A Greedy Search Algorithm for Maneuver-Based Motion Planning of Agile Vehicles

Charles B. Neas

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Aerospace and Ocean Engineering

Mazen Farhood, Chair
Christopher Hall
Craig A. Woolsey

November 30, 2010
Blacksburg, Virginia

Keywords: Motion Planning, A*, Motion Primitives, Heuristic Search

A Greedy Search Algorithm for Maneuver-Based Motion Planning of Agile Vehicles

Charles B. Neas

(ABSTRACT)

This thesis presents a greedy search algorithm for maneuver-based motion planning of agile vehicles. In maneuver-based motion planning, vehicle maneuvers are solved offline and saved in a library to be used during motion planning. From this library, a tree of possible vehicle states can be generated through the search space. A depth-first, library-based algorithm called AD-Lib is developed and used to quickly provide feasible trajectories along the tree. AD-Lib combines greedy search techniques with hill climbing and effective backtracking to guide the search process rapidly towards the goal. Using simulations of a four-thruster hovercraft, AD-Lib is compared to existing suboptimal search algorithms in both known and unknown environments with static obstacles. AD-Lib is shown to be faster than existing techniques, at the expense of increased path cost. The motion planning strategy of AD-Lib along with a switching controller is also tested in an environment with dynamic obstacles.

To my family

"Trust in the Lord with all your heart and lean not on your own understanding;

In all your ways acknowledge him, and He shall direct your paths."

Proverbs 3:5-6

Acknowledgments

First and foremost, I would like to thank my advisor, Dr. Mazen Farhood, for his support during my research and for giving direction to my efforts. I would also like to thank Dr. Chris Hall and Dr. Craig Woolsey for their time and effort in reviewing and providing feedback on my work as members of my committee.

To my research group, I owe many thanks for their assistance and encouragement. I would specifically like to thank Dave and Chris for pushing me to learn LaTeXand for helping to solve my frequent MatLab troubles. I also want to acknowledge my friends Lera, Ed, and Mike for keeping me sane through all these semesters. Roger, thank you for your guidance through the minefield of college and for serving as the font of knowledge.

Finally, I could not have accomplished this goal without the support and encouragement of my family. I hope I can continue to make you proud in my future endeavors. To my parents, thank you for pushing me when I needed it and for helping me survive this long. I thank my brother, Tom, for making everything fun.

Contents

1 Introduction **1**

2 Background **4**

2.1 Graph Search 4

2.1.1 Uninformed Search 4

2.1.2 Heuristic Search 9

2.1.3 Relaxations of Admissibility Condition 14

2.2 Local Search 15

2.2.1 Hill Climbing 15

2.2.2 Simulated Annealing 17

3 Maneuver-Based Graph Search **18**

3.1 AD-Lib 19

4 Four-Thruster Hovercraft Example **25**

4.1 Vehicle Dynamics and Maneuver Generation 25

4.2 Sensor Modeling Using Laser Range Finder 29

4.2.1 Heuristics from LRF Data 30

4.3 Known Static Obstacles 32

4.3.1 Test Results 34

4.4 Unknown Static Obstacles 37

4.4.1 Test Results 39

4.5 Dynamic Environments 40

5 Conclusion **45**

List of Figures

2.1 Breadth-first Search Ordering 6

2.2 Depth-first Search Ordering 7

3.1 A contour depicting the AD-Lib expansion with respect to the cost-to-go, $h(n)$. 20

3.2 A simple example of AD-Lib expansion with zero edge costs 23

4.1 Four-thruster hovercraft model and parameters 27

4.2 Transition maneuver from $\theta_i = 0$ rad to $\theta_f = \frac{\pi}{2}$ rad 28

4.3 Maneuver library for the four-thruster hovercraft at 1 m/s 29

4.4 Two heuristics used for testing: the line-of-sight heuristic, H1, and the Euclidean norm, H2 32

4.5 Environment with Known Static Obstacles 33

4.6 Maneuver libraries for the 0.5 m/s library (blue) and the 1 m/s library (red) 37

4.7 Environment with Sensor Sweep 41

4.8 Snapshots of dynamic obstacle environment 43

4.9 Dynamic Obstacle Simulation with errors shown in dashed red 44

List of Tables

4.1 Hovercraft parameters 26
4.2 Known static obstacles 35
4.3 Known static obstacles, two libraries, 5 obstacles 38
4.4 Known static obstacles, two libraries, 10 obstacles 39
4.5 Known static obstacles, two libraries, 15 obstacles 40
4.6 Unknown static obstacles 42

List of Algorithms

1 Breadth-first Search 6
2 Depth-first Search 7
3 Depth-first Iterative Deepening 8
4 Uniform Cost Search 11
5 Greedy Best-First Search 12
6 A* 13
7 Steepest Ascent Hill Climbing 16
8 AD-Lib 22

Chapter 1

Introduction

In many interesting applications, autonomous vehicular systems are expected to navigate increasingly complex environments, which may include uncertain dynamic obstacles and significant disturbances. The vehicle's *motion planner* constructs a motion plan as a feasible trajectory through the environment, altering the trajectory as knowledge of the environment changes. The *controller* handles exogenous disturbances and other uncertainties by modifying the reference inputs of the trajectory to compensate for these errors.

The system and expected operating environment should be accounted for when designing motion plans for an agile vehicle. Agile vehicles may operate in environments where the motion plan must be developed quickly before the safety of the vehicle is jeopardized. In such cases, solving the nonlinear equations of motion online increases the computation time required for the motion planning task. Maneuver-based motion planning produces simple, feasible maneuvers offline and stores them for planning use in a library.

Maneuver-based motion planners design feasible trajectories from the stored maneuvers by joining them at the trim states located at the start and end of the maneuvers. These

feasible maneuvers are defined as state and control trajectories which encompass two classes: trim trajectories and transition maneuvers. The trim trajectories are composed solely of steady-state motions, whereas transition maneuvers are maneuvers that begin and end at steady-state conditions. Trim trajectories have variable coasting times, and can be scaled as necessary. Existing methods for maneuver-based motion planning use online nonlinear optimization to find the optimal path from a given library by selecting the combination of transition maneuvers and coasting times of the trim trajectories which minimizes a given cost function [1]. In some scenarios, online optimization may require more computational resources and time than can be allotted, specifically in dynamic environments where there are usually significant changes in the search space.

To alleviate the need for online optimization, the problem can be formulated as a graph search problem by setting the coasting times for the trim trajectories in the library. Assuming zero-order hold sampling of the inputs, the trim trajectories will be applied over a pre-specified number of time steps. A trim trajectory with a set length has the same properties as the transition maneuvers in the library; therefore, no distinction will be made between the two classes, but they shall all be known as maneuvers. These maneuvers can be represent the arcs of a graph search problem whose corresponding endpoints serve as the nodes. A *graph* is a set of nodes connected by arcs [2], and algorithms which find a sequence of connected nodes through a graph are covered extensively in the field of graph search. A graph search returns a series of maneuvers connecting the starting state to the final state.

In this thesis, a new graph search algorithm for maneuver-based motion planning is described which uses greedy best-first search techniques combined with effective backtracking to find a feasible path through a space. In Chapter 2, existing graph search techniques and local search methods are examined to provide a basis for the development a new algorithm. The new algorithm, called algorithm, depth-first, and library-based (AD-Lib), is described

and several conclusions are drawn regarding its expansion properties in Chapter 3. In Chapter 4, AD-Lib is compared to dynamically weighted A*, a suboptimal search algorithm commonly used for suboptimal graph search, using libraries developed for a four-thruster hovercraft operating in a known environment. The performance of AD-Lib is examined in unknown environments using a laser range finder model to analyze the speed and cost of replanning. Finally, navigation in a dynamic obstacle field is demonstrated using AD-Lib along with hybrid control techniques. Chapter 5 summarizes the results of the thesis and suggests areas for future work.

Chapter 2

Background

2.1 Graph Search

Graph search is the task of finding a path through a graph. A graph is a series of nodes connected by arcs [3]. Graphs arise in a multitude of subject areas because the seach structure of nodes and arcs lends itself to any set of connected states [2]. Trees are simple graphs with one directed connection between nodes, where each node has one unique parent. More complex graphs may have multiple parents per node and undirected arcs. In this thesis, the nodes of the graph represent vehicle states, $n_i, \forall\ i = 1, 2, 3, ...$, corresponding to the start and end points of vehicle maneuvers. The arcs of the graph are sets of inputs to the system, defining maneuvers which the system can follow.

2.1.1 Uninformed Search

Initial efforts to solve graph search problems centered on a class of algorithms known as uninformed searches [4]. These algorithms search the entire graph to find a path, should

one exist. Because all possibilities are exhaustively searched until a goal is found, these algorithms are *complete*; meaning they return a path should one exist, or will return an error otherwise. However, the price of this completeness is an increase in computation needed to exhaustively search the space.

Uninformed search methods organize the graph into several sets: the open queue, $\mathcal{O}$, the closed set, $\mathcal{C}$, and the goal set, $\mathcal{G}$. The goal set contains all nodes at which the search can terminate. If the graph is not known *a priori*, the goal set can be defined by a set of termination conditions. The closed set contains all nodes which have been previously explored by the search. The difference between uninformed search techniques lies in the ordering of the open queue.

Breadth-First Search

The first type of uninformed search is breadth-first search. The nodes are expanded across the breadth of the graph before searching the nodes at the next depth. Initially, the start node is expanded, adding its successors to the open queue, $\mathcal{O}$. The start node is then added to the closed set, $\mathcal{C}$. The successors to the start node are then expanded in turn, adding their successors to $\mathcal{O}$ until a goal node $\mathcal{G}$ is found [3]. An example of the expansion order of a simple tree is shown in Figure 2.1.

Breadth-first search is a very simple algorithm, but it contains many of the main features of more complex algorithms, such as the process of node expansion. Node expansion, as briefly described above, is a process where search continues by adding subsequent nodes to a search list, in this case $\mathcal{O}$. In breadth-first search, each expansion adds every subsequent node, thus fully closing the node. A node is closed when all of its successors have been added to the search queues. The concept of maintaining a closed list also arises in later algorithms.

Algorithm 1 Breadth-first Search

Input: Start node n
Output: $n \in \mathcal{G}$

```
while n ∉ G do
    Add successors n' to the end of O
    n → C
    if O ∈ ∅ then
        No path exists
    else
        n ← O(1)
```

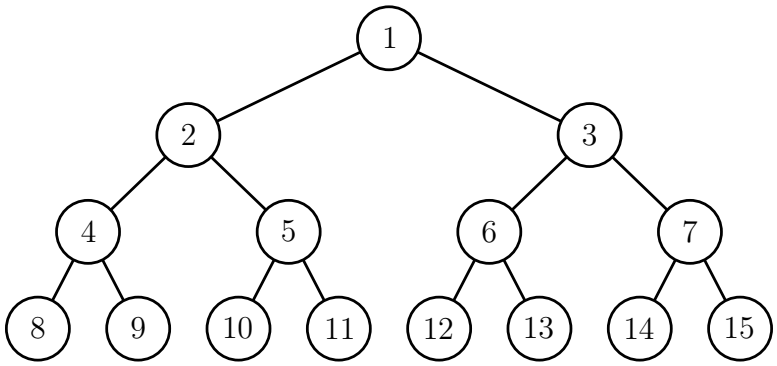

Figure 2.1: Breadth-first Search Ordering

The number of nodes expanded by the search depends on the average branching factor of the graph, b, which is the average number of successors added during each expansion. The worst-case run time for breadth-first search is $O(b^d)$, where d is the minimum depth of a node in $\mathcal{G}$ [2]. This worst-case scenario results in a brute-force search, wherein every node in the tree is added and searched before a goal state is found.

Depth-First Search

Depth-first search is identical to breadth-first search, except in how the nodes are ordered in $\mathcal{O}$. In breadth-first search, new nodes are added to the bottom of the list for expansion, causing the algorithm to expand nodes closer to the start before expanding nodes deeper in the tree. In depth-first search, newly added nodes are appended to the front of the search

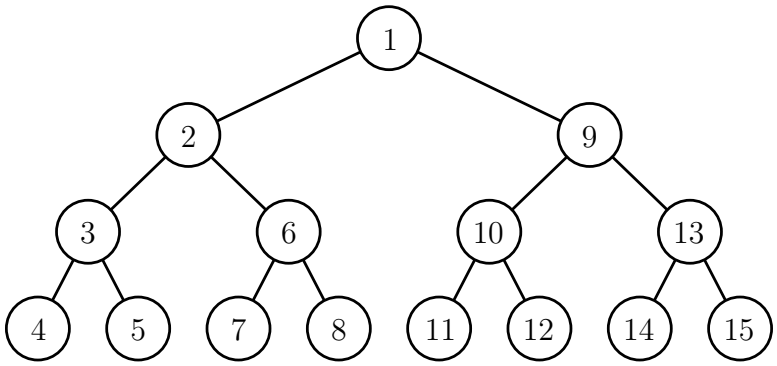

Figure 2.2: Depth-first Search Ordering

order, causing the search to expand subsequent depths before returning to search across the breadth. An example of depth-first search ordering is given in Figure 2.2.

Algorithm 2 Depth-first Search

Input: Start node n
Output: $n \in \mathcal{G}$

```
while n ∉ G do
    Add successors n' to the beginning of O
    n → C
    if O ∈ ∅ then
        No path exists
    else
        n ← O(1)
```

Depth-first search may experience reductions in the required number of expansions if a goal state is known to be deep in the search space. However, depth-first search performs as poorly in the worst-case as breadth-first search, with the worst-case time complexity being $O(b^d)$ [2, 3]. For some domains where a large set of goal states exist at a set depth, depth-first search can achieve faster search times than breadth-first search. In this case, depth-first search will "dive" towards the appropriate goal depth and have a higher chance of finding a goal node. For vehicle search applications, the depth to solution may not be known *a priori*, making either breadth-first or depth-first search a computationally intensive undertaking.

Depth-First Iterative Deepening

Since the goal depth may not be known beforehand, depth-first iterative deepening (DFID) limits the depth of the search initially, progressively pushing the limit forward until a goal state is found. The search begins as a depth-first search, stopping once it reaches either a goal state or a prespecified depth. The search continues until all nodes to a certain depth have been expanded or a goal has been found. If a goal state has not been reached, then the search continues by extending the depth limit by a user-specified amount, ϵ, then continuing the search [5].

Algorithm 3 Depth-first Iterative Deepening

Input: Start node n,Depth guess D_g,Increment ϵ
Output: $n \in \mathcal{G}$

```
while n ∉ G do
    while D_n ≤ D_g do
        Add successors n' to the beginning of O
        n → C
        if O ∈ ∅ then
            No path exists
        else
            n ← O(1)
    D_g = D_g + ε
```

DFID attempts to limit the inefficiencies of depth-first search by limiting the maximum depth during each iteration before searching along the breadth of the graph. If a set of goal states are known to be at a certain depth, the search effort can be reduced by not searching any nodes beyond that depth. DFID is useful when a set of goal states exist near a known depth. In this scenario, DFID will rapidly search towards the estimated goal depth, and then search along the breadth until the entire area has been searched. At this point, the estimated depth can be increased slightly and the next section of the graph will be searched.

DFID in the worst case can expand all nodes up to the goal depth, resulting in the same

inefficiencies as the other uninformed search methods, $O(b^d)$ [5]. While simple, uninformed searches may have to expand many nodes before a goal state is found. If additional information is incorporated into the search process, more ideal nodes can be expanded first as opposed to the arbitrary ordering of $\mathcal{O}$ in uninformed searches.

2.1.2 Heuristic Search

The wasted search effort inherent in uninformed search algorithms is a direct result of the absence of information during the planning process. Most algorithms employ a combination of the *cost* of each node and another value, known as the *heuristic* [3]. Heuristic search guides its search effort by employing information about each node to better inform the search process. A heuristic can be any value providing information to an algorithm about the relative value of each node in the search process. In the sense of vehicle search, a common heuristic is the cost-to-goal, or an estimate of the distance between a node and the goal states. By using the heuristic and cost in conjunction to motivate the expansion, heuristic searches can greatly reduce the amount of necessary computation.

Heuristic search maintains the open queue $\mathcal{O}$ and the closed queue $\mathcal{C}$, similar to uninformed search, but applies a different ordering to the open queue [2]. Instead of arbitrarily ordering the open queue as in uninformed searches, heuristic searches use a combination of the heuristic and the cost to order the nodes based on their relative importance to the search.

Dijkstra's Algorithm and Uniform Cost Search

One potential "best" solution to the search is the minimum-cost path from a start node, s, to a node in $\mathcal{G}$. In order to minimize cost, the cost must be used as a heuristic to guide the

search. The minimum cost of moving from s to node n is defined as $g(n)$. In a tree search, this value is unique to each node n, since only one path exists to any node n. In graph searches where multiple paths exist between nodes, the value of $g(n)$ changes to represent the minimum cost path to n from s [3, 6].

Dijkstra's algorithm uses $g(n)$ to guide the search effort in order to find the minimum-cost path to all nodes from a start node. From the start node, additional nodes are added in a breadth-first fashion to $\mathcal{O}$, which is then sorted according to decreasing values of $g(n)$ [7]. When a node is expanded, the costs of its successors are examined for consistency. Consistency demands that the cost $g(n)$ for the successors is minimal, i.e. $g(n) = \min[g_{old}(n), g(n_{pred}) + c(n_{pred}, n)]$. If a smaller $g(n)$ is found, the successors of n become inconsistent and must each be checked for consistency. Once all nodes are deemed consistent, the expanded node is placed in $\mathcal{C}$ and $\mathcal{O}$ is reordered [3]. Dijkstra's algorithm is prohibitively expensive in large graphs, since every node must be searched to ensure the minimum-cost path is found.

Dijkstra's algorithm does not solve the graph search problem, rather it solves the shortest path problem. In this formulation, the shortest path from a start node to all points is found. If Dijkstra's algorithm is set to terminate when a goal is expanded, the resulting algorithm is known as uniform cost search. The algorithm is given in Algorithm 4. Uniform cost search greatly reduces the number of nodes expanded from Dijkstra's algorithm.

The expansion of uniform cost search creates a cost frontier, where all paths below a certain maximum cost are expanded before moving ever closer to a goal state. Uniform cost search returns the minimum-cost path, unlike DFID which only returns the minimum-depth path. Also, the entire breadth of the graph may not be searched, depending on the costs associated with the graph. It should be noted for continuity that if the arc costs $c(n_i, n_j)$

Algorithm 4 Uniform Cost Search

Input: Start node n
Output: $n \in \mathcal{G}$

```
1: while n ∉ G do
2:     Add successors n' to O
3:     Check for consistency of n'
4:     if Any n' are inconsistent then
5:         Update inconsistent nodes
6:         Place inconsistent n' in O
7:     n → C
8:     if O ∈ ∅ then
9:         No path exists
10:    else
11:        Order O by g(n)
12:        n ← O(1)
```

are homogeneous, uniform cost search becomes breadth-first search.

Greedy Best-First Search

Other heuristics can be used to guide the expansion process. Heuristics other than the cost-to-go are most commonly estimates of the cost-to-goal, $h(n)$ [3]. In path planning problems, the cost estimate may include distance from the nearest goal state, expected fuel consumption, travel time or combinations thereof. Applying the heuristic naively to the search expands nodes with lower cost-to-goal estimates first. This search strategy is known as greedy best-first search [2]. More generally, the class of problems which expand the next-best option at every step are known as "greedy" algorithms [8].

Greedy best-first search is the informed analogue to depth-first search. The expansion does not account for travel costs which may result in path costs much higher than the minimum. Also, the expansion is *incomplete* since it may become caught in infinite loops, especially on graphs [2]. The worst-case time complexity of greedy best-first search is $O(b^m)$,

Algorithm 5 Greedy Best-First Search

Input: Start node n
Output: $n \in \mathcal{G}$

```
while n ∉ G do
    Add successors n' to O
    n → C
    if O ∈ ∅ then
        No path exists
    else
        Order O by h(n)
        n ← O(1)
```

where m is the maximum search depth. The average time complexity is highly dependent on the quality of $h(n)$ at estimating the actual cost-to-goal, $h^*(n)$. A good estimate will drive the search correctly towards the goal, whereas a less informed search resembles a depth-first search. There is a trade-off associated with improving the heuristic, as more complex heuristics require more time to compute and may offset reductions in graph expansion. This determination, like all heuristic choices, is problem dependent.

A*

Combining the search measures of uniform cost search and greedy best-first search, A* provides the guarantees necessary for a complete and optimal search finding the minimum-cost path. A* uses the total path cost, $f(n)$, to guide the expansion process. The total path cost is equal to the cost-to-go plus the cost-to-goal, $f(n) = g(n) + h(n)$. This total cost represents the estimated cost of a path between the start node and the goal which goes through n [3, 6].

The expansion of A* is ordered by the estimated path cost of the nodes in $\mathcal{O}$. Expansion using $f(n)$ combines the advantages of uniform cost search and greedy best-first search. It has been shown that A* returns the minimum-cost path should one exist, and expands the

fewest nodes necessary to find the minimum-cost path for any given heuristic [3, 6]. These guarantees only hold if the heuristic is admissible and consistent. A heuristic is admissible if it never overestimates the actual cost-to-goal, such that $h(n) \leq h^*(n)$ [3]. This added requirement ensures the expansion does not miss paths by assuming their cost to be greater than the actual value. The consistency requirement provides the same benefit as in Dijkstra's algorithm, ensuring that $g(n)$ remains minimal as alternative paths are found to the node n [7].

Algorithm 6 A*

Input: Start node n
Output: $n \in \mathcal{G}$

```
while n ∉ G do
    Add successors n′ to O
    Check for consistency of n′
    if Any n′ are inconsistent then
        Update inconsistent nodes
        Place inconsistent n′ in O
    n → C
    if O ∈ ∅ then
        No path exists
    else
        Order O by f(n) = g(n) + h(n)
        n ← O(1)
```

While A* is shown to minimize expansions in the process of finding the minimum-cost path, the number of expansions required can still be greater than the computational allowances for the path planning task. The time complexity of A* is dependent upon the quality of the heuristic. The quality of the heuristic is measured as the absolute error, $\Delta = h^*(n) - h(n)$. The time complexity of A* for simple problems is $O(b^{\Delta})$ [2]. If a heuristic $h_2(n) \geq h_1(n)$ for all n, then h_2 *dominates* h_1 [2]. A more dominant heuristic will reduce the search effort in relation to the absolute error decrease $h^*(n) - (h_2(n) - h_1(n))$.

2.1.3 Relaxations of Admissibility Condition

Weighted A*

The ability of A* to find the minimum-cost path has made it the standard to which most graph search programs are compared. A* produces excessive run times for complex graphs or trees, due to numerous similar paths which must all be checked for optimality [9]. Extensions of A* focus on achieving a reduction in the time necessary to find an acceptable path while maintaining bounds on optimality. The estimation of path cost by $f(n)$ in A* results in choosing nodes for expansion, ensuring no node on the minimum-cost solution path is missed. To emphasize speed, the weighted A* algorithm, denoted WA*, returns suboptimal solutions by relaxing the admissibility condition. The path cost formula becomes $f(n) = g(n) + (1 + w)h(n)$, where w limits the error incurred by relaxing the admissibility condition [10].

Modifications to WA* exist, the first of which is dynamically weighted A* (DWA*), emphasizing $h(n)$ close to the start and $g(n)$ as the depth increases. The path cost is calculated by $f(n) = g(n) + h(n) + w(1 - \frac{d(n)}{N})h(n)$, where $d(n)$ is the depth of the node n in the tree and N is the estimated depth of the goal region. This dynamic weighing initially favors promising routes first like WA* by increasing the contribution of the heuristic term. As the search approaches a goal state, the cost is examined more scrupulously, resulting in a path with a bounded cost [11].

The relaxed algorithm A^*_ϵ uses multiple heuristics to drive the search. The first of these heuristics is identical to that of A*: an estimate of the cost-to-goal. The second heuristic $h_F(n)$ estimates the remaining computational effort to complete a path from the node n. A^*_ϵ uses the open queue $\mathcal{O}$ to identify unexpanded nodes and the focal queue $\mathcal{F}$ which relates paths with similar costs $f(n)$. If a node has a cost $f(n)$ within a specified constant ϵ of the smallest current value, it is placed in $\mathcal{F}$. The expansion continues by selecting the node in

$\mathcal{F}$ with the smallest value of $h_F(n)$ [12].

Another means of relaxing the admissibility condition is to allow the estimate of cost-to-goal to be greater than the true value, that is $h(n) \geq h^*(n)$. Harris' "bandwidth" algorithm allows for $h(n) \geq h^*(n) + w$, which violates the admissibility condition [9]. This relaxation limits the additional incurred cost by w, such that the resulting goal is w-optimal. If $w = 0$, the algorithm becomes A*. While there are no guarantees that the runtime will be significantly smaller as a result of the relaxation, the "bandwidth" condition increases the number of solution paths from the minimum-cost path of A*.

2.2 Local Search

2.2.1 Hill Climbing

Unlike graph search, local search attempts to refine the current solution without regard for what has occurred previously. Local search methods originated to solve optimization problems where a global maximum (or minimum) was to be found. A basic example of local search is the Newton-Raphson method for finding roots of an equation [2]. Another popular example is known as hill climbing. Hill climbing, in its simplest form, approaches a global maximum by choosing the first available successor with an increase in cost given by a cost function, $J(n)$. Hill climbing can find a global minimum by selecting the node with the minimum value of $J(n)$ and ceasing when the cost increases. For the remainder of the discussion, maximization will be treated. A simple extension of hill climbing is steepest ascent hill climbing, which expands the successor with the highest cost. Steepest-ascent hill climbing selects the largest gradient towards the extrema, which gives the algorithm its name. As evidenced by Algorithm 7, the algorithm is very easy to implement.

Algorithm 7 Steepest Ascent Hill Climbing

Input: Start node n
Output: Minimum, $J(n)$ and node, n
1: **repeat**
2: Make n' with the $\max(J(n))$ the current node
3: **until** $h(n') < h(n)$

Such greedy behavior is not without its drawbacks. Since hill climbing does not retain any parent information, a misstep during the expansion can increase the number of iterations of the algorithm. Also, certain features of the cost function can "trap" the expansion. These features are local maxima, ridges and plateaux [2]. If the cost function is not convex, the expansion can terminate in local maxima before reaching the desired global maximum. Ridges are several local maxima located near one another whereas a plateau is a region of the cost function where there is little change in the cost from one state to the next. The landscape of these features is known as the state-space landscape, where "elevation" corresponds to a value of $J(n)$ [2].

The ideas from hill climbing are closely related to those of greedy best-first search discussed in Section 2.1.2. In both algorithms, the next-best successor to be expanded is determined by using a cost function. The difference between the algorithms is that greedy best-first search maintains the tree, whereas hill climbing only retains the current node. Also, the termination conditions are different between the algorithms. In greedy best-first search, the termination is dependent upon the search reaching a defined goal region, whereas hill climbing only seeks to find an extrema, which may or may not be global. These differences, however, are more of application than of substance, depending upon how the algorithms are applied.

For problems where the cost function is not convex and these trapping features exist, methods of extricating the search from the local extrema are necessary. Enforced hill climbing

combines local search with breadth-first search to escape local maxima. If a local maxima is found and known to not be the global maximum, a breadth-first search proceeds from the maxima until an increase in cost is found. The local search then proceeds from this new "best" point [13]. Breadth-first search is computationally intensive, and consequently, enforced hill climbing can result in increased computation to find the global maximum.

2.2.2 Simulated Annealing

A more advanced hill climbing algorithm is known as simulated annealing, named for the process of metal cooling in materials science [2]. In simulated annealing, the search space is initially assigned a high "temperature," where temperature is the current value of the cost function. The search then compares potential successors. A successor which improves the cost is accepted; if all nodes do not improve the cost, they are accepted with probability $e^{(J(n')-J(n))/T}$, which is a measure of the descent gradient [14]. The space is then "cooled" according to a given schedule. This process continues until the temperature is equal to zero. Simulated annealing is guaranteed to achieve the global minimum given a sufficiently slow cooling schedule [15].

While the probabilistic methods of simulated annealing are not as closely related to graph search, adapting the cost function has similarities to dynamically weighted A*. As the space is cooled, simulated annealing accepts increases in the cost function with a diminishing probability. Dynamically weighted A* instead decreases the gradient in the cost function by decreasing the weight on $h(n)$, keeping the expansion in an artificially generated valley of the cost function. While the implementation of these algorithms differs, the concept of modifying the cost function to restrict expansion is common to both algorithms.

Chapter 3

Maneuver-Based Graph Search

In uncertain dynamic environments, the motion planning task must be completed quickly to respond to potential changes in the environment which may jeopardize the safety of the vehicle. Motion planning for agile vehicles is accelerated by solving the equations of motion offline, then compiling the maneuvers into a library for use online. Such an approach is known as maneuver-based motion planning. Unlike existing methods for maneuver-based motion planning which rely on nonlinear programming, the problem can be formulated as a graph search by setting the coasting times of the trim trajectories. By using simple state transformations to connect the maneuvers in the library, a graph can be created to search through an environment. A consequence of fixing the coasting times of the trim trajectories is a decreased reachability set for the vehicle, which may require that the goal state be relaxed to make the problem feasible. The goal state is relaxed by allowing the graph search to terminate if a state within a fixed Euclidean distance around the goal is found.

With a relaxed graph search problem formulated, a fast algorithm is needed to find a feasible path from the start node to the goal region. Greedy search techniques offer the best solution for this type of problem, since greedy search seeks a feasible path, unlike A*,

which finds the minimum-cost path [16]. As discussed in Section 2.1.2, greedy best-first search relaxes most of the constructs of heuristic search [2]. Using the framework of greedy search, a new greedy algorithm can be constructed to suit the needs of agile vehicle motion planning. The new algorithm, known as Algorithm, Depth-first and Library-based, or AD-Lib, constructs a feasible path while trying to minimize wasted search effort.

3.1 AD-Lib

The AD-Lib algorithm has three main features: library-based greedy search, local hill climbing, and effective backtracking. Unlike A*-based algorithms, which add all possible successors to $\mathcal{O}$ and select from this queue for expansion, AD-Lib only adds the next-best node to the tree and then examines that node, similar to a local search. The successor with the minimum-cost path length, $f(n)$, is expanded from each node. Each potential successor, n', of a given node, n, has the total path cost $f(n') = g(n) + c(n, n') + h(n')$, where $c(n, n')$ is the cost-to-move between n and n'. To select the successor with a minimum $f(n)$, the common $g(n)$ term may be removed and the value $h_c(n') = c(n, n') + h(n')$ can be used to select the next node. Using h_c to guide the expansion alleviates the need to maintain a record of $g(n)$ for each node.

AD-Lib incorporates backtracking to avoid over-expansion due to greedy search. As a path is expanded, its nodes create a heuristic contour represented by the cost-to-go values, $h(n)$, of each of its nodes. An example of a heuristic contour is shown in Figure 3.1. The local maxima and minima of the heuristic contour represent nodes where a significant change in the expansion has occurred. For example, local heuristic minima may represent a marked change in estimated cost-to-goal arising from a nearby obstacle. Nodes representing the local maxima and minima of the heuristic contour are designated as "watch nodes", n_{peak}

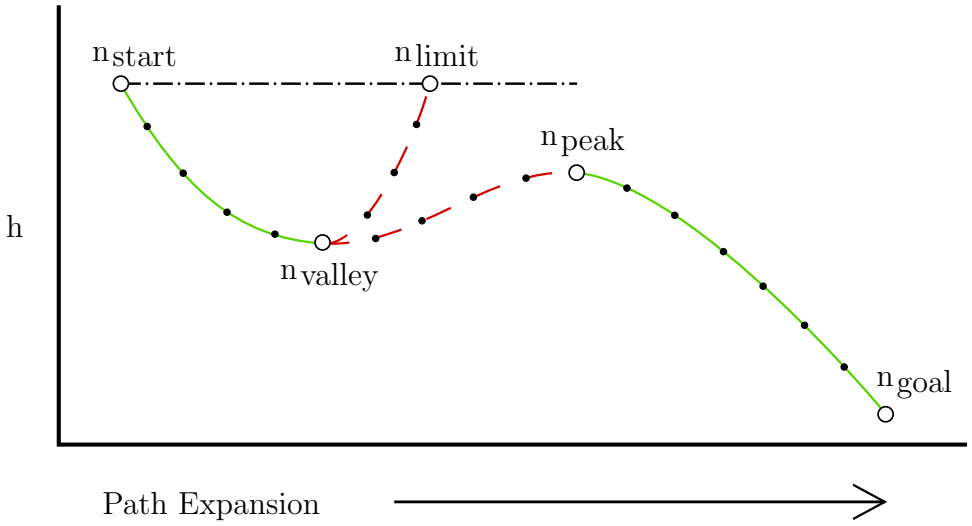

Figure 3.1: A contour depicting the AD-Lib expansion with respect to the cost-to-go, $h(n)$.

and n_{valley}, respectively. If the expansion finds itself in a local minimum, hill climbing is used to escape the minimum and find another descending gradient. Unlike hill climbing, the local search used in AD-Lib is wedded to a greedy search framework which maintains records of the parent nodes and corresponding maneuvers.

The falling and climbing phases indicate the two methods of expansion for AD-Lib. The falling phase indicates the h-value is decreasing between successive nodes; the climbing phase indicates the h-value is increasing. In the falling phase, the greedy search selects the node with the lowest f-value from all potential successors. The heuristic value of this successor, $h(n')$, is compared to the heuristic value of the current node, $h(n)$. If the value decreases, then expansion continues. If the value increases, then the expansion has entered a climbing phase. To avoid over-expansion, a limit is placed on the h-value of the successor n' equal to the heuristic value of the previous peak, $h(w_peak)$. If $h(n') \leq h(w_{peak})$, then n' is retained in the open queue and the search returns to the previous watch node to find a more promising path.

In the climbing phase, the expansion also selects the successor greedily by adding the

successor with the lowest f-value. If $h(n') < h(n)$, then the search has entered another falling phase, and n is saved as a watch node. Otherwise, the search enforces its climb until the peak limit is reached, prompting a return to the previous watch node and placing the peak node in $\mathcal{O}$. These two alternatives are illustrated in Figure 3.1, where from a valley either another peak or the limiting value is reached. By alternating between the falling and climbing phases, the search is forced to the goal by successively lowering the peak limiting value. In most cases, the path contour consists of a single falling phase connecting the start and goal states.

If a search requires, the algorithm may return to a watch node several times during expansion. Should a node become fully expanded, it cannot aid in backtracking. In this event, the watch node is closed. When a node is closed, any pointers to the closed node are updated to point to the previous watch node. A special case of closing watch nodes arises when the start node becomes fully expanded. If the start node becomes closed, there is no previous watch node for it to reference. The successors of the start node then become "start" nodes themselves, and watch nodes are updated to reference to those nodes. Conceptually, the start node is placed in the common "root system" of the newly separated trees. The common root structure makes the algorithm complete, as proved next.

Theorem 1. AD-Lib will return a path if one exists, and if one does not exist, AD-Lib will return failure.

Proof. As AD-Lib expands, all paths will either end at the goal state or achieve their respective peak limits. If a goal is not found during initial expansion, AD-Lib will return to search every path through every watch node. As the start node and its successors fall into the roots of the search tree, the remaining nodes will be in $\mathcal{O}$, located at their respective peak limits. In the worst case, all nodes on these limits will be expanded sequentially based upon their

Algorithm 8 AD-Lib

```
1: while n_current ∉ G do
2:     Expand current node
3:     if n has successors n' then
4:         Add successor state n' with lowest value of h_c(n')
5:     else if n has no successors then
6:         if n is start node then
7:             Make all successors of n start nodes
8:             Close n
9:             Current node is successor with minimum h(n)
10:        else
11:            Close n
12:            n ← watch(n)
13:            Return to while loop
14:    if State = Falling then
15:        if h(n') < h(n) then
16:            watch(n') = watch(n)
17:            n ← n'
18:        else if h(n') > h(n) then
19:            watch(n') = watch(n)
20:            n ← n'
21:            State = Climbing
22:        else if h(n') > h(peak(n)) then
23:            watch(n') = n
24:            n' ∈ O
25:            n ← watch(n')
26:    else if State = Climbing then
27:        if h(n') > h(n) then
28:            watch(n') = watch(n)
29:            n ← n'
30:        else if h(n') < h(n) then
31:            watch(n') = watch(n)
32:            n ← n'
33:            State = Falling
34:        else if h(n') > h(peak(n)) then
35:            watch(n') = n
36:            n' ∈ O
37:            n ← watch(n')
```

heuristic value $h(n)$. Eventually, the entire tree will be expanded until no nodes remain or a goal state is found. Once no nodes remain in $\mathcal{O}$, the search is terminated, returning

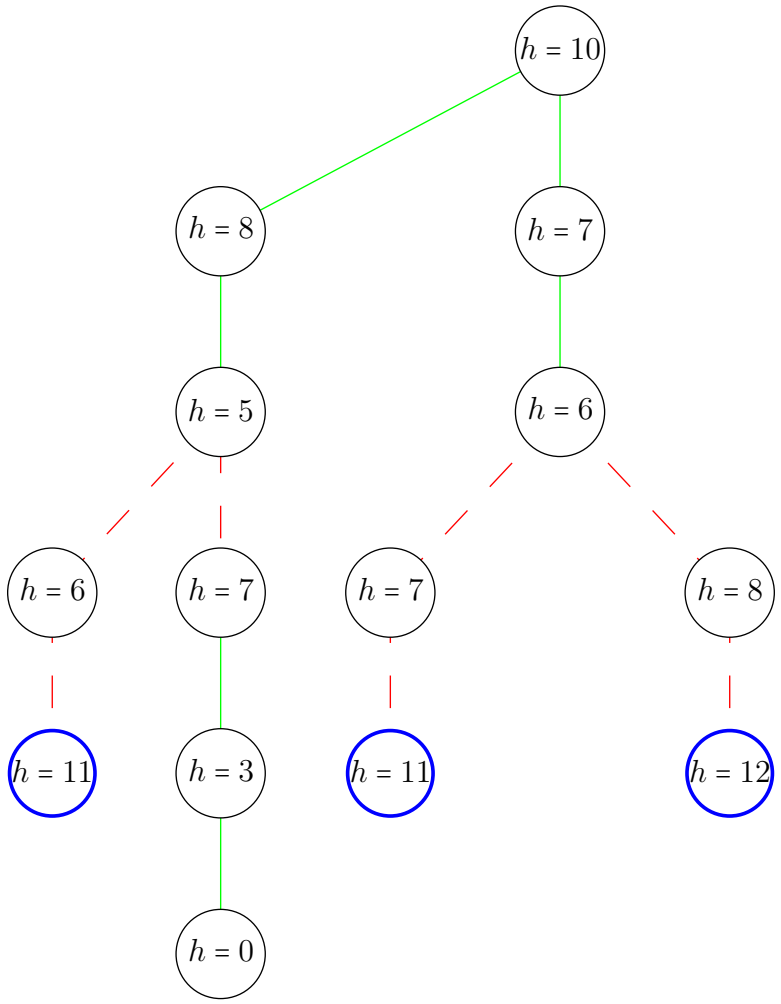

Figure 3.2: A simple example of AD-Lib expansion with zero edge costs

failure. □

As stated in the proof, the open queue is used to store possible paths until all the watch nodes have been closed. The open nodes represent the limit of the heuristic expansion, and are expanded greedily until a goal node is found. In practice, a time limit would cease expansion before this case was fully realized, unless no path exists beyond a few steps into the search.

An example of AD-Lib expansion is shown in Figure 3.2. In this example, the edge

costs are equal to zero to better illuminate the backtracking properties. Initially, the search examines the nodes on the right side of the graph until a local minimum is found where $h(n) = 6$. Then, nodes with increasing heuristics are added until $h(n) = 11 > 10$; the current limiting value for the heuristic. The search then returns to the $h(n) = 6$, where it continues until $h(n) = 12 > 10$. The terminal nodes on these branches are placed in $\mathcal{O}$, shown by their blue border. Since the valley node where $h(n) = 6$ is fully expanded, the search returns to the start node, expanding the branch on the left side of the tree. Once $h(n) = 5$, the search finds another peak value, then returns and searches to the goal state where $h(n) = 0$.

Chapter 4

Four-Thruster Hovercraft Example

The performance of AD-Lib is tested by simulating the motion planning of a four-thruster hovercraft. The tests are designed to accomplish three main goals: compare AD-Lib to existing suboptimal search algorithms, identify the performance of AD-Lib under different heuristics and libraries, and apply AD-Lib along with a controller in the presence of disturbances. In the first set of tests, AD-Lib is compared to DWA* in an environment containing known static obstacles to determine the performance gains of AD-Lib, if any. The second set of tests applies AD-Lib along with naive replanning and compares the results to AD-Lib with known static obstacles. The final simulation set uses AD-Lib in a dynamic known obstacle field to find a feasible trajectory through the obstacles, and then executes the trajectory using a hybrid controller.

4.1 Vehicle Dynamics and Maneuver Generation

This section provides the nonlinear equations of motion for the four thruster hovercraft used in the following tests and the associated library of dynamically feasible maneuvers. A

continuous flow of air from a perforated sheet supports the hovercraft, producing a near frictionless surface. The four thrusters are positioned equidistantly from the central axis, arranged as shown in Figure 4.1. The maximum force of the thrusters is 2.5 N, and the input is applied in discrete-time with a sampling frequency of 20 Hz, to account for hardware limitations. For each time step, each thruster must satisfy $\{u_i \mid u_i k \in [0, 2.5] \ \forall \ kT \leq t < (k+1)T\}$ $i = 1, 2, 3, 4$. The nonlinear equations of motion from the free-body diagram in Figure 4.1 are given by:

$$m\ddot{x} + b_t\dot{x} = (u_1 - u_3)\cos(\theta) + (u_2 - u_4)\sin(\theta) \tag{4.1}$$

$$m\ddot{y} + b_t\dot{y} = (u_1 - u_3)\sin(\theta) + (u_4 - u_2)\cos(\theta) \tag{4.2}$$

$$J\ddot{\theta} + b_r\dot{\theta} = L(u_1 - u_2 + u_3 - u_4) \tag{4.3}$$

Table 4.1: Hovercraft parameters

mass	m	1.731	kg
translational viscous friction	b_t	3.7×10^{-3}	N s/m
rotational viscous friction	b_r	3.65×10^{-4}	N s m/rad
polar moment of inertia	J	2.363×10^{-2}	kg m^2
half-radius	L	0.15	m

The procedure for generating the maneuvers is as follows. First, a smooth trajectory for the angular displacement θ is assumed. Given this trajectory, the problem is formulated as a convex optimization program, which is solved using `CVX`, a high-level convex program solver [17, 18]. The time is minimized by incorporating the bisection method to find the minimum number of time steps necessary for feasibility of the convex program. This procedure is repeated using different trajectories for θ until a satisfactory trajectory is found.

When designing the library, there exists a tradeoff between a large, maneuverable library and a small library which facilitates expansion. With fewer possible successors, small libraries

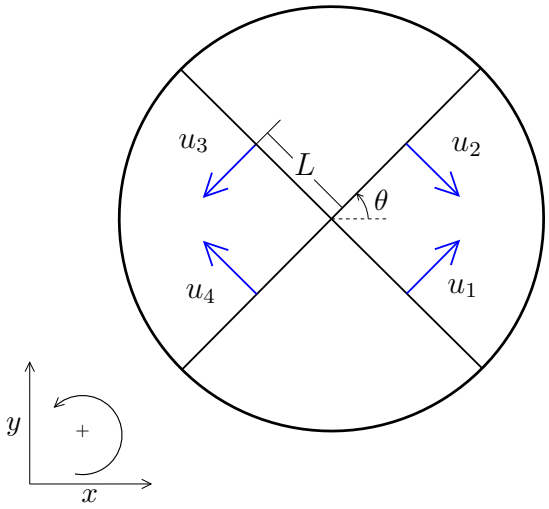

Figure 4.1: Four-thruster hovercraft model and parameters

allow for faster expansion by limiting the branching factor, b, of each node, which directly limits the worst-case expansion of heuristic searches. In contrast, a library without sufficient maneuverability could result in larger path costs or infeasibility when compared to a larger library.

For the hovercraft, the steady-state motion is forward in the body frame, where forward is defined along the $\theta = 0$ direction. Five maneuvers in the hovercraft library change the heading of the hovercraft from 0 rad to $[\frac{\pi}{2}, \frac{\pi}{4}, 0, \frac{-\pi}{4}, \frac{-\pi}{2}]$ rad. The initial and final velocities of each trajectory are both equal to 1 m/s in the direction of the vehicle heading. A detailed time history of the 0 rad to $\frac{\pi}{2}$ rad trajectory is given in Figure 4.2.

A steady-state maneuver is computed to keep the hovercraft on its current heading with a fixed velocity of 1 m/s. Nonlinear programming techniques do not set the coasting times of such trim trajectories, but treat their duration as variables of optimization. The length of this trajectory directly affects the number of nodes in the tree and ultimately the reachability set. For the current example, the trim trajectory coasting time is set to 0.4 s, giving a path length of 0.4 m, which is approximately the average length of the maneuver set. Since the cost-to-go,

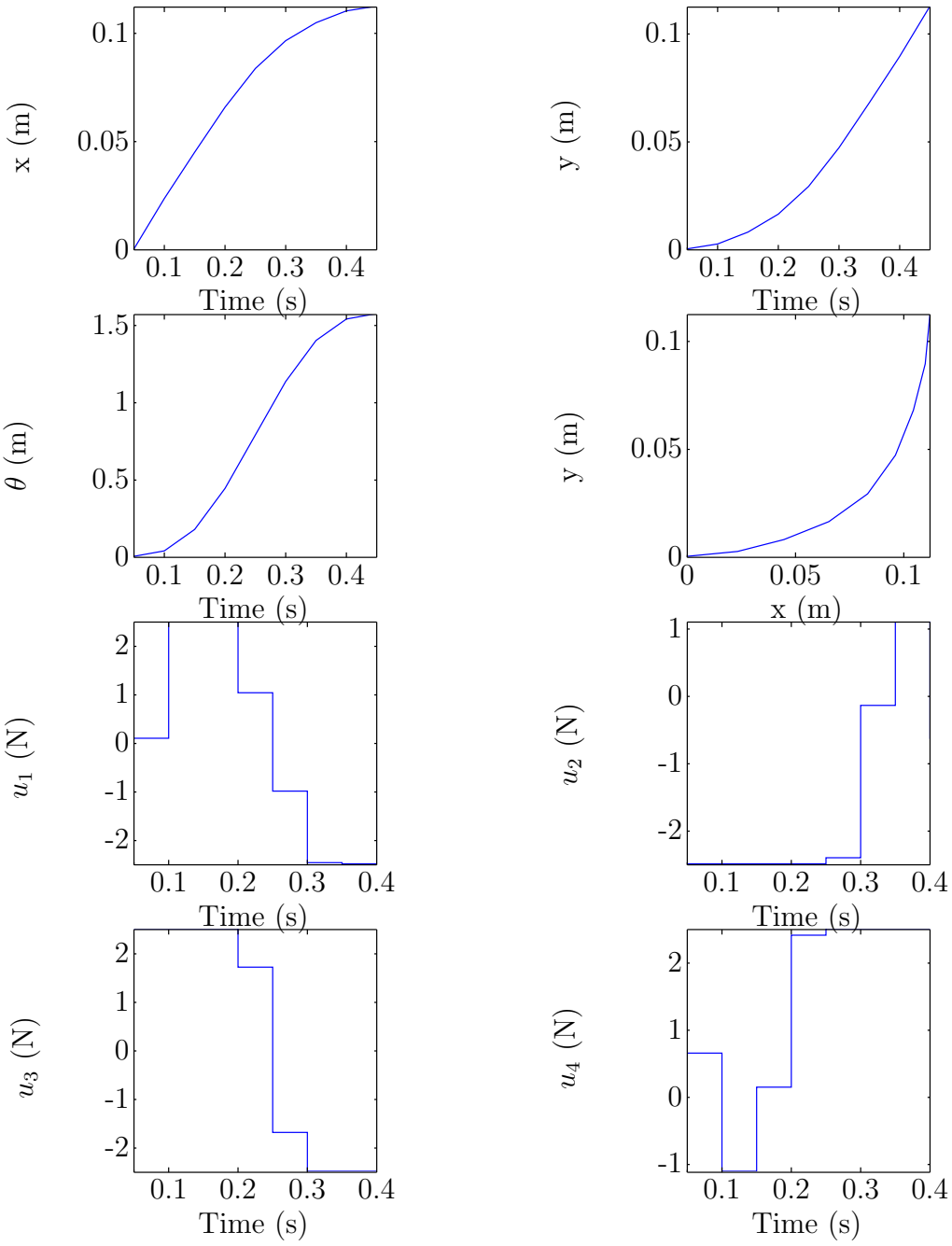

Figure 4.2: Transition maneuver from $\theta_i = 0$ rad to $\theta_f = \frac{\pi}{2}$ rad

$c(n, n')$, of each maneuver is relatively equal, the expansion process can be governed solely by the cost-to-goal, $h(n)$, instead of using $h_c(n)$. By removing this additional calculation,

run times may be slightly reduced. By setting the maneuver length around the library average, the expansion will not favor one trajectory more than another during expansion when examining $h_c(n)$.

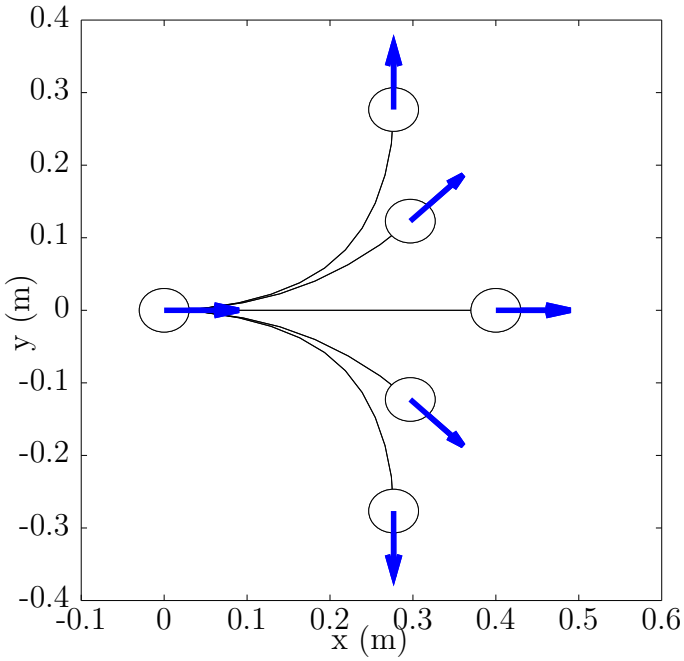

Figure 4.3: Maneuver library for the four-thruster hovercraft at 1 m/s

4.2 Sensor Modeling Using Laser Range Finder

Laser range finders (LRFs) record the time of flight from the laser to the obstacle and back to the LRF collector, and then multiply this time by the speed of light to find the distance to the obstacle. By traversing the laser across the environment, range and angle data are given by the sensor as polar coordinates in the vehicle frame, $R = [(r_i, \theta_i) \mid r, \theta, i = 1, 2, 3...]$. Typical systems have a field of view of 270° with an angular resolution of 0.36° scanning at a rate of 10 Hz. Noise in the LRFs can be characterized as zero-mean Gaussian noise with

typical σ value varying from a few centimeters to a few millimeters depending on sensor quality [19].

The LRF data is decomposed into line segments that represent obstacles in the 2D environment. Several methods exist to detect line segments from the data, such as Hough transforms, linear regressions, and split-and-merge [19–21]. Split-and-merge has been shown to be faster and more accurate than existing approaches [22]. From the polar data points, split-and-merge detects where there are breaks in the data, then recursively splits this data into smaller segments by finding the "corners" of the data. The final result of the line extraction is a set of line segments representing the obstacle edges seen by the hovercraft.

4.2.1 Heuristics from LRF Data

The LRF data provide the vehicle with domain-specific knowledge which can reduce search times by more accurately representing the perfect heuristic, $h^*(n)$. One means of accomplishing this estimation is to recreate the visibility graph from the LRF data. The visibility graph records which obstacle vertices can be "seen" by each obstacle vertex. For point-mass systems, the visibility graph connects the corners of obstacles by line-of-sight to provide $h^*(n)$ at any given point in the space. Computing the visibility graph can be computationally intensive, depending on the environment. A naive algorithm which checks every combination of vertices will operate in $O(n^2)$ time; in an environment with simple polygons without holes, the visibility graph can be computed in $O(E)$ time, where E is the number of edges of the final graph [23]. This output-sensitive behavior is typical for visibility graphs, where problem size dictates performance [24]. For more information on visibility graphs, the book by Ghosh describes in depth this field of study [25]. Some grid-based search algorithms, such as Theta* and Incremental Phi*, modify the grid path to resemble the visibility graph

after conducting an initial search[26, 27].

Relying on visibility graphs increases the computational requirements on the system, both in calculating the graph and storing the data. A proposed heuristic using the LRF data is the *line-of-sight* heuristic, which is intended to more accurately represent $h*(n)$ by penalizing states which are headed towards an obstacle. To check if a vehicle state is headed toward an obstacle, the obstacle line segments are checked for intersection with a ray, $\vec{r}$, projected from the vehicle. Given a line segment represented by two endpoints, $\vec{l_1}$ and $\vec{l_2}$, an intersection point (s,t) will be found which satisfies the following conditions:

$$\begin{aligned} (1+t)\vec{l_1} + t\vec{l_2} &= s\vec{r} \\ s &\geq 0 \\ t &\in [0,1] \end{aligned} \tag{4.4}$$

where s is the distance to the segment and t represents the location of the intersection on the line segment as a fraction of the length. Using the Euclidean norm as the nominal heuristic, represented by $h_2(n)$, a penalty is added if the state is directed towards on obstacle, which is equal to the distance of the segment midpoint to the goal, $h(seg)$. If multiple segments satisfy the conditions, the segment with the minimum s value is chosen. The line-of-sight heuristic is given as:

$$h_1(n) = \begin{cases} s + h(seg) & \text{if intersection} \\ h_2(n) & \text{otherwise} \end{cases} \tag{4.5}$$

The two heuristics are shown in Figure 4.4. For convenience the line-of-sight heuristic

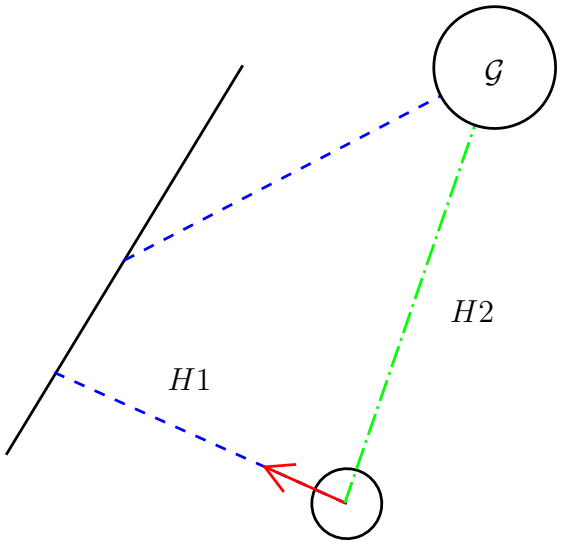

Figure 4.4: Two heuristics used for testing: the line-of-sight heuristic, H1, and the Euclidean norm, H2

will be referenced as H1 and the Euclidean norm heuristic as H2. The performance of both heuristics is compared simultaneosly in all of the following tests.

4.3 Known Static Obstacles

AD-Lib is compared to DWA*, an existing suboptimal heuristic search algorithm, in an environment containing obstacles whose positions are known beforehand. By varying the weighting coefficient of DWA*, the "greediness" of DWA* can be varied, allowing for a more flexible comparison to AD-Lib. The environment is constructed by placing randomly-sized, randomly-distributed obstacles in the search space. The obstacles are rectangular with a maximum side length of one meter. The length and width are chosen using MatLab's *rand* function, which provides uniformly distributed pseudorandom numbers between zero and one. The obstacle placement is also governed by the *rand* function. Obstacle centroids must reside in the square between (2,2) and (8,8) shown as a dashed blue box in Figure 4.5.

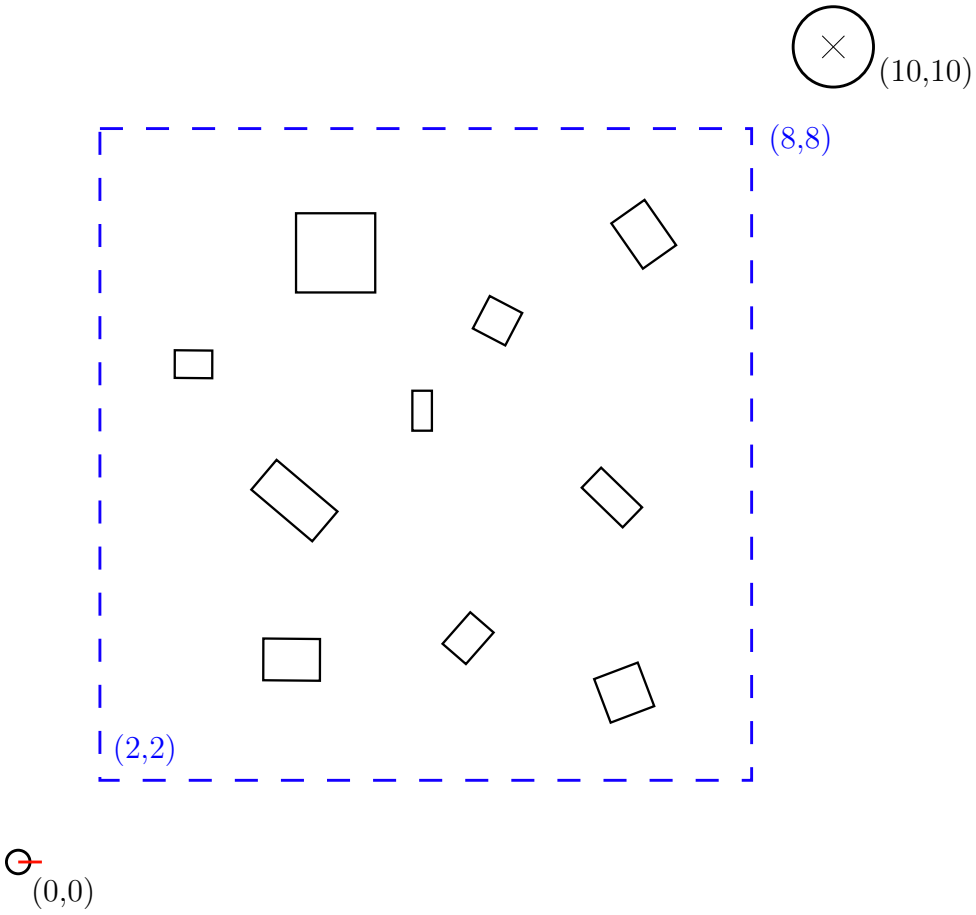

Figure 4.5: Environment with Known Static Obstacles

The output of *rand* is adjusted so the centroid resides in this region. The orientation of the rectangles is allowed to vary between zero and 2π, again scaled by the *rand* function. Obstacles are allowed to overlap, creating more complex shapes.

The algorithms are compared using four measures of performance: runtime, total number of nodes, number of nodes on the path, and path length. The first two measures, runtime and total number of expanded nodes, indicate the relative performance of each algorithm. For example, if one algorithm expands more nodes than the other, then this algorithm requires more computational resources and subsequently more time to find a trajectory. Each individual test is limited to 100,000 nodes. If this limit is exceeded, then the test is

terminated and recorded as an over-expansion (OE). The run time only provides a relative comparison between the algorithms, as the tests are run in MatLab which is not a real time language. The run times indicate if one algorithm requires significantly more time to expand than another. Such differences may be reduced by using a real time language such as C/C++. The final two measures, the path depth and path length, indicate the relative cost of the solution. Shorter paths are the goal of most search algorithms as discussed in Section 2.1. However, fewer nodes on the path result in a faster expansion when using greedy search. The resulting path from a search with fewer nodes may be longer than other feasible paths, and these two measures examine if the difference is significant.

4.3.1 Test Results

Several configurations of the algorithms are selected for testing. For DWA*, the weights used are 0.5, 0.75, and 1 for w. All algorithm configurations are tested using both the line-of-sight heuristic, H1, and the Euclidean norm heuristic, H2. Using two different heuristics shows what difference, if any, arises from different types of heuristics. The tests are run on a 3.07 GHz Intel Xeon processor with 6 GB of memory on the Windows 7 operating system.

Examining the results shown in Table 4.2, it is clear that AD-Lib expands far fewer nodes than all weightings of DWA*, in most cases by an order of magnitude. A higher percentage of the expanded nodes are on the path returned by AD-Lib, which is expected since AD-Lib attempts to reduce wasted search effort. The depth of the paths returned by AD-Lib are higher than for DWA*, which may be due to AD-Lib propensity for continuing the path instead of returning to more promising options like DWA*. These results are indicative of greedy expansion, which is designed to utilize expanded nodes at the expense of cost. For most of the test cases, the cost of the AD-Lib algorithm does increase, on average 150%

Table 4.2: Known static obstacles

5 obstacles, 4990 tests		Nodes	Depth	Cost	Time (ms)	OE
H1	DWA*, $w = 0.5$	825	39	14.68	95	1
	DWA*, $w = 0.75$	553	39	14.76	62	0
	DWA*, $w = 1$	599	39	14.82	89	0
	DWA*, all w	659	39	14.75	82	0
	AD-Lib	86	50	19.37	58	3
H2	DWA*, $w = 0.5$	755	38	15.23	49	4
	DWA*, $w = 0.75$	525	38	15.44	30	0
	DWA*, $w = 1$	471	39	15.54	38	1
	DWA*, all w	583	38	15.40	39	2
	AD-Lib	71	40	15.27	59	2
10 obstacles, 4841 tests		Nodes	Depth	Cost	Time (ms)	OE
H1	DWA*, $w = 0.5$	2 032	40	15.09	399	28
	DWA*, $w = 0.75$	1 607	41	15.24	322	16
	DWA*, $w = 1$	1 975	41	15.36	463	47
	DWA*, all w	1 871	41	15.23	395	30
	AD-Lib	170	69	27.01	211	25
H2	DWA*, $w = 0.5$	1 886	40	15.45	198	60
	DWA*, $w = 0.75$	1 110	40	15.64	74	30
	DWA*, $w = 1$	1 175	40	15.77	103	41
	DWA*, all w	1 390	40	15.62	125	44
	AD-Lib	103	48	18.27	24	26
15 obstacles, 4270 tests		Nodes	Depth	Cost	Time (ms)	OE
H1	DWA*, $w = 0.5$	4 109	42	15.50	1 166	135
	DWA*, $w = 0.75$	3 549	42	15.71	1 000	126
	DWA*, $w = 1$	4 761	42	15.86	1 568	250
	DWA*, all w	4 140	42	15.69	1 244	170
	AD-Lib	378	89	35.91	700	77
H2	DWA*, $w = 0.5$	4 273	41	15.71	641	291
	DWA*, $w = 0.75$	2 516	41	15.92	277	205
	DWA*, $w = 1$	2 914	41	16.07	396	250
	DWA*, all w	3 234	41	15.90	438	249
	AD-Lib	258	62	23.76	164	130

of the corresponding DWA* averages and as much as 230% for the 15 obstacle tests using the line-of-sight heuristic. While the magnitude of the cost increase may not be appropriate for some vehicular applications, AD-Lib is designed primarily for agile vehicles in situations where speed trumps cost. The run times also correspond to the expected behavior of the

algorithms, with AD-Lib taking more time per node than any weighting of DWA*. The time increase arises from the additional checks on $h(n)$ in AD-Lib during node expansion.

Examining the performance of both heuristics, the Euclidean norm heuristic, H2, clearly outperforms the line-of-sight heuristic, H1. For AD-Lib, line-of-sight has increases in every category except in the number of over-expansions. In DWA* expansion, the line-of-sight heuristic slightly improves the cost between the averages, but while incurring a significant time penalty. From the data, the additional complexity of the line-of-sight heuristic does not offset any minor gains in cost that may exist.

Two Library Testing

In maneuver-based motion planning, the configuration of the library directly affects the reachability set of the vehicle. To test what impacts the library have on path expansion, a second library was developed in a similar fashion, except it used a steady-state velocity of 0.5m/s. For convenience, the library with initial and final velocities equal to 0.5 m/s will be referred as L1, and the original library being called L2. The two libraries are shown in Figure 4.6, with L1 shown in blue and L2 shown in red. Using the same setup as previously, each configuration of DWA* and AD-Lib are tested using each library separately to identify the differences in expansion which arise by altering the reachability set. The results for the tests are shown in Table 4.3, 4.4, and 4.5.

Similar to the single library tests, both libraries expand fewer nodes than DWA*, and returns longer paths than the DWA* cases. Using the Euclidean norm, AD-Lib finds shorter paths than with the line-of-sight heuristic. Particularly for the cluttered cases, AD-Lib using L2 returns significantly longer paths than when L1 is used. The shorter paths of L1 allows the expansion to find shorter paths which may not be possible with longer paths, due to

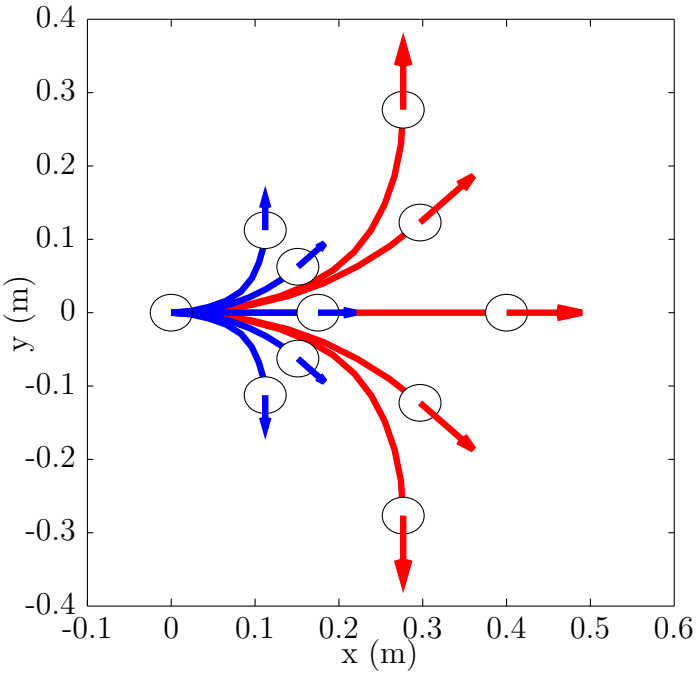

Figure 4.6: Maneuver libraries for the 0.5 m/s library (blue) and the 1 m/s library (red)

the limitations of the reachability set for such a library. In cluttered environments, libraries with shorter path lengths may return shorter trajectories; however, the number of nodes expanded will increase, particularly if the search must backtrack significantly.

4.4 Unknown Static Obstacles

Since AD-Lib is designed to respond rapidly to changes in the environment, the second set of test cases examines the performance of AD-Lib in an unknown obstacle field. Initially, the vehicle only knows of its immediate surroundings as relayed by the LRF sensor model. From this snapshot of the obstacle field, an initial trajectory through the environment is computed, and the vehicle begins to traverse the path. At each sensor time step of τ_s = 0.01 s, a new picture of the environment is captured from the LRF data. At each control time step,

Table 4.3: Known static obstacles, two libraries, 5 obstacles

5 obstacles, 4800 tests			Nodes	Depth	Cost	Time (ms)	OE
L1	H1	DWA*, $w = 0.5$	1 288	85	12.80	165	88
		DWA*, $w = 0.75$	944	86	13.12	110	63
		DWA*, $w = 1$	968	86	13.21	119	55
		DWA*, all w	1 067	86	13.05	132	69
		AD-Lib	205	100	15.40	132	49
	H2	DWA*, $w = 0.5$	1 236	84	12.53	149	113
		DWA*, $w = 0.75$	937	86	13.63	101	94
		DWA*, $w = 1$	947	87	14.08	103	79
		DWA*, all w	1 040	86	13.42	118	95
		AD-Lib	165	90	13.66	90	24
L2	H1	DWA*, $w = 0.5$	705	39	14.67	83	2
		DWA*, $w = 0.75$	505	39	14.74	60	0
		DWA*, $w = 1$	470	39	14.80	60	2
		DWA*, all w	560	39	14.74	67	1
		AD-Lib	66	49	18.68	37	3
	H2	DWA*, $w = 0.5$	539	38	15.22	55	5
		DWA*, $w = 0.75$	406	38	15.42	41	1
		DWA*, $w = 1$	346	38	15.52	35	0
		DWA*, all w	430	38	15.38	44	2
		AD-Lib	52	40	15.18	26	2

$\tau_c = 0.05$ s, the nodes on the path are checked for collision using the updated knowledge of the environment. If a collision is detected, a new path is computed from the next node on the path. The replanning continues until a path to the goal is found. An example of the environment with the sensor sweep of the vehicle is shown in Figure 4.7.

The strategy described is known as "naive" replanning, where previous search information is not used in subsequent plans. Naive replanning does not require active maintenance of the previous search tree while the vehicle traverses the path. Replanners, such as lifelong planning A* (LPA*) and D*-Lite, modify previous knowledge of the graph to accommodate changes in the environment [28, 29]. These replanners are designed to operate on well connected graphs, such as grids, and not on trees, where the connections between states are fixed.

Table 4.4: Known static obstacles, two libraries, 10 obstacles

10 obstacles, 4800 tests			Nodes	Depth	Cost	Time (ms)	OE
L1	H1	DWA*, $w = 0.5$	2 632	87	13.29	501	569
		DWA*, $w = 0.75$	1 936	88	13.50	358	400
		DWA*, $w = 1$	2 123	89	13.65	416	391
		DWA*, all w	2 231	88	13.48	425	453
		AD-Lib	382	115	18.14	381	298
	H2	DWA*, $w = 0.5$	2 186	86	12.92	387	659
		DWA*, $w = 0.75$	1 538	88	14.01	252	531
		DWA*, $w = 1$	1 481	89	14.33	248	470
		DWA*, all w	1 735	88	13.75	296	553
		AD-Lib	271	104	16.19	205	211
L2	H1	DWA*, $w = 0.5$	1 309	40	14.99	239	38
		DWA*, $w = 0.75$	1 043	40	15.12	190	36
		DWA*, $w = 1$	1 209	40	15.22	242	78
		DWA*, all w	1 187	40	15.11	224	51
		AD-Lib	138	66	26.19	130	26
	H2	DWA*, $w = 0.5$	970	39	15.37	157	83
		DWA*, $w = 0.75$	670	39	15.55	107	52
		DWA*, $w = 1$	649	40	15.66	108	66
		DWA*, all w	763	39	15.52	124	67
		AD-Lib	81	47	17.74	59	27

4.4.1 Test Results

The naive replanning AD-Lib performance is compared to AD-Lib with full knowledge of the environment to examine the costs of greedy expansion. The important test variables are number of plans, total number of expanded nodes, final path depth, and final path length. From these variables, the per-plan performance of the replanner can also be examined and compared to the fully known case. The test environment is identical to that in the known static obstacle case. The results of the tests are given in Table 4.6.

From the data, the total expanded nodes in the unknown case are relatively small when compared to the DWA* cases in Table 4.2. Naive replanning AD-Lib quickly conducts each separate replanning task, reducing the aggregate time for the motion planning task. As

Table 4.5: Known static obstacles, two libraries, 15 obstacles

15 obstacles, 2393 tests			Nodes	Depth	Cost	Time (ms)	OE
L1	H1	DWA*, $w = 0.5$	4 365	89	13.65	1 084	1 496
		DWA*, $w = 0.75$	3 026	90	13.85	735	1 189
		DWA*, $w = 1$	3 321	91	14.00	852	1 129
		DWA*, all w	3 571	90	13.83	890	1 271
		AD-Lib	618	128	20.35	735	708
	H2	DWA*, $w = 0.5$	3 530	87	13.24	806	1 552
		DWA*, $w = 0.75$	2 169	90	14.30	463	1 345
		DWA*, $w = 1$	2 124	90	14.56	473	1 215
		DWA*, all w	2 608	89	14.03	581	1 371
		AD-Lib	511	127	20.28	572	712
L2	H1	DWA*, $w = 0.5$	2 175	41	15.31	539	248
		DWA*, $w = 0.75$	1 869	41	15.46	473	225
		DWA*, $w = 1$	2 230	42	15.58	597	374
		DWA*, all w	2 092	41	15.45	536	282
		AD-Lib	174	75	29.85	167	82
	H2	DWA*, $w = 0.5$	1 467	40	15.53	319	369
		DWA*, $w = 0.75$	1 043	40	15.72	226	280
		DWA*, $w = 1$	1 059	41	15.84	242	334
		DWA*, all w	1 190	40	15.70	262	328
		AD-Lib	97	47	17.62	92	126

stated earlier, the time spent finding a trajectory could potentially be reduced by using a real time programming language. Similarly to the known static obstacle case, the line-of-sight heuristic, H1, performs poorly compared to the Euclidean norm, H2.

4.5 Dynamic Environments

In an environment with dynamic obstacles, AD-Lib is designed to quickly find a motion plan, and then this motion plan would be executed using a robust controller. For the simulated example below, a switching controller is designed using the H_∞ norm as a performance measure. A subcontroller is designed for each maneuver in the library; but unlike standard H_∞ control, extra care is taken in designing the subcontrollers to ensure that the switching

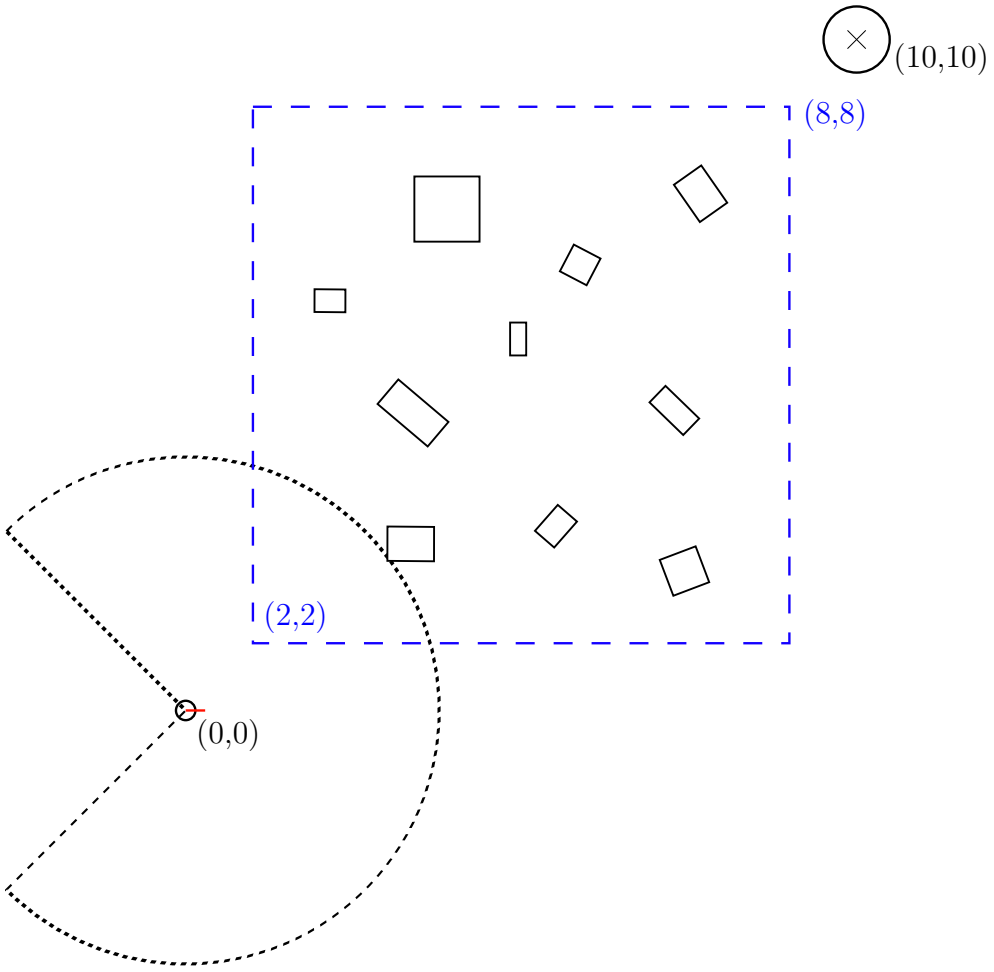

Figure 4.7: Environment with Sensor Sweep

takes place with stability and performance guarantees. The standard H_∞ synthesis conditions can be found in [30]. Note that the nodes on the path retained by AD-Lib represent the "switching points" between the subcontrollers. More on the switching control design is given in [31].

The switching controller attenuates the disturbance by modifying the reference inputs. Disturbances are modeled during simulation to show the utility of the switching controller about the trajectory generated by AD-Lib. At each time step k, disturbance forces act in the x and y directions, f_{xk} and f_{yk}, where $|f_{xk}|, |f_{yk}| \leq 1\text{N}$. A disturbance torque, τ_k, is

Table 4.6: Unknown static obstacles

5 obs., 3756 tests		Plans	Nodes	Depth	Cost	Time (ms)	OE
H1	Known	1	39	38	14.39	9	0
	Unknown (total)	8	305	40	15.03	144	713
H2	Known	1	51	40	15.13	10	2
	Unknown (total)	9	259	39	14.75	46	800
10 obs., 2682 tests		Plans	Nodes	Depth	Cost	Time (ms)	OE
H1	Known	1	40	38	14.53	11	0
	Unknown (total)	15	708	44	16.65	398	1 316
H2	Known	1	110	45	17.08	51	23
	Unknown (total)	15	530	42	15.60	103	1 548
15 obs., 1711 tests		Plans	Nodes	Depth	Cost	Time (ms)	OE
H1	Known	1	43	40	15.29	12	0
	Unknown (total)	19	1 006	46	17.27	572	1 943
H2	Known	1	247	59	22.41	204	109
	Unknown (total)	23	1 377	47	17.48	464	2 433

also placed on the vehicle, where $|\tau_k| \leq 0.15$N m. The magnitude of the iid disturbances in discrete-time are scaled by the *rand* function.

The obstacle dynamics are simulated using a simple impulse model, which provides an interesting and changing environment to test the motion planner [32, 33]. Using the impulse model, the obstacle positions are found for a time horizon well exceeding the expected path planning problem, generally 60 seconds. The obstacles at each time step are decomposed into line segments and are used in motion planning. These line segments are handed to AD-Lib and the maneuver sequence is given to the switching controller. Several test cases are run with various obstacle configurations to ensure the validity of the model, with a typical result shown below.

As shown in Figure 4.9, the vehicle maneuvers through the obstacle field and executes the desired motion plan with relatively little error in the states. Because of the additional guarantees which the switching controller provides, the vehicle is able to successfully switch between several subcontrollers over the course of the motion plan. Future research is set to

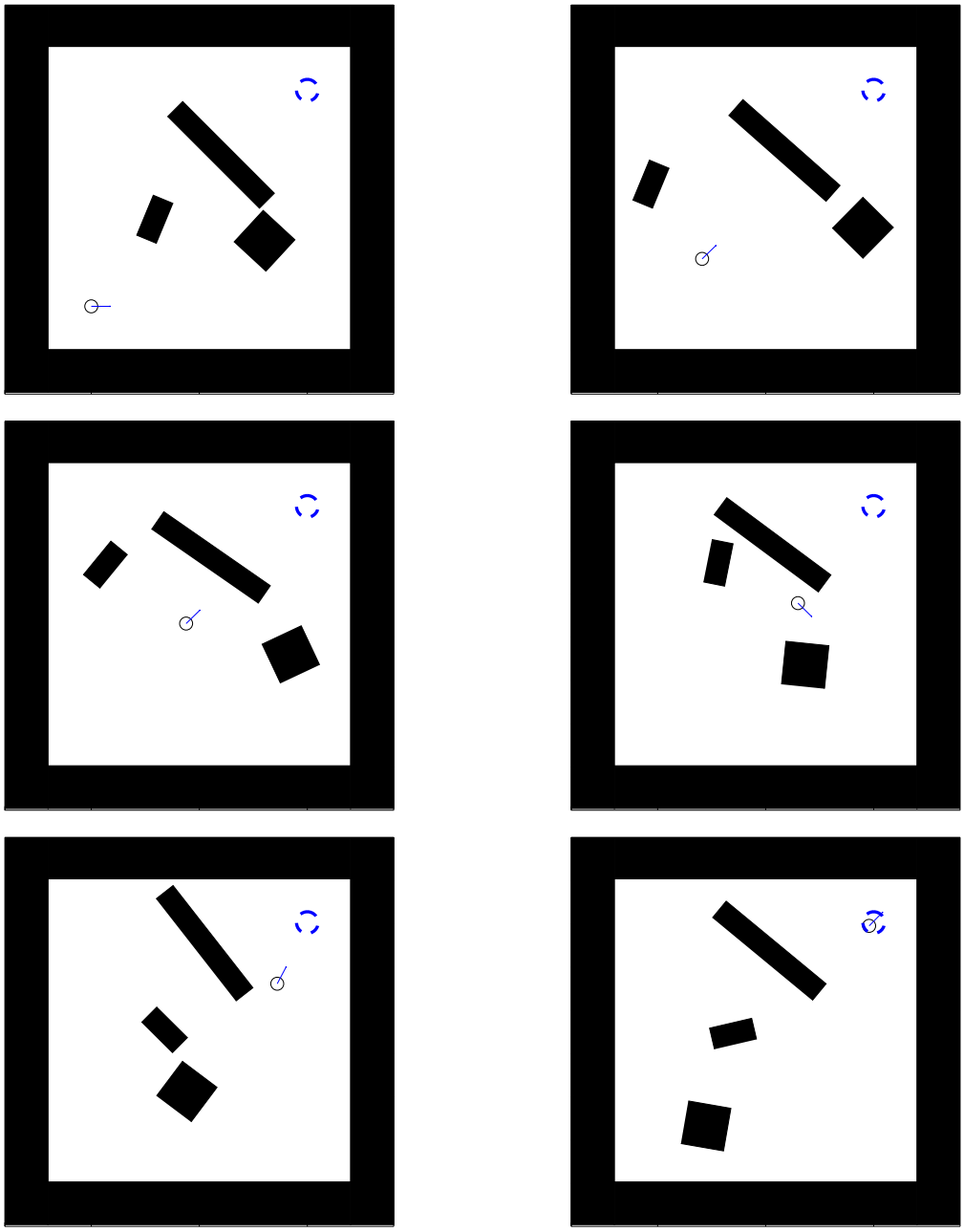

Figure 4.8: Snapshots of dynamic obstacle environment

examine the benefits of the switching controller to maneuver-based motion planning.

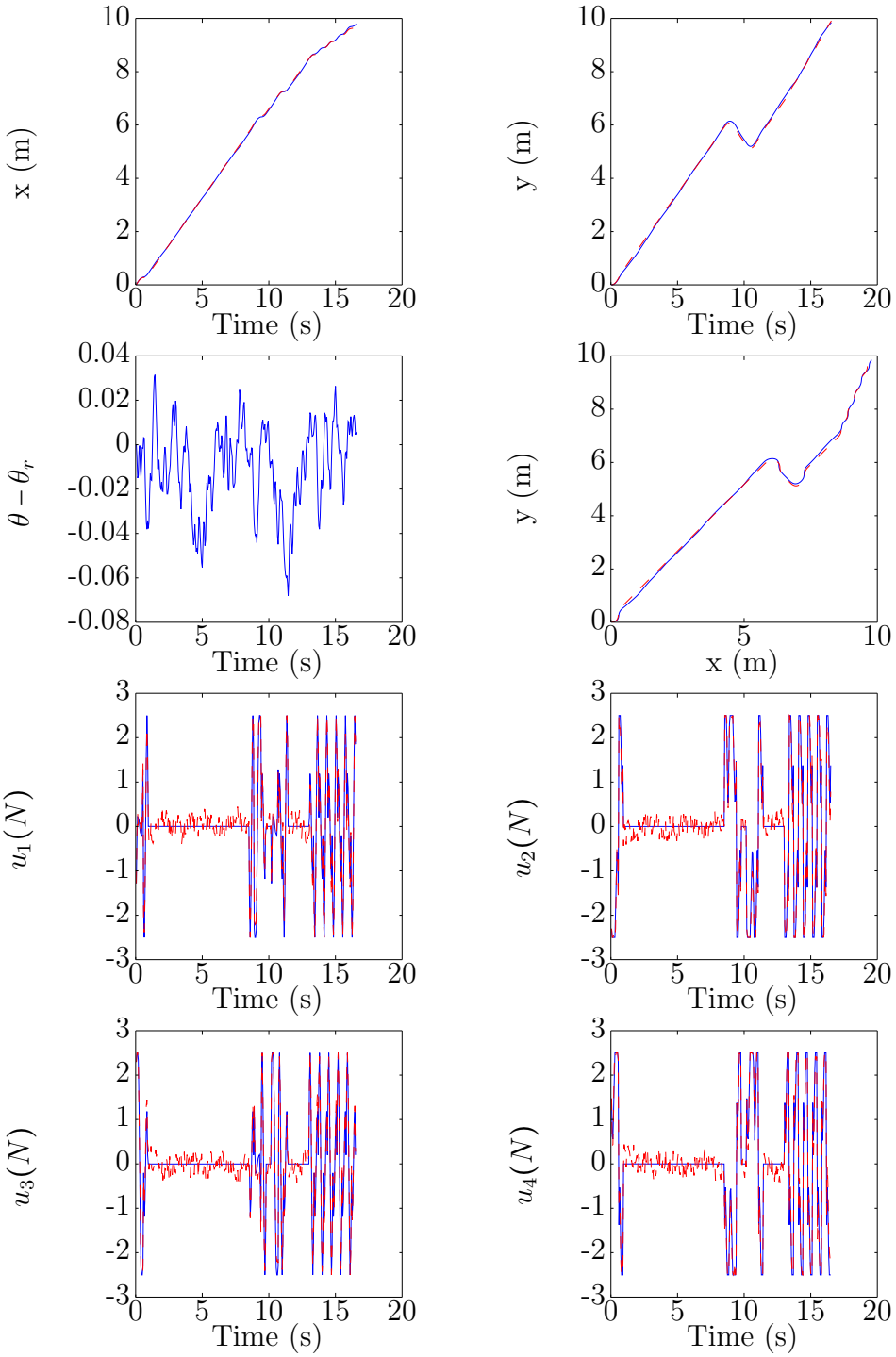

Figure 4.9: Dynamic Obstacle Simulation with errors shown in dashed red

Chapter 5

Conclusion

In this thesis, a new motion planning algorithm called AD-Lib is developed as an effective and fast motion planning strategy for agile vehicles. Using maneuver-based motion planning, AD-Lib uses the prespecified library of maneuvers to conduct a graph search through obstacle environments. AD-Lib uses greedy search with local hill climbing to quickly approach the goal region in order to generate a feasible trajectory while limiting over-expansion through effective backtracking. The backtracking approach uses the cost-to-goal, $h(n)$, to judge the quality of a given node as compared to previous nodes on the path. If a node exceeds limits set by the expansion, then the search returns to explore more promising options by revisiting the so-called "watch" nodes along the path. The greedy search strategy of AD-Lib reduces the time required for motion planning to avoid jeopardizing the safety of the vehicle.

From test data gathered in both known obstacle environments, AD-Lib expands fewer nodes than multiple configurations of DWA*, a standard suboptimal path planner; in most cases, the number of nodes expanded is reduced by an order of magnitude. By employing the framework of greedy search, AD-Lib returns path with a substantial cost increase, which may not be acceptable for all applications. However, the significant reductions in run time make

AD-Lib a good choice for agile vehicles where feasible trajectories are required as quickly as possible.

In unknown environments, AD-Lib using naive replanning is shown to successfully navigate through an obstacle field and quickly respond to changing information about the search space. The aggregate time required for motion planning averages under a tenth of a second for the fifteen obstacle cases examined during testing. Unlike other suboptimal search algorithms, the focus on speed makes AD-Lib a good choice for vehicles which operate in changing environments.

Finally, AD-Lib can produce motion plans suitable for use with a switching controller using the H_∞ norm as the performance measure. The switching controller alternates between subcontrollers designed for each trajectory, with extra care taken to ensure stability and performance around the switching points. Such an approach is advantageous for maneuver-based motion planning, where the maneuvers are independent and then pieced together. The switching controller provides the vehicle with the stability guarantees required to make the transition between maneuvers on the trajectory.

In future work, the motion planner can be used on more complex systems, such as small unmanned helicopters or fixed wing aircraft. The computational savings of using a maneuver-based approach should become more pronounced as the size and complexity of the system grows. However, it is difficult to determine how appropriate maneuver-based motion planning is for a given system compared to other motion planning methods, since implementing a motion planner is problem specific. A benchmark system, such as a helicopter, could be used to compare more diverse graph search methods.

Bibliography

[1] E. Frazzoli, M. Dahleh, and E. Feron, "Maneuver-based motion planning for nonlinear systems with symmetries," *IEEE Transactions on Robotics*, vol. 21, no. 6, pp. 1077–1091, Dec. 2005.

[2] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. New York: Prentice Hall, 2010.

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

[4] R. E. Korf, "Artificial Intelligence Search Algorithms," *Search*, pp. 1–40, 1996.

[5] ——, "Depth-first iterative-deepening: An optimal admissible tree search," *Artificial intelligence*, vol. 27, pp. 97–109, 1985.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions On Systems Science And Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, Dec. 1959.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. Cambridge, MA: MIT Press, 2009.

[9] L. Harris, "The heuristic search under conditions of error," *Artificial Intelligence*, vol. 5, no. 3, pp. 217–234, 1974.

[10] I. Pohl, "Heuristic search viewed as path finding in a graph," *Artificial Intelligence*, vol. 1, no. 3-4, pp. 193–204, 1970.

[11] ——, "The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving," *Proceedings of the 3rd international joint conference*, 1973.

[12] J. Pearl and J. H. Kim, "Studies in Semi-Admissible Heuristics," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-4, no. 4, pp. 392–399, Jul. 1982.

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

[14] S. Kirkpatrick, "Optimization by simulated annealing: Quantitative studies," *Journal of Statistical Physics*, vol. 220, no. 4598, pp. 671–80, May 1984.

[15] V. Granville and M. Krivanek, "Simulated annealing: A proof of convergence," *Pattern Analysis and*, vol. 16, no. 6, pp. 652–656, Jun. 2002.

[16] C. Wilt and J. Thayer, "A Comparison of Greedy Search Algorithms," *Artificial Intelligence*, 2010.

[17] M. Grant and S. Boyd, "CVX: Matlab software for disciplined convex programming (web page and software)," 2009. [Online]. Available: http://cvxr.com/cvx

[18] ——, *Graph implementations for nonsmooth convex programs*, ser. Lecture Notes in Control and Information Sciences. London: Springer London, 2008, vol. 371, pp. 95–110. [Online]. Available: http://stanford.edu/~{}boyd/graph_dcp.html

[19] G. Borges and M. Aldon, "Line extraction in 2D range images for mobile robotics," *Journal of Intelligent and Robotic Systems*, vol. 40, no. 3, pp. 267–297, 2004.

[20] D. H. Ballard, "Generalizing the Hough transform to detect arbitrary shapes*," *Pattern recognition*, 1981.

[21] A. Mendes, L. Bento, and U. Nunes, "Multi-target Detection and Tracking with a Laserscanner," in *Proc. of 2004 IEEE Intelligent Vehicles Symposium*. IEEE, 2004, pp. 796–801.

[22] V. Nguyen, A. Martinelli, N. Tomatis, and R. Siegwart, "A comparison of line extraction algorithms using 2D laser rangefinder for indoor mobile robotics," in *2005 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2005.(IROS 2005)*, 2005, pp. 1929–1934.

[23] J. Hershberger, "Finding the Visibility Graph in Time Proportional of a Simple Polygon to its Size," *Algorithmica*, no. 4, pp. 141–155, 1989.

[24] S. K. Ghosh and D. M. Mount, "An Output Sensitive Algorithm for Computing Visibility Graphs," *Proceedings of the 28th Annual Symposium on Foundations of Computer Science*, pp. 11–19, 1987.

[25] S. K. Ghosh, *Visibility Algorithms in the Plane*. Cambridge: Cambridge University Press, 2007.

[26] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta*: Any-Angle Path Planning on Grids," in *Proceedings of the National Conference on Artificial Intelligence*, vol. 22, no. 2, 2007, p. 1177.

[27] A. Nash, S. Koenig, and M. Likhachev, "Incremental Phi*: incremental any-angle path planning on grids," in *Proceedings of the 21st international jont conference on Artifical intelligence*. Morgan Kaufmann Publishers Inc., 2009, pp. 1824–1830.

[28] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artificial Intelligence*, vol. 155, no. 1-2, pp. 93–146, 2004.

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

[30] G. E. Dullerud and F. Paganini, *A Course in Robust Control Theory: A Convex Approach*. Springer, 2000.

[31] C. B. Neas and M. Farhood, "A Hybrid Architecture for Maneuver-Based Planning and Control of Agile Vehicles," in *IFAC World Congress*, [Awaiting Review], 2010.

[32] B. Mirtich, "Impulse-based dynamic simulation of rigid body systems," 1996.

[33] E. Neumann, "Rigid Body Collisions," 2004. [Online]. Available: http://www.myphysicslab.com/collision.html