Unifying Model-Based Programming and Randomized Path Planning Through Optimal Search

by

Aisha Walcott

Submitted to the Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

MASTER OF SCIENCE
IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
AT THE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2004

Author _____
Department of Electrical Engineering and Computer Science
May 2004

Certified by _____
Brian Williams
Associate Professor of Aeronautics and Astronautics
Thesis Supervisor

Accepted by _____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Unifying Model-Based Programming and Randomized Path Planning Through Optimal Search

by

Aisha N. Walcott

Submitted to the
Department of Electrical Engineering and Computer Science on

May 2004

In partial fulfillment of the requirements for the degrees of
Master of Science in Electrical Engineering and Computer Science

Abstract

The deployment of robots at the World Trade Center (WTC) site after September 11, 2001, highlighted the potential for robots to aid in search and rescue missions that pose great threats and challenges to humans. However, robots that are tele-operated and tethered for power and communication are restricted in terms of their operational area. Thus, rescue robots must be equipped with onboard autonomy that enables them to select feasible plans on their own, within their physical and computational limitations. There are three main characteristics that a rescue robot's onboard system must posses. First, the system must be able to generate plans for mobile systems, that is, plans with activities and paths. Second, in order to operate as efficiently as possible, particularly in emergency situations, the system must be globally optimal. Third, the system must be able to generate plans quickly.

This thesis introduces a novel autonomous control system that interleaves methods for spatial and activity planning, by merging model-based programming with roadmap-based path planning. The primary contributions are threefold. The first contribution is a model that represents possible mission strategies with activities that have cost and are constrained to a location. The second is an optimal pre-planner that reasons through the possible mission strategies in order to quickly find the optimal feasible strategy. The third contribution is a unified, global activity and path planning system. The system unifies the optimal pre-planner with a randomized roadmap-based path planner, in order to find the optimal feasible strategy to achieve a mission. The impact of these contributions is highlighted in the context of an urban search and rescue (USAR) mission.

Thesis Supervisor: Brian Williams

Title: Professor of Aeronautics and Astronautics

Acknowledgements

I would like to thank my family for their love and support. I especially would like to thank Reginald Bryant for his support.

I would like to thank my research advisor Brian Williams for his technical guidance. I would to thank the MERS (Model-based Embedded and Robotic System Group) members past and present. I would especially like to thank Robert Ragno, Andreas Wehowsky, Melvin Henry, John Stedl, Jon Kennel, and Seung Chung. I would also like to thank Margaret Yoon, and all the members of ACME.

Finally, this research would not be possible without the mentorship and direction of Mark Smith.

This thesis was supported by the Lucent CRFP Fellowship Program, and in part by MURI Airforce grant.

Contents

List of Figures

List of Equations

Chapter 1

Introduction

Today's emergencies, such as search and rescues, natural disasters, and fires, continue to pose great challenges and threats to rescuers and emergency personnel. The deployment of robots at the World Trade Center site after September 11, 2001, highlighted the potential for robots to aid in rescue missions. Robots can be constructed to operate in conditions that are hazardous and inaccessible to humans. Since the September 11th event, the Federal Emergency Management Agency (FEMA) and the robotics community have embarked on a joint effort to identify scenarios for which robots can be used to aid rescuers in certain emergency situations [24]. The focus of this thesis is to equip robots, used in emergency missions, with greater autonomy.

To quickly determine the set of activities that can be accomplished, a robot must employ automated reasoning and decision-making techniques in order to select valid strategies. Strategies are mission plans that encode high-level goals. They are comprised of one or more sets of activities that a robot or team of robots can execute in order to achieve the mission.

Many autonomous systems use automated reasoning to select a mission strategy or plan. Traditionally, activity and path planning exist in a decoupled form, as a two-step, feed forward process. A plan of activities is generated with only crude knowledge of movement. Then path planning is performed for a particular plan, without considering other options that could achieve the same objective. Although that two-stage process may find that no solution exists for a particular plan, without a backtrack mechanism, such a system would be incomplete. That is, other valid plans would not be explored if the activity planner is unable to backtrack to another option. One solution is to allow the activity planner to backtrack to the next plan, until it finds a feasible strategy. The difficulty, however, is that without a notion of cost, this decoupled approach could

produce a plan that is highly sub-optimal. Thus, we propose a unified system that uses activity and path costs along with a backtracking mechanism in order to generate optimal mission strategies.

This thesis introduces a novel autonomous system that interleaves methods for automated reasoning with spatial reasoning, by merging model-based programming with roadmap-based path planning. The primary contributions are threefold. The first contribution is a model that represents possible mission strategies with activities that have cost and are constrained to a location. The second is an optimal pre-planner that reasons through the possible mission strategies in order to quickly find the optimal feasible strategy. The third contribution is a unified activity and global path planning system. The system unifies the optimal pre-planner with a randomized roadmap-based path planner in order to find the optimal feasible strategy to achieve a mission. The impact of these contributions is highlighted in the context of an urban search and rescue (USAR) mission.

1.2 Application

Everyday emergency personnel are required to render aid in search and rescues, natural disasters, crime scenes, and many other dangerous situations. Frequently, these situations are hazardous, fatiguing, and oftentimes life-threatening for all those involved, with the most important factor being time. Generally, rescuers need to retrieve live victims within 48 hours to increase their chances of survival [32].

Robots have demonstrated their efficacy when placed in real, unpredictable, and high-risk conditions. Small robot rovers aided the rescue efforts at the World Trade Center (WTC) in New York City [32]. The small robots searched for survivors, investigating ditches where neither dogs nor humans could reach. The robots were controlled by joystick and were equipped with cameras, heat sensors (thermal cameras), microphones, and two-way radios for communication between victims and emergency personnel. The robots found several bodies and a set of human remains [26]. In addition, the FBI considered using an autonomous helicopter to map the terrain of the region in Pennsylvania where one of the planes went down on September 11, 2001 [31].

Many of the robots used in the rescue efforts at the WTC were tele-operated and tethered for power and communication. Operators were restricted by the field of view of the cameras, which created problems with localization [5]. These issues emphasize the need for "automatic capabilities intrinsic to the robot" [5] that would relieve tele-operators from having to control the robots and search for victims at the same time. In addition, it is unreasonable to rely on personnel training rescue to operate (fly or drive) robots. Therefore, rescue robots must be equipped with onboard autonomy that enables them to select feasible plans within their physical and computational limitations.

Autonomous vehicles (AVs) offer a number of benefits. They can communicate with each other and with rescue personnel, and can navigate regions that are inaccessible to humans, significantly improving the search and rescue process. Furthermore, AVs can apply vision, microphone, and other sensing technologies to explore hazardous areas, including wreckage, burning buildings, and toxic waste sites. Robots can be constructed to survive fire, thick smoke and dust, water, and sharp piercing materials [5].

1.2.1 Urban Search and Rescue Scenario

In order to prevent harm to humans, our goal is to devise technologies that allow heterogeneous teams of robots to coordinate and cooperate in urban search and rescue missions. Currently, during the initial phase of an urban search and rescue, a control center is setup and a preliminary reconnaissance is performed by the *first-responders*—emergency personnel who initially arrive on the scene [24]. We envision this as an ideal opportunity to deploy robots into unknown and potentially hazardous areas that are unsafe for the first responders. The robots would search for victims, detect chemicals, and sense and collect data [24]. In the spirit of RoboCup Rescue, a number of credible USAR scenarios have been proposed [24][32]. To ground the importance of planning for rescue robots, an example scenario is described below (Figure 1). The USAR scenario highlights the importance of activity planning and AV path planning.

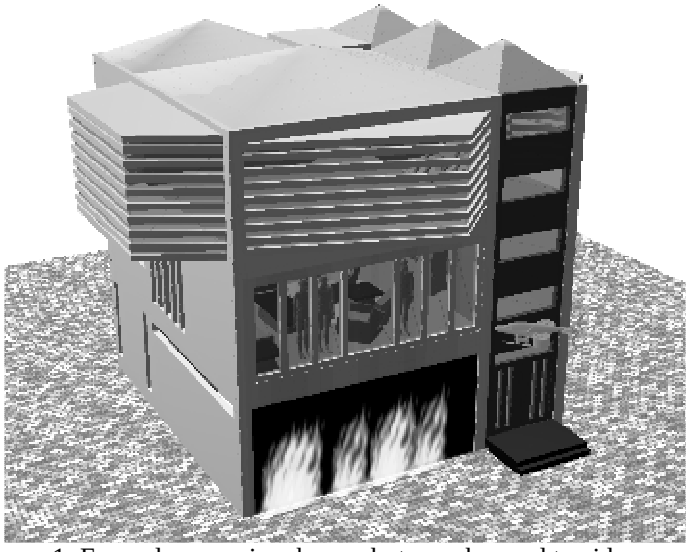


Figure 1: Example scenario where robots can be used to aid rescuers.

Urban Search and Rescue Scenario

It's 4pm, and sirens are roaring through the city as fire engines race to the scene of a burning office complex. The first few hours are critical to the survival of victims, and pose the greatest threat to rescue workers. After extinguishing the fire in one area of the complex, firefighters survey the potential dangers of entering the building. At this time, robots are summoned to explore the scene and begin searching for victims. An agile autonomous helicopter, called ANW1, is sent to aid the mission. ANW1 is carrying a team of small robots that will aid in the USAR mission. The team of small robots is divided by their sensing capabilities. One team, the *chembots*, is used to detect chemicals and monitor the environment. The other team, the *helpbots*, is used to bring first aid packages and a means for communication to the victims.

The location of the fire is provided as input to the helicopter, which then generate a plan to get to the scene. ANW1 has an *a priori* map of the city that it uses to plan paths to the office complex. Once it arrives on the scene, it waits for instruction from the control center.

After communicating with the control center, ANW1 uses its onboard camera and map of the complex to identify structural anomalies. Equipped with a range of sensors, ANW1 enters the building through a broken-out window. It flies to a drop-off location where it lands and releases a team of small robots. The robots explore a room, probing

and sensing for hazardous substances. ANW1 then navigates to a location in the building where the control center identified as having trapped victims. Once ANW1 navigates to the location, it carefully lowers *helpbots* with two-way radios and minimal first-aid supplies on the floor, enabling communication between victims and rescue workers. Then ANW1 goes back to the *chembots* and loads them into the helicopter's transport carrier. Finally, ANW1 exits the building and flies to a charging station where all the robots are recharged and their data is uploaded to the control center.

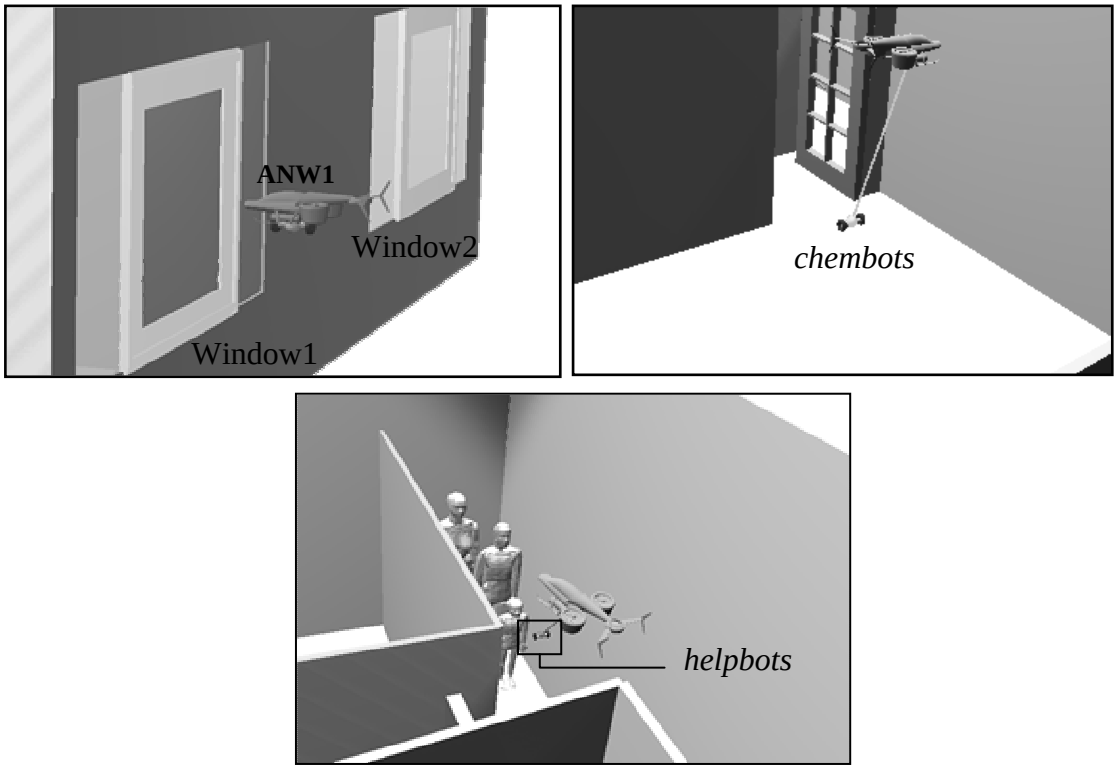


Figure 2: Example of the activities in the Search-Building mission

Figure 2 illustrates some of the major activities involved in the mission. We refer to the above mission as the Search-Building mission. The Search-Building mission is composed of a series of strategies, in this case three strategies. The first strategy, called Enter-Building, requires the AV to enter the building and deploy the robot teams, while gathering data. The second strategy, called Exit Building, requires ANW1 to deploy the *helpbots* and recover the *chembot* team, and then exit the building. The third strategy, called AtHome, requires ANW1 to upload data while recharging. The Search-Building

mission and the three strategies that comprise the mission serve as examples throughout this thesis. In the following section, we highlight the key characteristics of each strategy.

1.2.2 USAR Characteristics

There are two key characteristics of the Search-Building mission. First, a number of the activities in the mission require the autonomous vehicle, ANW1, to be in a particular location in the office complex, in order to execute an activity. For example, when ANW1 navigates and explores the building, it captures images and stores specific data gathered from its sensors. While capturing images might require the ANW1 to be in a specific region of the building, collecting data from its sensors can be done in any region of the building. Second, the set of activities chosen to accomplish the mission must make efficient use of AV resources such as, energy, fuel, and time. For example, during the Exit-Building strategy, it is more efficient for ANW1 to deploy the *helpbots* and then recover the *chembots*. In this case, ANW1 saves energy, but avoids carrying both teams of robots at the same time.

In order to encode the high-level goals that comprise a mission, such as USAR, the activities must be specified in a language that the AV understands. A class of languages called execution languages enables a mission designer to write complex procedures for a robot or team of robots to execute. Such languages are RAPs [10], ESL [12], and TDL [34]. In addition, the language must be able to describe the progression of the mission. That is, using a model of the AV, the mission should describe the steps which the AV should execute.

In order to select a feasible strategy, an activity planner can be used. The planner must be able to reason through the space of possibly feasible strategies in order to select the best strategy. A strategy is composed of activities and paths, therefore, the planner must not only plan activities, but generate paths as well. For an AV with complicated kinematics and dynamics, such as a helicopter, randomized roadmap path planning techniques have been applied [21][22][11]. These path planners construct a roadmap in the state space and connect roadmap nodes to each, in order to find a collision-free path. This is depicted in Figure 3.

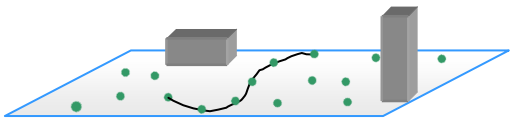


Figure 3: Example of a randomized roadmap based path planner exploring the robot's state space and the world in order to find a collision free path.

1.3 Problem

There are a number of issues that this thesis must address in order to develop algorithms for selecting an AV's plan of action, from a large space of strategies. Alternative mission strategies are encoded in an execution language. This language must be able to support activities with duration, activity costs, and location constraints for those activities that require the AV to be in a specific spatial region. The language must also be able to encode choice between functionally redundant methods in order to express alternative strategies. Given a description of a mission objective and strategies an execution language, fast planning techniques need to be developed to generate the most effective mission plan. The planning system must be able to perform both the spatial reasoning and reasoning about discrete actions. Spatial reasoning is required for those activities that are constrained to a specific region. Reasoning about action is required in order to determine the set of feasible activities.

The first objective of this thesis is to develop an optimal planner that selects the best strategy given a mission description. The second objective is to develop a unified optimal activity and path planning system that selects the best strategy for a mission that contains activities with location constraints. These algorithms are developed for single vehicle missions. However, these algorithms can be generalized to multiple AV missions (see Chapter 7).

1.3.1 Technical Approach

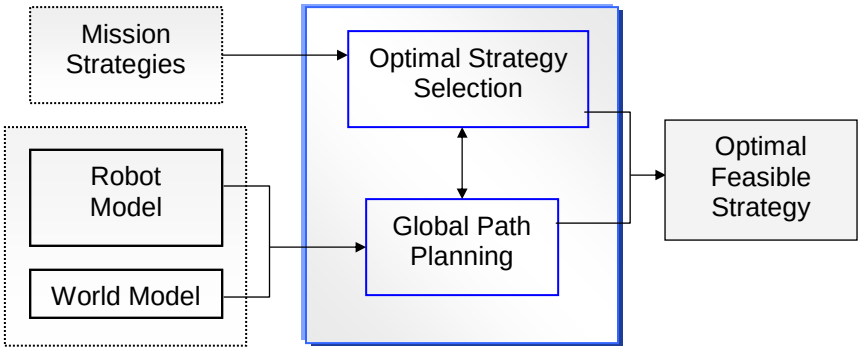


Figure 4: Overall unified activity and path planning system.

We approach the problem of finding the optimal mission strategy by merging model-based programming with roadmap-based path planning (Figure 4). Model-based programming enables the mission designer to encode a set of mission objectives, alternate strategies for achieving the objectives, and encodes models of the vehicles that perform the mission. Our first contribution is a language for planning optimal, mobile vehicle missions. This language is an extended subset of the Reactive Model-based Programming Language (RMPL) [38]. The extended RMPL subset includes activity costs and location constraints (ACLC). An example of an RMPL program is shown in Figure 5.

```
1.  (sequence
2.      ;;Choose type of vision sensing
3.      (choose
4.          ( (sequence ;;choice 1
5.              ( ANW1.Monocular-Vision(20) [10,20] )
6.              ( ANW1.Set-Compression(10, {low}) [5,10] )
7.          ) (HallwayA) [35,50])
8.          (sequence  ;;choice 2
9.              ( ANW1.Stereo-Vision(40, HallwayB) [10,20] )
10.             ( ANW1.Set-Compression(20, {high}) [8,13] )
11.         )
12.     ) ;;end choose
13.  ) ;;end sequence
```

Figure 5: Fragment of the Enter-Building program developed with the ACLC subset of RMPL.

Mission strategies encoded in RMPL are called control programs. Next, we describe a mission environment model, which has two components: 1) a physical

description of the AV involved in the mission and 2) a model of the world in which it will navigate (Figure 4).

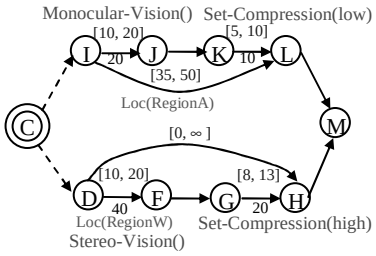


Figure 6: TPN representation of the program in Figure 5. There are two strategies for the AV to execute. The AV can either execute the Stereo-Vision activity and the Set-Compression activity, or the Monocular-Vision and the Set-Compression activity.

Given an RMPL control program, our planner maps the program to a compact graphical representation, called a Temporal Plan Network (TPN). The TPN encoding enables the planner to perform fast, online planning. The TPN model was first introduced in [40][39], and is extended here to support reasoning for missions with locations and activity costs. An example of the equivalent TPN representation of the control program in Figure 5 is shown in Figure 6.

For missions that require the vehicle to move from region to region, we extend the search space of our planner to the combined TPN and path planning space. The path planner uses the mission environment model in order to search for collision-free paths through which an AV can safely move from region to region. This is illustrated in path planning space shown below the TPN in Figure 7. We define this combined TPN and roadmap model as the Roadmap TPN (RMPTPN).

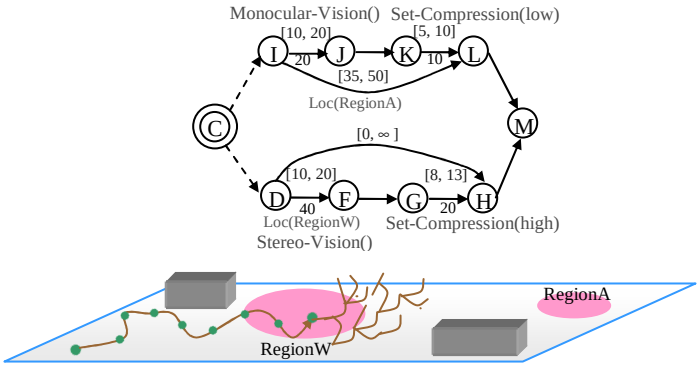


Figure 7: Unified activity and path planning space. The RMPTPN model is used to reason on the combined search spaces.

The best plan for vehicle activity and movement is then generated by a unified, globally optimal, activity and path planning system (UAPP). This system operates on the RMTPN model in order to select the best set of actions and to satisfy location constraints by planning collision-free paths in the space where the vehicle will carry out the mission.

1.4 Thesis Layout

This thesis is organized as follows. Chapter 2 provides background on three areas: execution languages and model-based programming, temporal reasoning, and randomized roadmap based path planning. Chapter 3 presents a subset of RMPL (the reactive model-based programming language) [38] that is extended to specify activity costs and location constraints (ACLC), this is referred to in this thesis as the ACLC subset of RMPL. Chapter 5 introduces a novel optimal pre-planner that operates on a TPN model in order to select the least cost strategy, according to the TPN activity costs. Chapter 6 combines this optimal pre-planner with roadmap based path planner, producing a unified, globally optimal, activity and path planning system (UAPP). Finally, Chapter 7 concludes this thesis with an empirical validation of our research and makes suggestions for future work.

Chapter 2

Background

The unified optimal activity and path planning system combines the following three areas of related work. The three areas are 1) model-based programming [37], 2) temporal planning [25], and 3) randomized kinodynamic path planning [23]. The first two have been combined to effectively encode coordinated air vehicle missions [39][33]. The third has been used to plan collision-free paths for robots with kinematic and dynamic (called "kinodynamic") constraints [22][23]. Drawing on these three areas, we develop algorithms that combine model-based programming and temporal reasoning with randomized roadmap path planning. This thesis builds on these three areas of research in order to create the unified optimal activity and path planning system.

2.1 Model-based Programming

A model-based program is a specification of the evolution of a system using a set of constructs for describing concurrent behavior. The constructs describes synchronous and constraint-based execution of the system. In this thesis, we focus on a mobile, autonomous vehicle (AV) system. The model-based program describes the high-level goals that the AV should execute during a mission.

To enable encoding of missions for autonomous and embedded systems, an execution language, called the Reactive Model-based Programming Language (RMPL), was introduced [37]. Its features include straightforward programming constructs that can be combined to describe the desired evolution of embedded and reactive system states. Additionally, RMPL has successfully been used to encode strategies for coordinated teams of autonomous vehicles [38]. Further, the language allows mission designers to express redundant methods for achieving a goal, by providing a set of constructs

necessary to describe flexible missions. The programs are compiled into a compact graphical representation to which automated reasoning techniques are applied in order to quickly select feasible threads of execution (sequences of activities that achieve the mission objectives).

2.2 Kirk Temporal Planner

This thesis uses the Kirk temporal planning framework in order to define a partial order on the activities encoded in a mission. The optimal pre-planner developed in this thesis extends Kirk by replacing the feasible search strategy with an optimal search strategy.

A temporal planner, called Kirk, was introduced in [39] and enables pre-planning of temporally flexible activities. Kirk uses RMPL constructs that express contingencies (nondeterministic choice) and temporal constraints. To perform temporal reasoning, Kirk operates on a temporal plan network (TPN) encoding of an RMPL control program.

Temporal Plan Networks express concurrent activities, decisions between activities, temporal constraints. An activity has both a start node and end node in the plan network that represents the duration of the activity. Nodes in a TPN represent an instance in time and are referred to as events. The arcs in a TPN contain temporal and symbolic constraints. The temporal constraints are given as an interval defining the lower bound L (the minimum duration of an activity) and upper bound U, denoted as [L, U], for the duration of an activity. These are expressed in the Simple Temporal Network representation summarized in Chapter 4. The symbolic constraints in a TPN are given in the form of Ask(C) and Tell(C), where C is a condition. An Ask(C) requires that a condition C be true, while a Tell(C) asserts the condition C. Symbolic constraints are specified on an arc in a TPN, and hold for the duration of that arc.

During planning, Kirk attempts to resolve temporal and symbolic constraints, while making decisions as needed. If no valid plan is found, Kirk backtracks and makes a new set of decisions that ignore the previously invalid portions of the plan space. The search process is repeated until a complete and consistent plan is found, or all candidate plans have been examined and no valid solution exists [39]. If Kirk finds a complete and

consistent plan then the plan is sent to a plan runner, which performs interleaved scheduling and execution, while adapting to execution uncertainties.

2.3 Path Planning

The unified activity and path planning system described in this thesis, requires a roadmap based path planner in order to find paths for an AV that is required to navigate from location to location. In this section, we review the general approached to finding a collision-free path for robots.

2.3.1 Overview

The general path-planning problem asks how a mobile robot can safely move from location A to location B, while avoiding obstacles. This solution can be found by asserting the start location and the goal location, and by employing a path planner that generates an obstacle-free trajectory from start to goal. Two key features of path planning algorithms are completeness and optimality. A path planner is complete if it finds a collision- free path when one exists; otherwise, it returns no solution. A path planner is optimal if it returns the shortest, or best, path from start to goal state [28]. There are several deterministic and non-deterministic (randomized) approaches to the path-planning problem, including mixed integer linear programming [2], convex cells [21], approximate cell decomposition [21], potential field methods [21], the freeway method [21], and roadmap methods [35][21].

Computing a collision free path for robots with complex dynamics has proven to be computationally hard. A path planner must explore a large state space that represents each dimensions of the robot. Complete path planners take an indefinite amount of time (depending on the problem size) in order to compute a collision-free trajectory and are usually used to solve small problems, that is, problems involving robots with only a few degrees of freedom. To avoid computationally time intensive path planning, a number of efficient randomized techniques have been explored. For example, the probabilistic roadmap path planner randomly generates nodes in the world in which the robot navigates (Figure 8). The nodes are in the obstacle-free space of the world. Completeness

for randomized path planners is referred to as probabilistic completeness; that is, if a solution exists, then it will find a path with high-probability [13].

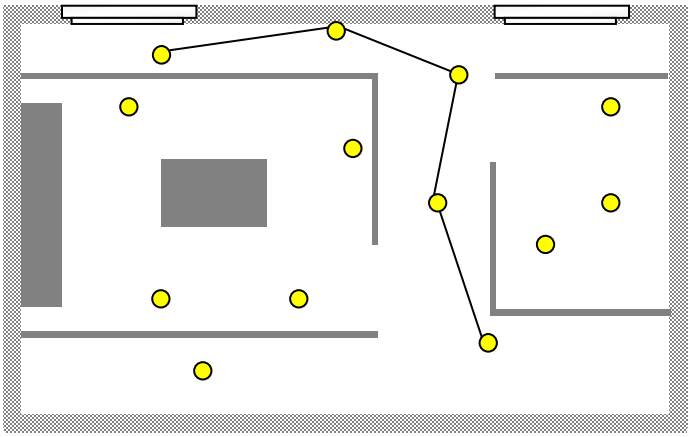


Figure 8: Example of randomly generated roadmap nodes used to find collision free paths in a space.

Rapidly-exploring Random Trees (RRTs) were introduced as a data structure that can quickly explore the state space of robots with kinematic and dynamic constraints—termed *kinodynamic* constraints. For example, a helicopter has inertia, and thus, this must be accounted for when planning a path for it. An RRT-based planner attempts to grow tree in the search space, from the initial state to the goal state (Figure 9). In our USAR mission we use a small helicopter, and thus, adopt an RRT-based path planner in order to satisfy location constraints in a model-based program.

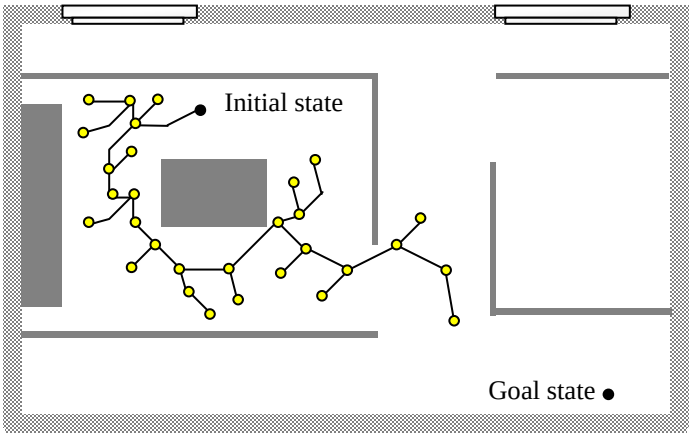


Figure 9: Example of an RRT exploring the state space.

2.3.2 Kinodynamic Path Planning

Kinodynamic path planning is an emerging area of research that explores path planning for complex robots with kinematic and dynamic constraints. Most often mobile robot motion is constrained by its velocity and acceleration [13][22]. Kinodynamic planning refers to the class of problems in which the motion of a robot must satisfy nonholonomic and/or dynamic constraints [22]. Nonholonomic systems are systems with fewer controllable degrees of freedom than total degrees of freedom.

A recent effort to solve a large class of kinodynamic problems, called Rapidly-exploring Random Trees (RRTs), has shown significant success with path planning for robots with a large number of degrees-of-freedom and complicated system dynamics [22][23].

To accurately encode the constraints of a robot, its equations of motion must be given. Equations of motion are a system of equations that govern the robot's motion. They are generally of the form $\dot{s} = f(s,u)$, where s is a state of the robot, $\dot{s}$ is its derivative with respect to time, and $u \in$ U (the set of all possible control inputs) is the control input(s) applied to the robot [22].

For non-complex robots, the path-planning problem can be solved using the robot's configuration space— all possible position and orientation pairs that describe the robot is reference to a fixed coordinate system [13]. Robots generally have physical limitations on their motion; thus, solving the path-planning problem in the configuration space does not suffice. Trajectories (collision-free paths) generated in the configuration space do not account for dynamic constraints on the robot's movement [22]. In the following section, we summarize the kinodynamic state space formulation given in [22].

State Space Formulation

The state space, denoted as X, is the search space for which kinodynamic path planners attempt to find a collision-free trajectory for a robot. Expressing the entire state, *s*, of a robot, in general, dramatically increases the dimensionality of the entire search space. Hence, most path planners approximate X by restricting the dimensionality. Systems described in the state space encode a state, $s \in$ X, as the robot's position (or

configuration) and the derivatives of the position (i.e., velocity). Thus, the configuration space is a subset of the state space.

The state space consists of the obstacle regions, X_{obst}, and free space regions, X_{free}. X_{obst} is the subset of X corresponding to states that would result in a collision between the robot and an obstacle. X_{free} represents the remaining set of states that result in no collision— i. e., the space where the solution trajectory must solely reside. A more detailed development of the state space formulation and equations of motion can be found in [22].

To incrementally construct a path, RRT-based kinodynamic planners integrate over the equations of motion and generate a path in the state space [22]. Given the current time *t* and the robot's current state *s*, by applying the selected control(s) *u* over the time interval [t, t+δ], then the state of the robot at time t+δ will be

Equation 1:

$$s(\mathrm{t}+\delta)=s(t)+\int_{t}^{\mathrm{t}+\delta} f(s(t),u)dt.$$

A solution is precisely defined in [22] as a time-parameterized (between [0, T_{final}]), continuous, collision-free trajectory from an initial state, $s_{init} \in X$, to a goal state (or goal region) $s_{goal} \in X$ that satisfies the robot's physical constraints. An input function for which each instant of time in [0, T_{final}] is mapped to its corresponding control(s), *u*: [0, T_{final}] $\rightarrow$ U, which results in a collision-free trajectory from start to goal. The trajectory at time $t \in T_{final}$ and state *s*, s(*t*) is determined by integrating:

Equation 2:

$$s(t) = \int_{0}^{t} f(s,u) .$$

2.3.2 Rapidly-exploring Random Trees (RRTs)

RRT-based planners are randomized incomplete planners that scale well for kinodynamic problems with high degrees-of-freedom and complicated system dynamics [10]. A key feature of RRTs is that they uniformly explore the robot's state space and, therefore, are heavily biased towards unexplored regions in the state space. RRTs incrementally

construct a roadmap (directed tree) in the robot's state space by applying control inputs to existing roadmap nodes. In this case, a connected roadmap is considered a RRT. A solution consists of a collision-free trajectory expressed as a finite sequence of robot states and corresponding control inputs [22] that drive the robot from state to state.

Constructing an RRT An RRT is best represented as a directed tree data structure. A tree, a directed graph with no cycles, is built in the robot's state space. All nodes in the tree have zero or more outgoing arcs (edges), each corresponding to another node called a child node. A child node can, in turn, have several children. All nodes, other than the root, have exactly one incoming arc deriving from its parent. The root of the tree has no parent.

RRTs have been adapted as kinodynamic path planners by storing specific information in RRT nodes and arcs. Data stored in an RRT node consists of a state and a list of children. Data stored in an RRT edge are the control inputs associated with that arc (form the transition between head node and tail node). The RRT is grown in the robot's state space. Nodes are added to the tree by integrating the equations of motion over a specified time interval, starting from some pre-existing node in the tree.

The general RRT construction is described as follows. A point is randomly generated in space and the nearest node (nearest neighbor) in the existing RRT to the random point is selected. Then, from the nearest node, the tree extends one step, by some pre-specified distance, towards the random point and adds a new arc and new node (a child node of the nearest node) at the end of the arc. This process is repeated by generating another random point in space and continuing with the steps described above until the maximum number of nodes is reached [22][23]. Figure 10 is an example of an RRT grown in free space with a path from start to goal.

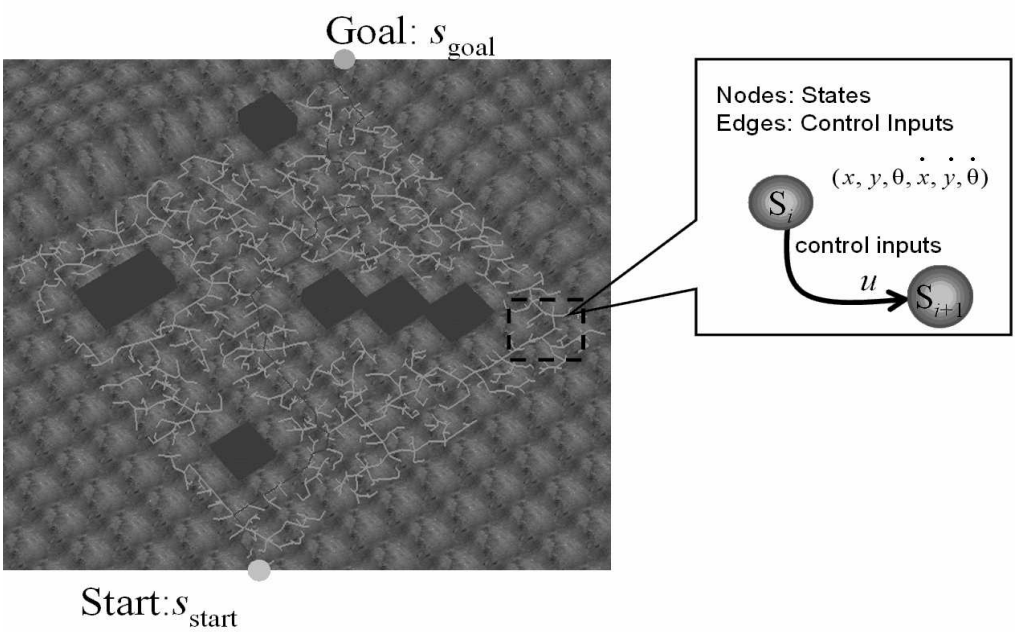


Figure 10: Example of an RRT growing from the start state to the goal state.

When extending the nearest neighbor node towards the random point, a collision test must be performed. If the edge from the nearest node to the new node collides with an obstacle then the edge is not added to the RRT. A number of efficient algorithms for incremental collision detection can be applied [22].

The algorithm terminates when the maximum number of iterations is reached or a solution is found. A valid solution is determined by testing the distance between the newly added node and the goal. The test is performed after each iteration and if s_{new} and s_{goal} are sufficiently close enough then the path is returned. A solution trajectory is established by working backwards from s_{goal} to s_{start} and recording the states and their corresponding controls.

Path planners that account for kinematic and dynamic constraints are important for agile autonomous vehicles used in urban search and rescues. Roadmap-based path planners naturally lend themselves to the compact graph-based planning model used in our planning system. The system employs a roadmap-based kinodynamic path planner to find collision-free trajectories and satisfy location constraints.

We draw on the areas of model-based programming, temporal planning, and roadmap based path planning in order to develop a globally optimal activity and path planning system.

Chapter 3

Model-based Programming for Autonomous Vehicles

We use the model-based programming approach to describe strategies that an autonomous vehicle (AV) must execute in order to achieve its mission. A strategy is a set of actions that the robots execute in order to achieve the mission goals. Adopting the model-based programming approach requires three key pieces of information. The first is a description of the possible strategies that make-up a mission. The second is a physical description of the AV involved in the mission. And the third is a model of the physical world in which the strategies are executed. These three pieces of information are specified within one of the two components that comprise a model-based program: a *control program* and an *environment model* [37]. In particular, mission strategies are specified in a control program, while the vehicle and world models are specified in an environment model.

A control program uses the constructs from the Reactive Model-based Programming Language (RMPL) [37] to encode strategies that contain activities, activity costs, location constraints, and simple temporal constraints. In general, activity costs are a function of the AV's resources and the environmental constraints, such as fuel or energy. An environment model contains a description of the world, where the mission will be carried out and a physical description of the AV operating within the world. The control program and environment model, together, are input to the unified activity and path planning system.

This chapter describes the encoding of a control program and environment model, used in unified optimal activity and path planning. We present a derivative of RMPL for

coding mission strategies, which contains activity costs and location constraints (ACLC). This derivative of RMPL is referred to as the ACLC subset of RMPL. Moreover, we define an encoding for an environment model that is used for planning collision free paths.

3.1 Control Program and Environment Example

An example scenario, where both activity planning and path planning are critical, is detecting a chemical leak. Strategies for this scenario are specified in the Enter-Building control program shown in Figure 11. In this scenario, a small autonomous helicopter, ANW1, is sent into a building, where it lowers its chemical robot team, *chembots*. The *chembots* then use their sensing technologies in order to detect the source of the chemical leak. The world where ANW1 will be navigating is shown in Figure 12. The gray areas represent obstacles that the robot must avoid. The two release points, ReleasepointA and ReleasepointB, are shown with black circles. The figure depicts a 2-D top-down floor layout, which is sufficient if ANW1 flies on a 2-D plane at a constant height. Otherwise, a 3-D representation of the floor plan must be provided.

Enter-Building Control Program

In the Enter-Building control program, the behavior of the autonomous helicopter, ANW1, is described using ACLC primitives. The mission goals are as follows. The first goal is to start ANW1's vision system. This is achieved by using either the monocular vision (Lines 5-8, Figure 11) or the stereo vision system (Lines 9-12). The next goal is for ANW1 to navigate from HallwayB to LaboratoryOne, while taking pictures (Lines 15-22). Once ANW1 arrives at the laboratory, it flies to a release point where the small, chemical-detecting robot team is lowered (Lines 24-27).

Standard RMPL constructs, such as `sequence`, `parallel`, `choose`, and temporal bounds, are used in creating the Enter-Building control program. The standard RMPL constructs are augmented by specifying cost and location constraints for each activity. For example, the Lower-Chembots activity on Line 25 has a cost of 65 units and is constrained to the region defined by ReleasepointA. Also, a location constraint can be

specified without an activity, as seen on Line 17. If a cost or location is not specified, then their default value is assumed. The default cost is 0 units and the default location is ANYWHERE, which refers to any region in the space. The addition of activity costs and locations to RMPL activities greatly enhances the expressivity of RMPL with respect to describing complex mission goals for mobile systems.

Enter-Building RMPL control program

```
1.  (Enter-Building
2.      (sequence
3.           ;;Choose type of vision sensing
4.           (choose
5.                ( (sequence ;;choice 1
6.                      ( ANW1.Monocular-Vision(20) [10, 20] )
7.                      ( ANW1.Set-Compression(10, {low}) [5, 10] )
8.                ) (0, HallwayA) [35, 50])
9.                (sequence  ;;choice 2
10.                   ( ANW1.Stereo-Vision(40, HallwayB) [10, 20] )
11.                   ( ANW1.Set-Compression(20, {high}) [8, 13] )
12.               )
13.           ) ;;end choose
14.          ;;Navigate from Hallway B to the Laboratory and take pictures
15.          (parallel
16.              (sequence
17.                  ( ANW1(HallwayB) [0, 0] )
18.                  ( ANW1.noOp() [5,+INF])
19.                  ( ANW1(LaboratoryOne) [0, 0] )
20.               )
21.               ( ANW1.Take-Pictures(50) [5, 50] )
22.          ) ;;end parallel
23.          ;; At the Laboratory lower the Chembots
24.          (choose
25.               ( ANW1.Lower-Chembots(65, ReleasepointA) [15, 25] )
26.               ( ANW1.Lower-Chembots(50, ReleasepointB) [10, 30] )
27.          ) ;;end choose
28.      );;end sequence
29.  );;end Enter-Building
```

Figure 11: The Enter-Building control program. Temporal constraints are show in brackets "[]", and activity costs and location constraints are specified with the activities using "( )". If temporal constraints, activity costs, or location constraints are not specified, then their default value is assumed. The default for a temporal constraint is `[0,+INF]`. The default value for an activity cost is 0, and the default for a location constraint is ANYWHERE.

Enter-Building World

A top-down 2-D view of the world where ANW1 will be navigating is shown in Figure 12. The occluded regions are shown in gray rectangles, labeled O_1-O_{10}, and the frame of references is shown in the lower left corner of the figure. A top-down view of ANW1 is shown at the top of the figure. The sphere with radius *r*, surrounding ANW1, represents

an estimate of ANW1's dimensions. ANW1 can enter the building through either Window1 or Window2 in order to carry out its mission.

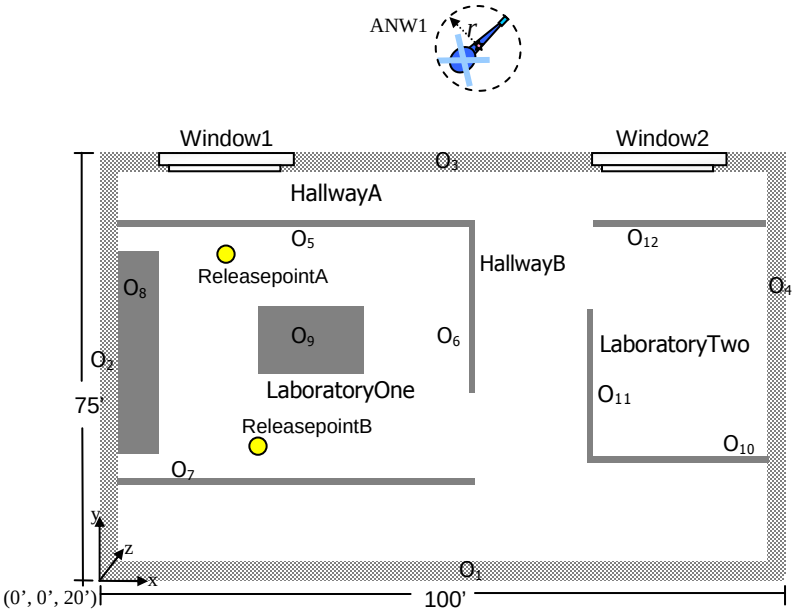


Figure 12: The world in which ANW1 will navigate in order to execute a strategy from the Enter-Building control program. The occluded regions are shown in gray. ANW1's dimensions are estimated by a sphere of radius *r*.

The world is encoded as an environment model, which is then transformed into its configuration space (or state space) **Error! Reference source not found.**. The path planner operates on the transformed model in order to find collision-free paths from location to location. In the following sections, we define the specifications for control programs and environment models.

3.2 Supported RMPL Specification

Control programs are written in RMPL and describe the high-level goals that AVs must achieve. A strategy is an execution of the RMPL program that achieves all the mission goals. More than one strategy can be specified by the `choose` combinator and functionally redundant methods. The ACLC subset of RMPL provides an efficient way to encode strategies in which the success of the mission critically depends on the costs of executing activities and navigation to multiple locations. Figure 13 shows the grammar for the ACLC subset of RMPL.

Expr → *Expr*(<**location**>)[ub,lb]| *Activity* | *Target*(<**location**>)[ub,lb] |
sequence(*Expr+*) | choose(*Expr+*) | parallel(*Expr+*)

Activity → *Target*.activityname(<**cost**>,<**location**>,{parameters})[ub,lb]

Target → robotname

Figure 13: The grammar for the ACLC subset of RMPL.

In the ACLC grammar, `location` is an element of the domain of locations that an AV might visit. The symbol "*Expr+*" refers to one or more expressions. The specification of activity costs and location constraints are highlighted in Figure 13 with bold type. Angle brackets "<>" signify that activity costs and location constraints are optional. The `parameters` associated with an activity is a list (which can be empty) of arguments to the activity. Temporal constraints are in the form `[lb,ub]`, where `lb` and `ub` are the lower and upper time bounds, which restrict the duration of a constraint, activity, or a combination of the two.

3.1.1 Description of Primitives

This section provides a detailed description of the ACLC primitives presented in the grammar. Activities and location constraints are referred to as commands. Commands can be recursively combined with the `sequence`, `parallel`, and `choose` combinators, in order to express complex mission strategies.

Primitive Commands

- *Target*`.activityname(cost,location,{parameters})[lb,ub]:`
  This expression depicts a primitive activity (or command) that the *Target* "knows" how to execute. It is labeled with an activity name, along with the corresponding cost, location constraint, and one or more parameters, specified in the `parameters`. The minimum and maximum time allowed to execute the activity is specified by temporal constraints `[lb,ub]`. The estimated cost of executing the activity and the activity's location constraint can be specified, but are not required. In general, we assume the default cost is 0, and the default location is ANYWHERE. An example of a primitive activity is `(ANW1.Set-`

`Compression(10,{low}) [5, 10])`. Its cost is 10 units and has a parameter `low`. The command requires the target vehicle, ANW1, to execute the Set-Compression activity ANYWHERE for at least `5` time units, but no more than `10` time units.

- *Target*`(location)[lb,ub]`: A location constraint asserts the region where *Target* must reside, for a duration of at least `lb` time units and at most `ub` time units. For example, `ANW1(Laboratory)[25,100]` asserts that ANW1 is at the laboratory for at least 25 time units and at most 100 time units. The process of satisfying location constraints is detailed in Chapter 6.

Primitive Combinators

- `sequence(`$c_1, c_2, \ldots, c_n$`)`: The sequence combinator is used to create RMPL expressions in which primitive commands or expressions denoting compositions of commands $c_1, c_2, \ldots, c_n$ should be performed in order. Lines 5-8 in Figure 11 are an example of a sequence of commands that are to be executed in the order in which they are presented. That is, ANW1 executes the `Monocular-Vision` activity before it executes the `Set-Compression` activity.

- `parallel(`$c_1, c_2, \ldots, c_n$`)`: The parallel combinator is used to encode concurrent threads of execution, where $c_1, c_2, \ldots, c_n$ are expressions that are executed in parallel. For example, the parallel expression on Lines 15-22 in Figure 11, requires ANW1 to execute the sequence of commands on Lines 16-20, while taking pictures (Line 21). The execution of parallel threads is complete when all commands on all the threads composed within a parallel expression have been executed.

- `choose(`$c_1, c_2, \ldots, c_n$`)`: The choose combinator is used to model decision theoretic choice between different threads of execution. Only one thread in a choose expression is selected for execution. The optimal pre-planner selects the thread whose combined activity costs are a minimum. A choose expression is

shown in Figure 11 on Lines 4-13. In this expression, ANW1 executes either the `Monocular-Vision` and the `Set-Compression` activities in sequence, or the `Stereo-Vision` and `Set-Compression` activities in sequence. The optimal pre-planner performs an informed search in order to decide which thread in a choose expression to execute (see Chapter 5).

3.1.2 Temporal Constraints

To encode strategies in a control program with flexible time bounds, temporal constraints are used. Lower and upper bound times are specified for a command or a composition of commands using the form `[lb,ub]`. `lb` is a positive number, representing the minimum time an AV must take to execute a command, and `ub` is a positive, number representing the maximum time allowed for executing the command. Note that `ub` must be greater than or equal to `lb`. If no temporal constraint is given, then the time bound is assumed to be `[0, +INF]`, this places no restriction on the duration of a command or composition of commands. For example, the expression defined by the sequence on Lines 9-12 of the Enter-Building control program does not contain a temporal constraint; thus, the activities within the sequence are constrained only by their respective temporal bounds. It is also possible, however, to constrain the duration of commands by adding time bounds on the combinator that surrounds the commands. For example, the sequence of commands on Lines 5-8 are further constrained by a `[35,50]` temporal bound. The technique used to test for temporal consistency is detailed in Chapter 4.

3.1.3 Location Constraints

In general, a location constraint is used to constrain the execution of an activity to a specific region. The statement `Target.Activity(region)` is interpreted as that `Target` must be in `region` throughout the entire execution, from start to end of `Activity`, and no where else. For example, `ANW1.Stereo-Vision(40, HallwayB)` states that for the duration of the `Stereo-Vision` activity ANW1 must remain in HallwayB. Like temporal bounds, location constraints can be used to constrain a set of activities to a specific region. This is achieved by constraining activities within an expression to a location. For example, the activities in the sequence on Lines 5-8 are

constrained to HallwayA. Additionally, to require an autonomous vehicle (AV) to be in the region, the target AV and region can be asserted in a control program. For example, `ANW1(HallwayB)[lb,ub]` requires the ANW1 to be in HallwayB for a minimum of `lb` time units and a maximum of `ub` time units. We refer to these commands as location assertions. Location assertions can specify waypoints for the AV to follow. For example, the sequence on Lines 16-20 requires ANW1 to be in HallwayB for an instant of time then execute a no-op (no operation), and, finally, visit LaboratoryOne for an instant of time. When location assertions are combined with a parallel combinator, as seen on Lines 15-22, the mission designer can explicitly express a set of waypoints for an AV to navigate to, and the activities that the AV must execute while navigating to the waypoints. The expression on Lines 15-22, for example, requires ANW1 to fly from HallwayB to LaboratoryOne, while concurrently executing the `Take-Pictures` activity.

A location constraint is entailed when the path-planner in the unified activity and path planning system finds a collision free path from the AV's current location, to the goal location specified in the control program. This process is explained in Chapter 6. Note that location constraints are similar to achievement constraints in [14], except that an achievement constraint specifies the intended internal state of an embedded system, rather than the location of an AV.

3.1.4 Activity Costs

Estimated activity costs, such as power or fuel, are determined by the mission designer. An activity and its cost are specified together in an RMPL control program. The activity costs are used in the optimal pre-planning process to guide the search towards the optimal strategy.

On the whole, RMPL primitives can be combined recursively in order to describe complex behaviors for autonomous vehicles. Costs provide the information needed to perform deterministic, decision theoretic execution. Location constraints specify where activities are to be performed, while abstracting away the specification of how to get

there. To execute ACLC RMPL programs, the pre-planner performs optimal unified activity and path planning.

3.2 Environment Model

The environment model contains the information about the world where the autonomous vehicle will be navigating. It includes terrain information, a physical description of the AV(s), and the domain of the locations where an AV can visit. Descriptions of the terrain and the AV(s) are used to create the state space used by the path planner.

3.2.1 Environment Model Specifications

The environment model is defined by a tuple $\langle M, R, T \rangle$ where:

- M is a pair $\langle$*World Dimensions*, *Obstacles*$\rangle$ encoding a map of the world where the robots will be navigating. The map is defined by its world dimensions, denoted $dim(M)$ and is comprised of length, width, height, and starting height, which specify the 2-D plane or 3-D space in which the robots will travel. The map also includes the obstacles or occluded regions $O = \{O_1, O_2, \ldots, O_n\}$ that the robots must avoid. The obstacles are described as polygons.

- R is a the physical description of the robot in the mission. This includes the dimensions of the robot and its minimum and maximum translational and rotational velocities. Also, the initial position (state) of each robot is given. For the purpose of this research, we encode R as a tuple $\langle$*radius, initial-state,* [*min-v, max-v*], [$min\theta$, $max\theta$]$\rangle$. R can be extended to incorporate additional information about the robot's dynamics for more complex problems.

- T is a symbol table, mapping locations in the domain of locations to their actual coordinates or regions in M.

Enter-Building Environment Model

In the Enter-Building world, the AV explores a 2-D plane with dimensions 80' × 100', at a starting height of 20' (Figure 12). This is encoded as $dim(M) = (80, 100, 0, 20')$. The

obstacles in the world are encoded as rectangles, which are specified by their minimum *x* and *y* coordinates, length, and width, (*minx*, *miny*, *l*, *w*). There is a total of 12 obstacles in the Enter-Building world. The encoding for LaboratoryOne, which contains five obstacles, is O_{lab1} = {O_5, O_6, …,O_9}, where, for example, O_5=⟨0', 55', 50', 1'⟩ states the obstacle 5 starts at coordinates (0, 55) with a length of 50 feet and a width of one foot. Thus, *M* = ⟨⟨80, 100, 0, 20'⟩, {O_1 ,O_2, …,O_{12}}⟩.

The Enter-Building example contains only one robot, ANW1. The dimensions of ANW1 are defined by a sphere with radius *r*, which encompasses the small helicopter. In this example, the radius is one foot. For simplicity, we represent the AV as a moving sphere with a heading. Its start state is hovering at a constant height $\langle x, y, \Theta, \dot{x}, \dot{y} \rangle$ = ⟨75, 80, 135°, 0, 0⟩. ANW1 is described by the following tuple, R = ⟨1', ⟨75, 80, 135°, 0, 0⟩, [0, 2 ft/sec], [0°, 360°]⟩. [31] gives a precise state formulation, including dynamics for a small, agile autonomous, helicopter.

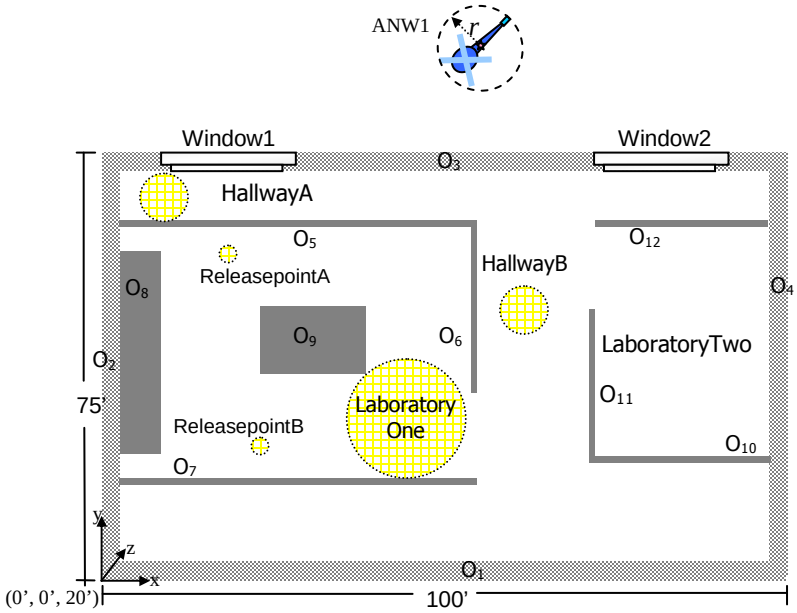


Figure 14: Regions specified by the locations in the symbol table *T*.

Finally, the domain of locations where ANW1 can visit is specified as the set of regions {HallwayA, HallwayB, ReleasepointA, ReleasepointB, LaboratoryOne} and are depicted with circles in Figure 14. The symbol table *T*, for the Enter-Building example,

maps each location to a coordinate, or a coordinate-radius pair that defines a region, and is given in Figure 15.

Name	Coordinates (x, y)
HallwayA	(3, 60) radius 5’
HallwayB	(53,40) radius 5’
ReleasepointA	(20, 25)
ReleasepointB	(20, 50)
LaboratoryOne	(15, 20) radius 15’

Figure 15: Symbol table for the Enter-Building control program.

3.3 Pre-planning and Execution System Overview

The control program and environment model, together, are used as input to the unified activity and path planning system, described in this thesis (Figure 16). The system is composed of an optimal pre-planner and a roadmap-based path-planner. The control program is mapped to a graph data structure, called a Temporal Plan Network (TPN), described in Chapter 4. The optimal pre-planner uses informed search techniques in order to search the TPN for safe threads of execution. The environment model is used with the roadmap-based path planner, which attempts to find a collision-free path to achieve the location constraints.

The output of the optimal pre-planner is the best strategy (plan) that does not violate any of its temporal or location constraints. The plan is then input to a plan runner. The plan runner is an executive that exploits the temporal flexibility in the TPN plan [25]. The plan runner consists of an offline pre-processing stage, which compiles the temporal constraints into a for that can be dispatched quickly, followed by an online scheduling phase that dynamically builds the schedule for the plan while activities are being executed [25].

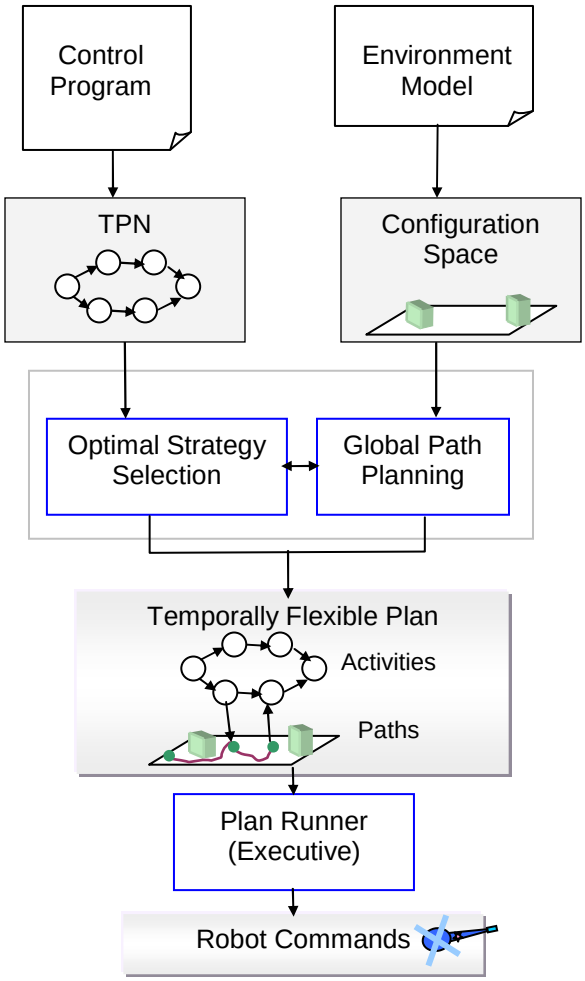


Figure 16: Overview of the unified activity and path planning system.

Chapter 4

Temporal Plan Networks

A model-based control program describes a mission strategy with high-level goals that autonomous vehicles must achieve during their mission. For many missions, movement between locations is an essential component of the planning process. In addition, minimizing the combined cost of these activities is key. To support these missions, we developed the activity cost and location constraint (ACLC) subset of RMPL in Chapter 3. To enable fast mission pre-planning, strategies written in a RMPL control program are compactly encoded in a graph-based data structure called a Temporal Plan Network (TPN) [39][40]. Planning is then performed by applying efficient graph algorithms to TPNs.

This chapter introduces the TPN model and presents an extension to the model that includes activity costs and location constraints. A mapping from the ACLC primitives to a TPN model is also discussed. We conclude this chapter with a description of a TPN generated plan.

4.1 Overview of Temporal Plan Networks

A temporal plan network is a compact graphical encoding of a RMPL control program. TPNs represent activities, activity costs, location and temporal constraints from the corresponding RMPL control program. The construction of a temporal plan network is based on the RMPL combinators used to express mission strategies. The RMPL combinators define the composition of sub-networks within a TPN, and the activities and constraints define the arcs and vertices of the network. Figure 17 depicts the TPN representation of the Enter-Building control program shown in Figure 11.

As seen in Figure 17, a temporal plan network is composed of vertices and arcs, each of which refers to a specific element in the corresponding control program. Vertices denote time points that correspond to specific events, such as the start or end of an

activity. The arcs in a TPN represent activities, activity costs, location constraints, and temporal bounds on activities. For example, the arc D→F states that the activity `Stereo-Vision()`has a cost of 40 units, must take a minimum of 10 time units and must not exceed 20 time units. In addition, the location constraint `loc(HallwayB)`, which is attached to arc D→F, requires ANW1 to be in HallwayB throughout the duration of the activity. Vertices in a TPN represent temporal events. The start and end of an activity are each signified by a vertex. For example, event D marks the start of the `Stereo-Vision()`activity and event F marks the end of that activity. The arcs correspond to the temporal distance between events [39][40]. An arc with temporal bound `[0,0]` indicates that the event at the head of the arc is executed immediately after the event at the tail of the arc. An arc with a `[0,+INF]` time bound indicates that the event at the tail of the arc may start at any time after the event at the tail of the arc. In addition, the direction of a TPN arc defines the order in which events on a thread are executed. In a TPN, a path from the start event to the end event of the high-level activity denotes a thread of execution. For example, any path from S to E in the Enter-Building TPN corresponds to a thread of execution.

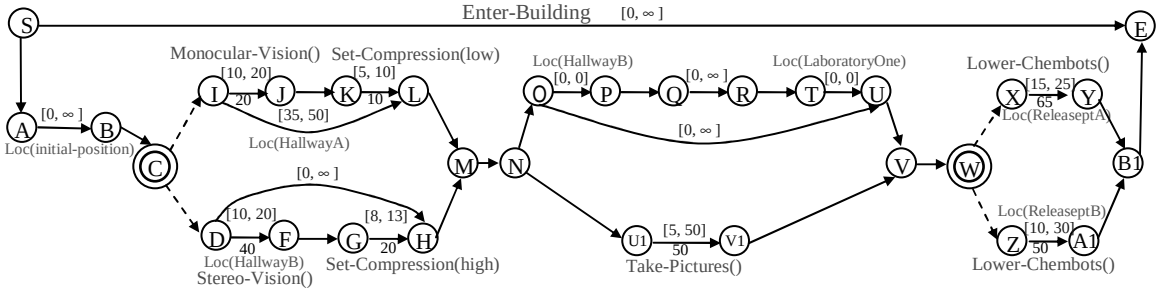


Figure 17: Example Temporal Plan Network for the Enter Building control program. In the figure, the name of the autonomous vehicle ANW1 involved in the mission is omitted and the activity names are abbreviated for clarity. In Figure 17, arcs without an explicitly labeled temporal constraint are assumed to have [0, 0] bounds and arcs without explicit location constraints are assumed to not be constrained by location.

TPNs encode choice between possible threads of execution using a special type of event, called a decision point. These are depicted in Figure 17 with double circles; these are the events C and W. The event C is a decision point that marks the start of a decision sub-network that ends at event M. Associated with each decision point is a *decision sub-*

network. The sub-network begins at the decision point and ends at an event where the threads of the choices re-converge. For example, for the decision sub-network starting at C and ending at M, the two possible threads of execution are C→I→J→K→L→M and C→D→F→G→H→M. Only one thread from each decision sub-network in a TPN is included in a plan.

TPNs encode concurrent threads of execution with sub-networks composed of the parallel threads. For example, the event N in Figure 17 denotes the start of two concurrent threads of execution. The end of the parallel sub-network occurs at the event where the two threads converge. For example, the event V (Figure 15) denotes the end of the parallel threads that start at N.

Location constraints in a TPN are specified using the "Loc" function and are parameterized with the name of the region where the autonomous vehicle must reside. Recall that the symbol table *T* in the environment model translates a region of the input map. Location constraints are resolved by the unified activity and path planning system, which operates on both the environment model and the control program, in order to satisfy the constraint. This process is described in further detail in Chapter 6.

4.2 RMPL to TPN Mapping

In this section we provide the mapping from the ACLC subset of RMPL to TPN sub-networks. This mapping shows how any control program with the ACLC subset can be encoded as a temporal plan network. The optimal pre-planner then searches for safe threads of execution in a control program by searching its corresponding temporal plan network representation.

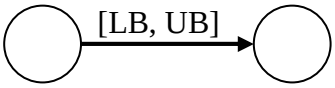
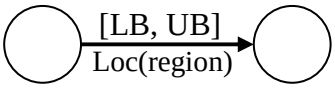
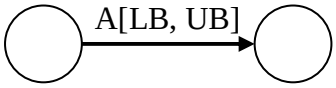
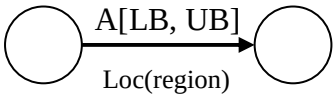
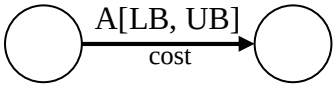
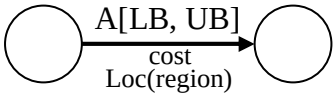
`Interval:` [LB, UB]	
`Interval + Location:` (region)[LB, UB]	
`Interval + Activity:` A[LB, UB]	
`Interval + Activity + Location:` A(region)[LB, UB]	
`Interval + Activity + Cost:` A(cost)[LB, UB]	
`Interval + Activity + Cost + Location:` A(cost, region)[LB, UB]	

Figure 18: Mapping from ACLC primitives to TPN sub-networks.

Recall that the RMPL notion for cost, location, and temporal bounds is given in the form (cost, location) [lower time bound, upper time bound]. Both Figure 18 and Figure 19 depict the information stored in a TPN sub-network. In general, events mark the start and end of an activity or the start and end of a TPN sub-network created by an RMPL combinator.

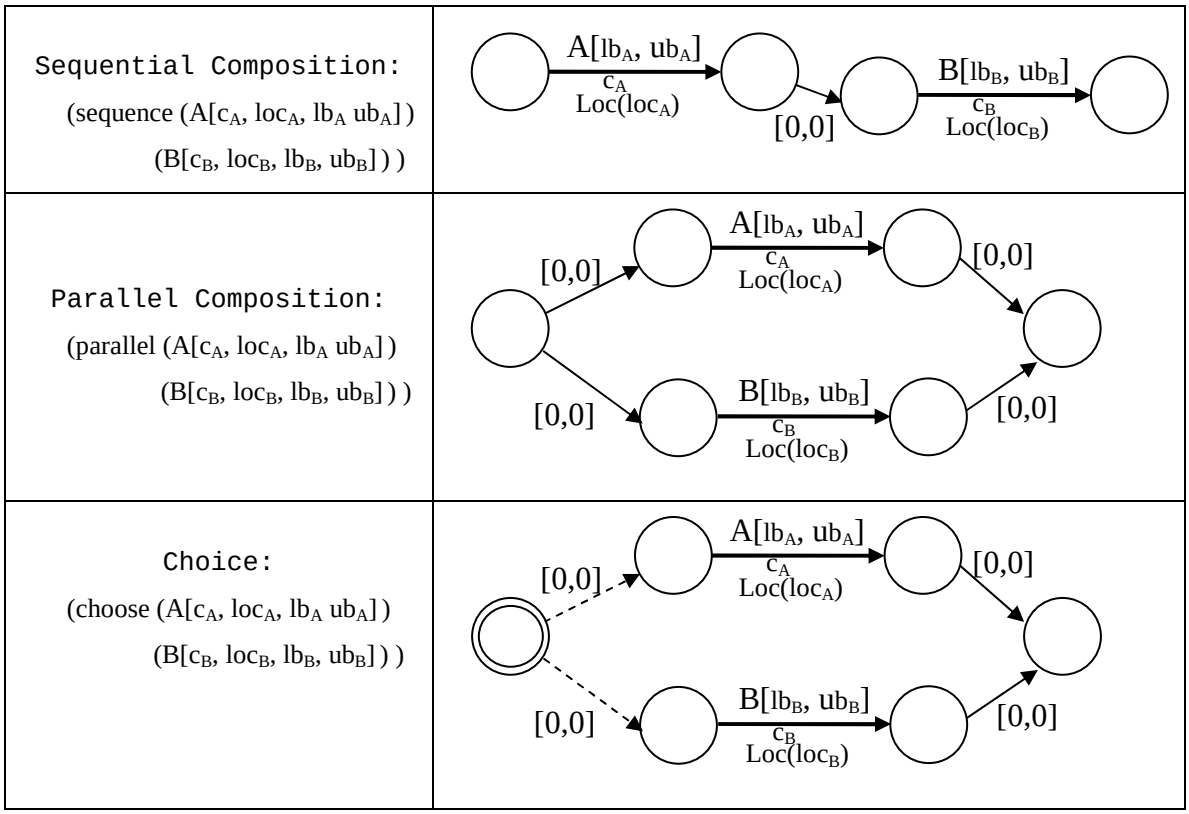


Figure 19: Mapping from ACLC combinators to TPN sub-networks.

Given the mapping of RMPL primitives and combinators to TPN sub-networks, any control program can be modeled as a temporal plan network. As a result, efficient network algorithms can be applied to TPNs in order to find the best executable strategy in a control program. In the remainder of this chapter, we define feasible and optimal plans, denoting feasible and optimal executions of an ACLC RMPL program, respectively.

4.3 TPN Plan Formulation

Recall that RMPL control programs specify possible strategies that will accomplish a mission. A TPN represents these possible strategies as disjunctive plans, that is, plans containing choices. Each possible plan in a TPN is distinguished by the set of chosen decisions in each decision sub-network. For example, the network of Figure 17 has two decision points, each with two potential choices, and hence represents four possible plans. The first possible plan contains the decisions C→I and W→X, the second contains

the decisions C→I and W→Z, the third contains C→D and W→X, and the fourth possible plan contains the decisions C→D and W→Z.

More formally, a plan is a complete, consistent execution of an RMPL control program. A TPN plan is represented as concurrent threads originating at the start of the TPN, S, and finishing at the end of the TPN, E. Each thread represents a string of activities, temporal constraints, and location constraints. A plan is complete if three properties hold. First, the TPN plan must be composed of a set of threads that originate at the start S and end at E. Second, if the plan contains a decision-point, denoting a decision sub-network in the corresponding TPN, then only one thread from the start to the end of that decision network is selected. Third, all threads extending from all non-decision points in the plan must be selected. A plan is consistent if it is both temporally and location consistent. To be location consistent, all location constraints in the plan must be satisfied. That is, there exists a collision-free path to each region specified in each of the location constraints. We refer to a plan that is complete and consistent as a *feasible* plan. A feasible plan for the Enter-Building program is shown in bold in Figure 20 below. In the example, the plan begins at S and ends at E. The out-arcs of each non-decision event in the plan are selected and only one thread from each decision sub-network is included in the plan. In addition, the temporal constraints of the plan highlighted in Figure 20 is temporally consistent, as defined in the next section, and there exists a collision free path for all locations. Therefore the plan is feasible. Finally, the plan in Figure 20 is optimal, that is, it is comprised of the least-cost complete and consistent set of threads.

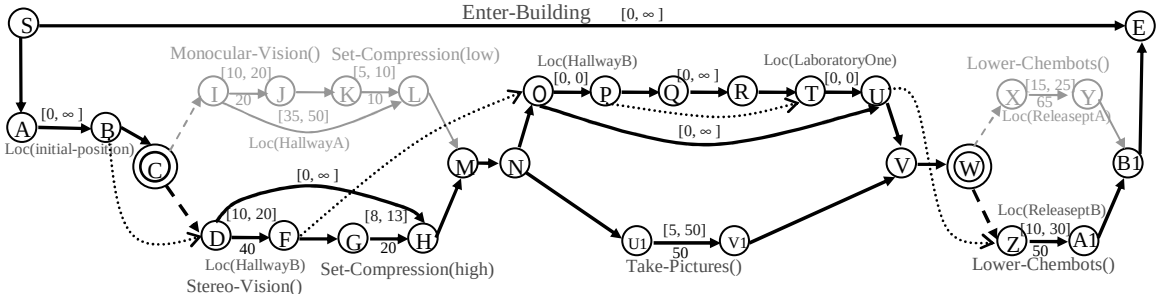


Figure 20: Example of an optimal, complete and consistent plan for the Enter-Building activity. The dotted arrows from location to location enforce the order in which each location is visited by ANW.

A solution to a temporal plan network is a feasible plan. The optimal pre-planner applies a best-first search strategy to the TPN, while resolving temporal and location constraints in order to find the optimal feasible plan,

4.4 Temporal Consistency in TPNs

To declare a plan consistent and feasible, it must be temporally consistent. We conclude this chapter by defining temporal consistency, and by reviewing algorithms for checking temporal consistency. The temporal constraints of a plan, generated by a TPN, form a Simple Temporal Network (STN). Thus, the techniques that verify the temporal consistency of an STN can be applied to TPN plans [39]. An STN is similar to a TPN in that it contains a network of arcs between events. It differs in that each arc of an STN specifies only a time bound (no cost or activity) and decision points are disallowed. One way to test for temporal consistency of an STN is by converting it into an equivalent representation, called a *distance graph*. If the distance graph of an STN contains no negative weight cycles, then the corresponding STN, and thus the TPN plan, is temporally consistent [9]. An STN is converted to a distance graph by maintaining its events and by creating two arcs between connected pairs of events *x* and *y*. Given an STN arc $x \rightarrow y$, labeled [lb, ub] the distance graph arc $x \rightarrow y$ with label ub specifies the upper bound time of the STN arc. The distance graph arc $y \rightarrow x$ with label –lb specifies the lower bound time from the STN arc $x \rightarrow y$. In general a distance arc $x \rightarrow y$ with label *l* specifies the constraint $y - x \leq l$. For the lower bound arc, its corresponding constraint $x - y \leq -lb$ is equivalent to $y - x \geq lb$ Figure 21 illustrates an STN and its corresponding distance graph.

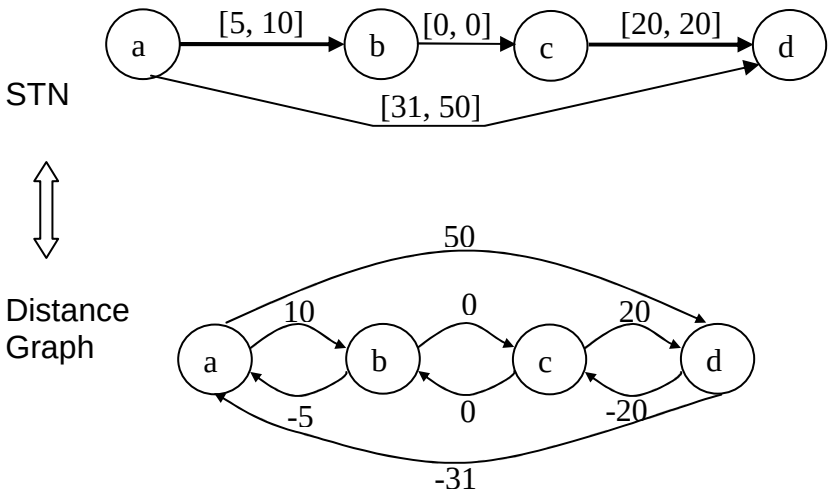


Figure 21: Example of STN and its corresponding distance graph.

In the above example, the sequence of events a, b, c, and d are constrained by the temporal bound `[31,50]` on STN arc a→d this specifies that the entire sequence of events must be completed at a minimum of 31 times units and a maximum of 50 time units. If this constraint is ignored, then the sequence of events are temporally consistent. However, with the a→d constraint included, the STN it is not temporally consistent. This inconsistency is identified by the negative weight cycle a→b→c→d→a in the distance graph. Intuitively, events a, b, c, and d can be executed within a minimum time of 5+0+20 = 25 time units, which is the sum of the lower bounds on each event. The events must be executed within 10 + 0 + 20 = 30 time units (the sum of the upper bounds). However, the additional constraint, a→d, over the entire sequence requires the events to take at least 31 time units to execute. This contradicts the maximum amount of time to execute the sequence (30 time units); therefore, the above example illustrates a temporal inconsistency in the STN and any plan that includes that STN as a sub-network is invalid.

As stated above, to detect temporal inconsistencies in a plan or partial plan, an algorithm that detects negative weight cycles in a distance graph is used. Some of the more common algorithms for testing negative weight cycles in a graph are the Floyd-Warshall all-pairs shortest path, the Bellman-Ford single-source shortest path, and the FIFO-Label-Correcting (where FIFO stands for first-in-first-out) algorithms [8].

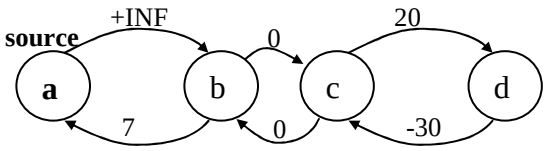


Figure 22: Distance graph from a sequence of STN constraints.

The Kirk temporal planner framed the problem of finding a negative weight cycle in a distance graph as a single-source shortest path (SSSP) problem. To detect a negative weight cycle in a distance graph, the FIFO-Label-Correcting algorithm was applied [1][18]. However, framing the problem of finding a negative weight cycle in a distance graph as a SSSP problem causes the FIFO-Label-Correcting algorithm to fail on particular distance graphs. The distance graph in which the algorithm will not detect a negative weight cycle is a distance graph with a positive infinity on an arc in a sequence. In this case, the positive infinity value would propagate to an inconsistent sub-network and cause that inconsistent sub-network to be declared temporally consistent. For example, the STN in Figure 22 has a negative cycle between c and d. By computing the shortest path from the source a to nodes b, c, and d, the `+INF` shortest path from arc a→b is included in the shortest path cost of a→c and a→d. Then, when attempting to reduce the shortest path from a to c, through the path a→b→c→d→c, the −30 distance on arc d→c is substracted from the current estimate of the shortest path from a to c. Since the current estimate of the shortest path from a to c is `+INF`, then −30 is subtracted from `+INF`, which results in `+INF`. By propagating the +INF cost forward, the negative cycle from c→d→c is not detected. This issue is also illustrated in Figure 23.

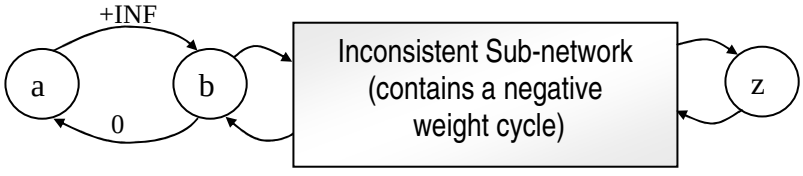


Figure 23: A sequence that starts with a +INF arc and contains a sub-network that is temporally inconsistent. The +INF arc gets propagated forward when applying the single-source shortest path algorithm and the shortest path between nodes in the graph all become +INF, and thus, a temporally inconsistent plan is mistakenly declared temporally consistent.

We make clear the distinction between a single-source shortest path problem and the algorithms used to solve it, as mentioned above. To address the issue of propagating `+INF` cost, we reformulate the problem of finding a negative weight cycle in a distance graph, as a single-destination shortest path (SDSP) problem [8]. The SDSP problem is to find the shortest path to a given destination node *s* in a graph from all other nodes in that graph.

The key to solving an SDSP problem is to reverse all the arcs in the input graph. Given this formulation a SSSP algorithm that detects negative weight cycles in a graph can be applied. Thus, we still use the FIFO-Label-Correcting algorithm, but on the SDSP formulation, where the source node, for example a, is specified as the destination node.

Intuitively, by reversing the arcs in the distance graph, the arcs with `+INF` are essentially ignored. These arcs are headed in the opposite direction of the destination node, and have a positive weight. Thus, they cannot be a part of the shortest path from a node to the destination. For example, in Figure 24 we want to detect the negative weight cycle b→c→b in the original distance graph. As stated, the shortest path from the source node a to nodes b and c is +INF, even though there exists a negative weight cycle. At the bottom of the figure, the arcs in the original distance graph have been reversed. If a single-source shortest path algorithm, like the FIFO-Label-Correcting algorithm, were applied to the graph at the bottom of Figure 24, then the arc with the +INF does not affect the shortest path from a to b, which is -5. Thus, the negative cycle b→c→b is detected, because the algorithm will repeatedly reduce the cost of node b, until it is examined more times than the number of nodes in the input graph (Line 11-12, Figure 25). At this point, since the cost of b is negative, it can be shown that a cycle must exist.

The FIFO-Label-Correcting algorithm is shown in Figure 25. The Push function adds a node to the end of the queue and the Pop function returns the node at the top of the queue. Note that Line 13 examines the incoming arcs (*j*, *i*), which is equivalent to examining a distance graph with its arcs reversed. In the original FIFO algorithm all the outgoing arcs (*i*, *j*) are examined [39][40]. The algorithm has an O(*nm*) run time, where *n* is the number of nodes in a graph and *m* is the number of edges in the graph. For a more careful discussion of the FIFO-Label-Correcting algorithm see [1].

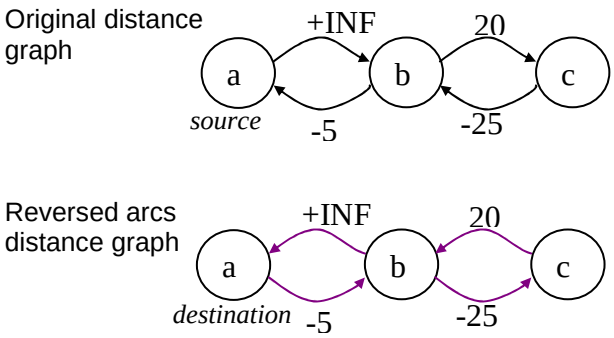


Figure 24: By framing the problem as a single-destination shortest path problem all negative weight cycles in a distance graph can be found.

---

function SDSP-FIFO (Graph *G*, Node *destination*) **returns** *true if negative weight cycle or false if no negative weight cycle.*

```
1.  initialize FIFO-queue Q ← ∅
2.  for each n_i ∈ Nodes( G ) do
3.     d( n_i ) ← +INF
4.     examined_count( n_i ) ← 0
5.  end
6.  d( destination ) ← 0
7.  Push( Q, destination )
8.  while Q ≠ ∅ do
9.     i ← Pop( Q ) //Removes the node at the top (or head) of the Q
10.    examined_count( i ) ← examined_count( i ) + 1
11.    if examined_count( i ) > |Nodes( G )| then
12.       return true //negative cycle detected, repeatedly revisiting a node
13.    for each arc ( j, i ) ∈ G do //Arcs are reversed by examining the incoming arcs j to i
14.       if d( j ) > d( i ) + weight( j, i ) then
15.          d( j ) ← d( i ) + weight( j, i )
16.       if j ∉ Q then
17.          Push( Q, j ) //Adds j to the end (or tail) of the Q
18.    end
19. end
20. return false
```

Figure 25: FIFO-Label-Correcting pseudo code used to detect a negative weight cycle in a TPN [1].

In summary, this Chapter presented the mapping from the ACLC derivative of RMPL to TPN sub-networks. We defined a complete and consistent execution of a TPN and presented the algorithm to test for temporal consistency of a TPN plan.

Chapter 5

Optimal Pre-planning of Activities

To find the best strategy for accomplishing mission objectives, we use the notion of optimal pre-planning. Optimal pre-planning is the process of taking a set of possible strategies, encoded in a TPN model of a control program, and searching for the optimal feasible plan. As a part of the optimal pre-planning process, we adopt A* search to find the best strategy. The optimal pre-planner is a forward heuristic search planner that uses a *TPN heuristic* to efficiently guide the search towards the best solution. A challenge in heuristic search planners is that the admissible heuristics, such as the max heuristic of HSP, tend to be extremely weak, and uninformative. The TPN heuristic is novel in its use of dynamic programming to provide an estimate that is more accurate and informative than max. The algorithm that guides the search is called TPNA*. TPNA* is also novel for its compact encoding of plan space. This chapter presents the terminology and develops the procedures that define the optimal pre-planning process.

To illustrate the optimal pre-planning process, we first present an example problem, and then apply the TPNA* search to that problem. The example is called AtHome. The RMPL control program for the AtHome example is given in Figure 26. Figure 27 illustrates the equivalent TPN representation. For the purpose of focusing on optimal pre-planning, the AtHome example does not contain location constraints. We address the combined problem of optimal pre-planning with location constraints in Chapter 6.

The objective of the AtHome program is twofold. The first objective is to refuel the autonomous vehicle, ANW1. This objective is achieved by Lines 3-11 of the control program, depicted in Figure 26. The second objective is to send sensor data to the control center. This can be achieved through two methods: either by uploading raw sensor data, or by fusing sensor data onboard and then uploading the fused data. The first method is

specified by the control program in Lines 14-20, and the second method is specified in Lines 21-24.

AtHome RMPL Control Program

```
1.  (AtHome [15, 100]
2.      (parallel
3.          ;;Refuel autonomous vehicle ANW1
4.          (sequence
5.              ( ANW1.Connect-To-Charger(80) [5, 20] )
6.              (choose
7.                  ( ANW1.Refuel-CellA(20) [0, 15] )
8.                  ( ANW1.Refuel-CellB(70) [0, 15] )
9.                  ( ANW1.Refuel-CellC(30) [0, 15] )
10.             )
11.         ) ;;end sequence
12.         ;;Send sensor data to control center by one of two ways
13.         (choose
14.             ( (sequence
15.                  ( ANW1.Upload-Raw-Data(25) [10, 30] )
16.                  (choose
17.                      ( ANW1.Purge-DataSet1(10) [10, 15] )
18.                      ( ANW1.Purge-DataSet2(20) [25, 35] )
19.                  )
20.              ) [0, 20]) ;;end sequence
21.             (sequence
22.                 ( ANW1.Sensor-Fusion(20) [1, 5] )
23.                 ( ANW1.Upload-Fused-Data(10) [1, 5] )
24.             )
25.         ) ;;end choose
26.     ) ;;end parallel
27. ) ;;end AtHome activity
```

Figure 26: Control program for the AtHome strategy. In this example there are no location constraints, thus the default value for all activities is ANYWHERE, as explained in Chapter 3.

The control program for the AtHome program maps to an equivalent TPN, shown in Figure 27. The arcs of the TPN are labeled with activity names, costs, and temporal constraints, specified by the AtHome control program. For example, the arc between nodes J and K represents the `Connect-To-Charger` activity. Stored on this arc are the estimated cost of 80 units and time bounds of `[5,20]`, which is specified in the control program (Line 5). In this example, arcs without an explicitly labeled cost are assumed to have a default cost of 5 units. The AtHome TPN contains no location constraints, thus, they are not included on the TPN arcs. The AtHome TPN is referenced throughout this chapter.

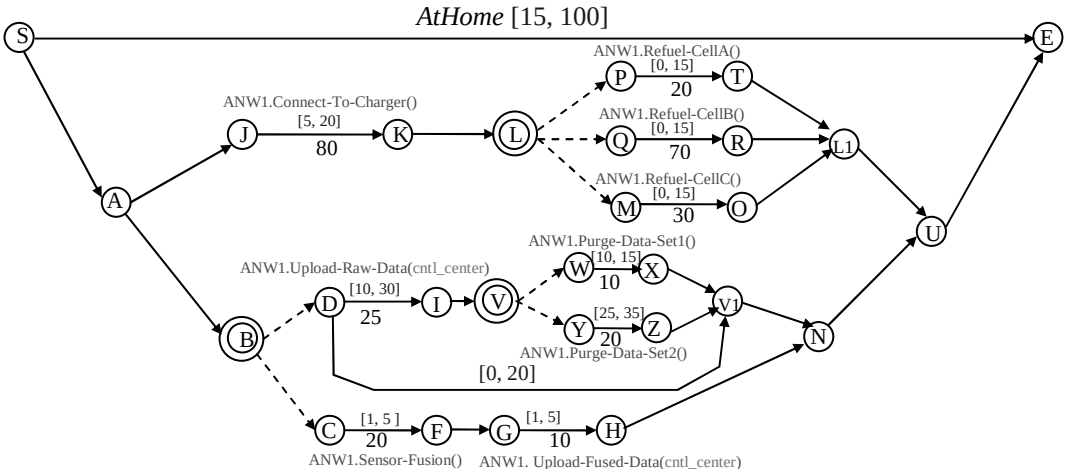


Figure 27: Corresponding TPN representation of the AtHome program in Figure 26. Decision points are marked with double circles. Arcs without an explicit time bound have a `[0,0]` time bound. Arcs without an explicit cost are assumed to have a default cost of 5 units. There are no location constraints in this example, thus, they are not included on the arcs.

An RMPL control program, like AtHome, is composed of one or more possible strategies for a robot to execute during a mission. In order to find a feasible plan, the original temporal planner within Kirk uses a modified network search that tests for temporal consistency, selects activities, and defines causal links between events [40][39]. Recall that a feasible plan is both complete and consistent. A plan is complete if it refers to a selected sub-network of the TPN such that 1) the sub-network originates at S and ends at E, 2) the sub-network contains only one thread extending from each decision point in the sub-network, and 3) the sub-network includes all threads extending from each non-decision point. For a complete plan to be consistent, and thus feasible, it must also adhere to its temporal constraints. That is, the simple temporal constraints of the sub-network must be satisfiable. While the original Kirk temporal planner searches a TPN for a feasible plan, it does not address pre-planning problems for which the cost of executing activities is critical to the success of the mission.

The optimal pre-planning process presented here extends Kirk by adopting a best-first search strategy, in particular A*, in order to find the optimal feasible plan. That is, from the set of all complete and consistent plans in a TPN, TPNA* search returns the feasible plan with the minimum cost.

The primary contribution of this chapter is threefold. The first is the formulation of an optimal pre-planning problem as a state space search problem. The second is a

compact encoding of a TPN state and a mapping from a search state to its corresponding TPN sub-networks. The third contribution is a procedure that extracts a heuristic from a TPN *a priori*, which is used to efficiently guide the search. We conclude this chapter with a discussion of the TPNA* search and its properties.

5.1 Review of A* Search

An optimal pre-planning problem can be framed as an optimization problem for which an informed search technique can be applied. We adopt the widely used A* search algorithm [26]. A* is a deterministic search that combines greedy-search with uniform-cost search, in order to find the optimal path from an initial state to a goal state of a search problem. The minimum-cost path is found by selecting search nodes according to an evaluation function. The evaluation function f estimates the cost of a search node n, by summing two values, g(n) and h(n). The first value, g(n), is the actual path cost from the start node, which represents the initial state, to node n. The second value, h(n), is a heuristic value that serves as an under-estimate of the cost from node n to the goal. The best solution is found by repeatedly selecting the path with the best f(n). The general A* search procedure is shown in [28].

One important element of A* is the dynamic programming principle. When A* finds a better path to an intermediate node n, it prunes the expansion of all other sub-optimal paths that reach n, thereby storing only the best-cost path to node n [26]. However, in the case of TPNA* search, the algorithm is systematic, that is, it visits search states at most once. Thus, in TPNA*, there is only one path to a node n, and the dynamic programming principle is not necessary.

```
procedure A*-Search ( Problem p )  returns minimum cost solution if, and only if, one exists.
1.  initial state s ← Initial-State( p )
2.  node n ← Make-Root-Node( s )
3.  f( n ) ← g( n ) + h( n )
4.  Insert( n, Open )
5.  while Open ≠ ∅ do
6.      n_best ← Remove-Best( Open )
7.      if Goal-Test( n_best, p ) then
8.          return solution n_best
9.      else if n_best ∉ Closed then
10.         new-child-nodes ← Expand( n_best )
11.         for each n_i ∈ new-child-nodes do
12.             if state( n_i ) ∉ Closed then
13.                 Insert( n_i, Open )
14.         Insert( n_best, Closed )
15. end
16. return no solution
```

Figure 28: The general A* search algorithm. The input is a problem that represents all possible states. A* search finds the shortest path from the initial state to the problem's goal state (checked by the Goal-Test function). Two sets, Open and Closed, are maintained. The Open set is a priority queue, which stores all nodes according to their evaluation function, $f(n)$. Nodes in Open are available for expansion. The Closed set contains all nodes that have been expanded.

A* search is both optimal and complete as long as $h(n)$ is an admissible heuristic, that is, $h(n)$ never overestimates the cost from a node n to the goal. The algorithm is complete for all problems in which each node has a finite branching factor. Proofs of optimality and completeness can be found in [26]. For a more careful introduction to A* search, see [26].

A Search Example*

To develop the TPNA* search algorithm, we first consider how searching for the optimal plan relates to searching for the shortest path in a graph. In a shortest path problem, every vertex in a graph is analogous to a decision point in a TPN. An example of a weighted graph is shown in Figure 29. Each vertex is a decision point and each arc is assigned a cost (also called a weight) if the start vertex is A and the goal vertex is G, then there are four possible paths from A to G: A→B→D→G, A→C→D→G, A→C→F→G, and A→C→E→G. The shortest path is the one with the least cost. In this case, the shortest

path is A→C→D→G which has a cost of 8 units. We demonstrate how A* search explores this graph to find the shortest path.

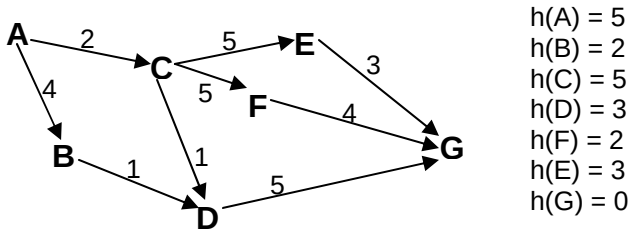


Figure 29: Example of a weighted graph (on the left) and the corresponding heuristic cost for each vertex to the goal G (on the right). The graph can be considered a state space search problem, where each vertex represents a state and each arc represents a specific action that takes the problem from one state to the next. The initial state is A and the goal state is G. Each vertex in the graph is analogous to a decision point in a TPN.

A* search can be applied to find the shortest path, the path with the least weight, from A to G in the graph in Figure 29. The heuristic cost, $h(v)$, for each vertex v is shown to the right of the graph in Figure 29. To expand vertices in best-first order, A* maintains a priority queue, called *Open*. Open is initialized with the start vertex A. During each iteration, the vertex with the least estimated cost, $f(v)$, is removed from Open and expanded, adding each of its target vertices to Open. Once a vertex is expanded, it is then inserted in a set containing all expanded nodes, called *Closed*. For example, when vertex A is expanded, targets B and C, with estimated costs $f(\text{B}) = 4 + 2 = 6$ and $f(\text{C}) = 2 + 5 = 7$, are inserted into Open, and A is inserted into Closed. Then B is expanded, adding its target D, $f(\text{D}) = 8$, to Open. This process continues until the goal G is reached. Figure 30 illustrates the priority queue for each iteration of the search.

Notice, that in this problem the dynamic programming principle applies. There are vertices in the graph, namely D and G, which can be reached via two or more paths. For example, there are two paths to D, A→B→D and A→C→D, where A→C→D is the least-cost path. First D is reached via the suboptimal path through B, giving D an estimated cost of 8 units (iteration 3). Then D is reached via its optimal path, A→C→D, and D is inserted again, but with a cost of 6 units (iteration 4). Then D with a cost of 6 is expanded, and G is inserted into Open. Next, D with cost of 8 units is removed from Open. Since vertex D already exists in the set Closed, this D with a cost of 8 units it is not expanded. [28] details the steps required to solve search problems with multiple paths to the same vertex.

Iteration	Open: (vertex, *f*(vertex))	Closed
1.	(A, 5)	
2.	(A B, 6) (A C, 7)	(A, 5)
3.	(A C, 7) (A B D, 8)	(A, 5) (B, 6)
4.	(A C D, 6) (A B D, 8) (A C F, 9) (A C E, 10)	(A, 5) (B, 6) (C, 7)
5.	(A B D, 8) (A C D G, 8) (A C F, 9) (A C E, 10) *D∈Closed, go to next iteration*	(A, 5) (B, 6) (C, 7) (D, 6)
6.	**(A C D G, 8)** (A C F, 9) (A C E, 10) *G is removed from Open, Goal-Test satisfied, shortest path found*	(A, 5) (B, 6) (C, 7) (D, 6)

Figure 30: Example of applying A* search to find the shortest path from vertex A to G in the graph in Figure 29. The priority queue Open contains partial paths with their estimated cost. The vertex expanded during each iteration is underlined. The goal is shown in bold (iteration 6).

5.2 Optimal Pre-Planning Overview

The input to the optimal pre-planning system is a temporal plan network. Recall that a TPN, by definition, represents the space of all complete executions of an RMPL program. There may be any number of complete executions represented by a TPN, depending on the number of choices at each decision point in the network. In the AtHome TPN, for example, the decision points B, L, and V create two parallel sets of three threads each, resulting in nine possible executions of the AtHome program.

The optimal pre-planner outputs the optimal complete and consistent execution with the least-cost, if and only if one exists. The optimal solution to the AtHome program is highlighted in bold in Figure 31. Although the plan with choices B→C and L→P has the least-cost, the activities within that plan create a temporal inconsistency with the overall AtHome program's `[15,100]` time bound. Consequently, the optimal feasible solution is the next best plan, which contains choices B→D, V→W, and L→P. This plan is both complete and consistent.

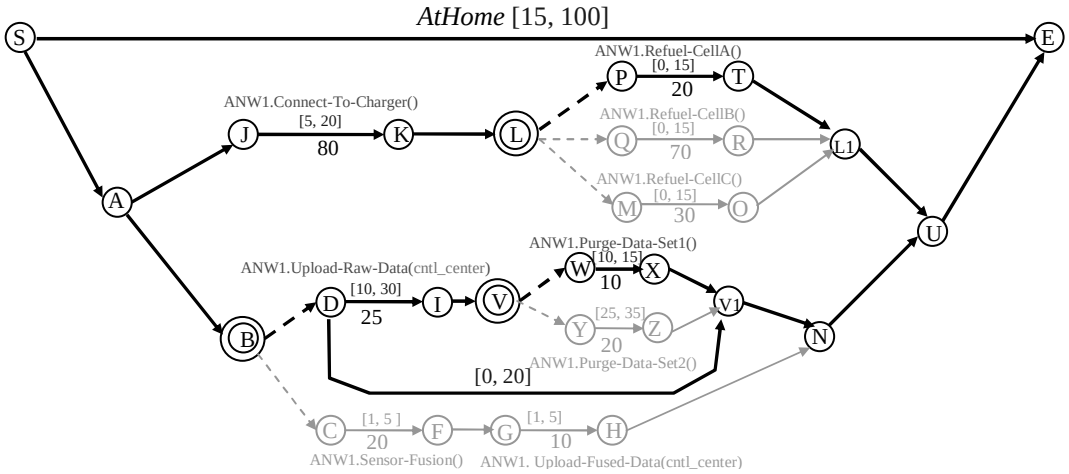


Figure 31: The bold portion of the TPN denotes the optimal plan for the AtHome program. We use a default arc cost of 5 units (not including arc S→E).

Our approach to solving an optimal pre-planning problem is an instance of A* search. In this section we define the search space in terms of TPNs, where each search state denotes a partial execution of the TPN. Then we formulate an optimal pre-planning problem as a state space search problem, and describe the search tree used to represent the search space.

5.2.1 TPNA* Search Space

The search space of an optimal pre-planning problem consists of the prefixes of all complete executions of a TPN. We refer to these prefixes as partial executions. More specifically, a partial execution is a set of contiguous concurrent threads that originate at the TPN start S and have not been fully expanded to the end event E. Each thread of a partial execution is comprised of activities, and temporal constraints. TPN events are implied by activity arcs. The events at the end of each thread that do not have an out-arc are terminal events. For example, Figure 32 illustrates a partial execution extracted from the AtHome TPN. The terminal events are E, L, and B. We refer to the set of terminal events as the fringe. A choice in a partial execution is an out-arc from a decision point to one of its target events. The partial execution in Figure 32 has two choices: B→D and L→P.

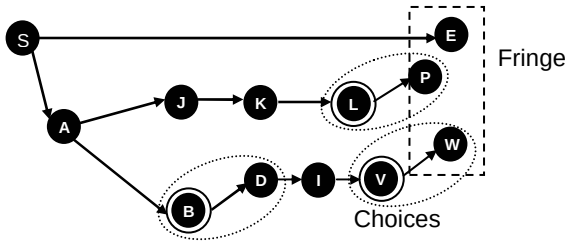


Figure 32: Example of a partial execution from the AtHome program TPN. The terminal events are E, P, and W and make-up the fringe of this partial execution. The choices are B→D, V→W, and L→P.

A partial execution is compactly encoded by its terminal events and choices. We define a formal mapping from a partial execution to its encoding as a pair ⟨*fringe*, *choices*⟩, where fringe is a set containing the terminal events in the partial execution, and choices is a set containing each choice in the partial execution. The encoding for the partial execution shown in Figure 32 is ⟨{E, P, W}, {B→D, L→P, V→W}⟩.

A state in the search space is a partial execution and is defined by the set of choices in that partial execution. For example, Figure 33 illustrates two partial executions that map to the same state. The two partial executions are encoded as a) ⟨{E, P, D}, {B→D, L→P }⟩ and b) ⟨{E, P, V}, {B→D, L→P }⟩. Both partial executions map to the same set of choices; thus, denote the same state. The optimal pre-planner, however, explores the search space in such a way that no search state is explored more than once. This is described by the expansion procedure in Section 5.3.

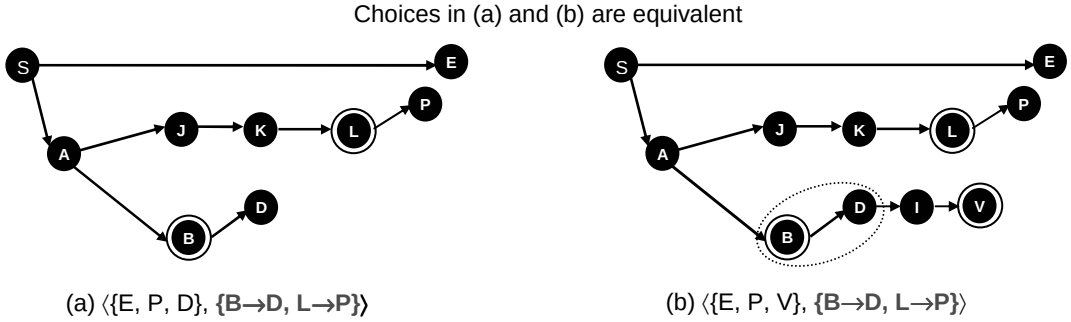


Figure 33: Two partial executions, (a) and (b), with equivalent choices {B→D, L→P}. Thus (a) and (b) map to the same state.

5.2.2 Search Tree

TPNA* explores the search space by expanding a search tree. A search tree is comprised of a set of nodes and branches, where each node denotes a search state. In general, the

root of the tree denotes the problems initial state. For TPN pre-planning, the root of the tree denotes the partial execution comprised of the set of all threads that originate at S, and either end at the first decision point reached or the end event E, whichever is reached first. Figure 34 shows the encoding of the root node s_1 and its corresponding partial execution, extracted from the TPN of Figure 15.

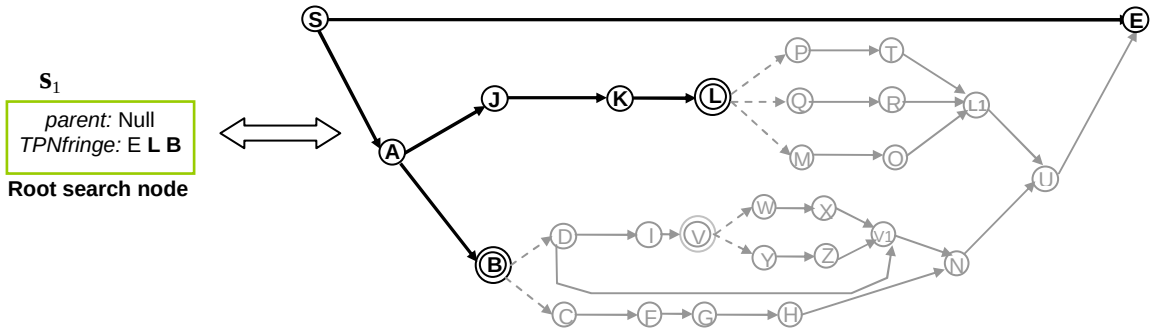


Figure 34: Root search node for the AtHome program (on the left) and it equivalent partial execution (on the right). The *TPNfringe* contains events E, L, and B. L and B in s_1 (shown in bold) are decision points that can be expanded further and E is the event signifying the end of that particular thread.

In general, each node and branch in the search tree contains a specific label, as shown in the complete search tree for the AtHome program in Figure 35. A node *n*, denoting a search state, is labeled with the following pair ⟨*parent*, *TPNfringe*⟩ where:

- *parent*(*n*): a reference to *n*'s parent node in the search tree. This is shown in the search tree as a branch from the parent to the child *n*.
- *TPNfringe*(*n*): a set of terminal TPN events in the corresponding partial execution.

For example, in Figure 32, *parent*(s_3) = s_1, and *TPNfringe*(s_3) = {V, E}. Correspondingly, a branch is labeled with a set {⟨d_1, e_1⟩, ⟨d_2, e_2⟩, … , ⟨d_n, e_n⟩}, where each pair in the set is a choice denoted by ⟨*decision-point*, *event*⟩. For example, branch (s_1, s_2) is labeled with the choices ⟨B, D⟩ and ⟨L, P⟩. These are the choices included in the partial execution denoted by s_2. The union of the labels along the unique path from a tree node *n* to the root node, and the events in *TPNfringe*(*n*) comprise an encoding of a partial execution. Figure 36 illustrates the partial execution corresponding to each node on the path from s_1 to s_{10}.

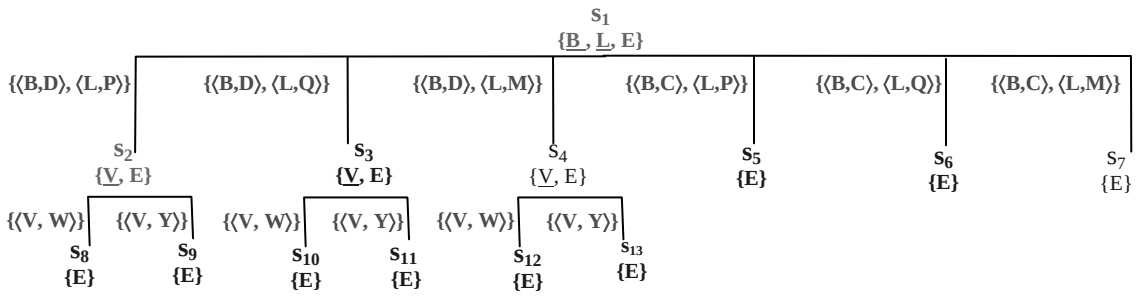


Figure 35: The complete search tree for the AtHome program. Each search node is labeled with a state name s_i and its corresponding TPN fringe events. The branches are labeled by choices. The choices for a node's state are the union of the branch choices along the path from the root to that node. The tree contains nine leaf nodes (s_5-s_{13}), each referring to the nine complete (not necessarily consistent) plans in the AtHome search space. The underlined TPN events are decision points that are expanded, in order to generate a node's children. Each set of TPN fringe events that contain E marks the end of another thread of execution from S to E. Node s_8 denotes the optimal execution.

To describe the mapping from a search tree node, s_i to its partial execution, we define the Node-To-Partial-Plan procedure in Figure 37. There are two main steps that comprise the procedure. The first step, Lines 1-7, is to create a set, $\Omega(s_i)$, containing all choices on each branch along the path from s_i to root. The second step, Lines 8-33, is to construct a partial-plan with events and arcs. This is done by beginning at the TPN start event S and tracing the threads that extend from S, using depth-first search until all threads end at an event in *TPNfringe*(s_i). While tracing each thread, if a decision point is encountered, then the thread corresponding to the choice in $\Omega(s_i)$ is added to the partial-plan (see Lines 20-24). For example, the search node s_9 (Figure 35) can be mapped to its corresponding state in the search space, creating Ω while walking upwards in the search tree along the path from s_9 to s_1. This results in $\Omega(s_9) = \{branch(s_9, s_2) \cup branch(s_2, s_1)\}$ = {⟨B, D⟩, ⟨L, P⟩, ⟨V, Y⟩}. Next, $\Omega(s_9)$ is mapped to a partial plan by starting at the start event S of the AtHome TPN, and tracing the threads that extend from S. Whenever a decision point is reached, the corresponding choice in $\Omega(s_9)$ is taken. For example, when the decision point L is reached on thread S→A→J→K→L the decision pair ⟨L, P⟩ directs the trace to continue the thread along P.

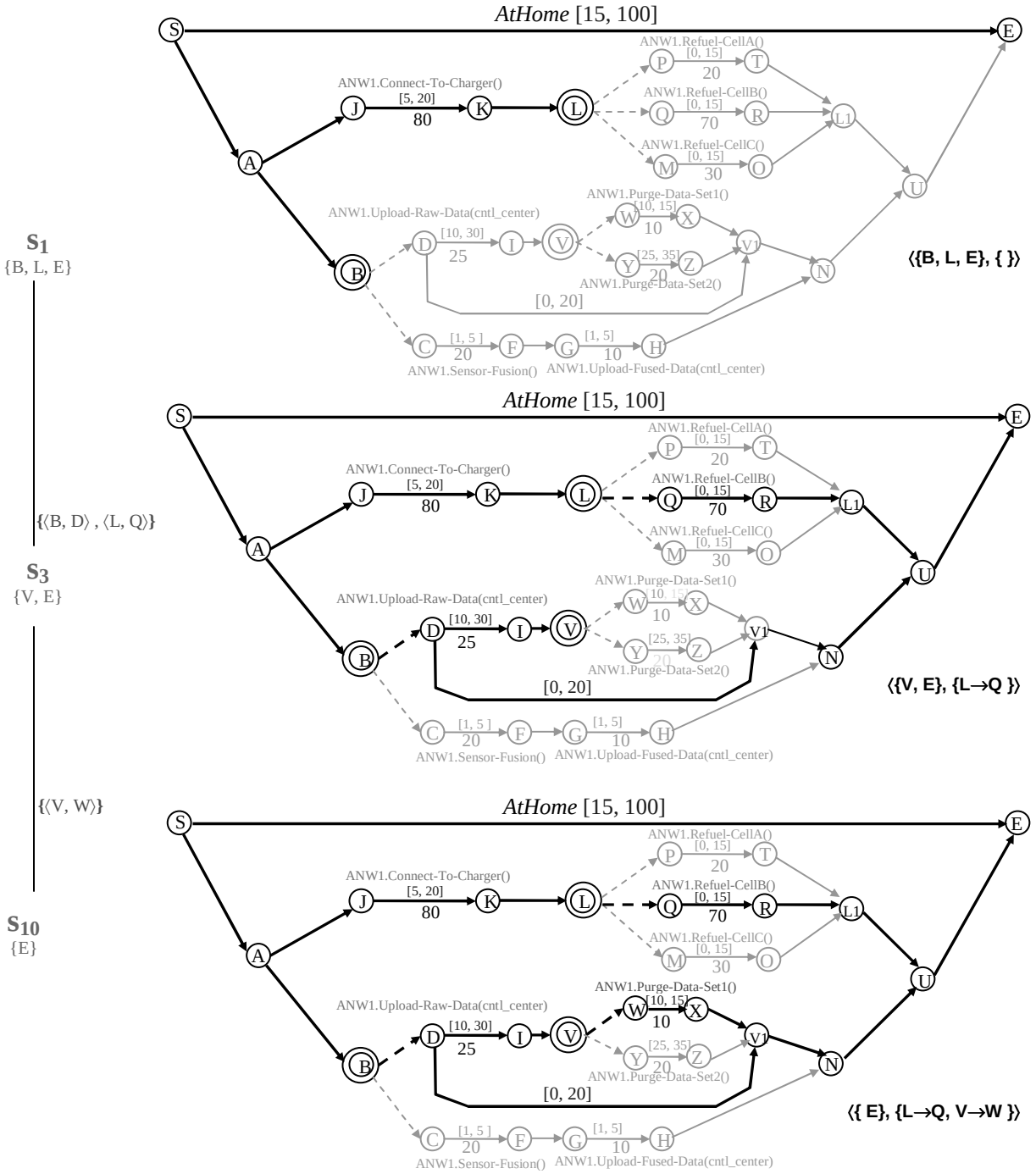


Figure 36: The search states corresponding to each search tree node along the path from the root s_1 to s_{11}.

```
procedure Node-To-Partial-Plan( Search Tree tree, Search Tree Node s, TPN tpn )
                                                    returns a set of arcs denoting a partial plan.
1.  //Step One: extract choices on path from n to root
2.  search tree node temp ← s
3.  Ω( s ) ← ∅
4.  while parent( temp ) ≠ root( tree ) do
5.      Ω( s ) ← Ω( s ) ∪ choice-set ( branch( temp, parent( temp ) ) )
6.      temp ← parent( temp )
7.  endwhile
8.  //Step Two: run a depth-first search on tpn following the choices  until each event in
    TPNfringe(n) is reached
9.  e ← start-event( tpn )
10. last-in-first-out stack stack ← ∅
11. Push( stack, e )
12. initialize visited( event ) of each TPN event to false
13. create partial plan partial-plan
14. events( partial-plan ) ← ∅
15. arcs( partial-plan ) ← ∅
16. while stack ≠ ∅ do
17.     e ← Pop( stack )
18.     if visited( e ) = false then
19.         visited( e ) ← true
20.         if decision-point( e ) = true then
21.             choice ← get-choice( Ω( s ), e )//returns a decision-pt, arc, and  target
22.             arcs( partial-plan ) ← arcs( partial-plan ) ∪ {get-arc( choice )}
23.             events( partial-plan ) ← events( partial-plan ) ∪ {get-target( choice )}
24.             Push( stack, target )
25.         else
26.             for each t_i ∈ target( e ) and visited( t_i ) = false do
27.                 arcs( partial-plan ) ← arcs( partial-plan ) ∪ {arc( tpn, e, t_i )}
28.                 events( partial-plan ) ← events( partial-plan ) ∪ {event( tpn, t_i )}
29.                 if t_i ∉ TPNfringe( e ) then
30.                     Push( stack, t_i )
31.             endfor
32. endwhile
33. return partial-plan
```

Figure 37: Algorithm to map a search tree node to its corresponding partial execution.

Detecting When Threads Re-converge

Notice that in Figure 36, TPN events E, U, N, and V1 are events where threads converge. For example, nodes s_3 and s_{10} denote partial executions with threads that converge at V1: thread D→V1 and W→X→V1, respectively. Node s_3 refers to a partial execution that includes the thread V1→N→U→E. Thus, when s_{10} is generated, it is only necessary to include the thread W→X→V1 and not continue the thread from V1 to E.

To detect when threads converge, a set of selected TPN events are maintained for each search tree node. The set contains the events that are currently included in the partial execution. For example the selected TPN events for node s_1 are {S, E, A, J, K, L, B} and the selected events for s_3 are {Q, R, L1, U, V1, N}. The union of these two sets is a set containing all the events in the partial execution denoted by s_3. The selected events of a search tree node contain all new events selected between the parent and the node. We create a table that maps each search tree node to its selected TPN events set (for example see Figure 38).

Node	Selected Events
s_1	{S, E, A, J, K, L, B}
s_3	{Q, R, L1, U, V1, N}.
s_{10}	{W, X}

Figure 38: An example of a table mapping the nodes along the path from s_1 to s_{10} to their corresponding selected TPN events. Each event included in a partial execution is added only once.

Figure 39 presents the function Threads-Converge, which returns true if it detects convergent threads, otherwise it returns false. Given search tree node *s*, the function proceeds upwards in the search tree along a path from a node *s* to the root. As it reaches each ancestor it tests if an event *e* is in a selected TPN events set of that ancestor. Threads-Converge is used in the node expansion procedure for two purposes: 1) to avoid extending threads that have already been extended, and 2) to detect when a cycle has been formed in the partial execution, prompting a test for temporal consistency (see Section 5.4).

```
function Threads-Converge?( Search Tree Node s, TPN event e, Table selected-events )
                                returns true if the TPN event e is included in the selected events
                                        set of an ancestor of Node n, otherwise false is returned.
1.  search tree node temp ← s
2.  while parent( temp ) ≠ NULL do
3.      if e ∈ selected-events[ temp ] then
4.          return true
5.      temp ← parent( temp )
6.  endwhile
7.  return false
```

Figure 39: Function that detects when an event is already included in a partial execution. It looks at the selected events in the ancestors of a search tree node. If the event in question is included in one of these sets, it is an indication that threads have converged, thus, forming a cycle in the partial execution.

Node Expansion and Goal Test

The TPN fringe events of a search tree node that are decision points are used to generate new child nodes. Recall that a decision point represents a choice between a set of threads in the decision sub-network. The set of all possible choices for the decision-points in the fringe of the search tree node is constructed by computing the cross-product of the sets of choices of each decision point in the fringe of the node. For example, the root node s_1 has two TPN fringe events, B and L, which are decision points. B has two choices, either C or D; and L has three choices, M, Q, or P. There are six possible sets of choices that result from performing the cross product operation, $\{C, D\} \times \{M, Q, P\}$, between the two decision points. The six combinations of choices, given as sets of ordered pairs, are:

1. $\{\langle B, D\rangle, \langle L, P\rangle\}$
2. $\{\langle B, D\rangle, \langle L, Q\rangle\}$
3. $\{\langle B, D\rangle, \langle L, M\rangle\}$
4. $\{\langle B, D\rangle, \langle L, P\rangle\}$
5. $\{\langle B, D\rangle, \langle L, Q\rangle\}$
6. $\{\langle B, D\rangle, \langle L, M\rangle\}$

During search tree expansion, a child node is created for each of the combinations that result from applying the cross-product to decision points in the fringe of a node. This is illustrated in Figure 35, where nodes s_2-s_7 are children of s_1. Note that the labels on each branch from s_1 to its children correspond directly to each combination of choices. Expansion only adds child nodes to the search tree that denote temporally consistent executions. This process is detailed in Section 5.4.

The expansion process terminates if a search tree node is removed from the queue, Open, that satisfies the Goal-Test, or if no consistent solution exists. Goal-Test tests if the partial execution denoted by a node is complete. That is, each thread in the partial execution originates at the TPN start event, S, and ends at the TPN end event, E.

A partial execution that is temporally consistent and satisfies the Goal-Test is referred to as a *feasible execution.* The feasible execution that minimizes the evaluation function *f*(*s*) is referred to as an *optimal execution*, and is the solution to the optimal pre-planning problem. If no feasible execution is generated then the expansion process terminates, returning no solution. The goal of optimal pre-planning is to find the optimal execution.

In summary, we frame the optimal pre-planning problem as a state space search problem. Given the encoding of a state as a partial execution, a search tree is constructed with the root denoting the problem's initial state. A child node is generated by expanding a node in the tree. At any point, a node in the search tree can be mapped to its corresponding partial execution by the Node-To-State procedure (Figure 37). The remainder of this chapter develops the expand procedure and TPNA* search.

5.3 Expansion

The process of generating new search tree nodes was first introduced in Section 5.2. We elaborate on this process by developing the Expand procedure, shown in Figure 45. The Expand procedure performs search tree node expansion in two phases. The first phase, given a search tree node *s*, is to extend the threads from each event in *TPNfringe*(*s*), until, along each thread, either a decision point is reached or the end event, E, is reached. While extending threads, if two or more threads re-converge, then a test for temporal consistency is performed (see Section 5.3.1 Figure 48). The first phase is accomplished by performing a modified version of a depth-first search (DFS), called Extend-Non-Decision-Events, which is invoked on Line 3 of the Expand procedure (Figure 47). This procedure is illustrated in Figure 44. In the figure, Extend-Non-Decision-Events is applied to the root search tree node. Each thread, originating at S is extended until either

the end event E is reached or a decision point in each thread is reached. In this example, the end event E is reached first, and then S extends its threads to the decision points L and B. The second phase creates new child nodes, where each node represents a unique combination of choices between decision points. The second phase, outlined on Lines 11-13 of Expand, is accomplished by executing the Branch procedure, given in Figure 50.

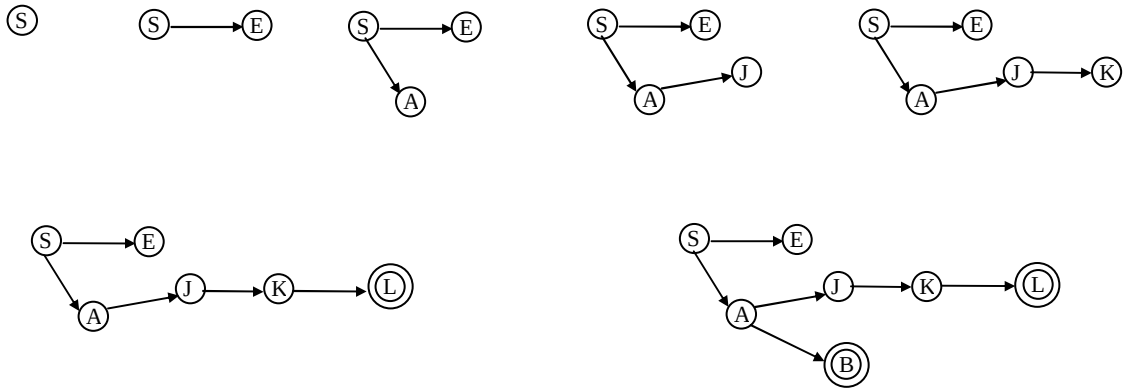


Figure 44: Example of Phase One, Extend-Non-Decision-Events, applied to the initial root tree node.

---

procedure Expand(Search Tree Node *s*) **returns** *set of search tree nodes.*

1. set of choices from decision points on the fringe of *s fringe-choices* ← ∅
2. boolean *consistent* ← false
3. set of TPN decision points *fringe* ← ∅
4. [*consistent*, *fringe*] ← Extend-Non-Decision-Events(*s*)//*Phase One: Extract Decision Points*
5. **if** *consistent* = false **then**
6. **return** ∅
7. **else if** *fringe* = ∅ **then**
8. complete(*s*) ← true
9. **return** { *s* } //*insert the updated s into Open*
10. **else**
11. initialize set *new-child-nodes* ← ∅
12. *new-child-nodes* ← Branch (*s, fringe*)//*Phase Two: Expand decision choices*
13. **return** *new-child-nodes*

Figure 45: Pseudo-code for expanding a node in the search tree.

The Expand procedure returns a set of new search tree nodes. The set is empty if the parent search tree node is determined to be temporally inconsistent in Phase One (Lines 4-5). For example, when s_6 in Figure 35 is expanded a cycle is formed. This

occurs when the thread from B converges at E. At this point, test for temporal consistency is done. The time bounds `[1,5]` on the thread extended from B are less than the minimum time bounds of the AtHome program, The node s6 is temporally inconsistent and Modified returns false and the empty set. If Extend-Non-Decision-Events extends all of the fringe events of *s* to the TPN end event E, without encountering any decision points, then *s* is marked completed and is the only element in the set returned by the Expand procedure, Lines 6-8. Otherwise, the set of search tree nodes will consist of child nodes generated by *s*, Lines 9-12.

5.3.1 Phase One: Extend-Non-Decision-Events

The objective of Phase One is to extend each thread of a search node up to its next decision point. Each event along a thread that is not a decision point represent a unique choice; hence, TPNA* can make these choices immediately rather than inserting these events into the queue, Open, for a delayed decision. The extension of threads is the responsibility of Extend-Non-Decision-Events. We precede the discussion of Extend-Non-Decision-Events with a brief summary of depth-first search.

In general, depth-first search operates on a problem specified as a graph with a start vertex, the initial state of the problem, and goal vertex, the goal state of the problem. Depth-first search grows a tree that, when complete, represents all unique paths from the start to the goal. During DFS, vertices at the deepest level in the tree are expanded first. This is done by maintaining a stack, where vertices are expanded in last-in-first-out order. Figure 46 illustrates the DFS tree created by applying DFS to the graph in Figure 29 (the heuristics and the weights are ignored).

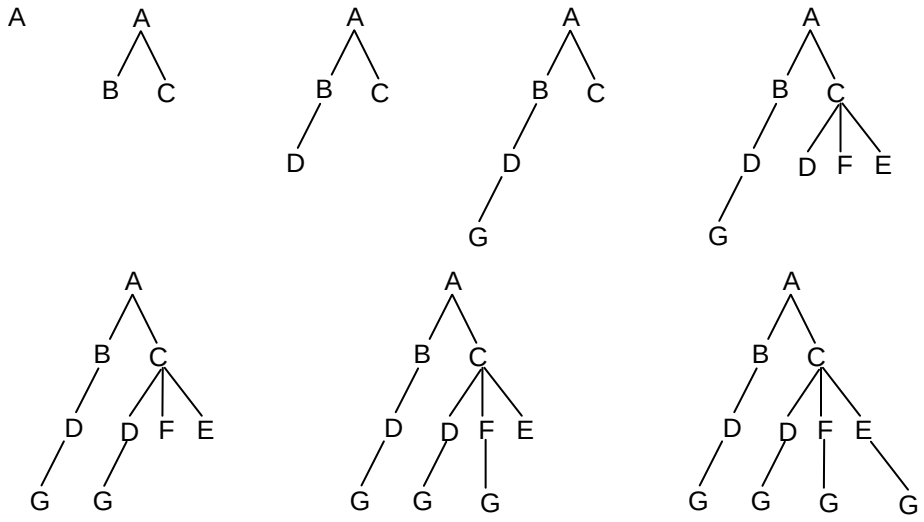


Figure 46: Progression of depth-first search applied to the graph in Figure 29. The start vertex is A and the goal vertex is G. The final tree shows the four possible paths from A to G.

The pseudo-code for Extend-Non-Decision-Events is shown in Figure 47. The argument to Extend-Non-Decision-Events is a search tree node *s*. A flag, indicating whether or not *s* is consistent, and a set of TPN events are returned. Each event in *TPNfringe*(*s*) is extended in depth-first order. While extending event e_i on a thread, one of three cases applies.

Case 1 (Lines 9-12): e_i is an event where two or more threads, in a partial execution, denoted by its search tree node *s*, re-converge.

Case 1 is detected by the Convergent-Threads function, described previously in Section 5.2. This case is significant for two reasons. First, it indicates that a cycle has been formed in the corresponding partial execution, as shown in Figure 48. Thus, a test for temporal consistency must be performed. In general, events that are not re-convergent must be consistent as long as their temporal constraint `[lb,ub]` has the property `lb` ≤ `ub`. To verify temporal consistency in a partial execution, the SDSP-FIFO algorithm, presented in Chapter 4, is executed on the temporal constraints of the partial execution encoded as a distance graph. For example, node s_5 corresponds to choices B→C and L→P (Figure 35). When s_5 is generated by s_1, its fringe initially contains C and P. When s_5 is expanded, C is extended to the end event E. E is already included in the selected node set of an ancestor node of s_5, namely s_1. An ancestor of a search tree node *n* is any search tree node on the path from the root to *n*. Thus, when E is reached by expanding s_5,

a cycle has been formed in the current partial execution, denoted by s_5. This signals a test for temporal consistency [38]. If the partial execution violates its simple temporal constraints, as in s_5, then Extend-Non-Decision-Events is terminated and the node being expanded is pruned from the search tree. Otherwise, the procedure continues.

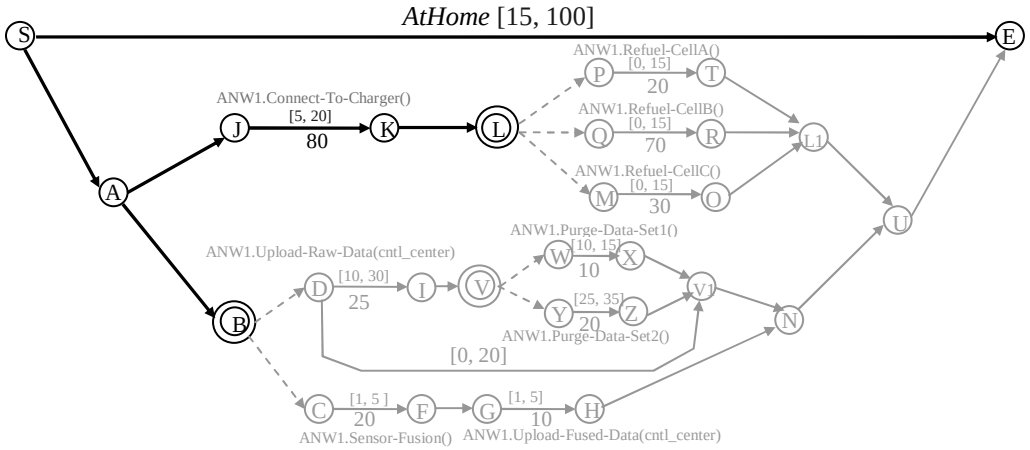
```
procedure Extend-Non-Decision-Events ( Search Tree Node s )
          returns true and the set of decision points reached by extending the
                  threads from the TPNfringe.
    of s, otherwise false is returned.
1.  initialize stack ← ∅
2.  initialize updated-fringe ← ∅ //contains decision-points reached by expanding threads in s
3.  for each event e_i ∈ TPNfringe( s ) do
4.      threadcosts ← 0
5.      Push ( e_i, stack )
6.      visited( e_i, true )
7.      while stack ≠ ∅ do
8.          event ←Pop( stack )
9.          //Case One: If e_i induces a cycle then check temporal consistency
10.         if Threads-Converge( event ) then
11.             if Not( Temporally-Consistent( distance-graph( s ) ) ) then
12.                 return false
13.         //Case Two: If e_i is a decision point then stop extending this thread
14.         else if decision-point( event ) = true then
15.             updated-fringe ← updated-fringe ∪ event
16.         //Case Three: Continue DFS, if target of e_i is visited then check temp consistency
17.         else
18.             for each t_i ∈ target( event ) do
19.                 if visited( t_i ) = false then
20.                     threadcosts ← threadcosts + cost( event, t_i )
21.                     visited( t_i, true )
22.                     selected-nodes( s ) ← selected-nodes( s ) ∪ { t_i }
23.                     Push( stack, t_i )
24.                 else
25.                     if Not( Temporally-Consistent( distance-graph( s ) ) ) then
26.                         return false, ∅
27.             endfor
28.     endwhile
29.     selected-nodes( s ) ← selected-nodes( s ) ∪ { e_i }
30. endfor
31. TPNfringe( s ) ← updated-fringe
32. g( s ) ← g( s ) + threadcosts
33. return true, updated-fringe
```

Figure 47: Pseudo-code for the Phase One Extend-Non-Decision-Events procedure.

The second reason Case 1 is significant is to avoid redundant extension of threads. Consider the search node s_{11}, for example. When s_{11} is expanded, the thread starting at V

converges at V1 with thread B→D→V1. TPN event V1 was extended during the expansion of an ancestor node of s_{11}, s_3. It would be redundant to extend V1 again, since it was already included in the partial execution by an ancestor of s_{11}.

Search tree node $\mathbf{s_1}$:
selected-nodes($\mathbf{s_1}$) = {S, E, A, J, K, L, B}



Search tree node $\mathbf{s_5}$ (during expansion, after extending fringe event C):

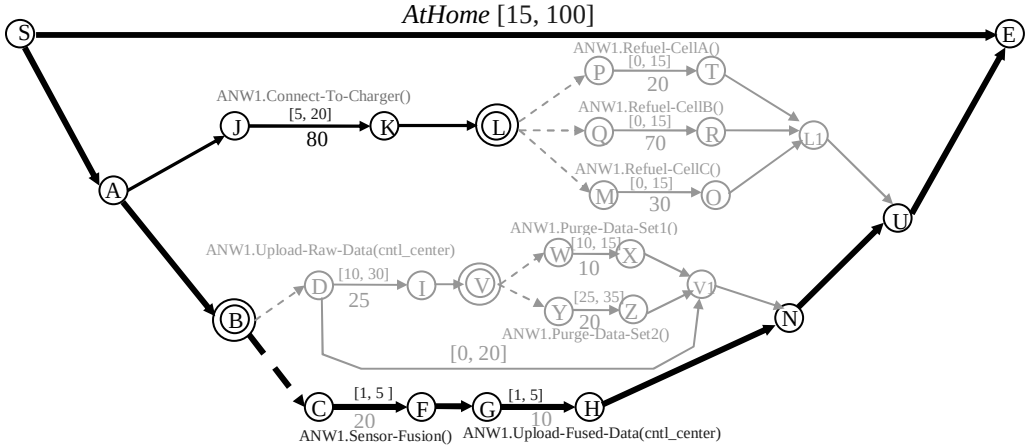


Figure 48: A cycle is formed at when search tree node s_5 is expanded. The cycle, between the partial execution of s_1 and s_5, is shown in thick bold. Temporal consistency is checked on the current partial execution, and it fails. Thus, expansion of s_5 is terminated and the node is pruned from the search tree.

Case 2 (Lines 13-15): e_i is a decision point signaling a choice between possible threads. Extend-Non-Decision-Events stops extending the current thread when a decision point is reached and updates the fringe with the decision-point. For example, the expansion of the root node s_1 extends three threads, two of which end at decision points B and L.

Case 3 (Lines 16-27): This case is applied when neither Case 1 nor Case 2 applies.

In Case 3, e_i is extended to each of its targets that have not been visited, as in the general depth-first search. If a target has been visited, then a cycle has formed in the corresponding partial execution. The cycle was formed during the expansion of the current node *s* and not between *s* and an ancestor node. For example, when the search tree node s_5 is first generated by its parent, it contains two decision points in its fringe, P and C (Figure 48). If C is extended first, then it extends to E along the thread C→F→G→H→N→U→E, marking each event as visited. Then P is extended and the event U, which was marked visited, is reached (Figure 49). At this point a cycle has been formed during the expansion of a search tree node, s_5, and a test for temporal consistency is performed (Lines 25-26).

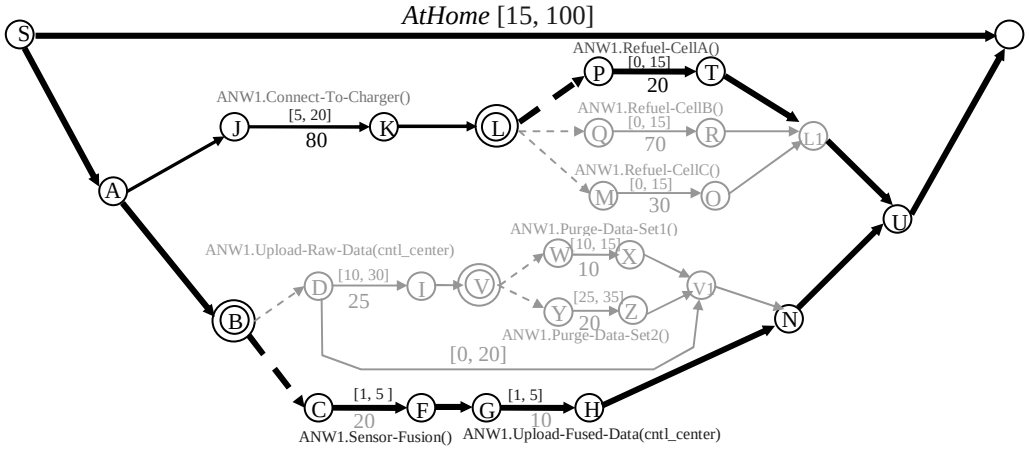


Figure 49: Extend-Non-Decision-Events applied to s_5 and the event U is reached by the threads from B and L.

A partial execution is complete if, when Extend-Non-Decision-Events is applied, all threads extending from a fringe event reach the end event E, without causing a temporal inconsistency. If this is the case, then there are no decision points in the fringe, and the partial execution is considered a feasible execution. This is detected in the Expand procedure (Figure 43, Lines 6-8).

5.3.2 Branch

The second phase of TPNA* search, introduced in Section 5.2, creates new child nodes from *s*. This is performed by the Branch procedure, Figure 50. The procedure begins by creating sets of choices for each decision-point in the TPN fringe of *s* (Lines 1-8). A set contains the target events of a decision point. For example, after Extend-Non-Decision-Events is applied to the root node s_1, the fringe of s_1 contains two decision points, B and L. Thus, the sets created from B and L are {P, Q, M} and {C, D} respectively. Next, to create all possible combinations of choices, the cross product is performed on the sets, Line 9. For each combination, a new search node, with parent *s*, is created, Lines 11-20. Finally, the Branch procedure returns the set of child nodes generated by their parent search tree node, *s*.

---

procedure Branch (Search Tree Node *s*, Set of Decision points *dec*) **returns** *set of child nodes.*

1. initialize sets of possible combinations *combinations* ← ∅
2. **for** each decision-point $d_i \in$ *dec* **do**
3. $choices_i \leftarrow \{ \{t_j\} | \{t_j \in$ target(d_i) $\}$
4. **endfor**
5. *combinations* ← Set-Cross-Product(*choices*)
6. initialize set of search tree nodes *children* ← ∅
7. **for** each set $cset_i \in$ *combinations* **do**
8. *current-cost* ← 0
9. search tree node *child* ← Make-Search-Node($cset_i$)//*initializes TPNfringe(child) w/ events in* $cset_i$
10. **for** each target $t_j \in$ *child* **do** //*get the cost on arc from decision-point of target to target*
11. *current-cost* ← *current-cost* + cost(decision-point(t_j), t_j)
12. **endfor**
13. Set-Parent-Node(*child, s*)
14. *children* ← *children* ∪ { *child* }
15. **endfor**
16. **return** *children*

Figure 50: Pseudo-code for the Branch procedure, which creates new child search nodes.

5.4 Computing the Estimated Cost of a Search Node

To define the optimal pre-planner as a forward heuristic search, we develop an evaluation function for search tree nodes in order to guide TPNA* search. The current description of TPNA* search excludes the activity costs, and thus, finds a feasible execution. To find

the optimal feasible execution, we introduce two measures of cost. The first, is the path cost g(*s*) for a search tree node *s*. The second is an admissible heuristic cost, *h*(*s*), for a each search tree node s. The sum g(*s*) and *h*(*s*), as in A* search, is the estimated cost of a solution through *s*, and thus, is used as our evaluation function.

5.4.1 Computing the Path Cost g(s)

The path cost g, of a search tree node is the sum of the costs of each arc in the partial plan of the search state. When a search tree node *s* is first created, it is initialized with the cost of its parent, *g*(parent(*s*)), (see Section 5.4.2). When *s* is expanded by Extend-Non-Decision-Events, the costs along the arcs of each thread being extended are summed (Figure 47, Line 20) and added to its initial cost. For example, the actual path cost of the partial execution denoted by node s_8 is the sum of each distinct arc, from the start event S, along each thread ending at E. For example, assume that the default arc cost for the AtHome TPN (Figure 51) is 5 units (not including the arc from S to E). Then $g(s_8) = g(s_2)$ + c(V, W) + c(W, X) + c(X, V1) = 190 + 5 + 10 + 5 = 210 units, where c(*x*, *y*) is the cost on the arc from TPN event *x* to TPN event *y*. If TPNA* only uses g as its cost function then the search operates as a uniform cost search. That is, no heuristic is used to guide the search.

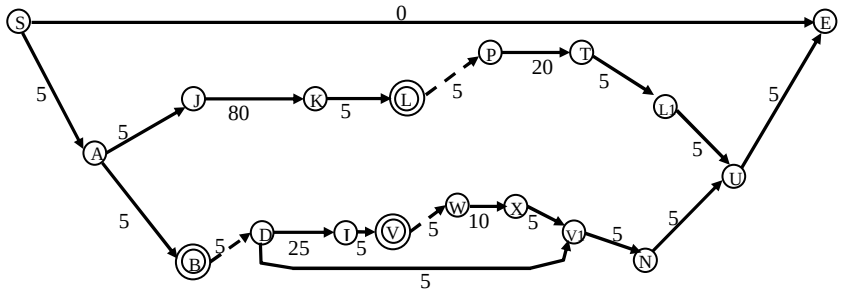


Figure 51: The partial execution denoted by search node s_8. The network contains a default arc cost of 5 units for all arcs without activity costs. The path cost for $g(s_8) = 210$ units.

5.4.2 Extracting the TPN Heuristic Cost

A uniform cost search will find the shortest path to all states in an optimal pre-planning problem. To efficiently focus TPNA* search towards the optimal execution, we develop an admissible heuristic for a search tree node. The heuristic cost of a search tree node is formed by computing a heuristic *h*(*e*) for all events *e* in the set of TPN events. Then,

given $h(e)$ for all events in the fringe of search tree node s, take the max $h(e)$ over all events in *TPNfringe*(*s*). We refer to this search tree node heuristic as DP-max.

Summary of the Heuristic used in the Heuristic Search Planner (HSP)

To develop a heuristic cost for search tree nodes, we first review to related work on heuristic search planners. Heuristic planners, such as FF [14] and HSP (Heuristic Search Planner) [1], compute heuristics based on the encoding of the problem. HSP considers two ways to compute the estimated cost of achieving a goal of a given planning problem: additive costs and max costs [1]. The additive heuristic cost of an event *e* is $h(e) = \sum_{t \in targets(e)} [c(e,t) + h(t)]$. The additive heuristic cost works well if the problem ensures that sub-goals are independent [1]. For example, consider the parallel sub-network in Figure 52. Each thread extending from M contains independent sub-goals that re-join at the end, or goal state, N. To compute an admissible heuristic cost for a planning problem with independent sub-goal, the additive heuristic can be applied. The cost of executing M is at least the sum of the costs of executing the commands on its parallel threads.

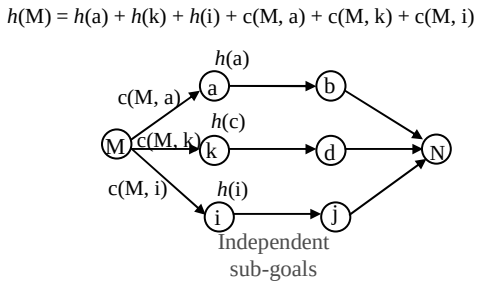


Figure 52: Example of additive heuristic cost for a parallel sub-network, where each thread contains its own independent sub-goals. The heuristic cost for event the event S can be expressed as the sum of the heuristic costs of its targets; *h*(a), *h*(k), and *h*(i), plus the costs from S to each of its targets; c(M, a), c(M, k), and c(M, i).

If the additive heuristic function is used to compute the heuristic cost of event J in the parallel sub-network in Figure 53, then *h*(J) is inadmissible. The inadmissible heuristic for J is caused by including the cost from the dependent sub-goal I in each heuristic *h*(a), *h*(d), and *h*(p). We refer to this issue as double counting.

To address the issue of double counting, [1] suggests a max heuristic. The max heuristic for an event *e* is $h(e) = \max_{t \in targets(e)} [c(e,t) + h(t)]$. For example, max heuristic for S

in Figure 53, below, is $h(J) = \max[\text{cost}(J, a) + h(a), \text{cost}(J, d) + h(d), \text{cost}(J, p) + h(p)]$. In this case, the heuristic cost of S is admissible. However, the max heuristic often severely underestimates, and thus, is not very informative. For example, if $h(a) = 1000$, $h(d) = 999$, and $h(p) = 900$, then $h(J) = 1000$. This estimate is significantly low, since all threads from S must be executed.

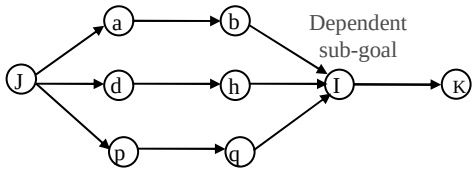


Figure 53: The additive heuristic applied to J results in an admissible heuristic for J. The heuristics for a, d, and p, each include the cost from their dependent sub-goal I→K. The max heuristic does result in an admissible heuristic for J, however, it is not informative.

To compute the heuristic cost of a set of TPN events, we exploit the structure of the TPN and apply the dynamic programming principle. As a result, we can get an improved heuristic cost of TPN events with dependent sub-goals.

Heuristic Cost for TPN Events Using the Dynamic Programming Principle

To develop a heuristic for TPNs, we note that a TPN is composed of explicitly defined sub-networks. That is, the mapping of each RMPL combinator; `sequence`, `parallel`, and `choose`, to a TPN sub-network, explicitly defines sub-goals, with a start event and end event, in the TPN. For example, in the AtHome TPN (Figure 27) the event B is the start event of a decision sub-network, ending at the sub-goal N, and the event A is the start event of a parallel sub-network, ending at the sub-goal U. Given a TPN with explicit sub-goals, we can define an exact heuristic on the *relaxed* TPN. A relaxed TPN is one in which temporal constraints are not considered. With a relaxed TPN, the optimal pre-planning problem is reduced to a shortest path problem in the problem state space, where the shortest path corresponds to a complete minimum cost execution through the TPN while ignoring time bounds. We compute the heuristic cost $h(e)$ for every TPN event e. We accomplish this by proceeding backwards from the end event E, applying the dynamic programming principle in order to compute the shortest path from each

preceding event *e* to E, until the start S is reached. The algorithm Compute-TPN-Heuristic is given in Figure 57.

Definition The heuristic cost of an event e_i, $h(e_i)$, in a TPN is defined by one of two equations:

1. The heuristic cost of a TPN event e_i that is a decision point, as seen in Figure 54, is defined as follows:

$$h(e_i) = \min_{t \in targets(e_i)} [c(e_i, t) + h(t)] \qquad \text{(Equation 3)}$$

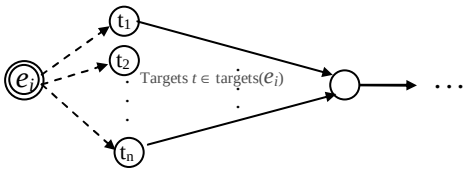


Figure 54: Example of a generalized decision sub-network.

An example of applying Equation 3 to a decision sub-network for the AtHome TPN is as follows. The AtHome TPN contains three decision sub-networks. The first one starts at B and ends at N, the second starts at L and ends at L1, and the third starts at V and ends at V1. Given a default arc cost of 5 units, the heuristic cost of decision point L, for example, is h(L) = min(c(L, P) + h(P), c(L, Q) + h(Q), c(L, M) + h(M)) = min(40, 90, 50) = 40.

2. The heuristic cost of an event e_i that is a non-decision point is defined as follows:

$$h(e_i) = \sum_{t \in targets(e_i)} [c(e_i, t) + h(t)] - [(|targets(e_i)| - 1) \times h(parallel\text{-}end(e_i))] \qquad \text{(Equation 4)}$$

where *targets*(e_i) are the events that are at the tail of each out-arc of e_i, |*targets*(e_i)| is the number of targets, and *parallel-end*(e_i) is the corresponding event where the threads re-converge (Figure 55).

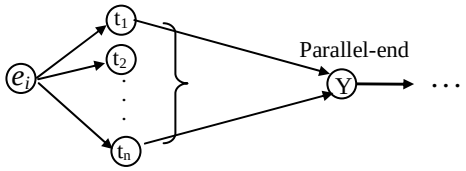


Figure 55: Example of a generalized parallel sub-network. The heuristic cost of a parallel start event e_i is the sum of each thread within the parallel sub-network, plus the heuristic cost of the parallel end.

Equation 4 can be applied to parallel sub-networks and simple command arcs in a TPN. Recall that a parallel sub-network is a point in the TPN where threads fork and re-converge. For example, the AtHome TPN contains three parallel sub-nets. The first one starts at event S and ends at event E, the second one starts at A and ends at U, and the third parallel sub-net starts at D and ends at V1. The heuristic cost of event D, for example, is h(D) = c(D, V1) + c(D, I) + h(V1) + h(I) – [h(V1)] = 5 + 25 +15 + 40 – 15 = 70, where the arcs that are not labeled with a cost have a default of 5 units.

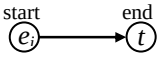


Figure 56: A simple command arc.

A simple command arc is defined by a start event that contains only one out-arc connected to its end event (Figure 56). Its heuristic $h(e_i)$ is defined as follows:

$$h(e_i) = c(e_i, t) + h(t) \qquad \text{(Equation 4a)}$$

For example, the event Y, in the AtHome TPN, has a heuristic cost of h(Y) = c(Y, Z) + h(Z) = 20 + 20 = 40. The heuristic cost of a simple command is a degenerate form of Equation 4. The first term of Equation 4 occurs only once for the one target of e_i. The last term on the right-hand-side of Equation 4 is 0, because the number of targets is one.

The heuristic cost of all events in a TPN can be computed *a priori* using the dynamic programming (DP) principle. The TPN heuristic is admissible, but approximate, because it computes the cost of a relaxed TPN that ignores all its temporal constraints. The procedure for computing the heuristic cost for all events in a TPN is shown in Figure 57.

```
procedure Compute-TPN-Heuristic ( TPN tpn, Start S, End E ) returns heuristic cost for
                                                              all TPN events.
1.  initialize list stack ← ∅
2.  initialize event current-event ← E
3.  h( current-event ) ← 0
4.  visited( current-event, true )
5.  estimated( current-event ) ← true
6.  Push ( current-event, stack )
7.  while stack ≠ ∅ do
8.      current-event ← Pop( stack )
9.      if all targets( current-event ) are estimated then
10.         //Case 1
11.         if decision-point( current-event ) = true then
12.             h( current-event ) ← min_{t∈targets(event)} [ c(event,t) + h(t) ]
13.         //Case 2: Non-decision point subnetwork
14.         else
15.             if | out-arcs( current-event ) | > 1 then //Case2a: Parallel sub-network
16.                 p-end ← parallel-end( current-event )
17.                 for each t_i ∈ targets( current-event ) do
18.                     temp-cost ← cost( current-event, t_i ) + h( t_i )
19.                 endfor
20.                 h( current-event ) ← temp-cost − [( | targets( current-event ) | − 1 ) × h( p-end )]
21.             //Case 2a.  simple command
22.             else
23.                 h( event ) ← cost( current-event, target ) + h( target )
24.         for each p_i ∈ predecessor( current-event ) do
25.             if visited( p_i ) = false then
26.                 visited( p_i, true )
27.                 Push( stack, p_i )
28.         endfor
29.         estimated( current-event ) ← true
30.     else
31.         Push-Back ( stack, current-event )
32. endwhile
33. return true
```

Figure 57: Pseudo-code to compute the heuristic cost for events in a TPN. The predecessors p_i of an event e, is the set of events at the head of each incoming-arc from $p_i \rightarrow e$.

The problem of finding the heuristic cost from the end event E from each TPN event is similar to the single destination shortest path problem, described in Chapter 4. In this case, the algorithm to solve the single-destination shortest path problem is applied to an entire TPN and additional book-keeping is required in order to avoid double counting of costs in the TPN. The algorithm in Figure 57 uses depth-first search and the dynamic programming principle in order to walk backward through the TPN, from the end E to the start S. During each iteration, if the heuristic for all targets of *current-event* have been

estimated, then *h*(*current-event*) is computed. Figure 58 shows the heuristic costs for the events in the AtHome TPN (Figure 59).

h(E) = 0	h(K) = 45	h(D) = 70
h(U) = 5	h(J) = 125	h(H) = 15
h(L1) = 10	h(N) = 10	h(G) = 25
h(T) = 15	h(V1) = 15	h(F) = 30
h(P) = 35	h(X) = 20	h(C) = 50
h(R) = 15	h(W) = 30	h(B) = 55
h(Q) = 85	h(Z) = 20	h(A)=185
h(O) = 15	h(Y) = 40	h(S) = 190
h(M) = 45	h(V) = 35	
h(L) = 40	h(I) = 40	

Figure 58: The heuristics for the event in the AtHome TPN using the default cost of 5 units.

The cost computed above specifies the cost to go, starting at any event in a single TPN thread. Our remaining step is to use this cost to compute the cost to go of a search tree node.

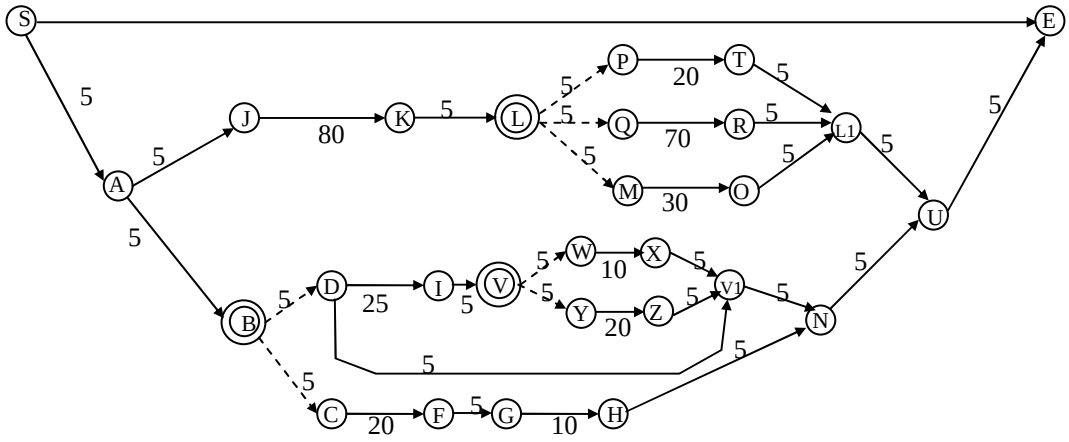


Figure 59: TPN for the AtHome strategy with all its arc costs.

DP-Max: *A Heuristic Cost for Search Tree Nodes*

Note that a search tree node may contain one or more decision points in its *TPNfringe*. The heuristic cost of the search tree node is the sum of the heuristic costs of each of the decision points d_i in its fringe. However, this still can result in some double counting. By

definition of a TPN, all threads re-converge either at sub-goals, or at the end event E. Thus, the heuristic cost for each decision point in the fringe may include the cost of a shared sub-goal. Figure 60 depicts a partial execution with decision points d_i and d_j in its fringe. If h(d_i) and h(d_j) are summed, then $h(p_1)$, their shared sub-goal, would be counted twice.

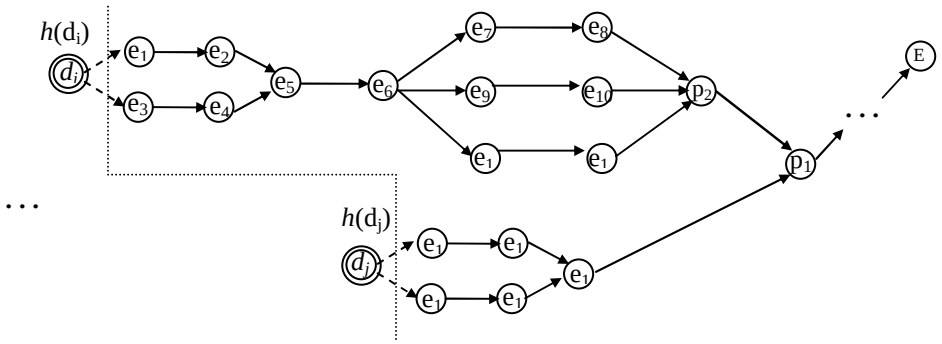


Figure 60: Example of the fringe of a search state with decision points d_i and d_j. Thread from the choices of either decision point will re-converge at p1.

To avoid this remaining element of double counting, we can apply the max heuristic in order to select the cost of the decision point with the highest cost, of those in the fringe of a search node. We refer to the function used to compute the heuristic cost of a search node as DP-Max.

Our essential contribution is that the DP-Max estimate is much closer than that of the HSP Max heuristic. All computation of the partial execution denoted by a search tree node *s*, starting from the max decision point is exact. This is a result of the heuristic, *h*, computed for all TPN events by dynamic programming.

In summary, the Compute-TPN-Heuristic procedure computes the heuristic cost for each TPN event computed prior to the execution of TPNA* search. As a result, to compute the estimated cost of search tree node, a constant table-lookup operation, is executed to find the heuristic value for each event during optimal pre-planning.

5.5 TPNA* Search and Properties

This section discusses the overall TPNA* search algorithm and its properties. The search is systematic, and therefore, each state, denoted by a partial execution, is visited at most once. We argue that our formulation of the search space is complete, that is, TPNA* is

guaranteed to find the optimal solution if, and only if, a temporally consistent, complete execution exists.

5.5.1 The Algorithm

The pseudo-code for the TPNA* search algorithm is shown in Figure 61. The algorithm begins by creating a root node s_{root} based on the start event of the input TPN (Lines 3). The estimated cost of s_{root}, is computed and s_{root} is inserted into the priority queue, Open (Lines 4-5). When the main loop, shown on Lines 6-18, is executed, the search tree nodes with the minimum cost, $f(n)$, are removed from Open and expanded in best-first order. As each node, s_{best}, is removed from Open, the Goal-Test is performed to test if the partial execution is complete. If the Goal-Test is satisfied, then the optimal solution has been reached. Otherwise, the expand procedure is invoked (Line 11). If there are no nodes generated by applying the Expand procedure to a search tree node *s*, then there exists no consistent solution that extends *s*. Thus, *s* is pruned from the search tree, Line 18.

---

procedure TPNA* (TPN *tpn*, Start Event *S*, End Event *E*) **returns** *an optimal solution that is complete and consistent if, and only if, one exists.*

```
1.  Compute-Heuristic( tpn )
2.  Open ← ∅
3.  create root node s_root ← Make-Root( s_root, S )
4.  f( s_root ) ← g( s_root ) + h( s_root )
5.  Insert( s_root, Open, f( s_root ) )
6.  while Open ≠ ∅ do
7.      s_best ← Remove-Best( Open )
8.      if Goal-Test( s_best, tpn, E ) then
9.          return Extract-Solution( s_best )
10.     else
11.         set of search tree nodes tree-nodes ← Expand( s_best )
12.         if tree-nodes ≠ ∅ then
13.             for each s_child ∈ new-nodes do
14.                 Insert( s_child, Open, f( s_child ) )
15.             endfor
16.         else
17.             Prune( s_best )
18. endwhile
19. return no solution
```

Figure 61: TPNA* is the driving procedure for the optimal pre-planning system.

The Prune procedure removes a search tree node *s* that violates its temporal constraints. If, after removing *s*, the parent of *s* has no children, then the parent also is pruned from the tree. This process continues walking up the tree towards the root until the search tree node with at least one sibling is pruned. In the worst case, the Prune procedure will require walking up the search tree from the deepest leaf in the tree to the root.

```
procedure Prune ( Search Tree Node s,  Search Tree search-tree )
1.   search tree node temp ← s
2.   while ( temp ) ≠ NULL do
3.       if root( temp ) = true then
4.           Delete( search-tree, temp )
5.           return
6.       else
7.           parent ← parent( temp )
8.           Remove-Child( parent, temp )
9.           Delete( search-tree, temp )
10.          if  parent has no children then
11.              temp ← parent
12.          else
13.              temp ← NULL
14.  endwhile
```

Figure 63: Pseudo-code to prune a search tree node from the search tree.

AtHome Example

The TPNA* search process for the AtHome strategy is illustrated in Figure 64. Each node is denoted by ⟨*s* : iteration# : *f*(*s*) : g(*s*) : *h*(*s*)⟩, where iteration# represents the iteration at which the node was expanded. The search tree nodes that are not included in the search tree were pruned from the search space. Notice that the size of the tree is much smaller than the complete search tree in Figure 35. For example, the choice B→C results in a temporal inconsistency. Thus, when s_1 attempts to generate child nodes s_5, s_6, and s_7, the search detects the temporal inconsistency and s_5, s_6, and s_7 are never added to the search tree. Additionally, by using the admissible heuristic for a search tree node, it is possible for some states to not be explored. In this example, the heuristic guided the search directly to the optimal execution denoted by s_8. However, if s_8 were inconsistent, TPNA*

search would prune s_8 and then expand the next best search tree node in Open. In the worst case, the only consistent execution is the feasible execution with the highest cost.

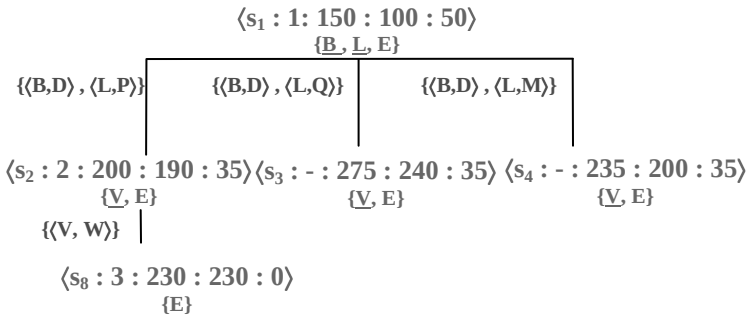


Figure 64: Search process for TPNA* applied to the AtHome control program. Each search tree node is denoted by ⟨*s* : iteration# : *f*(*s*) : g(*s*) : *h*(*s*)⟩. The heuristic cost of s_1 is max(*h*(B), *h*(L)) = 50.

The TPNA* search algorithm is complete, consistent, and systematic. The search can be improved by using better techniques for using inconsistencies (e.g. conflicts) to explore the search space [41]. For example, all partial executions that contain choices B→C or V→Y result in a temporal inconsistency. Once this is first detected, then using conflict-directed search technique, the portion of the search space can be pruned that contains partial executions with either of those choices.

5.6 Summary

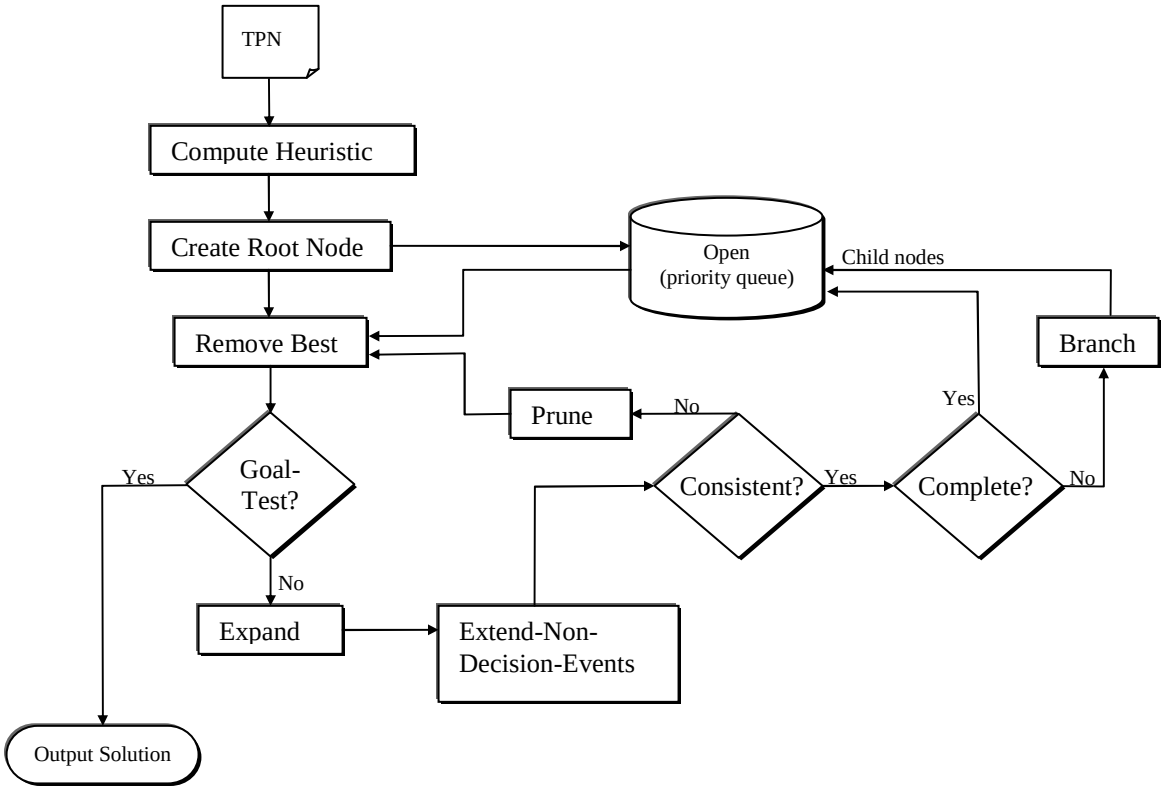


Figure 65: Flow chart for the optimal pre-planning process.

This chapter described the optimal pre-planning process. The main algorithm driving the process is a variant of A* applied to temporal plan networks, TPNA* search. We defined a function to compute the admissible heuristic value for events in a TPN in Section 5.2. Then we formulated the search space and search tree with nodes and branches in Section 5.3. Finally we presented the expansion procedure for generating nodes in the search tree. A flow chart of the optimal-pre planning process is shown in Figure 65.

Chapter 6

Optimal Activity Planning and Path Planning

In Chapter 5, we developed an optimal pre-planner that adopted an A* search strategy to find the least-cost complete and consistent execution of a TPN. The optimal pre-planning problem was formulated as a TPN search problem with activities, costs, and temporal constraints. In this chapter, we extend optimal pre-planning problems to satisfy location constraints on activities. To address this class of problems, we extend the optimal pre-planner, to include spatial reasoning, via a roadmap-based path planner.

Movement of an AV or ground vehicle is often constrained by the vehicle dynamics (e.g. turning radius). To address this, we exploit a particular effective kinodynamic path planner, called Rapidly-exploring Random Trees (RRTs). An RRT is a roadmap that explores the state space by randomly growing towards the destination. This chapter introduces a globally optimal, unified activity planning and path planning algorithm, called UAPP (Unified Activity and Path Planning). UAPP interleaves activity and path planning by searching a hybrid graph, called a Roadmap TPN (RMTPN) that connects TPNs and RRTs. Search over a RMTPN allows UAPP to find the globally optimal complete and consistent plan composed of activities and paths. We focus on mission strategies developed for a single autonomous vehicle (AV) that navigates from region to region in order to execute activities.

6.1 Motivation

Chapters 3 and 5 described two of the three strategies, Enter-Building and AtHome, which comprise the Search-Building mission described in Chapter 1. In this chapter, we focus on the third strategy, called Exit-Building. The Exit-Building strategy requires the autonomous vehicle (AV), ANW1, to deploy its team of tiny *helpbots*, which are equipped with two-way radios and minimal first-aid supplies. These are deployed in a specific location, an office complex, where potential victims may be trapped. ANW1 first deploys its *helpbots*, and then recovers the chemical detecting robots, *chembots*, and quickly scans their data for any extreme hazard. Finally, ANW1 exits the office building and communicates with the mission control center. The control program for the Exit-Building strategy is shown in Figure 66, and its equivalent TPN representation is shown in Figure 67.

Exit-Building RMPL Control Program

```
1.  (Exit-Building [0,150]
2.      (sequence
3.           ;;Lower helpbots
4.          (choose
5.              ((sequence
6.                  ( ANW1.Open-Rear-Hatch(20) )
7.                  ( ANW1.Lower-Helpbots(30 chembots.location) [5, 20] )
8.              ) (LaboratoryTwo) )
9.              (sequence
10.                 ( ANW1.Open-Rear-Hatch(20) )
11.                 ( ANW1.Lower-Helpbots(45 HallwayC) )
12.             )
13.          ) ;;end choose
14.         ;;Retrieve chembots and take pictures
15.         (parallel
16.             ( ANW1.Explore(50 LaboratoryTwo) [0, 10] )
17.             ((parallel
18.                 (sequence
19.                     ( ANW1.Retrieve-Chembots(40 chembots.location) [15, 20] )
20.                     ( ANW1.Scan-Chembot-Data(20) [10, 20] )
21.                 )
22.                 ( ANW1.Take-Pictures(10 hallwayB) [10, 30] )
23.             ) [5, 50] )
24.         ) ;;end parallel
25.         ;;ANW1 leaves building
26.         (choose
27.             ( ANW1.Contact-Control-Center(50 OutsideA) )
28.             ((sequence
29.                 ( ANW1.(HallwayA) [20, 30] )
30.                 ( ANW1.Contact-Control-Center(20 OutsideA) [25, 40] )
31.             ) [20, 40] )
32.             ( ANW1.Contact-Control-Center(70 OutsideB) )
33.         ) ;;end choose
34.     ) ;;end sequence
35. ) ;;end Exit-Building
```

Figure 66: Control program for the Exit-Building strategy.

The Exit-Building program contains a number of activities that are constrained to a specific location. For example, the activity `Contact-Control-Center` has the location constraint OutsideA. This means that ANW1 must remain in the region OutsideA throughout the entire duration of the `Contact-Control-Center` activity.

The vehicle dynamics are important when executing a mission strategy. A vehicle, such as a helicopter, has a high dimension state space. Thus, to quickly explore a vehicles state space and find a collision-free path through the world an RRT-based path planner is used.

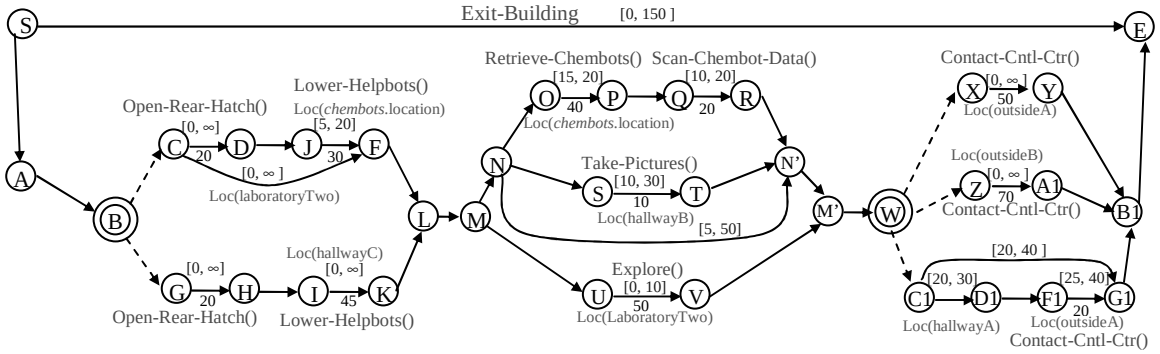


Figure 67: TPN representation of the Exit-Building strategy.

The main contributions of this chapter are threefold. The first is the description of an extended TPN model, called a Roadmap TPN (RMTPN), which grows RRTs from its events in order to satisfy location constraints. The second is a reformulation of the search space as a unified representation that consists of partial executions comprised of activities, temporal constraints, and RRT paths. The third is an extension to the TPNA* algorithm, developed in Chapter 5, that resolves location constraints by dynamically updating partial executions with paths.

6.2 Overview

UAPP searches an RMTPN model to find the optimal, complete and consistent execution. We describe the RMTPN model in terms of RRTs, but it can be generalized to any roadmap-based path planning model, such as the probabilistic road-map (PRM) planner [3] and the moving obstacles path planner [21]. We specify a number of assumptions in order to simplify the combined activity and path planning framework, and then provide suggestions for extending the UAPP algorithm.

6.2.1 Assumptions

We make the following three assumptions. The first is a control program for the mission may specify multiple threads of activities, such as communicating and taking pictures; however we impose the assumption that only one thread of activities constrains the location of the vehicle. This means that, given a set of parallel threads, all threads but one

have the location constraint “ANYWHERE”. The second assumption we make is that all regions specified in the domain of locations are disjoint, except “ANYWHRE”. The third assumption is if an activity requires the AV to be in region B, for example, and the AV is currently in region A, then if a path from region A to region B is found, we assume that, the robot arrives in region B by the beginning of the activity, and remains in region B for the duration of the activity. This might require a helicopter to hover, a rover to stop in one place, or an airplane to fly in a holding pattern.

RRTs are probabilistically complete and incrementally explore their space for a given number of iterations (see Chapter 2). UAPP fails to satisfy a location constraint if no collision-free path is found within a specified number of RRT iterations.

6.2.2 RMTPN Model

The Roadmap TPN (RMTPN) model is an extension to a temporal plan network that grows RRTs in order to satisfy location constraints. Recall that a location constraint is attached to a TPN arc and is used to either require an activity or set of activities to be performed in a specific spatial region, or to assert a waypoint that the AV (Autonomous Vehicle) must navigate to. The RMTPN model supports dynamic path updates by growing RRTs in order to achieve location constraints (Figure 68).

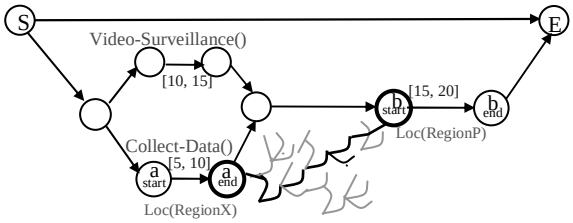


Figure 68: High-level example of a RMTPN growing an RRT from RegionX to RegionP.

6.2.3 UAPP Overview

By unifying activity planning with path planning, activities that are not constrained to a specific region may be executed in parallel, while the AV is navigating. This is a result of dynamically updating the RMPTPN when a location constraint is satisfied. For example, in Figure 69, `Activity2` is not constrained to a specific region, but must be executed after `Activity1` and before `Activity3`. In this case, it is

possible to navigate from RegionA to RegionW while performing `Activity2` in parallel. Likewise, Activity5 can occur in parallel to Activity3, since it poses no additional location constraint.

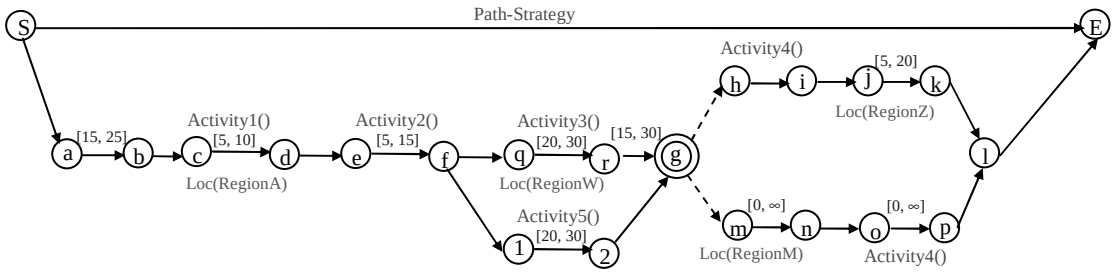


Figure 69: Example RMTPN representation of a control program that has a total ordering on its activities. We refer to this strategy as Path-Strategy

Recall that the motivation for the RMTPN is the introduction of location constraints on all TPN activities. If a location constraint is not explicitly specified, then the default ANYWHERE is assumed. Each location constraint is satisfied by finding a path for the AV that satisfies the constraint within the time bounds on the activity. This process is detailed in Section 6.4.

6.3 UAPP Search Space Formulation

This section develops the unified TPN and path planning space that UAPP explores. The search space has two layers. The first is a layer contains the activities in the initial TPN, while the second layer denotes where path planning will be done (Figure 70). These two layers are inherent to a RMTPN model.

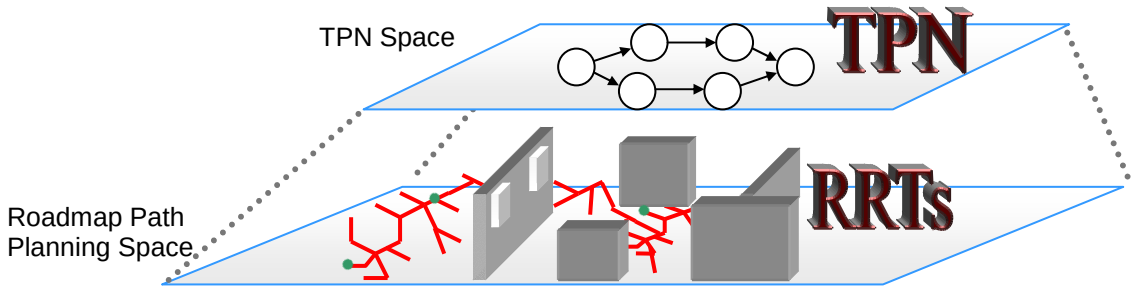


Figure 70: Unified search space - Illustrates the two spaces in which the unified planner searches. The circles points in the TPN represent regions where the AV will navigate to.

The input to the UAPP algorithm is a TPN representation of a control program and an environment model. The TPN is treated as an RMTPN, allowing it to grow RRTs and dynamically update its representation with paths. The path planner operates on the configuration space (or state space) model of the world. The output of UAPP is the optimal execution, moving through the TPN and path planning layers. The optimal execution must not violate any of its temporal constraints, and all of its location constraints must be satisfied. UAPP generalizes the TPNA* by adding in a procedure to satisfy location constraints.

6.3.1 Partial Execution

Recall, from Chapter 5, that the search space of an optimal pre-planning problem consists of the prefixes of all complete executions in a TPN. Likewise the RMTPN search space consists of the prefixes of all complete executions in an RMTPN. These prefixes are, again, referred to as partial executions. There are two elements of a RMTPN partial execution. First, the RMTPN execution contains a set of contiguous threads that originate at the start event S and have not been fully expanded to the end event E. Second, an RMTPN partial execution contains a single additional thread, which is a path. A feasible path connects all locations in the partial execution, within the temporal constraints, specified in the partial execution's corresponding STN. An example of a partial execution of the Path-Strategy (Figure 69) is given in Figure 71.

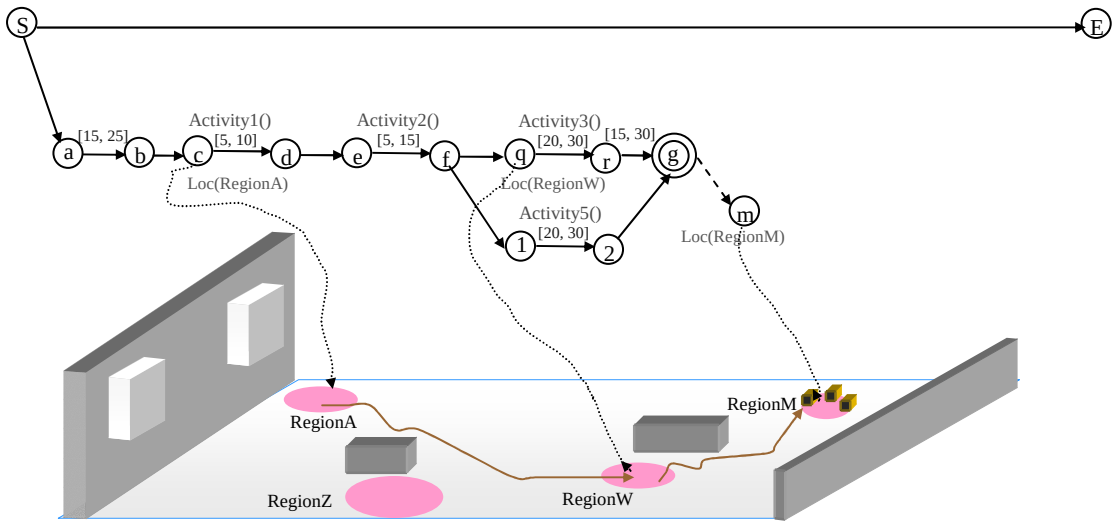


Figure 71: Example of a RMTPN partial execution with a path from RegionA to RegionW to RegionM.

A RMTPN partial execution is compactly encoded by its terminal events, choices, and path connecting regions. The formal mapping of a RMTPN partial execution to its encoding is defined by the tuple ⟨*fringe*, *choices*, *RRT-path*⟩ where:

- *fringe*: is a set containing terminal events in the partial execution (as described in Chapter 5).
- *choices*: is a set containing each choice in the partial execution (as described in Chapter 5).
- *RRT-path*: is a path between activity regions that does not violate the temporal constraints. The RRT-path can be traced from the current RRT node back to the initial AV state by following the parent RRT nodes on the path.

The encoding for the partial execution shown in Figure 71 is ⟨{E, m}, {g→m}, RRT-path(rrt-node(RegionM)) ⟩.

A complete execution of an RMTPN is a complete execution through the TPN layer, which satisfies all of the location constraints of the execution, and a continuous roadmap path, through the path planning layer, to each region specified by a sequence of location constraints.

6.3.2 Search Tree

Recall that a search tree is constructed by TPNA* search (Chapter 5). A search tree for an optimal pre-planning problem specified by a RMTPN is comprised, once

again, of nodes and branches. An example search tree for the Path-Strategy is shown in Figure 72. Initially, the root of the search tree denotes a partial execution encoded as ⟨S, {}, initial-AV-location⟩. A search tree node *n*, is encoded by the tuple ⟨*parent*(*n*), *TPNfringe*(*n*), *location*(*n*)⟩, where *parent*(*n*) and *TPNfringe*(*n*) are defined the same as in Chapter 5. For a given search tree node *n*, *location*(*n*) is the current RRT node in the path planning layer. For example, *location*(s_3) in Figure 72 is rrt-node(RegionZ). Finally, branches in the search tree denote a specific set of choices in the search space, as described in Chapter 5.

The RRT path containing all regions of a partial execution denoted by a search tree node *n* can be obtained by following the path from *location*(n) backwards to the initial state of the AV.

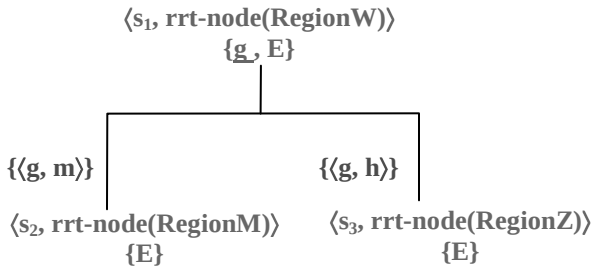


Figure 72: Search tree Path-Strategy in Figure 69.

Node Expansion and Goal Test

As described in Chapter 5, the TPN fringe events of a search tree node are decision points in the TPN layer, and an RRT node in the path planning layer. The decision points of a search tree node are used to generate new child nodes. For example, *TPNfringe*(s_1) is comprised of the two events g and E. Event g is a decision point, and thus, for each choice from g, a new child node is generated. In this case, there are two choices g→m and g→h, which are denoted as search tree nodes s_2 and s_3 in Figure 72.

When a search tree node *n* is expanded to its first location constraint, location(*n*), the search tree node looks-up *location*(parent(*n*)) and plans a path from the RRT node returned by that procedure to the location(*n*). For example, when the search tree node s_2, is expanded a location constraint, Loc(RegionM) is satisfied and the *location*(s_2) is updated to rrt-node(RegionM). This is illustrated in Figure 73 and Figure 74. The

additional TPN events and arcs are inserted when location constraints are satisfied (see Section 6.4).

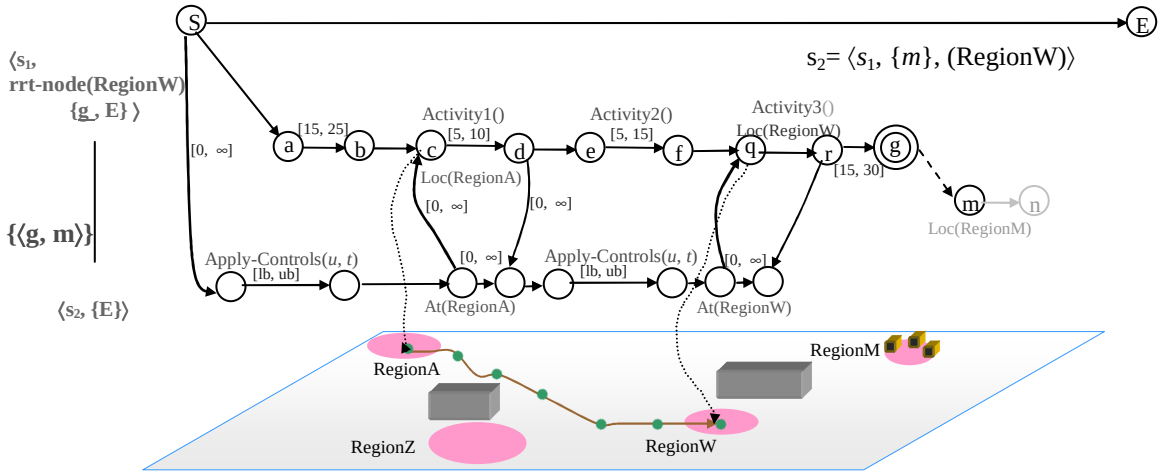


Figure 73: Example of an RMTPN partial execution for the search tree node s_2 (Figure 72) before satisfying location constraint Loc(RegionM) during expansion. The corresponding search tree is shown on the left, and the partial execution is shown on the top right.

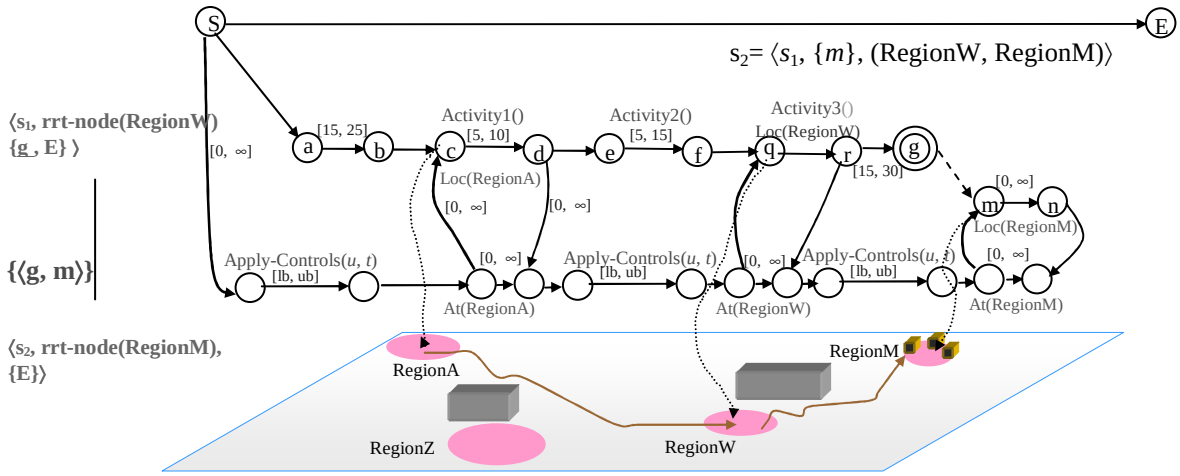


Figure 74: Example of partial execution for the search tree node s_2 (Figure 72) after satisfying location constraint Loc(RegionM) during expansion.

The UAPP node expansion procedure is detailed in Section 6.4. The procedure extends the expansion procedure described in Chapter 5 by growing RRTs in attempt to satisfy location constraints of a search tree node *n*. If the location constraint is satisfied, then the RRT node denoting the goal location, rrt-goal, is recorded with the search tree node, and location(*n*) = rrt-goal.

The UAPP expansion procedure terminates if a search tree node *s* is removed from the priority queue, Open, which satisfies the Goal-Test. The Goal-Test verifies that

the RMTPN partial execution denoted by *s* is complete. That is, the corresponding TPN partial execution is consistent and complete, and *s* contains a collision-free path through the path planning space, which connect all regions in the partial execution denoted by *s*.

We refer to the definitions outlined in Chapter 5 for *feasible execution*, and *optimal execution* to describe a feasible and optimal RMTPN execution. A feasible RMTPN execution is a partial execution through the TPN and path planning space that is consistent and complete and satisfies the Goal-Test. An optimal RMTPN execution is feasible execution that minimizes the evaluation function *f*. If no feasible execution exists, then either one of two cases has occurred. The first is that there exists no complete consistent execution in the corresponding TPN. The second is that, within the given number of iterations to grow an RRT, a location constraint was not satisfied, and thus the algorithm terminated.

6.4 Expansion and Satisfying Location Constraints

Recall from Chapter 5 that the TPNA* Expand procedure is comprised of two phases. The first phase, Phase One (Figure 47), is to extend threads from each event in the *TPNfringe*(*s*) of a search tree node *s*, until, along each thread, either a decision point is reached or the end event, E, is reached. If, while extending threads, two or more threads re-converge, then temporal consistency is tested. The second phase, Phase Two, involves generating new child search tree nodes, and is accomplished by the Branch procedure (Figure 50).

The UAPP expansion procedure extends the TPNA* expansion procedure by adding one more step to Phase One. This step grows the RRTs from the *location*(*n*), of a search tree node *n*, in an attempt to satisfy location constraints. If *location*(n) is empty then the RRT grows from the *location*(*parent*(*n*)). The procedure to satisfy location constraints is called Satisfy-Location-Constraint and is given in Figure 75. More specifically, as a thread of search tree node *s* is extended, if an activity with a location constraint is extended, then the Satisfy-Location-Constraint procedure is invoked. The procedure grows an RRT for a specified number of iterations from the *location*(*s*) to the region specified in the location constraint, referred to as the goal region (Lines 4). This is

illustrated in Figure 76, where the thread is extended to `Activity3`, which has the location constraint Loc(RegionW).

---

procedure Satisfy-Location-Constraint (Search Tree Node *s*, Path Planner *rrt_pp*
Location Constraint *loccon*)
returns true if location constraint is satisfied, otherwise false.

1. initialize RRT node *start*
2. *start* ← location(*s*)
3. create RRT node *goal* ← Make-RRT-Node(get-region(*loccon*))
4. set *roadmap-path* ← Grow-RRT(*start*, *goal*)
5. **if** pathfound(*roadmap-path*) **then**
6. initialize *end-at-assertion* ← end-event(most-recent-At-assertion(*s*))
7. create navigation *navactivity*
8. set-command(*navactivity*, Apply-Controls)
9. set-parameters(*navactivity*, control-inputs(*roadmap-path*))
10. [lb, ub] ← get-time-bounds(*roadmap-path*)
11. set-temporal-constaints(*navactivity*, [lb, ub])
12. insert-arc(*end-at-assertion*, *navactivity*, [0, 0])
13. create At-Assertion *at-region*
14. set-temporal-constraint(*at-region*, [0,+INF])
15. insert-arc(end-event(*navactivity*), start-event(*at-region*), [0, 0])
16. insert-arc(start-event(*at-region*), start-event(*loccon*), [0,+INF])
17. insert-arc(end-event(*loccon*), end-event(*at-region*), [0,+INF])
18. **if** Not(Temporally-Consistent(distance-graph(*s*)) **then**
19. **return** false
20. **else**
21. location(s) ← (rrt-goal(*roadmap-path*))
22. **return** true
23. **else**
24. **return** false

Figure 75: Procedure to attempt to satisfy location constraints, during Phase One.

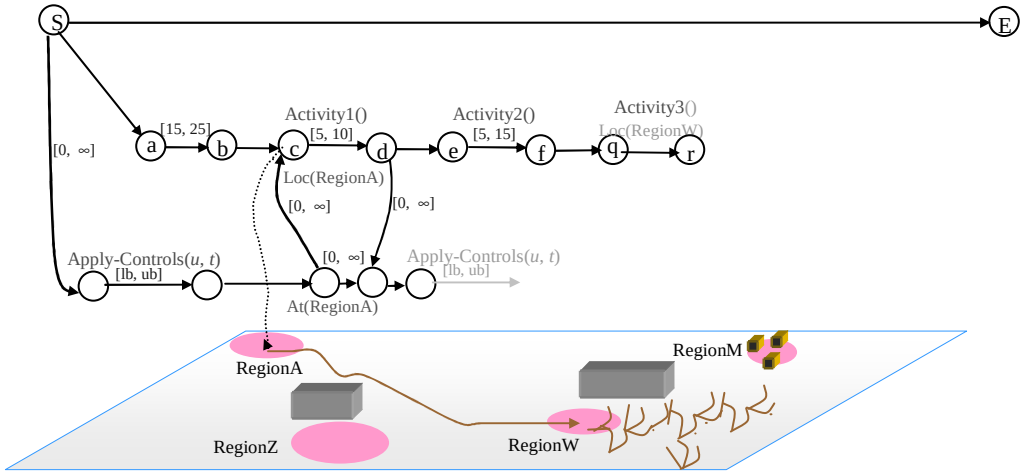


Figure 76: Attempting to satisfy the location constraint Loc(RegionM) by growing an RRT from the current region, RegionW.

Inserting TPN Arcs for Temporal Consistency

Planning a path from region to region adds more time to the RMTPN partial execution. Thus, when a path is found, we insert a set of arcs and a navigation activity in the corresponding TPN, with specific time bounds. Primarily, we want to test if the time needed to navigate along the path does not induce a temporal inconsistency.

A location constraint is satisfied if, and only if, the path planner returns a collision-free path from the AV's current position to the specified location within a specified number of iterations. In this case, a navigation activity, that abstracts the RRT path, is added to the partial execution (Lines 7-11). We use an "At" assertion which means the AV is At the location for the duration of the activity. For example, in Figure 77 the `Apply-Controls2`, shown on arc p5→p6, is inserted from the previous At assertion, At(RegionA). The temporal bound on the `Apply-Controls2` activity is the minimum and maximum time it takes to get from RegionA to RegionW.

In addition to the navigation activity, an At assertion is added that holds over the duration of the location constraint (Lines 13-17). This is a result of inserting two temporal constraints. The first constraint is on the arc from the start event of the At assertion to the start event of the location constraint. This arc is given a temporal bound of [0, +INF]. The second, is an arc from the end of the location constraint to the end of the At assertion, also with a [0,+INF] temporal bound. The addition of these arcs is illustrated in Figure 77 with arcs p7→q and arc r→p8. Once the navigation activity and the At assertion are inserted into the partial execution; threads in the partial execution converge. Thus, a test for temporal consistency is performed (Lines 18). The test verifies whether or not the path can be followed by the AV within the temporal constraints of the partial execution. The minimum and maximum time bounds of the path are computed based on the AV's minimum and maximum velocities.

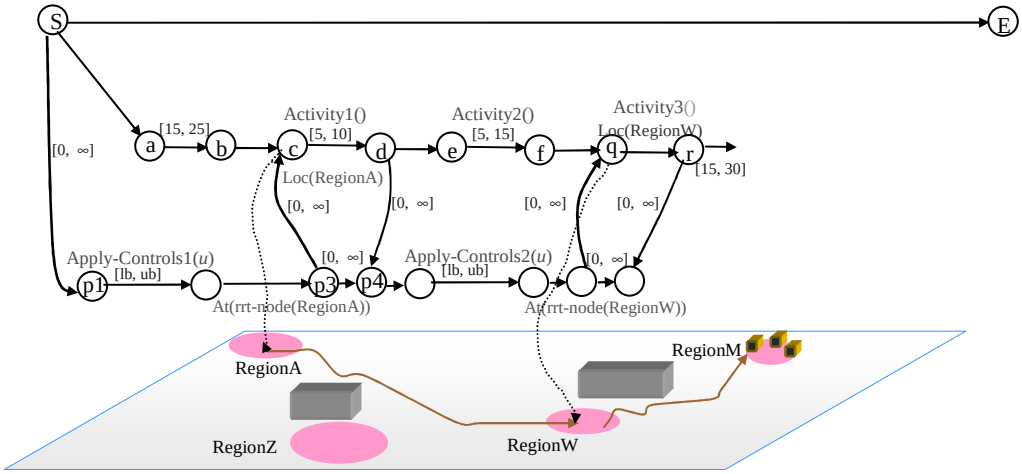


Figure 77: After the location constraint Loc(RegionW) is satisfied.

There are two cases in which the Satisfy-Location-Constraint procedure returns failure. The first case occurs when a collision-free path to the goal region is not found within the given number of RRT iterations. The second case occurs when a collision-free path is found, but the amount of time to navigate does not fit within the temporal bounds of the partial execution. In this case, the RRT continues exploring until the maximum number of iterations is reached. If the maximum number of iterations is reached and no path is found, or if the temporal bounds of the path cause a temporal inconsistency, then the RMTPN partial execution is inconsistent (Figure 77, Line 19 and Line 24).

If the Extend-Non-Decision-Events procedure succeeds, then the Branch procedure is invoked. Recall that the Branch procedure, as described in Chapter 5, is responsible for generating new child nodes from a parent *p*. Child nodes are generated for each possible set of choices between decision points in the fringe of *p*.

Converting an RMTPN Path to a Sequence of Activities

As stated, an RMPTN grows RRTs in order to achieve location constraints. A path generated by an RRT path planner can be converted into a sequence of activities, as shown in Figure 78. This is done by creating an activity, `Apply-Controls`, which is parameterized by the requisite control inputs to get from state to state in the RRT path. Once a complete execution is found, the navigation activities inserted during the Satisfy-Location-Constraint procedure can be removed and replaced with a sequence of `Apply-Controls` commands for each RRT node pair.

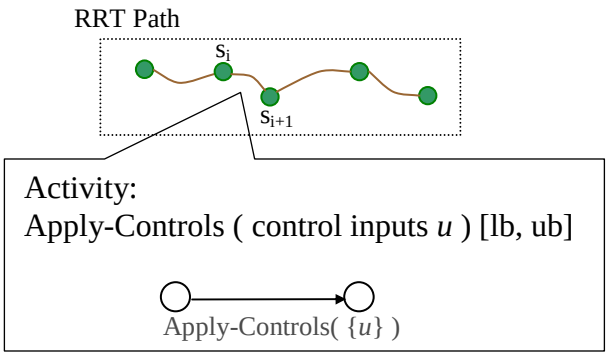


Figure 78: Example of mapping from a path sequence with nodes and edges to a sequence of activities. Each activity refers to the action of navigating to a location by applying the control inputs u for the duration of the activity.

An example of a complete RMTPN execution with the control actions is shown in Figure 79.

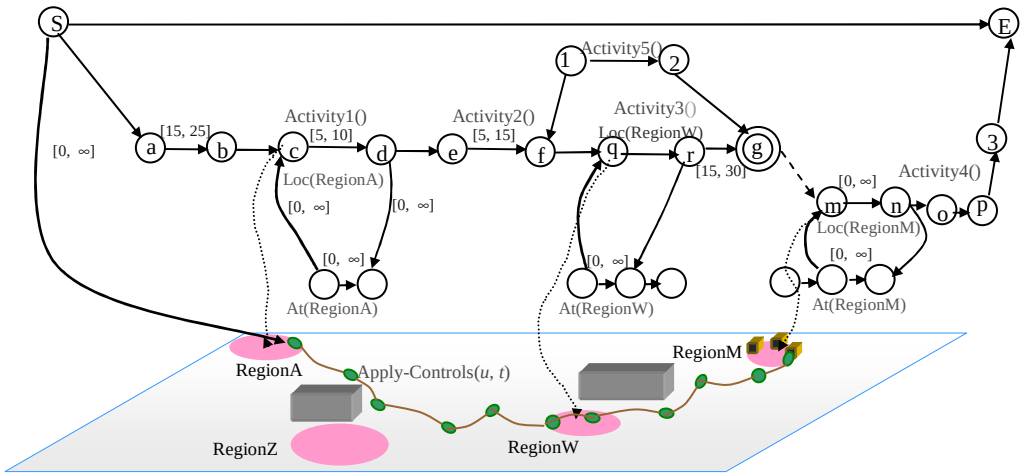


Figure 79: Example of a complete RMTPN execution.

6.5 Discussion

In summary, we developed a solution to satisfy location constraints for control programs based on the assumptions presented in Section 6.2.1. We use the RMTPN model and apply the UAPP algorithm in order to satisfy location constraints in a given partial

execution. Given this solution, we are able to unify activity planning and path planning by searching over their combined layers, in order to find the globally optimal plan. The UAPP algorithm can use the cost of the RRT paths, found while expanding a search tree node, with the cost of the activities of the search tree node in order to get a global cost. Moreover, given a roadmap based path planner with an optimal admissible heuristic, such as a visibility graph, the UAPP algorithm can use the heuristic in order to partially expand paths of a search tree node to get a better estimated cost of a solution through that node.

The process of satisfying location constraints can be generalized to control programs with multiple threads by employing the standard threat resolution techniques as described in [39][28]. These techniques are used to place an ordering on two intervals that assert two different locations. We describe this further in Chapter 7.

Chapter 7

Performance and Discussion

This chapter describes the implementation and experimental results of the optimal pre-planner and the unified optimal activity and path planning system (UAPP). We demonstrate both capabilities on a suite of randomly RMPL control program. Then give a discussion of the results and suggestions for future work. Finally, we conclude with a summary of the thesis contributions.

7.1 Implementation

Both the optimal pre-planner and unified activity and path planning (UAPP) system were implemented in C++ on a Windows based system using the gcc compiler. A random RMPL generator, created to empirically validate the system, was also implemented in C++.

7.2 Empirical Validation

We first analyze the performance of the optimal pre-planner on a set of random RMPL control programs. We use a random RMPL generator to create classes of problems that are parameterized by the number of decision points, the number of choices per decision point, the number of parallel sub-networks, and the solution depth. We discuss the performance of optimal pre-planning using the uniform cost and DP-Max heuristic to guide the search. Finally we conclude with preliminary results of the UAPP applied to 10 RMPL control programs with increasing number of location constraints.

7.2.1 Analysis of TPNA* Search with Feasible, Uniform-cost, and DP-Max

We begin our analysis of the optimal pre-planner with a general comparison of the three search strategies: feasible, uniform-cost, and DP-Max on a large state space. Recall that the feasible search strategy introduced in [39], implements a modified network search in order to find a feasible execution through the TPN. When the search reaches a decision point, it immediately makes an arbitrary choice, and proceeds forward through the TPN, until either a temporal inconsistency is detected, or a feasible solution is reached. If the feasible search detects a temporal inconsistency, it backtracks to its last decision point and selects a different choice.

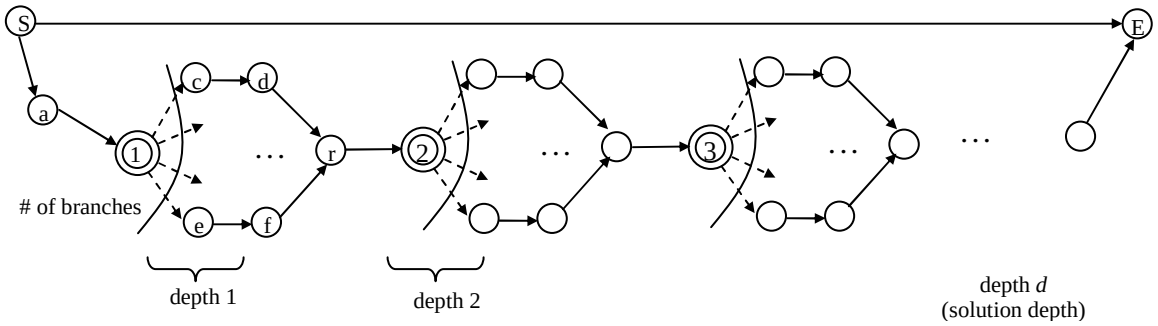


Figure 80: Example of type of problem instance analyzed.

We compare the trade-off of finding a feasible solution versus finding an optimal solution in a large state space. The of problems used to compare the two search strategies involved a sequence of choices, with varying depth, as shown in Figure 80. The number of branches per decision point was set to 2, and the solution depth ranged from 1-10. We allotted each search algorithm the same maximum amount of time to solve all 10 problems. Note that random problems of this type will not illustrate the effectiveness of the DP-Max heuristic. This is due to the fact that there is only one activity per thread in a decision sub-network. As a result the uniform cost and DP-max heuristic will perform comparable to each other.

Table 1: General analysis of the three search strategies.

Search strategy	Avg. Time to solve
Feasible	47,869 μs
Uniform-cost	54,972 μs
DP-Max	51,930 μs

In this experiment, the feasible search only slightly out-performed the uniform cost and DP-Max searches, as was expected. Table 1 shows the average computational time for each search strategy to solve 10 problems. The graphs in Figure 81 and Figure 82 illustrate the computational time to solve optimal pre-planning problems with a sequence of decision sub-networks. The graph implies that the search strategies are comparable in the amount of time to compute either a feasible solution or an optimal solution for problems with less than 6 decision points in a sequence. The feasible search strategy found a solution to the problem with a depth of 6 fairly quickly. We hypothesis that the feasible search "got lucky" with the set of arbitrary choices it made.

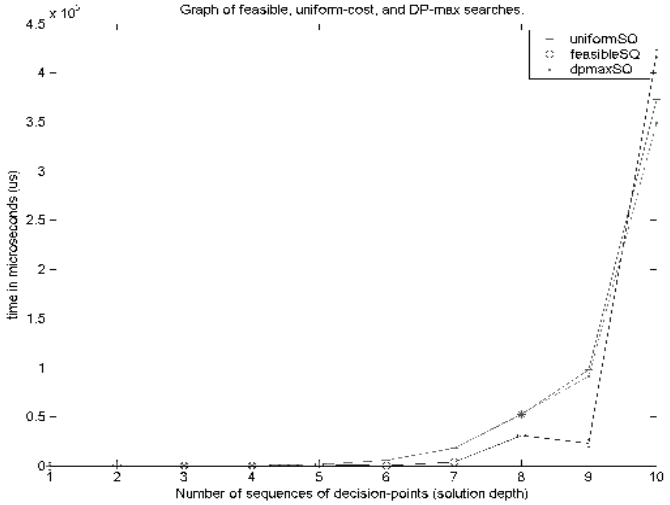


Figure 81: Plot of the computation time of the feasible, unified-cost, and DP-max search strategies on RMPL control programs with a sequence of choices and a solution depth from 1-10.

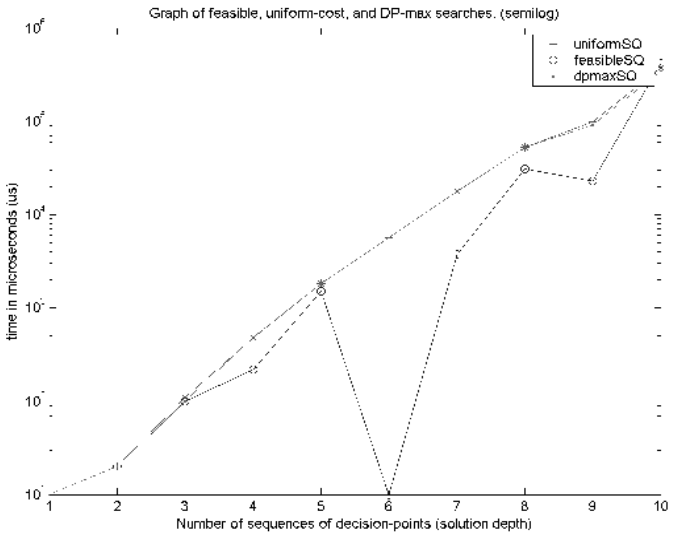


Figure 82: Semi-log plot of the graph in Figure 81.

We used this preliminary data to develop the experiments described in Section 7.2.2 and Section 7.2.3. Note that, to conclude which search strategy performs the best in practice, 10 random problem instances is insufficient. More analysis can be performed using a bank of random examples per solution depth, and by analyzing the average performance of each search strategy overall.

7.2.2 Analysis of TPNA* Search on Class A Problems

This section presents a performance analysis of the optimal pre-planner on problems without location constraints. We compare the results of optimal pre-planning with the DP-Max heuristic to pre-planning with a uniform-cost search, that is, optimal pre-planning with only activity costs. For the types of problems in this class, we expect the search with DP-Max to perform comparably to the search with uniform-cost search. In Section 7.2.3 we show a class of problems for with the DP-Max heuristic outperforms the uniform-cost search.

To analyze the performance of the optimal pre-planner on a large state space, a set of 100 problems, encoded as RMPL control programs, was generated. The problems were composed of a `sequence` of `choose` expressions with a varied number of choices and solution depths, as shown in Figure 80. To simulate a temporal inconsistency in a TPN, threads within each `choose` were randomly selected to be made either temporally

consistent or inconsistent. This is done by setting the temporal bounds of a thread(s) to be inconsistent. That is, the lower bound time is greater than the upper bound time.

We designed two trials, where each trial referred to the number of choices per decision point, which corresponds to the branching factor of each problem. A trial consisted of problems with b = 2, or 3 where b is the number of branches. We set the branching factor range based on the results in Section7.2.1. A problem instance was randomly created based on the following parameters:

- Depth of solution ranged from 2 to 6 decision sub-networks within a problem.
- The cost was randomly generated ranging from 10 to 100.
- The maximum number of temporally inconsistent branches within a decision sub-network ranged from 1 to b-1.

The results are shown in Table 1 and Table 2.

In each trial, we analyze the maximum number of search tree nodes in the priority queue, Open, for each problem instance. We also show the average number of enqueue operations per data set, where an enqueue operation is the process of inserting a search tree node into Open. Finally, the average time to solve the problem instances for each trial is given.

Table 2 highlights the performance of TPNA* applying a feasible, uniform-cost, and DP-Max search strategy on the randomly generated class of problems. We provide the range of values for the maximum number of search tree nodes in Open problem instance, and the range of enqueue operations per problem instance.

Table 2: Analysis of uniform-cost and DP-Max search on 100 problem instances with a branching factor of 2.

	Uniform	DP-Max
Avg. Time to solve	1,070.3 μs	1,090.7 μs
Avg. of the Maximum Number of Nodes in Open	17.1	17.1
Avg. Number of Enqueue Operations	44.4	44.4
Minimum(of the Maximum Number of Nodes in Open)	4	4
Maximum(of the Maximum Number of Nodes in Open)	49	49
Minimum(Number of Enqueue Operations)	8	8
Maximum(Number of Enqueue Operations	128	128

As expected, the DP-Max heuristic proved to be uninformative when solving this class of problems. Note that this class of problems, which is composed of a sequence of `choose` expressions, demonstrates the worst case for temporal consistency checking. That is, a partial execution consisted of solely one thread. Consequently, an inconsistent partial execution was not detected until the thread re-converged at the TPN end event E.

Table 3: Analysis of uniform-cost and DP-Max on 100 problem instances with a branching factor of 3.

	Uniform	DP-Max
Avg. Time to solve	12,500 μs	12,741 μs
Avg. of the Maximum Number of Nodes in Open	120	120
Avg. Number of Enqueue Operations	265	265
Minimum(of the Maximum Number of Nodes in Open)	9	9
Maximum(of the Maximum Number of Nodes in Open)	530	530
Minimum(Number of Enqueue Operations)	14	14
Maximum(Number of Enqueue Operations	1,105	1,105

In general, any of the two optimal search strategies would benefit greatly by extracting conflicts (inconsistent choices) from the search space. This can be done with a Conflict-directed search [41] [33]. We are currently characterizing the class of problems for which the DP-Max heuristic outperforms the uniform-cost search.

7.2.3 Analysis of UAPP

We present preliminary data of the UAPP algorithm. We focus on the time to compute a combined activity and path plan for 10 RMPL control programs with an increasing number of locations.

Table 4: Analysis of UAPP on ACLC RMPL control programs.

Time to Solve(μs)	Number of locations	Number of Sequence	Number of Choose	Number of parallel
831	3	1	1	0
370	4	2	1	0
7,300	4	3	1	0
610	7	3	2	0
5,337	8	3	2	1
270	9	3	2	2
3,465	12	3	3	2
9,463	13	4	3	2
14,971	14	5	3	2
7,470	15	5	3	3

To gain an insight on the amount of time to compute a combined activity and path solution in a simple world, we created an environment with dimensions 100'x100' and no obstacles. The robot dimensions were estimated with a sphere of radius 1'. The UAPP algorithm is dominated by the time to grow RRTs. This is to be expected since the search space of the optimal pre-planning problem has increased. A number of more analyses need to be performed in order to completely characterize the UAPP algorithm. For example, a comparative analysis of UAPP using various RRT parameters and UAPP using other roadmap based path planners would provide greater insight into the computational and memory costs of unifying activity planning and path planning.

7.3 Future Work

This section gives suggestions for future work. We focus on three main areas: 1) improving heuristic costs of search tree nodes, 2) solving location constraints on multiple threads within a control program, and 3) employing an optimal memory-bounded search strategy to improve space efficiency.

7.3.1 Improving Heuristic Costs for Search Tree Nodes

An issue for computing an accurate admissible cost is that search tree nodes may contain more than one TPN event in their fringe. Threads corresponding to each event may re-converge at a sub-goal before the end event E. In this case the sum of the heuristic costs of the TPN events in the fringe, computed by the dynamic programming principle, will double count the cost of their common sub-goal. Without combined knowledge of where the threads containing these TPN events converge, the heuristic cost of a search tree node can become inadmissible, while a max heuristic is less informative. This can be addressed, by performing a depth first search from each of the TPN fringe events, in order to obtain the point where the threads converge. This only requires at most an O(E) forward search, where E is the number of edges, each time the heuristic cost of a search tree node is computed.

7.3.2 Extending UAPP to Multi-Threaded Control Programs

In a single AV strategy, in order to plan paths from region to region, the UAPP must determine the order in which the regions are visited. If two location constraints are asserted over two intervals of time that overlap, then the two constraints pose a threat to each other, and are called conflicts. For example, threads in a parallel sub-network are executed concurrently. If, however, two threads of a parallel sub-network have activities that are to be executed in different regions, then these activities cannot be executed at the same time. For example, consider the parallel sub-network for the Exit-Building program (Figure 83). The parallel sub-network contains location constraints on each of its threads. Thread M→N→O→P→R→N'→M' has the location constraint Loc(*chembots*.location). Thread M→N→S→T→N'→M' has the location constraint Loc(HallwayB). Finally, the thread M→U→V→M' has the location constraint Loc(LaboratoryTwo). The UAPP algorithm must be able to determine the order in which each of regions specified by the location constraints, are visited.

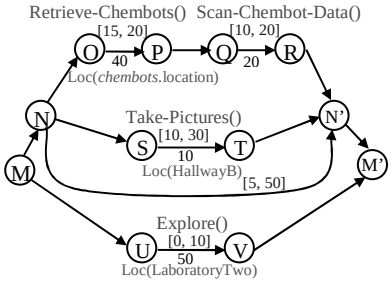


Figure 83: Snapshot of the parallel sub-network in the Exit-Building TPN (Figure 67). The location constraints within a parallel sub-network must have an order that specifies the order in which regions are visited.

To determine the order in which each region is visited and guarantee that two activities constrained to different locations are not asserted over the same time interval, we apply the standard threat resolution techniques described in [28][39]. The UAPP algorithm approaches threat resolution into two steps: 1) Conflict detection and 2) Resolution. Conflict detection requires determining the feasible time ranges over which the activities in the partial execution can be executed, and then test each interval to determine whether or not, more than one location constraint is asserted over an interval. If a conflict has been detected, then UAPP attempts to resolve the conflict by imposing an ordering forcing one conflict before the other.

Intervals of activities on one thread of a parallel sub-network may overlap with intervals of activities on another thread within the same sub-network. Location constraints within a parallel sub-network may also overlap, and thus, considered incompatible. This would require the AV to be in two regions at the same time, which is not possible. We refer to this incompatibility as a conflict. A conflict, as described in [39], occurs if two constraints within the same interval assert the negation of each other. For example, given a condition *C*, if *C* and *Not(C)* are asserted over the same interval then they conflict. A condition *C*, is analogous to a location constraint. For example, if two location constraints are asserted over the same time interval, then the two constraints conflict.

To detect location constraints that conflict within a given partial execution, we first determine when two intervals overlap. To detect when two intervals overlap we computing the feasible time ranges for each event in the partial execution, as described in [39]. A feasible time range of an event is given by the upper bound and the lower bound times for the event to occur. To compute the feasible time range of events in partial execution, we use the distance graph encoding of the partial execution. The upper bound time range for each event is given by the shortest path distance from the start event S to each event [39]. The lower bound time range for each event is given by the negative shortest path distance of an event to the start event S.

Once the feasible time ranges for each event in the partial execution have been computed, then overlapping intervals with location constraint conflicts can be detected. If a conflict is detected, then we use the threat resolution technique described in [39]. This technique involves applying the standard promotion/demotion technique [28] by inserting a TPN arc with a non-zero lower bound from the end of one conflict to the start of another. Inserting this arc imposes an ordering on which region, specified in the conflicting location constraints, will be visited first. Inserting this arc may result in a temporal inconsistency. Thus, if both promotion and demotion result in a temporal inconsistency, then the conflict is unrecoverable, and the partial execution is invalid. Otherwise, if imposing an ordering on conflicts succeeds then we can grow an RRT to in attempt to resolve the location constraints, as described in the previous section.

7.3.3 Improving Space Efficiency

Best-first search, operates similar to breadth-first search, in that space consumption grows exponentially in the depth of the search. A memory-bounded optimal plan search can compensate for the tendency of A*-based algorithms to use excessive space. Thus, a search strategy such as TPNA* search may not be suitable for certain AVs. This is of particular importance for mobile robots, given their limited computing resources. Initial versions of this type of search strategy were presented in [7] and [29]. These memory-bounded searches are guaranteed to find the optimal plan that can be stored within their allocated memory. They do this by partially expanding search nodes and pruning high-cost nodes from the explicit memory, while backing up heuristic estimates along partial paths in the search tree. The most recent version of these algorithms, Zhou and Hansen's SMAG*-propagate and SMAG*-reopen [43] improve speed by accounting for heuristic inconsistencies that occur during the search. A memory-bounded search strategy can adapt the SMAG* optimal search algorithm and apply it to find the optimal execution in an RMTPN within a specified amount of memory.

7.4 Summary

There are three main research contributions presented in this thesis. The first is a language, the ACLC (activity cost and location constraint) subset of RMPL, used to specify mission strategies for autonomous vehicles. The mission strategies are specified in a control program. The language supports activities with costs and location constraints. Location constraints are specified alongside an activity. They constrain an autonomous vehicle (AV) to a specific spatial region throughout the duration of the activity. To resolve location constraints, an environment model is described. The environment model contains a description of the autonomous vehicle used in the mission and the world in which it will navigate. In addition to location constraints, the ACLC subset of RMPL allows the mission designer to specify the estimated cost of executing an activity. Given the ACLC specification for mission strategies, a control program is mapped in to a compact graphical representation, called a temporal plan network (TPN). TPNs were

introduced in [40][39], but are extended in this thesis to encode activity costs and location constraints.

The second contribution is an optimal pre-planner (Chapter 5). The optimal pre-planner operates on a given TPN encoding of a mission strategy. We define partial execution of the TPN as search states in the search space. The primary algorithm driving the optimal pre-planner is TPNA* search. TPNA* constructs a search tree in order to explore the search space for an optimal feasible execution. In addition, we provided an optimal heuristic for TPN events, called DP-Max, which can be used to efficiently guide TPNA* search.

The third contribution is the unified optimal activity and path planning (UAPP) system. This system supports optimal pre-planning of activities and location constraints. UAPP operates on a road-map TPN (RMTPN) and explores the combined TPN search space and path planning space. Location constraints are resolved by determining a temporally feasible ordering on the locations within a partial execution. Then an RRT based path planner grows a tree in attempt to entail each constraint (Chapter 6).

In conclusion, unifying model-based mobile programming with roadmap-based path planning can provide enhanced autonomy for robots. Robots equipped with robust automated reasoning capabilities and sensing technology can be used to aid emergency personnel in rescue missions, potentially saving more lives.

Bibliography

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

[2] J. Bellingham, A. Richards, and J. How. "Receding Horizon Control of Autonomous Aerial Vehicles" I*EEE American Control Conference*, May 2002.

[3] R. Bohlin and L. E. Kavraki. "Path Planning Using Lazy PRM." *Proceedings of the International Conference on Robotics and Automation*, 2000.

[4] Bonet and H. Geffner. "Planning as heuristic search." *Artificial Intelligence: Special Issue on Heuristic Search,* 129:5-33. 2001

[5] M. Blackburn, H. R. Everett, and R.T. Laird. "After Action Report to the Joint Program Office: Center for the Robotic Assisted Search and Rescue (CRASAR) Related Efforts at the World Trade Center". San Diego, CA, August 2002.

[6] J. Bruce and M. Veoloso. "Real-time randomized path planning for robot navigation." *Proceedings of IROS-2002*, Switzerland, October 2002.

[7] P. Chakrabarti, S. Ghose, *et al*. Heuristic search in restricted memory. AIJ, 41, 197-221, 1989.

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

[9] R. Dechter, I. Meiri, J. Pearl. "Temporal constraint networks." *Artificial Intelligence*, 49:61-95, May 1991.

[10] R. James Firby. An investigation into reactive planning in complex domains. In *InProc. of the 6th National Conf. on AI, Seattle, WA, July 1987.*.

[11] E. Frazzoli. "Robust hybrid control for autonomous vehicle motion planning." Dissertation, MIT, June 2001.

[12] E. Gat. "Esl: A language for supporting robust plan execution in embedded autonomous agents." *AAAI Fall Symposium: Issues in Plan Execution*, Cambridge, MA,1996.

[13] D. Hsu, J-C. Latombe, and R. Motwani. "Path planning in expansive configuration spaces." In IEEE *International Conference on Robotics and Automation*, Albuquerque, NM, 1997.

[14] J. Hoffman and B. Nebel. "The FF planning system: Fast plan generation through heuristic search." *Journal of Artificial Intelligence Research,* 14:253-302, 2001.

[15] M. Ingham. "Timed Model-based Programming: Executable Specifications for Robust Mission-Critical Sequences." Dissertation, MIT, June 2003.

[16] H. Kaindl and A. Khorsand. "Memory-bounded bidirectional search." *Twelth International Joint Conference on Artificial Intelligence (AAAI-94),* pages 1359--1364, Menlo Park, California, 1994.

[17] P. Kim. “Model-based Planning for Coordinated Air Vehicle Missions.” M. Eng. Thesis, Massachusetts Institute of Technology, July 27, 2000.

[18] P. Kim, B. C. Williams, and M. Abramson. “Executing Reactive, Model-based Programs through Graph-based Temporal Planning.” *Proceedings of the International Joint Conference on Artificial Intelligence*, Seattle, Washington, 2001.

[19] R. Kindel, D. Hsu, Latombe, and S. Rock, “Randomized Kinodynamic Motion Planning Amidst Moving Obstacles” in *IEEE Conference on Robotics and Automation*, 2000.

[20] H. Kitano, S. Tadokoro *et. al.* “RoboCup Rescue: Search and Rescue in Large-Scale Disasters as a Domain for Autonomous Agents Research.” In Proc. Of the IEEE Conference on Systems, Men, and Cybernetics. 1999.

[21] J.-C. Latombe. Robot Motion Planning. Kluwer Academic Publishers. Norwell, MA, 1991.

[22] S. M. LaValle, and J. J. Kuffner. “Randomized Kinodynamic Planning.” *Proc. IEEE International Conference on Robotics and Automation.* 1999.

[23] S. M. LaValle. “Rapidly –Exploring Random Trees: A New Tool for Path Planning.”.

[24] R. R. Murphy, J. Casper, and M. Micire. “Potential Tasks and Research Issues for Mobile Robots in RoboCup Rescue.” RoboCup Workshop 2000.

[25] N. Muscettola, P. Morris, and I. Tsmardions. “Reformulating temporal plans for efficient execution.” NASA Ames. 1998.

[26] N. J. Nilsson. Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York. 1971.

[27] B. Orfringer. “Robot responders at WTC site fit into tight spaces.” *Disaster Relief*. October 17, 2001.

[28] I. Pohl. “First results on the effect of error in heuristic search.” Machine Intelligence 6, pp 127-140. Edinburgh University Press, Edinburgh, Scottland. 1971.

[29] S. Russell. “Efficient memory-bounded search methods.” *In Proceedings Tenth European Conference on Artificial Intelligence*, pages1—5, Chichester, England, 1992.

[30] S. Russel, and P. Norvig. Artificial Intelligence: A Modern Approach. Prentice Hall, 1995.

[31] T. Schouwenaars, B. Mettler, E. Feron and J. How. “Robust Motion Planning Using a Maneuver Automaton with Built-in Uncertainties.” *Proceeding of the IEEE American Control Conference*, June 2003.

[32] J. Selingo. “Pilotless Helicopter takes a whirl as an investigation tool.” *The New York Times*, September 2001.

[33] I. Shu. "Enabling Fast Flexible Planning through Incremental Temporal Reasoning." M. Eng. Thesis, Massachusetts Institute of Technology, September, 2003.

[34] R. Simmons. A task description language for robot control. In Proceedings of the Conference on Intelligent Robots and Systems (IROS), Victoria Canada, 1998, 1998.

[35] A. Stentz. “Optimal and Efficient Path Planning for Partially-Known Environments.” *IEEE International Conference on Robtics and Automation*, May 1994.

[36] B. P. Trivedi. Autonomy urban search and rescue. *National Geographic Today*, September 2001.

[37] B. C. Williams, M. Ingham *et. al.* Model-based Programming of Intelligent Embedded Systems and Robotic Space Explorers. *Proceedings of IEEE: Special Issue on Modeling and Design of Embedded Software*. January 2003.

[38] B. C. Williams, M. Ingham *et. al.* Model-based Programming of Fault-Aware Systems. *AI Magazine*. 2004.

[39] B. C. Williams, P. Kim *et. al.* Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration. 2001.

[40] B. C. Williams, P. Kim, M. Hofbaur *et al.* Model-based Reactive Programming of Cooperative Vehicles for Mars Exploration. *Int. Symp. on Artificial Intelligence, Robotics and Automation in Space*, St-Hubert, Canada, June 2001.

[41] B. C. Williams, and R. Ragno. “Conflict-directed A* and its Role in Model-based Embedded Systems." Special Issue on Theory and Applications of Satisfiability Testing, in *Journal of Discrete Applied Math*. January 2003.

[42] M. Yim, D. G. Duff, and K. Roufas. “Modular Reconfigurable Robots, An Approach To Urban Search and Rescue.” 1st Intl. Workshop on Human-friendly Welfare Robotics Systems. Taejon, Korea, Jan. 2000

[43] R. Zhou and E. A. Hansen. “Memory-Bounded A* Graph Search.” *15th International FLAIRS Conference*. Pensecola, Florida, May 2002.