Computational Science and Engineering (Int. Master's Program)

Technische Universität München

Master's Thesis

Parallel Best-First Heuristic Search applied to Cooperative Planning for Automated Vehicles

Author:	Agamirzov Evgeny
1st examiner:	Prof. Dr.-Ing. Matthias Althoff
2nd examiner:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Assistant advisor(s):	Dipl. Inform. Daniel Hess
Thesis handed in on:	June 31, 2016

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

June 29, 2016 Agamirzov Evgeny

Acknowledgments

First of all, I wish to thank Prof. Dr.-Ing. Matthias Althoff for providing me a great opportunity to do my thesis in Deutsches Zentrum für Luft- und Raumfahrt (DLR), Institut für Verkehrssystemtechnik.

Also, I sincerely thank DLR's staff and especially Daniel Hess for their guidance and encouragement in carrying out this project work.

Finally, I would like to give my special thanks to all my family and friends who have been supporting me throughout this eight month.

Abstract

This paper studies the utilization of multi-core processors for path planning algorithms. A* best-first heuristic search algorithm is used for path finding, where the state space is a search tree that is built from discrete trajectory paths (motion primitives). Particularity of the given search problem is computationally expensive expansion step and large branching factor of the tree. Most recent parallelization techniques has been evaluated to identify which one fits best for the existing planning problem (considering significant differences between A* parallelization approaches). Finally, two schemes were chosen to be tested in real life scenarios - PA* and PRA* (HDA*). These algorithms represent two basic strategies of the best-first search parallelization that are shared OPEN list (PA*) and private OPEN lists (PRA*, HDA*). Both approaches has been tested for execution time and memory usage, and eventually compared head to head. Experiments were conducted for different road scenarios with various complexity. The main goal of this work was to show that multi-core machines can be successfully applied to path finding (planning) problems for automated vehicles. The thesis was funded and supported by Deutsches Zentrum für Luft- und Raumfahrt (DLR), Institut für Verkehrssystemtechnik. All implemented algorithms were embedded into DLR's simulation environment where real car models and road scenarios are being tested.

Contents

Outline of the Thesis

Part I: Introduction and Theory

CHAPTER 1: INTRODUCTION
Overview of the thesis and it purpose.

CHAPTER 2: THEORY
Theoretical background of path planning problems and algorithms.

Part II: Vehicle and Environment Models

CHAPTER 3: VEHICLE MODEL AND MOTION PRIMITIVES
Mathematical model of the vehicle and trajectory generation.

CHAPTER 4: STATE SPACE
State space description for single and multi-agent planners.

Part III: Path Planners

CHAPTER 5: SINGLE AGENT CONTINGENCY PLANNER
Emergency breaking path planning algorithm.

CHAPTER 6: MULTI-AGENT CONTINGENCY PLANNER
Cooperative emergency path planner where more than one vehicle is involved.

Part IV: Best-First Search Parallelization Applied to Path Planning

CHAPTER 7: STRATEGY CHOOSING
Reasoning the parallelization algorithms architecture.

CHAPTER 8: PARALLEL A*
Standard parallelization approach applied to multi-agent contingency planner.

CHAPTER 9: HASH-DISTRIBUTED PARALLEL A*
Parallel A* with hash function based domain distribution.

Part V: Results and Conclusions

Part I.

Introduction and Theory

1. Introduction

Nowadays automated driving systems are being developed by many institutions and automotive industry companies around the world. These systems are intended to optimize traffic flow on future roads and make driving experience less stressful for an every day user. Ideally, a vehicle has to navigate in the surrounding environment based on sensor data without any human input [11]. Such capabilities usually require a set of various sensors, a data processing module and a control system. Raw sensor data along with the current vehicle parameters is used to construct an environmental model and make a control decision which has to consider all probable road scenarios and be as safe as possible for the driver and the surrounding environment [23]. In case of autonomous vehicles decision making means computing safe driving path in finite period of time. One of the most popular path planning techniques is the utilization of deterministic heuristic-based search algorithms (Dijkstra, A*, etc.) [9]. In this case path planning is defined as a search problem where parameters depend on the route criteria. Path planning problem in the continuous space is reduced to a search problem in the discrete space (e.g. graph) by applying state space sampling.

Deutsches Zentrum für Luft- und Raumfahrt (DLR), Institut für Verkehrssystemtechnik is developing it's own automated driving system that includes a simulation environment, a set of path planners for distinct road conditions and two prototype vehicles that carry a number of sensors, controllers and on-board computers. The system is currently under development and thus many modules like path planners are advancing and being actively tested. The project is created in collaboration with the UnCoVerCPS consortium that provides cutting edge technologies in cyber-physical systems. This work was funded by DLR to continue the development of path planning algorithms in the direction of parallel algorithms application and also a research of multi-agent planners.

This paper focuses on defining a computationally expensive path planning problem for a set of agents and then evaluate how modern parallelization schemes for heuristic search algorithms can be applied to this particular problem [25]. It starts with describing a discretized search domain, proceeds to the environment description, continues with heuristic search specifics and finally studies the effectiveness of multi-core machines utilization.

This work is based on already existing single agent path planning algorithm developed at DLR for a real prototype of the robot vehicle [6]. This algorithm is intended to compute a contingency maneuver path (emergency breaking), that means it is only initiated when an emergency road scenario occurs. It is operating in real time (finds solutions below 0.1 seconds margin) and at the same time provides a safety guarantee. Generally speaking, the domain of the single agent emergency planner (base algorithm [6]) is the same as for the multi-agent planner that is introduced by this paper. The search algorithm is also the

same, which is Anytime Weighted A* [24]. However there are differences in the search space structure that are described in detail in chapter 4 .

To build a search space one has to define a vehicle model first. This would apply restrictions on the search domain according to the vehicle capabilities (e.g. steep turns become impossible on high velocities). The vehicle model used in this paper is based on clothoids [10] and Dubin's curves. Given that a vehicle has a set of continuous parameter ranges (e.g. velocity, acceleration, etc.), the clothoid model provides a set of possible vehicle trajectories according to every parameter combination, considering that the parameter ranges are discretized. In other words, it generates all possible outgoing maneuvers depending on the current vehicle state. The maneuver's trajectory is finite in time and represent a transition between two vehicle states. Such trajectory is also called a motion primitive [16]. For every discretized vehicle state there is a set of motion primitives i.e. a set of possible successor states that the vehicle can reach from the current state. Utilization of motion primitives represent a continuous space discretization by building a search tree with the root in the initial vehicle position.

Using this vehicle model and the base single agent planner this paper introduces a multi-agent (cooperative) path planning algorithm where more than one vehicle is involved. This multi-agent algorithm represents a complex search problem that has to be parallelized to achieve faster operation. Ideally, it should match the performance of the single agent contingency planner that is capable of finding the solution in 0.1 seconds time frame. It should be pointed out that this problem was introduced as an example for a tree search with large branching factor and expensive expansions, which are typical properties of any deterministic path planner algorithm [9]. The heuristic function of the contingency planner is defined by velocity, i.e. the search progresses to the slowing down direction (contingency planning). However, in case other planning tasks like lane changing or parking are required, the heuristic can be adjusted and only minor changes have to be made to the planner to fulfill newly given requirements.

There exist many parallelization strategies for best-first search algorithms [4]. One of the first approaches was to use a standard dynamic scheduling to provide parallel state expansions. A number of states with the lowest cost were drawn from the OPEN list and expanded in parallel [18]. Though the algorithm sequence seems natural, it had two major drawbacks that were significantly reducing the performance. First one is synchronization overhead. Since OPEN and CLOSED lists are shared between all threads, the access to those lists shall be locked to avoid concurrent read and write. Second one is state re-expansions when the same state can be expanded more than once by different threads [28][18]. These difficulties occur for any parallel search, thus other attempts on developing robust multi-threaded search algorithms where aimed for reducing or completely avoiding these effects. One of the ways to reduce the synchronization overhead is to use private lists for each thread. This scheme would only require communication when distributing successors among other threads according to some hash function [8][19]. There are also several techniques on how to avoid or reduce the number of re-expansions [28][33].

In general most papers on parallel best-first search provide testing results for a most

common problems like puzzles or grid path-finding [3][19][28] which makes it hard to estimate possible performance for real world applications. Thus, all parallelization schemes had to be carefully evaluated before applying them to the existing planner problem to make sure that the applied strategy is feasible. This paper describes a few parallel search strategies that are a combination of several techniques existing nowadays and are intended to work exclusively with path planner problems or problems with very similar architecture. Since state re-expansion (one of the parallel search bottlenecks) does not occur for the tree search, this problem could be discarded for the path planner application. Therefore, it has been decided to assess the classical A* parallelization approach and attempt to make locking operations as effective as possible [9]. A hash-distributed approach was chosen as another strategy to appraise because of it's flexibility and the ability of search space distribution [8][19]. This work illustrates how different parallelization strategies behave for distinct road scenarios in terms of performance and memory usage. It also describes future improvements that might be applied to path planning parallelization techniques.

2. Theory

2.1. Motion Planning

Robot motion planning remains a challenging problem in artificial intelligence. Operation environments are often partially known, complex and dynamic while the computational power and planning time are limited. This leads to a difficult trade-off between computational capabilities and algorithm intelligence [17]. The goal of the planning algorithm is to find a sequence of actions that will transform an initial state of the robot to the desired goal state [9]. The problem can be modeled with the following non-linear differential equation:

$$\dot{x} = f(x, u) \tag{2.1}$$

Where x is a state, and u is an input (control action). This equation is known as a state transition equation that shows the state transition under a certain external input [29].

In most applications a deterministic heuristic-based search is used for path planning computation, considering that the dimensionality of the problem is relatively low [9]. For that, one has to define a search space that represent a given problem.

2.2. Continuous State Space Sampling

A robot is operating in the continuous space where computations are extremely expensive. In this case it seems natural to define a finite number of discrete states and an associated set of actions on top of a given continuous space [23]. In other words, environment of the robot will be represented by a graph $G = (S, E)$, where S is a vertex i.e. a state, and E is an edge associated with a corresponding action that has certain cost [9]. Such space discretization is a common approach of reducing computational complexity of planning. However it comes at the expense of algorithm **completeness**, meaning that only a limited number of actions is available for the current state (unlike a state in the continuous space) [31].

In path planning, a robot state is normally represented by a spacial coordinate and often the cost of the action is the distance from one state to another. Thus a typical problem of path planning is to reach from point A to point B in an optimal way. A path is called **optimal** if the cost sum of all state transitions from the initial state to the goal state is minimal over all possible path combinations. An algorithm is called **complete** if in finite time it can find a solution or indicate that there is no solution for the given problem [9]. The

complexity of the search grows with increasing search space dimensionality and real-time planning further contribute to the difficulty of the problem [29].

2.3. Control Space Sampling

The information about connectivity of the sampled state space is called the lattice [30]. The lattice introduces parameter constraints according to the given vehicle model and determines the interconnection between discrete states of the search space i.e. it makes sure that every action taken at the particular state is feasible for the robot.

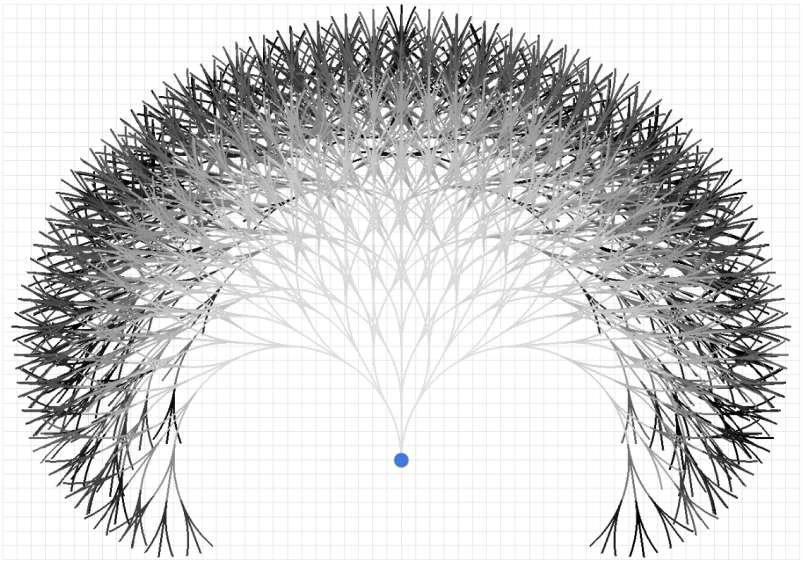

Figure 2.1.: Expansions of the control set in best-first order [30].

Lattice's structure is defined by robot's control system i.e. it's agility. Knowing the control system parameters one can compute a lattice i.e. a set of feasible motions to satisfy the requirements of the given problem [29]. A major advantage of such approach is off-line computation of the motion primitives (actions) that means it does not affect the runtime of the deterministic search algorithm, which is very important since the contingency planner introduced in this paper is time critical [30]. An example of the state lattice control set is shown in figure 2.1 .

2.4. Best-First Heuristic Search

After state and control space are defined, an actual path finding sequence shall be initiated. The planning is formulated as a search problem on a graph that is generated from a given state space.

There are classical graph search algorithms like Dijkstra and A* that are proven to be effective for many applications. However, since A* is an informed search i.e. has prior information about the search domain, it is able to search towards most promising states of the graph, that would eventually lead to a smaller computational effort [9].

2.4.1. A* Search

Let us define a set of all possible states S in some finite discrete state space where the A* computes a path from the initial state $s_{start} \in S$ to the goal state $s_{goal} \in S$. At each step A* considers two estimates of the state. First, the estimate of the path cost $g(s)$ from the initial to the current state. Second, a heuristic estimate of the cost to the goal $h(s)$ from the current to the goal state. Both estimates define a total cost of the state and introduce an evaluation function [15]:

$$f(s) = g(s) + h(s), s \in S \tag{2.2}$$

The heuristic $h(s, s_{goal})$ usually underestimates the path cost to the goal and is used to focus the search [9]. Before the search starts all states are initialized to $g(s) = \infty, \forall s \in S$. The search then begins with setting $g(s_{start}) = 0$ and add the s_{start} to the OPEN list. The OPEN list is a data structure that holds all states that are to be explored by the algorithm, that is also known as the frontier. A* pops a node with the lowest f value from the OPEN list and explore it by generating successors. The explored state is then added to the CLOSED list and the OPEN list is updated by adding previously generated successors that are satisfying the equation:

$$g(s) + c(s, s\prime) < g(s\prime), s \in S \tag{2.3}$$

Where $c(s, s\prime)$ is the transition cost from the current state s to the successor state $s\prime$, and $g(s\prime)$ is the cost of the successor state. The CLOSED list is a data structure that holds all previously explored states and when the successors are generated is first checked if none of them are in the CLOSED list already. The search ends when the algorithm pops the goal state from the OPEN list.

To satisfy the optimality criteria the heuristic $h(s)$ of the search should be **admissible**, which means it should be less or equal than the actual cost from the current state s to the goal state s_{goal} i.e. it underestimates the cost to the goal [15]. A slightly stronger definition is **consistency**, where:

$$h(s) \leq c(s, s\prime) + h(s\prime) \tag{2.4}$$

Every consistent heuristic is also admissible.

2.4.2. Anytime Weighted A* Search

In many applications optimal search is not required. Moreover, it is often infeasible e.g. in real-time algorithms when the solution has to be found in a very limited time frame. Anytime algorithms are used instead to satisfy the constraints provided by the peculiar properties of the problem [14]. Basic idea of anytime search is to compute a suboptimal solution in a given time frame, and proceed to the optimal solution if there is still time left. One of the drawbacks of anytime algorithms is that they do not provide a bound for sub-optimality of the algorithm. However, [24] introduces an algorithm that is capable of providing such bound and eventually guarantee an optimal solution.

The algorithm is called Anytime Repairing A* (ARA*). The main difference to standard A* is a so called inflated heuristic that overestimates the cost to the goal by a factor of inflation coefficient ϵ. The evaluation function then becomes:

$$f(s) = g(s) + \epsilon * h(s), \epsilon > 1, s \in S \tag{2.5}$$

Please note that the heuristic is not admissible anymore. The inflation factor ϵ is a bound on sub-optimality of the algorithm, meaning that the first solution found will be close to optimal by the ϵ factor. The more heuristic is inflated, the more the cost to the goal is overestimated and the search becomes depth-first [14]. Utilization of inflated heuristic proves to find a slightly less optimal solution much faster than regular admissible heuristic finds an optimal solution, that pays off for complex problems [24].

The ARA* algorithm is executing A* with inflated heuristic multiple times, starting with large ϵ and decreasing it's value with every execution until $\epsilon = 1$ (classic A*). Running a search from scratch at every iteration would be very computationally expensive, thus the ARA* reuses the results of previous iterations. Due to inconsistent heuristic of ARA* there might occur re-expansions of states, however it has been shown [24] that restricting the states to be expanded not more than once still provide a sub-optimality bound ϵ. Therefore, nodes that has been expanded at previous iterations are simply ignored and not inserted into the OPEN list. This allows ARA* to avoid redundant computations and converge to the optimal solution while finding sub-optimal solution on the way.

2.5. Parallel Best-First Heuristic search

Computer processor development is moving towards multi-core architectures, thus, algorithms and software has to be modified in order to exploit the power of modern CPUs. Parallelization of deterministic heuristic search algorithms is a challenging problem that arise in many applications. Nevertheless, there is no general parallelization approach that would work for any given problem [21]. Usually, parallel search techniques can be categorized according to lists structures (OPEN and CLOSED lists) and utilized memory architecture (shared or distributed memory).

2.5.1. Shared lists

The most basic approach on best-first search parallelization is to have OPEN and CLOSED lists stored in shared memory. All threads are accessing same memory space at the same time while managing states inside shared lists, that makes it necessary to use locking to restrict list access for only one thread at a time [18]. Since shared lists are updated after each expansion step, such method introduces a significant synchronization overhead by forcing threads to compete for the list access.

Synchronization overhead is also caused by multiple state re-expansions introduced by a parallel search nature. Standard A* never expands the state more than once because during expansion node's g value is guaranteed to be optimal. However, this rule does not hold when the expansion step is done in parallel and several items of the OPEN list are expanded simultaneously. In this case the search expands states with suboptimal g values and some of them might be re-expanded later during the search. In another words, a particular thread finds shorter (optimal) path to a state that has already been expanded by another thread, so that this state has to be expanded more than once [28]. This effect increases the load on the OPEN list, that causes even more synchronization overhead. Due to this fact, even a sequential version of A* can outperform such parallelization approach for a certain set of problems [3].

2.5.2. Private lists

This strategy implies possession of private lists for each thread that manages expansions. Though, communication is still required to distribute successors evenly among all threads to maintain load balancing. However, a communication overhead due to states distribution is much less compared to shared lists strategy [20]. States distribution is usually done by introducing a problem specific hash function, that assigns a thread index to a particular node. This assignment can either attempt to balance the load as much as possible, or partition the search space according to some problem specific parameters [21].

Private lists are normally chosen as a technique for distributed memory architectures and utilization of Message Passing Interface (MPI), however, it is also possible to use shared memory architecture with a proper memory access management. [19].

2.5.3. Bottlenecks

There exist a number of parallel best-first search strategies that were developed for different kind of problems [28][19][3]. Some of them claim to be effective only for path planning while others might be much better for puzzle solving. Unfortunately, there is no universal method that would effectively parallelize any given search problem.

To build a feasible parallelized search, first thing that should be considered is memory architecture of the system were the search is supposed to be executed. Often, path computation is performed on board of a robotic system that might have specific software or hardware architecture, and probably limited access to dedicated frameworks and libraries.

Second, the sequential search algorithm shall be analyzed for the state space structure, problem size, computational cost etc. Careful evaluation of search properties and given computer system allows choosing a best fitting strategy that would eventually lead to a successful parallelization attempt. Ideally a parallel version of A* has to overcome the following challenges:

- Load balancing
- State re-expansions
- Syncronization overhead due to list access

Keeping these in mind and considering path planning problem together with the given computer architecture we will have to choose a parallelization scheme that would suite the requirements of the multi-agent contingency planner.

2.6. Problem Statement

One of the existing planners developed in DLR computes an emergency breaking path to zero velocity and provides a solution within of 0.1 seconds margin, which is a threshold for computational runtime [6]. It was planned to continue the development of such planning algorithms, including multi-agent planners, with increased accuracy and efficiency that requires higher computational capabilities.

This work is intended to develop a number successful parallelization schemes for a set of path planners (single and multi-agent), study it's behavior and embed it into DLR's development environment. To exploit the effect of parallelization, this thesis introduces a cooperative (two-agent) contingency planner that has similar architecture as [6], but now planning is done for two vehicles. This planner represents a more complex search problem in higher dimensional domain to which we apply two parallelization techniques (chapter 8 , chapter 9).

Part II.

Vehicle and Environment Models

3. Motion Primitives

This chapter introduces a dynamic vehicle model based on Dubin's curves [23]. The motion primitives that will define a state lattice control set, will be generated according to this model.

3.1. Kinematic Vehicle Model

The model is defined by the following kinematic equations:

$$\begin{cases} \frac{dx}{dt} = V_x cos(\theta) \\ \frac{dy}{dt} = V_x sin(\theta) \\ \frac{d\theta}{dt} = V_x K \\ \frac{dV_x}{dt} = A_x \\ \frac{dA_y}{dt} = C \\ K = A_y / V_x^2 \end{cases} \tag{3.1}$$

Where x and y are spacial coordinates, V_x, V_y and A_x are velocities and accelerations along the axes, C is a constant, $K = 1/R$ is the curvature and θ is an orientation angle [6]. The illustration of the model is given in figure 3.1 .

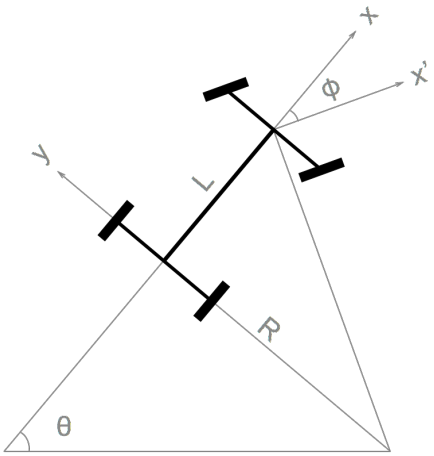

Figure 3.1.: Dubin's car model. R - curvature radius; L - vehicle length; ϕ - steering angle; θ - orientation angle; x, y - local coordinate system.

3.2. Constraint Graph

Since a real vehicle has physical constraints e.g. maximum braking acceleration, maximum speed, steering angle etc. these constraints should be considered by the vehicle model. The following set of constraints shall be used [6]:

$$\begin{cases} 0 \leq V_x \leq 30 \\ -7.8 \leq |A_{y\,max}| \leq 7.8 \\ A_y \leq |K_{max}| V_x \end{cases} \tag{3.2}$$

The first constraint in equation 3.2 is longitudinal velocity with the maximum value of 30 $[m/s]$ [6]. The second constraint is a bound of vehicle's lateral acceleration. The threshold value comes from the traction circle equation [27]:

$$\begin{aligned} \sqrt{F_x^2 + F_y^2} &\leq F \\ \sqrt{A_x^2 + A_y^2} &\leq \mu g \end{aligned} \tag{3.3}$$

Where μ is a traction coefficient and g is a gravitational constant. This work assumes that the traction coefficient μ_y has much greater impact on the resulting traction than μ_x, thus the μ_x can be discarded for the simplicity of the model. The equation 3.3 becomes:

$$A_y \leq \mu g \tag{3.4}$$

This is a constraint equation that provides a value for maximum reachable lateral acceleration according to the traction coefficient. In this paper the traction coefficient is chosen at the value $\mu = 0.8$ as the most average one, that results in $|A_y| = 7.8\ [m/s^2]$.

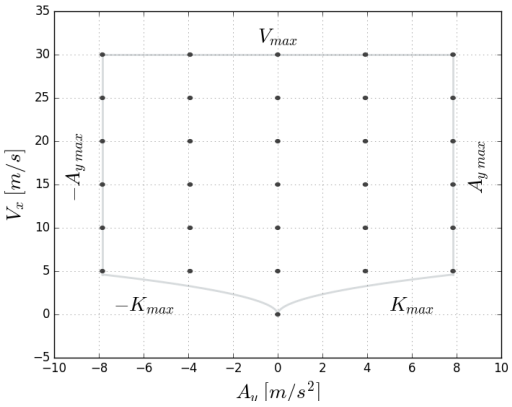

Figure 3.2.: Constraint graph that is used for motion primitives generation.

The third constraint of equation 3.2 is the curvature. Using the maximum value of the steering angle $\phi = \pi/4$ and the wheel base length of the vehicle $L = 2.7[m]$ one can compute the maximum curvature value:

$$K_{max} \leq \frac{tan(\phi_{max})}{L} \approx 0.37 \tag{3.5}$$

This is later used in equation 3.1 . Physically, the curvature constraint denotes that at lower velocities certain lateral accelerations are not available anymore. An interval enclosed by all physical constraints of the car (equation 3.2) is discretized with respect to A_y and V_x. The discretization introduces a state space sampling (chapter 2) or a constraint graph where every vertex represents a state of the vehicle (figure 3.2). A transition from one discrete vehicle state to another is done through a graph edge and called a motion primitive i.e. a trajectory with certain initial and final parameters.

3.3. Trajectory Curves

In order to build up a trajectory curve that is connecting two discrete states, few more assumptions shall be inserted into the vehicle model of equation 3.1 :

$$\begin{cases} A_x = sign(V_{x1} - V_{x0})\sqrt{\mu^2 g^2 - max(|A_{y1}|^2, |A_{y0}|^2)} \\ \Delta t = \frac{V_1 - V_0}{A_x} \\ \frac{dA_y}{dt} = \frac{A_{y1} - A_{y0}}{\Delta t} \\ A_y^2 + \frac{(V_{x1} - V_{x0})^2}{\Delta t^2} \leq \mu^2 g^2 \\ 0.5 \leq \Delta t \leq 2.5 \end{cases} \tag{3.6}$$

First formula in equation 3.6 provides an estimation of a longitudinal acceleration using a traction circle, that is later used to compute the duration of the state transition Δt (time that vehicle requires to cover the distance of motion primitive). The model assumes that the lateral acceleration is changing linearly with the state transition. Also, equation 3.6 introduces one more constraint for the trajectory duration. The lower bound makes sure that state transitions differ enough from each other and the upper bound eliminates unsafe long trajectories [6]. Since this work is dedicated to contingency trajectory planners, velocity of the vehicle is always decreasing along the trajectory curve. This means that the difference between initial and final longitudinal velocity $\Delta V_x = V_{x1} - V_{x0}$ will always be negative i.e. $sign(V_{x1} - V_{x0}) < 0$.

Satisfying equation 3.2 , equation 3.6 and solving the system in equation 3.1 provides a solution for state transitions in (x, y) coordinates. The illustration in figure 3.3 shows the evolution of vehicle's state progressing through the constraint graph (sampled state space). The corresponding discrete trajectories are shown in figure 3.4 . The introduced model is used to compute motion primitives with required constraints for every state in the constraint graph. All motion primitives are computed off-line, meaning that it is not affecting the runtime of path planning.

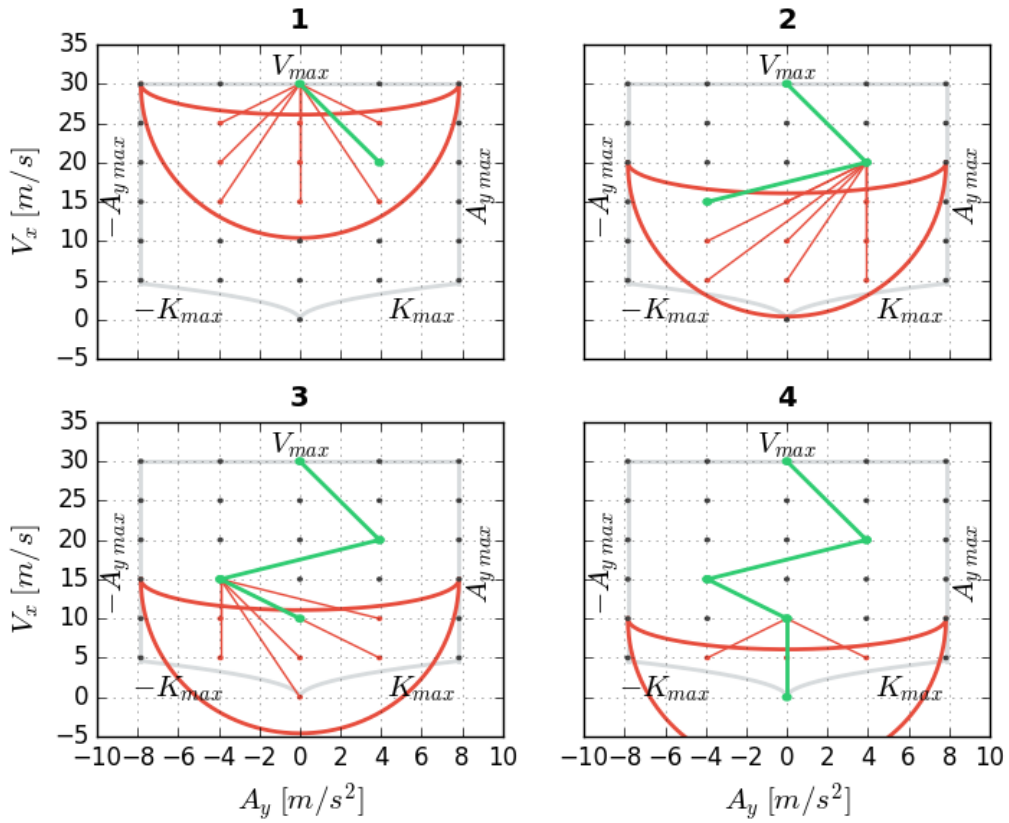

Figure 3.3.: State transitions on the constraint graph; half-ellipses denote time constraints from equation 3.6 ; **1** - initial state with the highest velocity; **2 - 4** - progressing through the graph to the zero velocity and lateral acceleration.

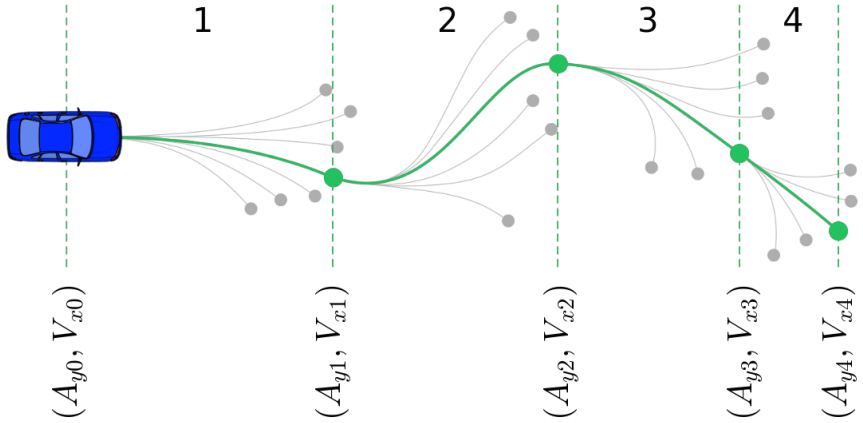

Figure 3.4.: A combination of discrete trajectories in (x, y) coordinates; numbers **1-4** correspond to plots at figure 3.3 .

3.4. Trajectory Enclosure Region

During path planning discrete trajectories computed with the model of equation 3.1 have to be checked for collisions against obstacles and road boundaries. Since a car has real physical dimensions, a trajectory curve of the vehicle's center of mass transforms to an enclosure region in three dimensions:

$$g(x, y, t) \leq 0 \tag{3.7}$$

There are different ways to define an enclosure set. This work utilizes OBB trees that provide a balanced solution both in computational effort and enclosure region resolution [12]. OBB tree model is depicted in figure 3.5 and figure 3.6 .

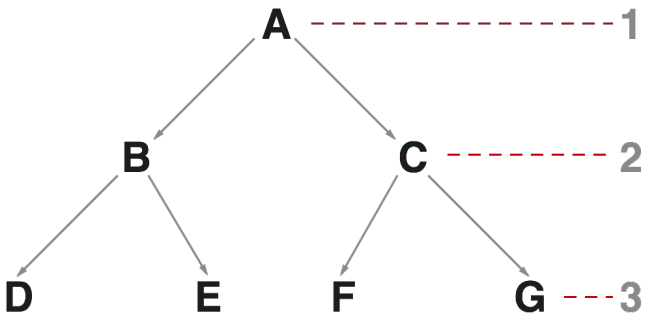

Figure 3.5.: OBB tree structure.

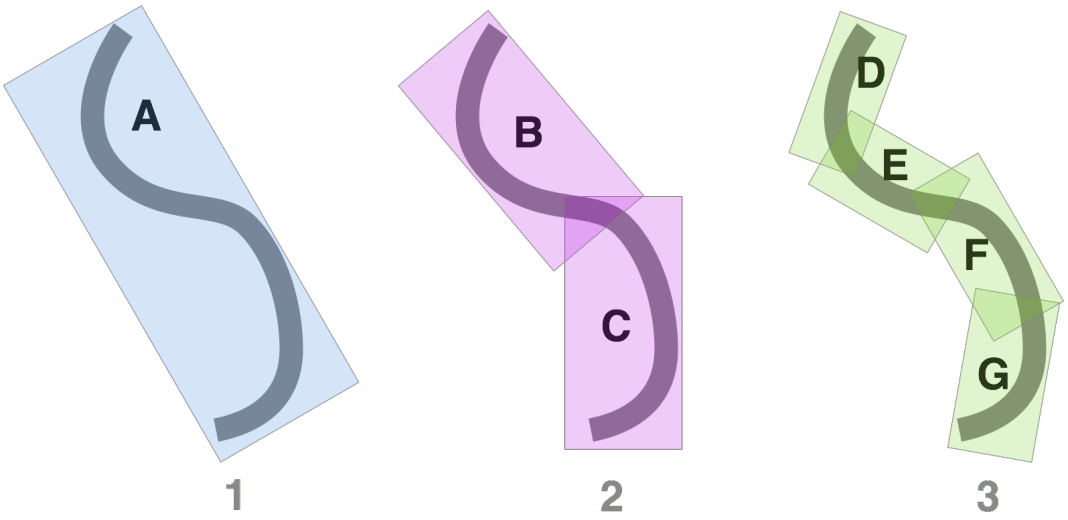

Figure 3.6.: OBB tree levels; numbers **1-3** correspond to tree's depth; rectangles **A-B** denote schematic enclosure of car's real physical boundaries (thick stripe).

Each generated motion primitive is provided with the corresponding OBB tree that defines it's physical boundary. An example of real enclosure region of vehicle's motion primitive is shown in figure 3.7 . Motion primitives also evolve in time that is considered during collision checking.

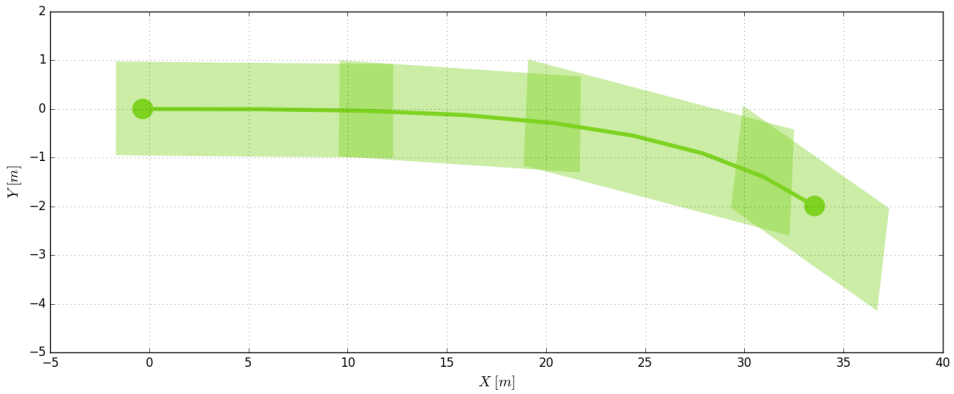

Figure 3.7.: An example of the motion primitive with OBB tree enclosure; an OBB is presented on maximum tree's depth of **3**.

4. State Space

After defining a vehicle model and generating motion primitives in chapter 3 , this chapter proceeds with building a state space for the heuristic search.

4.1. Single-Agent Planner State Space

As observed on figure 3.5 , every state is defined by lateral acceleration A_y and longitudinal velocity V_x. State transitions i.e. motion primitives are represented as curves in (x, y, t, Θ) space, where Θ is a global direction angle. Thus, according to the model of equation 3.1 , the state space of the planner should have six dimensions:

$$s_i(x, y, t, \Theta, A_y, V_x) \in S,\ i = 1...n,\ s_i \in \mathbb{R}^6 \tag{4.1}$$

Where S is a set of all discrete states satisfying the constraints provided by equation 3.2 and n is a number of discrete states. Every state s_i have a unique set of associated motion primitives computed as transitions to other states in the graph at figure 3.2 , assuming they are satisfying trajectory duration constraints in equation 3.6 . Such approach generates a tree of motion primitives in (x, y) space.

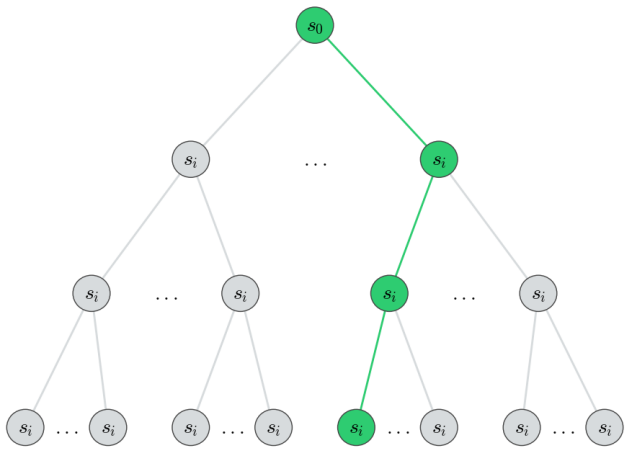

Figure 4.1.: Schematic representation of a single agent search tree with non-fixed branching factor and six-dimensional nodes; possible solution is shown in green.

This tree is a six dimensional discretized state space that is well suited for deterministic search algorithms. Tree's branching factor is varying depending on the current vehicle state and space discretization resolution (figure 3.2). A schematic representation of such search tree is presented in figure 4.1 .

A state space of equation 4.1 is used in [6] for single vehicle contingency planner algorithm that is further extended for cooperative (multi-agent) planning later in this work.

4.2. Multi-Agent Planner State Space

Since this work is focused on cooperative path planning, one should define a proper search space that is able to consider multiple agents (vehicles). Using the results from chapter 3 we will build a state space suitable for multi-agent planning.

4.2.1. States Combination Node

There are many techniques used for multi-agent search [7]. However, almost all of them can be classified in two categories - centralized and distributed. With the distributed approach each agent runs it's own independent search algorithm on private state spaces and when the goals are found, the paths are corrected accordingly, making sure that there occurs no conflicts (collisions) between agents [13] [26]. Centralized approach utilizes only one search algorithm for all agents and corrects every search step with regard to mutual collision avoidance during the runtime of the algorithm [7]. Latter method uses shared state space between agents and requires higher computational power since it considers all possible state combinations for a given number of agents. Nevertheless, this paper is going to use centralized approach for it's better suitability for a problem of vehicle trajectory planning with a given constraints of the model and hardware (see chapter 6).

Considering that every agent is associated with a certain initial discrete state and a set of successor states, a cooperative planner has to take into account every combination of successors to make an optimal decision at a current search step. This means that a cooperative state space will have states as a union of agent's states e.g.:

$$s_i^{coop} i = s_i^1 \cup s_i^2 \cup ... \cup s_i^m, \; i = 1...n, \; j = 1...m \tag{4.2}$$

Where m is a number of agents. In general, for an arbitrary number of agents it is required to find an n-fold Cartesian product of all successor state vectors, that will provide all possible combinations of outgoing trajectories. For instance, if there are three agents and each generates four successors states, all possible combinations can be written in an array of 3-tuples.

4.2.2. Computational Demands

Introduced strategy on cooperative planning increases the branching factor of a search tree exponentially with the increasing number of agents. Taking into account that a state space for a cooperative planner should operate in six dimensions and computational requirements of a centralized multi-agent approach (exponential growth of a branching factor), we will construct a state space for only two agents that will be enough to satisfy a stated goal of this thesis (chapter 1), since two-agent search is already enough to introduce a computationally demanding path planner algorithm for further parallelization. An illustration of a two agent search space is depicted in figure 4.2 .

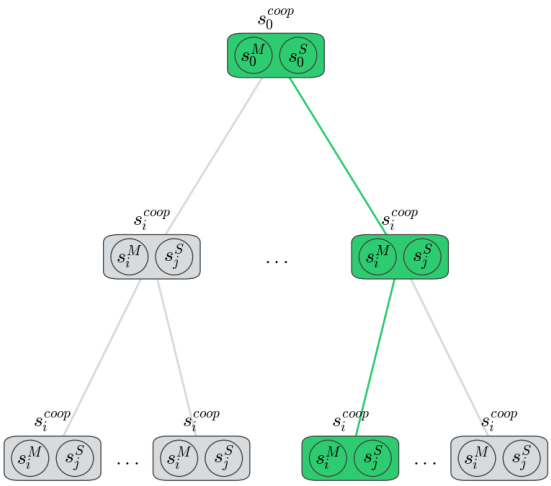

Figure 4.2.: Multi agent search tree; each node contains two vehicle states of six dimensions; branching factor is squared compared to a single agent search tree; M - index of a ”master” vehicle, S - index of ”slave” vehicle; possible solution is shown in green.

Essentially, a cooperative (two agent) search space is the same tree with non-fixed branching factor, however each node of this tree now holds two vehicle states. Due to centralized multi-agent strategy, only one vehicle will perform path computation. Therefore we will refer to those vehicles as ”master” - computes path, and ”slave” - recieves computed trajectory (see figure 4.2 and equation 4.2).

$$s_i^{coop} = s_i^M \cup s_i^S \cup, \; i = 1...n \tag{4.3}$$

An introduced vehicle model (chapter 3) provided a continuous space discretization technique that allowed to build a search space for a cooperative path planner. Despite the fact that chosen model of a search space (equation 4.2) is not optimal in terms of computational effort, it has some significant benefits for the given planning problem de-

scribed in chapter 1 . Since multi-agent search strategy is centralized, there is required no communication between agents that is a major advantage for the real world application. Radio transmission communication between vehicles (agents) deliver decent delays compared to the overall planning time, which makes a distributed multi-agent search strategy infeasible due to communication overhead.

Part III.

Path Planners

5. Contingency Planner

This chapter provides a brief introduction to a single agent contingency planner that was developed by DLR in 2015 [6] and is later extended to a cooperative planner (chapter 6). A contingency planner is intended to compute an emergency breaking trajectory so that it stops as soon as possible while not crossing the road boundaries and hitting road obstacles (e.g. other cars). This planner operates in a single agent discrete state space from chapter 4 with 16×9 discretization grid on a constraint graph (for V_x and A_y respectively).

5.1. Heuristic

The planner uses anytime search algorithm that with inflated heuristic and deliver suboptimal solutions before converging to the optimal one (equation 2.5) [24]. Since an optimal solution is often not required for path planners, this approach allows significantly reducing the planning time [14]. Planner's goal is to find zero velocity state while avoiding collisions on the way, therefore the following heuristic function was chosen:

$$h(s) = V_x / A_{max} \tag{5.1}$$

Where V_x is velocity in the direction of curve tangent and a_{max} is an absolute value of maximum acceleration defined by equation 3.2 . The cost from the initial to the current state $g(s)$ is defined by the duration of state transitions Δt i.e. motion primitives (see equation 3.6). Anytime search utilizes inflated heuristic (equation 2.5) which means that the cost to the goal is always overestimated and the search becomes more depth-first oriented. Initially, the inflation coefficient $\epsilon \geq 1$ is chosen arbitrary, but after the first solution is found, gets recalculated according to:

$$\epsilon_{new} = \min_{s_i \in S}(\epsilon_{old}, \frac{f(s_{sln})}{f_{min}(s_i)}),\ \forall s_i \in S,\ i = 1...n \tag{5.2}$$

Where s_i is a state in the OPEN list, ϵ_{old} is a previous inflation coefficient value, $f(s_{sln})$ is the non-inflated cost of the solution and $f_{min}(s_i)$ is a minimum non-inflated cost of the state among all states in the OPEN list. Anytime algorithm converges to classical A* with every newly found solution according to the decrease of inflation coefficient.

5.2. Planning Sequence

At first algorithm loads a precomputed set of motion primitives and parses the environment representation from the simulator that contains vehicle positions and road coordinates. Afterwards, the search is initialized by setting a number of default constants and

generating a root node with the initial parameters of the ego vehicle that are defined by a simulator (algorithm 1). Note that the velocity of ego vehicle provided by a simulator is continuous, meaning that it has to be fitted into a closest possible discrete state of the constraint graph figure 3.2 . Therefore, the discretization grid of lateral acceleration and longitudinal velocity has to be fine enough to minimize this discretization error. The root node is then added to the OPEN list and the search sequence is started. Since at this moment the OPEN list contains only one state (root node), the planner expands this node in order to generate successors. A set of successors is defined by two parameters of the state A_y and V_x that determine a node of the constrain graph figure 3.2 . Note that OPEN and CLOSED lists are heap data structures meaning that list ordering according to the f value is done automatically when the state is added to the list [5].

Algorithm 1 Initialize Planner Instance

```
function prepeareSearch():

    // Read input files
     graph = Graph("MotionPrimitives.csv")
     planner = Planner("Single-Agent")
     envRep = Environment("Scenario.xml")
     planner.setDefaultParameters()

    // Generate root node
     rootNode = envRep.getRootNode()
     OPEN.push(rootNode)
```

5.2.1. Exploration

Exploration means evaluating successors of the given state. Search object reserves memory and assigns values of f, g and h for the generated successor and also updates it's value of $f_{min}(s_i)$ that is later used to update the inflation coefficient. Each generated node is then added to the OPEN list (*evaluateState* function in algorithm 2).

5.2.2. Expansion

After successors evaluation is complete, the search picks a state with the lowest f value from the OPEN list. Prior to be expanded the node is examined to be inside the CLOSED list and safety verified by checking trajectory collisions against obstacles and road boundaries. If all checks are successful, the state is expanded i.e. generates successors (algorithm 2). The node is also checked if the expanded node is a goal node. If this holds, the search stops, inflation coefficient gets updated and OPEN list is being reordered. If there are still nodes to explore (OPEN list not empty), the search is going to continue with the updated evaluation function. In any case the expanded node is being added to the CLOSED list. Search results that has been found within the given time frame are written to the SOLU-

TIONS array. Nodes whose trajectories are intersecting with obstacles or ones that have f value above maximum, are being added to the DISCARDED list.

Algorithm 2 Search Step

```
function executeSearchStep(node):

      // Expand
      if node.getF() > fmax:
           return False
      elif planner.collisionRoadObstacle(node):
           return False
      elif node.isGoalNode():
           fmax = fsol
           planner.updateEpsilon(node, fmin)
           SOLUTIONS.push(node)
           CLOSED.push(node)
           OPEN.reorder()
           return True
      else:
           children = graph.generateChildren(node)

      // Explore
      for child in children:
           planner.reserveMemoryForState(child)
           planner.evaluateState(child)
           planner.updateFmin(child)
           if child.getF() < fmax:
                OPEN.push(child)

      // Successful Search Step
      return True
```

5.2.3. Collision Detection

Every motion primitive that is chosen to be expanded has to go through a safety verification check. This is done by examining an intersection of the motion primitive's trajectory enclosure region (chapter 3) with road obstacles and boundaries (*collisionRoadObstacle* function in algorithm 2).

5.3. Example

Planner routine implementation is done by algorithm 3 . The search terminates if all states were explored before the time horizon. A solution is a node with zero absolute velocity

from where the full path to the initial node can be restored. Resulting path represents a set of connected motion primitives that form a continuous closed six-dimensional set equation 4.1 . An example of contingency planner solution is shown in figure 5.1 .

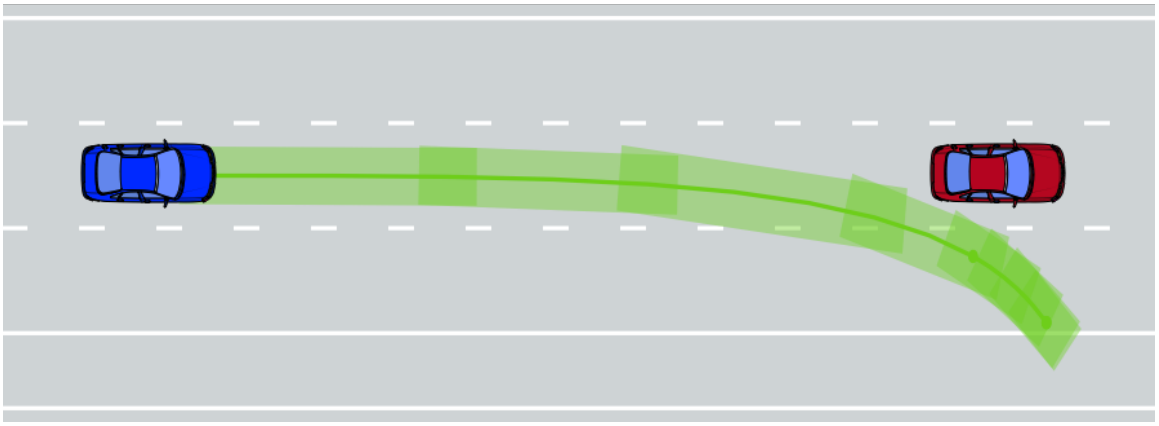

Figure 5.1.: Contingency planner example solution trajectory (algorithm 3)

Algorithm 3 Contingency Planner Path Computation

```
function computePath():

    // Initialize and generate root node
    prepeareSearch()

    // Search sequence
    while searchTime < timeHorizon:

        // Check if the frontier is not empty
        if OPEN.notEmpty():
            node = OPEN.pop()
            if CLOSED.contains(node) or DISCARDED.contains(node):
                continue
            if executeSearchStep(node):
                CLOSED.push(node)
            else:
                DISCARDED.push(node)
        else:
            return
```

Contingency planner with precomputed set of motion primitives finds optimal or sub-optimal solutions within a time frame of 0.1 seconds. In the next chapter are we going to build a contingency planner extension that would perform two agent path planning.

6. Cooperative Contingency Planner

This chapter introduces a two-agent contingency path planning algorithm that utilizes a state space from section 4.2 and the same set of motion primitives from chapter 3 for each agent of the system.

6.1. Modification to the Contingency Planner

The main difference between a standard single agent planner and a cooperative contingency planner is that each node now holds two vehicle states. All possible successor state combinations can be obtained by a Cartesian product of two successor sets (figure 6.1).

	S1	S2	S3
M1	(M1, S1)	(M1, S2)	(M1, S3)
M2	(M2, S1)	(M2, S2)	(M2, S3)
M3	(M3, S1)	(M3, S2)	(M3, S3)
M4	(M4, S1)	(M4, S2)	(M4, S3)
M5	(M5, S1)	(M5, S2)	(M5, S3)

Figure 6.1.: An example of discrete states combination; M1-5 (green cells) denote successors of the master vehicle state and S1-3 (blue cells) - successors of the slave vehicle state; table body (yellow cells) denotes all possible successors of the cooperative node.

Since a centralized multi-agent search strategy was chosen (section 4.2), path computation will be performed by only one vehicle. We will refer to a vehicle that calculates a path as a master vehicle and the second one as a slave vehicle. The idea of centralized planning for two agents is to minimize the expensive communication between real cars through a radio channel. As it was noted before, a method of centralized multi-agent path planning significantly increases a branching factor of a search tree. However, there are few techniques that allow minimizing this effect during successors generation process (see subsection 6.2.1).

6.2. Planning Sequence

The execution order of two-agent planner is not much different from the one presented in section 5.2 , however the process of nodes creation and evaluation is very particular. Let us define s_m as a current state of the master vehicle and s_s as a current state of a slave vehicle.

Discrete states of each vehicle are exactly the same as ones that were used for a single vehicle planner i.e. $s_i(x, y, t, \Theta, A_y, V_x)$. Generating one cooperative state of the search space requires both s_m and s_s discrete states (see algorithm 4 root node generation).

Algorithm 4 Initialize Cooperative Planner Instance

```
function prepeareSearch():

    // Read input files
     graph = Graph("MotionPrimitives.csv")
     planner = Planner("Multi-Agent")
     envRep = Environment("Scenario.xml")
     planner.setDefaultParameters()

    // Generate root node
     masterRootNode = envRep.getMasterRootNode()
     slaveRootNode = envRep.getSlaveRootNode()
     rootNode = CoopNode(masterNode, slaveNode)
     OPEN.push(rootNode)
```

6.2.1. Advanced Node Generation

State combination nodes are required to be generated, thus there shall be introduced a certain algorithm that computes a Cartesian product for the given discrete successor states. Before computing a Cartesian product one has to generate successors for both agent's states separately. To minimize the number of combined successor nodes (i.e. cooperative states), the algorithm first checks the intersection of enclosed motion primitives (OBB trees) with obstacles and road boundaries for each agent (algorithm 6). This early assessment allows significantly reduce the number of state combinations since many individual motion primitives will be discarded due to this check. Note that generated nodes (individual or cooperative) preserve parent-child relationship that makes it possible to restore the final solution path individually for each vehicle.

6.2.2. Node Assessment

Evaluation process of cooperative nodes has to provide a valid estimate for the search heuristic and evaluation function. For this purpose it has been decided to use averaged values of both master and slave states:

$$\begin{aligned} h(s_{coop}) &= (h(s_m) + h(s_s))/2 \\ g(s_{coop}) &= (g(s_m) + g(s_s))/2 \\ f(s_{coop}) &= (f(s_m) + f(s_s))/2 \end{aligned} \tag{6.1}$$

A cooperative node is considered to be a goal node if both master and slave are goal states (i.e. have zero absolute velocity). If only one of two vehicles has reached the goal state at a particular search step, the search is going to continue only for the moving one, while other vehicle will be assigned a zero-to-zero transition node (algorithm 5).

Algorithm 5 Cooperative Planner Search Step

```
function executeSearchStep(coopNode):

    // Expand
    if coopNode.getF() > fmax:
        return False
    elif planner.mutualTrajIntersection(coopNode):
        return False
    elif coopNode.isGoalNode():
        fmax = fsol
        planner.updateEpsilon(coopNode, fmin)
        SOLUTIONS.push(coopNode)
        CLOSED.push(coopNode)
        OPEN.reorder()
        return True
    else:
        children = graph.generateCoopChildren(coopNode)

    // Explore
    for child in children:
        planner.reserveMemoryForState(child)
        planner.evaluateState(child)
        planner.updateFmin(child)
        if child.getF() < fmax:
            OPEN.push(child)

    // Successful Search Step
    return True
```

6.2.3. Mutual Collision Detection

Another important difference to the standard contingency planner (chapter 5) is that cooperative successor trajectories have to be also examined for mutual intersection. The process is identical to collision detection for obstacles and road boundaries, however in this case both OBB's of vehicle's successor state pair are checked against each other. If this check fails, the search discards this node and proceed with the next one that has a lowest f value. As trajectories may have varying durations, each motion primitive has to be checked not only against the currently investigated maneuver of the cooperation partner, but also against all previous maneuvers in the search tree, which overlap in time. This

is necessary to avoid collisions between vehicles in cases when motion primitives with similar spacial coordinates and time boundaries occur at different tree levels (algorithm 7). An illustration of such example is shown in figure 6.2 .

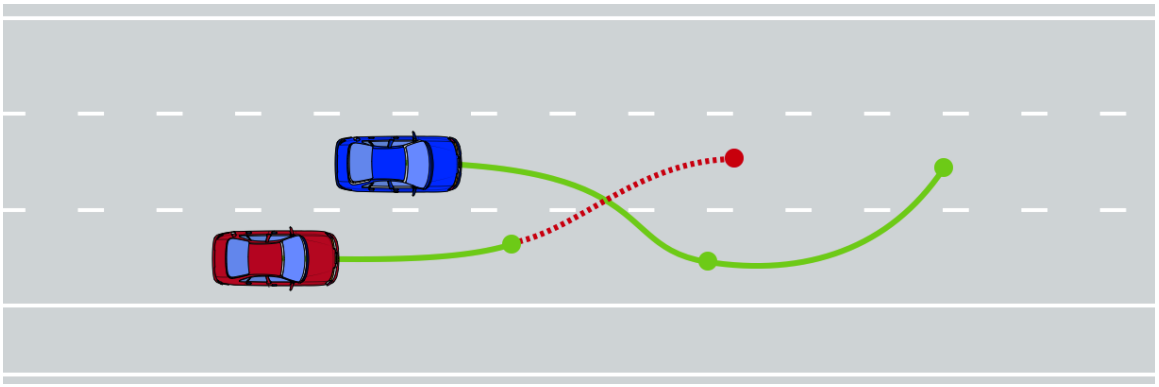

Figure 6.2.: Illustration of possible motion primitives intersection within distinct tree levels; to handle such cases mutual collision checking is implemented for motion primitives that are overlapping in time.

Algorithm 6 Successors Generation with Cartesian Product

```
function generateCoopChildren(coopNode):

    // Generate individual successors
    mChildren = graph.generateChildren(coopNode.masterNode())
    sChildren = graph.generateChildren(coopNode.slaveNode())

    // Cartesian product
    for mChild in mChildren:
        if planner.collisionRoadObstacle(mChild):
            continue
        for sChild in sChildren:
            if planner.collisionRoadObstacle(mChild):
                continue
            coopChildren.push(CoopNode(mChild, sChild))

    return coopChildren
```

6.3. Example

A cooperative planner computes two suboptimal (converging to optimal for unlimited time frame) trajectories that are safe from colliding with each other and surrounding environment (algorithm 2). An example of cooperative contingency planner solution is

presented in figure 6.3 .

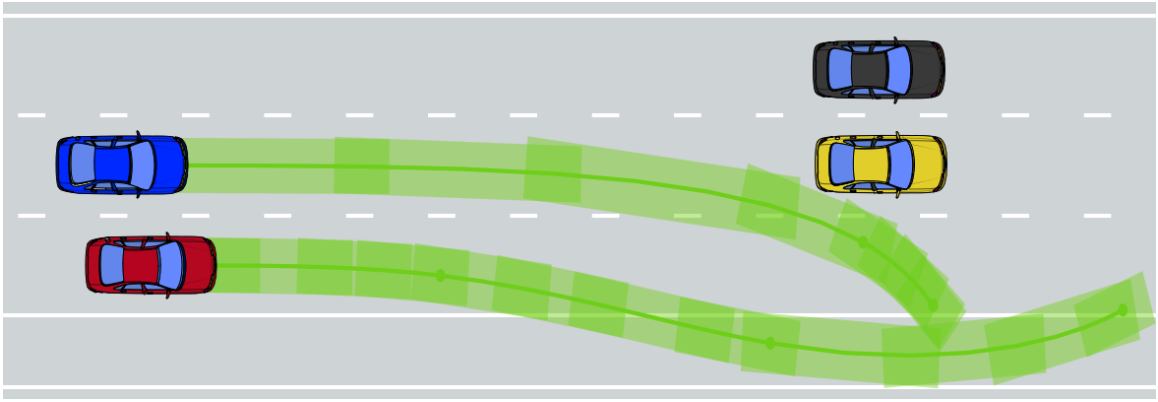

Figure 6.3.: Cooperative planner solution example; intersection region is not overlapping in time, even though it seems that trajectory enclosure sets are intersecting.

Algorithm 7 Mutual Trajectory Intersection Check

```
function mutualTrajIntersection(coopNode):

    // Get individual states
    mNode = coopNode.masterNode()
    sNode = coopNode.slaveNode()

    // Determine fixed node
    if mNode.getT() > sNode.getT():
        fixedNode = mNode
        floatingNode = sNode
    else:
        fixedNode = sNode
        floatingNode = mNode

    // All nodes that are intersecting in time with the fixed node
    nodesToCheck = fixedNode.intersectInTime(floatingNode)

    // Check intersections
    for node in nodesToCheck:
        if node.is IntersectingWith(fixedNode)
            return True

    return False
```

Part IV.

Best-First Search Parallelization Applied to Path Planning

7. Parallelization Strategy

In chapter 6 we introduced a computationally demanding two-agent path planning algorithm that is intended to find emergency breaking trajectories for two vehicles. This cooperative planner has been created as an example of a complex path planner with computationally expensive expansions that should benefit from parallelization. Considering a theoretical background on heuristic search parallelization presented in chapter 2 , this chapter will reason the choice of parallelization schemes applied in chapter 8 and chapter 9 .

7.1. Software Environment Limitations and Memory Architectures

All algorithms developed in this work were implemented in DLR's software environment called Dominion (Visual Studio 2010 Project), that simulates roads, vehicles and their properties. It also includes a number of interface classes for implementation of distinct path planners. Working in this environment puts some restrictions on utilization of specific libraries and classes e.g. for the development of a new path planner one must use provided interface classes. Furthermore, in case attaching additional frameworks or libraries to the Dominion requires fastidious dependency analysis, since some modules of the project might cause conflicts with new ones.

In this regard, attaching MPI (Message Passing Interface [1]) to the Dominion would be extremely difficult due to compiler differences. This fact encourages us to use shared memory architecture instead of distributed memory. There are two main multi-threading APIs for the shared memory architectures:

- POSIX Threads [2]
- OpenMP [32]

Both are powerful tools for parallel programs development. However, OpenMP provides less control when it comes to memory management due to pragma based syntax [22], that also makes it difficult to control the program execution when several classes and sources are involved.

Despite the complexity of POSIX threads, they provide the required robustness, flexibility and full control of memory [22]. Moreover, pthreads module was already included to the Dominion environment, that made it an ideal parallel computing tool to use under certain limitations of the project.

7.2. Lists Structures

To determine which list structure (shared of private) would fit better for the introduced planner, one has to evaluate the state space of the search. As we have already observed in chapter 4 , the state space of the planner is a tree with a big branching factor (tens and hundreds successors). Note that the given problem is not a general graph search planning, thus some properties or restrictions of best-first search parallelization might not be applicable anymore. For instance, state re-expansions due to expanding nodes with suboptimal g values. Since in the tree search each node generates a unique set of successors, re-expansion does not occur, thus, we can consider using a standard parallelization scheme [18] with the shared OPEN and CLOSED lists chapter 8 . Though this approach still has drawbacks like significant synchronization overhead, it also has some advantages like relative simplicity.

Since this work tries to find an optimal parallelization scheme for path planning routines, it would be interesting to try out another approach that deviate from classical techniques for shared memory architectures. Therefore, hash-distributed approach ([20][8]) that utilizes private lists, was chosen. Such scheme is usually used for distributed memory architectures. However, with the correct memory distribution it can be realized in the shared memory environment. Advantages of this scheme are less synchronization overhead on the list access and flexibility in terms of the hash function choice (load balancing and domain distribution).

8. Parallel A* Planner

This chapter describes a standard parallelization approach for the A* search described in [18] and [28].

8.1. Algorithm Overview

The operation principal of parallel A* is based on dynamic task assignment during search execution. Threads take turns removing states with the lowest f value from the OPEN list by locking it. Then the states are expanded in parallel and successors that were generated by each thread are added to OPEN list. As soon one of the threads finishes state expansion it waits for the OPEN list to be unlocked and takes the next state to expand. Note that OPEN and CLOSED lists are shared between all threads which means that any list access should be locked in order to avoid concurrent read/write inconsistency.

For this parallel search strategy one may expect significant synchronization overhead due to thread execution locking. Introduced cooperative search problem (chapter 6) utilizes a tree with the large branching factor as a state space representation that result in thousands of successors generated at every search step. Such load on the shared lists will have a great impact on algorithm performance, therefore we will attempt to structure search implementation in such a way that this load will be minimized.

8.2. Planner Architecture

Algorithm starts by initializing a planner instance and adding root node to the OPEN list. Afterwards, the threads are allocated and each thread is given a pointer to the planner instance and a unique thread ID. After initialization is done, threads start to compete for states in the OPEN list. Since at the first search step OPEN list contains only the root node, it will be expanded by one thread while others remain idle.

As it was mentioned before, shared lists have to be locked in order to maintain consistent access. Each list (i.e. OPEN - the frontier; CLOSED - expanded states; DISCARDED - states discarded due to large f value or detected collisions) possess it's own mutual execution lock ($pthread\_mutex\_t$ type) that only allows accessing the list for one thread at a time.

Before a thread generates successors, safety verification checks shall be executed (i.e. collision detection). For the sequential planner motion primitives generation implies generating an OBB tree (chapter 3) that is later used for safety examination by checking the intersection region of the OBB enclosure with road obstacles. Since a parallel version

of the algorithm may access the same OBB tree from different threads, there should be a mechanism to avoid concurrent read of the OBBs.

Algorithm 8 Cooperative Planner Parallel Path Computation (PA*)

```
function computePathParallel():

    // Initialize and generate root node
    prepeareSearch()
    allocateThreads(numThreads)

    // Search sequence
    while searchTime < timeHorizon do in parallel:

        // Lock OPEN list
        lock(OPEN)
        if OPEN.notEmpty():
            coopNode = OPEN.pop()

            // Unlock OPEN list
            unlock(OPEN)

            // Execute search step in parallel
            if CLOSED.contains(coopNode) or
               DISCARDED.contains(coopNode):
                continue
            if executeSearchStepParallel(coopNode, threadID):
                lock(CLOSED)
                CLOSED.push(coopNode)
                unlock(CLOSED)
            else:
                lock(DISCARDED)
                DISCARDED.push(coopNode)
                unlock(DISCARDED)
        else:
            // Unlock OPEN list
            unlock(OPEN)
            if explorations(threadID) > 0
                finishThreadExecution(threadID)
```

Instead of using another mutual execution lock, the search holds copies of OBB trees for each motion primitive. The number of copies should be equal to the number of threads times the number of search agents (i.e. two). In this case memory consumption is not very significant due to a relatively ”lightweight” and simple OBB tree implementation in the Dominion environment (chapter 7).

Algorithm 9 Cooperative Planner Parallel Search Step (PA*)

```
function executeSearchStep(coopNode, threadID):

    // Expand
    if coopNode.getF() > fmax:
        return False
    elif planner.mutualTrajIntersection(coopNode, threadID):
        return False
    elif coopNode.isGoalNode():

        lock(CLOSED)
        fmax = fsol
        planner.updateEpsilon(coopNode, fmin)
        SOLUTIONS.push(coopNode)
        CLOSED.push(coopNode)
        unlock(CLOSED)

        lock(OPEN)
        OPEN.reorder()
        unlock(OPEN)

        return True
    else:
        children = graph.generateCoopChildren(coopNode, threadID)

    // Explore
    for child in children:
        planner.reserveMemoryForState(child)
        planner.evaluateState(child)
        planner.updateFmin(child)
        if child.getF() < fmax:

            lock(OPEN)
            OPEN.push(child)
            unlock(OPEN)

    // Successful Search Step
    return True
```

Note that the search step (algorithm 9) requires a thread ID. This is done to make sure that a particular thread will utilize a private OBB tree for collision detection checks.

On the final stage of the search step generated successors are added to the locked OPEN list by each of the threads. When a solution is found by one of the threads, shared OPEN list gets locked and reordered according to the new f_{min} value. Afterwards, all threads proceed working with the updated cost function until another solution is found. Threads proceed until OPEN list is empty or until time horizon is reached. Algorithm pseudo code representation is implemented by algorithm 8 .

8.3. Optimization and Evaluation

Presented parallel search scheme (algorithm 8 , algorithm 9) has scalability difficulties as was discovered experimentally. Significant synchronization overhead was detected when adding states to the OPEN list on the stage of successors generation. The cause was adding states one by one from each executing thread, that triggered as many locking requests as there are successors generated at the moment, and as a consequence OPEN list was always locked. Logically that situation was getting even worse with increasing number of threads.

Algorithm 10 Advanced Explore Step For Parallel Execution (PA*)

```
// Fill private thread buffer without locking
for child in children:
     planner.reserveMemoryForState(child)
     planner.evaluateState(child)
     planner.updateFmin(child)
     if child.getF() < fmax:
          BUFFER[threadID].push(child)

// Lock OPEN and add all states from the private buffer
lock(OPEN)
for state in BUFFER[threadID]:
     OPEN.push(state)
unlock(OPEN)
```

To reduce the number of locking requests it was decided to use private buffer lists holding successor states of each thread (see algorithm 10). Generated states are added to the buffer of the dedicated thread without locking. Afterwards each buffer is added to the open list as a chunk of states that needs locking of the OPEN list maximum the number of executing threads. Final version of the parallel A* planner is scaling well and outperform sequential version of the algorithm (see chapter 10). The flow diagram of the algorithm is presented in figure **??** .

9. Hash-Distributed Parallel A* Planner

9.1. Algorithm Overview

Unlike standard parallel A* implementation (chapter 8), hash-distributed approach implies utilization of private OPEN and CLOSED lists for managing states [19], [20], [8]. Each list has it's private mutual execution lock that allows triggering a locking request only for one particular list while not affecting others. Such scheme requires communication between threads when distributing generated successor states and is usually implemented on distributed memory architectures. However, it can also be effective for the shared memory architectures with certain optimizations.

With this approach successor states generated by a particular thread have to be distributed among other threads according to one-to-all principal. State distribution is done with a hash function that determines which state has to be sent to which process. Hash function introduce a lot of flexibility to the architecture of the search algorithm. Depending on the given problem a hash function can serve as a domain distribution tool (search space partitioning) or it can maintain load balancing according to some parameters. In this work we will use a simple random function to distribute nodes so that the domain is evenly distributed among all processes.

9.2. Planner Architecture

The search starts as usual by generating a root node and adding it to the OPEN list. First thread to expand the root node, generate successor states and distribute them to other threads. As soon each thread has states in it's private OPEN list, it starts working on the search independently but gets locked when other threads try to add states into it, or when popping or pushing states from the list (algorithm 11).

9.2.1. Hash Function

As it was mentioned before, successor state distribution is done using a hash function, which in this work is a random function from the *STL* library (*std::rand()*). When the successor is created, hash function generates a natural number that denotes a thread ID that will be assigned to the newly generated state. Random function was chosen in order to maintain load balancing of the parallel implementation so no thread remains idle.

Algorithm 11 Cooperative Planner Parallel Path Computation (HDA*)

```
function computePathParallel():

    // Initialize and generate root node
    prepeareSearch()
    allocateThreads(numThreads)

    // Search sequence
    while searchTime < timeHorizon do in parallel:

        // Lock local OPEN list
        lock(OPEN[threadID])
        if OPEN[threadID].notEmpty():
            coopNode = OPEN[threadID].pop()

            // Unlock local OPEN list
            unlock(OPEN[threadID])

            // Execute search step in parallel
            if CLOSED[threadID].contains(coopNode) or
               DISCARDED[threadID].contains(coopNode):
                continue
            if executeSearchStepParallel(coopNode, threadID):
                lock(CLOSED[threadID])
                CLOSED[threadID].push(coopNode)
                unlock(CLOSED[threadID])
            else:
                lock(DISCARDED[threadID])
                DISCARDED[threadID].push(coopNode)
                unlock(DISCARDED[threadID])
        else:
            // Unlock local OPEN list
            unlock(OPEN[threadID])
            if explorations(threadID) > 0
                finishThreadExecution(threadID)
```

9.2.2. Communication

Since hash-distributed parallel search is usually implemented for distributed memory architecture, communication is commonly done through *MPI* send/recv functions. However, in our case send/recv operations are replaced with writing to a memory location that is ”private” for one single thread. In another words, shared memory location is logically divided into ”private” spaces that are assigned to each thread. Each private memory par-

tition has an OPEN, CLOSED and DISCARDED lists that have to be locked during access. Using separate mutual execution locks allows reducing synchronization overhead, since lists are locked independently of each other.

Algorithm 12 Cooperative Planner Parallel Expand Step (HDA*)

```
if coopNode.getF() > fmax:
      return False
elif planner.mutualTrajIntersection(coopNode, threadID):
      return False
elif coopNode.isGoalNode():
      CLOSED[threadID].push(coopNode)

      // Lock solution update
      lock()

      fmax = fsol
      planner.updateEpsilon(coopNode, fmin)
      SOLUTIONS.push(coopNode)

      // Reorder local OPEN lists
      for i in range(numThreads):
           lock(OPEN[i])
           OPEN[i].reorder()
           unlock(OPEN[i])

      // Unlock solution update
      unlock()

      return True
else:
      children = graph.generateCoopChildren(coopNode, threadID)
```

After a solution is found all private OPEN lists have to be reordered (see algorithm 12). OPEN lists update should be done by only one thread, therefore there is a mutual execution lock that restrict the access to the solution update.

Algorithm 13 Cooperative Planner Parallel Explore Step (HDA*)

```
for child in children:
     planner.reserveMemoryForState(child)
     planner.evaluateState(child)
     planner.updateFmin(child)

     if child.getF() < fmax:

          // Get random thread ID
          sendID = rand() % numThreads
          lock(OPEN[sendID])
          OPEN[sendID].push(child)
          unlock(OPEN[sendID])
```

Part V.

Results and Conclusions

10. Performance Evaluation

10.1. Criteria

This chapter evaluates the performance of implemented algorithms for distinct road scenarios. All scenarios will be tested with parallel cooperative path planners from chapter 8 and chapter 9 . Each algorithm will be executed fifty times for different number of threads for every scenario. The algorithms will be tested for speedup, scalability, synchronization overhead, number of explored nodes and compared to each other. Testing was performed on a *Windows 7* machine with *Visual Studio 2010* IDE on the following hardware:

- CPU: Intel Core i7-2600 3.4 GHz (8 logical cores)
- RAM: 8.00 GB DDR3

10.2. Road Scenarios

To provide a valid estimation of algorithm effectiveness we will test it for several road scenarios of distinct complexity, that require different amount of time and computational power. In all road scenarios blue cars represent *EGO* vehicles (i.e. cars for whom trajectories are computed) and gray cars act as steady obstacles(zero velocity). Both *EGO* vehicles move with velocity of thirty meters per second ($30\ [m/s]$). Scenarios supposed to represent a standard emergency situation when cars in front suddenly stop or appear on the road.

10.2.1. Normal Scenario

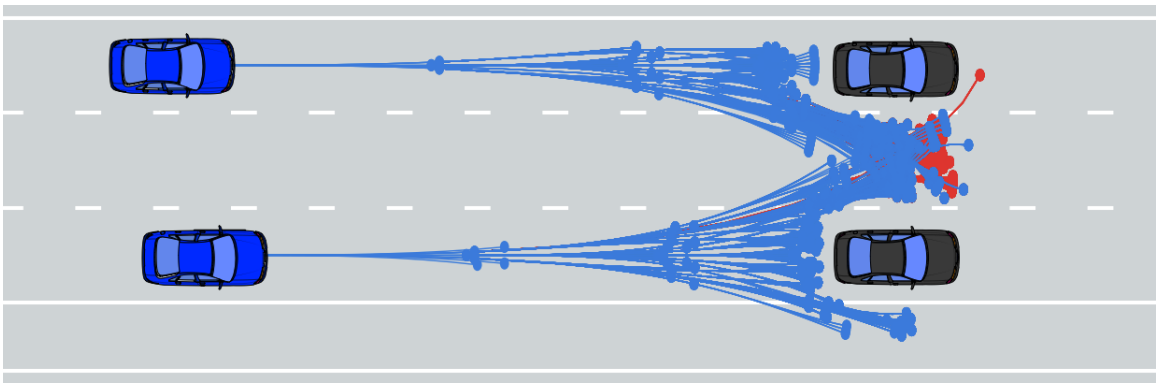

Figure 10.1.: Normal scenario; search tree of motion primitives; blue nodes - expanded states; red nodes - states discarded due to trajectory intersection.

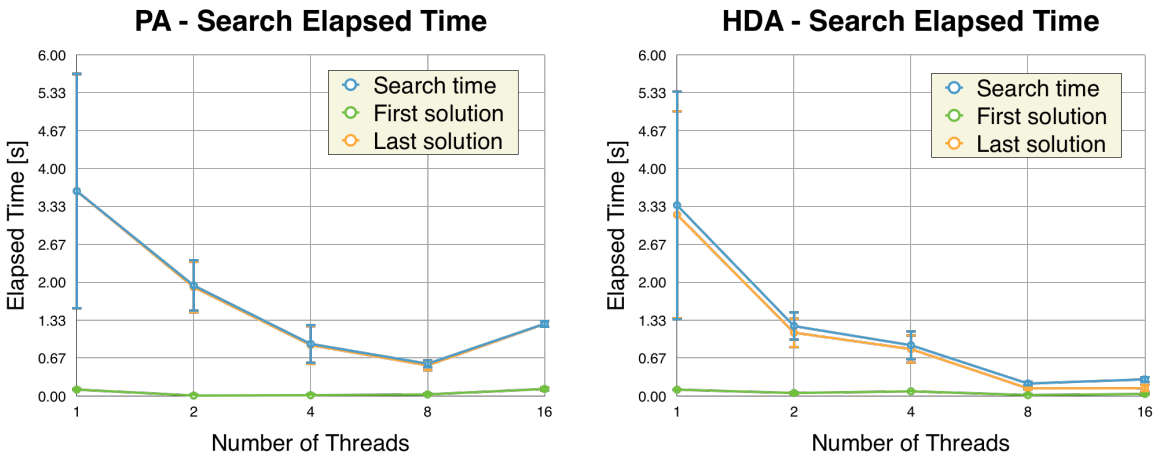

Figure 10.2.: Normal scenario; parallel and hash-distributed A* search elapsed time.

It was decided to arrange scenarios according to the elapsed time needed to compute a sub-optimal solution. Figure 10.1 illustrates a search tree for two *EGO* vehicles where blue lines are expanded nodes and red lines are the nodes discarded due to mutual trajectories intersections. There are also nodes that have been discarded because of collisions with obstacles or road boundaries, however they are not shown in the search tree in figure 10.1. Those invalid nodes are dismissed by the search on an early stage of successors generation as described in chapter 6 . Figure 10.3 shows the fastest sub-optimal solution found by one of the search algorithms. Since solutions may differ depending on the planner and the number of processes, this particular path represents one of the possible solutions that might be found.

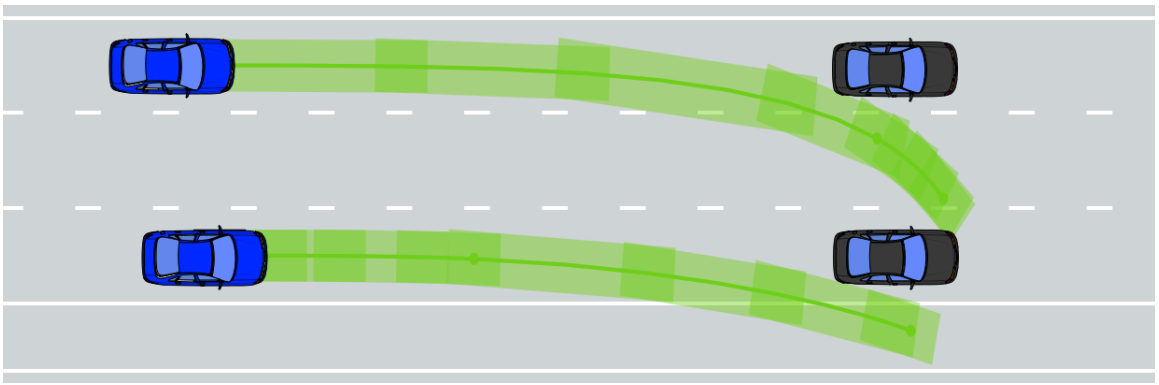

Figure 10.3.: Normal scenario; fastest sub-optimal solution.

Figure 10.2 presents the elapsed time of the path planning for both parallel search strategies implemented in this work. One may observe that parallel execution delivers speedup with increasing number of processes for both planners (PA* and HDA*). Note that for sixteen threads elapsed time is starting to increase because of the hardware limitations. As

mentioned earlier, Intel Core i7-2600 only has four physical cores with hyper-threading, meaning that it cannot effectively maintain more than eight processes at a time. Each plot (Figure 10.2) shows the elapsed time of a full tree exploration (empty OPEN list), the time of first solution found and the time of last found solution.

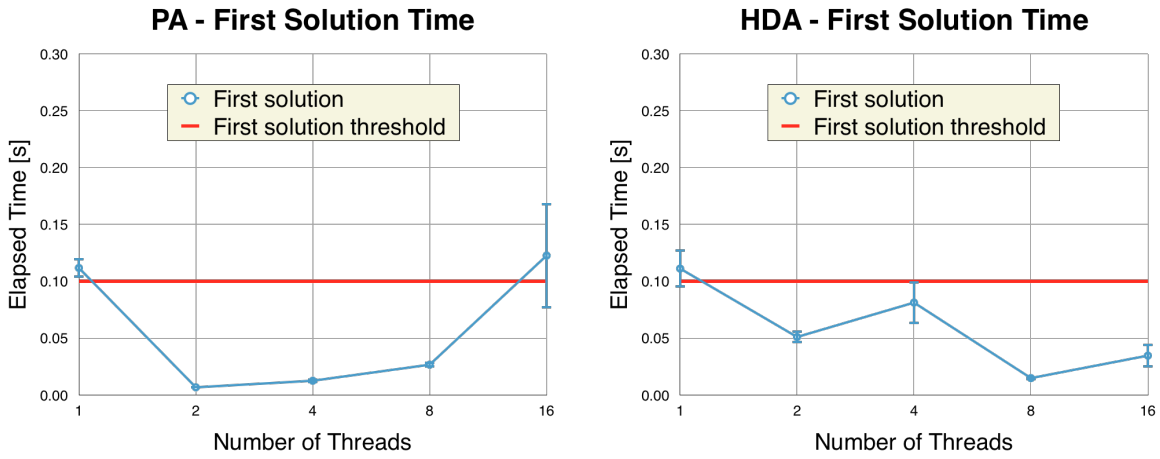

Figure 10.4.: Normal scenario; fastest sub-optimal solution compared to an execution threshold (red line).

figure 10.4 shows a comparison between the elapsed time of the fastest solution and execution time threshold (real time computation margin). As one may observe, both parallel planners are able to guarantee path computation within a provided time frame for this particular scenario.

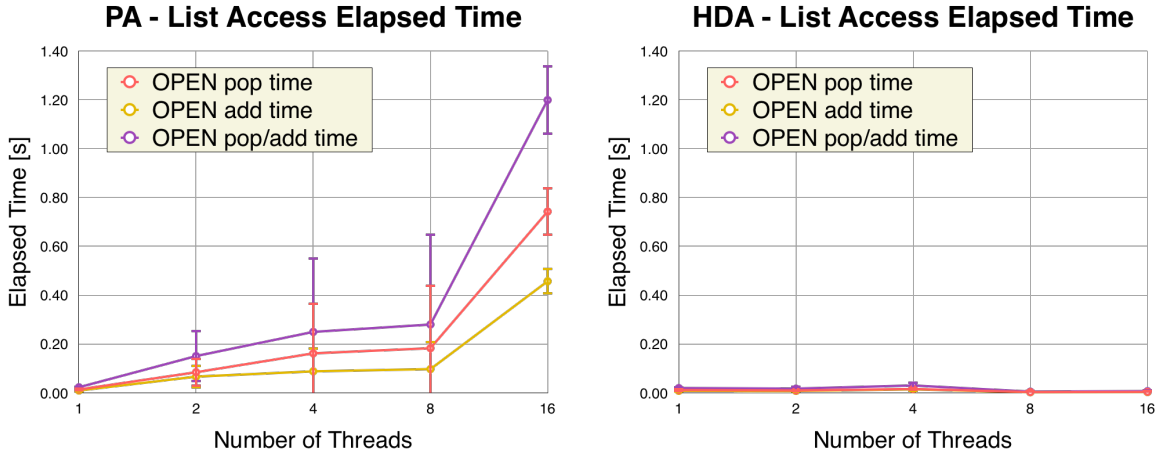

Figure 10.5.: Normal scenario; OPEN list access time.

Synchronization overhead can only be evaluated by measuring the time that every thread spent waiting to access an OPEN list (CLOSED list access is not important due to little load). Figure 10.5 shows OPEN access times during the entire search sequence for both parallel planners. It can be observed that standard parallel A* implementation introduces

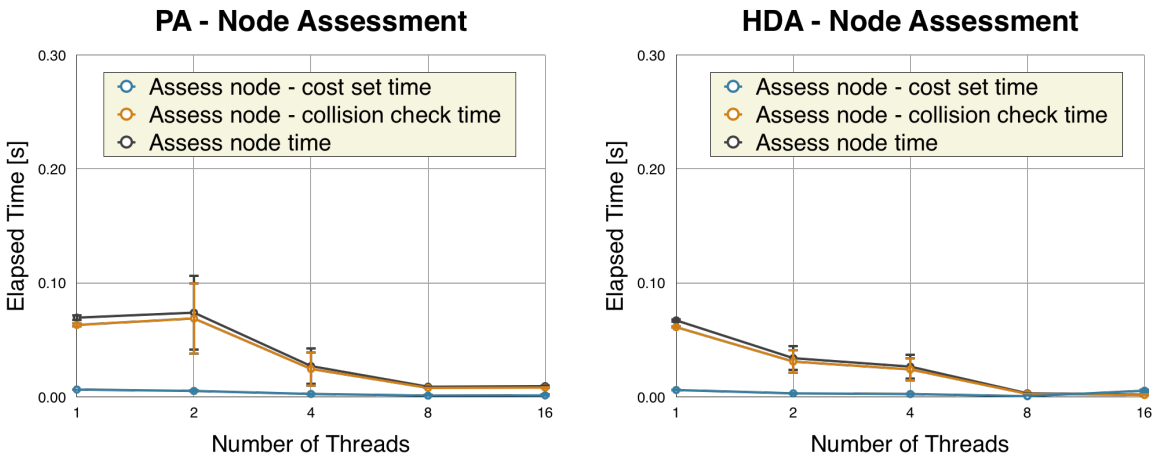

Figure 10.6.: Normal scenario; time spent for node evaluation and mutual collision checks.

significant overhead while a hash-distributed search shows very little of it, which was expected and implied by algorithm's architecture. Figure 10.6 shows how parallelization affects node assessment during the search. Clearly, node assessment time should decrease with increasing number of threads since it is the most time consuming part of both algorithms.

10.2.2. Difficult Scenario

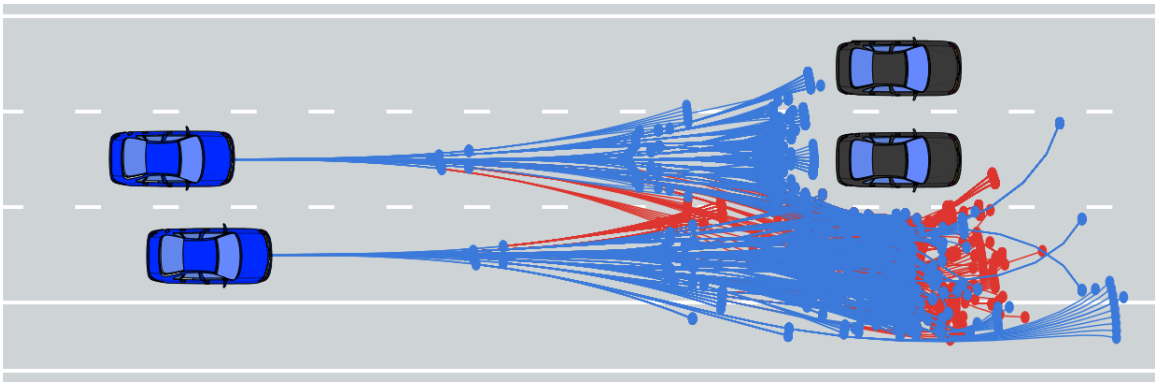

Figure 10.7.: Difficult scenario; search tree of motion primitives; blue nodes - expanded states; red nodes - states discarded due to trajectory intersection.

This scenario denotes a more difficult case for the planner, since both *EGO* vehicles have to fit into the rightmost part of the road without hitting each other (figure 10.7), that result in a bigger size of the search tree. A solution in figure 10.8 has an intersection region of two trajectories in the right bottom corner. Though is might seem that vehicles are colliding and the solution could be wrong, in fact, they are not. Since motion primitives are six dimensional and the plot in figure 10.8 is two dimensional, one cannot see the time axis. Trajectories are not intersecting in time, that can be observed only on a 3D plot.

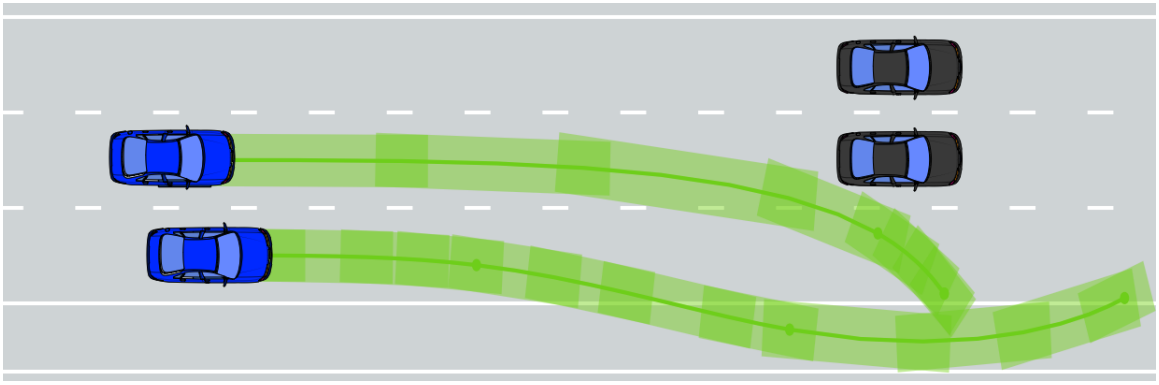

Figure 10.8.: Difficult scenario; fastest sub-optimal solution.

By looking in figure 10.9 and figure 10.10 one can see that the algorithm is still scaling, however, first solution is no longer fitting a 0.1 seconds threshold, since the scenario complexity has increased. The fastest solution time is not scaling as well as full tree exploration time. If there exist a sub-optimal solution that is relatively close to the root node, performance of the parallel implementation is not much different from a sequential one since it is reachable within very few search steps. In this case parallel expansions do not deliver desired effect, because speedup is compensated by the overhead on list access. This denotes that for easy solutions path finding parallelization might not be that effective as for complex solutions. Similarly to the *Normal* scenario, synchronization overhead is increasing for parallel implementation with shared lists (figure 10.11) and node assessment time is going down (figure 10.12). Note that the overhead estimation is approximate since there where no special tools used (e.g. parallel execution time measuring frameworks or libraries). Time measurement was performed with *Windows* system functions that might have caused some minor inconsistency is the results of overhead estimations.

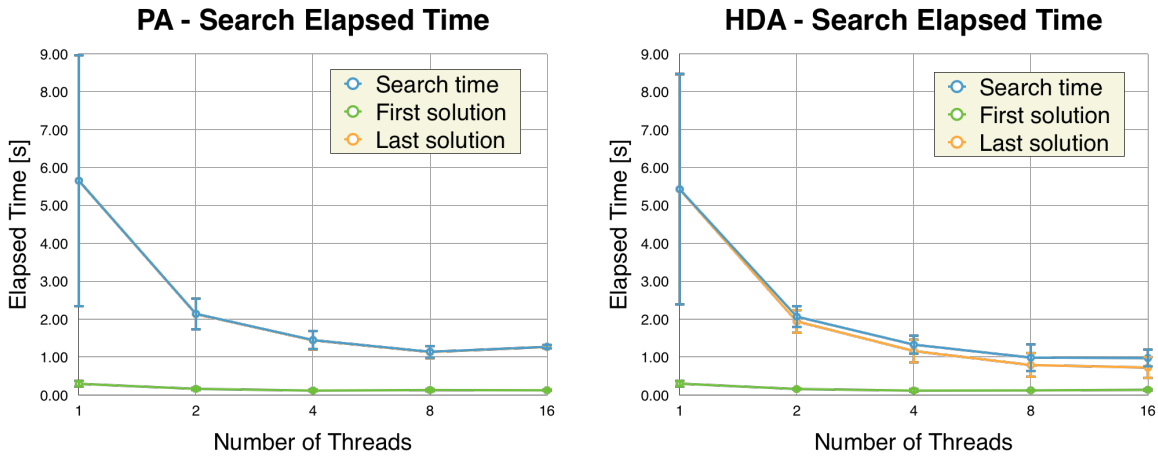

Figure 10.9.: Difficult scenario; parallel and hash-distributed A* search elapsed time.

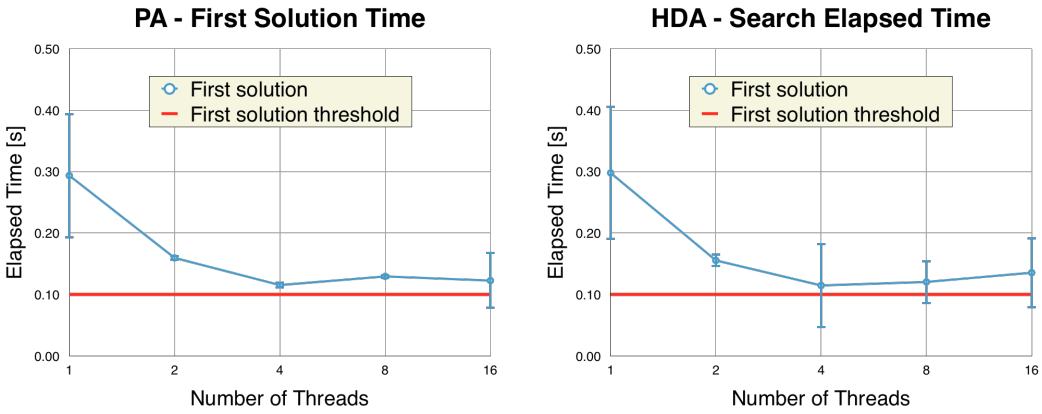

Figure 10.10.: Difficult scenario; fastest sub-optimal solution compared to an execution threshold (red line).

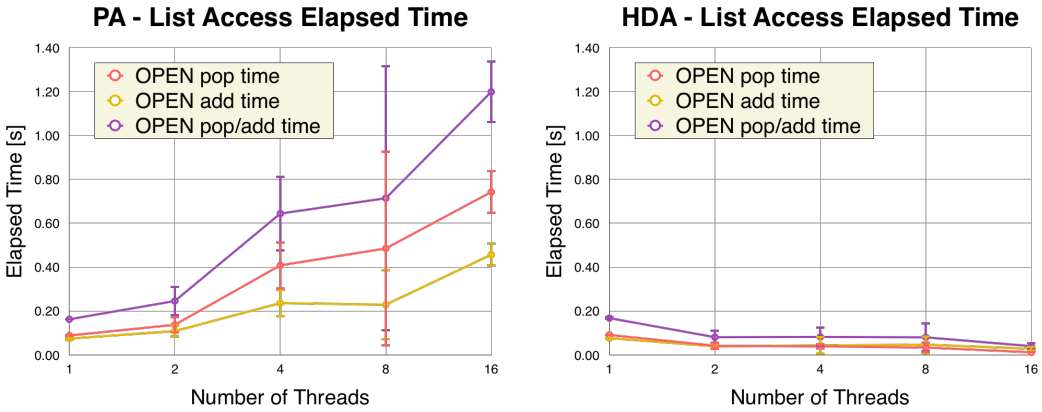

Figure 10.11.: Difficult scenario; OPEN list access time.

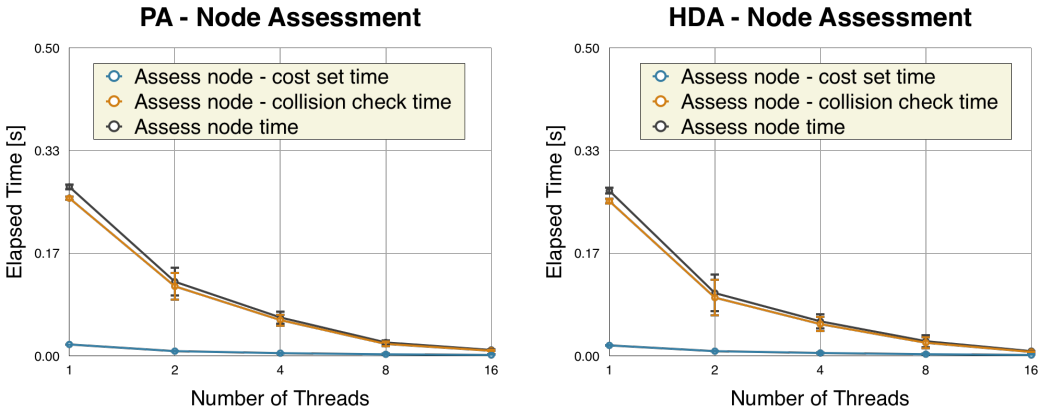

Figure 10.12.: Difficult scenario; time spent for node evaluation and mutual collision checks.

10.2.3. Extreme Scenario

The goal of this scenario was to see algorithm's behavior in extreme situations. Moving with over 100 km/h both vehicles are forced to perform emergency breaking while fitting one lane of the road (figure 10.13 and figure 10.14).

Figure 10.15 shows that the search execution time is much greater than for other two scenarios presented in this paper. Also, the fist solution is far from 0.1 seconds threshold and unpredictable in terms of scaling (figure 10.16).

Figure 10.17 shows that even for hash-distributed parallelization there is a list access overhead which was not the case for two previous scenarios. One may conclude that the amount of nodes added to the private lists is significant to introduce an overhead if the problem size is big enough.

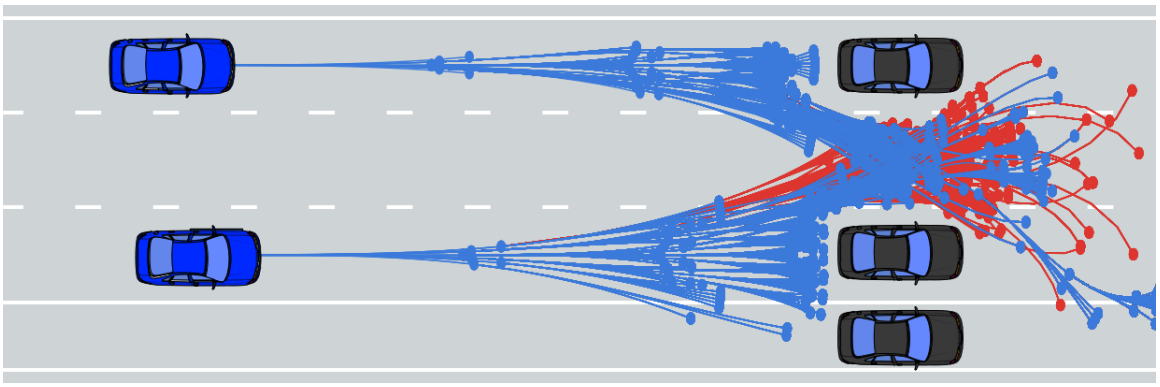

Figure 10.13.: Extreme scenario; search tree of motion primitives; blue nodes - expanded states; red nodes - states discarded due to trajectory intersection.

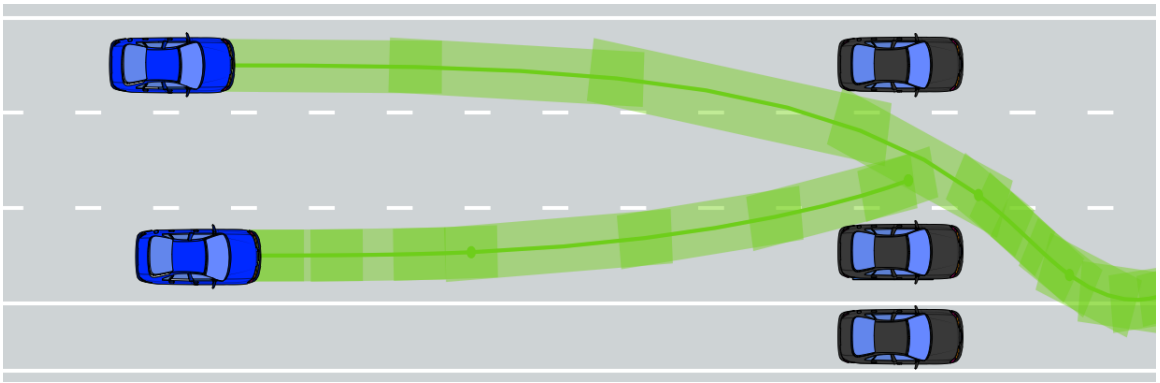

Figure 10.14.: Extreme scenario; fastest sub-optimal solution.

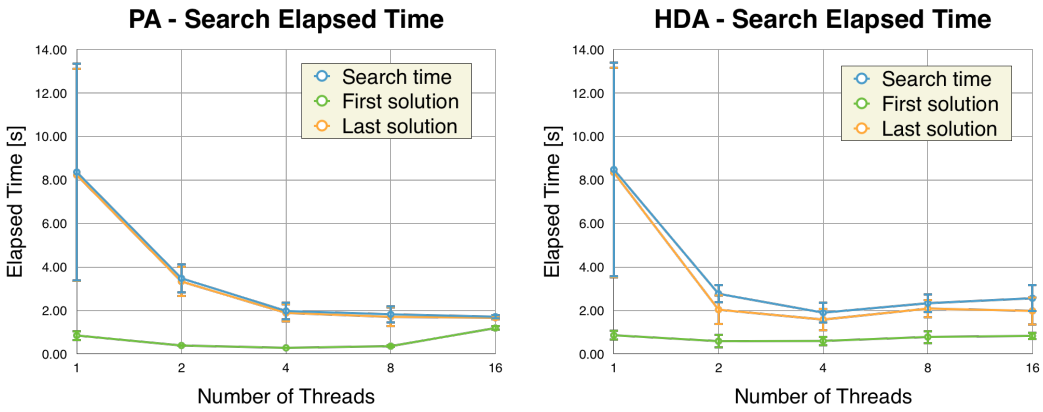

Figure 10.15.: Extreme scenario; parallel and hash-distributed A* search elapsed time.

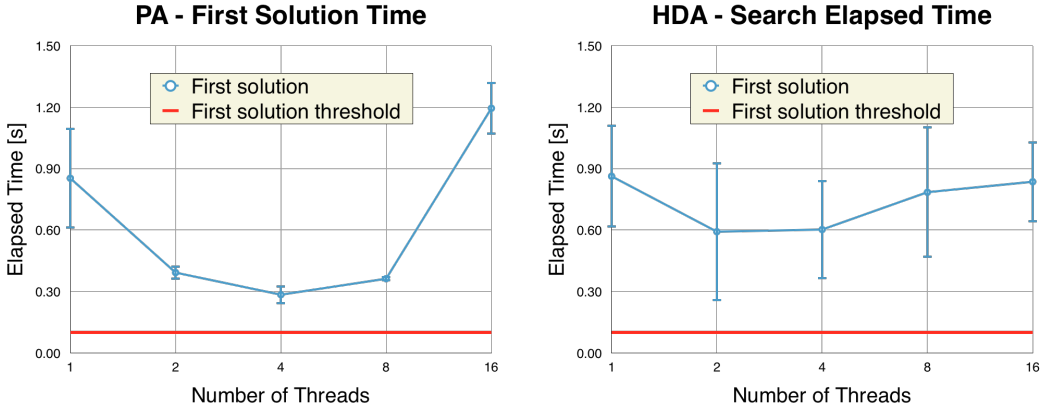

Figure 10.16.: Extreme scenario; fastest sub-optimal solution compared to an execution threshold (red line).

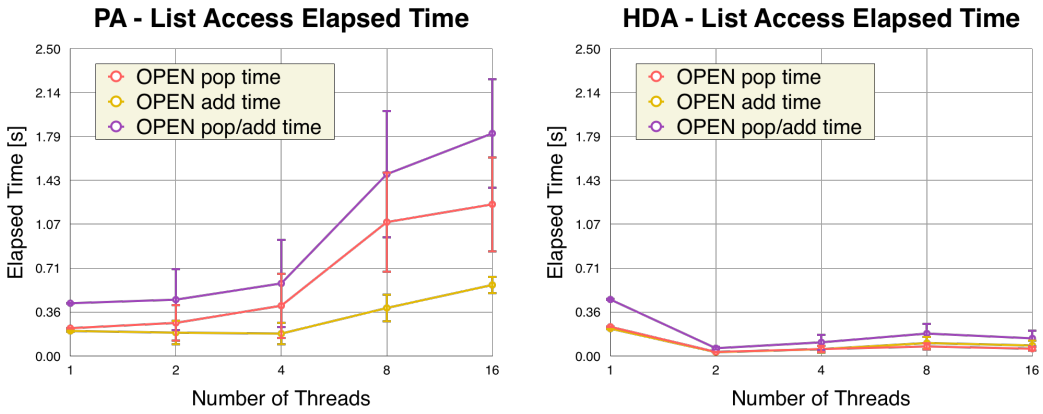

Figure 10.17.: Extreme scenario; OPEN list access time.

10.3. Summary

Both parallelization strategies proved to deliver speedup for all three road scenarios. While standard parallel A* was slightly slower because of the synchronization overhead, the difference to hash-distributed algorithm was not drastic. Though, not all scenarios were successfully resolved within the real time boundary of 0.1 seconds. Complex road situations still require faster algorithms.

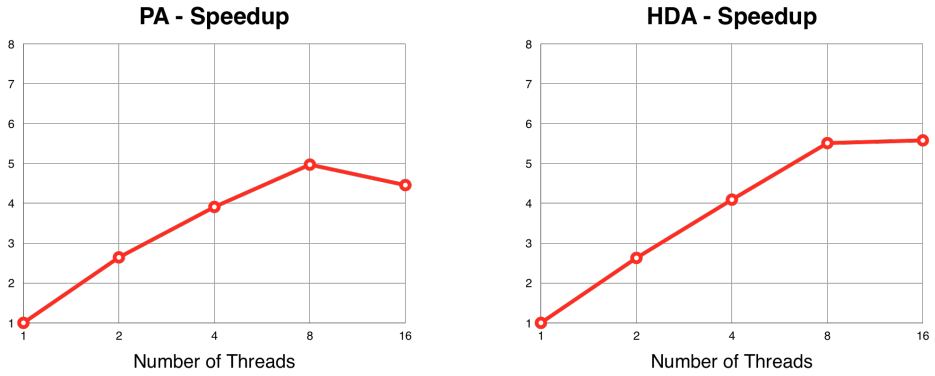

Figure 10.18.: Difficult scenario; speedup.

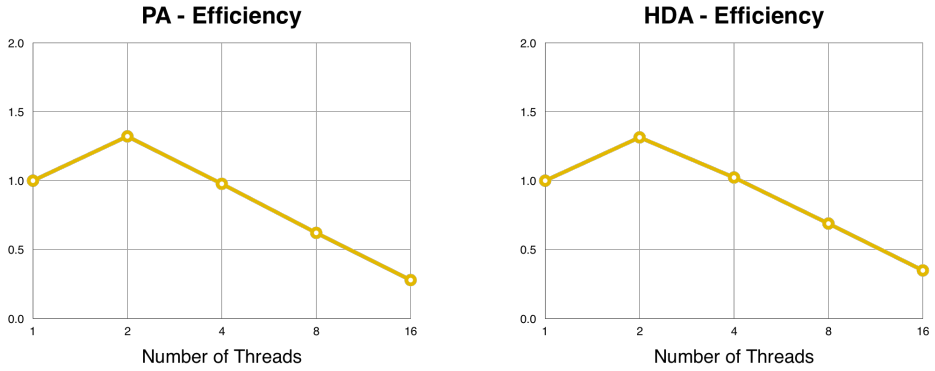

Figure 10.19.: Difficult scenario; efficiency.

Summarizing the outcomes from all three scenarios one may conclude that the difficult scenario is a most generic one, thus it was chosen to represent generalized parallel search parameters (subsection 10.2.2). Figure 10.18 and 10.19 respectively show speedup and efficiency comparison for both parallel implementations for a difficult scenario. Note that both parameters are given for the time of full tree exploration i.e. time of search termination. The speedup is super linear up to 4 processors for both algorithms, that can be explained by two factors. First, synchronization overhead is low since only two processors are involved in computations (see figure 10.11). Second, problem size got smaller due bigger size of the frontier and consequently better quality expansions (see figure 10.20) that results in many nodes being discarded due to large f_{max} value and eventually less problem size. With increasing number of threads, the overhead takes over the computa-

tion time, while problem size is still shrinking. Nevertheless, speedup decreases and with sixteen processors decline rapidly due to hardware limitations as was explained earlier. Efficiency is defined as speedup divided by a number of processors and shows how effective can algorithm perform for particular number of processors (another measure of speedup). Logically that efficiency is decreasing with growing number of processors because of reasons explained earlier in this paragraph.

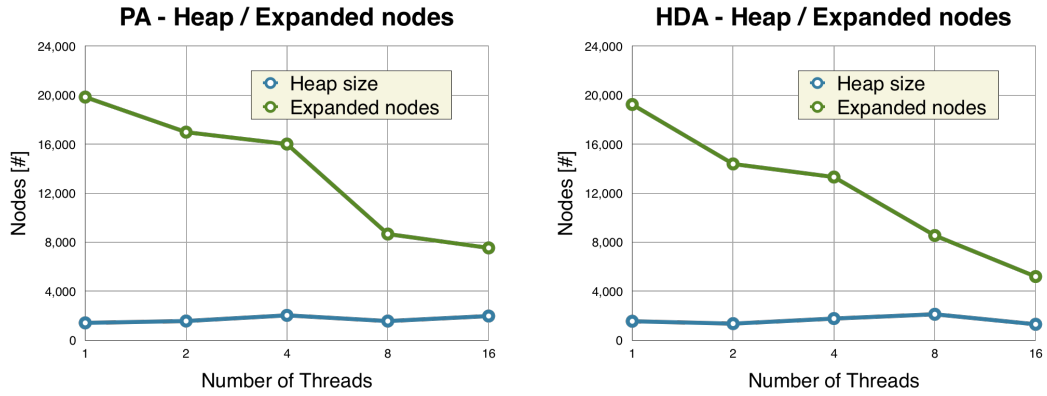

Figure 10.20.: Difficult scenario; maximum heap size and number of expanded nodes.

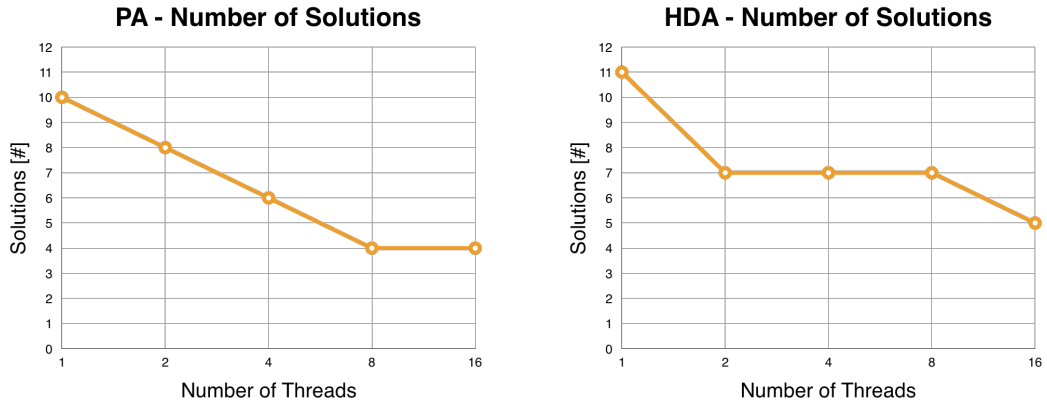

Figure 10.21.: Difficult scenario; number of solutions found after full tree exploration.

Even though the fastest solution is not scaling very well for both parallelization strategies (e.g. figure 10.12), parallel search tend to find solutions that are closer to the optimal one within the same time margin as a sequential version. With parallel implementation the frontier size is slightly greater compared to sequential version, meaning that the search has more options to pick a node with the lowest f value to expand, that eventually lead to better quality solutions and less expanded nodes (figure 10.20). Moreover, for the same reason, a number of found solutions and expanded nodes is decreasing with increasing number of processes for both planners (figure 10.21). In terms of memory usage we can conclude that both parallel implementations require to reserve more memory to store the frontier, but the search tree is smaller compared to sequential algorithm which means we need less storage to manage the CLOSED list.

11. Conclusions and Future Work

11.1. Conclusion

This work has studied an effect of parallelization applied to path planning algorithms. It was implied that parallelization should significantly improve algorithm's performance by providing much greater data processing capabilities. Since there has been very little research in this area [28], Deutsches Zentrum für Luft- und Raumfahrt has funded the work and provided all development tools like simulation environment and real vehicle prototype.

First task was to introduce a complex search problem that might utilize advantages of parallel algorithms. For that purpose an existing contingency path planner developed by DLR was extended to work with two agents [6] which significantly increases search complexity. Two-agent planner was supposed to compute emergency breaking trajectories for two vehicles while avoiding obstacle collisions. In order to create such planner, a new state space had to be defined that would hold a combination of individual vehicle states. Both single-agent and two-agent planners use informed best-first heuristic search to compute a path to the goal.

For a cooperative algorithm search evaluation function represents a summation of these functions for individual states. By such manipulation the state space preserve a search tree structure with a large branching factor. State transitions are represented by motion primitives with corresponding enclosure regions. During the search motion primitives (discrete trajectory curves) are checked for intersections against obstacles (other vehicles) and road boundaries. Also, trajectories are examined for intersections against each other, since there is more than one vehicle for whom a path is computed.

Parallelization strategies were chosen according to the search space structure and Dominion environment capabilities chapter 7 . Two approaches were chosen to parallelize a cooperative contingency planner:

- Standard Parallel A*
- Hash-Distributed A*

First approach uses shared lists to maintain search execution. The algorithm is considered to be the most standard A* parallelization technique that requires minimum modifications to the sequential version, though it has major drawbacks like large synchronization overhead. Second approach is considered more advanced. It uses shared lists to keep track of visited states and distribute them according to a defined hash function, that demands

interprocess communication.

Both parallelization strategies were tested for three different road scenarios to determine their performance (chapter 10). It was identified that planners with chosen parameters provide good results on speedup for all presented scenarios. It has been proven that parallelization of path computation can be effectively implemented and applied for real world applications. All algorithms were implemented in Dominion environment (chapter 7) with POSIX Threads API.

11.2. Future Work

Parallel path planners presented in this work are capable of computing cooperative emergency paths for two agents but also can be easily extended to any other planning problem that utilizes motion primitives. Dominion environment (chapter 7) allows using one search strategy for different types of problems and not restricting the usage of parallel search only for cooperative planning.

This fact provides a lot of flexibility in terms of testing parallel algorithms applied for path planning, which is very important considering that there are endless opportunities for tuning and adjustment for better performance.

11.2.1. Possible Improvements on the Sequential Planner

The base contingency planner used a so called triple set heuristic that was intended to improve performance of the search [6]. The idea was to split a set of generated successors into three subsets that were given weights according to the obstacle distance on their way. Distance to the goal was underestimated when the obstacle was closer and vise versa, that allowed to direct a planner towards an obstacle free area avoiding expensive collision checking.

However, this strategy could't adequately work when obstacle density was high enough, and testing it with cooperative planner identified that it works even slower than simple anytime heuristic. Though, the idea of speeding up the search by establishing obstacle-heuristic dependency remains interesting. To make it work properly one might introduce an obstacle density function that would determine heuristic weight according to the final coordinate of the motion primitive. Same strategy can be applied to road boundaries so that trajectories closer to the edge have less chance to be expanded.

11.2.2. Hash Function Choice

As it was mentioned before, parallelization strategy with the hash-function successors distribution was chosen for it's flexibility (chapter 9). In this case flexibility means that a hash function can be anything depending on the given task. For instance, if we would want to assign parts of the search space to different threads, the hash function would produce a coordinate-thread mapping that would determine which thread would process which

state. Such approach would be effective when the state space is required to be explored independently that does not require a lot of communication, however is not providing desired load balancing.

Random hash function on the other hand can maintain perfect load balancing. Nevertheless, it requires interprocess communication at every search step that introduces some synchronization overhead. In this regard, we may conclude that it is very important to arrive at the correct balance between load balancing and synchronization overhead for the choice of the hash-function. In this work (chapter 9) it has been decided to focus on a random hash-function, however it would be also interesting to try out other techniques and their effect on planner performance.

For instance, a hash function can sort out motion primitives according to their length and distribute them between threads. Since every pack of successors have trajectories of distinct length, the load balancing would be preserved, while search space is distributed.

Also, there is a possibility to speedup hash-distributed parallelization by utilizing buffers during node distribution, similarly to chapter 8 . Instead of sending nodes one by one to other threads with many locking requests, we can introduce small node buffers for each thread we are sending nodes to. Each buffer will be added to the private OPEN list as a chunk, which requires less locking operations.

11.2.3. Multi-Agent Planning

Introduced two-agent planner (chapter 6) theoretically could be extended to an arbitrary number of agents. One of the advantages of such cooperative planner is it's completeness (eventually it finds an optimal path for each agent). However, search tree branching factor explodes exponentially with every new agent added to the system (section 4.2), thus practical implementation will most likely be infeasible.

Nevertheless, there are ways to reduce a tree branching factor by introducing early node assessment before generating successors. Something similar was done in chapter 6 when collision checking of individual trajectories was done before generation of combined states. Such technique can be extended so that individual successors are discarded according to some other parameters. Together with improved heuristic-obstacle mapping such strategy would allow having a number of agents in the system while still preserving required performance.

Appendix

12. Appendix

This chapter contains most important code listings (C++) from the Dominion project parallel planners implementation.

12.1. Main Function

Parallel strategy depends on the planner instance used by the main function (defined as a class member variable).

```
// Main function (re-called everytime the search is completed)
void DCooperativePlannerTest::run(void)
{
      // Threads initialization
      Timer timer;
      long long startTime;
      pthread_t *thread =
           (pthread_t *) calloc(NUM_THREADS, sizeof(*thread));
      struct pthread_args *thread_arg =
           (pthread_args *) calloc(NUM_THREADS, sizeof(*thread_arg));

      // Search initialization (including parsing the environment)
      planner->prepareSearch();

      // Set start timer
      startTime = timer.getRawCurrentTime();
      timer.setRawStartTime(startTime);

      // Create and fork threads
      for (unsigned int i = 0; i < NUM_THREADS; i++)
      {
           thread_arg[i].threadID = i;
           thread_arg[i].startTime = startTime;
           thread_arg[i].planner = planner;
           pthread_create(thread + i,
                          NULL,
                          &perform_search_par,
                          thread_arg + i);
      }
```

```
    // Join threads
    for (unsigned int i = 0; i < NUM_THREADS; i++)
    {
        pthread_join(thread[i], NULL);
    }

    // Stop timer
    timer.stop();

    // Free thread variables
    free(thread);
    free(thread_arg);

    /*
    Data gathering and plotting ...
    */

    // Resset before next search starts
    resetSearchParams();
}
```

12.2. Thread Function

A function executed by all threads. Called from the main function.

```
// Thread variables
struct pthread_args
{
    unsigned int threadID;
    long long startTime;
    Planner *planner;
};

// Thread function
void *perform_search_par(void *ptr)
{
    // Get thread local variables
    Timer timer;
    struct pthread_args *args = (pthread_args *) ptr;
    Planner *planner = args->planner;
    unsigned int id = args->threadID;

    // Set start time (identical for each thread)
    timer.setRawStartTime(args->startTime);
```

```
    // Perform search
    while(timer.getElapsedTimeFromStartSeconds() < SEARCH_END_TIME)
    {
        planner->executeSearchStep(id);
    }

    return NULL;
}
```

12.3. Root Node Generation

Generating root node by getting parameters from environment representation.

```
// Generate root
NodeFactory::child_iterator* getRootNodes()
{
    float tM, xM, yM, vM, ayM, thetaM;
    float tS, xS, yS, vS, ayS, thetaS;

    // Master vehicle parameters
    tM = (float)(nodeFactory->envRep->VEH_T_GPS);
    xM = (float)(nodeFactory->envRep->VEH_X_GPS);
    yM = (float)(nodeFactory->envRep->VEH_Y_GPS);
    vM = (float)(nodeFactory->envRep->VEH_V_ABS);
    ayM = (float)(nodeFactory->envRep->VEH_AY_GPS);
    thetaM = (float)(nodeFactory->envRep->VEH_HEADING_GPS);

    // Slave vehicle parameters
    tS = (float)(nodeFactory->envRep->VEH2_T_GPS);
    xS = (float)(nodeFactory->envRep->VEH2_X_GPS);
    yS = (float)(nodeFactory->envRep->VEH2_Y_GPS);
    vS = (float)(nodeFactory->envRep->VEH2_V_ABS);
    ayS = (float)(nodeFactory->envRep->VEH2_AY_GPS);
    thetaS = (float)(nodeFactory->envRep->VEH2_HEADING_GPS);

    // Fit analog coordinate values to the trajectory set grid
    findClosestGraphValue(&vM, &ayM);
    findClosestGraphValue(&vS, &ayS);

    return new root_coop_child_iterator(tM, xM, yM, thetaM, vM, ayM,
                                        tS, xS, yS, thetaS, vS, ayS);
}
```

12.4. Search Step

Called by each thread.

```
// Performs one search step (identical for both parallel strategies)
inline void AWAstarPA::executeSearchStep(unsigned int threadID)
{
      // Evaluates a node with the lowest cost
      expand(threadID);

      // Generates successors
      while (currentExpansion[threadID])
      {
           explore(threadID);
      }
}
```

12.4.1. Expand

Hash-distributed strategy expand step.

```
// Generatin successors
inline void AWAstarHDA::expand(unsigned int threadID)
{
      // Lock "pop" operation of the local open list
      Node* parent;
      pthread_mutex_lock(&mutex_open[threadID]);
      if (!localOPEN[threadID]->ol_isEmpty())
      {
           parent = localOPEN[threadID]->ol_pop();
           pthread_mutex_unlock(&mutex_open[threadID]);
      }
      else
      {
           pthread_mutex_unlock(&mutex_open[threadID]);
           if(exploration_count[threadID] > 1)
           threadFinished[threadID] = true;
           return;
      }

      // Check if current cost is less then the upper bound
      if(parent->getFOpt() >= fMax)
      {
           explorations_discarded_awabound[threadID] += 1;
           nodeManagement->unreserve(parent);
      }
```

```
else
{
    // Mutual collision detection
    if(!parent->isAssess2Done())
    {
        costStrategy->assess2(parent, threadID);
        exploration_count[threadID] += 1;
    }

    // If not collision free, discard the node
    if(!parent->isValid())
    {
        invalid_edge_count[threadID] += 1;
        localDISCARDED[threadID]->cl_add(parent);
        nodeManagement->unreserve(parent);
    }
    else
    {
        if(parent->isGoalNode())
        {
            // Add solution node to CLOSED
            localCLOSED[threadID]->cl_add(parent);

            // Lock shared variables update
            pthread_mutex_lock(&mutex_solution);

            fMax = parent->getFOpt();
            SOLUTION->cl_add(parent);
            costStrategy->adaptStrategy_solutionFound(parent,
            f_uninflated_min);

            // Adapt strategy
            if(useAdaptiveCostStrategy)
            {
                for (int i = 0; i < numThreads; i++)
                {
                    // Lock local frontier
                    pthread_mutex_lock(&mutex_open[i]);
                    for(OpenList::iterator* it =
                    localOPEN[i]->items(); it->hasNext())
                    {
                        costStrategy->reAssess(it->next(),
                                useMultiSetStrategy);
                    }
                    localOPEN[i]->reorder();
                    pthread_mutex_unlock(&mutex_open[i]);
```

```
                    }
                }
                pthread_mutex_unlock(&mutex_solution);
            }
            else
            {
                Node* previous =
                localCLOSED[threadID]->getPreviousExploration(parent);
                if( previous
                && previous->getFOpt() <= parent->getFOpt())
                {
                    re_explorations_discarded[threadID] += 1;
                    nodeManagement->unreserve(parent);
                }
                else
                {
                    if( !localOPEN[threadID]->ol_isEmpty()
                    && parent->getF()
                    > localOPEN[threadID]->ol_peek()->getF())
                    {
                        // Lock "add" operation of the local open list
                        pthread_mutex_lock(&mutex_open[threadID]);
                        localOPEN[threadID]->ol_add(parent);
                        pthread_mutex_unlock(&mutex_open[threadID]);
                    }
                    else
                    {
                        if(previous)
                            re_exploration_count[threadID] += 1;
                        else
                            exploration_count[threadID] += 1;

                        // Generate successors
                        localCLOSED[threadID]->cl_add(parent);
                        if(currentExpansion[threadID]!= 0)
                            delete currentExpansion[threadID];
                        currentExpansion[threadID] =
                        samplingStrategy->getChildren(parent, threadID);
                    }
                }
            }
        }
    }
}
```

12.4.2. Explore

Hash-distributed strategy explore step.

```
// Evaluating successors
inline void AWAstarHDA::explore(unsigned int threadID)
{
      if(currentExpansion[threadID]->hasNext())
      {
           // Lock memory reservation
           pthread_mutex_lock(&mutex_reserve);
           Node* child = nodeManagement->reserve();
           updateMaxHeapSize(threadID);
           pthread_mutex_unlock(&mutex_reserve);
           currentExpansion[threadID]->createNext(child);
           costStrategy->assess1(child, useMultiSetStrategy, threadID);

           if(!child->isValid())
           {
                localDISCARDED[threadID]->cl_add(child);
                invalid_edge_count[threadID] += 1;
                return;
           }

           if(child->getFOpt() >= fMax || !child->isValid())
           {
                nodeManagement->unreserve(child);
                if(child->getFOpt() >= fMax)
                {
                     explorations_discarded_awabound[threadID] += 1;
                }
                if(!child->isValid())
                {
                     invalid_edge_count[threadID] += 1;
                }
           }
           else
           {
                // Lock uninflated cost update
                pthread_mutex_lock(&mutex_uninflated);
                if(!child->isGoalNode())
                {
                    setFUninflatedMin(min(f_uninflated_min, child->getFOpt()));
                }
                pthread_mutex_unlock(&mutex_uninflated);
```

```
                // Get send ID
                unsigned int sendID = rand() % numThreads;

                // Add to the local list according to send ID
                pthread_mutex_lock(&mutex_open[sendID]);
                localOPEN[sendID]->ol_add(child);
                pthread_mutex_unlock(&mutex_open[sendID]);
            }
        }
        else
        {
            // Reset current expansion and children counter
            currentExpansion[threadID] = 0;
            children_count[threadID] = 0;
        }
}
```

12.5. Node Generation

Successor states generation process.

```
// Create CoopNode object based on motion primitives sets
void createNext(Node *freeMemory)
{
      // Create a cooperative node
      CoopNode *coopFreeMemory = (CoopNode *) freeMemory;
      *coopFreeMemory = CoopNode(parent,
                                 parent->getMasterVehNode(),
                                 *masterIterator,
                                 parent->getSlaveVehNode(),
                                 *slaveIterator);

      // Increment individual iterators
      masterIterator++;
      if (masterIterator == masterTrajs.end())
      {
          slaveIterator++;
          if(slaveIterator != slaveTrajs.end())
          {
              masterIterator = masterTrajs.begin();
          }
      }
}
```

```
// Returns cooperative children iterator
virtual coop_child_iterator* getChildren(Node* parentNode,
                                         int threadID = 0)
{
      CoopNode *coopParent = (CoopNode *) parentNode;
      Node6d *master = coopParent->getMasterVehNode();
      Node6d *slave = coopParent->getSlaveVehNode();

      // Get super sets for both vehicles
      TrajectorySuperSet *masterVehSet =
             setMap->getTrajectorySuperSet(master->getV(),
                                         master->getAy());
      TrajectorySuperSet *slaveVehSet =
              setMap->getTrajectorySuperSet(slave->getV(),
                                         slave->getAy());

      return new coop_child_iterator(coopParent,
                                     masterVehSet,
                                     slaveVehSet,
                                     envRep);
}
```

12.6. Node Assessment

Evaluation of the states.

```
// Evaluate node
void assess1(Node *node,
              bool useMultiSetStrategy,
              unsigned int obbID = 0)
{
      // Extract vehicle nodes
      CoopNode *coopNode = (CoopNode *) node;
      CoopNode *coopParent = (CoopNode *) coopNode->getParent();
      Node6d *master = coopNode->getMasterVehNode();
      Node6d *slave = coopNode->getSlaveVehNode();

      master->setValid(true);
      slave->setValid(true);
      node->setValid(true);
```

```
        // Execute early assessment for each vehicle
        strategyM->assess1(master, useMultiSetStrategy);
        strategyS->assess1(slave, useMultiSetStrategy);

        // Set cooperative node as a goal node
        if (master->isGoalNode() && slave->isGoalNode())
        {
            node->setGoalNode(true);
        }

        // Set node parameters
        node->setC((master->getC() + slave->getC()) / 2.0);
        node->setG((master->getG() + slave->getG()) / 2.0);
        node->setH((master->getH() + slave->getH()) / 2.0);
        node->setFOpt((master->getFOpt() + slave->getFOpt()) / 2.0);
        node->setF((master->getF() + slave->getF()) / 2.0);
        node->setAssess1Done(true);
}



// Checks for trajectories intersection
void assess2(Node *node, unsigned int obbID = 0)
{
        // Initialization
        float tStartFixed, tEndFixed, tStartFloating, tEndFloating;
        Node6d *fixed, *floating;

        // Extract vehicle nodes
        CoopNode *coopNode = (CoopNode *) node;
        CoopNode *coopParent = (CoopNode *) coopNode->getParent();
        Node6d *master = (Node6d *) coopNode->getMasterVehNode();
        Node6d *slave = (Node6d *) coopNode->getSlaveVehNode();
        Node6d *parentM = (Node6d *)(master->getParent());
        Node6d *parentS = (Node6d *)(slave->getParent());

        // Check if not root
        if (coopParent != 0)
        {
            // Assign the node to be checked against
            if (parentM->getT() < parentS->getT())
            {
                fixed = master;
                floating = slave;
            }
            else
```

```
            {
                fixed = slave;
                floating = master;
            }

            // Get initial time intervals
            tStartFixed = ((Node6d *) fixed->getParent())->getT();
            tEndFixed = fixed->getT();
            tStartFloating = ((Node6d *) floating->getParent())->getT();
            tEndFloating = floating->getT();

            // Moving towards the root
            while (tStartFixed < tEndFloating)
            {
                // Check if time intervals are intersecting
                if ((tEndFixed > tStartFloating)
                && (tStartFixed < tEndFloating))
                {
                    if (trajectoriesAreIntersecting(fixed, floating, obbID))
                    {
                        node->setValid(false);
                        break;
                    }
                }

                // Move one node back
                floating = (Node6d *) floating->getParent();
                if (floating->getParent() != 0)
                {
                    // Reset start and end time
                    tStartFloating =
                    ((Node6d *) floating->getParent())->getT();
                    tEndFloating = floating->getT();
                }
                else
                {
                    break;
                }
            }

            node->setAssess2Done(true);
        }
}
```

Bibliography

[1] Blaise Barney. Message passing interface introduction. https://computing.llnl.gov/tutorials/mpi/#What.

[2] Blaise Barney. Posix threades introduction. https://computing.llnl.gov/tutorials/pthreads/.

[3] Ethan Burns, Seth Lemons, and Rong Zhou. Best-first heuristic search for multi-core machines, 2009.

[4] Ethan Burns, Sofia Lemons, Wheeler Ruml, and Rong Zhou. Parallel best-first search: The role of abstraction. In *Abstraction, Reformulation, and Approximation, Papers from the 2010 AAAI Workshop, Atlanta, Georgia, USA, July 12, 2010*, 2010.

[5] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[6] Joao Pedro de Campos Salvado. Contingency planning for automated vehicles in urban traffic. Master's thesis, Tecnico Lisboa, November 2015.

[7] Mathijs de Weerdt and Brad Clement. Introduction to planning in multiagent systems. *Multiagent Grid Syst.*, 5(4):345–355, December 2009.

[8] Matthew Evett, Ambuj Mahanti, Dana Nau, James Hendler, and James Hendler. PRA*: Massively parallel heuristic search. *Journal of Parallel and Distributed Computing*, 25:133–143, 1995.

[9] David Ferguson , Maxim Likhachev, and Anthony (Tony) Stentz . A guide to heuristic-based path planning. In *Proceedings of the International Workshop on Planning under Uncertainty for Autonomous Systems, International Conference on Automated Planning and Scheduling (ICAPS)*, June 2005.

[10] C. Gackstatter, S. Thomas, Dr. P. Heinemann, Prof Gudrun Klinker, Audi Electronics Venture Gmbh, Leibniz Universitat Hannover, and Technische Universitat Muenchen. Stable road lane model based on clothoids, 2006.

[11] Stefan K. Gehrig and Fridtjof Stein. Cartography and dead reckoning using stereo vision for an autonomous car. In *ICIP (4)*, pages 30–34, 1999.

[12] S. Gottschalk, M. C. Lin, and D. Manocha. OBB tree: A hierarchical structure for rapid interference detection, 1996.

[13] Ariel Felner Guni Sharon, Roni Stern and Nathan Sturtevant. Meta-agent conflict-based search for optimal multi-agent path finding. In *to appear in SoCS*, 2012.

[14] Eric A. Hansen and Rong Zhou. Anytime heuristic search. *Journal of Artificial Intelligence Research (JAIR)*, 28:267–297, 2007.

[15] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science, and Cybernetics*, SSC-4(2):100–107, 1968.

[16] Daniel Hess, Matthias Althoff, and Thomas Sattel. Formal verification of maneuver automata for parameterized motion primitives, 2014.

[17] Thomas M. Howard, Colin J. Green, Alonzo Kelly, and Dave Ferguson. State space sampling of feasible motions for high performance mobile robot navigation in complex environments. *Journal of Field Robotics*, pages 325–345, 2008.

[18] Keki B. Irani and Yifong Shih. Parallel A* and AO* algorithms: An optimality criterion and performance evaluation. In *International Conference on Parallel Processing, ICPP'86, University Park, PA, USA, August 1986.*, pages 274–277, 1986.

[19] Akihiro Kishimoto and et al. Scalable, parallel best-first search for optimal sequential planning, 2009.

[20] Akihiro Kishimoto, Alex Fukunaga, and Adi Botea. Evaluation of a simple, scalable, parallel best-first search strategy, 2012.

[21] Stepan Kopriva, David Sislak, and Michal Pechoucek. Towards parallel real-time trajectory planning, 2009.

[22] Bob Kuhn and Paul Petersen. Openmp versus threading in c/c++, 1999.

[23] Steven M LaValle. Planning algorithms, 2004.

[24] Maxim Likhachev, Geoff Gordon, and Sebastian Thrun. ARA*: Anytime A* with provable bounds on sub-optimality. In *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 16: PROCEEDINGS OF THE 2003 CONFERENCE (NIPS-03)*. MIT Press, 2004.

[25] Raz Nissim and Ronen Brafman. Distributed heuristic forward search for multi-agent planning, 2014.

[26] Raz Nissim and Ronen I. Brafman. Distributed heuristic forward search for multi-agent planning. *J. Artif. Intell. Res. (JAIR)*, 51:293–332, 2014.

[27] Hans B. Pacejka. *Tire and vehicle dynamics*. SAE, 2006.

[28] Mike Phillips, Maxim Likhachev, and Sven Koenig. PA*SE: Parallel A* for slow expansions, 2014.

[29] Mikhail Pivtoraiko. *Differentially Constrained Motion Planning with State Lattice Motion Primitives*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, February 2012.

[30] Mikhail Pivtoraiko and Alonzo Kelly . Efficient constrained path planning via search in state lattices. In *The 8th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, September 2005.

[31] Mikhail Pivtoraiko and Alonzo Kelly . Generating near minimal spanning control sets for constrained motion planning in discrete state spaces. In *Proceedings of the 2005 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '05)*, pages 3231 – 3237, August 2005.

[32] Joel Yliluoma. Openmp guide. http://bisqwit.iki.fi/story/howto/openmp/.

[33] Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *In National Conference on Artificial Intelligence (AAAI)*, pages 1217–1222, 2007.