Stratified Heuristic POCL Temporal Planning based on Planning Graphs and Constraint Programming

Ioannis Refanidis

University of Macedonia, Dept. of Applied Informatics, Thessaloniki, Greece
yrefanid@uom.gr

Abstract

This paper elaborates on the temporal planning graphs with mutual exclusion reasoning of the known TGP planning system, in order to build a heuristic temporal Partial Order Causal Link (POCL) planner. The planner exploits the temporal planning graph in two ways: First, it obtains heuristic estimates for sets of open goals during the plan construction phase, in order to solve the symbolic dimension of the planning problem. And second, it obtains additional disjunctive constraints, based on both permanent and temporary mutex relations, which are used by a CSP solver, in order to solve the temporal dimension of the planning problem. Furthermore, in the paper we simplify the temporal planning graph construction process, we propose two completeness preserving pruning rules and a heuristic function that takes into account clusters of permanently mutexed open goals. Preliminary results demonstrate the effectiveness of the various choices.

Introduction

Partial Order Causal Link (POCL) planning was the dominant planning paradigm until the middle of the previous decade. It has been adopted by numerous planning systems, such as SNLP (McAllester and Rosenblitt, 1991) and UCPOP (Penberthey and Weld, 1992) for symbolic domains, (Penberthy and Weld, 1994) for temporal and metric domains and (Peot and Smith, 1992; Pryor and Collins, 1996) for domains with uncertainty, among others. The main problem of the POCL approach was its inability to solve problems of moderate size, although it provably managed to reduce the size of the search space, with respect to the older state-space or regression-based planning approaches. This was mainly due to the lack of effective domain independent heuristics. Researchers mainly concentrated on devising alternative flaw selection strategies, e.g. (Peot and Smith, 1993; Joslin and Pollack, 1994; Schubert and Gerevini, 1995) among others.

The introduction of planning graphs (Blum and Furst, 1997) caused a significant increase in the performance of planning systems. State-space planners (Bonet, Loerings and Geffner, 1997; Refanidis and Vlahavas, 2001; Bonet and Geffner, 2001; Hoffmann and Noebel, 2001) exploited planning graph structures to devise heuristics to guide either progression or regression, achieving great performance, both in terms of solution speed and in the size of the tractable problem instances. TGP (Smith and Weld, 1999) extended planning graphs in a temporal setting.

The first attempt to exploit planning graph structures for POCL planning in symbolic domains was RePOP (Nguyen and Kambhampati, 2001). (Younes and Simmons, 2003) utilize also planning graphs to extract heuristics in a POCL setting, but they emphasize in flaw selection strategies.

Constraint programming techniques have been used in POCL, especially in temporal domains, before the advent of planning graphs and graph-based heuristics. Examples are IxTeT (Ghallab and Laruelle, 1994), ZENO (Penberthy and Weld, 1994) and Deviser (Vere 1983). These techniques are very efficient in reasoning about the time-windows. A common approach adopted by the POCL planning community is to interleave planning and scheduling, solving conflicts by introducing simple ordering constraints between actions (Smith, Frank and Jonsson, 2000; Vidal and Geffner, 2004).

In this paper we utilize the temporal planning graphs with mutual exclusion reasoning (Smith and Weld, 1999) to create a heuristic POCL planner that will use constraint programming to schedule the actions. Our approach consists of two phases: Initially a temporal planning graph is completely created, involving eternal and conditional mutex relations between pairs of propositions, pairs of actions and pairs of a proposition and an action. Then, a POCL planning phase is started, progressively solving open goals and posting disjunctive constraints to solve threats. Heuristics extracted from the temporal planning graph are used to select among the alternative ways to support open goals. Threats are detected using the information encoded in mutex relations. Our approach does not rely on the presence of no-op actions or of a minimum quantum of time. After a plan with no open goal is found, a CSP solver is employed to search for a feasible schedule. Suitable completeness preserving pruning rules have been employed to reduce the branching factor.

According to (Smith, Frank and Jonsson, 2000), our approach could be classified as stratified P&S. However, there is some interleaving between P&S, since non-disjunctive temporal constraints, originating by the causal link relations, are taken into account during the plan construction phase and may lead to plan rejection, in case of tight bounds. Moreover, temporal information is taken into account by the heuristic.

The rest of the paper is as follows: First we review the temporal planning graph structure, as it has been implemented in our system. Then we rewrite the graphplan plan-extraction phase, using POCL planning and constraint programming. On top of this, we build then a heuristic POCL temporal planner, with the heuristic being extracted from the planning graph. In the same section we present two additional pruning techniques that we found useful in our setting. Finally, we present preliminary results from the evaluation of the various techniques implemented in our system and pose future directions.

Temporal Planning Graphs

In this section we present the way the temporal planning graphs are constructed in our system. We use a slightly different and simplified approach than in (Smith and Weld, 1999), which does not lead to arbitrarily complex formulae that need simplification. Actually, the mutex relations we compute are "simplified" from their beginning, in a way that we think is more intuitive and closer to the original graphplan.

As in (Smith and Weld, 1999), we define eternal mutex (emutex) relations and conditional mutex (cmutex) relations between pairs of propositions, pairs of actions and pairs of a proposition and an action (in the following we will refer to propositions and actions as nodes of the planning graph). An emutex relation between two nodes denotes that they can never be true simultaneously (being true for an action means being executed). A cmutex relation between two nodes determines a time point, before which the two nodes cannot be true simultaneously (this time point may be infinite, denoted with *inf*). In the following we will use *Prec*(*A*) and *Eff*(*A*) to denote the preconditions and effects of an action *A* respectively.

Two nodes are emutexed in the following cases:

- Each proposition *P* is emutexed with its negation $\neg P$.
- An action *A* and a proposition *P* are emutexed, iff $P \in Eff(A)$ or $\neg P \in Eff(A)$ or $\neg P \in Prec(A)$.
- Two actions *A* and *B* are emutexed iff either *A* or *B* deletes the preconditions or effects of the other, or *A* and *B* have emutexed preconditions.

Cmutex relations are computed in a recursive manner, during the construction of the planning graph. The basic data structures of the temporal planning graph are the following:

- *prop*(*P*,*T*): Time *T* is the earliest time proposition *P* may become true.
- *action*(*A*,*T*): Time *T* is the earliest time action *A* may start execution.
- *emutex*(N_1,N_2): Nodes N_1 and N_2 are emutexed.
- *cmutex*(N_1,N_2,*T*): Nodes N_1 and N_2 are cmutexed until time *T* (*T* may be *inf*).

To avoid duplicate effort in storing mutex relations, we used a lexicographic ordering in their arguments N_1 and N_2.

During planning graph creation, a time-ordered queue of events is maintained. There are two kind of events:

- *new_prop*(*P,T*): Proposition *P* has been achieved at time *T*.
- *end_cmutex*(*P*,*Q*,*T*): The cmutex relation between propositions *P* and *Q* ends at time *T*.

The procedure for creating the temporal planning graph is initialized by asserting *prop*(*P*,0) and *new_prop*(*P*,0) for each initial state's proposition *P*. Then, the planning graph is created according to the following steps (Figure 1):

1. Retract the earliest event *E*. Let *T* be its time. If no event exists, then stop.
2. If *E*=*new_prop*(*P,T*), let *Actions* include each new action *A* having *P* in its preconditions. If *E*=*end_cmutex*(*P*,*Q*,*T*) event, let *Actions* include each new action *A* having *P* and *Q* in its preconditions.
3. For each action *A* in *Actions*, such that all of its preconditions have been achieved in time earlier than or equal to *T* and there is no pair of preconditions being emutexed or cmutexed until time greater than *T*:
   a. Let D_A be *A*'s duration.
   b. Call ADD_EFFECTS(*Eff*(*A*), $T+D_A$)
   c. Call NEW_EMUTEX(*A*)
   d. Call CMUTEX_ACTION_PROP1(*A*,*T*)
   e. Call CMUTEX_ACTIONS(*A*,*T*)
   f. Call CMUTEX_PROPS(*A*)
   g. Call STOP_CMUTEX_PROPS(*A*,$T+D_A$)

Note that the order of procedure calls in step 3 is important. In the following we explicate these procedure calls.

ADD_EFFECTS(*Eff*(*A*),*T*+D_A): For each newly achieved fact $P \in Eff(A)$, this procedure asserts *prop*(*P*,$T+D_A$) and event *new_prop*(*P*,$T+D_A$). Moreover, *emutex* relations to existing nodes are created, according to the related definitions. For those propositions in *Eff*(*A*) that already exist in the planning graph with time reference earlier than or equal to $T+D_A$, nothing happens. Finally, for those propositions in *Eff*(*A*) that already exist in the planning graph, with time reference later than $T+D_A$, their time reference is replaced by $T+D_A$ and the corresponding pending *new_prop* event is updated accordingly.

NEW_EMUTEX(*A*): This procedure creates *emutex* relations between the new action A and existing planning graph nodes, according to the related definitions.

CMUTEX_ACTION_PROP1(A,T): This procedure creates new *cmutex* relations between the newly inserted action *A* and already existing propositions that are cmutexed with *A*'s preconditions and are not emutexed with *A*. For each such proposition *Q* that is cmutexed with any proposition $P \in Prec(A)$ until time T_1, a cmutex relation between *A* and *Q* until (the maximum possible) T_1 is asserted.

CMUTEX_ACTIONS(*A*,*T*): This procedure creates cmutex relations between *A* and other existing actions, to which *A* is not emutexed. *A* is cmutexed to another action *B* until time T_1, if there is a pair of propositions $P \in Prec(A)$ and $Q \in Prec(B)$, such that *cmutex*(*P*,*Q*,T_1), and T_1 is the latest time for any such pair of propositions.

CMUTEX_PROPS(*A*): This procedure creates cmutex

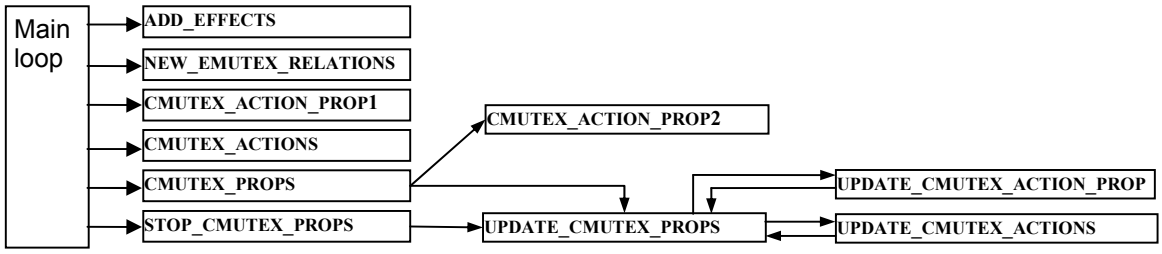

Figure 1: Flow of procedure calls while creating temporal planning graphs.

relations between propositions achieced by A, and other propositions. In particular:

- Each proposition P firstly achieved by A is cmutexed until *inf* to every other proposition Q, to which A is emutexed and P is not emutexed. A call to CMUTEX_ACTION_PROP2(A,P,Q,*inf*) is issued.
- Each proposition P firstly achieved by A is cmutexed to every other proposition Q, to which A is cmutexed. The end T_1 of *cmuxex*(P,Q,T_1) is equal to the sum between T_0 and D_A, where T_0 is obtained by *cmutex*(A,Q,T_0). Event *end_cmutex*(P,Q,T_1) is created, in case T_1 is not *inf* (T_1 is *inf* if T_0 is also *inf*). A call to CMUTEX_ACTION_PROP2(A,P,Q,T_1) is issued.
- For each proposition P reachieved by A, for every other proposition Q to which A is cmutexed until time $T_0<inf$, call UPDATE_CMUTEX_PROPS(P,Q,T_1), where $T_1=T_0+D_A$.

STOP_CMUTEX_PROPS(A,$T+D_A$): This procedure terminates cmutex relations between propositions in the following four cases:

- For each pair of propositions P and Q, such that $P\in Eff(A)$, $Q\in Eff(A)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each proposition $P\in Eff(A)$, for each proposition $Q\in Prec(A)$, such that $\neg Q\notin Eff(A)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each proposition $P\in Eff(A)$, for each proposition $Q\notin Prec(A)$, such that $\nexists\ R\in Prec(A)$, such that *emutex*(Q,R) or *cmutex*(Q,R,T_1), for some T_1, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$).
- For each newly achieved proposition $P\in Eff(A)$, for each action $B\neq A$, such that P is not mutexed with B, for each proposition $Q\in Eff(B)$, call UPDATE_CMUTEX_PROPS(P,Q,$T+D_A+D_B$)

The following procedures were referenced above:

CMUTEX_ACTION_PROP2(A,P,Q,T): For each action $B\neq A$, such that $Q\in Prec(B)$, assert *cmutex*(P,B,T).

UPDATE_CMUTEX_PROPS(P,Q,$T+D_A$): If *cmutex*(P,Q,T_1), where $T_1>T+D_A$, exists in the planning graph, then retract *cmutex*(P,Q,T_1), delete the event *end_cmutex*(P,Q,T_1) (if any), assert *cmutex*(P,Q,$T+D_A$), create event *end_cmutex*(P,A,$T+D_A$) and perform the following calls:

- UPDATE_CMUTEX_ACTION_PROP(P,Q,$T+D_A$)
- UPDATE_CMUTEX_ACTIONS(P,Q,$T+D_A$)

Finally, the following two procedures perform are responsible for recursively updating the cmutex relations:

UPDATE_CMUTEX_ACTION_PROP(P,Q,$T+D_A$): This procedure updates any preexisting cmutex relation between proposition P and any action C, such that $Q\in Prec(C)$, as well as between proposition Q and any action B, such that $P\in Prec(B)$. Concerning the former case (the second is symmetric), the update is performed only if, taken into account the rest of C's preconditions and the cmutex relations between them and P, the new end time T_1 of the cmutex relation between P and C is earlier than the existing one. For each such update, and for each $R\in Eff(C)$, UPDATE_CMUTEX_PROPS(P,R,T_1+D_C) is called, where D_C is the duration of C.

UPDATE_CMUTEX_ACTIONS(P,Q,$T+D_A$): This procedure updates any preexisting cmutex relation between pairs of actions B and C, such that $P\in Prec(B)$ and $Q\in Prec(C)$. The update is performed only if, taken into account the cmutex relations between any pair of preconditions of B and C, the new end time T_1 of the cmutex relation between B and C is earlier than the existing one. In that case, UPDATE_CMUTEX_PROPS(R,S,T_2) is called for every pair of propositions R and S, such that $R\in Eff(B)$ and $S\in Eff(C)$. Time T_2 is determined as $max(T+D_A+min(D_B,D_C), T_B+D_B, T_C+D_C)$, where T_B, T_C are the earliest possible start times of B and C, according to the planning graph, and D_B, D_C are their durations.

Plan Extraction as Constraint Satisfaction Problem

In this section we formulate the plan extraction process from the planning graph as a constraint satisfaction problem in the space of partial plans. Our approach works with ground actions and represents their start times with constraint variables.

The main structures are the following:

- An *Agenda* with entries of the form $\langle G,T\rangle$, where G is an open goal and T is a constraint variable, always unified with the start time of an action of the plan.
- A partial *Plan* with entries of the form $\langle A,T_A\rangle$, denoting that action A starts execution at time T_A, where T_A is a constraint variable.
- A list *Persistence* with entries of the form $\langle G,T_1,T_2\rangle$, meaning that proposition G has to be true between T_1

and T_2, where T_1 and T_2 are constraint variables.

Agenda is initialized with the goal propositions and the time at which the goals have firstly achieved in a non-mutexed way, whereas the other structures are initially empty. The main loop of the algorithm is the following:

1. Dequeue $\langle G,T\rangle$ from *Agenda*. If *Agenda* is empty, call the CSP solver to schedule the actions. In case of success, return current instantiated plan, otherwise backtrack.
2. Non-deterministically try to support $\langle G,T\rangle$ in one of the following options:

2a. Support $\langle G,T\rangle$ from the initial state. Call:
PERSISTENCE_PLAN(G,0,T,*Plan*)
PERSISTENCE2(G,0,T,*Persistence*)
Add $\langle G,0,T\rangle$ in *Persistence*.

2b. Non-deterministically try to support $\langle G,T\rangle$ from an existing action $\langle A,T_A\rangle$, where $G\in Eff(A)$. Post the constraint $T_A+D_A\leq T$. Call:
PERSISTENCE_PLAN(G,T_A+D_A,T,*Plan*)
PERSISTENCE2(G,T_A+D_A,T,*Persistence*)
Add $\langle G,T_A+D_A,T\rangle$ in *Persistence*.

2c. Non-deterministically choose a new action A to insert in *Plan*, such that $G\in Eff(A)$. Create a new constraint variable, T_A, for the start time of A. Post the constraint $T_A+D_A\leq T$. Call:
ACTION_PLAN(A,T_A,*Plan*)
ACTION_PERSISTENCE(A,T_A,*Persistence*)
PERSISTENCE_PLAN(G,T_A+D_A,T,*Plan*)
PERSISTENCE2(G,T_A+D_A,T,*Persistence*)
Add $\langle G,T_A+D_A,T\rangle$ in *Persistence*. For each precondition P of A, add $\langle P,T_A\rangle$ in *Agenda*.

In the following we explicate the procedure calls.

PERSISTENCE_PLAN(G,T_1,T_2,*Plan*): For every entry $\langle A,T_A\rangle$ in *Plan*:
- If there is an *emutex* relation between A and G, or a *cmutex* that lasts until *inf*, post the following constraint:
$$T_A\geq T_2 \text{ or } T_A+D_A\leq T_1$$
- If there is a *cmutex* relation between A and G that ends at finite time T, post the following constraint:
$$T_A+D_A\leq T_1 \text{ or } T_A\geq T_2 \text{ or } T_1\geq T \text{ or } T_A\geq T$$

PERSISTENCE2(G_1,T_{11},T_{12},Persistence): For every entry $\langle G_2,T_{21},T_{22}\rangle$ in *Persistence*:
- If G_1 and G_2 are emutexed or cmutexed until *inf*, post the following constraint:
$$T_{11}\geq T_{22} \text{ or } T_{21}\geq T_{12}$$
- If G_1 and G_2 are cmutexed until the finite time T, post the following constraint:
$$T_{11}\geq T_{22} \text{ or } T_{21}\geq T_{12} \text{ or } T_{11}\geq T \text{ or } T_{21}\geq T$$

ACTION_PLAN(A,T_A,*Plan*): For every entry $\langle B,T_B\rangle$ in *Plan*:
- If $A=B$ (different instances of the same action), or if A and B are emutexed, or if they are cmutexed until *inf*, post the following constraint:
$$T_B\geq T_A+D_A \text{ or } T_A\geq T_B+D_B$$
- If A and B are cmutexed until the finite time T, post the following constraint:
$$T_B\geq T_A+D_A \text{ or } T_A\geq T_B+D_B \text{ or } T_B\geq T \text{ or } T_A\geq T$$

ACTION_PERSISTENCE(A,T_A,*Persistence*): For every entry $\langle G,T_1,T_2\rangle$ in *Persistence*:
- If A and G are emutexed or cmutexed until *inf*, post the following constraint:
$$T_A+D_A\leq T_1 \text{ or } T_A\geq T_2$$
- If A and G are cmutexed until the finite time T, post the following constraint:
$$T_A+D_A\leq T_1 \text{ or } T_A\geq T_2 \text{ or } T_A\geq T \text{ or } T_1\geq T$$

The innovation in the above algorithm is that temporary *cmutex* relations are taken into account and produce threats, whereas alternative ways to promotion and demotion are available to resolve these threats. In particular, consider two nodes A and B that have to be true in $[T_{A1},T_{A2}]$ and $[T_{B1},T_{B2}]$ respectively (for an action the interval corresponds to the duration of the action, whereas for a proposition it corresponds to a protected interval). If these nodes are cmutexed until finite time T, then, apart from usual promotion and demotion, there are two additional ways to resolve this threat. Indeed, we can demote one of the two nodes in such a way that it starts after T, i.e. $T_{A1}\geq T$ or $T_{B1}\geq T$.

What is interesting with threats caused by cmutex relations is that it is not necessary to resolve them. We can just ignore them and let the POCL procedure search for a solution. The planner will not produce any incorrect solution (as it would happen if we ignored an emutex relation); it will detect any inconsistency later in the planning procedure, after having spent more time in search.

Another interesting remark concerns the way these threats are resolved. The four disjuncts in e.g.:
$$T_{11}\geq T_{22} \text{ or } T_{21}\geq T_{12} \text{ or } T_{11}\geq T \text{ or } T_{21}\geq T$$
do not constitute mutual exclusive cases (as it happens with the promotion and demotion in solving threats caused by permanent mutexes). Actually in a solution plan it may be the case that more than one of these disjuncts is true. An alternative settlement that produces mutual exclusive cases would be the following one:
$$(T_{11}\geq T_{22} \text{ and } T_{11}<T \text{ and } T_{21}<T) \text{ or}$$
$$(T_{21}\geq T_{12} \text{ and } T_{11}<T \text{ and } T_{21}<T) \text{ or}$$
$$(T_{11}\geq T \text{ and } T_{21}<T) \text{ or}$$
$$T_{21}\geq T$$
This resolution poses more constraints and may lead faster to fail, but it also produces plans with constraints that are not intuitive.

Heuristic POCL Temporal Planning

As with the original TGP, the main problem of the plan extraction process described in the previous section is that it must be repeated for each time step, starting from the time point at which the goals are firstly achieved without mutexes between them, where the time step is equal to the greatest common divisor of the durations of the problem actions. This is necessary if we want to get an optimal plan with respect to its makespan. However, this is unpractical

for realistic domains, especially for temporal ones, so we disregard optimality and we attempt to obtain near optimal plans, using heuristic guidance and branch and bound techniques.

There are several ways to derive heuristics from planning graphs, which can easily be adapted in a temporal setting. The most used is the sum heuristic (Nguyen and Kambhampati, 2000), which scores a partial plan with the sum of the earliest possible times, in which each proposition G in *Agenda* can be achieved. These times are obtained by the $prop(G,T)$ relations of the temporal planning graph.

In our system we adopted a slightly different heuristic, adapted by (Younes and Simmons, 2003). For each set of open goals:

- We do not consider duplicate goals. This prevents planner from inserting into the plan actions that are "biased" to the initial state configuration, but allows for inserting actions that "connect" pairs of subgoals.
- We do not consider goals that can potentially be supported by actions already inserted in the plan. This favors actions that, although they inserted in the plan to support a specific open goal, could be used to support other goals too.
- From the remaining goals, we sum the maximum of the heuristic values for each cluster of goals that are emutexed or cmutexed until the infinite to each other. This takes into account that permanently mutexed propositions have usually to be achieved in sequence, so the cost to achieve them is estimated by the cost of the most expensive one,
- In the above sum, we add the number of the goals, taking into account both duplicate goals, goals supported by existing actions and mutexed goals. This aims at solving ties between partial plans.

The procedure for POCL heuristic planning is quite similar to that described in the previous section. The basic difference is the following: For a selected partial plan, a number of refined new plans are created simultaneously, corresponding to the various ways to support an open goal, according to the nondeterministic choices of the algorithm in the preceding section. For each child plan, the whole constraint pool and the domains of the constraint variables must be copied. The new plans are scored by the heuristic function and stored in memory together with other existing partial plans. That is, we adopted a best-first schema to search the space of the partial plans. Finally, the best partial plan is retracted and refined in the next step. Ties are broken by preferring newer plans.

This setting led us in a variety of choices that reduce the branching factor without sacrificing completeness. The first one of them concerns the delay of the ordering decisions. As it has been described in the previous section, threats in the plan are resolved by posting disjunctive constraints, instead of creating separate child-plans without disjunctions. Having disjunctions reduces the possibilities for propagating constraints, which may lead to inability for early inconsistency detection. On the other hand, this choice overcomes the problem of the numerous similar child-plans having the same sets of open goals and actions but different orderings. Generating separate child plans for the alternative ways to resolve threats would lead to a continuously increasing number of similar plans with equal heuristic values that would dominate every other area of the search space. In case of an early wrong estimate of the heuristic function, it would be very difficult to overcome it. In our system new childs are generated only by the alternative ways to support open goals.

Another tuning of the planning process concerns the way open goals are selected. We adopted the approach to resolve the most costly goal of each selected partial plan, according to the heuristic function. Note that the order in which open goals are selected does not constitute a backtracking point, however, it has large impact on the efficiency of any POCL planner. Selecting the most costly goal allows generally for faster downgrading of the heuristic values, towards a solution plan. On the other hand, with our heuristic function, which is a hybrid max-sum function, selecting any other than the most costly goal could lead to the generation of child-plans with the same heuristic value to their parent.

Repeated Subgoal Pruning

In this section we present a pruning technique we implemented in our planner, which in some problems led to significant reduction in the branching factor. We call it *repeated subgoal pruning* and, in order to present it we have first to define the notion of primitive subgoal chains.

Def. 1: A *primitive subgoal chain* is an ordered list of subgoals $\langle G_n, G_{n-1}, \ldots G_0\rangle$, where each subgoal G_i has been inserted in *Agenda* as a precondition of action A_i, where action A_i was initially inserted in the plan to support subgoal G_{i-1}. Subgoal G_0 is an original goal of the problem instance. We denote the primitive subgoal chain starting by any specific subgoal G with PCHAIN(G).

Note that in the above definition we do not care about other subgoals that action A_i might have been used to support, after its introduction in the plan. So, the preconditions of A_i cannot belong to the same primitive subgoal chain with these alternative subgoals that A_i was used afterwards to support. A primitive subgoal chain is extended only when a new action is introduced in the plan to support its left-most subgoal. In this case, various new primitive subgoal chains are generated, one for each precondition of the new action.

The main idea of repeated subgoal pruning is the following:

Repeated subgoal pruning rule: A new action A with $G \in Eff(A)$, cannot be inserted in a plan to support a specific subgoal G, if there is any proposition $P \in Precs(A)$ such that $P \in$ PCHAIN(G).

The above pruning rule sacrifices completeness. Indeed, there are many plans where a single proposition may

change its truth value many times. For example, in Figure 2 the task is to clean a dirty room, which can be done only if the light is *on*. The light is initially *off*, and we have to leave it *off* at the end. So, in order to achieve all goals, light has to get once *on* and then *off* in the plan. On the other hand, changing repeatedly and many times the status of the light would be irrational. We want to avoid irrational loops without sacrificing completeness. To achieve this, we adopted the following approach:

The repeated subgoal pruning rule is implemented in our system as follows: Suppose we are processing subgoal $\langle G,T\rangle$. There are several possibilities to support G (from the initial state, from several existing actions and from several new actions) and all of them are examined in order to generate the corresponding child-plans. Suppose there are some (among others) new actions that could support G, but the pruning rule is activated for all of them. For all these actions, a single new child-plan is generated, in which no new action has been inserted. The new child-plan differs from its parent one only in that subgoal $\langle G,T\rangle$ has been marked as inactive (initially all subgoals are marked as active). Moreover, $\langle G,T\rangle$ has been labeled with the number of actions the current plan has. In case no active goal exists in the new *Agenda*, the new child plan is pruned.

Inactive subgoals differ from active ones in that they cannot be supported by new actions or by the initial state: They can only be supported by existing actions that have been added in the plan after they became inactive (this is why its inactive subgoal is labeled with the number of actions in the plan at the time the subgoal became inactive).

The child-plan generated when a subgoal becomes inactive is identical to its parent one, so it is scored with the same value by the heuristic function. So, it may happen that this child-plan will be selected in the next loop of the POCL planning procedure for further refinement, thus entering in an infinite loop. We cope with this problem in two ways: First, we penalize inactive subgoals, so that their plans will not be selected in case other plans with the same heuristic value but fewer inactive subgoals exist. This is done by adding for each inactive subgoal a small constant in the heuristic value. Second, when a plan with inactive subgoals is selected for further refinement, the inactive subgoal is not selected as the most costly subgoal (event if it is), except if new actions have been inserted into the plan after this subgoal became inactive. This last technique eliminates the risk of entering into infinite loops when refining partial plans with inactive subgoals.

Figure 2 illustrates all these. We have two subgoals, *off* and *clean*. Suppose that initially we choose SWITCH_OFF to support *off*, because this is the most costly subgoal. So, the primitive subgoal chain $\langle on, off\rangle$ is generated from the precondition of SWITCH_OFF. Suppose next we have to support *clean*, and we insert CLEAN in the plan. The primitive subgoal chains $\langle on,clean\rangle$ and $\langle dirty,clean\rangle$ are generated. Suppose next we have to support the precondition *on* of SWITCH_OFF. The only way to do this is to insert SWITCH_ON in the plan. But this activates the

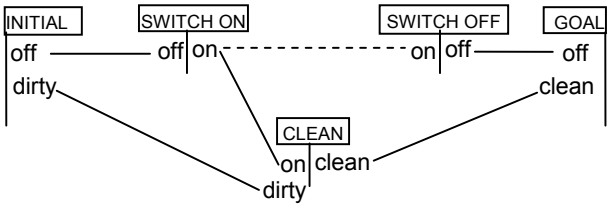

Figure 2: Subgoal chains. Solid lines denote primitive subgoal chains.

pruning rule, and the insertion is rejected. The subgoal becomes inactive and then subgoal *on* of CLEAN is tried (although it may be less costly than *on*). Now, action SWITCH_ON can be inserted into the plan, since subgoal chain $\langle off,on,clean\rangle$ does not activate the pruning rule. Next, subgoal *dirty* is supported by the initial state, subgoal *on* of SWITCH_OFF can be supported by the already existing action SWITCH_ON (which has been inserted into the plan after the subgoal became inactive) and finally subgoal *off* is supported by the initial state.

An alternative, but less efficient way to cope with repeated subgoals would be to prune altogether any child-plan with new actions falling under the pruning rule, without marking goals in *Agenda* as inactive. In this case, in order to preserve completeness, all subgoals in *Agenda* should be processed for each partial plan (and not only the most costly one). This approach, although more general and intuitive, is less efficient.

The last paragraph gives rise to an alternative formulation of the pruning rule. First, we generalize the notion of the primitive subgoal chains:

Def. 2: A *subgoal chain* is an ordered list of subgoals $\langle G_n, G_{n-1}, \ldots, G_0\rangle$, where each subgoal G_i is a precondition of some action A of the plan, where an effect of A supports subgoal G_{i-1}. Subgoal G_0 is an original goal of the problem instance.

This definition does not care about the order in which actions have been inserted in the plan and subgoals were resolved, as happens with primitive subgoal chains. With this definition, the pruning rule can be rewritten in a more declarative (but less operational) way:

> For any subgoal G appearing in a causal link of a partial order plan, there must exist at least one subgoal chain starting with G, were G is not repeated.

One could claim that this pruning rule is nothing more than repeated state pruning, so this pruning could be achieved by employing a closed list of visited states. However, in a temporal setting with disjunctive constraints it is not easy to check whether two partial plans with different sets of actions are equivalent. The pruning rule we presented manages at least to catch some of these cases.

The repeated subgoal pruning rule preserves completeness (we reserve for a proof in an extended version of this paper), whereas it reduces the search space, since it avoids irrational loops. It can be used both for symbolic and for temporal domains, being more effective in the latter case

#	*Full*		*Older*		*-Pruning 1*		*-Pruning 1, Older*		*STP*	
	time	*makespan*	*time*	*makespan*	*time*	*makespan*	*time*	*makespan*	*time*	*makespan*
satellite1	1.39	177	2.03	269	81.83	1984	3.52	269	374.47	323
satellite2	26	453	24.25	182	-	-	30.17	182	-	-
satellite3	43.9	441	48.3	251	-	-	50.77	251	-	-
satellite4	242.59	501	-	-	-	-	-	-	-	-
satellite5	1154	688	-	-	-	-	-	-	-	-

Figure 3: Preliminary results for various configurations of the planner (time in *secs*).

due to the increased probability of irrational loops with low duration actions.

Deleted Supports

A second pruning rule that has been used in our system concerns pairs of actions being supported by the same proposition, whereas both actions delete this proposition. Suppose there is a proposition P in the plan, being either an initial proposition or an add effect of an existing action. This specific proposition supports two actions A and B, for which $P \in Prec(A)$, $P \in Prec(B)$ *but also* $\neg P \in Eff(A)$ and $\neg P \in Prec(B)$ hold. It is obvious that this is an inconsistency and this partial plan must be discarded. However, the use of disjunctive constraints in our setting renders this inconsistency undetectable; it is detected only at the scheduling phase, when no feasible labeling can be found.

We implemented a mechanism in our system, which, each time a new causal link is created, checks which other existing actions are supported by the same proposition. For such pairs A and B of actions, being supported by the same proposition, the following constraints posted:

- If neither A nor B deletes P, no constraint is posted.
- If A deletes P but B preserves it, A is demoted after B.
- If B deletes P but A preserves it, B is demoted after A.
- If both A and B delete P, the plan is discarded.

This pruning rule has significant impact in the efficiency of the planning system, according to our experiments, as it reduces the number of unsuccessful trials to solve the constraint satisfaction problem.

Solution Generation and Presentation

Each time a partial plan is found, with *Agenda* being empty of open goals, a constraint solver is employed to search for feasible assignments of values to the constraint variables, i.e. the start times of the actions. Due to the extensive use of disjunctive constraints, the domains of the constraint variables are usually very broad before labeling begins. We use a branch and bound schema for solving the CSP problem, with the makespan of the plan being the minimized quantity. This ensures that we will get the shortest temporal plan, with respect to the specific partial plan at hand. Of course this is not the globally optimal plan, since alternative partial plans might result in even shorter makespans. If we want to find even better plans, we can start the POCL planning procedure from scratch, setting the upper bound slightly shorter than the makespan of the found plan.

Concerning the presentation of the solution plan, what we get from the CSP solver is an assignment of time points to the start times of the actions. If we want a more flexible presentation, we can present also the constraints between the actions. In case of disjunctive constraints, we can remove those disjuncts that are not satisfied by the current assignment. However, even with this removal, more than one disjuncts from each disjunction may be present, since, as we explicated earlier in this paper, the disjunctions do not correspond to mutually exclusive cases. Our current implementation keeps all these satisfiable disjuncts and present them as part of the solution.

Preliminary Results

In this section we evaluate the various techniques we presented in the paper. We implemented a POCL temporal planner in Prolog and used ECLiPSe 5.8 as our environment. We ran our experiments in a Pentium 4, 3GHz, 1GB memory machine and we set a time limit of 20 mins. Our intention was not to create a high-performance planning system, so we did not give attention in implementation details but in declarativeness. As a result, our planner is not quite efficient to solve complex temporal problems.

We ran our system in the easiest of the temporal Satellite domain of the 4th International Planning Competition. The only modification we made was that we truncated the action durations to the nearest integer[1]. We compared the full featured system with alternative configurations, where one feature each time was deactivated. The results are shown in Figure 3. *Time* in all cases includes the creation of the temporal planning graph. We set a high enough upper bound for time, so all problems were solvable.

Columns *Full* present the performance of the full featured system. Columns *Older* present a configuration where ties are broken by preferring the older plans. We can see that in this case shorter plans are found, however more time is needed in general, making the larger problems unsolvable under the specified time limit.

Columns *STP* (for simple temporal problem) concern a configuration of the system, where threats were resolved by generating alternative childs. This choice may increase dramatically the branching factor, so we applied it only to

[1] The code and the problem files can be found at http://ai-server.uom.gr/pocl

threats generated by emutex or permanent cmutex relations. In these cases, ties were broken by favoring promotion of new actions. As we can see, there was a significant increase in time needed to solve even the easiest problem. Of course, one could apply heuristics for selecting among alternative resolutions of threats, but this was out of the scope of this paper.

Columns *–Pruning 1* concern the case where the repeated subgoal generation pruning rule was not used. We observe a significant negative impact in system's performance. In columns *–Prunning 1 older* ties are broken by preferring older plans. The results in this case are similar as in the *Older* case, although we receive slightly longer solution times.

We do not present results for the case where the second pruning rule was inactivated. In this case the planner failed to solve any problem, because the CSP subproblem generated by the planning procedure was unsolvable. Note that with the second pruning rule being active, no unsolvable CSP problem was generated ever.

Finally, we tried to run the system without considering threats generated by the temporary cmutex relations. In this case we did not notice any significant difference to the *Full* case. This was the case even when we tightened the upper bound. This is explained by the fact that large disjunctive constraints do not allow for propagation, especially in the presence of good heuristics. However, we think that the role of temporary cmutex relations needs further investigation.

Conclusions and Future Work

In this paper we presented a temporal POCL planning system, which exploits a temporal planning graph in order to extract heuristic values and post disjunctive constraints to resolve threats. We used a domain independent temporal heuristic that, among others, takes into account groups of permanently mutexed open goals and sums only the maximum of their heuristic values. Finally, we presented two completeness preserving pruning rules that are well suited in the framework of stratified (i.e. separated planning and scheduling) POCL temporal planning.

We believe that the use of disjunctive constraints is well suited in the framework of stratified heuristic POCL temporal planning, since it avoids the problem of overgenerating child plans with the same or similar heuristic values. On the other hand, disjunctive constraints demand for stronger propagation rules to exploit pruning challenges. In the future we aim to work at this direction.

References

Blum, A.L., and Furst,M.L. 1997. Fast planning through planning graph analysis. *Artificial Intelligence*, 90 (1-2), 281-300.

Bonet, B., and Geffner, H. 2001. Planning as Heuristic Search. *Artificial Intelligence*, 129 (1-2), 5-33.

Bonet, B., Loerincs, G. & Geffner, H. (1997). A robust and fast action selection mechanism for planning. *In Proceedings of 14th National Conf. on Artificial Intelligence*, 714-719, Providence, RI. AAAI Press.

Ghallab, M., and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. In *Proc. of 2nd International Conf. on Artificial Intelligence Planning Systems*, 61-67, Chicago, IL: AAAI Press.

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

Joslin, D., and Pollack, M. 1994. Least-cost flaw repair: A plan refinement strategy for partial-order planning. In *Proc. of 12th National Conf. on AI*, 1004-1009, Seattle, WA. AAAI Press.

McAllester, D.A., and Rosenblitt, D. 1991. Systematic nonlinear planning. *Proceedings of Ninth National Conference on Artificial Intelligence*, 634-639. Anaheim, CA: AAAI Press.

Nguyen, X., and Kambhampati, S. 2000. Extracting effective and admissible heuristics from the planning graph. In *Proc. of 17th National Conf. on Artificial Intelligence*, 798-805, Austin, TX: AAAI Press.

Penberthy, J.S. and Weld, D.S. 1994. Temporal planning with continuous change. In Proc. of the 12th National Conference on Artificial Intelligence, 1010-1015, Seattle, WA: AAAI Press.

Penberthy, J.S., and Weld, D.S. 1992. UCPOP: A sound and complete, partial order planner for ADL. *In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, 103-114. Cambridge, MA: Morgan Kaufmann Publishers.

Peot, M., and Smith, D. 1992. Conditional nonlinear planning. *In Proceedings of the first conf. on AI planning systems*, 189-197.

Peot, M., and Smith, D. 1993. Threat-removel strategies for partial order planning. *Proc. of the 11th National Conf. on AI*, 492-499.

Pryor, L., and Collins, G. 1996. Planning for contingencies: a decision-based approach. *Journal of Artificial Intelligence Research*, 4, 287-339.

Refanidis, I., and Vlahavas, I. 2001. GRT: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research*, 15 (2001), pp. 115-161.

Schubert, L., and Gerevini, A. 1995. Accelerating partial order planners by improving plan and goal choices. In *Proceedings of the 7th Intern. Conf. on Tools with Artificial Intelligence*, 442-450, Hernodn, VA. IEEE Computer Society Press.

Smith, D., and Weld., D. 1999. Temporal planning with mutual exclusion reasoning. *Proc. of the 16th Intern. Joint Conf. on Artificial Intelligence*, 326,333.

Smith, D.E., Frank, J., and Jonsson, A.K. 2000. Bridging the gap between planning and scheduling. *Knowledge Engineering Review*, 15(1), 47-83.

Vere, S.A. 1983. Planning in time: Windows and durations for activities and goals. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 246-267.

Vidal, V., and Geffner, H. 2004. Branching and Pruning: An Optimal Temporal POCL Planner based on Constraint Programming. In *Proc. 19th National Conference on Artificial Intelligence*, San Jose, CA: AAAI Press.

Younes, L.S.H., and Simmons, R.G. 2003. VHPOP: Versatile Heuristic Partial Order Planner. *Journal of Artificial Intelligence Research*, 20, 405-430.