16.412J Cognitive Robotics
Course Project

Tactic-Guided Cooperative Path Planner

Thomas Léauté

12 May 2004

In cooperation with
Seung H. Chung
Robert T. Effinger
Steven D. Lovell

Table of Contents

1. Introduction

This is the final report for the Spring 2004 16.412J course project. I chose to focus my project on the design and implementation of a Tactic-Guided Cooperative Path Planner, using Linear Programming. This personal project was part of a larger group project involving the design of a fairly complex activity and path planning architecture for cooperative autonomous agents.

In this report, I will first quickly present the overall planning architecture of this group project, how my path planner folds into this design and how it interacts with the other components of the system. I will then give an overview of the functionalities of the path planner, explain to what extent its design builds upon previous work in the planning research field, and describe in more details the innovations of the approach I took to tackle the problem.

One part of this report will also be dedicated to explaining the software interface I have been working on in order to demonstrate the path planner and the whole architecture on a flight simulation testbed. I will also present some results and runs we made on this simulator to demonstrate the capabilities of our activity and path planning architecture.

Finally, the last section will gather the imperfections and weaknesses of the current design that will have been introduced in the previous sections and suggest improvements to the path planner.

2. Overall Activity and Path Planner Architecture

2.1. Motivation and Previous Work

The concept of autonomous cooperative vehicles have recently found applications in many various fields, from managing a team of Unmanned Aerial Vehicles for fire-fighting or military missions, to coordinating a group of rovers to explore unknown environments or perform search-and-rescue scenarios. In all these applications, one of the most challenging issues is to be able to coach and monitor a group of autonomous agents organized in an architecture with which the human only has to interact at the highest level, delegating the low-level tasks to the autonomous systems.

In order to tackle this issue, this project builds upon several previous projects that tried to deal with the problem of coaching groups of cooperative autonomous vehicles. In particular, the overall architecture of the activity and path planner described in this report

was inspired by the work done under the MICA initiative, which will be briefly presented here.

The *Mixed Initiative Control of Automa-Teams* (MICA)[1] is a DARPA project conducted at the AI Lab and the Space Systems Lab a few years ago. This project dealt with the coordination of UAVs and ground vehicles for military missions, using an approach based on the tight integration of a temporal activity planner (Kirk) and a path planning architecture based on Mixed Integer Linear Programming (MILP)[2]. Kirk was used at the mission level to describe the list of goals to be achieved by the team of vehicles (mainly, waypoints to be reached) and the temporal constraints on the execution of these tasks. The lower-level MILP trajectory planner was responsible for executing the temporal activity plan generated by Kirk and also provided real-time re-planning and task re-assignment capabilities, using a receding-horizon framework [3].

More recently, the Software Enabled Control (SEC) group at MIT [4] designed a path planning architecture for multi-vehicle systems, involving a MILP receding-horizon framework. This architecture described in [5] inspired in many ways the architecture of our activity / path planner, and it is worth describing in some level of detail before introducing the actual planning architecture we came up with for this term project.

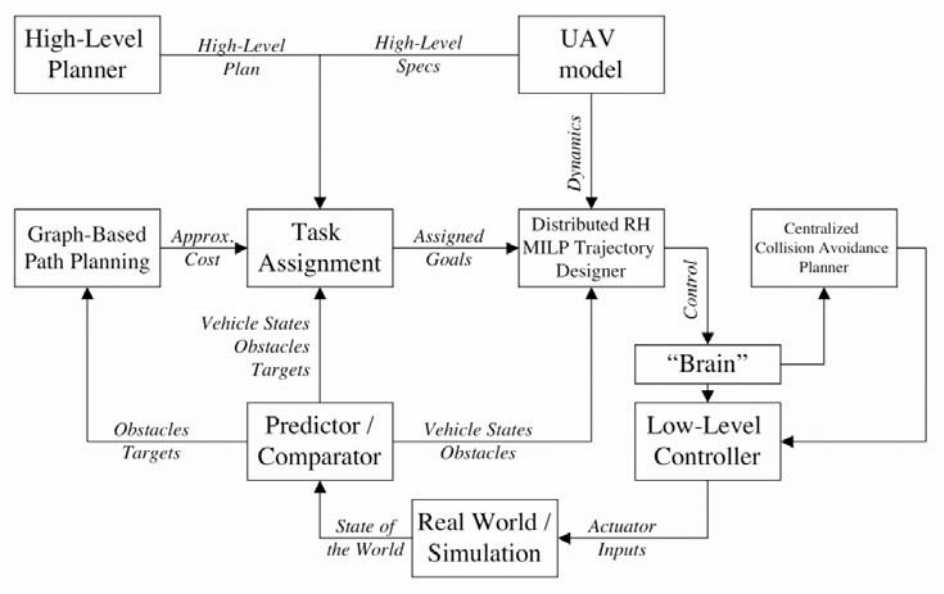


Figure 1: Software Enabled Control Planning Architecture [5]

Figure 1 illustrates the architecture of the Software Enabled Control planner. A High-Level Planner takes mission goals from the user (possibly through a natural language interface). Typically, for missions involving a team of UAVs supervised by a manned leader aircraft, these goals would be "go to this waypoint" commands issued by the pilot onboard the leader aircraft. Task assignment is done by a component that formulates the resulting multi-agent traveling salesman problem as a Mixed Integer Linear Program. To compute an optimal task assignment with respect to some cost

function (like fuel consumption, or some measure of safety) and/or some temporal constraints on the completion of the mission, the Task Assignment component requires estimates of the costs for each vehicle to go to each waypoint. These costs are provided by a Graph-Based Path Planner using a visibility graph approach to estimate the costs to go, avoiding the obstacles. A distributed MILP Trajectory Designer computes for each vehicle the path to go to its assigned goal, using a receding horizon framework which will be introduced in more details in Section 3.4. Finally, a centralized "Brain" checks for vehicle collision and calls a Collision Avoidance Planner to modify the path plans if necessary.

2.2. Overall Architecture and Innovations

The overall architecture of the activity and path planner was clearly inspired by the SEC framework described in 2.1. However, some major differences and/or innovations exist between the two approaches, as illustrated in Figure 2.

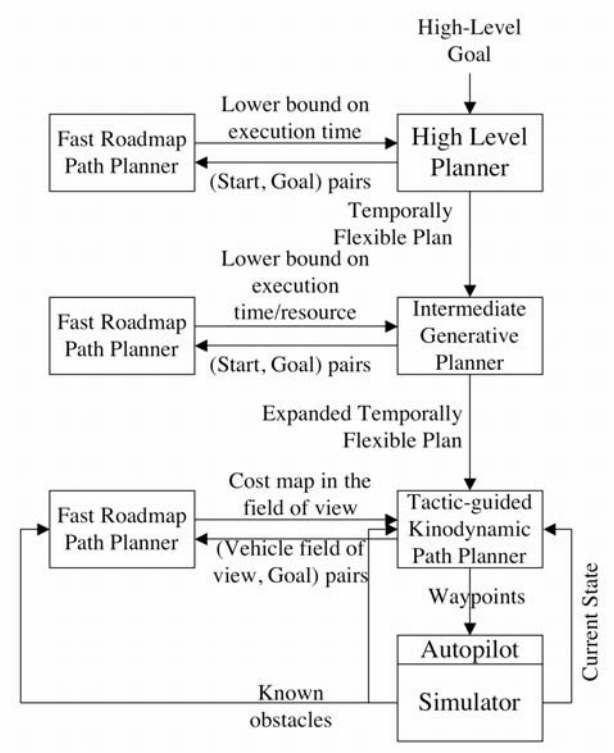


Figure 2: Overall Architecture of the Activity / Path Planner

As in the SEC framework, the user interacts with a High-Level Activity Planner (Kirk) sitting at the top of the architecture. However, while the SEC High-Level Planner allowed the user to specify goal waypoints to be reached within a given period of time, the use of Kirk as an even higher-level planner is a significant innovation. In this architecture, the user can specify very high-level mission tactics in order to accomplish a

set of abstract goals while satisfying mission temporal constraints. The goals are no longer commands to go to given waypoints; they are requirements on the final state of the system, such as having extinguished the specified forest fires. The intermediate Generative Activity Planner is then responsible for elaborating a lower-level activity plan in order to accomplish the high-level plan outputted by Kirk.

Both activity planners interact with a Fast Roadmap Path Planner in order to assure feasibility of the activity plan. The Roadmap Path Planner uses the D* Lite algorithm on a discretized map in order to compute the minimum distance to go from a given start waypoint to a given goal waypoint. This minimum distance is used by the activity planners to compute a lower bound on the execution time of activities involving motion. Note that in this architecture the task assignment is made at the highest level by Kirk. This will be discussed briefly in Section 5.

The use of this intermediate Generative Activity Planner allows the user to supervise the system at an even higher-level. Consider the goal sent by Kirk to a fire-fighting UAV to extinguish a specified forest fire. The intermediate planner can now relieve the user from wondering about resource management: it can decide that the commanded UAV must first go to an intermediate fuel station in order to refuel or go to a lake to get water before it can accomplish the goal specified by the user.

Finally, at the lowest level, a Tactic-Guided Kinodynamic Path Planner similar by some aspects to the SEC Trajectory Designer is responsible for designing the paths with respect to the temporal plan provided by the activity planners. It uses a receding horizon framework to plan within a limited time window and calls the Fast Roadmap Path Planner to plan beyond its horizon. This is presented in more details in Section 3.

3. Description of the Path Planner

3.1. Main Features and Functionalities

The main role of the Tactic-guided Kinodynamic Path Planner is to further expand the temporal activity plan provided by the higher-level planners by designing paths to lead the autonomous vehicles through the corresponding consecutive waypoints in the map. To do so, the path planner could simply pre-compute the entire paths beforehand according to its knowledge of the map. However, this knowledge might not be complete (it might encounter unknown obstacles along the way) or the environment might be dynamic and non-deterministic (pop-up obstacles). For those reasons, and also because the system must be able to monitor the execution of the plan and re-plan if necessary, an important feature of the path planner is that it runs in real time, designing paths as the vehicles evolve in their environment.

To design paths that take into account the dynamics of the vehicles, the path planner formulates the problem as a linear program. A short introduction to linear programming and how it has been applied to path planning are presented in Sections 3.2 and 3.3. But solving such linear programs can be very computationally intensive, so in

order the have the path planner design paths in real time, it must be restricted to designing partial paths within a limited time window. This approach is called Receding Horizon Path Planning and is presented in Section 3.3.4.

However, the role of the Tactic-guided Path Planner goes beyond simply designing paths towards provided waypoints. Because it is "tactic-guided", it is also responsible for enforcing the "tactic", i.e. the temporal activity plan it takes as an input. This plan is a temporally flexible plan, with lower and upper times bounds on the duration of activities so that re-planning is made possible at the lower level without changing the activity plan elaborated at a higher level. So the path planner also has to schedule the activities in the input plan by fixing the start and end times of each activity, as described in Section 3.5.

3.2. Brief Introduction to MILP and Clausal LP

A linear program is a constraint satisfaction and optimization problem in which constraints are linear inequalities on the variables and the optimization function is a linear combination of the variables. Variables are constrained to take real values. An optimal solution to the linear program is an assignment to all the variables that satisfies all the constraints while minimizing (or maximizing) the value of the optimization function. A linear program can be represented as illustrated in Figure 3.

$$\begin{aligned} &\textit{Minimize} \quad c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n \\ &\textit{Subject to} \quad \begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n \le b_1 \\ \qquad\qquad \ldots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n \le b_m \end{cases} \end{aligned}$$

Figure 3: Usual Representation of a Linear Program

Many optimization problems under constraints can be formalized as linear programs. However, the formulation described above can only describe *conjunctions* of constraints. Some problems require the notion of disjunctions of constraints, such as the obstacle avoidance problem: at any time instant, the vehicle must be either north of the obstacle *or* east *or* south *or* west. One approach to solve this issue is to extend the variable domains to discrete domains, mainly integer intervals. The resulting linear program is called a Mixed Integer Linear Program (MILP). How MILPs can be used to encode disjunctions of constraints as described in [5] for the path planning problem. But this approach is rather awkward and tricky, and can become very cumbersome for problems involving large numbers of nested disjunctions of constraints.

Another much more natural approach that has been demonstrated in this project and previously introduced in [6] is the Clausal Linear Programming approach. A Clausal Linear Program (CLP) is an extended linear program in which the set of constraints is no

longer a conjunction of inequalities but can be any logical combination of inequalities. The definition of a CLP in its Conjunctive Normal Form is illustrated in Figure 4.

$$\begin{aligned} &\textit{Minimize } c_{11}x_1 + c_{12}x_2 + \ldots + c_{1n}x_n \\ &\textit{Subject to} \begin{cases} \bigwedge_{1 \le i \le k_1} \left[a^i_{11}x_1 + a^i_{12}x_2 + \ldots + a^i_{1n}x_n \le b^i_1 \right] \\ \ldots \\ \bigwedge_{1 \le i \le k_m} \left[a^i_{m1}x_1 + a^i_{m2}x_2 + \ldots + a^i_{mn}x_n \le b^i_m \right] \end{cases} \end{aligned}$$

Figure 4: Definition of a CLP in Conjunctive Normal Form

No reliable solver currently exists for clausal linear programs. However, ILOG CPLEX [7] provides supports for some evolved logical constraints such as disjunctive or if-then constraints. The user can define such logical constraints and the solver then translates them into a MILP to apply the usual solving methods available. In the rest of this report, linear program encodings will be presented as CLPs.

3.3. Encoding of the Vehicle Dynamics as a Linear Program

The linear programming framework allows us to consider state variables taking real values. However, in order to express the constraints relative to the evolution of these state variables over time, we need to discretized time and consider the values of the state variables at consecutive time steps. From a linear programming perspective, there has to be one unknown for each vehicle, state variable, and time step. The resulting list of unknowns is presented in Table 1.

Table 1: List of Unknowns to Encode the State of the System

x[i][t]	x-coordinate of aircraft i at time step t
y[i][t]	y-coordinate of aircraft i at time step t
vx[i][t]	x-component of the velocity of aircraft i at time step t
vy[i][t]	y-component of the velocity of aircraft i at time step t
ax[i][t]	x-component of the acceleration of aircraft i at time step t
ay[i][t]	y-component of the acceleration of aircraft i at time step t

3.3.1. Initialization of the Unknowns

The state variables must be initialized to some value in order to track the continuous evolution of the state of the system from one planning step to the following re-planning step. To do so, the initialization constraints presented in Figure 5 are added to the problem. The choice of t_i^{ini}, x_i^{ini}, y_i^{ini}, vx_i^{ini}, vy_i^{ini}, ax_i^{ini} and ay_i^{ini} will be discussed in 3.3.4.

$$
\begin{cases}
x[i][t_i^{init}] = x_i^{ini} \\
y[i][t_i^{init}] = y_i^{ini} \\
vx[i][t_i^{init}] = vx_i^{ini} \\
vy[i][t_i^{init}] = vy_i^{ini} \\
ax[i][t_i^{init}] = ax_i^{ini} \\
ay[i][t_i^{init}] = ay_i^{ini}
\end{cases}
$$

Figure 5: Initialization Constraints for Aircraft i

3.3.2. Dynamic Model for the Aircrafts

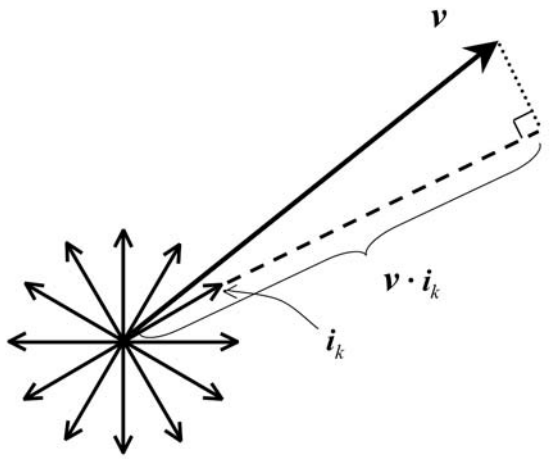
The dynamic model for the aircrafts is the zero-order linear point mass model presented in [5] and reproduced in Figure 6. Δt is the time increment between two time steps.

$$
\begin{cases}
x[i][t+1] = x[i][t] + \Delta t \cdot vx[i][t] + \Delta t^2 / 2 \cdot ax[i][t] \\
y[i][t+1] = y[i][t] + \Delta t \cdot vy[i][t] + \Delta t^2 / 2 \cdot ay[i][t] \\
vx[i][t+1] = vx[i][t] + \Delta t \cdot ax[i][t] \\
vy[i][t+1] = vy[i][t] + \Delta t \cdot ay[i][t]
\end{cases}
$$

Figure 6: Dynamic Model Constraints for Aircraft i at Time Step t

3.3.3. Upper-Bound Constraint on the Velocity

At each time step, the velocity of the aircrafts must be lower than some maximal velocity v_{max}. The issue with expressing this constraint is that it has to be linearized; one method introduced in [5] is presented in Figure 7.



$$\bigwedge_{0 \le k \le K-1} \left[vx[i][t] \cdot \cos(2k\pi/K) + vy[i][t] \cdot \sin(2k\pi/K) \le v_{max} \right]$$

Figure 7: Part-wise Linearization of the Upper-Bound Velocity Constraint for Aircraft *i* at Time Step *t* [5]

3.3.4. Lower-Bound Constraint on the Velocity

At each time step, the velocity of the aircrafts also has to be greater than some minimal velocity v_{min}. The CLP encoding for this constraint is very similar to the upper-bound velocity constraint, although it is now a disjunction of inequalities, as presented in Figure 8.

$$\bigvee_{0 \ge k \ge K-1} \left[vx[i][t] \cdot \cos(2k\pi/K) + vy[i][t] \cdot \sin(2k\pi/K) \ge v_{min} \right]$$

Figure 8: Part-wise Linearization of the Lower-Bound Velocity Constraint for Aircraft *i* at Time Step *t*.

3.3.5. Upper-Bound Constraint on the Acceleration

Similarly, at each time step the acceleration of the aircrafts also must be smaller than some maximal acceleration a_{max}. The CLP encoding for this constraint is the same as for the velocity and is presented in Figure 9.

$$\bigwedge_{0 \le k \le K-1} \left[ax[i][t] \cdot \cos(2k\pi/K) + ay[i][t] \cdot \sin(2k\pi/K) \le a_{max} \right]$$

Figure 9: Part-wise Linearization of the Upper-Bound Acceleration Constraint for Aircraft *i* at Time Step *t*.

3.3.6. Obstacle Avoidance Constraints

An aircraft is outside of an obstacle or a no-fly zone if and only if it is either to the north, the east, the south or the west of this obstacle. If we only allow rectangular obstacles o represented by the 4-tuple $(y_{North}[o], x_{East}[o], y_{South}[o], x_{West}[o])$ of their edge coordinates, then the obstacle avoidance constraints can be expressed as in Figure 10.

$$\begin{cases} y[i][t] \leq y_{North}[o] \\ x[i][t] \geq x_{East}[o] \\ y[i][t] \geq y_{South}[o] \\ x[i][t] \leq x_{West}[o] \end{cases}$$

Figure 10: Obstacle Avoidance Constraint for Aircraft *i* and Obstacle *o* at Time Step *t*

Note that since the aircrafts are modeled by a point mass, the obstacles have to be slightly extended in order to take into account the actual size of the aircrafts, plus a probable safety factor. This is described in numerous details in [5].

As for the constraints imposing to the aircrafts to stay within a given region of the map, those can be encoded in a rather straight-forward way as conjunctions of inequalities, with one inequality for each edge of the region, assuming that it is concave.

Collision avoidance between two aircrafts has unfortunately not been implemented.

3.4. The Receding Horizon Framework

As mentioned in 3.1, the CLP path planner only designs paths within a limited time window in order to lower the computation time necessary to solve the linear program. If the final goal towards which the path planner is designing a path is outside of that scope, the CLP planner requires a heuristic to guide its path in the correct direction. This heuristic is provided by a fast path planner as a set of sub-goals lying within the planning horizon and to which costs have been assigned. This set of sub-goals is called a *local cost map*. To design cost-optimal paths, the Kinodynamic Path Planner must choose a path to one of the sub-goals in the cost map while minimizing the total cost of the path to the final goal [3]. This is illustrated in Figure 11.

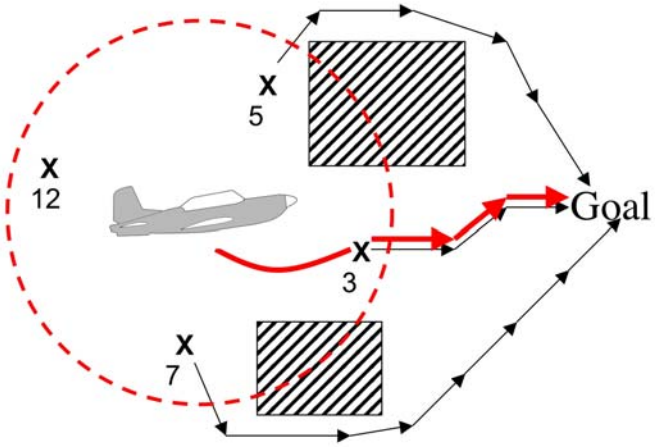


Figure 11: Receding Horizon Framework

Note that this method is very likely to lead the aircraft to a crash into an obstacle if the path planner re-plans once the aircraft has reached the planning horizon, because the final velocity of the aircraft as it reaches the planning horizon might not allow it to avoid an obstacle that would not have been detected before because it was beyond the planning horizon. To prevent this, we can introduce a second shorter-term horizon called the *execution horizon*: the path planner plans up to the planning horizon, but re-plans as soon as the aircraft reaches the execution horizon [3]. This method proved to be very effective during the test runs on the simulator.

Another approach to the same problem is the make sure that every path designed by the path planner ends in a safe loop [8]. This has not been implemented, although it is actually what the aircraft is driven to do in the current version of the path planner if CPLEX fails to find a solution to the CLP before the aircraft reaches the executing horizon: at each re-planning step, the aircraft is sent a linked list of waypoints, with the last waypoint pointing to the previous one, so that an aircraft which would have reached its last waypoint would start going back-and-forth between its two last waypoints. However, this closed path is not guaranteed to be obstacle-free.

The constraint formulation presented in Figure 12 is used in order to add the choice of a sub-goal to the CLP. Note that in the current version of the path planner, the cost to go to the sub-goal is not taken into account in order to speed-up the algorithm. The width ε of the tolerance region around sub-goals also helps speeding up the algorithm by loosening the constraints on the position.

$$Minimize \quad c[i]$$

$$Subject\ to \quad \bigvee_{sub\text{-}goals\ g} \begin{vmatrix} x_g - \varepsilon < x[i][t_{horizon}] < x_g + \varepsilon \\ y_g - \varepsilon < y[i][t_{horizon}] < y_g + \varepsilon \\ c[i] = c_g \end{vmatrix}$$

Figure 12: Constraint on the Choice of the Sub-goal for Aircraft *i*

Despite many other approximations similar to the ε sub-goal tolerance and not mentioned in this report, the algorithm still runs in more than 10 seconds. As a consequence, the path planner cannot wait until the aircraft reaches the execution horizon to start designing the next partial path; it has to start solving the CLP as soon as possible, i.e. as soon as it has finished solving the previous one. This means that the path planner must design the following path starting at an estimated future position of the aircraft. In practice, it starts the new path at the last waypoint of the current path (see Section 3.3.1). If CPLEX finishes solving the CLP "early", then the path planner waits until the aircraft has started tracking its last waypoint before it sends the new linked list of waypoints and the TRACK command (see Section 4.3). If CPLEX has not found the optimal solution in time, the path planner uses the best found solution as the new path. The algorithm fails if no feasible path is found in time.

3.5. Activity Scheduling and Execution Monitoring

The Tactic-guided Path Planner has to choose the start and end times of all the activities in the temporally flexible plan outputted by the activity planners. Section 3.4 describes a method that enables the path planner to solve a small part of the whole planning problem by focusing only on planning within a limited time window. The exact same method cannot be applied to the activity scheduling problem: if the planner ignores the activities that do not overlap the current time window, it will ignore lower time bounds on the duration of future activities and very likely leave too little time to perform them.

This is the reason why the planner must consider the whole scheduling problem every time it re-plans. However, as will be shown in the following paragraphs, the scheduling problem involves few variables and is not very computationally intensive, which makes it somewhat easy to re-solve.

A temporally flexible activity plan consists in nodes, arcs between the nodes, and activities associated with the arcs. Each arc describes a temporal constraint on the times at which its two corresponding nodes can occur. Each activity might impose constraints on the spatial position of the agent responsible for executing it. This is described in the following sections.

3.5.1. Associating Variables to the Nodes

The scheduling problem consists in assigning a time value to each node in the temporal plan. Hence, a new unknown $t[n]$ is created for each node n.

3.5.2. Encoding the Temporal Constraints on the Arcs

Each arc between two nodes is associated with a time interval constraining the time delay that may occur between the two nodes. Hence, if we consider an arc between two nodes n_1 and n_2 associated with the time interval $[t_{min}, t_{max}]$, the corresponding temporal constraint will be added to the CLP in the following form:

$$t_{\min} - t[n_2] - t[n_1] - t_{\max}$$

3.5.3. Encoding the Spatial Constraints on the Activities

We can think of two different spatial constraints an activity could impose on an agent: it could either impose the agent to be at a specified point at the end of the activity ("Go To" activity) or impose that the agent stays within a specified region during the execution of the activity ("Stay At" activity). The current version of the path planner supports "Go To" activities, but does not support "Stay At" activities.

Every time the path planner needs to re-plans, it first monitors the state of each activity in the temporal plan. Five cases can occur for a given activity:

- If the end node is older than the beginning of the previous time window, then the activity was completed long before and the path planner ignores it.
- Else, if the time value for the end node is within the previous time window, then the activity was just completed at the previous iteration. The path planner then freezes the time variable for this end node to its current value and removes any spatial constraint that might be associated with this activity.
- Else, if the start node is older than the beginning of the current time window, this means that the activity is being executed. The Tactic-Guided Path Planner then requests a cost map from the Roadmap Path Planner and adds the corresponding constraints (see Section 3.4) to the CLP.
- Else, the activity has not started yet, but it might still start within the current time window, depending on the status of the activities preceding it in the temporal plan:
    - If one of the immediately preceding activities is being executed and is about to be completed during the current time window, then the path planner adds its spatial constraints to the CLP (same as for the previous bullet point). Note that this method looks only at the immediate preceding activities, so that it is only valid if all activities are assumed to last longer than the width of the time window.

- Else, the activity is not about to start, so the path planner ignores its spatial constraints.

At the end of each re-planning phase, the path planner also checks if the aircrafts reach the goal position corresponding to their "Go To" activity by the end of its new path plan. If an aircraft is not within a given small distance of its goal position, then the completion of the "Go to" activity is postponed by adding a constraint on its end node stating that it cannot occur before then of the next time window. Else, the path planner considers that the aircraft will have reached its goal by the end of the current time window, so it adds a constraint on the end node of the "Go To" activity so that the activity has to terminate within the next time window.

4. Interfacing with the Cloud Cap Flight Simulator

In order to develop, debug and demonstrate our planning architecture, we had to choose a multi-vehicle testbed and design a software interface with that testbed. In the interest of time, we decided to choose a simulation testbed rather than a hardware architecture, because hardware integration would probably have taken more time than we had in the context of our course project.

The Software Enable Control group [4] at the Space Systems Laboratory was using a flight simulator provided by Boeing, and we first decided to use this simulator. We met with Mario Valenti, the PhD student responsible for maintaining the simulator and developing a software interface to it. However, it turned out that the requirements on the use of this simulator were rather stringent; one of the strongest requirements was probably the runtime constraint imposing our real-time planner to run in less than one second.

We finally decided to choose a simulation platform provided by Cloud Cap Technologies. This testbed had already been used in the context of the MICA project (see 2.1). The following sections briefly describe this hardware-in-the-loop simulation testbed and the software interface we had to build in order to interact with it.

4.1. Architecture of the Simulator

The Cloud Cap simulator is a hardware-in-the-loop testbed. It consists in a set of Piccolo autopilot units, connected to a PC via USB ports. The PC runs the actual software simulator, whose goal is to take the commands sent by the autopilots and simulate the corresponding behavior of the aircrafts according to a given kinodynamic model. The simulator then computes the sensor information corresponding to the current state of the aircrafts, and sends this data back to the autopilots.

In order to send commands to the simulated aircrafts, the user must interact with an Operator Interface. This software is preferably installed on another PC, because Cloud

Cap recommends that the simulator be the only application running on its PC in order to assure real-time computation. The PC running the Operator interface is connected to a Ground Station that relays the data from the user to the autopilots through radio waves. The user can send commands, such as waypoints or autopilot mode switches. The Operator Interface also displays status data sent by the autopilots every second, such as GPS position, velocity or index of the waypoint currently being tracked. The overall simulator architecture is presented in Figure 13.

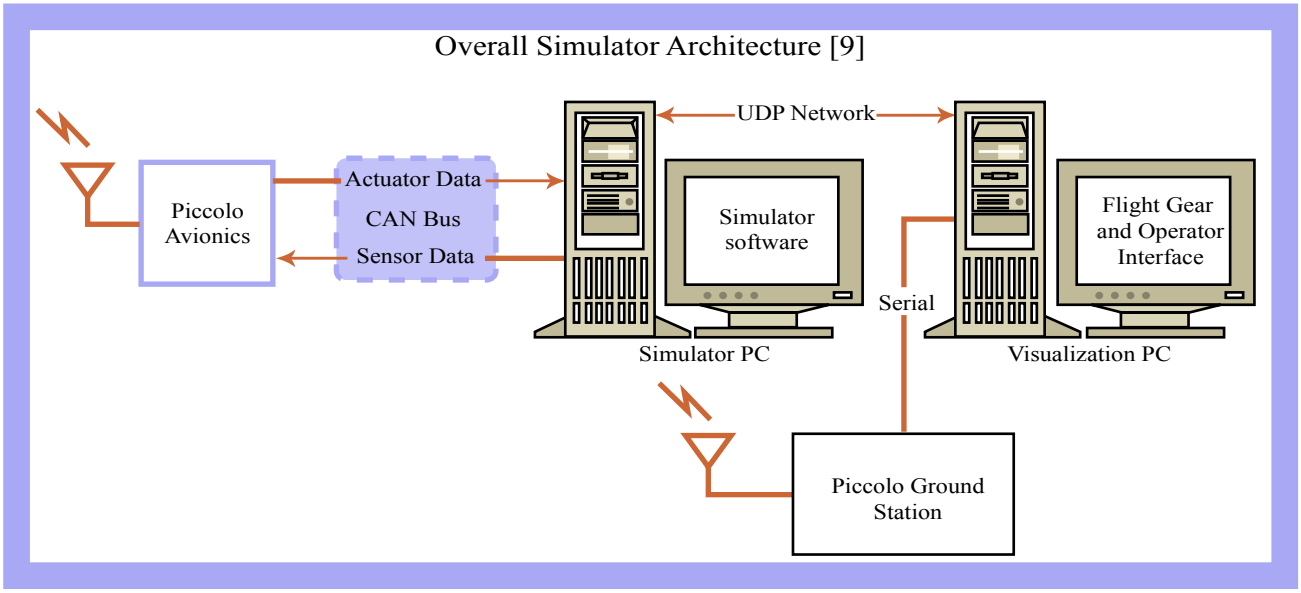


Figure 13: Overall Simulator Architecture [9]

The user can also take control of the simulated aircrafts over the autopilots by using a manual pilot console connected to the Ground Station. Finally, as presented in Figure 13, the simulator offers the possibility to use an external visualization software such as Flight Gear or Microsoft Flight Simulator to see the aircrafts evolve in a virtual environment.

4.2. Pros and Cons of the Cloud Cap Simulator

First of all, the choice of an aircraft simulator is much more challenging than demonstrating on a rover testbed: while rovers can stay still at a given location, UAVs have a minimum speed requirement that creates real-time issues and imposes computation time specifications on the path planner. If we ignore temporal constraints on the fulfillment of the mission, a rover path planner has a theoretically infinite amount of time to design a path as long as the rover is able to stay at a safe location. On the contrary, a flying vehicle is constantly evolving in its environment and the path planner must assure collision avoidance at any time instant while planning future moves towards a specified goal waypoint.

The principle advantage of the Cloud Cap simulator is that it is a hardware-in-the-loop testbed. This framework offers the opportunity to demonstrate the capabilities of a

path planner on a very realistic flight simulation, only one step away from a real-world demonstration with actual UAVs. Contrary to the Boeing simulator used by the SEC group, this testbed is fairly customizable in that the user has direct access to the aircraft dynamics model and can modify it or replace it completely. This allows scenarios involving heterogeneous teams of UAVs with various capabilities and specifications. The external visualization tool (Flight Gear) also enables the user to use various types of aircraft models and provides a very nice tool to monitor the dynamic behavior of the different UAVs, which is not possible on a simple map display.

However, the possibility to choose different aircraft models also makes it difficult to calibrate the autopilots. In particular, we had to spend a fair amount of time using a trial-and-error method to calibrate the command gains. With erroneous gains, the autopilots are unable to control the aircrafts properly; they can even often completely loose control and remain unable to go back to a stable state, even with the help of the manual pilot console.

Finally, it is also important to mention that, while the current version of the Operator Interface allows the user the load customized maps of the vehicle environment, it does not support the display of multiple UAVs on the same map: each aircraft is associated with its map display window, and the user cannot visualize the relative positions of multiple UAVs in the same window.

4.3. Description of the Software Interface

A software interface is necessary for our planning architecture to interact with the simulator. This interaction is made possible with the Operator Interface by starting it in "server mode". In this mode, the Operator Interface can be connected through TCP/IP to an external client and can relay to the client all the data it receives from the autopilots. The client can also send commands to the Operator Interface, which relays them to the autopilots.

Cloud Cap Technologies provides a Software Development Kit consisting of C++ libraries in order to make the interaction with the Operator Interface easier. Unfortunately, those libraries are only available for Windows, and since neither Kirk nor CPLEX were available for this platform, we had to re-write them from scratch for UNIX-based platforms. The resulting set of classes and methods are far from being complete, but they were designed to be as comprehensible as possible so that they could easily be extended to give more features to the interface with the simulator and more capabilities to the planning architecture. How to extend these capabilities is also fairly comprehensibly described in the Cloud Cap communications manual [10].

The software interface currently supports the following capabilities:

- The interface can listen to the TCP streams relayed by the Operator Interface and identify their sender and their type. The type of streams mostly used by the planner architecture is the AUTOPILOT type relative to the aircraft status and autopilot mode.

- For any incoming *AUTOPILOT* stream, the interface can also identify its sub-type. In particular, more capabilities are provided for the following sub-types:
  - TELEMETRY streams: These incoming streams contain telemetry data about the autopilot that sent it. The interface can currently extract from those streams the following information: latitude, longitude, altitude (both GPS altitude and barometric altitude), groundspeed, and time at which these measurements were made.
  - AUTOPILOT_COMMAND streams: These streams sent by the autopilots contain in particular the reference to waypoint that the corresponding aircraft is currently tracking.
  - WAYPOINT_LIST streams: Such streams can be sent to the autopilots in order to add or remove waypoints from their flight plans.
  - WAYPOINT streams: These streams are used to upload waypoint descriptions to be added to a given autopilot's flight plans.
  - TRACK streams: used to command an aircraft to start tracking a given waypoint stored in one of its flight plans.

This interface is sufficient to command the autopilots by sending them waypoints and track commands. However, it currently does not give access to some important telemetry and sensor information, such as acceleration.

5. Results and Future Work

The Tactic-guided Path Planner was demonstrated on a simple single-aircraft fire-fighting test run that is commented in this section in order to outline the weaknesses of the current version of the path planner and suggest future work to improve its performances. A short movie is available for this demonstration. For this test run, the Tactic-guided Path Planner what given the following temporal activity plan consisting of two parallel branches starting and ending at the same time. The intervals before the description of activity correspond to the time constraints on the duration of this activity.

First branch: (this first thread describes the whole mission and imposes minimum and maximum time bounds on its duration)
- [30, 1800] Put out the two fires

Second branch:
- [30, +INF] Go to the Eastern "water" waypoint of the Southern lake
- [30, 30] Get water
- [30, +INF] Go to southern fire
- [30, 30] Drop water
- [30, +INF] Goto the Southern "water" waypoint of the Northern lake
- [30, 30] Get water
- [30, +INF] Go to Northern fuel station
- [30, 30] Get fuel

- [30, +INF] Go to Northern fire
- [30, 30] Drop water
- [30, +INF] Go back to base

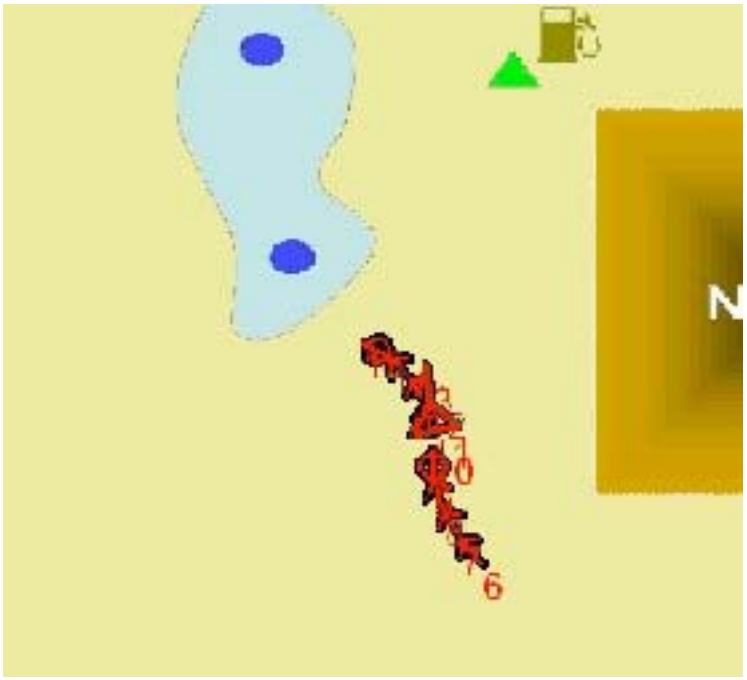


Figure 14: Zigzag Phenomenon as the Aircraft Gets Closer To its Destination

Figure 14 illustrates one of the weaknesses of the path planner: as it gets closer to its destination, an aircraft tends to start zigzagging. This is due to the fact that the path planner uses a discrete cost map as a heuristic to guide its path in the direction of the destination when this destination is outside of its planning horizon. This test was run with a 200-meter resolution for the cost map, so as the aircraft gets closer to the destination, the optimal sub-goal in the cost map is often slightly off-course. This might be solved by increasing the resolution; however, the greater the number of sub-goals to take into account in the CLP, the longer CPLEX needs to run to solve the CLP. Some tests were run with lower resolutions, and those tests showed that CPLEX tends too take too much time to solve the LP when the resolution of the cost map is lower than 200 meters.

Figure 15: Late Obstacle Avoidance Problem

Figure 15 also illustrates the problem with late obstacle avoidance. On its way back to the base, the aircrafts heads straight to the destination and turns to go around the no-fly zone ate the last minute. This is due to the fact that the Tactic-Guided Path Planner and the Roadmap Path Planner were never integrated; for this test run, the heuristic used to guide the path design is straight-line distance to the destination.

6. References

[1] Website of the MICA project
[2] Arthur Richard's webpage on Model Predictive Control of vehicles:

[3] MICA's webpage on Receding Horizon Control of Autonomous Aerial Vehicles

[4] The webpage of the Software Enabled Control Group

[5] Yoshi Kuwata, *Real-time Trajectory Design for Unmanned Aerial Vehicles using Receding Horizon Control*, Masters Thesis at MIT, June 2003
[6] Raj Krishnan, *Solving Hybrid Decision-Control Problems Through Conflict-Directed Branch & Bound*, PhD Thesis at MIT, February 2004.
[7] ILOG Concert Technology Manual
[8] Tom Schouwenaars, Éric Féron, Jonathan How, *Safe Receding Horizon Path Planning for Autonomous Vehicles*, 40th Allerton Conference on Communication, Control, and Computation October 2002
[9] Cloud Cap Quick Setup manual
[10] Cloud Cap Communications Manual