Factored Planning Using Decomposition Trees

Elena Kelareva
University of Melbourne
Melbourne, Victoria 3010 Australia
e.kelareva@ugrad.unimelb.edu.au

Olivier Buffet,* **Jinbo Huang, Sylvie Thiébaux**
National ICT Australia and Australian National University
Canberra, ACT 0200 Australia
{firstname.lastname}@nicta.com.au

Abstract

Improving AI planning algorithms relies on the ability to exploit the structure of the problem at hand. A promising direction is that of factored planning, where the domain is partitioned into subdomains with as little interaction as possible. Recent work in this field has led to an detailed theoretical analysis of such approaches and to a couple of high-level planning algorithms, but with no practical implementations or with limited experimentations. This paper presents `dTreePlan`, a new generic factored planning algorithm which uses a decomposition tree to efficiently partition the domain. We discuss some of its aspects, progressively describing a specific implementation before presenting experimental results. This prototype algorithm is a promising contribution—with major possible improvements—and helps enrich the picture of factored planning approaches.

1 Introduction

Improving AI planning algorithms relies on the ability to exploit the structure of the problem at hand. Being particularly interested in composite systems, i.e., in planning in a network of components, we investigate a promising direction known as *factored planning*, where the domain is partitioned into subdomains with as little interaction as possible.

Factored planning is related to *abstract planning*, as both involve planning on abstracted versions of the original problem. Abstract planning usually involves progressively refining a plan while going down a sequence of decreasingly abstract domains, backtracking if the current plan cannot be refined further [Knoblock *et al.*, 1991]. Factored planning may take a different path: two recent approaches amount to simultaneously finding all plans (up to a given length) in different subdomains, then trying to merge them into a global solution [Amir and Engelhardt, 2003; Brafman and Domshlak, 2006]. They avoid backtracking at the cost of computing all possible plans for the subdomains, which may be potentially very expensive.

*Olivier Buffet is now at LAAS-CNRS, France.

Our approach is different in that our algorithm is a form of abstract planning which makes use of the factorisation of the problem. In this respect, it is more related to localised planning [Lansky and Getoor, 1995], having the same advantage of not being limited to domains with explicit hierarchical structure. A characteristic of our algorithm is that it uses a decomposition tree (dtree) [Darwiche, 2001], rather than a junction tree, for domain factorisation. The resulting algorithm, `dTreePlan`, is rather generic, and is a first attempt to design a new algorithm for factored planning with backtracking. It can benefit from different improvements and different choices for the underlying planner are possible.

We also present a particular implementation based on planning as satisfiability [Kautz and Selman, 1992], using `zChaff` as the low-level planner [Moskewicz *et al.*, 2001]. We explore several implementation details, mostly aiming at reducing backtracking, such as different forms of abstraction and the use of caching techniques. This results in automated algorithms both for factorisation and for planning. The experiments show encouraging results, and the analysis of `dTreePlan` leads to various directions where improvements are possible, making it a very promising algorithm.

Our setting and background knowledge on factored planning are presented in Section 2. The generic `dTreePlan` algorithm is introduced in Section 3. Details of the implementation and improvements are described in Section 4. Section 5 presents experimental results, followed by a discussion and conclusion.

2 Background

2.1 Planning Problem

Our planning problem formulation uses STRIPS operators with conditional effects and goals as conjunctions of literals. Yet, a large part of our algorithm extends to more complex settings (negative preconditions, multi-valued variables...). Only the restricted form of objective is a rigid assumption in this paper. The optimality criterion we consider is the length of the complete plan.

A running example we will use in the remainder of this paper is the *window problem*, described in Figure 1, where someone wants to throw a ball into a room without breaking the window.

Variables:

```
open ? {true, false}
broken ? {true, false}
ball ? {inside, outside}
```

Actions:

```
  Open:
(open=false)                 -> Set(open=true)
  Close:
(open=true & broken=false)   -> Set(open=false)
  Throw:
(open=true & ball=outside)   -> Set(ball=inside)
(open=false & ball=outside)  -> Set(ball=inside
                & broken=true & open=true)
```

Init state: `ball=outside & open=false & broken=false`
Goal state: `ball=inside & open=false & broken=false`

Figure 1: The window problem.

2.2 Factored Planning

Factored planning raises two questions: how to factor a given problem and how to plan using this factorisation. Factoring is the process of partitioning a domain into possibly overlapping *subdomains*. Then, planning starts by solving simple abstracted problems, and tries to merge their solutions or improve them to get solutions to more complex problems, up to solving the original problem.

To create subdomains, we need to cluster actions from the original domain. A subdomain d_i is then defined by:

1. the variables appearing in its cluster of actions c_i and
2. its actions, made of *real actions* (actions within the cluster) plus possibly *abstract actions* (abstracted versions of actions from some other clusters).

A sequence of consecutive abstract actions is a *hole* in a plan. In the window problem we consider one cluster per action. Abstract actions make it possible for the `Open` cluster to close the window.

We are mainly interested in composite systems, where a set of controllable components are interacting through shared variables (representing communications and possibly indirect interactions) and organised under a network topology. This leads us to a first suggestion for factoring: using components $c \in C$ as building blocks, seen as *clusters of actions* in the causal graph of the problem. This is an intuitively appealing choice as actions within a component are prone to act and depend on the same internal variables. Yet this does not tell us how this set of blocks should be organised. Indeed, we will not apply the same tree decomposition of the components' network as most other approaches.

Existing Factored Planners

Two recent factored planners are `PartPlan` [Amir and Engelhardt, 2003] and `LID-GF` [Brafman and Domshlak, 2006]. Both use a tree decomposition with actions distributed among all nodes, and plan recursively by computing all possible plans (with holes for other actions) in all subtrees of a node before planning in the current node by merging subplans and inserting actions local to the node. Among the incomplete plans produced in a node, many will turn out to be infeasible, if there is no possibility to complete them (fill their holes). This gives a dynamic programming algorithm since planning starts at the leaves and goes up to the root, where the plan to be executed is selected. The only form of backtracking is when restarting the algorithms with plans of length $l+1$ if no plan of length at most l was found.

Even though actions in lower levels of the tree are abstracted out in higher levels and many subplans disappear because they cannot be merged in some way, the approach has potentially huge memory requirements and its practicability has not been established yet. Moreover, the local planners (inside a given node) do not know anything about actions in non-descendant nodes, whereas some knowledge could help prune some infeasible plans: plans with holes that cannot be filled. Finally, these two planners extend the length of plans in all subdomains at the same time, leading to local optimality, but not guaranteeing minimum length of the complete plan [Brafman and Domshlak, 2006].

3 Algorithm

3.1 An Abstract Planner

We first introduce a particular abstract planning procedure, using an arbitrary ordering over the action clusters $c_1, \ldots, c_{|C|}$ (supposed to be already selected). The planning process consists in planning in these clusters in order (c_1 being the root). In cluster c_i, a planning problem is defined by:

Actions = c_i's real actions, plus abstract actions replacing actions from $c_{i+1}, \ldots c_{|C|}$ when in c_i.

Variables = Variables appearing in c_i's real actions only. This is not necessarily a subset or a superset of variables in other clusters.

Goal = For c_1, the goal is defined by the original problem. For any other cluster c_i, the goal is to fill a set of holes $\Lambda = \{\lambda_1, \cdots, \lambda_{|\Lambda|}\}$ passed on by c_{i-1} and projected on c_i's variables.

Actions and variables define the *subdomain at level* i, which depends on c_i and on the clusters down the tree.

The complete plan can be seen as a tree in which a node is a plan whose holes are completed by children subplans. Depending on whether c_{i-1} passes on its holes to c_i one at a time or all at once, the process is either a depth-first or a breadth-first traversal. If no plan is found that can be completed by child c_i, a failure is signified to its parent c_{i-1}. The window problem (Figure 2a) is easy to solve if the `Throw` cluster is c_1, as it quickly produces a first plan with `Throw` preceded by some abstract action making `open` true and followed by another one making `open` false.

The depth-first version is detailed in Algorithm 1, omitting projections on subdomains. Taking a hole λ_{i-1} given by c_{i-1} (except when $i = 1$), c_i tries each plan π_i made of c_i's real and abstract actions and satisfying λ_{i-1} until π_i's holes can be filled by FillHoles(). Each call of PlanForHole() returns a new plan, up to exhaustion. Subplans returned by FillHoles() are attached to π_i's holes (as children in a tree). AbstractReplan() and ReFillHoles() are used when FillHoles() has to backtrack. They make it possible to look for the next plan (and subplans) filling a given hole. The loop stops when c_i returns a valid plan with subplans, or when there is no more plan to try.

Algorithm 1: Abstract Planner (Depth-First)

```
AbstractPlan(λ_{i-1}: hole passed on by c_{i-1})
if isLeaf(this) then return PlanForHole(λ_{i-1})
repeat
    π_i ← PlanForHole(λ_{i-1})
    if failure(π_i) then return failure
    Π' ← FillHoles(π_i.holes())
until ¬ failure(Π')
return π_i.attach(Π')
```

```
FillHoles(Λ_{i-1}: list of holes)
if isEmpty(Λ_{i-1}) then return ∅
π ← child.AbstractPlan(Λ_{i-1}.head())
if failure(π) then return failure
Π' ←FillHoles(Λ_{i-1}.tail())
while failure(Π') do
    π ← child.AbstractReplan(Λ_{i-1}.head())
    if failure(π) then return failure
    Π' ←FillHoles(Λ_{i-1}.tail())
return {π} ⋃ Π'
```

```
AbstractReplan(λ_{i-1}: hole passed on by c_{i-1})
if isLeaf(this) then return PlanForHole(λ_{i-1})
Π' ← ReFillHoles(π_i.holes())
while failure(Π') do
    π_i ← PlanForHole(λ_{i-1})
    if failure(π_i) then return failure
    Π' ← FillHoles(π_i.holes())
return π_i.attach(Π')
```

```
ReFillHoles(Λ_{i-1}: list of holes)
if isEmpty(Λ_{i-1}) then return failure
Π' ←ReFillHoles(Λ_{i-1}.tail())
while failure(Π') do
    π ← child.AbstractReplan(Λ_{i-1}.head())
    if isEmpty(π) then return failure
    Π' ←FillHoles(Λ_{i-1}.tail())
return {π} ⋃ Π'
```

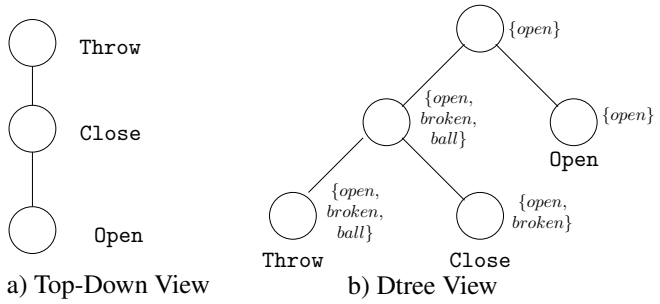
From now on, the clusters visited after (before) cluster c_i are called *future* (*past*) *clusters* for c_i.

3.2 Ordering the Action Clusters

To make Algorithm 1 more efficient, it is very important to choose the visiting order of clusters (which specifies the subdomains). The ordering could be dynamic: which cluster to visit next could depend on which variables appear in the holes to fill. But we will focus on fixed ordering in this work.

Choosing an ordering requires analysing the actions and the variables they depend on. We have already mentioned that subdomains should regroup real actions whose preconditions and effects are linked to the same group of variables (here using actions within a cluster). But if two partitions of a domain should share as few variables as possible, the *ordering* of the resulting subdomains should be such that two consecutive subdomains share as many variables as possible, with a view to early backtracking by quickly identifying unsatisfiable constraints.

To sort the clusters we propose using a *decomposition tree* (dtree) [Darwiche, 2001], which recursively splits the original domain into subdomains up to leaves corresponding to individual clusters of actions. A dtree's subtrees can be seen as neighbourhoods, which are a good basis for finding an ordering on its leaves. Dtrees correspond to *branch decompositions* known in graph theory [Robertson and Seymour, 1991], and were used in [Darwiche, 2001] to specify recursive decompositions of a Bayesian network down to its families. In [Huang and Darwiche, 2003], dtrees were used to specify recursive decompositions of a CNF formula down to its clauses. For our purposes here, a dtree is a full binary tree whose leaves correspond to clusters of actions of the planning problems; see Figure 2b, where all clusters (leaves) contain a single action and each node is annotated with the variables shared between its subtrees. We assume that the children of a dtree node will be visited from left to right.



a) Top-Down View b) Dtree View

Figure 2: Two views of our running example

To generate a dtree, we use the tool described in [Darwiche and Hopkins, 2001], which employs the technique of *hypergraph partitioning*. Our hypergraph is constructed by having a node for each action and a hyperedge for each variable, connecting all nodes (actions) in which this variable appears. The hypergraph is then recursively partitioned into two (balanced) parts while attempting to minimise the number of edges across. The resulting dtree is expected to have relatively small cutsets, i.e., few variables shared between subtrees.

Such a dtree specifies a recursive decomposition of the domain down to actions. However, one will generally wish for a subdomain to contain more than just a single action. This can be accomplished simply by regarding an internal node of the dtree as a subdomain (containing all the actions represented by the leaves under that node). Specifically, we implemented a clustering process where, given a dtree of depth d and a clustering level $cl \in [0, 1]$, all dtree nodes at depth $(1 - cl) \times d$ are treated as subdomains in which no further decomposition takes place. In Section 5 we also experiment with an alternative scheme where action clusters correspond to components in the original planning problem, and a dtree is built with these pre-formed clusters as leaves.

Reordering Subtrees

As explained above, a dtree only specifies neighbourhood relationships between leaves. Since we assume that children of a dtree node are visited from left to right, flipping the children of any node produces a different ordering of subdomains. To decide on the order of children we employ a simple heuris-

tic inspired from goal-regression. It takes the list of variables appearing in the goal definition and rearranges the dtree to put the cluster able to modify most of them in the first visited (leftmost) leaf. Then the process is repeated with the list of variables appearing as (pre)conditions in this first cluster, so as to place the next "best" cluster on the second leaf. This means that, assuming the `Close` node on Figure 2 was a complex subtree and leaf `Throw` had just been placed at the leftmost position, variables appearing in `Throw`'s preconditions would be used in `Open`'s subtree to position its best leaf. This is a simple recursive process happening while traversing the tree (and rearranging it on the way); see Algorithm 2.

Algorithm 2: reOrder

```
REORDER()
goalVars ← variables in the goal definition
reOrderRec(goalVars)
REORDERREC(vars: list of variables)
if isLeaf(this) then return this.preCondVars()
#l ←nSharedVars(vars, leftChild.modifiedVars())
#r ←nSharedVars(vars, rightChild.modifiedVars())
if #l < #r then swap(leftChild, rightChild)
vars' ←reOrderRec(leftChild, vars)
vars" ←reOrderRec(rightChild, vars')
return vars"
```

3.3 `dTreePlan`

Visiting the leaves in order can be achieved by a simple dtree traversal. The result is shown in Algorithm 3, where we omit ReDTreePlan() and ReFillHoles(). dTreePlan returns a plan-tree whose leaves may have unfilled holes. π.holes() returns these holes in an ordered list. Note that the traversal of the plan—seen as a tree—is neither a depth-first nor a breadth-first traversal: holes are refined one at a time within each "neighbourhood" (sub-dtree).

4 Implementation Details, Improvements

This section presents a particular implementation of `dTreePlan` used for experimentation and including several improvements.

4.1 Abstract Actions

Abstract actions can be more or less informative regarding future clusters' abilities. We have defined two types of abstract actions with a view to evaluating their relative efficiency. Definitions below take the point of view of one cluster (one level of abstraction):

- **H(ole) actions:**[1] Actions changing any variable shared with future clusters. One way of implementing them is to create one action violating the frame axiom in that any shared variable can get any value after this action. This action should not have any preconditions and should not modify any other variable.

[1]Hole actions are quite similar to fluent-setting actions in [Amir and Engelhardt, 2003].

Algorithm 3: dTreePlan (Depth-First)

```
DTREEPLAN(λ: hole passed on by parent)
if isLeaf(this) then return PlanForHole(λ)
π_l ← leftChild.dTreePlan(λ)
if failure(π_l) then return failure
Π_r ← FillHoles(π_l.holes())
while failure(Π_r) do
    π_l ← leftChild.ReDTreePlan(λ)
    if failure(π_l) then return failure
    Π_r ← FillHoles(π_l.holes())
return π_l.attach(Π_r)
FILLHOLES(Λ': list of holes)
if isEmpty(Λ') then return ∅
π_l ← rightChild.dTreePlan(Λ'.head())
if failure(π_l) then return failure
Π' ←FillHoles(Λ_{i-1}.tail())
while failure(Π') do
    π ← rightChild.ReDTreePlan(Λ_{i-1}.head())
    if failure(π) then return failure
    Π' ←FillHoles(Λ_{i-1}.tail())
return {π} ⋃ Π'
```

- **V(irtual) actions:** Abstracted versions of the actions in future clusters (with non-shared variables removed). This is a many-to-one mapping from real actions in future clusters to abstract actions in the current cluster.

Note: "hole" = "sequence of abstract actions" ($\neq$ "H action").

Other types of abstract actions could be proposed. H actions have the advantage of not requiring any knowledge of future clusters apart from the variables they share with the current cluster. This could help design a planning process going through independent components who want to keep their capabilities private. V actions are much more informative, which should help avoid infeasible holes, but which also brings us closer to central planning.

4.2 Tree Traversal

We use the depth-first algorithm because it makes use of the domain factorisation: it turns out to be a compromise between refining one hole at a time and working preferably within neighbourhoods. Note that the breadth-first version is strictly equivalent to a classical breadth-first abstract planner.

The tree traversal also depends on π.holes(). This function can return one hole for each sequence of consecutive abstract actions, or for each abstract action. Although the former seems more promising to reduce backtracking, our experiments use the latter.

4.3 Caching

When backtracking, it will be quite common for a given node to be asked several times to solve the same subproblem. A good way to avoid such replanning is to cache solutions to such subproblems, or to keep track of subproblems which cannot be solved. The strategy we adopted is mixing both ideas:

- in each leaf node are stored all already computed plans for each already encountered subproblem, and

- in each internal node are stored all encountered subproblems known to be unsolvable.

The latter is less informative and thus produces smaller savings, but helps reduce space complexity. Note that caching all plans and using H actions would lead to an algorithm sharing similarities with `PartPlan` or `LID-GF` as it would compute subplans only once and store them in memory.

4.4 Planning as Satisfiability

SAT planners, as introduced in [Kautz and Selman, 1992], encode a planning problem into a propositional formula to be solved by a SAT solver. We adopt this approach in the implementation of `dTreePlan`, using the publicly available `zChaff` SAT solver [Moskewicz *et al.*, 2001].

The choice of a SAT planner as the low-level planner has several advantages. First, it makes it possible to give an incomplete plan as a problem for some subdomain: this amounts to specifying values for some variables at appropriate time steps. Second, H actions are easily encoded by SAT by locally removing the frame axiom for variables shared with future clusters: it suffices to specify that their values at time t are not related to their values at time $t+1$. Third, one can use abstract actions placed by past clusters as constraints, requiring that the action replacing it matches its definition.

Controlling the plan length

As is typical with SAT planners, we search for increasingly longer plans while no solution is found. This guarantees optimality while `PartPlan` and `LID-GF` only guarantee local optimality.

Moreover, as the need for backtracking partially depends on the number of holes generated in each plan, `dTreePlan` also iterates over an increasing number of abstract actions in each subplan. This increase is not linear, as it would lead to adding a huge number of clauses. Instead, the algorithm plans with (1) no abstract action, (2) one abstract action, and then (3) any number of abstract actions.

5 Experiments

5.1 Improvements

A first set of experiments has aimed at evaluating the effect of the different improvements added to our basic `dTreePlan` algorithm (with H actions). It uses the Robot-Charger problem described in [Amir and Engelhardt, 2003], where a robot wants to upgrade its battery charger but needs to charge its batteries while achieving this goal. We designed variants by changing the number of batteries and lines (named x**bat**y**line**).

Table 1 reports computation times (in seconds averaged over 100 runs) on **1bat1line** with a clustering level of 0.5 and the following versions of `dTreePlan`:

Basic basic dTreePlan, with no improvement from Section 3.
IncrNAct Basic, with incremental number of actions.
IncrMaxH IncrNAct with incremental number of H actions.
Caching IncrMaxH with plan caching.

The reOrder() function was not used, so that four different tree shapes were possible, denoted by the order of visited clusters: Switch (S), Connect+AddLine (CA), Charge (C).

Table 1: Effect of various improvements

Algorithm	Dtree Shape			
	S(CA)C	CS(CA)	(CA)CS	C(CA)S
Basic	1.950	4.463	8.993	107.186
IncrNAct	1.036	2.698	25.655	6.731
IncrMaxH	1.169	2.645	24.613	6.447
Caching	0.898	1.159	2.332	1.263

As can be observed, each improvement clearly speeds up the planning process, except the increase in the number of holes, where the gain is limited. The main exception is with dtree shape (CA)CS, where the vanilla `dTreePlan` is made slower by IncrNAct and IncrMaxH. Finally, it is important to be aware that the reOrder() procedure would produce dtree shape S(CA)C, which proves to be the most efficient.

5.2 Various Planning Parameters

Having decided to use the improvements mentioned in previous section, we conducted a second set of experiments to evaluate `dTreePlan` with different parameters. In Table 2, each column presents planning time (in seconds) for a given clustering level (2 = component clustering) and a given type of abstract action (H = H action, V = V action). The last column shows results with central planning (C): `zChaff` used with no factorisation. The first column indicates the number of batteries and segments in the Robot-Charger problem considered.

Table 2: Effect of clustering level and type of abstract action

$\#_{batt}$ x$\#_{seg}$	Planning Parameters					
	0 H	0.5 H	0 V	0.5 V	2 V	C
1x1	0.71	0.38	0.43	0.51	0.36	0.37
1x2	-	0.51	0.61	0.92	0.40	0.47
2x1	2.55	0.49	0.43	0.43	0.63	0.49
2x2	-	0.65	0.49	0.49	1.90	0.81
1x3	-	0.85	1.71	3.19	0.37	0.73
2x3	-	1.24	0.85	0.85	26.48	1.94
3x1	99.42	4.09	0.97	0.92	2.41	1.95
3x2	-	5.28	1.09	1.03	15.26	3.32
3x3	-	13.48	1.63	1.52	-	6.08
3x4	-	20.04	4.20	4.03	-	13.15
4x3	-	-	13.33	12.38	-	-
4x4	-	-	16.50	15.60	-	-
4x5	-	-	30.71	30.03	-	-

On small domain instances, most parameter settings appear more or less equivalent. Only clustering level 0 with H actions has bad performance from the beginning. Clustering level 0 or 0.5 with V actions appears to scale better than the centralised algorithm, even though their planning time appears unimpressive on some instances (as 1x3). This confirms that V actions are a good means to prune out infeasible plans. This experiment also shows that components may not be an effective basis for decomposition. Finally, unknown values (-) correspond to durations over 2 minutes or failures due to memory exhaustion (probably linked to our caching strategy).

6 Discussion, Future Work

`dTreePlan` is a first attempt to design a new algorithm for factored planning with backtracking. This paper has presented a specific implementation based on SAT planning together with several improvements. The experiments presented in previous section show that there is an advantage over direct central planning if action clusters are visited in an appropriate order.

A first direction for improvements is that of rearranging the dtree. As discussed in Section 3, a dtree does not define the ordering of subtrees: it specifies neighbourhood constraints for the leaves rather than a precise ordering. It has been found particularly useful to choose the ordering with a very crude heuristic; hence an interesting avenue is to make use of more knowledge of the planning problem to rearrange the dtree. As in [Helmert, 2004; Brafman and Domshlak, 2006], the causal graph of a planning problem seems to be a good starting point for that. The order could also be changed on the fly, since the current plan could suggest following the right child of a node first, rather than the left one. But this would require recomputing abstract actions for each cluster as they mainly depend on future clusters.

In the `dTreePlan` framework, both the caching strategy and the SAT planner used could be replaced by alternative solutions. For example, it may not be necessary to store complete feasible plans: one may just store the holes they generate—the only thing required when searching for a solution—and replan when the details of a stored plan are required. It is also possible to replace `zChaff`—which is a generic SAT solver— by a SAT planner such as SATplan or MAXplan.

A final possible improvement we mention is to compute a lower bound on the plan duration. Given a node of the dtree and its two children, a lower bound on the node's plan length can be computed by planning in both children on problem restricted to their internal (non-shared) variables, or more precisely to variables no one else can modify. Adding the resulting plan lengths gives a lower bound for the node. This process can even be applied recursively from the leaves to the root. A possible heuristic is also to estimate the overall plan length by blindly adding the length of plans computed within leaves using the overall goal and V actions in each case.

7 Conclusion

We have presented `dTreePlan`, a generic factored planning algorithm aiming at avoiding the space complexity of dynamic programming approaches and using a dtree for factorisation. Preliminary results show that a SAT-based implementation scales better than raw SAT planning. The algorithm is complete, and could still benefit from major improvements.

This work also illustrates the connection between abstract and factored planning. Apart from direct improvements, research directions include the management of more complex goals (objectives specified with temporal logic), the use of non-deterministic/stochastic actions, and the problem of distributing a plan over several components for execution.

Acknowledgments

This work has been supported in part via the SuperCom project at NICTA. NICTA is funded through the Australian government's *Backing Australia's Capabilities* initiative, in part through the Australian research council.

References

[Amir and Engelhardt, 2003] E. Amir and B. Engelhardt. Factored planning. In *Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[Brafman and Domshlak, 2006] R. Brafman and C. Domshlak. Factored planning: How, when and when not. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence (AAAI'06)*, 2006.

[Darwiche and Hopkins, 2001] A. Darwiche and M. Hopkins. Using recursive decomposition to construct elimination orders, join trees, and dtrees. In *Proceedings of the Sixth European Conference for Symbolic and Quantitative Approaches to Reasoning under Uncertainty (ECSQARU'01)*, 2001.

[Darwiche, 2001] A. Darwiche. Recursive conditioning. *Artificial Intelligence Journal*, 125(1–2):5–41, 2001.

[Helmert, 2004] M. Helmert. Planning heuristic based on causal graph analysis. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS'04)*, pages 161–170, 2004.

[Huang and Darwiche, 2003] J. Huang and A. Darwiche. A structure-based variable ordering heuristic for SAT. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI'03)*, 2003.

[Kautz and Selman, 1992] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI'92)*, 1992.

[Knoblock *et al.*, 1991] C.A. Knoblock, J.D. Tenenberg, and Q. Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI'91)*, 1991.

[Lansky and Getoor, 1995] A.L. Lansky and L.C. Getoor. Scope and abstraction: Two criteria for localized planning. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI'95)*, pages 1924–1930, 1995.

[Moskewicz *et al.*, 2001] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the Thirty-Eighth Conference on Design Automation*, pages 530–535, 2001.

[Robertson and Seymour, 1991] N. Robertson and P. D. Seymour. Graph minors X: Obstructions to tree-decomposition. *Journal of Combinatorial Theory, Series B*, 52(2):153–190, 1991.