Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling

Transition Trees for Cost-Optimal Symbolic Planning

Álvaro Torralba
Planning and Learning Group
Universidad Carlos III de Madrid, Spain
atorralb@inf.uc3m.es

Stefan Edelkamp
TZI Universität Bremen, Germany
edelkamp@tzi.de

Peter Kissmann
Universität des Saarlandes, Germany
kissmann@cs.uni-saarland.de

Abstract

Symbolic search with binary decision diagrams (BDDs) often saves huge amounts of memory and computation time.

In this paper we propose two general techniques based on transition relation trees to advance BDD search by refining the image operator to compute the set of successors. First, the conjunction tree selects the set of applicable actions through filtering their precondition. Then, the disjunction tree combines each transition result. Transition trees are used to combine several transition relations, speeding up BDD search.

Experiments with bidirectional symbolic blind and symbolic A* search on planning benchmarks are reported showing good performance on most IPC 2011 domains.

Introduction

Binary decision diagrams (BDDs) (Bryant 1986) are memory-efficient data structures used to represent Boolean functions as well as to perform set-based search. Symbolic search with BDDs avoids (or at least lessens) the costs associated with the exponential memory requirement for the state set involved as problem sizes get bigger. Symbolic planning with BDDs links to symbolic model checking (McMillan 1993) and has been pioneered by Cimatti et al. (1997) and many existing symbolic planners like MBP are built on top of symbolic model checkers. Symbolic A* (alias BDDA*) has been invented by Edelkamp and Reffel (1998) and integrated in the planning context by Edelkamp and Helmert (2001) in the model checking integrated planning system MIPS. ADDA* (Hansen, Zhou, and Feng 2002) is an alternative implementation with ADDs, while Set A* (Jensen, Bryant, and Veloso 2002) refined the partitioning in a matrix representation of g- and h-values.

In IPC 2011, the only BDD-based planner that participated was GAMER, already including the improvements proposed by Kissmann and Edelkamp (2011). In that IPC, explicit-state heuristic search planning showed advantages over symbolic planning, indicating that the increased quality of search heuristics sometimes exceeds the structural savings for representing and exploring large state sets in advanced data structures. However, implementation details may be relevant for evaluating different techniques. For example, it has been shown that an optimized version of explicit A* can be up to 5.8 times more efficient with a memory consumption 2.5 times lower (Burns et al. 2012).

One of the main computational bottlenecks in symbolic planning is successor generation. The *transition relation* is a function of predecessor and successor states, which encodes the planning actions. A traditional approach in symbolic model checking is using a disjunctive or conjunctive partition of the transition relation (Burch, Clarke, and Long 1991). In planning, as actions usually refer to a subset of variables, a disjunctive partitioning is the most natural technique. For example, Jensen, Veloso, and Bryant (2008) propose to partition the planning actions into several transition relations based on the difference in heuristic values.

In this work, we experiment with two improvements to BDD exploration, both based on trees of transitions. The conjunction tree filters the states supporting all preconditions of an action. The disjunction tree computes the disjunction of the individual images for each action. These transition trees are also used as a criterion for deciding the transition relation partitioning. We analyze the relevance of these two techniques for symbolic A* search, increasing the performance of symbolic planning.

The IPC 2011 version of GAMER performed bidirectional symbolic breadth-first search (BFS) for unit-cost domains and symbolic A* for the rest. Comparison with explicit search is even more difficult due to the use of different algorithms for each domain. Therefore, we extended the bidirectional symbolic BFS algorithm to bidirectional symbolic Dijkstra search for supporting action costs, in order to compare informed and blind symbolic algorithms.

The remainder of this paper is organized as follows. First, we introduce symbolic planning concepts, followed by the two algorithms considered in this paper: symbolic A* and bidirectional symbolic Dijkstra search. Then, we study some approaches for successor generation in symbolic planning. Experimental results show that using the new image computation algorithms increases the efficiency of both symbolic algorithms, making them competitive with other state-of-the-art techniques.

Symbolic Planning with BDDs

A *planning task* consists of variables of finite domain, so that states are assignments to the variables, an initial state,

the goal, and a finite set of actions, each being a pair of preconditions and effects. In *cost-based planning*, actions are associated with action cost values, which often are integers. The task is to find a solution path from the initial state to a goal. A solution path is *optimal* if its cost is smallest among all possible solutions. A *heuristic* is a distance-to-goal mapping, and admissible if for all possible states the value is not greater than the cost of an optimal solution path. A planning task *abstraction* is a planning task based on a relaxation for the initial state, goal state as well as the actions.

In symbolic search, binary decision diagrams (BDDs) represent all binary state vectors that evaluate to 0 or 1. More precisely, a BDD is a directed acyclic graph with one root and two terminal nodes, called sinks, with values 0 and 1. Each internal node corresponds to a binary variable of the state vector and has two successors (low and high), one representing that the current variable is false and the other representing that it is true. For any assignment of the variables on a path from the root to a sink the represented function will be evaluated to the value labeling the sink. Moreover, BDDs are unique, given a variable ordering, by applying the two reduction rules of (1) eliminating nodes with the same low and high successors and (2) merging two nodes representing the same variable that share the same low successor as well as the same high successor.

Symbolic search classifies states in layers S_g, according to the cost g needed to reach them. State sets are represented as BDDs by their corresponding characteristic functions. We may assume that the BDD variable ordering is fixed and has been optimized prior to the search. To generate the successors of a set of states, planning actions are represented in the *transition relation*. The transition relation is a function defined over two sets of variables, one (x) representing the current states and another (x') representing the successor states. To find the successors with cost $g + c$ of a set of states S_g represented in the current state variables (x) given a BDD T_c (the transition relation) for the entire set of actions with cost c, the $image$ is computed, i. e., $image(S, T_c) = \exists x \,.\, S(x) \wedge T_c(x, x')[x' \leftrightarrow x]$. Thus, the image is carried out in three steps. First, the conjunction with $T_c(x, x')$ filters preconditions on x and applies effects on x'. Existential quantification of the predecessor variables x is a standard BDD operation which removes the value relative to the predecessor states. Finally, $[x' \leftrightarrow x]$ denotes the swap of the two sets of variables, setting the value of the successor states in x. Similarly, search in backward direction is done by using the $pre\text{-}image$ operator (i. e., $pre\text{-}image(S, T_c) = \exists x' \,.\, S(x') \wedge T_c(x, x')[x \leftrightarrow x']$).

Symbolic A* Planning

Symbolic A* search classifies the states according to their g and h values, with g being the cost needed to reach the states and h a heuristic estimate of the remaining cost to the nearest goal. It expands the state sets in ascending order of $f = g + h$. Thus, when using an admissible heuristic, expanding all the states with $f < f^*$ (the cost of an optimal solution) guarantees to obtain on optimal solution. In this paper we consider the List A* implementation (Edelkamp, Kissmann, and Torralba 2012), which organizes the generated states in a list according to their g value and computes the conjunction with the heuristic only on demand.

The word *pattern* in the term pattern database (PDB) coined by Culberson and Schaeffer (1998) was inspired by a selection of tiles in the sliding-tiles puzzle, and has been extended to the selection of state variables in other domains. More general definitions have been applied, shifting the focus from the mere selection of care variables to different state-space abstractions that are computed prior to the search. Following the definition in (Edelkamp and Schrödl 2012), a PDB is characterized by memorizing an abstract state space, storing the shortest path distance from each abstract state to the set of abstract goal states. An efficient implementation for explicit-state PDB planning is given by Sievers, Ortlieb, and Helmert (2012). A *partial pattern database* (Anderson, Holte, and Schaeffer 2007) is a PDB that is not fully calculated, but rather its calculation is stopped after either a pre-defined distance to the nearest goal or a pre-defined time-out has been reached. If all states with a distance of d have been generated, then all the other states can safely be assigned a value of $d + 1$.

Symbolic PDBs (Edelkamp 2005) are PDBs that have been constructed symbolically as decision diagrams for later use either in symbolic or explicit heuristic search. In contrast to the posterior compression of the state set (Ball and Holte 2008), the construction in (Edelkamp 2005) works on the compressed representation, allowing larger databases to be constructed. The savings observed by the symbolic representation are substantial in many domains. For symbolic heuristic search (Jensen, Veloso, and Bryant 2008) it is often more convenient to represent the PDB as a vector of BDDs, i. e., where the heuristic relation is partitioned into $\mathit{Heur}[0](x), \ldots, \mathit{Heur}[\max_h](x)$.

For symbolic PDB construction in unit-cost state spaces, backward symbolic breadth-first search is used. For a given abstraction function the symbolic PDB is initialized with the projected goal. As long as there are newly encountered states, we take the current backward search frontier and generate the predecessor list with respect to the abstracted transition relation. Then we attach the current BFS level to the new states, merge them with the set of already reached states, and iterate. When action costs are integers this process can be extended from breadth-first to cost-first levels.

The implementation we based our experiments on (the planner GAMER by Kissmann and Edelkamp (2011)) automatically decides whether to use a procedure similar to the one proposed by Haslum et al. (2007) to automatically select a pattern or to calculate a partial PDB based on the original non-abstracted input. In both cases, PDB creation can take up to half the available time. In this paper, we only consider the automatic pattern selection procedure, since partial PDBs on the original state space are similar to bidirectional blind search.

Bidirectional Symbolic Shortest Path Search

As another extreme, we implemented bidirectional shortest path search on domains with non-unit action costs (cf. Algorithm 1). The motivation for this was that in GAMER the automatic decision procedure often chose to not abstract the

Algorithm 1 Bidir. Symbolic Dijkstra: $\mathcal{A}, \mathcal{I}, \mathcal{G}, w, T$

$fClosed \leftarrow bClosed \leftarrow \emptyset$
$fReach_0 \leftarrow \mathcal{I}$
$bReach_0 \leftarrow \mathcal{G}$
$g_f \leftarrow g_b \leftarrow 0$
$w_{total} \leftarrow \infty$
while $g_f + g_b < w_{total}$
  if $NextDirection = Forward$
    $\{g_1, \ldots, g_n\} \leftarrow fStep(fReach, g_f, fClosed, bClosed)$
    for all $g \in \{g_1, \ldots, g_n\}$
      for all $i \in \{i \mid i < g_b$ **and** $bReach_i \neq \emptyset$ **and** $g + i < w_{total}\}$
        if $fReach_g \wedge bReach_i \neq \emptyset$
          $w_{total} \leftarrow g + i$
          update plan π
    $g_f \leftarrow g_f + 1$
  else // same in backward direction
return π

Procedure $fStep(fReach, g, fClosed, bClosed)$
$Ret \leftarrow \{\}$
$fReach_g \leftarrow \mathbf{BFS}_{c=0}(fReach_g) \wedge \neg fClosed$
for all $c \in \{1, \ldots, C\}$
  $Succ \leftarrow \mathbf{image}(fReach_g, T_c)$
  if $Succ \wedge bClosed \neq \emptyset$
    $Ret \leftarrow Ret \cup \{g + c\}$
  $fReach_{g+c} \leftarrow fReach_{g+c} \vee Succ$
$fClosed \leftarrow fClosed \vee fReach_g$
return Ret

domains, but rather calculate the partial PDB until a timeout was reached. In those cases bidirectional blind search is more flexible, as it is able to select whether to perform a backward or a forward step at any time.

The algorithm takes a set of actions $\mathcal{A}$, the initial state $\mathcal{I}$, the goal states $\mathcal{G}$, the action costs w and the transition relation T, which is a set of BDDs T_c for each action cost c, as input. The forward search starts at $\mathcal{I}$, the backward search at $\mathcal{G}$. In case a forward step is to be performed next, the procedure *fStep* is called, which removes the already expanded states from the bucket to be expanded next in the open list $fReach$. Then it computes the image to find the set of successor states and inserts them into the correct buckets of $fReach$. Whenever a newly generated successor, which is inserted into bucket $g+c$ in $fReach$, has already been solved in backward direction we store the index $g + c$.

After the *fStep* the algorithm continues with the set of stored indices corresponding to states that were just generated in forward direction and already expanded in backward direction (those indices stored in Ret). It searches for the same states in the backward buckets $bReach$. When it finds a bucket that contains such a state the sum of the indices of backward and forward direction corresponds to the smallest cost of a solution path over such a state. If this cost is smaller than the smallest cost found so far (w_{total}) then w_{total} is updated and the corresponding solution path π can be created. The procedure for the backward step looks nearly the same, only that the forward and backward sets are swapped and pre-images instead of images are applied.

The stopping criterion in the BDD implementation of a bidirectional version of Dijkstra's (1959) algorithm is not immediate, as individual shortest paths for the states cannot be maintained and many improvements for bidirectional explicit state search are not immediate (Rice and Tsotras 2012). In the context of external search the following criterion has been established (Goldberg and Werneck 2005): *stop the algorithm when the sum of the minimum g-values of generated states for the forward and backward searches is at least w_{total}, the total of the cheapest solution path found so far.* They have shown that this stopping condition guarantees an optimal solution to be found. Since the g-value for each search is monotone in time, so is their sum. After the condition is met, every state s removed from a priority queue will be such that the costs of the solution paths from $\mathcal{I}$ to s and from s to $\mathcal{G}$ will be at least w_{total}, which implies that no solution path of cost less than w_{total} exists.

To decide whether to perform a forward or backward step, the algorithm selects the direction with lower estimated time. The estimated time for step k, t_k, is based on the time spent on the last step t_{k-1} and the BDD sizes for the state set to be expanded and the state set expanded in the last step, s_k and s_{k-1}, respectively. A linear relation between BDD size and image time is assumed:

$$t_k = t_{k-1} \frac{s_k}{s_{k-1}}$$

Though the estimation is not perfect, it is often accurate enough to determine the best direction. In some domains a backward step takes a lot longer than a forward step, so that we have implemented the possibility to abolish backward search altogether and continue only in forward direction in case a backward step takes significantly longer than the slowest forward step.

Efficient Image Computation

Having a single transition relation per action cost c, $T_c = \bigvee_{a \in \mathcal{A}, w(a)=c} T_a$ is often infeasible due to the exponential memory cost required to represent it. GAMER considers a disjunctive partitioning of the transition relation with a transition for each grounded action of the planning task, $\mathcal{T} = \{T_a \mid a \in \mathcal{A}\}$. The image is computed as the disjunction of the individual images wrt. the transition relation of each grounded planning action:

$$image(S, T_c) = \bigvee_{a \in \mathcal{A}, w(a)=c} image(S, T_a).$$

This guarantees some properties on the transition relations of the problem. In particular, the effect of an action is expressed by a list of assignments of variables to values, which do not depend on the preconditions[1]. All the variables not appearing in the effects must retain their values. It is possible to take advantage of this structure. When computing

[1]More complex effects such as conditional effects can be compiled away with an exponential growth in the number of actions (Nebel 1999).

the image wrt. an action T_a, the existential quantification and variable swapping need only to be applied over variables modified by its effects, $x_a \subseteq x$:

$$image(S, T_a) = \exists x_a \,.\, S(x) \wedge T_a(x, x')[x'_a \leftrightarrow x_a].$$

Furthermore, it is possible to divide the transition relation $T_a(x, x')$ into its precondition $T_a^{pre}(x)$ and effect $T_a^{eff}(x_a)$, such that $T_a(x, x') = T_a^{pre}(x) \wedge \left(T_a^{eff}(x_a)[x_a \leftrightarrow x'_a]\right)$. Applying the precondition and the effect separately we can express transition relations using only the set of predecessor variables, so that variables representing the successor states x' as well as the swap operation are no longer needed:

$$image(S, T_a) = (\exists x_a \,.\, S(x) \wedge T_a^{pre}(x)) \wedge T_a^{eff}(x_a).$$

When considering precondition matching and effect application as two separate operations, the conjunction with the precondition of each action might be reused by several actions sharing the same precondition. First, we extend this idea by defining a conjunction tree that filters states in which each action is applicable. Then, we consider combining several transition relations, avoiding the exponential memory blowup by imposing a limit on the transition sizes.

Conjunction Tree

In many explicit-state planners there are speed-up techniques for filtering the actions that match, such as the successor generators used in (Helmert 2006). In those successor generators, the actions are organized in a tree (see Figure 1a). Each leaf node contains a set of actions having the same preconditions, $\mathcal{A}_{pre}$. Every internal node is associated with a variable v and splits the actions with respect to their precondition for v, one child for every possible value that v can take and an additional one for the actions whose precondition is independent of v. To compute the successors of a state the tree is traversed, omitting branches labeled with unsatisfied preconditions.

We found that the approach carries over to BDD search as follows. As all the variables are binary[2], each internal node only has three children c_0, c_1 and c_x, dividing the actions into three sets: those that have $\overline{v}$ as a precondition, those that have v as precondition and those whose precondition does not depend on v at all, respectively. When computing the successor states, the actions are not applied over a single state as in explicit search. Instead, all the successors of a set of states represented as a BDD are computed. Still, actions in a branch of the tree only need to be applied over states satisfying the corresponding precondition. We take advantage of this by computing the subset of states satisfying the precondition as the conjunction of the state set with the precondition label. We call this method *conjunction tree* (CT).

Algorithm 2 shows how to compute the image of a BDD using the conjunction tree structure. It takes as input the set of states S and the root node of the conjunction tree and returns a set of BDDs per action cost, corresponding to the images wrt. each transition relation. For each internal node one recursive call is applied for each child node, applying the corresponding conjunction between the state set S and the precondition associated with it. This conjunction is reused for all the actions in that branch. Moreover, if there are no states satisfying the preconditions of a branch, it is not traversed. In the leaf nodes it computes the image wrt. all the actions in that node. The image wrt. each action can be computed using a transition relation or with separated precondition and effect BDDs.

Algorithm 2 CT: Image using the conjunction tree

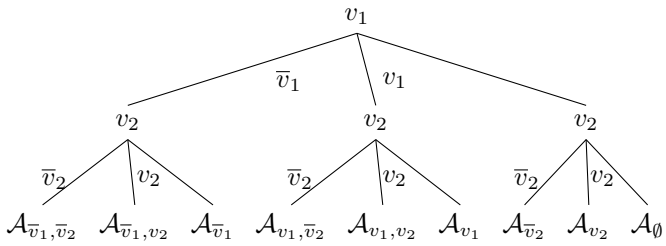
CT-image *(node, S)*:
  if $S = \emptyset$:
    return $\emptyset$
  if *node is leaf*:
    return $\bigcup_{a \in node.\mathcal{A}} \{w(a), image(S, T_a)\}$
  $v \leftarrow$ *node.variable*
  $r_0 \leftarrow$ CT-*image*(*node.c0*, $S \wedge \neg v$)
  $r_1 \leftarrow$ CT-*image*(*node.c1*, $S \wedge v$)
  $r_x \leftarrow$ CT-*image*(*node.cx*, S)
  return $r_0 \cup r_1 \cup r_x$

Conjunction trees for backward search are initialized taking into account the preconditions of the inverted actions. In the presence of 0-cost actions, before expanding a layer both algorithms (symbolic A* and bidirectional symbolic Dijkstra search) need to apply a BFS based only on the 0-cost actions until a fixpoint is reached. Since cost and 0-cost actions are applied over different sets of states, two different conjunction trees are needed: one containing the 0-cost actions and another with the rest. Thus, in order to apply bidirectional search in domains with 0-cost actions, up to four different conjunction trees are needed.

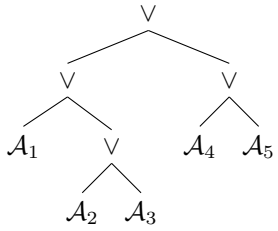
The conjunction tree has two advantages over the previous approach of just computing the images for every transition independently. On the one hand, if the state set does not contain any state in a branch of the tree, all the actions regarding that branch are ignored reducing the total number of images. On the other hand, if several actions have the same precondition, the conjunction of the state set with that precondition is only computed once, reducing the size of the BDDs we are computing the images for.

However, there might be an overhead in computing and storing intermediate BDDs in memory. To address this, the conjunction tree should only be used when the benefits are estimated to be enough to compensate the overhead. This is the case when the conjunction with some precondition is shared between several actions. Thus, the conjunction tree can be parametrized with a parameter *min operators conjunction*, so that the intermediate conjunction with a partial precondition is only computed when needed for at least that number of actions. If the number of actions with that precondition is not enough, they are placed on the *don't care* branch of the tree and marked so that we know that an extra conjunction with their preconditions is needed. Notice that in case that some preconditions have been conjoined and others not, the conjunction with the preconditions is needed,

[2]Finite-domain state variables are compiled into several binary variables in the BDD representation.



(a) Conjunction tree for applying an operator's precondition on a state set



(b) Disjunction Tree

Figure 1: Transiton trees clustering operators.

but there may nevertheless be some benefit because the state set has been reduced. When *min operators conjunction* is set to 1 we have the full tree strategy and when it is set to ∞ the transition tree consists of only one leaf node, which is equivalent to not having a tree at all. Setting the parameter to intermediate values produces intermediate strategies.

Unifying Transition Relations

Even though having a monolithic transition relation per action cost is often infeasible due to its size, unifying transition relations of several planning actions may optimize image computation. The union of a set of transition relations $\mathcal{T}$ is a new transition relation $Union(\mathcal{T})(x, x')$ such that the image wrt. the new transition is equivalent to the disjunction of the images of the individual transitions. The new transition relation cannot be represented with separated precondition and effect BDDs, so that predecessor and successor variables (x and x') are needed again. As the cost of reaching a state must be preserved, only actions with the same action-cost may be merged.

The new transition relation modifies the value of variables in the effect of any of the actions it is composed of. Thus, the set of effect variables of $Union(\mathcal{T})$ is the union of the effect variables of all transitions in $\mathcal{T}$: $x_{\mathcal{T}} = \bigcup_{T \in \mathcal{T}} x_T$. The image operation wrt. $Union(\mathcal{T})$ applies an existential quantification over all the variables in $x_{\mathcal{T}}$, replacing their value with the corresponding successor variables $x'_{\mathcal{T}}$. Transition relations of actions not affecting some variable in $x_{\mathcal{T}}$ are combined with the biimplication between the predecessor and successor state variables they must not modify, $biimp\,(x_i, x'_i) = (x_i \wedge x'_i) \vee (\overline{x}_i \wedge \overline{x}'_i)$. Thus, the new transition relation is computed as:

$$Union(\mathcal{T}) = \bigvee_{T \in \mathcal{T}} \left(T \wedge \bigwedge_{x_i \in x_T, x_i \notin x_a} biimp\,(x_i, x'_i) \right).$$

A critical decision is which actions to merge into one transition relation. Since the result of the image wrt. the new transition is the same as the disjunction of the image wrt. the individual transitions, a criterion to decide which actions should be merged could be the disjunction tree used by GAMER to disjoint the result of each transition relation.

The disjunction tree classifies the set of actions with a given action-cost in a binary tree (see Figure 1b) to decide in which order to apply the disjunction of the individual image results. Each internal node applies a disjunction of the result of its left and right branch. Leaf nodes are associated with the image result of single transition relations. Different organizations of disjunction trees may influence overall performance, since the order of the disjunctions determines the complexity of intermediate BDDs. GAMER constructs a balanced disjunctive tree by arbitrarily splitting the actions in each internal node in two equally sized partitions.

Algorithm 3 U_{DT}: Unifying transitions using the disjunction tree

```
U_DT (node):
  if node is leaf:
    return T_node.A
  T_l ← U_DT(node.l)
  T_r ← U_DT(node.r)
  if |T_l| = 1 ∧ |T_r| = 1:
    T' ← Union(T_l ∪ T_r)
    if size(T') ≤ MAX_TR_SIZE:
      return {T'}
  return T_l ∪ T_r
```

Algorithm 3 takes as input the root of a disjunction tree and returns a set of transition relations. It decides which transition relations to merge by recursively merging branches with only two transition relations. In leaf nodes, it simply returns the transition relation of the action associated with the leaf. For internal nodes, first it recursively merges its left and right branch. If both branches were successfully merged, it merges their corresponding transition relations. If the result does not violate maximum memory requirements it is returned, otherwise it is discarded and the results of the left and right children are returned (so that parent nodes will not unify any more transitions).

However, U_{DT} is not a good criterion to merge transitions when using the conjunction tree. The combined transition relation must be applied to all the states satisfying the preconditions of any action it combines. Thus, in order to apply both ideas together, we define another criterion for merging the operators taking into account their preconditions: U_{CT}. U_{CT} merges transition relations of actions present in the same branch of the conjunction tree. Actions present in a CT

Algorithm 4 U_{CT}: Unifying operators using the CT

U_{CT} *(node)*:
  if *node is leaf*:
    return $\bigcup_{c\in\{1,\ldots,C\}}$ **disjLeaf** $(node.T^c)$
  $\mathcal{T}_0 \leftarrow \mathbf{U}_{\mathrm{CT}}$ *(node.c0)*
  $\mathcal{T}_1 \leftarrow \mathbf{U}_{\mathrm{CT}}$ *(node.c1)*
  $\mathcal{T}_x \leftarrow \mathbf{U}_{\mathrm{CT}}$ *(node.cX)*
  for all $c \in \{1,\ldots,C\}$:
    $\mathcal{T}_x^c \leftarrow \mathcal{T}_x^c \cup$ **merge**$(\mathcal{T}_0^c, \mathcal{T}_x^c)$
    $\mathcal{T}_x^c \leftarrow \mathcal{T}_x^c \cup$ **merge**$(\mathcal{T}_1^c, \mathcal{T}_x^c)$
    $\mathcal{T}_x^c \leftarrow \mathcal{T}_x^c \cup$ **merge**$(\mathcal{T}_0^c, \mathcal{T}_1^c)$
  return $\mathcal{T}_x \cup \mathcal{T}_0 \cup \mathcal{T}_1$

disjLeaf$(\mathcal{T})$
  $\mathcal{T}_{leaf} = \emptyset$
  while $|\mathcal{T}| > 1$:
    $\mathcal{T}' \leftarrow \emptyset$
    for $T_i, T_j \in \mathcal{T}$:
      $\mathcal{T} \leftarrow \mathcal{T} \setminus T_i, T_j$
      $T_i' \leftarrow$ *Union*$(\{T_i\} \cup \{T_j\})$
      if *size*$(T_i') \leq$ *MAX_TR_SIZE*:
        $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{T_i'\}$
      else:
        $\mathcal{T}_{leaf} \leftarrow \mathcal{T}_{leaf} \cup \{T_i\} \cup \{T_j\}$
    $\mathcal{T} \leftarrow \mathcal{T}'$
  return $\mathcal{T}_{leaf} \cup \mathcal{T}$

merge $(\mathcal{T}_i, \mathcal{T}_j)$
  if $|\mathcal{T}_i| = 1$ **and** $|\mathcal{T}_j| = 1$:
    $T' \leftarrow$ *Union*$(\mathcal{T}_i \cup \mathcal{T}_j)$
    if *size*$(T') \leq$ *MAX_TR_SIZE*:
      $\mathcal{T}_i \leftarrow \emptyset$
      $\mathcal{T}_j \leftarrow \emptyset$
      return $\{T'\}$
  return $\emptyset$

branch are guaranteed to share the preconditions checked in the previous inner nodes. Thus, the new transition relation only needs to be applied over states satisfying the preconditions shared by all those actions.

Algorithm 4 shows the pseudo-code of the U_{CT} procedure. It takes as input a CT node and returns a set of transition relations. In the leaf nodes, the *disjLeaf* procedure merges individual transitions until the maximum transition size is reached or only one transition is left. On internal nodes, it recursively gets sets of operators associated with the current node's variable. Then, for each action cost it attempts to merge transitions from different children. The preconditions of the new transitions no longer are related to the variable (they are now encoded in the transition relation BDD), so that they are placed in the T_x bucket. The *merge* procedure gets two sets of transitions and, if both have only one transition, it attempts to generate the union of both transitions. If this is successful it returns the new transition, removing the old ones from their corresponding sets. Otherwise, they could not be merged by the maximum transition size restriction so that the algorithm returns the empty set and does not change the original sets. In this case, it will not attempt to merge them again.

Whenever the resulting BDD exceeds MAX_TR_SIZE, we interrupt the union operation and drop the result, so that calculating the union of two transitions cannot get worse than $O(\text{MAX\_TR\_SIZE})$. In Algorithms 3 and 4 we have a linear (in the number of operators) number of calls to $Union$, so that both run in linear time in the number of operators. This is adequate for the planning problems considered in our benchmarks, where the huge number of actions in some domains could be prohibitive for algorithms with greater complexity.

Experiments

As the basis for the experiments we take GAMER (Kissmann and Edelkamp 2011) for performing symbolic search optimal planning. The software infrastructure is taken from the resources of the International Planning Competition (IPC) 2011. We implemented several refinements in GAMER using the CUDD library of Fabio Somenzi (compiled for 64-bit Linux using the GNU gcc compiler, optimization option -O3). For the experiments we used an 8-core Intel(R) Xeon(R) X3470 CPU @2.93 GHz with the same settings concerning timeout (30 minutes) and maximal memory usage (6 GB) as in the competition.

In order to evaluate the efficiency of the image computation approaches, we try a number of different configurations when doing symbolic bidirectional blind search. The coverage of each planner is not completely informative, since in some domains the exponential gap between problems causes all the versions to perform the same, even if there is a huge performance gap. To compare the planners' efficiency we use the time metric proposed for the learning track of IPC 2011. It evaluates the performance on a problem taking into account the time used to solve it (t) and the minimum time required by any planner (t^*):

$$\frac{1}{1+\log_{10}(t/t^*)}.$$

This metric assigns a score between 0 and 1 to each planner for each problem solved, depending on the time it took to solve the problem with respect to the time of the best planner. Thus, the fastest planner receives 1 point, while all the others receive less points according to a logarithmic scale.

Bidirectional blind search gives a better comparison of the image techniques than A* search because (1) it is more straightforward, not depending on more parameters like the PDB generation procedure and (2) time comparisons with A* are not fair due to the fixed 15 minutes time window for generating the PDBs.

Table 1 shows a comparison of the time score for each proposed image computation technique. The basic approach, TR, uses a transition relation BDD for each action. $\mathrm{TR}_{\mathcal{A}}$ takes advantage of the planning actions structure to get rid of the successor variables in the image computation. CT and CT_{10} apply a conjunction tree, the latter setting the *min operators conjunction* to 10. $\mathrm{U}_{\mathrm{DT}}^{1k}$ and $\mathrm{U}_{\mathrm{DT}}^{100k}$ merge transition relations following the disjunction tree criterion with a

Domains	% fw	TR	$TR_{\mathcal{A}}$	CT	CT_{10}	U_{DT}^{1k}	U_{DT}^{100k}	U_{CT}^{1k}	U_{CT}^{100k}	$U_{CT_{10}}^{1k}$
BARMAN	94	4.56	4.99	4.60	4.69	6.41	7.86	6.75	**8.00**	6.70
ELEVATORS	74	11.94	14.04	13.63	12.99	17.58	**18.69**	18.35	18.22	18.34
FLOORTILE	37	4.15	6.91	5.54	4.83	8.81	9.76	8.92	**9.77**	9.14
NOMYSTERY	57	9.17	10.58	10.28	10.34	14.36	**15.73**	15.40	14.74	14.39
OPENSTACKS	90	11.61	11.99	12.03	12.11	16.29	19.34	15.15	**19.43**	16.53
PARC-PRINTER	62	5.46	6.55	6.31	5.49	7.57	7.60	7.38	**7.61**	7.41
PARKING	100	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
PEG-SOLITAIRE	100	15.74	15.91	15.30	15.83	16.45	16.88	16.26	**16.94**	16.46
SCANALYZER	53	5.55	6.29	7.23	6.92	8.16	**9.00**	8.40	8.92	8.29
SOKOBAN	94	11.49	11.83	9.57	11.49	16.19	18.75	13.71	**18.83**	16.26
TIDYBOT	100	6.17	6.19	13.38	13.31	10.32	7.63	**13.78**	6.64	13.42
TRANSPORT	77	4.56	6.93	7.10	7.03	8.74	8.66	8.48	8.76	**8.82**
VISITALL	53	8.48	10.84	10.52	10.38	10.47	10.69	**10.78**	10.76	10.65
WOODWORKING	40	10.21	12.41	10.41	11.04	14.67	15.08	**15.19**	15.09	15.17
Total	74	109.09	125.46	125.90	126.45	156.02	**165.67**	158.55	163.71	161.58

Table 1: Time score comparison of image functions for the IPC 2011 domains.

threshold on the maximum size of each transition relation of 1,000 and 100,000 nodes, respectively. Finally, U_{CT}^{1k}, U_{CT}^{100k} and $U_{CT_{10}}^{1k}$ use the CT as criterion to unify transition relations combining both approaches.

Column %fw shows the average percentage of steps taken in forward direction by all versions. In PARKING, PEG-SOLITAIRE and TIDYBOT no backward search is accomplished, while in BARMAN, OPENSTACKS, and SOKOBAN the amount of backward search is negligible. Thus, in 6 out of 14 domains bidirectional search behaves like blind forward search. In contrast, FLOORTILE and WOODWORKING take more advantage from backward than from forward search. In general, bidirectional search is able to detect the most promising direction for the search.

The comparison of $TR_{\mathcal{A}}$ and TR reveals that, when considering transition relations associated with a single planning action, it is possible to take advantage of their structure increasing the performance across all domains. The use of CT is helpful in some domains, especially in TIDYBOT, TRANSPORT, and VISITALL, though it worsens the results in SOKOBAN and PEG-SOLITAIRE. When choosing a suitable value for *min operators conjunction* to limit the application of the CT, CT_{10} obtains more robust results on all domains, not being worse than the original TR in any domain and giving a good alternative for the conjunction tree.

However, versions unifying transitions get the best overall performance, dominating all the previous approaches in all domains except TIDYBOT where for some configurations the memory bound is reached while unifying the transition relations. This suggests that unifying transition relations always helps when there is enough memory to do so. This is also true when comparing versions with a maximum transition size of 100,000 nodes and versions merging transitions only up to 1,000 nodes. Both the conjunctive and disjunctive tree show good potential for merging transition relations. The comparison between them depends on the parameter for the maximum transition size. With a value of 1,000 nodes, U_{CT}^{1k} and $U_{CT_{10}}^{1k}$ show better performance than U_{DT}^{1k}, especially in TIDYBOT where the CT usually works best. On the other hand, when increasing the size of transitions up to 100,000 nodes, the number of transitions decreases resulting in the CT being less effective and reaching the memory bound in TIDYBOT, causing U_{CT}^{100k} to work worse than U_{DT}^{100k}.

In order to compare the different search algorithms, some of the previously analyzed image techniques are selected. TR gives a good baseline comparison, CT gives similar coverage results to CT_{10} being more different wrt. other configurations and U_{DT}^{100k} and $U_{CT_{10}}^{1k}$ are the versions with the best performance in the time and coverage metrics, respectively.

Regarding the heuristic used by the A* version, the IPC 2011 GAMER selected whether to apply abstraction or not depending on how costly backward exploration is in the original space. This was done by looking at the CPU times for the first backward step. If it is too hard, then abstraction is applied instead of calculating a non-abstracted partial PDB. As the partial PDBs are closer to bidirectional search (but in a non-interleaved manner – backward search is stopped after 15 minutes if it does not finish earlier), we use the abstracted PDBs for all the experiments. The patterns are automatically selected as explained in (Kissmann and Edelkamp 2011).

For comparing our performance against state-of-the-art results, we select two different competitors. One is the winner of the last IPC, FAST DOWNWARD STONE SOUP-1 (Helmert, Röger, and Karpas 2011), a static portfolio using explicit state A* with several abstraction and landmark heuristics. The other one is the best coverage obtained by any planner in the competition excluding GAMER (which are FAST DOWNWARD STONE SOUP-1 results plus three PARC-PRINTER problems solved by CPT-4 (Vidal and Geffner 2006) and one problem in VISITALL solved by FORKINIT (Katz and Domshlak 2011)). Notice that there is a significant difference between a single-technique planner, such as GAMER, and portfolio approaches. Indeed, the optimal portfolio for IPC 2011 includes GAMER among other explicit search planners (Núñez, Borrajo, and López 2012).

Table 2 shows the coverage results on the IPC 2011 tasks

	GAMER Bidirectional				GAMER A*				IPC 2011	
Domain	TR	CT	U_{DT}^{100k}	U_{CT10}^{1k}	TR	CT	U_{DT}^{100k}	U_{CT10}^{1k}	FDSS-1	BEST
BARMAN	**8**	**8**	**8**	**8**	4	4	5	4	4	4
ELEVATORS	**19**	**19**	**19**	**19**	17	17	**19**	**19**	18	18
FLOORTILE	7	8	10	10	11	11	**12**	**12**	7	7
NOMYSTERY	13	14	16	16	13	14	14	15	**20**	**20**
OPENSTACKS	**20**	**20**	**20**	**20**	**20**	**20**	**20**	**20**	16	16
PARC-PRINTER	7	7	8	8	9	9	9	9	14	**17**
PARKING	0	0	0	0	1	1	1	1	**7**	**7**
PEG-SOLITAIRE	17	17	17	17	17	17	17	17	**19**	**19**
SCANALYZER	9	9	9	9	8	9	9	9	**14**	**14**
SOKOBAN	18	18	19	18	**20**	19	**20**	**20**	**20**	**20**
TIDYBOT	8	**14**	11	**14**	12	**14**	**14**	**14**	**14**	**14**
TRANSPORT	7	**9**	**9**	**9**	6	6	6	6	7	7
VISITALL	9	11	11	11	11	11	11	11	13	**14**
WOODWORKING	16	16	16	16	16	16	**19**	**19**	12	12
Total	158	170	173	175	165	168	176	176	185	**189**

Table 2: Planners' coverage for the problems of the optimal track of IPC 2011.

for symbolic bidirectional blind and A* search with the selected successor generation methods. With the new image computation approaches the results of GAMER show that symbolic search in both variants – bidirectional blind and symbolic A* search – is competitive with state-of-the-art optimal planning techniques. With 175 and 176 solutions, respectively, both versions are almost tied and they would have been the runner-up of IPC 2011, only beaten by both FAST DOWNWARD STONE SOUP versions. This makes GAMER the best non-portfolio approach to cost-optimal planning. Furthermore, it dominates all the other planners in 6 out of 14 domains: BARMAN, ELEVATORS, FLOORTILE, OPENSTACKS, TRANSPORT, and WOODWORKING. Taking GAMER into account, the best performance achieved by any planner for each domain reaches 212 solutions instead of 189. When comparing the performance of both algorithms, it is noticeable that for domains where backward search works better (the bidirectional search decided to take more backward steps than forward), FLOORTILE and WOODWORKING, the A* version gets even better results than bidirectional blind search. Also, in domains where the bidirectional search takes almost no backward steps, sometimes the use of abstraction is better (in PARKING or SOKOBAN) but sometimes the h-partitioning makes the forward exploration more difficult, e. g., in BARMAN.

The domains where symbolic search does not perform well are PARC-PRINTER, PARKING, and SCANALYZER. In those domains the new successor generator techniques may help a bit but the performance is still far away from state-of-the-art planners. The reason for PARC-PRINTER is the huge amount of different action costs, resulting in a huge number of small state sets with different g values, so that symbolic search does not seem a good choice over explicit state search. In PARKING and SCANALYZER the BDD-representation for the state sets grows exponentially with g. In PARKING this effect is even worse because backward search does not scale and bidirectional search is reduced to blind forward search.

Conclusion

According to the outcome of the last two international planning competitions, heuristic and symbolic search are two leading methods for sequential optimal planning. There is no doubt that symbolic search has the effectiveness to explore large state sets, while the explicit-state heuristics are often more informed. While the result of IPC 2011 suggested a clear advantage for heuristic explicit-state search, with the contributed new options for computing the relational product for the image using conjunctive and disjunctive transition trees we have pushed the performance of symbolic search planners, showing they are competitive with state-of-the-art non-portfolio planners and closing the gap to the portfolio ones.

Also, while the competition version of GAMER used bidirectional symbolic search for unit-cost domains and symbolic A* for the rest, we have compared the performance of both approaches separately, showing close performance. The good results of symbolic blind search are especially surprising as after many years of research of finding refined heuristics in the AI planning domain this form of blind search still outperforms all existing planners on some domains, while being close on most others. Flexibility for deciding whether to advance the search in forward or backward direction compensates the use of heuristics.

One important step is to split the transition relation into a precondition part and an effect part. This avoids referring to successor variables and documents a significant change to the way symbolic search is typically performed, reducing the variable set by a half. Future research avenues for refining the image operator might be a partitioning based on the state set size or by combining symbolic with explicit search.

Acknowledgments

This work has been partially supported by MICINN projects TIN2008-06701-C03-03 and TIN2011-27652-C03-02 and by DFG project ED 74/11.

References

Anderson, K.; Holte, R.; and Schaeffer, J. 2007. Partial pattern databases. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*, 20–34. Springer.

Ball, M., and Holte, R. C. 2008. The compression power of symbolic pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2–11. AAAI Press.

Bryant, R. E. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Burch, J. R.; Clarke, E. M.; and Long, D. E. 1991. Symbolic model checking with partitioned transition relations. In *International Conference on Very Large Scale Integration*, 49–58. North-Holland.

Burns, E. A.; Hatem, M.; Leighton, M. J.; and Ruml, W. 2012. Implementing fast heuristic search code. In *Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via model checking: A decision procedure for AR. In *European Conference on Planning (ECP)*, 130–142. Springer.

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

Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik* 1:269–271.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system MIPS. *AI-Magazine* 67–71.

Edelkamp, S., and Reffel, F. 1998. OBDDs in heuristic search. In *German Conference on Artificial Intelligence (KI)*, 81–92.

Edelkamp, S., and Schrödl, S. 2012. *Heuristic Search – Theory and Applications*. Academic Press.

Edelkamp, S.; Kissmann, P.; and Torralba, Á. 2012. Symbolic A^* search with pattern databases and the merge-and-shrink abstraction. In *European Conference on Artificial Intelligence (ECAI)*, 306–311. IOS Press.

Edelkamp, S. 2005. External symbolic heuristic search with pattern databases. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 51–60. AAAI Press.

Goldberg, A. V., and Werneck, R. F. F. 2005. Computing point-to-point shortest paths from external memory. In *Workshop on Algorithm Engineering and Experiments and Workshop on Analytic Algorithmics and Combinatorics, ALENEX /ANALCO*, 26–40.

Hansen, E. A.; Zhou, R.; and Feng, Z. 2002. Symbolic heuristic search using decision diagrams. In *Symposium on Abstraction, Reformulation and Approximation (SARA)*.

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

Helmert, M.; Röger, G.; and Karpas, E. 2011. Fast downward stone soup: A baseline for building planner portfolios. In *ICAPS-Workshop on Planning and Learning (PAL)*.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Jensen, R. M.; Bryant, R. E.; and Veloso, M. M. 2002. SetA*: An efficient BDD-based heuristic search algorithm. In *National Conference on Artificial Intelligence (AAAI)*, 668–673. AAAI Press.

Jensen, R. M.; Veloso, M. M.; and Bryant, R. E. 2008. State-set branching: Leveraging BDDs for heuristic search. *Artificial Intelligence* 172(2–3):103–139.

Katz, M., and Domshlak, C. 2011. Planning with implicit abstraction heuristics. In *7th International Planning Competition (IPC)*, 46–49.

Kissmann, P., and Edelkamp, S. 2011. Improving cost-optimal domain-independent symbolic planning. In *AAAI Conference on Artificial Intelligence (AAAI)*, 992–997. AAAI Press.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Nebel, B. 1999. Compilation schemes: A theoretical tool for assessing the expressive power of planning formalisms. In *German Conference on Artificial Intelligence (KI)*, 183–194. Springer.

Núñez, S.; Borrajo, D.; and López, C. L. 2012. Performance analysis of planning portfolios. In *Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Rice, M. N., and Tsotras, V. J. 2012. Bidirectional a* search with additive approximation bounds. In *Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Sievers, S.; Ortlieb, M.; and Helmert, M. 2012. Efficient implementation of pattern database heuristics for classical planning. In *Symposium on Combinatorial Search (SoCS)*. AAAI Press.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.