Enabling Fast Flexible Planning through Incremental Temporal Reasoning with Conflict Extraction

I-hsiang Shu, Robert Effinger, and Brian Williams

MIT Computer Science and Artificial Intelligence Laboratory (CSAIL)
Bldg. 32-G273, 77 Massachusetts Ave., Cambridge, MA 02139
{ishu, effinger, williams}@mit.edu

Abstract

In order for an autonomous agent to successfully complete its mission, the agent must be able to quickly re-plan on the fly, as unforeseen events arise in the environment. This is enabled through the use of temporally flexible plans, which allow the agent to adapt to execution uncertainties, by not over committing to timing constraints, and through continuous planners, which are able to replan at any point when the current plan fails. To achieve both of these requirements, planners must have the ability to reason quickly about timing constraints.

We enable continuous, temporally flexible planning through a temporal consistency algorithm (ITC), which supports incremental consistency testing on a new type of disjunctive temporal constraint network, the Temporal Plan Network (TPN), and supports focused search through incremental conflict extraction. The ITC algorithm combines the speed of shortest-path algorithms known to network optimization with the spirit of incremental algorithms such as Incremental A* and those used within truth maintenance systems (TMS). Empirical studies of ITC applied to the Kirk temporal planner demonstrate an order of magnitude speed increase on cooperative air vehicle scenarios and on randomly generated plans.

Introduction

Autonomous robots and vehicles are quickly becoming an integral part of modern society. These autonomous agents have long been building and assembling our automobiles. In the future, these agents will perform more complex tasks, such as Mars exploration and flying unmanned aerial vehicle missions for search and rescue.

Due to the dynamic and unpredictable nature of these planning environments, complex autonomous missions will require planners that are capable of *continuous planning* (Estlin et al. 2000). Continuous planners, such as ASPEN (Rabideau et al. 1999) are capable of quickly generating a new plan as soon as an environment change breaks the current mission plan.

A downside of these continuous planners is that they do not allow for temporal flexibility in the execution time of activities, as they assign hard execution times to activities. Temporally flexible planners, such as HSTS (Muscettola et al. 1998), are able to adapt to perturbations in execution time without breaking the entire plan. These planners only impose those temporal constraints required to guarantee a plan's success, leaving flexibility in the execution time of activities. This flexibility is then exploited, in order to adapt to uncertainty, by delaying the scheduling of each activity until it is executed.

To be robust to major disturbances that lead to plan failure, a temporally flexible planner must be able to replan quickly. However, state of the art temporally flexible planners have not yet achieved the efficiency of continuous planners, like Aspen. Our objective is to provide the computational building blocks that enable *continuous, temporally flexible planning*.

The core task repeatedly performed by a continuous, temporally flexible planner is to determine the temporal consistency of each candidate plan. Simply put, all such planners generate a candidate plan and then test the plan for temporal consistency. This generate and test loop highlights two ways to increase planning speed: 1) Increase the speed of the testing phase by speeding up the temporal consistency checking algorithm, and 2) decrease candidate plan generation, by improving the generator's ability to prune candidates without generation.

We achieve dramatic increases along both fronts by drawing upon principles of incremental reasoning (Koenig and Likhachev 2001) (McAllester 1990) (Gerevini et al. 1996) (Cesta and Oddi 1996), and conflict-directed search (Ginsberg 1993) (Williams and Ragno 2002), which have been used to achieve efficient consistency checking of simple temporal networks (STNs), and to achieve efficient model-based diagnosis, respectively. We increase efficiency during testing by providing an incremental temporal consistency algorithm (ITC) that reasons in terms of only the differences between the temporal constraints of successive candidate plans. Our empirical results show that these differences and their logical consequences are small relative to the overall plan size, resulting in a significant decrease in the number of temporal inferences (arc updates) performed. We increase efficiency of candidate plan generation by identifying the subset of temporal constraints that lead to temporal inconsistency, known as *conflicts*, and use these conflicts to prune sets of infeasible candidate plans, without explicitly generating them. Our empirical results also show that the number of candidate plans generated using conflicts is significantly reduced.

The central focus of this paper is the ITC algorithm and its empirical evaluation. While ITC is planner independent, we benchmark it using the Kirk planner, reported elsewhere in (Kim, Williams, and Abrahmson 2001). First, we introduce background in temporal consistency checking of STNs, using shortest path algorithms on distance graphs. Second, we introduce ITC's algorithm for checking incremental temporal consistency. The ITC algorithm has similar ties to work by (Cesta and Oddi 1996) and (McAllester 1990) in which a set of support is used to perform incremental updates. Third, we augment ITC with an algorithm for conflict extraction that is itself incremental. The challenge of this task is to maintain a correct set of support incrementally, as an STN moves from inconsistent back to consistent. Fourth, we describe how ITC is incorporated within the generate and test loop of a temporally flexible planner in general, and specifically for the Kirk planner. Kirk is part of an executive that generalizes temporally flexible plan execution to the execution of temporally flexible contingent plans. Kirk selects a feasible plan from the set of contingencies, by encoding the contingent plan in a Temporal Plan Network (TPN) (Kim, Williams, and Abrahmson 2001), and by solving the TPN as a temporal conditional CSP. The TPN encapsulates a novel disjunctive temporal network, distinct from the Disjunctive Temporal Problem(DTP) (Stergiou and Koubarakis 1998), and the Conditional Temporal Problem(CTP) (Tsamardinos et al. 2003). Finally, we evaluate the performance of our ITC implementation within Kirk, applied to a range of structured and unstructured, randomly generated planning problems.

Background: Consistency of STNs

The temporal constraints of a candidate plan are expressed as an STN. An STN is checked for temporal consistency by first converting the STN to an equivalent representation, called a *distance graph*. The STN is temporally consistent if and only if its corresponding distance graph does not contain a negative cycle (Decter, Meiri, and Pearl 1991).

Simple Temporal Network (STN)

An STN is comprised of a set of nodes, representing temporal events, and labeled arcs between nodes, representing constraints on the duration between two events. Each arc has a label *[l,u],* representing the lower l and upper u bounds on the duration from the event at the tail of the arc to the event at the arc's head. For example, the simple temporal constraint in Figure 1 says that *End-engine-start* must occur between 1 and 5 time units after *Begin-engine-start*.

STN to Distance Graph Conversion

A distance graph is similar to an STN, in that the nodes in a distance graph represent time points. In a distance graph, however, an arcs label u specifies only an *upper bound* on the duration from the tail event t to the head event h of the arc ($h - t \leq u$).

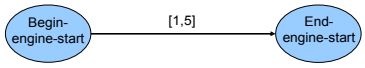


Figure 1 Example STN

An STN is converted to a distance graph by copying the nodes and by mapping each arc of the STN to two additional arcs, one in the forward direction and one in the reverse direction. The forward arc is labeled with the value of the upper time bound and the reverse arc is labeled with the negative of the lower time bound value (Figure 2).

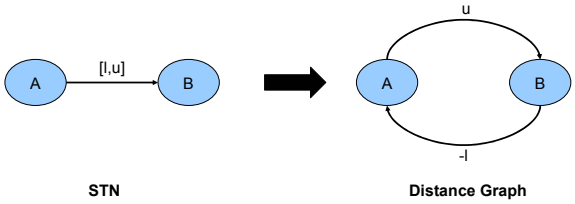


Figure 2 STN to Distance Graph Conversion

The equation below specifies how each timepoint constraint for an STN is converted to a constraint for the distance graph for an arbitrary arc_i.

$$\underset{\text{STN}}{T_j - T_i \in [l,u]} \quad \Rightarrow \quad \underset{\text{Distance Graph}}{T_j - T_i \leq u \;\cap\; T_i - T_j \leq -l}$$

Figure 3 Equation for STN to Distance Graph Conversion

As an example, in Figure 2, timepoint *B* is executed at most *u* time units after timepoint *A*. Similarly, since timepoint *A* occurs before timepoint *B*, timepoint *A* must be executed at most *-l* time units after timepoint *B*, or equivalently, timepoint *A* must be executed at least *l* time units before timepoint *B*.

Detecting Temporal Inconsistency through Negative Cycle Detection

In order for an STN to be temporally consistent, the equivalent distance graph of the STN must not contain a negative cycle. This is proved rigorously in (Decter, Meiri, and Pearl 1991). Intuitively, since the edge weights in the distance graph represent the amount of time that an event must happen before another event (e.g. event *B* must happen at least *l* time units after event *A* and event *A* must happen at least *u* time units before *B*), then a negative cycle in the distance graph would correspond to having a temporal constraint saying that a timepoint must happen at most some positive time units before the same timepoint (e.g. event *A* must happen at least 5 time units before event *A*). Having a constraint such as this makes little sense and is the basis for the intuitive argument.

Negative Cycle Detection Using Label-Correcting Algorithms

In order to find a negative cycle in the distance graph, it is unnecessary to compute the shortest-path for every pair of nodes, as compiled by APSP algorithms. If a negative cycle exists, it can be detected by just computing the shortest-paths from one single node to all the other nodes, that is, the single source shortest path (SSSP). The reason is, if a node is involved in a negative cycle, then the shortest-path to that node from any source node connected to it is $-\infty$. This follows because a shortest-path can continually loop along the negative cycle, reducing path distance indefinitely.

There are several ways for a shortest path algorithm to determine it has entered a negative cycle. Most of these algorithms are based on the concept of label correction, in which an edge weight is incrementally reduced to its shortest path value. The simplest, and most conservative cycle detection algorithm, is to observe that a node's shortest-path value drops below –nC, where n is the number of nodes in the STN, and C is the value of the largest forward arc label in the STN. A faster technique is to keep a spanning tree of the shortest-path support for each node, and terminate as soon as a cycle is detected in the spanning tree (Cesta and Oddi 1996). A third method is to check if any node's shortest-path value has been updated twice. This method, however, can only be used when the candidate STN begins from a consistent context.

Using only a SSSP algorithm offers significant savings over an APSP algorithm. As an example, the runtime for Floyd-Warshall's APSP algorithm is $\theta(n^3)$, where n is the number of nodes in the graph. The SSSP algorithm, such as the FIFO label-correcting algorithm, has a worst-case runtime of O(nm), where n is the number of nodes and m is the number of arcs in the graph.

```
FIFO Label -Correcting Algorithm
(Graph G )

{01}  for all s   ∈ V(G)
{02}     d(s) =   ∞
{03a}  d(Sstart) = 0
{03b}  Q.insert( Sstart )
{04a} while !Q.empty()
{04b}     u = Q.pop()

{05a}    for v   ∈ Succ(u)
{05b}      dval = Update(u,v)
{06}        if(dval) < -nC
{07}                 return false;
{08}  return true;

value Update (p,x)
{09}  if (d(x) > d(p) + c(p,x))
{10}     d(x) := d(p) + c(p,x);
{11}     Q.Insert(x, d(x));
{12}  return d(x);
```

Initialize → {03a}
Check for Violating Arcs → {04a}
Update → {05a}

Figure 4 Pseudo-code for FIFO Label-Correcting Algorithm

FIFO Label-Correcting Algorithm

ITC is a variant of a label-correcting algorithm. Label-correcting algorithms find shortest-paths by performing three key procedures. It first initializes shortest-path values, d, to ∞, scans arc costs, c, for whether shortest-paths can be improved (if $d(x) > d(p) + c(p,x)$), and then updates these arcs with new values. The algorithm then iterates until all violating arcs have been updated.

The FIFO label-correcting algorithm simply refers to an efficient implementation of the generic label-correcting algorithm in which a queue of updated nodes is maintained, in order to check for outgoing arcs that might be potentially violating. If during a particular iteration of the algorithm, the shortest-path distance, *d(i)*, from the source to node *i* was not updated, then no new information is learned about the shortest-path to that node. Any arc emanating from that node that was not violating before the update is still not violating after the update, and need not be scanned. Conversely, if an update occurs for a particular node *i*, then *d(i) + c(i,j)* may have become less than *d(j)*, thus any outarc *(i,j)* may have become violated. Hence to find violated arcs, it is sufficient to add each update node to a queue and then examine all of the outarcs of a node on the queue.

At initialization of the FIFO label-correcting algorithm, only the start node's outarcs are potential violating arcs, because the other node's start distances are set to ∞. Thus, only the start node is put initially in the queue. As nodes are taken out of the queue and updates occur, these updated nodes are added to the queue, requiring additional examination of the outarcs of the queued node. Once the queue is empty and consequently no arcs remain, we have the optimal shortest-path solution. The pseudo code for the FIFO label-correcting algorithm is shown in Figure 4.

The worst-case running time for a label-correcting algorithm is much faster than any all-pairs shortest-path algorithm, O(nm) versus $O(n^2 \log n + nm)$ of Johnson's APSP algorithm. However, using the modified label-correcting algorithm with an efficient implementation of the update queue, the average case runtime of the algorithm can be reduced significantly, sometimes to O(m) (Ahuja, et al. 1993). For simplicity, we show the conservative –nC method for negative cycle detection, but a more efficient method, such as the spanning tree method, could also be used.

Next we develop a variant of the FIFO Label-Correcting Algorithm which is incremental and has three novel characteristics: 1) doesn't assume the candidate STN starts from a consistent context, 2) has a conflict extraction mechanism, and 3) allows multiple arc changes at once.

The Incremental Temporal Consistency Algorithm (ITC)

Overview

A temporal planner requests temporal consistency checks on STNs of candidate plans as they are built up, constraint by constraint, and as constraints are removed, when shifting to alternative candidates. As a result, the STN of the new plan differs from the previous STN only by a few arcs and nodes. This means that only the previously computed shortest-path values that are affected by the newly changed arcs and nodes need to be updated. Temporal consistency of an STN can therefore be determined with fewer node updates. Additionally, if the ITC algorithm finds that a candidate STN is inconsistent, it will return a set of simple temporal constraints involved in the inconsistency. This ultimately increases the speed at which the planner finds a consistent candidate plan. A discussion of conflict extraction algorithms for optimal search together with a performance analysis can be found in (Williams and Ragno 2002). Figure 5 shows how the ITC algorithm communicates with the plan generation algorithm.

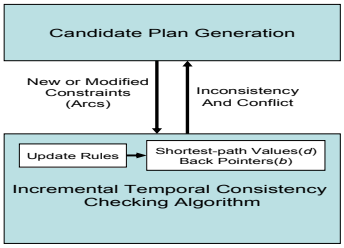


Figure 5 ITC Algorithm

Given a new temporal constraint, the ITC algorithm performs the following steps in order to quickly determine the temporal consistency of the graph and, if inconsistent, the conflict involved in the inconsistency.

ITC Pseudo-Code

When the planner requires a temporal consistency check on a candidate STN, it calls *CheckTemporalConsistency*. Depending on whether the consistency check is starting from scratch or incrementally, the planner will call either *Initialize* or *ModifyConstraint,* respectively. When *CheckTemporalConsistency* returns, it will either return a conflict if there is an inconsistency or it will return no conflict if the graph is consistent.

If an inconsistency is found, the algorithm calls *ExtractConflict to* collect all nodes in the negative cycle and then returns them collectively as the conflict. Next, the algorithm needs to call *updateITCwithNegativeCycle,* which resets all shortest-path values of nodes in the negative cycle, and also resets any shortest-path values of nodes that depend on the negative-cycle for support.

```
void
Initialize()
{01}  Q := ∅;
{02}  for all s ∈ V(G)
{03}    d(s) := ∞;
{04}    p(s) := unknown;
{05}    d(sstart) := 0;
{06}    Q.insert(sstart);

Conflict
CheckTemporalConsistency(G)
{07}  while !Q.empty()
{08}    u = Q.pop();
{09}    for v ∈ Succ(u)
{10}      dval = Update(u,v);
{11}      if(dval) < -nC
{11a}     // or if cycle in spanning tree
{12}        return ExtractConf(c, ∅);
{13}  return ∅;

value
Update(p,x)
{14}  if ( d(y) > d(x) + c(x,y))
{15}    d(y) := d(x) + c(x,y);
{16}    p(y) := x;
{17}    Q.insert(y);
{18}  return d(y);

Conflict
ExtractConflict(c,L)
{19}  if L.contains(c)
{20}    return L;
{21}  else
{22}    L.add(c);
{23}    ExtractConflict(p(c),L);

void
ModifyConstraint(x,y,l,u)
{24}  ModifyArc(arc(y,x),-l)
{25}  ModifyArc(arc(x,y),u)

void
ModifyArc(arc,c)
{26}  setCost(arc,c);
{27}  if (d(arc.head > d(arc.tail) + c)
{28}    d(arc.head) := d(arc.tail) + c;
{29}    p(arc.head) := arc.tail;
{30}    Insert(arc.head);
{31}  elseif (d(arc.head) < d(arc.tail) + c
      AND (p(arc.head) == arc.tail))
{32}    d(arc.head) :=   ;
{33}    p(arc.head) := unknown;
{34}    set<Node> nodes_reset =   ;
{35}    nodes_reset.insert(arc.head);
{36}    nodes_reset.insert(InvalidateSupports(arc.head))
{37}    InsertParents(nodes_reset)

void
InsertParents(set<Node> reset_nodes)
{38}  for all n ∈ reset_nodes
{39}  for all m ∈ Pred(n)
{40}    if( d(n) != ∞ || p(s) != unknown )
{41}      Insert(m);

set<Node>
InvalidateSupports(Node n)
{42}  set<Node> nodes_reset = ∅;
{43}  for all s ∈ Succ(n)
{44}  if( p(s) == n )
{45}    if( s == sstart )
{46}      d(s) := 0;
{47}      nodes_reset.insert( InvalidateSupports(s) );
{48}    elseif( d(s) != ∞ OR p(s) != unknown )
{49}      d(s) := ∞;
{50}      p(s) := unknown;
{51}      nodes_reset.insert( InvalidateSupports(s) );
{52}  return nodes_reset;

void
updateITCwithNegativeCycle(set<Node> neg_cycle)
{53}  set<Node> nodes_reset = ∅;
{54}  for all n ∈ neg_cycle
{55}    d(n) := ∞;
{56}    p(n) := unknown;
{57}    nodes_reset.insert(n)
{58}    nodes_reset.insert( InvalidateSupports(n) );
{59}    InsertParents( nodes_reset );
```

Figure 6 ITC Pseudo-Code

After all nodes that depend on the negative cycle are reset, *modifyConstraint* may be called (multiple times if needed) to make any necessary changes to resolve the conflict. *CheckTemporalConsistency* may then be called again to test consistency of the new candidate STN.

Insufficiency of FIFO Label-Correcting to Perform Incremental Temporal Consistency

In order to perform a temporal consistency check, we must use an algorithm that is capable of detecting negative cycles. As discussed earlier, the FIFO label-correcting algorithm is a good choice because of its efficiency. It also has some of the capabilities needed to perform incremental updates. In particular, the label-correcting algorithm can handle a change that improves a node's shortest-path distance, since all it needs to do is add the node to the queue and propagate the update down the line. However, the label correcting algorithm is not capable of handling cases in which an edge distance *increases the shortest-path* to a node and, as a consequence, a new shortest-path exists. To handle this case, we introduce a set of support for keeping track of which shortest-path distances on nodes affect each other. Additionally, the label correcting algorithm halts when a negative cycle is detected, so it can't reuse any previous computation. Thus, to reuse all previous computations that remain valid, ITC maintains a correct set of support incrementally as a negative cycle is discovered and the STN moves from inconsistent back to consistent.

ITC Algorithm's Incremental Update Rules

ITC's incremental update rules for an arc change are divided based on how that arc change affects the shortest-path distance at its head node. There are three types of effects (1) no effect to the current shortest-path, (2) improves the shortest-path, and (3) invalidates the current shortest-path.

(1) Arc Change without Effect to Shortest-Path

An arc can change in such a way that the shortest-path to a node is unaffected. This may be the case either as a result of an arc increase or decrease. The graph in this case requires no updates, because the shortest-path distances have not changed.

For example, in Figure 7, the current best way to reach

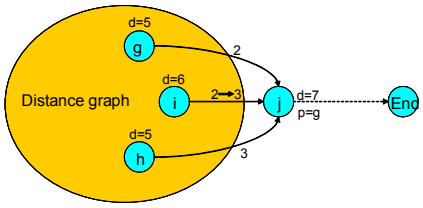


Figure 7 Arc Change Without Effect to Shortest-Path

node *j* is to go through node *g*, as specified by the predecessor pointer (p=g) of node *j*. This path reaches node *j* with a cost of 7. The figure indicates that arc_{ij} increases from a cost of 2 to a cost of 3. With the distance increased, the *d(j)* for a path through the newly changed arc would be *9*. This value is still worse than the current best value of *7*, therefore, the shortest path value at node *j* does not need to be updated, and no further updates need to be performed.

(2) Arc Change Improves Shortest-Path

An arc distance decrease can improve the shortest-path to one or more nodes. This can happen when the changed arc is either on or off the current shortest-path to the head node. In either case, the shortest-path distance value of the node at the head of the changed arc needs to be updated, and this updated distance value is propagated to successor nodes.

For example, in Figure 8, arc_{ij} reduces in cost from *3* to

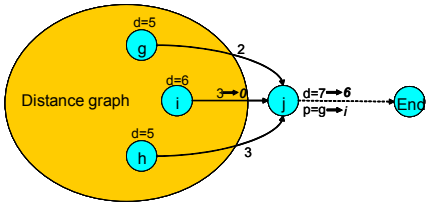


Figure 8 Shortest-Path Improvement

0. With this change, the shortest-path distance to node *j* can be decreased from *7* to *6*, through node *i*. The predecessor pointer should now point to node *i,* instead of node *g*, and the shortest-path value should be updated to *6*. As a final update step, since the successor nodes of node *j* can be affected by the improvement to node *j*'s shortest-path distance, node *j* is added to the algorithm's update queue. When the node is subsequently dequeued, the outgoing arcs from node *j* are examined for updates.

Cases (1) and (2) are already handled by the FIFO label-correcting algorithm but case (3) ,below, is not.

(3) Arc Change Invalidates Shortest-Path

In the final case, an increase in arc distance can *worsen* the current shortest-path to a node. In this case, the node at the tail of the arc is the predecessor for the node at the head of the arc. The set of parent nodes for the changed arc's head node must then be re-examined to determine the new best shortest-path. Additionally, since all nodes supported by this affected node also have invalid shortest-

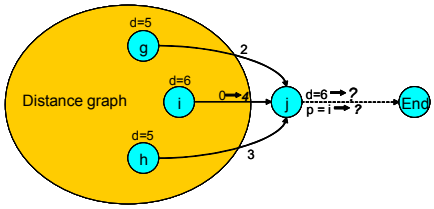


Figure 9 Shortest-Path Invalidated

path distances, a recursive function must be called to invalidate all nodes supported in the chain. Once these shortest-path distances have been invalidated, the parents of the affected node can be enqueued and a new start distance may be propagated from this node.

For example, in Figure 9, arc_{ij} has increased in value, and since node *j*'s shortest-path value of *6* was calculated by traversing through the changed arc, the value at node *j* is no longer valid. ITC first invalidates the shortest-path value for node j and then recursively invalidates the shortest-path values and predecessor pointers of all nodes that use node j in their shortest path. Finally, node j's parents are added to the Q so that new shortest-path values and predecessor pointers can be calculated for all of the invalidated nodes. The predecessor pointer allows ITC to focus only on updating a small set of relevant nodes, similar to how the set of support focuses a truth maintenance system.

This completes the development of the incremental shortest path algorithm when the STN is consistent. Next we augment ITC to extract conflicts and reason incrementally when an inconsistency arises.

Negative Cycle Detection with Conflict Extraction

ITC detects negative cycles in the same manner that the FIFO label-correcting algorithm detects negative cycles. Thus, ITC can use any of the methods described previously for negative cycle detection using label-correcting algorithms. After ITC detects a negative cycle, the set of inconsistent edges are collected by following the predecessor pointers around the cycle. Consider the inconsistent graph shown in Figure 10. It shows the

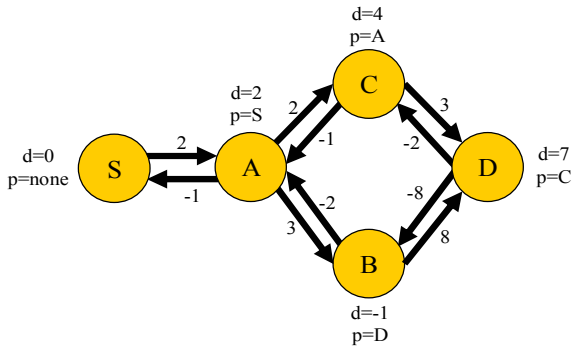


Figure 10 Snapshot of Negative Cycle Just Before Detection

shortest-path values just before a negative cycle is detected.

Notice that in this graph, the set of edges involved in the inconsistency cannot be extracted by following the predecessor pointers. This is because the negative loop has not yet been closed at arc *BA*.

Depending on which method is used to detect the negative cycle, ITC will either continue walking around the negative cycle until it is eventually detected (-nC bound), or the cycle will be detected immediately (spanning tree). Once the cycle is detected, the predecessor pointers are ensured to be cyclically dependant so that the source of the conflict can be identified. Figure 11 shows the state of the algorithm a few steps after the negative cycle has been closed. If using the spanning tree method to detect negative cycles, ITC would have detected this negative cycle as soon as it was closed, and if using the –nC bound to detect negative cycles, ITC would need to continue walking the cycle until a node value drops below -40.

As Figure 11 shows, node *A*'s predecessor pointer now

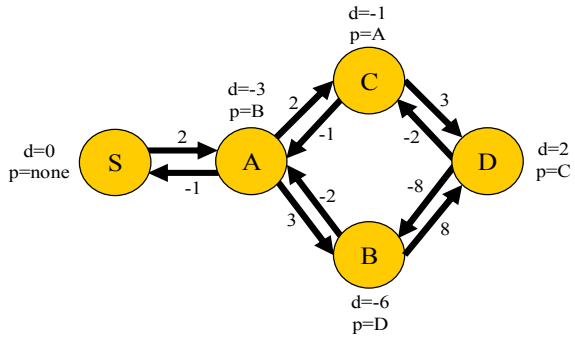


Figure 11 Snapshot of Negative Cycle After Detection

points to node *B*, completing the cycle. We can now extract the conflict by walking the predecessor pointers and report that this graph was found to be inconsistent with the negative cycle, *ACDBA*. Notice that once ITC detects a negative cycle, any shortest-path distance values computed for nodes in the negative cycle are meaningless. This is evident in Figure 11, because negative start times from the start node are realistically impossible. This means that before resolving the inconsistency, ITC must invalidate all nodes in the negative cycle, and must also invalidate any nodes that depend on the negative cycle for support. This is accomplished by a call to *updateITCwithNegativeCycle*.

Incremental Update after Negative Cycle Detection

ITC's *updateITCwithNegativeCycle* takes as input the negative cycle, and must perform three steps to maintain a correct set of support: 1) reset every node in the negative cycle by setting d(n) to ∞, and the predecessor pointer to unknown, 2) reset all nodes that depend on the negative cycle by calling *InvalidateSupports* on each node in the

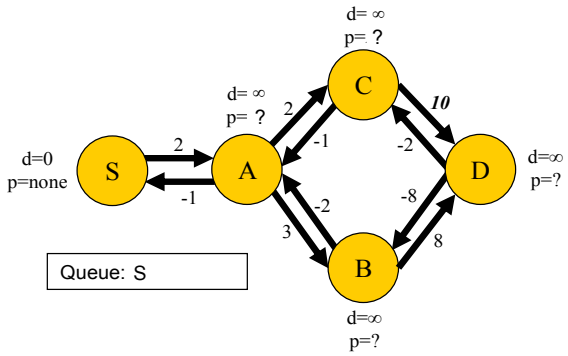


Figure 12 Snapshot after calling *updateITCwithNegativeCycle*

negative cycle, and 3) call *InsertParents* on the set of all nodes that were reset in steps 1 or 2. This inserts onto the Q any parent node that has not also been invalidated. For example, Figure 12, shows the graph after a call to *updateITCwithNegativeCycle*. In this small example, all shortest-path values but S were reset, however, for a larger STN, all shortest-path calculations upstream of node S remain valid, and can be reused without examination.

Next we show how the conflict returned by ITC is used to incrementally generate a new candidate plan.

Inconsistency Resolution

A planner will take the conflict from the ITC algorithm and then use it to generate a new candidate plan that does not contain the conflict. Consider how ITC performs an incremental update to shift from an inconsistent candidate plan to a new candidate. For example, imagine that the planner changes activity *CD* of a plan so that its upper bound is increased to *10*. This corresponds to an increase in the distance of *CD* from *3* to *10*. ITC incrementally updates the graph by calling *modifyConstraint* on *CD*.

Notice, in Figure 12, that by calling modifyConstraint, neither node C or D are added to the Q. This is because they have both already been invalidated during the negative cycle update. Nodes C and D are already guaranteed to be updated as new shortest-path values propagate through the graph initiating from node S.

Since changing arc *CD* to *10* greatly increased the path that was on the negative cycle, this altered graph or new candidate plan is temporally consistent. As shown in Figure 13, the ITC algorithm will return this answer after *checkTemporalConsistency* has updated and removed all nodes from the queue.

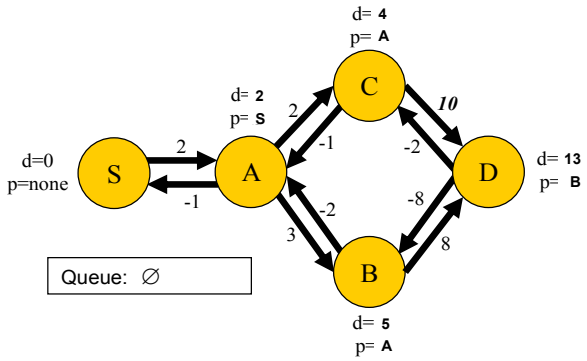


Figure 13 ITC Algorithm After Finding a Consistent Solution

Note that a unique feature of ITC is that *modifyConstraint* can be called multiple times before calling *checkTemporalConsistency*. This is important for cases when multiple arcs need to be modified in order to resolve an inconsistency. Also note that multiple and intertwined negative cycles pose no problem for ITC's incremental conflict extraction and inconsistency resolution framework.

Continuous Planning with ITC

Next we return to how a typical temporally flexible planner uses ITC to achieve efficiency. The generate and test loop of a planner using ITC was previously depicted in Figure 5. The candidate plan generator takes as input a conflict, supplied by ITC, generates a new candidate plan, and outputs the STN differences between the successive candidate plans. ITC then takes these changes to the STN as input, applies its incremental update rules to modify the STN, incrementally tests the new STN for temporal consistency, and outputs a conflict if one is found. The planning algorithm terminates if an empty conflict is returned from ITC (signaling that a consistent STN was found), or if no new modifications are suggested by the candidate plan generator (signaling that there are no more candidate plans to try, and planning has failed).

We have implemented and evaluated an instance of a continuous, temporally flexible planner by incorporating ITC within the Kirk planner. Kirk can be viewed as a hierarchical task network planner that supports temporal flexibility. Kirk supports efficient planning by compiling its planning domain knowledge into a graphical structure called a *temporal plan network (TPN)*. A TPN is similar to a temporally flexible plan, that is, it includes activities, predecessor and successor relations between activities, and simple temporal constraints that relate the start and end times of activities. Additionally, a TPN represents options or contingencies in a plan by augmenting a temporally flexible plan with choice nodes. Figure 14 presents a concise definition of the TPN; the complete definition of a TPN, which includes mutex and resource support, is developed in (Kim, Williams, and Abhramson 2001).

TPN := A [lb , ub] |
(Parallel TPN1 , TPN2 , …) |
(Sequence TPN1 , TPN2 , …) |
(Choose TPN1 , TPN2 , …)

with the graphical equivalents:

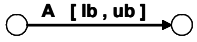
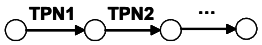
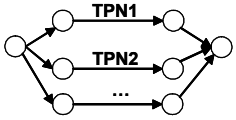
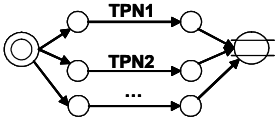
A [lb , ub]	
Sequence (TPN1 , TPN2 , …)	
Parallel (TPN1 , TPN2 , …)	
Choose (TPN1 , TPN2 , …)	

Figure 14 Definition of a Temporal Plan Network (TPN)

In a TPN, a choice between alternative courses of action is represented by a choice start node, (represented by a double circle), a choice end node (represented by a circle with two parallel lines), and alternative subplans between them. Figure 15 shows an example TPN with a parallel set of activities branching at node P and converging at node F. The example TPN also has a choice between two possible subplans, C1 and C2.

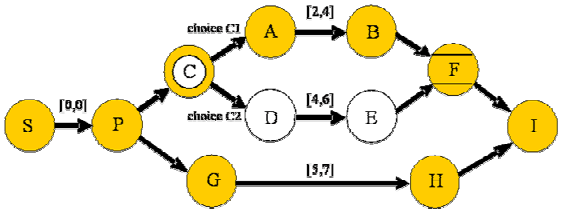


Figure 15 An Example TPN

Roughly speaking, Kirk picks a candidate plan, and its corresponding STN, by choosing one and only one execution path thru each of the choice start and choice end nodes in the TPN. Therefore, a TPN represents a family of closely related plans, and corresponding STNs, that consist of all possible permutations of choices that can be made in the TPN. For example, the TPN in Figure 15 represents two closely related plans, one corresponding to the plan in the figure when choice C1 is selected, and one corresponding to the plan in the figure when choice C2 is selected.

To search the TPN efficiently, Kirk compiles the TPN into a conditional CSP, in which the conditional variables are the choice nodes. The conditions describe the upstream relationship between choice nodes, and the constraints are simple temporal constraints. Conflict-directed candidate plan generation then corresponds to performing a conflict-directed search through assignments to the conditional CSP.

There are several conflict-directed search algorithms in the literature that are suitable for this task. Three of the most popular are Conflict-Directed Backjumping (Prosser 1993), Dynamic Backtracking (Ginsberg 1993) and Conflict-directed A* (Williams and Ragno 2002). For the purpose of evaluating ITC, we implemented Dynamic Backtracking within the Kirk planner. Dynamic Backtracking ensures a complete, systematic, and memory-bounded search, while leveraging conflicts to only generate candidate plans that resolve all known conflicts. In addition, dynamic backtracking performs dynamic variable reordering in order to preserve assignments, when possible. See (Ginsberg 1993) for the pseudocode of Dynamic Backtracking. Our implementation is a straightforward generalization of Dynamic Backtracking that is extended to handle conditional variables. After discussing related work, we consider the effectiveness of ITC at enabling continuous, temporally flexible planning, by benchmarking this implementation of Kirk on a range of structured and unstructured, randomly generated examples.

Related Work

The ITC algorithm combines the speed of shortest-path algorithms known to network optimization with the spirit of incremental algorithms such as Incremental A* and those used within truth maintenance systems (TMS).

The TPN (Kim, Williams, and Abhramson 2001) is similar to a DTP (Stergiou and Koubarakis 1998) in that a TPN allows disjunctive choice between entire subplans, and the DTP allows disjunctive choice between simple temporal constraints.

Another disjunctive temporal constraint network, the CTP, has subsequently been defined by (Tsamardinos et al. 2003) which converts conditional temporal constraint networks into a CSP, similarly to the way a TPN is converted into a conditional CSP. However, we note that solving a conditional CSP directly is often much quicker than solving its equivalent CSP representation (Gelle and Sabin 2003). Intuitively, this makes sense because a conflict-directed search strategy can reason directly on the conditional CSP's structure to efficiently prune out conditional variables. This is not possible if the conditional variables are flattened out by converting the problem into its equivalent CSP representation.

Experimentation

Overview

The Kirk planner was tested on a set of randomly generated plans, a set of realistic aerial vehicle mission plans, and also a structured plan instance that illustrates the advantage of conflict-directed search. Kirk's planning speed was compared with three search algorithm implementations:

1.) Chronological Backtracking without ITC
2.) Chronological Backtracking with ITC
3.) Dynamic Backtracking with ITC

Random TPN Generator

A random TPN generator was developed to test Kirk's performance on a wide variety of TPN plans. The random generator varies over three parameters:

1) branching factor of the parallel and choice nodes (b)
2) max-level of nested parallel and choice nodes (n)
3) number of subTPNs in sequence (s)

Figure 15 shows a typical randomly generated TPN, and the black arrows represent each parameter that can be varied to change the dimension of the generated TPN.

Each activity in the TPN was randomly assigned upper and lower time constraints. The lower time constraint for each activity was randomly selected from 1 to 6 time units with a uniform distribution, and the upper time bound was randomly selected from 5 to 10 time units with a uniform distribution.

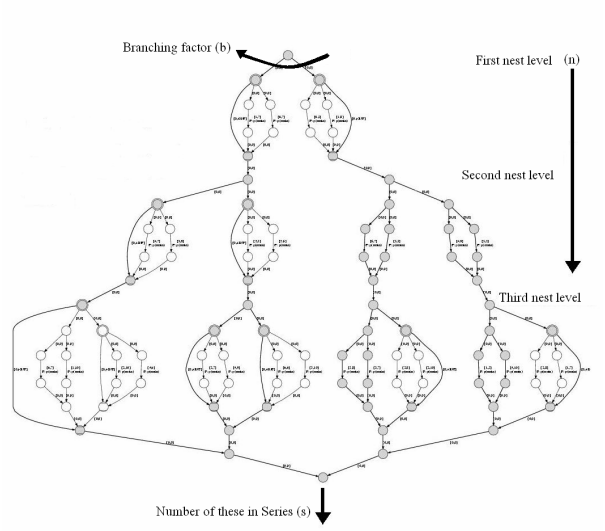


Figure 15 Randomly generated TPN

Randomly Generated TPN Test Results

For the random TPN test cases, the branching factor(b) was fixed at 2, the number of nested nodes(n) was fixed at three, and the number of subTPNs in sequence(s) was varied from 1 to 10. There were 10 choice nodes per subTPN, so this corresponds to testing TPNs that increment by 10 in the number of choice nodes for each test case.

Ten random TPNs were generated for each data point, (100 total) and for each case, the number of TPN arc updates until plan completion was counted. Plan completion corresponds to either plan success or plan failure, depending on the TPN. The results, presented in Figure 16, shows an order of magnitude improvement in

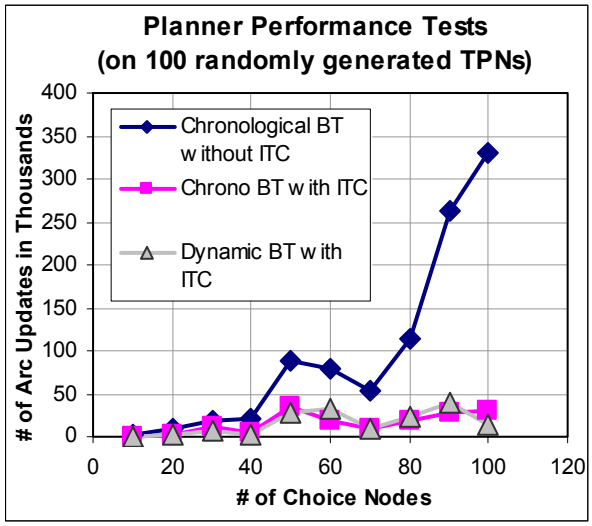


Figure 16 Performance Data on Random TPNs

planning speed with ITC vs. without ITC, as the number of choice nodes in the problem increases. Interestingly, Dynamic Backtracking with ITC shows no significant improvement over Chronological BT with ITC for these random problems. However, random problems do not offer the structure common in real world instances.

Air Vehicle Scenario Test Results

To evaluate performance on structured problems, a set of air vehicle test plans were designed specifically to test the improvement of ITC over the non-incremental planning algorithm. These plans involved multiple cooperative aerial vehicles performing a sequence of temporally consistent activities. In the scenarios, each aerial vehicle is required to image two locations but has a choice between two different sets of locations. The planner must choose one set of locations for each aerial vehicle to image. Once this choice is made, each unmanned aerial vehicle performs five activities, (1) *fly to target1*, (2) *image target1*, (3) *fly to location2*, (4) *image target2*, (5) *return to base*. In all test cases the activities are temporally consistent, so conflict-direction would not improve performance, since there are no conflicts in the plans. The graph in Figure 17 once again shows an order of

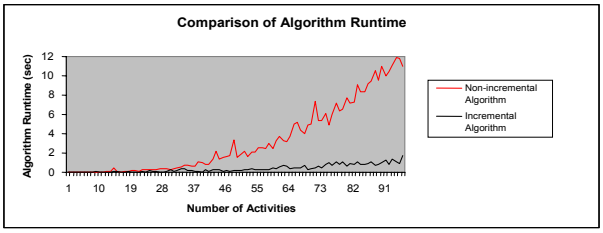


Figure 17 Runtime of Incremental versus Non-incremental Temporal Consistency Checks

magnitude improvement in runtime of ITC versus the traditional FIFO label-correcting algorithm as the number of activities is increased. These test cases illustrate ITC's ability to improve Kirk's planning speed by an order of magnitude, and to plan realistic coordinated air vehicle missions.

Structured Test to Highlight the Advantage of Dynamic Backtracking with ITC

Recall that the randomly generated test cases in Figure 20 do not indicate a clear advantage for conflict-directed

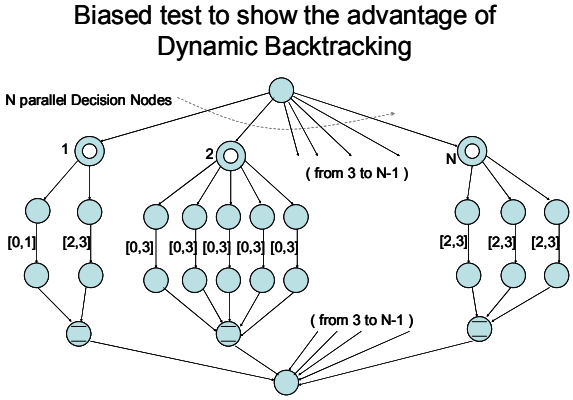


Figure 18 Structured Test to Highlight Dynamic BT with ITC

search, that is, using Dynamic Backtracking with ITC over Chronological Backtracking with ITC. However, further experiments show the key result that for many structured TPNs, such as the one in Figure 22, Dynamic Backtracking with ITC significantly outperforms Chronological Backtracking with ITC.

In Figure 18, choices are assigned in order from 1 to N, (left to right). The first activity for choice 1, has time bounds of [0,1]. This choice is consistent with the subsequent choices 2 to N-1, with time bounds of [0,3]. However, there are no choices for node N that are consistent with the the first assignment to choice 1, which has the time bounds [0,1]. Therefore, choice 1 and choice N represent a conflict in the TPN, and the only resolution is to change choice 1. When trying to resolve this inconsistency, Chronological Backtracking with ITC backtracks through half of the entire search space until choice 1 is changed to the only consistent alternative, with bounds [2,3]. Dynamic backtracking with ITC, on the other hand, can immediately utilize ITC's conflict extraction capability to determine that the conflicting choice is 1, and immediately backtrack to this inconsistent choice. The results are presented in Figure 19.

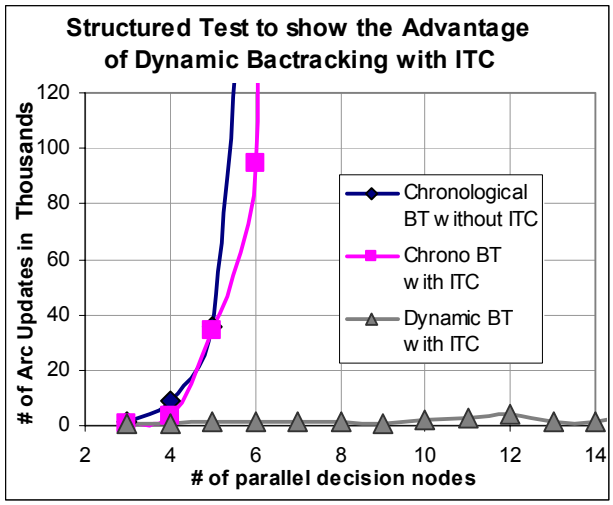


Figure 19 Structured Test Results

Discussion

The test results on both randomly generated TPNs and on realistic aerial vehicle scenarios show that ITC improves planning speed by an order of magnitude on both randomly generated TPNs and on realistic aerial vehicle scenarios. A structured test case is then presented in which the conflict-directed algorithm, Dynamic Backtracking with ITC, performs in real-time (<1sec), and both of the chronological search techniques, Chronological BT and Chronological BT with ITC, become intractable. This result suggests that search techniques without conflict-direction can run across relatively simple TPNs in which planning is intractable. Dynamic Backtracking with ITC, however, can counter this intractability by identifying conflicts and then focusing the search towards feasible regions of the search space.

The key accomplishment of this paper has been to demonstrate an order of magnitude speed improvement in temporally flexible planning through an Incremental Temporal Consistency (ITC) algorithm with incremental conflict extraction and inconsistency resolution.

Acknowledgments

This research was supported in part by the DARPA MICA program under contract N66001-01-C-8075, and the NASA IS program under contract NCC-2-1235.

The authors would also like to thank Jon Kennel, Seung Chung, and the anonymous reviewers for their excellent comments and suggestions.

References

Ahuja, R.; Magnanti T.; Orlin J, 1958. Network Flows: Theory, Algorithms, and Applications. Prentice Hall.

Cesta, A. and Oddi, A., 1996. Gaining Efficiency and Flexibility in the Simple Temporal Problem, *3rd Workshop on Temporal Representation and Reasoning*.

Dechter, R.; Meiri, I.; Pearl, J., 1991. Temporal Constraint Networks. *Artificial Intelligence*, 49:61-95.

Estlin, T.; et al., 2000. Using Continuous Planning Techniques to Coordinate Multiple Rovers. *Electronic Transactions on Artificial Inttligence,* 4:45-57.

Gelle, E. and Sabin M., 2003. Solving Methods for Conditional Constraint Satisfaction. In *IJCAI-2003.*

Gerevini, A., et al., 1996. Incremental Algorithms for Managing Temporal Constraints. *8th International Conference of Tools with Artificial Intelligence.*

Ginsberg, M.L., 1993. Dynamic backtracking. Journal of AI Research, 1:25-46.

Kim, P.; Williams, B.; and Abrahmson, M, 2001. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. *IJCAI-2001.*

Koenig, S. and Likhachev. M., 2001. Incremental A*. In *Adv. in Neural Information Processing Systems 14.*

McAllester, D., 1991. Truth Maintenance. In *Proceedings of AAAI-90*, 1109-1116.

Muscettola N., et al., 1998. Issues in temporal reasoning for autonomous control systems. In *Autonomous Agents*

Prosser, P., 1993. Hybrid algorithms for the constraint satisfaction problem, *Comp. Intelligence. 3* 268-299.

Rabideau, G., et.al., 1999. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. *ISAIRAS.*

Stergiou, K, and Koubarakis, M., 1998. Backtracking Algorithms for Disjunctions of Temporal Constraints. *15th Nat. Conference of Artificial Intelligence*, 81-117.

Tsamardinos, I., Vidal T., Pollack, M., 2003. CTP: A new constraint-based formalism for conditional, temporal planning. *Constraints.*, vol. 8, no. 4, pp. 365-388.

Williams, B. and Ragno, J., 2002. Conflict-directed A* and its role in model-based embedded systems. *Journal of Discrete Applied Math.*