TimeSplice: Planning with the Affordances of Preemption and Concurrency

Adam Perer and Nikhil Swamy and Taowei David Wang
Department of Computer Science
University of Maryland
College Park, MD 20742
{adamp,nswamy,tw7}@cs.umd.edu

Abstract
Despite preemption’s prominence in the real world, current temporal planners have failed to address its existence. In this paper, we present a new planner, TimeSplice, that supports this ability to plan for actions over non-contiguous intervals of time. By examining TimeSplice’s success in comparison to an existing temporal planner, we show that having an enhanced expressivity for preemption has clear benefits.

Introduction
Planning in the real world often has to take into account the consequences of actions with duration, or effects contingent on the time of their execution. An obvious extension to the classical planning problem is the introduction of temporal constraints. Time was presented as an additional constraint in several of the problems at the AIPS 2002 planning competition.

Actions that execute in a temporal context can be partially ordered - this can lead to mutual exclusion between actions, or under the appropriate conditions an ordering may permit the concurrent execution of actions. HTN planners such as SHOP2 require that actions be added to the plan in some linear sequence and thus do not have an explicit representation for concurrent actions. Other planners such as TimeLine (or SHOP2 with the MTP extension) may be able to extract such concurrency properties due to the timestamps associated with each action.

Concurrency is often associated with the notion of preemption - it is possible to have the agent partially execute some of its ordered subtasks, and then execute some other task, before finishing the first task to completion.

Preemption is an important area for planning to address because it is a widely-used technique of managing resources, from CPU scheduling to job-shop problems. In this paper, we introduce TimeSplice, which we believe is the first temporal planner that encapsulates this notion of preemption.

Starting off with a brief summary of other temporal planners in the field today, we will then explain the concept of preemption and why these planners fail to produce a complete model of it. The full details of how our implementation of a preemptive temporal planner will then be discussed, followed by several experiments exhibiting the strengths of weakness of planning with preemption.

Related work
There are many aspects of time that are still not fully handled by AI planning algorithms (Yaman et al. 2003). For this very reason, there are a variety of planners attempting to handle these different temporal characteristics which haven’t been addressed in the past.

One such planner is TimeLine. TimeLine is an extension of the SHOP HTN planner that is able to reason about temporal actions. Actions, and methods composed of these actions, can be provided with durations and the sequencing of these actions may be subject to temporal constraints. TimeLine possesses the notion of a clock – when generating a plan, the algorithm chooses only those actions (or methods) that may be applicable to the state of the world, at the current time. This choice is determined by the set of temporal constraints that apply to the actions in conjunction with the traditional preconditions on the application of an action. TimeLine considers multiple actions that may be applicable at a particular instant of time and as such is able to handle concurrent actions.

Another approach to temporal planning is to use Multi-timeline preprocessing (MTP). This algorithm transforms temporal-planning domains into domains which have operators that do the temporal reasoning themselves. This technique has most extensively been tested with the HTN planner, SHOP2 (Nau et al. 2003). While SHOP2 with MTP was an order of magnitude faster than TimeLine in a head-to-head comparison, it also created significantly worse plans (Yaman et al. 2003).

TLPlan (Bacchus & Ady 2001) and TalPlanner (Kvarnstrom & Doherty 2001) are also planners that plan for actions in the same order the actions will occur. They differ from SHOP2 and TimeLine by using their temporal formulas to figure out which portions of the search space can be ignored, instead of HTN methods. These planners are more efficient than TimeLine but there are certain types of temporal problems that they fail to solve due to assumptions they make (Yaman et al. 2003).

Approach
While all of the temporal planners listed above can reason about actions that take some duration of time, none directly support the ability for an action to be performed over a set
of non-contiguous intervals of time. When an executing action halts to an idle state only to be resumed at a later point in time, it has exhibited this notion of preemption which we will continue to refer to throughout the remainder of the paper.

We decided to add these enhancements to an existing temporal planner, TimeLine, due to the characteristics of the planner listed above – TimeLine’s notion of a clock would give us explicit control over rescheduling and preemption.

To understand TimeSplice, the basics of TimeLine need to be understood. An interesting enhancement that TimeLine provides over classical HTN planners is the concept of operators with incremental effects. Consider an operator A that has duration \( D(A) > 1 \). If the start time of the operator is \( t(A) \) then it is possible to specify effects that occur at each unit of time in the interval \([t(A), (t(A) + D(A))]\). This is a natural model for the specification of many real world actions.

An obvious extension of this model is to permit the execution of an action over a set of non-contiguous intervals of time: Now consider two operators \( A \) and \( B \) with durations \( D(A) \) and \( D(B) \) respectively. Suppose the execution of the actions corresponding to \( A \) and \( B \) both require some resource \( R1 \). This resource is typically modeled as a precondition for the execution of the actions. If the earliest start time eligible for the execution of \( A \) is \( t(A) \) and the resource \( R1 \) is available at \( t(A) \) then it makes sense to start \( A \) at time \( t(A) \). Now, let \( t(A) = t(A) + \delta < D(A) \) be the earliest possible start time for operator \( B \), and furthermore suppose that \( D(B) < D(A) - \delta \). Under a shortest-job-first heuristic, it might make sense to stop the action corresponding to \( A \), release the resource \( R1 \) and begin the execution of the action corresponding to \( B \). Once \( B \) is completed \( A \) can resume until it completes. In this case, the execution period of the action corresponding to \( A \) is composed of the union of the disjoint time intervals \([t(A), t(A) + \delta] \cup [t(A) + \delta + D(B), t(A) + D(B) + D(A)]\).

TimeSplice is a greedy algorithm in the sense that it decides which tasks have priority over another using a greedy heuristic. If task \( T1 \) is currently executing and using resource \( R \) at \( t0 \), and then \( T2 \) becomes a legal task at a future time point \( t1 \) which also requires resource \( R \), \( T2 \) will be able to preempt \( T1 \) for control of \( R \) if \( D(T2) \) is less than the remaining duration of \( T1 \). Due to the fact that a local heuristic is used, our algorithm will not necessarily lead to globally optimal solutions. In fact, globally optimal scheduling is NP-hard. However, this simple heuristic is a tractable starting point.

### Implementation

Both TimeLine and TimeSplice were written in Java and are platform-independent. Since TimeSplice is an extension of TimeLine, only details pertinent to preemption and modifications/additions to TimeLine will be described in this section. For full details on TimeLine’s implementation, please refer to (Yaman & Nau 2002).

---

**Operator Definitions**

The ability for operators to execute over non-adjacent intervals of time may make sense only under certain conditions. Operators that permit such execution will be termed preemptible operators. It should be clear that such operators can lead to the generation of plans that can execute to completion in less time. TimeLine currently does not provide a means to model the execution of such operators. Consequently, the representation of TimeLine’s operators needed to be enhanced. An example of a TimeSplice operator is shown in Figure 1. Our operators are identical to TimeLine’s operators, except for the section labeled preemptive block. For every operator, there are four characteristics that are used to define its preemptible nature:

1. **stop-preconditions** - These are the preconditions that must be satisfied in order for preemption to begin.
2. **stop-effects** - These are the effects that will happen once the operator enters its preemptive state.
3. **resume-preconditions** - These are the preconditions that must be satisfied in order for the operator to resume and exit its preemptive state.
4. **resume-effects** - These are the effects that will happen once the operator resumes and exits its preemptive state.

At this time, it is worth mentioning that TimeLine’s domains are expressive enough to represent a naive version of preemption. Essentially, this can be done by writing methods to represent every possible combination of points of preemption. The full details of how TimeLine can represent preemption naively is described in the implementation section below. Even beyond the obvious weakness of having an extremely large number of methods, this naive implementation will be shown to be much weaker than TimeSplice.

---

Figure 1: Example of TimeSplice operator

```plaintext
(operator (drive-truck ?truck ?loc-from ?loc-to) 
  (distance ?loc-from ?loc-to ?dist) 
  (speed ?truck ?speed) 
  (:effects 
    (assign ?duration (call ceil (call / ?dist ?speed) ) ) 
    (road ?loc-from ?loc-to ?road) 
    (not (moving ?truck ?some-location)) 
    (truck-at ?truck ?loc-from)) 
  ://preemptive block 
  ;;stop-preconditions 
  ;;stop-effects 
  (\[0\] (not (available ?road))) 
  \[resume-preconditions\] 
  (\[0\] (not (available ?road))) 
  \[resume-effects\] 
  (\[0\] (not (available ?road))) )
```

---

---
Temporal Constraints

Extending TimeLine to provide support for preemptive operators entailed a redesign of the task notification and scheduling model. TimeLine supports temporal constraints on tasks of the following forms:

1. \( (\geq (\text{start task1}) (\text{end task2})) \)
2. \( (\geq (\text{start task3}) (\text{start task2})) \)

These constraints are implemented in TimeLine using the notion of the "notification list". A notification list is a list of tasks that are associated with a given task. Each task maintains two types of notification lists - constraints of the form of (i) result in the addition of task1 to task2’s endNotification list while constraints as in (ii) result in the addition of task3 to task2’s startNotification list. We will refer to task1 as the notifying task and task2 as the notified task. This notification relation between tasks is interpreted as follows: in case (i) once the end time of task2 has been determined, the scheduler is notified of the corresponding earliest start time of task1 (the analogous notification takes place in the case of constraint (ii)).

This concept of notification extends naturally to the case of preemptible tasks. In the case of endNotification constraints, the preemption of the notifying task should result in the rescheduling of the notified task, until such a time as when the preempted notifying task is resumed and its true end time can be determined. In the case of startNotification constraints the appropriate interpretation is not immediate – here if the notifying task is preempted, it is likely that the notified task has already been scheduled and its not immediate what action, if any, should be taken on the notified task. We choose to perform no action on the notifying task and allow it to execute at the time designated by the original start notification.

While this model carries the benefit of simplicity, this policy is clearly inadequate in certain situations. Consider, for instance, the case of two communicating processes each modeled as a preemptible operator. Here one task of the communicating pair is likely to be the recipient of a start notification from the other task. Preemption of either of these processes should result in the preemption of both – i.e. these are lock-step processes. We choose not to model such scenarios.

In addition to the adjustment or rescheduling of previously calculated temporal constraints, TimeSplice introduces two other kinds of constraints. When a task \( T_1 \) is preempted by another task \( T_2 \), an endNotification constraint is induced from \( T_2 \) to \( T_1 \). The rationale behind such a decision is as follows: since \( T_2 \) needs to preempt \( T_1 \) to be scheduled, it must be the case that \( T_1 \) and \( T_2 \) contend for the same resource. Since that resource will not be released until such a time as when \( T_2 \) completes, it makes sense to constrain \( T_1 \) to attempt to resume only once \( T_2 \) has completed. Also, if \( T_2 \) is itself preempted before it completes, the resource under contention is made available at that time – TimeSplice supports an option to allow for a notification to occur under such circumstances too.

Finally, when it is determined that some task should be preempted to make available a resource for the preempting task, a hard constraint is imposed to ensure that the newly available resource be consumed by the preempting task only. That is the next task scheduled must be the preempting task and no other.

When preemption is in use, there are many more constraints that are pertinent to the planner. For instance, if a task \( T_2 \) was able to preempt \( T_1 \) in order to obtain resource \( R \), there would have to be a constraint that \( T_1 \) cannot go back and use \( R \) until \( T_2 \) has finished with it. Such constraints need not be specified in the preemptible methods – TimeSplice will determine them implicitly. In fact, once TimeSplice induces such a constraint, it will do so only for the current subspace of the search tree. These constraints also reduce the branching factor of the search tree because they will dissuade the planner from attempting to schedule a task before it becomes legal. So if TimeSplice backtracks to find a better plan, it is flexible enough to alter its original constraints and induce new ones that make sense for the planner’s current placement of preemption.

Pseudocode

```plaintext
1: procedure apply_operator(OP:operator, S:state, A:agenda)
2:   if \( S \) satisfies preconditions(OP)
3:     add ground_effects(OP) to A
4:   return success
5: else for each action AC that is currently in A
6:     if \( ((S + \text{stop_effects}(AC)) \text{ satisfies } \text{preconditions}(OP)) \) AND
7:         \( (S + \text{stop_effects}(AC)) \text{ satisfies } \text{preconditions}(OP) \)
8:       if \( \text{ground}(OP) \text{ ends before AC} \)
9:       remove effects(AC) from A
10:   add end_notification from OP to AC
11:   add ground_effects(OP) to A
12: return success
13: return failure
```

Domains

Since domains built for TimeLine did not incorporate the ability for preemption, we of course had to build our own domains from scratch. We built two, the Trucks domain and the TA domain, and will be referring to them throughout the paper.

Truck Domain

The simpler of the two, the Truck domain involves a set of trucks of various speeds and starting points, a set of roads of varying lengths, and a set of cities. For the purposes of instantiating preemption, we made it so that one truck can only be on one road at a time. The goal in most of our experiments was to see how fast the planner can get the trucks to their designated cities. Due to the fact that the trucks would need to share the roads to get to their destination, the act of preemption could be invoked.

TA Domain

Our somewhat more complicated domain was the TA domain, modeled after the horrors graduate teaching assistants (TAs) tend to face when there are plenty of students requiring their services. Featuring a set of TAs and students of varying intelligences and a set problems of varying difficulty, the goal is to help all of the students finish their problems in the shortest amount of time. A student can work with a TA until their problem is finished. However, if a TA is forced to switch his attention towards another student, the preempted student will be able to work on its own since guidance was received, albeit at a slower rate. Once the TA
becomes available again, the student can resume working alongside the TA.

**Experiments**

The following experiments were performed on a Pentium 4 computer with 2GHz of processing power and 256 megabytes of RAM. The operating system used to compute the results was Linux (kernel 2.4.20-ac1).

**Naive preemption with TimeLine**

TimeLine specification allows for multiple methods of the same name as long as all operators have different signatures. The idea of using this operator overloading for preemption is to simulate preemption/resumption pairs by specifically calling operators that contain three periods of activity:

1. `begin-operator-period` - Applies operator, claims resources needed.
2. `suspend-operator-period` - Releases resources needed, suspends all TimedEffects.
3. `continue-operator-period` - Reacquires resources needed, continues suspended effects.

The idea is that during the suspend stage, any claimed resource is released, allowing for another task to claim that resource and begin itself. The problem is that once a suspended operator wants to continue, it does not check to see if all its preconditions are satisfied since suspension is only a simulated event. The operator has been executing the whole time. This creates a concurrency problem where two processes enter the same critical section. To solve this problem, the naive domain needs locks and careful calculation of times when resources are released and claimed. In addition, a separate operator is written to handle the instances where no preemption is allowed to take place when this operator is in effect. That is the reason why there are two operators, as shown in Figure 5.

To express preemption in TimeLine’s domains, there will be a requirement of an exorbitant number of methods. Suppose we have a very simple planning problem with three tasks ($T_1$, $T_2$, $T_3$), each with running times of $D(T_1) = 4$, $D(T_2) = 16$ and $D(T_3) = 8$. Each of these running times can be broken up into two parts in $D(T_n)$ ways. For example, $T_1$ could execute by running for one timestep, then halting, then finishing its remaining three timesteps. All $D(T_1)$ combinations for $T_1$ are $(1, 3)$, $(2, 2)$, $(3, 1)$, $(4)$.

Furthermore, there will also need to be methods representing all possible combinations of breaking an operator into three parts, namely the begin-stage-duration, the suspend-stage-duration and the continue-stage-duration. We will represent these three parts by the symbols $A$, $B$ and $C$ respectively. If $|A|$ is the number of choices to choose $A$ from, then the number of all possible combinations for all three durations is $|A + C| \cdot |B|$. Although there is an infinite length for the suspend-stage-duration, to make it expressible in TimeLine’s domain, we made the reasonable assumption that this duration will never be greater than the duration of the longest operator. Thus, in our simple planning problem, the number of choices for $A$, $B$, and $C$ will all be $D(t_2)$.

Even with this restriction, the number of methods we would have so far is:

$$D(t_1) \cdot |B| \cdot D(t_2) \cdot |B| \cdot D(t_3) \cdot |B| = 4 \cdot 16 \cdot 8 \cdot 16 \cdot 16 = 2097152$$

Having over two million methods was too many for TimeLine to handle in a reasonable amount of time. So, to further reduce the number of methods we made use of our prior knowledge of the optimal solution (e.g. in the best solution, $T_2$ should have a suspend-stage-period of length 8). There were also methods among the two million that were homogeneous and these duplicates were pruned. However, there were still 262,144 methods. To further reduce the number of methods, we decided to make $T_1$ and $T_3$ non-preemptible tasks and this gave us a reasonable number of 112 methods. Several of these methods can be seen in Figure 6.

**Enhanced Expressivity for Preemption**

One of the main goals to be accomplished by TimeSplice was to create a temporal planner that is proficient at handling domains where preemption makes sense. While we mentioned that it is indeed possible to express a naive version of preemption in TimeLine, there are some serious limitations. Even with the restrictions on the naive domain for TimeLine listed above in the Implementation section, it couldn’t compare with the speed of TimeSplice on average. The results can be seen in Table 1.

The `Nodes Examined` rubric tells us how many “high-level” operations that TimeSplice and TimeLine performed. Each of those operations essentially decomposes into a bunch of Theorem Prover (TP) calls. The number of these decompositions for TimeSplice per operation is typically greater. This can be thought of as the cost of a “pruning operation” which reduces the number of nodes in the search space – so the total expenditure for TimeSplice must take into account these extra TP calls (e.g. of the 22 TP calls by TimeSplice, 10 were due to preemption). As Table 1 tells us, both TimeLine and TimeSplice found optimal plans (both had a total plan length of 24, which is the shortest amount of time all of the tasks can be completed with preemption, and longer without), however TimeLine took much longer.

An example of an optimal plan generated by TimeLine with naive preemption is shown in Figure 2 and one by TimeSplice is shown in Figure 3.

Due to the fact that TimeLine checks overloaded methods sequentially, it is possible that if you put all of the methods that lead to a solution at the top of the domain file, it will indeed compute more efficiently than TimeSplice. This is illustrated in Figure 2, which shows that even in TimeLine’s best case, TimeSplice is still not much worse. However, it is quite unreasonable to expect to have knowledge about which methods are best prior to running a planner. Thus, we decided to run a head-to-head comparison of the Naive TimeLine versus TimeSplice. To remove any bias from where the methods where placed in the naive domain files, we averaged the results from ten different trials with random placement of the methods.

There are also more things that TimeSplice makes it easy to express, including:
Table 1: Comparison of Naive TimeLine to TimeSplice

<table>
<thead>
<tr>
<th></th>
<th>TimeSplice</th>
<th>TimeLine</th>
<th>Best TimeLine</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of TP Calls</td>
<td>22</td>
<td>5142.6</td>
<td>9</td>
</tr>
<tr>
<td>Best Plan Length</td>
<td>24</td>
<td>24</td>
<td>24</td>
</tr>
<tr>
<td>CPU Time (s)</td>
<td>0.77</td>
<td>14.296</td>
<td>0.70</td>
</tr>
</tbody>
</table>

Figure 2: A plan found by TimeLine with naive preemption for the Trucks domain. The whole plan is inside a pair of parentheses. Each line represents a fully instantiated operator and a duration in which the operator is in effect. The operators have 1 or 3 values passed in. The last value is the ID of the truck that is using the road. The two first values, if they exist, designate the start point of the suspend period and the start point of the continue period. These time points are relative to the begin time of their respective operator. The entire durations are bounded by brackets.

```plaintext
{ (!drive-truck truck1 city1 city2 1.0 ) [0 , 4]  
 (!drive-truck truck2 city2 city3 7.0 15.0 2.0 ) [0 , 24]  
 (!drive-truck truck1 city5 city3 1.0 ) [7 , 15]  
 (!unload-truck truck1 ) [12 , 17]  
 (!park-truck truck1 ) [20 , 21]  
 }
```

Figure 3: A plan found by TimeSplice for the Trucks domain. The format is the same as any plan found by TimeLine with the exception of the appearance of the durations. Each operator now has a list of durations. Each element on the list denotes a period during which the operator is in effect. The time periods not described in the duration denote the periods the operator is preempted, hence not in effect. For example, the second line denotes that this operator is in effect from [0, 4], preempted from [4, 12] and again in effect from [12, 24]. At time 24, the operator ends its execution.

```plaintext
{ (!drive-truck truck1 city1 city2 ) [0 , 4]  
 (!drive-truck truck2 city2 city3 ) [[0, 4],[12 , 24]]  
 (!drive-truck truck1 city2 city3 ) [4 , 12]  
 (!unload-truck truck1 ) [12 , 17]  
 (!park-truck truck1 ) [17 , 18]  
 }
```

**Multiple-Preemption** Another advantage that TimeSplice has over TimeLine with naive preemption is the fact that TimeSplice can support multiple preemption. Suppose there is a situation in which a task $T_1$ is interrupted by task $T_2$, and $T_2$ completes and allows $T_1$ to resume. Then a third task, $T_3$ interrupts $T_1$ once more, this is an instance of double preemption. There is no bound on the number of multiple preemptions – and the planner will examine if these will lead to a better plan. Naive TimeLine could simulate multiple preemption by simply adding more methods to describe those scenarios, but not only would that be a combinatorial blowup, but it also would be limited to the finite number of of preemptions that was supported by the domain.

An example of a plan that makes use a multiple preemption is shown in Figure 10. This plan took place within the TA domain, with two students, one TA, and five problems. Its plan length is noticeably shorter than a run without using any preemption, as shown in Figure 11.

**Discard Remaining Effects** TimeSplice supports a feature which permits an operator to have a completely different set of effects after it has been preempted. This allows for a number of useful scenarios to be modeled – for instance, whatever progress an action has made towards the completion of a task may be degraded over the idle-period of that action and that work may have to be redone once the action resumes. Equivalently, a task may make progress towards completion even though the action was preempted; in the case of our TA domain, a student makes slow progress towards solving a problem independently and this reduces the amount of work that a TA must do to help the student reach a solution.

**Notify on Preempt** A preempting task can notify the task it preempted if it is preempted itself. If an operator is flagged as notifyOnPreempt, at line 10 in the pseudocode in the Implementation section above, procedure apply-operator, a preempt-notification is added from OP to AC. The effect of this is that AC receives a notification that it can try to resume if OP is either preempted or completes its execution. This information can be useful if you wish to model simple background processes. With the intention of not modifying much of our Trucks domain, we were interested in having an operator that would be active whenever a certain truck wasn’t on the road. As in the simple trucks domain, the resource under contention in the modified domain is the road. A new operator is introduced, !background-free-road-use, that permits a truck to use the road without actually consuming the resource. In particular, when this action is being executed for some truck, then that truck makes slow progress towards its destination while still allowing another truck to consume the road resource. This feature is demonstrated in Figure 4.

**Conclusions**

Temporal domains are often used to specify problems with hard time constraints or for problems that contain critical tasks. For instance, in the TA domain, one could specify
that the student Bob must catch a bus at a particular time in addition to solving the homework problems. The hard time constraint makes the task of catching the bus the critical task in any schedule for such a problem. The shortest job first heuristic used by TimeSplice has an effect on whether or not such a deadline will be met. Consider a problem instance in which the critical task \( T_1 \) is required to follow some other task \( T_2 \). If \( T_1 \) requires a long time to complete, TimeSplice will plan for non-preemption first for the long task \( T_2 \), then preempt it by a shorter task \( T_3 \) hence prolonging the execution time of \( T_2 \). Later TimeSplice realizes that such plan will not satisfy the hard time constraint imposed by \( T_1 \), hence the planner will backtrack, causing extra overhead that a non-preemptive planner avoids by not being aware of the notion of preemption. On the other hand, if \( T_1 \) was a short task, then \( T_2 \) will be scheduled favorably by TimeSplice and the hard constraint will be met in the first try. In the average case, the performance of TimeSplice will be no worse than that of a non-preemptive scheduler.

However, there are also situations where preemption can actually lead to longer plans. When this happens, it generally is due to non-unique resources. For instance, in our TA domain, suppose at time \( t_n \) that all of the TAs are in use. One student, Bob, is working with the teaching assistant William. Then, at time \( t_{n+1} \), student Don comes along and suddenly gets William’s attention because his problem can be finished before Bob’s can (this is our shortest-jobs heuristic in action). Bob will be forced into a state of preemption and will resume only when Don is finished with William. This is due to our endNotification constraint. But suppose another TA, Xavier, becomes available while Bob is halted. Xavier is now idle and could be helping Bob, except Bob is frozen and waiting for William. Due to the fact that a TA’s time was being unused could lead to a worse solution. This was demonstrated in one of the domains we created, where run under nonpreemption mode (Figure 8) created a better plan than under preemption mode (Figure 9).

As exhibited through our experiments, TimeSplice’s biggest strength is its enhanced expressivity for preemption. TimeSplice was able to create plans with preemptible operators with much more efficiency than TimeLine. TimeSplice is also a sound and complete planner. This stems from the fact that TimeLine was both sound and complete and none of our enhancements affected those characteristics.

Due to its prominent existence in a realistic setting, preemption is a useful concept to model in temporal planning.

Acknowledgements

We would like to thank Fusun Yaman for being kind enough to share her extensive knowledge of temporal planning, as well as the source code to her own planner, TimeLine. We’d also like to thank Ugur Kuter and Dr. Dana Nau for inspiration and guidance throughout TimeSplice’s development.

References


from a local shortest-job-first to a branch-and-bound heuristic might help guide our planner in a more efficient manner. The addition of a load-balancer would help remedy the non-unique resources problem addressed earlier. And finally, TimeSplice could be enhanced with priority-based scheduling, which is a natural extension to preemption for use in the real world.
Appendix
(operator (!drive-truck ?truck ?loc-from ?loc-to ?thisTruckID)
  (distance ?loc-from ?loc-to ?dist)
  (speed ?truck ?speed)
  (= (foobar ?inc-var) ?fred-val)
  (call >= ?fred-val 0)
  (assign ?duration (call ceil (call / ?dist ?speed)))
  (assign ?newDuration (call + ?duration (call - ?restartTime ?stopTime)))
  (road ?loc-from ?loc-to ?road)
  (available ?road)
  (call < ?stopTime ?restartTime)
  (call > ?newDuration ?restartTime)
  (call > ?stopTime 1)
  (not (moving ?truck ?some-location))
  (truck-at ?truck ?loc-from))
)
;;//effects (all ref to time are relative to start of this op)
{
  (1) [ (moving ?truck ?loc-to)
        (not (truck-at ?truck ?loc-from))
        (not (available ?road))]
  (1, (call = ?stopTime 1) )
  { [+ (foobar ?inc-var) ?dist]
    (not (available ?road))
    (= (userID ?road) ?thisTruckID)
  }
  [?stopTime] ;;stop and release the road
          (available ?road)
}

([?restartTime, (call = ?newDuration 1) ])
  { [+ (foobar ?inc-var) ?dist]
    (= (userID ?road) ?thisTruckID) }

([?newDuration] ((available ?road)
         (truck-at ?truck ?loc-to)
         (not (moving ?truck ?loc-to)))
  )
)

(operator (!drive-truck ?truck ?loc-from ?loc-to ?thisTruckID)
  (distance ?loc-from ?loc-to ?dist)
  (speed ?truck ?speed)
  (= (foobar ?inc-var) ?fred-val)
  (call >= ?fred-val 0)
  (assign ?duration (call ceil (call / ?dist ?speed)))
  (road ?loc-from ?loc-to ?road)
  (available ?road)
  (not (moving ?truck ?some-location))
  (truck-at ?truck ?loc-from))
)
;;//effects (all ref to time are relative to start of this op)
{
  (1) [ (moving ?truck ?loc-to)
        (not (truck-at ?truck ?loc-from))
        ]
  (1, (call = ?duration 1) )
  { [+ (foobar ?inc-var) ?dist]
    (= (userID ?road) ?thisTruckID) }

  [?duration] ((available ?road)
         (truck-at ?truck ?loc-to)
         (not (moving ?truck ?loc-to)))
  )
)

Figure 5: Operators for naive preemption in TimeLine
Figure 6: A few of the 112 methods for supporting naive preemption in TimeLine
(defdomain ta-room

( operator (!help-student ?student ?problem)
  (not (helping ?someTA ?student ?someproblem))
  (ta ?ta)
  (available ?ta) ;;ta can't be helping someone else
  (ta-factor ?ta ?factor)
  (= (prob-progress ?student ?problem) ?progress)
  (prob-difficulty ?problem ?diff)
  (intelligence ?student ?iq)
  (assign ?inc-rate (call * ?factor ?iq))
  (assign ?maxhelptime (call ceil (call / (call - ?diff ?progress)
               ?inc-rate)))
  (assign ?absoluteFinishTime (call + ?maxhelptime (call currentTime))))
  (not (available ?ta))
  (helping ?ta ?student ?problem))
  ([0] ((= (time ?student ?problem) (call currentTime))
    (not (available ?ta))
    (helping ?ta ?student ?problem)))
  ([1,?maxhelptime ] ((+= (time ?student ?problem) 1)
    (+= (prob-progress ?student ?problem) ?inc-rate)))
    (= (last-help-time ?student) 0.0)
    (available ?ta)
    (not (helping ?ta ?student ?problem))))
)

(discardRemainingEffects
  (assign ?currentTime (call currentTime)) ;;no stop precondition
  ([0] ((not (helping ?ta ?student ?problem))
        (available ?ta))));immediate stop effect
  ([1] (set (last-problem-worked ?student) ?problem)
        (last-help-time ?student) 0.0)
        (available ?ta)
  ((ta ?anyta)
    (available ?anyta) ;;resume precondition
    (not (helping ?someTA2 ?student ?someproblem2)) ;;basic soundness conditions
    (= (prob-progress ?student ?problem) ?curProgress)
    (val last-problem-worked ?student ?anyProblem)
    (= (last-help-time ?student) ?stopTimeTemp)
    (assign ?stopTime (call cond ?stopTimeTemp ?stopTimeTemp ?resumeTime))
    (assign ?remainingTime (call + ?resumeTime ?remainingTime))
    (assign ?progressInc (call - ?resumeTime ?stopTime)))
)

((ta ?anyta)
  (available ?anyta) ;;resume precondition
  (not (helping ?someTA2 ?student ?someproblem2)) ;;basic soundness conditions
  (= (prob-progress ?student ?problem) ?curProgress)
  (val last-problem-worked ?student ?anyProblem)
  (= (last-help-time ?student) ?stopTimeTemp)
  (assign ?stopTime (call cond ?stopTimeTemp ?stopTimeTemp ?resumeTime))
  (assign ?remainingTime (call + ?resumeTime ?remainingTime))
  (assign ?progressInc (call - ?resumeTime ?stopTime)))
)

(:method (process-student ?student ?prob-list)
  (student ?student)
  (call equal ?prob-list 0))
  ()
  )

;case : student arrives ... start back-ground study process
(method (process-student ?student ?prob-list)
  (student ?student)
  (call notequal ?prob-list 0)
  (call (car ?prob-list))
  (assign ?prob (call cdr ?prob-list))
  (assign ?rest (call cdr ?prob-list))
  (problem ?prob)
  (= (last-help-time ?student) 0.0))
  (t1 (!help-student ?student ?prob)
    t2 (process-student ?student ?rest))
  ()
  (>= (start t2) (+ (start t1) 1)))))

(method (process-student ?student ?prob-list)
  (val last-problem-worked ?student ?prob)
  (student ?student)
  (call notequal ?prob-list 0)
  (assign ?prob (call cdr ?prob-list))
  (assign ?rest (call cdr ?prob-list))
  (problem ?prob)
  (= (last-help-time ?student) 0.0))
  (t1 (!help-student ?student ?prob)
    t2 (process-student ?student ?rest))
  ()
  (>= (start t2) (+ (start t1) 1)))))

;state axioms
[[- (range ?prob-progress 0 10000) ()]
  [- (range ?last-help-time 0 10000) ()]
  [- (range ?time 0 10000)])

Figure 7: The definition of the TA domain.
Figure 8: A demonstration of how non-unique resources leads to worse times in preemptive mode. Here, using no preemption, a plan is found with length 113. The best plan with preemption took length 189.

Figure 9: A demonstration of how non-unique resources leads to worse times in preemptive mode. Here, using preemption, a plan is found with length 189. Without preemption, a plan could have been found with length 113.

Figure 10: A plan generated by TimeSplice that exhibits multiple preemption.

Figure 11: A plan for the same problem as Figure 10, but instead using TimeLine which doesn’t support multiple preemption.