Flexible Execution of Plans with Choice

Patrick R. Conrad and **Julie A. Shah** and **Brian C. Williams**
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory, MERS
32 Vassar St. Room 32-D224, Cambridge, MA 02139
prconrad@mit.edu, julie_a_shah@csail.mit.edu, williams@mit.edu

Abstract

Dynamic plan execution strategies allow an autonomous agent to respond to uncertainties while improving robustness and reducing the need for an overly conservative plan. Executives have improved this robustness by expanding the types of choices made dynamically, such as selecting alternate methods. However, in methods to date, these additional choices introduce substantial run-time latency. This paper presents a novel system called Drake that makes steps towards executing an expanded set of choices dynamically without significant latency.

Drake frames a plan as a Disjunctive Temporal Problem and executes it with a fast dynamic scheduling algorithm. Prior work demonstrated an efficient technique for dynamic execution of one special type of DTPs by using an off-line compilation step to find the possible consistent choices and compactly record the differences between them. Drake extends this work to handle a more general set of choices by recording the minimal differences between the solutions which are required at run-time. On randomly generated structured plans with choice, we show a reduction in the size of the solution set of over two orders of magnitude, compared to prior art.

As autonomous systems become more capable and common, they will need to reason about complex tasks and robustly execute plans in uncertain environments. In previous work, Williams et al. introduced the Reactive Model-Based Programming Language (RMPL), which is designed to allow engineers to simply and intuitively express the desired behavior of the system (2003). Then the agent's *executive* determines the correct sequence of actions to accomplish this behavior, relieving the programmer of explicitly coding that logic. RMPL programs often involve temporal constraints which the executives must reason over.

Kim, Williams, and Abramson previously developed Temporal Plan Networks (TPNs) as a temporal constraint language for a subset of these problems (2001). TPNs specify simple interval constraints, providing an upper and lower bound on the durations between plan events. However, they also include choice points, where the executive may choose between different possible threads of execution, which increases the agent's flexibility in responding to disturbances. These choices require disjunctive constraints, making the temporal constraints of TPNs a special case of the Disjunctive Temporal Problem (DTP) (Dechter, Meiri, and Pearl 1991).

Although some instances of temporal networks can be statically scheduled in advance, this strategy typically leads to brittle plans since the executive cannot adjust to disturbances. To account for this inflexibility, pre-scheduled plans are often overly conservative. One effective technique for handling uncertainty is to follow a strategy of least commitment and to delay decision making until execution time, thereby maintaining maximal flexibility to respond to uncertainty (Muscettola, Morris, and Tsamardinos 1998; Tsamardinos, Muscettola, and Morris 1998). Previous work has developed efficient execution strategies for Simple Temporal Problems (STPs), a non-disjunctive temporal network, by breaking the executive into a *dispatcher* and a *compiler* (Dechter, Meiri, and Pearl 1991; Muscettola, Morris, and Tsamardinos 1998; Tsamardinos, Muscettola, and Morris 1998). The off-line compiler transforms the network into *dispatchable form*, exposing all the implicit constraints in the original plan. The *dispatcher* uses the dispatchable form to quickly make dynamic scheduling decisions.

However, developing flexible executives for plans with choices, has been more difficult. Kim, Williams, and Abramson present an executive called Kirk, which uses a deliberative planning step to change the execution sequence on-line (2001). Although their results show improvement over prior planning systems, the latency is still too high for tightly coupled systems, for example robots working with humans or walking robots with fast dynamics. Recently, Shah and Williams extended the *compiler* and *dispatcher* model to Temporal Constraint Satisfaction Problems (TCSPs), a type of temporal problems with choice, by compactly recording the possible set of solutions and efficiently reasoning over the possible options (2008). Their executive improved upon the response time of previous systems by reducing the storage and propagation of redundant information.

Shah and Williams' dispatcher represented the full set of plans by creating a relaxed plan and storing the differences in the plans based on the choices the executive can make.

This has an intriguing analogy to previous work in Assumption Based Truth Maintenance Systems (ATMSs) (De Kleer 1986). We generalize their algorithm to work with DTPs by leveraging more ideas from prior ATMS work to record the minimal differences between the plans required by the dispatcher. We begin by reviewing temporal networks and techniques for efficiently dispatching them online. Then we present our algorithm for to the compiler and the dispatcher. Finally, we present empirical results for Drake, which demonstrates significant reduction in the encoding of the dispatchable form of DTPs. These encouraging results suggest that Drake will support fast flexible execution of plans with choice.

Background

Simple Temporal Problems

Temporal networks are often used to represent temporal constraints in planning and scheduling systems. A simple temporal problem is defined as a collection of real-valued time point variables V corresponding to instantaneous events (Dechter, Meiri, and Pearl 1991). There is also a collection of simple interval constraints C of the form

$$l_{XY} \leq Y - X \leq u_{XY}, \tag{1}$$

where l and u are the lower and upper bound, respectively, of the time elapsed between the execution of the two events. There can only be one constraint of this form per pair of time points X and Y. A *solution* to an STP is a set of assignments to the time points that respect all the constraints and a *consistent* STP is one with at least one solution. Testing for consistency is typically done by reformulating the problem as a directed, weighted graph $\langle E, V \rangle$ where each time point is represented with an edge, and each constraint of the form

$$Y - X \leq b_{XY} \tag{2}$$

is represented by a weighted edge from X to Y with weight b_{XY}. An STP is consistent iff its associated distance graph has no negative cycles, which can be checked by computing the All-Pairs Shortest Path (APSP) graph with the Floyd-Warshall algorithm in $O(n^3)$ time (Dechter, Meiri, and Pearl 1991).

The APSP form of the distance graph is also *dispatchable,* because a dispatcher can make scheduling decisions on-line while only requiring local propagation to guarantee a solution (Dechter, Meiri, and Pearl 1991). Since all the implicit constraints given by the set C are explicitly enumerated by the APSP graph, the dispatcher can make assignments to the variables without search. During dispatch the executive tracks *execution windows* for each event specifying the constraints on its execution times given the local propagation of previous execution times through the distance graph. At each step of dispatch, the executive finds a time point whose predecessors have all been executed and whose execution window includes the current time. Muscettola, Morris, and Tsamardinos show that the APSP contains redundant information; an edge is said to be *dominated* by another edge, meaning that the dominated edge always propagates a looser constraint and therefore is not needed by the dispatcher (1998). These redundant edges may be trimmed after consistency is determined, resulting in a *minimal dispatchable network.*

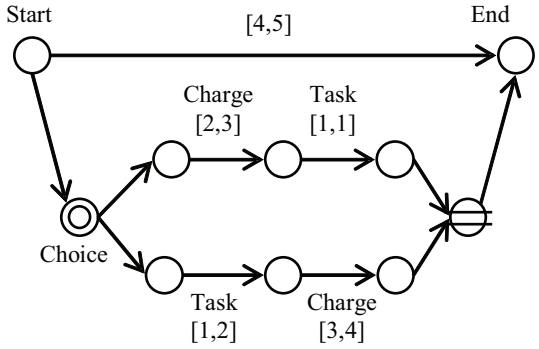

Figure 1: A TPN of the rover example. Unlabeled arcs represent $[0, 0]$ constraints.

Temporal Plan Networks

Temporal Plan Networks are a representation of contingent temporal plans introduced by Kim, Williams, and Abramson (2001). TPNs are STPs with the addition of *symbolic constraints* and *choice nodes*. The primitive element in a TPN is an activity, comprised of two time points connected by a simple interval constraint. Networks are then created by hierarchically composing sub-networks or activities in series, in parallel, or by providing a choice between them. This hierarchical nature places structural restrictions on the network, since parallel threads cannot be arbitrarily constrained. The symbolic constraints do provide one way to constrain activities from parallel threads, but we will not discuss them in detail here.

To illustrate TPNs as plans with choice, we introduce a simple motivating example, shown in Figure 1. Consider a rover with a partially charged battery and a task to complete. The ordering changes the durations of these activities because the battery will take longer to charge if it completes the task first. The entire sequence must be completed within four or five hours so that the rover is ready to receive its next command. The two activity orderings are represented by creating a choice node between the two serial sequences. The total duration is enforced by placing the $[4, 5]$ constraint between the start and end node of the TPN. The choice in the plan indicates that the rover can decide between the activity orderings at run-time. Although this flexibility is not obviously useful in our toy example, choices can be important in larger plans. This example suggests two tools a programmer can take advantage of when creating high level specifications for a system: introducing choice between possible execution orderings of activities and the ability to select between different simple interval constraints. Drake is designed to take a TPN as an input and to flexibly schedule the events on-line consistently with the input plan specifications.

Disjunctive Temporal Problems

Disjunctive temporal problems provide a richer language than TPNs because they lack the hierarchical structural requirements. DTPs are defined similarly to STPs, but each constraint $C_i \in C$ is allowed to be disjunctive, in the form

$$c_{i1} \vee c_{i2} \vee ... \vee c_{in}, \quad (3)$$

where n may be any number (Dechter, Meiri, and Pearl 1991). As before, a solution is a set of assignments to each time point in V while meeting at least one STP constraint per DTP constraint in C. The disjunctive constraints make DTPs an expressive formulation, allowing encoding of problems with choices and resources, to name a few important capabilities. Most modern approaches for determining consistency for DTPs search for a consistent component STP, which is the STP induced by selecting a single STP constraint from each disjunctive one (Stergiou and Koubarakis 2000; Oddi and Cesta 2000; Tsamardinos and Pollack 2003). Any solution to a component STP is also a solution to the DTP.

Tsamardinos, Pollack, and Ganchev presented a flexible dispatcher for DTPs which first enumerates all consistent component STPs and then uses them in parallel for decision making (2001). At run-time, the dispatcher propagates timing information in all the STPs simultaneously. The dispatcher may make scheduling decisions that violate timing constraints in some of the component STPs as long as it never removes all the remaining possible plans.

Assumption Based Truth Maintenance Systems

Assumption Based Truth Maintenance Systems (ATMSs) were introduced by De Kleer as a model based reasoning system specifically designed to allow the user to quickly make queries based on different sets of assumptions (1986). Previous systems would categorize facts as either "true" or "false" by reasoning over the evidence and logical relationships provided. However, new evidence might require sweeping changes to the model's understanding of the world, especially if a key piece of evidence was invalidated. Additionally, it was previously costly to query the system with hypothetical questions, e.g. "if A is true, would X be true or false?" de Kleer's innovation was to replace the *labels* of "true" and "false" with the minimum set of assumptions necessary for the fact to hold, making it a simple matter to change the context of a query.

The ATMS architecture uses *environments* and *labels* to represent the support for a fact. Consider a simple universe with three assumptions, A, B, and C, and two facts X and Y, presented to an ATMS. An environment is a minimal set of assumptions required for a fact to hold; for example, if X is true under any context where A is true, regardless of the other assumptions, then the set $\{A\}$ in an environment under which X holds. A *clause* is an arbitrary set of assumptions and is said to be *subsumed* by an environment if all the assumptions in the environment are present in the clause, regardless of the presence or absence of any other assumptions. Therefore, the clause $\{A, C\}$ is subsumed by the environment $\{A\}$.

Since there may be more than one set of assumptions that make a fact hold, each fact is given a label, the minimal set of environments under which it holds. So, if X holds if either A or B is true, it is given the label $\{\{A\}, \{B\}\}$. A fact holds under any clause subsumed by its label, so determining whether a fact is true under a set of assumptions only requires checking the label, which can be done quickly. The labels are kept minimal to make subsumption checking as quick as possible, meaning that no environments in a label are allowed to subsume each other. Imagine that Y can be given the label $\{\{A\}, \{B\}, \{A, B\}\}$. The environment $\{A, B\}$ is redundant because $\{A\}$ and B together subsume every clause $\{A, B\}$ does, so it should be discarded from the label of Y. These minimal labels are built up at runtime as necessary and are the key to the ATMS's efficiency. Contradictory sets of assignments are kept in a database of no-goods for easy identification, since a label cannot specify that X holds *unless* C does.

When implementing a minimal label system, care is required to ensure that performing label operations will not dominate the computation time (De Kleer 1986; Forbus and De Kleer 1993). These works use specialized encodings of the labels, including bit vector representations and tree data structures optimized for performing subsumption tests. To avoid obfuscating the core algorithms, we will not discuss these ideas further here and will simply treat environments and labels as sets.

The ATMS concept is related to disjunctive temporal networks because the choices among the disjuncts can be treated as assumptions. Since different temporal constraints hold under different choices, the ATMS framework provides a way to calculate the minimal dependence of the derived constraints on the possible choices. Drake will use an ATMS inspired labeling system to compactly record the dispatchable form of the DTP.

Fast Dynamic Dispatching of TCSPs

Shah and Williams approached the problem of dispatching Temporal Constraint Satisfaction Problems, a special case of DTPs, by removing redundant storage and calculations performed by Tsamardinos, Pollack, and Ganchev's algorithm (2008; 2001). Temporal Constraint Satisfaction Problems (TCSPs) are DTPs with restricted structure, where for one constraint C_i, every simple interval constraint per DTP constraint involves the same two variables, so the problems can only represent choosing different execution bounds for a given pair of events, but cannot include exchanging a constraint on one pair for a constraint on a different pair of time points. We will discuss the details of solving TCSPs further because the insights from this algorithm easily extends to DTPs.

Shah identifies that often the component STPs differ by only a few constraints, so that the space required to store the dispatchable representation can be reduced by only keeping the difference between them. The changes are kept as constraint tightenings on a relaxed plan that are calculated by Dynamic Back-Propagation Rules introduced by Shah et al., which only calculate the changes required to maintain dispatchability (2007). This technique is in the spirit of an

ATMS, although the labels are not minimal. The approach of explicitly listing the component STPs also requires copying the events for each STP, which is unnecessary. By removing these redundancies, Shah et al.'s results show dramatic reductions in the size of the dispatchable TCSP (2008). Their dispatching algorithm uses the compact encoding to reduce the execution latencies by several orders of magnitude for medium sized problems.

Although Shah's technique exploits the underlying structure of the TCSP, it does not identify every possible shared constraint since the labels are not minimal. Since DTPs have less common structure than TCSPs, a DTP algorithm would benefit from identifying the shared elements more completely. Therefore, we present Drake, which generalizes Shah's work to DTPs by calculating minimal labels indicating the set of choices under which a particular constraint is active. We will first explain Drake's compilation technique, then describe the dispatcher, and finally evaluate Drake's performance on random problems.

Compilation

Drake's compilation produces a compact dispatchable form of the DTP by identifying common elements between the possible solutions. This process is designed to directly reflect the standard STP compilation algorithm except that *labels* are calculated to distinguish the choices between different constraints. The process is comprised of three steps: first the DTP is converted to a labeled distance graph. Second, the DTP is compiled using a variant of an All-Pairs-Shortest-Path (APSP) algorithm which reasons over these labels. Third, redundant edges are trimmed. We begin our discussion of the compiler by introducing the labeling system and then discuss modifications to the standard STP compilation technique.

Converting TPNs to Labeled Distance Graphs

In preparation for Drake's compilation and dispatch algorithms, input TPNs or DTPs are first converted into distance graphs augmented with ATMS style labels. To abstract the choices represented in the labels from the simple intervals in the disjunctive constraints, we introduce finite domain state variables. Each disjunctive constraint C_i is represented by a state variable with one element in the domain correlated to each of the disjunctive clauses the executive may choose to enforce. The rover example requires a single variable to represent the choice between the two execution paths, A, which has two values in its domain, $\{1, 2\}$, where "1" represents charging first and "2" represents performing the task first. Non-disjunctive constraints, such as the $[4, 5]$, are not given variables to avoid unnecessary book-keeping. A complete set of state assignments therefore represents a selection from each of the disjunctive temporal constraints, which specify a component STP. Choosing $A = 1$ in the rover example leads to a STP consisting of the sub-network where the rover charges first. A partial set of assignments therefore correlates to a family of component STPs that include all the other possible assignments to other variables. To quickly reason over the possible consistent choices at runtime, Drake maintains a list of the consistent full assignments to the variables, S. A solution to a DTP is an assignment to each of the time points that satisfies at least one simple interval constraint per disjunctive constraint. In the state variable formulation a single assignment to each variable correlates to enforcing exactly one interval constraint per disjunctive constraint. Although it is sometimes possible to enforce more than one STP constraint per DTP constraint in a solution, which correlates to allowing more than one assignment to each state variable, doing so creates a more difficult problem. Therefore, we never consider multiple concurrent assignments to a state variable.

To store the set of possible solutions as compactly as possible, it is necessary to note that a particular constraint holds in an entire family of STPs. In the most obvious case, if a component constraint c_{ij} is paired with assignment $x_i = j$, then c_{ij} holds in all STPs where $x_i = j$, so it is only necessary to record it once. The compiler and dispatcher use *labels* with these assignments to identify what set of choices imply that a particular constraint is active, allowing the constraints for different component STPs to be stored in a common repository. Now, a constraint holds under a set of choices if its label is consumed by the clause formed from the assignments associated with those choices. Although Shah's compact encoding of the solution uses a similar idea, the primary contribution of our work is that the labels are computed to be minimal, allowing an especially compact representation of the compiled DTP.

To illustrate this process, consider the transformation of the rover problem from a TPN into a DTP, drawn as a labeled distance graph in Figure 2. After creating the state variable A to represent the choice of activity orderings, we must convert the temporal constraints. Since a DTP need not have the hierarchical structure of a TPN, we can collapse the four nodes on the left connected by $[0, 0]$ arcs (shown without labels in Figure 1) into a single start node W and the four nodes on the right into a single end node Z. Non-disjunctive constraints are added to the graph as usual, i.e. upper bounds become weighted edges and lower bounds are negated and placed on edges in the opposite direction, except they are also given universal labels. For example, the plan completion deadline of four to five hours becomes the two arcs between W and Z, which are given universal labels (not shown) because those constraints always hold. Each possible simple interval from the disjunctive constraints are similarly translated into the distance graph and each is labeled with a single environment specifying its correlated assignment. Hence, the arcs along $\overline{WXZ}$ are all given labels $A = 1$ and the arcs along $\overline{WYZ}$ are all given labels $A = 2$.

The edges E_{jk} between any pair of vertices j and k are maintained as an ordered set, so that the edge which holds under environment e is the one with the lowest weight, which is the tightest constraint, whose label subsumes e. This ordering keeps the labels from growing overly complicated when a new edge that has a lower weight does not totally subsume the label of the old edge. For example, an edge with weight 1 and label $A = 1, B = 1$ takes precedence over an edge with weight 5 of label $A = 1$ without specifically altering the later one. Additionally, any edge

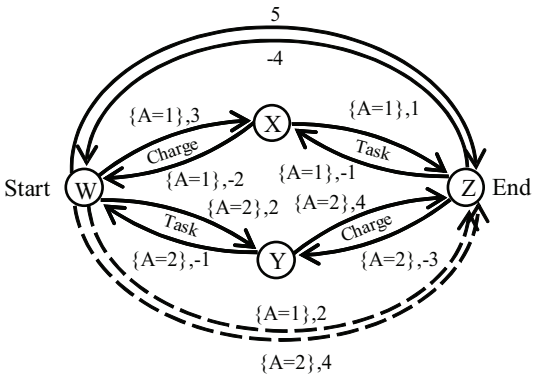

Figure 2: The rover example in distance graph form with state variables. The constraints are shown after the label. The dashed edges are examples calculated by LABELED-APSP.

whose label is subsumed by the union of the labels of edges with lower weights is never actually identified as the edge corresponding to any clause and may be discarded. In this small example, any edge with label $A = 1$ and a weight higher than 5 would always be replaced by the tighter existing constraint and is unnecessary. This conversion to labeled distance graph form compactly represents the constraints of the plan so that Drake's compiler can transform it into a compact dispatchable form.

All-Pairs-Shortest-Path with Labels

The labeling technique allows all the constraints from the families of STPs to be combined into a single labeled distance graph. The compilation process needs to (1) expose the implicit constraints in the DTP to place it in a dispatchable form and (2) record the dependence of each derived constraint on the choices available to the dispatcher. Additionally, the compilation needs to identify and remove any inconsistent sets of choices under consideration by the dispatcher. The labeled distance graph can be be compiled and tested for consistency with a simple modification of a standard STP compilation algorithm. Here, we use a variant of the Floyd-Warshall All-Pairs Shortest-Paths (APSP) algorithm called LABELED-APSP since Floyd-Warshall is the simplest available STP compilation algorithm. Note that this variant does not have polynomial run-time because there may be multiple edges between any given pair of nodes which the algorithm will have to search over.

The modified compilation algorithm LABELED-APSP, shown in Algorithm 1 keeps the essential framework of the Floyd-Warshall algorithm, but introduces management of the labels into the update step in the inner loop in lines 5-8. Its input is a weighted directed graph with vertices V, edges E, and full assignments S as above. Each edge is defined by start and end vertices, a weight, and a label. As in the Floyd-Warshall algorithm, Labeled-APSP updates the shortest paths by looking for a route $j \rightarrow i \rightarrow k$ that has lower weight than the existing $j \rightarrow k$. GENERATECAN-

Algorithm 1 Labeled APSP Algorithm

1: **procedure** LABELED-APSP(V, E, S)
2: **for** $i \in V$ **do**
3: **for** $j, k \in V$ **do**
4: $C_{jk} \leftarrow$ GenerateCandidates(E_{ji}, E_{ik}, S)
5: **if** j=k **then**
6: $S \leftarrow$ CheckForNegCycles(C_{jk}, S)
7: **else**
8: $E_{jk} \leftarrow$ MergeCandidates(E_{jk}, C_{jk}, S)
9: **end if**
10: **end for**
11: **end for**
12: **return** E, S
13: **end procedure**

14: **procedure** GENERATECANDIDATES(E_{ji}, E_{ik}, S)
15: $C_{jk} \leftarrow \{\}$
16: **for** $a \in E_{ji}$ **do**
17: **for** $b \in E_{ik}$ **do**
18: $w \leftarrow a.weight + b.weight$
19: $l \leftarrow a.label \cap b.label$
20: **if** l is consistent with S **then**
21: $C_{jk} \leftarrow C_{jk} \cup Edge\,(j, k, w, l)$
22: **end if**
23: **end for**
24: **end for**
25: **return** C_{jk}
26: **end procedure**

27: **procedure** CHECKFORNEGCYCLES(C_{jk})
28: **for** $c \in C_{jk}$ where $c.weight < 0$ **do**
29: RemoveFromAllLabels($c.label$)
30: $S \leftarrow S - c.label$
31: **if** S is empty **then**
32: **return** inconsistent DTP
33: **end if**
34: **end for**
35: **return** S
36: **end procedure**

37: **procedure** MERGECANDIDATES(E_{jk}, C_{jk}, S)
38: **for** $c \in C_{jk}$ **do**
39: partition E_{jk} into $E_{\leq}$ and $E_{>}$
40: $l \leftarrow c.label - \bigcup_a E_{\leq,a}.label$
41: **if** l is consistent with S **then**
42: **for** $a \in E_{>}$ **do**
43: $a.label \leftarrow a.label - l$
44: **end for**
45: $E_{jk} \leftarrow E_{\leq} \cup E_{>} \cup Edge\,(j, k, c.weight, l)$
46: **end if**
47: **end for**
48: **return** E_{jk}
49: **end procedure**

DIDATES finds the possible candidates C_{ij} by searching for pairs of edges a, b from E_{ji} and E_{ik} (lines 16-17) that can form an alternate path. This shortcut's weight is the sum of a and b and its label is the intersection of a and b's labels (lines 18-19). If the new label l is consistent with the set of possible assignments S, the edge is created and added to the set of candidates (lines 20-21).

After generating the candidate edges representing shorter paths on line 4, LABELED-APSP must use them to update the graph (lines 5-8). If the candidates are edges that start and end at the same node, they are used to check consistency. Recall that negative cycles imply inconsistency in component STPs, so if a negative self-loop candidate is found, its label must be removed from the set of possible full assignments S and from all other edge labels. This process is performed by CHECKFORNEGCYCLES and effectively removes all component STPs with that negative cycle from consideration so that the dispatcher can avoid them at runtime without searching. All other candidates are inserted into the graph with MERGECANDIDATES on line 8. To insert a candidate c into the ordered list E_{jk}, first split the list into two parts: $E_{\leq}$ contains edges with lower or equal weights and $E_{>}$ holds the edges with higher weights (line 39). To ensure that no tighter constraints exist than c, all environments subsumed by labels of edges in $E_{\leq}$ are removed from the label of c (line 40). If c still has a non-empty, consistent label, it is inserted into the list and all environments subsumed by the modified label are similarly removed from all edges with higher weights (lines 41-45).

In Figure 2 the distance graph with the rover example is shown with the two edges derived for the path $\overline{WZ}$, shown with dashed arcs. The edges with weights two and four were are created by generating candidates using nodes X and Y as short-cuts, respectively. All three $\overline{WZ}$ edges remain in the compiled form, kept ascending in order. Although the universally labeled five is actually unnecessary, the algorithm presented would not remove it because none of the lower weight edges totally subsume its label. Note that no edge is ever derived for $\overline{XY}$ because any constraint between those points requires two concurrent assignments to A, which the executive does not need to consider, although the executive may incidentally satisfy both constraints in some executions as the choices do not need to be mutually exclusive. These labels are noted as unnecessary and the edges are discarded immediately when the candidate edges are generated.

The graph resulting from LABELED-APSP is either identified as inconsistent or is a dispatchable network. To make a minimal network, the redundant edges are trimmed from the graph. Trimming is essentially unchanged from the STP algorithm, except that for an edge a to be *dominated* by another edge b, the label of a must be subsumed by the label of b and the third edge of the triangle. Otherwise, b might provide evidence that a is unnecessary even though b does not actually hold in all the situations where a does. Since Drake generates minimal labels for each of the derived constraints and the graph is then trimmed of redundant edges, the compiled dispatchable representation of the DTP is compact, reducing the space required to store the network.

Dispatching the Compact Encoding

The LABELED-APSP algorithm provides Drake with a compact representation of the DTP, which is dispatchable with a variant of the standard STP executive. The dispatcher uses this shared information identified in the compact form to quickly make scheduling decisions on-line. As with the compiler, we generalize the standard STP dispatching algorithm to handle the labeled distance graph. Since Drake dispatches DTPs, it must dynamically choose execution times for the events and select among the possible disjuncts. Here we explain how the standard STP local propagation and event execution algorithms are modified to work with the labeled distance graph and the additional choices available.

The STP dispatcher uses *execution windows* to track the current restrictions on when future events may be scheduled, which are updated by propagating the time of event executions through the constraints in the distance graph. Drake again modifies these operations by maintaining ordered lists of the upper and lower bounds with labels. As with the edges, the lists are sorted to place the tightest constraints at the beginning of the list. Propagating an execution time through an edge creates a candidate bound with the same label as the edge, which is inserted into these lists in essentially the same manner as shown by MERGECANDIDATES above, except that the partition process is reversed for lower bounds. This process allows Drake to compactly track the execution windows for all possible futures without making copies of the events for each component STP.

Although a solution to a DTP might obey every STP constraint in the disjunctive constraints, in general a solution will only enforce a subset of the disjuncts. Constraints from different possible solutions are mixed together, so Drake needs to separate out only a single set of constraints to enforce. To maintain a least commitment approach at run-time, Drake begins by considering all possible solutions and selectively removing some choices from the disjuncts to make progress through the plan, converging to at least a single disjunct from each choice that it will enforce. In this way, it is able to delay commitment until run-time and only use local search to ensure that the final execution will satisfy the DTP.

For an event to be executed by a dispatcher, it must be *alive* and *enabled*, meaning that all events constrained to happen before it have already occurred and that the current time falls within event's execution window. However, both of these conditions depend upon the set of constraints the dispatcher enforces. Drake may execute an event i at the current time if it can violate all the contradicting constraints: any upper and lower time bounds excluding the current time and any edges specifying that non-executed events must occur before i. These constraints may be violated if there is an alternative choice of constraints that Drake could still enforce. Drake checks this condition by testing whether the full assignments in the set S that are subsumed by the labels of those constraints can be removed from S and still leave possible full assignments. This checks whether all the remaining possible solutions require enforcing that constraint, in which case the event cannot be executed. If the event is executed, then S is pruned of those assignments.

The algorithm for this reasoning is shown in Algorithm

2, which determines the new set of assignments S' if the given event e is executed at the current time t. Its inputs are the event under consideration e, the edges of the graph E, the set of assignments S, the list of events that have already been executed V_{exec}, and the current time t. Lines 3-15 find all the constraints that preclude the execution of the event e at the current time and unions their labels to determine which disjuncts would not be satisfied if the constraints were broken. Then line 16 determines if there are still available solutions to the DTP if those disjuncts are not enforced. If there are, then lines 17-18 return true and the new set of full assignments pruned of options that become infeasible if e is actually executed. Otherwise lines 19-20 return false.

Algorithm 2 Event Selection Algorithm

```
 1: procedure EVENTEXECUTABLE?(e, E, S, V_exec, t)
 2:     l ← {}
 3:     for ub ∈ e.upper_bounds, ub.bound < t do
 4:         l ← l ∪ ub.label
 5:     end for
 6:     for lb ∈ e.upper_bounds, lb.bound > t do
 7:         l ← l ∪ lb.label
 8:     end for
 9:     E_e− ← non-positive lower bound edges
10:     starting with e
11:     for edge ∈ E_e− do
12:         if edge.start is not in V_exec then
13:             l ← l ∪ edge.label
14:         end if
15:     end for
16:     S' ← S − l
17:     if S' is not empty then
18:         return true, S'
19:     else
20:         return false
21:     end if
22: end procedure
```

To conclude our presentation of Drake's algorithms, we explain how the dispatch process works on the compiled rover example. After compilation, S includes both assignments to A, as both produce consistent STPs. At the first time step, arbitrarily chosen as $t = 0$, the start event W is executed. This propagates through the graph, putting upper and lower bounds on X and Y with the labels on those paths. Z will have three upper and lower bounds, one for each assignment to A and a third with a universal label because the compiled form has six total arcs between W and Z. At later time steps, Drake can choose to schedule either X or Y, but not both, at an arbitrary time (or elect not to schedule it). Doing so violates a constraint labeled with $A = 1$ or $A = 2$, respectively, which is allowed if the other assignment's constraints are enforced. At a later time $t = 5$, Z needs to be executed to satisfy the global constraint, and Drake may do so even if X was never scheduled as long as Y was scheduled according to the $A = 2$ constraints. In this case, X does not need to come before Z because Drake already committed to neglecting the $A = 1$ constraints by failing to schedule X. Note that once one of the assignments is discarded when Drake violates a constraint, all the constraints with the other label become binding to ensure that the execution is correct. Drake's dispatcher leverages the compact recording of the constraints from the labeled compilation algorithm to perform a smaller number of computations.

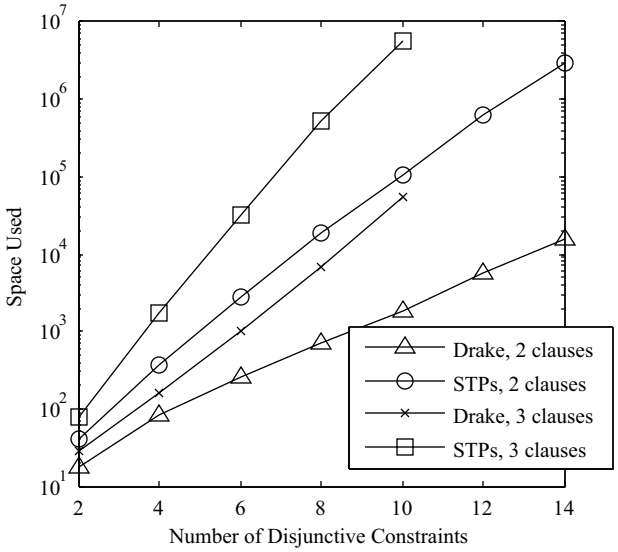

Figure 3: The space used by the dispatchable graphs compiled with Drake's method and by listing component STPs with disjunctive constraints with two and three clauses each.

Results

We evaluated Drake's performance on randomly generated structured problems to characterize the space savings on realistic problems. The problem generator was based on Stedl's random structured STPU generator, which we will briefly outline (2004). First, a series of activities are laid out on a coordinate grid with random ordered constraints. Then constraints are added to nearby events with magnitudes proportional to their separation on the grid, which may be ordered or unordered. Finally, local sets of edges are selected to be the simple intervals in the disjunctive constraints. In our setup we varied the number of activities in the plan and used one disjunctive constraint per activity. Plans are generated with twice the number of events as activities and roughly four times the constraints as activities, following previous work that realistic problems are not highly constrained (Tsamardinos 2001). Twenty problems are tested at each size, and the number of simple interval clauses per disjunctive constraint is varied between two and three.

We compile the random problems using both Drake's compilation technique and using Tsamardinos's approach of explicitly enumerating all the consistent component STPs and compared the space requirements (Tsamardinos, Pollack, and Ganchev 2001). The space is counted as the number of edges plus the number of nodes in the representation. To represent the costs of the labels Drake stores, we also

count the number of full assignments in S. The results are shown in Figure 3. Especially on larger problems, Drake uses much less space than prior art, by two orders of magnitude when compared for disjuncts of order two and three. We also note that for the larger problems tested, roughly those with 10^5 or more storage space used, the technique in prior art is infeasible because all the component STPs could not fit in memory at the same time.

Discussion

Based on Shah and Williams' results, we expect that reducing the size of the dispatchable form of the disjunctive problem will significantly reduce the response time at execution. Since Drake is a proof of concept implementation, it did not include several features known to make ATMS systems perform well, so the execution latency observed in the dispatcher is still relatively high. However, these results indicate that a more advanced implementation of the labeling system will yield dramatic reductions in the execution latency (De Kleer 1986). Furthermore, a better labeling system will improve the compiler by providing a more compact version of the solution set S.

As we have tried to emphasize throughout, the elegance of Drake is that it involves relatively simple modifications to algorithms for non-disjunctive forms of the temporal problem to reason efficiently over the shared structure of the disjunctive problem. This method is effective because during the compilation process the algorithm directly calculates the dependence of each derived constraint on the choices in the original plan and is thus able to extract the minimal number of constraints to represent all the possible choices. We believe that recent work in efficient STP reformulation (Tsamardinos, Muscettola, and Morris 1998; Xu and Choueiry 2003; Planken et al. 2008) may be adapted to consider labels, thereby creating a more efficient compilation technique than the LABELED-APSP algorithm shown here, but this is left as future work.

Conclusion

We have presented Drake, a novel system for dynamically executing temporal plans with choices. By deferring choice until execution time, autonomous systems can avoid overly conservative plans and improve robustness by allowing uncertainty in the world to unfold before making decisions. Drake takes the plan and reformulates it as a Disjunctive Temporal Problem, which is then compiled using a variant of the standard STP compilation algorithm designed to exploit common structure to compress the dispatchable form of the plan. Drake shows improvement in the space to store the solution set by up to two orders of magnitude, which is an important step towards reducing the execution latency for the flexible dispatch of these plans.

References

De Kleer, J. 1986. An assumption-based TMS. *Artificial intelligence* 28(2):127–162.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artificial Intelligence* 49:61–95.

Forbus, K., and De Kleer, J. 1993. *Building problem solvers*. MIT press.

Kim, P.; Williams, B.; and Abramson, M. 2001. Executing reactive, model-based programs through graph-based temporal planning. In *International Joint Conference on Artificial Intelligence*, volume 17, 487–493.

Muscettola, N.; Morris, P.; and Tsamardinos, I. 1998. Reformulating temporal plans for efficient execution. In *Principles of Knowledge Representation and Reasoning-International Conference*, 444–452.

Oddi, A., and Cesta, A. 2000. Incremental forward checking for the disjunctive temporal problem. In *ECAI*, 108–112.

Planken, L.; de Weerdt, M.; van der Krogt, R.; Rintanen, J.; Nebel, B.; Beck, J.; and Hansen, E. 2008. P 3 C: A New Algorithm for the Simple Temporal Problem. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, 256–263. AAAI Press.

Shah, J. A., and Williams, B. C. 2008. Fast Dynamic Scheduling of Disjunctive Temporal Constraint Networks through Incremental Compilation. In *Proceedings of the International Conference on Automated Planning and Scheduling*.

Shah, J.; Stedl, J.; Williams, B.; and Robertson, P. 2007. A Fast Incremental Algorithm for Maintaining Dispatchability of Partially Controllable Plans.

Stedl, J. 2004. Managing temporal uncertainty under limited communication: a formal model of tight and loose team coordination. Master's thesis, Massachusetts Institute of Technology.

Stergiou, K., and Koubarakis, M. 2000. Backtracking algorithms for disjunctions of temporal constraints. *Artificial Intelligence* 120(1):81–117.

Tsamardinos, I., and Pollack, M. 2003. Efficient solution techniques for disjunctive temporal reasoning problems. *Artificial Intelligence* 151(1):43–89.

Tsamardinos, I.; Muscettola, N.; and Morris, P. 1998. Fast transformation of temporal plans for efficient execution. In *Proceedings of the National Conference on Artificial Intelligence*, 254–261.

Tsamardinos, I.; Pollack, M.; and Ganchev, P. 2001. Flexible dispatch of disjunctive plans. In *6th European Conference on Planning*, 417–422.

Tsamardinos, I. 2001. *Constraint-based Temporal Reasoning Algorithms with Applications to Planning*. Ph.D. Dissertation, University of Pittsburgh.

Williams, B.; Ingham, M.; Chung, S.; and Elliott, P. 2003. Model-based programming of intelligent embedded systems and robotic space explorers. *Proceedings of the IEEE* 91(1):212–237.

Xu, L., and Choueiry, B. 2003. A new effcient algorithm for solving the simple temporal problem. In *Temporal Representation and Reasoning, 2003 and Fourth International Conference on Temporal Logic. Proceedings. 10th International Symposium on*, 210–220.