Enabling Fast Flexible Planning through Incremental Temporal Reasoning

by
I-hsiang Shu

S.B. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2002

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

September 8, 2003

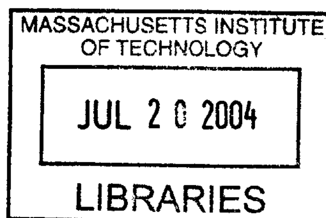
Copyright 2003 I-hsiang Shu. All rights reserved.

The author hereby grants to M.I.T. permission to reproduce and
distribute publicly paper and electronic copies of this thesis
and to grant others the right to do so.

Author ______________________________________
Department of Electrical Engineering and Computer Science
August 8, 2003

Certified by ______________________________________
Brian C. Williams
Thesis Supervisor

Accepted by ______________________________________
Arthur C. Smith
on Graduate Theses


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUL 2 0 2004
LIBRARIES

BARKER

Enabling Fast Flexible Planning through Incremental Temporal Reasoning

by
I-hsiang Shu

Submitted to the
Department of Electrical Engineering and Computer Science

September 8, 2003

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In order for a team of autonomous agents to successfully complete its mission, the agents must be able to quickly re-plan on the fly as unforeseen events arise in the environment. This requires temporally flexible plans that allow the agent to adapt to execution uncertainties by not overcommitting on time constraints, and a continuous planner that replans 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.

This thesis provides a fast incremental algorithm, ITC, for determining the temporal consistency of temporally flexible plans. Additionally, the temporal reasoning capability of ITC is able to return the conflict or the nature of the inconsistency to the planner, such that the planner can resolve inconsistencies quickly and intelligently. 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 algorithm has been implemented and integrated into a temporal planner, called Kirk. It has demonstrated an order of magnitude speed increase on cooperative air vehicle scenarios.

Thesis Supervisor: Brian C Williams
Title: Associate Professor of Aeronautics and Astronautics

Acknowledgements

I would like to thank my mom, Sue-yun Shu, my dad, Paul yet-Chao Shu, my brother, I-Wei Shu, and my grandma Mei-yu Lin for all their support during my years at MIT.

I also would like to thank my advisor, Brian Williams, for his knowledge and guidance through which this thesis would not have been possible.

Thanks especially to Jon Kennell, Raj Krishnan, and Sean Lie for giving me the algorithms background that I needed.

Thanks to everyone at the Model-based Embedded Robotic Systems group for being the coolest research group at MIT.

Thanks to Margaret Yoon for her incredible administrative assistant skills and her ability to get all of us free lunches.

Thanks to Highway "Joe" Cattell for thanking me in his thinking. Thanks to John Mcbean in the hopes that this will get him to thank me in his thesis. Thanks to Ahmed Elmouelhi because I am sure he did not thank me in his thesis.

Table of Contents

Chapter 1 .... *10*

Introduction **10**

1.1 Motivation 10
1.2 Preliminary Problem Statement 11
1.3 Mission to the Goal 11
1.4 Enabling Continuous Temporally Flexible Planning 12
1.4.1 Temporal Consistency Requirement 12
1.4.2 Achieving Continuous Temporally Flexible Planning 13
1.5 Problem Statement 13
1.6 Approach 13
1.6.1 Fast Temporal Consistency with Incremental Methods 14
1.6.2 Continuous Temporally Flexible Planning through Conflict Extraction 15
1.6.3 Solution 16
1.4 Thesis Outline 16

Chapter 2 .... *17*

Temporal Consistency in Flexible Temporal Planning **17**

2.1 RMPL – Reactive Model-based Programming Language 17
2.2 Temporal Networks 18
2.2.1 Simple Temporal Network (STN) 18
2.2.2 Temporal Plan Network (TPN) 19
2.2.3 Example TPN for Soccer Example 20
2.2.4 Example Candidate STN for Soccer Scenario 21
2.3 Kirk – A Temporally Flexible Planner 22
2.4 Temporal Consistency and Candidate Plan Generation 24
2.5 Incremental Temporal Consistency on Candidate STNs 25

Chapter 3 .... *27*

Temporal Consistency Checking Algorithms **27**

3.1 Determining Temporal Consistency of an STNs 27
3.1.1 An STN and its Distance Graph 27
3.1.2 Detecting Temporal Inconsistency through Negative Cycle Detection 29
3.2 Negative Cycle Detection Algorithms 29
3.2.1 All-pairs Shortest-path Algorithms 30
3.2.2 Single-source Shortest-Path Algorithms 30
3.2.3 Modified Label-Correcting Algorithm 37
3.3 Temporal Consistency of "Mission to the Goal" Scenario 39

Chapter 4 .... *42*

The Incremental Temporal Consistency Algorithm (ITC) **42**

4.1 The ITC Algorithm Overview 43
4.2 Insufficiency of Modified-label Correcting to Perform ITC 44
4.3 Truth-Maintenance Systems and Unit Propagation 45
4.3.1 LTMS 45
4.3.2 Unit Propagation and Support 45
4.3.3 Clause Deletion and Unsupport 46
4.3.4 Incremental Ideas from LTMS 47
4.4 ITC Algorithm's Incremental Update Rules 48
4.4.1 Arc Change without Affect to Shortest-Path 49
4.4.2 Arc Change Improves Shortest-Path 50
4.4.3 Arc Change Invalidates Shortest-Path 52
4.4.4 Addition and Removal Arcs 53
4.5 Incremental Temporal Consistency Algorithm Pseudo-Code 54

4.6 Negative Cycle Detection with Conflict Extraction 57
4.7 Inconsistency Resolution 59
4.8 Algorithm Analysis 60
4.9 ITC Algorithm on "Mission to the Goal" 61

Chapter 5........ 65

Discussion 65
5.1 Implementation........ 65
5.2 Performance........ 65
5.3 Future Work........ 68
5.3.1 ITC Implementation Work........ 68
5.3.1 ITC Evaluation Work........ 68
5.3.1 ITC Improvement Work 68
5.3 Conclusion........ 69

References........ 70

Chapter 1
Introduction

1.1 Motivation

Autonomous robots and vehicles are quickly becoming an integral part of modern society. These autonomous agents have long been building and assembling our automobiles and some are even beginning to perform more everyday tasks, such as mowing our lawns and vacuuming our floors. In the future, these agents will perform even more complex tasks, such as exploring and analyzing the Martian landscape and flying unmanned aerial vehicle (UAV) 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* [5]. Continuous planners, such as ASPEN [10], developed by JPL, are capable of quickly generating time critical maneuvers given that a change in environment breaks the current mission plan. For example, excessive temperatures can cause hardware to begin to fail. A new plan needs to be devised quickly to prevent further damage to that hardware component. A downside to these continuous planners is that they do not allow for temporal flexibility, as they assign hard execution times to activities. For example, these planners schedule a fixed time in the plan for charging batteries. However, if the charging process actually takes a little longer because of a decreased charging rate, then this plan would break because activities scheduled to be performed later in the plan would not be able to start on time. Temporally flexible planners, such as HSTS [16], allow these smaller perturbations to not break the entire plan. These planners impose temporal constraints that guarantee a plans success, but delay assigning fixed times to activities until execution. The temporal constraints only constrain activities to a minimum and maximum duration. Least commitment gives a temporally flexible planner the ability to adapt to unknown environments and execution uncertainties during plan

execution and scheduling because some slack is allowed for when activities must occur.

1.2 Preliminary Problem Statement

For an autonomous agent to be robust to minor uncertainties such as time delays and also operate in dynamic environments where plans can fail, a continuous temporally flexible planner must be enabled.

1.3 Mission to the Goal

To motivate the need and use for temporally flexible continuous planning, we introduce a soccer scenario involving autonomous robotic players. In addition, we will return to this example later in the thesis, to illustrate the representations and algorithms necessary for determining temporal consistency.

Teams of autonomous robots have already demonstrated their versatility by playing games of soccer against other teams of autonomous robots. These robotic soccer teams are able to plan optimal strategies based on models of the opponents [8]. Consider a scenario in which two Blue autonomous robotic soccer players are on a 2-on-1 breakaway ready to attack the Red goal, see Figure 1.

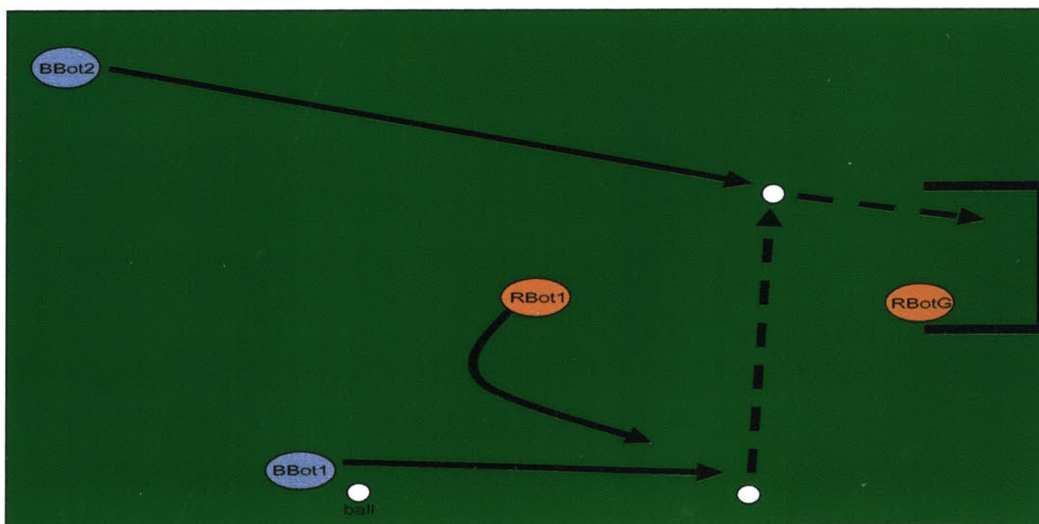


Figure 1 – Breakaway 2-on-1

In this scenario, there are two Blue robots, *Blue1* and *Blue2*, and two Red robots, *Red1* and *RedG*. *RedG* is the goalie for the Red team. Figure 1 shows the desired paths of the autonomous players with solid black lines and the paths of the ball in dashed lines.

Blue1 attempts to drive the ball towards the goal, but *Red1* challenges *Blue1* and forces *Blue1* to dribble the ball towards the sideline away from the goal. *Blue1* anticipates this action by *Red1* and tries to race towards the corner, beating *Red1,* and then centering the ball to *Blue2*. *Blue2* receives the centering pass and shoots the ball into the goal for the score.

This is a common soccer strategy, however to be successful it is essential that the robotic soccer players coordinate properly so that *Blue2* can take the shot. In addition, suppose that *Red1* decides not to decisively challenge *Blue1* and instead defend a little closer towards *Blue2* in order to try and prevent the quick centering pass. The Blue team must change its plan of attack in order to compensate for this change in Red's strategy. The planner must be quick and agile enough to continuously plan and recover from this change.

1.4 Enabling Continuous Temporally Flexible Planning

1.4.1 Temporal Consistency Requirement

A key task that must be performed by a temporally flexible continuous planner is to evaluate a candidate plan do determine whether or not an autonomous agent has sufficient time to complete all of the assigned activities, given the timing constraints. This is referred to as the temporal consistency of a plan. For example, an autonomous rover begins work at 9am and needs to sample and analyze a deep layer of the Martian soil before sunset at 7pm. The activities of drilling and analyzing together are constrained to last no more than 11 hours, giving the rover a flexible window to perform the activities. However, if it takes at least 12 hours to drill for samples, the rover will inevitably fail its mission, since the constraint on drilling time conflicts with the constraint of finishing before sunset. The requested plan is temporally inconsistent.

1.4.2 Achieving Continuous Temporally Flexible Planning

All planners, put simply, generate a plan and then test this plan for validity before it is executed. This allows for two ways to support continuous planning with temporal flexibility. Increase the speed of the testing phase by speeding up the temporal consistency checking algorithm and increase the speed of the candidate plan selection process by identifying the subset of temporal constraints that lead to temporal inconsistency. Knowledge of these inconsistent temporal constraints can then be used by the plan generator to intelligently select the next candidate plan.

1.5 Problem Statement

To enable a planner to be continuous and temporally flexible, this thesis will create a fast temporal consistency algorithm with conflict extraction.

1.6 Approach

Our approach to enabling a continuous temporally flexible planner is developed in the context of the Kirk temporally flexible planner. In this section we present Kirk and analyze what is needed to make Kirk fast.

The planning process for a temporally flexible planner contains four basic phases, as shown in Figure 2. First, high level goals are specified. Second, candidate plans are chosen. Third, the candidate plan is verified and checked for consistency. Steps 2 and 3 are repeated until a consistent plan is found. Finally, the consistent plan is passed down to the plan executive for execution.

Temporal planners repeatedly ask whether candidate plans are temporally consistent as they search to find a feasible plan. As shown in Figure 2, plan inconsistencies and execution failures, common to dynamic environments, require numerous iterations through the plan selection and plan verification phases. Therefore, optimizing the algorithm that performs temporal consistency checking in the plan verification phase and focusing the ability of the plan selection phase to choose consistent plans would significantly improve the performance of the planning process.

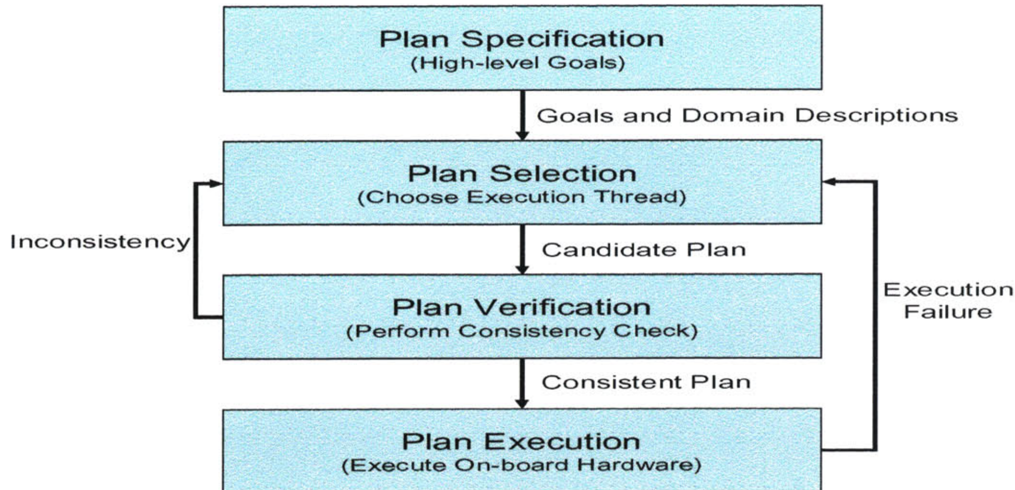


Figure 2 – Planning Process

1.6.1 Fast Temporal Consistency with Incremental Methods

The speed of the temporal consistency checking algorithm can be significantly increased by using incremental methods to remember previous work that need not be recomputed. Since all plans chosen in the plan selection phase in Figure 2 are derived from the same set of high-level goals in the plan specification phase, the candidate plan involved in successive queries to plan verification differs only incrementally. Hence, It is not necessary to start the temporal consistency check from scratch, but to only check constraints that differ from the previous candidate. To achieve this, we monitor the difference from candidate plan to candidate plan and check the temporal consistency by computing only from the differences. Figure 3 below shows the modified planning process with incremental temporal consistency checking.

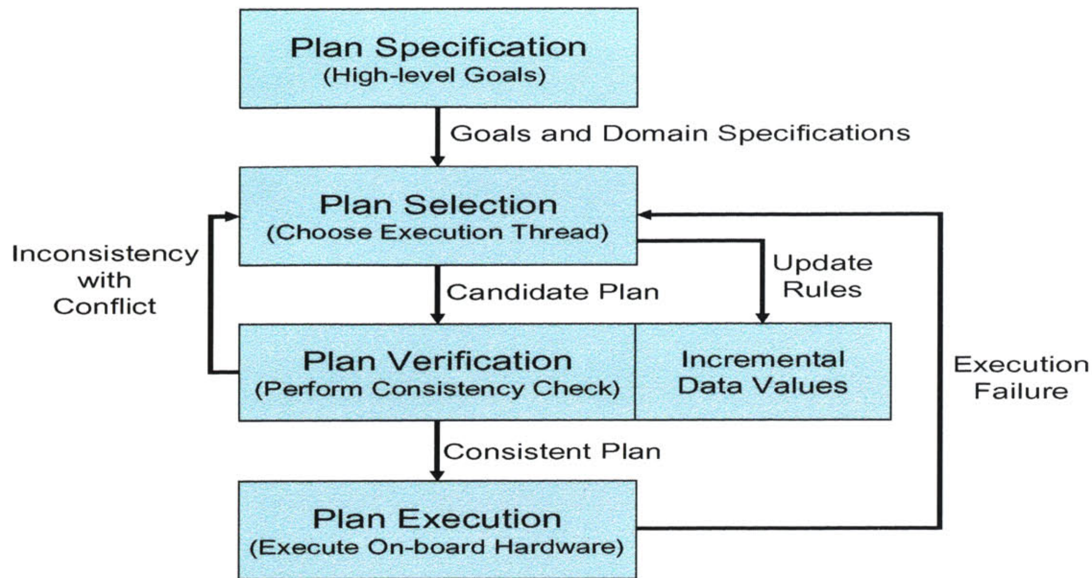


Figure 3 – Planning Process with Incremental Temporal Consistency

The additional module in Figure 3 is added to store the data values calculated to determine temporal consistency by the plan verification phase. When a new plan is selected in the plan selection stage, the algorithm monitors the parts of the plan network that has changed from the previous candidate plan and then modifies the incremental data values with update rules. The update rules guide the incremental temporal consistency algorithm to perform less work.

1.6.2 Continuous Temporally Flexible Planning through Conflict Extraction

A temporally flexible planner can increase the speed in which candidate plans are found through the use of *conflict extraction*. A conflict is a set of temporal constraints that force a candidate plan to be temporally inconsistent. The conflict can be used to guide the plan selection of successive plans. These plans resolve, or do not contain these conflicts, and consequently are more likely to be verified as consistent by the temporal consistency algorithm. Focused search over the plan space reduces the number of iterations through the generate and test loop of the planning process and thus helps enable continuous temporally flexible planning.

1.6.3 Solution

In this thesis, we introduce and explain a fast incremental algorithm for checking temporal consistency in order to support continuous temporally flexible continuous planning, called ITC (Incremental Temporal Consistency). It uses modifications of a fast shortest-path algorithm from network optimization, FIFO label-correcting algorithm, and incremental update rules in the spirit of incremental search algorithms such as Incremental A* [8] and TMS [4] to accelerate the temporal reasoning process. Additionally, if a candidate plan is inconsistent, ITC uses a built in conflict extraction mechanism to return temporal constraints responsible for the inconsistency. This guides the plan generation phase to resolve or return candidate plans without these conflicts, ultimately increasing overall planning speed.

1.4 Thesis Outline

The following chapters first give a background of a temporal planner, Kirk, capable of planning with temporal flexibility and then explain how temporal consistency is determined within this planner. Next, an overview of the general approach of how to determine temporal consistency is given. In Chapter 4, the ITC algorithm is introduced and demonstrated. Chapter 5 shows the experimental results of the speed improvements of temporal planning using this incremental method. It also summarizes the thesis and suggests ideas for future work.

Chapter 2

Temporal Consistency in Flexible Temporal Planning

This thesis introduces an incremental temporal consistency algorithm for temporally flexible planners. In order to understand how temporal consistency fits into temporally flexible planners, we provide an overview of Kirk [7] and the basic structures and algorithms it uses for temporally flexible planning.

Kirk takes as input a high-level goal specification program written in RMPL, converts this program to a Temporal Plan Network (TPN) which represents possible threads of execution, selects threads of execution from the TPN, resolves symbolic constraints, and finally takes this resulting Simple Temporal Network (STN) and executes it on low-level hardware. The subsequent sections explain the Kirk planning structures and how they fold into Kirk planning.

2.1 RMPL – Reactive Model-based Programming Language

As input, Kirk takes an RMPL program specifying high-level goals. RMPL includes constructs that allow mission designers to express maintenance conditions, concurrency, synchronization, metric constraints, and contingencies when creating plans for autonomous robotic teams. An RMPL program written to control the two Blue robotic soccer players from the example soccer scenario in Section 1.3 would look as shown in Figure 4.

```
Score-Goal()
      (parallel
            (sequence
                  (BBot1.goto(corner)[1,8])
                  (choose
                        (BBot1.centeringpass-low()[2,2])
                        (BBot1.centeringpass-high()[9,9])))
            (sequence
                  (BBot2.goto(goal)[1,5])
                  (BBot2.wait()[0,5])))
      (BBot2.shoot()[1,1])
```

Figure 4 – RMPL Program for Soccer Scenario

This example RMPL program shows two concurrent sequences of activities for the mission *Score-Goal()*, one for *Blue1* and one for *Blue2*. Each sequence has a series of activities that needs to be successfully completed in order for the Blue team to score the goal. Every activity has time bounds associated with it, specified in brackets, *[l,u]*. A time bound constrains a particular activity to last at least *l* time units and at most *u* time units. The *parallel* RMPL construct constrains the two sequences of activities for *Blue1* and *Blue2* to start and finish at the same time. This eventual synchronization of *Blue1* and *Blue2*'s activity threads means that *Blue2* must finish waiting for the ball as soon as the centering pass reaches the front of the goal. The *choose* RMPL construct allows the planner to make a non-deterministic choice between two alternative sets of activities. In this example, *Blue1* has the option of kicking either a low centering pass or a high centering pass to *Blue2*. Additional details about RMPL and the supported constructs can be found in [5].

2.2 Temporal Networks

Kirk converts the RMPL program into a graph, called a Temporal Plan Network (TPN) that represents the possible threads of execution and timing constraints between activities. It uses the TPN in order to select threads of execution (what Kirk considers planning), to check the execution feasibility, and finally to schedule activities to be executed on the fly. Timing constraints are represented as a simple temporal network (STN), defined in Section 2.2.1. These pre-compiled graph structures allow for a compact and easily understandable representation of the plan, support temporal flexibility, and allow for fast and easy search through the space of possible plans. The objective of this thesis is to support efficient temporal reasoning and to enable continuous, but temporally flexible planning. To accomplish this, we will focus on fast algorithms for reasoning on STNs.

2.2.1 Simple Temporal Network (STN)

An STN has three basic components, nodes, arcs, and binary time constraints. Each node represents a point in time, such as beginning to turn the car key when starting a car. An arc represents the existence of a time constraint

between two nodes, where the head of the arc represents a timepoint and is later in time than the node at the tail of the arc. Binary time constraints are enclosed within brackets, *[l,u]*, similar to the RMPL example, and are shown above the arc. These represent the absolute lower and upper bounds of the duration between two timepoints. For an activity *engine-start()[1,5]*, the STN structure would look as shown in Figure 5.

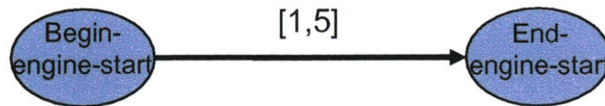


Figure 5 – Example STN

Based on the time constraint on the activity, the duration between the two timepoints *Begin-engine-start* and *End-engine-start* must be less than or equal to 5 time units and greater than or equal to 1 time unit.

2.2.2 Temporal Plan Network (TPN)

A TPN extends an STN by adding decision nodes as well as symbolic constraints.

Decision nodes allow TPNs to represent the multiple feasible threads of execution specified in an RMPL program. The planner must select the best path as it is trying to determine a consistent plan of execution. For example, if a Mars exploration rover has an option to explore a mountainous region or a flatland region, based on the constraints on the agent, the planner must choose a path that will be executable. The TPN will represent this as a decision node branching to a sequence of mountainous activities and a sequence of flatland activities.

A symbolic constraint is used within a mission plan to express conditions that must be true in order for an activity to be executed. For example, in order for a UAV to attack a target, the condition must be true that the munitions are armed. Thus, the planner for the UAV must determine that it needs to arm the missiles before the missiles can be launched. This is specified within an RMPL program and in a corresponding TPN by an *Ask(condition)* and a complementary *Tell(condition)*. These *Ask* and *Tell* conditions are attached to the arcs of a TPN,

which constrains each *Ask* and *Tell* to have the duration specified by the arc. An example TPN is shown in Figure 6.

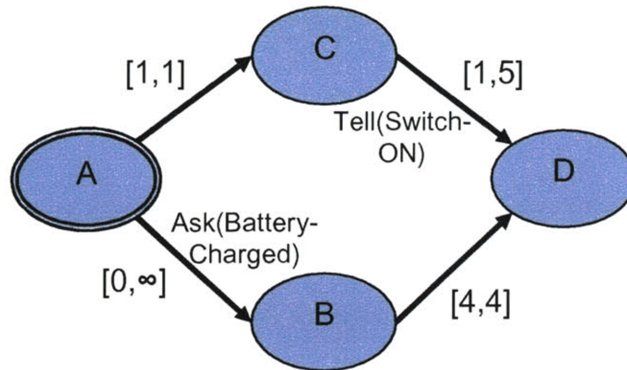


Figure 6 – Example TPN

In this TPN example, node *A* is a decision node, specified by the double lines forming the oval. The planner has the option of either choosing to take path *ACD* or path *ABD,* but not both. Arcs *CD* and *AB* have symbolic constraints associated with them. The symbolic constraint on arc *AB* specifies that during the time between timepoint *A* and timepoint *B,* we must have the condition *Battery-Charged* for some amount of time. The symbolic constraint on arc *CD* specifies that during the time between timepoint *C* and timepoint *D* we will assert the condition *Switch-ON* for at least 1 time unit, but for no more than 5 time units.

2.2.3 Example TPN for Soccer Example

The TPN for the soccer scenario described in Section 1.3 of the Introduction is shown in Figure 7. The TPN for this scenario does not contain any symbolic constraints; however, it does contain a decision node specifying that *Blue1* needs to choose the type of centering pass it will kick.

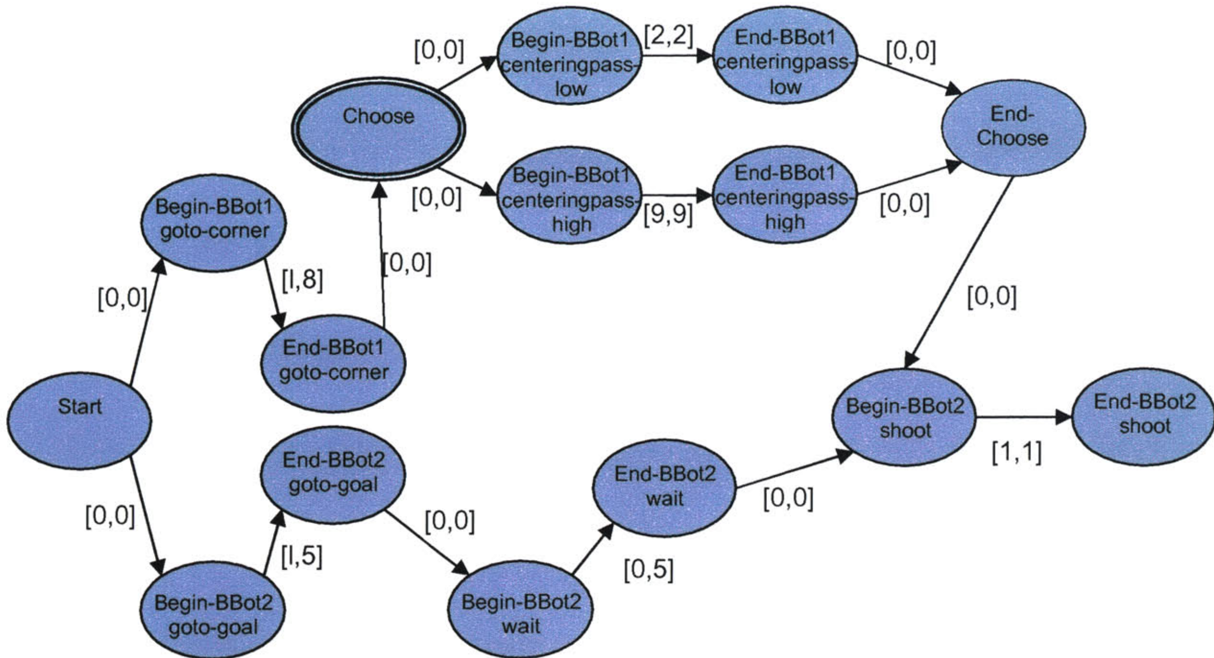


Figure 7 – TPN of Soccer Scenario

In this TPN, the *Start* node is not a decision node; hence the two parallel sequences emanating from it must be selected for execution and started simultaneously. However, the node labeled *Choose* is a decision node, designated by the double lines. For this scenario, the planner must choose one of the two threads emanating from the *Choose* node, corresponding to the two types of centering passes that *Blue1* can kick. The graph also contains many *[0,0]* timing constraints. These *[0,0]* constraints mean that the next timepoint following the constraint happens instantaneously after its predecessor. For example, as soon as *Blue1* is near the corner, it will immediately kick the ball to be centered in front of the goal. The goal state is the last node in the graph, since it has no outarcs, signifying the end of the plan. In this example, the goal state is the node containing the event *End-Blue2-shoot*.

2.2.4 Example Candidate STN for Soccer Scenario

It shows an example of the data structure that Kirk generates as candidate plans. From the TPN in Figure 7, since there are no symbolic constraints, we just need to ensure that decisions have been made. For this soccer scenario, we will have Kirk make the decision for *Blue1* to kick a low centering pass. The

resulting candidate plan is shown in Figure 8, with the selected nodes outlined in bold.

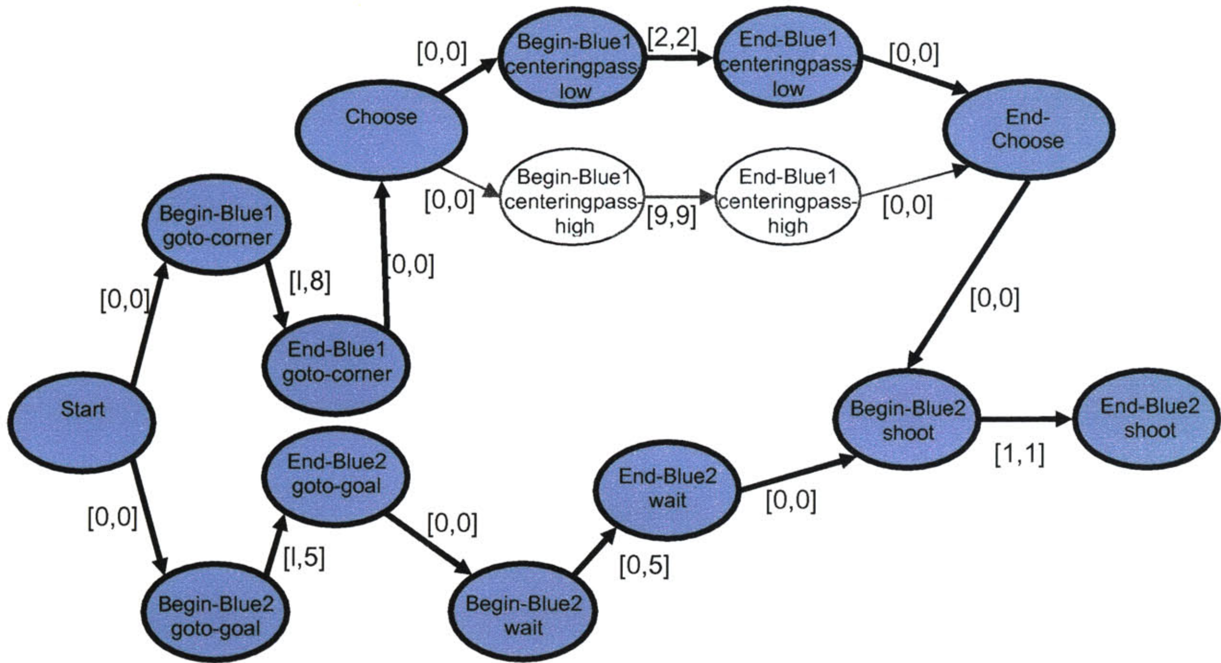


Figure 8 – Soccer Scenario Candidate STN

2.3 Kirk – A Temporally Flexible Planner

Now that we have described the basic representations manipulated by Kirk, this section explains how Kirk uses these representations for planning, with particular focus on how temporal reasoning interacts with plan generation. The basic Kirk Planning architecture is shown in Figure 9.

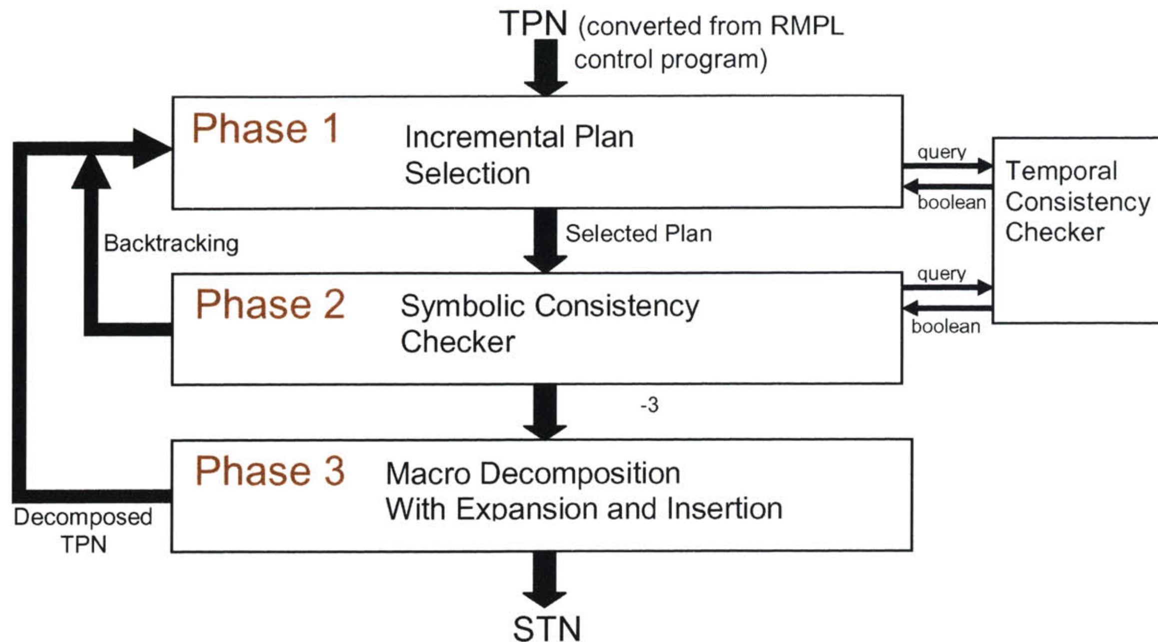


Figure 9 – Kirk Planning Architecture

In Phase 1 of Kirk planning, a TPN, generated from a high-level RMPL program specifying mission objectives, is searched in order to find a temporally consistent plan. In Phase 2, Kirk resolves symbolic constraints and ensures this resolution is also temporally consistent. In the final phase, Kirk decomposes high-level macro activities into lower-level primitive activities.

The backtracking arrow leading from Phase 2 to Phase 1, allows Kirk to make new decisions for a new candidate plan if the current plan is found to be inconsistent, either temporally or symbolically. The arrow leading from Phase 3 to Phase 1 allows Kirk to select and examine the new nodes that have been introduced into the network from the decomposition step. The final plan output of Kirk has all decision nodes and symbolic constraints resolved and is executed in the plan runner as described in [13].

Recall that this thesis focuses on developing an incremental temporal reasoning capability for temporally flexible planning, thus we are concerned primarily with Phase 1 of Kirk planning. This phase is where Kirk creates candidate plans, constantly requesting a temporal consistency check, and thus this phase can benefit the most from a fast incremental temporal consistency algorithm. This is explained in the Section 2.4 and 2.5. Additional, details

regarding the algorithms found in the other phases of Kirk TPN planning not discussed in this thesis can be found in [5].

2.4 Temporal Consistency and Candidate Plan Generation

In order to understand, how an incremental temporal consistency algorithm can be incorporated into Kirk TPN planning, we must first understand how temporal consistency is integrated into Kirk plan generation. This section describes exactly how Kirk selects a candidate plan and ensures its temporal consistency.

Phase 1 of Kirk TPN planning chooses a candidate plan by searching through the TPN graph. Beginning with the start node of the graph, the algorithm checks whether the node is a decision or non-decision node. If it is a non-decision node, the algorithm simply extends the plan to the head node of all outgoing arcs and adds these nodes to the set of nodes to be examined later. If the node is a decision node, then the algorithm decides on one particular outgoing arc and extends the mission plan to the head of this arc. The algorithm terminates when there are no more nodes left to expand, meaning that all branching paths have reached the final goal node. The candidate plan is then passed on to the next phase.

As the candidate plan is built up node by node and arc by arc, it must be checked for temporal consistency. Temporal consistency means that there exists an assignment of times to each timepoint in the temporal network such that all of the temporal constraints are satisfied. For example, in the STN shown in Figure 10, we would like our Mars exploration rover to take a picture of the sunset as seen from Mars.

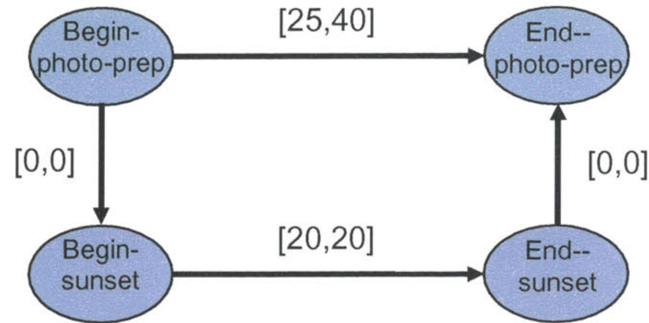


Figure 10 – STN for photographing Martian sunset

We know that the sunset on Mars lasts for exactly 20 minutes, however it takes the rover at least 25 minutes to drive to a location where it can prepare the camera and take the picture. Thus we know that if we try to execute the STN shown in Figure 10, where the rover begins preparations for the photo at the same time the sunset begins, the plan is guaranteed to fail. This is called a *temporal inconsistency* since it is impossible for the rover to take the photograph of the sunset, given the requirement that it must start preparation as soon as the rover sees the Martian sunset beginning.

Within Phase 1 of Kirk, candidate plans are checked for temporal consistency every time two paths in the search converge, indicating that there is a synchronization in the plan. This ensures that there will exist a possible assignment of times to timepoints such that an autonomous agent will be able to complete its mission task. If Kirk finds a partial candidate plan to be temporally inconsistent, it will backtrack, select a different branch at a decision node and test this new partial candidate for consistency. It is useless to continue to build up a partial candidate plan that is inconsistent, because the candidate will remain inconsistent, no matter how many nodes or arcs are added.

2.5 Incremental Temporal Consistency on Candidate STNs

As described in the previous section, candidate plans are tested for temporal consistency as a plan is built up, every time two paths converge. Because these candidates are incrementally constructed, successive candidate plans are very similar to the preceding candidate, often differing only by a few

nodes and arcs. The new or changed nodes and arcs typically comprise a small percentage of the overall candidate plan. This suggests that when checking for temporal consistency, it is unnecessary to check the entire candidate starting from scratch. The consistency of successive plans can be determined by storing the data values used to calculate the consistency of the previous graph, by analyzing how the new candidate differs from the previous candidate and by updating only those affected data values. This mechanism is in the spirit of a range of incremental algorithms such as Incremental A* [8], where the best start distance at each node is carried over from search to search, and truth maintenance systems where logical consequences are carried over, as clauses are added and removed from the propositional theory [4].

An incremental temporal consistency algorithm that returns the minimum set of constraints that result in the temporal inconsistency can also speed up candidate plan generation. The plan generator can be much more focused in finding a temporally consistent plan by making use of *conflicts*, the minimum subset of constraints that lead to inconsistency. If every time the incremental algorithm discovers an inconsistency, then the planner is capable of focusing its plan generation by not choosing plans containing the conflict. This strategy can further improve the performance of the incremental algorithm used to speed up Phase 1 of Kirk planning.

Chapter 3

Temporal Consistency Checking Algorithms

For enabling continuous temporal planning, the planner needs a fast algorithm to test for temporal consistency. This chapter begins by describing exactly how temporal consistency is determined. It then introduces two groups of algorithms that can be used to check for consistency; specifically, all-pairs shortest-path algorithms (APSP) and single-source shortest-path algorithms (SSSP). The chapter gives a more in-depth treatment of the SSSP label-correcting algorithm since the incremental temporal consistency algorithm, developed in Chapter 4, is based on this SSSP algorithm. The subsequent chapter discusses how this algorithm is generalized to perform incremental temporal consistency checking.

3.1 Determining Temporal Consistency of an 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 [3].

3.1.1 An STN and its Distance Graph

An STN and its distance graph have the same nodes. An STN is converted to a distance graph 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 11 shows this conversion

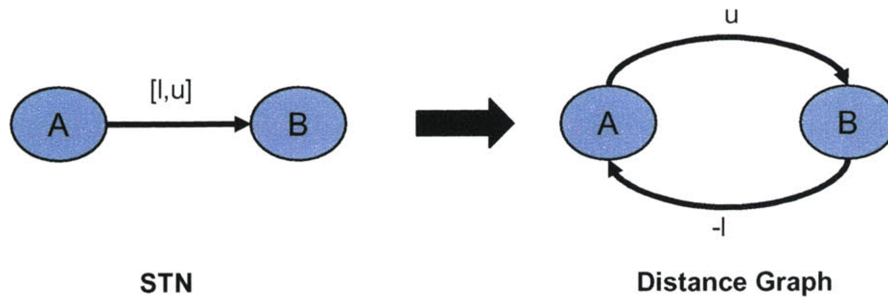


Figure 11 – STN to Distance Graph Conversion

The nodes in a distance graph represent timepoints just like in an STN. The arcs in the distance graph, however, correspond to an upper bound on the distance between the two timepoints. For every arc, the difference in time between the timepoint at the head of the arc and the timepoint at the tail of the arc must differ by a value less than or equal to the distance on that arc. The equation below shows specifies how each timepoint constraint for an STN is converted to a constraint for the distance graph for an arbitrary arc_{ij}.

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

As an example, in Figure 11, 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*.

Figure 12 shows the distance graph of the soccer scenario candidate STN, given in Section 2.6.

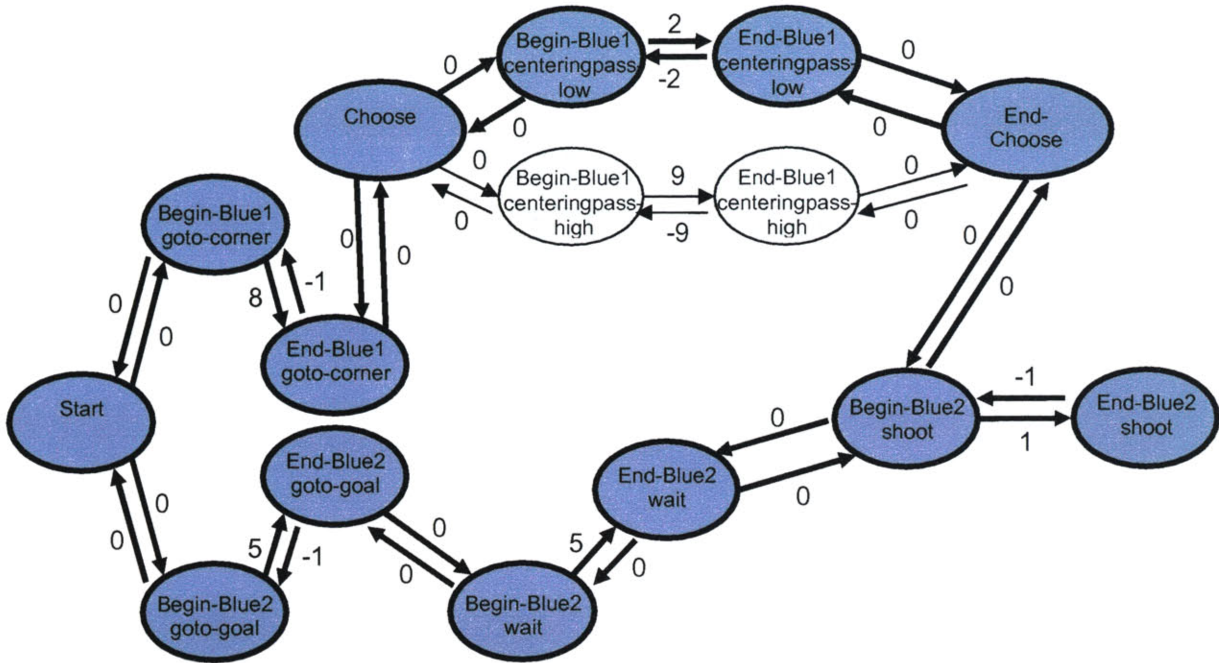


Figure 12 – Distance Graph of Soccer Scenario

3.1.2 Detecting Temporal Inconsistency through Negative Cycle Detection

As mentioned above, 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 [3]. Intuitively, since the edge weights in the distance graph represent the amount of time that an event must happen before another event (i.e. 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 (i.e. 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.

3.2 Negative Cycle Detection Algorithms

Several algorithms exist for detecting negative cycles in graphs that contain negative edges. Many of these methods are applied to network optimization problems in which it is possible that, as an arc is traversed, some of the cost that has already been accumulated can be regained or decreased. In this section, we review two classes of negative cycle detection algorithms, All-

Pairs Shortest-Path (APSP) and Single-source Shortest-Path (SSSP). A more thorough treatment is given to the SSSP label-correcting algorithm, since it is the basis for the incremental temporal consistency checking algorithm contributed by this thesis.

3.2.1 All-pairs Shortest-path Algorithms

An all-pairs shortest-path algorithm returns the shortest-path from *u* to *v* for every pair of nodes *u* and *v* in a graph. This information can be represented in the form of an N by N matrix, where N is the number of nodes in the graph and each element a_{ij} in the matrix represents the shortest path from node *i* to node *j*. Figure 13 shows on the right the matrix that is returned when an APSP algorithm is run on the distance graph shown on the left..

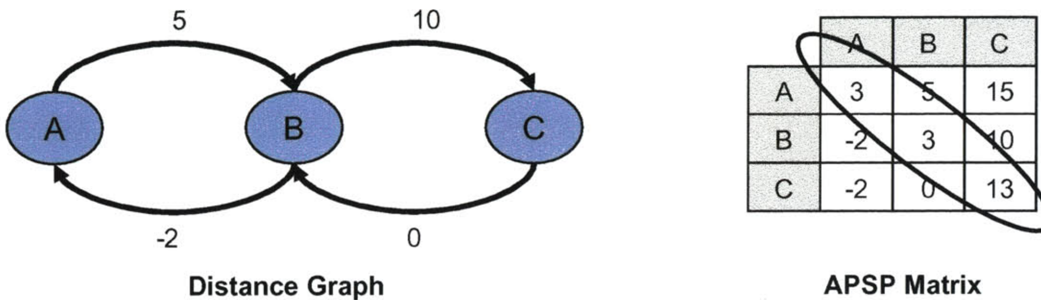


Figure 13 – APSP Example

For an APSP algorithm, a negative cycle is detected if a diagonal element of the APSP matrix, a_{ii}, is less than zero. In the example shown in Figure 13, we see that there are not any negative values in the diagonal elements of the matrix, and consequently, this graph is temporally consistent. Thus, to determine temporal consistency, any APSP algorithm can be run, such as Floyd-Warshall's algorithm, and the resulting APSP matrix can be scanned for negative diagonal elements. A detailed presentation of Floyd-Warshall's algorithm and other APSP algorithms can be found in [2].

3.2.2 Single-source Shortest-Path 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, SSSP. The reason only a SSSP needs to be performed is because 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 is because a shortest-path can continually loop along the negative cycle, reducing path distance indefinitely.

Using only a SSSP algorithm offers significant saving over an APSP algorithm 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 given in Section 3.2.3, 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. Before fully introducing the FIFO label-correcting algorithm, this section first discusses the generic label-correcting algorithm and then gives insight into a modified label-correcting algorithm.

Generic Label-Correcting Algorithm

The basic pseudo-code for the generic label-correcting algorithm is shown below.

Definitions:

d(i): the best known start distance or the temporary distance from the start node to node *i* before termination of the algorithm.

$d^*(i)$: the true shortest path distance from the start node to node *i*.

V(G): the nodes of graph G.

Violating arc: any arc(i,j) where d(j) > d(i) + c(i,j)

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

{01} for all s ∈ V(G)
{02}    d(s) = ∞
{03} d(s_start) = 0
{04} while some arc(i,j) is
     violating,
{05}    d(j) = d(i) + c(i,j)
```

Figure 14 – Pseudo Code fore Generic Label-Correcting Algorithm

The generic label-correcting algorithm computes an upper bound on the shortest-path distances and then iteratively tightens these bounds [1]. The generic label-correcting algorithm is based on the concept of *violating arcs*. A violating arc is an arc(i,j) that has d(j) > d(i) + c(i,j) and identifies to the algorithm where these shortest-path distance may be updated. If there are not any violating arcs, then the algorithm is finished and $d(i) = d^*(i)$, meaning that we have found the shortest-path from the start node to all other nodes.

Looking at the code, the generic label-correcting algorithm starts off by first initializing all start distances to the largest upper bound possible, ∞, since at the start, it is unknown what the path length to each node is (lines `{01-02}`). The start distance for the start node is then initialized to 0, since the best distance from the start to itself must be 0 (line `{03}`). In the iterative step, lines `{04-05}`, the algorithm continually updates the start distances for the nodes at the head of violating arcs until there are no longer any violating arcs. When the loop exits, the algorithm is finished and the shortest-paths have been found.

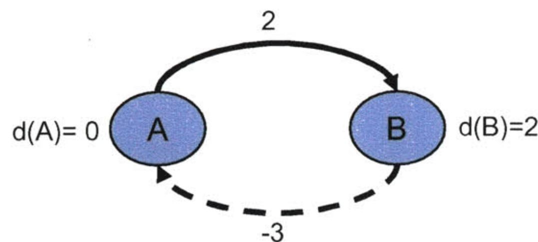


Figure 15 – Simple Example of the Generic Label-Correcting Algorithm

The example in Figure 15 shows the first iteration step of the generic label-correcting algorithm on a simple distance graph. The start node, *A*, has an initial start distance of 0. This start distance is then propagated along the first violating arc, *AB*, and therefore updating the start distance value at *B* to be 2. Figure 15 shows the snapshot of the algorithm at this point, after one update step. The dashed line shows which arcs are violating and still need updating.

Termination Conditions

The generic label-correcting algorithm terminates when no more violating arcs exist. If the distance graph contains a negative cycle, the algorithm will never terminate. Instead, it will continuously update the nodes on violating arcs forever since the absolute shortest-path for a path containing a negative cycle is $-\infty$

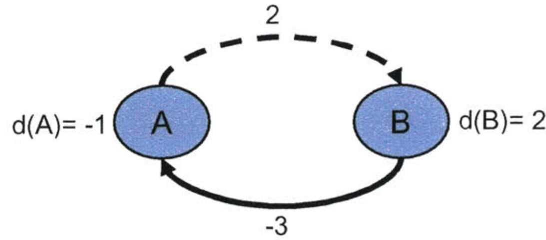


Figure 16 – Violating Arcs from Distance Graph with Negative Cycle

For example, in the above simple distance graph, Figure 16, currently arc *AB* is violating. If we update arc *d(B)* to equal 1, d(A) + c(arc_{AB}), then arc *BA* will then become violating. The algorithm will then update *d(A)* to equal -2. Now, again arc *AB* is violating. Therefore, since there exists a negative cycle, either arc *AB* or arc *BA* will always be violating arc and the generic label-correcting algorithm will continually update the start distance values of nodes *A* and *B*.

There are a few basic ways to terminate the generic label-correcting algorithm given that the search graph contains a negative cycle. One such method is to stop the algorithm as soon as a shortest-path distance, *d*, becomes smaller than a specified lower bound. Generically, for any graph, the lower bound $-nC$ can be used, where *n* is the number of nodes in the graph and *C* is the $max(\mid c_{ij} \mid)$, or in other words, the maximum absolute value of a cost on an arc. The value $-nC$ is the lower bound because the greatest cost acyclic path for a graph can have at most *n-1* arcs. If there exists a shortest path to a node with cost less than $-nC$, and the largest possible return path to the source node has cost nC, then there must exist a negative cycle. Thus, the algorithm can terminate if any distance value is less than this bound.

Termination for STNs Chosen From TPNs

The magnitude of the bound affects how long the SSSP algorithm must cycle before it terminates and detects the negative cycles. A concern about the bound $-nC$ is that it can be quite conservative. In this section we show that for STNs selected and built from TPNs, a tighter lower bound of 0 can be used.

When running the label-correcting algorithm on a distance graph converted from an STN that was selected from a TPN, the distance values do not need to reach all the way down to $-nC$ for the algorithm to terminate, the

algorithm can stop when any shortest-path distance value for any node reaches below zero. The property of these STNS that allows for zero to be the lower bound is that STNs always contain timepoints (excluding the start) that must be assigned times that are equal to or later than the start node. The input temporal network always contains timepoints (excluding the start) that must be assigned to times equal or later than the start node. A network containing a timepoint that is constrained to happen before the start node is not allowed as input.

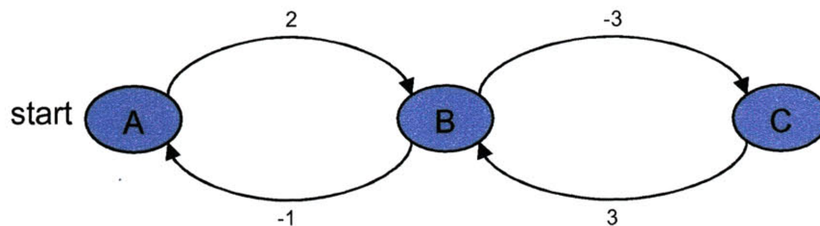


Figure 17 – Illegal Distance Graph of STN of a Plan

Figure 17 shows a distance graph that is not allowed in Kirk planning. This network constrains node *C* to happen at least 1 time unit before the start node since the shortest path from node *A* to node *C* is -1. Since node *C* precedes *A*, a correct STN would need to label C as the start node. STNs selected from TPNs naturally have this start node and therefore constrains all other nodes to happen after the start.

Given that all timepoints of candidate STNs of plans must occur after the start node, it guarantees that all timepoints have a negative cost path back to the start node. Thus, if the shortest-path distance value at a node or timepoint is computed to be negative during some iteration of the SSSP algorithm, then since there exists a negative cost path back to the start node from the argument stated above, then we are guaranteed to have a negative cycle. Because of this property of candidate STNs created from TPN plans, the algorithm can terminate as soon as it discovers a distance value less than zero.

Generic Label-Correcting Algorithm as a Consistency Procedure

There are a few modifications that need to be made in order to change the generic label-correcting algorithm to be a consistency procedure for STNs. Generally, the algorithm either calculates the SSSP or it fails and returns that

there is a negative cycle. In this thesis, we need a temporal consistency procedure to return true or false indicating the temporal consistency of the input graph. The procedure should return true if the graph does not have a negative cycle and false if the graph does contain a negative cycle. In the remainder of the thesis, the consistency procedure version of the label-correcting algorithm will always be referred to when the label-correcting algorithm is mentioned.

Generic Label-correcting Example with Soccer Candidate STN

The figure below shows the first few steps of the generic label-correcting algorithm run on a part of the distance graph from the soccer scenario. In the initial step, all of the *d*-values are initialized to ∞ except for the start node which is initialized to a value of 0. All violating arcs in the graph are represented by dotted and dashed lines. The arbitrary arc chosen for update is shown with dotted lines and all other violating arcs are shown in dashed lines.

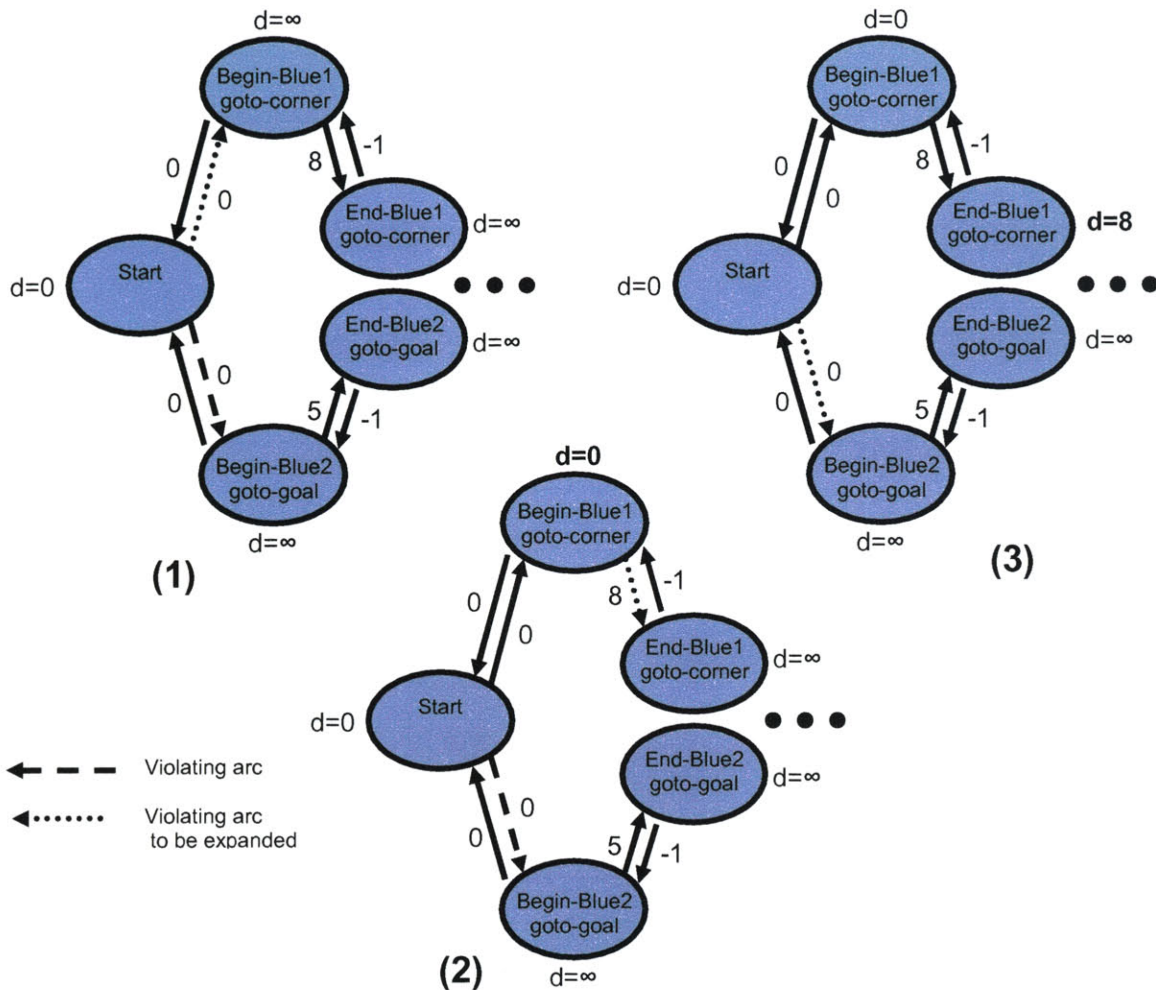


Figure 18 – Generic Label-Correcting Algorithm On Partial Soccer Scenario STN

At step (1), there are two violating arcs, the arc between *Start* and *Begin-Blue2-goto-goal* and the arc between *Start* and *Begin-Blue1-goto-corner*. The first violating arc is arbitrarily chosen for update and *Begin-Blue1-goto-corner* is updated and assigned a *d=0*. Once this violating arc is updated (step (2)), additional arcs may also become violating. In this scenario, only the arc between *Begin-Blue1-goto-corner* and *End-Blue1-goto-corner* becomes a newly violating arc. In step (3), we choose to update this newly violating arc, assigning *d=8* to *End-Blue1-goto corner*. The algorithm continues until there are either no more violating arcs or a termination condition has occurred.

3.2.3 Modified Label-Correcting Algorithm

A key issue for label-correcting algorithms is finding an effective mechanism for implementing efficiently. For label-correcting algorithms, the efficiency is dependent on the process of searching for violating arcs. At the simplest level, the algorithm can simply scan all the arcs in the graph until it finds one that is violating. This process is repeated for every violated arc until termination. However, this is very time consuming since a large number of non-violating arcs are scanned. To support fast temporal consistency detection, we build upon the modified label-correcting algorithm, that implements violated arc detection very efficiently. We will also use the same idea underlying this modification as the basis for our incremental algorithm.

The modified label-correcting algorithm simply refers to an implementation of the generic label-correcting algorithm where a queue of updated nodes is maintained, in order to check for outgoing arcs that might be potentially violating. Consider why only the updated nodes need to be examined. If during a particular iteration of the algorithm, the *d*-value for a node 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 $d(j)$, hence any out arc (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 the outarcs of any node on the queue.

At initialization of the modified 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, theses 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 worst-case running time for a label-correcting algorithms 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) [1].

In this thesis, we build upon a variant of the modified label-correcting algorithm that is implemented with a FIFO (first-in first-out) queue to perform negative-cycle detection for temporal consistency. The FIFO queue removes nodes from the queue in the same order that they were added into the queue. We choose the FIFO label-correcting algorithm is the fastest available polynomial time algorithm for determining shortest-path.

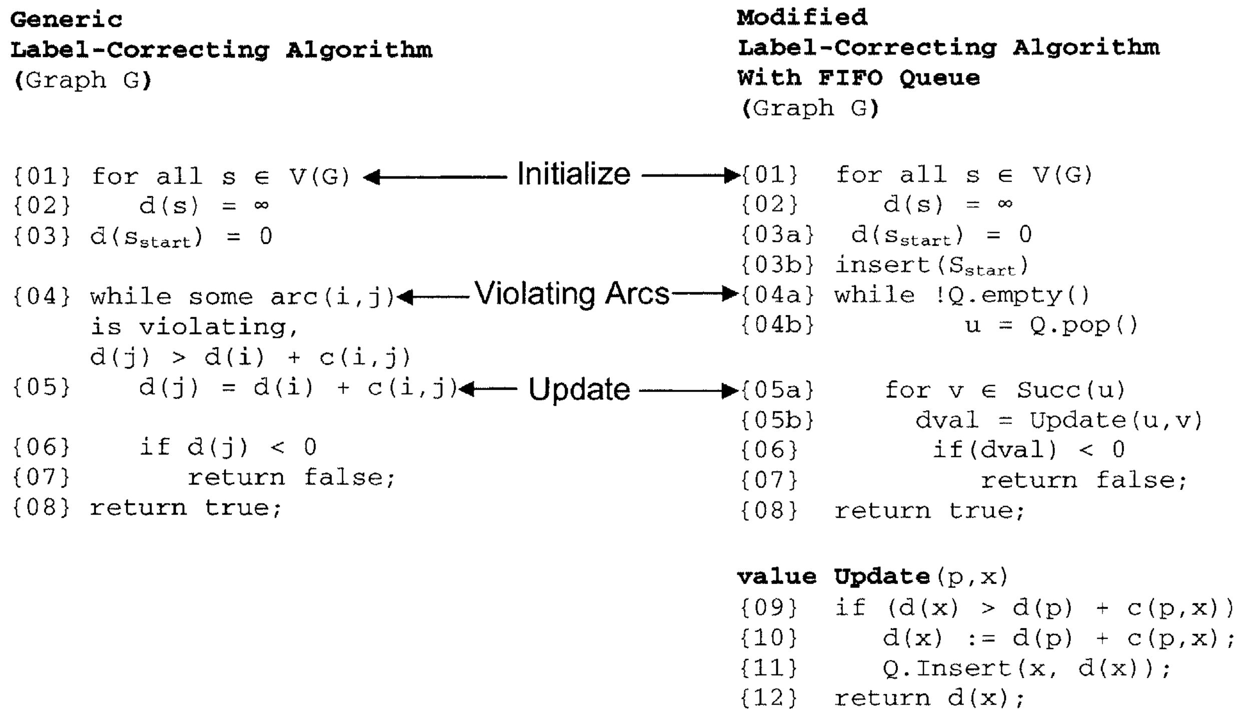


Figure 19 – Pseudo Code for Modified Label Correcting Algorithm

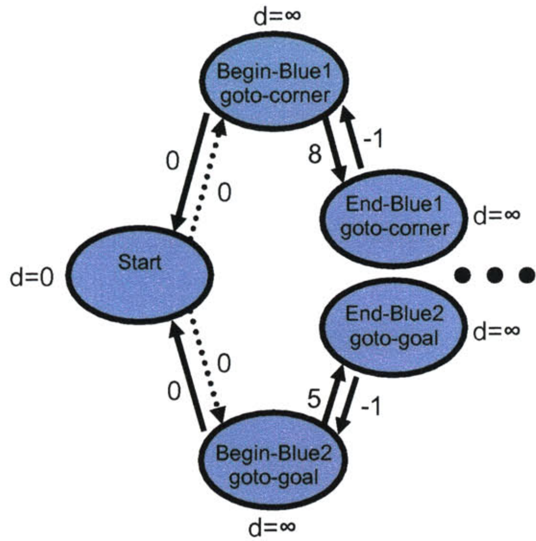
This pseudo-code for the FIFO modified label-correcting algorithm has the same basic structure as the generic label-correcting algorithm. There is an initialization where all distance values are initialized to ∞ (lines {01-02}), and the start node d-value is set to 0 (line {03}). In the modified label-correcting algorithm, there is an additional initialization step of adding the start node into the

queue, since the start node is the only node that contains outarcs that are potentially violating at the start. Both algorithms continue to update *d*-values by selecting violating arcs, for the generic label-correcting algorithm via arc examination and for the modified label-correcting algorithm via removing nodes from the queue, until there are no more violating arcs. In both cases, if a *d*-value falls below the threshold for negative cycles, the algorithms returns false, signifying an inconsistency. If the graph is consistent, both of the algorithms terminate, either after the queue is empty for the modified label-correcting case or and no violating arcs exist for the generic label-correcting case.

3.3 Temporal Consistency of "Mission to the Goal" Scenario

Consider the application of the label-correcting algorithm to the soccer example. When applied to the "Mission to the Goal" candidate STN, shown in Figure 8, the label-correcting algorithm determines that the graph is temporally consistent. We can verify this visually by looking at the paths that lead up to the synchronization node in Figure 8, *Begin-Blue2-shoot*. At this synchronization timepoint, *Blue1* must have centered the ball from the corner and *Blue2* must be at that destination point of the centered ball. If we add up the upper and lower time constraints for *Blue1* to perform its task by the synchronization node, we see that it must take *Blue1* at least 3 time units and at most 10 time units to complete all the activities. Similarly, it must take *Blue2* at least 1 time unit and at most 10 time units to complete its tasks. Thus, there is a consistent overlap where, if we force *Blue1* and *Blue2* to execute within 3 and 10 time units, then the mission will succeed.

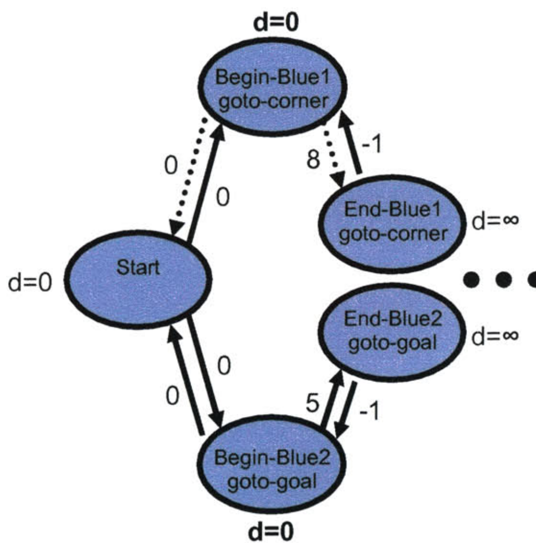
Figure 20 demonstrates the beginning three steps of the FIFO label-correcting algorithm run on the STN for the soccer scenario candidate plan. As before, only part of the candidate STN will be used to illustrate how the algorithm operates. The dotted lines in this figure show which arcs will be updated or need to be examined for update on the next iteration.



Step 1

FIFO Queue

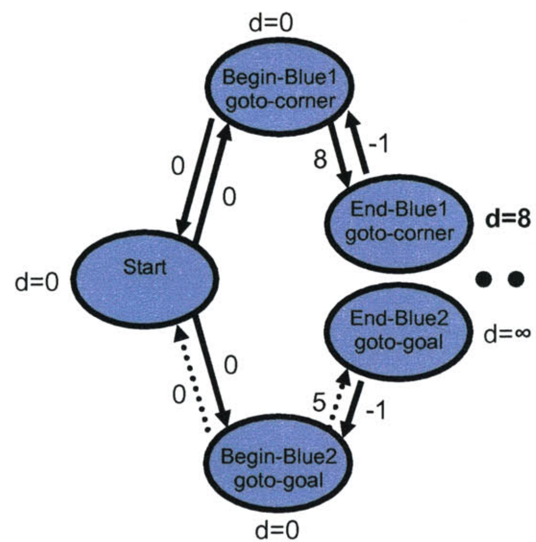
(1) Start



Step 2

FIFO Queue

(1) Begin-Blue1 goto-corner
(2) Begin-Blue2 goto-goal



Step 3

FIFO Queue

(1) Begin-Blue2 goto-goal
(2) End-Blue1 goto-corner

Figure 20 – FIFO Label-Correcting Algorithm On Soccer Scenario STN

The example shows that at initialization (Step 1, Figure 20), only the start node has been inserted into the queue, meaning that all outarcs from the start node need to be examined for violation. Once the algorithm is finished checking the outarcs and updates the heads of the violated edges, it removes the start node from the queue and adds the updated nodes into the queue. In this case, since both *Begin-Blue1 goto-corner* and *Begin-Blue2 goto-goal* were updated in Step 1, both nodes are added into the queue. Step 2 expands the node *Begin-Blue1 goto-corner.* For this expansion, only *End-Blue1 goto-corner* needs to be updated and is therefore added to the queue. The arc leading from *Begin-Blue1 goto-corner* back to the *Start* does not improve the shortest-path value of the *Start* node, therefore, the *Start* node is neither updated nor added to the queue. For Step 4 (not shown), *Begin-Blue2 goto-goal* would be expanded next, resulting in additional nodes being added to the queue. As previously stated, this graph is temporally consistent and therefore the algorithm terminates with non-negative start distances, *d-values*, when the queue is empty.

Chapter 4

The Incremental Temporal Consistency Algorithm (ITC)

Incremental algorithms can significantly increase the speed of a task because much of the work that was performed for previous calls to tasks can be reused in successive searches. These algorithms are most advantageous when the successive tasks that the incremental algorithm is run on are similar to previous tasks.

As a simple example of where an incremental algorithm can be useful, consider a planning task for an autonomous taxi that navigates through a large metropolitan city. Suppose the autonomous taxi has planned a route from the airport to a hotel. However, early on in the drive, the taxi learns of a road block in the city near where the hotel is located. The taxi must plan a new path in order to reach its destination. It would be wasteful to throw away what is currently known to be the best path to the hotel and start a new search from scratch, since the path has only changed near the hotel and not near the airport. A new search would require re-examination of all paths going from the airport to the hotel. This is how a non-incremental algorithm works on the taxi problem. It has no mechanism to remember what has been computed previously.

It would be much more efficient to start with the optimal path that is already known from the airport to the hotel, and then update the parts of the path affected by the road block in the city. Incremental algorithms for path planning problems exist, such as Incremental A* [8] and D*[15]. Reusing previous work is the main idea behind incremental algorithms, and consequently the ITC algorithm develops this idea for STNs.

This chapter first introduces the ITC algorithm. It then gives a quick introduction to truth maintenance systems (TMS) and discusses how concepts from TMSs and the modified label-correcting algorithm (Section 3.2.3) are combined in order to achieve the ITC algorithm. This is followed by an example

of how the ITC algorithm detects negative cycles and extracts the conflicts, which summarize temporal inconsistencies.

4.1 The ITC Algorithm Overview

The Kirk temporal planner requests temporal consistency checks on STNs of candidate plans as they are built up node by node (Section 2.4). 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 chosen arcs and nodes need to be updated. Temporal consistency of an STN can therefore be determined with fewer nodes updated. Sometimes, this results in an order of magnitude in savings, as is empirically demonstrated in the Chapter 5. Additionally, if the ITC algorithm finds that a candidate STN is inconsistent it will return a set of simple temporal constraints that result in the inconsistency, referred to as a *conflict*. The conflict tells the plan generation algorithm which decisions contributed to the inconsistency. This allows it to make more informed decisions about what candidate to consider next in order to resolve the inconsistency. This ultimately speeds up the planner's ability to find a consistent candidate plan. A discussion of conflict extraction algorithms for optimal search together with a performance analysis can be found in [14].

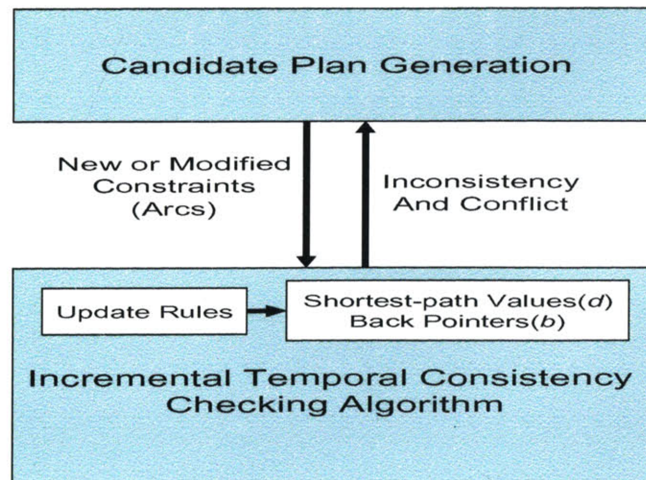


Figure 21 – Diagram of ITC Algorithm

Figure 21 shows how the ITC algorithm interacts with the Kirk temporal planner, as it performs incremental temporal consistency. Once a candidate plan is initially generated by the plan generation phase, it is sent to the ITC algorithm to be checked for temporal consistency. The ITC algorithm either finds the candidate plan temporally consistent and the planner passes the consistent plan to its next stage or it finds the candidate temporally inconsistent and returns a conflict, consisting of simple temporal constraints. The plan generation phase can then use the conflict to find a new candidate plan that resolves the conflict. As plan generation makes modifications to the plan, it communicates these changes to the ITC algorithm. ITC uses this information to determine which nodes need their start distances updated when consistency is checked and which ones do not.

4.2 Insufficiency of Modified-label Correcting to Perform ITC

In order to perform any type of temporal consistency checking, we must use an algorithm that is capable of detecting negative cycles. As discussed in Section 3.1, the FIFO modified-label correcting algorithm is a good choice because it is very fast, and consequently supports the needs of a fast continuous planner. It also has some of the capabilities needed to perform incremental updates. In particular, the modified label-correcting algorithm can handle an arc that improves a node's shortest-path distance since all it needs to do is add this node to the queue and propagate down the line. However, the modified-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 must be found. To handle this case, a new strategy of keeping track of which shortest-paths distances on nodes affect each other needs to be incorporated. We develop this strategy in Section 4.4. However, we first examine the incremental update methods of a truth maintenance system (TMS), since a TMS deals with an analogous issue for truth updates when clauses are removed.

4.3 Truth-Maintenance Systems and Unit Propagation

Truth maintenance systems (TMS) [4], developed in the late-1970s, were widely used in the AI community for solving problems where the truth of facts are added and then later retracted. A TMS determines the truth of propositions. It provides justifications for its conclusions, recognizes inconsistencies, remembers previous derivations, and guides searching by identifying propositions responsible for inconsistencies. TMS have been used frequently in applications for system analysis, diagnosis, and other deductive tasks [6]. The incremental temporal consistency algorithm described in this thesis offers and analogous set of four capabilities, and uses concepts analogous to a TMS' set of supports in order to intelligently reuse previous calculations. This section describes how a truth maintenance system works, specifically, the LTMS [9]. Later on in Section 4.3.4, we show how this algorithm is analogous to the incremental temporal consistency algorithm.

4.3.1 LTMS

LTMS operates on propositional sentences, containing clauses and literals. A literal is a proposition, representing a fact, or the proposition's negation, (e.g. Q or $\neg$Q). A clause is a disjunction of literals (e.g. $X \vee Y \vee Z$). Lastly, a propositional sentence is a conjunction of clauses (e.g. $(\neg X \vee M \vee \neg C) \wedge (\neg J \vee \neg K \vee M) \wedge (X \vee J \vee Z)$). For additional details on propositional logic, see [12].

The job of an LTMS is to maintain and return the truth of propositions, given some initial premise. Therefore, the LTMS has two basic tasks. First, given a propositional sentence and the premises, it must be able to identify those literals that must be true. Second, once it has determined the truth of the propositions, whenever a clause is added or removed, it quickly determines how this change alters the truth of the propositions. An LTMS achieves these tasks with two functions, *propagate* and *unsupport*.

4.3.2 Unit Propagation and Support

If some literal in a sentence is known to be true, then the consequences of this knowledge must be propagated to all clauses containing this literal.

Additionally, an LTMS also keeps track of how a literal was assigned a truth value by storing which clause entailed this literal. This clause is called a support. The pseudo-code for this propagation step is shown in Figure 22.

```
procedure propagate(clause A)
{01}   if all literals in A are false except l, and l is
       unassigned
{02}   then assign true to l and
{03}        record A as a support for l and
{04}   for each clause C mentioning "not l",
{05}        propagate(C)
```

Figure 22 – Pseudo Code for Propagation in LTMS

The first line updates clause *A* by searching for an unassigned literal of clause *A* that must be assigned to *true*. If the literal exists, it is set to *true* and the algorithm remembers that clause *A* is the support for why literal *l* is true. This truth assignment is then propagated by updating any clause containing the literal ¬*l*, which has just become false (line {05}). At the completion of running this procedure, any literal that is true by unit resolution on the clauses will be set to *true*. Additionally, the clause that determined the truth of each literal will be stored as the support for that literal.

4.3.3 Clause Deletion and Unsupport

If a clause is deleted, then just as in the support case above, this information is propagated to in order to update which literals are true.

```
procedure unsupport(clause D)
{01}   if D supports some proposition p
{02}   then delete p's support and truth assignment;
{03}        for each  support C containing p    // Delete consequences
{04}           unsupport(C);
{05}        for each clause A containing p    //Resupport p
{06}        then propagate(A);
```

Figure 23 – Pseudo Code for Unsupport in LTMS

In this procedure, if removing a clauses unsupports proposition, *p*, then clauses that contain *p* and support other propositions may no longer be a valid support. Unsupport is run on each of these clauses to recursively invalidate the truth of all propositions that they support (lines {03-04}). Hence, Lines {05-

`06}` try to re-support each proposition that lost its support, once all the consequences of that proposition are unsupported.

4.3.4 Incremental Ideas from LTMS

We see from the pseudo-code for an LTMS that a key method that the system uses to determine what needs to be updated is through a tree of support. The truth of a proposition is supported by the clause that entailed the truth. When that clause is removed, the proposition that it supported is no longer entailed from that clause, therefore all clauses that contain this proposition need to be re-examined. With this method, a TMS can quickly find all clauses and propositions that depend on the removed clause to derive the current truth assignments. If the removed clause does not support any propositions as identified by the *unsupport* function, then when the clause is removed no update is required beyond just removing this clause.

Through the use of supports, the algorithm finds the exact number of clauses that need to be reconsidered when unit propagation step is restarted, thus saving a significant amount of recomputation.

4.3.5 ITC analogs to LTMS

The ITC algorithm needs to maximize speed by minimizing work, thus in a negative cycle detection algorithm this means speeding up the SSSP algorithm, by reducing the number of distances that need to be calculated. This can be achieved in much the same way as LTMS uses support to find invalidated clauses. For ITC, it uses its support tree to find invalidated shortest-paths.

We begin to develop the analogy between the two algorithms by first stating that updating a truth value for a proposition is analogous to assigning a shortest-path distance to a node. Both must determine if the new value should replace the old. In addition, removing a clause that supports a proposition thereby invalidating its truth assignment, is similar to removing or increasing the distance on an arc in the distance graph, such that the distance value assigned to that node is invalidated (it becomes more than the shortest-path distance). For both the LTMS and ITC, all consequences that depend on an invalidated truth assignment or distance value must also be invalidated. With this parallel,

ITC can use a recursive *unsupport* function similar to that for the LTMS, in order to invalidate all shortest-paths that have been invalidated by the changed arc. This procedure quickly identifies which nodes needs to be re-evaluated in order for the algorithm to find the shortest-path distances; saving the work of re-examining all nodes.

4.4 ITC Algorithm's Incremental Update Rules

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

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

Second, a decrease in an arc distance can improve the shortest-path distance to a node such that it is now better to traverse through that arc. The improved arc can either previously be in the shortest-path of the node or not be in the previous shortest-path. This case can be handled using the modified-label correcting algorithm strategy for updates because the improved distance at this node can be propagated further down the graph simply by adding it to the queue and checking for violating arcs.

Lastly, an increase in arc distance can alter the value on the shortest-path to a node such that the shortest-path is now worse than what it was before. This case cannot be handled by the modified label-correcting algorithm. The modified label-correcting algorithm requires that all start distance values be upper bounds, however, when the arc distance increases, this is no longer guaranteed. To get on track we must identify the start distance values that are no longer valid because of the edge cost increase. In particular, all paths that depend on the previous shortest-path through the node also have incorrect distance values. For this case, the strategy used in a TMS of tracing the set of support to determine

consequences can be applied. ITC must recursively invalidate all start distances that are supported by the invalid distance on the node directly affected by the arc increase and all successor nodes that depend on this node.

To allow for successors to be invalidated recursively, the ITC algorithm adds a predecessor pointer, *p*, to every node. The predecessor pointer of a particular node *n* points to the node that directly precedes node *n* in the best known shortest-path. For example, suppose that reaching node *Y* from the start node in the shortest manner requires traversing through arc *XY*, then the predecessor pointer for node *Y* would be set to *X*. This tells us that in the best path to node *Y* from the start node, node *X* must be visited directly prior to visiting node *Y*. This is equivalent to saying node *Y* is supported by node *X* in the LTMS terminology. If it is unknown how to get to a particular node, then that node's predecessor is set to *unknown*.

The next three subsections describe how the ITC algorithm deals with the three cases previously outlined.

4.4.1 Arc Change without Affect to Shortest-Path

Recall that the first case involves any arc change that does not affect the shortest-path of the head node. The arc distance can increase as long as it is not on the current shortest-path of the head node. It can decrease as long as the path through this edge to the head node is better than the current shortest-path at the head node. Figure 24 shows the instance of this case where an arc increases in its distance.

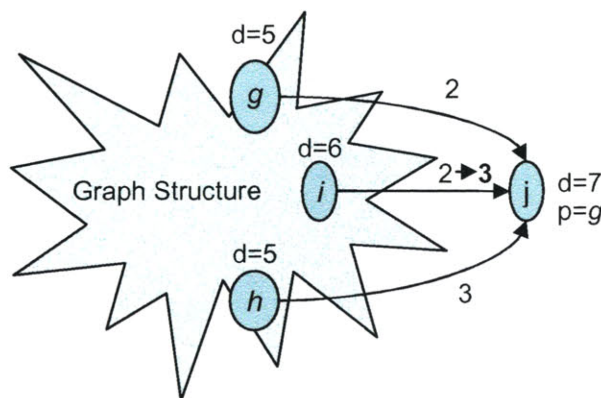


Figure 24 – Arc Change without Improvement or Affect

In Figure 24, the current best way to get to node *j* is to go through node *g*, as specified by the predecessor pointer 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*-value for node *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 *d*-value at node *j* does not need to be updated. If no *d*-values are affected by an arc change, then no further updates need to be performed. All start distances are up-to-date and the consistency of the graph is preserved.

The pseudo-code for this case is shown in Figure 25, however, in the complete pseudo-code for the algorithm in Section 4.5, this case is not shown because it performs no action.

```
{01} if (d(arc.head) < d(arc.tail) + c) AND
         p(arc.head) ≠ arc.tail)
{02}     return;
```

Figure 25- Pseudo Code for Arc Change without Affecting Shortest-Path

The first condition in line `{01}` tests that the start distance for the head node has not improved. The second condition tests that the arc is not on the current shortest-path for the head node. If both conditions are true, then we have the situation as described above, and no action is performed.

4.4.2 Arc Change Improves Shortest-Path

Frequently, an arc change will improve the cost of an arc, and consequently the shortest-path to one or more nodes. This can happen both when the changed arc is on the current shortest-path or not on the current shortest-path to the head node. In either case, the rules are applied the same way. The distance value of the node at the head of the arc needs to be updated appropriately and this updated distance value propagated to successor nodes. Figure 26 below shows the case when the arc that improves the distance value at the head node is not on the shortest-path for that node.

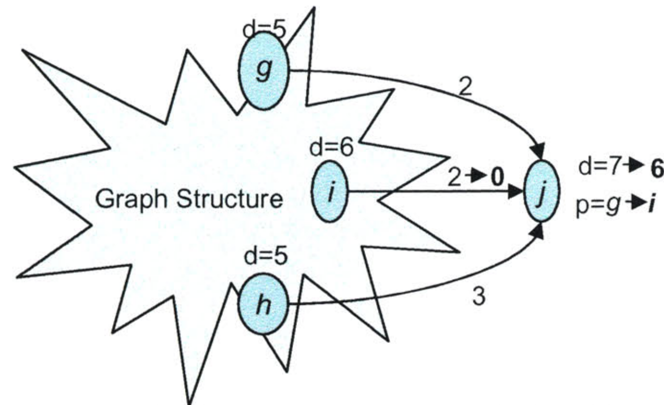


Figure 26 – Arc Change Improves Shortest-Path

The figure shows arc_{ij} reducing in cost from *2* to *0*. With this change, the shortest-path distance to node *j* can be decreased from *7* to *6*, by first going through node *i*. Both the *d*-value and the predecessor pointer for node *j* therefore need to be updated. The predecessor pointer should now point to node *i* instead of node *g*, representing that we should traverse through node *i*, in the shortest path to node *j* and the *d*-value should be updated to represent this new shortest-path of distance of *6*. As a final update step, since the successor nodes of node *j* can be affected by the improvement to node *j*'s *d*-value, 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. The pseudo-code for this case is shown in Figure 27.

```
{01} if (d(arc.head) > d(arc.tail) + c)
{02}      d(arc.head) := d(arc.tail) + c;
{03}      p(arc.head) := arc.tail;
{04}      Insert(arc.head);
```

Figure 27 – Pseudo Code for Arc Change Improves Shortest-Path

The code starts by testing whether the start distance value has improved for the node at the head of the arc (line `{01}`). If it has, then ITC first updates the head node's *d*-value (line `{02}`) sets the predecessor pointer to the node at the tail of the arc,(line `{03}`) and then inserts the changed head node so that the successors are updated (line `{04}`).

4.4.3 Arc Change Invalidates Shortest-Path

In the final case, an increase in the distance worsens 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-path distances, a recursive function must be called to invalidate all nodes supported in the chain. Once the *d*-values have been updated, the parents of the affected node can be enqueued and a new start distance may be propagated from this node. Figure 28 below shows this scenario.

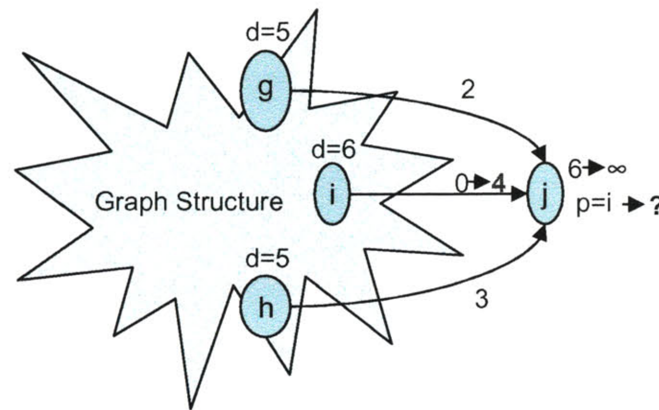


Figure 28 – Arc Change without Improvement and Shortest-Path Affected

Again arc_{ij} has increased in value, but this time it cannot be treated as just an arc increase with no affect. Since node *j*'s *d*-value of *6* was calculated by traversing through the changed arc, the value at *j* is no longer valid. An update must be performed on all of node *j*'s parents in order to find the new shortest-path distance to node *j*. This is done by adding all of node *j*'s parents to the queue. Additionally, since node *j*, may support other nodes elsewhere in the graph, ITC must recursively invalidate the *d-values* and predecessor pointers of all nodes that use node *j* in their shortest-path, as well as the *d-values* and predecessor pointers of node *j*. This is done by setting their *d-values* to ∞ and changing the predecessor pointers to *unknown*. When the temporal consistency algorithm is restarted and nodes are evaluated from the queue, the algorithm

calculates and updates node *j* with the new best path since node *j's* parents are in the queue. The pseudo-code for this case is given in Figure 29.

```
{01}   if (d(arc.head) < d(arc.tail) + c) AND
{02}       p(arc.head) == arc.tail
{03}       d(arc.head) := ∞;
{04}       p(arc.head) := unknown
{05}       InvalidateSupports(arc.head);
{06}       InsertParents(arc.head);
```

Figure 29 – Pseudo Code for Arc Change Invalidates Shortest-Path

Line `{01}` of the pseudo-code checks that the path for the changed arc is longer than the shortest-path. Additionally, line `{02}` checks that this path was on the shortest-path of the head node. If both conditions are true then, in line `{03}`, ITC resets the distance value of the head node and in line `{04}`, the predecessor pointer is set to *unknown*. Lines `{05-06}` perform the recursive invalidation of supported successors and insert the invalidated node's parents into the queue.

4.4.4 Addition and Removal Arcs

Often the change in the graph is not a changed arc distance, but an addition of a new arc or the removal of an existing arc. With additions and removals one of the above three scenarios above can still be applied by mapping the arc addition and removal to a distance decrease and increase.

If a new arc is added, then the new arc distance is treated as being previously set to ∞ and now changed to a the arc distance. Then the arc addition can fall into either the case where the shortest-path distance to a node is improved (Section 4.4.2) or the case where the shortest-path to a node is unaffected (Section 4.4.1).

If an arc is removed, then the new arc distance is treated as being set to ∞ from a previous value. The arc removal case can then fall into either the case where nothing is affected by the removal of the arc, Section 4.4.1, or the case where a new path to the head node of the arc must be found, Section 4.4.3.

4.5 Incremental Temporal Consistency Algorithm Pseudo-Code

When the planner requires a temporal consistency check on an STN of a candidate plan, *G*, it will call *CheckTemporalConsistency*. Depending on whether the consistency check is starting from scratch or incrementally, the planner will call either *Initialize* or *ModifyConstraint,* respectively, before the call to *CheckTemporalConsistency*. 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.

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

Conflict
CheckTemporalConsistency(G)
{07}    while !Q.empty()
{08}      u = Q.pop()
{09}      for v ∈ Succ(u)
{10}        dval = Update(u,v)
{11}        if(dval) < 0
{12}          c = CompletedCycle(v);
{13}          if(c)
{14}            return ExtractConf(c, ∅);
{15}    return 0;

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

Node
CompletedCycle(v)
{21}    if L.contains(v)
{22}      return v;
{23}    else
{24}      L.add(v)
{25}      return 0;

void
ModifyConstraint(x,y,l,u)
{26}    ModifyArc(arc(y,x),-l)
{27}    ModifyArc(arc(x,y),u)

Conflict
ExtractConflict(c,l)
{28}    if l.contains(c)
{29}      return l;
{30}    else
{31}      l.add{c};
{32}      ExtractConflict(p(c),l);

void
ModifyArc(arc,c)
{33}    setCost(arc,c);
{34}    if (d(arc.head > d(arc.tail) + c)
{35}       d(arc.head) := d(arc.tail) + c;
{36}       p(arc.head) := arc.tail;
{37}       Insert(arc.head);
{38}    elseif (d(arc.head) < d(arc.tail) + c)
           AND (p(arc.head) == arc.tail))
{39}       d(arc.head) := ∞;
{40}       p(arc.head) := unknown;
{41}       InvalidateSupports(arc.head);
{42}       InsertParents(arc.head);

void
InsertParents(n)
{43}    for all m ∈ Pred(n)
{44}       Insert(m);
{45}       if(p(m) == n)
{46}          if(m == s_start)
{47}             d(m) := 0;
{48}          else
{49}             d(m) := ∞;
{50}          p(m) := unknown;
{51}          InsertParents(m);

void
InvalidateSupports(n)
{52}    for s ∈ Succ(n)
{53}    if(p(s) == n)
{54}       d(n) := ∞;
{55}       p(n) := unknown;
{56}       InsertParents(s);
{57}       InvalidateSupports(s);
```

Figure 30 – ITC Algorithm Pseudo Code

The *Initialize* function empties the queue, Q, resets all *d-values* to be ∞, and resets all predecessor pointers, *p(i)*, to be unknown, lines `{01-04}`. It then sets the start node's *d-value* to be 0 and adds the start node into the queue, lines `{05-06}`. This sets up the algorithm data structures such that when *CheckTemporalConsistency* is called, it is a completely new run and all paths to nodes need to be examined.

The *CheckTemporalConsistency* function will return either the conflict resulting in inconsistency or no conflict when the graph is consistent. It is called whenever temporal consistency needs to be determined. In line `{11}`, the function checks for the termination condition to see if the lower bound is reached signifying a negative cycle. Once the lower bound is reached, the algorithm calls *CompletedCycle* at every update step (line `{12}`). *CompletedCycle* adds the node that is currently being updated to a list so that it can be checked whether the algorithm has finished walking through the negative cycle. When *CompletedCycle* discovers a cycle it will return the first node discovered in the negative cycle, otherwise it will not return a node. Line `{13}` checks if a node is returned by *CompletedCycle* and allows the algorithm to call *ExtractConf* (line `{14}`) with this so that the negative cycle can be determined and returned by the algorithm.

The *Update* function performs the update step as in the modified label-correcting algorithm. If there is a better path to a node *y* by going through node *x,* as checked by line `{16}`, then the function sets the predecessor pointer of node y to traverse through node *x* and also updates the *d*-value at node *y* to the new cost of traversing through node *x* (lines `{17-18}`). The insert step at line `{19}`, allows the algorithm to update successors of this node by adding them to the queue. Returning the *d*-value in the final step is important for the termination step, where the *d*-value is checked to make sure it has not dropped below zero as described in Section 3.2.2.

InsertParents is a helper function to the *ModifyArc* function. As discussed in the previous section, sometimes the shortest-path to a node needs to be re-determined because the previous shortest-path was altered by changing the

distance on an arc. *InsertParents* inserts all of a node's parents into the queue (lines {43-44}), such that they will all be examined by the algorithm and a new shortest-path to that node can be determined. Built into the code beginning at line {46} is a special case condition where while inserting a parent node we find that the predecessor pointer, *p(i)*, points to the node whose parents we are inserting. The potential for this is frequent with the distance graphs of temporal networks because every temporal constraint has both a forward arc and a return arc, representing the upper and lower bounds. If this happens, the algorithm needs to reset all the values of this particular parent and insert the parent's parents into the queue. If this parent is the start node, we can automatically set the *d*-value to 0 as opposed to ∞, since the distance to the start node is always 0. Simply, if we want to know what the shortest-path to a node is, we check its parents. If we do not know what the shortest-path to the parent is then we check the parent's parent and so on.

The *InvalidateSupports* function performs the recursive call to invalidate all nodes dependent on a node whose start distance value is no longer valid. It checks all the children of the node that is invalidated, resets their *d-values* and predecessor pointers if the successor node uses the invalidated node as a support (lines {53-55}). It then calls *InsertParents* on all nodes that have invalidated *d-values* so that new shortest-paths can be found to these nodes (line {56}). Lastly, it performs the recursive call to *InvalidateSupports* on the child node so that the successors of the successors can be invalidated (line {57}).

The function *CompletedCycle* returns a node contained in a negative cycle if the algorithm has finished walking through the cycle. It returns a 0 node otherwise. *CompletedCycle* detects cycles simply by storing each updated node in a list and compares successive updated nodes with this list. If the node is contained in the list, then that node is returned as a member of the negative cycle. If it is not contained in the list, the node is added and a 0 is returned meaning that no cycle was found.

ModifyConstraints takes as input two nodes and the upper and lower bound constraints between them. This function is a convenience function that

allows the planner to change constraints on the STN of the candidate plan. *ModifyConstraints* translates STN constraints to distance graph constraints as described in Section 3.1.1. It then calls *ModifyArc* on these distance graph constraints.

ExtractConf is a conflict extraction function that returns the nodes involved in the negative cycle. It takes as input an initial node within the cycle and recursively walks the predecessor pointers at each point, adding the current node to a list of nodes already traversed. It detects a cycle by checking whether the current node being walked is in the list of nodes already traversed. This list is returned as the conflict.

The last function, *ModifyArc*, is the main function involved with the incremental temporal consistency algorithm. Given a changed arc, it updates this arc and performs the appropriate steps, as illustrated in the discussion in Section 4.4, to initialize *Q*, *d*-values, and *p(i)* in such a way that a call to *CheckTemporalConsistency* will return a correct answer quickly. For each changed arc, this function must be called separately.

4.6 Negative Cycle Detection with Conflict Extraction

ITC detects negative cycles in the same manner that the modified-label correcting algorithm detects negative cycles. The difference for ITC is that it cuts off as soon as a *d-value* becomes negative, rather than less than *–nC* (Section 3.2.2). As ITC updates the start distances of each node, it checks to see if that updated start distance has surpassed the lower bound of zero. Once this lower bound is surpassed the algorithm continues until a negative cycle has been completely traversed. The set of inconsistent edges can the be found by following the predecessor pointers. Consider the inconsistent graph shown below in Figure 31. It shows the values just after a negative cycle has been detected. ITC has stopped at node *B* because the *d-value* has fallen below zero.

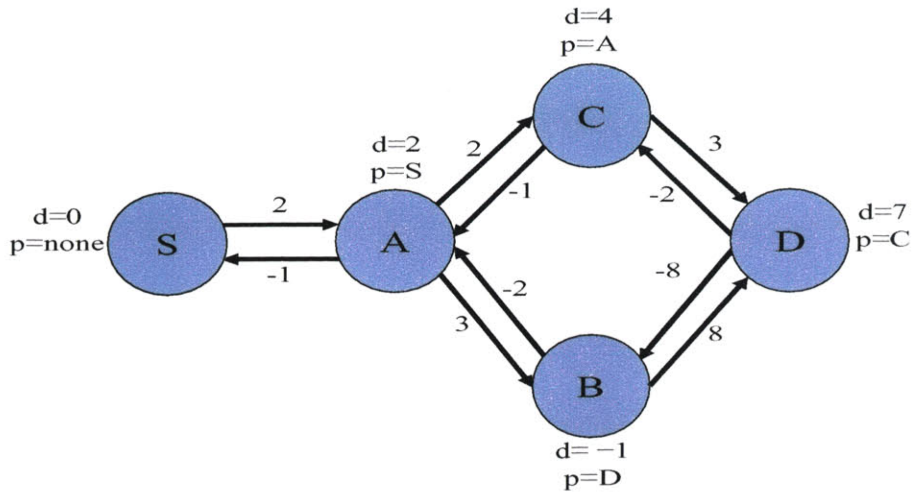


Figure 31 – Example ITC Negative Cycle Detection

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*. The reason the loop has not been closed is because the negative cycle was detected early due to an extremely negative edge, *DB*, which plunges the *d-value* dramatically. The algorithm stops here because of the property discussed in Section 3.2.2 on why termination can occur for temporal networks at a lower bound of zero. Thus, the negative cycle has not been completely traversed.

In order to extract the nodes involved in the conflict, ITC needs to continue walking the negative cycle until it comes back to the node at which it detected the inconsistency. This will set all predecessor pointers so that the source of the conflict can be identified. Figure 32 shows the state of the algorithm once the conflict extraction step has been performed.

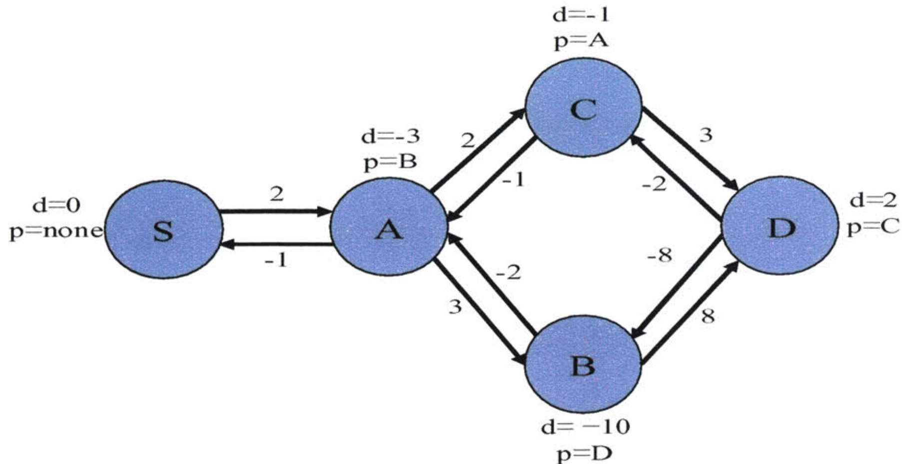


Figure 32 – Example ITC After Conflict Extraction

As the figure above shows, node *A*'s predecessor pointer now points to B, completing the cycle. We can now extract the conflict with the predecessor pointers and report that this graph was found to be inconsistent with the negative cycle *ACDBA*. Notice that once ITC initially detects a negative cycle, it shifts gears from a shortest-path algorithm to a conflict extraction algorithm. Therefore, the start distance values that are computed in the extraction step only help to identify the negative cycle. The values themselves are useless to the shortest-path algorithm because they will never converge since the shortest-path to a node in the graph connected to a negative cycle has the distance $-\infty$ (Section 3.2.2).

4.7 Inconsistency Resolution

A planner will take the conflict from the ITC algorithm and intelligently select a new candidate plan that does not contain this inconsistency. Consider how ITC performs an incremental update after a planner has shifted from an inconsistent candidate plan to a new candidate. For example, imagine in Figure 31, the planner changes activity *CD* so that its upper bound is increased to *10*. This corresponds to an increase in the distance of *CD* from *3* to *10*. Using the update rules, from Section 4.4, the resulting values for the ITC algorithm is shown in Figure 33.

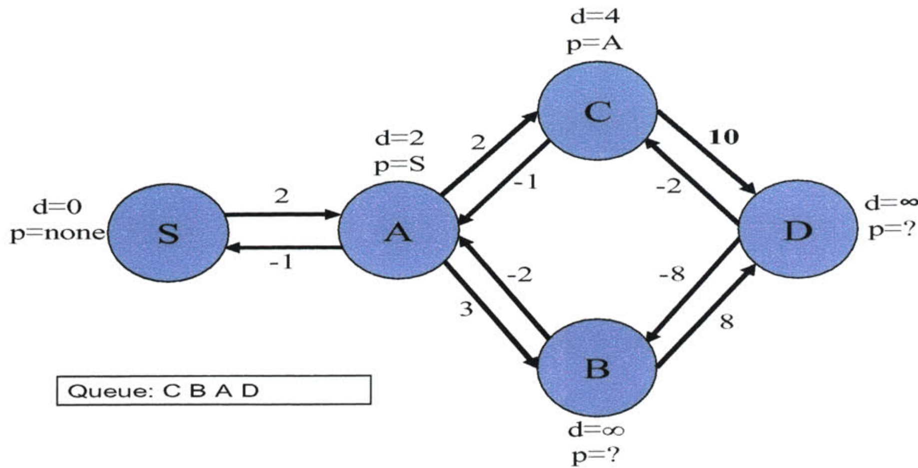


Figure 33 – ITC Algorithm After Inconsistency Repair

ITC detects that the new *CD* arc has invalidated the shortest-path distance to node *D*. Thus, it first invalidates *D* by setting its start distance value to ∞ and predecessor pointer to *unknown* and then goes on to invalidate all nodes supported by *D*, in this case node *B*. Since node *B* supports no other nodes, once it is invalidated the invalidation algorithm terminates. For every node that is invalidated a new shortest-path for that node must be found. Thus, the parents of both *B* and *D* are added into the queue as seen in Figure 33.

Since changing arc *CD* to *10* greatly increased the path that was on the negative cycle, this altered graph is temporally consistent. The ITC algorithm will return this answer after it has updated and removed all nodes from the queue.

4.8 Algorithm Analysis

The label-correcting algorithm is guaranteed to find the shortest-path distances and negative cycles given that the *d-values* are always an upper bound to the true shortest-path distance and the final graph does not contain any violating arcs [1]. Since ITC uses exactly the same mechanism that the label-correcting algorithm uses to detect temporal inconsistencies, then as long as the update rules for ITC maintain *d-values* to be upper bounds on the true shortest-path distance, and do not miss potential violating arcs, then ITC will also be guaranteed to find the shortest-path distances and negative cycles. In the paragraphs below, we will give informal arguments to why this is true for ITC.

ITC guarantees that violating arcs will always be examined. The only way for an arc that is not violating to become violating is for the *d-value* at the tail of the arc to decrease or if the *d-value* at the head of the arc to increase. Given a decrease in the *d-value* at the tail of the arc, ITC update rules add the tail node to the queue meaning that the arc will later be examined later. Given an increase at the head of the arc, ITC adds all the parents of this node into the queue in order to examine potential violating arcs going into this node. Additionally, since an increase at the head node can potentially increase the *d-values* of nodes supported by this head node, a recursive check must be performed on each an every node that is supported. This ensures that nodes that arcs that need to be updated are added to the queue.

ITC guarantees that *d-values* are always upper bounds, or the true start distance. When an arc changes, the *d-values* can potentially no longer be the upper bound on the true start distance. When ITC no longer knows how a *d-value* is calculated, meaning that that node is no longer supported, ITC will recursively invalidate that node and all nodes supported by it. Invalidation sets each *d-value* to ∞, restoring the upper bound guarantee.

4.9 ITC Algorithm on "Mission to the Goal"

The ITC algorithm can save a significant amount of computation when the planner has to re-determine the temporal consistency of a similar STN graph. To illustrate this on a larger planning domain then the examples given above, let us consider ITC on another candidate STNs from the soccer scenario described in Section 1.3.

The Red defender decides that it will change its strategy to try and confuse the Blue team. *Red1* now decides to play closer to *Blue2* instead of challenging *Blue1's* attack on the goal. This new strategy prevents *Blue1* from sending a quick centering pass to *Blue2* because *Red1* would be able to intercept it. However, if *Blue1* can predict this move by *Red1* and re-plan such that the centering pass is made by kicking the ball high over *Red1*, then the plan to score a goal would again succeed.

The difference is that this type of centering pass takes significantly more time to complete, resulting in changed temporal bounds on the centering pass activity. The TPN illustrating both STNs is shown in Figure 34, with the new contingency plan represented with the shaded nodes.

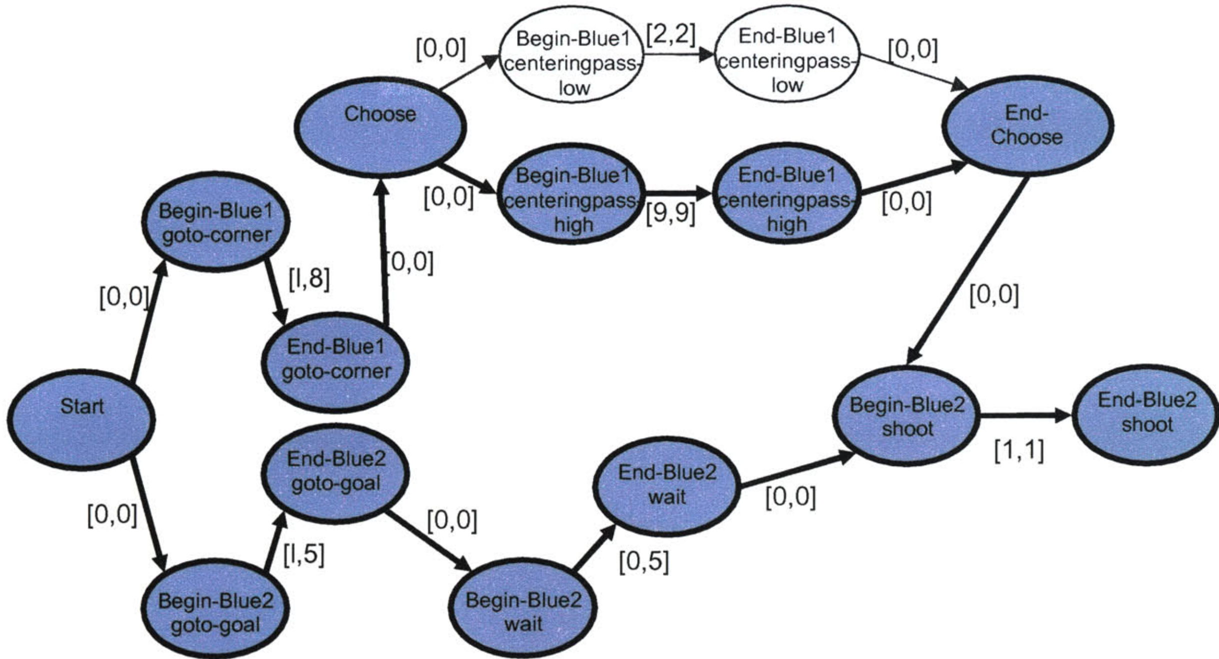


Figure 34 – "Mission to the Goal" with New Temporal Bounds

The new temporal bounds on the high centering pass constrain the pass to take 9 time units, which is about 5 times longer then the previous low centering pass. However with the temporal bounds specified for the "mission to the goal", the mission is still feasible even with the high floating centering pass selected.

First, we can again visually inspect the STN and determine that it is temporally consistent and the "mission to the goal" may succeed. Adding up temporal bounds again before the synchronization node shows that *Blue1* now must complete its set of activities no sooner than 10 time units and no later 17 time units. *Blue2's* time constraints have not changed from the previous STN and must complete its set of activities no sooner than 1 time unit and no later than 10 time units. Again, there is a non-empty overlap (even though much smaller) and we can see that if both Blue robots complete there task in exactly 10 time units then the mission will succeed.

Using an ITC, we find that only a few nodes need to be updated in order to determine the temporal consistency of the new plan. The figure below shows which nodes need to be examined in order to determine the temporal consistency of the newly revised STN given that we have already calculated the consistency of the previous mission plan.

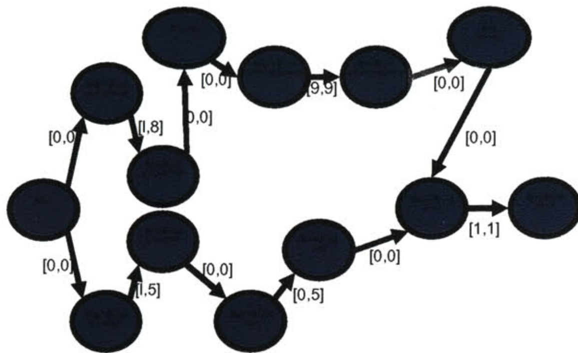


Figure 35 – Examined Nodes for Non-Incremental Algorithm

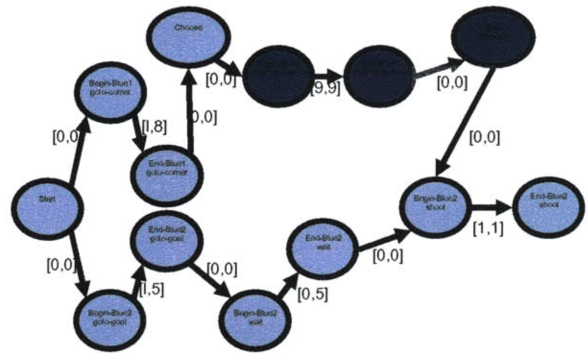


Figure 36 – Examined Nodes for Incremental Algorithm

In Figure 35 and Figure 36, the darker ovals are nodes that require examination and the lighter ovals are those that are not examined. Figure 35 shows that only the nodes directly affected by the change in the *centering-pass* activity need to be updated. In this robotic soccer scenario, the ITC algorithm performed 77% less work than if a non-incremental algorithm. However, for larger more complex STN graphs, the savings can be much more dramatic when using an incremental temporal consistency algorithm. Chapter 5 will empirically demonstrate this claim.

Next consider the case where the centering pass takes longer than 9 time units, in which case the mission plan, the plan to score a goal, would become temporally infeasible. The reason is because the mission can succeed only if *Blue1's* pass reaches *Blue2* when *Blue2* is in front of the goal. If the pass takes any longer, *Blue2* will no longer be waiting for the pass in front of the goal and thus will not be able to shoot the ball. A visual inspection by adding up the timing constraint for *Blue1* and *Blue2* shows that there is no overlap between possible execution times, if the centering pass takes longer than 9 time units. Figure 37

below shows the number of nodes that the ITC algorithm will traverse in order to determine this temporal inconsistency.

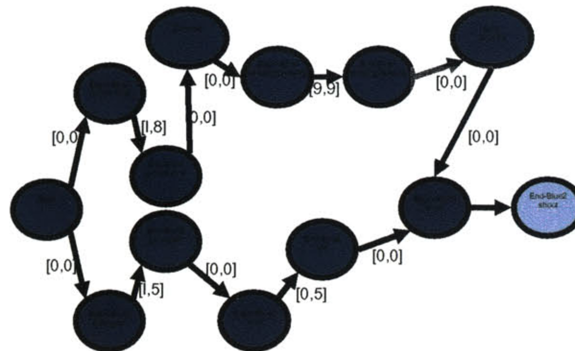


Figure 37 – Examined Nodes for Incremental Algorithm with Inconsistency

Although, the savings is less significant for this scenario when the centering-pass activity produces an inconsistency, only 8%, the one node not remained unexamined demonstrates the exact motivation behind incremental algorithms. All of the nodes that need to be re-examined in Figure 37 are exactly the nodes that are involved in the temporal inconsistency. In general, it is the case that temporal inconsistencies will see less savings then successive searches that return temporal consistency because inconsistencies usually have to be propagated through the cycle before it to be detected. However, with incremental candidate generation as described in Section 2.4 and conflict direction, the majority of candidate plans will be consistent and a smaller fraction will be inconsistent.

Chapter 5
Discussion

This chapter describes how the ITC algorithm is implemented and then integrated into the Kirk Flexible Temporal Planner. It then gives the performance data of the ITC algorithm compared to previous temporal consistency algorithms. The chapter then concludes with a summary of the main contributions in the thesis and some suggestions for future work.

5.1 Implementation

The ITC algorithm is implemented in C++ and integrated into the Kirk Temporal Planner/Executive. The algorithm is implemented as a separate stand-alone module, with its own STN representation. Thus, the module is capable of returning the temporal consistency of any input STN and does not have to be run through Kirk.

5.2 Performance

The incremental algorithm was tested on a real world cooperative air vehicle scenario, where UAVs attack two targets. In the scenario, each UAV is required to destroy two targets but has a choice between two different sets of targets. The planner must choose one set of targets for each UAV to attack. Once this choice is made, each UAV performs five activities, (1) *fly to target1*, (2) *attack target1*, (3) *fly to target2*, (4) *attack target2*, (5) *return to base.*

Figure 38 shows that the size of the plan on which the two algorithm were tested on. Notice that it grows linearly with the number of UAVs being considered because each added UAV performs a constant number of activities.

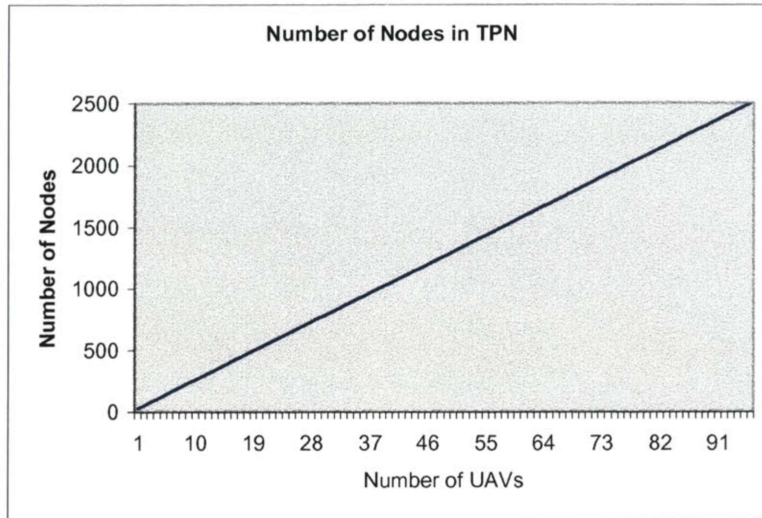


Figure 38 – Number of Nodes in TPN for MICA scenario

The data shown in Figure 38 gives the basic idea of the size the planning structure that Kirk is dealing with.

The number of UAVs versus the number of queue insertions for the corresponding plan is graphed in Figure 39. The thesis claims that the ITC algorithm reduces the amount of work a temporal consistency algorithm has to perform by examining fewer nodes than the modified label-correcting algorithm. The number of queue insertions is directly proportional to the number of nodes examined.

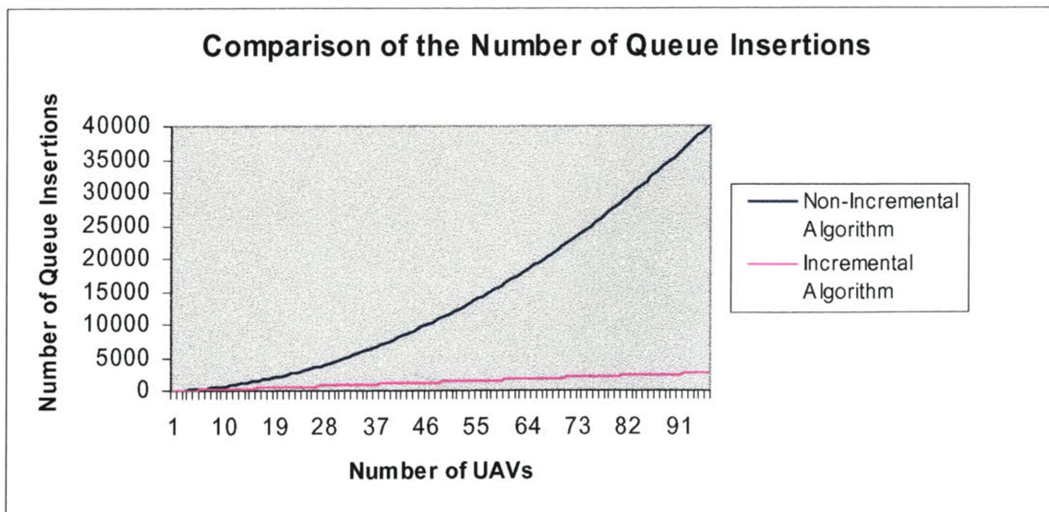


Figure 39 – Queue Insertion Comparison

Figure 39 shows that the trend for both the incremental and non-incremental temporal consistency algorithm is that the number of queue

insertions increase as the number of UAVs is increased. This is because the resulting graph of the plan given to the algorithm becomes larger and larger as shown in Figure 38. Additional UAVs add additional activity sequences, resulting in more nodes and arcs, which ultimately leads to added examination when determining temporal consistency. For the ITC algorithm, the number of queue insertions grows much slower than for the non-incremental algorithm.

Figure 40 shows the time it took the algorithms to determine the temporal consistency of each plan.

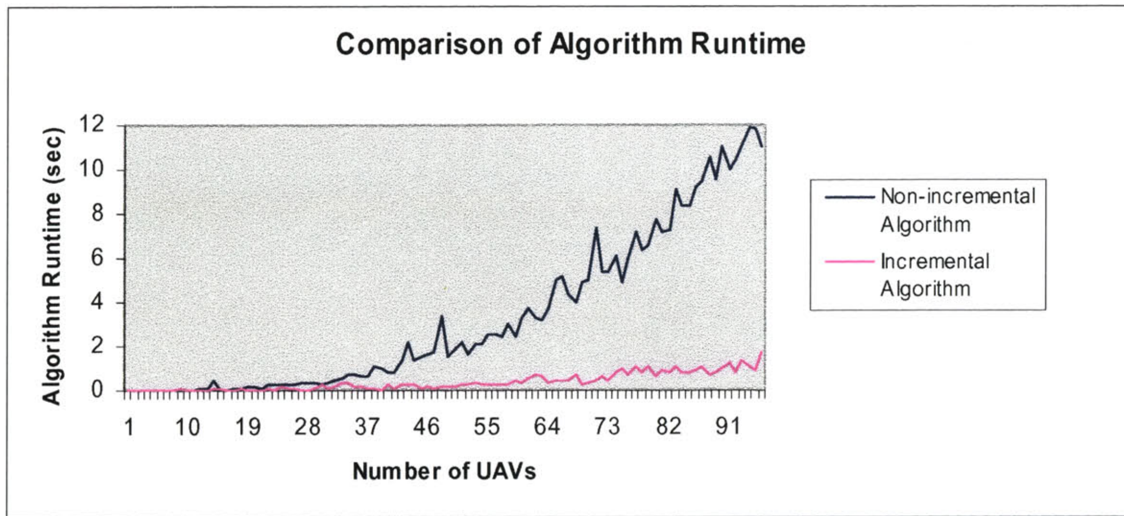


Figure 40 – Runtime Comparison

Figure 40 shows that both algorithms again increase in runtime in the same manner as they increase in the number of queue insertion when the number of UAVs in the scenario is increased. (The jagged edges in the graph corresponds to context switching between processes on the test computer.)

The graph in Figure 40 is consistent with the graph in Figure 39 because the number of queue insertions should be directly proportional to the amount of time the algorithm takes. This is because the majority of time spent determining temporal consistency is in examining nodes from the queue. Again, the ITC algorithm has a much slower rate of growth.

Both graphs show at least an order of magnitude improvement on ITC vs repeated FIFO label-correcting algorithm on this particular planning domain.

5.3 Future Work

The future work that needs to be performed can be broken up into three sections, (1) ITC implementation work, (2) ITC evaluation work, and (3) innovative ideas that improve ITC further.

5.3.1 ITC Implementation Work

The interface into Kirk translates the Kirk TPN planning object with selected activities and converts it into a complementary C++ data structure that the ITC algorithm can understand. This is not the optimal method to integrate the algorithm into Kirk since the translation process takes a significant amount of time, especially on very large input graphs. However, this implementation is sufficient to evaluate the performance of Kirk over the current temporal consistency algorithm, FIFO label-correcting algorithm. The ITC algorithm will need to be integrated into Kirk so that Kirk can take full advantage of ITC's capabilities.

The interface into the Kirk plan selection process is also yet to be implemented. Interfaces and protocols at both the Kirk plan generation and ITC conflict extraction phase first need to be determined.

5.3.1 ITC Evaluation Work

It would be very interesting to see if the conflict returned by the ITC algorithm is capable of significantly speeding up the plan generation step by focusing the search. However, the algorithm for how the Kirk planner would resolve inconsistencies has not yet been determined. This algorithm involves deciding which decisions to change given the set of temporal conflicts.

A true random plan generator needs to be implemented so that performance evaluations on ITC are not so domain specific. Currently, the plans tested on the ITC program increased breadth of the graph as the size of the graph increases. More insight on the performance of ITC might be gained by evaluating larger sized graphs with increased depth.

5.3.1 ITC Improvement Work

For the ITC algorithm, in the case where a shortest-path distance value becomes invalidated, many nodes are added to the queue because both nodes

containing distance values supported by this changed shortest-path, and the parents of these nodes need to be re-reexamined. With this many nodes, the ITC algorithm could achieve additional time savings by being able to determine which node would be best to be examined first. There is an ordering of these nodes for which the ITC algorithm performs the least amount of arc and node examinations. The ITC algorithm currently uses a FIFO queue, which just pops the node that has been in the queue the longest. However, a use of a priority queue could be provide additional speed savings.

5.3 Conclusion

The ITC algorithm has been shown to be capable of enabling continuous temporally flexible planning. Based on the results shown in Section 5.1, ITC achieves fast temporal reasoning by reusing work as demonstrated with Figure 39, graphing the number of queue insertions. This is demonstrated empirically by the order of magnitude cost savings that ITC has over non-incremental algorithms for temporal consistency.

ITC combines a fast shortest-path and negative cycle detection algorithm from network optimization along with the incremental update rules based from incremental algorithms such as Incremental A* and truth maintenance systems. This allows ITC to quickly determine the temporal feasibility of a candidate plan, thus speeding up the verification phase of a temporally flexible planner.

Additionally, ITC also guides temporally flexible planners to choose candidate plans that are more likely to be temporally consistent, by returning to the planner the minimum set of temporal constraints, or conflict, that caused a previously considered candidate plan to be temporally inconsistent. The plan generation phase of a temporally flexible planner can then use this information to bias the search against candidate plans that contain the conflicting temporal constraints.

References

[1] R. Ahuja, T. Magnanti, J. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, 1993.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest. *Introduction to Algorithms*. MIT press, Cambridge, MA, 1990.

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

[4] J. Doyle. A Truth Maintenance System. *Artificial Intelligence* 12 (1979):231-272

[5] T. Estlin, G. Rabideau, D. Mutz, S. Chien. Using Continuous Planning Techniques to Coordinate Multiple Rovers. *Electronic Transactions on Artificial Inttligence,* 4:45-57, 2000.

[6] K. Forbus and J. de Kleer. *Building Problem Solvers.* The MIT Press, 1993.

[7] P. Kim, B. Williams, and M. Abrahmson. Executing Reactive, Model-based Programs through Graph-based Temporal Planning. In *Proceedings of IJCAI-2001,* Seattle, WA, 2001.

[8] S. Koenig and M. Likhachev. Incremental A*. In *Advances in Neural Information Processing Systems 14,* 2001.

[9] D. McAllester. Truth Maintenance. In *Proceedings of AAAI-90*, 1990, 1109-1116.

[10] G. Rabideau, R. Knight, S. Chien, A. Fukunaga, A. Govindjee. Iterative Repair Planning for Spacecraft Operations in the ASPEN System.

International Symposium on Artificial Intelligence Robotics and Automation in Space (ISAIRAS), Noordwijk, The Netherlands, June 1999.

[11] P. Riley and M. Veloso. Planning for Distributed Execution Through Use of Probabilistic Opponent Models. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning Systems*, Toulouse, France, April 2002.

[12] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[13] I. Tsmardinos, N. Muscettola, and P.Morris. Fast transformation of temporal plans for efficient execution. In *AAAI-98*, 1998.

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

[15] A. Stentz. *Optimal and efficient path planning for partially known environments*. In Proceedings of IEEE International Conference on Robotics and Automation, May 1994.

[16] N. Muscettola, P. Morris, B. Pell, and B. Smith. Issues in temporal reasoning for autonomous control systems. In *Autonomous Agents,* 1998.