AIPATH: MOVEMENT PLANNING IN PHYSICALLY CONSTRAINED DOMAINS

Benjamin Schnieders

Master Thesis DKE 14-09

Thesis submitted in partial fulfilment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Faculty of Humanities and Sciences
of Maastricht University

Thesis committee:

Dr. Mark H.M. Winands
Prof. Dr. Gerhard Weiss
Dr. Marc Lanctot

Maastricht University
Faculty of Humanities and Sciences
Department of Knowledge Engineering
Master Artificial Intelligence
July 1st, 2014

Preface

This master thesis is written in partial fulfillment of the requirements for the degree of Master of Science of Artificial Intelligence at the Faculty of Humanities and Sciences of Maastricht University. The thesis covers the fields of path and trajectory planning in computer games and simulation. A novel approach to movement planning for computer games, aiPath, is presented.

I would like to thank the whole DKE staff for the ongoing support throughout the study, most of all my thesis supervisor Dr. Mark Winands for his guidance during the thesis research. Thanks also to Prof. Dr. Gerhard Weiss and Dr. Marc Lanctot for assessing my thesis. Special thanks also to my mother, who unfortunately passed away in the course of this study, for her support.

Benjamin Schnieders
June 2014

Summary

While pathfinding is an extensively studied field in computer games, planning the actual movements to be performed while following the path is an open field to more in-depth research. Different vehicles may require a different set of movement actions and may be subject to different physical limitations. Following a path that was planned without considering optimal action selection and physical possibility might produce visible movement artifacts, which in turn can be noticed by human players. Movement artifacts may include abrupt movement and stopping, taking visibly too narrow corners and "glitching", i.e., performing sudden, unreasonable movement. Detecting irrational, impossible or simply uncanny behavior easily triggers a repulsive reaction in the player, destroying believability of the game world, and with it the created immersion. Believing in the game world and immersing into it is an important concept in computer gaming; without it many games would lose their entertainment value. Minimizing movement artifacts and providing life-like AI movement is thus a goal for computer games. Physical movement planning is a technique to plan an optimal path while regarding physical possibility of actions at all times, annihilating movement anomalies.

The problem statement is given as "*In order to ensure believability of characters in a simulated environment, current pathfinding techniques have to be modified to incorporate planning with physical limitations of the controlled vehicle*". The two research questions answered in the course of the research are "*Can physical movement planning approaches provide additional depth in gaming by providing more realistic movement?*", and "*Can pathfinding techniques be enhanced to provide more realistic movement, while still being computationally feasible?*" by employing physical movement planning in a computer game.

This thesis research examines the use of AI search techniques to plan a path in the physical state space of a vehicle. Different discretization strategies are proposed for every state space variable, and different state space representations are proposed for a variety of vehicles, including pedestrians and cars. Two search algorithms are collaborating to find a path in the discretized physical state space. An initial probing search step makes use of a greedy best-first search in order to quickly expand nodes towards the goal. If the probing step fails to find a path, A* is used instead, utilizing the already expanded nodes as a vantage starting configuration. Both algorithms make use of a heuristic evaluation function guiding them towards the goal. The proposed heuristic evaluation function incorporates Pinter's method to calculate shortest trajectories for vehicles with a turning radius, but is extended to regard velocity and acceleration limits to calculate the traversal time.

The resulting approach is called aiPath. The search produces physically possible, shortest paths, because only physically possible actions are considered in every node of the state space. Experiments testing the quality of the heuristic evaluation function and the generated paths are performed and their results analyzed. An integration into a game making heavy use of open street traffic, Emergency 5, provides detailed results in a real-life scenario. aiPath is used as a local planner for EM5, improving curve shapes and ensuring physical possibility of the paths generated by the EM5 pathfinding system.

The integration into Emergency 5, as well as comparisons to other approaches and qualitative analysis of the results answer the research questions.

List of Terms

A* is a heuristic search algorithm to find the shortest path on a graph, see also Subsection 2.1.1.

Dijkstra's algorithm is an algorithm to perform an uninformed search on a graph, see also Section 2.1.

EM5: short for Emergency 5, an open-world disaster simulation game presented in Section 1.3.

Euclidean Distance is the distance between two points as obtained with a ruler, or "as the crow flies".

Heuristic or a **heuristic evaluation function**, is a function providing an estimate of expected costs. Section 5.4 provides more detail.

IDA* or **Iterative Deepening A*** is an A* variant designed to use lower amounts of memory by iteratively deepening the search horizon.

Open Street Network is a connected set of roads, like commonly given in urban situations, as opposed to racetracks with just one possible traversal route.

Path Following describes a procedure invoked every simulation step to move an entity along a path. Different levels of realism are possible.

Path Quality is a rather subjective measure of path realism, as opposed to the objective measure of path length.

Pathfinding is a technique to obtain a possible sequence of waypoints in order to reach a goal. Typically, this is achieved by graph search algorithms.

Search Graph is typically an abstraction of a more complicated space to enable efficient searching.

State Space is a description for the set of all possible states an object can have.

Uncanny Valley or the **Uncanny Valley Effect** is a description for a sudden reduction in acceptance towards simulated humans if they behave quite, but not exactly like humans [36].

Contents

Chapter 1

Introduction

Movement planning is a technique to schedule movement actions in a way so that the resulting movement fulfills certain constraints, regarding physical possibility or aesthetic criteria. This thesis presents aiPath*, an approach to plan shortest paths that do not violate physical constraints. It uses the well-established search algorithm A* to find a shortest path in the physically constrained state space of a vehicle.*
Video games, becoming increasingly more realistic simulations of a wide array of real-life scenarios, provide a growing domain for AI research, most importantly intelligent movement [34]. Physically correct movement helps preserving immersion in simulated environments and creates feasible paths in real-world situations. This chapter provides an introduction to the topic of movement planning and shows its use in video game AI and real world applications.

Chapter contents: Introduction to the topic of constrained movement planning and areas of use.

1.1 Pathfinding and Movement Planning

This section introduces the key techniques in the field of pathfinding and movement planning. It discusses the differences between finding the shortest path and finding the optimal sequence of actions, and why the latter provides a significant improvement in observed path quality.

1.1.1 Pathfinding

Pathfinding is a common AI technique to find the shortest route between two nodes, a start node and the goal node, on a graph representing a physical environment [44]. In order to find the shortest route, costs are assigned to edges of the graph, and search algorithms are subsequently used to find a path from the start to the goal node. A path consists of a list of nodes that have to be visited in order to reach the goal. Notable search algorithms are (depth-limited) depth first search, Dijkstra's algorithm [16] and heuristic search algorithms such as A* [23] and IDA* [29]. To find a path in a physical world representation, the physical space is converted to a search graph, which is then searched on; nodes on the resulting path are re-translated into the spatial world representation. Pathfinding is widely used in computer games AI, producing paths for user controlled units or AI opponents [34]. Another domain making heavy use of pathfinding is movement in robotics [10]. However, pathfinding is not restricted to work on spatial worlds. Other areas of use include packet routing in networks, where the search graph corresponds directly to the network structure.

1.1.2 Movement Planning

Pathfinding only provides a list of nodes to be traveled in order to reach a goal, but no detailed instructions what actions to perform in between each two nodes. A separate path following algorithm can be used in order to convert a path into a sequence of movement instructions. Alternatively, a movement planning algorithm can immediately search on the space of available actions, producing a sequence of actions required to achieve the goal. If at any time during path traversal, multiple different actions with varying effects are available, some kind of movement planning is required to appropriately select actions. In sophisticated computer game movement procedures, ducking and jumping might be planned at certain points on the path. For a parking assistant, selecting a gear and steering angle are actions that have to be planned in order to maneuver a car into a parking space. While selecting a movement action during path traversal is possible, an optimal sequence of actions is only guaranteed if the whole path is considered.

1.1.3 Motion-Constrained Path Planning

While movement planning already provides the appropriate action to be executed at any given point in time, until the goal state is reached, planning actions based upon an already planned spatial path may pose certain problems. For example, the shortest path between two points in space might be physically impossible for a vehicle to follow. Figure 1.1 provides two examples of a shortest path that is impossible for the vehicle to follow, and one feasible shortest path. Post-processing techniques can improve the feasibility of spatial paths, but only up to a certain extent [34]. In order to find the shortest path regarding all physical limitations of a vehicle, a movement planning algorithm has to find the shortest spatial path, such that each two consecutive nodes are physically possible to traverse. This process is also called physical path planning, or motion-constrained path planning, if arbitrary constraints are to be met.

Physical path planning is widely used in robotic applications [39]. Computer games, however, often strip this expensive extra realism in favor of computationally cheaper approaches, taking into account the risk that the resulting behavior might be suboptimal. Paths are often post-processed in games; giving them a mere appearance of physical correctness [8]. Nevertheless, post-processing cannot fix physically infeasible paths, and thus players can notice these fallacies.

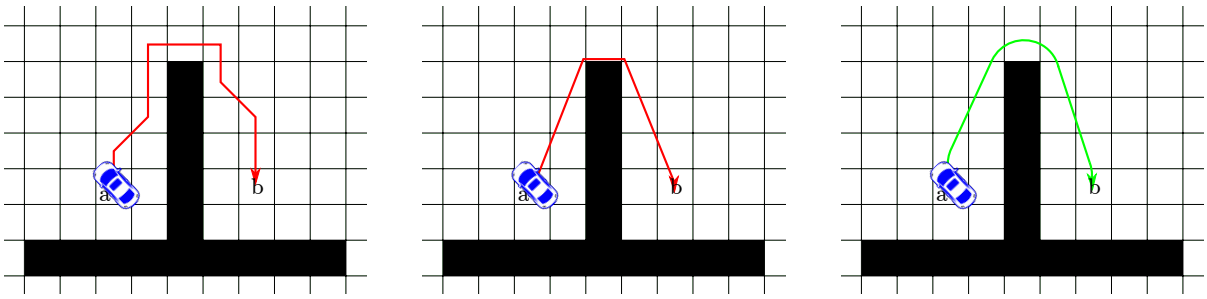

Figure 1.1: A vehicle planning to move from a to b. Left: Shortest path on a grid. Middle: Shortest path after path smoothing or obtained through any-angle path planning. Right: The only feasible path concerning the physical restrictions of the vehicle.

1.2 Game AI

This section distinguishes game AI designed to be a strong opponent from AI that was specifically designed to be entertaining. It furthermore focuses on movement in games, and provides an overview about game AI techniques commonly used for navigation, pathfinding and path following. It discusses the importance of AI behavior which is understandable by the player for an immersive gaming experience.

In a certain subset of games, an AI opponent is desired to play the game as challenging as possible. These are typically games in which a human player can outperform state-of-the-art AI, such as Go [51].

In other cases, such as when an AI is capable of outplaying humans by far, an opponent mimicking the skill level of the human player is desired [30]. Many video games are designed to tell a story rather than modeling an intelligent opponent. In these games, AI functions to bring the simulated environment to life [37][47].

1.2.1 Video Game AI Goals

Video game AI has a main purpose: to entertain the player. A key component in entertainment is generating a believable game world. In order to achieve more realism, sophisticated graphics and physics engines are used. Developments in CPU architecture, away from monolithic, high performance CPU cores towards multi-core machines and their intrinsic capability of performing multiple different tasks at once, have led to an increase of available computational resources for AI as well. AI aids generating an immersive gaming experience in two ways. For once, more believable computer opponents or allies can be simulated. Secondly, AI is used to provide seamless background action, making a game world appear alive. Players can easily identify with the character they are playing, and thus, they provide an overall acceptance of game graphics; i.e., a computer controlled character looking like a human controlled character might be accepted as the character of another player, or, more generally, as an authentic piece of the world. If however small, but inexplicable, faults in the behavior of a computer controlled character are noticed, suspicion rises and the AI character quickly falls victim to the uncanny valley effect, which triggers a rejective attitude in the player [4][36]. Believability of the game world and immersion are quickly destroyed, and the gaming experience suffers [12]. Serious games, while not designed to be entertaining, but rather to convincingly present real-life scenarios, rely on the fact that players have to identify with characters in the simulated environment, and thus take playing the game seriously. Destroying the immersion in a simulated environment might ruin its applicability as a serious game. As most games make use of some kind of motion, generating believable movement is an important prerequisite for generating believable game worlds. AI character movement should ideally appear equal to human controlled character movement. Realistic AI movement is therefore a key component for preserving overall believable characters and games [50].

1.2.2 Movement Techniques in Games

Typically, computer games make use of spatial pathfinding to produce paths. These paths are then commonly postprocessed by algorithms like path smoothing to look more realistic [8]. In order to achieve the AI agent actually following the path, another post-processing step transforms the sequence of waypoints to a sequence of actions. This transformation is typically done just-in-time during path execution, and is called path following. Path following is commonly combined with other, collision-preventing or purely cinematic navigation approaches to a dedicated steering system, which weights different desires and produces an actual movement [34]. Naive implementations, like a simple interpolation in between waypoints, make use of a stateless path following algorithm, which may take some, but certainly not all physical limitations of a vehicle into account. More sophisticated steering approaches plan ahead on the path, calculating the optimal speed at any time, or giving at least the impression, like the "chase the rabbit" path following variant, shown in Figure 1.2, as described by Millington [34]. However, all these systems fail to accurately incorporate physical limitations, as this cannot be achieved by a post processing step. It is also to be noted that the most commonly described approaches are only applicable for ground based vehicles. Figure 1.3 displays visual path improvement by post-processing, but the technique cannot produce a physically possible path for the vehicle.

A certain amount or error tolerance is innately present in computer games for their entertainment value. This also allows players and AI to reach goal points within a certain radius, and still be considered to have arrived at the specified location exactly. Tolerances like these have to be known to the AI in order to exploit them the same way a human player would do. However, tolerances and their subsequently introduced errors may also harm the believability and integrity of the game world. For serious or simulation type games, tolerances should be kept as low as possible. Providing state-of-the-art AI techniques in games with acceptable resource consumption may bridge the gap between academic AI and AI in games [6].

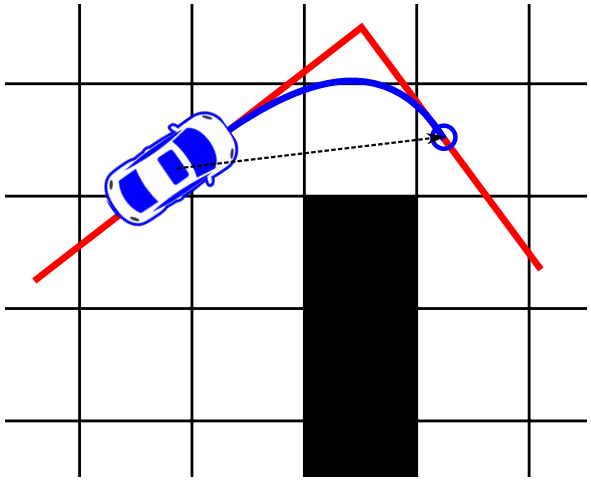

Figure 1.2: Chasing the rabbit: The vehicle is attracted towards the blue circle ahead on the path. The originally planned, red, path is smoothed and a trajectory like the blue one is generated.

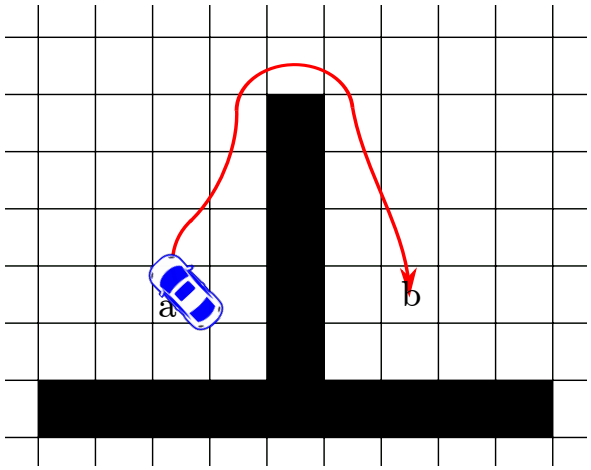

Figure 1.3: Path post-processing can generate more visually pleasing paths, however, these approaches can neither guarantee shortest paths nor physical correctness.

1.3 Emergency 5

Emergency 5, abbreviated EM5, is an upcoming title of German game development studio Promotion Software, and the latest successor in the Emergency series [46]. The gameplay in the Emergency series revolves around micromanaging emergency services in different catastrophe and accident situations, for example distribute firefighters over several burning buildings and assigning rescue personnel to individual civilian units. While EM5 is meant to be a game for entertainment purposes, it is located on the boundary of serious games and disaster simulation, with high-fidelity graphics and realistic level design to provide an immersive experience. EM5 features large, open world street maps resembling major German cities. Playing the game, the user sends individual commands to rescue personnel, which the game AI in turn realizes by performing movement, playing animations and modifying the game world. Another essential task for EM5s AI is providing ambient entertainment. The game world should be filled with pedestrians, commuters and bystanders, all of which should behave in a natural way. Figures 1.4 and 1.5 provide examples of ambient behavior generated by the AI. Furthermore, vehicular traffic should populate roads in a realistic manner. For vehicular traffic, this means following traffic laws, avoiding accidents and, in general, submit to physical constraints that are not enforced, yet should be followed. As the gameplay features disaster scenarios, blocked roads or even whole city blocks should be circumnavigated believably, that is, vehicles should not appear to be all aware of the whole map, but merely a smaller region around them.

Figure 1.4: A crowd of bystanders gathering near an accident site in EM5. The debugging grid to measure in-game distances is activated.

Figure 1.5: AI providing ambient micro-actions to simulate a living game world, like moving birds, pedestrians using cell phones and bike unlocking before usage.

EM5s vehicular movement system makes use of lane graphs, which represent the center lines of each lane on the street network. Figure 1.6 illustrates an example of lane center graphs in an intersection. The pathfinding system plans paths on this representation, providing a sequence of straight and circular segments. While following this path, a steering component ensures that the vehicle moves with the maximally sustainable speed, but still respects speed limitations from traffic laws. Collision prevention and priority rules are partially implemented through a *reservation system*, exclusively or collaboratively reserving lane segments for one or more vehicles at a time. Temporarily blocked lanes, e.g. a road segment in use by an emergency vehicle, should not be entered. Planned arrival and departure times are taken into account when reserving lane segments in order to achieve higher realism.

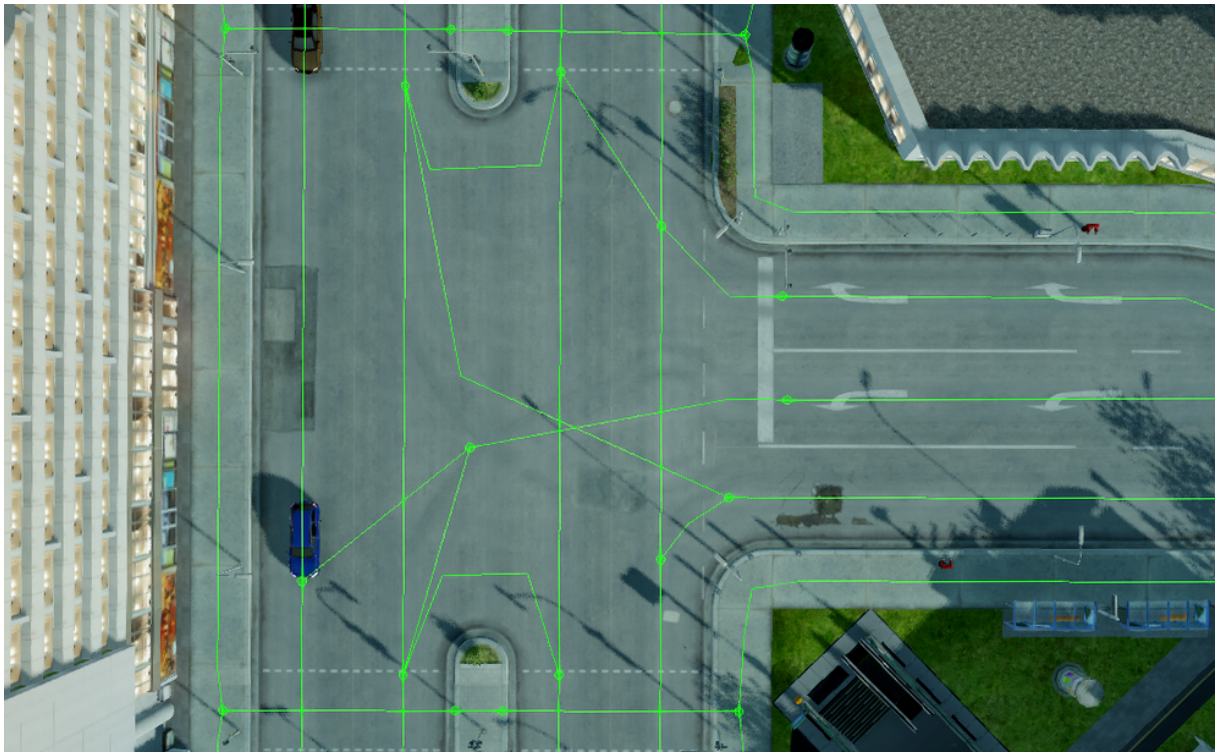

Figure 1.6: Street and sidewalk navigation graph obtained from lane centers. Paths obtained from this structure have to be smoothed in order to look realistic.

1.4 Problem Statement

From above sections one can conclude that game AI techniques, though typically providing adequate level of entertainment, fail to guarantee error free behavior. Users can discover faulty movement easily in many cases, which might disrupt an immersive experience. The problem can be stated as follows:

> *In order to ensure believability of characters in a simulated environment, current pathfinding techniques have to be modified to incorporate planning with physical limitations of the controlled vehicles.*

The main research question to answer is therefore:

> *Can physical movement planning approaches provide additional depth in gaming by providing more realistic movement?*

Additionally, highly computationally expensive, state-of-the-art AI techniques are typically not available in computer games, due to the amount of resources spent for, among others, graphics and physics simulation. This has led to the second research question, concerning the evaluation and optimization of the computational efficiency of the approach.

> *Can pathfinding techniques be enhanced to provide more realistic movement, while still being computationally feasible?*

A method to search physically sound paths in a discretized state space description of the planning vehicle is presented, aiPath. aiPath uses A* in the customary discretized physical state space of the vehicle to find a physically correct path to any goal configuration. The underlying A* algorithm uses a customized heuristic evaluation function to be able to search the state space efficiently. Comparisons of aiPath to other state-of-the-art techniques, like using A* search with different heuristic evaluation functions or using the pathfinding/steering ensemble in EM5, answer the first research question. As EM5 makes heavy use of open street network traffic, an integration of aiPath into EM5 is used to answer the second research question concerning the computational, and thus commercial, viability of the approach.

1.5 Outline

This outline concludes the first chapter of this thesis, indicating the problem statement and exploring the research domain. Chapter 2 provides further, more in-depth information about the field of research, explains algorithms and procedures and provides an overview of comparable work in the area. Chapter 3 presents the state space descriptions aiPath works on. Chapter 4 presents improvements to discretization methods and heuristic evaluation functions, resulting in aiPath, a proposed solution to answer the research questions. Comparisons of aiPath to existing approaches, qualitative and quantitative tests are described in Chapter 5, furthermore, the results to all experiments are presented. Concluding this thesis and answering the research questions, Chapter 6 reflects upon the results achieved during research and analyzes them critically. Finally, further topics of interest and newly exposed research questions are conglomerated.

Chapter 2

Background

This chapter describes techniques evaluated and compared in this thesis. It discusses searching algorithms, discretization measures to convert a continuous domain to a search graph and how physical limitations of vehicles can be mathematically modeled and incorporated into the search algorithm.

Chapter contents: How to search on a graph and how to obtain a search graph from a simulated environment.

2.1 Search

Search is an intensively studied field in AI [44]. This section provides an overview of commonly used search techniques. Most search algorithms operate on a search graph, iteratively expanding child nodes of the currently selected node. Search algorithms return a set of nodes which, if followed one by one, representing a path from the start node to the goal node. Figure 2.1 depicts a search graph with two shortest paths. Simple search algorithms like breadth-first search and depth-first search perform an exhaustive search on the whole graph, however, runtime and memory complexity does not scale well with an increase in path length. Simple search algorithms also merely test for the presence of a path to the goal node. Breadth-first search is guaranteed to find the path with a minimal number of nodes, when every edge has a uniform distance. More sophisticated search algorithms are required to find shortest paths on graphs featuring non-uniform edge distances. To provide a custom distance measure, connections between nodes typically are assigned a specific cost, c, which should be considered during search. A minimum cost path then corresponds to the shortest path using the custom measure. For spatial searches, that is, searches on a graph with nodes representing points in space, Euclidean distance between the nodes is an appropriate cost measure for edges. If traversal speeds should be considered, the traversal time is to be preferred. Most commonly, Dijkstra's algorithm and variants thereof are used to perform point to point searches minimizing costs [16]. If minimal cost paths between one start node and all other nodes in the search graph are desired, the Bellmann-Ford algorithm is preferred, cf. [9]. Dijkstra's algorithm is guaranteed to return the shortest path as long as no negative-weight edges are present in the search graph; the Bellman-Ford Algorithm and the Floyd-Warshall Algorithm can handle negative edge weights as well [9].

2.1.1 Informed Search

All presented search algorithms so far suffer from the naivety to expand all nodes on the graph until the goal node is found. This includes many nodes not even close to the shortest path. In order to reduce the number of node expansions, a heuristic evaluation function can provide an estimated distance from any node to the goal node, h. As the heuristic introduces additional information beyond the realm of the search graph, algorithms making use of it are called informed search algorithms. Using a heuristic evaluation function typically reduces the number of nodes expanded, especially those farther away from the shortest path. An easily obtainable and usually sufficient heuristic in spatial searches is the Euclidean

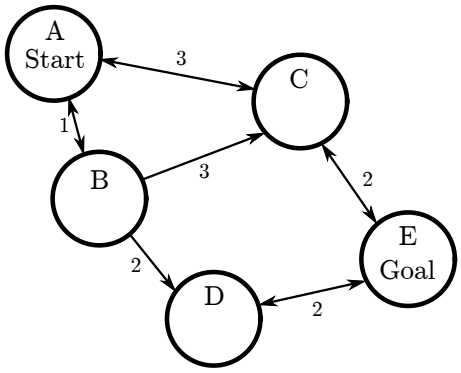

Figure 2.1: A search graph. Edges are weighted by their costs of traversal, leaving $\{A, C, E\}$ and $\{A, B, D, E\}$ as shortest paths from the start to the goal node.

distance between the positions corresponding to the current and goal nodes. A naive search algorithm making use of heuristic evaluation is greedy best-first search, always expanding the child node with lowest cost estimation. Greedy best-first search is very fast, however, depending on the heuristic function used, it may not necessarily return the shortest path, if the heuristic value and true costs diverge at some point [13]. A* is a search algorithm extending Dijkstra's algorithm with a heuristic evaluation function [23]. It is commonly used in video games pathfinding for its performance and simplicity of implementation. A* keeps nodes which are encountered, but not yet expanded, in a structure called *open list*. This structure is sorted by the estimated cost of the whole path, $c + h$. Each iteration, the smallest element e is removed from the open list, all child nodes are added instead, and the original node e is added to the *closed list*, a second container used to avoid re-opening already fully explored nodes. As A* keeps a list of all previously expanded nodes, it may use up a significant amount of memory on larger graphs. Variants like IDA* reduce memory usage in exchange for a higher run-time [29].

In order for A* to be optimal, the used heuristic function has to be *admissible*, i.e., must not overestimate the costs to the goal. Furthermore, *consistency* is a desirable property of the heuristic evaluation function. The consistency constraint is fulfilled if the heuristic score decreases monotonically towards the goal node.

2.2 World Representations

Some worlds are designed in a discrete way. For example, commonly used in education are the following two simple examples; the vacuum cleaner world and the Wumpus-World, from the video game Hunt the Wumpus [44]. As discrete worlds, each possible state can be mapped one-to-one onto nodes on a search graph. Most board games, for example chess and checkers, are set in discrete environments as well, which allow humans and computers to search for strategies and solutions in a more restricted environment. Some video game genres, like turn-based strategy games, closely represent the board game model, as they are modeled to be discrete in space and time as well. A main drawback of discrete game worlds is the lack of realism, as humanly perceivable physical space and time is of continuous nature. Measuring and visualizing continuous attributes on a board game is a difficult task, however, variables can easily be measured and visualized as real valued numbers or charts on a computer screen. This technological advantage allows for the usage of continuous game worlds in computer games.[1] The search techniques evaluated in the previous section only function in discrete domains, or those with a specially featured search graph. This section discusses ways to transform continuous domains to discrete ones by discretization.

[1] While the computer is still a finite machine, even floating point numbers are discrete representations of numbers, however, the discretization is sufficiently fine to be negligible in most cases. Also, a search graph containing all possible floating point values would be infeasibly large.

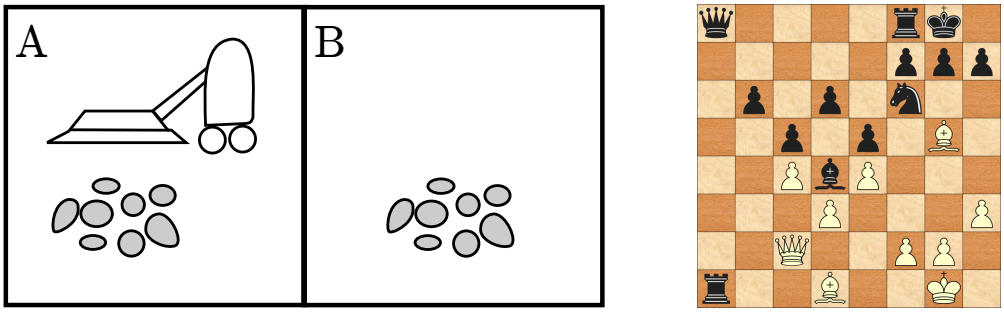

Figure 2.2: Vacuum cleaner world and a chess board: Two examples of discrete game worlds.

2.2.1 Grid Discretization

The possibly easiest method to discretize a continuous domain to a discrete one is done by rounding each point in space towards the closest cell in an evenly spaced grid. Figure 2.3 shows how arbitrary points in space are assigned to discrete cells. A search graph can be obtained from the grid by using the cell center points are node positions in space, as also illustrated in Figure 2.4. Because a grid is evenly spaced, most implementations allow random access on cells. Custom costs can be introduced by weighting cells. Being simple to implement and flexible in usage, grids are commonly used in robotics maneuvering and game world discretization [3][48]. Sparsely filled grids reserve great portions of memory to represent empty cells. If random access is not a necessity, the grid can also be represented by a collection of only the filled cells, reducing memory usage. While grid discretization provides good results describing physical spaces, not all search spaces can be discretized well. If variables in a search space description are mutually dependent, their relationship may be broken by the process of discretization. While the components of position vectors are independent, direction vectors may fulfill additional constraints. For example, a normalized direction vector may not have unit length after discretization any more. While the possible error on an independent variable is at most the width of a grid cell, the error on dependent variables can easily be higher. To avoid unexpected discretization artifacts, it is suggested to break down mutually dependent variables to independent components.

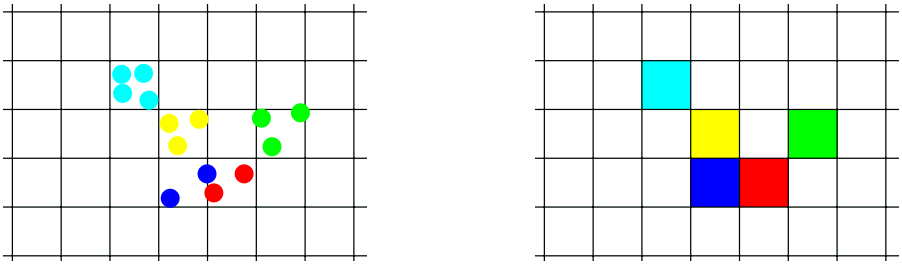

Figure 2.3: A number of points in space assigned to their respective grid cells. Any position is now limited to cell center points. Finer detail is lost.

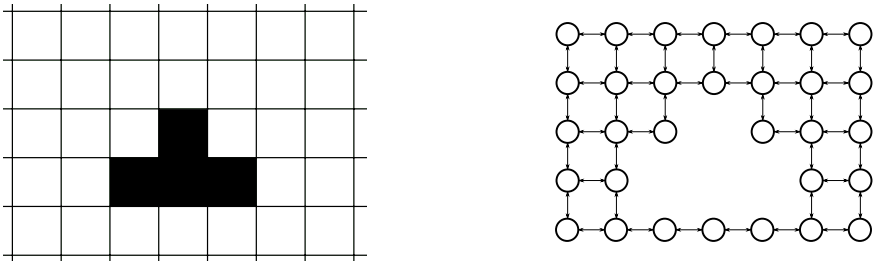

Figure 2.4: Left: A grid with four inaccessible cells. Right: The search graph derived from the grid. All edges are weighted the same.

Remapping Continuous Domains. In certain cases, the world layout is not properly reflected by a search grid. For example, representing a two dimensional ring structure by a two dimensional grid may lead to longer searching times, as heuristics have to work around the occupied area in the middle. Remapping such spaces to more uniform representations before discretization can aid in subsequent path searches [43]. Figure 2.5 presents an example of a layout remapping. Remapping can also reduce empty cell overhead in less densely blocked areas while increasing resolution in regions lacking it.

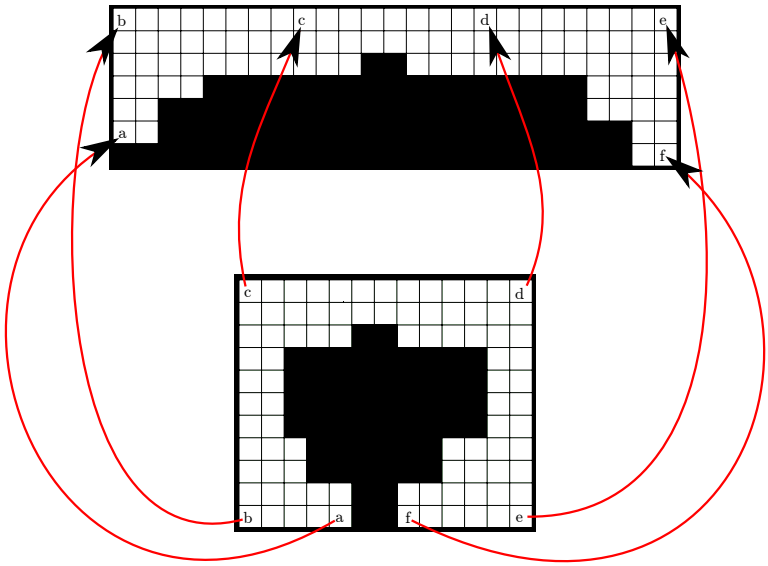

Figure 2.5: Remapping of a grid to better reflect the underlying geometry. Note that the Euclidean distance between a and f is deceivingly low in the original version. The distance between cells a and f in the remapped version models the path costs between them more realistically.

Variable Grid Resolution. For large, unevenly filled grids, investing most of the grid cells just do describe empty space might be considered a waste of memory. Hierarchical fixed-resolution grids provide a viable alternative to more complicated approaches [24]. Variable cell size and geometry can be obtained by using variable resolution techniques such as a kD-Tree. More sophisticated approaches may even make use of decision trees to map high dimensional state spaces efficiently [35].

2.2.2 Navigation Mesh

A navigation mesh, or navmesh for short, is a mesh, commonly triangular and two-dimensional, situated in three dimensional space, providing an abstraction step between the world representation and a search graph. Cost-intensive techniques generate a mesh that fits and follows the environment, leaving holes where insurmountable objects prevent movement. Navigation meshes can be generated using Delaunay triangulation [14], for which many improvements exist [15]. The search graph is subsequently obtained from each shared edge producing a node connected to the other nodes on the same triangle. For sparsely blocked game worlds, navmeshes typically produce smaller search graphs, which are easier to store and faster to search on. Figure 2.6 provides an example of a navigation mesh and the resulting search graph. For large simulated environments, hierarchical pathfinding like HPA* [5] can provide a significant performance increase over non-hierarchical algorithms [31].

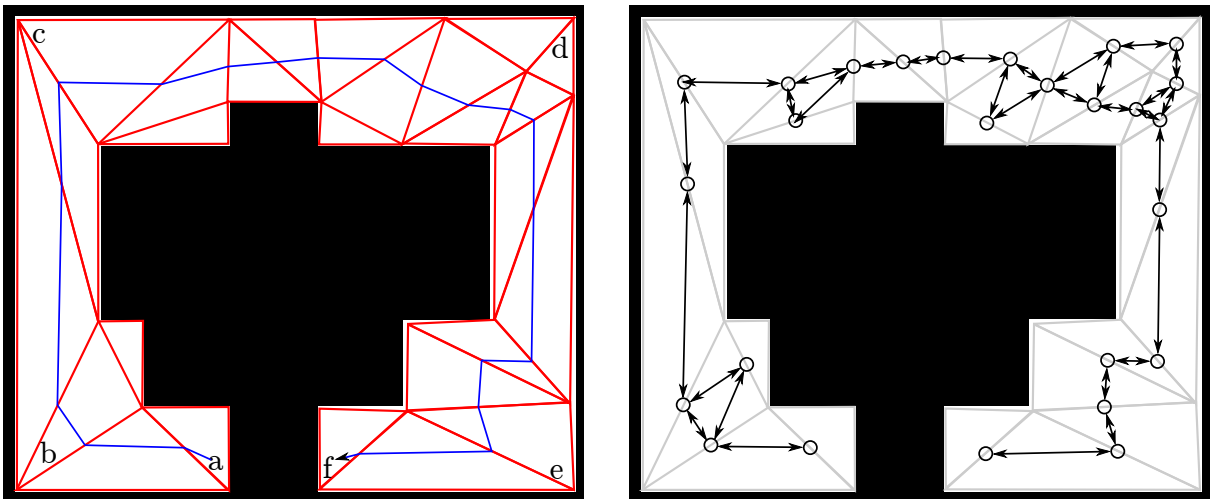

Figure 2.6: LEFT: Red: A navmesh, describing the navigable areas in space. All free regions are included in a triangle, and all triangles share one edge. Blue: a path between a and f, planned on the center points of shared triangle edges. Path post-processing is required to provide a realistic path. RIGHT: The corresponding search graph. Note the reduction in number of nodes compared to a grid based approach.

2.3 Mathematical Background of Physical Constraints

This section presents common physical limitations that should be adhered in games, and their Newtonian physics equivalents. Shown are acceleration and deceleration force constraints, linear speed limits, angular acceleration and speed limits, plus some special limitations, as minimal turning radii or maintaining a limited centripetal force. All physical constraints are swiftly explained for which vehicular types they apply and of what significance they are. This section assumes usage of the SI-Units meters, kilograms and seconds in the game world. Also, acceleration and deceleration coefficients a and d are always given in positive quantities, acting as undirected forces.

2.3.1 Linear Acceleration

Linear acceleration is described by the force currently exerted on an object. This measure is often expressed as $g - force$, with $1g$ being a equal to 9.80665 $^{m}/_{s^2}$, which is the acceleration towards the center of gravity of earth on sea level. Acceleration limits for unprepared humans are found at around $5-7g, \approx$ 50-70 $^{m}/_{s^2}$, with symptoms such as loss of consciousness when exceeded [2]. Disregarding rocket driven vehicles, typically, achievable accelerations are much lower. Not only the inertia of the vehicle has to be overcome, also, the connection between the vehicle and the ground only provides a certain traction. The amount of force generated vs. the amount of force acting on the ground is typically denoted by the friction coefficient μ. A sports car, accelerating from 0 to 100 $^{km}/_{h}$ in 3 seconds would then experience an acceleration of 9.25925 $^{m}/_{s^2}$. Deceleration forces may be higher, as they do not require engine generated force to apply, but naturally, they also succumb to physical traction limits.

In games with relatively uniform surfaces, the friction coefficient may be kept constant for all surfaces, and can effectively be incorporated into an acceleration limit of a type of vehicle. For games with distinct off-track surfaces, or games with a wide array of surface types, possibly being affected by changing weather conditions, modeling the friction coefficient explicitly may be a necessity.

Other factors such as air resistance affecting the maximally achievable acceleration are widely ignored in this thesis. For airplane movement, these factors have to be taken into account. Also, certain racing games might simulate increased air resistance at higher speed, as well as an additional downforce generated by negative lift.

For a vehicle operating in n dimensions, typically $2n$ different acceleration limits can be derived. These limitations are applied directed along the unit axes of the local coordinate system of a vehicle and in their opposite directions. A vehicle may accelerate quickly straight ahead, but less quickly backwards.

The formula for distance traveled while uniformly accelerating is given by:

$$s = \frac{1}{2}at^2 + v_0 t \tag{2.1}$$

Where v_0 denotes the velocity prior to the acceleration procedure, and t denotes the duration of the acceleration phase.

A second formula, describing the derivative of Formula 2.1, expresses the velocity gained.

$$v_t = v_0 + at \tag{2.2}$$

Using these formulas, the time it takes to pass any distance with fixed acceleration and given initial and target velocity can be calculated. In case both acceleration and deceleration are allowed as movement action, or to calculate heuristics spanning a wider distance than the standard discretization step, Formula 2.3 provides the relationship between optimal acceleration distance and deceleration distance for given initial and target velocities and fixed acceleration and deceleration force.

$$\begin{aligned} s &= s_a + s_d \\ as_a - ds_d &= \frac{v_t^2 - v_0^2}{2} \end{aligned} \tag{2.3}$$

Using Formulas 2.1, 2.2 and 2.3, the time it takes to travel a predefined distance, with known acceleration and deceleration forces and start and goal velocities can be obtained. Figure 2.7 illustrates the geometric relationship of the variables. The traversal time calculated is used in Subsection 4.2.2 to weight the distance heuristics, based on the current and goal point velocity and the acceleration limits of the vehicle.

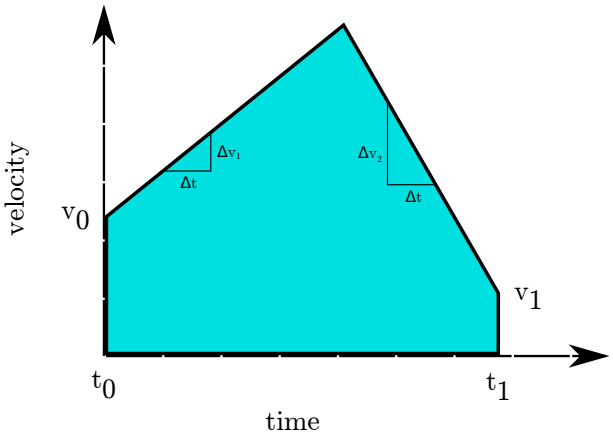

Figure 2.7: A time/velocity diagram showing the relationship between the entering velocity, v_0 at time t_0, the departure velocity after the segment is passed v_1, at time t_1. The slopes $^{\Delta v_0}/_{\Delta t}$ and $^{\Delta v_1}/_{\Delta t}$ represent the acceleration force a and the deceleration force d, respectively. The distance traveled is represented by the tinted area. Using Formula 2.3, the traversal time $t_1 - t_0$ can be calculated, as all other variables are given.

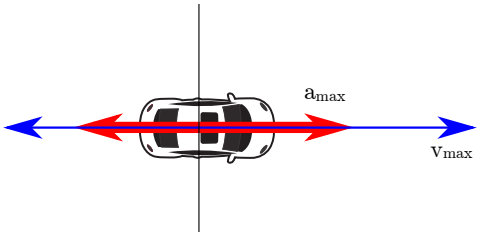

Figure 2.8: Linear acceleration limit a_{max} and speed limit v_{max} for a car. As cars are able to move backwards, the backwards acceleration and speed limits are nonzero.

2.3.2 Linear Speed Limit

Typically, special relativity is not simulated in computer games, thus virtually no universal speed limits exist. However, maximum engine output only allows a limited speed for vehicles. Humanoid characters traveling on foot are obviously subject to speed limitations as well. Besides the maximum speeds, a vehicle can reach, often a lower speed is desired, for example in simulated traffic environments. Here, roads each may have a distinct speed limit, for example $50^{km}/_{h}$ or $13.88^{m}/_{s}$.

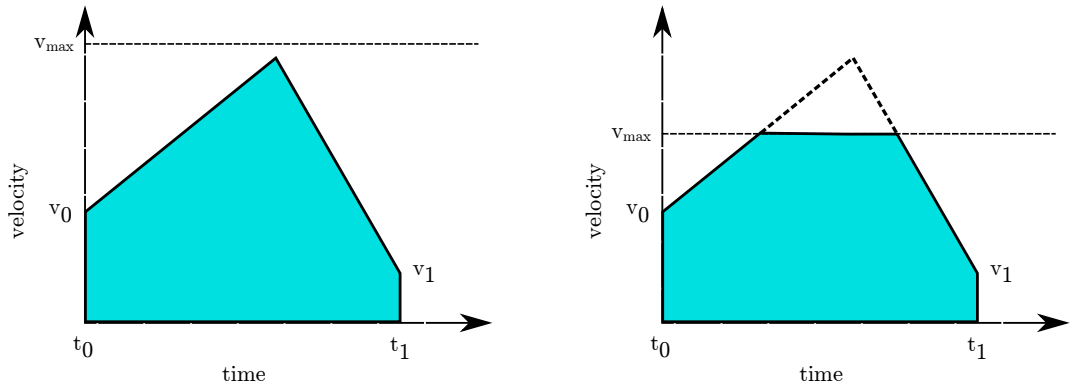

Figure 2.9: Time/Velocity diagrams with speed limit v_{max}. LEFT: v_{max} cannot be reached in the available distance. RIGHT: v_{max} is reached.

Incorporating a speed limit is straightforward. Depending on initial and goal velocity, two valid cases can occur: Figure 2.9, left, presents the case in which the maximal speed cannot be exceeded on the distance available. In this case, travel time is given by Formula 2.3.

The right half of Figure 2.9 displays the case in which maximal speed can be exceeded: In this case, acceleration time and distance to attain the speed limit are calculated using Formula 2.1 and 2.2; as well as distance and time needed to brake to the target velocity. The remaining distance is then calculated to be traveled by the maximal speed allowed. Again, speed limitations may be depending on the local orientation of the velocity vector. A car is typically able to drive faster forwards than backwards.

2.3.3 Angular Acceleration

A vehicle may be limited in the change of rotation speed per second. This not only concerns vehicle types that are able to turn on the spot, like tanks, helicopters or humans but also vehicles like airplanes, as for example rolling is a rotation around an axis within an airplane. Depending on the local axis of movement, different angular accelerations may be sustainable for the vehicle and the pilot. Figure 2.10 shows the yaw, pitch and roll axes for an aircraft. Any rotation around one of these axes with limited angular acceleration thus have to begun and ended slowly. As cars typically are not able to rotate in place, this limitation is no further investigated in this research.

2.3.4 Angular Speed Limit

Akin to the linear speed limitation, angular speed can be limited for various reasons. From a purely aesthetic point of view, humanoid characters should not turn in place unbelievably fast; also, nausea may occur for any occupant of a fast spinning vehicle. A too high angular speed may even lead to centripetal forces large enough to damage the vehicle. As before, vehicles are usually able to sustain angular speeds better in certain directions than others.

2.3.5 Centripetal Acceleration

Except for moving in a straight line, a certain curvature of the path is inevitable. In these cases, without centripetal force, an object's momentum would simply keep the object on a linear trajectory. In order to move in a circular way, a constant force pointing towards the center of the sphere is necessary, the so

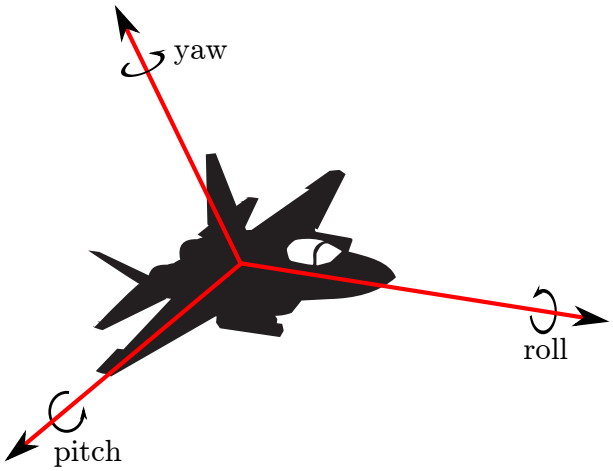

Figure 2.10: *Yaw*, *pitch* and *roll* angles for an aircraft. Angular movement may be restricted for each angle by both acceleration and turning speed limits.

called centripetal force. Centripetal acceleration for a circle with radius r is calculated with the following formula:

$$a_c = \frac{v^2}{r} \tag{2.4}$$

Interaction between the variables concerning centripetal acceleration is illustrated by Figure 2.11. For real vehicles, the available centripetal force is limited, limiting achievable turning radii in turn. For ground based vehicles such as cars, centripetal force is limited by the available traction. Other vehicles may only sustain limited lateral acceleration before damage is inflicted.

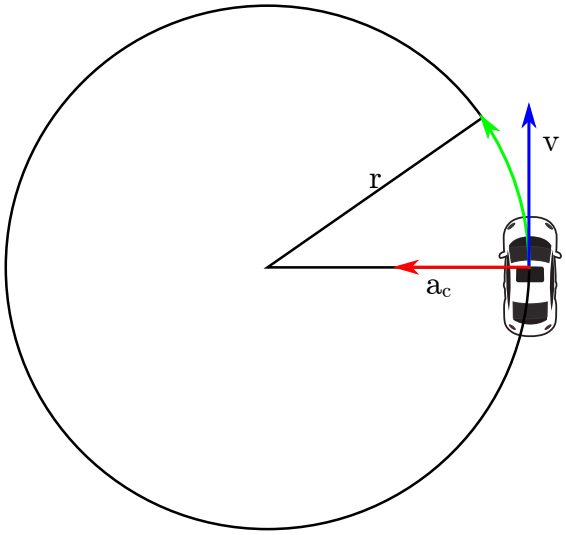

Figure 2.11: Centripetal force a_c, forcing a vehicle moving with velocity v onto a circular trajectory with radius r. If a_c was absent, the vehicle would continue moving in a straight line.

2.4 Related Work

This section refers to similar approaches of movement planning and producing more realistic movement in general from both academic and game AI.

2.4.1 Any-Angle Path Planning

Any-angle pathfinding is a cumulative term to describe techniques to find a shortest path between points on a two-dimensional grid in a way such that the resulting path is not artificially constrained to the grid coordinates in the plane. Being restricted to the grid coordinates, although all points on the plane are theoretically possible, creates suboptimal and artificially looking paths. Commonly used strategies to improve paths in games are path smoothing post-processing techniques like *string pulling* [41]. Post-processing paths however is bound to improve upon a potentially suboptimal path, and may not reach as good results as planning an any-angle path directly. *Field D** [19], an extension to *D** [49], and *Theta** [38] both are able to eliminate the artificial coordinate restriction during planning time using linear interpolation between each node expansion or planning shortcuts to any successor node if a line of sight check succeeds. Both however are not guaranteed to find optimal paths. *Anya*, a pathfinding method inspired by the *Continuous Dijkstra Algorithm*, is designed to provide true optimal paths without additional memory overhead [22]. Being able to traverse grids in a less artificially constrained manner is vital for motion constrained pathfinding, however, existing any-angle pathfinding algorithms focus on finding shortest distance paths only, disregarding possible angular constraints.

2.4.2 Motion Planning in Robotics

High dimensional state spaces occur whenever a robotic appliance features multiple joints. Planning in these state spaces usually involves preprocessing and remapping input states [10]. A planned trajectory is then commonly achieved by inverse kinematic calculations [33]. Other approaches produce a graph of connecting collision free states in the configuration space of the robot, then use a planner to find shortest paths on the generated graphs [27]. These methods perform efficiently even in high dimensional spaces, however, they quickly become infeasible for non-stationary robots and dynamic environments [17]. Motion planning techniques for mobile robotics tend to plan a spatial path first, then employing a local planner to generate a physically possible plan for the robot to follow. This approach is applied in the Robot Operating System, ROS [52], using a *Dynamic Window Approach* local path planner [20]. Rather than finding the shortest path, these local planners focus on steering robots regarding their physical limitations and without hitting obstacles [28]. Alternatively, a spatial path may serve as heuristic for a more complex state space search [32]. Recently developed algorithms in the field of unmanned aerial vehicle control compute a coarse, roughly physically possible, shortest path first, then refine the path at run-time using a local path planner [26]. While keeping a constant velocity, this approach is not unlike the one presented in this thesis.

2.4.3 Racing Game AI

For racing games with a static racetrack, often an ideal racing line is pre-calculated during the track design stage. The ideal racing line may include optimal velocities to be driven at any point. Also, as different vehicle types can have different friction coefficients, maximal speeds and accelerations, multiple ideal lines may be calculated, one per vehicle type. Commonly, domain experts sketch a rough outline of the ideal line; iterated tests then provide information to optimize parameters manually or automatically [7]. Racing under open street conditions faces the task to provide ideal racing lines for every possible path through a street network of potentially lifelike dimensions. Here, instead of calculating actual racing lines, center lines of straight road segments are supplied, as well as - potentially subdivided - circular segments for curves [1]. Deviating from the center lines is allowed to a fixed degree, and shortest paths are searched using standard AI search techniques, most commonly the A* algorithm [8]. Velocities are calculated during path following, looking ahead as far as needed to securely slow down in order to stop or drive safely along a circular segment. These techniques only generate locally optimal behavior, as velocities and physical constraints are not taken into account during planning time. For background AI populating the streets this might suffice, but in racing scenarios, more sophisticated approaches are required.

Chapter 3

Search Domain of AIPATH

This chapter describes the search domain AIPATH *operates in. Using an underlying A* algorithm,* AIPATH *requires a discrete representation of the state space. State spaces of multiple vehicle types are presented.*

Chapter contents: The state space description of various vehicle types.

3.1 State Space Description

In pathfinding techniques, only the represented position in space is included in each search graph node. In order to plan a path according to physical constraints, the current physical state has to be added to each node. Performing motion constrained pathfinding for a car thus implies that the vehicular orientation and the velocity of the car have to be incorporated in each node in addition to the position in space. In order to reduce the size of the search graph, a minimal state space description is desirable. For a surface bound car, at least a two dimensional position vector, and an orientation angle is required to describe the vehicle at rest. Assuming no loss of traction, the two dimensional velocity vector can be further reduced by one dimension by projecting it onto the orientation of the car. Thus, a minimal state space representation of a car can be described by a *state space vector* (x, y, ϕ, v), with x and y describing the position on the plane, ϕ describing the rotational offset of the vehicle to the x-axis in an angular measure, and v denoting the real-valued speed of the vehicle. The state space primarily used in this thesis and in all experiments is thus a four dimensional one, with two spatial components, one rotational component and the linear velocity of the vehicle.

As a car is not able to turn on the spot, and lateral forces are much higher than rotational accelerations in a typically possible curve, angular speed, and therefore its change, are not considered. For other vehicle types, the rotational speed would at least add another data field to the state, see also Subsection 3.1.1.

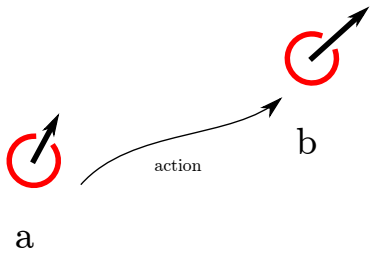

Figure 3.1: An action transforming the point a in state space to point b, altering position, velocity and orientation.

Possible *actions* in this state space are accelerating, decelerating, and changing the heading ϕ. Each action can be executed for a time t, producing an altered state space (x', y', ϕ', v'). Actions therefore describe edges in the search graph. Not all actions are mutually exclusive, so multiple actions may be executed at the same time, forming a *composite action*. Figure 3.1 shows an example of an action changing

the state space vector, and Figure 3.2 shows an iteration of a node expansion in the state space. Some actions, like accelerating and decelerating may cancel each other out, while other actions, like turning, depend on the state space to have a nonzero velocity. The listed actions may all be executed with a certain *intensity*, for example, an acceleration force a is bound by 0 and the maximal possible linear acceleration force such that $0 \leq a \leq a_{max}$. For memory and run-time reasons, intensities should be discretized so that only a limited subset of possible actions is explored. For child node generation, every possible combination of actions and intensities is generated, with a fixed positional change of either 0, -1 or 1 for both x and y. The time t required to execute the composite action is then calculated and assigned to be the cost of performing this action. It is assumed that each action is performed for the whole time t with uniform intensity.

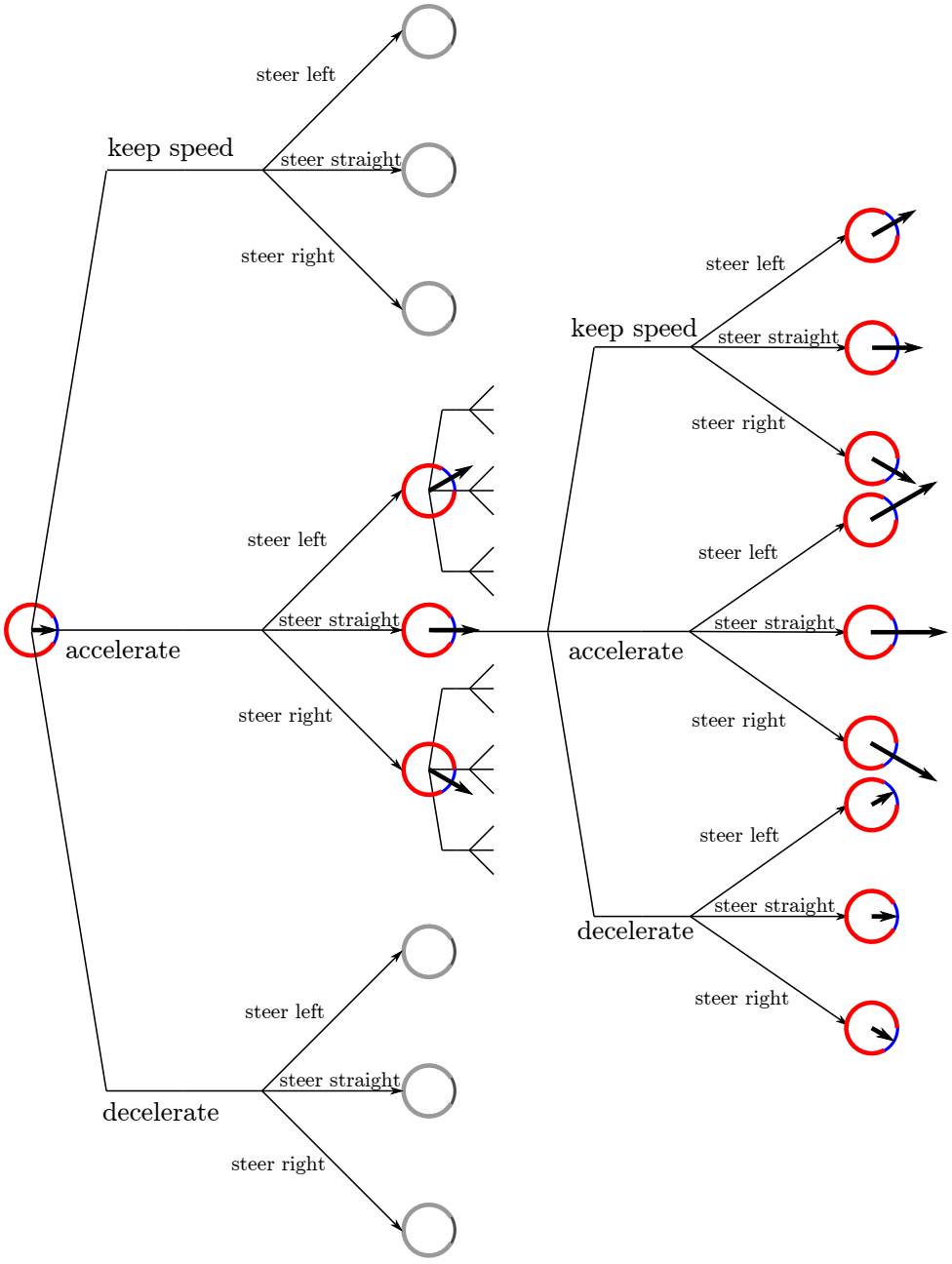

Figure 3.2: Expanding nodes in the vehicle state space. Acceleration and deceleration are performed with all discretely possible values, and combined with all possible steering angles. Only one possible discretization value is shown. Greyed out nodes are discarded after checking their validity – keeping zero speed is forbidden, as is decelerating when standing still. The generated node network functions as search graph input for A*.

3.1.1 Proposed Descriptions

This subsection presents proposed state space descriptions for a selection of vehicle types. Besides pedestrians and cars, EM5 contains boats and helicopters. Airplanes, tracked vehicles and possibly hovercraft are thinkable vehicles to be added in extensions. As the state space descriptions are very generic, it should be possible to transform vehicle representations of most games and simulations into one of the proposed descriptions.

Surface Bound Vehicles

Surface bound vehicles are vehicles that are limited to movement on a plane, like earthbound vehicles are limited to the ground. While ships are not ground based, they still behave much like ground based vehicles. Most computer games feature AI movement for surface bound vehicles only, leaving this vehicle type as the most relevant.

Pedestrians typically move slowly enough so that acceleration is not a decisive factor. They are able to turn on the spot, as they have no turning radius, so rotational speed should be limited. Again, as sustainable rotational speeds for humans are low, limiting rotational acceleration is a feature that may be dropped. Humans are able to walk backwards, but rarely do so. Limiting backwards speed to 0 or a small percentage of forward speed seems appropriate. Humans, due to their high center of gravity, may however be subject to lateral acceleration limits. Limiting centripetal acceleration, the planner can ensure that with rising speed, larger circles are taken, without limiting the ability to turn on spot. Proposed state space description is (x, y, ϕ, v), tracking position, orientation and speed. Because pedestrians are very agile in comparison to other vehicle types, path post-processing and sophisticated steering algorithms can improve their paths to a higher degree, lowering the demand for movement planning for pedestrians.

Cars occur in many games, from racing game simulations to open city games. Unlike pedestrians, cars can benefit a lot from improved movement planning. Cars feature a distinctive turning radius as well as they are subject to lateral acceleration at higher velocities. Furthermore, both linear acceleration and deceleration have to be taken into account when planning. The interesting set of physical limitations and the fact that cars are a common AI vehicle type in games make this vehicle type most valuable for further testing in this research. As noted above, the proposed state space representation is given by the state space vector (x, y, ϕ, v), tracking position, orientation and speed. Rotational speed is, unlike for the pedestrian type, not of much significance for the car vehicle type.

Tanks or other tracked vehicles are commonly able to turn without forward movement. In this respect, they behave like pedestrians. However, as tracked vehicles typically can make perfect use of their ability to move backwards, their backwards speed should not be artificially limited. Due to the mass and size of tanks, rotational accelerations manifest more obviously, thus they should be limited and included in the description. The proposed state space vector is $(x, y, \phi, \Delta\phi, v)$, adding the rotational velocity $\Delta\phi$ to the state space to allow putting a limit on the rotational acceleration.

Ships are another example of surface bound vehicles. Limiting acceleration and deceleration forces is obligatory due to their large masses. While some ships, like hovercraft and pump-jet powered ships may turn on the spot, this is not a common trait. Because of the low maximum speeds and large turning radii, centripetal acceleration is insignificant in modeling ships.

Aerial Vehicles

Aerial vehicles are vehicles that move detached from the ground. Additional to the forces generated by the vehicle, the gravitational pull has to be considered. Aerial vehicles, as they are able to move in three dimensions, feature six dimensional state vectors, $(x, y, z, \phi, \theta, \psi)$ only to describe a static configuration, here x, y and z denoting the position in space while ϕ, θ and ψ denote the yaw, pitch and roll angles, respectively.

Airplanes generate lift from forward movement. Unless for more precise physical simulations, movement in other directions is to be forbidden and their possible external occurrence neglected. Thus, the velocity of an airplane can be solely represented in direction of flight. Acceleration forces of both linear and rotational kind have to be considered and limited to viable extents. The proposed description breaks down the state of an airplane to three dimensional spatial coordinates, three orientation angles, three angular speeds and the forward speed of the aircraft. The resulting state space vector is $(x, y, z, \phi, \theta, \psi, \Delta\phi, \Delta\theta, \Delta\psi, v)$, adding the rotational yaw, pitch and roll speeds $\Delta\phi$, $\Delta\theta$ and $\Delta\psi$ as well as the linear forward speed v to the static configuration state space vector.

Helicopters have a limited range of pitch and roll angles, beyond which they cannot generate sufficient lift to hover. Speed and acceleration limits on all rotational variables are required. Besides pitch and roll angles, the angle of attack of the rotor blades provides upwards thrust, which can be modeled by a locally upwards facing speed and acceleration. The state space representation is identical to the airplane model, except that the upwards facing main thrust generates lift directly, as opposed to indirectly in the airplane case.

Other

Other vehicles may include a variety of less commonly simulated vehicles, such as hovercraft, rockets and submarines. Due to the manifold of possible vehicle types, and the fact that most vehicles share common traits, only the most exotic one, namely spacecraft, is listed here.

Spacecraft are typically set in a state of no acceleration. Depending on the configuration, they can freely rotate around any axis. Naturally, these rotations are subject to limited speeds and accelerations. The biggest difference between spacecraft and any other vehicle type presented so far is that they do not feature a distinctive deceleration force, as there is only negligible to no drag in their environment. It means that to maneuver from resting position to another resting position, a vehicle has to accelerate, then negate the former acceleration by rotating and accelerating an equal amount in the opposite direction. The state space can be described by a vector equal to the one used for airplanes, $(x, y, z, \phi, \theta, \psi, \Delta\phi, \Delta\theta, \Delta\psi, v)$, however, the set of possible actions differs.

Chapter 4

Physical Movement Planning with aiPath

This chapter describes aiPath*, the state space discretization technique in use and the heuristic evaluation function for the underlying A* algorithm. An additional search step, performing a non-backtracking best-first search before A*, called the probing search step, is briefly introduced.*

Chapter contents: Introduction of aiPath and how it works.

4.1 Introducing aiPath

In this thesis research, a standard A* pathfinding algorithm is equipped with a heuristic evaluation function estimating the time between any two points in the car-type vehicle state space described in Chapter 3. The A* algorithm is then employed to search a path in a given state space using a custom discretization strategy. Depending on the urgency, aiPath can also produce an incomplete path by using the heuristic evaluation function to perform a non-backtracking best-first search to probe for the goal node. That way, a vehicle may start moving along the probed path well before A* provides an optimal result. If the search space contains few obstacles, the probing search may find a valid shortest path without constructing an open and closed list. If the probing step fails to produce a valid path, all visited nodes can be re-used for A* by inserting them into the open list. Algorithm 1 provides pseudocode explaining how aiPath combines the probing search and A*.

Algorithm 1 aiPath Pseudocode

```
1:  procedure aiPath(startNode, goalNode) returns a sequence of nodes towards goalNode
2:      path ← ProbingSearch(startNode, goalNode)
3:      if GoalTest(path.last, goalNode) then
4:          return path
5:      closedList ← ∅
6:      openList ← Insert(startNode)
7:      for each node in path do
8:          openList ← Insert(node)
9:      path ← aStar(startNode, goalNode, openList, closedList)
10:     return path
```

4.2 Heuristic Evaluation Function

A perfect heuristic evaluation function returns the exact costs of the rest of the path for any node, allowing for an optimal $O(n)$ best-first search, with n being then number of nodes on the path. Due to possible obstructions on the path, implementing a perfect heuristic function is infeasible. Any imperfect heuristic evaluation function can be used in combination with search algorithms like A* to provide an improved order of nodes to expand next, reducing the number of search iterations. In order to be guaranteed to find an optimal path, a heuristic has to be admissible, that is, the estimated costs have to be less or equal to the actual costs to. Using Euclidean distance as spatial heuristic works well in low dimensional spaces. As with higher dimensional search spaces the branching factor increases steeply, a more focused heuristic function is required in order to comply with reasonable memory and time constraints. While consistency is a desirable property [40], it can not be trivially achieved due to the discretization of the search space. Tests with inconsistent heuristics show show that any overhead due to inconsistency is comparatively low for practical purposes [18]. The heuristic evaluation function used in this research is designed to be admissible while being as close as possible to the actual costs.

The proposed heuristic evaluation function for AIPATH estimates the distance to the goal node, and, in a second step, uses the given speed and acceleration values to calculate an accurate traversal time.

4.2.1 Distance Estimation

Calculating the distance between two oriented points on a plane for a turning radius limited vehicle is derived from the work of Pinter [41]. His work is based upon the fact that any oriented point can be reached by moving a distance along one circle segment, continuing movement in a straight line and entering another circle segment. A circle segment may either be a clockwise or counterclockwise turn; that is, a vehicle turning right or left in a local coordinate frame. In order to find the minimal distance, the turning radius is used for the radius of the circular segments. Subsequently, all combinations of clockwise and counterclockwise turns are calculated; the lowest outcome denoting the distance between the two oriented points using a turning radius. Figure 4.1 shows all four combinations of clockwise and counterclockwise turning. Pinter's algorithm does not include the use of variable turning radii, like a speed dependent minimal turning radius, which would be required to model centripetal force limits.

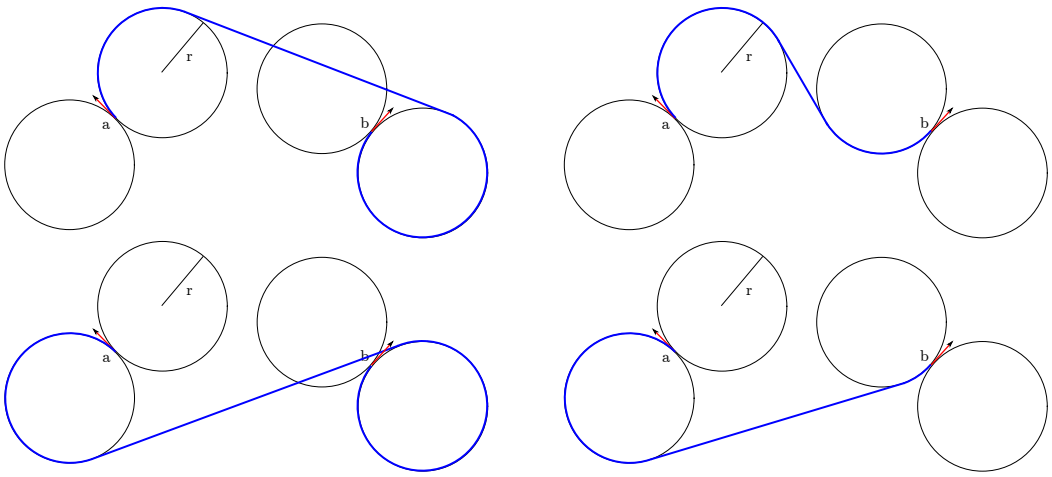

Figure 4.1: All four ways to navigate between two oriented points a and b on the plane using a turning radius r. The lengths of the different trajectories are calculated and the shortest one is chosen.

Figure 4.2 shows the value of the heuristic function plotted in a heat map style. Clearly visible are the discontinuities along the turning circle, as points within the turning circle cannot be reached without additional maneuvering. From the heuristic, the optimal goal orientation at any point can be determined as well. Figure 4.3 displays a quiver plot, illustrating both the optimal arrival orientation as well as the heuristic value at any point, assuming a trajectory from the origin.

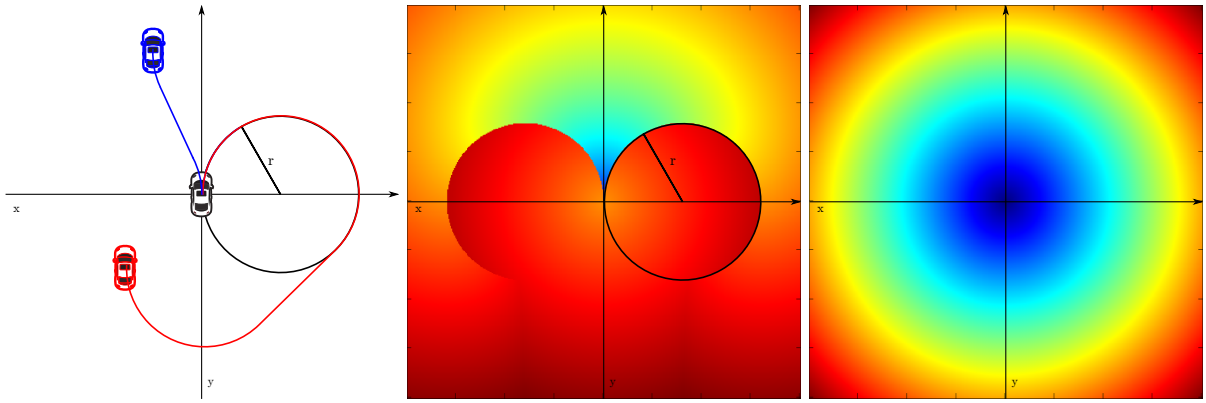

Figure 4.2: LEFT: The search setup used to plot the distance estimating heuristic function. MIDDLE: A heat map plot of the heuristics value from the origin towards any point, using a turning radius r. Origin and goal are both facing upwards, i.e., in positive y direction. Note that the area directly ahead of the vehicle behaves like Euclidean distance, which is included for comparison on the RIGHT.

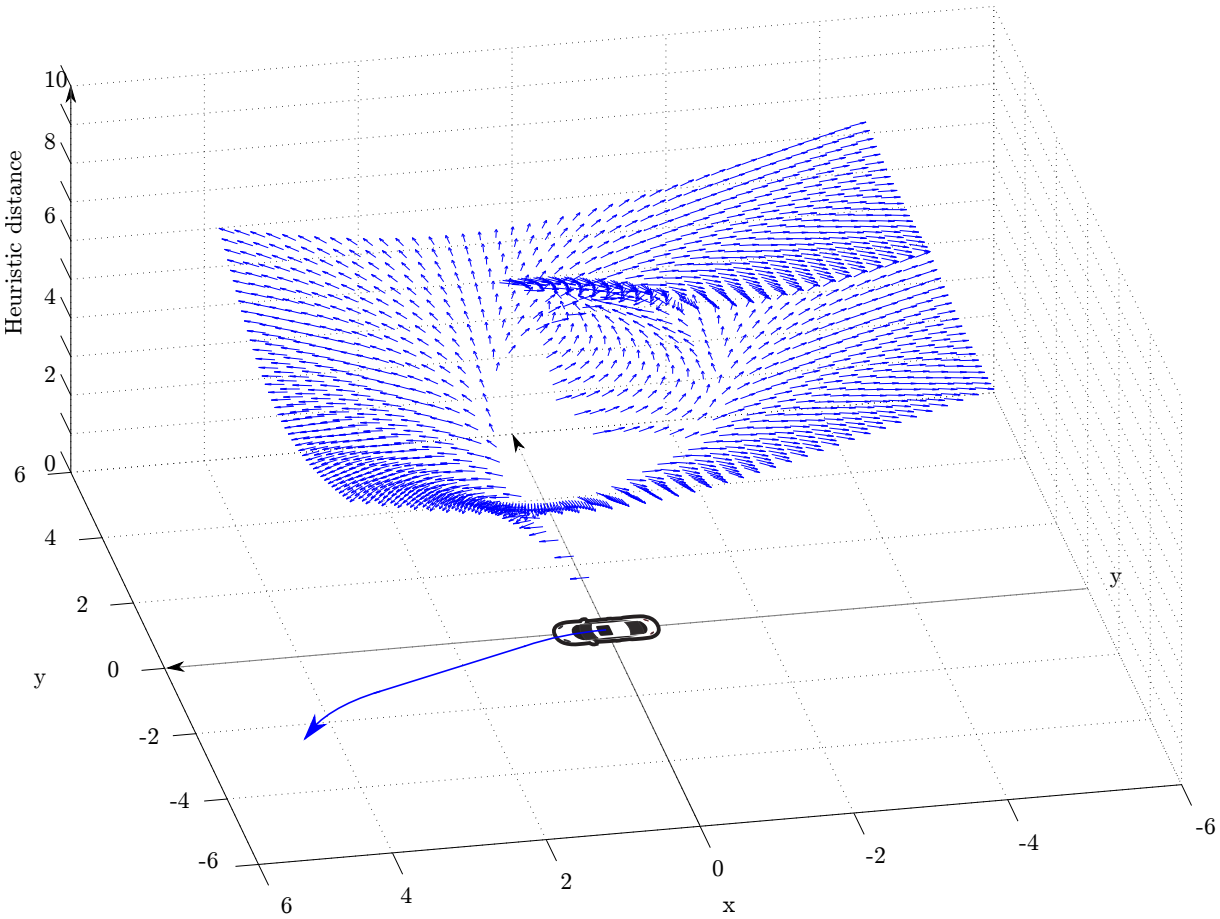

Figure 4.3: A 3d-quiver plot of the ideal arrival orientation for any point. The heuristic distance from the origin to any point is indicated by the height of the arrow. Every arrow points in the direction of optimal arrival, i.e. the goal direction with the lowest heuristic value. All distances are given in meters. The discontinuity in both direction and heuristics value along the turning radius can clearly be seen.

4.2.2 Traversal Time Estimation

The previously discussed distance estimation function takes only three out of the four state space variables of the car-type state space described in Chapter 3 into account; the position given by x and y and the orientation ϕ. To be able to operate efficiently in the complete four dimensional state space, a feature concerning the fourth component, the velocity v, is required. Taking the velocity into account produces a heuristic evaluation for any node in *seconds until arrival*, as opposed returning the *distance to the goal*.

The proposed heuristic evaluation function, estimating traversal time, makes use of the calculated distance to the goal node, then takes the additional three factors into account to calculate the minimal time for arrival:

(1) Velocities at the start and goal state are considered, as well as (2) possible speed limitations, to calculate a movement time with maximum viable speed. In order not to overestimate costs, (3) acceleration and deceleration limits are used, ensuring the vehicle to move as fast as possible at all times.

The traversal time is then calculated by inserting the given distance s and the initial and goal velocities v_0 and v_g into the Formulas 2.1 and 2.2. A test is performed whether s suffices to accelerate to v_{max}, then decelerate to v_g. If this test fails, solving Formulas 2.1 and 2.3 for t provides the minimal traversal time. If the test succeeds, meaning the vehicle spends some time with the maximum velocity, the time to pass the remaining distance at v_{max} is added to the partial result. Formula 4.1 shows the mathematical formulation of the traversal time calculation, and Algorithm 2 provides pseudocode describing how to detect under- or overshooting, how to react on it and how to calculate the traversal time in detail.

$$t(s, v_0, v_1, a, d) = \begin{cases} \frac{\sqrt{v_0^2 + 2a \cdot \frac{2ds - v_0^2 + v_1^2}{2ad}} - v_0}{a} + \frac{\sqrt{v_1^2 + 2d \cdot \frac{2ds - v_0^2 + v_1^2}{2ad}} - v_1}{d} & \text{if } \frac{s}{v_{max}} < \frac{v_{max} - v_0}{a} + \frac{v_{max} - v_1}{d} \\ \frac{s}{v_{max}} + \frac{1}{2} \cdot \left(\frac{v_{max} - v_0}{a} + \frac{v_{max} - v_1}{d}\right) & \text{else} \end{cases} \tag{4.1}$$

Algorithm 2 Traversal Time Estimation Heuristics Pseudocode

1: **procedure** EstimateTraversalTime(*startNode*, *goalNode*, *config*) **returns** the traversal time between two nodes
2: $distance \leftarrow$ EstimateTraversalDistance(*startNode*, *goalNode*)
3: $v_0 \leftarrow startNode.v$
4: $v_1 \leftarrow goalNode.v$
5: $t_{maxSpeed} \leftarrow {}^{distance}/_{config.maxForwardSpeed}$
6: $t_{acc.} \leftarrow {}^{(config.maxForwardSpeed - v_0)}/_{config.maxAcceleration}$
7: $t_{dec.} \leftarrow {}^{(config.maxForwardSpeed - v_1)}/_{config.maxDeceleration}$
8: **if** $t_{maxSpeed} \geq (t_{acc.} + t_{dec.})$ **then**
9: **return** $t_{maxSpeed} + {}^{(t_{acc.} + t_{dec.})}/_2$
10: $accelerationDistance \leftarrow \frac{2 \cdot config.maxDeceleration \cdot distance - v_0^2 + v_1^2}{2 \cdot (config.maxAcceleration + config.maxDeceleration)}$
11: $decelerationDistance \leftarrow distance - accelerationDistance$
12: **if** $accelerationDistance < 0$ **or** $accelerationDistance > distance$ **then**
13: **return** HandleOvershooting(v_0, v_1, *distance*, *config*)
14: $accelerationTime \leftarrow \frac{\sqrt{v_0^2 + 2 \cdot config.maxAcceleration \cdot accelerationDistance} - v_0}{config.maxAcceleration}$
15: $decelerationTime \leftarrow \frac{\sqrt{v_1^2 + 2 \cdot config.maxDecceleration \cdot decelerationDistance} - v_1}{config.maxDecceleration}$
16: **return** $accelerationTime + decelerationTime$

Handling Overshooting

During traversal time estimation, two kinds of error can occur. The calculation of the acceleration distance can return a negative distance or a distance that is larger than the total distance available. These cases correspond to under- or overshooting, i.e. not being able to accelerate or decelerate, as the absolute speed difference $|v_g - v_0|$ is too high for the available acceleration forces a and d. A naive approach is to forbid expansion of these nodes by returning a very high or infinity value as heuristic, $Cost_{Max}$. As a valid setup may, however, include a seemingly impossible setting, a more sophisticated approach is to be preferred. Depending on the vehicle configuration, the vehicle either has to make a turn in order adapt the speed, or may brake, turn in place and try again. If both options are present, the one taking less time is preferred. Algorithm 3 provides pseudocode explaining the overshooting handling in detail. Appendix A.2 contains heat map plots of the heuristic function using different overshooting strategies.

Algorithm 3 Overshooting Handling Pseudocode

```
1: procedure HandleOvershooting(v_0, v_1, distance, config) returns the time to handle over- or undershooting
2:     t_circular ← ∞
3:     t_linear ← ∞
4:     if config.maxLateralAcceleration ≠ 0 then
5:         t_circular ← HandleCircularOvershooting(v_0, v_1, distance, config)
6:     if config.maxTurnOnSpot then
7:         t_linear ← HandleLinearOvershooting(v_0, v_1, distance, config)
8:     return Min(t_circular, t_linear)
```

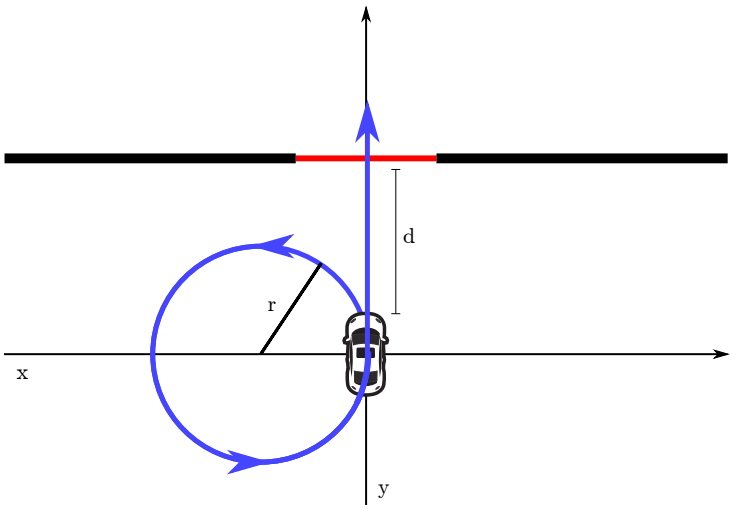

Figure 4.4: A vehicle unable to accelerate to its goal velocity on a straight line with distance d will deliberately plan a circle of sufficient circumference to gain enough speed. A lower bound for the radius of the circle r is given by the turning radius of the vehicle.

Circular Overshooting Handling. If the vehicle has a defined turning radius, it has to drive a circle in order to gain or lower additional speed. Figure 4.4 shows an example of circular backup to prevent undershooting. Note that the minimal distance for a turning circle, s_{turn}, is $2\pi r$, r being the turning radius. The minimal distance s_{min} to achieve v_g can be calculated using the Formulas 2.1 and 2.2. If $s_{min} < s_{turn}$, the vehicle can adapt speed before finishing the circle. The time on the remaining section of the circle is calculated using the traversal time estimation Formula 4.1. If $s_{min} \geq s_{turn}$ however, performing a single turning radius turn does not suffice to adapt the velocity. In this case, the radius of

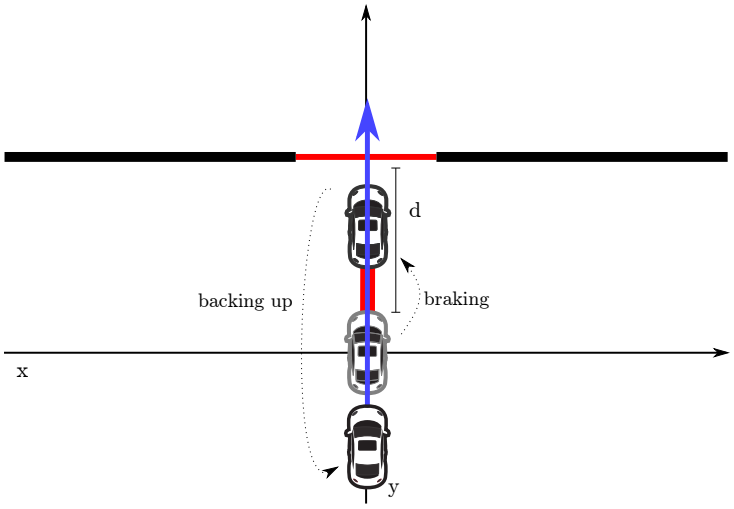

Figure 4.5: A vehicle unable to accelerate to a certain velocity in the required distance d may have to brake first, back up by either turning on the spot or driving backwards, then accelerate again from a point further away.

the circle is adjusted so that the length fits s_{min} exactly. Pseudocode illustrating the circular overshooting handling is available in Algorithm 4.

Algorithm 4 Circular Overshooting Handling Pseudocode

procedure HANDLECIRCULAROVERSHOOTING(v_0, v_1, *distance*, *config*) **returns** the time to handle over- or undershooting
  if $v_1 < v_0$ **then**
    $t_{adjustSpeed} \leftarrow {}^{(v_1 - v_0)}/_{config.maxAcceleration}$
    $d_{adjustSpeed} \leftarrow {}^1/_2 \cdot config.maxAcceleration \cdot t^2_{adjustSpeed}$
  else
    $t_{adjustSpeed} \leftarrow {}^{(v_0 - v_1)}/_{config.maxDeceleration}$
    $d_{adjustSpeed} \leftarrow {}^1/_2 \cdot config.maxDeceleration \cdot t^2_{adjustSpeed}$
  $backupCircumferenceNeeded \leftarrow d_{adjustSpeed} - distance$
  $minCircumference_{turningRadius} \leftarrow 2 \cdot \pi \cdot config.turningRadius$
  $minCircumference_{lateralForce} \leftarrow 2 \cdot \pi \cdot {}^{config.maxSpeed^2}/_{config.maxLateralAcceleration}$
  $minCircumference \leftarrow$ MAX($minCircumference_{turningRadius}$, $minCircumference_{lateralForce}$)
  if $backupCircumferenceNeeded \geq minCircumference$ **then**
    return $t_{adjustSpeed}$
  return ESTIMATETRAVERSALTIME(v_0, v_1, $d_{adjustSpeed}$, *config*)

Linear Overshooting Handling. If the vehicle is allowed to change its movement direction while standing still, instead of driving a lengthy curve to match velocities, it may be faster to stop, back up an appropriate distance and retry from there. Figure 4.5 illustrates this procedure. Limited angular acceleration and speed determine the time required to turn around on the spot. Calculating deceleration time to a complete stop and accelerating to the desired velocity is straightforward, the remaining, backing up section, is calculated using Formula 4.1 and the appropriate acceleration and deceleration values. Algorithm 5 provides pseudocode, explaining the linear overshooting handling. In case the vehicle is allowed to move backwards, backing up can also be done without turning on the spot. Other speed and acceleration limits may apply when moving backwards.

Algorithm 5 Linear Overshooting Handling Pseudocode

1: **procedure** HANDLELINEAROVERSHOOTING(v_0, v_1, *distance*, *config*) **returns** the time to handle over- or undershooting
2: $t_{brake} \leftarrow {}^{v_0}/_{config.maxDeceleration}$
3: $d_{brake} \leftarrow {}^1/_2 \cdot config.maxDeceleration \cdot t_{brake}^2$
4: $d_{Segment1} \leftarrow d_{Brake} - distance$
5: $t_{acc.} \leftarrow {}^{v_1}/_{config.maxAcceleration}$
6: $d_{acc.} \leftarrow {}^1/_2 \cdot config.maxAcceleration \cdot t_{acc.}^2$
7: $d_{Segment2} \leftarrow d_{acc.}$
8: $d_{Segment3} \leftarrow d_{Segment1} + d_{Segment2}$
9: $t_{backup} \leftarrow$ ESTIMATETRAVERSALTIME(0, 0, $d_{Segment3}$, *config*)
10: $t_{overshoot} \leftarrow t_{brake} + t_{backup} + t_{acc.}$
11: **return** $t_{overshoot}$

4.2.3 Performance Considerations

Because the complete heuristic evaluation function is calculated for any visited node during search, that is, both distance and traversal time are calculated, performance requirements may render optimization of the heuristic evaluation function for speed a necessity. A performance improvement may be achieved by replacing the costly distance estimation procedure by computationally less expensive estimations. To achieve similar results with lower calculation costs, goal nodes within a certain angle ahead of the vehicle may be calculated using underlying Euclidean distance, as the results of this function are quite similar, as also illustrated by Figure 4.2. The summed difference between heuristic distance and Euclidean distance can be calculated by systematically summing over all possible translation angles α between the origin and the goal position, and over all possible goal orientations ϕ. The average offset as a function of Euclidean distance $\bar{o}(d_e)$ is then calculated as described by Formula 4.2, summing up the difference between heuristic score h and Euclidean distance d_e. Note that the distance estimating heuristic h takes two parameters, the start node, situated in the origin, and the goal configuration. Both configurations are given in the three dimensional, incomplete state space description (x, y, ϕ), with x and y describing the position and ϕ providing the orientation angle.

Figure 4.6 shows the difference between heuristically estimated distance and Euclidean distance, and clearly illustrates that the offset loses significance if the goal is more than about 4 turning radii away. Thus, for heuristic evaluations of distant points, a weighted Euclidean distance may suffice, saving calculation overhead. The traversal time is then calculated based on the estimated distance. Figure A.1 provides a more detailed plot, with offset values for all goal translation angles.

$$\bar{o}(d_e) = \frac{1}{360} \sum_{\alpha=0^\circ}^{360^\circ} \frac{1}{360} \sum_{\phi=0^\circ}^{360^\circ} h\left(\begin{pmatrix} 0 \\ 0 \\ 0^\circ \end{pmatrix}, \begin{pmatrix} cos(\alpha) * d_e \\ sin(\alpha) * d_e \\ \phi \end{pmatrix} \right) - d_e \tag{4.2}$$

4.3 Discretization Strategies

This section presents the different discretization strategies used on the state space variables. As in this thesis research the main focus was cast on car-type vehicles, the discretization strategies described are designed to work well with cars as well, results for other vehicle types may vary. The car state space is given by the vehicle position, orientation and speed, or the vector (x, y, ϕ, v). As x and y describe spatial coordinates, *spatial discretization* is applied. The orientation of the vehicle is discretized by the means of *directional discretization*. The current velocity of the vehicle is discretized using *uniform discretization*. As the search graph not only features nodes, but also edges, the possible *actions* transforming one state space vector into another have to be discretized as well in order to keep the search graph edge number finite. Actions are discretized using special *action intensity* and *action duration* discretization strategies.

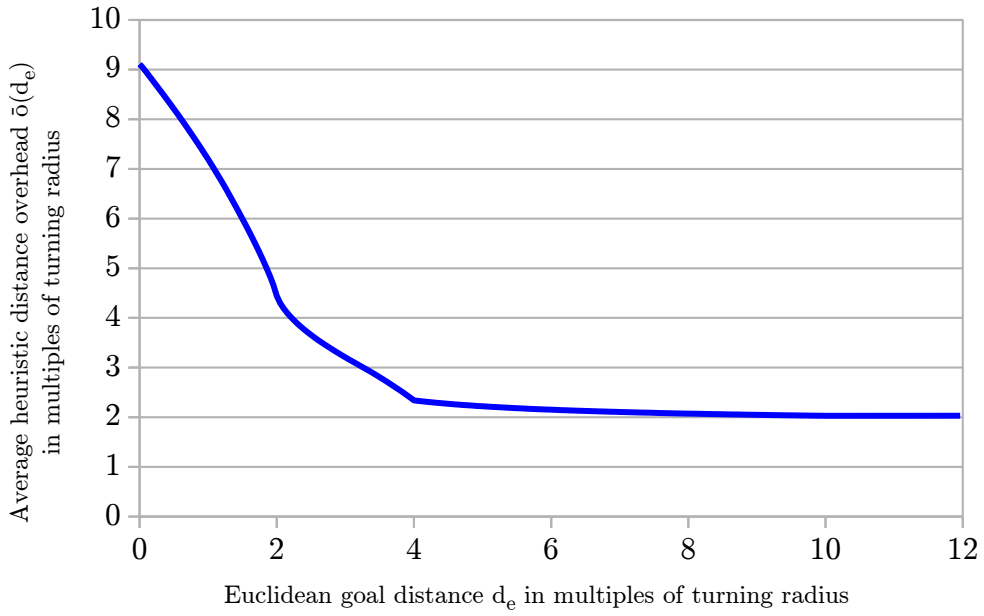

Figure 4.6: The average difference between heuristic distance estimation and Euclidean distance plotted against the Euclidean distance in multiples of turning radii. It is clearly visible that beyond 4 turning radii, the estimated distance converges towards a static offset of the Euclidean distance. The average overhead for distant goal point appears to be 2 turning radii.

4.3.1 Uniform Discretization

Uniform discretization describes the most simple form of discretization by rounding any given value to the closest one supported by the discretization resolution. This is the one dimensional equivalent of a grid discretization.

4.3.2 Spatial Discretization

Besides static non-uniform discretization and adaptive discretization methods, a simple grid remains one of the most basic and thus frequently used discretization methods [21]. However, using a grid loses resolution detail by assigning any point entered the positional coordinates of the center point of the associated grid. In Figure 4.7, the left and middle illustrations describe the naive grid discretization and show the loss of detail by marking the whole cell in one color, indicating that the sub-cell resolution is lost. The spatial discretization strategy for AIPATH exploits the fact that A* mainly uses discretization to reduce the number of search nodes. Furthermore, the fact that already explored search nodes may be re-opened and re-assigned with differing costs and predecessor nodes allows for a *selection* which node's exact, undiscretized position may be stored. Figure 4.7, right, illustrates the discretization procedure. Any grid cell may at most hold one node, as in standard uniform grid discretization. The *cost* and *heuristic* value of the node determine, which node is kept. All nodes that A* considers inferior to the one preserved are discarded, but the exact position of the node with lowest sum of cost and heuristic value is preserved. If a better node is detected in the same grid cell during path planning, it replaces the previously best node and overwrites its exact position, cost, heuristic value and predecessor.

4.3.3 Directional Discretization

Discretizing direction vectors using uniform grid discretization is straightforward, but produces a higher amount of discretization error. It is proposed to represent direction vectors as mutually independent rotation angle and magnitude values instead. The orientation of the vehicle is thus expressed as a rotation angle measured counterclockwise to the x-axis. Figure 4.8 illustrates the reduced error when discretizing direction vectors with the proposed strategy. The direction of the velocity vector can be converted to an angular value as well, using a separate data field to denote the speed, i.e., the length of the velocity vector projected to the orientation vector. Exploiting the assumption that the vehicle does

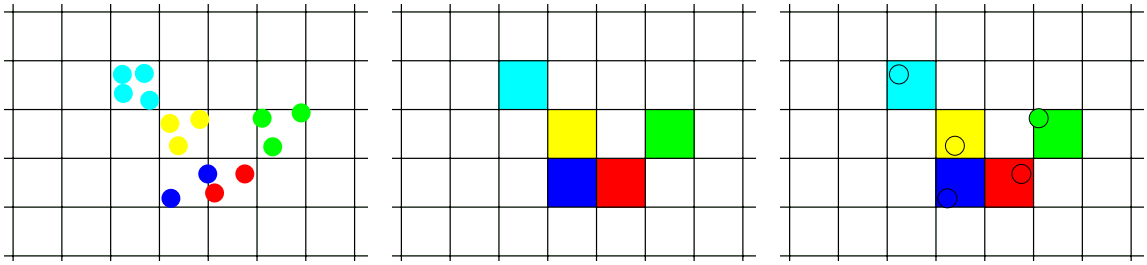

Figure 4.7: LEFT: Some points in space associated with grid cells. MIDDLE: Using naive discretization, every point on the grid is merely associated with a whole cell. RIGHT: Using the presented discretization approach, any cell may only contain one node, but the exact position of the node is preserved.

not lose traction, the velocity vector is expected to be always collinear to the orientation vector. Thus, the orientation component of the composite velocity can be dropped. Vehicle speed is discretized with a user-defined resolution, much like the spatial discretization. Angular orientation is discretized to degrees, as they are a human friendly representation for angles and prove to be of sufficiently fine resolution to successfully search with.

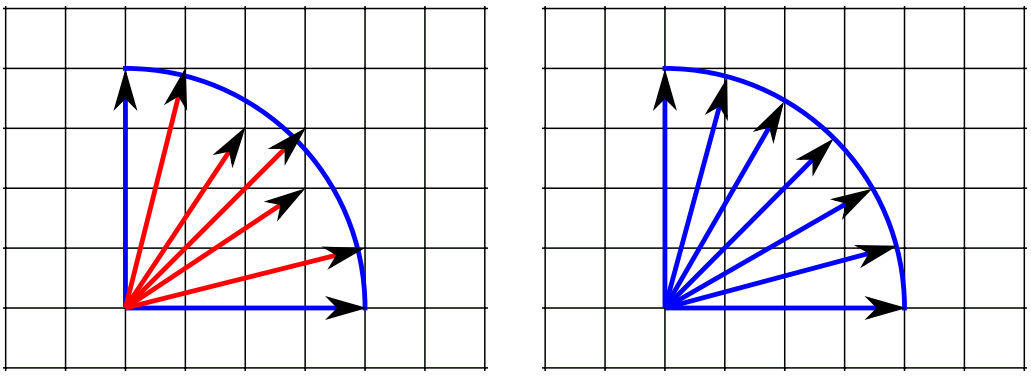

Figure 4.8: LEFT: Discretization of direction vectors results in nonuniform angles and differing lengths for the vectors marked red. RIGHT: Representing direction vectors as an angle and a magnitude, using discrete angular increments for directions produces constant magnitudes and uniform angular intervals.

4.3.4 Action Intensity Discretization

Being a transformation between state spaces, action intensities do not necessarily have to be discretized using the same mechanism or coarseness as state space variables. Acceleration and deceleration forces were limited to either zero force, in case the vehicle should maintain a certain, possibly its maximally allowed, velocity, or full force. Adding a third possible discrete step in between no acceleration and the acceleration limit allows the vehicle to travel more smoothly. However, this additional intensity is only chosen during search if the path layout demands it, as reaching, then keeping maximum velocity as quickly and for as long as possible, then braking down using maximal force always produces a minimum cost path. Due to discretizational error, however, interjecting an action with lower-than-maximal force may produce a shorter path. Steering angles, describing the intensity of the turning action, may be discretized to a user defined number of degrees. Using the same resolution as for directional discretization of the state space can generate shorter paths, however, limiting the possible steering angles to multiples of 5 or 10 provides a significant reduction of the branching factor, speeding up the search. Figure 4.9 shows an example of action intensity discretization for steering right at various intensities. An important difference in action intensity discretization to uniform discretization is that the intensity limits should always be conserved as real values. For example, a vehicle that is allowed to turn 15° per step should be able to perform a steering action with 15° intensity, even if the discretization allows only intensities with the value being a multiple of 10°.

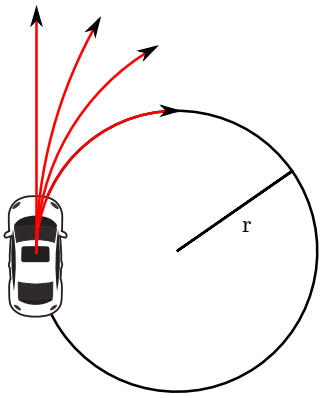

Figure 4.9: A vehicle turning right with different, discrete action intensities. The intensity limits should always be included, for turning this means allowing no turning and turning with exactly the turning radius limitation.

4.3.5 Action Duration Discretization

Instead of discretizing the action duration directly, the spatial change, that is, the Euclidean distance between the node to be expanded and the child nodes, is discretized to the resolution of the spatial grid. This strategy ensures that at least a distance of one spatial grid cell is traversed. The necessary action duration to perform the spatial change is then calculated and used for child node generation.

4.3.6 Temporal Discretization

An alternative, additional discretization of temporal variables is possible, but provides little to no benefit while degrading the precision of the result of the heuristic evaluation function. Time, measuring costs and heuristic estimates of costs, is thus not discretized in the course of this research. Costs are not a state space variable - they are adapted if another, shorter, path is found. Incorporating costs to the state space leads to insertion of a new node each time a cost is adapted. As node reduction is a main goal of discretization, and discretization of time does not give a reduction in the number of nodes, this procedure is not applied in AIPATH. However, when searching with a maximum vehicle speed, minimal costs between two adjacent nodes, $cost_{min}$, can be easily calculated, and subsequently, cost values may be discretized using $cost_{min}$ as a basis. Discrete cost values may lead to more ties during node selection, placing higher emphasis on the heuristic function as tie-breaker. Discretizing both cost and heuristic values leads to more unbroken ties, and thus a higher number of expanded nodes is expected.

4.4 Probing Search

Due to the high branching factor in the four dimensional search space, a greedy best-first search with no backtracking can be used in a pre-search step to perform a *probing search* led by the heuristic function towards the goal. Algorithm 6 explains the algorithm in detail. If no obstacles or other problems unregistered for by the heuristic arise on the path, a shortest path can be found without using the A* algorithm at all. If the probing step fails, all visited nodes can be used as starting points for the subsequent A* search as shown in Figure 4.10, conserving their already calculated heuristic values and costs. Performing a probing search and starting movement according to it can reduce latency times, when it is more important that vehicles do not appear to be unresponsive then that they always follow the shortest path. Physical correctness is always given due to the fact that the probing search makes use of the physical state space description. If the underlying heuristic is admissible and consistent, the shortest path is found in abundance of obstacles. If the underlying heuristic is not consistent, suboptimal paths may be found, as no history of traversed nodes is kept.

Algorithm 6 Probing Search Pseudocode

```
1:  procedure ProbingSearch(startNode, goalNode) returns a sequence of nodes towards goalNode
2:      node ← startNode
3:      global_best_h ← Heuristic(node)
4:      do
5:          path ← Insert(node)
6:          if GoalTest(node, goalNode) then return path
7:          successors ← Expand(node)
8:          local_best_h ← global_best_h
9:          for each successor in successors do
10:             h ← Heuristic(successor)
11:             if (h < local_best_h) then
12:                 local_best_h ← h
13:                 node ← successor
14:         improvement ← (local_best_h < global_best_h)
15:         if improvement then
16:             global_best_h ← local_best_h
17:     while (improvement);
18:     return path
```

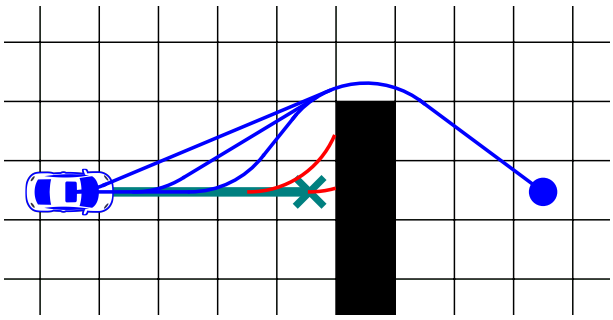

Figure 4.10: Even though the probing search terminates unsuccessfully due to obstacles on the path, the nodes visited can be re-used as starting points for A*, to plan the shortest of the three presented trajectories.

4.5 Implementation Details

As of today, C++ is still the main programming language used in creation of video games [34]. For ease of integration with EM5 and other video games, the search environment was programmed in C++, making use of the Boost C++ libraries [11] and the aiTools library [45]. Importance was attached to the ability to wrap any search graph easily, without forcing the user to keep certain functionality in the nodes or the search graph description itself. By design, static search graphs, keeping the nodes in an unaltered structure and keeping a separate priority queue for node expansion as well as dynamic search graphs, that are expanded whenever a node is visited, are supported. Furthermore, costs, heuristic score and parent/child nodes are queried using a uniform interface, allowing the internal structure to either keep these values in the nodes for easier access, or even implement sophisticated caching and indexing structures for cost tables or the graph structure. The search graph and its nodes were generically set up to work with any underlying data types, allowing to switch from floating-point values to fixed point arithmetic at any point.

Costs of the path are measured in seconds, describing the time it takes a vehicle to follow the path. Besides the state space, each node thus holds a time in seconds describing the costs of the path to this point. Another field, the heuristic measure, provides the estimated time of the rest of the path. Also, as the search graph is generated on the fly, a pointer to the lowest-cost parent is held in each node, as shown in Figure 4.11. Both the actually calculated position for each successor and also the discretized value is saved in the node. This allows for sorting and selecting nodes from a grid cell, as well as following a planned path with higher accuracy. If another node is expanded in the same grid cell, the node with lower costs is preserved, just as in regular A*.

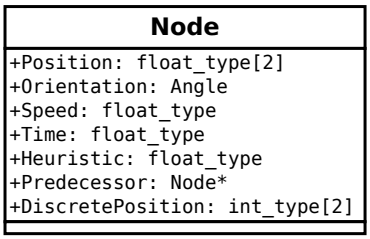

Figure 4.11: UML diagram of the Node class used for the A* search.

The default implementation makes use of a *boost::multi_index_container* for an open list, allowing to keep nodes sorted according to multiple criteria. This way, removing the node with lowest cost and heuristic score ($c+h$) can be achieved in $O(1)$, and searching nodes by their state space can be performed in $O(log(n))$ by using a binary search; n being the number of nodes in the open list.

4.5.1 EM5 Integration

Due to the modular structure of EM5s code, modifying and extending the game is made easy. The steering component can be replaced by a custom component, receiving the planned path and altering the curve segments in question before relaying the improved path to an encapsulated original steering component. Figure 4.12 shows how the local path planner class integrates within the existing EM5 infrastructure.

The code produced for this thesis research as well as the code used from the aiTools library is designed to work with arbitrary graph representations and used utility functions, as long as a common interface is used, easing the integration. In order to improve path layouts in EM5 without letting AI vehicles leave the roads and to comply with the EM5 reservation system, AIPATH is used as local planner for vehicles with a limited allowed deviation from the already planned path by the EM5 pathfinding system.

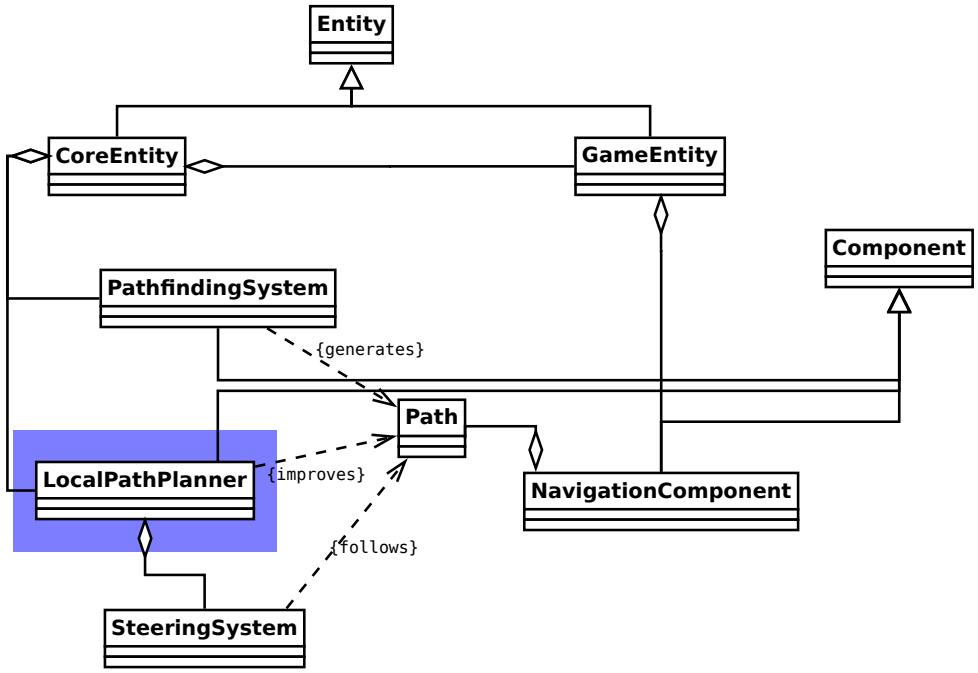

Figure 4.12: The main classes concerning steering in EM5 and their interactions. The blue shaded area marks the integration boundary between original and new code. The *PathfindingSystem* plans center line paths for vehicles, the *SteeringSystem* translates them to actual movement. Replacing the *SteeringSystem* by a local path planner, improving upon the path and then relaying the improved path to an encapsulated *SteeringSystem*, the path can be intercepted without changing the original program structure much.

Chapter 5

Experiments and Results

This chapter contains all experiments carried out in this research and their results. The quality of the generated paths is assessed visually and compared to paths produced by the work of Pinter and paths used in EM5. The quality of the heuristic evaluation function is compared to the distance-estimating approach by Pinter. Lastly, the results of the probing search are assessed.

Chapter contents: Experiments performed and their results.

5.1 General Test Setup

Most experiments are conducted with similar search setups. Unless otherwise specified, paths are planned by A* only, making use of the proposed traversal time estimation heuristic. Because the probing search might find suboptimal paths due to the non-monotonicity of the heuristic evaluation function, it is only included for probing search specific tests. For all tests performed, the goal test requires a node to be spatially within 0.3 discrete steps, the rotational deviation to be no more than one degree and a speed difference of at most $0.1^m/_s$. As the physical state space of a vehicle is not intrinsically bound, limiting the number of search iterations prevents aiPath from running out of memory. For all tests, the maximum allowed number of iterations is set to $100,000$ (10^5), after which a search is considered failed. All tests including random variables are repeated at least 1000 times in order to obtain meaningful averages. If paths to random locations are planned, typically all four variables of the goal node plus the velocity of the start node are randomized according to a uniform distribution. Furthermore, the turning radius of the vehicle and the lateral acceleration possible are selected randomly. Possible values for the turning radius, as derived from EM5 data, are between 5 and 8 meters. Lateral acceleration is bound by 10 and $20^m/_{s^2}$, allowing reasonable speeds in curves. Acceleration and deceleration values as well as speed limits are kept constant for all tests, with maximum acceleration $a_{max} = 1.5^m/_{s^2}$, deceleration limit $d_{max} = 5^m/_{s^2}$ and speed limit $v_{max} = 13.8^m/_{s^2}$. In order to plan interesting paths with respect to the turning radius used, goal positions are generated uniformly with a distance to the origin between 0 and 8 turning radii. Goal orientations are sampled from the whole possible range of orientations, from 0° to 360°. Velocities are, if not specified otherwise, generated randomly between 0 and v_{max}. The hardware in use for all tests is an Intel i7 quad-core processor with 32GB of RAM. EM5 internal tests are performed under Windows within the EM5 framework, all other tests are run under Linux.

5.2 Improvements in Path Quality

For varying input parameters, aiPath can reliably return physically valid paths. Input parameters may vary in search parameters, that is, start and goal node settings like position, velocity and orientation, or vehicular parameters such as turning radius, acceleration and deceleration limits and velocity limits. Figure 5.1 provides examples of planning a curve with limited turning radius. If the turning radius of the

vehicle surpasses the radius of the curve, an elaborate turning maneuver is planned rather than breaking physical limitations. As common path following methods are limited to follow the planned path, such flexibility in pathfinding is unachievable with path post-processing methods. Figure 5.2 shows planned paths for naively impossible situations, in which the vehicle has to accelerate to a goal speed that is too high to be reachable in the available distance. AIPATH solves the issue by planning a backup circle, to increase the distance traversed in order to be able to accelerate to the goal velocity. Note that, while the heuristic evaluation function estimates costs based on a circle and straight segments, as shown in Figure 5.2 on the far right, AIPATH is able to find better paths even if restrained by the lateral acceleration limit. Instead of planning a curve with uniform diameter, a curve with growing radius is planned. As the vehicle starts with zero velocity, the first planned steps can only be traveled slowly due to the linear acceleration limit of the vehicle. Here, the turning circle size is determined by the turning radius. After the vehicle gains speed, the turning circle radius gradually increases, in order for the vehicle to stay under the lateral acceleration limit.

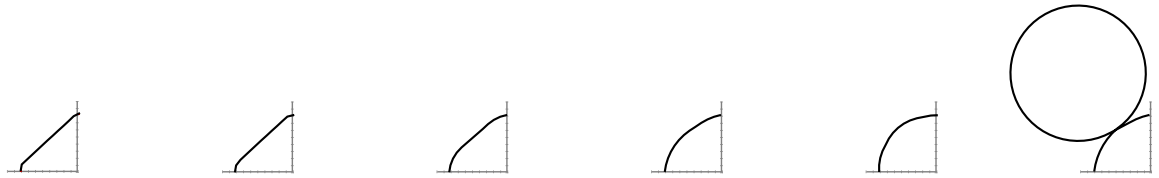

Figure 5.1: Planning a curve with increased turning radius. When the turning radius exceeds the radius of the curve, a backup circle is planned, allowing to reach the goal without violating physical constraints.

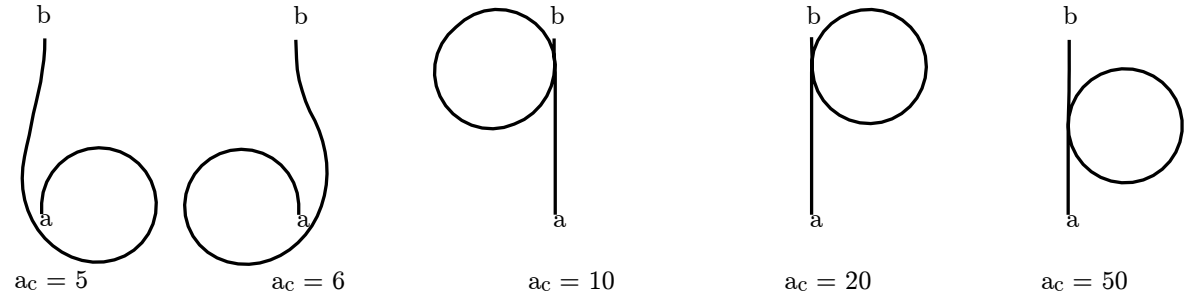

Figure 5.2: Deliberately planned backup circles on the path from a to b. Acceleration limits and the target velocity at b are set such that the vehicle is unable to reach the goal in a straight line. For different lateral acceleration limits a_c, different backup circles are planned. With $a_c = 50$, the centripetal acceleration limit has no effect on the path. At lower lateral acceleration limits, the effect is more noticeable.

5.2.1 Visual Path Quality Comparison

Fixing the speed and disabling lateral force limits, search parameters as used by Pinter [41] can be replicated. Searches with random layouts show that both approaches deliver visually comparable search results. Figure 5.3 displays an overlay of paths planned by both approaches under equal circumstances. It is to be noted that Pinter's algorithm relies on expanding the next 8, 24 or 48 neighbors, making certain angled straight lines impossible to plan. Effects of this restriction can be found in the middle of the path in Figure 5.3, where the green path deviates to the right seemingly without reason. AIPATH suffers from no such restrictions, unless introduced on purpose by discretizing the direction more coarsely. Other, smaller deviations between both approaches are to be attributed to the different discretization techniques used. Unlike Pinter's implementation, AIPATH is not constrained to the limitations imposed by pre-calculated lookup tables for heuristic values and grid cells crossed, allowing to vary turning radii at runtime. Naturally, AIPATH also allows planning optimal velocities already during path planning.

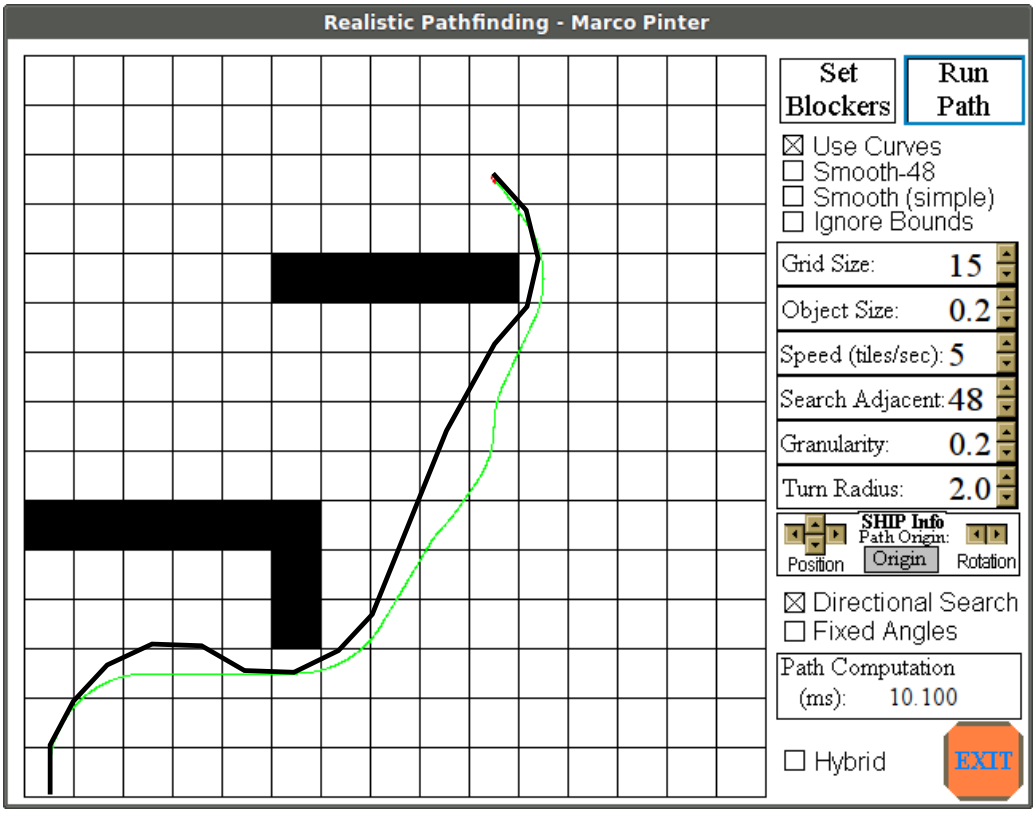

Figure 5.3: Overlay of a path planned by AIPATH (black) over a path planned by Pinter's approach (green). Equal spatial and directional discretization settings as well as equal turning radii and obstacle grids were used. Path smoothing can improve the visual appearance for both paths.

5.3 Testing in EM5

While EM5 has a working pathfinding and steering routine, generated paths can still be improved by employing AIPATH. The EM5 pathfinding system relies solely on static information in the graph on how to plan curves, i.e., vehicular constraints and differing speeds are not taken into account during planning time. Traffic should obey traffic laws and follow distinct lanes, so neither the EM5 pathfinding nor the steering system, producing a behavioral sound vehicle, can be replaced without further action. To be able to benefit from AIPATH, it is implemented as a local path planner, improving curves while still following each lane, and still employing the original EM5 steering system to obey traffic laws.

A local planner can improve path optimality locally, as it can plan to cut curves slightly or taking slightly wider curves in order to fulfill physical constraints of the vehicle. These optimized curve shapes may, depending on the speed and lateral acceleration limits, be traveled faster than the originally planned curve. As an additional benefit, slight variations in driving behavior let the traffic seem less sterile and artificial. For every road segment, EM5 not only provides the center line, but also possible deviations to both sides. Figure 5.4 shows a typical right curve. The blue shaded region marks the area which the vehicle may use to navigate the curve. Some grid cells are illustrating a discretized representation of blocked and free cells. Alternatively, AIPATH allows custom functions to assess child node validity, making the construction of an extra blocked grid superfluous. A custom validity check function can measure the distance of the child node position to the ideal line provided by EM5, and reject all nodes outside the blue, allowed area.

To allow cutting corners or purposely selecting a wider radius, multiple start and goal nodes are provided. Each node deviating from the center line receives a cost penalty, as the vehicle has to drive a slightly longer diagonal in the segment before. Figure 5.5 illustrates the increase in distance before reaching the curve. The additional time spent on the preceding segment used to initialize the costs of the deviating starting points.

The penalty is calculated with an estimate of the traversal time, assuming the previous area was

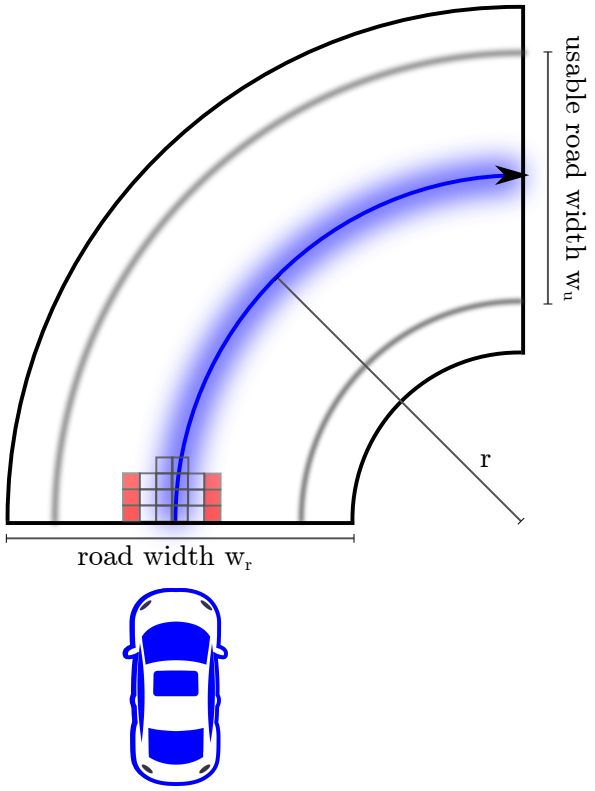

Figure 5.4: A 90° right curve, illustrating the available parameters to plan an EM5 curve:

- Blue center line: Supplied by EM5 as naive optimal trajectory.
- r, the radius of the curve.
- w_r, the width of the road as supplied by EM5
- w_u, the maximal possible distance to deviate from the center line before the vehicle may collide; depends on the vehicular dimensions.
- Blue shaded region: Possible vehicular center points during curve traversal.
- Grid cells: Transparent if unblocked, red if blocked due to obstacles or vehicle dimensions.

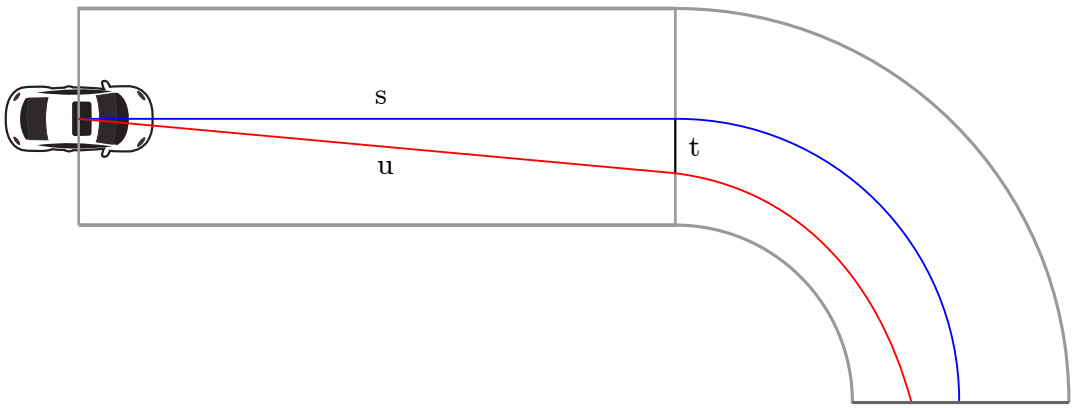

Figure 5.5: Blue: The ideal line provided by EM5. Red: a possible planned trajectory with deviation t from the center line. The length of the segment preceding the curve to be planned, s, and the lateral translation from the ideal line, t, form a right-angled triangle with hypotenuse u. The difference in estimated path traversal time, $h(u) - h(s)$ is added to the costs of the deviated starting point.

entered on the center line and continuously traveled at maximum velocity towards the starting node of the curve segment. Figure 5.6, right, shows possible deviations from the EM5 ideal curve shape, both purposely selecting a wider turning radius and cutting the curve. This test modifies the usable road width w_u, to produce different levels of curve cutting, simulating different widths of vehicles. Figure 5.6, left shows planned curve paths with varying usable road width, leading to different levels of curve cutting. Because the planned curves respect turning radius and lateral acceleration limitations, they may be traveled slightly faster than following the circle perfectly. Table 5.1 presents traversal time improvements for the planned trajectories. A decrease of traversal time up to 31.57% is only possible because the newly found curve has an improved shape, and thus can be traveled faster due to less lateral acceleration.

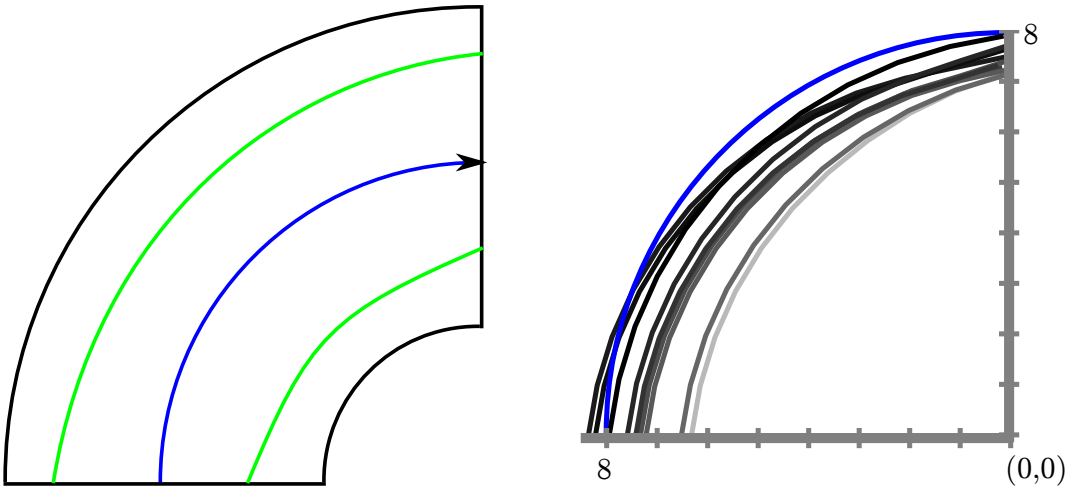

Figure 5.6: Left: An example of curve cutting and one of choosing a deliberately wider turning radius. Right: Resulting paths in an 8 meter curve with changing usable road width. Blue: The ideal line as used by EM5. Black: Minimal usable road width. Grey shades: Increased usable road width allows shorter paths.

Type	Length	Travel Time	% Improvement
EM5 Ideal Line	12.56	0.89	0 %
w_u 0.6 Meters	12.68	0.73	18.04%
w_u 1.2 Meters	13.38	0.78	12.46%
w_u 1.8 Meters	13.62	0.73	18.04%
w_u 2.4 Meters	12.39	0.65	26.84%
w_u 3.0 Meters	12.39	0.72	18.95%
w_u 3.6 Meters	12.32	0.67	24.54%
w_u 4.2 Meters	12.24	0.64	27.72%
w_u 4.8 Meters	12.16	0.71	20.42%
w_u 5.4 Meters	11.69	0.64	28.38%
w_u 6.0 Meters	11.59	0.61	31.57%

Table 5.1: Improved path lengths and travel times with increased w_u, the usable road width. Distance and time penalties for diverting from the center on the previous segment are included. A 30 meter straight street segment is assumed.

5.3.1 Performance Feasibility Testing

EM5 maps feature mostly similar junctions and road intersections, only few are designed by hand to resemble famous landmarks in real world cities. The most commonly used curves in intersections are shown in Appendix A.3. Using these curve, searches with varying turning radii, velocity and lateral acceleration limits are performed to estimate the average workload introduced by aiPath. In this test, start and goal velocity are selected to be equal.

For each of the four most common curve types, 1000 paths were planned, with a total average number of iterations of 1098. Table 5.2 shows results for each curve type, and Appendix A.4 provides histograms illustrating the deviation of number of iterations in detail. The histograms clearly show that the majority of path searches take well below 1000 iterations, only a few outliers skew the average.

Curve Type	Curve Radius	Average Number of Iterations
Short Right, A.6	6 meters	230
Short Left, A.7	13 meters	1161
Long Right, A.8	8 meters	678
Long Left, A.9	15 meters	2323
All		1098

Table 5.2: Average number of iterations to plan common EM5 curves with varying desired velocities, acceleration limits and vehicle turning radii.

The EM5 navigation systems currently consumes a significant amount of calculation resources, up to 25% of the whole workload, in certain situations. Considering that the steering system alone used 5 milliseconds to update in early EM5 versions, it seems natural to impose a 5 millisecond update time for aiPath as well. However, like in many video games, the level of detail is scaled in such a way that maximal detail is possible on a target machine. Adding a highly complex movement planning system to EM5 in its current state is certainly computationally infeasible. For a path search taking 1000 iterations, the generic proof of concept implementation of aiPath uses 10.8 milliseconds on the testing hardware. Figure 5.7 provides a histogram of search time deviations. The majority of searches take about8 milliseconds, only a small percentage of searches takes longer than 10 milliseconds. These outliers appear to be normally distributed around 27 milliseconds. A possible explanation for the outliers can be found in the way the open list is implemented. Being a sorted, heap-like structure, the order of insertions into the open list matters. It is possible, that for a small percentage of searches, a costly restructuring or re-allocation procedure has to be invoked. Improving the open list implementation may thus eliminate the outliers. Still, with a median path search time of 8 milliseconds, only one path can be planned every two updates, allowing only 25 path queries per second. Using the generic, single-threaded implementation seems to be infeasible for EM5. Implementing aiPath efficiently for a specific game, and optimizing the implementation for calculation speed, significantly reduced overhead is expected [25][42].

In EM5, most calculations are performed sequentially on a single core. A few specialized systems, for example the pathfinding system, transfer their calculations into a separate thread, to make use of multi-core processors. However, EM5 typically does not utilize a multi-core machine at full capacity. When outsourcing aiPath to another CPU core and reducing planning latency by following the originally planned path before the planned improvements are returned, aiPath can be successfully integrated as local movement planner. Figure 5.8 shows vehicles in EM5 following curves planned by aiPath. On a quad-core processor almost exclusively running EM5, enough CPU resources are idle to allow integration of aiPath with close to zero calculation overhead. The only overhead present in EM5 is the asynchronous waiting for the planner to return, which is negligible.

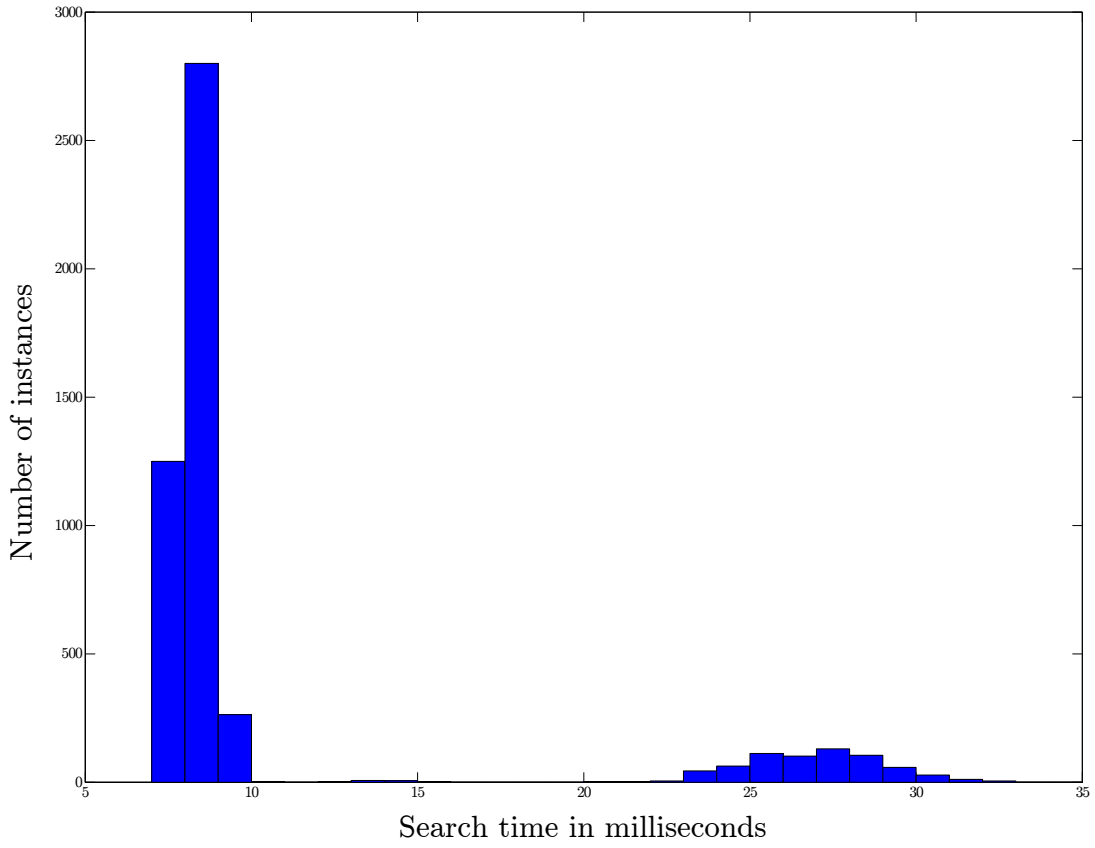

Figure 5.7: Histogram of the wall clock search time in milliseconds, obtained from 5000 searches with 1000 iterations each.

Figure 5.8: Vehicles planning their trajectories physically correct in EM5 curves.

5.4 Quality of the Heuristic Evaluation Function

The quality of the heuristic evaluation function used for aiPath's underlying A* algorithm is assessed on multiple levels. First of all, admissibility is verified. The estimated costs returned by the heuristic function are evaluated visually against the values produced by the planner, resulting in an estimate of how well the evaluation function represents the actual planning domain. The heuristic evaluation function is then tested against the distance estimation function used by Pinter by performing A* searches with both heuristics.

Admissibility

In the ideal case, heuristics should accurately describe the actual costs for the rest of the path. Typically though, obstacles may occur on the path, which are left unaccounted for by the heuristics. In order for A* to provide an optimal path, any used heuristic has to underestimate the remaining path costs, i.e. at most return the actual costs. The heuristics used for aiPath, described in Section 4.2, first calculates the shortest possible trajectory between two points, and subsequently produces the ideal speed profile on the path. As the traversal time estimation uses the physical acceleration and deceleration limitations of the vehicle, it can guarantee that the vehicle cannot arrive in less time than estimated. The heuristic evaluation function does, however, ignore the limited centripetal acceleration of a vehicle. A limited centripetal acceleration may force the vehicle to select a wider than minimal turning radius, and thus can produce a longer path. Any possible speed increase due to a wider turning circle increases traversal time as well, because the increase in distance traveled is always higher.

$$t_{circle}(r) = \frac{2 \cdot \pi \cdot r}{\sqrt{a \cdot r}} \tag{5.1}$$

Using Formula 2.4 for centripetal force and the circumference formula for a circle, the time pass a whole circle with a given centripetal acceleration limit is given by Formula 5.1. As this is a monotonically increasing function, the ideal circle to follow is always the smallest turning circle possible. Thus, ignoring lateral acceleration limits can only lead to underestimating path length. Because the heuristic disregards the discretization of the state space, it is not consistent and thus does not fulfill the triangle inequality, an otherwise desirable property [40]. Not fulfilling the triangle inequality also does not guarantee a successful result of the probe search step even in the absence of obstacles, as it may get stuck in a local minimum.

Visual Assessment

Using the search setup described in Figure 5.9, the heuristic evaluation function was plotted. Performing searches to all possible goal positions provides the actual minimal costs to arrive at the goal position respecting discretization and lateral acceleration limits. Figure 5.10 shows the results for the heuristic evaluation function, and is to be compared to Figure 5.11, which shows the plotted results obtained from the actual search. The effect of discretization is quite noticeable, also the *action duration discretization* presented in Subsection 4.3.5 can be noticed. A path close in length to an integer multiple of the discretization cell size is found cheaper than neighboring paths. Because of this, a path of one discretization cell size is the shortest possible path, any closer point has to be reached by at least two discrete steps. Differences in colors are to be attributed to different scaling of the color space; as the search requires at least one search iteration, no zero-cost paths can be found without introducing an additional subroutine. Still, the heuristic evaluation function resembles the actual path costs visually quite well, and thus high quality paths are expected.

Comparison to Distance Estimation Heuristic

To compare aiPath's heuristic evaluation function to estimating only the distance, paths were planned to random locations with random orientation and velocity. Different car setups with random turning radii in the range of 5 to 10 meters were used, a range inspired by EM5 vehicles, but also sensible for real world applications. As clearly visible in Figure 4.6, the influence of a turning radius on the heuristic function diminishes for evaluated positions further away than four times the turning radius. For this reason, random goal positions were chosen within the range of 0 to 8 turning radii, to incorporate varying effects of the turning radius. Arrival orientations were selected uniformly on the set of 0° to 360°. Table 5.3

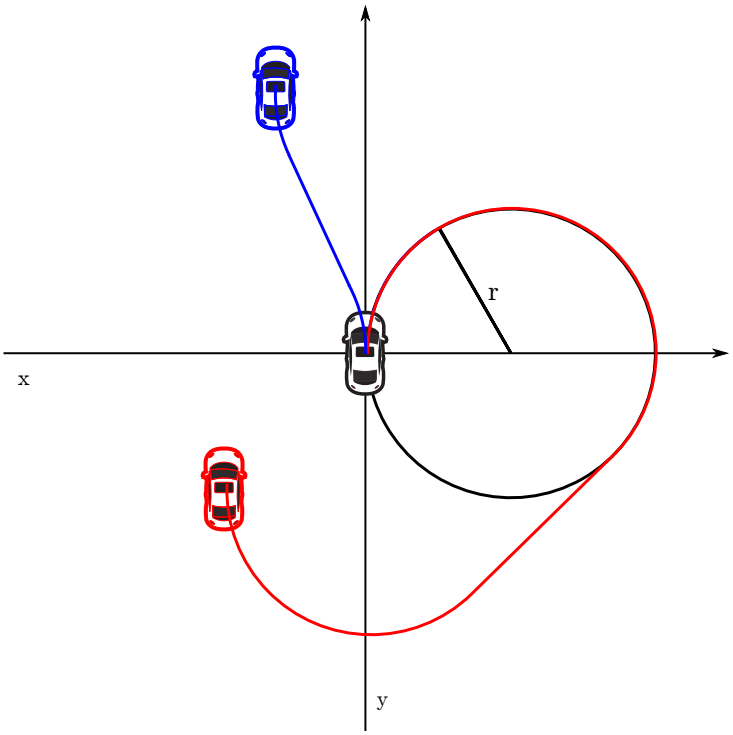

Figure 5.9: Search setup to plot the heuristic evaluation function: A vehicle positioned in the origin, with $\phi = 90°$ heading and zero velocity plans a path to any other point on the plane in a certain region with the same goal heading and velocity. The path traversal time is indicated through color information at the goal position.

shows a significant reduction in search iterations when improving Pinter's spatial heuristic evaluation function with the travel time estimation introduced in Subsection 4.2.2. On average, using the traversal time heuristic improved search iteration count by 97.52%. Shorter paths feature less improvement, with a possible explanation that the velocity varies less on shorter paths, due to the acceleration and deceleration limits. Using Euclidean distance as a heuristic has proven to be infeasible for the problem due to memory limitations of the hardware used. Table 5.4 also shows a slight improvement in path traversal time. While the distance based heuristic merely ignores the traversal speed, AIPATH uses a static move ordering with a bias towards acceleration and keeping speed, producing high quality paths even when ignoring the velocity for the heuristic evaluation function. Because of this static move ordering, the traversal time estimation heuristic only improves upon the distance estimation heuristic on average by 2.50%. Appendix A.5 contains histograms showing the distribution of improvement percentages for both iteration count and path length.

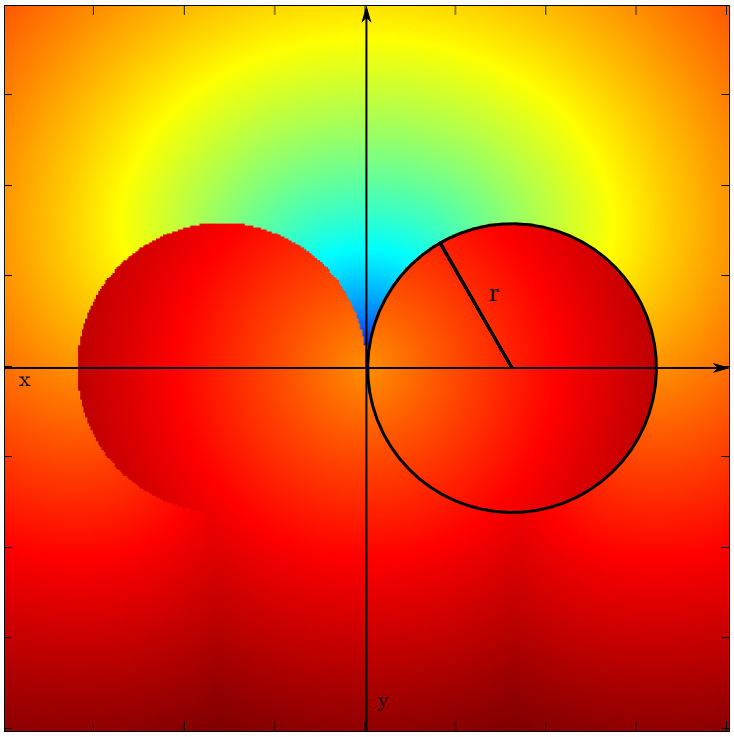

Figure 5.10: A heatmap style scatter plot showing the heuristic evaluation of any position in the (x, y) plane in a certain range, with $\phi = 90°$ i.e. facing in positive y direction, and zero velocity. The turning radius r used for the vehicle is indicated.

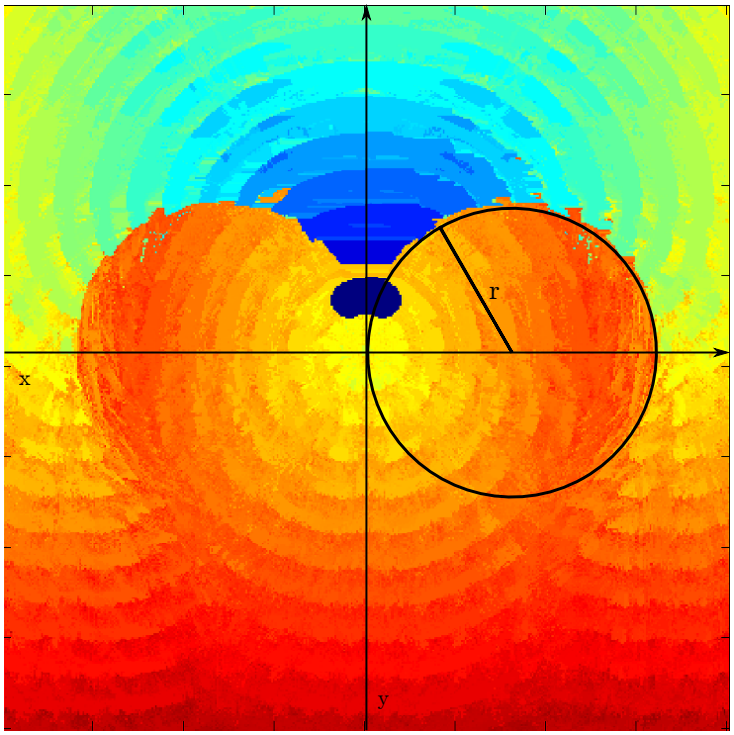

Figure 5.11: A heatmap style scatter plot showing the actually found costs by AIPATH to any position in the (x, y) plane in a certain range, with $\phi = 90°$ i.e. facing in positive y direction, and zero velocity.

Average Number of Iterations			
Euclidean Path Length (Quantity)	Distance-Based	Time-Based	Reduction by
0 - 10 (167)	13651	842	93.83%
10 - 20 (524)	18961	705	96.28%
20 - 30 (811)	21393	868	95.94%
30 - 40 (884)	35344	585	98.34%
0 - 50 (467)	48894	732	98.50%
50 - 60 (160)	49854	1051	97.89%
60 - 70 (47)	51566	1160	97.75%
All (3060)	34238	849	97.52%

Table 5.3: Average number of search iterations of distance and movement time based heuristics. Calculation cycles were reduced on average by 97.52% compared to the distance based heuristic. Averages were obtained from 3060 path searches to random locations.

Average Traversal Time			
Euclidean Path Length (Quantity)	Distance-Based	Time-Based	Reduction by
0 - 10 (167)	12.53	11.89	5.10%
10 - 20 (524)	12.19	11.60	4.78%
20 - 30 (811)	11.94	11.66	2.38%
30 - 40 (884)	12.47	12.32	1.20%
40 - 50 (467)	14.29	14.07	1.51%
50 - 60 (160)	16.32	16.04	1.73%
60 - 70 (47)	17.98	17.69	1.59%
All (3060)	13.96	13.61	2.50%

Table 5.4: Average path traversal times achieved by distance and movement time based heuristics. Using time based heuristics, traversal time was reduced by 2.50% compared to the result obtained with distance based heuristics.

5.5 Discretization Resolution Results

Discretization resolution has an impact on both path quality as well as search iteration count. A coarse discretization allows larger step sizes and reduces the number of nodes expanded, but it also increases discretizational error, resulting in possibly longer paths. Path searches to random goal positions in the vehicle state space with different discretization settings provide an insight into optimal discretization settings for the search configuration used.

5.5.1 Spatial Discretization Tests

The effect of the grid resolution used for spatial discretization on path length and search iteration count is evaluated in this test using searches to 1000 random locations with varying discretization resolution. Figure 5.12 shows resulting average number of iterations during search and path length plotted against the grid resolution used for spatial discretization. Note that both path length and number of iterations increase slowly beyond the point where the optimal resolution was reached; a steeper gradient is expected. This is to be attributed to the fact that failed searches, i.e., searches which take longer than a user defined number of steps, are not counted, as the real iteration count is unknown. Increasing problem difficulty by increasing discretization coarseness leads to more failed searches. These failed searches, already taking up more iterations than allowed, are most likely also longer than succeeding paths. Stripping these paths explains the slow rise in both path length and iteration count for coarser discretization values. Figure 5.14 shows the rising percentage of failed paths for coarser discretization resolutions. Table 5.5 contains the number of iterations for given cell sizes and their standard deviation, both of which feature a common minimum at a discretization cell size of 1.2. Table 5.6 contains the same information for the path length. Note that the standard deviation of path length rises more quickly than the average path length with an increase of cell size, as also shown in Figure 5.13. This is to be attributed to the fact that a coarser discretization may introduce certain shortcuts, but also generally lowers path quality, increasing their length.

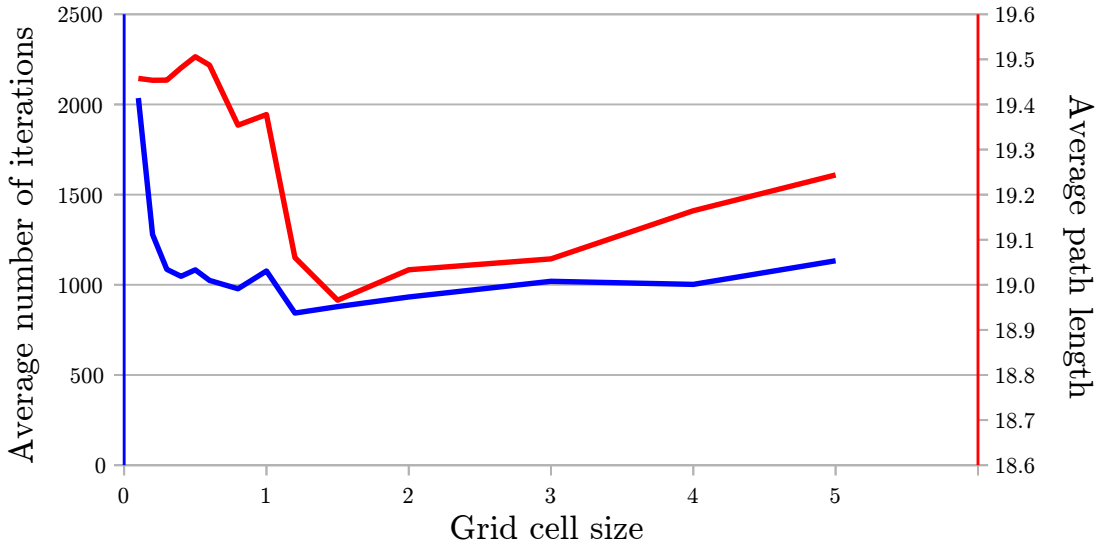

Figure 5.12: Plotting average number of search iterations (blue) and path length (red) against spatial discretization grid size. The average number of iterations is expectedly higher for finer discretization resolutions, then grow as the search problems become more difficult to solve because of the more limited possibilities. Note that a discretization grid cell size of 1.5 seems to be optimal for this search setup.

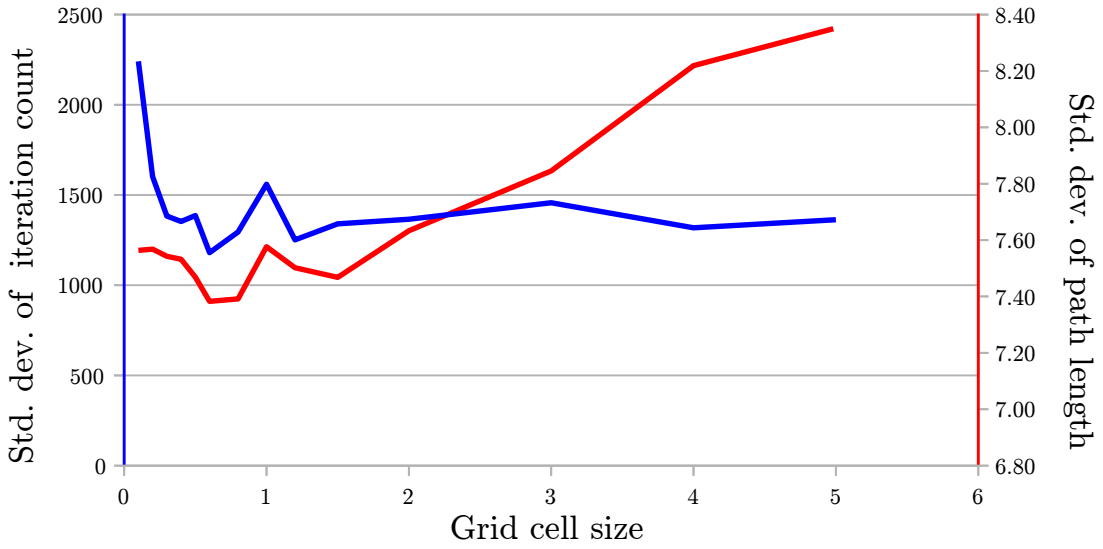

Figure 5.13: Standard deviations of search iterations (blue) and path length (red) plotted against spatial discretization grid size. Note that the standard deviation of path length increases significantly for grid cell sizes greater than 1.5.

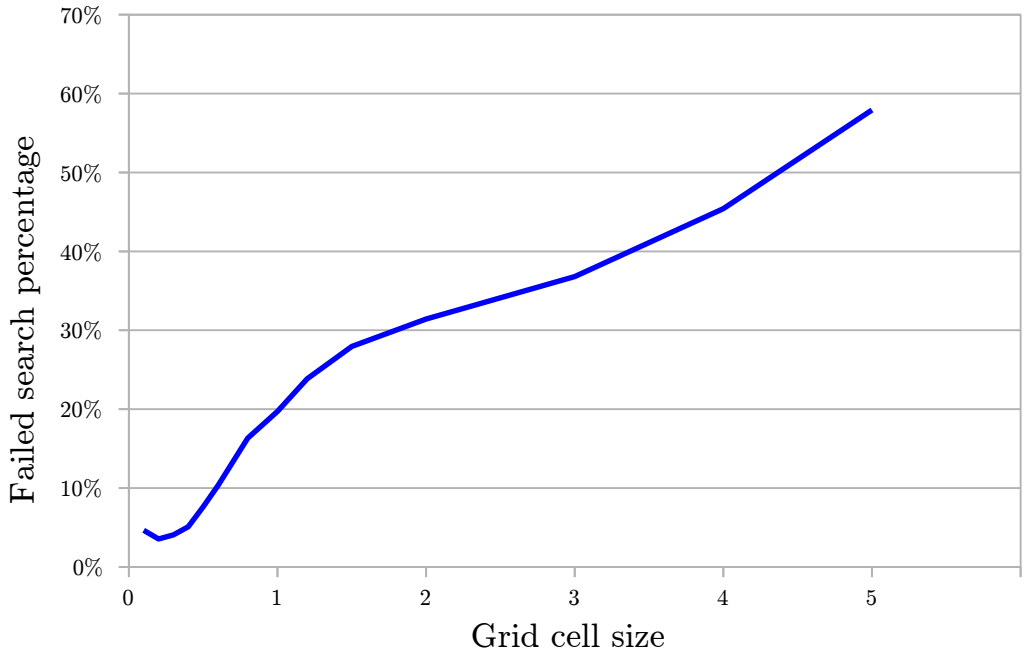

Figure 5.14: The percentage of failed paths plotted against spatial discretization grid size retrieved from 1550 path searches to random locations. With growing cell size, more paths fail due to the added complexity.

5.5.2 Directional Discretization Tests

Selecting optimal rotational discretization values is not as straightforward as finding an optimal positional discretization. Figure 5.15 illustrates that, even for a comparatively fine rotational discretization using 10°, the turning radius of the vehicle can disturb path planning as it requires rotational adjustment in a very precise manner. Still, reducing the rotational resolution from 1° to 2° lowered the number of search iterations from 22868 to 5307, or by 76.79%, with a negligible change in path length. Optimal discretization parameters should be selected for every distinct search problem, to provide reduced search times and optimal path quality.

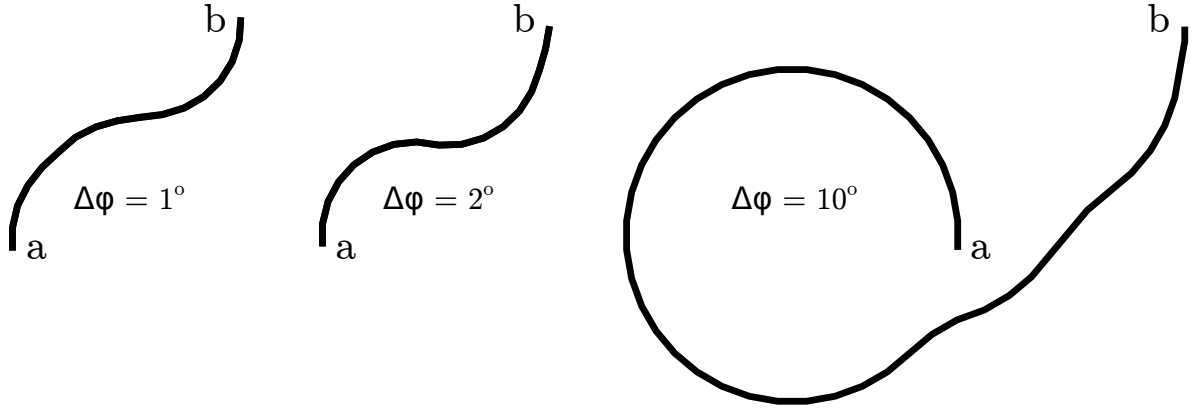

Figure 5.15: Planned paths from a to b with rotational discretization coarseness value $\Delta\phi$ at 1°, 2° and 10°. Beyond 2°, the discretization conflicts with the turning radius, resulting in suboptimal paths.

Cell Size	Avg. # of Iterations	Iteration Count Std. Dev.	Iteration Count Change
0.1	2036	2241	-89.11%
0.2	1279	1602	-18.82%
0.3	1085	1383	-0.83%
0.4	1047	1353	2.75%
0.5	1083	1386	-0.60%
0.6	1024	1180	4.86%
0.8	978	1295	9.14%
1.0	1076	1560	0.00%
1.2	843	1252	21.66%
1.5	879	1340	18.33%
2.0	933	1366	13.36%
3.0	1020	1457	5.28%
4.0	1002	1318	6.88%
5.0	1133	1363	-5.29%

Table 5.5: Average number of iterations for varying spatial discretization grid sizes. Iteration count changes are based on the unit grid cell resolution with cell size 1. For each discretization resolution, the same 1000 planned paths to random locations provide the average values. Note that both the iteration count and its standard deviation feature a minimum at discretization cell size 1.2.

5.6 Probing Search Results

The probing search step of aiPath uses a non-backtracking best-first search algorithm to provide a physically correct, preliminary path that can be followed during path-planning using A*. The probing search is thus assessed for performance and path quality in comparison to A*. Using a search setup comparable to the one used in measuring the heuristic evaluation function performance as explained in Section 5.4, the probing search was used to plan paths to various locations on the plane. Figure 5.16 shows

Cell Size	Avg. Path Length	Path Length Std. Dev.	Path Length Change
0.1	19.46	7.56	-0.41%
0.2	19.45	7.57	-0.39%
0.3	19.45	7.54	-0.39%
0.4	19.48	7.53	-0.53%
0.5	19.51	7.47	-0.66%
0.6	19.49	7.38	-0.57%
0.8	19.35	7.39	0.12%
1.0	19.38	7.58	0.00%
1.2	19.06	7.50	1.64%
1.5	18.97	7.47	2.12%
2.0	19.03	7.63	1.78%
3.0	19.06	7.85	1.65%
4.0	19.16	8.22	1.10%
5.0	19.24	8.38	0.69%

Table 5.6: Average path lengths for varying spatial discretization grid sizes. Path length changes are based on the unit grid cell resolution with cell size 1. For each discretization resolution, the same 1000 planned paths to random locations provide the average values. The standard deviation of path lengths rises noticeably with increasing discretization cell size.

that the probing search finds a path to at least the grid cell neighboring the goal every time, and that in most cases, sub-cell resolution precision can be achieved. As discretization parameters should typically be chosen in a way that one cell with deviation is negligible, using only the probing search is a viable option in absence of obstacles. Table 5.7 shows improvements in the number of search iterations for path searches to 1446 random locations using both A* and probing search. On average, the probing search uses about 99.42% less iterations to produce a path. The distribution of improvement percentages is displayed by Figure A.16. As the heuristic evaluation function in use is admissible, but not consistent, the probing search is not guaranteed to find the shortest path. Table 5.8 shows that especially for relatively close goals, the probing search tends to construct longer paths than necessary. Obtained from 4323 searches to random goals, an average traversal time overhead of 11.87% is found for the probing search. Figure A.17 provides a histogram of the traversal time overhead deviation. Even though the probing search is not optimal, the resulting behavior is always comprehensible, as it follows the heuristic evaluation function in every situation.

Average Number of Iterations			
Euclidean Path Length (Quantity)	A* Search	Probing Search	Reduction by
0 - 10 (79)	1442	25	98.25%
10 - 20 (247)	3961	34	99.14%
20 - 30 (383)	6770	39	99.42%
30 - 40 (418)	7653	44	99.43%
40 - 50 (222)	9141	49	99.47%
50 - 60 (75)	15140	56	99.63%
60 - 70 (22)	11152	73	99.34%
All (1446)	7894	46	99.42%

Table 5.7: Average number of search iterations of A* and probing search. Calculation cycles were reduced on average by 99.42% compared to A*. This average improvement is the result of 1446 searches to random locations.

Average Path Traversal Times				
Euclidean Path Length (Quantity)	A* Search	Probing Search	Probing Overhead	Overhead Std. Dev.
0 - 10 (55)	2.46	15.37	83.94%	26.70%
10 - 20 (307)	7.22	11.04	34.51%	15.78%
20 - 30 (798)	12.00	14.50	17.19%	7.55%
30 - 40 (1522)	16.82	19.11	11.98%	5.35%
40 - 50 (1266)	21.29	23.31	8.65%	8.43%
50 - 60 (337)	25.62	27.41	6.51%	2.47%
60 - 70 (38)	31.70	33.21	4.53%	2.62%
All (4323)	17.19	19.64	12.44%	11.87%

Table 5.8: Average path traversal times produced by planning 4323 paths to random locations using A* and the probing search. The probing search produces suboptimal results especially on short paths, with an overhead of up to 83.94%. On longer paths, the overhead drops, and with it, its standard deviation.

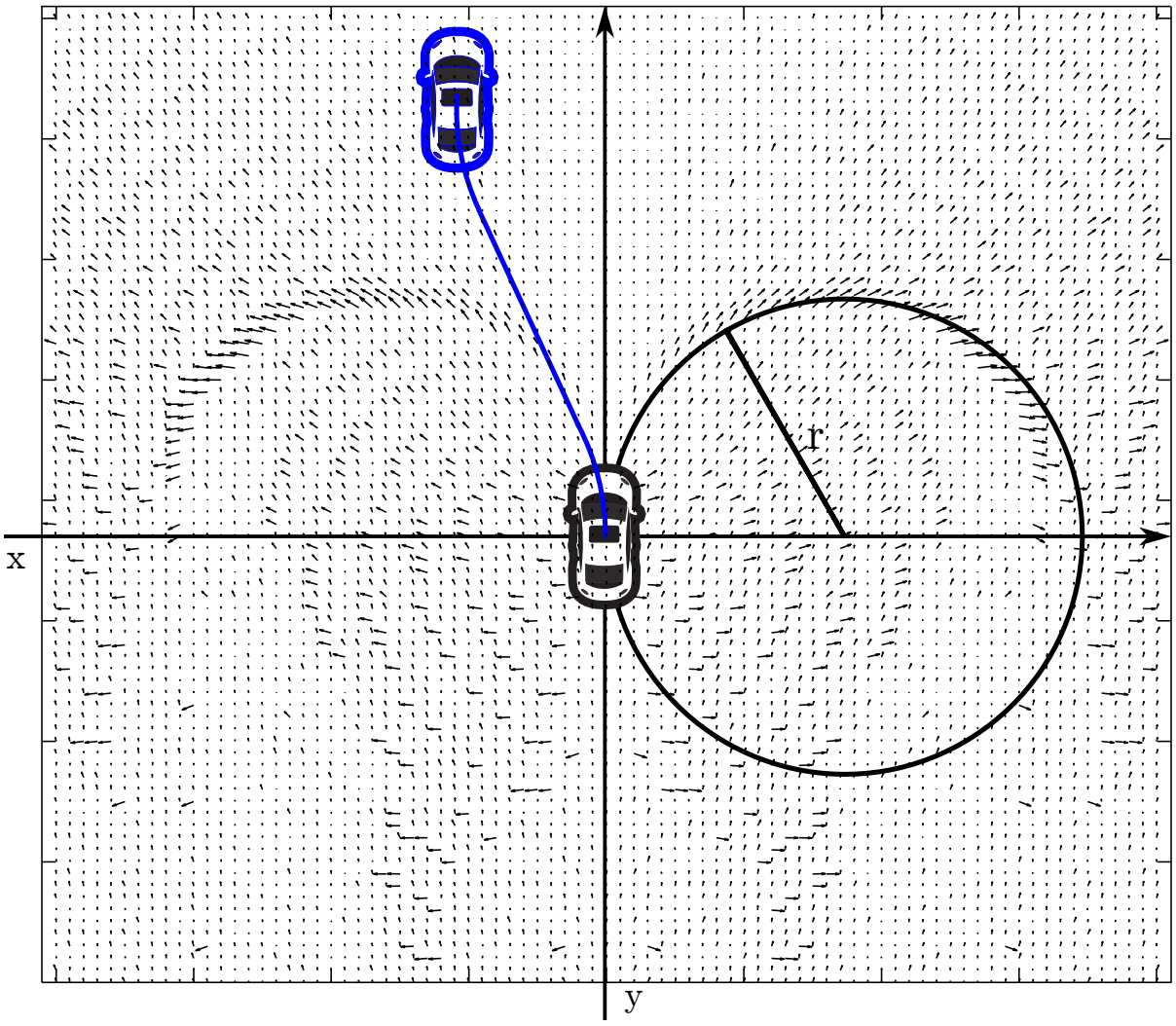

Figure 5.16: Deviations between probing search end positions and goal positions plotted as arrows pointing from found position to target position. Most deviation can be found at the rim of the turning radius, as the heuristic evaluation function is discontinuous there.

Chapter 6

Conclusions

Searching a physically correct path using aiPath *in a multidimensional vehicle state space has proven to be a viable technique to generate realistic paths. This chapter answers the research questions, presents general conclusions and examines possible enhancements and future research.*

Chapter contents: Conclusions about aiPath.

6.1 Answering Research Questions

aiPath is designed to answer the research questions positively. This section analyzes whether the results confirm that aiPath meets the demands. To reiterate, the first research question is formulated as:

> *Can physical movement planning approaches provide additional depth in gaming by providing more realistic movement?*

aiPath can indeed improve the quality of planned paths compared to approaches not regarding physical constraints. More realistic movement is implied by the physical feasibility of the path, other improvements regard path traversal time, which, rather than path length, is optimized using aiPath. The approach is robust, because the underlying A* algorithm is guaranteed to find a possible path if one exists, as long as the heuristic evaluation function in use is admissible. While path post-processing techniques face natural limitations if unforeseen circumstances arise, aiPath can find alternative solutions. Path post-processing techniques most commonly have to resort to breaking physical constraints, for example by forcing the vehicle into a smaller turning radius than possible. Situations in which a locally steered vehicle may get stuck, or in which a vehicle is given a seemingly impossible task of accelerating in a distance too short for its acceleration limit can be resolved by using aiPath. Flexibility in finding solutions is inherently present due to the searching nature of the underlying A* algorithm; where steering routines can only improve paths, aiPath can find the true shortest path regarding physical limitations.

Integrating aiPath as a local planner into EM5, variations in movement for differing vehicle types can be achieved. It is to be noted that, even though random deviations are much simpler to implement, aiPath always guarantees physical possibility as it searches in the physical state space of the vehicle directly. As an additional benefit, paths planned by aiPath are in general faster to traverse than original EM5 curves.

The second research question concerns the algorithmic complexity of physical path planning and whether the approach can be computationally feasible:

> *Can pathfinding techniques be enhanced to provide more realistic movement, while still being computationally feasible?*

The computational complexity of A* depends heavily on the branching factor, which in turn depend on the discretization parameters used. A finer discretization generates more successor nodes, but provides

smoother paths and may produce solutions missed by a coarser discretization. A certain range of discretization coarseness values can produce aesthetically pleasing paths within reasonable planning time and memory bounds.

For hundreds of simultaneously moving vehicles in EM5, aiPath is deemed to be computationally too complex. EM5 already spends a large share of CPU time on navigation, meaning that any increase in navigation computations become infeasible quickly.

However, EM5 does not use all CPU cores to their full potential, allowing lazy evaluation of path planning requests by aiPath. While the vehicle is following the path provided by the default EM5 pathfinding system, aiPath may compute a higher quality, physically correct path in a separate thread. Upon completion, the inferior path can be replaced. Performing a probing search step before A* search, making use of the heuristic evaluation function to provide a physically valid path, implements a similar concept. During path following, a complete path can be planned concurrently in a separate thread. Using aiPath for AI controlled vehicles can enhance realism of produced paths and driving behavior.
Concluding this research, the problem statement is reviewed:

> *In order to ensure believability of characters in a simulated environment, current pathfinding techniques have to be modified to incorporate planning with physical limitations of the controlled vehicles.*

Planning a path concerning the physical limitations of the controlled vehicle can be achieved by aiPath, using A* with a domain-aware heuristic evaluation function on the physical state space of the vehicle. aiPath can therefore be used to improve believability of characters in simulated environments, and thus convey immersive experiences while playing.

6.2 General Conclusions and Recommendations

As A* is an optimal algorithm, it searches until either a path is found or it is proven that there is no path. The latter requires an expansion of all search graph nodes, which is infeasible in the case of possibly quasi-infinite physical domains. In order to achieve good performance even in search failure cases, the number of iterations may be limited to a number scaling with Euclidean distance or heuristic distance between the start and goal node. It is also advised to prohibit node expansion strictly from areas the vehicle is not meant to navigate. In a game setup using urban traffic, this may mean to discard all nodes which are expanded off road. This limitation may, however, cause vehicles to turn or navigate more restricted than a player is. In general, care has to be taken to constrain the search using finely tuned discretization parameters and a good heuristic function modeled after the state space description for the vehicle type in question.

In many computer architectures, integer calculations are still faster than floating point calculations. As the state space used for searching and representing nodes is discretized anyway, fixed point arithmetic may improve both calculation time as well as reproducibility of results, as floating point calculations are prone to provide inaccurate results.

Usage of aiPath beyond Computer Games

In the field of robotics, using a two planner setup in order to improve performance may be necessary, especially if the robot deviates from the path and needs to re-plan. A global planner, set up with a coarser discretization, generates a rough plan towards the goal. Using slightly more restrictive physical parameters for the global planner makes sure it is possible to follow the path even in case of minor deviations. A local planner is then used to plan short segments ahead of the robot to any point on the global path. If the robot deviates from the path, only the local plan has to be revised, rather than a complete path to the goal.

6.3 Future Work

Testing aiPath in various domains, from different computer game genres and different vehicle types to testing with live robots is a task to be performed in future work. As discretization settings heavily influence the planning performance as well as the path performance, machine learning techniques assisted

by automated tests in the actual problem domain may aid game designers in choosing the appropriate trade-off between path quality and performance. Replacing the wrapped A* by an anytime search algorithm may provide higher quality paths with less overhead, as the vehicle may start following the path at an early stage, and the planner may continue to refine the path while the vehicle is following it. ARA* [32], an anytime A* variant, may be a viable alternative.

Future Work for EM5 If aiPath is widely used to plan paths for EM5 vehicles, performance should be improved. Naturally, many performance enhancing improvements can be made to any pathfinding algorithm. Commonly planned (sub–)paths may be cached per vehicle type, such as recurring curves in EM5, and using lookup tables or defining a piecewise heuristic evaluation function, as described in Subsection 4.2.3, can provide significant reduction in computation time. Lastly, not all background AI must necessarily perform physically correct movement. Limiting the usage of physical path planning for situations like car chases or user-commanded, but AI-controlled, vehicles, aiPath can add realism to especially the situations with importance to the player.

Bibliography

[1] Adzima, Joseph C. (2002). Competitive AI Racing under Open Street Conditions. *AI Game Programming Wisdom*, Vol. 1, pp. 460–471.

[2] Balldin, Ulf I. (2002). Acceleration Effects on Fighter Pilots. *Medical Aspects of Harsh Environments*, Vol. 2, pp. 1014–1027.

[3] Barraquand, Jerome, Langlois, Bruno, and Latombe, J.-C. (1992). Numerical Potential Field Techniques for Robot Path Planning. *IEEE Transactions on Systems, Man and Cybernetics*, Vol. 22, No. 2, pp. 224–241.

[4] Bartneck, Christoph, Kanda, Takayuki, Ishiguro, Hiroshi, and Hagita, Norihiro (2007). Is the Uncanny Valley an Uncanny Cliff? *In Proceedings of the 16th IEEE, RO-MAN 2007*, pp. 368–373, Jeju, Korea.

[5] Botea, Adi, Müller, Martin, and Schaeffer, Jonathan (2004). Near Optimal Hierarchical Path-Finding. *Journal of Game Development*, Vol. 1, pp. 7–28.

[6] Botea, Adi, Bouzy, Bruno, Buro, Michael, Bauckhage, Christian, and Nau, Dana (2013). Pathfinding in Games. *Artificial and Computational Intelligence in Games* (eds. Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius), Vol. 6 of *Dagstuhl Follow-Ups*, pp. 21–31. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany.

[7] Botta, Matteo, Gautieri, Vincenzo, Loiacono, Daniele, and Lanzi, Pier Luca (2012). Evolving the Optimal Racing Line in a High-End Racing Game. *IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 108–115, IEEE.

[8] Buckland, Mat (2005). *Programming Game AI by Example*. Jones & Bartlett Learning.

[9] Cormen, Thomas H., Leiserson, Charles E., Rivest, Ronald L., and Stein, Clifford et al. (2001). *Introduction to Algorithms*, Vol. 2. MIT Press and McGraw-Hill.

[10] Craig, John J. (2013). *Introduction to Robotics: Mechanics and Control*. Pearson, 3rd edition.

[11] Dawes, Beman, Abrahams, David, and Rivera, Rene et al. (2014). Boost C++ libraries. Retrieved May 7th, 2014, http://boost.org.

[12] Byl, Penny Baillie de (2004). *Programming Believable Characters for Computer Games*, Chapter 1, pp. 2–5. Charles River Media.

[13] Dechter, Rina and Pearl, Judea (1985). Generalized best-first search strategies and the optimality of A*. *Journal of the ACM (JACM)*, Vol. 32, No. 3, pp. 505–536.

[14] Delaunay, Boris (1934). Sur la sphère vide. *Izv. Akad. Nauk SSSR, Otdelenie Matematicheskii i Estestvennyka Nauk*, Vol. 7, pp. 793–800.

[15] Demyen, Douglas and Buro, Michael (2006). Efficient Triangulation-Based Pathfinding. *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, Vol. 6, pp. 942–947, AAAI Press.

[16] Dijkstra, Edsger W. (1959). A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, Vol. 1, No. 1, pp. 269–271.

[17] Donald, Bruce R. (1987). A Search Algorithm for Motion Planning with Six Degrees of Freedom. *Artificial Intelligence*, Vol. 31, No. 3, pp. 295–353.

[18] Felner, A., Zahavi, U., Holte, R., Schaeffer, J., Sturtevant, N., and Zhang, Z. (2011). Inconsistent heuristics in theory and practice. *Artificial Intelligence*, Vol. 175, No. 9-10, pp. 1570–1603.

[19] Ferguson, Dave and Stentz, Anthony (2006). Using Interpolation to Improve Path Planning: The Field D* Algorithm. *Journal of Field Robotics*, Vol. 23, No. 2, pp. 79–101.

[20] Fox, Dieter, Burgard, Wolfram, and Thrun, Sebastian (1997). The Dynamic Window Approach to Collision Avoidance. *IEEE Robotics & Automation Magazine*, Vol. 4, No. 1, pp. 23–33.

[21] Harabor, Daniel Damir and Grastien, Alban (2011). Online Graph Pruning for Pathfinding on Grid Maps. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*, pp. 1114–1119.

[22] Harabor, Daniel and Grastien, Alban (2013). An Optimal Any-Angle Pathfinding Algorithm. *Proceedings of the 23rd International Conference on Automated Planning and Scheduling*, pp. 308–311.

[23] Hart, Peter E., Nilsson, Nils J., and Raphael, Bertram (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, pp. 100–107.

[24] Hartmann, Daniel, Meinke, Matthias, and Schröder, Wolfgang (2008). An Adaptive Multilevel Multigrid Formulation for Cartesian Hierarchical Grid Methods. *Computers & Fluids*, Vol. 37, No. 9, pp. 1103–1125.

[25] Higgins, Dan (2002). How to Achieve Lightning-Fast A*. *AI Game Programming Wisdom*, Vol. 1, pp. 133–145.

[26] Hwangbo, Myung, Kuffner, James, and Kanade, Takeo (2007). Efficient Two-phase 3D Motion Planning for Small Fixed-wing UAVs. *2007 IEEE International Conference on Robotics and Automation*, pp. 1035–1041, IEEE.

[27] Kavraki, Lydia E., vestka, Petr, Latombe, Jean-Claude, and Overmars, Mark H. (1996). Probabilistic Roadmaps for Path Planning in High-Dimensional Configuration Spaces. *IEEE Transactions on Robotics and Automation*, Vol. 12, No. 4, pp. 566–580.

[28] Kelly, Alonzo (1994). An Intelligent Predictive Controller for Autonomous Vehicles. Technical report, The Robotics Institute, Carnegie Mellon University.

[29] Korf, Richard E (1985). Depth-first Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, Vol. 27, No. 1, pp. 97–109.

[30] Laird, John and VanLent, Michael (2001). Human-Level AI's Killer Application: Interactive Computer Games. *AI Magazine*, Vol. 22, No. 2, pp. 15–25.

[31] Lanctot, Marc, Sun, Nicolas Ng Man, and Verbrugge, Clark (2006). Path-Finding for Large Scale Multiplayer Computer Games. *Proceedings of the 2nd Annual North American Game-On Conference*, pp. 26–33, Eurosis.

[32] Likhachev, Maxim, Gordon, Geoffrey J., and Thrun, Sebastian (2004). ARA* : Anytime A* with Provable Bounds on Sub-Optimality. *Advances in Neural Information Processing Systems 16* (eds. S. Thrun, L.K. Saul, and B. Schölkopf), pp. 767–774. MIT Press.

[33] Lozano-Perez, Tomas (1987). A Simple Motion Planning Algorithm for General Robot Manipulators. *IEEE Journal of Robotics and Automation*, Vol. 3, No. 3, pp. 224–238.

[34] Millington, Ian and Funge, John (2009). *Artificial Intelligence for Games*. Taylor and Francis US, 2nd edition.

[35] Moore, Andrew and Atkeson, Chris (1995). The Parti-game Algorithm for Variable Resolution Reinforcement Learning in Multidimensional State-spaces. *Machine Learning*, Vol. 21, No. 3, pp. 199–233.

[36] Mori, Masahiro (1970). Bukimi no tani (The uncanny valley). *Energy*, Vol. 7, No. 4, pp. 33–35.

[37] Nareyek, Alexander (2004). AI in Computer Games. *Queue*, Vol. 1, No. 10, pp. 58–65.

[38] Nash, Alex, Daniel, Kenny, Koenig, Sven, and Felner, Ariel (2007). Theta*: Any-Angle Path Planning on Grids. *Proceedings of the National Conference on Artificial Intelligence*, Vol. 22, p. 1177, Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.

[39] O'Hara, Keith J., Bigio, Victor L., Dodson, Eric R., Irani, Arya J., Walker, Daniel B., and Balch, Tucker R. (2005). Physical Path Planning Using the GNATs. *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, pp. 709–714, IEEE.

[40] Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Pub. Co., Inc., Reading, MA.

[41] Pinter, Marco (2001). Toward More Realistic Pathfinding. *Game Developer*, Vol. 3.

[42] Rabin, Steve (2000). A* Speed Optimizations. *Game Programming Gems*, Vol. 1, pp. 272–287.

[43] Rayner, C., Bowling, M., and Sturtevant, N. (2011). Euclidean Heuristic Optimization. *AAAI Conference on Artificial Intelligence*, pp. 81–86.

[44] Russell, Stuart and Norvig, Peter (2002). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Second edition.

[45] Schnieders, Benjamin (2014a). aiTools, a free, lowlevel C++ library to ease AI related tasks. Retrieved May 7th, 2014, http://airesearch.de/aiTools/.

[46] Schnieders, Benjamin (2014b). Designing a Steering System for Vehicular Traffic for Emergency 5. Retrieved June 12th, 2014 from http://airesearch.de/written/SteeringSystemForEM5.pdf. Internship Report for Maastricht University.

[47] Scott, Bob (2002). The Illusion of Intelligence. *AI Game Programming Wisdom*, Vol. 1, pp. 16–20.

[48] Smed, J. and Hakonen, H. (2006). *Algorithms and Networking for Computer Games*. Wiley.

[49] Stentz, Anthony (1995). The Focussed D* Algorithm for Real-Time Replanning. *IJCAI*, pp. 1652–1659.

[50] Sturtevant, Nathan R. (2012). Moving Path Planning Forward. *Motion in Games (MIG)*, pp. 1–6.

[51] Werf, Erik C.D. van der and Winands, Mark H.M. (2009). Solving Go for Rectangular Boards. *ICGA Journal*, Vol. 32, No. 2, pp. 77–88.

[52] Willow Garage (2014). Robot Operating System. http://www.ros.org.

Appendix A

Additional Figures

A.1 Heuristic Values by Distance and Angle

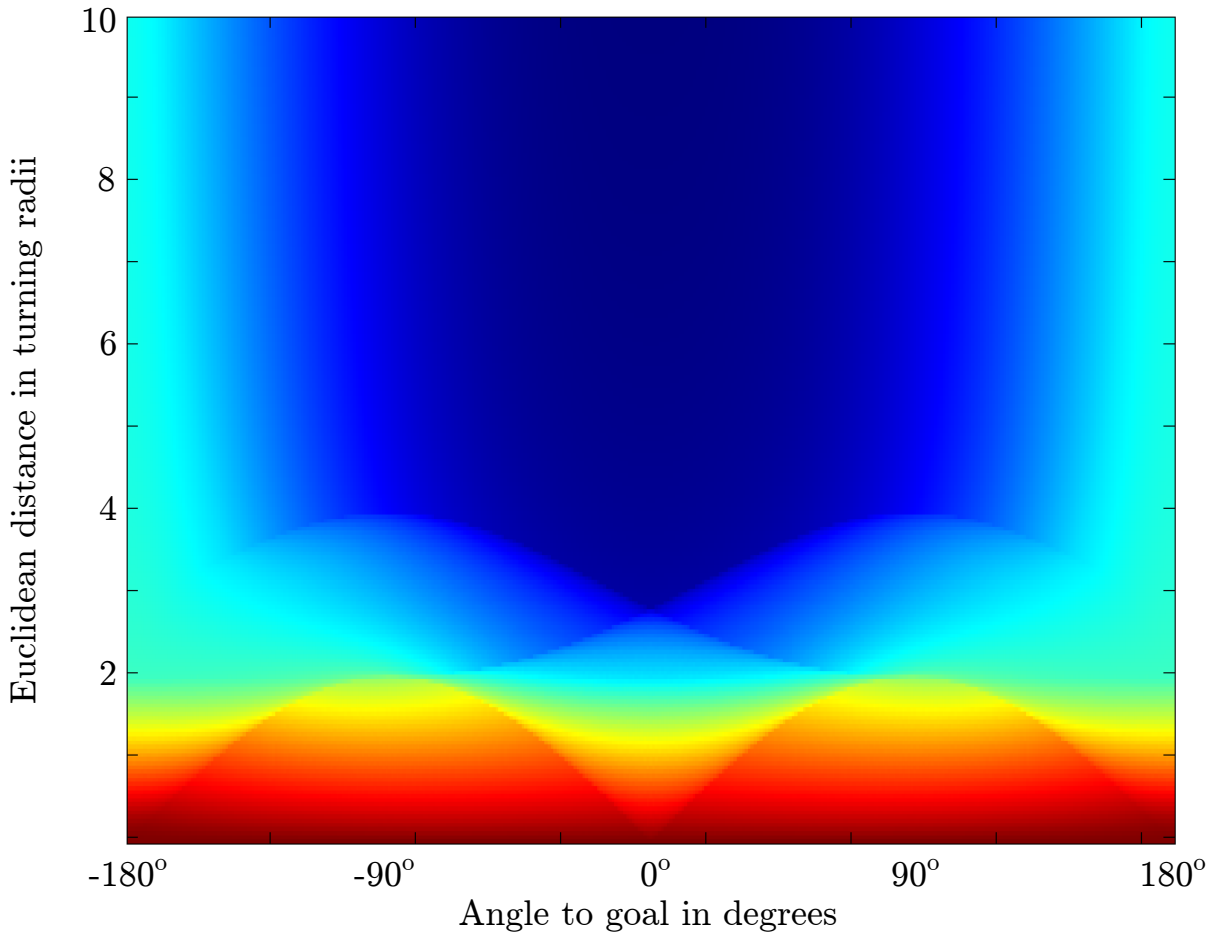

Figure A.1: Difference between heuristic distance and Euclidean distance to goals at certain angles and distances plotted as a heatmap. Distances to goal points are given in multiple of turning angles. Note that for goal points farther than 4 turning radii away, the effect of the turning radii quickly diminishes for all goal angles. Still, all goal points ahead of the vehicle are closer by a certain factor than goal positions behind the vehicle.

A.2 Overshooting Handling Effects on the Heuristic Evaluation Function

In order to correctly estimate the traversal time between two state space configurations with known distance, under- and overshooting have to be taken into account. Depending on the vehicle configuration, linear, circular or a combination of both overshooting handling strategies are used. Each diagram plots the traversal time as color encoded value onto the goal distance versus initial velocity plane. Distances to the goal are given in meters, velocities are given in meters per second.

No Overshooting Handling

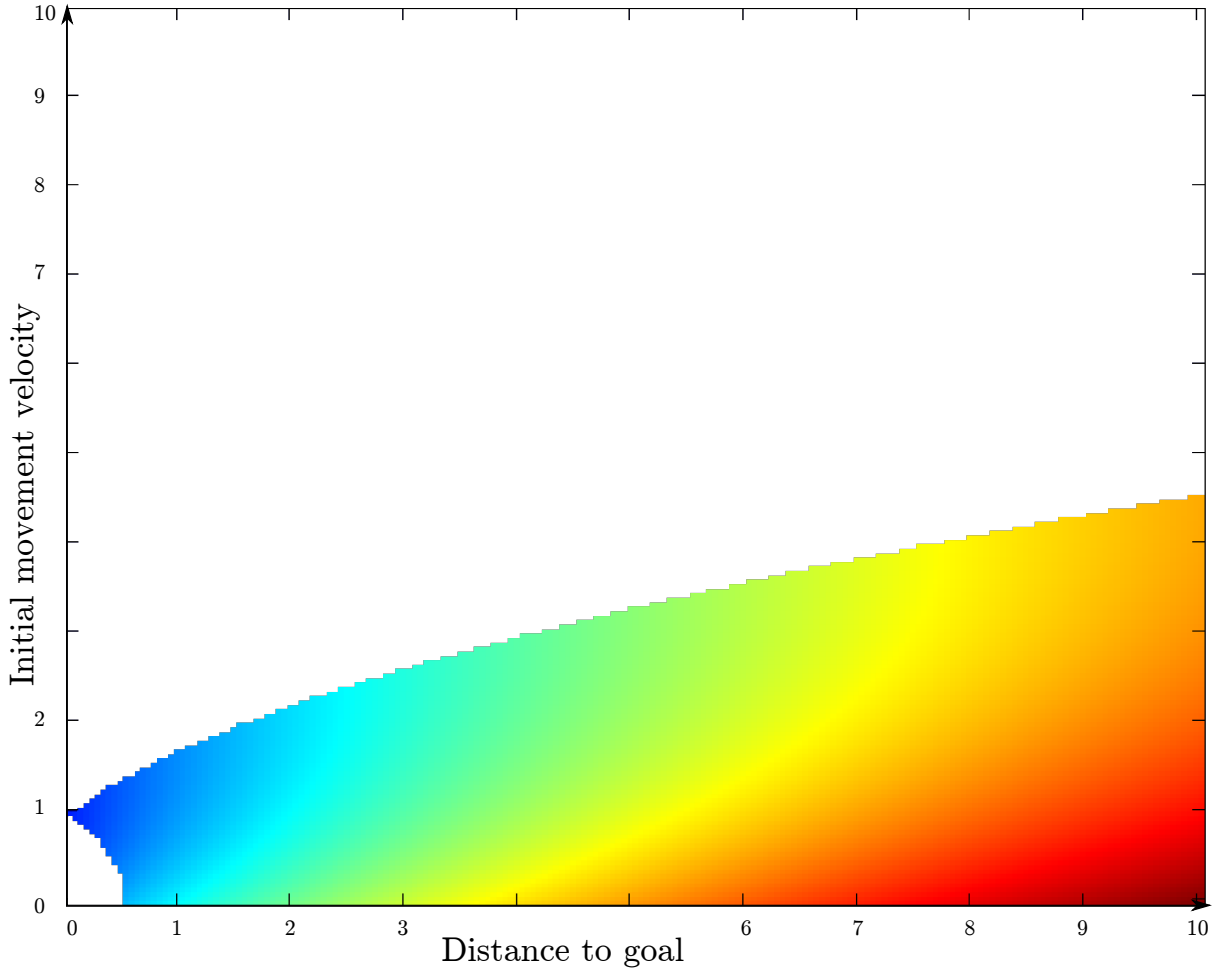

Figure A.2: Heatmap of heuristic evaluation function values for variable distances and initial velocities. The target velocity is constant at $1^{m}/_{s}$. Acceleration and deceleration limits were also set to $1^{m}/_{s^2}$. The large white region is impossible to reach without backing up due to overshooting, i.e., the initial velocity is too high to decelerate to the goal velocity with the given distance. Also note the smaller white region near the origin created through undershooting, in which the initial velocity is too low to reach the goal velocity in the available distance. Naturally, the edge between possible and impossible to reach configurations is a discontinuity, which will prevail in all overshooting handling strategies.

Linear Overshooting Handling

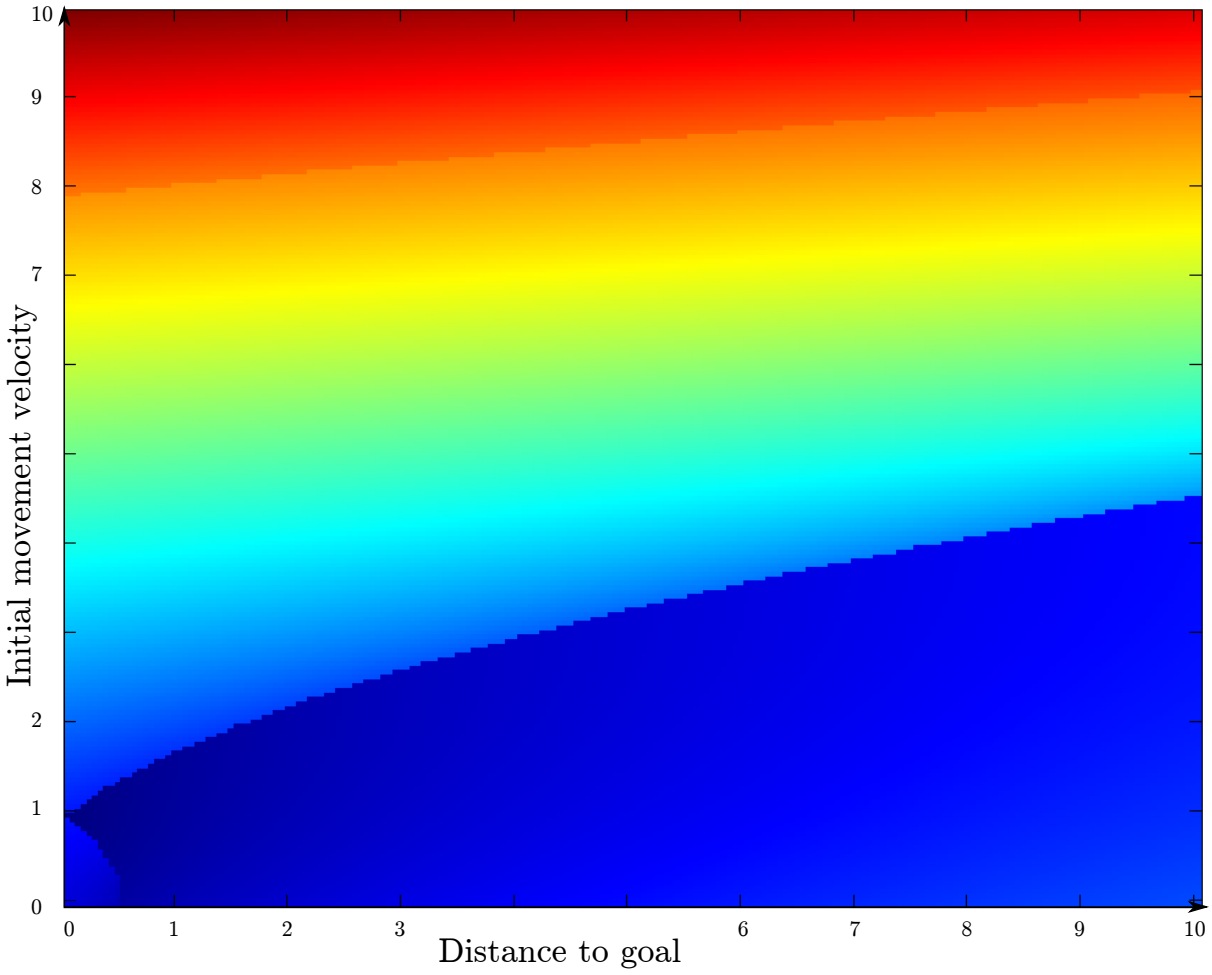

Figure A.3: Heatmap of heuristic evaluation function values for variable distances and initial velocities obtained with linear backing up allowed. No more regions are impossible, but moving too fast quickly increases the time needed to travel especially shorter distances. Note the slight discontinuity in the yellow region, due to reaching the maximally allowed velocity.

Circular Overshooting Handling

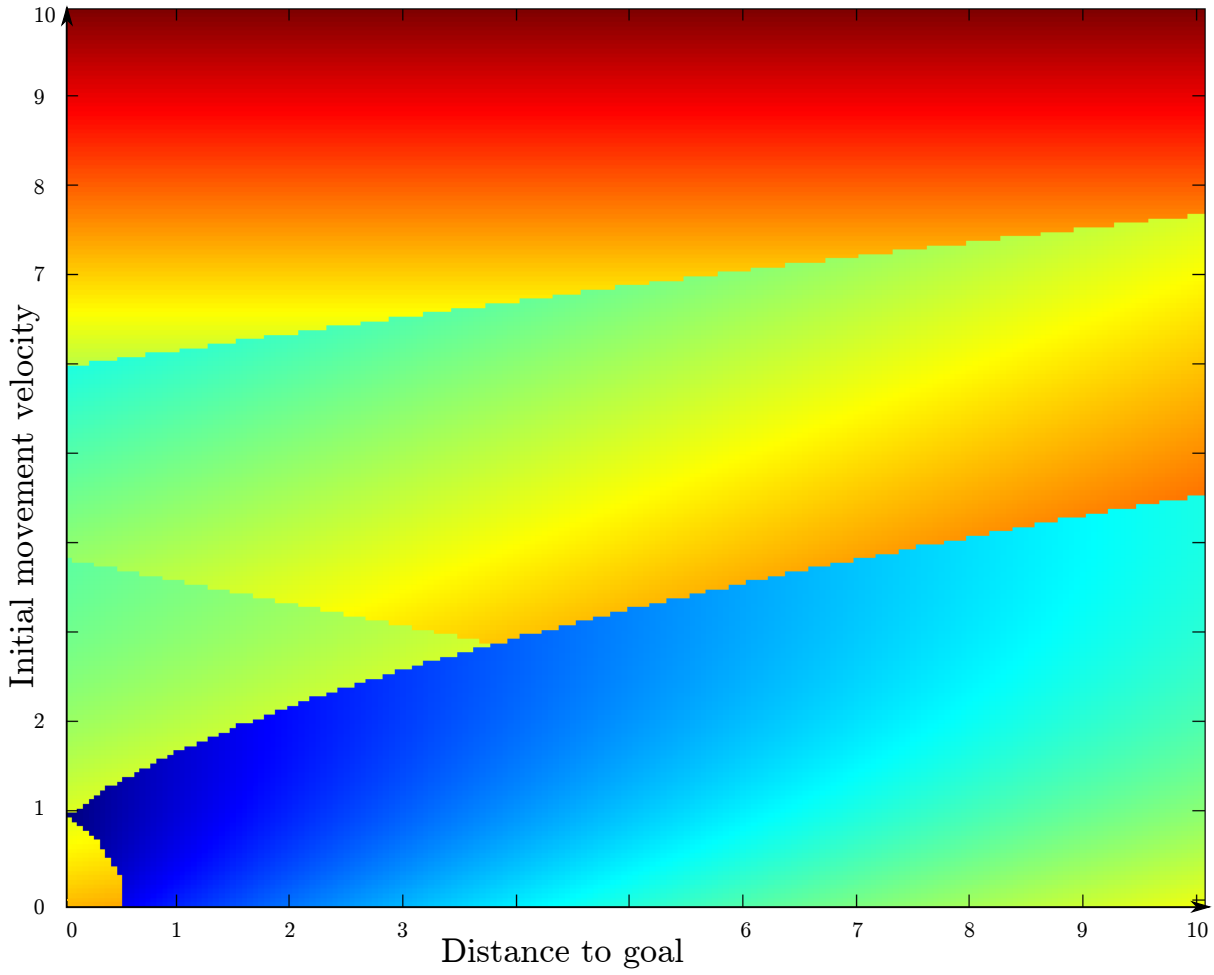

Figure A.4: Circular backing up leads to this heatmap of the heuristic evaluation function values for variable distances and initial velocities. The undershooting effect is much more pronounced than for the linear backing up strategy, as it requires to maneuver a whole turning radius circle with low starting velocity. Note the discontinuities introduced by correctly assuming a circle radius fitting to both the turning radius of the vehicle, as well as the lateral acceleration limit. Note also that the discontinuity introduced by reaching the maximum velocity is more defined, as possibly a longer distance with the maximum velocity is traveled.

Combined Overshooting Handling

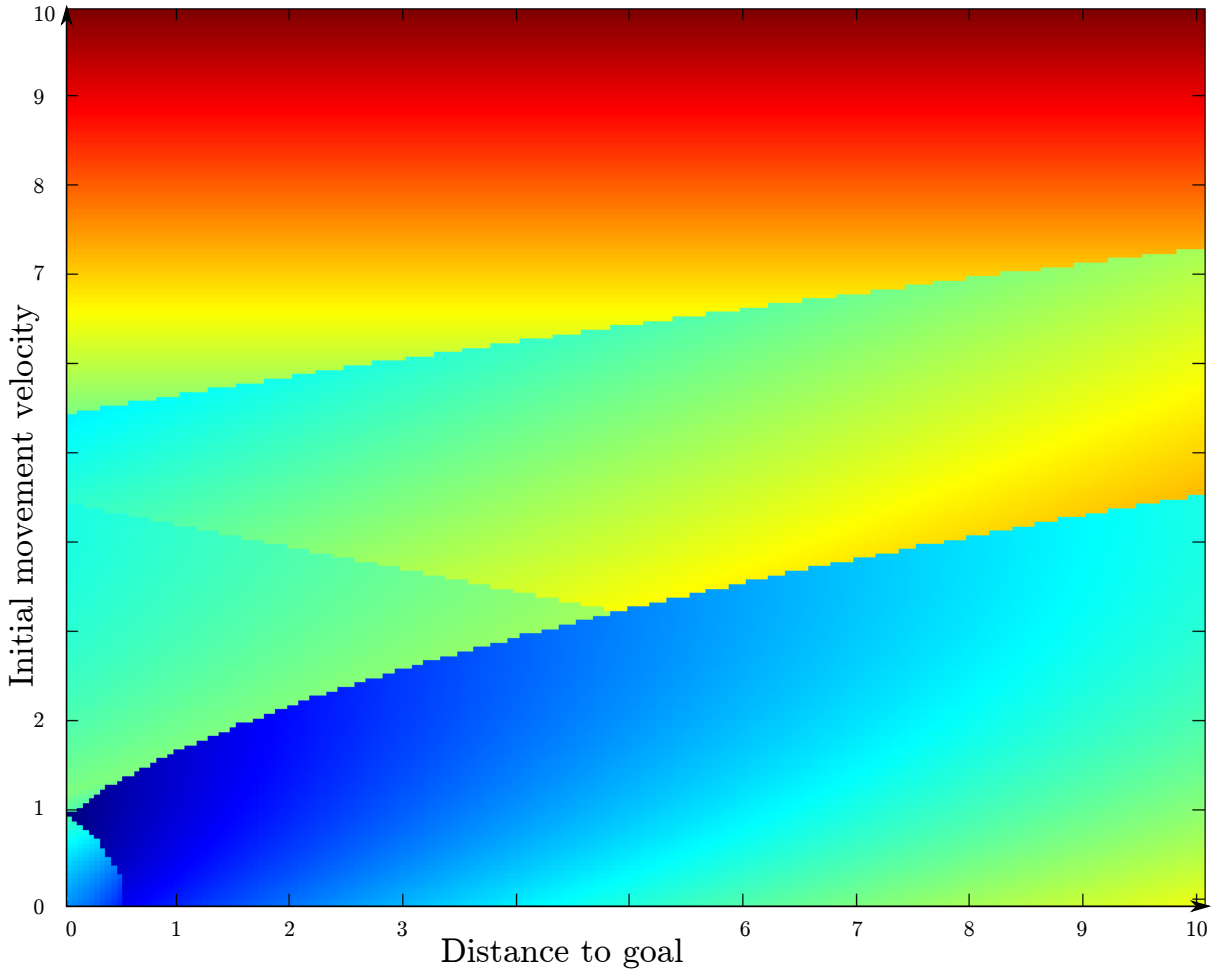

Figure A.5: Heatmap of the heuristic evaluation function using the minimum of both linear and circular backup. If possible for the vehicle, this strategy should be chosen, as it lowers the traversal time and reduces the discontinuity effect for undershooting.

A.3 EM5 Curve Types

The most commonly occurring curves in EM5 maps are presented in this section.

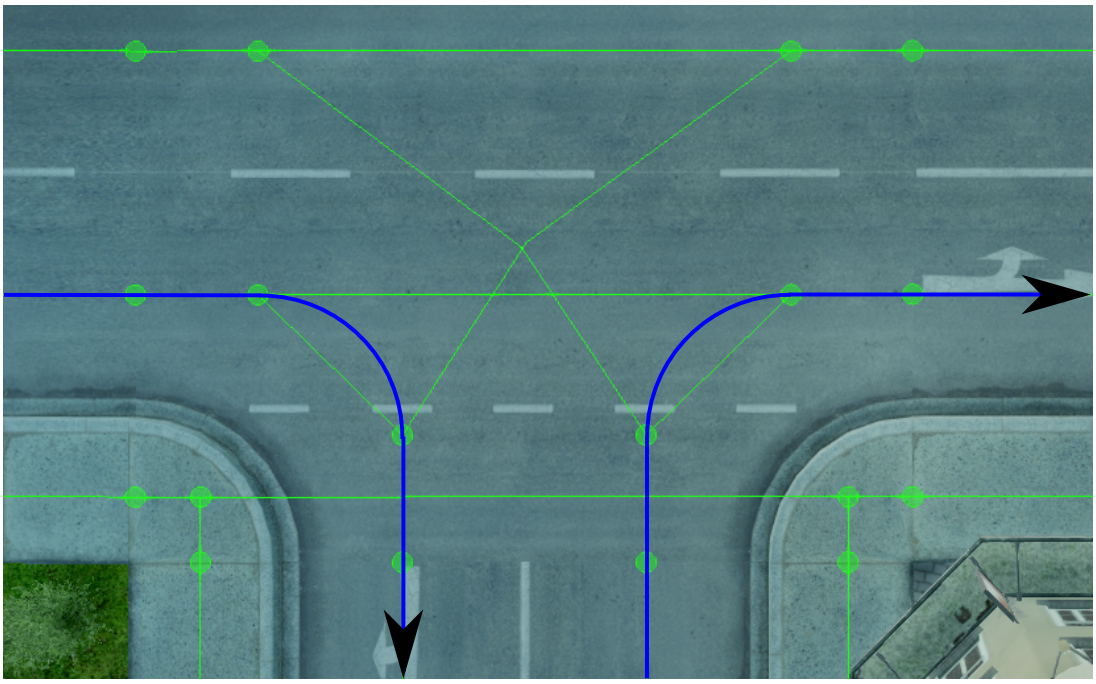

Figure A.6: Typical right curves in EM5. This curve is the most narrow curve occurring with a curve radius of only 6 meters.

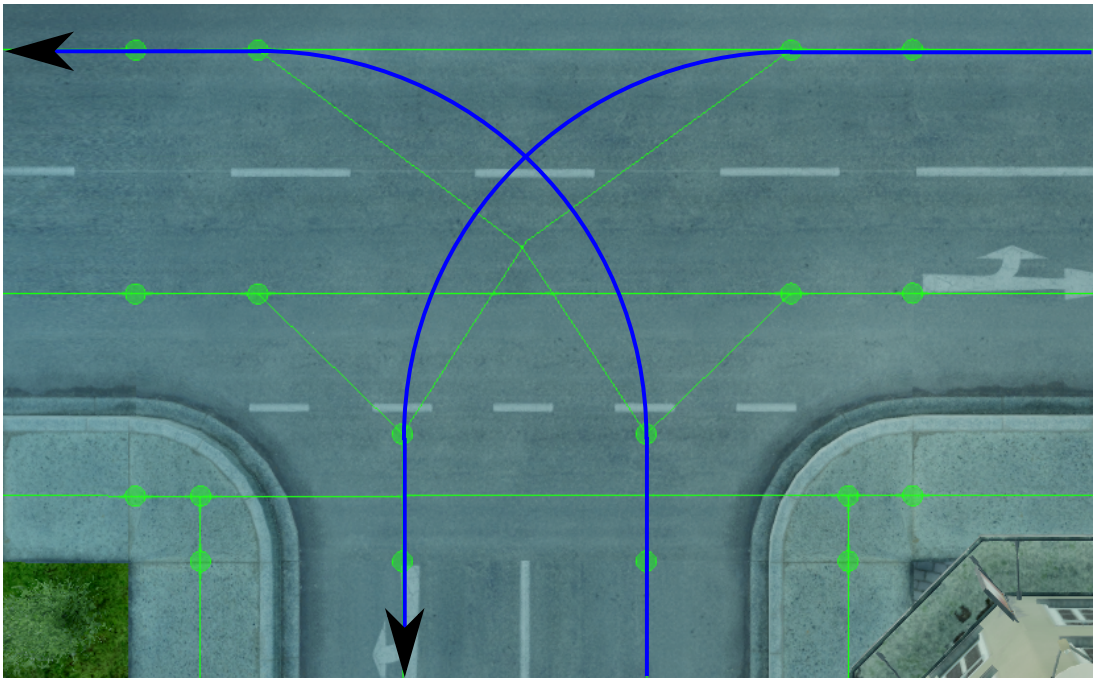

Figure A.7: Typical left curve in EM5. Compared to the right curve, this curve is a lot wider, with a 13 meter turning radius.

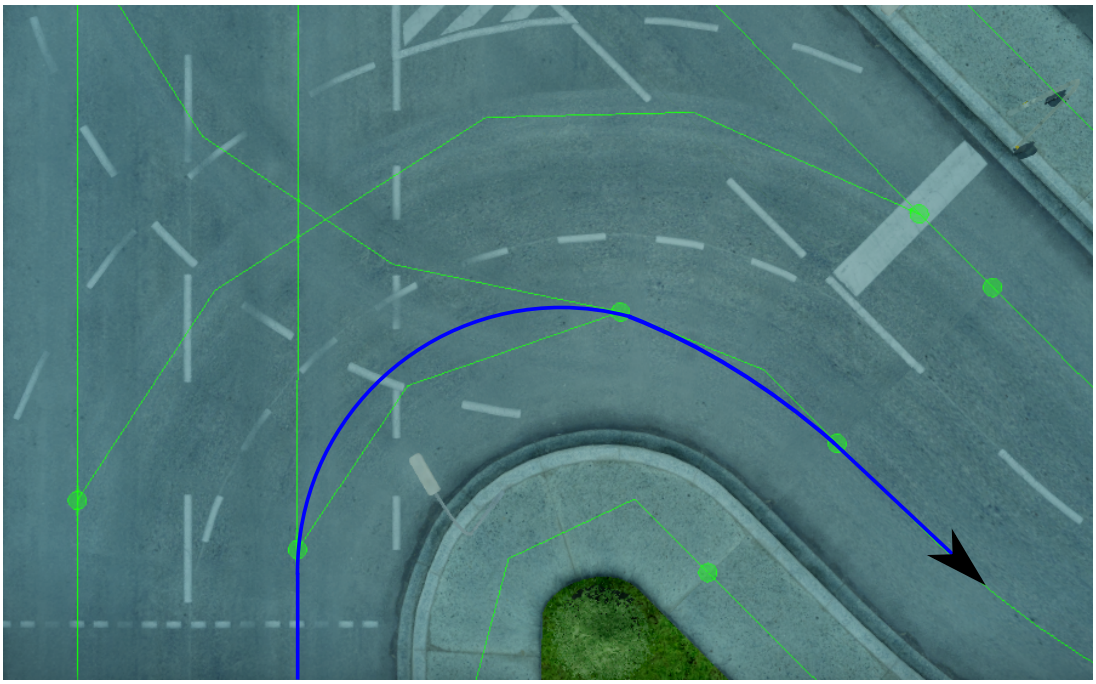

Figure A.8: A sharp, longer right curve with an approximately 8 meter minimal curve radius.

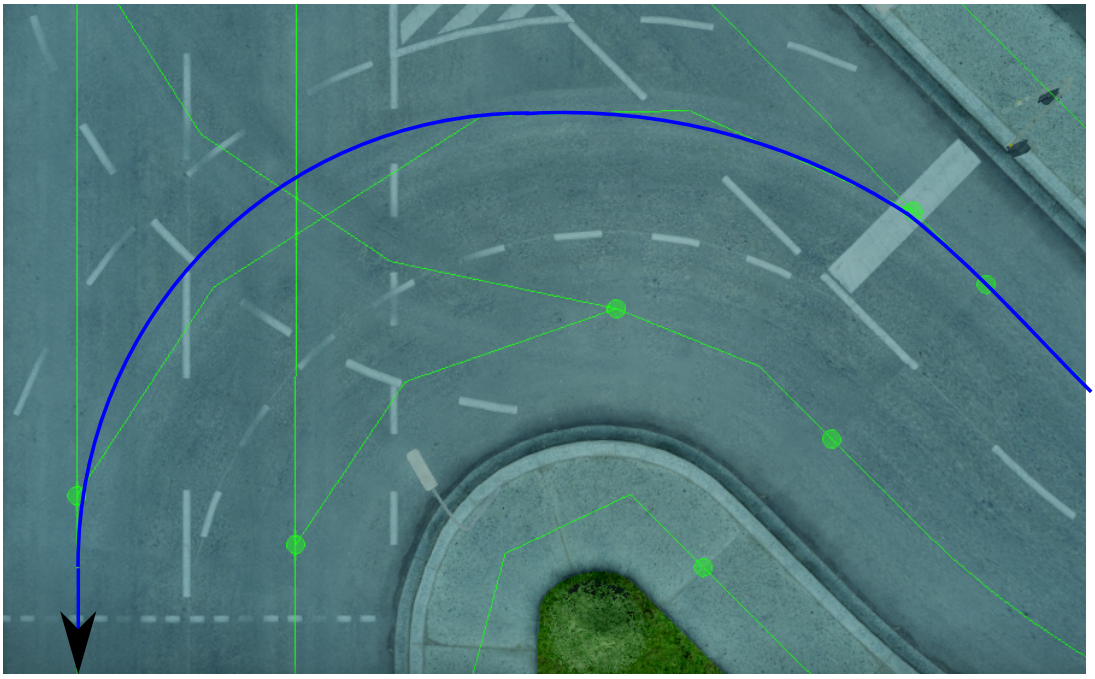

Figure A.9: A long left curve with an approximate minimal curve radius of 15 meters.

A.4 EM5 Curve Search Iterations

Histograms describing the deviations in number of iterations to plan the most common EM5 curves.

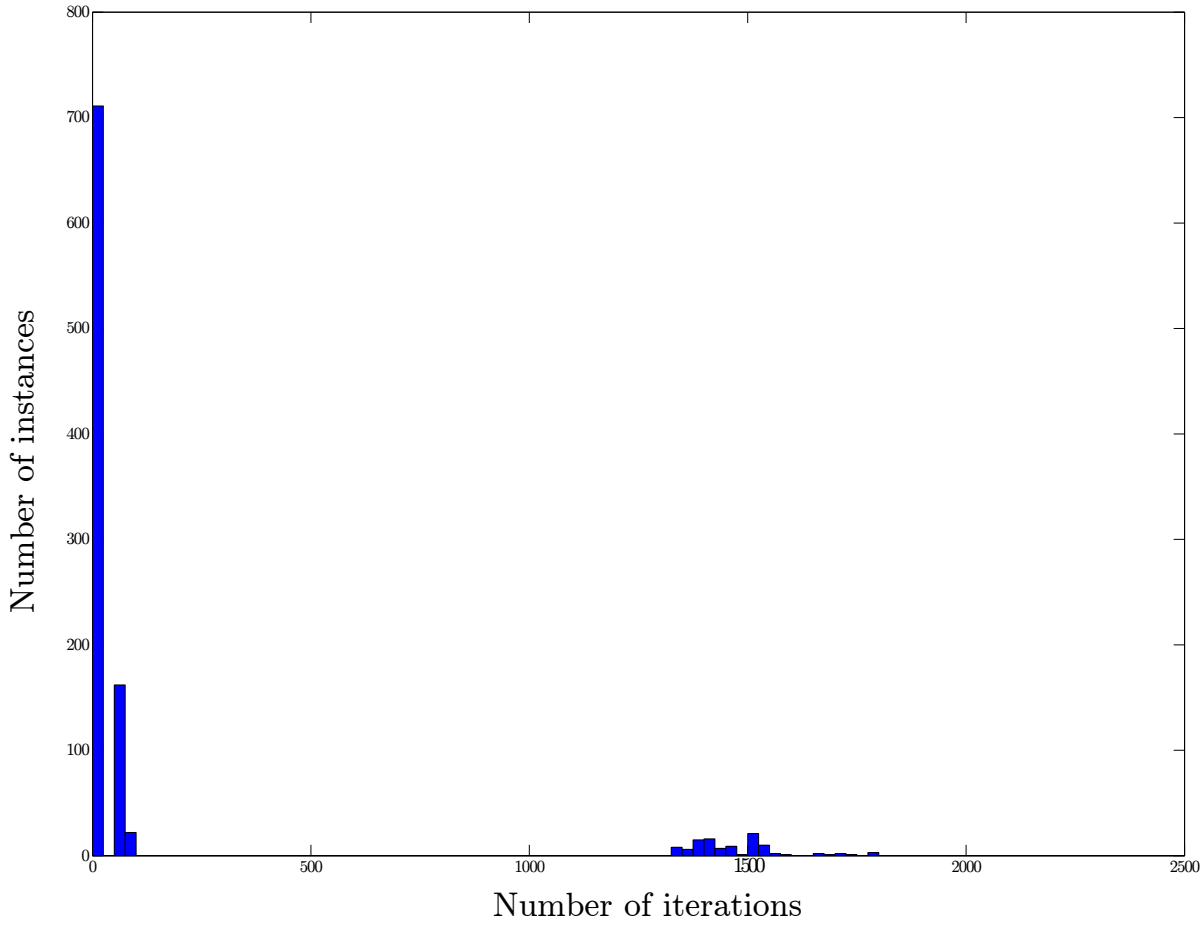

Figure A.10: Histogram describing the deviation of number of iterations to plan the EM5 6 meter radius curve shown in Figure A.6. Most searches can be performed with a low number of iterations, but varying turning radius and input speeds surpassing the lateral acceleration limit in the curve produce some more complex backup maneuvers with higher number of iterations.

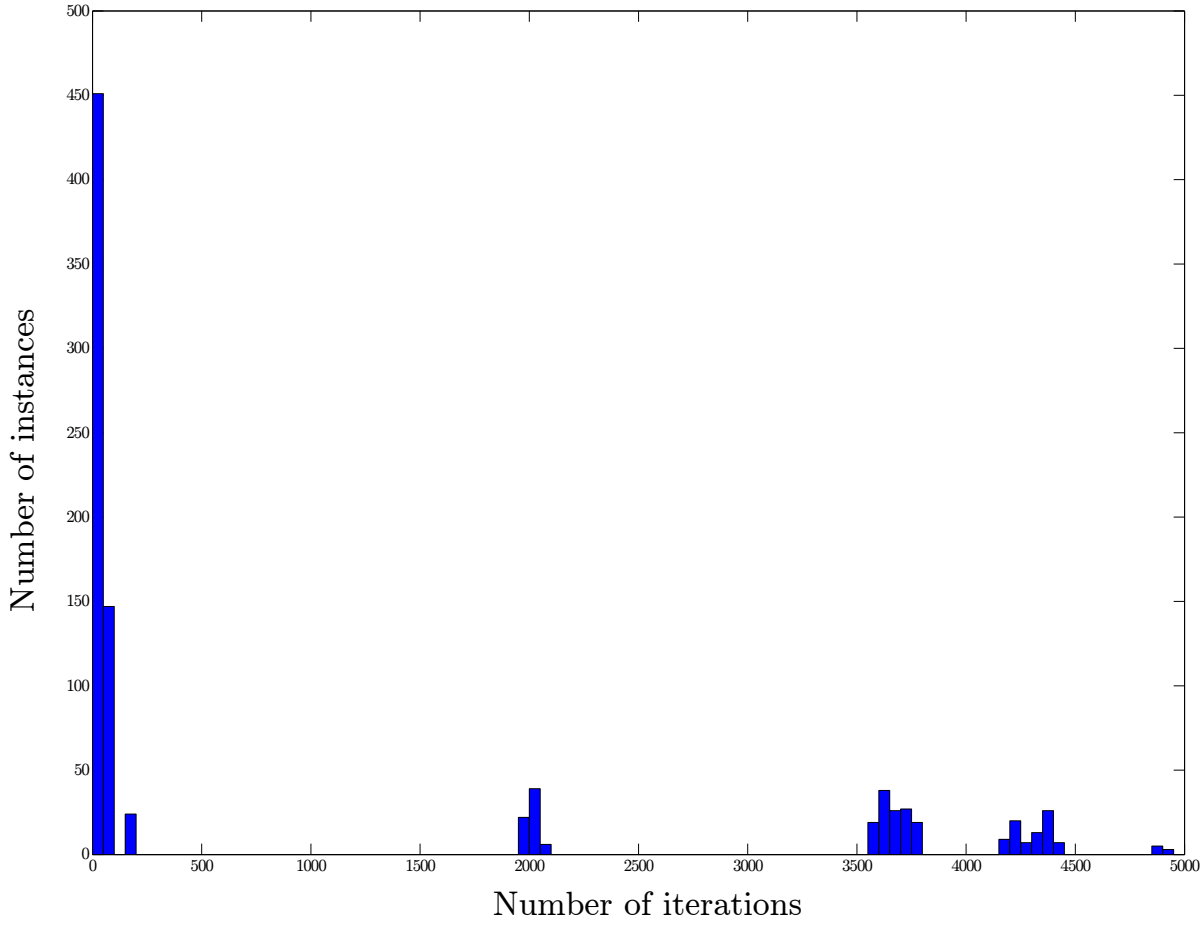

Figure A.11: Histogram describing the deviation of number of iterations to plan the EM5 13 meter radius left curve shown in Figure A.7. For the 13 meter curve, the lateral acceleration limit is not reached, but due to the longer curve, more possibilities arise to produce shorter traversal times. Planning maneuvers like cutting curves may increase the number of iterations.

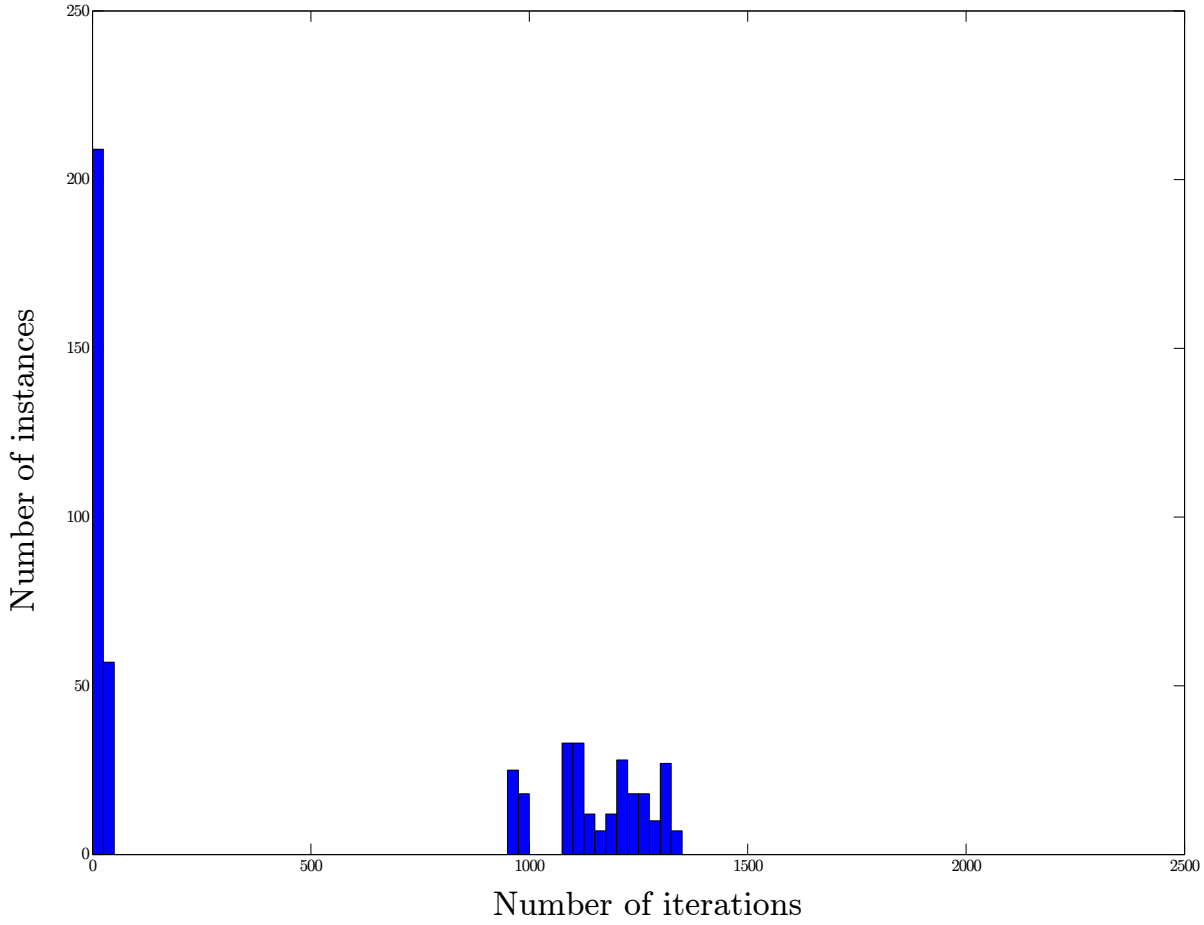

Figure A.12: Histogram describing the deviation of number of iterations to plan the EM5 8 meter radius curve shown in Figure A.8. For the 8 meter curve, the lateral acceleration limit is not reached, lowering the upper bound of search iterations. The overall number of longer path searches is increased, because the path is substantially longer.

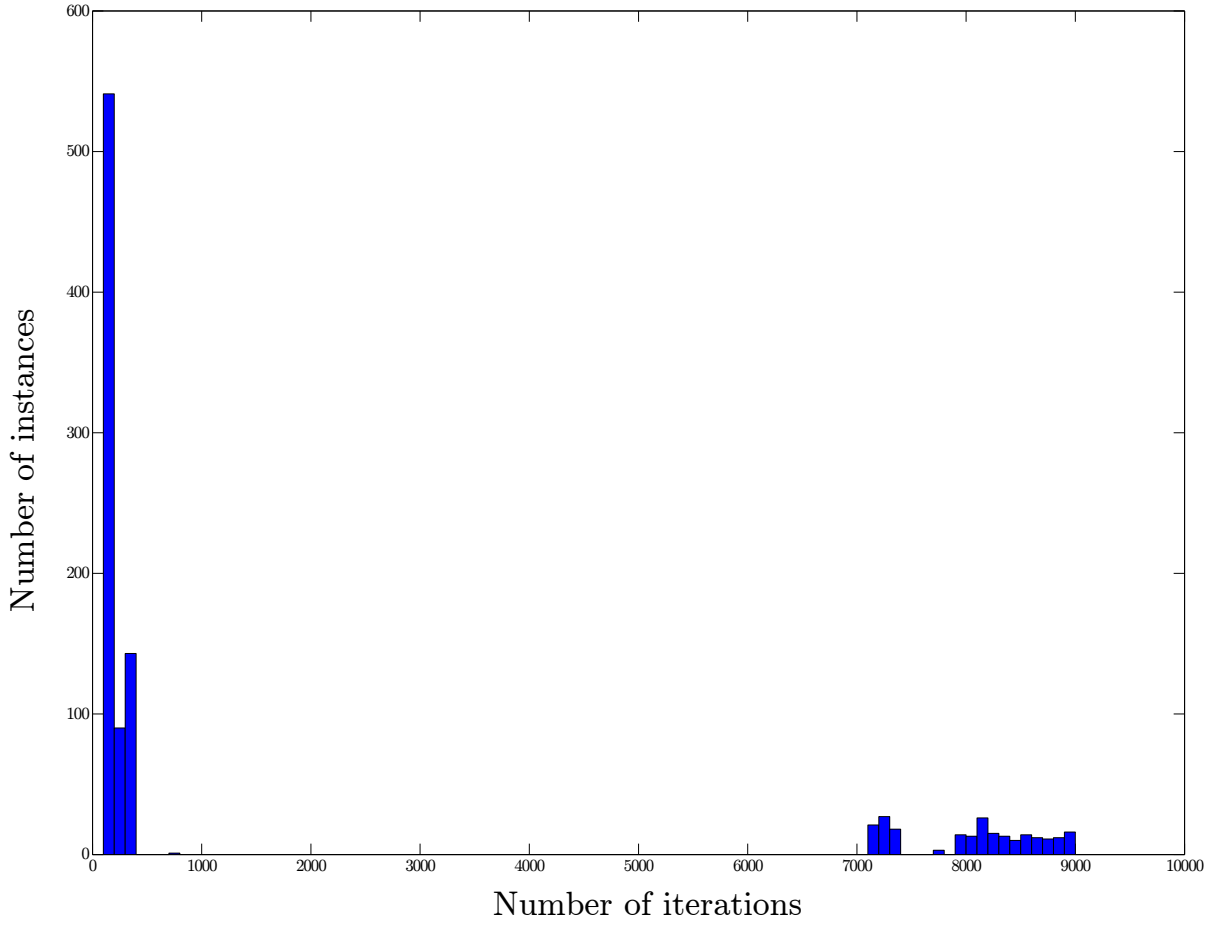

Figure A.13: Histogram describing the deviation of number of iterations to plan the EM5 15 meter radius long left curve shown in Figure A.9. Due to the length of the curve, every backtracking operation in A* requires a high number of iterations, producing a significant number of outliers.

A.5 Histograms of Heuristic Improvement Result Tables

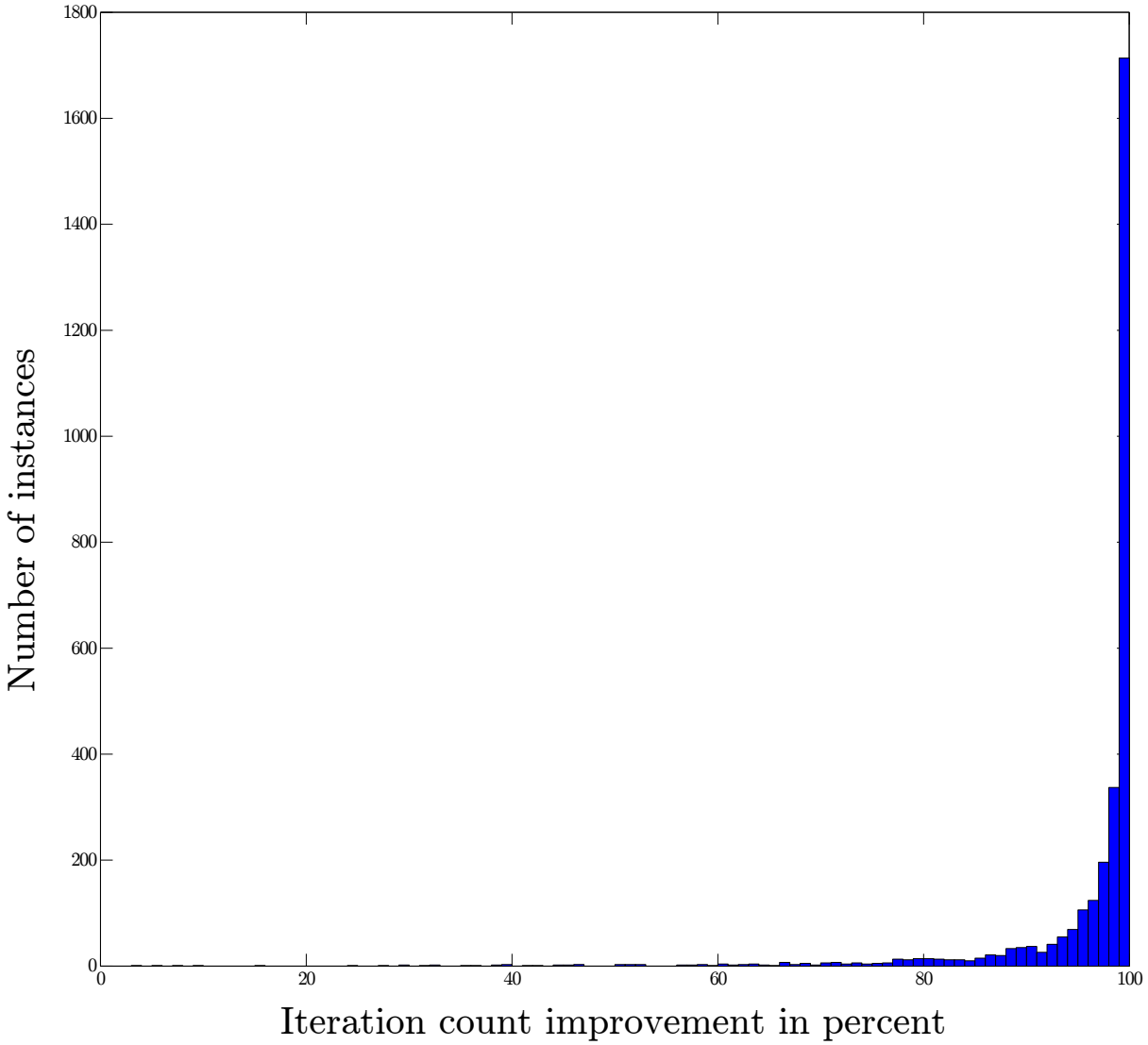

Figure A.14: Histogram of the iteration count improvement in percent of all 3060 path searches to random locations. Improvement of the time estimation heuristic is calculated based on the distance based heuristic. The majority of searches were improved by over 90%

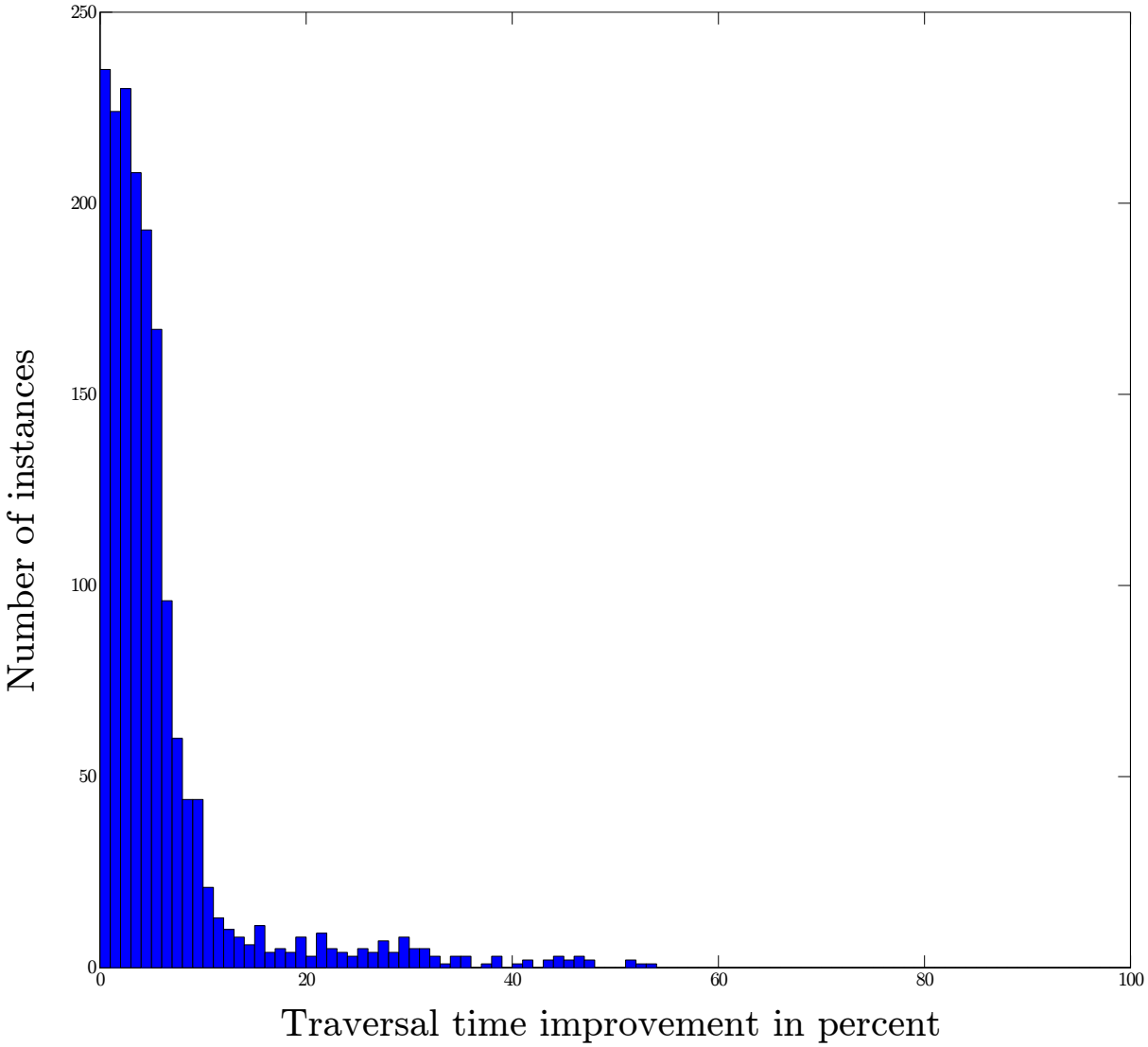

Figure A.15: Histogram of the traversal time improvement in percent of all 3060 path searches to random locations. Improvement of the time estimation heuristic is calculated based on the distance based heuristic. Most path traversal times could be improved between 0 to 20%.

A.6 Histograms of Probing Search Result Tables

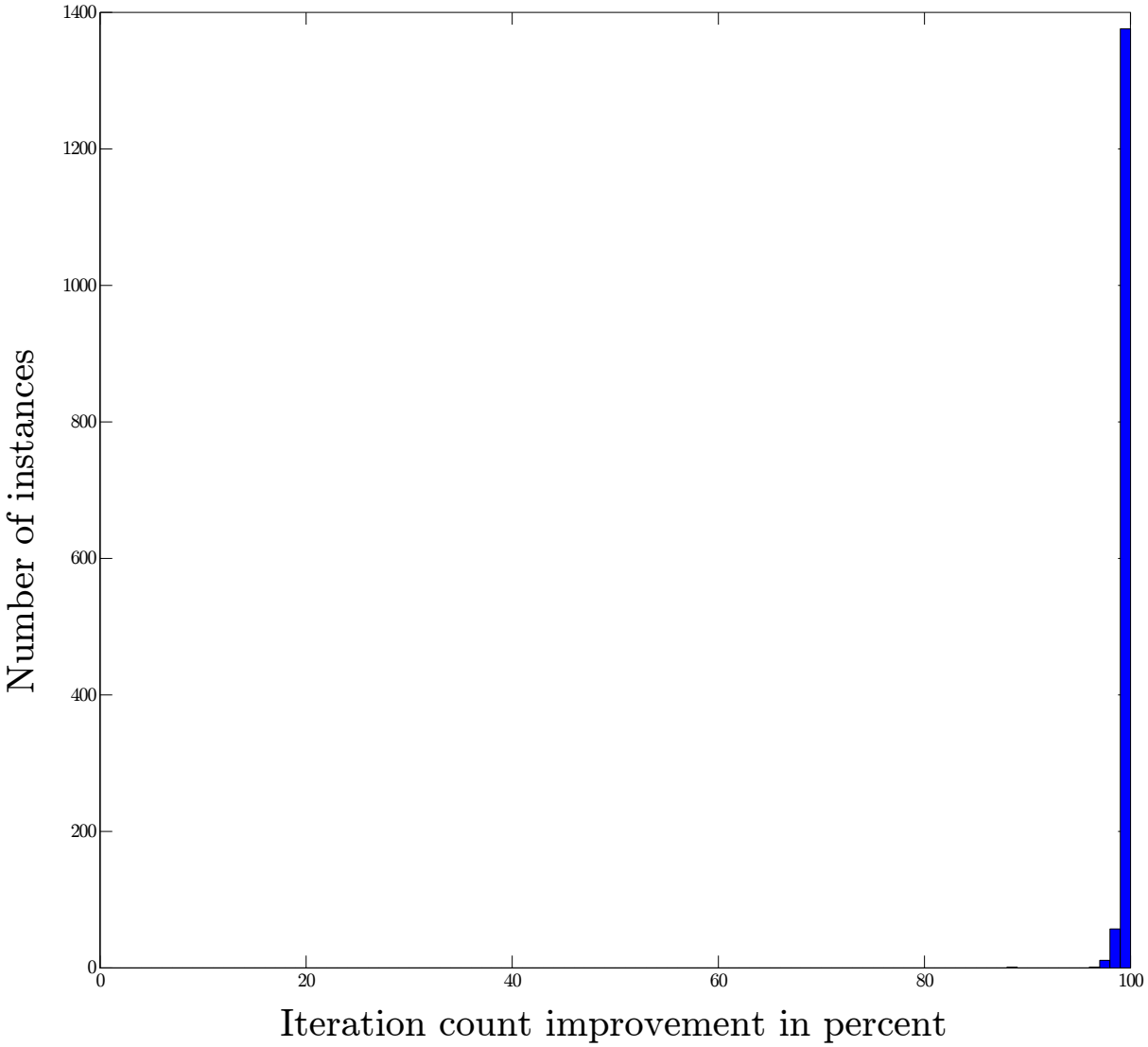

Figure A.16: Histogram of the iteration count improvement in percent of all 1446 path searches to random locations. Improvement of the iteration count compares the number of iterations required for the probing search versus the number of iterations required for A*.

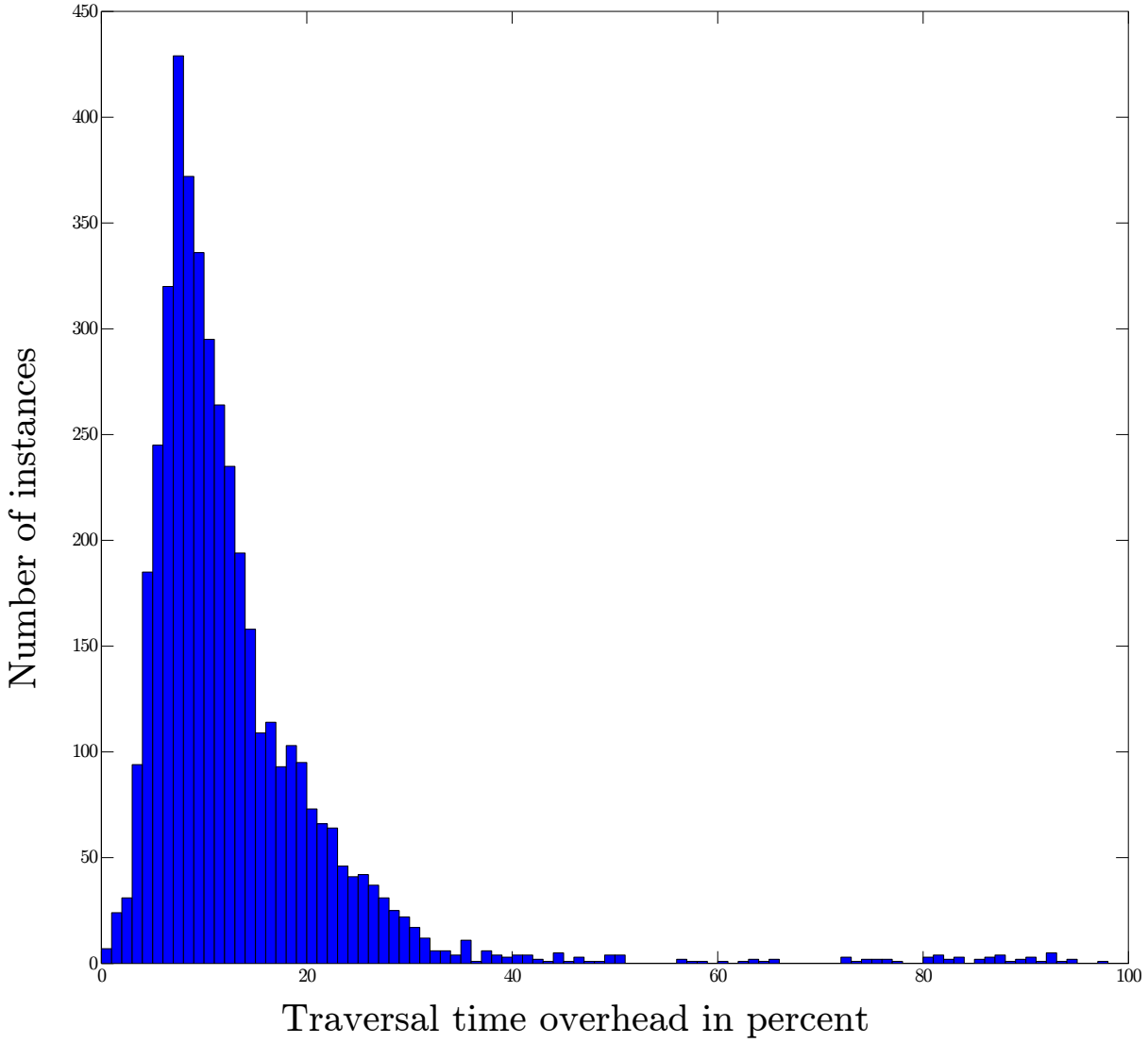

Figure A.17: Histogram of the traversal time overhead in percent of all 4323 path searches to random locations. As A* finds shorter paths than the probing search, the overhead compares path traversal times of the probing search with A* path traversal times.