ABSTRACT

Title of Document: AN ADAPTIVELY SAMPLED PATH PLANNER USING WAYPOINTS: AN ANY-ANGLE VARIANT

Yonatan Gefen, ECE, 2014

Directed By: Associate Professor, Nuno C. Martins, Department of Electrical and Computer Engineering

This thesis develops a low-cost grid-based path planner that intrinsically supports smooth, curved vehicle dynamics. There are many advantages to grid-based planners, including working natively in the digital space of most sensors, and efficiency in low dimensional space. However, discrete planners create jaggedness in most paths. Further, the dimensionality must be limited for efficiency, usually by limiting vehicle steering angles to a small finite set.

The algorithm presented here, Waypoint-A*, extends A* to produce low-cost curved trajectories, taking the dynamics of the vehicle into account explicitly post-planning. Considering the path generated by A* as composed of a set of waypoints, Waypoint-A* calculates the minimum-cost heading on a continuum, to direct the vehicle to the waypoint at the location resulting in the lowest total cost. Smoothness of these curves is invariant to terrain resolution and computation.

AN ADAPTIVELY SAMPLED PATH PLANNER USING WAYPOINTS: AN ANY-ANGLE VARIANT

By

Yonatan Gefen

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park, in partial fulfillment
of the requirements for the degree of
Masters of Science
2014

Advisory Committee:
Professor Nuno C. Martins, Chair
Sarah Bergbreiter
Petr Svec

© Copyright by
Yonatan Gefen
2014

Dedication

There are too many people to thank for the completion of this culminating piece of my masters at the University of Maryland, College Park. I would, however, like to briefly acknowledge those who played a most central role in assisting me. Firstly, I would like to thank my parents and family who have been willing to help in more ways than is possible to list. I would also like to acknowledge my advisor, Professor Nuno Martins, for his motivational and intellectual support throughout my degree. Finally, I must thank my good friends in the university and outside who have also given daily support in the form of meaningful friendships.

Acknowledgements

This work was supported in part by AFOSR - FA955009108538 “MURI - Distributed Learning and Information Dynamics in Networked Autonomous Systems”, NSF CNS 0931878 CPS:Medium:Ant-Like Microrobots - Fast, Small, and Under Control, and NSF ECCS 1135726 CPS: Medium: Collaborative Research: Remote Imaging of Community Ecology via Animal-borne Wireless Networks.

Table of Contents

Dedication ii
Acknowledgements iii
Table of Contents iv
List of Tables vi
Table 1: Elements of a Discrete Planner 7
.... vi
Table 2: Elements of A* 11
.... vi
List of Figures vii
Chapter 1: Introduction 1
  Motivation 1
  Organization 3
Chapter 2: General Background 5
  Discrete Planners 5
  The A* Algorithm 10
Chapter 3: Sampling Considerations 19
  Motivation 19
  Considerations 20
    Denseness 20
    Completeness 22
    Maximal Relevant Granularity 23
  Adaptive Sampling 25
  Quad-Tree Decomposition 27
  Interface Challenges 30
  Rapidly Exploring Random Trees 36
Chapter 4: Planning Under Differential Constraints 38
  Motivation 38
  Model 39
  Differential Curves 40
  Motion Primitives 41
  Any Angle 46
Chapter 5: Waypoint-A* 53
  Motivation 53
  Elements 54
  Examples 56
Chapter 6: Conclusion 73

List of Tables

Table 1: Elements of a Discrete Planner ………………………………………………….. 7

Table 2: Elements of A* ………………………………………………………………… 11

List of Figures

Figure 1 Three Step State Propogation. Three steps of state propagation with constant-time steps and three steering angles: {-π/4, 0, π/4} for the Ackerman car model. Notice how much is unexplored in the map, and how avoiding obstacles might create excessive swerving. Usually a larger, better set of motion primitives is used. ... 2

Figure 3: Observe how in the image on the right, not only loses much of the relevant information of the detailed photo on the left, but creates a blocky representation of the map. This can be remedied by increasing the sampling rate, at the cost of computation time increasing at $n^2 \log n^2$ with resolution. Source for photo on left: [11] ... 8

Figure 4: Stages Lettered (a)-(f) from left to right ... 14

Figure 5 In this example fragment the planner has almost terminated, but at the last moment, when the closest explored state is but one grid space away, the planner finds another expansion to that same terminal state and the cost is updated. ... 15

Figure 6: The results of different queue sorting methods on uniform open terrain. From left to right: only cost-to-go, only estimated cost-to-come, and A*. In open terrain one can easily see that only considering cost-to-go wastes much computation time expanding needless states. Meanwhile both the cost-to-come estimate only and A* are much more directed in this search. The paths, though they appear different, all cost the same. Source [16]. ... 16

Figure 7: The results of different queue sorting methods on uniform terrain with a trap. From left to right: only cost-to-go, only estimated cost-to-come, and A*. While the cost-to-go only finds a lower-cost path that eludes the cost-to-come estimate only method, it still wastes much computation time. A* reveals its usefulness as it is the best of both worlds here. Source: [16] ... 16

Figure 8: Dispersion with L_2 and L_∞ metrics. Minimizing the dispersion while sampling reduces the gaps where important information might be missed. Source: [8] ... 21

Figure 9 Multiresolution Field D* uses higher resolution search in regions where it might help create lower cost plans. Field D* will be explored later, and the paths that do not traverse through green nodes will be explained. Source: [23] ... 26

Figure 10 Sample quad-tree decomposition. Here pixel intensity delineates different terrain types. ... 29

Figure 11 Note how there is increasing sampling in tight regions whereas wideopen regions have minimal sampling. Sampled nodes are marked with blue circles.. 30

Figure 12 Interfacing Challenges 1. Here each Cell has a node at center, denoted by an uppercase letter. Consider a path from A to D. To traverse the lower resolution cell it must first travel to C, creating a rather circuitous route. ... 32

Figure 13 Interfacing Challenges 2. Travelling through A-C-D is now lower cost, however for any single node orientation there will be circuitous paths. Consider E-C-D for example. Further, B-C-E will clip the red obstacles leaving such a transition forbidden, however, logically a transition between B and E should be possible. In this way feasible paths in the pixel-level state space may appear infeasible. ... 32

Figure 14 Interfacing Challenges 3. Placing nodes at all corners of each cell reduces many of the previously stated problems. There is now a feasible path and direct path from B to E and A to D. However, traversing from cell F to D remains circuitous as F and C have no aligned nodes. These problems result from misaligned nodes at different sampling densities. 33
Figure 15 Framed Quad-Tree Approach. Here all of the aforementioned problems have been addressed. There is now a nicely transition defined between cells of different sizes. However, there are a lot of unnecessary nodes such as C5-6, C9-14, C17-20, and C23-28. 34
Figure 16: Node Mirroring. Note how in the single resolution transition this yields all the same transition options as the framed approach, with much fewer nodes to expand. 35
Figure 17 Difficulty with Mirrored Nodes. Note that traversing from any node in F to any in E an indirect path must be taken via C3,G6, and G5m where a line of sight exists. The line of sight suggests that a straight-line path exists between these points. This will be remedied in Waypoint-A*. 35
Figure 18 Framing the Cell Can lead to direct paths. As opposed to the Mirrored node approach above. Here traversing from F to E can be done more directly via C26,C12,G25, and G14. However, note there are many unused nodes in the search. 36
Figure 19: FWD Model for Ackerman Steering Car. Source [29] 39
Figure 20 Cell Approach to Planning under Differential Constraints. Source [8]...... 43
Figure 21: An intelligent Choice of Motion primitives. Source [25] 45
Figure 22: Each cell is framed with a predetermined number of nodes. A path is free to travel between any of these nodes, thus increasing path angle possibilities. Source: [30]... 47
Figure 23: The framed cell approach on the left definitely creates a more direct path than a normal grid search on the right. In fact it outperforms the grid search even with 9 times the grid spaces. Source: [30] 48
Figure 24: Classically the node is considered to be at the center of the cell: (a). In order to interpolate more easily Field D* moves the node to the upper left corner (b), in that way when interpolating cost-to-go as shown in (c) the path travels through only one type of terrain. This makes the interpolation calculation easier. Also note, that any path leaving **s** must pass through the boundary outlined by $\mathbf{s_1}$-$\mathbf{s_8}$. Source: [2]... 49
Figure 25: Limiting the state transitions to only corners of cells can lead to error and unnatural paths. Here the error induced is 8%, and worse the obvious straight-line path cannot be taken. Source: [2].. 50
Figure 26: The path generated by Field D*.Different grid colors indicate different terrain costs, darker implying higher cost. Note that the path is not restricted to nodes at cell corners. Source: [2].. 51
Figure 27: Any-Angle Advantage. Searching an 8-way neighbor grid results in the solid circuitous path, whereas there clearly exists a more direct path to the goal (dotted path). Waypoint-A* will find this direct path as well. Source: [2] 57

Figure 28: Example 1, Nominal Path. Note how the nominal path is of the same length as in Figure 26 with fewer nodes to search. The nodes are the blue circles. 58
Figure 29: Example 1, Big Picture heading vector field. The region on the bottom left is the initial region, any point in there is a valid start. The region on the right is the goal region, any pose in this region is a valid end point. The planner, however, selects a single state in these regions for the nominal path. The white central block is an obstacle, it has infinite traversal cost. 59
Figure 30: Example 1, Zooming in on Obstacle. Above, below, and to the right of the obstacle, where there is a direct line of sight to the final waypoint the heading field points directly to that final waypoint. However, when an obstacle blocks the line of sight, as on the left of the obstacle, the heading field instead points to the furthest waypoint it can travel to in a straight line. 60
Figure 31: Example 1, Sample Low-Cost Trajectories. Initial poses for Red, Green, Cyan, and Blue are up, right, down and left, respectively. When applicable the vehicle can reverse, as seen in cyan and blue, to head towards the goal in a more direct manner than U-turning without reversal. Notice that the paths are smooth, direct, and predictable with no swerving. These paths avoid the obstacle. 61
Figure 32 Example 2. This map was instructive for the discussion of quad-tree decomposition, so it is natural to show how it performs with the full Waypoint-A* algorithm. The gray region on the right is the initial region, and the goal region is the white on the left. The central white pillar is an obstacle. The addition of the white high cost terrain on the top will be used to illustrate the non-greedy nature of the waypoint selection. 62
Figure 33: Example 2, Waypoint-A* is not Greedy. On the right of the high cost terrain the algorithm draws the vehicle away from the high cost region and to a waypoint that is not as far along as can be maximally reached with a line-of sight. This happens because it is deemed more expensive to traverse the high cost terrain than to take a longer route. Even once inside the high cost region, for a while it is better to turn around and leave the region. However, if the vehicle finds itself deep enough into the region, at a certain point is it calculated to be better to continue through the obstacle to a waypoint closer to the goal. 63
Figure 34 Example 2, Overshoot Due to no Look-Ahead. Due to no line of sight with further waypoints, all heading vectors to the bottom right drive the vehicle to the waypoint at the center o f the sink, with no consideration for later waypoints. Thus, the vehicle does not set itself up to make the turn without overshooting the corner slightly. This will be a problem later in tighter spaces. 64
Figure 35: Example 3, Varied Terrain. In this example each block of terrain is assigned a different cost, as indicated by the color. The lighter blocks are more costly. There are no obstacles here. The initial region is 3 down on the right, and the goal region is on the bottom left. At the interfaces between cells of varying terrain there are sometimes non-smooth changes in the heading field. Although the heading field does not drive the vehicle through there non-smooth interfaces, do to the dynamics of the vehicle it may cross these interfaces. Sometimes this will cause the vehicle to turn around, and sometimes it will continue through, possibly changing heading. Regardless the paths are smooth and natural. 65

Figure 36: Example 3, Trajectories Through Varied Terrain. The trajectories red, blue, cyan, and green start pointing up, right, down and left respectively. Notice that despite what might be predicted looking at the rough heading field in the previous figure, the trajectories are smooth and do not swerve or jitter. 66
Figure 37: Example 3, Zoomed In. The trajectories indicated with blue, green, and cyan drive forward in nice arcs towards the goal. Meanwhile, in red the vehicle is initialized pointing in the wrong direction, and it reverses performing a 2-point turn. Then it cannot make the second turn through the black region, and it is deemed cheaper to perform a 3-point turn than to traverse the costlier terrain. . 67
Figure 38: Example 4, Tight Maneuvers. This example was used earlier to exhibit the completeness under quad-tree decomposition. However, the tight nature of the turns will cause the robot to crash... 68
Figure 39: Example 4, Heading Vector Field. Here one can see a similar vector field layout as in the previous examples with the exception of the undefined regions. In these regions there is no line of sight to waypoint. 69
Figure 40: Example 4, Crashes in Tight Spaces. In this case one can see several crashes as the vehicle cannot make some turn radii (middle of the map), heads to a waypoint without regard for the next waypoint (upper left collision), and cycles about the goal region (upper left).. 70

Chapter 1: Introduction

Motivation

This thesis presents a discrete motion planner focused on finding smooth, low cost, trajectories for robots whose dynamics drive them in curves. Discrete planners usually operate on grids or more general graphs. Due to the discrete nature, these planers are easy to reason about in the context of computing. They also manipulate data in a discrete format, very similar to the output of most sensors and computers, hence working in the same space as the data. Additionally, as long as the dimensionality of the problem space is not large, these discrete searches can be quite efficient as they fit in nicely with the architecture paradigm of modern computers and instruction sets. Finally, there is a wealth of literature about discrete planners, making their study quite interesting.

While discrete motion planners are appealing for their computational efficiency and ease of implementation, they are not directly suited for motion planning on robots with higher order states such as heading. Robots with motion described by Ackerman's car model, for instance, have a dependence on the current state's heading to determine the orientation and location of future states. Intuitively, propagating orientation-state increases the complexity of the algorithm. For each additional state variable another dimension must be added to the discrete search. Furthermore, the size of this extra dimension is dependent on how many heading angles are considered in the search. Clearly, if such a search considered a continuum of steering angles, then

the size of the new dimension would be infinite, leading to an intractable search. Usually such planners compromise by searching over a small and finite set of headings such as {-π/4, 0, π/4} over a constant time-step (See Figure 1) [1]. Yet, the feasibility of traversing through obstacles and varied terrain smoothly is dictated by the ability to steer continuously, as opposed to providing choppy steering inputs. Finite headings lead to a very limited number of trajectories that can cause unnatural, choppy paths [2]. This becomes very important in video games for aesthetics, as well as for vehicles that must be predictable to other humans operating in the vicinity.

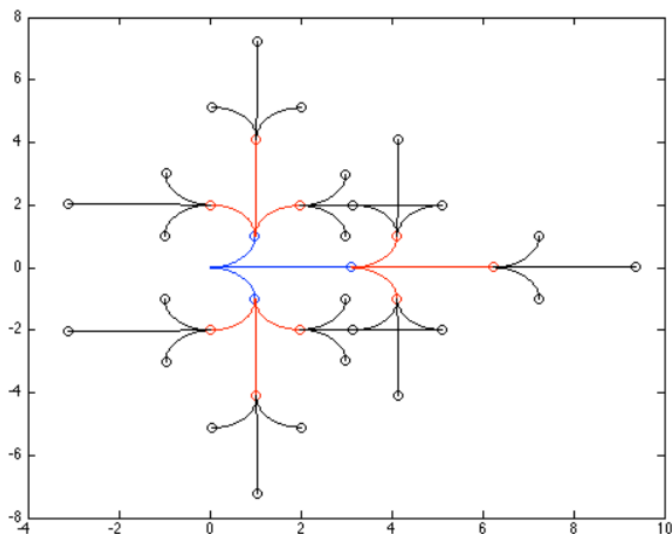

Figure 1 Three Step State Propagation. Three steps of state propagation with constant-time steps and three steering angles: {-π/4, 0, π/4} for the Ackerman car model. Notice how much is unexplored in the map, and how avoiding obstacles might create excessive swerving. Usually a larger, better set of motion primitives is used.

There are many other algorithm classes that can create smooth and natural paths. Potential field algorithms, as an example, produce a gradient field with sinks at the goal state and sources at obstacles. This creates a continuous cost-field that can be used to drive a vehicle smoothly to the goal. These methods have their own

shortcomings however, such as falling into local minima [1]. These are successful algorithms, but this thesis will focus exclusively on discrete planners. The Waypoint-A* algorithm presented here, therefore, attempts to extend the discrete planner to support curved paths.

Waypoint-A* has three stages. In the first stage, it adaptively samples the terrain map into grid cells containing uniform terrain. Over open terrain these cells are large, reducing the amount of calculation required. In tight areas with narrow openings the cells can get as small as needed to ensure that if a pass exists in the tight terrain, it will be found. In the second stage, Waypoint-A* searches the adaptively sampled grid and generates an optimal nominal path through the grid cells from the initial state to the goal state. This path is considered to be the optimal path given the terrain sampling rate and under the assumption that the robot can turn on the spot.

Finally, Waypoint-A* defines a gradient field over the continuous terrain to draw the robot as close as possible to the ideal nominal path while considering the terrain map. Each vertex of the nominal path is treated as a waypoint. At any point in the map the planner can then generate an ideal heading angle for the vehicle. This heading calculation considers the remaining cost at each waypoint on the nominal path as well as the traversal cost to get there given the terrain data. This ideal heading is integrated into a feedback loop using the robot's dynamics to direct the robot in an valid, low-cost trajectory.

Organization

This presentation will have the following structure. First, a general foundation will be

laid in the *General Background* section. The section will formally and intuitively define the terminology related to discrete path planners. The focus of this section will be specifically the A* algorithm which is at the heart of Waypoint-A*.

In the following section,*Sampling Considerations* ,the presentation will discsuss the issues surrounding sampling and sampling based planning approachs. The section will cover topics such as low-dispersion sampling policies, adaptive sampling, and som interfacing problmes with adaptive sampling. This will segway into a discussion of the necessary elements involved in creating the smooth trajectories in Waypoint-A*. It will cover other work done to apply discrete searches to curved paths. This section is called *Planning Under Differential Constrains.*

The next section, *Waypoint-A*,* discusses the elements of the Waypoint-A* planner. The purpose of this section is to describe the planner in a mode removed from its implementation. In this way the hope is not to muddle the core concepts of the algorithm with specifics and compromises of its implementation. The planner applied to an instructive example, and the results are discussed.

Finally, a concluding discussion of future work will be supplied in the *Conclusion* section.

Chapter 2: General Background

Discrete Planners

Generally speaking, *path planning algorithms* are a class of algorithms that are concerned with transforming a set of states from some initial values to goal values. These algorithms are used in a vast number of applications. They can be found in AI that play discrete games such as Bridge, Poker, and most famously IBM's Deep Blue that in 1998 defeated chess grandmaster Kasparov [3]–[5]. Furthermore, these algorithms can be used for such automotive tasks as automated parallel parking: now available in some modern luxury vehicles [6], [7]. Additionally, vehicles participating in the DARPA autonomous vehicle challenges, such as CMU's BOSS have used algorithms such as Anytime D* Lite algorithm that helped their vehicle take the prize for the 2007 Urban Challenge [8].

There are also many less intuitive examples of *planning algorithm* applications. Planners can be used to design maintainable and serviceable machines by automatically determining how to reach, maneuver, and extract different components [9]. Furthermore, these techniques can determine how to construct complex structures so that all the components can fit together, as well as determine *how* to fit them together [1]. In this way, a task that has historically been a laborious activity involving mock-ups and trial-and-error has been converted into the domain of the computer [9]. Another non-intuitive yet powerful application of these planners is

in the study of RNA folding. Some of these newer, more efficient algorithms can approximate the behavior of very large RNA sequences, which may help scientists advance their understanding in this biological field [10].

Evidently, these planners serve many purposes, most of which are out of the scope of this project. For this presentation a specific sub-class of planners called *motion planners* will be considered. *Motion planners* pertain to "the automation of mechanical systems that have sensing, actuation, and computation abilities...usually ignor[ing] dynamics and other differential constraints [1]." This presentation considers algorithms that determine a collision-free and low cost path through variable terrain, while not ignoring dynamics.

Moreover, these planners work in many different problem spaces. Some problem spaces are naturally discrete, such as chess games, whereas some are naturally continuous, such as the navigation of an autonomous car. Of the algorithms that address problems in a continuous space, some operate on the continuous space itself, whereas others discretized this space first. This presentation will focus on those algorithms that operate on discrete spaces that are sampled from continuous spaces. This class of planner is called a *discrete planner.* There are several formulations for discrete planners, however, the following is appropriate for the present discussion.

Table 1	Elements of a Discrete Planner
State Space	Denoted X, which is a set of all possible relevant values pertaining to the robot's pose. A single state in this set is denoted $x \in X$. X should be both *non-empty* and either *finite* or *countable* in cardinality.
Action Set	Denoted *U*, the *action set* is often a function of $\boldsymbol{x}$, in which case it is denoted $u(s)$. *U* is the set of all possible actions at all possible states. $u(s)$ represent the available actions a robot can take at any given state.
State Transition Function	$f: (x, u) \rightarrow s' \in S$. This defines the result of applying an action $u(s) \in U$ on the robot in state $x \in X$. The *state transition function* determines which states *x'* are the neighbors of each *x*.
Initial & Goal State	$x_i, x_g \in X$ respectively. These define the starting and final state of the robot.
Cost Assignment Function	$L: (s, u) \rightarrow \mathbb{R}^{++}$. L outputs the cost associated with making any state transition. In this context the cost assignment will be embodied in a terrain map denoting the cost of the terrain at each discrete grid cell. This will be explained further when discussing the A* algorithm.

It is natural to work with discrete planners in the digital age. Robots receive data about the world digitally (i.e. in a discrete format), reason about the data digitally, and even drive output actuators digitally. Thus working natively in the discrete realm makes code easier to reason about. These discrete spaces are often represented as grids or graphs, both of which run very efficiently due to specialized digital hardware and compiler optimizations. Furthermore, representing continuous terrain as a discrete set can be accurate enough for real world applications – simply looking at the plethora of applications in the literature using A* alone-- yet tractable

in terms of computation time. Finally, these algorithms are intuitive to reason about, lending themselves to easier debugging and shorter development cycles, a feature that is of critical value in industry scale software.

There are, of course, disadvantages inherent to discrete planners when applied to a continuous problem space. The introduction touched briefly on the limitations of finite heading angles. Figure 1 shows the effects of having a small set of heading angles. This will be covered in greater detail in the next sections. Additionally, the grid nature of many discrete planners can create blocky terrain if the sampling is not high enough (Figure 2). The resolution and accuracy are then traded for computational efficiency. If the dimensionality of the problem space gets too large, or the resolution is too high, the computation time may get out of hand.

Figure 2: Observe how in the image on the right, not only loses much of the relevant information of the detailed photo on the left, but creates a blocky representation of the map. This can be remedied by increasing the sampling rate, at the cost of computation time increasing at $n^2 \log n^2$ with resolution. Source for photo on left: [11]

With any planner, one differentiates between finding a *feasible solution* and an *optimal solution*. In *feasible planning* the planner returns a sequence of actions that drive the initial state to the final state, cost aside, insofar as the robot does not travel through forbidden states such as obstacles. These types of planners are used to answer questions of the *existence* of a solution. In *optimal planning* the solution is a series of

actions that drive the robot from the initial state to the final state, without entering any forbidden state and while minimizing some criterion. Usually this criterion is the minimization of the accumulated traversal cost in traveling from the *initial state* to the *goal*. *Suboptimal* plans also play an important role in practical planning. In suboptimal plans optimal cost is willingly traded for a supra-optimal cost and faster or easier calculation. Many algorithms, such as A* give optimal solutions while certain reasonable conditions are met.

It is important to insure that a discrete planner finds a solution, if one exists, in finite time. This is called *completeness*. To guaranty this behavior in a finite *state space* it is sufficient to enforce that the planner does not re-explore states that have already been considered in the path. This is called being *systematic*, and insures that there are no cycles in the path evaluation. In a countable *state space* the requirement for being *systematic* demands that in the limit every reachable state is explored. It should be noted that if no solution exists, it is still acceptable that the planner not return a failure value in finite time.

Another important element of planners is the *principle of optimality.* This principle states: “An optimal policy has the property that whatever the initial state and initial decisions are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision [12].” This means that any fragment of an optimal path remains optimal in isolation with respect to getting from the first state in the fragment to the original final state. This is critically important for planners that dynamically re-plan based on changing information about the terrain [8], [13]. The algorithm presented here does not dynamically re-plan, although it

could be extended to do so. The *principle of optimality* is useful because it is a statement of the continuity of propagation of traversal costs from the initial state to the goal state.

Discrete planners can be used both to search for a path from the *initial state* to the *goal state* by working from initial to final in what is known as a *forward search* or so the reverse in what is known as a *backwards search. Backwards searches* are commonly preferred in planners that dynamically recalculate the path, because as the robot is moving from the start to finish the last part of the path remains largely invariant; this in combination with the *principle of optimality* decreases the amount of the path that must be recalculated. Although Waypoint-A* is not, at this juncture, dynamic, it is based around the *backwards search.*

The A Algorithm*

Within the discrete planner framework the A* algorithm is a very successful, well studied, and extended discrete planner and it will be the basis for Waypoint-A* [14], [15]. A* employs a knowledge-plus-heuristic to focus the search by considering likely low cost paths as well as the directing the search towards the *initial state*. This decreases computational strain (See Figures 5 and 6). It can be used both as a *feasible* path planner as well as a low cost (optimal in many conditions) planner.

Before describing the operation of A* it is helpful to describe the elements of A*. Their usefulness will become evident in the following discussion of the operation of A*.

Table 2	Elements of A*	
State Space	There are several relevant sets in the *state space*. These states are mutually exclusive and form a partition of the *state space*.	
	Unvisited States	A collection of those $\boldsymbol{x} \in \boldsymbol{X}$ that have not been considered by the path planner, hence have not been evaluated for their value. At the onset, all states besides $\boldsymbol{x_g}$ are in this set.
	Alive States	A collection of those $x \in X$ that have entered consideration for their value in the path, but which have not yet been expanded to consider their neighbors. Practically, this means that the state is in the *queue* as has not yet been extracted for evaluation.
	Dead States	A collection of those $x \in X$ which have been considered and whose neighbors have been expanded. This state is dead to the planner, it is never loaded into the queue again, because no better path can found by reconsidering those paths through it. This guarantees that A* is *systematic*.
Cost-To-Go	$G(x_k)$ is an accumulating cost metric quantifying the cost from each state, x_k, to the final state, x_g. This cost is determined for x_k as that state is searched; it is not initially known. $G(x_k) = L(x_{k-1}, u(x_k)) + G(x_{k-1})$.	
Cost-To-Come	$C(x_k)$ is like *cost-to-go* except that it measures the cost to come to x_k from x_i. In *backwards search* this is not known or calculated, because the planner works backwards from x_g. It is however estimated in A* as $\hat{C}(x_k) = \lVert x_k - x_i \rVert_2$ or in some cases $\hat{C}(x_k) = \lVert x_k - x_i \rVert_1$.	

Priority Queue	*Q* is the prime distinguishing element between different discrete planners. A *priority queue* is a queue that sorts, or prioritizes, its members given some comparison function. In A*, *Q* is sorted by a knowledge-plus-heuristic function of both the *cost-to-go* of each candidate state as well as an estimate of the *cost-to-come* of that state. Hence, the front of *Q* is that state x_k which has associated with it $min(G(x_k) + \varepsilon * \hat{C}(x_k))$**.** Where ε can be used to influence the balance of *cost-to-come* and *cost-to-go* in the metric. The default is $\varepsilon = 1$, and although other values can be used for various effect, in iterative methods, we do not do so here.

The simplest way to understand the A* algorithm's basics is via the example in Figure 3. In (a) one can see a cost map for the terrain over which the planner will find a path. The red blocks are obstacles and have an infinite cost, therefore, because no finite cost path can pass through an obstacle the planner ignores all such paths. The yellow box is the *goal state*, from which the planner searches towards the *initial state* in gray. The green boxes are reachable states, and the number inside is the cost of traversing that terrain square. Note that it is assumed that full knowledge of the terrain costs is known at planning time. There are methods that work with uncertainty in this cost map and others that explore on the fly, but that is not considered here.

In (b) the A* algorithm begins by expanding the goal state in eight directions, because the *action set U* for this example is such that at any state x_k its neighbors are the surrounding 8 nodes. This is a fairly standard *action set.* Prior to expansion, only the *goal state* was *alive* and the rest were *unvisited.* After the expansion the surrounding states that are not obstacles are loaded into the *priority queue*. A *cost-to-go,* $G(x_k)$, is calculated for each of these states as an accumulated cost of the initial cost, 0, plus the new terrain traversal cost. Also an estimate of the *cost-to-come,* $\hat{C}(x_k)$, is calculated in the parenthesis. $\hat{C}(x_k)$, will be used only in sorting the priority

queue for path creation later. Note, although Waypoint-A* will use the Euclidean norm, here the planner is using the L_1 norm for easy numbers.

In (d) the *goal state* becomes dead as it was removed from the queue in the last step. The next element is extracted from the priority queue such that it has $min(G(x_k) + 1 * \hat{C}(x_k))$. This new state then repeats the previous step, expanding to all its neighbors, making them *alive*, and making itself *dead*. Note, that some of this *state's* neighbors are already visited. For those next *states* that are visited, their new cost is the minimum between the previous cost they held and the cost propagated from the current *state*. In this case the cost remains the same for all previously visited states, but in the next example such a situation will be examined. This continues in (d).

The only thing to note in (e) is that A* does not always expand its search in the direct route towards the *initial state.* Rather, because the cost of the state to the immediate left of the *initial state* was high enough to offset the fact that it was a little closer than the *state* on the upper left, the upper left *state* was the explored next. The planner terminates this exploration stage when one of the neighbors expanded to is the *initial state* itself.

This results in the optimal path, shown in (f). This is the best path given the discretization and chosen action set. It is created while traversing from the *initial state* back to the *goal state* by performing a gradient decent. To do this consider the furthest most *state* in the path (treating the *initial state* as the first state in the path) connect the path to neighboring state with the lowest *cost-to-go.* Iterate this until the *goal state* is reached. As long as all terrain costs are strictly positive the *cost-to-go*

propagates in a strictly monotonic fashion out from the *goal state*. This guaranties that the gradient decent will return to the *goal state*. Note that the path is jagged, this is something that needs to be addressed.

2	3	1	1	1
1	2	OBSTACLE	1	1
1	GOAL	OBSTACLE	1	1
1	1	OBSTACLE	1	1
1	1	1	3	INITIAL

ALIVE COST 1+(7)	ALIVE COST 2+(6)	OBSTACLE		
ALIVE COST 1+(6)	GOAL	OBSTACLE		
ALIVE COST 1+(5)	ALIVE COST 1+(4)	OBSTACLE		
				INITIAL

ALIVE COST 1+(7)	ALIVE COST 2+(6)	OBSTACLE		
ALIVE COST 1+(6)	GOAL (DEAD)	OBSTACLE		
ALIVE COST 1+(5)	ALIVE COST 1+(4)	OBSTACLE		
ALIVE COST 2+(4)	ALIVE COST 2+(3)	ALIVE COST 2+(2)		INITIAL

ALIVE COST 1+(7)	ALIVE COST 2+(6)	OBSTACLE		
ALIVE COST 1+(6)	GOAL (DEAD)	OBSTACLE		
ALIVE COST 1+(5)	DEAD COST 1+(4)	OBSTACLE	ALIVE COST 3+(2)	
ALIVE COST 2+(4)	ALIVE COST 2+(3)	ALIVE COST 2+(2)	ALIVE COST 5+(1)	INITIAL

ALIVE COST 1+(7)	ALIVE COST 2+(6)	OBSTACLE		
ALIVE COST 1+(6)	GOAL (DEAD)	OBSTACLE		
ALIVE COST 1+(5)	DEAD COST 1+(4)	OBSTACLE	ALIVE COST 3+(2)	ALIVE COST 4+(1)
DEAD COST 2+(4)	DEAD COST 2+(3)	DEAD COST 2+(2)	ALIVE COST 5+(1)	INITIAL

		OBSTACLE		
	X	OBSTACLE		
	X	OBSTACLE	X	
		X		X

Figure 3: Stages Lettered (a)-(f) from left to right

Earlier it was mentioned that when expanding the *neighbors* of the *state* at the

front of the *priority queue*, if one of the neighbors is already alive then the planner must resolve which cost to keep. Figure 4, which could be a fragment of some terrain map, shows such an example. Such a scenario can happen if the planner is attracted to the goal along a corridor of low cost terrain heading directly to the *initial state,* which before reaching the *initial state* suddenly gets very expensive. In this case the planner will backtrack and evaluate a less direct route over cheaper terrain. If this less direct, yet cheaper, sequence of states then intersects the first sequence of explored states, the planner may swap the costs if it has found a cheaper way to get to the same place.

1	OBSTACLE	1	…	DEAD COST: 10+(3)	OBSTACLE	**ALIVE COST: 9+(3)**	DEAD COST: 10+(3)	OBSTACLE	ALIVE COST: 9+(3)
OBSTACLE	2	OBSTACLE		OBSTACLE	ALIVE COST: 12+(1)	OBSTACLE	OBSTACLE	**ALIVE COST: 11+(1)**	OBSTACLE
OBSTACLE	INITIAL	OBSTACLE		OBSTACLE	INITIAL	OBSTACLE	OBSTACLE	INITIAL	OBSTACLE

Figure 4 In this example fragment the planner has almost terminated, but at the last moment, when the closest explored state is but one grid space away, the planner finds another expansion to that same terminal state and the cost is updated.

In understanding the usefulness of the A* knowledge-plus-heuristic search it will be helpful to look at some examples on a more intuitive level. In Figures 5 and 6 one can see that in an open uniform terrain the directedness of the A* priority queue leads to a much more efficient search, with no penalties on the cost of the path it will find. A* tends to out perform Dijkstra's algorithm which sorts the priority queue by cost-to-go only, as well as a greedy search which uses only the distance from the *initial state* as its sorting metric.

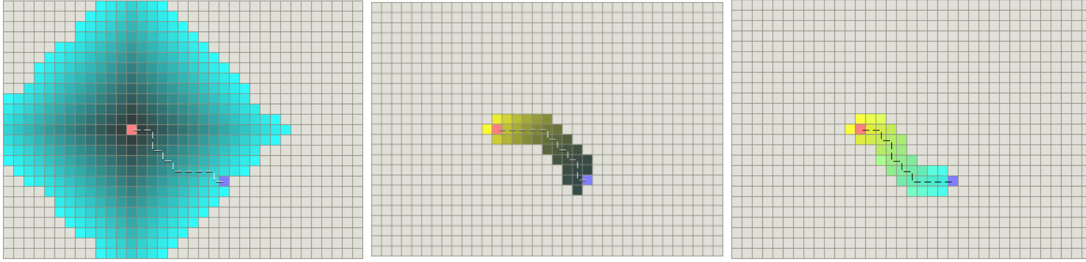

Figure 5: The results of different queue sorting methods on uniform open terrain. From left to right: only cost-to-go, only estimated cost-to-come, and A. In open terrain one can easily see that only considering cost-to-go wastes much computation time expanding needless states. Meanwhile both the cost-to-come estimate only and A* are much more directed in this search. The paths, though they appear different, all cost the same. Source* [16].

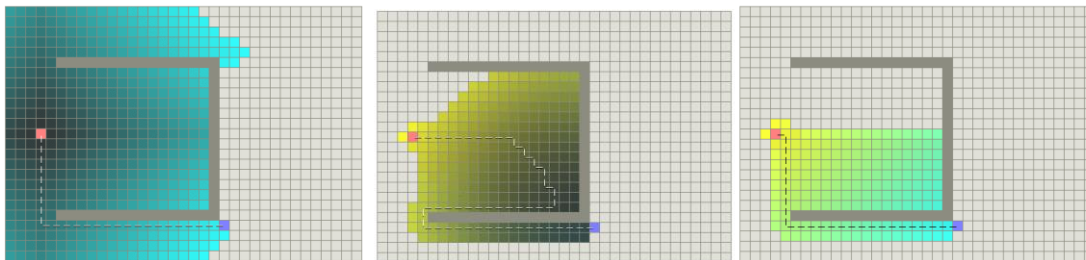

Figure 6: The results of different queue sorting methods on uniform terrain with a trap. From left to right: only cost-to-go, only estimated cost-to-come, and A. While the cost-to-go only finds a lower-cost path that eludes the cost-to-come estimate only method, it still wastes much computation time. A* reveals its usefulness as it is the best of both worlds here. Source:* [16]

Having discussed the search paradigm, cost propagation, and path construction of A* it is important to ensure that A* is *complete*. *Completeness,* as discussed, will guaranty that the search phase is *systematic* (has no cycles) and will complete in finite time if solution exists. In both the finite and countable cases, it is sufficient that the search not reopen *dead states*. In this way, it is not possible to generate cycles in the finite case. Further, not reopening *dead states* means the number of *reachable states* explored is strictly monotone: in each step the planner will increase the number of visited states if there are any unvisited states left. *This* guaranties that in the limit every *reachable state* is explored. Hence, both in the finite and countable case A* is complete.

It is also critical to have strictly monotonically increasing *cost-to-go* in order

to determine a path. Strict monotonicity in *cost-to-go* can be ensured by enforcing that the *cost assignment* function, L, maps to strictly positive numbers. In the present case, if the terrain cost is always positive, then the *cost-to-go* monotonically increases as *states* are visited with each iteration from the *goal state.* This can easily be seen by the accumulation nature of *cost-to-go*: $G(x_k) = L(x_{k-1}, u) + G(x_k)$**.** Under this strict monotonicity requirement one can construct a path from the *initial state* to the *goal state* by performing a gradient decent as explained above. The monotonicity guarantees that this gradient decent will indeed lead back to the *goal state*.

This path will always be the lowest cost path through all explored *states*. As such optimality of the A*-generated path over <u>all</u> *reachable states* depends on whether the correct states are expanded in the exploration phase of A*. This is guaranteed to happen as long the cost-to-come estimate heuristic remains an under estimator of the actual cost-to-go [1].

Consider the *cost-to-come* estimate $\hat{C}(sk) = \|x_k - x_i\|_2$. As long as $\hat{C}(x_k) < C^*(x_k)$, optimum *cost-to-come,* for all x, the path will be optimal among <u>all</u> states in the *state space*. This is usually the case as the Euclidean distance estimate for cost-to-go does not take into account obstacles or expensive terrain. The only way that the condition will fail is if there are zero or negative costs in the terrain. When all costs are strictly positive one can guaranty that all the appropriate states are expanded such that the lowest cost path through all explored *states* is also the lowest cost path through all the *reachable states* in the state space [17]. In this case the A* heuristic which directs the search has sped up the algorithm without penalty to path cost.

There are many extensions of A*. Some of the relevant one will be discussed

in the next sections. Specifically, those extensions of A* that are geared towards finding paths for robots which travel in curves. These are all discrete searches, which while they are rather efficient and intuitive to reason about, do not lend themselves directly to curved paths. Difficulties associated with curved path generation and several key approaches will be discussed in the next sections.

Chapter 3: Sampling Considerations

Motivation

Grid searches over a finite discretized state space have quite a few nice features some of which have been mentioned earlier. Briefly, the discrete grid nature of the search aligns itself with the native language of modern computers and digital sensors. Secondly, ease of reasoning about a program written on this space may reduce errors and debug time, a property that cannot be overstated in industry scale applications. Also, discrete searches, such as A* are well studied in the literature and have a strong theoretical backing. Moreover, in general, sampling approaches, in general, avoid explicitly representing possibly complicated geometric spaces of other approaches. Finally, under the assumption that there is a maximal state space resolution relevant for a given problems allows discrete searches to be used effectively in practice [1].

In many planning applications, such as vehicle path planning over a terrain map, the underlying search space is continuous, hence uncountably infinite. Such a space cannot be searched directly with discrete planners and must first be sampled in some manner. This poses some challenges that need to be considered when choosing sampling resolution of the state space. Even in the limit, the cardinality of this sample set of the state space can only be countable [1].

How can one guaranty that the search is complete? I.e., that relevant states critical for a feasible or optimal path have been sampled. Preferably, these critical states also will have been sampled in (small) finite time, and not in the limit. Lastly,

even if a sampling policy does do this eventually, how will the planner ascertain when this has happened?

These questions will be covered in the following section, and sufficient conditions for the present problem will be provided. Additionally, various methods of adaptive sampling will be considered to further reduce the sampled state space for efficiency while maintaining completeness.

Considerations

The state space for many problems, certainly the navigation problem considered here, is uncountably infinite. Yet, sampling based approaches, including grid search, attempt to quantize this space with what amounts to finite set of points. Even in the limit, sampling only yields a countable number of samples. This cardinality mismatch makes it hard to guaranty that a sufficient amount of sampling has been used in the correct regions to find a feasible solution, let alone an optimal one [1].

In the previous section feasibility and optimality of A* were discussed only within the context of the sampling assumption. That is, given the current data, what is the least-cost path? Here, however, comparison to the continuous ground truth is mandatory. After all, the robot vehicle will need to traverse the ground truth, not the abstraction the computer internally uses for calculation. As such, the sampling policy is of critical importance for the real-world application of any vehicle path.

Denseness

At minimum, it is important to require that a sampling policy has a topological property called *denseness*. Denseness of the sampling space requires that in the limit,

the set of sampled points get arbitrarily close to any point in the state space. This guaranties that in the limit any piece of relevant data will be revealed via sampling.

Focusing on grid-based searches we define a notion called *dispersion.* Dispersion is defined intuitively as the largest region of un-sampled state space. Formally: $\delta(P) = \sup_{x \in X} \{\min_{p \in P} \rho(x, p)\}$, where X is the state space, P is the set of samples, and ρ is a distance metric. For a graphical idea of what this means, see Figure 7. This can be viewed as a generalization of resolution of sampling. Ideally we wish to minimize this metric, so that the largest gap in sampling becomes very small.

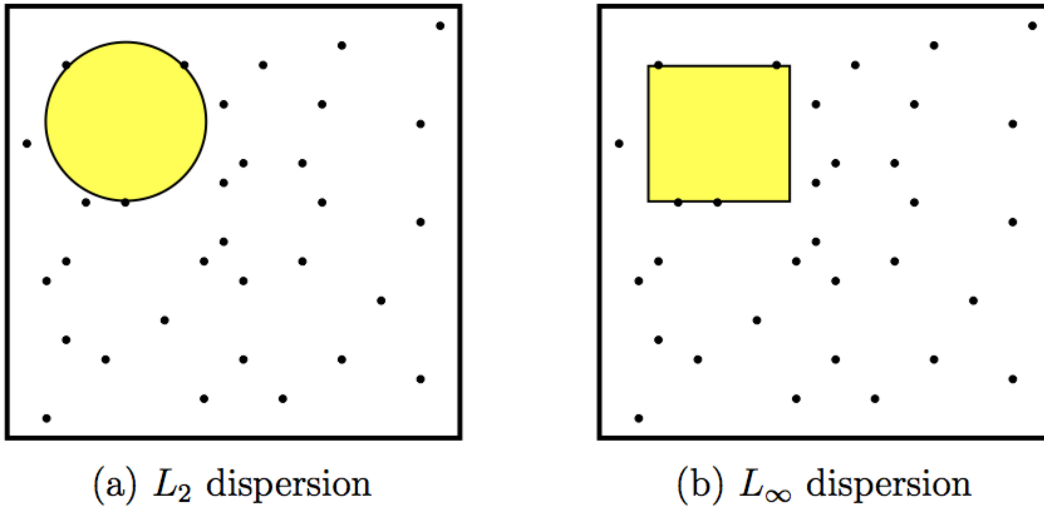

Figure 7: Dispersion with L_2 and L_∞ metrics. Minimizing the dispersion while sampling reduces the gaps where important information might be missed. Source: [8]

The path planning discussed in this thesis occurs entirely on a map in $\mathbb{R}^n$ state space and so optimal dispersion takes the form of a uniform grid. Consider L_∞ as ρ and $X = [0,1]^n$, and a finite number of samples k. Here the state space is a unit cube for simplicity of notation, but it can easily be extended to any closed bounded subset of $\mathbb{R}^n$. In this case the optimal distribution of these points is at the center of squares, such that the number of cubes per axis is $floor\ (k^{\frac{1}{n}})$. When $k^{\frac{1}{n}}$ is not an integer, the

remaining points may be placed arbitrarily without affecting dispersion. This creates a lower bound on dispersion for any set P of k samples: $\delta(P) \geq \frac{1}{2*floor\left(k^{\frac{1}{n}}\right)}$, [1].

It should be noted that nothing about variable terrain, obstacles, or variable resolution sampling has been mentioned in this context. The prior discussion of grids of nodes for searching should be distinguished from the present section. In this discussion there is no prior knowledge of the map, we are sampling to discover terrain. Later we will define how to distribute nodes on this terrain for A* cost propagation and path searching. This discussion, however, is very important for the notion of completeness.

Completeness

To recap on the previous discussion of completeness in the introduction to A*: An algorithm is considered to be *complete* if it can return whether a path solution exists in finite time. This is an important metric, because it determines the tractability of the search algorithm as a whole.

In the previous context we talked about this notion within the framework of an existing searchable node lattice. That is, given a lattice of nodes with traversal costs, find a path from one node to another. There it was sufficient that the search be systematic to terminate in finite time. This essentially avoided cycles both in the finite and countable case. In the previous context we also talked about the relaxation allowing enforcing finite answer time only if a solution exists, there will be a parallel here.

In the present context of determining how to even create terrain sampling, however, challenges to completeness come from uncertainty that we have sampled

the relevant data to form any solution, let alone a good one. Consider a goal state that is a single point in a continuous space, the probability of sampling this point is 0. Realistically, the goal state in this example would be a region, and hence have non-zero probability of being sampled, however it should still be sampled with high probability and as fast as possible. Until the sampling policy places a sample in critical regions such as this, there would not be a complete planner, because although a solution may exist it would never be found [1].

Herein lies the importance of the denseness and dispersion. The sampling policy must have denseness, so that it can guaranty that at least in the limit every point will be sampled. Further, because in a practical sense one would prefer that the algorithm terminate faster, the policy should try to minimize dispersion. In this way for a given sampling density, using a uniform grid creates minimal dispersion [1].

Similar to the discussion covering A*, here we define a relaxation of completeness. If the sampling policy is deterministic we require that it be *resolution complete:* i.e., that given a dense sampling policy, if a solution exists then it will be returned in finite time. With probabilistic samplers we use *probabilistically completeness* stating that with probability 1 a solution will be found in the limit, should one exist. In both cases if a solution does not actually exist, then it is acceptable that the algorithm not return in finite time [1].

Maximal Relevant Granularity

In the present case it is assumed that sampling of the terrain has occurred due to some digital sensor. Usually these sensors output data about the world in a uniform grid format. Relating this to the previous discussion this is good, because, given the

resolution of the sensor (and no prior knowledge about the world) this grid provides minimum dispersion sampling of the continuous space.

This sensing however is not dense, because the sampling resolution of the hardware is limited. As such, if some piece of terrain data is too small to be picked up by the sensors, even at their maximal resolution, then the sampling density cannot be increased to sense it.

Practically, however, this is not a problem. If we assume that there is a maximum relevant resolution for a given problem, then any additional sampling once this granularity has been reached does not improve the solution. In the present case it can safely be assumed that for vehicle navigation, the data contained in one pixel of a high definition sensor is probably of much higher resolution than the so-called relevant resolution. The terrain map built of pixels still represents a continuous space, yet now it is constructed of a finite number of patches with nonzero area. Hence, with a number of samples equal to the number of pixels each patch of relevant data can be sampled.

Therefore, from here on out, the pixel is considered the maximally relevant piece of terrain data. Each pixel can be mapped to an area in the ground truth, and so we treat the pixel level terrain map as a low fidelity continuous mapping of the ground truth. In this sense the denseness criteria is met trivially, because in a finite number of samples maximum terrain knowledge has been achieved. This means that any search that is systematic is also complete in the present discussion.

Adaptive Sampling

At this point, given the pixel level map we could apply A* on the pixels as discussed previously, and it would be a complete planner. However, there would be a lot of wasted computation as the data is likely sampled at a rate such that the grid sizes are quite small with regard to the vehicle. This means that there will be large swaths of uniform terrain. In these uniform regions it is computationally wasteful to use the same sampling rate as terrain of high variability.

Adaptive sampling of the terrain is performed for a few different reasons. Firstly, as discussed, it is used to reduce the complexity of the search by reducing the sampling rate in certain regions. On the other side of the same coin, we wish to retain sufficient resolution in tight or varied areas of the terrain to find low-cost paths. This can also be used to guaranty completeness by ascertaining that in tight regions the appropriate sampling rate (up to the maximum available) is used so as to distinguish different relevant terrain types. For example, the planner in [18]and [19] have higher resolution sampling near more rugged terrain, so reflect the need for a more careful search in these areas.

Some algorithms apply this adaptive *sampling incrementally*. Most simply, in [20] a certain sampling rate is attempted, and resolution is increased if a solution could not be found. Since we assume that at maximum available resolution the planner is certainly complete, this incremental version is also complete. In another similar scheme A* is applied on a lower resolution grid, producing a rough path. Then, incrementally, the resolution is increased and the rough plan is refined. This method is called a staged planning [21]. This method can increase performance if in

each increment the previous rougher path built on for the new higher resolution search. Another benefit is that the robot can start to move once the rough plan is made, working on the better plan on the fly.

Another way to determine resolution is based on distance from current location. In [22] the planner uses a higher resolution locally near the robot, and reduces the resolution linearly with distance from the robot. This reflects higher sensor certainty for nearby objects. There is also a larger immediacy in the planning for nearby objects. For distant obstacles only rougher plan is needed until they get closer.

In a similar fashion, Multi-Resolution Field D* [23] increases the sampling rate, expanding regions as higher accuracy short-range sensor determine a high level of terrain variability. Multi-Resolution Field D* uses a built in replanner to modify the search space on the fly in response to the increased sampling rate. In this way the vehicle can find lower cost paths as it finds new options in the higher resolution map regions. See Figure 8.

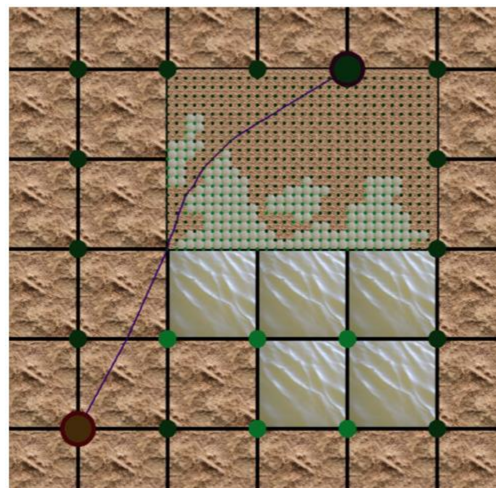

Figure 8 Multiresolution Field D uses higher resolution search in regions where it might help create lower cost plans. Field D* will be explored later, and the paths that do not traverse through green nodes will be explained. Source: [23]*

Another way to look at variable resolution searches is the idea of *adaptive dimensionality.* While a vehicle might have some higher order states that need considering, heading being a rather simple example, there are many portions of a trajectory that likely do not need all the state space dimensionality. For instance, a car navigating a map will likely have many regions where it is driving straight forward. In these cases, a simpler (x,y) state space can be considered as the car is not changing heading. In [24] an incremental A* planner introduces higher dimensional states into the search only in regions it sees fit. In these regions more of the robot kinematics must be considered. Then the incremental planner re-plans over these regions. In this way the algorithm still maintains completeness while gaining speedups due to the lower dimensionality of the search and efficient re-planning that reuses searched states.

In a similar tactic [25] defines a notion called *graduated fidelity* which varies the quality and quantity of pre-computed curved motion primitives to use in different regions. In a low fidelity pass only rough motions are considered. This plan can be refined with a finer and larger set of actions in regions requiring it. They also define a re-planner to allow for several passes over the path, efficiently reevaluating sections previously considered in lower fidelity. In this way they can efficiently calculate dynamically complex trajectories quickly by considering differing numbers of motion primitives in different regions.

Quad-Tree Decomposition

There are many ways to divide the discretized state space into regions of various resolutions. In some cases discussed this happens in pre-planning phase, whereas in

others it happens incrementally or on the fly. These methods come in various levels of complexity. In this thesis we will use the quad-tree decomposition, because it is simple and fast. We will also only consider a single decomposition that happens pre-planning, because this thesis showcases Waypoint-A* in scenarios that do not consider incremental or dynamic re-planning. However, there is no reason why it could not be extended in the future.

Quad-Tree decomposition is a recursive division of the state space into quadrants. First the space is divided into 4 quadrant cells. Then, each of those cells that contains pixels representing 2 or more different terrain types is split in 4 once more. This process is repeated until each cell is of uniform terrain type. This is used because it is fast, simple, and works. There is no guaranty of optimality for this division, however it is a nice heuristic alternative to the NP-hard box-packing problem.

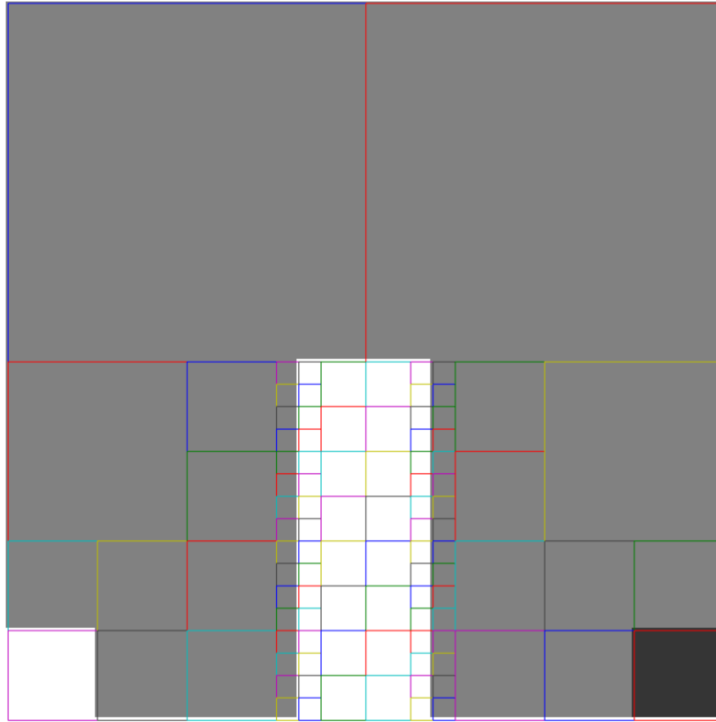

Figure 9 Sample quad-tree decomposition. Here pixel intensity delineates different terrain types.

By splitting the space into square cells of uniform terrain type (in conjunction with appropriate node placement) we can guaranty completeness. Because the quad-tree decomposition keeps dividing cells until it has only unique terrain, any small passage through obstacles can be found, even if it only of 1 pixel width (See Figure 10). As long as we define an action set that allows the A* search to transition from any cell to all its tangential neighbors, then a feasible path between the initial and goal states will be found if it exists. This is discussed in the next section.

Figure 10 Note how there is increasing sampling in tight regions whereas wideopen regions have minimal sampling. Sampled nodes are marked with blue circles.

<u>*Interface Challenges*</u>

When considering varying resolution of state discretization there is ambiguity in how to transition between cells of varying resolutions. In this section we will consider some different approaches for dealing with this interface problem. To do so we will consider some instructive examples.

Each cell groups pixels, the finest grain discretization of terrain, together. However because these pixels still represent a continuous space, we must associate with each cell one or more nodes to discretize the state space for searching over.

There is ambiguity in where to place these nodes in the quad-tree cells. This section attempts to show that node placement can affect the accuracy of the traversal cost propagation in the path planner as well as completeness.

Here we consider the robot to be traversing the low fidelity continuous space from node to node. Here when talking about adaptive sampling we are concerned with adapting the size of each cell. This dictates that there must be a finite number of nodes in each cell, otherwise we have all the same completeness challenges raised earlier. Hence, presently the adaptive sampling changes the resolution of the search space, not the action set as in some of the previous examples.

We must place a certain number of nodes in each of the quad-tree-generated cells. However, problems arise when traveling between different resolutions because the nodes do not align nicely. Consider, as an example, cells with nodes placed in the centers, as in Figure 11, and with nodes at the corners as in Figure 12. Note how sometimes rather circuitous routes are forced, whereas in others infeasibility is introduced, when a path clearly exists. Such problems arise whether considering centered nodes, corner nodes, or even nodes at all corners

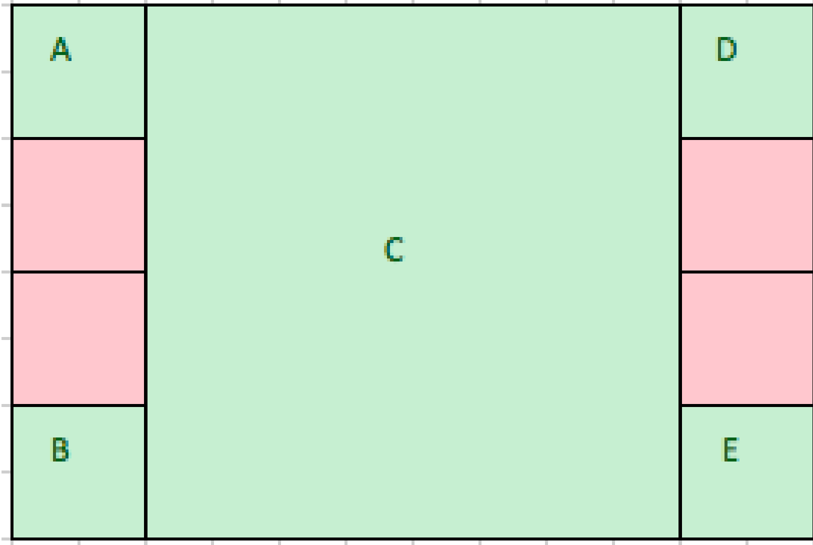

Figure 11 Interfacing Challenges 1. Here each Cell has a node at center, denoted by an uppercase letter. Consider a path from A to D. To traverse the lower resolution cell it must first travel to C, creating a rather circuitous route.

Figure 12 Interfacing Challenges 2. Travelling through A-C-D is now lower cost, however for any single node orientation there will be circuitous paths. Consider E-C-D for example. Further, B-C-E will clip the red obstacles leaving such a transition forbidden, however, logically a transition between B and E should be possible. In this way feasible paths in the pixel-level state space may appear infeasible.

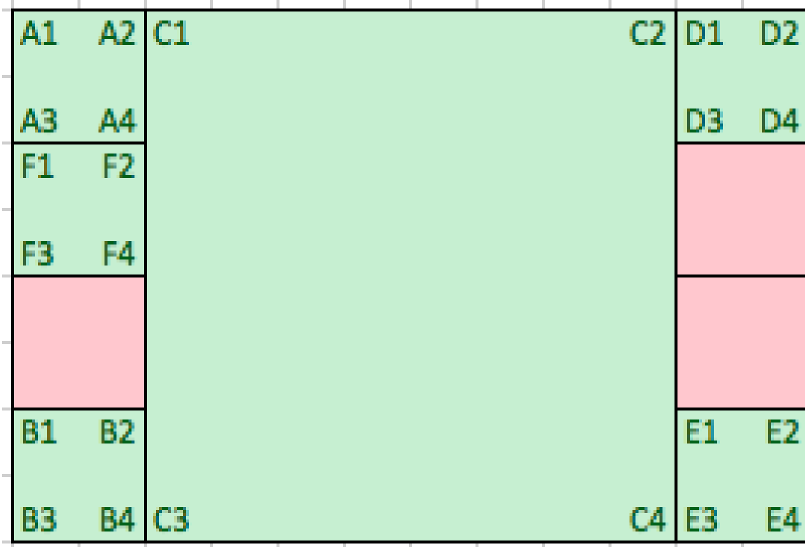

Figure 13 Interfacing Challenges 3. Placing nodes at all corners of each cell reduces many of the previously stated problems. There is now a feasible path and direct path from B to E and A to D. However, traversing from cell F to D remains circuitous as F and C have no aligned nodes. These problems result from misaligned nodes at different sampling densities.

In the above configurations there is no nice way to transition between cells of different sizes. This issue is explored in [26] where the authors suggest framing each cell with very small cells of the same terrain type (See Figure 14). This defines an efficient way to transition between cells of different sizes. While this framed approach does remedy the situation it creates a large number of additional nodes to explore over the whole map.

A1 A2 C1 C2 C3 C4 C5 C6 C7 C8 D1 D2
A4 A3 C28 C9 D4 D3
F1 F2 C27 C10
F4 F3 C26 C11
C25 C12
C24 C13
B1 B2 C23 C14 E1 E2
B4 B3 C22 C21 C20 C19 C18 C17 C16 C15 E4 E3

Figure 14 Framed Quad-Tree Approach. Here all of the aforementioned problems have been addressed. There is now a nicely transition defined between cells of different sizes. However, there are a lot of unnecessary nodes such as C5-6, C9-14, C17-20, and C23-28.

Node mirroring, introduced here, adapts the framed approach by only adding nodes where needed to define an interface. This is shown in Figure 15. There are much fewer nodes to expand in the mirrored approach. For a single transition, mirroring nodes addresses the interface problem more directly, however for multiple transitions as in Figure 16 the framed approach would yield a better path. We will see that this is remedied in Waypoint-A* without incurring the additional computational load of the framed approach.

A1 A2 C1 C2 D1 D2
A4 A3 C10 C3 D4 D3
F1 F2 C9
F4 F3 C8
B1 B2 C7 C4 E1 E2
B4 B3 C6 C5 E4 E3

Figure 15: Node Mirroring. Note how in the single resolution transition this yields all the same transition options as the framed approach, with much fewer nodes to expand.

A1 A2 C1 C2 G1 G2 D1 D2
A4 A3 C8 G3 D3 D4
F1 F2 C7
F4 F3 C6
B1 B2 C5 G4 E1 E2
B4 B3 C4 C3 G6 G5 E3 E4

Figure 16 Difficulty with Mirrored Nodes. Note that traversing from any node in F to any in E an indirect path must be taken via C3,G6, and G5m where a line of sight exists. The line of sight suggests that a straight-line path exists between these points. This will be remedied in Waypoint-A.*

A1	A2	C1	C2	C3	C4	C5	C6	C7	C8	G1	G2	G3	G4	G5	G6	G7	G8	D1	D2
A4	A3	C28							C9	G28							G9	D3	D4
F1	F2	C27							C10	G27							G10		
F4	F3	C26							C11	G26							G11		
		C25							C12	G25							G12		
		C24							C13	G24							G13		
B1	B2	C23							C14	G23							G14	E1	E2
B4	B3	C22	C21	C20	C19	C18	C17	C16	C15	G22	G21	G20	G19	G18	G17	G16	G15	E3	E4

Figure 17 Framing the Cell Can lead to direct paths. As opposed to the Mirrored node approach above. Here traversing from F to E can be done more directly via C26,C12,G25, and G14. However, note there are many unused nodes in the search.

While in some cases the mirrored approach yields a less direct path than the framed approach, it works nicely for Waypoint-A* since it will treat the nominal path as a set of waypoints that can be skipped if needed. The mirrored approach avoids the obstacle clipping discussed earlier by introducing a well-defined transition between each cell. This ability to transition between any two adjacent cells guarantees, in addition to the ability to find the smallest openings guarantees that using this method is complete. To rephrase, there are no adjacent cells, outside of obstacles, between which a path cannot traverse.

Rapidly Exploring Random Trees

It is good to note that there are other very successful sampling based techniques for differentially constrained vehicles. Of note is the Rapidly Exploring Random Tree (RRT) algorithms. These algorithms work by building a random space-filling tree to efficiently sample paths through the terrain. When dealing with differentially

constrained vehicles, BVP is used to see if each randomly sampled point is feasible to be reached.

These algorithms avoid maintaining a grid-lattice type sampling of the terrain; this is more memory efficient. Additionally, these algorithms tend to find a feasible path in significantly less sampling than grid-based searches. However, RRT planners can only find feasible plans, regardless of the sampling resolution.

RRT planners have an advantage for higher dimensional state spaces, and many have been successfully used for differentially constrained motion planning. A good discussion of these methods can be found in [27]. Another interesting approach: [28] samples whole path segments instead of nodes to achieve faster and more accurate results for systems with more complex dynamics.

Chapter 4: Planning Under Differential Constraints

Motivation

The stated goal of Waypoint-A* is to create low cost, natural, curved paths that respect the kinematics or dynamics of the robot, using a discrete planner such as A*. This work is clearly not the first attempt at doing such a thing, however it creates a nice extension to existing algorithms. This section will look at some other methods that relate to this topic.

In planners aimed at finding feasible, low-cost, or optimal paths for real-world vehicles, it is critically important to consider the dynamics and kinematics of the vehicles involved. In the examples in the introduction to A* section the planner did not incorporate any knowledge of any dynamics or kinematics. Indeed in those examples the path could veer in any direction instantly, creating a jagged path. In many real-world vehicles kinematic constraints, such as turning radius, and dynamic constraints, such as steering rate, must be considered. Otherwise the vehicle may not be capable of following the path from the motion planner.

In this section we review some background with regard to planning with differential constraints. We consider a basic model to work with, state the general problem, and give some examples of how this task is accomplished. We focus on two approaches in this section: one approach encodes the differential constraints directly into the planning phase, and the other one creates so called any-angle planners that create paths that are not constrained to the grid state space.

Model

Most vehicles have non-holonomic differential constraints. This means that the differential equations cannot be fully integrated, i.e., there are certain states that cannot be arbitrarily changed. For instance, in the Ackerman's steering model for a car, Figure 1, the vehicle cannot instantly change heading without moving forward of backward. Nor can this vehicle move sideways in isolation of other states, rather it must perform a relatively complex parallel parking maneuver to move sideways. For the analysis of Waypoint-A* we choose the front-wheel-drive (FWD) version of Ackerman's model, because it avoids asymptotic behavior as the steering angle reaches 90 degrees. Although this is not a realistic steering angle, it made the software easier to work with and did not hamper the value of the algorithm.

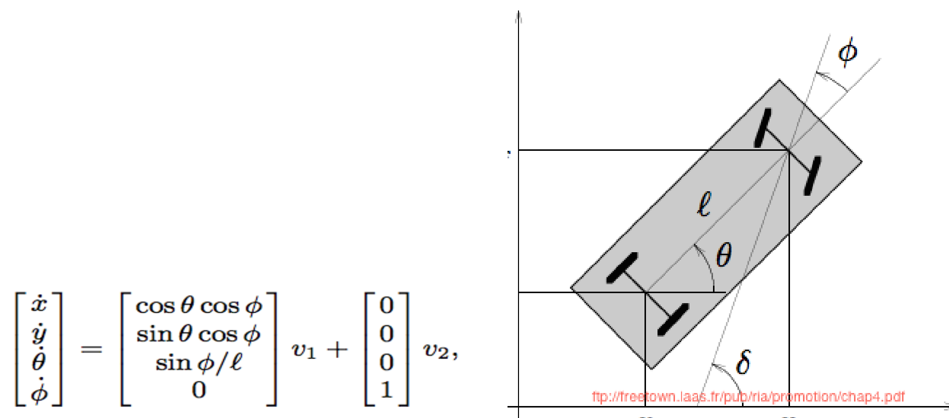

Figure 18: FWD Model for Ackerman Steering Car. Source [29]

Thus for non-holonomic vehicles the present system state depends on the path that was taken. This introduces some additional challenges to the planning. Firstly, the state transition function is now no longer a static mapping. The set of available next states depends on the current state. For example a vehicle must consider which

neighboring states are outside its minimum turning radius. Secondly, within the reachable states, a 2-sided boundary value problem (BVP) must be solved to determine what trajectory the vehicle must take to get to the next state. To make matters harder, there are usually infinitely many ways to get between 2 points, yet we can only select a finite number to search over for A* to be complete. Moreover, as we shall see, the BVP methods are not equipped to easily determine if a trajectory passes through an object, let alone weigh different trajectories between the same points for traversal cost [1].

Additionally, we want natural looking curves. These vehicles will likely either have human occupants or operate in the viscinity of humans, and as such creating paths that do not swerve excessively and transition from turn to turn smoothly are important. Meanwhile, A* must have a finite number of motion primitives to consider or else it will not be complete. There is a tradeoff between number of heading angles and computation speed, and so while technically a countably infinite set of heading angles could still be complete, the computation time would be severely impacted. Some algorithms discussed attempt to deal with this problem by choosing appropriate motion primitive, others try use post search techniques to free the path from the search grid lattice. For the most part attempting to solve the general BVP problem from initial to goal state is not done [1].

Differential Curves

When planning for vehicles with differential constraints we consider the system defined as $\dot{x} = f(x, u)$ such that $x \in X, the\ state\ space: a\ smooth\ manifold$, and

$action\ set\ u \in U \subset R^{m}\ bounded.$ Further, given $f(x, u)$ we can find $x(t) =$

$x(t_0) + \int_{t0}^{t} f\big(x(T), u(T)\big)dT$, [1].

When there are no obstacles or variable terrain to be considered, finding a feasible trajectory from the initial pose to the goal pose falls under the category of a BVP. However, BVP is not well suited for obstacle-laden terrain with varied terrain. When there are obstacles to consider the BVP must only search in regions that are not *regions of inevitable collisions*: $X_{ric} = \{x(0) \in X\ |\forall\ \tilde{u} \in U, \exists t > 0\ ST\ x(t) \in X_{obs}\}$. These are the regions for which, given the dynamics of the vehicle, no control action can divert a collision with an obstacle. Perhaps the vehicle is moving too fast to stop, or has nowhere to turn. As dimensionality of the vehicle increases, X_{ric} becomes increasingly difficult to calculate. Further, we consider what is called *the reachable set* $R(x_0, U) = \{x_1 \in X\ |\ \exists u \in U\ and\ t \in [0, \infty)\ st\ x(t) =\ x_1\}$. Together, these sets consider to which states the robot can go such that it will not result in a crash. A complete algorithm will find a trajectory from x_i to x_g with the dynamics, through the reachable set, avoiding X_{ric} and weighing in the traversal cost over terrain. This is very difficult to achieve, and so the problem is relaxed in most approaches [1].

<u>*Motion Primitives*</u>

Due to the difficulty of planning for vehicles with differential constraints, most planners in this application are sampling based. Instead of planning the entire trajectory, these planners consider what is called a *time-limited reachable set* $R(x_0, U, t) = \{x_1 \in X\ |\ \exists u \in U\ and\ t' \in [0, t]\ st\ x(t') =\ x_1\}$. This set determines to which states the vehicle can travel, given its dynamics, in a constant time frame [1].

Given a feasible state transition, there are usually infinitely many trajectories between the two-state that respect the vehicle dynamics. As discussed, a discrete planner cannot search them all so a finite subset must be chosen. The discrete planner considers this finite set of state transitions as the motion primitives for A*. In the *General Background* section the discussion considered an 8-way set of motion primitives, whereas in the present discussion possibly curved motion primitives are considered as in the previous figure. The general format of A* remains the same outside of the different motion primitives.

There is a tradeoff here. If the set of primitives is not rich enough then a vehicle may not be able to avoid an obstacle or make a turn in a critical section. Additionally, with a small set of primitives as in the figure above, turns may appear to swerve excessively, over-steering when only a slight deflection is needed. Too many state transitions, on the other hand, hampers efficiency of the code. It is not enough to have a large set of motion primitives, significant study has gone into creating a good toolbox of motion primitives to minimize dispersion and other metrics as in [26].

In some cases it can be difficult to enforce trajectories that comply to a grid lattice of state spaces. In these cases, one approach is to use a cell decomposition. In this case the terrain maps is broken into quantized regions as discussed earlier, but instead of associating some finite set of searchable nodes in this region, the whole region is considered as one state, see Figure 19.

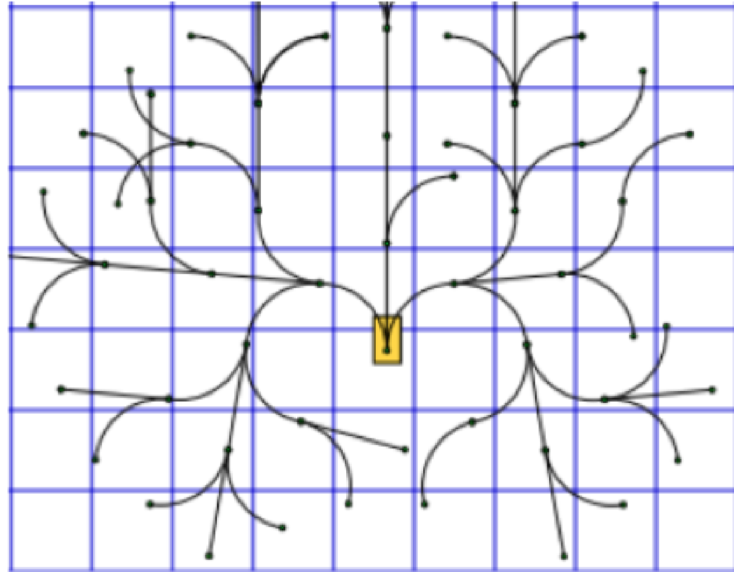

Figure 19 Cell Approach to Planning under Differential Constraints. Source [8]

In order for A* to converge in a discrete search there must be a finite search space in steering angles. With a continuum of steering angles there is no way to guarantee that the search is *systematic*, because we are not likely to ever arrive at exactly the same state twice. So some form of discretization of the steering angle is needed. Generally a small number of headings are considered, leading to either unnatural trajectories or not finding paths through obstacles when they exist [2].

Furthermore, the discretization of the search space into grids opens aliasing and quantization issues familiar from discrete signal processing. As an example see Figure 19. At the upper right grid space (8,1) there is an aliasing error. Here two curves that clearly (given the scale of the grid) are quite separate will both be considered to have entered the same (x,y) state as far as A* is concerned. This may create propagating error as far as future reachability of the paths as well as creating discontinuous paths if not attended to. On the other end of the spectrum, in the region

about (2,2) there is a quantization problem: there is a tight clustering of points there. The first problem with this case is that there is uneven sampling in (2,2) while (3,3) is entirely skipped. Now it is likely that as A* expands this reachability graph there will be some path that loops back to (3,3) however this is clearly unnatural as a direct path to (3,3) from the robot's start is within the turning radius of the vehicle. Moreover, there are three curves, one going to (1,2) and another to (2,2) and (2,1) that are considered to be in different states in the grid, but are quite close to one another and not truly representative of a difference as large as the grid spacing implies.

These issues are addressed by doing one of two things: *trimming* or specialized choice of search space. *Trimming* only allows for one end-point of a curve per grid square, gaining its name by "trimming" out curves leading to aliasing as in (8,1). This will remedy the discontinuity possibilities and will lead to smooth trajectories [1].

An additional challenge is that it is not trivial to calculate the transition costs over curved motion primitives. In this case A* is transitioning between states as a knight in a chess game: moving between non-tangential grid spaces at times. How does one calculate the traversal cost from grid cell A to grid cell B when it partially passes through grid cell C in a curve?

One can also custom mold the motion primitives for grid spacing. This would be set up the search so that all actions always land in the middles of each grid space. This eliminates all these aliasing and quantization issues. This is explored next.

If instead one selects motion primitives that do start and terminate on lattice nodes, one can utilize much of the powerful discrete grid search algorithm techniques.

For example, in [25] they use a pre-calculated library of motion primitives, some of which are replicated in Figure 20. Actually the method presented in this paper utilizes some of the adaptive sampling techniques discussed earlier, varying the number and quality of motion primitives in different regions of the map as necessary. This technique produces nice smooth trajectories, with its clever choice of predefined motion primitives.

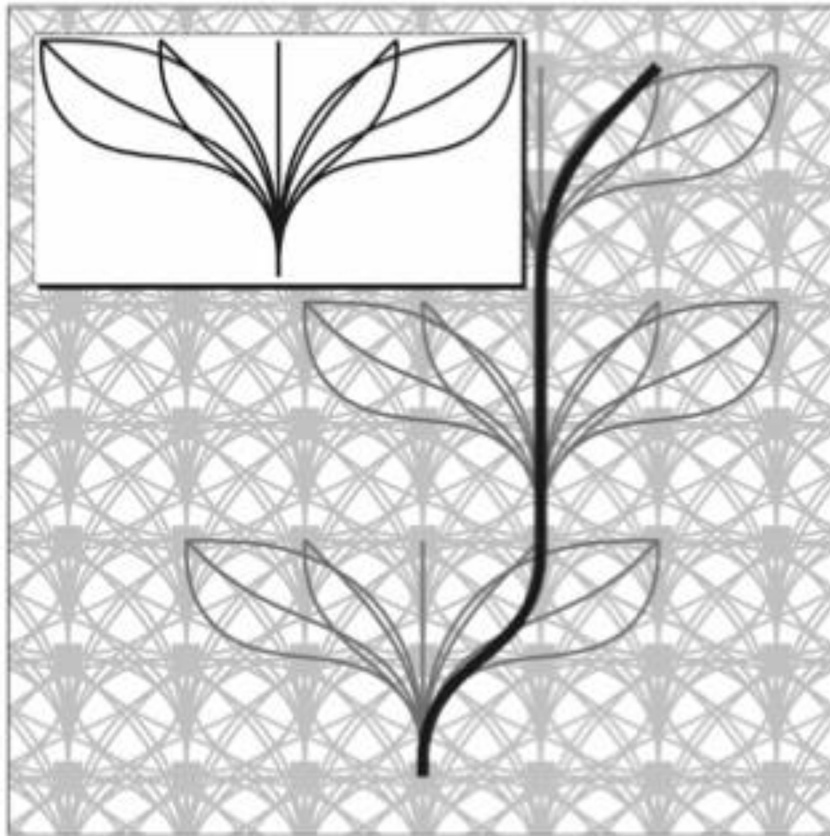

Figure 20: An intelligent Choice of Motion primitives. Source [25]

Any Angle

Another approach to addressing the finite number of departure headings in a discrete search is to create a framework that can calculate a cost for a vehicle departing a node at any angle. These algorithms, like Waypoint-A*, do not consider the dynamics of the vehicle during the path construction; this is perhaps their greatest drawback. However, they do allow for a very large number of departure angles, sometimes infinite number. They are also lightweight and mathematically simple.

The next method frames each grid space with searchable nodes [30]. It uses straight line segments native and efficient in the discrete search but adds an additional insight. It separates the notion of the discrete grid of terrain, called a *cell* here, from the (x,y) location that is traversed to in the search over the terrain called *node* here. In the past two algorithms it was assumed that the concepts are one in the same. Here, for each *cell*, there are several *nodes*.

Instead of treating each quantized terrain unit as one location to travel to, the framed cell approach claims that although the terrain and search locations must be sampled to create a finite search space, they do not have to be sampled at the same rate. Here they analogy is thus: the robot on its path traversing through some *cell* will have to enter that cell somewhere along the boundary and also leave somewhere along the boundary of that *cell*. Given knowledge of the terrain cost of the cell in question as well as the segment length of the path through the cell we can calculate the cost of traversal as the segment length times the cost of the cell (insofar as the cost is a constant).

However, the discrete planner must still have a finite number of nodes to search. This method then frames each cell with some number of nodes per each edge. This frees the planner to travel in many more arrival and departure angles from each cell than in previous methods. This can be seen in Figure 21.

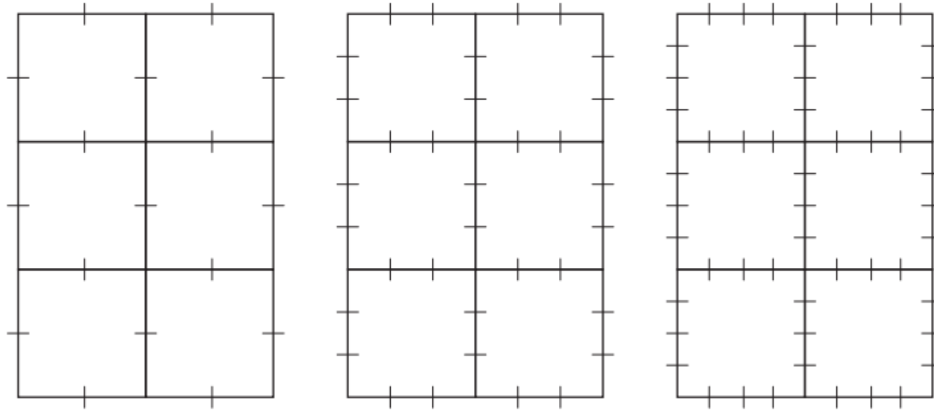

Figure 21: Each cell is framed with a predetermined number of nodes. A path is free to travel between any of these nodes, thus increasing path angle possibilities. Source: [30]

The framed approach yields some nice improvements. Firstly it does indeed allow for a larger variety of departure and arrival angles between cells. This, they argue, allows for paths that are more natural and cheaper, because of reduced zigzagging. Additionally, as cell resolution increases the generated paths look even smoother. In this way applying a post-processing smoothing operator incurs less potential error since the path is closer to smooth.

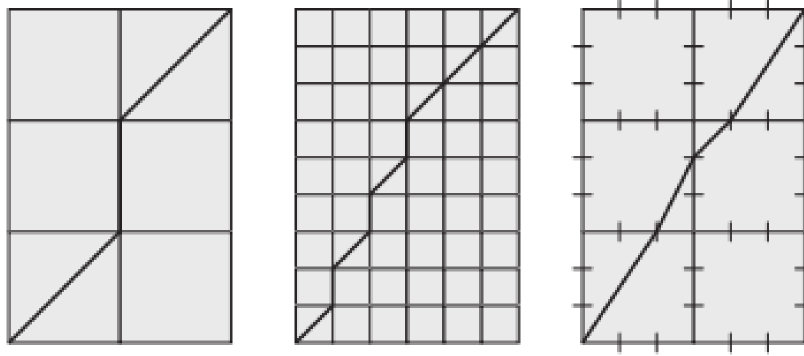

Figure 22: The framed cell approach on the left definitely creates a more direct path than a normal grid search on the right. In fact it outperforms the grid search even with 9 times the grid spaces. Source: [30]

Although this is a nice idea there are some drawbacks that will lead on to the next algorithm. Firstly there are still only a finite number of departure headings that the robot can take. As mentioned it is desired to allow for any-angle of departure, to allow the robot steering to be expressed as a continuum. Again, true that if the cell resolution is high enough or the node frame density is high enough the path gets smoother, but this is at the cost of computational complexity and does not actually solve the problem.

Field D* [2] answers some of the concerns raised with the framed cell approach. It extends the notion of arriving and departing through edges of a frame by using interpolation. Instead of calculating the paths through a finite number of points as framed cell does, Field D* uses just the corners of each cell and interpolates the cost of leaving through anywhere on the edges between these corners. This interpolation is an approximation, however the authors of the paper show that it works well in practice. In this sense, this is the first so called a*ny-angle* algorithm considered here; The planner can find a cost -- perfect accuracy aside -- for departing each node at any angle on a continuum.

For clarity's sake, Field D*, as presented in the cited work, is built with D* lite not A*. D* lite is a dynamic evolution of A* which allows for fast re-planning. As mentioned Waypoint-A* is not dynamic, even though at its core there is no reason it could not be. Regardless, the major concepts presented apply equally well to A* searches where we want *any-angle* trajectories.

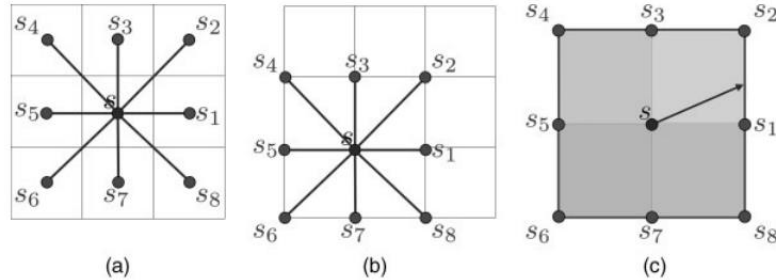

Figure 23: Classically the node is considered to be at the center of the cell: (a). In order to interpolate more easily Field D moves the node to the upper left corner (b), in that way when interpolating cost-to-go as shown in (c) the path travels through only one type of terrain. This makes the interpolation calculation easier. Also note, that any path leaving* ***s*** *must pass through the boundary outlined by* $\boldsymbol{s_1}$*-*$\boldsymbol{s_8}$*.* *Source:* [2]

As in any grid search, every *node* has a set of *neighbors* reachable in one action u in the *action set* U. In Field D* a standard 8-way *action set* is used, resulting in the set of neighbors of state s, $ngbr(s)$, s_1 through s_8, as seen in the Figure 23(c). In a standard discrete planner with this *action set,* state s would only be allowed to transition to any of $ngbr(s)$ in the path creation stage, possibly creating one or both a jagged and inefficient path. See Figure 23 below.

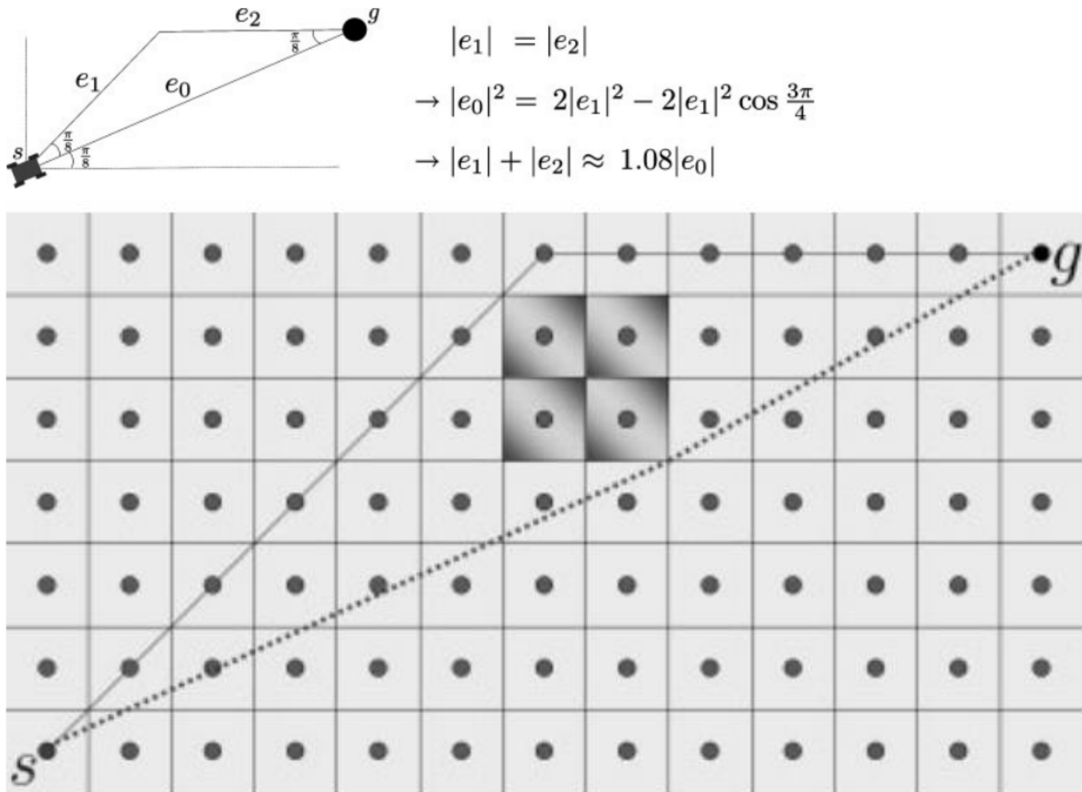

Figure 24: Limiting the state transitions to only corners of cells can lead to error and unnatural paths. Here the error induced is 8%, and worse the obvious straight-line path cannot be taken. Source: [2]

Field D* however, notes that $ngbr(s)$ outlines a boundary around s through which any path must cross, whether it be through one of the neighbors themselves or any line segment of this boundary. See Figure 23, above. Call this boundary S^b and the points in it: s^b.

If the cost was known for all points in S^b then optimal departure spot on this boundary could be calculated by minimizing $G(s) + C(s, s_b)$ over all points s^b in the boundary. In this formulation $G(s)$ is the *cost-to-go* of state s and $C(s, s_b)$ is the *cost-to-come* from s to s^b and can be calculated as a line integral from s to s^b over the terrain traversal cost.

To simplify the cost calculation Field D* considers each *node* to be located on the upper left hand pixel of each *cell* as in Figure 23. In this way traversing in any

angle between 0 to -π/2 results in traveling in the same cell. Then each traversals is over a uniform terrain. Now the line integral simplifies to $L(s, f^{-1}(s, s^b)) = \|s - s_b\|_2 * cellcost.$

A challenge still remains: there are an uncountably infinite number of points $s^b \in S^b$. This essentially reduces to the same issue mentioned above necessitating finite heading angles. However, Field D*, presents a linear interpolation assumption: given the minimal *cost-to-go* at each *node* at each *cell* corner, one can use geometry to interpolate the minimal *cost-to-go* for any point on any line segment between any two nodes. They create a closed-form expression for this cost, which they minimize.

In fact, with this method we treat the nodes as a sampling of a continuous space. Thus entrance and departure into cells is not restricted to cell corners. A path might start at a node, but need not travel between nodes. In this sense, this algorithm has abstracted nodes to a further degree than the framed cell method. Now, nodes are explicitly treated as a discrete sampling of an underlying continuous cost field from the *initial state* to the *goal state*. The authors provide an analysis of the appropriate geometry and linear interpolation to resolve this continuous cost field at all S^b.

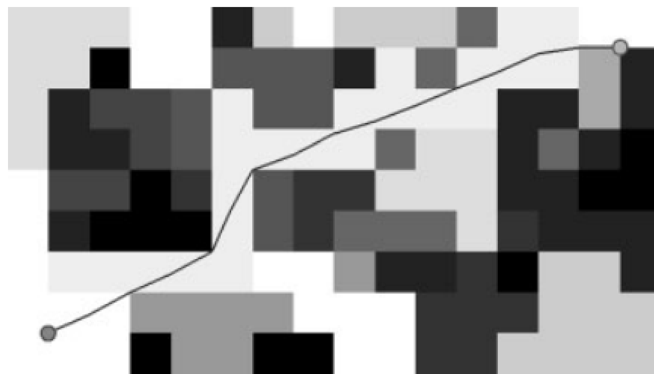

Figure 25: The path generated by Field D.Different grid colors indicate different terrain costs, darker implying higher cost. Note that the path is not restricted to nodes at cell corners. Source:* [2]

Field D* is the first true *any-angle* algorithm considered, because it truly supports a continuum of heading angles. However, the path is still constructed of jagged line segments. Hence as with the framed cell approach there is still a trade-off between resolution and smoothness versus computational speed. Although, Field D* needs less resolution increase for the same level of smoothness as the framed cell method due to the continuum of headings, one must still increase the resolution to reduce the jagged appearance of the path. Finally, even if the path is smoother than other approaches, the curves in the path were created with no consideration for the dynamics of the vehicle itself, and as such there is no guaranty that the path is even feasible for a given real robot.

Other similar approaches have been carried about in theta* [31] and T* [32] incremental phi* [33]. But it is not particularly helpful to expand on this.

Chapter 5: Waypoint-A*

Motivation

So far this thesis has discussed the concepts of grid search, adaptive sampling, and dealing with differential constraints. Waypoint-A* falls in the intersection of these categories. Waypoint-A* is a lightweight, grid-based search that uses quad-tree decomposition to adaptively sample, and is aimed at application for vehicles with non-holonomic dynamic constraints.

Waypoint-A* does not pre-calculate curved motion primitives, nor is it an any-angle algorithm. Instead Waypoint-A* calculates a discrete grid search to find a jagged nominal path from x_i to x_g. It then considers the nodes involved in this path as waypoints along a feasible, low-cost path through the continuous ground truth. For a vehicle in a given position the algorithm applies a line-of-sight cost calculation to determine to which waypoint to travel to minimize the cost to the goal. This direction is treated as the ideal heading, on a continuum, for the vehicle, which then attempts to turn in that direction as governed by its dynamics. Hence, path production is a two-step process: a jagged discrete path with no consideration for the constraints of the vehicle, and a second step that allows the planner to express those constraints. In this sense, Waypoint-A* falls somewhere in between those algorithms that incorporate the dynamics directly during the planning phase as in BVP or curved motion primitives,

and those any-angle algorithms that are free to depart in any direction, but do not account for dynamics in the planning at all.

Due to the second phase of planning that uses waypoints, Waypoint-A* can benefit greatly from adaptive sampling. If the vehicle is not constrained to the nominal path, but can, under the correct conditions to be discussed, skip waypoints altogether, then there is no need to sample densely over uniform terrain. Instead the waypoint finder simply calculates the best path regardless of lower node density in these regions. This reduces calculation time. Waypoint-A* does benefit from the increased sampling rate near varied terrain, however.

Elements

Waypoint-A* has three main phases: one sampling phase, and two planning phases. During the sampling phase, the algorithm takes in as input a traversal cost map. This map data is assumed to be sampled at the highest relevant resolution, and cleaned up from any noise. That is, each pixel is assumed to be relevant data at the highest relevant resolution for the current application. This map is then adaptively sampled using quad-tree and mirrored nodes as discussed in the sampling section. Next the two planning phases start.

In the first planning phase, called nominal path finding, a grid search is executed with no regard for the kinematics or dynamics of the vehicle. The only feature differentiating this from standard A* is how it defines the valid state transitions. For a given node, its neighbors, i.e., those nodes that can be travelled to as dictated by the state transition function, are all other nodes in the same adaptively sampled cell as well as the other nodes in its mirrored group. In this way the planner

is allowed to consider all defined transitions within a cell of uniform terrain, and also has a defined, controlled method of crossing into other cells.

The output of the first planning phase is a nominal, jagged path from the initial state to the final state. As discussed if such a path exists given the 1-pixel resolution then this phase will find it. In this sense this phase is resolution complete. The *examples* section below will illustrate these steps.

The second phase decomposes the nominal path to the set of nodes it passes through, named vertex nodes: $v_k \in V$. In Figure 27, for instance, these are the blue circles intersecting the red nominal path. Additionally, denote *Full resolution terrain cost:* $T(x, y)$. The cells in the planner are a grouping of quantization of the terrain, i.s., pixels. $T(x, y)$ is the cost of each of these smallest units of terrain from the robot's sensors. Then given the vehicle position, whether the vehicle is on the nominal path or off, the planner calculates to which waypoint the vehicle should travel next using a cost estimate. It should be noted, that although the vehicle starts on the path at the initial pose, there is nothing stopping it from leaving the path. Indeed this is critical and will be discussed below.

To find which waypoint is ideal, Waypoint-A* uses a *cost estimate* $C_E(x, y, v_k) = G(v_k) + \int T(C)dC$. C is a parameterized straight line path from the vehicle position at (x,y) to the coordinates of any one v_k. At the robot's current location, the cost of traversing through the terrain to the given vertex is added to the *cost-to-go* of this vertex. The heading vector points the robot towards the vertex $v_k \in V$ corresponding to minimum among costs calculated.

Although in the figures below the heading vectors are calculated at a regular

interval for graphing, Waypoint-A* calculates these vectors continuously at the location of the robot. This closes the feedback loop and provides the vehicle with a means to correct for drift due its dynamics. The current examples uses the front wheel drive model presented earlier in the paper in Figure 18. The vehicle's controller, while respecting the steering dynamics, adjusts the steering input to match the ideal heading. In this way the next state of the vehicle is calculated directly through the dynamics.

Examples

This section covers several instructive examples that illustrate some nice features of this algorithm as well as some limitations that need to be addressed. The first example was brought up earlier in the discussion of any-angle algorithms. It is reproduced below. In the context of Field-D* it was notes that their any-angle approach allowed a direct path to be found between the initial state and the goal state. Here too, Waypoint-A* accomplishes the same thing, while allowing for curved paths.

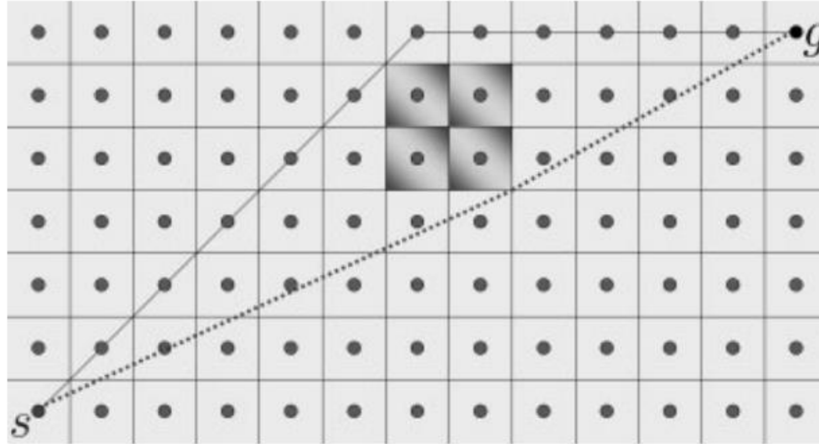

Figure 26: Any-Angle Advantage. Searching an 8-way neighbor grid results in the solid circuitous path, whereas there clearly exists a more direct path to the goal (dotted path). Waypoint-A will find this direct path as well. Source:* [2]

The nominal path that results from applying Waypoint-A* to a similar map as in Figure 26, is shown in Figure 27. Note how the adaptive quad-tree based sampling divides the space so that each cell has only one type of terrain. Additionally, the adaptive sampling results in a path of the same length as in Figure 26 with fewer nodes to search. The nominal path here travels beneath the obstacle instead of above, this choice is arbitrary to the planner as it results in the same cost.

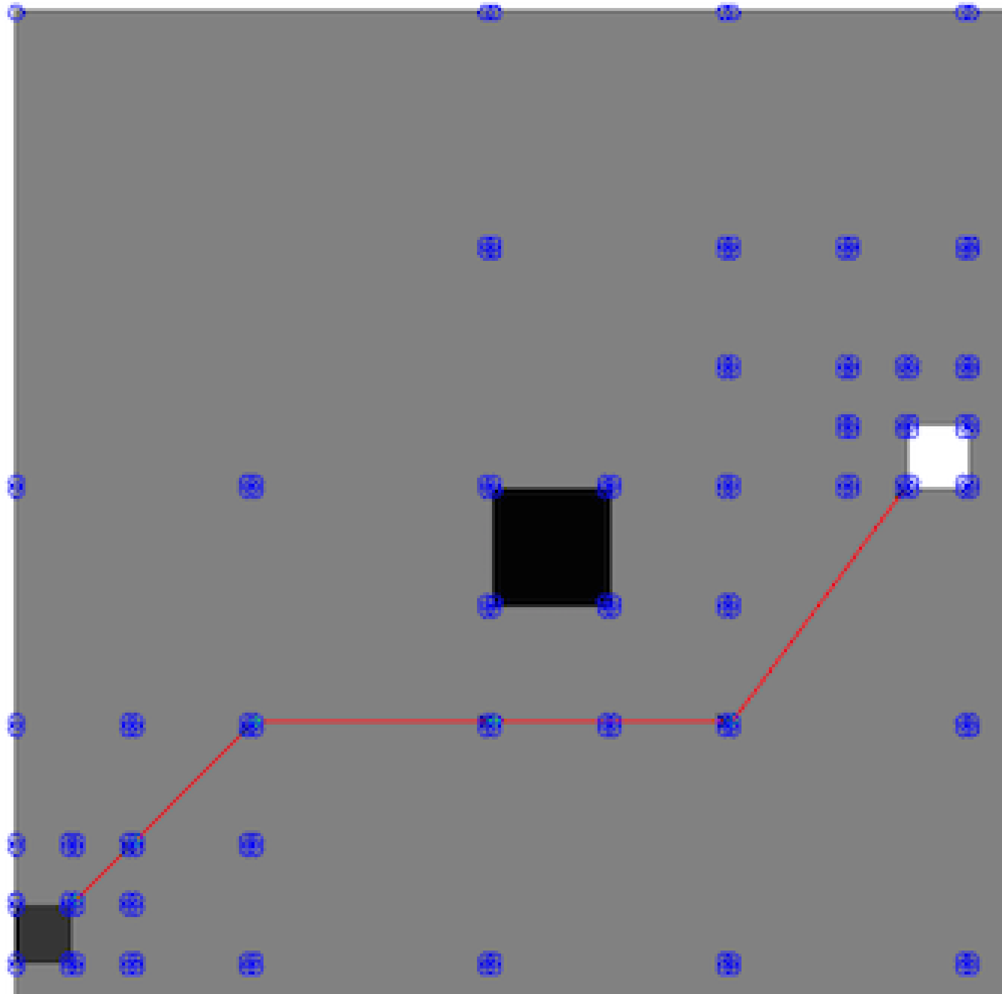

Figure 27: Example 1, Nominal Path. Note how the nominal path is of the same length as in Figure 26 with fewer nodes to search. The nodes are the blue circles.

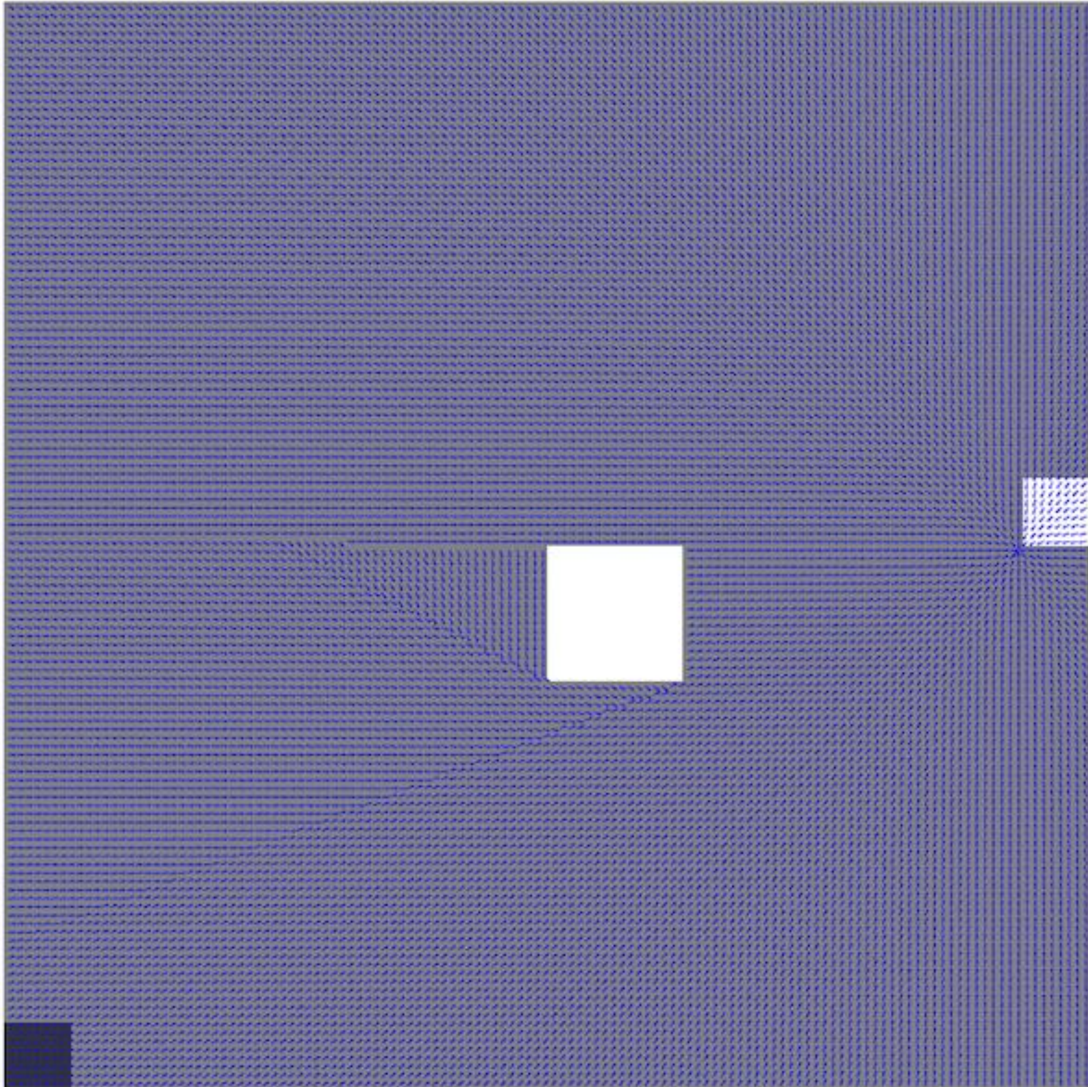

Figure 28: Example 1, Big Picture heading vector field. The region on the bottom left is the initial region, any point in there is a valid start. The region on the right is the goal region, any pose in this region is a valid end point. The planner, however, selects a single state in these regions for the nominal path. The white central block is an obstacle, it has infinite traversal cost.

For demonstrative purposes, Figure 28 shows the vector field of headings produced by the waypoint phase of the algorithm at a regular spacing on the map. There are some interesting behaviors here. With the exception of the obstacle, the terrain is uniform. As such, whenever there is a line of sight to the waypoint at the goal, the heading vector points skips all intermediate waypoints and points the vehicle directly to the goal. This happens because with no varied terrain to consider, a straight

line is the shortest -- hence lowest cost -- path between two points. When there is no direct line of sight to the goal, Waypoint-A* directs the vehicle toward the waypoint with the lowest cost factoring in cost-to-go from the waypoint and cost to get to that waypoint. In the present case where the terrain is of uniform cost, this amounts to guiding the vehicle to the closest waypoint to the goal. These features can be seen in Figure 29, which zooms in around the obstacle.

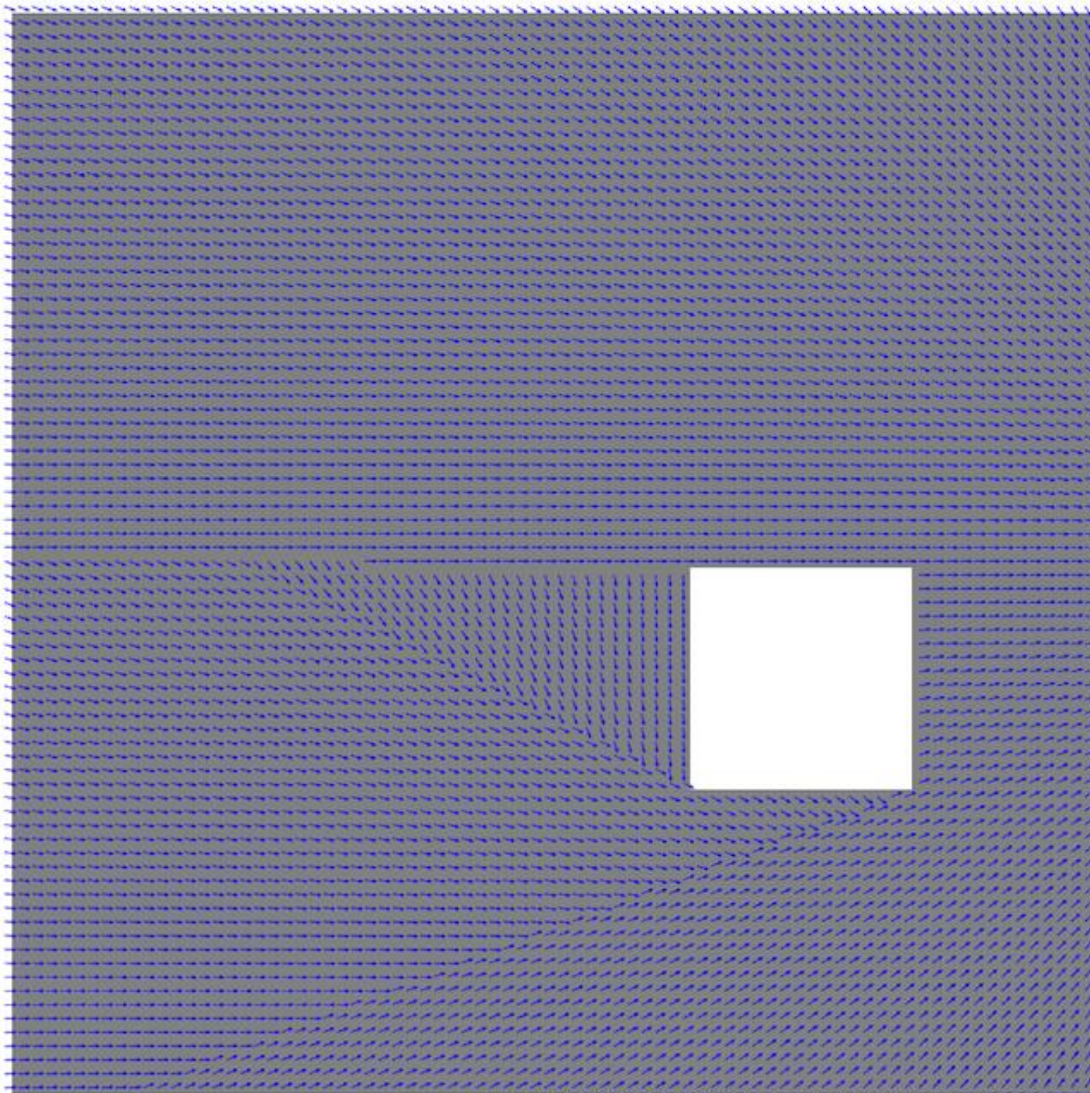

Figure 29: Example 1, Zooming in on Obstacle. Above, below, and to the right of the obstacle, where there is a direct line of sight to the final waypoint the heading field points directly to that final waypoint. However, when an obstacle blocks the line of sight, as on the left of the obstacle, the heading field instead points to the furthest waypoint it can travel to in a straight line.

In actual application, Waypoint-A* does not calculate the heading vector field, rather creates a heading vector at the vehicle's instantaneous position as part of a feedback controller or forward simulator. A forward simulation of an FWD Ackerman car in several starting poses can be seen in Figure 30. Note how these drive the vehicle smoothly at a variety of initial headings in a direct route to the goal state. When the vehicle is facing towards the goal state already, the path is a straight shot. However, when turning is required it is performed in a smooth, predictable, and direct manner. The trajectories are created by forward simulating the dynamics of the robot so the results are feasible curves for the vehicle. The vectors, represent the heading for the most direct path to the best waypoint to the goal. The forward simulator treats these waypoint headings as the ideal heading and so sets the steering angle towards the direction of the heading. Here we assume instantaneous steering rate, but a more complex model could easily be considered.

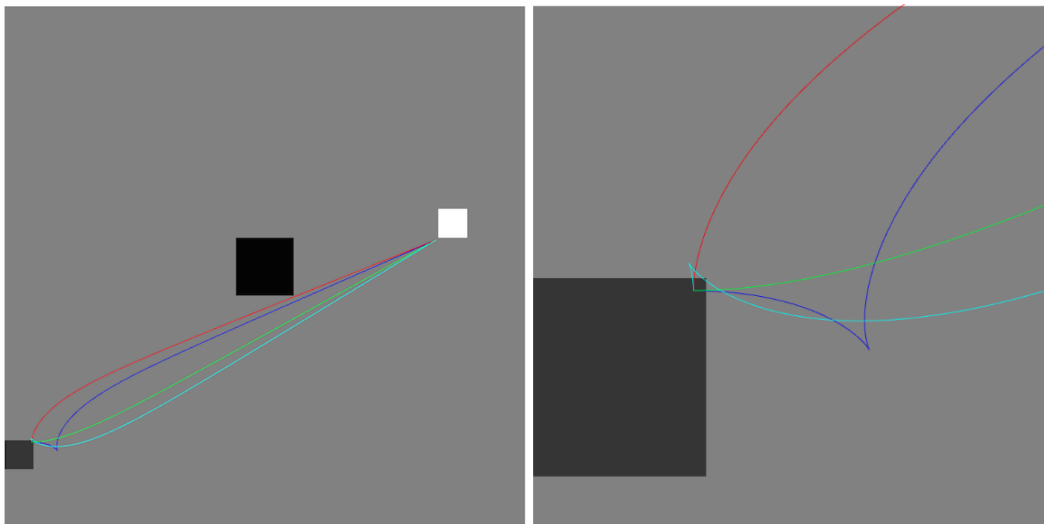

Figure 30: Example 1, Sample Low-Cost Trajectories. Initial poses for Red, Green, Cyan, and Blue are up, right, down and left, respectively. When applicable the vehicle can reverse, as seen in cyan and blue, to head towards the goal in a more direct manner than U-turning without reversal. Notice that the paths are smooth, direct, and predictable with no swerving. These paths avoid the obstacle.

In the next example we consider the feat of rounding a corner. This example was used earlier to exhibit adaptive sampling, and one can see its efficacy here as well. Again, over uniform terrain sections the heading field opts for the waypoint furthest towards the goal that is within a line of sight. Waypoint shortcuts cannot travel through the obstacle in the middle, because this would result in an infinite cost.

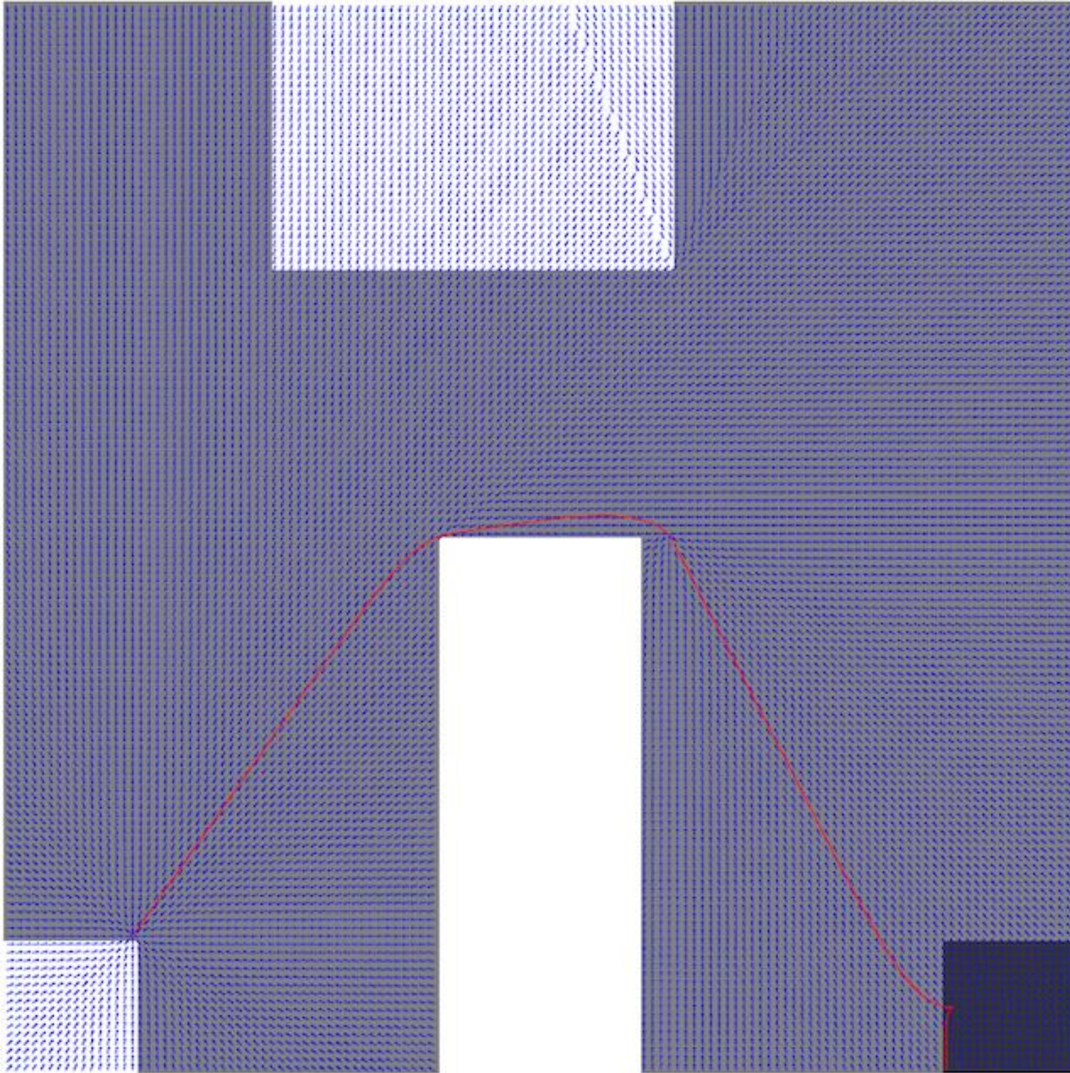

Figure 31 Example 2. This map was instructive for the discussion of quad-tree decomposition, so it is natural to show how it performs with the full Waypoint-A algorithm. The gray region on the right is the initial region, and the goal region is the white on the left. The central white pillar is an obstacle. The addition of the white high cost terrain on the top will be used to illustrate the non-greedy nature of the waypoint selection.*

Additionally, note that the white block at the top is not an obstacle, rather a region of high cost. The waypoint algorithm is not greedy, it does not just point to the furthest waypoint that does not traverse an obstacle. Rather, it considers the cost of traversing all the terrain in a straight line to each waypoint. This is exhibited by the fact that for a certain distance into the heavy terrain from the right, the field indicates it is best to turn around. Yet after the vehicle is deep enough it is deemed better to just plow through the high cost terrain to the best available waypoint. This is zoomed-in to in Figure 32.

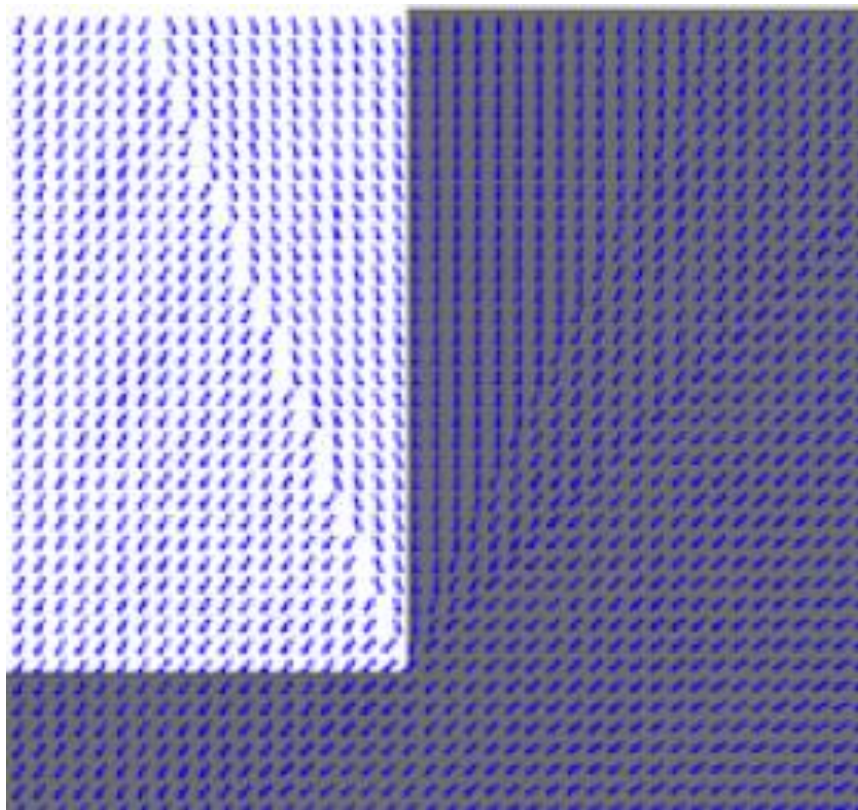

Figure 32: Example 2, Waypoint-A is not Greedy. On the right of the high cost terrain the algorithm draws the vehicle away from the high cost region and to a waypoint that is not as far along as can be maximally reached with a line-of sight. This happens because it is deemed more expensive to traverse the high cost terrain than to take a longer route. Even once inside the high cost region, for a while it is better to turn around and leave the region. However, if the vehicle finds itself deep enough into the region, at a certain point is it calculated to be better to continue through the obstacle to a waypoint closer to the goal.*

Finally, note a disadvantage here: there is no look-ahead and the heading calculator only considers straight line paths. In other words, the waypoint algorithm finds the best waypoint to travel to without considering what happens next. This causes the sink near the top of the right corner, shown in Figure 33. This leads to the overshoot on the right corner. If the planner could incorporate the knowledge that the waypoint in question is followed by a sharp left turn, then it could alter the heading calculation and set the vehicle in a better trajectory to navigate the corner with less overshoot. On the other hand, even with this drift the vehicle knows how to effectively return to the path.

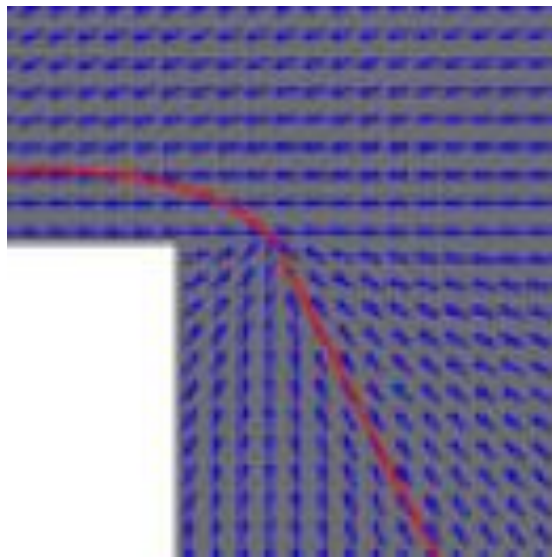

Figure 33 Example 2, Overshoot Due to No Look-Ahead. Due to no line of sight with further waypoints, all heading vectors to the bottom right drive the vehicle to the waypoint at the center of the sink, with no consideration for later waypoints. Thus, the vehicle does not set itself up to make the turn without overshooting the corner slightly. This will be a problem later in tighter spaces.

Next consider a map with varied terrain, as in Figure 34. In this case the straight line of sight to the waypoint closest to the goal will not always be the best answer. Notice that the vector field is no longer as aesthetically smooth and

continuous, rather it changes at the interface between different terrain types. Yet, regardless the trajectories are always smooth.

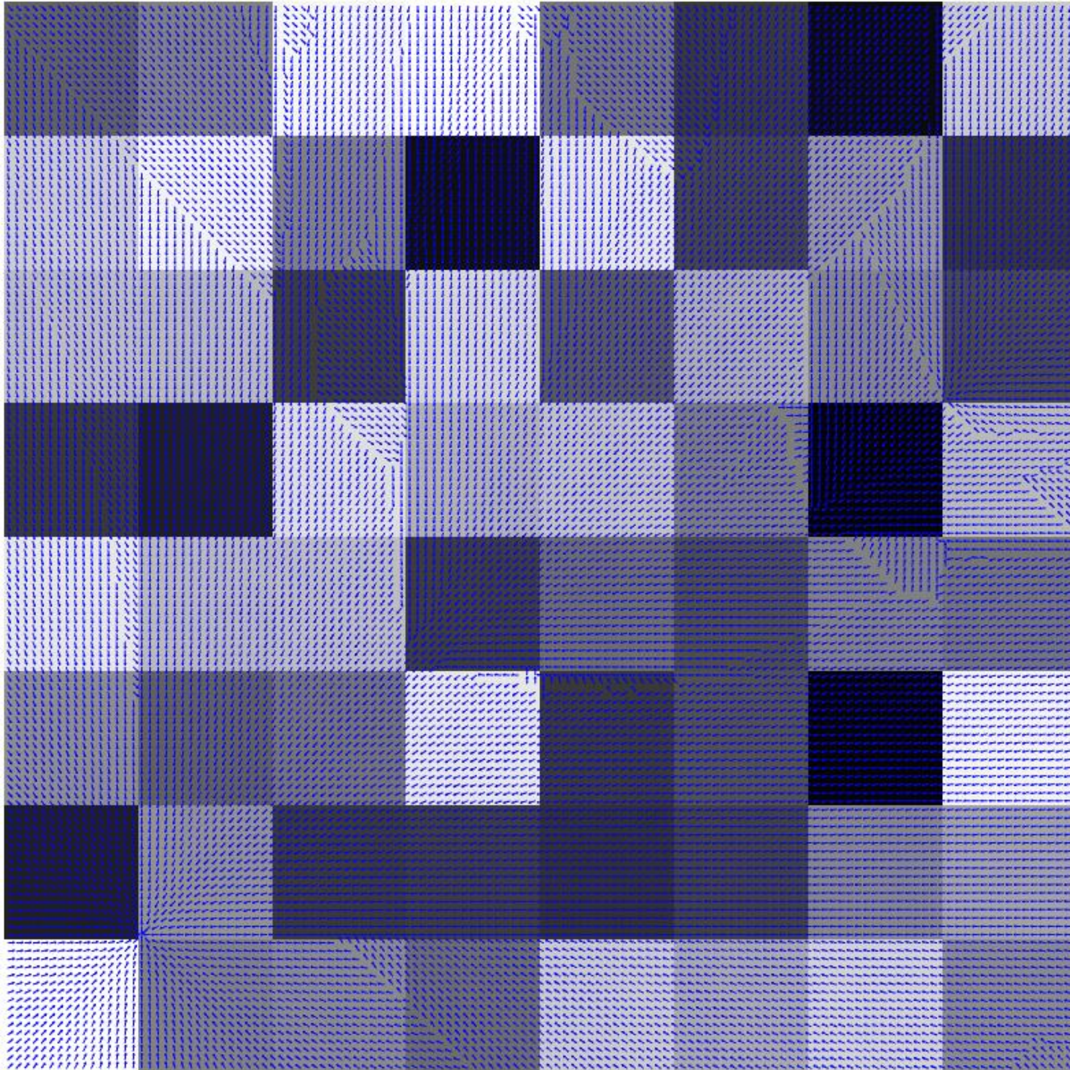

Figure 34: Example 3, Varied Terrain. In this example each block of terrain is assigned a different cost, as indicated by the color. The lighter blocks are more costly. There are no obstacles here. The initial region is 3 down on the right, and the goal region is on the bottom left corner. At the interfaces between cells of varying terrain there are sometimes non-smooth changes in the heading field. Although the heading field does not drive the vehicle through there non-smooth interfaces, due to the dynamics of the vehicle it may cross these interfaces. Sometimes this will cause the vehicle to turn around, and sometimes it will continue through, possibly changing heading. Regardless the paths are smooth and natural.

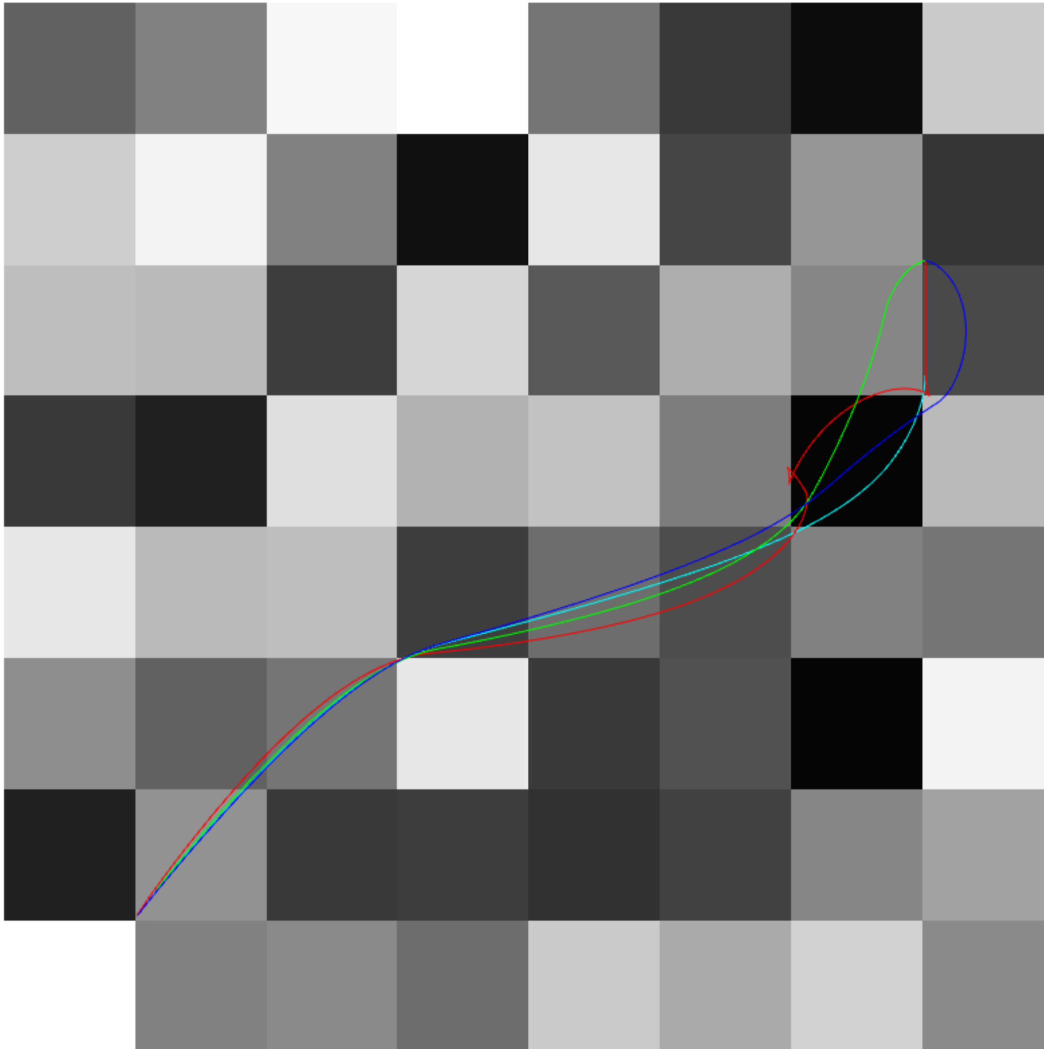

Figure 35: Example 3, Trajectories Through Varied Terrain. The trajectories red, blue, cyan, and green start pointing up, right, down and left respectively. Notice that despite what might be predicted looking at the rough heading field in the previous figure, the trajectories are smooth and do not swerve or jitter.

Notice how in red trajectory there is one 2-point turn and one 3-point turn. The first is due to the fact that the vehicle is initialized pointing upwards and it is faster to simply reverse than turn around. Then the vehicle turns to point towards the goal, however it cannot make the turn given its turning radius, so it performs another 2-point turn so that it can make the turn. All of this is performed by simply adjusting the steering angle to point the vehicle toward the calculated heading. This simplicity, however reinforces a weakness of this algorithm: there was no look-ahead to verify if

the vehicle driving the red path would be able to complete the turn, this will be discussed further in the next example.

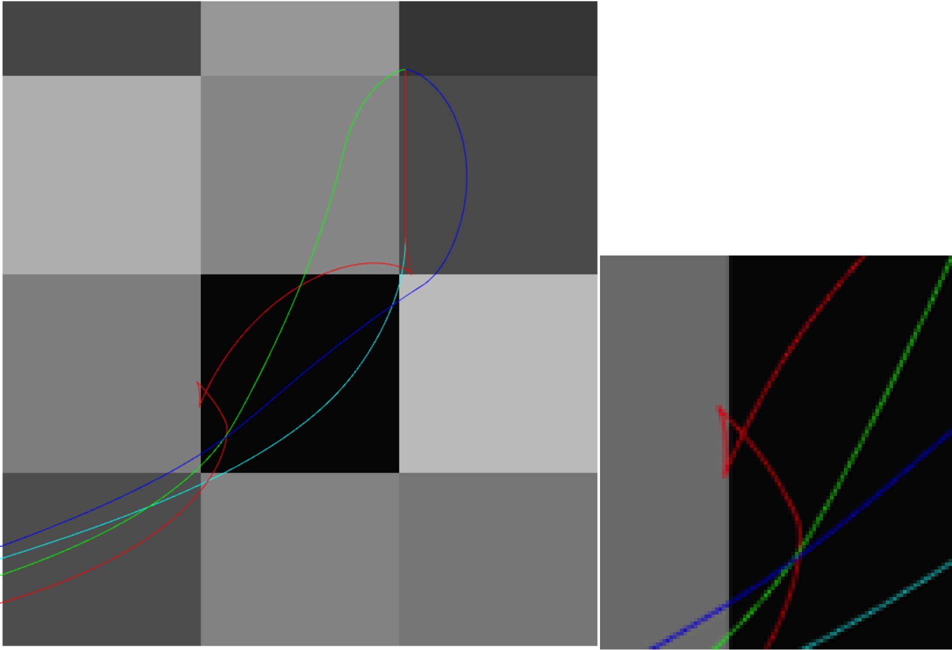

Figure 36: Example 3, Zoomed In. The trajectories indicated with blue, green, and cyan drive forward in nice arcs towards the goal. Meanwhile, in red the vehicle is initialized pointing in the wrong direction, and it reverses performing a 2-point turn. Then it cannot make the second turn through the black region, and it is deemed cheaper to perform a 3-point turn than to traverse the costlier terrain.

Finally, consider the example shown earlier to illustrate the completeness of the quad tree decomposition replicated below in Figure 37. This will outline some limitations of the algorithm. This example was used to show that using quad-tree Waypoint-A* will find a nominal path if one exists, even if the path must squeeze through tight areas. Indeed an optimal (given the sampling) nominal path was found, while saving much computation with adaptive sampling. However, the lack of consideration for the ability of the vehicle to actually make a turn causes the vehicle to crash in this scenario.

Figure 37: Example 4, Tight Maneuvers. This example was used earlier to exhibit the completeness under quad-tree decomposition. However, the tight nature of the turns will cause the robot to crash.

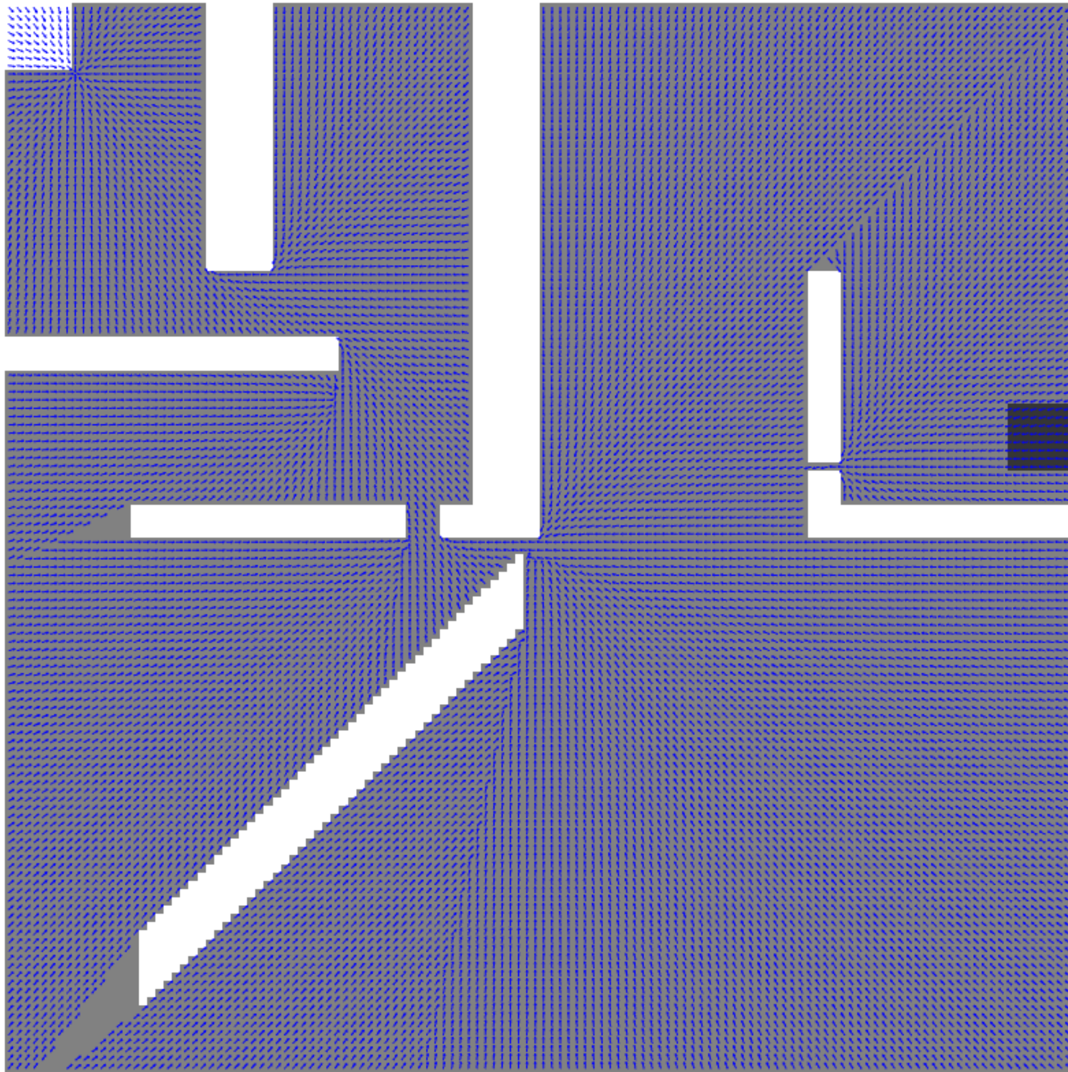

Figure 38: Example 4, Heading Vector Field. Here one can see a similar vector field layout as in the previous examples with the exception of the undefined regions. In these regions there is no line of sight to waypoint.

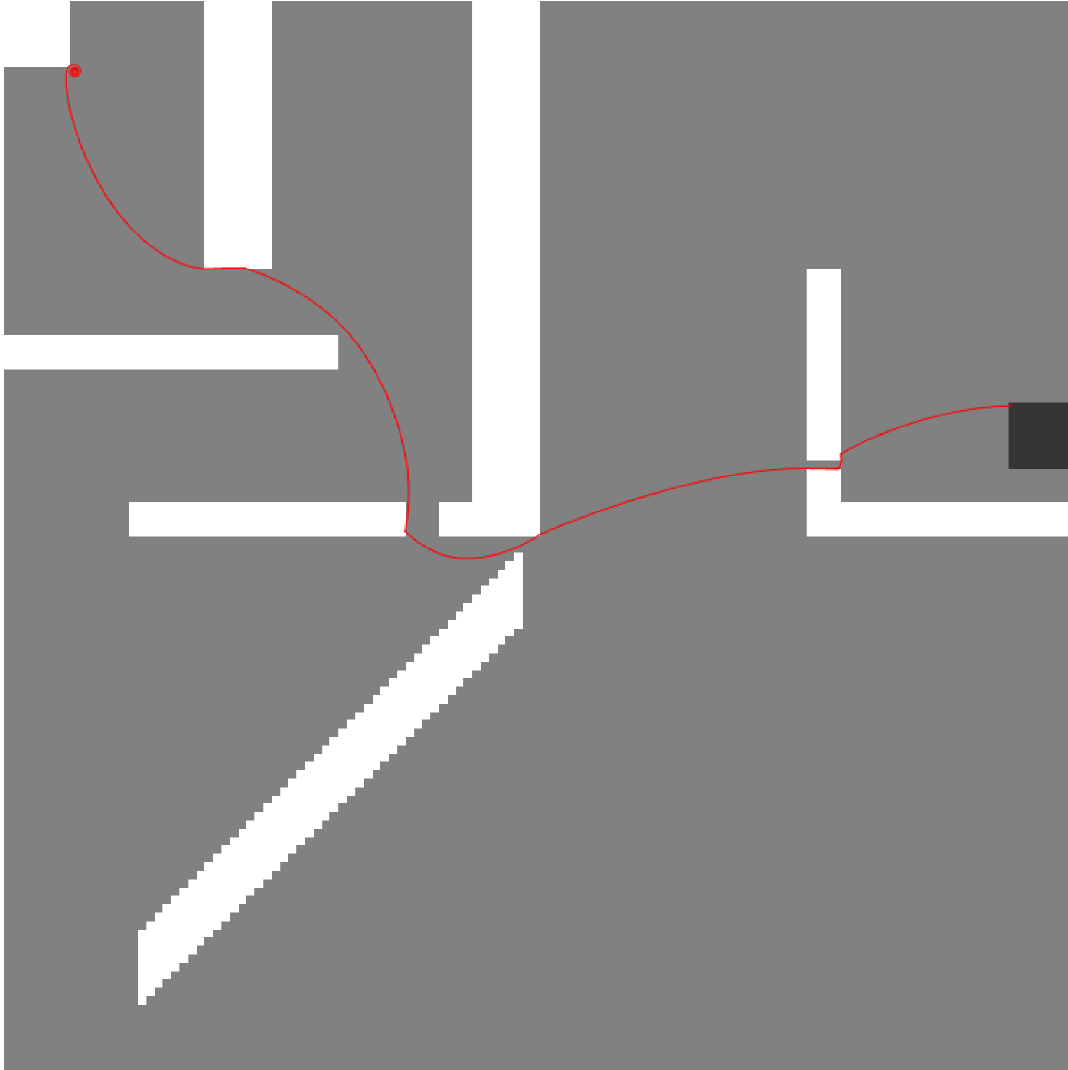

Figure 39: Example 4, Crashes in Tight Spaces. In this case one can see several crashes as the vehicle cannot make some turn radii (middle of the map), heads to a waypoint without regard for the next waypoint (upper left collision), and cycles about the goal region (upper left).

A good way to fix these crashes in Figure 39 in the future would be to utilize knowledge of the kinematics of the vehicle and not create nominal paths that require turns that are too tight. For example, we might consider limiting the angle formed by any two segments of the nominal path, so as not to create paths that are too removed from vehicle kinematics. Additionally, when considering which waypoint to travel to it would be beneficial to consider 2-point BVP-valid curves so that trajectories beyond straight line paths can be considered towards waypoints. 3-point BVP would

also then consider the next waypoint to travel to so that the vehicle does not drive itself towards a waypoint without regard for the next waypoint.

When a point in the map has no line of sight to any waypoint, the Waypoint-A* cannot generate a heading vector. This can be observed in the blank areas of Figure 39. There are two possible ways to address this problem. First by considering curved paths and not just line of sight paths, as discussed previously, would alleviate many of these blind regions. Another approach would be to relax the necessity of travel to waypoints. Rather it would be valid to travel to any point on any segment connecting two waypoints. Interpolation would be used to estimate the cost-to-go at these intermediate points, and an algebraic closed form solution would be used to find the best place to intersect the segment. Another interesting extension would be to use some RRT techniques at each waypoint to branch out possible paths from each waypoint. This would be used to describe possible sub-waypoints to increase visibility of each waypoint.

Fixing the cycling noted near the goal in Figure 40 is straightforward. That is the primary cause for setting up goal regions. Although the nominal path must be directed to a single node in the goal region, whenever the vehicle is in any part of the goal region, the forward simulation terminates. This minimizes cycles because the vehicle is not trying to converge to a single point in a sink.

Next it will be important to build in an explicit crash prevention element to the code. Although the smooth trajectory finding is not taking place in an entirely removed post-planning phase, there are currently no explicit safeguards against collisions. Fixing this limitation will come in two forms. Firstly, the creation of the

nominal path should take measures to avoid situations such as getting too close to obstacles that may cause problems for turning vehicles. The nominal path should use knowledge of the minimum turning radius to create this buffer from the obstacles. Secondly, in the actual heading creation, additional factors such as keeping safety margins from the obstacles should weigh in to the decision for best waypoint in addition to minimum cost.

Chapter 6: Conclusion

This thesis has presented the novel Waypoint-A* algorithm, some of its nice features, and its limitations, as well as suggested a way to remedy these limitations. A motivation for this work has been supplied in Chapter 1, and a background presentation on discrete planners has been covered in Chapter 2. The algorithm has been related to other relevant work in the field in Chapters 3 & 4, in order to provide context. Additionally, a breakdown of the elements of the algorithm and instructive examples have been featured in Chapter 5.

The key features of Waypoint-A* are its simplicity, and smoothness of path. Waypoint-A* runs efficiently by reducing the higher dimensionality of the vehicle dynamics to just (x,y) during the nominal path finding and heading finding, only incorporating higher order states explicitly in the forward simulator. Adaptive sampling also reduces the number of states that must be expanded in this search. Because the final path trajectories are created by forward simulating the vehicle's dynamics in a continuous space, resolution of the nominal path search does not affect the smoothness of the trajectory, a limitation in the any-angle path planners.

Similar to the any-angle planners, Waypoint-A* can define a heading on a continuum. As discussed this is a key feature in creating natural and smooth paths, as the space of headings is now rich enough to describe any combination of turns and maneuvers. This is contrasted with the approaches that utilize a finite, if not intelligently chosen, set of motion primitives. Although the works exhibited do create nice paths, for any finite set of motion primitives one can always construct a

pathological terrain map for which the finite set is not rich enough to find a path or find a paths that does not appear to swerve erratically.

Waypoint-A* does have some considerable limitations at this moment, however. These almost exclusively stem from the way that the algorithm determines the heading. It finds the best waypoint to travel to in a straight line without considering what happens next. The fact that it only searches straight-line paths, where admittedly the vehicle itself travels in curves, does not express the best heading the vehicle could actually take. This leads to visibility problems where in regions with no line of sight to waypoints the heading is undefined, even if a curved path could return the vehicle to the path. Further the vehicle cannot always comply with these straight-line directives as it is dynamically bound: this causes turn overshoot and possible crashes. There is also no look-ahead feature; the heading drives the vehicle to the best waypoint in a line of sight, but does not consider what it takes to continue following the waypoints once the first has been reached. This causes the overshoot problem discussed, as well as some of the crashes.

Solutions to these limitations have been proposed and will be implemented in the future. Regardless, Waypoint-A* showcases a nice way to think of the nominal path created by A*. By treating a path as a set of waypoints, many nice behaviors can be created to suit the needs of the vastly diverse applications of path planners.

Bibliography

[1] S. LaValle, *Planning Algorithms*. Cambridge Press, 2013.

[2] D. Ferguson and A. Stentz, "Using interpolation to improve path planning: The Field D* algorithm," *J. F. Robot.*, vol. 23, no. 2, pp. 79–101, Feb. 2006.

[3] S. J. J. Smith, D. Nau, and T. Throop, "Computer Bridge—A Big Win for AI Planning," *AI Magazine*, vol. 19, no. 2, pp. 93–106, 1998.

[4] D. Billings, L. Peña, J. Schaeffer, and D. Szafron, "Using Probabilistic Knowledge and Simulation to Play Poker," in *The Sixteenth National Conference On Artificial Intelligence*, 1999.

[5] "How Deep Blue Works," *IBM Research*. [Online]. Available: https://www.research.ibm.com/deepblue/meet/html/d.3.2.html. [Accessed: 07-Apr-2014].

[6] A. Gupta, R. Divekar, and M. Agrawal, "Autonomous parallel parking system for Ackerman steering four wheelers," in *2010 IEEE International Conference on Computational Intelligence and Computing Research*, 2010, pp. 1–6.

[7] C. Rebuzzi, "Top Five Self Parking Cars." [Online]. Available: http://motorburn.com/2012/06/top-five-self-parking-cars/. [Accessed: 07-Apr-2014].

[8] M. Likhachev, D. Ferguson, G. Gordon, A. Stentz, and S. Thrun, "Anytime Dynamic A *: An Anytime , Replanning Algorithm," in *International Conference on Automated Planning and Scheduling*, 2005, pp. 262–271.

[9] N. Ladeveze, J.-Y. Fourquet, and B. Puel, "Interactive path planning for haptic assistance in assembly tasks," *Comput. Graph.*, vol. 34, no. 1, pp. 17–25, Feb. 2010.

[10] B. B. Kirkpatrick, S. L. Thomas, N. M. Amato, B. Kirkpatrick, and W. P. St, "Modeling RNA Folding Landscapes with Probabilistic Roadmap Methods Parasol Lab Technical Report # TR03-004," 2013.

[11] "Roses Bunch of Flowers." [Online]. Available: http://ibmsmartercommerce.sourceforge.net/wp-content/uploads/2012/09/Roses_Bunch_Of_Flowers.jpeg. [Accessed: 07-Apr-2014].

[12] R. . Bellman, *Dynamic Programming*. Dover Publications, 2003.

[13] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," *IEEE Trans. Robot.*, vol. 21, no. 3, pp. 354–363, Jun. 2005.

[14] D. Delling, P. Sanders, D. Schultes, and D. Wagner, "Engineering Route Planning Algorithms," *Algorithmics of Large and Complex Networks*, vol. 2, pp. 117–139, 2009.

[15] W. Zeng and R. L. Church, "Finding shortest paths on real road networks: the case for A*," *Int. J. Geogr. Inf. Sci.*, vol. 23, no. 4, pp. 531–543, Apr. 2009.

[16] A. Patel, “Game Programming 4000 word version.” [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/4000.html. [Accessed: 07-Apr-2014].

[17] R. Dechter and J. Pearl, “Generalized best-first search strategies and the optimality af A*,” *J. ACM*, vol. 32, no. 3, pp. 505–536, Jul. 1985.

[18] D. K. Pai and L.-M. Reissell, “Multiresolution rough terrain motion planning,” in *Proceedings 1995 IEEE/RSJ International Conference on Intelligent Robots and Systems. Human Robot Interaction and Cooperative Robots*, 1995, vol. 2, pp. 39–44.

[19] D. K. Pai and L.-M. Reissell, “Multiresolution rough terrain motion planning,” *IEEE Trans. Robot. Autom.*, vol. 14, no. 1, pp. 19–33, 1998.

[20] R. Bohlin, “Path planning in practice; lazy evaluation on a multi-resolution grid,” in *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*, 2001, vol. 1, pp. 49–54.

[21] S. Kambhampati and L. Davis, “Multiresolution path planning for mobile robots,” *IEEE J. Robot. Autom.*, vol. 2, no. 3, pp. 135–145, 1986.

[22] S. Behnke, “Local Multiresolution Path Planning,” in *RoboCup 2003: Robot Soccer World Cup VII*, D. Polani, B. Browning, A. Bonarini, and K. Yoshida, Eds. Springer Berlin Heidelberg, 2004, pp. 332–343.

[23] D. Ferguson and A. Stentz, “Multi-resolution Field D *,” in *IAS*, 2006, pp. 65–74.

[24] K. Gochev, A. Safonova, and M. Likhachev, “Incremental Planning with Adaptive Dimensionality,” in *International Conference on Automated Planning and Scheduling*, 2013, pp. 82–90.

[25] M. Pivtoraiko and A. Kelly, “Differentially constrained motion replanning using state lattices with graduated fidelity,” in *2008 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2008, pp. 2611–2616.

[26] C. Liu, W. Liu, H. Zhou, and Z. Wei, “Path Planning for Mobile Robot Based on Improved Framed-Quadtree,” in *Intelligent Robotics and Applications*, 2008, pp. 844–852.

[27] S. M. Lavalle, M. S. Branicky, and S. R. Lindemann, “On the Relationship Between Classical Grid Search and Probabilistic Roadmaps,” in *The International Journal of Robotics Research*, 2004.

[28] L. E. Kavraki, “Kinodynamic Motion Planning by Interior-Exterior Cell Exploration,” in *Algorithmic Foundation of Robotics VIII*, 2009, pp. 449–464.

[29] A. De Luca, G. Oriolo, and C. Samson, “Feedback control of a nonholonomic car-like robot,” in *Robot Motion Planning and Control*, J. P. Laumond, Ed. Springer Berlin Heidelberg, 1998, pp. 171–253.

[30] G. A. Mills-tettey, A. Stentz, and M. B. Dias, “Continuous-Field Path Planning with Constrained Path-Dependent State Variables,” in *CRA 2008 Workshop on Path Planning on Costmaps*, 2008.

[31] A. Nash, K. Daniel, and S. Koenig, “Theta *: Any-Angle Path Planning on Grids,” *Proc. Natl. Conf. Artif. Intell.*, vol. 22, no. 2, pp. 533 – 579, 1999.

[32] Z. Liang, X. Ma, and X. Dai, “A novel path planning algorithm based on twofold interpolations,” in *2008 IEEE International Conference on Mechatronics and Automation*, 2008, pp. 718–723.

[33] A. Nash, S. Koenig, and M. Likhachev, “Incremental Phi *: Incremental Any-Angle Path Planning on Grids ∗,” in *Proc. of the 21st Int. Joint Conf. on Artifical Intelligence*, 2009, pp. 1824–1830.