DPPlan: an Algorithm for Fast Solutions Extraction from a Planning Graph

M. Baioletti, S. Marcugini and A. Milani
Dipartimento di Matematica e Informatica
Università degli Studi di Perugia
06100 Perugia - Italy
Tel. +39–075–5855049, Fax +39–075–5855024
e-mail: {marco, gino, milani}@dipmat.unipg.it

Abstract

The most efficient planning algorithms recently developed are mainly based on Graphplan system or on satisfiability approach. In this paper we present a new approach to plan generation based on planning graph analysis, which can be considered as a bridge between the two planning approaches.

The method exploits the propagation of planning axioms and constraints in order to make deductions on the planning graph and therefore to prune the search space. The consequences of decisions made during search have backward and forward impact on the planning graph. In contrast with Graphplan based backward algorithms, our approach allows to search the planning graph without committing to any specific direction. The experimental results obtained with DPPlan, a planner implementing the presented propagation approach by systematic search, are encouraging even if compared with approaches based on SAT.

DPPlan has not the huge memory requirements as SAT solvers and keeps a strong connection with the planning problem allowing the development of search strategies which incorporate domain dependent heuristics. Moreover, the typical SAT techniques, such as stochastic and incomplete strategies, can easily be transferred and integrated in this framework.

Introduction

The most recent relevant results in domain independent planning are represented by Graphplan, SATPLAN, Medical, IPP and Blackbox (Blum and Furst 1995; Kautz and Selman 1996; Ernst *et al.* 1997; Koehler *et al.* 1997; Kautz and Selman 1998). According to (Kambhampati 1997) these planners follow approaches which relate plan synthesis with constraint satisfaction. The most interesting performance results have been firstly obtained by Graphplan, which outperformed all previous domain independent plan synthesis algorithms by various many magnitude orders and by SATPLAN and its successor Blackbox, which has been shown to outperform Graphplan itself.

Graphplan is based on the construction of a data structure, called the planning graph, which is a very compact representation of set of states and actions that are possible after n step levels. The algorithm interleaves the planning graph construction phase, which adds a new step level to the graph, and a search phase, in which a solution to the planning problem is sought. The algorithm goes on until a solution is found or when the termination condition becomes true.

The search engine implemented in Graphplan is basically a backward-chain search, which starts from the goals layer and proceeds backwardly by finding operators which supports goals, then it continues by finding operators in the previous time levels supporting preconditions of newly added operators and so on. At each step the mutual exclusion relation is propagated during searching and conflicts can lead the algorithm to backtrack. As noted in (Kambhampati *et al.* 1997) the propagation of mutual exclusion constraints can be seen as a form of constraints propagation.

SAT planners are based on the transformation of a planning problem into the satisfiability problem of a propositional formula: if the formula is satisfiable then a solution plan can be obtained by the truth assignments which verify the formula. The formula can be generated in several ways: by hand, by expressing operator precondition/effect and frame axioms (through different types of encoding) and by translating the planning graph into a propositional formula. The latter is the way used in Medical and in Blackbox.

The formula, however is generated, is then passed to SAT solver in order to see if it is satisfiable and in this case to find a satisfying assignment, which corresponds to a solution of the initial planning problem.

Thus some existing planners, Medical and Blackbox, propose to use a planning graph construction phase interleaved to a search phase, with the difference, with respect to Graphplan, that this search is done in the SAT framework, by converting the planning graph into a propositional formula. This is one of the possible way to combine Graphplan and SATPLAN approaches.

We propose a completely different way to combine these two approaches: instead of converting a planning graph in a form that is useful to a SAT solver, we have built a search engine which is able to search for a solution within the planning graph and which behaves like

a SAT solver.

Compared to Blackbox and the other SATPLAN planners, our system will not need any translation phase to search a solution. Moreover if the search is performed into the planning graph, the search engine can exploit all the information contained in the planning graph and which are usually hidden after the translation into a propositional formula. Consider, for instance, that a deterministic SAT solver can receive a good speed up, as was shown in (Giunchiglia, Massarotto and Sebastiani 1998), if it is able to distinguish between action nodes and fact nodes by trying to assign values only to action nodes.

On the other hand, an apparent advantage of our approach on Graphplan–based systems is that the direction of the search is not fixed, but it can be unidirectional (either forward or backward), bidirectional or mixed, according to the strategy selected. This is possible because we have found propagation rules that allow the planner to choose any planning variable, either an action or a fact, (independently to which other variables have already been assigned) and to try to give that variable any value (true of false), by computing and propagating all the possible consequences of that choice.

The rules used in our approach have the following form: if the variable attached to a node N receives a certain value B, then the variables associated to nodes belonging to a set S must receive the value B', where S is, for the most part of rules, a set of nodes connected to N. Thus the modification of a single node can lead to the modification of a subgraph, which in some case can be quite large.

Note that one of the most important feature of SATPLAN planners, which is also one of the most influential cause of their speed, is that search has no direction: variables can be given values in any order, without being restricted to follow a determinate direction.

Another important point of our approach is to bring into the planning graph context the principle underlying Davis-Putnam algorithm (Davis *et al.* 1962), that is, if a given choice (e.g. assume an operator node is selected to be executed) is inconsistent with the value of some other variable, through the propagation rule, then the only way to obtain a solution is to assume the corresponding opposite choice (i.e. that operator cannot be executed). Positive and negative choices are then propagated through the planning graph and can have impact as both in the previous and the following time levels. For example determining that a fact must be false, has as a consequence of avoiding the choice of operators in next (previous) time level for which the fact is a precondition (effect).

A closely related approach is presented in (Rintanen 1998), where a planning algorithm on the background of SAT/CSP also based on non directional search is described. The main difference between our approach and Rintanen approach is that the latter is not based on the Graphplan data structure and uses a less number of rules, some of which are slightly different from ours. The higher number of rules of our approach can cause in most cases backtracking at earlier points in the search.

Experimental results show that the application of the principle of maximum propagation of consequences and Davis–Putnam principle leads to a powerful pruning of the search space and a problem can be solved with a low number of search branching points, if compared to standard Graphplan backward search. Furthermore DPPlan, our planning system based on the principles described above, keeps a strong connection with the planning problem, i.e. the planning graph information is directly available to the search strategy, this feature allows the easy development of search strategies which incorporate domain dependent or domain independent heuristics, with respect to a blind "blackbox" search.

In the next two sections the basic elements of our approach and the rules for reasoning and propagating search decisions on the planning graph are presented. In Section 4 different search strategies for solution extraction are discussed. Section 5 presents experimental results obtained by DPPlan. Comparisons with previous and related works and formal issues, as completeness, systematicity and termination are discussed in section 6 and conclusions are drawn in section 7.

The algorithm

In this section we describe DPPlan, the planning system that we have developed and tested. DPPlan is mainly based on Graphplan (Blum and Furst 1995), with which it shares the plans representation (the so called planning graph) and the description and syntax of problems and planning domains.

Both the planners interleave the phase of construction of planning graph, level by level, with the phase of solution search.

The main difference between our planner and Graphplan is the procedure used for searching a possible solution within the planning graph: the latter uses a backward-chaining algorithm (starting from the users goals to the initial state), while the former uses a Davis-Putnam-like procedure combined with several possible search strategies.

One of the basic elements of DPPlan is that it represents the choices made on the planning graph by associating to each node N a variable $Value(N)$ analogously to what happens in SAT-based approaches.

For an action node A at time t, $Value(A)$ has value *true* if A is used at time t in the solution, otherwise it has value *false*.

For a given fact F at a time t, $Value(F)$ would be true in two different cases, we therefore will use two different values: $Value(F)$ is *asserted* when F is achieved by any true action node at time $t-1$ and $Value(F)$ is *required* if F is a precondition of some true action at time t or it is one of the user goals.

Similarly we distinguish two ways of being false: $Value(F)$ is *denied* if F is deleted by a true action at

time $t-1$ and $Value(F)$ is *required-false* if F is required to be false by the search engine.

During search phase it can happen that a fact, whose value was *required*, is added by an action. In this case we set its state to a new value *required-and-then-asserted* in order to be able to restore the previous state when doing backtracking. Also for state *required-false* a new value *required-false-and-then-denied* should be used. In the rest of the paper we shall not use this distinction, because except for the backtracking phase, a node whose value is *required-and-then-asserted* (*required-false-and-then-denied*) can be handled like a node with value *asserted* (*denied*).

The values *required, asserted* and *true* are called *positive* values, while *required-false* and *false* are called *negative* values. It is clear that a **positive goal**, or just a **goal** as meant in ordinary planner, is a fact F whose value is *required*, while a **negative goal** is a fact whose value is *required-false*.

The algorithm consists in an initialization step of the planning graph (see *initialize*) followed by a solution extraction step (see *search*).

In the first step all the variables associated to every node in the planning graph are set to the undefined value; then the facts in the initial state are asserted while the user goals are required to be false.

```
procedure initialize
begin
    for each N in PLANNING-GRAPH do
        Value(N) := undefined
    for each F in INITIAL-FACTS do
        assert(F)
    for each F in USER-GOALS do
        require(F)
end
```

The core of the DPPlan search algorithm is described in the following recursive procedure:

```
procedure search
begin
    if the goals list is empty
    then return success
    else if there are no undefined nodes
    then return failure
    else begin
        choose(V,B)
        set(V,B)
        if search() then return success
        backtrack(V)
        set(V,not B)
        return search()
    end
end
```

where *set(V,B)* is the procedure assigning the boolean B to $Value(V)$ i.e.

```
procedure set(V,B)
begin
    if typeof(V)=fact then
        if B then require(V) else require-false(V)
    else /* V is an action node */
        if B then use(V) else exclude(V)
end
```

require, require-false, use and *exclude* are the procedures, which will be described in the next section, responsible to change the value of a given node and propagating this change through the planning graph. *backtrack* is a procedure which undoes the changes performed by the procedure *set*, thus restoring the situation before the execution of *set*.

When the algorithm ends with success then the solution is composed by those actions whose value is true.

The most important feature of this search procedure is that decisions about the variable to be tried next can be taken in any order without restrictions on the level at which search can be performed and on the type of node to choose (both facts and operators nodes are allowed to be chosen). The choice is made by the procedure *choose*, which will be described in section 4.

Graphplan search procedure is more rigid than our system in that it first tries to satisfy all the possible goals at some level before going the previous level.

The propagation rules

The basic propagation rules are described by means of recursive procedures described below. For fact nodes we have the procedures *assert, require, require-false* and *deny*, while for operator nodes we have the procedures *use* and *exclude*.

These procedures modify the value of the node which is passed as their argument, and propagate, if necessary, this change by calling some other procedures.

Propagating positive values

The following procedures give the variable attached to their argument a positive value. They return without effect if the node has already that positive value. They return a result which is *success* when no contradiction was found, or *failure* when if a contradiction has been reached.

The direct way of obtaining a contradiction is when the node has already a negative value, while an indirect source is when at least one of the procedures called inside fails at their turn.

Note that each of these procedures uses the mutexes precomputed in generation graph phase, in that if a node N, whether an operator or a fact, is set to a positive value, then every node which is mutually exclusive with N must be set to a negative value.

The first procedure, *use*, gives an operator node the value *true*: in order to use an action its preconditions must be true (so they becomes new subgoals in the procedure *require*), while its positive (negative) effects becomes *asserted* (*denied*).

```
procedure use(O)
begin
```

Value(O) :=executed
  for each p **in** PRECONDS(O) do
    require(p)
  for each e **in** ADD-LIST(O) do
    assert(e)
  for each e **in** DEL-LIST(O) do
    deny(e)
  for each e **in** MUTEX(O) do
    exclude(e)
end

The procedure *assert* gives a fact node the value *asserted*. Its main effect is to remove its argument from the goals list, if it was a goal. It is called by the procedure *use*.

procedure assert(F)
begin
  if Value(F)=required **then**
    remove F from the goals list.
    Value(F) :=required-and-then-asserted
  else
    for each e **in** MUTEX(F) **do**
      require-false(e)
    Value(F) :=asserted
end

The procedure *require* gives a fact node the value *required*, i.e. the creation of a new sub-goal. It returns without performing any action if the node value was *asserted*.

procedure require(F)
begin
  Value(F) := required
  add F to the goals list
  for each e **in** MUTEX(F) do
    require-false(e)
  for each d **such that** F $\in$ DEL-LIST(d) **do**
    exclude(d)
end

It is worth noting that the deletion (call to procedure *require-false*) of mutexes can be performed even if the node is required to be true, but it is not already asserted by any action: this can cause early pruning in the search space and can even fail, causing an anticipate backtracking.

Another way of getting an early pruning is to inhibit (through the procedure *exclude*) the execution of any action that can delete the node we require. It is unnecessary to do the same operation in the procedure assert because the action that adds F (rememver that assert is called only by use) will be mutually exclusive with any action which deletes F and therefore any such action will be already excluded.

Propagating negative values

The following procedures give the variable associated to their argument a negative value. They return without performing any operation if the node has already that negative value. As we saw in previous section these procedures return a result which is *success* when no contradiction was found, or *failure* when if a contradiction has been reached, i.e. the node value is already a positive value, or one of the procedures called fails.

The procedure *deny* gives a fact node F the value *denied*. As a consequence, any action which adds F must be excluded, because if it were true F would be true too, and also any action which has F as a precondition must be excluded, because it cannot be executed.

procedure deny(F)
begin
  if Value(F)=required-false then
    Value(F) := required-false-and-then-denied
    delete $\neg$F from the goal lists
  else
    Value(F) := denied
    for each p **such that** F $\in$ ADD-LIST(p) **do**
      exclude(p)
    for each c **such that** F $\in$ PRECONDS(c) **do**
      exclude(c)
end

The procedure *require–false* gives a fact node F the value *required–false*. It is the dual of the procedure *require*, in that it adds $\neg F$ to the goals list. Moreover it performs the same propagation as in the procedure *deny*.

procedure require-false(F)
begin
  Value(F) := required-false
  add $\neg$F to the goals list
  for each p **such that** F $\in$ ADD-LIST(p) **do**
    exclude(p)
  for each c **such that** F $\in$ PRECONDS(c) **do**
    exclude(c)
end

The procedure *exclude* gives an operator node O the value *false*. For a given fact F, we say that an operator O is a *possible adder* of F if F belongs to the add-list of O and O has value *undefined*. Similarly we say an operator O is a *possible deleter* of F if F belongs to the del-list of O and O has value *undefined*.

If an operator O will not be executed then some facts F cannot be reached (if O was their last possible adder). A more involuted consequence is that if a goal G is in the add-list of O and there is only one more possible adder for G, then that operator must be used, otherwise G would not be reachable. More complex propagations are done for facts which could be deleted by O:

- if a fact F at time t is a negative goal then
  - if it has no more possible deleters then F must be false at $t-1$
  - if has no still one possible deleter and F has a positive value at $t-1$ then there is only one chance to delete this goal: apply the last possible deleter of F

- if a fact F at time t is undefined, has no more possible deleters and F has a positive value at $t-1$ then F must be true at time t

Note that this asymmetry in handling facts added and deleted arise from the existence of NO–OP, which is always a possible adder to a positive fact and pushes the goal at the previous time (through the precondition), while this is not available for negative facts.

```
procedure exclude(O)
begin
    Value(O) := false
    for each f in ADD-LIST(O) do
        /* No Adder Rule */
        if number-of-possible-adders(f)=0
        then require-false(f)
        /* Unique Adder Rule */
        if Value(f)=required and
          number-of-possible-adders(f)=1
        then
          O'=last-possible-adder(f)
          use(O')
    for each f in DEL-LIST(O) do
        f-prev := copy-at-previous-time(f)
        if number-of-possible-deleters(f)=0 then
          /* No Deleter Rule 1*/
          if Value(f)=undefined and
            is_positive(Value(f-prev))
          then require(f)
          /* No Deleter Rule 2*/
          if Value(f)=required-false then
            delete ¬f from the goal lists
            require-false(f-prev)
        /* Unique Deleter Rule */
        if number-of-possible-deleters(f)=1 and
            is_positive(Value(f-prev))
        then
          O'=last-possible-deleter(f)
          use(O')
end
```

Note that the rule *No Adder* can lead to a backtracking if it is applied to a positive goal, because it means that this goal has no possibility of being achieved. A backtracking can also be generated by the rule *No deleter rule* if the fact at the previous time cannot be falsified.

The rules *No Adder* and *Unique Adder* may be checked in the procedure *require* in order to have a possibly earlier pruning of the search tree. And similarly the rules *No Deleter 1*, *No Deleter 2* and *Unique Adder* may be checked in the procedure *require-false*.

Necessary Truth and Falsity

The following rules directly derive from the frame axioms. They can be added to the procedures described above even if they are not strictly necessary (because of the NO–OP actions) during search phase:

Persistence of Truth Rule "When a fact F has a positive value true at time t and every action that can delete F is false then the fact F must be true (*required*) also at time $t+1$."

Its dual is:

Persistence of Falsity Rule "When a fact F is false at some time t and every its possible adder is false then the fact F must be false *required-false* also at time $t+1$."

Extended Propagation Rules

We also found some propagation rules that can be applied in order to prune the search space. These propagation rules do not belong to the class of propagation rules seen before because they do not follow from classical planning reasoning but they arise from some domain-independent considerations about efficiency of solutions.

We empirically found that their application can in some planning problems dramatically reduce the search space, while in some other cases their effects is not so relevant.

Avoid repeated consecutive actions Rule The following rule is based on the consideration that it is never useful to use the same instance of some actions in two consecutive levels:

"When an action is true at time t exclude its possible copy (i.e. the same operator applied to the same arguments) at times $t-1$ and $t+1$".

Avoid consecutive inverse actions Rule The following rule can help in domains where some actions a has its own inverse a^{-1}, i.e. the action which restores the situation before the execution of a. For instance in the logistics domain the couple LOAD and UNLOAD applied to the same arguments (e.g. LOAD(obj1,truck2) and UNLOAD(obj1,truck2)) are the inverse of each other.

It is quite obvious that the consecutive execution of a and a^{-1} is useless and should be avoided.

In general a and a^{-1} can have different arguments, but only in the order: for instance the inverse of MOVE(o,x,y), which moves the object o from place x to place y, is MOVE(o,y,x).

The strategy is "When an action at time t is true exclude its possible inverse at time $t-1$ and $t+1$"

Exclude useless actions Rule Another rule that can reduce the number of branching points by finding useless actions, i.e. actions whose execution would not produce any change in the state.

"If an undefined operator O is useless, i.e. its positive effects are already true (and produced) and its negative effects are already false, then exclude O".

Strategy rules

One of most critical points in our algorithm, similarly to other algorithms based on Davis-Putnam procedure,

is the choice of the undefined node variable and the boolean value to try. A good choice in this stage can affect the rest of search leading easily to a solution or to a dead end.

A strategy in DPPlan is a function, like *choose* in the procedure *search*, returning an undefined node variable and a boolean value to try first.

All the strategies we have implemented try the value TRUE before the value FALSE, so in their description we do not mention the boolean value returned.

For example a trivial, but valid, strategy is the choice of the first undefined node variable. In the first version of DPPlan we have implemented various strategy rules which reflects common planning heuristics.

Almost all the following strategy rules try to implement the principle "try in some way to reduce the number of goals to be achieved". In the rest of section we will call an action O *achiever* of a fact F if F belongs to the add-list of O and F is a positive fact, or if F is equal to $\neg G$ and G belongs to the del-list of O. Moreover a *possible achiever* of a fact F will mean an *achiever* whose value is *undefined*.

Use Any Achiever Strategy

The simplest strategy to implement a strategy that ensure that at least a goal will be achieved is to use an action which adds a positive goal or deletes a negative goal :
"Choose a goal G (either positive or negative) and return a possible achiever of G".

A quick and easy way to implement the Use–Any–Achiever strategy is choosing the first goal G in the goal list and selecting the first available adder or deleter. A randomized version, which can avoid problems derived from having a fixed order of tries consists in choosing randomly goal G and returning a randomly selected possible achiever of G.

Most convenient action Strategy

This strategy is a modification of the previous one and take into account of the number of further sub-goals that an achiever can require:

"Return the operator node O which has the maximum difference between the number of goals which O can achieve and the number of goals which O requires. Break ties deterministically or randomly".

Forward Strategy

The following strategy is almost equivalent to performing planning in a forward way, i.e. from initial state to goals. At the beginning of the search the only executable actions are actions belonging to the first level and therefore the first action to be tried is surely at the first level :

"Return either deterministically or randomly an undefined variable corresponding to an action which is executable, i.e. whose preconditions have value asserted".

Achieve the hardest goal Strategy

This strategy is similar to a principle found also in constraint satisfaction algorithms, that is, try to assign the value to the variable which has the smallest number of ammissible values. Here the more "constrained" goals are those which have the smallest number of ways to achieve them:

"Take the hardest goal G, i.e. the goal with the minimum number of possible achievers and return one of its achievers of G (deterministically or randomly selected). Break ties deterministically or randomly".

Backward Strategy

This strategy is somehow similar to Graphplan search method in that it tries to satisfy all the goals at some level before trying to attach the goals at a previous level.

"Take the latest goal G, i.e. the goal at the maximum level and return one of its achievers (deterministically or randomly selected). Break ties deterministically or randomly".

Mixed and Randomized Strategies

It is also possible to build new strategies by mixing some of them. A *randomized mixed strategy* takes two or more strategies, randomly choose one of them and apply it.

An important feature in the design of a mixed strategy is to be able to apply a component strategy on the contexts on which it gives the best results. For instance we found empirically that *Use any achiever* strategy performs better than *Achieve the hardest goal* when the goals number is large.

The understanding of search strategies for DPPlan is fairly preliminary and needs further investigations, with special regards to the effect of randomization (Gomes *et al.* 1998).

Implementation Issues and Experimental Results

DPPlan has been developed by modifying Graphplan (Blum and Furst 1995). Special modules implement propagation rules and the search strategies, while the planning graph generation code is directly taken from Graphplan except for extensions in the node structure.

The propagation rules are implemented through recursive procedures that call in their turn other propagation procedures.

Each procedure uses the information stored in the planning graph. For instance the procedure *use* explores the preconditions and effects of its argument through the scansion of lists *in_edges, out_edges, del_edges.*

The current version of DPPlan allows the optional activation of additional propagation rules and the selection of the search strategy.

The following preliminary experimental tests have been held on a 500 MHz Pentium PC with 512 MB RAM and time are expressed in seconds. In order to compare the time of pure solution extraction phase, the times do not include problem parsing and planning graph generation time. Due to large memory requirements of Blackbox, the time for the largest problem can be influenced by memory swapping overhead. The times in the table are given as average time on 50 trials per problem. Blackbox has been run with default parameters which uses the SATZ-Rand solver, while DPPlan has been executed with the *Achieve the Hardest Goal* systematic strategy without activating the extended propagation rules.

Problem	Levels	Blackbox	DPPlan
logistics.a	11	1.42	0.47
logistics.b	13	2.16	0.48
logistics.c	13	5.60	0.68
bw_large.a	12	24	0.66
bw_large.b	18	37.5	46.82
rocket_ext.a	7	2.56	0.22
rocket_ext.b	7	3.47	0.02
phil_8	6	0.440	0.08
phil_16	6	3.070	0.17
phil_32	6	8.920	0.40
phil_64	6	23.930	1.07

The *phil_8, phil_16, phil_32, phil_64* are a set of planning problems which model the classical concurrent automata problems for the $n = 8, 16, 32, 64$ dining philosophers (the automaton behaviour is encoded following (Baioletti, Milani 1999)).

The results for DPPlan confirm that the extensive application of propagation rules to the planning graph generally outperforms Graphplan on solution extraction, for example DPPlan solves logistics.c which cannot be solved by Graphplan after a 10 hrs run. The positive results, i.e. rocket and logistics domain, can be explained by the structure of the planning graph that makes an extensive use of pointers, which allow to implement an efficient propagation of the nodes variables updates. The dramatic increment of time between bw_large.a and bw_large.b, and the anomaly of rocket_ext.b (which is generally assumed to be harder than rocket_ext.a) are probably due to the simplicity of our current search strategies. These preliminary results are encouraging even if DPPlan has a lower performance for some problems with respect to Blackbox.

DPPlan has not the huge memory requirements of SAT solvers. The only memory structure needed by DPPlan is the planning graph, the nodes contains few additional information with respect to Graphplan structures. As a comparison Blackbox (Kautz and Selman 1998) first generates the planning graph and it then allocates memory for encoding it as a SAT problem. The space scalability of SAT solvers is a problem connected with the increasing number of clauses.

Comparison with previous work and formal issues

It is worth to compare and discuss DPPlan with respect to Graphplan, Blackbox and the approach in (Rintanen 1998).

The backward search algorithm of Graphplan is basically characterized by four steps:

- goals achievement, which consists on selecting only operators nodes which reach a needed fact node;
- mutual exclusive propagation of the selected node, which consists on pruning the search space of all nodes which are in mutual exclusion with the selected one.
- propagation of effects and their exclusive nodes to the next level.
- backtracking and memoizing on failure.

The main improvement that DPPlan adds is that:

- any undefined node can become a candidate member of the current solution, thus the search algorithm is not committed to be directional;
- negative choices are propagated and can produce positive impact on the planning graph;
- mutual exclusive nodes and effects propagation rules are greatly enhanced (see section 3)

The advantage of DPPlan approach is that the pruning effect is dramatically increased by propagation rules and by Davis-Putnam principle: if a choice of p is not possible, choose $\neg p$.

Negative choices were not considered in Graphplan therefore they could not produce any pruning effect: if the choice of an operator node leads to failure, Graphplan algorithm simply backtracks to another candidate operator choice without taking advantage of the information provided by the failure propagation (except for the memoization phase). On the other hand no memoization is currently done by DPPlan.

As noted in (Kautz and Selman 1996), a planning graph is quite similar to a propositional formula, therefore a main difference with Blackbox is that no encoding is required in DPPlan and searching and propagation are made directly on the planning graph, moreover a dditional propagation rules have been easily implemented in DPPlan with no additional structures or memory overhead.

It is certainly possible to find a correspondence with techniques used by SAT solvers and constraint propagation rules used in DPPlan.

For instance the Unique Adder Rule, which forces to use an operator $\mathcal{O}_k$ if it produces a needed fact $\mathcal{F}$ and any other adder is negated, roughly corresponds to

simplification of single literal clause, because the clause corresponding to action/effects axioms for $\mathcal{F}$ that is

$$\neg\mathcal{F} \vee \mathcal{O}_1 \vee \mathcal{O}_2 \vee \ldots \vee \mathcal{O}_n$$

would be reduced to the single literal $\mathcal{O}_k$ which will be forced to be true.

In addition DPPlan implements propagation rules which are not implemented in Blackbox, for example the Avoid Repeated Consecutive Actions Rule or Persistence of Truth Rule are not encoded in Blackbox, in particular the notion that action deletes an effect is not explicit in the usual SAT encodings.

It could be possible to encode these new rules as a SAT formula, i.e. the first rule could be encoded as $\neg\mathcal{O}^t \vee \neg\mathcal{O}^{t+1}$ for each $\mathcal{O}$ and t, thus requiring one additional binary clause for each operator.

Moreover the work described in (Rintanen 1998) is much closely related to DPPlan: in this work the non directional search proceeds by emulating the application of planning axioms and failed literals detection techniques, which come from problem encoding and solution searching in planning as satisfiability. While strictly related to SAT approach (Rintanen 1998) avoids, as DPPlan, the generation of an explicit encoding.

One of the remarkable differences is that our planning algorithm directly works on the planning graph and exploits the constraints embedded in it (such as mutex relations) during constraints propagation. The use of the planning graph structure has several advantages: the pre-computation of mutex relations avoids additional overhead during search on visited nodes, planning graph pre-pruning techniques can be applied (i.e. removing unneeded nodes), finally the presence of nonmutex final goals can be used to determine a lower bound to the solution plan lenght, as in standard Graphplan algorithm.

Completeness and Soundness

The algorithm implemented in the current version of DPPlan is complete in that if a solution exists within a fixed time level of the planning graph it is able to find it, otherwise it will produce a negative answer. To see why this is true consider that the main procedure, *search*, performs a Davis-Putnam-like search by trying to assign both boolean values to each undefined variable. Since the propagation rules assign values only to those variables for which there are no other possibilities, the search is systematic and every possible assignment will be taken into account.

Our algorithm is also sound, in that if it halts with success, the plan returned is a solution of the planning problem. The proof is related to the fact that DPPlan can stop with success only when all the goals are achieved. Since in the goals list all the user goals and the preconditions of all used actions will be present and a fact is removed from the goals list only when it is achieved, it is straightforward that in a plan produced by DPPlan all the user goals are true in the final situation and all the actions are executable.

Termination

Termination of a planner on planning graph can be defined as the following problem: if there is no solution in planning graph with n levels, would it be useful to add the $n+1$-th level to the graph, i.e. is there a maximum number of levels, after which no solution can be found ?

Graphplan algorithm use a termination criterion which combines the concept of memoization and the concept of "level off" (Blum and Furst 1995). The planning graph is said to "level off" at level n when the number of action and fact nodes and the number of mutual exclusive nodes in this level are the same as the next level $n+1$. When in two consecutive stages $t-1$ and t, after that the level off situation has been reached at level n, the number of unsolvable goals at level n is constant then, as shown in (Blum and Furst 1995), the problem is unsolvable.

Since DPPlan is not currently performing any memoization and since the search for the solution is not done in a backward way, then the Graphplan termination criterion cannot apply to DPPlan.

SAT based algorithms have a similar problem for determining the unsolvability. They can use cutoffs on CPU time, cutoff on the number of trials, or cutoff on time level in order to decide when do not generate the next time level and terminating with failure. Therefore the planner is incomplete because it is unable to find a solution at greater time levels.

Currently the only complete termination criterion available for DPPlan is that no solution can ever exist if the graph has leveled off at level n and a stage $n+a$ has passed without finding a solution, where a is the number of actions at level n.

This sufficient condition provides an unpractical upper bound, therefore further investigations are needed.

Conclusions and future work

DPPlan is a systematic planner based on the propagation of search decision through the planning graph.

The outperforming improvements with respect to Graphplan demonstrate the effectiveness of algorithms based on constraints propagation which are searching directly on the planning graph.

Further investigations are needed in order to demonstrate that the approach is really competitive and scalable with respect to the SAT based approach, but the preliminary experimental results are encouraging.

An important feature of DPPlan is the memory requirement, which is much smaller than SAT approach. Moreover, any search algorithm, in DPPlan framework, is not committed to a particular direction of search in that the propagation of a choice at a certain time level can produce useful consequences in any direction, or level, of the graph. The easy extendibility of search

strategies is also an important feature of DPPlan: we have experimented systematic forward and backward search algorithms, as well as systematic randomized algorithms. An easy extension of our planning system is to allow the user to express negative goals and preconditions: the machinery needed for handling negative subgoals is already available in the algorithm. Future work will experiment the effects of introducing in DPPlan framework some typical SAT techniques such as stochastic and incomplete strategies.

Another direction for future extensions is the integration in DPPlan context of other planning graph based algorithms which take into account of costs or resources consumption associated to operator nodes such as in (Koehler 1998)

Work needs to be done also on the formal side: firstly, the definition of an effective termination criteria is needed; then, the investigation of new planning dependent propagation rules which can simplify the search space, and the definition of forms of memoizing within the DPPlan framework.

References

A.L. Blum and M.L. Furst. Fast Planning through Planning Graph Analysis. *Artificial Intelligence 90(1-2): 281-300, 1997*

H. Kautz and Bart Selman. Pushing the envelope: Planning, Propositional Logic and Stochastic Search. In *Proc. of AAAI 1996*

M. D. Ernst, T. D. Milstein, and D. S. Weld. Automatic SAT-Compilation of Planning Problems. In *Proc. of IJCAI 1997*

J. Koehler. Planning under Constraints Resource. In *Proc. of ECAI 1998.*

Henry Kautz, D. McAllester, and Bart Selman. Encoding Plans in Propositional Logic. In *Proc. of KR 1996.*

Subbarao Kambhampati, Eric Lambrecht, and Eric Parker. Understanding and Extending Graphplan. In *Proc. of ECP 1997*

Henry Kautz and Bart Selman. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In *AIPS-98 Workshop on Planning as Combinatorial Search.*

J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending Planning Graphs to an ADL Subset In *Proc. of ECP-1997.*

M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *CACM 5, 1962, 394-397.*

Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proc. of AAAI-98.*

Henry Kautz and Bart Selman. Planning as Satisfiability. In *Proc. of ECAI 1992.*

R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence, 2 (3/4), 189-208*

Subbarao Kambhampati. Challenges in bridging plan synthesis paradigms. In *Proc. of IJCAI-97.*

Jussi Rintanen A Planning Algorithm not based on Directional Search In *Proc. of KR-98*

E. Giunchiglia, A. Massarotto, R. Sebastiani. *Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability.* Proc. of AAAI 1998.

M. Baioletti, A. Milani. *A Planning Model for Concurrent Asynchronous Automata* Submitted to AIPS-2000