Fast Planning by Search in Domain Transition Graph*

Yixin Chen, Ruoyun Huang, and Weixiong Zhang
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130, USA
{chen,rh11,zhang}@cse.wustl.edu

Abstract

Recent advances in classical planning have used the SAS+ formalism, and several effective heuristics have been developed based on the SAS+ formalism. Comparing to the traditional STRIPS/ADL formalism, SAS+ is capable of capturing vital information such as domain transition structures and causal dependencies. In this paper, we propose a new SAS+ based incomplete planning approach. Instead of using SAS+ to derive heuristics within a heuristic search planner, we directly search in domain transition graphs (DTGs) and causal graphs (CGs) derived from the SAS+ formalism. The new method is efficient because the SAS+ representation is often much more compact than STRIPS. The CGs and DTGs provide rich information of domain structures that can effectively guide the search towards solutions. Experimental results show strong performance of the proposed planner on recent international planning competition domains.

Introduction

The SAS+ formulation of planning problems has drown increasing attention recently in classical planning (Bäckström & Nebel 1995). In contrast to the traditional STRIPS formalism, the SAS+ formulation can provide compact constructs such as domain transition graphs (DTGs) and causal graphs (CGs) to capture vital domain information of domain transition structures and causal dependencies.

Many applications of the SAS+ formalism have been studied. In the Fast Downward planner (Helmert 2006), a heuristic function was developed by analyzing the causal graphs on top of the SAS+ models. Another SAS+ based heuristic for optimal planning was recently derived via a linear programming model encoding DTGs (van den Briel *et al.* 2007). Moreover, long-distance mutual exclusion (mutex) constraints based on a DTG analysis was proposed and shown to be effective in speeding up SAT-based optimal planners (Chen, Zhao, & Zhang 2007).

However, one limitation of the existing methods using the SAS+ formulation is that their high-level problem-solving strategy is searching for solution graphs or plans in STRIPS state spaces. In other words, the previous works focused mainly on deriving new heuristics or mutex constraints, while exploring a search space where each state is represented by binary STRIPS facts. It is critical to mention that a STRIPS state space can be very large for a typical planning problem. In particular, the number of binary facts (F) in a planning problem is typically in the order of 10^3 to 10^4, and the size of the state space is $\Omega(2^F)$, resulting in huge time and memory complexities in the worst case.

Although the compact constructs provided by the SAS+ formulation have been used to develop effective heuristics to speed up search (Helmert 2006; Chen, Zhao, & Zhang 2007), the domain transitions and causal dependencies encoded in SAS+ have not been fully exploited. Our research was motivated by the possibility of searching directly in graphs composed of DTGs which are further inter-connected by CGs. We are inspired by the observation that the compact constructs from the SAS+ formulation can give rise to small, compact graphs, which can be substantially smaller than a binary-fact state space. In particular, the size of any DTG for a given problem is often very small. The number of DTGs is typically in the order of 10 to 100, and the number of nodes in a DTG varies from a few to around 20 for a problem in the recent years' planning competitions. Therefore, the size of each DTG is small and a direct search on DTGs can be efficient.

On the other hand, the search of a plan cannot be completely decomposed into searching individual DTGs. DTGs can depend on one another due to causal dependencies, which lead to complex orderings among actions. Hence, causal dependencies are indeed the culprit of the difficulty of automated planning. One possible approach is to merge individual DTGs under the constraints of their causal dependencies, while maintaining the overall graph as small as possible so as to make the search process efficient. However, an effective DTG merging scheme seems to be difficult to come by. This certainly calls for a new approach to utilize DTGs and deal with their causal dependencies.

In this paper, we propose a new planning algorithm that directly searches in a space of DTGs and CGs rather than a binary-fact search space. Based on the DTG structures, our algorithm directly extracts plans from a graph composed of DTGs. We distribute the decision making into several hierarchical levels of search to deal with causal dependencies. At each level, we design heuristics to order branching choices

*This work was supported by Microsoft New Faculty Fellowship and NSF grants IIS-0713109, ITR-0113618 and IIS-0535257.
Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

and prune nonpromising alternatives. We show that the direct search of DTGs can work well across a variety of planning domains, showing competitive performance.

The proposed method has at least two advantages over the traditional heuristic search algorithms on STRIPS models such as FF (Hoffmann & Nebel 2001). First, unlike the popular relaxed-plan heuristic that ignores delete effects, the DTGs preserve much structural information and help avoid deadends in problems from many domains. Second, the proposed method introduces a hierarchy of decision-making where heuristics can be accordingly designed for different levels of granularity of search. In contrast to requiring a single good heuristic in planners such as FF, the proposed method provides an extensible framework in which heuristics and control rules can be designed, incorporated and improved at multiple levels.

The proposed algorithm is based on similar ideas as that of previous hierarchical decomposition planning algorithms such as HTN planning. However, the new algorithm it is fully automated and does not require domain-specific control knowledge to specify how an action can be decomposed.

Background

In STRIPS planning, a **fact** f is an atomic fact that can be true or false. Given a set of facts $F = \{f_1, f_2, \ldots, f_n\}$, a STRIPS state S is a subset of facts in F that are set to true. An **action** a is a triplet a = (pre(a), add(a), del(a)), where pre(a)$\subseteq F$ is the set of preconditions of action a, and add(a)$\subseteq F$ and del(a)$\subseteq F$ are the sets of add facts and delete facts, respectively. A **planning task** is a triplet $(O, S^{initial}, S^{goal})$, where O is a set of actions, $S^{initial} \subseteq F$ the initial state, and $S^{goal} \subseteq F$ the goal state.

The SAS+ formalism (Helmert 2006; Jonsson & Bäckström 1998) of a planning domain includes a set of multi-valued **state variables**, where a variable represents a group of mutually exclusive facts from which only one can be true in any state. We denote the set of state variables as $X = (x_1, x_2, \ldots, x_m)$, where x_i takes a value from a finite discrete set $\mathcal{D}_i$. For a SAS+ task, the value assignment of a state variable corresponds to a binary fact in the traditional STRIPS formalism.

Definition 1 (Domain transition graph (DTG)) *Given a state variable $x \in X$ defined over $\mathcal{D}_x$, its DTG G_x is a directed graph with vertex set $\mathcal{D}_x$ and arc set $\mathcal{A}_x$. A directional arc (v, v') belongs to $\mathcal{A}_x$ if and only if there is an action o with $v \in del(o)$ and $v' \in add(o)$, in which case we say that there is a **transition** from v to v'. We use $T_{v,v'}$ to denote the set of actions that can **support** the transition from v to v': $T_{v,v'} = \{o | v \in del(o), v' \in add(o)\}$.*

For simplicity, we assume that each fact exists in only one DTG. A domain violating this assumption can be transformed into one satisfying it (Helmert 2006). Given a STRIPS fact f, if it corresponds to a state variable assignment in a DTG G, we denote it as $DTG(f) = G$, or simply $f \in G$. Given any two vertices $f, h \in G$, we define the **minimum DTG cost** $\Delta_G(f, h)$ as the distance of the shortest path from f to h in G.

There may exist **causal dependencies** between DTGs. Consider two DTGs G and G'. If an action o in G' has a precondition f in G, denoted as $DTG(f) = G$, we say G' depends on G. We denote $dep(G)$ as the set of DTGs depending on G.

Given a SAS+ planning domain with state variable set $X = (x_1, x_2, \ldots, x_m)$, each state variable x_i corresponds to a DTG G_{x_i}. A state corresponds to a complete value assignment to all the state variables. In the following, we use states to refer to SAS+ states. Given a state S, we use $\pi(G, S)$ to specify the fact that is true in the DTG G.

For a DTG G and two facts f and h in G, we define the **transition path set** $\mathcal{P}(G, f, h)$ to be the set of all possible paths from f to h in G, with the restriction that each vertex can appear at most once in a path.

The DTG Planning Algorithm

The task of extracting a plan from DTGs can be viewed as selecting a sequence of actions from the DTGs. At the top level, for each subgoal $g_i, i = 1..N$, we find their DTGs $DTG(g_i)$, then find a sequence of transitions in $DTG(g_i)$ that reaches g_i from the initial state. The transitions for various subgoals may need to be interleaved instead of sequentially concatenated. We need to find a proper way to interleave these transitions. For each selected transition, we need to select an action to materialize the transition. To satisfy the preconditions of the action, we may need to insert a sequence of necessary "supporting actions" before the action.

As described above, there are three major components in our search: 1) selecting a path of transitions between two nodes in a DTG, 2) deciding the execution order of facts (subgoals or preconditions), and 3) selecting an action for a transition. To speedup the search, we use various heuristics to help choose promising directions and reduce the branching factors. Since we prune many branching choices at the three places above, our algorithm is an incomplete search.

Finding transition paths for subgoals

The top-level procedure search_goals(), listed in Algorithm 1, finds a high level plan of transitions for achieving all subgoals. We emphasize that the approach here is not incremental planning that concatenates sequentially plans for subgoals. Instead, the approach takes the interactions between different subgoals into consideration.

Before the searching starts, we order the subgoals using a **forced ordering** method (Koehler & Hoffmann 2000). In the forced ordering, a subgoal f is placed before h if it is impossible to reach f without invalidating h. We use $f \prec h$ to denote the forced ordering relationship. We generate an ordered list $subgoal_list$ that honors all forced orderings during preprocessing.

Each call to search_goals() (Algorithm 1) works as follows. We choose the first unsatisfied subgoal g on the $subgoal_list$, find its DTG $G = DTG(g)$, and generate the transition path set $\mathcal{P}(G, \pi(G, S), g)$, where $\pi(G, S)$ is the value of G in current state S (Lines 3-5). Rather than completing a path from S to g, we only try to move one transition further towards g each time we call search_goals() (Lines 6-9). The search_transition() procedure tries to find a concrete

Algorithm 1: search_goals(S)

Input: a state S
Output: a plan sol that reaches all subgoals from S; upon exit, S is the state after executing sol

1 **if** *all subgoals are true in* S **then** return $\{\}$;
2 **foreach** g *in the ordered subgoal_list* **do**
3 **if** g *is true in* S **then** continue;
4 $G \leftarrow DTG(g)$;
5 **foreach** $p \in \mathcal{P}(G, \pi(G, S), g)$ **do**
6 let $T_{u,v}$ be the first transition along p;
7 $S' \leftarrow S$;
8 $sol_1 \leftarrow$ search_transition(S',G,$T_{u,v}$,
9 forced_pre($T_{u,v}$,p));
10 **if** sol_1 *is not valid* **then** continue;
11 **if** $v = g$ **then** $protect_list \leftarrow protect_list \cup \{v\}$;
12 sol_2 = search_goals(S') ;
13 **if** $v = g$ **then** $protect_list \leftarrow protect_list \backslash \{v\}$;
14 **if** *plan* $sol2$ *is valid* **then**
15 $sol \leftarrow sol_1 + sol_2$;
16 $S \leftarrow S'$;
17 return sol;
18 return $no_solution$;

Algorithm 2: search_transition(S, G, T, forced_pre)

Input: a state S, a DTG G, a transition T, a set of facts forced_pre
Output: a plan sol that realizes T from S; upon exit, S is the state after executing sol

1 let O be the set of actions supporting T;
2 sort all $o \in O$ by $cost(o)$;
3 **foreach** *action* o *in* O **do**
4 **if** $\exists f, f \in del(o), f \in protect_list$ **then** continue;
5 F = { f | $f \in$ pre(o) or $f \in$ forced_pre };
6 $S' \leftarrow S$;
7 $sol \leftarrow$ search_fact_set(S' , F);
8 **if** sol *is valid* **then**
9 $S \leftarrow S'$;
10 apply the effects of o to S;
11 $sol = sol$ + $\{o\}$;
12 return sol;
13 return $no_solution$;

plan that realizes the transition $T_{u,v}$. There are two possibilities. If the transition $T_{u,v}$ can be carried out, we then update the state and recursively call search_goals() to complete the search (Line 12). If it is impossible to make the transition $T_{u,v}$ from the current state, we backtrack and choose another path in $\mathcal{P}(G, \pi(G, S), g)$. If no path in $\mathcal{P}(G, \pi(G, S), g)$ can have its first transition materialized, we try the next goal on the $subgoal_list$. Any subgoal, when achieved, will be added to a $protect_list$ to avoid being invalidated before backtracking on that subgoal.

The significance of using the forced goal ordering is as follows. The search_goals() procedure allows subplans for subgoals to be interleaved when necessary. However, the $subgoal_list$ assigns priorities to the subgoals. The search_goals() procedure can be viewed as a depth-first search that tries to meet the subgoals ordered earlier in $subgoal_list$ whenever possible. When the subgoals are ordered properly, it is often possible to generate plans for each subgoal sequentially with little backtracking. In this case, the solution plan will be close to those generated by incremental planning. However, search_goals() offers a complete search that can interleave the plans for subgoals for difficult problems where the subgoal interactions need to be considered.

A key observation is that, in almost all the IPC domains, the goal-level DTGs all have smaller size than other DTGs and the $\mathcal{P}(G, \pi(G, S), g)$ set often contains very few paths. In fact, many goal-level DTGs contain only one path to the subgoal and the paths are typically short. Therefore, the complexity of search_goals() tends to be low.

Materializing a transition

We now consider search_transition(), listed in Algorithm 2. It takes as parameters a given state S, a DTG G, a transition T, and a set of forced preconditions to be explained shortly. In this procedure we choose an action o to materialize the transition T. Before executing o, its preconditions and forced preconditions must also be true. Therefore, the procedure typically returns a plan consisting of a sequence of actions achieving the preconditions, followed by the action o at the end.

A transition usually has a number of supporting actions, from which one must be chosen. To choose one action, we sort all supporting actions by a heuristic value in Line 2 of Algorithm 2. Given the transition T in a DTG G, we evaluate each of its supporting actions by estimating the number of steps needed to make all its preconditions true. Formally, given an action o and the current state S, we estimate the cost to enable o as the total minimum DTG cost of all preconditions of o:

$$cost(o) = \sum_{\forall f \in pre(o)} |\Delta_G(\pi(G, S), f)|$$

. In search_transition(), we sort all supporting actions in an ascending order of their costs. Actions with lower costs are tried earlier because their preconditions are likely to be easier to achieve.

Forced preconditions We denote the set of forced preconditions as forced_pre(T,p). When materializing a transition T in a path p in a plan, it is beneficial to look ahead at the full path p. A wrong plan for the transition may lead the search into a deadend. Although backtracking can solve this problem, it is more efficient to detect such situations beforehand and avoid deadends.

Consider an example of the Blocksworld domain shown in Figure 1. There are three blocks A, B and C. Figure 1.i) illustrates the initial state, in which A and B are on the table and C is on B. The goal is "ON A B". Figure 1.ii) shows a part of the DTG of block A. Because of the transition edges among the three vertices, they must be true at different time steps.

Suppose we make the first transition (Pickup A TABLE),

Algorithm 3: search_paths(S, G, f_1, f_2)

Input: a state S, a DTG G, facts f_1,f_2
Output: a plan sol; upon exit, S is the state after executing sol;

```
1  if f1 ∈ protect_list then return no_solution;
2  find out P(G, f1, f2);
3  foreach p ∈ P(G, f1, f2) do
4      compute forced_pre(T, p) for each transition T in p;
5      sol ← {} ;
6      foreach T ∈ p do
7          S' ← S;
8          sol ← sol+ search_transition(S' , G , T,
9                          forced_pre(T, p));
10         if sol is not valid then break;
11         S ← S';
12     if sol is valid then
13         return sol;
14 return no_solution;
```

Algorithm 4: search_fact_set(S, F)

Input: a state S, a set of facts F
Output: a plan sol that achieves facts in F from S; upon exit, S is the state after executing sol;

```
1  if all facts in F are true in S then
2      return {};
3  generate the partial orders O1, O2, and O3;
4  while new valid permutation exists do
5      sol ← {};
6      S' ← S;
7      foreach f, f ∈ F do
8          G ← DTG(f);
9          h ← π(G, S);
10         sol1 ← search_paths(S', G, h, f);
11         if sol1 is valid then
12             S ← S';
13             sol ← sol + sol1;
14         else
15             break;
16     if sol is valid then break;
17 return sol;
```

the state of block A becomes "HOLDING A". To achieve "ON A B", the next action in the path is (Put A B), which has two preconditions. The first precondition is "HOLDING A" which is true. The second precondition is "CLEAR B". Since C is on top of B, we need to remove C from B. But that requires the arm to be free, resulting in "HOLDING A" being invalidated. A deadend is thus encountered.

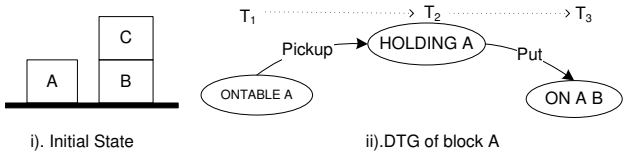


Figure 1: Initial state of an example and the DTG of block A

It is clear that the way to avoid this deadend is to achieve "CLEAR B" first before executing (Pickup A TABLE) to make "HOLDING A" true. In general, for each transition T, there may be some facts that need to be made true before an action materializing T is executed. Below we define the forced preconditions, a set of facts that need to be true before transition T is executed.

Given a path p of transitions in a DTG G, and a transition T on path p, we define the forced preconditions of T with respect to p as follows. Assume T turns a fact f to g, and then T_1, which is the next transition following T on path p, changes fact g to h. If there exists a fact q such that $q \in pre(o)$ for any action supporting T_1 and there is a forced ordering $q \prec g$ (Koehler & Hoffmann 2000), we call q a forced precondition of transition $T_{f,g}$. The set force_pre(T, p) includes all such forced preconditions of T.

The rationale for the above definition is the following. The fact q is required in order to execute T_1. We need to find a plan to satisfy q when we call search_transition() for T_1. However, in our case, it would be too late due to the $q \prec g$ ordering, which means that g has to be invalidated before achieving q. Thus, we need to achieve q *before* g. In search_transition(), a forced precondition, such as q, will be treated as a precondition for transition T (Line 5).

In the Blocksworld example, "CLEAR B" is a precondition of (Put A B), and there is a forced ordering "CLEAR B" $\prec$ "HOLDING A". Thus, we recognize it as a forced precondition of the transition from "ONTABLE A" to "ON A B". Computing forced preconditions can avoid many deadends during search. It is useful not only for Blocksworld, but also for other domains such as TPP and Rovers.

Search for a valid transition path

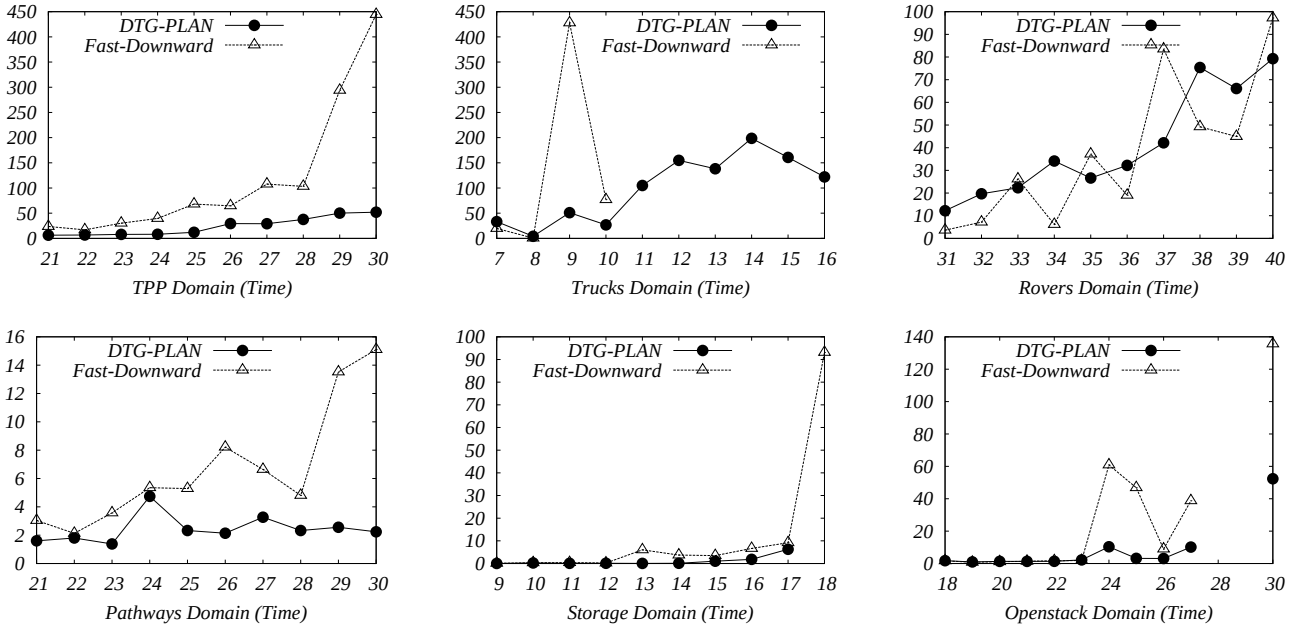
Given a state S, a graph G and two facts $f_1, f_2 \in G$, procedure search_paths() tries to find a valid transition path from f_1 to f_2. First, we compute the set of possible paths $\mathcal{P}(G, f_1, f_2)$. For each p in $\mathcal{P}(G, f_1, f_2)$, we first compute its forced preconditions and then traverse along p to form a plan (Lines 6-11). For each transition T in p, we call procedure search_transition() to search for a plan. A plan is returned when solutions to all transitions of a path p is found.

Search for a set of preconditions

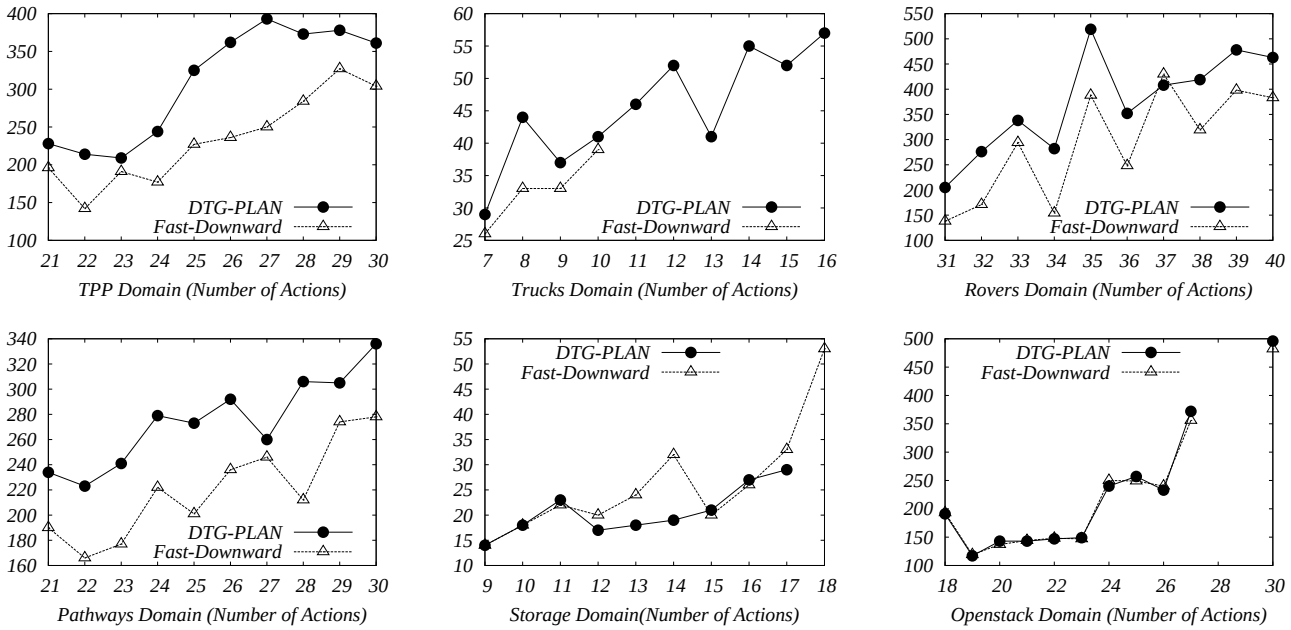
Given a state S and a set of facts F, starting at S, the procedure search_fact_set() searches for a valid plan that can make all facts $f \in F$ be true at a final state. The facts in F may require a particular order to be made true one by one.

For the facts in F, we derive the following partial orders between them, listed with a descending priority.

O1. For two facts $f, h \in F$, if $DTG(h) \in dep(DTG(f))$, we order f before h. To understand the reason, consider a transportation domain with a cargo and a truck. The DTG of a cargo depends on that of a truck. If we fix the truck at a location, then we may not be able to move the cargo without moving the truck. Thus, it makes sense to first deliver the cargo before relocating the truck.



a) Comparison of running time (in seconds).



b) Comparison of number of actions.

Figure 2: Experimental results on IPC5 domains.

O2. For each fact $f \in F$, we evaluate a degree-of-dependency function defined as

$$\beta(f) = |dep(DTG(f))| - |dep^{-1}(DTG(f))|,$$

where $dep^{-1}(G)$ is the set of graphs that G depends on. For two facts $f, h \in F$, if $\beta(f) > dep(DTG(h))$, we order f before h. This is a generalization of O1.

O3. For two facts $f, h \in F$, if there is a forced ordering $f \prec h$, we order f before h.

In search_fact_set(), we first consider all the permutations that honor the above partial orderings (Lines 3-4). In very rare cases, when all permutations that meet O1, O2 and O3 fail, we continue to compute the remaining orderings.

In some domains, an action may have a large number of preconditions that share the same predicates. Given two facts f and g, if f and g have the same predicate, and they differ by only one object arity, we call f and g **symmetric facts.** Given an action o, if there exist a number of facts $f_1, f_2, \ldots, f_n$, $f_i \in pre(o)$, such that every two f_i and f_j are symmetric facts, we ignore the differences between all of these facts when generating orderings. Based on the partial-order heuristic and symmetric-fact pruning, we have a good chance to succeed quickly. We usually get a feasible ordering among the first three candidates.

Experimental Results

We tested our new planner, DTG-PLAN, on the STRIPS domains of the 5th International Planning Competition (IPC5). We compared DTG-PLAN with Fast-Downward (Helmert 2006). Fast-Downward is one of the top STRIPS planners, which uses the DTGs in SAS+ formalism to compute the heuristic. Therefore, we can directly compare a method of directly searching in DTGs (DTG-PLAN) against a method using DTGs for deriving a heuristic (Fast-Downward). To carry out the comparison, we compiled the latest version of Fast-Downward on our computers, and ran all experiments on a PC workstation with a 2.0 MHZ Xeon CPU and 2GB memory and within a 500 second time limit for each run.

Table 1 summarizes the results in six out of seven STRIPS domains used in IPC5. Figure 2 gives details on some larger instances. For each domain, we show the ten highest numbered instances for which at least one of two methods can solve. DTG-PLAN currently does not have competitive performance on the seventh domain, Pipesworld. The reason is that solving this domain requires interleaving of paths not in goal-level DTGs but in other DTGs. We are developing a more general algorithm for handling such cases.

It is evident from the experimental results that DTG-PLAN is faster than Fast-Downward on most larger instances, sometimes by more than ten times. For instance, in the Trucks domain, DTG-PLAN can solve six large instances that Fast-Downward failed.

Regarding the number of actions, the two planners differ by less than 20% in most cases. DTG-PLAN is better on the Storage domain, while Fast-Downward is better on TPP, Pathways, and Rovers. In the current version of DTG-PLAN, we focused on heuristics that improve time efficiency. The fast speed of DTG-PLAN allows us to deliver high-quality anytime solutions for a given time limit.

	Instances		Average Time	
	DP	FD	DP	FD
OpenStack	28	28	3.39	10.91
Pathways	30	30	1.69	2.90
Rovers	40	40	12.05	9.69
Storage	17	18	0.58	1.78
TPP	30	30	9.16	47.16
Trucks	16	10	19.19	53.04

Table 1: The overall results of DTG-PLAN (DP) and Fast-Downward (FD) on IPC5 domains. The results are shown in terms of: 1) the number of instance solved by each approach and 2) the average time (sec) to solve one instance. Instances solved by both methods are used to calculate 2).

Conclusions

In this paper, we presented a new search algorithm for classical planning based on the SAS+ formalism. The algorithm directly searches for plans in a graph composed of DTGs. We distributed the decision making over several search hierarchies to deal with causal dependencies. For each level, we developed heuristics to order branching choices and to prune nonpromising alternatives. We experimentally showed that the strategy of direct search in DTGs can work well across a variety of planning domains. Comparison against a leading STRIPS planner showed that our method is competitive with the state-of-the-art heuristic planner that uses the SAS+ formalism for constructing heuristics.

The hierarchy of the proposed search algorithm may seem to be complex, but very often the search can extract a plan quickly along a depth-first path with little backtracking. Since the sizes of DTGs are generally small, the search can be very efficient with proper pruning and ordering heuristics.

The proposed work provides an extensible framework in which strategies can be designed and refined at multiple levels of search, leaving many opportunities for future improvement. Our study is the first attempt to develop search heuristics in the DTG space instead of the traditional state space. We are currently working on designing stronger heuristics for each hierarchical search level and studying plan merging techniques to improve plan quality.

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS+ planning. *Computational Intelligence* 11:17–29.

Chen, Y.; Zhao, X.; and Zhang, W. 2007. Long distance mutual exclusion for propositional planning. In *Proc. of International Joint Conference on Artificial Intelligence*, 1840–1845.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research* 14:253–302.

Jonsson, P., and Bäckström, C. 1998. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence* 100(1-2):125–176.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In *Proc. Constraint Programming Conference*.