Drudgery Relieving Commands for Mixed Initiative Planning

Anthony Barrett, Thomas Starbird

Jet Propulsion Laboratory
California Institute of Technology
4800 Oak Grove Drive, M/S 126-347
{Anthony.Barrett, Thomas.Starbird}@jpl.nasa.gov

Abstract

Not only has the number of JPL spacecraft launched per year exploded over the past few years, but also the spacecrafts' behaviors are getting more complex as flyby missions give way to remote orbiters, which in turn give way to rovers and other *in situ* explorers. With this complexity there comes an increased need for rapid response to execution uncertainties arising from dynamic partially-understood environments. Thus operations staffs are forced to ever more rapidly generate and uplink short sequences. For instance the Mars Exploration Rovers' staffs had to analyze downlinked data and generate a new set of commands every Martian day during the mission. To speed up the activity planning process while keeping operators fully in control, this paper describes a set of simple useful commands for manually building plans without leaning on the machinery of an automated planner.

Introduction

While past flight projects had the luxury of weeks to generate a spacecraft sequence, present missions like the Mars Exploration Rovers (MER) and future ones like the Mars Science Laboratory (MSL) and Dawn need to build sequences for an uplink within hours. While MER's use of the MAPGEN planning environment [Bresina *et al*. 2005] improved the feasibility of rapidly generating a plan, there were occasions when it hindered operators from making straight forward changes. MAPGEN helped an operator maintain the validity of a plan's constraints, but it also hindered an operator from invalidating constraints at any time. This has forced operators to perform unintuitive plan manipulations to make simple changes.

For example, it was common for the operator (the Tactical Activity Planner in MER's prime mission) to wish to place a particular set of activities one after another, as a starting point for a plan, while maintaining the ability to change order later. Making such a chain of activities often required introducing one activity at a time into its approximate desired location in the timeline, and then shifting it to a start time equaling the end time of the previous activity. In other cases, features of the MAPGEN environment could be used to create a chain of activities, by placing them in the correct order manually and then dragging the right-most one, causing the group to align themselves one immediately after the other like a row of boxcars. This method would work if the activities were sorts that MAPGEN knew could not overlap. But that knowledge could sometimes get in the way. Many activities could not overlap communications activities, so pushing an accumulating chain from after to before a comm activity was not possible. One trick that was sometimes employed was to delete the comm activity temporarily to allow such movement. Such a deletion is highly unintuitive, since the comm activities were generally the only ones that were known ahead of time and fixed in time, and since it would be a major mistake to forget to re-insert the comm activity. As another example, to swap the positions in the timeline of two activities, it was often necessary for the operator to delete one activity from the plan, move the other activity to its new position, and then insert the deleted activity. These sorts of unintuitive and somewhat roundabout steps ate up valuable time in the tight operations schedule for producing a plan, and added a bit to the specialized training needed by a user.

This paper takes a more human centered approach toward mixed initiative planning, where operators manipulate a plan instead of guiding a planner, letting them make any plan manipulation including those that make a plan inconsistent. Instead of enforcing consistency at all times, this approach allows constraint breaking and helps an operator manipulate a plan to improve its consistency. As such, this paper presents three contributions. First it shows how to add a constrained-move functionality to any plan manipulation interface without including a general purpose planner. Second it shows how a constrained move can be used to manipulate plans even when they have inconsistent constraints. Finally, it defines a number of drudgery relieving commands identified during MER operations and shows how to combine them with constraint enforcement.

In the next section we introduce planning constraints and describe how they are defined in terms of temporal constraints. The next two sections show how constrained move maintains a set of consistent constraints and how to support satisfying violated constraints even when the full set is inconsistent. Given this improvement to constrained move, the subsequent sections define a set of drudgery relieving plan manipulation commands, place this work in the context of related work on mixed initiative planning, and finally conclude.

Planning Constraints

In general, plans are represented as sets of temporally constrained activities, where each activity may or may not have an assigned start time depending on whether or not there is a desire to maintain temporal flexibility. Typically manual ground operations planners work with fixed start times to simplify reasoning about a plan. This facilitates drawing activities as boxes on a screen, which an operator drags left/right to change their start times. For the purpose of this paper, we will stick to fixed time planning as it is typically used in graphical based manual plan editors. As such, each activity in a plan has the eight following fields in addition to activity types and parameters.

- start – the start time
- duration – how long the activity lasts
- temporals – explicitly required temporal constraints
- MUTEXs – other activities that are mutually exclusive
- pinned – boolean for freezing at current start time
- constraints – valid constraints with transitive closure
- earliest – inferred earliest possible start time
- latest – inferred latest possible end time

While an activity's start, duration, pinning, earliest, and latest fields contain integers, its MUTEXs field is a possibly empty set of pointers to other plan activities that are not allowed to overlap with it. Finally, temporals and constraints contain sets of constraints with the form

$$(A, [\text{lowest}, \text{highest}], B),$$

where A is the current activity and activity B is temporally constrained such that

$$A.start + lowest \le B.start \le A.start + highest.$$

While most of the fields are manually set and altered, the last three are computed. Essentially, the MUTEXs, temporals, and pinning combine to set the initial set of constraints, and an algorithm for solving the all-pairs shortest path problem computes the transitive closure over a simple temporal network [Dechter 2003]. The Floyd-Warshall algorithm computes such a closure in $O(n^3)$ time.

For instance, Figure 1 illustrates four activities with three temporal constraints that make them occur serially with the second occurring within a time bound after the first, and the three extra constraints are from the temporal closure. Actually each illustrated constraint refers to two constraints. In the case of the rightmost constraint in the diagram, the two constraints are

$$(C,[C.duration,\infty], D) \in C.temporals \text{ and}$$

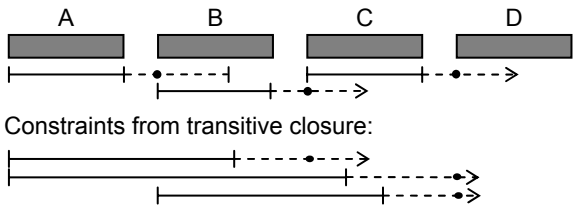

Figure 1. Example set of four temporally constrained activities and transitive constraint closure

$$(D, [-\infty,-C.duration], C) \in D.temporals.$$

More precisely, each B pointed to in A.MUTEXs determines a constraint

$$(A, [-\infty,-B.duration], B) \text{ or } (A, [A.duration,\infty], B)$$

A succeeds or precedes B respectively. Since pinning explicitly freezes an activity's start time, an activity A's pinned flag's truth would turn into constraint

$$(e, [A.start,A.start], A),$$

where "e" refers to an initial epoch at time "0" such that activity A starts A.start time units after "e". Finally A.temporals are added to A.constraints without modification, and the result is used to compute the transitive closure.

Active Constraint Enforcement

The arguably most used feature of MAPGEN was its active constraint enforcement. This feature lets an operator change a selected activity and has other activities change to enforce the constraints. For instance, an operator moves activities as shown in Figure 2. At the start (t1) the operator selects the middle two activities and subsequently drags them to the right. During the drag the third activity meets the fourth (t2) and active constraint enforcement then starts moving the fourth activity also. If the operator keeps dragging the selected activities, enforcing the required proximity between the first two activities will start moving the first activity too (t3).

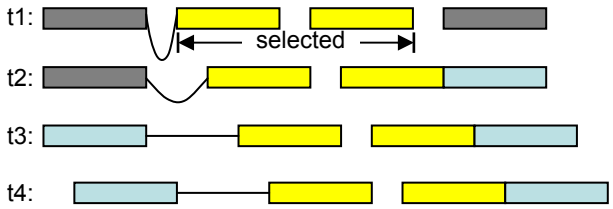

Figure 2. Four instants in a constrained move for a selected set of activities

At first glance, active temporal constraint enforcement looks rather complex in that there is a network of activities and constraints, and constraint enforcement requires determining which activities to move to keep current constraints satisfied. In actuality, the process is fairly simple after computing the transitive closure of temporal constraints. Given this closure, Algorithm 1 implements moves with active constraint enforcement in $O(sn)$ time,

```
For each A ∈ SELECTED {
  A.start ← A.start + Δ ;
  For each (A,[Lowest,Highest],B) ∈ A.constraints do
    If B ∉ SELECTED then
      B.start ← max(A.start + Lowest,
                    min(A.start + Highest, B.start )) }.
```

Algorithm 1. Moving SELECTED activities by Δ (positive or negative) while maintaining constraints

where *s* and *n* denote the number of selected activities and the total number of activities.

The main points to observe is that the transitive closure short circuits the need to traverse the graph of constraints and activities to determine which activities to move. This traverse is captured in the closure. Thus iterating through A.constraints finds all activities that need to be moved, and the new values are computed in the max-min equation. Essentially the max() moved activities to the right to maintain the lower bounds, and the min() moves them to the left to maintain the upper bounds.

Dragging and Nudging

At its most basic, an operator alters an activity's start time earlier or later by either dragging it or nudging it to the left or right respectively. In the case of nudging, pressing a key will move the activity by a preset step size. Dragging moves the activity by some arbitrary delta depending on how far the activity is dragged. While these features are endemic to any graphical plan editing interface, combining this with active constraint enforcement results in limiting how far an activity might be dragged. For instance, suppose that the leftmost activity in Figure 2 was pinned to its current location. In this case the illustrated drag would stop at time t3 due to the constraints keeping the activities from moving.

Algorithm 2 shows how simple it is to correctly constrain dragging and nudging in O(*s*) time after computing the transitive closure. One of the side effects of computing the transitive closure with respect to an initial epoch "e" is knowledge of the earliest and latest times when an activity can start. With this information clipping delta to restrict a drag or a nudge is a matter of reducing its absolute value to keep all of the selected activities from moving out of their earliest-latest start time bounds. All of the other activities are kept in their start time bounds by virtue of the transitive closure.

```
For all A ∈ SELECTED {
  LIMIT ← max(A.earliest,
              min(A.latest, A.start + Δ)) – A.start;
  if Δ < LIMIT ≤ 0 or 0 ≤ LIMIT < Δ then Δ ← LIMIT }.
```

Algorithm 2. Limiting Δ based on activity constraints

Jumps

While dragging and nudging suffice for moving activities around, they are also relatively imprecise in that it is easy to nudge one too many times or drag an activity too far or not far enough on an activity display. For this reason, we introduce other moves to qualitatively interesting points. The first such move is to "jump" a set of selected activities to the left or right until any loose constraint becomes taut, which will increase the number of moving activities on the next jump. For instance, the first three lines in figure 2 correspond to jumps.

Jumping selected activities forward in time is a matter of computing the desired delta with Algorithm 3, limiting delta with Algorithm 2, and then performing the move with Algorithm 1. Given these algorithms, a jump can be performed in O(*sn*) time, and the steps for computing left jumps have the same structure and time complexity.

```
Let Δ ← ∞;
For all A ∈ SELECTED do {
  Δ ← min(Δ, A.latest – A.start);
  For each (A,[Low,High],B) in A.constraints do
    If (B ∉ SELECTED & A.start + Low < B.start) then
      Δ ← min(B.start – (A.start + Low), Δ) }.
```

Algorithm 3. Computing Δ for jumping activities right (later) to the point where the next stationary activity starts moving

Hops

While jumping is useful, it is quite conceivable that jumping will move a set of activities too far. In this event, another qualitative movement called a "hop" moves a set of activities until any two temporally ordered time points become simultaneous. With this move any two unrelated activities can be moved back to back or to start/end at the same time.

Algorithm 4 shows how to compute the deltas Δ_L and Δ_R for left and right hops respectively. It starts by computing the deltas for left and right jumps and then reduces the magnitudes of these deltas by analyzing the time points from earliest to latest to find the two closest points that would be made simultaneous by moving left/right. The time complexity of this algorithm is O(*sn*) due to computing the jump deltas and O(*n*log(*n*)) for reducing the magnitudes, due to a need to order the time points.

```
Let  ΔL ← –leftJump(), ΔL ← rightJump(),
     POINTS be the points that start/stop activities,
     L be the tuple ⟨t ← 0, stay ← false, move ← false⟩,
     R be the tuple ⟨t ← 0, stay ← false, move ← false⟩;
For each P ∈ POINTS ordered from first to last do {
  If P.time ≠ R.time then {
    L ← R; R.stay ← false; R.move ← false; R.t ← P.t; }
  If P starts/stops an activity that will move
    then R.move ← true
    else R.stay ← true;
  If L.move and R.stay then ΔR ← min(ΔR, R.t – L.t);
  If R.move and L.stay then ΔL ← min(ΔL, L.t – R.t); }
Return ΔR or –ΔL depending on right or left hop.
```

Algorithm 4. Computing Δ for hopping left or right to a point where two time points will change ordering.

Building Plans

So far the focus here has been on activity movement, but the previously described MER experience also points to simple drudgery relieving commands when adding activities, but they are manual. While most approaches toward planning focuses on representing activities and how

to automatically add and schedule them to achieve goals, this approach becomes problematic when the precise resource needs, preconditions, and effects of activities are not known at plan time. This was often the case during early operations – a time when the activities were often specialized to squeeze as much performance as possible from a rover. Typically activity definitions evolve over time to have numerous parameters that combine in complex ways to determine preconditions, effects, and resource needs.

While the ASE experience [Chien *et al*. 2005] on the EO1 satellite showed that modeling these activities for automated planning is possible, ASE was not developed until after EO1 had been in orbit for over a year. During early operations the drive is to build plans manually and simulate them in order to determine correctness as the planned activities get progressively better defined. Still, even without complete activity definitions, coarse properties like timing and mutual exclusion relationships between activities can be defined and used early in the planning process.

Adding Activities and Constraints

There are many ways to manually add activities to a plan, but they all derive from a copy/paste metaphor where the activities are copied from an activity dictionary or from elsewhere in the plan. The main point where they differ revolves around how to paste them into a plan. The simplest way is to require explicit manual placement of the pasted activity, but that can become a bit arduous as described in the introduction. Algorithm 5 shows how to perform a simple insertion of activities from left to right in constant time, and the algorithm for adding from right to left would be identical except for the computation of B.start.

```
When SELECTED has only one activity A
   Add new activity B with B.start ← A.start + A.duration
   SELECTED ← {B}.
```

Algorithm 5. Chain-anchor-first to add activities from left to right.

Adding constraints is a matter of selecting activities and choosing how to constrain them by adding temporal constraints to each activity's temporals set. During early MER operations, two useful drudgery relieving commands related to adding constraints were determined: freeze-temporal-relations to freeze the relative start times of a selected set of activities and freeze-ordering-relations to freeze the ordering of start and endpoints of a selected set of activities. In the case of a freeze-temporal-relations command, all that is needed is to add constraints between any one selected activity and all of the others to fix the relative start times of the activities. Freezing the ordering requires a little more effort to constrain the order of each pair of selected activities. In both cases the complexity of these commands is $O(n^3)$ due to their requiring a transitive closure computation. Related unfreeze commands were also determined to be useful and are simply a matter of removing the constraints added by a freeze and then updating the transitive closure. Thus these commands also exhibit an $O(n^3)$ time complexity.

Supporting Inconsistent Constraint Sets

While freezing is guaranteed to add constraints that are consistent with the current plan, it is possible to add constraints that are not consistent. An activity can be inserted on top of another to invalidate a MUTEX relation or a temporal constraint might be added that cannot be satisfied while enforcing all of the other temporals. While we can specify an inconsistent constraint, adding it to the plan invalidates the transitive closure, and determining which constraints are at fault, to find a maximally sized consistent subset, is a nontrivial problem.

By far the easiest way to deal with this problem is to only enforce constraints that currently hold. In this event Algorithm 6 is used to determine which constraints to actively enforce depending on whether or not the user wishes to enforce temporal constraints and/or MUTEX constraints. Notice that in each case a constraint is added to CONSTRAIN only if it already holds. Thus an operator can select which valid constraints to enforce and then update the enforced constraints as subsequent activity movement makes them hold, and an enforcement update will never unexpectedly move an activity.

```
Let CONSTRAIN ← { };
For each activity A do {
   If enforcing temporals then
      For each (A,[Low,High],B) ∈ A.temporals do
         If Low ≤ B.start – A.start ≤ High
            then Add (A,[Low,High],B) to CONSTRAIN;
   If enforcing MUTEXs then
      For each B ∈ A.MUTEXs do
         If A starts before B and does not overlap
            then Add (A,[A.duration,∞],B) to CONSTRAIN;
   If A.pinned = true
      then Add (e,[A.start,A.start],A) to CONSTRAIN
      else Add (e,[0,∞],A) to CONSTRAIN }
Update each activity's constraints, earliest, and latest
   values with results of calling Floyd-Warshall algorithm
   to compute transitive closure of CONSTRAIN.
```

Algorithm 6. Updating enforced constraints.

In general, this approach to constraint enforcement lets the operator choose which constraints to actively enforce and then facilitates manipulating a plan to monotonically reduce the number of unsatisfied constraints. In all cases the operator is given the power to break a constraint at any time and then the tools to remove the drudgery from repairing the breaks.

Computing Transitive Closures

As previously mentioned, the simplest way to compute the transitive closure of a set of temporal constraints involves

an adaptation of the Floyd-Warshall algorithm [Dechter 2003]. For the case of our work, this adaptation takes the form of Algorithm 7, where the $O(n^3)$ time complexity comes from the third nested FOR loop. The first two loops initialize an n by n distance matrix where n is the number of activities. They work by first assuming less than infinite distances between activity start times and then reducing these distance limits with each temporal constraint. The third nested loop is the heart of the Floyd-Warshall algorithm, which computes the transitive closure of the distance limits to solve the all-pairs-shortest-path problem. Given this solution, the fourth loop sets each activity's earliest and latest possible start times, and the fifth loop sets the relative constraints among activities.

```
For i ← 1 to n do
   For j ← 1 to n do
      If i = j then dij ← 0 else dij ← ∞;
For each (Ai, [low,high], Aj) ∈ CONSTRAIN do {
   dij ← min(high, dij); dji ← min(–low, dji) };
For k ← 1 to n do
   For i ← 1 to n do
      For j ← 1 to n do dij ← min(dij, dik + dkj);
For i ← 2 to n do {
   Ai.constraints ← { }; Ai.earliest ← –di1; Ai.latest ← d1i };
For i ← 2 to n do
   For j ← 2 to n do
      If |dij| ≠ ∞ or |dji| ≠ ∞
         then Add (Ai, [–dji,dij], Aj) to Ai.constraints.
```

Algorithm 7. Computing transitive closure of CONSTRAIN's constraints, where A_1 is "e"

For example, Figure 3 denotes the constraints alluded to back in Figures 1 and 2, where the activity durations are 5 time units long and the start time of activity B is less than 10 time units after the start time of activity A. All activities start after a set epoch time "e", resulting in the [0,∞] arcs. Finally, the [5,∞] arcs come from the requirement that activities C and D are consecutive after B.

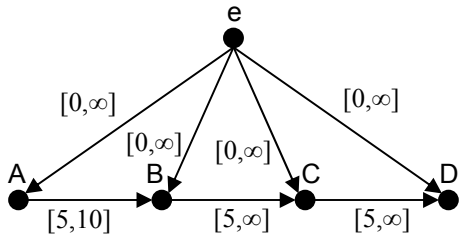

Figure 3. Example set of constraints among activity start times

Given our example, the first two loops compute the $d_{row,column}$ distance matrix in Figure 4. Explaining this computation comes from the observation that a constraint $(A_i, [low,high], A_j)$ denotes the two equations:

$$A_i.start - A_j.start \le high \text{ and}$$
$$A_j.start - A_i.start \le -low.$$

Thus this matrix has zero elements on the diagonal since the distance for an activity to itself is zero, and the elements above the diagonal denote limits from the highs in constraints and the elements below the diagonals denote negatives of limits from the lows.

	e	A	B	C	D
e	0	∞	∞	∞	∞
A	0	0	10	∞	∞
B	0	-5	0	∞	∞
C	0	∞	-5	0	∞
D	0	∞	∞	-5	0

Figure 4. Matrix representation $d_{row,column}$ of constraints among activity start times.

While the above matrix encodes all of the information from CONSTRAIN, computing the transitive closure with Algorithm 7's third loop to get Figure 5 makes this information easier to work with. For instance, the original constraint was for activity C to occur after the epoch time, but computing transitive closure shows us that C cannot occur in less than ten time units after epoch. As the fourth loop shows, each activity's earliest and latest start times can be quickly acquired from the matrix's first column and row respectively, and the fifth loop computes the constraints in Figure 6.

	e	A	B	C	D
e	0	∞	∞	∞	∞
A	0	0	10	∞	∞
B	-5	-5	0	∞	∞
C	-10	-10	-5	0	∞
D	-15	-15	-10	-5	0

Figure 5. Matrix representation of constraints among activity start times after computing transitive closure

Actually, Figures 3 and 6 are not precisely correct due to the fact that they only show half of the constraints. Essentially, for each $(A_i, [low,high], A_j)$ constraint shown in the figures there is another $(A_j, [-high,-low], A_i)$ constraint in the graph. These constraints add no information to the transitive closure computations, but they are computed by Algorithm 7 and required by the algorithms that implement constrained moves and other drudgery relieving commands.

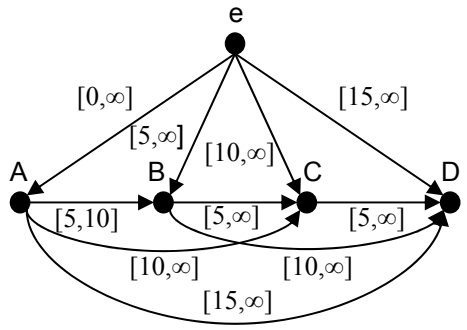

Figure 6. Resultant transitive closure of constraints

Group Manipulations

With our approach toward breaking and repairing constraints, we can now define a number of higher level drudgery relieving commands for manipulating groups of activities. In order to keep each command's definition as simple as possible, it is quite feasible for each of these commands to fail if disallowed by the currently enforced constraints, but then succeed upon relaxing constraint enforcement.

Activity Set Manipulations

The simplest of the group manipulations just iterates over a set of selected activities and moves them into some specific pattern. Essentially three such patterns were determined to be useful during MER operations: synchronize activity start times, riffle activities to push them as far as possible together while maintaining their original ordering, and synchronize activity end times. Pictorially, examples of these three manipulations appear in Figure 7. Notice how riffle is very similar to synchronize start times, with the exception that some set step size is added to the start times to maintain activity ordering.

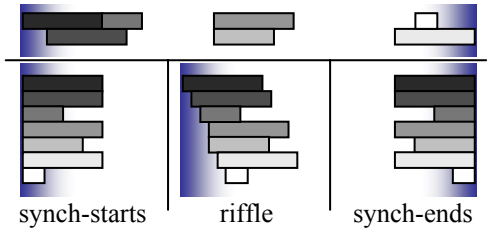

Figure 7. How three simple manipulations move selected activities

Algorithm 8 shows how simple any of these commands can be. It just finds the time to synchronize to and performs a series of moves with constraint enforcement. In this case the command can fail if one of the selected activities cannot be moved to the start time due to an enforced constraint, at which point an operator can just relax constraint enforcement if needed.

```
Let S ← A.start of the earliest A ∈ SELECTED;
If (S < B.earliest for some B ∈ SELECTED or
    two selected activities cannot start simultaneously)
  then return failure
  else for each B ∈ SELECTED do
    move {B} by S – B.start.
```

Algorithm 8. Synchronizing the start times of SELECTED activities without breaking constraints.

There are possibilities for different versions of this command that break as few constraints as possible instead of failing. Such versions might be more useful, but it would be harder to anticipate the effects of such alternative versions due to their unexpectedly breaking constraints. A simpler version of this command would turn off all constraints, move, and then turn them back on. Although there is a computational problem with that approach in that turning constraints on requires an $O(n^3)$ transitive closure computation. As it stands, this algorithm takes $O(sn)$ time. In general, there is a tradeoff between a version's power, time complexity, and understandability. The primary desire here is to keep commands as simple and fast as possible. In any case, this command can be forced to succeed by turning constraint enforcement off.

Cluster Set Manipulations

Quite often a selected set of activities can be separated into overlapping subsets as shown at the top of Figure 8. By identifying the gaps we can determine overlapping clusters of activities, and these overlaps often denote a desired absolute temporal relationship between interacting activities. Cluster set manipulations move these activities around while maintaining the absolute temporal relationships of overlapping activities.

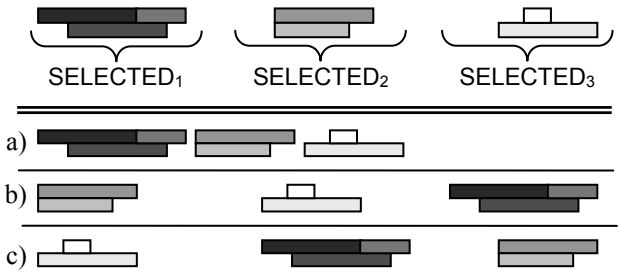

Figure 8. Three clusters of selected activities with three cluster manipulations: (a) splay by gap size, (b) cycle left, and (c) cycle right

Given this goal to maintain overlaps, cluster manipulations start with Algorithm 9 to compute the clusters in $O(s\log(s))$ time, where the temporal complexity comes from sorting the $\langle T, A\rangle$ elements. This algorithm performs this sort in order to sweep over the activities from left to right, identifying the subsets that do not overlap. The $\mathsf{SELECTED}_i$ subsets are progressively ordered as a side effect of the sweep direction.

```
Let  i ← 1, j ← 0,
     POINTS ← {⟨T, A⟩ | A in SELECTED and
                        T ∈ {A.start, A.start+A.duration}},
Sort POINTS by increasing T with T = A.start points last
  for a given T;
For each ⟨T, A⟩ ∈ POINTS in sorted order do {
  If T = A.start
     then { put A into SELECTED_i ; j ← j + 1; }
     else j ← j – 1;
  If j = 0 then i ← i + 1; }.
```

Algorithm 9. Identifying the overlapping clusters of SELECTED activities.

Once the clusters are determined, a manipulation command becomes fairly trivial. For instance, Algorithms 10 and 11 perform the splay and cycle-left manipulations as they are illustrated in (a) and (b) of Figure 8. In both cases the computations start with identifying clusters, then they test to determine if the activity movements are possible given the currently enforced constraints, and finally they perform the constrained moves. In both algorithms StartTime() and EndTime() compute the times when a cluster's first activity starts and last activity ends respectively in $O(s_i)$ time for each $SELECTED_i$. In total, each algorithm takes $O(sn)$ time.

```
Identify the M clusters of SELECTED;
If (any move below would be constrained by A.earliest
   or A.latest for any A ∈ SELECTED)
   then return failure;
For i ← 2 to M do
   move SELECTED_i by EndTime(SELECTED_i–1) –
      StartTime(SELECTED_i) + gapSize.
```

Algorithm 10. Splaying a set of clusters by gapSize by shifting left or right depending on gapSize.

```
Identify the M clusters of SELECTED
Let S ←  StartTime(SELECTED_1) –
         StartTime(SELECTED_2)
If (any move below would be constrained by A.earliest
   or A.latest for any A ∈ SELECTED)
   then return failure;
move SELECTED_1 by
   EndTime(SELECTED_M) – EndTime(SELECTED_1)
For i ← 2 to M do move SELECTED_i by S.
```

Algorithm 11. Cycling clusters of selected activities to the left, moving the first cluster to the end.

Just like the activity set manipulations, cluster set manipulations have varying effects depending on the enforced constraints. Each can fail if it will violate an actively enforced constraint, and each is guaranteed to only move activities in the currently selected set if no constraints are being enforced. If a manipulation command can succeed with the currently enforced constraints, then it may move more than the selected set of activities just as a constrained move would have.

Future Manipulations

So far the described activity movement manipulations have taken $O(sn)$ time, which is fast enough to appear instantaneous when manipulating a day's worth of activities through a GUI. There are three other manipulations identified during early MER operations that we have yet to identify the appropriate algorithm for due to their complexity and interaction with constrained move:

- Spread-obey-temporal – from left to right spread activities later to resolve constraints without changing ordering;
- Repair-if-possible – same as spread-obey-temporal, but can change ordering of selected activities; and
- Repair-or-discard – same as repair-if-possible, but can remove activities.

The first two commands' implementations look like fairly simple constraint additions followed by a transitive closure computation, but their computational complexity rapidly explodes to NP complete problems when considering which ordering constraint to add to enforce each MUTEX relation. The last manipulation is even ambiguously defined. The question is, "Which activities should be discarded?" Obviously discarding all selected activities and their constraints resolves all broken constraints, but that is not likely to be the operator's intent.

By virtue of their ambiguity and time complexity, these commands may not be ultimately useful. Still, they were identified and their intuitive definitions provide a place for the more powerful automated planners to infiltrate into a plan manipulation tool.

Related work

As this paper's title suggests, this work is most related to other mixed initiative planning systems. Examples of these systems include MAPGEN [Bresina *et al.* 2005] to assisted operators in planning a MER rover's day on Mars, VAL [Howey et al. 2004] with experiments on a Beagle 2 style Mars lander domain, and PASSAT [Myers *et al.* 2003] to plan missions in a special operations forces domain. Given the number of hard to compare mixed initiative planning systems, Cortellessa and Cesta (2006) developed an experimental approach toward evaluating a mixed initiative system, which focused on two issues: how to make the underlying problem solver better serve users and how to empower users to enhance there active involvement in the planning process.

Our work challenges an assumption made by most other related systems and exposed by this evaluation, an assumption that we start with an automated planner when building an mixed-initiative system. Instead of starting with automated planners and focusing on how to make them work with users, this work starts with a set of

deterministic drudgery relieving commands that users would like to see, and shows O(sn) algorithms that automate them. There is no search or search heuristics underlying this planning automation, and a given command will always have the same effect.

In terms of capability, MAPGEN is the most closely related system to this work, but our approach to active constraint enforcement makes no assumption that all constraints are valid and consistent. Our approach only only enforces a subset of the valid constraints, and this subset can be either reduced or enlarged at any time with an O(n^3) algorithm, where this algorithm can only enlarge the set to include all currently valid constraints.

Conclusions

The experience of constructing daily plans for the MER rovers represents a valuable case study for the role of different levels of aid that software can give the human planner. Much of the operator's time on MER was spent on low-level manipulations of the plan's elements. The algorithms offered in this paper strike a balance, having software automatically make changes to the plan, yet leaving the operator in complete control. Each algorithm performs an action that is easily understood by the operator, an attribute that is vital for remaining in control, and each relieves the operator of some low-level drudgery.

The constrained move functionality was very much used and liked by MER operators. The algorithms in this paper give a variety of such functions without the need for a general purpose planner in the software. In addition, the algorithms do not assume that at every stage the entire plan is consistent. This feature gives the operator the freedom to concentrate on one portion of a plan and get use from the algorithms even if another part of the plan is broken.

One other class of drudgery removing operations not addressed here involves those that add support activities. For instance, the MER operators found the automatic addition and deletion of CPU turn off commands. While one can easily point to this functionality as a motivation for a planner, it can also be implemented using a relatively simple algorithm that simulates a plan and applies a few simple expert system rules.

Acknowledgements

This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The authors would also like to thank James Kurien and Michael McCurdy for discussions contributing to this effort.

References

Bresina, J., Jónsson, A., Morris, P., and Rajan, K. 2005. "Mixed-Initiative Planning in MAPGEN: Capabilities and Shortcomings." In *Proceedings of the ICAPS Workshop on Mixed-Initiative Planning and Scheduling*. Monterey, CA.

Chien, S., Sherwood, R., Tran, D., Cichy, B., Rabideau, G., Castano, R., Davies, A., Mandle, D., Frye, S., Trout, B., D'Agostino, J., Shulman, S., Boyer, D., Hayden, S., Sweet, A., Christa, S. 2005. "Lessons Learned from Autonomous Sciencecraft Experiment." In *Proceedings of Autonomous Agents and Multi-Agent Systems Conference*. Utrecht, Netherlands.

Cortellessa, G. and Cesta, A. 2006. "Evaluating Mixed-Initiative Systems: An Experimental Approach**"**. In *Proceedings of the 16th International Conference on Automated Planning & Scheduling (ICAPS-06)*. Cumbria, UK.

Dechter, R. 2003. *Constraint Processing*. San Francisco, CA.: Morgan Kaufmann.

Howey, R., Long, D., and Fox, M.2004. "VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning Using PDDL." In *Proceedings of 16th IEEE International Conference on Tools with Artificial Intelligence*, Boca Raton, FL.

Myers, K., Jarvis, P., Tyson, W., and Wolverton, M. 2003. "Mixed-initiative Planning in PASSAT." In *Proceedings of the 13th International Conference on Automated Planning and Scheduling – Demonstration Track*. Trento, Italy.