Sapa: A Domain-Independent Heuristic Metric Temporal Planner

Minh B. Do & Subbarao Kambhampati *
Department of Computer Science and Engineering
Arizona State University, Tempe AZ 85287-5406
{binhminh,rao}@asu.edu
http://rakaposhi.eas.asu.edu/sapa.html

Abstract

Many real world planning problems require goals with deadlines and durative actions that consume resources. In this paper, we present *Sapa*, a domain-independent heuristic forward chaining planner that can handle durative actions, metric resource constraints, and deadline goals. The main innovation of *Sapa* is the set of distance based heuristics it employs to control its search. We consider both optimizing and satisficing search. For the former, we identify admissible heuristics for objective functions based on makespan and slack. For satisficing search, our heuristics are aimed at scalability with reasonable plan quality. Our heuristics are derived from the "relaxed temporal planning graph" structure, which is a generalization of planning graphs to temporal domains. We also provide techniques for adjusting the heuristic values to account for resource constraints. Our experimental results indicate that *Sapa* returns good quality solutions for complex planning problems in reasonable time.

1 Introduction

For most real world planning problems, the STRIPS model of classical planning with instantaneous actions is inadequate. We normally need plans with durative actions that execute concurrently. Moreover, actions may consume resources and the plans may need to achieve goals within given deadlines. While there have been efforts aimed at building metric temporal planners that can handle different types of constraints beyond the classical planning specifications [14, 9, 11], most such planners either scale up poorly or need hand-coded domain control knowledge to guide their search. The biggest problem faced by existing temporal planners is thus the control of search (c.f. [17]). Accordingly, in this paper, we address the issues of domain independent heuristic control for metric temporal planners.

At first blush search control for metric temporal planners would seem to be a very simple matter of adapting the work in heuristic planners in classical planning [3, 12, 7]. The adaptation however does pose several challenges. To begin with, metric temporal planners tend to have significantly larger search spaces than classical planners. After all, the problem of planning in the presence of durative actions and metric resources subsumes both the classical planning and scheduling problems. Secondly, the objective of planning may not be limited to simple goal satisfaction, and may also include optimization of the associated schedule (such as maximum lateness, weighted tardiness, weighted completion time, resource consumption etc. [15]). Finally, the presence of metric and temporal constraints, in addition to subgoal interactions, opens up many more potential avenues for extracting heuristics (based on problem relaxation). Thus, the question of which relaxations provide best heuristics has to be carefully investigated.

In this paper, we present *Sapa*, a heuristic metric temporal planner that we are currently developing. *Sapa* is a forward chaining metric temporal planner, whose basic search routines are adapted from Bacchus and Ady's[1] recent work on temporal TLPlan. We consider a forward chaining planner because of the advantages offered by the complete state information in handling metric resources [17]. Unlike temporal TLPlan, which relies on hand-coded control knowledge to guide the planner, the primary focus of our work is on developing

*We thank David E. Smith, Terry Zimmerman and three anonymous reviewers for useful comments on the earlier drafts of this paper. This research is supported in part by the NSF grant IRI-9801676, and the NASA grants NAG2-1461 and NCC-1225.

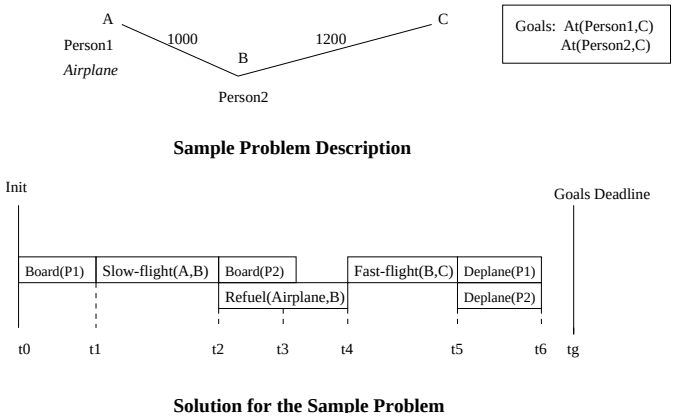

Figure 1: Sample problem description and its solution

distance based heuristics to guide the search. In *Sapa*, we estimate the heuristic values by doing a phased relaxation: we first derived heuristics from a relaxation that ignores the delete effects and metric resource constraints, and then adjust these heuristics to better account for resource constraints. In the first phase, we use a generalization of the planning graphs [2], called relaxed temporal planning graphs (RTPG), as the basis for deriving the heuristics. Our use of planning graphs is inspired by (and can be seen as an adaptation of) the recent work on *AltAlt* [12] and FF [7]. We consider both optimizing and satisficing search scenarios. For the former, we develop admissible heuristics for objective functions based on makespan or slack. For the latter, we develop very effective heuristics that use the characteristics of a "relaxed" plan derived from the planning graphs. Finally, we present a way of improving the informedness of our heuristics by accounting for the resource constraints (which are ignored in constructing the relaxed planning graphs).

Sapa is implemented in Java. Our empirical studies indicate that *Sapa* can solve planning problems with complex temporal and resource constraints quite efficiently. *Sapa* also returns good quality solutions with short makespans and very few irrelevant actions. This is particularly encouraging given that temporal TLPlan, the leading contender of *Sapa* that uses hand-coded control knowledge, tends to output many irrelevant actions.

The rest of this paper describes the development and evaluation of *Sapa.* We start in Section 2 with a discussion of action representation and the general search algorithm used in *Sapa.* In Section 3, we present the relaxed planning graph structure and discuss different heuristics extracted from it. We also describe how to adjust the heuristic values based on the metric resource constraints. We present empirical results in Section 4 and conclude the paper with a discussion of related work and future work in Sections 5 and 6.

2 Handling concurrent actions in a forward state space planner

Sapa addresses planning problems that involve durative actions, metric resources, and deadline goals. In this section, we describe how such planning problems are represented and solved in *Sapa.* We will first describe the action representation, and will then present the forward chaining state search algorithm used by *Sapa.*

To illustrate the representation and the search algorithm used in *Sapa*, we will use a small example from the flying domain discussed in [14]. In this domain, which we call *zeno-flying*, airplanes move passengers between cities. An airplane can choose between "slow flying" and "fast flying" actions. "Slow flying" travels at 400 miles/hr and consumes 1 gallon of fuel for every 3 miles. "Fast flying" travels at 600 miles/hr and consumes 1 gallon of fuel every 2 miles. Passengers can be *boarded* in 30 minutes and *deplaned* in 20 minutes. The fuel capacity of the airplane is 750 gallons and it takes 60 minutes to *refuel* it. Figure 1 shows a simple problem from this domain that we will use as a running example throughout the paper. In this problem, Person1 and the Airplane are at cityA, Person2 is at cityB and the plane has 500 gallons of fuel in the initial state. The goals are to get both Person1 and Person2 to cityC in 6.5 hours. One solution for this problem, shown in the lower half of Figure 1, involves first *boarding* Person1 at cityA, and then *slow-flying* to cityB. While *boarding* Person2 at cityB, we can *refuel* the plane concurrently. After finishing refueling, the plane will have enough fuel to *fast-fly* to cityC and *deplane* the two passengers.

```
(:action BOARD
    :parameters
    (?person - person ?airplane - plane ?city - city)
    :duration (st, + st 30)
    :precondition
    (and (at ?person ?city) - (st,st)
         (in-city ?airplane ?city) - (st,et))
    :effect
    (and (not (at ?person ?city)) - st
         (in ?person ?airplane) - et))
```

```
(:action SLOW-FLYING
    :parameters
    (?airplane - plane ?city1 - city ?city2 - city)
    :duration
    (st, + st (/ (distance ?city1 ?city2)
                 (slow-speed ?airplane)))
    :precondition
    (and (in-city ?airplane ?city1) - (st,st)
         (> (fuel ?airplane) 0) - (st,et))
    :effect
    (and (not (in-city ?airplane ?city1)) - st
         (in-city ?airplane ?city2) - et
         (-= (fuel ?airplane)
            (* #t (sf-fuel-cons-rate ?airplane))) - #t))
```

Figure 2: Examples of action descriptions in *Sapa*

2.1 Action representation

Planning is the problem of finding a set of actions and their respective execution times to satisfy all causal, metric, and resource constraints. Therefore, action representation has influences on the representation of the plans and on the planning algorithm. In this section, we will discuss the action representation used in *Sapa*. Our representation is influenced by the PDDL+ language proposal[4] and the representations used in Zeno[14] and LPSAT[19] planners.

Unlike actions in classical planning, in planning problems with temporal and resource constraints, actions are not instantaneous but have durations. Their preconditions may either be instantaneous or durative and their effects may occur at any time point during their execution. Each action *A* has a duration D_A, starting time S_A, and end time E_A (= $S_A + D_A$). The value of D_A can be statically defined for a domain, statically defined for a particular planning problem, or can be dynamically decided at the time of execution.[1] Action *A* have preconditions $Pre(A)$ that may be required either to be instantaneously true at the time point S_A, or required to be true starting at S_A and remain true for some duration $d \leq D_A$. The logical effects *Eff(A)* of *A* will be divided into three sets $E_s(A)$, $E_e(A)$, and $E_m(A, d)$ containing respectively instantaneous effects at time points S_A, E_A and $S_A + d$ ($0 < d < D_A$). Figure 2 illustrates the actual representations used in *Sapa* for actions *boarding* and *slow-flying* in the zeno-flying domain. Here, *st* and *et* denote the starting and ending time points of an action, while *#t* represents a time instant between *st* and *et*. While the action $boarding(person, airplane, city)$ requires a person to be at the location *city* only at its starting time point *st*, it requires an airplane to stay there the duration of its execution. This action causes an instant effect $(not(at(?person, ?city)))$ at the starting time point *st* and the delayed effect $in(?person, ?airplane)$ at the ending time point *et*.

Actions can also consume or produce metric resources and their preconditions may also well depend on the value of the corresponding resource. For resource related preconditions, we allow several types of equality or inequality checking including ==, <, >, <=, >=. For resource-related effects, we allow the following types of change (update): assignment(=), increment(+=), decrement(-=), multiplication(*=), and division(/=). In Figure 2, the action *slow-flying* requires the fuel level to be greater than zero over the entire duration of execution and consumes the fuel at a constant rate while executing.

Currently we only model and test domains in which effects occur at the start or end time points, and preconditions are required to be true at the starting point or should hold true throughout the duration of that action. Nevertheless, the search algorithm and the domain representation schema used in *Sapa* are general enough to represent and handle actions with effects occurring at any time point during their durations and preconditions that are required to hold true for any arbitrary duration between the start and end time points of an action. In the near future, we intend to test our planner in domains that have more flexible temporal constraints on the preconditions and effects of actions.

[1] For example, in the zeno-flying domain discussed earlier, we can decide that boarding a passenger always takes 10 minutes for all problems in this domain. Duration of the action of flying an airplane between two cities will depend on the distance between these two cities. Because the distance between two cities will not change over time, the duration of a particular flying action will be totally specified once we parse the planning problem. However, *refueling* an airplane may have a duration that depends on the current fuel level of that airplane. We may only be able to calculate the duration of a given *refueling* action according to the fuel level at the exact time instant when we execute that action.

2.2 A forward chaining search algorithm

Even though variations of the action representation scheme described in the previous section have been used in the partial order temporal planners such as IxTeT[9] and Zeno[14] before, Bacchus and Ady [1] are the first to propose a forward chaining algorithm capable of using this type of action representation and allow concurrent execution of actions in the plan. We adapt their search algorithm in *Sapa.*

Before going into the details of the search algorithm, we need to describe some major data structures that are used. *Sapa*'s search is conducted through the space of time stamped states. We define a time stamped state S as a tuple $S = (P, M, \Pi, Q, t)$ consisting of the following structure:

- $P = (\langle p_i, t_i \rangle \mid t_i < t)$ is a set of predicates p_i that are true at *t* and the last time instant t_i at which they are achieved.[2]
- *M* is a set of values of all functions representing all the metric-resources in the planning problem. Because the continuous values of resource levels may change over the course of planning, we use *functions* to represent the resource values.
- Π is a set of persistent conditions, such as action preconditions, that need to be protected during a period of time.
- *Q* is an event queue containing a set of updates each scheduled to occur at a specified time in the future. An event *e* can do one of three things: (1) change the True/False value of some predicate, (2) update the value of some function representing a metric-resource, or (3) end the persistence of some condition.
- t is the time stamp of S

In this paper, unless noted otherwise, when we say "state" we mean a time stamped state. It should be obvious that time stamped states do not just describe world states (or snap shots of the world at a given point of time) as done in classical progression planners, but rather describe both the state of the world and the state of the planner's search.

The initial state S_{init} is stamped at time 0 and has an empty event queue and empty set of persistent conditions. However, it is completely specified in terms of function and predicate values. In contrast, the goals do not have to be totally specified. The goals are represented by a set of n 2-tuples $G = (\langle p_1, t_1 \rangle ... \langle p_n, t_n \rangle)$ where p_i is the i^{th} goal and t_i is the time instant by which p_i needs to be achieved.

Goal Satisfaction: The state $S = (P, M, \Pi, Q, t)$ *subsumes* (entails) the goal *G* if for each $\langle p_i, t_i \rangle \in G$ either:

1. $\exists \langle p_i, t_j \rangle \in P$, $t_j < t_i$ and there is no event in *Q* that deletes p_i.
2. There is an event $e \in Q$ that adds p_i at time instant $t_e < t_i$.

Action Application: An action A is *applicable* in state $S = (P, M, \Pi, Q, t)$ if:

1. All instantaneous preconditions of *A* are satisfied by *P* and *M*.
2. *A*'s effects do not interfere with any persistent condition in Π and any event in *Q*.
3. No event in *Q* interferes with persistent preconditions of *A*.

When we apply an action *A* to a state $S = (P, M, \Pi, Q, t)$, all instantaneous effects of *A* will be immediately used to update the predicate list *P* and metric resources database *M* of *S*. *A*'s persistent preconditions and delayed effects will be put into the persistent condition set Π and event queue *Q* of *S*. For example, if we apply action *Board(P1,airplane)* to the initial state of our running example in Figure 1, then the components of resulting state *S* will become $P = \{\langle At(airplane, A), t_0 \rangle, \langle In(P1, airplane), t_0 \rangle, \langle At(P2, B), t_0 \rangle\}$, *M* = {*Fuel(airplane)=500*}, $\Pi = \{\langle At(airplane, A), t_1 \rangle\}$, and $Q = \{\langle In(P1, airplane), t_1 \rangle\}$.

Besides the normal actions, we will have one special action called **advance-time**[3] which we use to advance the time stamp of *S* to the time instant t_e of the earliest event e in the event queue *Q* of *S*. The

[2] For example, at time instant t_1 in Figure 1, $P = \{\langle At(airplane, A), t_0 \rangle, \langle At(Person2, B), t_0 \rangle, \langle In(Person1, t_1) \rangle\}$

[3] *Advance-time* is called **unqueue-event** in [1]

State Queue: *SQ*={S_{init}}
while *SQ*≠{}
    S:= *Dequeue*(*SQ*)
    Nondeterministically select *A* applicable in *S*
        S' := *Apply*(*A*,*S*)
        if *S'*⊨ *G* **then**
            PrintSolution
        else *Enqueue*(*S'*,*SQ*)
end while;

Figure 3: Main search algorithm

advance-time action will be applicable in any state *S* that has a non-empty event queue. Upon applying this action, we update state *S* according to all the events in the event queue that are scheduled to occur at t_e.

Notice that we do not consider action *A* to be applicable if it causes some event *e* that interferes with an event e' in the event queue, even if *e* and e' occur at different time points. We believe that even though an event has instant effect, there should be some underlying process that leads to that effect.[4] Therefore, we feel that if two actions cause instant events that are contradicting with each other, then even if the events occur at different time points, the underlying processes supporting these two events may contradict each other. Thus, these two actions are not allowed to execute concurrently. Our approach can be considered as having a *hold* process [5] extending from the starting point of an action to the time point at which an event occurs. The hold process protects that predicate from violations by conflicting events from other actions. This also means that even though an effect of a given action *A* appears to change the value of a predicate at a single time point *t*, we implicitly need a duration from the starting point *st* of *A* to *t* for it to happen. We are currently investigating approaches to represent constraints to protect a predicate or resource more explicitly and flexibly. Additionally, in handling metric resource interactions between two actions, *Sapa* follows an approach similar to the ones used by Zeno[14] and RIPP[8]: it does not allow two actions that access the same metric resource to overlap with each other. By not allowing two actions affecting the same resource to overlap, we can safely change the resource condition that needs to be preserved during an action to be an instantaneous condition or an update at the start or end point of that action. For example, the condition that the fuel level of an airplane should be higher than 0 while flying between two cities, can be changed to a check to see if the level of fuel it has at the beginning of the action is higher than the amount that will be consumed during the course of that action. This helps in simplifying the search algorithm. In future, we intend to investigate other ways to relax this type of resource interaction constraints.

Search algorithm: The basic algorithm for searching in the space of time stamped states is shown in Figure 3. We proceed by applying all applicable actions to the current state and put the result states into the sorted queue using the $Enqueue()$ function. The $Dequeue()$ function is used to take out the first state from the state queue. Currently, *Sapa* employs the A* search. Thus, the state queue is sorted according to some heuristic function that measures the difficulty of reaching the goals from the current state. The rest of the paper discusses the design of heuristic functions.

3 Heuristic control

For any type of planner to work well, it needs to be armed with good heuristics to guide the search in the right direction and to prune the bad branches early. Compared with heuristic forward chaining planners in classical planning, *Sapa* has many more branching possibilities. Thus, it is even more critical for *Sapa* to have good heuristic guidance.

Normally, the design of the heuristics depends on the objective function that we want to optimize; some heuristics may work well for a specific objective function but not others. In a classical planning scenario, where actions are instantaneous and do not consume resources, the quality metrics are limited to a mere count of actions or the parallel execution time of the plan. When we extend the classical planning framework to handle durative actions that may consume resources, the objective functions need to take into account

[4]For example, the *boarding* action will cause the event of the passenger being inside the plane at the end of that action. However, there is an underlying process of taking the passenger from the gate to inside the plane that we are not mentioning about.

Heuristic	Objective Function	Basis	Adm.	Use res-infor
Max-span	minimize makespan	RTPG	Yes	No
Min-slack	maximize minimum slack	RTPG	Yes	No
Max-slack	maximize maximum slack	RTPG	Yes	No
Sum-slack	maximize sum-slack	RTPG	Yes	No
Sum-action	minimize number of actions	relaxed plan	No	No
Sum-duration	minimize sum of action durations	relaxed plan	No	No
Adj. sum-act.	minimize number of actions	relaxed plan	No	Yes
Adj. sum-dur.	minimize sum of action durations	relaxed plan	No	Yes

Table 1: Different heuristics investigated in *Sapa*. Columns titled "objective function", "basis", "adm" and "use res-infor" show respectively the objective function addressed by each heuristic, the basis to derive the heuristic values, the admissibility of the heuristic, and whether or not resource-related information is used in calculating the heuristic values.

other quality metrics such as the makespan, the amount of slack in the plan and the amount of resource consumption. Heuristics that focus on these richer objective functions will in effect be guiding both planning and scheduling aspects. Specifically, they need to control both action selection and the action execution time.[5]

In this paper, we consider both satisficing and optimizing search scenarios. In the former, our focus is on efficiently finding a reasonable quality plan. In the later, we are interested in the optimization of objective functions based on makespan, or slack values. We will develop heuristics for guiding both types of search. Table 1 provides a high level characterization of the different heuristics investigated in this paper, in terms of the objective functions that they are aimed at, and the knowledge used in deriving them.

For any type of objective function, heuristics are generally derived from relaxed problems, with the understanding that the more constraints we relax, the less informed the heuristic becomes [13]. Exploiting this insight to control a metric temporal planner brings up the question of what constraints to relax. In classical planning, the "relaxation" essentially involves ignoring precondition/effect interactions between actions [3, 7]. In metric-temporal planning, we can not only relax the logical interactions, but also the metric resource constraints, and temporal duration constraints.

In *Sapa*, we estimate the heuristic values by doing a phased relaxation: we first relax the delete effects and metric resource constraints to compute the heuristic values, and then modify these values to better account for resource constraints. In the first phase we use a generalization of the planning graphs [2], called relaxed temporal planning graphs (RTPG), as the basis for deriving the heuristics. Our use of planning graphs is inspired by (and can be seen as an adaptation of) the recent work on *AltAlt* [12] and FF [7]. The RTPG structures are described in Section 3.1, and Sections 3.2 and 3.3 describe the extraction of admissible and effective heuristics from the RTPG. Finally, in Section 3.4, we discuss a technique for improving the informedness of our heuristics by adjusting the heuristic values to account for the resource constraints (which are ignored in the RTPG).

3.1 Building the relaxed temporal planning graph

All our heuristics are based on the relaxed temporal planning graph structure (RTPG). This is a Graphplan-style[2] bi-level planning graph generalized to temporal domains. Given a state $S = (P, M, \Pi, Q, t)$, the RTPG is built from S using the set of relaxed actions, which are generated from original actions by eliminating all effects which (1) delete some fact (predicate) or (2) reduce the level of some resource. Since delete effects are ignored, RTPG will not contain any mutex relations, which considerably reduces the cost of constructing RTPG. The algorithm to build the RTPG structure is summarized in Figure 4. To build the RTPG, we need three main datastructures: a fact level, an action level, and an unexecuted event queue.[6] Each fact *f* or action *A* is marked *in*, and appears in the RTPG's fact/action level at time instant t_f/t_A if it can be achieved/executed at t_f/t_A. In the beginning, only facts which appear in *P* are marked *in* at *t*, the action level is empty, and the event queue holds all the unexecuted events in *Q* that add new predicates. Action *A* will be marked *in* if (1) *A* is not already marked *in* and (2) all of *A*'s preconditions are marked *in*. When action *A* is

[5] In [17], Smith et. al. discuss the importance of the choice of actions as well as the ordering between them in solving complicated real world planning problems involving temporal and resource constraints.

[6] Unlike the initial state, the event queue of the state *S* from which we build the RTPG may be.

while(true)
  forall A $\neq$*advance-time* applicable in S
    S := Apply(A,S)
    if S $\models$ G **then** *Terminate*{*solution*}

  S' := Apply(*advance-time*,S)
  if $\exists$ $\langle p_i, t_i \rangle \in$G such that
    $t_i <$ Time(S') and $p_i \notin$S **then**
    Terminate{*non-solution*}
  else S := S'
end while;

Figure 4: Algorithm to build the relaxed temporal planning graph structure.

in, then all of *A*'s unmarked instant add effects will also be marked *in* at *t*. Any delayed effect *e* of *A* that adds fact *f* is put into the event queue *Q* if (1) *f* is not marked *in* and (2) there is no event e' in *Q* that is scheduled to happen before *e* and which also adds *f*. Moreover, when an event *e* is added to *Q*, we will take out from *Q* any event e' which is scheduled to occur after *e* and also adds *f*.

When there are no more unmarked applicable actions in *S*, we will stop and return *no-solution* if either (1) *Q* is empty or (2) there exists some unmarked goal with a deadline that is smaller than the time of the earliest event in *Q*. If none of the situations above occurs, then we will apply *advance-time* action to *S* and activate all events at time point $t_{e'}$ of the earliest event *e'* in *Q*. The process above will be repeated until all the goals are marked *in* or one of the conditions indicating *non-solution* occurs. Figure 5 shows the RTPG for the state *S* at time point t_1 (refer to Figure 1) after we apply action *Board(P1)* to the initial state and *advance* the clock from t_0 to t_1.

In *Sapa*, the RTPG is used to:

- Prune the states that can not lead to any solution.
- Use the time points at which goals appear in the RTPG as the lower bounds on their time of achievements in the real plans.
- Build a relaxed plan that achieves the goals, which can then be used as a basis to estimate the distance from S to the goals.

For the first task, we will prune a state if there is some goal $\langle p_i, t_i \rangle$ such that p_i does not appear in the RTPG before time point t_i.

Proposition 1: *Pruning a state according to the relaxed temporal planning graph (RTPG) preserves the completeness of the planning algorithm.*

The proof is quite straight forward. Since we relaxed the delete effects and resource related constraints of all the actions when building the graph structure, and applied all applicable actions to each state, the time instant at which each predicate appears in the RTPG is a *lower bound* on its real time of achievement. Therefore, if we can not achieve some goal on time in the relaxed problem, then we definitely will not be able to achieve that goal with the full set of constraints.

In the next several sections, we will discuss the second task, that of deriving different heuristic functions from the RTPG structure.

3.2 Admissible heuristics based on action durations and deadlines

In this section, we will discuss how several admissible heuristic functions can be derived from the RTPG. First, from the observation that all predicates appear at the earliest possible time in the relaxed plan graph, we can derive an admissible heuristic which can be used to *optimize the makespan* of the solution. The heuristic is defined as follows:

Max-span heuristic: *Distance from a state to the goals is equal to the length of the duration between the time-instant of that state and the time the last goal appears in the RTPG .*

The max-span heuristic is admissible and can be used to find the smallest makespan solution for the planning problem. The proof of admissibility is based on the same observation made in the proof of Proposition

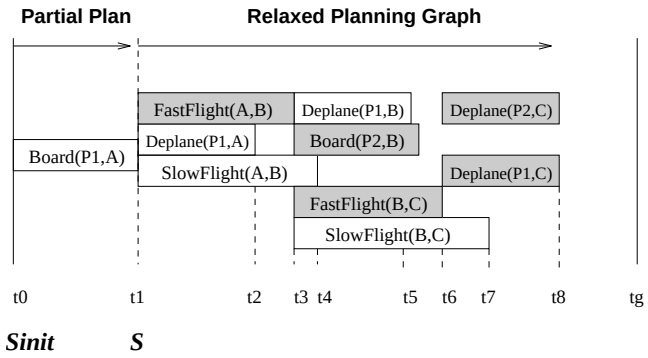

Figure 5: Sample relaxed temporal planning graph for durative actions. Shaded actions are the ones appear in the relaxed plan.

1. Because all the goals appear in the RTPG at the time instants that are *lower bounds* on the their real time of achievements, the time instant at which the last goal appears in the RTPG will be the lower bound on the actual time point at which we can achieve all the goals. Thus, it is a lower bound on the makespan of the solution.

The max-span heuristic discussed so far can be thought of as a generalized version of the max-action heuristic used in HSP [3] or max-level heuristic in AltAlt [12]. One of the assumptions in classical planning is that the goals have no deadlines and they need only be achieved by the end of the plan. Therefore, all heuristics concentrate on measuring how far the current state is to the point by which *all* the goals are achieved. However, in temporal planning with deadline goals, we can also measure the 'slack' values for the goals as another plan quality measurement (where slack is the difference in time between when the goal was achieved in the paln, and the deadline specified for its achievement). The slack values for a given set of goals can also be a good indication on how hard it is to achieve those goals, and thus, how hard it is to solve a planning problem from a given state. Moreover, slack-based objective functions are common in scheduling.

We will consider objective functions to maximize the minimum, maximum, or summation of slack values of all the goals for the temporal planning problems. In our case, the slack value for a given goal g is estimated from the RTPG by taking the difference between the time instant at which g appears in the RTPG and its deadline. We now present admissible heuristics for these three slack based objective functions.

Min-slack heuristic: *Distance from a state to the goals is equal to the minimum of the slack estimates of all individual goals.*[7]

Max-slack heuristic: *Distance from a state to the goals is equal to the maximum of slack estimates of all individual goals.*

Sum-slack heuristic: *Distance from a state to the goals is equal to the summation of slack estimates for all individual goals.*

The min-slack, max-slack, and sum-slack heuristics target the objective functions of maximizing the minimum slack, maximum slack, and the summation of all slack values. The admissibility of the three heuristics for the respective objective functions can be proven using the same argument we made for the max-span heuristic. Specifically, we use the observation that all goals appear in the RTPG at time instants earlier than the actual time instants at which they can be achieved, to prove that the slack estimated calculated using the RTPG for any goal will be the *upper bound* on its actual slack value for the non-relaxed problem.

3.3 Heuristics for efficient satisficing search

We now focus on efficiently finding reasonable quality plans. In the last section, we discussed several admissible heuristics which can be used to find optimal solution according to some objective functions. However, admissible heuristics such as max-span and slack-based heuristics are only concerned about the time points at which goals are achieved, and not the length of the eventual plan. In classical planning, heuristics that use an estimate on the length of the plan have been shown to be more effective in controlling search [7, 12]. To

[7] If all the goals have the same deadlines, then maximizing the minimum slack is equal to minimizing the makespan of the plan and the two heuristic values (max-span and min-slack) can be used interchangably.

estimate the length of the solution, these planners typically use a valid plan extracted from the relaxed planning graph (the relaxation typically involves ignoring negative interactions). We can use a similar heuristic for temporal planning.

Sum-action heuristic: *Distance from a state to the goals is equal to the number of actions in the relaxed plan*.

The relaxed plan can be built backward from the goals in a manner nearly identical to the procedure used in Graphplan algorithm[2] in classical planning. We first start with the goals and add actions that support them to the solution. If we add an action to the solution, then its preconditions are also added to the set of current goals. The search continues until we "reach" the initial state (i.e the goals are entailed by the initial state). In our continuing example, the shaded actions in Figure 5 are the ones that appear in the relaxed plan when we search backward.

Finally, since actions have different durations, the sum of the durations of actions in the relaxed plan is another way to measure the difficulty in achieving the goals.

Sum-duration heuristic: *Distance from a state to the goals is equal to the sum of durations of actions in the relaxed plan*.

If all actions have the same durations, then the sum of durations of all actions in the relaxed plan will be equivalent to taking the number of actions in the plan. Thus, in this case, sum-action and sum-duration will perform exactly the same. Neither of these heuristics are admissible; searches using the sum-action or sum-duration heuristics do not guarantee to return the solutions with smallest number of actions, or solutions with smallest summation of action durations. The reason is that these two heuristics have their values based on a *first* (relaxed) plan found. There is no guarantee that that first relaxed plan will be smaller than the smallest real (non-relaxed) plan in terms of number of actions, or summation of durations of actions in the plan.

3.4 Using metric resource constraints to adjust heuristic values

The heuristics discussed in the last two sections have used the knowledge about durations of actions and deadline goals but not about resource consumption. By ignoring the resource related effects when building the relaxed plan, we may miss counting actions whose only purpose is to give sufficient resource-related conditions to other actions.[8] Consequently, ignoring resource constraints may reduce the quality of heuristic estimate based on the relaxed plan. We are thus interested in adjusting the heuristic values discussed in the last two sections to account for the resource constraints.

In real-world problems, most actions consume resources, while there are special actions that increase the levels of resources. Since checking whether the level of a resource is sufficient for allowing the execution of an action is similar to checking the predicate preconditions, one obvious approach to adjust the relaxed plan would be to add actions that provide that resource-related condition to the relaxed plan. For reasons discussed below, it turns out to be too difficult to decide which actions should be added to the relaxed plan to satisfy the given resource conditions. First, actions that consume/produce the same metric-resource may overlap over the RTPG and thus make it hard to reason about the resource level at each time point. In such cases, the best we can do is to find the upper bound and lower bound values on the value of some specific resource. However, the bounds may not be very informative in reasoning about the exact value. Second, because we do not know the values of metric resources at each time point, it is difficult to reason as to whether or not an action needs another action to support its resource-related preconditions. For example, in Figure 5, when we add the action *fast-flying*(B, C) to the relaxed plan, we know that that action will need $fuel(airplane) > 400$ as its precondition. However, without the knowledge about the value (level) of $fuel(airplane)$ at that time point, we can hardly decide whether or not we need to add another action to achieve that precondition. If we reason that the fuel level at the initial state ($fuel(airplane) = 500$) is sufficient for that action to execute, then we already miss one unavoidable $refuel(airplane)$ action (because most of the fuel in the initial state has been used for the other flying action, *fast-flying*(A, B)). A final difficulty is that because of the continuous nature of the metric resources, it is harder to reason if an action *gives* a resource-related effect to another action and whether or not it is logically relevant to do so. For example, suppose that we need to fly an airplane from *cityA* to *cityB* and we need to refuel to do so. Action $refuel(airplane, cityC)$ gives the fuel that the airplane needs, but it is totally irrelevant to the plan. Adding that action to the relaxed plan (and its preconditions to

[8] For example, if we want to drive a truck to some place and the fuel level is low, by totally ignoring the resource related conditions, we will not realize that we may need to *refuel* the truck before *driving* it.

the goal set) will lead to the addition of irrelevant actions, and thus reduce the quality of heuristic estimates it provides.

In view of the above complications, we introduce a new way of readjusting the relaxed plan to take into account the resource constraints as follows: we first preprocess the problem specifications and find for each resource *R* an action A_R that can increase the amount of *R* maximally. Let Δ_R be the amount by which A_R increases *R*, and let $Dur(A_R)$ be the duration of A_R. Let $Init(R)$ be the level of resource *R* at the state *S* for which we want to compute the relaxed plan, and $Con(R)$, $Pro(R)$ be the total consumption and production of *R* by all actions in the relaxed plan. If $Con(R) > Init(R) + Pro(R)$, we use the following formula to adjust the heuristic values of the sum-action and sum-duration according to the resource consumption.

Sum-action heuristic value h:

$$h \leftarrow h + \sum_R \left\lceil \frac{Con(R) - (Init(R) + Pro(R))}{\Delta_R} \right\rceil$$

Sum-duration heuristic value h:

$$h \leftarrow h + \sum_R \frac{Con(R) - (Init(R) + Pro(R))}{\Delta_R} * Dur(A_R)$$

We will call the newly adjusted heuristics **adjusted sum-action** and **adjusted sum-duration**. The basic idea is that even though we do not know if an individual resource-consuming action in the relaxed plan needs another action to support its resource-related preconditions, we can still adjust the number of actions in the relaxed plan by reasoning about the total resource consumption of *all* the actions in the plan. If we know how much excess amount of a resource *R* the relaxed plan consumes and what is the maximum increment of *R* that is allowed by any individual action in the domain, then we can infer the minimum number of resource-increasing actions that we need to add to the relaxed plan to balance the resource consumption.

For example, in the relaxed plan for our sample problem, we realize that the two actions *fast-flying*(A, B) and *fast-flying*(B, C) consume a total of: 1000/2 + 1200/2 = 1100 units of fuel, which is higher than the initial fuel level of 500 units. Moreover, we know that the maximum increment for the airplane's fuel is 750 for the *refuel(airplane)* action. Therefore, we can infer that we need to add at least $\lceil (1100-500)/750 \rceil = 1$ refueling action to make the relaxed plan consistent with the resource consumption constraints. The experimental results in Section 4 show that the metric resource related adjustments are quite important in domains which have many actions consuming different types of resources.

The adjustment approach described above is useful for improving the sum-action and sum-duration heuristics, but it can not be used for the max-span and slack-based heuristics without sacrificing their admissibility. In future, we intend to investigate the resource constraint-based adjustments for those heuristics that still preserve their admissibility.

4 Experimental results

We have implemented *Sapa* in Java. To date, our implementation of *Sapa* has been primarily used to test the performance of different heuristics and we have spent little effort on code optimization. We were primarily interested in seeing how effective the heuristics were in controlling the search. In the case of heuristics for satisficing search, we were also interested in evaluating the quality of the solution. We evaluate the performance of *Sapa* on problems from two metric temporal planning domains to see how well it performs in these complex planning problems. The first one is the zeno-flying domain discussed in Section 2.2 [14]. The second is our version of the temporal and metric resource version of the logistics domain. In this domain, trucks move packages between locations within one city, and planes carry them from one city to another. Different airplanes and trucks move with different speeds, have different fuel capacities, different fuel-consumption-rates, and different fuel-fill-rates when refueling. The temporal logistics domain is more complicated than the zeno-flying domain because it has more types of resource-consuming actions. Moreover, the *refuel* action in this domain has a dynamic duration, which is not the case for any action in the zeno-flying domain. Specifically, the duration of this action depends on the fuel level of the vehicle and can only be decided at the time we execute that action.

Table 2 and 3 summarize the results of our empirical studies. Before going into the details, we should mention that among the different types of heuristics discussed in the Section 3, max-span and slack-value

	sum-act		sum-act adjusted		sum-dur		sum-dur adjusted	
prob	time (s)	node	time (s)	node	time (s)	node	time (s)	node
zeno1	0.272	14/48	0.317	14/48	0.35	20/67	0.229	9/29
zeno2	92.055	304/1951	61.66	188/1303	-	-	-	-
zeno3	23.407	200/996	38.225	250/1221	7.72	60/289	35.757	234/1079
zeno4	-	-	37.656	250/1221	7.76	60/289	35.752	234/1079
zeno5	83.122	575/3451	71.759	494/2506	-	-	-	-
zeno6	64.286	659/3787	27.449	271/1291	-	-	30.530	424/1375
zeno7	1.34	19/95	1.718	19/95	1.374	19/95	-	-
zeno8	1.11	27/87	1.383	27/87	1.163	27/87	1.06	14/60
zeno9	52.82	564/3033	16.310	151/793	130.554	4331/5971	263.911	7959/10266
log_p1	2.215	27/159	2.175	27/157	2.632	33/192	2.534	33/190
log_p2	165.350	199/1593	164.613	199/1592	37.063	61/505	-	-
log_p3	-	-	20.545	30/215	-	-	-	-
log_p4	13.631	21/144	12.837	21/133	-	-	-	-
log_p5	-	-	28.983	37/300	-	-	-	-
log_p6	-	-	37.300	47/366	-	-	-	-
log_p7	-	-	115.368	62/531	-	-	-	-
log_p8	-	-	470.356	76/788	-	-	-	-
log_p9	-	-	220.917	91/830	-	-	-	-

Table 2: Solution times and explored/generated search nodes for *Sapa* in the zeno-flying and temporal logistics domains with sum-action and sum-duration heuristics with/without resource adjustment technique. Times are in seconds. All experiments are run on a Sun Ultra 5 machine with 256MB RAM. "-" indicates that the problem can not be solved in 500 seconds.

based heuristics are admissible. However, they do not scale up to reasonable sized problems. As a matter of fact, the max-span heuristic can not solve any problems in Table 2 in the allotted time. The sum-slack heuristic returns an optimal solution (in terms of makespan and sum-slack values) for the problem *Zeno1* in zeno-flying domain in 7.3 seconds, but can not solve any other problems. However, both are able to solve smaller problems that are not listed in our result tables. Because of this, most of our remaining discussion is directed towards sum-action and sum-duration heuristics.

Table 2 shows the running times of *Sapa* for the *sum-action* and *sum-duration* heuristics with and without metric resource constraint adjustment technique (refer to Section 3.4) in the two planning domains discussed above. We tested with 9 problems from each domain. Most of the problems require plans of 10-30 actions, which are quite big compared to problems solved by previous domain-independent temporal planners reported in the literature. The results show that most of the problems are solved within a reasonable time (e.g under 500 seconds). More importantly, the number of nodes (time-stamped states) explored, which is the main criterion used to decide how well a heuristic does in guiding the search, is quite small compared to the size of the problems. In many cases, the number of nodes explored by the best heuristic is only about 2-3 times the size of the plan.

In general, the sum-action heuristic performs better than the sum-duration heuristic in terms of planning time, especially in the logistics domain. However, there are several problems in which the sum-duration heuristic returns better solving times and smaller number of nodes. The metric resource adjustment technique greatly helps the sum-action heuristic, especially in the logistics domain, where without it *Sapa* can hardly solve the bigger problems. We still do not have a clear answer as to why the resource-adjustment technique does not help the sum-duration heuristic.

Plan Quality: Table 3 shows the number of actions in the solution and the duration (makespan) of the solution for the two heuristics analyzed in Table 2. These categories can be seen as indicative of the problem's difficulty, and the quality of the solutions. By closely examining the solutions returned, we found that the solutions returned by *Sapa* have quite good quality in the sense that they rarely have many irrelevant actions. The absence of irrelevant actions is critical in the metric temporal planners as it will both save resource consumption and reduce execution time. It is interesting to note here that the temporal TLPlan[1], whose search algorithm *Sapa* adapts, usually outputs plans with many more irrelevant actions. Interestingly, Bacchus

	sum-act		sum-act adjusted		sum-dur		sum-dur adjusted	
prob	#act	duration	#act	duration	#act	duration	#act	duration
zeno1	5	320	5	320	5	320	5	320
zeno2	23	1020	23	950	-	-	-	-
zeno3	22	890	13	430	13	450	17	400
zeno4	-	-	13	430	13	450	17	400
zeno5	20	640	20	590	-	-	-	-
zeno6	16	670	15	590	-	-	14	440
zeno7	10	370	10	370	10	370	-	-
zeno8	8	320	8	320	8	320	8	300
zeno9	14	560	13	590	13	460	13	430
log_p1	16	10.0	16	10.0	16	10.0	16	10.0
log_p2	22	18.875	22	18.875	22	18.875	-	-
log_p3	-	-	12	11.75	-	-	-	-
log_p4	12	7.125	12	7.125	-	-	-	-
log_p5	-	-	16	14.425	-	-	-	-
log_p6	-	-	21	18.55	-	-	-	-
log_p7	-	-	27	24.15	-	-	-	-
log_p8	-	-	27	19.9	-	-	-	-
log_p9	-	-	32	26.25	-	-	-	-

Table 3: Number of actions and duration (makespan) of the solutions generated by *Sapa* in the zeno-flying and logistics domains with sum-action and sum-duration heuristics with/without resource adjustment technique.

& Ady mention that their solutions are still better than the ones returned by LPSAT[19], which makes our solutions that much more impressive compared to LPSAT.

The pure sum-action heuristic without resource adjustment normally outputs plans with slightly higher number of actions, and longer makespans than the sum-duration heuristic. In some cases, the sum-action heuristic guides the search into paths that lead to very high makespan values, thus violating the deadline goals. After that, the planner has harder time getting back on the right track. Examples of this are zeno-4 and log-p3 which cannot be solved with sum-action heuristic if the deadlines are about 2 times smaller than the optimal makespan (because the search paths keep extending the time beyond the deadlines). The resource adjustment technique not only improves the sum-action heuristic in solution times, but also generally shortens the makespan and occasionally reduces the number of actions in the plan as well. As mentioned earlier, the adjustment technique generally does not help the sum-duration heuristics in solving time, but it does help reduce the makespan of the solution in most of the cases where solutions can be found. However, the set of actions in the plan is generally still the same, which suggests that the adjustment technique does not change the solution, but *pushes* the actions up to an earlier part of the plan. Thus, it favors the execution of concurrent actions instead of using the special action *advance-time* to advance the clock.

When implementing the heuristics, one of the decisions we had to make was whether to recalculate the heuristic value when we advance the clock, or to use the same value as that of the parent node. On the surface, this problem looks trivial and the correct way seems to be to recalculate the heuristic values. However, in practice, keeping the parent node's heuristic value when we advance the clock always seems to lead to solutions with equal or slightly better makespan. We can explain the improved makespan by the fact that recalculating the heuristic value normally favors the *advance-clock* action by outputting a smaller heuristic value for it than the parent. Using many such *advance-clock* actions will lead to solutions with higher makespan values. The solving time comparison is somewhat mixed. Keeping the parent heuristics value speeds up 6 of the 9 problems tested in the logistics domain by average of 2x and slows down about 1.5x in the 3 zeno-flying problems. We do not have a clear answer for the solution time differences between the two approaches. In the current implementation of *Sapa*, we keep the parent node's heuristic value when we advance the clock.

Although we wanted to compare *Sapa* to other planners, there are very few implementations of metric temporal planners with capabilities comparable to *Sapa* that are publicly available and even they tend to scale up poorly. For example, although Zeno is a more expressive planner than *Sapa*, it can not scale up to bigger

problems. The easiest problem in the zeno-flying domain in Table 2 (*Zeno1*) is reported in [14] to be solved by Zeno in several minutes with hand-coded domain control rules.[9] IxTeT is another known planner that we would have like to compare to, but the code is not available and IxTeT's results reported in the literature have concentrated on a class of temporal problems that use discrete, but not metric, resources. In the near future, we intend to compare our planner with TGP[16] and TP4[6] on a simpler set of temporal planning problems that can be handled by all three of them.

5 Related work

There have been several temporal planning systems in the literature that can handle different types of temporal and resource constraints. Among them, planners such as temporal TLPlan[1], Zeno[14], IxTeT[9], and HSTS[11] can solve problems that are similar to the one solved by *Sapa*. There are also planners such as Resource-IPP[8], TP4[6], TGP[16], and LPSAT[19] that can handle a subset of the types of problems discussed in this paper.

Closest to our work is the temporal TLPlan [1], which originates the algorithm to support concurrent actions in the forward state space search. The critical difference between this planner and *Sapa* is that while temporal TLPlan is controlled by hand-coded domain-specific control rules, *Sapa* uses domain-independent heuristics. Experimental results reported in [1] indicate that while Temporal TLPlan is very fast, but it tends to output plans with many irrelevant actions.

There are several partial order planners that can handle various types of temporal and resource constraints. Zeno[14] can solve problems with a wide range of constraints, as well as actions with conditional and quantified effects. However, Zeno lacks heuristic control and scales poorly. IxTeT[9] is another hierarchical partial order planner that can handle many types of temporal and resource constraints. Most of IxTeT's interesting innovations have been aimed at on handling discrete resources such as robots or machines but not on metric resources. HSTS[11] is a partial order planner that has been used to solve NASA temporal planning problems. Like TLPlan, HSTS uses hand-coded domain control knowledge to guide its search. parcPlan[10] is a domain-independent temporal planner using the least-commitment approach. parcPlan claims to be able to handle a rich set of temporal constraints, but the experiments in [10] do not demonstrate its expressiveness adequately.

Resource-IPP (RIPP)[8] is an extension of the IPP planner to deal with durative actions that may consume metric resources. RIPP considers time as another type of resource and solves the temporal planning problem by assuming that actions are still instantaneous. Like IPP, RIPP is based on Graphplan[2] algorithm. A limited empirical evaluation of RIPP is reported in [8]. TP4[6] by Haslumn & Geffner is a recent planner that employs backward chaining state space search with temporal or resource related admissible heuristics. The results of TP4 are promising in a subset of temporal planning problems where durations are measured in unit time, and resources decrease monotonically.

There are several planners in the literature that handle either temporal or resource constraints (but not both). TGP[16] is a temporal planner based on the Graphplan algorithm. TGP extends the notion of mutual exclusion relations in the Graphplan algorithm to allow constraints between actions and propositions. RTPG can be seen as a relaxed version of the planning graph that TGP uses. While TGP might provide better bounds on slacks and times of achievement, it is also costlier to compute. Cost of computation is especially critical as *Sapa* would have to compute the planning graph once for each expanded search node. It is nevertheless worth investigating the overall effectiveness of heuristics derived from TGP's temporal planning graph. LPSAT[19] can handle metric resource constraints by combining SAT and linear programming. As noted in Section 4, LPSAT seems to suffer from poor quality plans.

6 Conclusion and future work

In this paper, we described *Sapa*, a domain-independent forward chaining heuristic temporal planner that can handle metric resource constraints, actions with continuous duration, and deadline goals. *Sapa* does forward search in the space of time-stamped states. Our main focus has been on developing effective heuristics to control the search. We considered both satisficing and optimizing search scenarios and proposed effective

[9] We tried to run Zeno on the same machine used to test *Sapa* without control-knowledge for that problem, but Zeno indicated that it can not solve and returned a partial solution.

heuristics for both. Our heuristics are based on the relaxed temporal planning graph structure. For optimizing search, we introduced admissible heuristics for objective functions based on the makespan and slack values. For satisficing search, we looked at heuristics such as sum-action and sum-duration, that are based on plans derived from RTPG. These were found to be quite efficient in terms of planning time. We also presented a novel technique to improve the heuristic values by reasoning about the metric resource constraints. Finally, we provided an extensive empirical evaluation demonstrating the performance of *Sapa* in several metric temporal planning domains.

In the near term, we intend to investigate the problem of finding better relaxed plans with regard to the resource and temporal constraints of actions in the domain. We are interested in how to use the resource time maps discussed in [8] in constructing the relaxed plan. Moreover, we want to use the binary mutex information, *a la* TGP [16] to improve heuristics in both optimizing and satisficing searches. Our longer term plans include incorporating *Sapa* in a loosely-coupled architecture to integrate planning and scheduling, which will be the logical continuation of our work with the *Realplan* system[18].

References

[1] Bacchus, F. and Ady, M. 2001. Planning with Resources and Concurrency: A Forward Chaining Approach. *Proc IJCAI-2001*.

[2] Blum, A. and Furst, M. 1995. Fast planning throught planning graph analysis. *Proc IJCAI-95*.

[3] Bonet, B., Loerincs, G., and Geffner, H. 1997. A robust and fast action selection mechanism for planning. *Proc AAAI-97*

[4] Fox, M. and Long, D. 2001. PDDL+: An Extension to PDDL for Expressing Temporal Domains

[5] Ghallab, M. and Laruelle, H. 1994. Representation and control in IxTeT, a temporal planner. *Proc AIPS-94*

[6] Haslum, P. and Geffner, H. 2001. Heuristic Planning with Time and Resources *Workshop on Planing with Rersource, IJCAI-01*

[7] Hoffmann, J. 2000. http://www.informatik.uni-freiburg.de/ hoffmann/ff.html

[8] Koehler, J. 1998. Planning under Resource Constraints. *Proc ECAI-98*

[9] Laborie, P. and Ghallab, M. 1995. Planning with sharable resource constraints. *Proc IJCAI-95*.

[10] Liatsos, V. and Richards, B. 1999. Scaleability in planning *Proc ECP-99*.

[11] Muscettola, N. 1994. Integrating planning and scheduling. *Intelligent Scheduling.*

[12] Nguyen, X., Kambhampati, S., and Nigenda, R. 2001. Planning Graph as the Basis for deriving Heuristics for Plan Synthesis by State Space and CSP Search. *To appear in Artificial Intelligence.*

[13] Pearl, J. 1985. Heuristics. *Addison-Wesley*

[14] Penberthy, S. and Well, D. 1994. Temporal Planning with Continuous Change. *Proc AAAI-94*.

[15] Pinedo, M. 1995. Scheduling: Theory, Algorithms, and Systems. *Prentice Hall*

[16] Smith, D. and Weld, D. 1999. Temporal Planning with Mutual Exclusion Reasoning. *Proc IJCAI-99*

[17] Smith, D., Frank J., and Jonsson A. 2000. Bridging the gap between planning and scheduling. *The Knowledge Engineering Review*, Vol. 15:1.

[18] Srivastava, B., Kambhampati, S., and Do, M. 2001. Planning the Project Management Way: Efficient Planning by Effective Integration of Causal and Resource Reasoning in RealPlan. *To appear in Artificial Intelligence.*

[19] Wolfman, S. and Weld, D. 1999. Combining Linear Programming and Satisfiability Solving for Resource Planning. *Proc IJCAI-99*