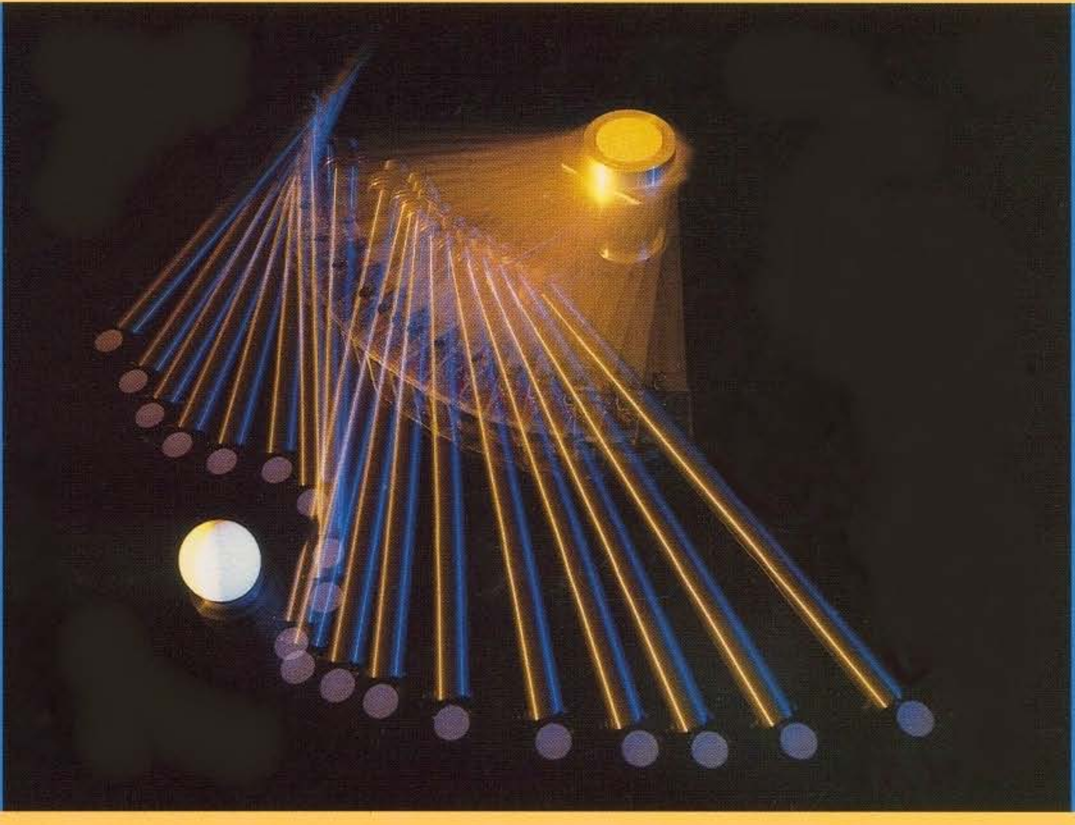
A* Planning in Discrete Configuration Spaces of Autonomous Systems



Karen I. Trovato

A* Planning in Discrete Configuration Spaces of Autonomous Systems

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr. P.W.M. de Meijer
ten overstaan van een door het college van decanen ingestelde
commissie in het openbaar te verdedigen in de Aula der Universiteit
op maandag 9 september 1996 om 12.00 uur

door

Karen Irene Trovato

geboren te Cold Spring, New York USA

A* Planning in Discrete Configuration Spaces of Autonomous Systems

Dit academisch proefschrift is goedgekeurd door de promotores:
Prof. dr. ir. F.C.A. Groen
Dr. ir. L. Dorst

Samenstelling promotiecommissie:

Rector Magnificus, voorzitter
Prof. dr. ir. F.C.A. Groen, promotor
Dr. ir. L. Dorst, co-promotor
Prof. dr. J.L. Crowley, LIFIA, St. Martin-d'Heres, Cedex, France
Prof. dr. J.F.A.K. van Benthem, Universiteit van Amsterdam
Prof. dr. H.J. van den Herik, Rijks Universiteit Limburg, Maastricht
Prof. dr. N. Petkov, Rijks Universiteit Groningen, Groningen
Prof. dr. ir. A.W.M. Smeulders, Universiteit van Amsterdam

On the Front Cover: Newman's Two-Link Robot Arm
Photo: Frank Molinaro
Design: Marjorie Garrett and Lisa Cherian
Philips Reference: MS-96-030

To Steve and Mark

Table of Contents

List of Figures

“The Future Belongs To Those Who Plan For It”

- Charles E. Schroeder

Chapter 1

Introduction

1.1 Motivation for Planning

Planning our actions provides a way to achieve our goals with a certain level of optimality. The perfect plan can be constructed if we know all of the factors, past and future, that affect us reaching our goal. Therefore we can estimate the price of reaching our goal before we begin our actions, and see the hurdles that are inevitable. The best we can hope for is to be well informed about the past, guess well about the future, and have the perseverance to meet the hurdles or the power to remove them.

Even with this information in hand, few people analyze the impact of these factors on their own goals. This is the purpose of planning. It avoids duplication of effort by clearly directing the course of action, and can achieve optimal performance by anticipating likely hurdles. Perhaps just as important, it provides a way to decide if the price required to achieve the goal is higher than the value of the goal. Plans also provide a mechanism to understand the impact of unanticipated changes.

Plans for businesses and business related activities are derived from the overall strategy which defines the overall objective (i.e. what is to be done). Plans are common, because others must understand the direction in order to support it. There is also a direct measure of success corresponding to the plan, such as profit, growth or market share. Indirectly, the people who cause this success, in part because the plan is sound, are given rewards to motivate them further. The plan typically includes a set of scenarios describing the course of events dependent on the success or failure to meet major milestones (hurdles). Estimates to meet these milestones require (at a minimum) assumptions about levels of productivity of finance, people, and assets.

Whether machines (as assets) operate within a company or as a final product, the productivity and flexibility (ability to adapt to new tasks) of the machine will determine its longevity. For machines that produce commodity products, fixed automation which is fine tuned for a specific process work best. Flexible automation performs best when the quantities of products are small, or are custom made. This type of automation is the general application area of this thesis. Ideally, a machine can adapt itself to a new task autonomously, that is, without requiring any programming. The next generation of machines can be both flexible and productive by the use of software that provides intelligent, autonomous control.

Commercial robotic machines are often required to be taught every precondition and motion [56] because many do not have the luxury of sensory systems, and therefore perform tasks by rote. Advanced sensory systems, including the fusion of information from different sensing sources, are still a subject of research. We envision the day when robots have sufficiently inexpensive sensors so that they can perform complex tasks automatically, and at no higher cost than fixed automation. Once the sensing system is in place, we would like the machine to incorporate this knowledge of the environment and quickly revise the perhaps rote plans into optimal actions.

Latombe's vision for an intelligent autonomous system [21] is one where a machine can be given very simple, non-specific verbal orders, which the machine then carries out. It must first extract every nuance from the orders to define the overall objective. It then must recall relevant knowledge, identify lacking information, formulate a plan to acquire it (by sensors, perhaps), and finally create a series of action plans to cause the overall objective to be met.

The challenge is to define a system that takes high level, possibly vague, (possibly voice) commands describing what should be achieved, and then produces a complete and actionable plan describing how it should be achieved. Ideally, "Go to the moon, get some rocks, and return to earth by next Sunday. Oh, and don't spend more than we've budgeted" would be enough. In practice, the challenges for far more mundane tasks are still not solved. Each new vehicle model requires a new 'robot painting' program, which may or may not be optimal in any sense. Although methods to reuse parts of programs are quite common, the goal is to have machines behave intelligently, and autonomously. A major step in the right direction is to define what optimality means for our task (i.e. minimize time, cost, etc.), define the environment to the best of our sensing ability, and set out a goal, and allow the machine to deduce the motion. For example, "park my car in this tight spot, and do it quickly" would be a test of such a system. General purpose knowledge would allow the machine to behave sensibly whether it is a bus, a helicopter, or a bulldozer. Some of these will be presented as examples later in this thesis.

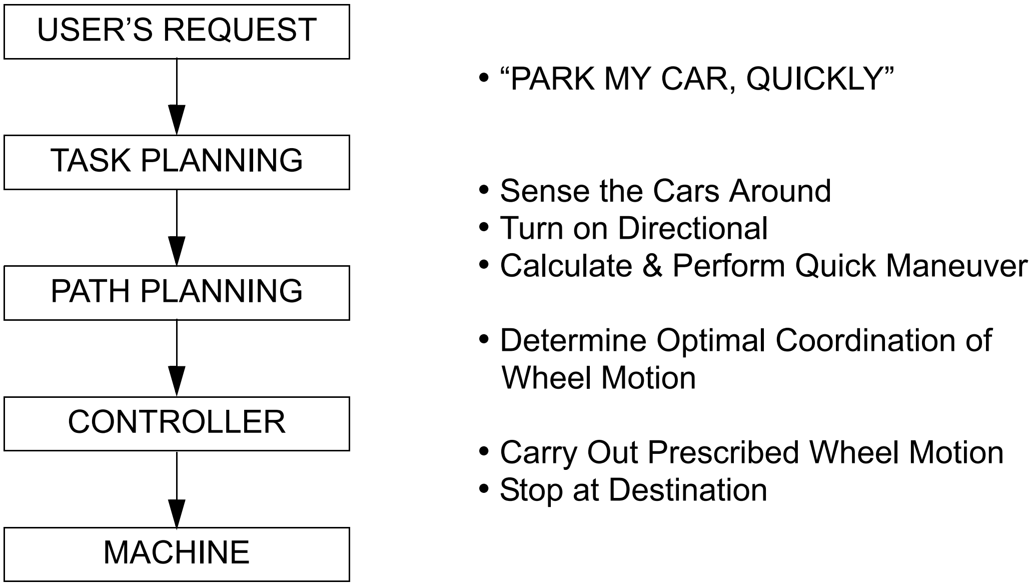


Figure 1: Framework for an Autonomous System.

The vision of an autonomous system is assumed to have several conceptual layers. The objective of the system is to achieve a user's request efficiently. The input may be as sophisticated as speech, or be provided by another interface to the computer. The task planner helps refine the definition of the task based on prior knowledge including strategies for responding to similar requests. It breaks down the initial request into smaller tasks, including sensing tasks to acquire more information. Some of the subtasks require an optimal, planned motion to a desired ending location. The path planner computes the best path that avoids obstacles, and sends the request for desired actions to a controller. The controller adjusts the position of each of the actuators to achieve the action as prescribed. Ideally, there is a sensing system that gives information about the state of the environment and the system itself, to each of the levels on request. For the control level, it verifies that the action is occurring as specified, and uncovers any differences between the assumed and observed environment.

Research is being done on each layer with the hope that one day the vision for an autonomous system can be realized. The planning method presented in this thesis provides only one piece, a fast, globally optimal path planner. The other levels (task planning, sensing, control, etc.) are not the current focus.

Objectives of this thesis

The primary objective of the thesis is to define a new framework and methods applicable to planning. Emphasis is on demonstrating path planning applications for intelligent autonomous systems and demonstrating varied and flexible applications.

This thesis presents a modular framework for A* planning in discrete configuration spaces. The plans are illustrated in the domain of intelligent autonomous systems, where a machine such as a robot can detect the surrounding world, have an understanding of the objective (goal), and determine the appropriate actions based on its own limitations. The fundamental motion of a particular mechanism can be encapsulated in what is introduced as a *neighborhood*. Example neighborhoods define the unique form of motion for a variety of mechanisms such as a mobile robot, robot arm, car, and tractor, based on their inherent limitations. The framework then uses the neighborhood, an optimality criterion (e.g. minimize distance) and specification of the goal to select a specific optimal motion based on the range of collision free configurations of the mechanism. The selection is based on an A* search method from the field of Artificial Intelligence (AI).

The thesis also introduces 'Differential A*', a new graph method that quickly adapts existing A* plans to changes. When applied to autonomous systems, as previously unforeseen obstacles appear in the environment, for example, the plan is adjusted to account for their affect in the new environment. This allows optimal motion to resume quickly. This fast replanning can improve robot reaction time and productivity.

Finally, the thesis introduces 'rendezvous planning' a direction of research that extends the basic framework into the area of task planning. The objective of rendezvous planning is to coordinate the actions of multiple machines or 'actors'. An initial description of this method and a simple application are described briefly.

1.2 Approaches to Planning

Two factors are involved in path planning. The first is the overhead required to find some path, or the optimal path. The second is the effort required to follow the path. These factors determine whether or not planning is worthwhile. If an activity regularly requires travel from a given origin to a destination, then the gain achieved from planning (ideally optimal) paths is clear. On the other hand if a trip will only occur once, and the overhead of planning is vast, then finding any reasonable path may suffice. Planning methods vary in their level of overhead, and must be chosen in balance with the gain.

There are many methods for task planning, including commercial software such as Microsoft Project [27] for project planning and scheduling. This type of software provides task plans that assign time ranges and costs, based on user input of resource requirements and task dependencies. Time frames can be adjusted to fit the resources, or resources defined to fit the time frame. It can highlight the series of critical tasks, which affect the total time from start to finish. While this is primarily targeted at commercial task planning, it must still rely on fundamental search algorithms and hierarchical graphs to compute results such as the critical path.

More traditional motion plans include schemes to escape a maze. As complicated as a maze seems, there is typically only one solution path, requiring no optimization criterion. A maze is constructed with no ‘free space’, that is, it has constraints such that arbitrary travel through a portion is prohibited, and can be hand crafted to have a high branching factor from the start to the goal and designed to cause heuristics to lead down erroneous paths. We can solve mazes using the ‘right hand rule’, where by walking along the maze with one hand on the wall, the exit will always be found. Unfortunately this involves walking up and down many dead ends (i.e. backtracking). We learn early on to find the path from the opposite direction (i.e. the goal) because it typically has a smaller branching factor.

An overview of robot path planning methods is given by Latombe [21]. These plans determine a motion for the robot. Artificial potential fields described by Khatib [20] are often used at the robot planning level. While typically used to plan in the task space (real world) environment, they can also be used to plan based on the set of possible joint angle parameters (i.e. the configuration space) of the robot. They are analogous to electric fields, whereby there is an attraction ‘pole’ at the goal and repelling forces at the obstacles. The sum of the forces gives a potential field used by the controller to direct the actions of a given machine toward the goal. This is a relatively fast calculation to provide a safe path around obstacles. One problem with the method is that the artificial potential fields produce paths that tend to follow medial boundaries between obstacles. While safe, these plans are not optimal. Another problem is that artificial potential fields can produce *local minima*, where the attraction of the pole is insufficient to provide a path from around a cluster of obstacles. The result is that the attractor must be adjusted. If a path is not possible, potential fields do not identify this ahead of time. A controlled machine follows the path until it stops at a local minimum. Several other approaches that target robotic applications are discussed in Section 3.7.

Another approach is to compute a global distance map, that for every location, gives the distance to a goal. By examining the surrounding locations, the location with the least distance (i.e. steepest gradient) is the one to follow toward the goal. Distance maps can be computed with the Constraint Distance Transform (CDT) of Dorst and Verbeek [13] to provide globally optimal paths with a localized template that is repeatedly swept over the space. Every location in the space gives the cost to the nearest goal and a pointer to the next node in the sequence leading to that goal. The computation of the distance map is relatively inefficient (i.e. $O(N^2)$), since for each of the N states, the template may be swept N times. A fundamental limitation of the distance map computed with the CDT is that it is restricted to the Euclidean (straight line) measure of optimality, and a regular pattern of connected nodes. Although a hardware implementation of this inherently parallelizable method would improve the inefficiency, the other limitations remain.

As an alternative to recomputing the entire distance map for each change, a method developed by Boult [5] exists to recompute only the portion of the distance map affected by the change. In many examples, this is less than the entire map. The method relies on the CDT method in a two dimensional distance map using the Euclidean cost criterion. It adapts the map to the introduction of new polygonal obstacles, which is achieved by determining the area that is affected by the change and then updating it. This limits the number of locations that must be recomputed. Even with this improvement, the method suffers from the same limitations as the Constraint Distance Transform.

Using a Hopfield neural network rather than a standard von Neuman architecture, reasonably reliable paths have been computed by Cavalieri, Di Stefano and Mirabella [7] between pairs of nodes in a graph. The network was arranged so that the transitions in and out of N nodes were each represented as neurons. Therefore the complete graph is represented by connecting each transition into a given node to the N-1 other outgoing transitions (neurons). The graph is topologically characterized by the presence of feedback between each pair of neurons. The network parameters are tuned until convergence to an optimal path is achieved. While the work in neural networks remains an interesting and potentially powerful method and architecture, this thesis will be focused on techniques for conventional von Neuman computers.

The A* approach proposed in this thesis can solve the maze problem at least as quickly because the method does not require backtracking, a relatively slow process. It can also provide solutions for higher dimensional maze problems than the classic 2-D maze. Compared to the artificial potential field technique, the A* approach provides globally optimal paths, always finds a path if one exists, and is never trapped in local minima. Optimality can be defined according to a variety of possible cost measures (minimum time, energy, distance etc.) rather than only the minimum Euclidean distance. The constrained distance transform is limited to the type of cost measures that can be computed and it is much less efficient than A* in terms of the time required for computation. However, the method for updating distance maps suffers from the same problems as the CDT, and is restricted to polygonal objects in two dimensions.

1.3 Overview of the Thesis

We begin in Chapter 2 with a discussion of the A* graph search algorithm, on which the remainder of the thesis is based. A comparison to other standard search algorithms helps highlight the strength of the method and place it into context with them. In reviewing the traditional A* algorithm, several specific improvements are made to the performance for later implementations. A detailed analysis of the time complexity of the algorithm helps identify the parameters that contribute to the total time spent computing, and is used in later chapters.

Chapter 3 defines a *framework* to apply the A* graph search algorithm to path planning in autonomous systems. A configuration space is used to describe the set of all configurations of the system, so that with a single point the status is known. The definition of the problem is transformed into the configuration space, solved, and then transformed back to the task at hand. The framework provides a modular way to identify the relevant components of the problem so that they are transformed properly to the new space and so that the components become easily interchanged to address new problems. For example, the overall behavior of a machine is based on its *neighborhood* of permissible motions and the machine's objective defined by an *optimality criterion*. Because the type of motion and measure of optimality are separated, they can be selected specifically for the application. So, the behavior for a given robot arm to move quickly (minimum time), can be switched to a motion which is graceful (minimum effort), by straightforward substitution of a new cost measure. Different cost measures are illustrated on an actual robot arm in several obstacle strewn environments to show the resulting optimal behavior.

Several aspects of the planner's design are also analyzed. Typically the environment to be modeled resides in the continuous space of the 'natural world'. The internal representation within the computer however, casts this into a discretized space. The issues surrounding the fineness of the discretization versus the quality of the produced paths are presented and briefly analyzed. The time complexity of Chapter 2 is used to characterize the computation time for the robot application. This gives insight into the variables that most affect the performance of the planner.

Chapter 4 applies the framework to a more complex problem, autonomous vehicle maneuvering. The complexity arises because the vehicle (a car) is constrained to move outside the limits defined by the turning radius (i.e. the car cannot move laterally). This complicates the planning process because these (kinematic) restrictions must be considered. Without them, we would immediately discover that the vehicle cannot follow any arbitrary line around obstacles. The paths generated from the planner are versatile, since from every starting position the vehicle determines automatically how to maneuver optimally to reach the goal. For a goal residing between two parked cars, the vehicle's behavior is to parallel park. If the environment has many obstacles, then the maneuver is much more complex, but is computed using the same systematic planning framework, with equivalent ease. Similar to Chapter 3, the time complexity is given for this type of calculation using the analysis of Chapter 2.

The true validation of the plan occurs when an instrumented vehicle is controlled to follow the path. The motion plan is converted so that appropriate pieces of the plan (setpoints) are provided to a controller. This controller must be implemented so that it appropriately coordinates the wheel motion to move the vehicle to the setpoint.

Chapter 5 introduces a new method called Differential A* which adapts graphs based on changes in the nodes or transitions. The objective is to recompute only the required portions of a graph based on the changes, rather than the entire graph (using A*). These changes affect the nodes surrounding the change, and the chain of nodes dependent on them. For each individual case, describing a type of change, the repercussions of the changes are analyzed so that only affected portions of the graph are recomputed. Each case has an associated set of actions required to correctly identify and recompute the affected areas. The effect of each of the actions is analyzed in terms of the factors contributing to the computation time. By identifying the relevant cases and combining the actions for a specific application, an algorithm can be defined, with an estimate of the computation time.

Chapter 6 illustrates the Differential A* algorithm in the dynamic environment of robotics. Robotic environments are rarely static. Obstacles move through the workspace, and goals vary from task to task. Depending upon the sensing system, some unexpected obstacles may be sensed. In these situation, the ability to replan quickly and intelligently (i.e. optimally) improves the overall response and quality of the system. The complexity of this type of environment provides a way to challenge and measure the Differential A* method. This chapter defines a specific algorithm based on relevant cases for robotics, and tests it with a number of representative changes in the environment. Finally, the method is compared to A* to characterize the comparative advantages.

Chapter 7 is a discussion of the key points that are highlights of this thesis, with a view toward future work and identification of unsolved problems.

Chapter 2

Background: A* Search

2.1 Introduction

It has been said that 80% of the world's computing consists of searching and sorting. They are clearly the staples of computing, giving rise to numerous techniques to achieve each. A* is an efficient method for searching through graphs. To appreciate the elegance of the A* algorithm, we must discuss other standard methods, their limitations and application. One objective of the search methods is to find a path between two nodes in a graph, giving the transitions required between intermediate nodes.

There are may types of search, but they fall primarily into two categories: uninformed and informed. Uninformed searches perform fast, but rote calculations to find the path. They can result in exhaustive searches, with high inefficiency and duplicated effort. These methods are best used when the graph size is small. Informed searches make use of characteristics of the problem so that fewer parts of the graph are searched. These searches can also be grouped by the way decisions are made. The next branch to be explored can be determined by the current view from a node in the graph or based on the best of all the nodes.

Four very common search methods are: Depth First Search, Breadth First Search, Hill Climbing, and Branch and Bound. Each of the methods produce *search trees* from the graphs such that for each explored node, the subsequent reachable nodes are added to the next level (L) of the tree. If a node is encountered that has already been *visited*, it is not added so that search cycles (infinite loops) are not generated. The decision about the sequence in which to produce the tree is governed by the specific search method. Each of these methods will be reviewed briefly for completeness, but can be examined in detail in many texts [30,63].

2.1.1 Depth First Search (DFS)

This is an uninformed search method to traverse the vertices of a graph. At each vertex, an unexplored successor (vertex directly connected to the current vertex via an edge) is identified and added to the search tree. When there are no unexplored successors, alternatives are identified by backtracking up the tree, until one is found. This continues until the desired vertex is found, or all vertices are explored. DFS is used for solving mazes. It uses the 'right hand rule' to find a way out of the maze; put your right hand on the entrance wall of the maze and continue until the exit. The backtracking occurs automatically at each dead end of the maze.

While this method will eventually find a solution if one exists, it is inefficient. If there are V vertices in a graph, and E edges, then each of the V vertices must be initialized, and then (worst case) each of the edges must be explored. Therefore the running time of the algorithm is O(V+E).

2.1.2 Breadth First Search

This is also an uninformed search method. At each vertex, all successors are identified for each level of the tree in sequence. This is achieved by adding all successors of the starting vertex to a list and then adding the successors of each of the listed nodes until the list is empty. Therefore, the tree is built uniformly from the nearest vertices to the farthest.

The running time of the algorithm is O(V+E), because each of the vertices are initialized as in the DFS method, and (worst case) each edge is explored.

2.1.3 Hill Climbing

This is an informed search method that uses a single viewpoint at a time, and is an improvement on the Depth First method. It orders the transitions such that the most promising successor is added to the tree next. The successors are measured against one another, using a cost estimate of the likelihood that it will be part of the best path, and the most promising successor becomes the origin of the next search step.

This method has many limitations, because it suffers from premature celebration. The maximum may appear to have been reached, when in fact it has not. Several situations cause this: the *foothill problem* (local maxima) may be reached where the search has lead to what appears to be a maximum in the shadow of the true maximum; a *ridge problem* prevents exploration in the true direction because the selection of successors is restricted (i.e. considering only north, west, south and east, when northeast leads to the maximum); and a *plateau problem*, where a single step in any direction yields no clue as to the proper direction, yet there are non-local, but major improvements available in unpredictable locations.

Hill climbing also has difficulty in managing a search when the problem space has many dimensions, because the number and sensible evaluation of the successors can be complex. However higher dimensions are difficult for most other searches as well.

2.1.4 Branch and Bound

Branch and bound is an informed search method based on a viewpoint of all nodes, and can be classified as a *best first* method. It finds the optimal path by always adding to the shortest path so far. This results in many incomplete paths (i.e. paths stopped midway), but concludes with one complete path. An optimal path is guaranteed when the shortest of all the incomplete paths is greater or equal to the one including the destination. For example, in Figure 2, the path arriving at Intermediate node B will cause the path, in the next step, to reach the Destination with cost 109. Clearly though, this should not be considered the optimal path until the incomplete paths costing less than 109 have been extended to at least 109.

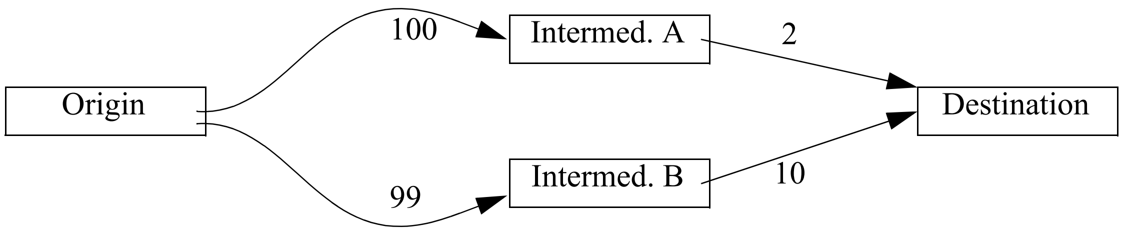


Figure 2: Branch and Bound Termination

This branch and bound technique can be improved by using an underestimate of path length (equal to the distance already traveled plus an underestimate of distance remaining) as the cost at each node rather than only distance already traveled [63]. This will guide the path search to reduce the number of nodes evaluated, while still generating the optimal path. The optimal path can be assured because there cannot be a completed non-optimal path which has a lower cost than an underestimate of an incomplete optimal path. The best case is when only the nodes on the optimal path are opened, which can occur if the estimate is precisely the distance remaining.

2.2 Best First Searches for Optimal Paths, A*

Best first searches are now further defined and focused to pertain to searches from one or more alternative starting nodes toward a goal state in the class of OR graphs. Problems often use OR graphs to describe alternative approaches to solve the problem vertex from which they are derived. So for example, in Figure 2, the alternatives are to move from the origin to the destination via Intermediate node A OR Intermediate node B. This is in contrast to AND graphs which describe the subdivision of a problem into subproblems, where for instance, both A and B might be required to arrive at the Destination. A* is used for OR graphs only.

The search begins with the starting nodes by expanding successor nodes that are selected based upon their likely relevance to a final path. The selection process takes into account *heuristic* rules or approximations that focus the search, minimizing the number of expanded nodes. The final result is an *explicated graph*. This graph identifies the optimal path(s) by specifying optimal transitions between successive nodes such that the overall path cost is minimized. Commonly, the path is shown as a sequence of pointers, or a *pointer path*. In the event that no goal node is reached when all branches of the search are exhausted, then there is no solution that connects the start to the goal. If the graph is finite, the algorithm halts and reports the failure. If the graph is infinite, the algorithm may not terminate.

The Best First class of algorithms can be essentially classified based upon two features, the recursive weight computation (RWC), and delayed termination (DT). The recursive weight computation assures that each node is a recursive function of previous costs. This therefore assumes that each subsequent state depends on data from its *parent* node, itself, and a function that characterizes the cost of the transition. Therefore finding the total distance so far is a recursive operation requiring addition. Delayed termination ensures that the final result is not reported at the moment there is a connection between the start and goal, but rather waits until it can be assured that no other node could improve upon this solution. Pearl [34] suggests that the program wait until the expansion of the goal node occurs, but it may be preferable to wait until the nodes expanded have an actual cost greater than the goal. Therefore, special names are given to the searches that have one feature but not the other.

Pearl includes a useful hierarchical diagram ([34] pg. 63) that includes the relationships between nine best-first algorithms, including those searching on generalized AND/OR graphs. Only the five that relate to OR graphs are shown in Figure 3.

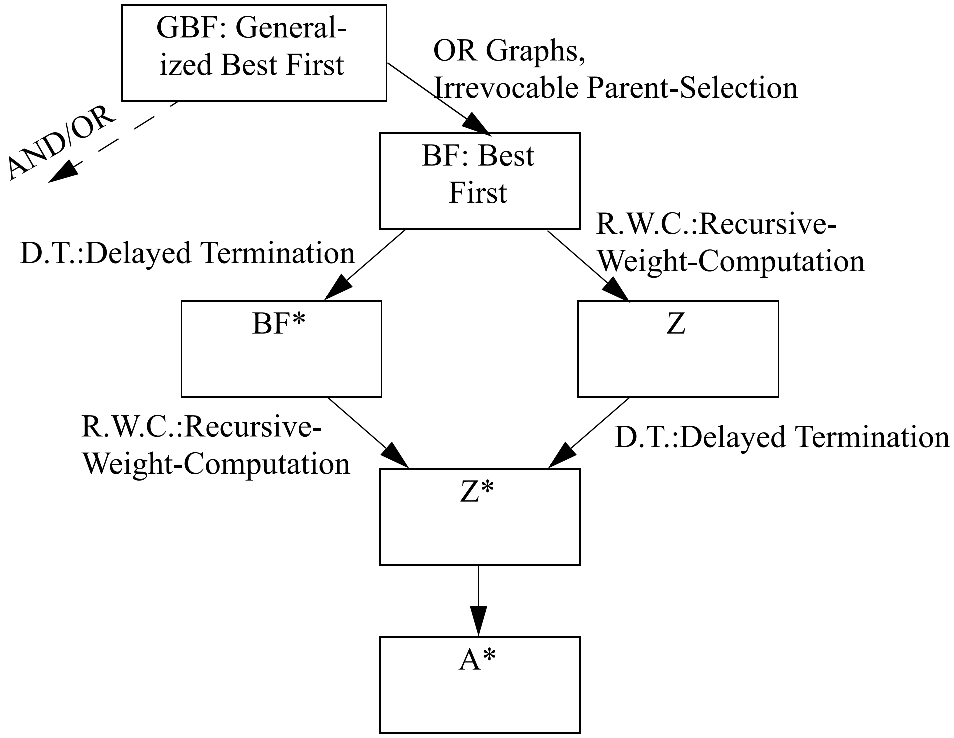


Figure 3: Pearl's Hierarchical Diagram.

Best First * (BF*) computes DT, but does not assume that the weights can be recursively computed (RWC). Z assumes RWC, but not that Delayed Termination (DT) is needed. Both features taken together result in Z*. Z* does not assume that the cost is dependent on any particular restriction on the cost measure. A* is a specific version of Z*, where an arbitrary, but recursively computed cost function can be used. This means that rather than having a simple additive formula such as total_cost = previous_cost + transition_cost, for the actual costs so far, the formula might include references to other nodes than the locally adjacent nodes and transitions as well as more complex mathematics.

2.2.1 Admissible Heuristics

Nilsson [31] proved that using heuristics to guide the A* search can result in an optimal solution, if the heuristic function is *admissible*. The heuristic is admissible if it is an optimistic value, lower or equal to the actual cost between the current node n and the goal. A heuristic h_1 is considered to be *more informed* than a heuristic h_2 if both are admissible and $h_1 > h_2$. This causes less of the graph to be searched, and the solution to be found more immediately. The properties of admissible heuristics have already been defined and proven by others [31,34], and will be used here primarily for applications.

2.3 A* in Detail

The A* algorithm [34] is reviewed here because the algorithm, its limitation and some of the contemporary notation, will be used in the description of Differential A* in Chapter 5. It also is important to understand the standard method, as extensive applications are described in the remainder of the thesis. For those already familiar with A*, it may be best to proceed to Section 2.5 where the algorithm is analyzed in detail.

The A* algorithm will be described along with a simple travel example. The example is relevant to many Philips Research employees who travel from the New York area to the corporate headquarters in Eindhoven, The Netherlands.

Graphs contain nodes and transitions as shown in Figure 4. The graph shown here connects a subset of the cities for flights between New York and Eindhoven, the Netherlands (NL). The dashed-line transitions show the possible flights that connect the cities. In graphs treated by A*, there must be a finite number of nodes and transitions. This therefore excludes problems with continuous or infinite bounds. In many instances, problems have finite limitations or can be quantized into discrete value ranges, making A* a practical method applicable to a wide range of problems.

2.3.1 A* Algorithm

By examining the differences between the branch and bound (B&B) method of Section 2.1.4 and the A* method, the notion of OPEN and CLOSED sets are new. OPEN refers to the set of terminating nodes in the search tree of the B&B method at any point in the search. CLOSED refers to nodes that were on OPEN, but have been expanded (i.e. the lowest cost path at some point in the search). Also different is the specific notation and definition for:

$c(n,n')$: the transition cost from a node n to a successor, n',
$g(n)$: the actual cost from the start to the current node n, where $g(n') = g(n') + c(n,n')$,
$h(n')$: the heuristic (underestimate) of the cost from n' to the goal node of the search, and
$f(n')$: the total of $g(n') + h(n')$.

The generally accepted A* algorithm steps, taken from Pearl [34] (page 64), are shown below. For completeness, the A* definitions are included in Appendix A.

A* Algorithm Steps

1. Put the start node s on OPEN.
2. If OPEN is empty, exit with failure.
3. Remove from OPEN and place on CLOSED a node n for which f(n) is minimum.
4. IF n is a goal node, exit successfully with the solution obtained by tracing back the pointers from n to s.
5. Otherwise expand n, generating all its successors, and attach to them pointers back to n. For every successor n' of n:
   - a. If n' is not already on OPEN or CLOSED, estimate h(n') (an estimate of the cost of the best path from n' to some goal node), and calculate f(n') = g(n') + h(n') where g(n') = g(n) + c(n,n') and g(s) = 0.
   - b. If n' is already on OPEN or CLOSED, direct its pointers along the path yielding the lowest g(n').
   - c. If n' required pointer adjustment and was found on CLOSED, reopen it.
6. Go to step 2.

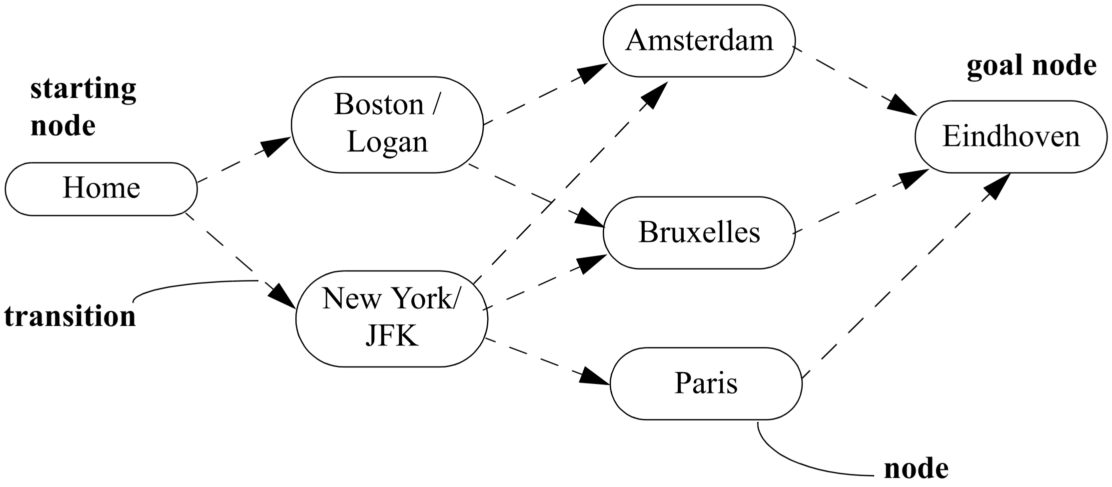


Figure 4: Graph Notation.

2.3.2 A* Related to a Travel Example

A* graphs are characterized by having a single start node which originates the search toward one goal γ of possibly many goal nodes in a set Γ. The nodes and transitions form the general graph, denoted by G_0. Nodes may have *successors*, that is, transitions that are possible directly from the node to other nodes. In Figure 4, the starting state may be considered as the Home node and the goal as Eindhoven, NL.

Successors are the nodes in G that can be reached immediately from the current node. So in Figure 5, the possible successors of the Home node are Boston and New York. The successors of the New York node are Amsterdam, Bruxelles, and Paris, etc. For each successor n' of n, the cost g(n') that leads from n to n' is the actual cost g(n) plus the transition cost c(n,n') between n and n'. So in A*, g(n') = g(n) + c(n,n').

Every transition from a node n to a node n' has a cost c(n,n'). In A*, these transitions must have a positive cost (i.e. greater than zero). In addition, a *heuristic* must be provided to guide the search in an efficient manner. This must be an optimistic estimate of the cost from a given node to the final node. The heuristic should always be optimistic, that is, a lower bound on the actual cost. If it is too low, for example zero, then the resulting search will not be guided well, and more nodes will be opened than necessary. If the heuristic is zero, then all nodes with costs less than the goal node will be opened. The more accurate the heuristic is at predicting the optimal path, the fewer nodes will be opened. If the heuristic is incorrect, that is, if it estimates a higher cost than reality, then it may cause the search to traverse nodes in improper order, possibly requiring re-evaluation of the same node several times. It may also result in an incorrect solution, for example if the search is pulled in the opposite direction from the correct one.

In Figure 5, the objective is to find the minimum time path between Home and Eindhoven. The transitions have costs that represent the number of hours to traverse the distance. Therefore the time may be affected by the mode of transportation taken (i.e. Home to Boston is 6 hours by car). The heuristic might be an estimation of the time using the air distance in ‘statute miles’ using the fastest possible transportation (i.e. a fast passenger plane with no delays) at 1000 km/h. Based on this, an estimate for the fastest transportation time from a node to Eindhoven is given in italics above that node.

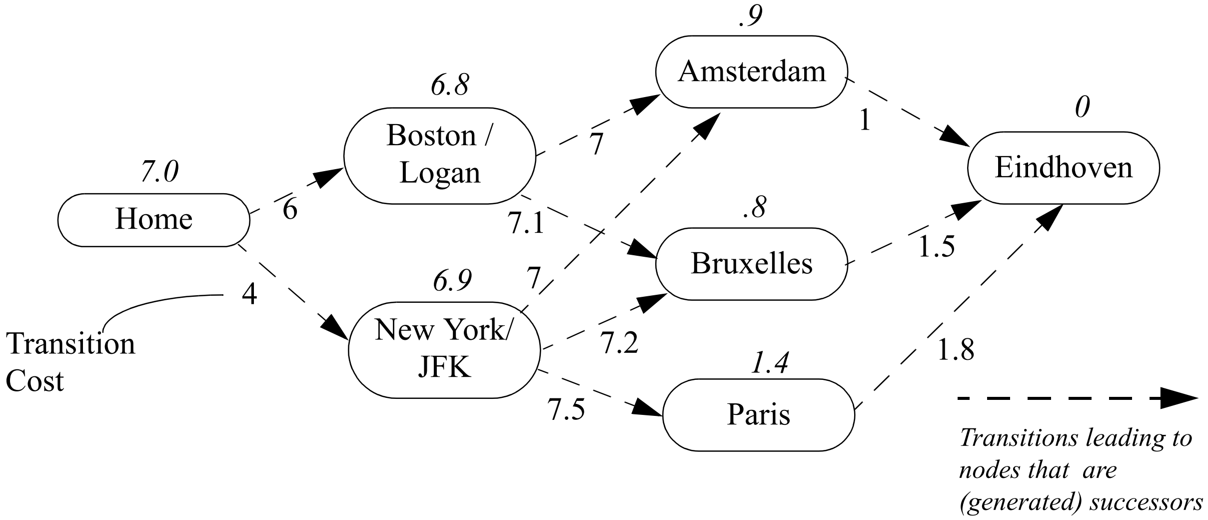


Figure 5: Transition Costs and Heuristics.

For every node during the search, there are two components: the cost of the search path from the start to the current node n, g(n), and a heuristic estimate of the cost from the current node to the goal node, h(n). If n is the starting node, then g(n) = 0. Each node has a total cost f(n) = g(n) + h(n).

The search begins with the home node. The starting cost of g(Home) = 0, and h(Home) = 7. Therefore f(Home) = g(Home) + h(Home) = 7. The home node, being the lowest f() (and only) node in OPEN is then expanded to the New York and Boston successors. The costs are:

f(New York) = g(New York) + h(New York),
g(New York) = g(Home) + c(Home, New York) = 0 + 4 = 4.

Since h(New York) = 6.9, f(New York) = 10.9, and a pointer is created from New York to Home, and New York is added to OPEN. Similarly, f(Boston) = 12.8 and a pointer is created to Home and added to OPEN.

Since New York is now the lowest f(), it is expanded next. It results in the evaluation of Amsterdam, Bruxelles and Paris as:

f(Amsterdam) = g(New York) + c(New York,Amsterdam) + h(Amsterdam) = 4 + 7 + .9 = 11.9.
f(Bruxelles) = g(New York) + c(New York, Bruxelles) + h(Bruxelles) = 4 + 7.2 + .8 = 12.
f(Paris) = g(New York) + c(New York,Paris) + h(Paris) = 4 + 7.5 + 1.4 = 12.9.

Each of these three successor nodes is added to OPEN and pointers point back to New York.

Of the four nodes now in OPEN, Amsterdam has the lowest f() at 11.9. Once expanded, it reaches Eindhoven giving f(Eindhoven) = g(Amsterdam) + c(Amsterdam,Eindhoven) + h(Eindhoven) = 11 + 1 + 0 = 12. This is therefore a path that may be time optimal between Home and Eindhoven. In this case, the other nodes still in OPEN have a minimum f() value of 12. This means that the best case for all other nodes in the search is to result in a cost that is 12. Therefore the search may terminate. If the minimum f() was less than 12, then the search should continue because another path could be less costly.

Many nodes and costs were never evaluated, and because the heuristic is *admissible*, they would never have resulted in a better solution. The transitions that were evaluated and resulting search traversal is given in Figure 6. The solid arrows indicate the optimal path that can be taken between the start and a selected node by following the arrows and then reversing them. The dashed arrows indicate the possible transitions. In Figure 7 the *explicated graph* arising from the traversal can be seen more clearly. Note that although from several nodes there is a pointer leading back to the Home node, this path is not guaranteed to be optimal since the heuristic used was intended for a different goal node (namely Eindhoven).

In addition, a delay cost for being in n may be incurred either as part of g(n) or c(n,n') depending upon the application. If there is a layover associated with the waiting time between the New York to Amsterdam and Amsterdam to Eindhoven flights, then the delay will more naturally be incurred as part of the c(n,n') cost. This makes the measure of optimality for the problem time, rather than specifically flight time. If the time for traveling through passport control is added, then this would more naturally be added to the g(n) cost associated with the New York to Amsterdam flight. Finally, (travel) delays can be incorporated into the graph itself as an interim node and cost between two others. This final case is not how the term delay will be interpreted for the rest of the text because this case is treated directly in any computations.

As stated in Pearl ([34] pg. 74) “The A* explores the state-space graph G using a traversal tree T. At any stage of the search, T is defined by the pointers that A* assigns to the nodes generated, with the branches in T directed opposite to the pointers.” From this we have the notion of a set of pointers leading from the goal node toward the starting node of the search since the transitions are opposite the branches (see Figure 6 below). To find a path between the starting node to any other node selected as a goal node, pointers are traced node to node from goal to start. This path then must be reversed to form the path.

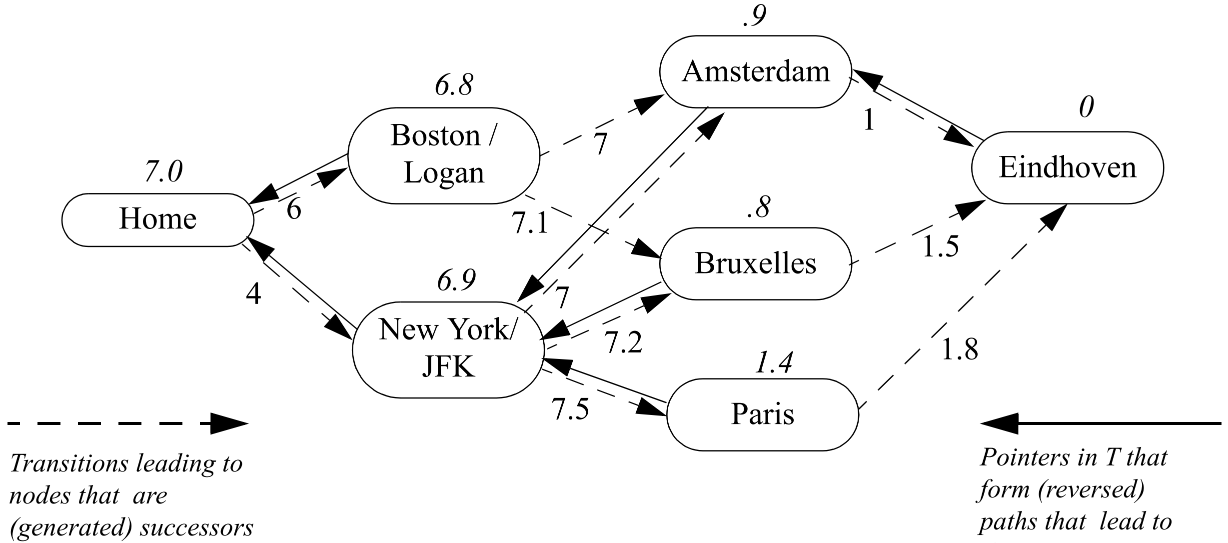


Figure 6: Transitions and Paths.

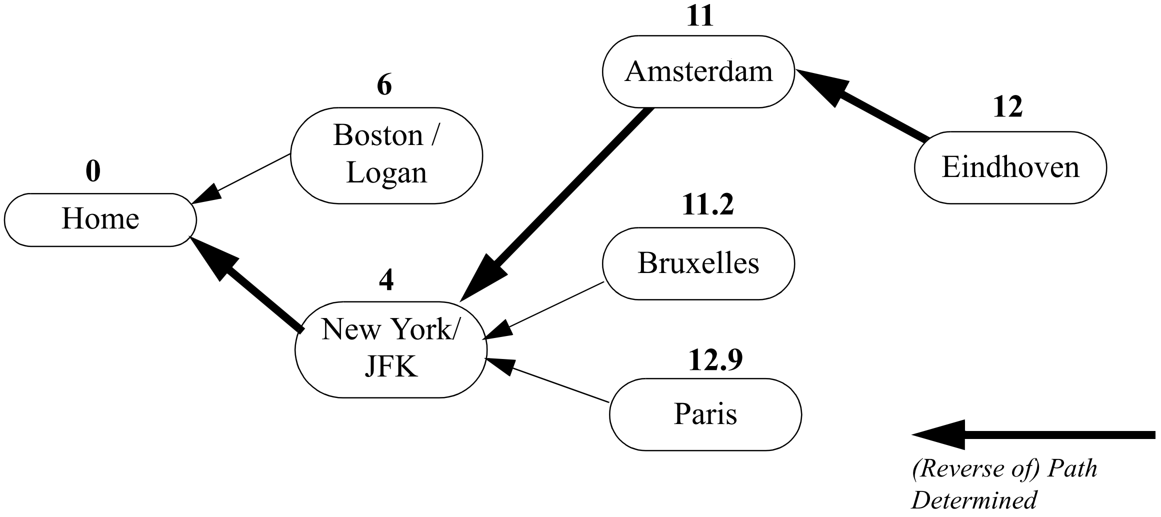


Figure 7: Explicated Graph G_e with Actual Costs g(n)

The explicated subgraph G_e is the union of all of the branches in the traversal tree. If the search were to retain alternative but cost-equivalent pointers, then the tree may become a graph. This did not occur in the previous simple example, but it is likely when traversing a grid of Manhattan city streets. For example it may be equivalent to drive north two blocks and east one block vs. east one and north two. Therefore at the far corner, there are two equivalent alternatives both leading to the origin.

2.4 An Improved Implementation of A*

Two main factors affect the performance of the A* algorithm. The first is the choice of heuristic. The second is the efficiency of implementation of the A* algorithm itself. The heuristic should be selected based on the consideration of the cost of its implementation versus the gain expected in improving the search. Because our improved implementation of the A* algorithm will be used and contrasted later, a specific implementation is described next. First, programmatically precise data structures are proposed and some of the frequently cited structures used by Pearl [34] are slightly adapted. Then an algorithm using these new structures is described and analyzed for its performance.

Another variation of A* should also be mentioned here, A^*_ε , (said A - epsilon - star). This improvement [33] presumes that the selection process (of the minimum cost node) is the most time consuming part of the calculation. It computes several nodes at a time (i.e. those with costs no greater than a factor of $1+\varepsilon$ ($\varepsilon \geq 0$) of the best node. This causes the nodes to be calculated possibly out of turn. Although the resulting path is not guaranteed to be optimal, it attacks a potential difficulty in the selection process. This thesis will not rely on this variation.

2.4.1 OPEN

OPEN is not specified to be a list, tree, or other structure in Pearl. Here the selection of the best structure is investigated. The OPEN object has three functions associated with it, namely node insertion, node removal and lowest-cost node identification. The identification requires that the list must be searched regularly, or preferably, remain sorted. As such, the number of comparisons will be analyzed to evaluate the performance of several structural choices.

If OPEN is stored as a list, then each insertion will require a fixed time of O(1), assuming that the element is added to one end of the list. If this is the case, then the removal of the lowest cost node is O(N) since each node must be searched to find the lowest cost node. Since this is performed N times, then each node will require total processing of $O(N^2)$. Alternatively, if the insertion occurs by placing the element into its proper linear ordering sequence, then the time required is O(N) and the removal from the head of the list is O(1). In this case, the total processing is again $O(N^2)$.

A better structure selection is the heap [39], since it is not necessary for the elements to be stored in sorted order, only for the lowest value item to be accessible at any time. This structure is defined as a complete binary tree having a lower (or equal) value at each node than either of the node’s children (if they exist). It requires $O(\log_2 N)$ for insertion and $O(\log_2 N)$ for removal. In practice, since the nodes increase in cost as the A* search progresses, the $O(\log_2 N)$ for insertion (along the bottom of the tree) is likely to be a coarse upper bound. Taken together, the total time for the algorithm is $2N\log_2 N = O(N\log_2 N)$ for N elements.

Further, the sorting structure is best stored as an *indirect heap*, namely storing only pointers to nodes rather than nodes themselves. This saves time in copying the nodes, as well as space in the storage of the nodes.

Finally, the structure should be *sifted*, that is, no node should be entered into the heap more than once. This can be accomplished by storing a flag in the node itself which denotes whether the node is in the OPEN heap.

One problem with the sifted heap is that when an indirectly addressed node is added to the heap and its value is later changed by a subsequent interaction, there must be a mechanism to ensure that the heap position of the indirectly addressed node is updated. Depending upon the implementation, the efficiency in time required to ensure the absolute correctness if the heap may be greater than the cost of recalculating erroneous nodes. This will be discussed further in Section 3.6.3.

2.4.2 CLOSED

CLOSED is used in step 5 of the A* algorithm with OPEN (steps a and b) such that to determine if a node is in OPEN or CLOSED, both sets must be searched. The notion of maintaining a structure named CLOSED does not seem to contribute much to the search other than to help identify nodes that have been opened in the past.

2.4.3 *Uninitialized* Nodes

The same information can be determined by maintaining a flag in the node that gives this information. If the node would have been on neither list, it is 'Uninitialized', otherwise it is on OPEN. When the node is created it is identified as *Uninitialized* or U. This eliminates the need for maintaining and searching the CLOSED structure. Once again a trade-off for time savings is made in exchange for a small amount of space.

2.4.4 Resulting Algorithm

Based upon the structures and assumptions given above an implementation is created for A*. A pseudo-code description is given in Figure 8. Because the core of the computation is in maintaining the OPEN heap structure, we might expect the algorithm to be approximately $O(N\log_2 N)$. But other factors such as the average number of successors play a role in more precisely defining the performance of the algorithm. The algorithm is next analyzed to give a precise formula for the performance, which will verify the initial intuition.

2.5 Complexity

The above described A* algorithm will first be analyzed to determine a mathematical expression for the total time of the algorithm with a collection of variables characterizing the timing function. From this, the key variables will be extracted and described in comparison with conventional big-O notation.

2.5.1 Detailed Analysis

Using the modified Pseudo-Code of Figure 8, the costs that are incurred in each portion of the algorithm can be described. The total time for the algorithm to run is the cost incurred by removing each node from the heap, then for each successor node, computing the transition cost, heuristic cost and actual cost and performing comparison to the other successor's costs. Those successors with lowered costs are then added to the heap. To express the time required for the algorithm, several assumptions are made, definitions are given, and then the algorithm is expressed mathematically.

```
Main()
    Put Starting nodes on OPEN

    While OPEN is not empty
        Remove the lowest cost node n from OPEN
            If the node n  is a goal node
                compute_path(n)
            Else expand_successors(n)
    endwhile
    exit(Failure)
End main()

Expand_successors(n)
    For each successor n' of n
        estimate h(n')
        compute g(n') = g(n) + c(n,n'), where g(starting node) = 0
        compute f(n') = g(n') + h(n')
        if n' is Uninitialized, or g(n') < the existing g(n') pointing elsewhere
            Direct the pointers of n' to n
            Add n' to OPEN

Compute_path(n)
    Trace pointers from n to s
    exit(success)
```

Figure 8: Modified A: Pseudo-Code.*

Definitions

| |	Number of elements in a set
start	Starting nodes
n(t)	Node opened at iteration t
newopen(t)	Function to return the set of successor nodes of the node having minimum cost f() at iteration t
$\Omega(t) = \lvert open(t) \rvert$	Number of nodes in OPEN at iteration t
$\lvert start \rvert = \lvert open(0) \rvert$	Number of starting nodes at iteration zero
$m(t) = \lvert newopen(t) \rvert$	Number of successors of the node opened at iteration t
γ	Characteristic growth factor. Example: average or max(m(t)) for all t
K_1	Overhead for calling remove( )
K_2	Overhead for calling add( )
M	Time to determine the m successors
H	Time to compute h(n')
C	Time to compute c(n,n')
G	Time to lookup g(n), flag checking, comparison of proposed and existing g(n'), directing pointers
b_1, b_2	Time for heap management

Assumptions

Each of the functions below is given with the expected cost in time (if relevant) and a brief explanation of the function.

remove(x):	$b_1 \log_2 x + K_1$	Time to remove an item from OPEN based upon the number of elements in a heap structure
add(x):	$b_2 \log_2 x + K_2$	Time to add an item to OPEN based upon the number of elements in the heap structure
heap constants:	$b_1 = b_2 = b$	Time for fundamental heap management operations is equal

The time required for this algorithm can be expressed mathematically by:

$$\text{Time} = \sum_{t=1}^{T} \left[\text{remove}(\Omega(t)) + m(t)\ (G+C+H) + \sum_{i=\Omega(t)}^{\Omega(t)-1+m(t)} \text{add}(i) \right]$$

The time for expanding T nodes is the time required to remove the low cost node, and then for each of the successors the costs must be calculated and then each of the cost-improved successors must be added to OPEN. In many cases, the number of newly opened nodes is the same and the typical case can be fixed:

$$\underset{t}{\text{mean}}\ (m(t)) = \gamma$$

If this is not the case, then m(t) must be left in place of γ.

Expanding the last sum,

$$\sum_{i=\Omega(t)}^{\Omega(t)-1+m(t)} \text{add}(i) = \sum_{i=\Omega(t)}^{\Omega(t)-1+\gamma} b \log_2(i) + K_2$$

$$= \sum_{i=\Omega(t)}^{\Omega(t)-1+\gamma} \left[b \log_2(i) + K_2 \right] = \gamma K_2 + b \sum_{i=\Omega(t)}^{\Omega(t)-1+\gamma} \log_2(i)$$

Assuming that γ is small compared to $\Omega(t)$, the last sum can be approximated by:

$$\sum_{i=\Omega(t)}^{\Omega(t)-1+\gamma} \log_2 i \cong \gamma \log_2 \Omega(t)\ ,$$

To check whether the assumptions hold for this approximate formula for the time, we have to gain insight into the error we are making.

This error, E, is given by:

$$E = b\left[\sum_{i=\Omega(t)}^{\Omega(t)+\gamma-1} \log_2 i - \gamma \log_2 \Omega(t)\right]$$

$$= b[\log_2([\Omega(t)+\gamma-1][\Omega(t)+\gamma-2]\ldots[\Omega(t)]) - \gamma\log_2\Omega(t)]$$

$$= b[\log_2(\Omega(t)^{\gamma} + \gamma\binom{\gamma}{1}\Omega(t)^{\gamma} + \ldots) - \log_2\Omega(t)^{\gamma}]$$

$$= b[\log_2(1+\gamma^2\,\Omega(t)^{-1} + \ldots)]$$

Discarding the higher negative order terms, the error in the first approximation is:

$$= b\log_2\left[1+\frac{\gamma^2}{\Omega(t)}\right]$$

$$E = \frac{b}{\log_e(2)} \bullet \frac{\gamma^2}{\Omega(t)}$$

The total time using the approximated sum is then:

$$\text{Time} = \sum_{t=1}^{T}\left[\text{remove}(\Omega(t)) + m(t)(G+C+H) + \gamma[K_2 + b\log_2\Omega(t)]\right]$$

Let us assume:

$m(t) = M$, a fixed upper bound on the # of successors, not depending on time or location

Recalling:

$\text{remove}(\Omega(t)) = b\log_2\Omega(t) + K_1$, for an insertion into a heap

$$\text{Time} = \sum_{t=1}^{T}\left[b\log_2\Omega(t)+K_1 + M(G+C+H) + \gamma[K_2 + b\log_2\Omega(t)]\right]$$

$$= b \sum_{t=1}^{T} \log_2 \Omega(t) + TK_1 + MT\,(G+C+H) + TK_2\gamma + \gamma\, b \sum_{t=1}^{T} \log_2 \Omega(t)$$

Assuming $K_1 = K_2 = K$,

$$\boxed{\text{Time} = (\gamma+1)\, b \sum_{t=1}^{T} \log_2 \Omega(t) + T(M(G+C+H) + K(\gamma+1))}$$

Formula 1

The percent error P is then given by:

$$P = \frac{E}{\text{remove}(\Omega(t)) + m(t)(G+C+H) + \gamma(K+b \log_2 \Omega(t))}$$

Assuming that the appropriate terms (K, m(t)) are small, we obtain:

$$P \cong \frac{E}{(\gamma+1)\, b \log_2 \Omega(t)}$$

$$P \cong \frac{b\gamma^2}{(\gamma+1)\, \Omega(t) \log_e(2)\, b \log_2\Omega(t)}$$

We get a percent error:

$$\boxed{P \cong \frac{\gamma^2}{(\gamma+1)\Omega(t)\bullet \log_e\Omega(t)}}$$

2.5.2 General Analysis

To compare this algorithm to other search algorithms, the order of the computation is typically given using ‘Big O notation’ [39] for the worst case behavior of an algorithm. Based on the more specific description of time given in Formula 1, assumptions about the worst case behavior of some variables can be given.

In the worst case, such as when the heuristic is zero (i.e. no guidance for the search), it may be that T approaches N, the total number of nodes. Also, if the earlier nodes open almost all the nodes, then it is possible that |open(t)| may approach N.

$$\text{Time} = (\gamma+1)\, b\, N \log_2 N + N(M(G+C+H) + K(\gamma+1))$$

Since the values K, M, G, C, H and γ are constants, then the algorithm is worst case, $O(N \log_2 N + N) = O(N \log_2 N)$ for large N. This illustrates how critical the size of the heap is to the performance of the algorithm.

2.6 Chapter Summary

In this chapter the conventional A* algorithm was discussed in the context of four other search methods namely depth first, breadth first, branch and bound and hill climbing. The A* algorithm was reviewed. This algorithm efficiently computes the globally optimal path planning solution. The search is guided by admissible heuristics which direct the search in the most likely direction of the solution, and reduces the amount of searching required. The notion of an *uninitialized* node and the specific use of a *sifted heap* structure for OPEN node management is introduced to improve the performance of the A* implementation. While viewed generally as a worst case time O(N log N) algorithm, the complexity is detailed and determines that the major time is attributable to the number of OPEN nodes. Specific time and error estimates are derived for an A* computation.

Chapter 3

Path Planning Framework Using an A* Search

3.1 Path Planning as Part of an Autonomous System

An autonomous system works at different levels of abstraction and at different speeds to produce a robust and seemingly intelligent mechanism. The levels are: task planning, path planning, reflex control, servo control, and at the lowest level the hardware. These levels are shown in Figure 9.

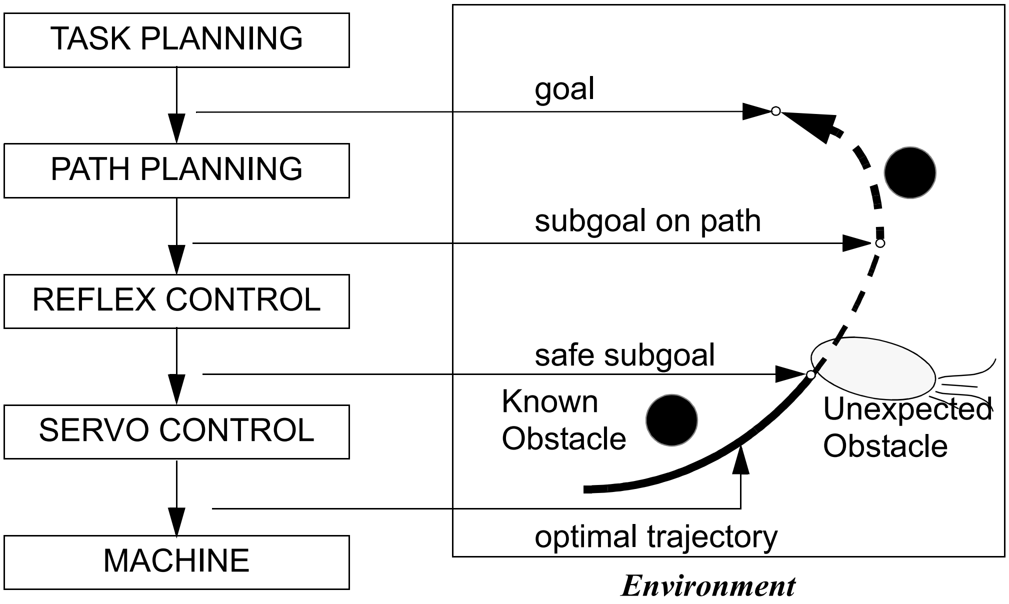


Figure 9: Framework for an Autonomous System.

The task planner determines the proper strategic moves for the task at hand. For instance it may determine strategies for a manufacturing process or identify the best move based on rules of a game. It generates goal states for the system without detailed concern for how they will be carried out. It can also identify alternative, but equally good goals, for example, picking from two different sources of the same part in an assembly. Naturally a good task planner should not regularly plan tasks that are impossible for the machine in the given environment. Typically, the task planner has the longest cycle time of the elements in the system.

The path planner uses these goal(s), and determines the optimal path from the current position to the goal position. The optimal path is the shortest according to a given criterion (time, distance, etc.), must avoid known forbidden regions (obstacles), and be executable based on the limitations of the machine (kinematics, control, etc.). The path must define the proper motion for each of the actuators and is broken down into *sub-goals* (*setpoints* in robotics). This element can be a bottleneck in some systems. It is not a requirement that the path planner operate in real-time, however in practice it can take longer to plan than carry out the motion. Therefore efficient planning methods are critical to the utility of an autonomous system.

The reflex controller is available in an ideal system. It uses the sub-goal and sensors to monitor and ensure safe, collision-free motion. This is an additional safety mechanism that is provided near the control level for faster response in the case where the sensors detect a sudden problem. If the motion is not safe, then it determines the nearest *safe sub-goal* along the trajectory of the path planner's sub-goal. In addition, it determines appropriate (smaller) control points for the servo controller. In the event that an unexpected obstacle is sensed, then the reflex controller can halt the motion (and initiate an exception handling step) or take localized corrective action in time by changing the control points to the servo controller. The reflex controller is not found on many machines, but is an improvement to a simpler 'control program' that directly divides sub-goals into control points. Other forms of reflex control can be used simply to identify major errors between the predicted control and the results and determine the proper response. This response may be only to stop and replan, report the problem to the path or task planner, or respond with an alternative control strategy.

The servo control uses the control points to find optimal (usually time) trajectories for the individual actuators (motors). Servo controllers are commonly available today in hardware with programmable attributes. The actuators move the machine hardware to the desired position.

Generally speaking, these tasks can be performed in parallel. There are real time performance issues, but they will not be addressed in this thesis. The extent to which the planner affects the various levels will be discussed, along with the breadth of the types of machines that have been simulated or controlled. The planning framework and the method for using it will be the subject of the next sections.

3.2 General Mapping Between the System and *Configuration Space*

The path planner receives goals from the task planner and determines the optimal way to move the system from the current status toward the goal, such that a variety of constraints are not violated. The terms used in the description of the problem can be directly mapped into the graph terminology of A* using a discretized structure called a *configuration space*.

Figure 10 gives an overview of the terms as they are described below. Beyond the mapping of the problem and terms, there are also issues of granularity of discretization. They will be touched upon here, and then discussed further in Section 3.6.1.

3.2.1 Terms for a Framework

The defined terms below form a framework [44] that will later be used to compute paths that can be carried out by a robot. Each term is a component that must be defined before the computation can be carried out.

Configuration Space / System Status

The autonomous systems considered must be able to be described in discretized form. That is, the systems are characterized by specific key properties (or parameters), each property having one or more ranges of valid discrete values. A *system status* therefore gives a unique setting for each of these properties. The span of all the possible parameter ranges is called the *configuration space*, abbreviated *CS*.

Nodes / States/ Events / Transitions

Because it is a discretized space, the status of the system can be considered identical to individual *nodes,* or *states* in robotics. Additionally, any *events* in the system that can cause the change between one system state and another can be viewed as *transitions* between the nodes.

Criterion / Cost

The objective of the system often has a *criterion* for success, such as fastest, shortest, least expensive, etc. In many cases, this can be directly translated to a *cost* incurred for a particular transition between nodes.

System / Graph / Configuration Space (continued)

Taken together, the *system* as a whole, translated to be nodes, transitions and costs, forms a configuration space *graph.* Even though it is clearly a graph, in this thesis it will continue to be referred to as a configuration space to be compatible with the original definition [22] and current literature.

Atomic Actions / Neighborhood / Successors

The allowed *atomic actions* that cause changes or transitions from one state in the configuration space to another are encapsulated as the *neighborhood.* This neighborhood is a collection of permissible successors. For robots, the permissible successors represent the core capability of the machine to make certain motions. Because this usually does not vary based on location, the neighborhood can be defined once for all locations. *Adjacency* is defined only through the neighborhood, not through proximity of the configuration space. The neighbors also may be determined based upon ‘rules of the game’, so there may be a few neighbors that are selected by a particular attribute of the controlled object. Assigned to each transition is the cost imposed for changing between the original state and the neighbor state. Therefore the combination of the states in configuration space with the transitions between them can be thought of as a graph with the states as nodes and permissible transitions as directed edges.

Constraints / Forbidden Regions / Obstacles

For many applications, the system also has *constraints*. These define illegal system states, often because of mechanical limits, interaction with the environment (i.e. obstacles), or imposed rules. These must be transformed into *forbidden regions* of nodes in the configuration space. In some graphs, the transitions into these nodes are removed, along with the nodes themselves. Alternatively, the nodes may be marked as illegal, or transitions into the node may have infinite (unattainable and high) cost, denoted by ∞. Each of these techniques will cause the search to avoid the nodes.

Goal / Start

The system *goal* may be mapped to one or more equivalent *goal nodes* in the discretized configuration space. Multiple goal nodes may exist because the formulation of parameters expressing the system may have more than one solution describing the system goal. (For example both left handed and right handed configurations of your arm can reach the same location.)

The system *start* is simply transformed to a specific *starting node*.

Series of Events / Optimal Path

The most desirable *series of events* leading from the current system status to the desired goal is analogous to finding the *optimal path* of transitions from the current node to the goal node requiring the minimum cost while avoiding all illegal nodes.

This desirable series of events therefore can be found by planning a path using the configuration space nodes, transitions, costs, forbidden regions, goal, and by knowing the starting state. A graph search method such as A* provides an efficient mechanism to determine the path.

The components described above can be mapped to many types of problems, but in this thesis a large number are illustrated from the field of robotics. Once the components are in place, the A* algorithm can be used to compute solutions.

Config. Space with A* for Robotics	***Task Based***	***Config. Space with A****
STATE	SYSTEM STATUS	NODE
TRANSITION	EVENT	TRANSITION
COST	CRITERION	COST
DISCRETIZED CONFIGURATION SPACE	SYSTEM	GRAPH
FORBIDDEN REGIONS OF STATES	CONSTRAINTS	FORBIDDEN REGIONS OF NODES
GOAL STATE(S)	SYSTEM GOAL	GOAL NODE(S)
STARTING STATE	SYSTEM START	STARTING NODE
OPTIMAL PATH	SERIES OF EVENTS	OPTIMAL PATH
NEIGHBORHOOD	ATOMIC ACTIONS	PERMISSIBLE SUCCESSORS

Figure 10: Terms Mapped from a Task to CS/A and CS/A*/Robotics*

3.3 Mapping A* to Robotics

Robotics applications can best use A* with a few modifications to improve A*'s performance and to gain greater utility from the result. In particular, Pearl's description of A* [34] begins the search of a graph from one or more 'starting nodes' toward the goal node. The nodes generate the permissible successors, and use transitions that represent permissible motion from those nodes to their successors. Through the process however, the 'path taken' results in pointers that point back toward the start, which then have to be reversed to find the actual path of travel.

In the configuration space examples, and the robotics description in Figure 10, we will make the distinction between the original method [30], which uses 'nodes', and the current modification which use 'states' for the clarity of the discussion. The reason is that the A* search has terminology that assumes the search begins at a starting node seeking the goal node. In the robotics application, we prefer to begin the search at the goal and seek the start, and have certain assumptions about the way the graph is generated. Although subtle, we must remain clear about the domain (general or robotics). These issues will be discussed next in more detail.

It must first be observed that a path found between the start and goal node cannot simply be reversed to obtain the path from the goal to the start, such as travel through cities with one-way streets. However the search from the start to the goal can produce the same path as a search performed from the goal to the start, as long as the permissible transitions and their costs are always taken consistently as directed edges in the graph.

In this framework, using the robotics terminology, the search will begin from start <u>nodes</u> which represent the goal <u>states</u>. If the starting nodes of the search are in fact the goal states, then the successor function of Pearl's A* is performed by the 'Neighborhood' which gives the states that point into a given node. A current state is said to have a neighborhood consisting of transitions that are the permissible motions *from* the neighboring states *to* the current state. This is similar to the original A*. Because the search is performed from the goal state however, the pointers that result from the search will point toward the goal state in the same direction as the transitions. See Figure 11. This has two practical consequences.

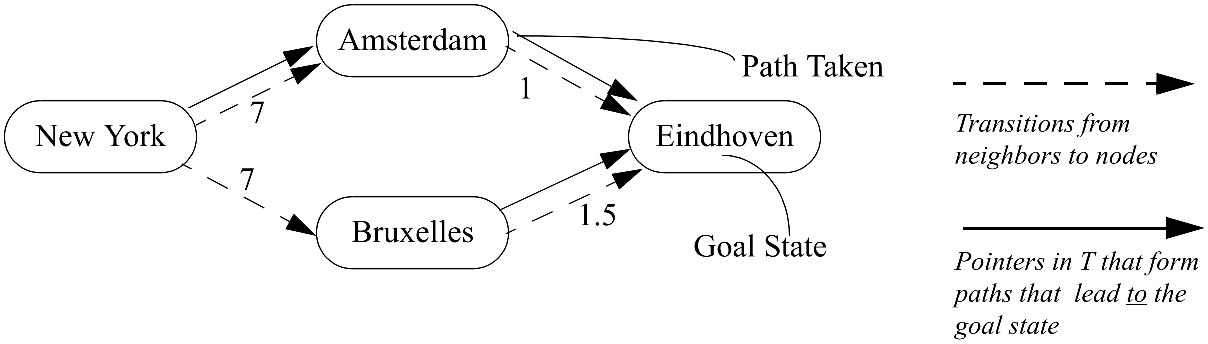


Figure 11: Computation 'Starts' from Goal State.

The first is that it is not necessary to find the path and then reverse it to find the actionable transitions. Because the pointers point from all states toward the goal, it is immediately possible to follow the pointers toward the goal. This has the added benefit for some systems where the system may miss one of the transitions (due to sensing or control problems), and find itself in an unexpected state. From that state it can immediately correct for the error by optimally moving toward the goal without recomputing the explicated graph. The explicated graph (described previously in Section 2.2) gives the optimal path in terms of pointers that lead to the nearest goal.

The second is that the starting state does not need to be known to begin the computation. The search can be performed so that the path from all starting states is essentially computed simultaneously. This can be achieved by using a zero heuristic.

As a final note, not every application is well suited to computations originating from the goal state, such as when the cost of a transition is based upon the accumulated cost from the starting state. This is the case for example when computing paths for cars factoring in traffic conditions. The estimate of any particular cost transition is a function of the time we expect to arrive there. This must be computed from our current location toward the goal.

This framework provides all the information required to compute the explicated graph containing transitions and pointers leading toward the goal. This may also be referred to as the *navigation map*, since it can be used for that purpose.

Example problems will now be explored along with their use of each of the components in the framework identified above, and the results that they achieve.

3.4 A Simple Mobile Robot Example

The first example problem will be using a simple mobile robot and a two level floorplan to compute the optimal paths from a current position to the nearest goal. The robot must travel through the building to obtain parts from a source (and then later deliver them). The 'task planner' will determine the possible equivalent locations that are sources for parts. The mobile robot must then independently plan and travel a path. Other factors may also be known, such as the difficulty (traffic density, sticky floors, etc.) in traveling some routes. These may have 'cost penalties' which are higher localized costs of travel which can be incorporated into the plan.

The building in Figure 14 has two floors, two exits, and one set of stairs. The first floor is on the left, and the second floor is on the right. The goal locations are in the upper left and lower left corners of the first floor. The elevator is in the lower right corner and leads to the second floor. This shows the level of complexity that can be handled easily. It shows that the robot situated on the second floor in the upper left corner has a long path to follow to reach the nearest of the two goals.

Configuration Space

In this case the configuration space is three dimensional. It consists of a 32 by 32 by 2 state grid in which the walls represent obstacles, and the range of motion is considered to be only those grid spaces within the building. The two levels are joined by the elevator in the lower right corner. One way to ensure the separation between the first and second floor is to interleave an additional 'solid' floor between them, leaving a free space in the lower right corner for the elevator. A preferable representation simply forbids 'up/down' transitions unless there is an elevator.

Neighborhood

The neighborhood is the set of permissible motions from any given current position. The motions are simplified here to be those motions shown in Figure 12, which are the simple east, west, north, south motions, which neglects the possibility for diagonal moves. This is done to make Figure 14 easier to read. As a special case at the elevator, there are additional arrows that point up or down, but are taken only if it makes sense (i.e. within the valid range of floors).

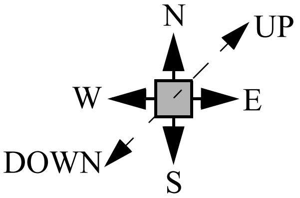


Figure 12: Neighborhood.

Cost Measure

The cost of making a motion is simplified to be one (1.0), for a transition in any of the six defined neighbors (east, west, north, south, up, down). An additional penalty occurs in areas where there is difficulty in passage. The amount of penalty is determined by the level of difficulty (such as 1000*level of difficulty). In this example, there are four levels of difficulty (numbered 0-3), three induce a penalty, the fourth indicates no penalty. The darker color represents a higher penalty. Thus the cost for moving from n to n' = c(n,n') + 1000*difficulty.

In Figure 14 there is a penalty found on both floors. These states incur a large (but not infinite) penalty. This allows a high cost to traverse the area, providing arrows through that area only if there is no alternative.

Constraint Transformation

There is no difficulty in performing obstacle transformation. It is a one to one correspondence with the task space, where the walls are the only forbidden regions.

Start and Goal States

The A* computation will use the goal states (exits) as the initial (starting) nodes for the search, and then use the neighborhood to determine the successors of a given node *a*. The cost incurred will be the cost of traveling from the successor to the node *a*.

A* to Compute Costs and Paths

Because this example relies on computing the paths from all starting states to the nearest goal state, a zero heuristic is used, and the A* search is not complete until all OPEN nodes are exhausted.

Precedence Order

In the A* process there may be a critical ordering of the neighborhood which does not permit the evaluation of one neighbor because a nearer logical neighbor is forbidden. For example, in Figure 13, if diagonal motion was permitted, it would not be sensible to have obstacles blocking a movement in x and y, but permit motion diagonally. Although the discretization of the obstacles represented in these two areas, may in fact permit this motion, it is certainly risky. The evaluation of an OPEN node should therefore take this into account if all illegal states are to be avoided completely.

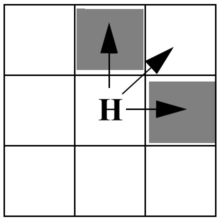


Figure 13: Precedence of Neighbors for Robot Examples.

3.4.1 Planned Solutions

The pointers that result from the A* algorithm form the 'navigation map' of Figure 14. This navigation map can be used by the reflex control system to direct the servos to achieve a particular motion. The feedback used can be as simple as feedback from sensors in the floor to complicated vision. This is used continually to correct the current state of the system.

If there is no exit from a particular location (perhaps because of being locked in a closet), then the robot would report this as an error to the calling task planner, rather than struggle.

3.4.2 Variation: Intelligent Emergency Exits

Using the same floorplan as Figure 14, we can compute the optimal emergency exit paths for all inhabitants of a smoky building. In this case, the objective is to provide exit directions for all the people of an unfamiliar building (conference center, hotel, etc.) by guiding them step by step. Observe that currently in a fire, we are all trained to follow blithely the red exit signs even if they lead directly into the blaze of a fire. Oddly enough, the fire/emergency systems today can specifically pinpoint the problem areas, but cannot convey that information to the inhabitants. Greater safety can be achieved by intelligent use of the public address loudspeakers, or by providing lighted arrows at shoulder height as a guide at logical locations such as junctions. The directions should lead the inhabitants out of the building by the most direct route, but by avoiding the problem areas as much as possible.

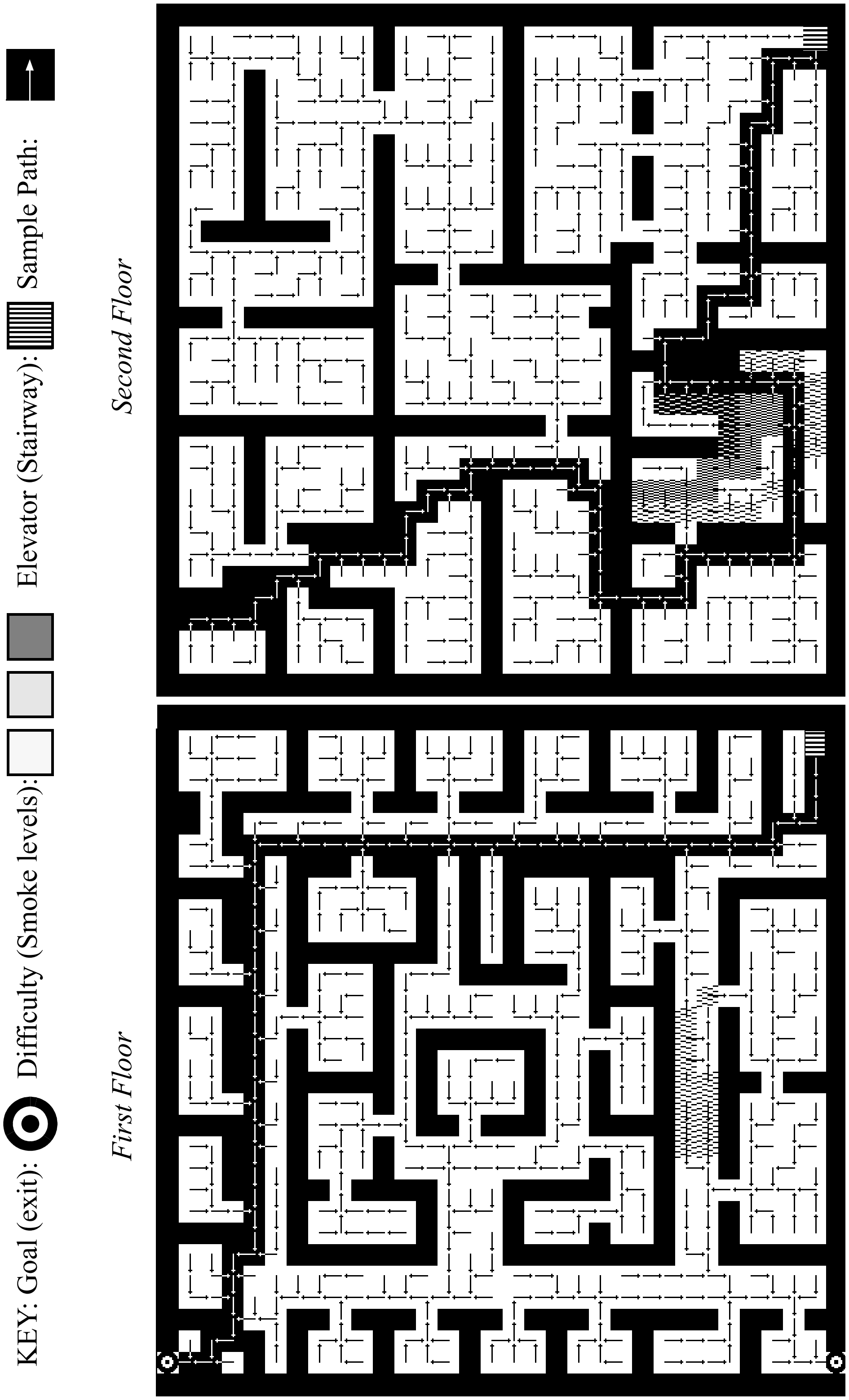


Figure 14: Mobile Robot / Intelligent Emergency Exits

Even though the problem is simple, the results may still be commercially and personally important. Computationally, the configuration space, obstacle transformation, and neighborhood are identical to the robot scenario except that the cost penalties might be increased for paths leading up stairs. An additional penalty (delay) occurs in areas where there is smoke, radiation or other problem. Similar to the robot, the amount of penalty is determined by the type of alarm sensed. In this example, there are three types of alarm. The darker color represents a more severe alarm.

By indicating crucial arrows from this map on exit-signs, a person located anywhere in the building will be led to the nearest safe exit by simply following the direction arrows. Notice that there is a smoky room on the second floor through which the people must travel in order to exit the building. This is not an enviable position to be in, but as this case shows, the arrows can lead the trapped people more confidently to safety. This also has the advantage that if people can proceed in a way that avoids emergency areas, then dangers of trampling are reduced because no person finds he must reverse his direction when discovering the emergency area. If there is no exit from a particular location (perhaps because of being blocked in by an impassable alarm) causing people to be trapped, then this could also be apparent to the emergency personnel monitoring the system.

3.5 Path Planning for a Robot Arm

In the previous example, the configuration space layout was a one-to-one mapping to the task space layout. In other words, the definition of movement in the space and the obstacle transformation were trivial. Industrial robots however are often robot arms, having several links and joints. The joints can be revolute or prismatic. The same motion planning method can be applied to such a robot. This will be discussed next using an example two-link, two jointed robot built by Newman [28,29]. It is shown in Figure 15 and on the front cover.

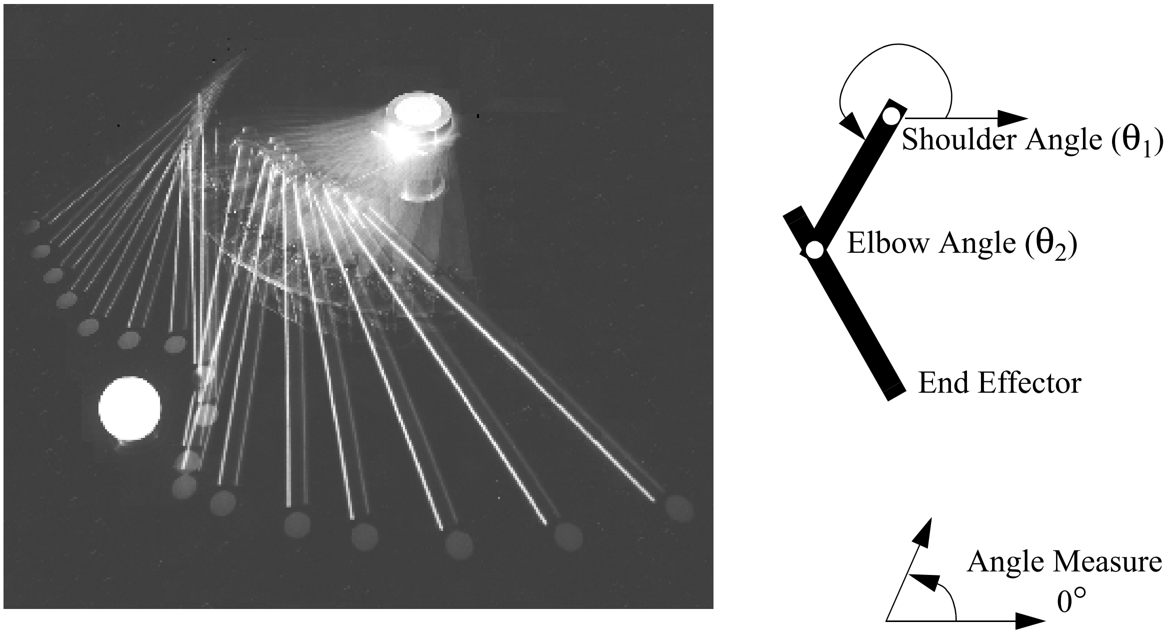


Figure 15: Newman's Two-Link Robot.

Robot arms have multiple joints, defining the number of degrees of freedom of the machine. They control the motion of the links in ways that are much more complicated than controlling a single point through a space. The objective however is still the same, namely to move a robot from one configuration to another without hitting anything, and to move optimally in some sense (least end-effector distance, effort, etc.).

Configuration Space

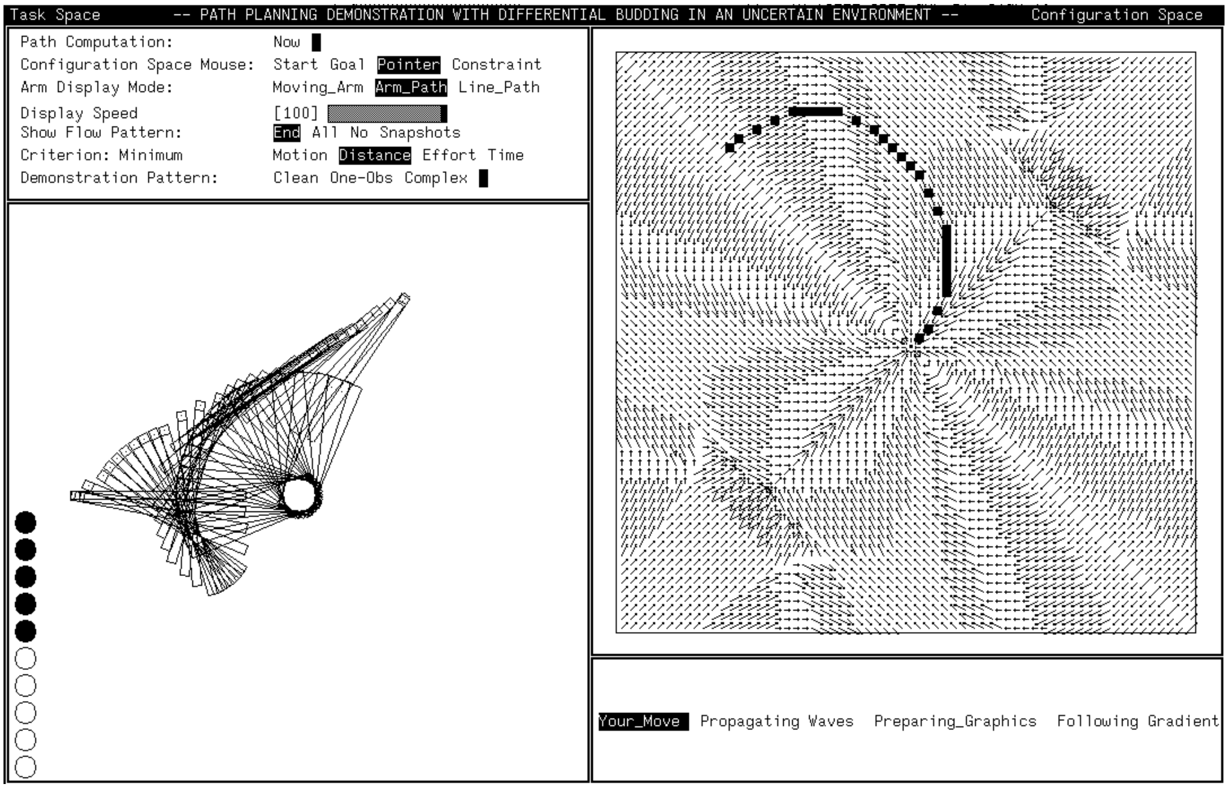
Because a robot is not a point, but rather a complex interconnection of joints and bodies, it is necessary to find a transformation from the real world (task space*)* to another problem space, where the robot can be thought of as a point. This is precisely the configuration space introduced in Section 3.2.1.

The state of the system must be characterized by one or more parameters which can uniquely describe that state. For example, the two joint angles of a two-link robot arm uniquely describe the position of the robot.

Continuing with the two-link robot example, a robot that has shoulder and elbow joints that are fully revolute, would have a coarsely discretized configuration space as in Figure 17. Because the robot can continuously rotate either joint, the configuration space in essence wraps around so that the right edge of the space is not a limit, but is connected again to the left side, and the top is similarly connected to the bottom, forming a torus topology. Each state is specified by the values assigned to the parameters. Figure 18 shows an individual state representing the robot nominally at the quantized pose 120°,60°. Naturally, all poses within the quantization range [120°± 22.5°, 60°± 22.5°] are mapped to the same state because of the coarse discretization. The mapping from configuration space to task (or system) space yields a specific nominal pose which can then be rendered. This coarseness of discretization will result in some level of inaccuracy, since the machine will be controlled through the nominal setpoints bringing the machine through possibly unnecessary locations.

The more standard representation measures the second joint angle relative to the first. The choice of representation is often a function of the way in which the (motor) actuators move. If the actuators are placed at a joint, then it becomes easier to control each of the motors separately as they move mounted to the robot arm rather than require all motors to move relative to some arbitrary origin. The two-link robot built by Newman [29] is belt driven, and because of the coupling, the angle of each joint is controlled based on the same fixed origin. The robot joint angles are each measured relative to 0°, which points to the right of the page. Although non-standard, it also makes it easier to determine the angle of the second joint by eye.

A user interface was built to show the configuration space of the robot (on the right) and the pose or path of the robot in task space (on the left). It is also used to gather images and gave insight that led to other fundamental improvements to the method. There are round obstacles of various types along the lower left side of the task space that can be dragged into the task space. In addition, the user interface provides a way to specify the start and goal position, set the optimality criterion, vary the speed of animation and set individual obstacle states directly in the configuration space. A snapshot of the user interface for the simulation is shown in used to gather later images is shown in Figure 16. This simulation shows the motion of the robot arm in task space on the left side of the interface, and the configuration space on the right side. The round obstacles are available on the left side of task space and are dragged into place.



Task Space -- PATH PLANNING DEMONSTRATION WITH DIFFERENTIAL BUDDING IN AN UNCERTAIN ENVIRONMENT -- Configuration Space

Path Computation: Now ▮
Configuration Space Mouse: Start Goal [✓ Pointer] Constraint
Arm Display Mode: Moving_Arm [✓ Arm_Path] Line_Path
Display Speed [100] ▨
Show Flow Pattern: [✓ End] All No Snapshots
Criterion: Minimum Motion [✓ Distance] Effort Time
Demonstration Pattern: Clean One-Obs Complex ▮

Figure 16: Robot User Interface for Simulation.

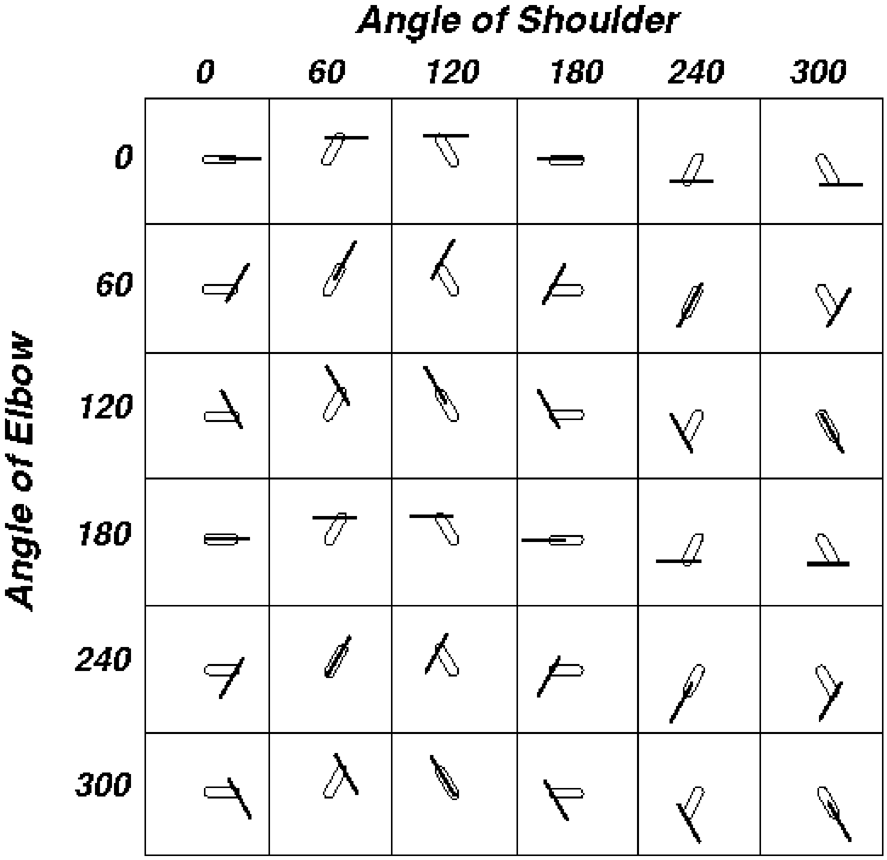


Figure 17: Coarse Configuration Space for a 2-Link Robot.

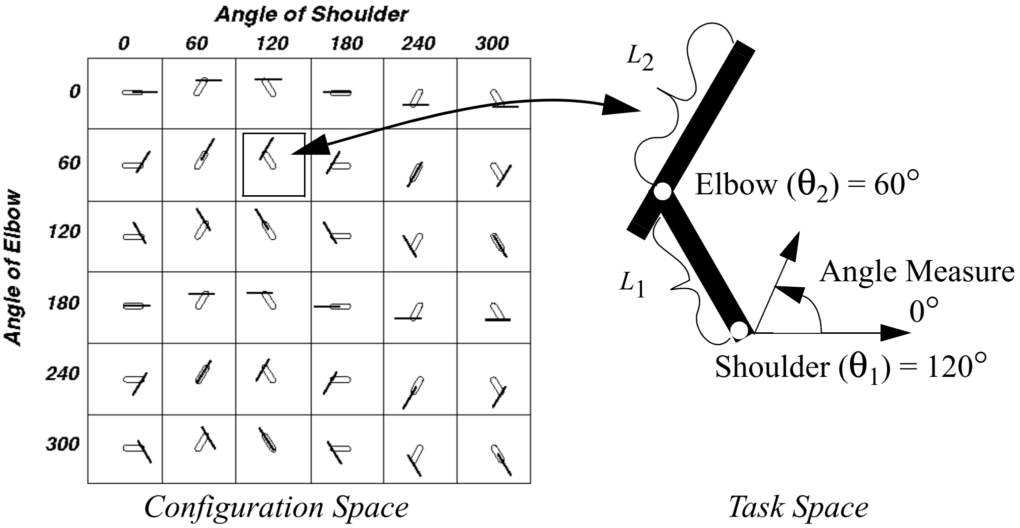


Figure 18: One Configuration.

Neighborhood

As stated in Section 3.2.1, the *neighborhood* in the configuration space is a set of possible motions for moving ‘legally’ from one state to another assuming no obstacles. It describes the fundamental motions of the system, and is depicted by an arrow showing the direction of a particular motion. Therefore if the example robot has one motor that is switched between the shoulder and elbow joint so that only one can move at a time, then the neighborhood of possible motions is as in Figure 19. Note that this neighborhood represents one unit of motion in the horizontal direction corresponding to a unit change in the shoulder angle, and one unit of motion in the vertical direction corresponding to a unit change in the elbow angle such as shown in Figure 20. If this neighborhood is used on the doubly revolute system shown in Figure 18, which has no joint limits, then the neighborhood would allow the elbow or shoulder transitions representing motion from 300° to 360° (identically 0°).

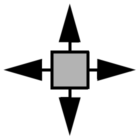


Figure 19: Simple 4-connected Robot Neighborhood.

Figure 19 is a simplified example of the neighborhood. If the joints could be moved simultaneously then four diagonal arrows would be seen as well. In later examples, more neighbors are used. Here things will be kept simple for the sake of explanation.

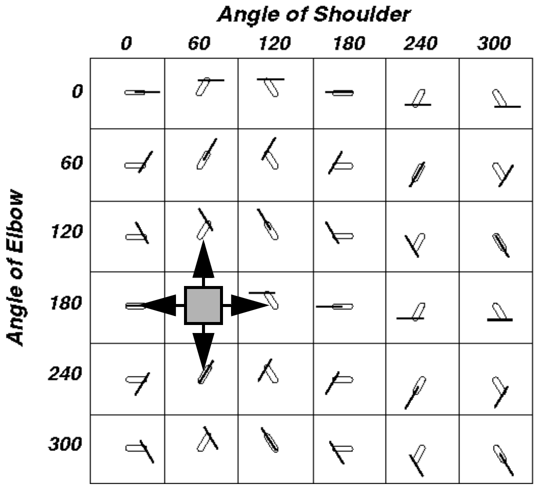


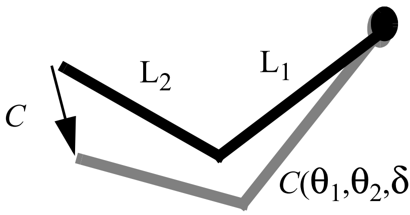
Figure 20: Neighborhood Applied to a Coarsely Discretized CS Graph.

Cost Measure

In accordance with Section 3.2.1, a *cost measure* must be associated with each motion, resulting in a cost-weighted neighborhood relative to a given state. It is usually expressed as a function of the 'current' state and the neighbor state, or more specifically, the cost of the transition for moving from the neighbor state to the current state. The measure need not be memoryless, that is, it may have cost factors that are a function of prior transitions or states along the path generated so far. Admissible heuristics also may be employed as part of the cost measure. The measure can be used to implement an optimality criterion, for example, a cost measure C that will minimize the total joint angle motion (giving the straightest possible path in configuration space) could be expressed as the Euclidean distance:

$$C(\theta_1,\theta_2,\delta\theta_1,\delta\theta_2) = \sqrt{(\delta\theta_1)^2 + (\delta\theta_2)^2}$$

Alternatively, a cost representing the distance traveled by the two-link robot's end effector (giving the straightest possible motion in the task space) can be expressed as in Figure 21.



$$C(\theta_1,\theta_2,\delta\theta_1,\delta\theta_2) = \sqrt{(L_1\delta\theta_1)^2 + (L_2\delta\theta_2)^2 + 2L_1\delta\theta_1 L_2\delta\theta_2\cos(\theta_1\text{-}\theta_2)}$$

Figure 21: Formula for Straightest Motion in Task Space.

Constraint Transformation

System constraints as described in Section 3.2.1 are either permanent or temporary. If a constraint is 'permanent', such as a joint limit, then rather than creating states that cannot be visited, the configuration space can represent the joint limits implicitly. This can be achieved by using the joint limits to define the parameter range in the configuration space. If there are temporary constraints, or constraints between other permanent constraints, in practice it is often simpler to mark the nodes as illegal. This maintains the homogeneity of a regular structure which is not only easier to manage computationally, but makes it simpler to restore the nodes, if the constraint is lifted. Rules can define the location of obstacles or forbidden states, such as the 'legal' squares on a checkerboard.

Obstacles and self-intersections (the robot body hitting itself), are represented as forbidden regions; one difficulty using the configuration space technique is determining them. The *transformation* of obstacles from task space to configuration space can be very difficult, particularly for high-dimensional spaces with complex shaped obstacles. More information about transformations can be found in [58].

In some simple two dimensional cases, the transformation can be obtained by ‘growing’ the existing obstacle by the size of the controlled mechanism. For example, a mobile robot that can rotate about its center axis, is treated as a circular object moving around obstacles. In this case, as long as the center position is no closer than the radius distance to the obstacle, then no collision can occur. Therefore it is sufficient simply to grow every dimension of the obstacle by the size of the radius of the robot. The robot can then be treated as a point location (at the center axis) moving about the grown obstacles.

Once the problem grows in complexity because of irregularly shaped obstacles or robots, robots that change shape with time (like a robot arm) and more complex issues like dynamics, the task of uncovering the illegal regions becomes more difficult. Others [1,29] have examined such cases, but no fast, general transform methods currently exist. Fortunately in a variety of problems the transformation can be performed in real time such as the two dimensional transform above. To understand the framework however, the relationship between the task space and configuration space obstacles is required.

In the simple coarse configuration space of Figure 22 the obstacle in the task space on the right would transform into the three illegal states in the configuration space on the left. Because each state in the configuration space actually represents a range of configurations, it is necessary to require that if any configuration causes a collision, the entire range of configurations will be restricted by marking the corresponding configuration space state illegal. These states are therefore forbidden by either deleting transitions into the state, creating an infinite cost transitions into the state, or by having an infinite delay cost for existing in those states. Regardless of the implementation, the computed paths will avoid these states.

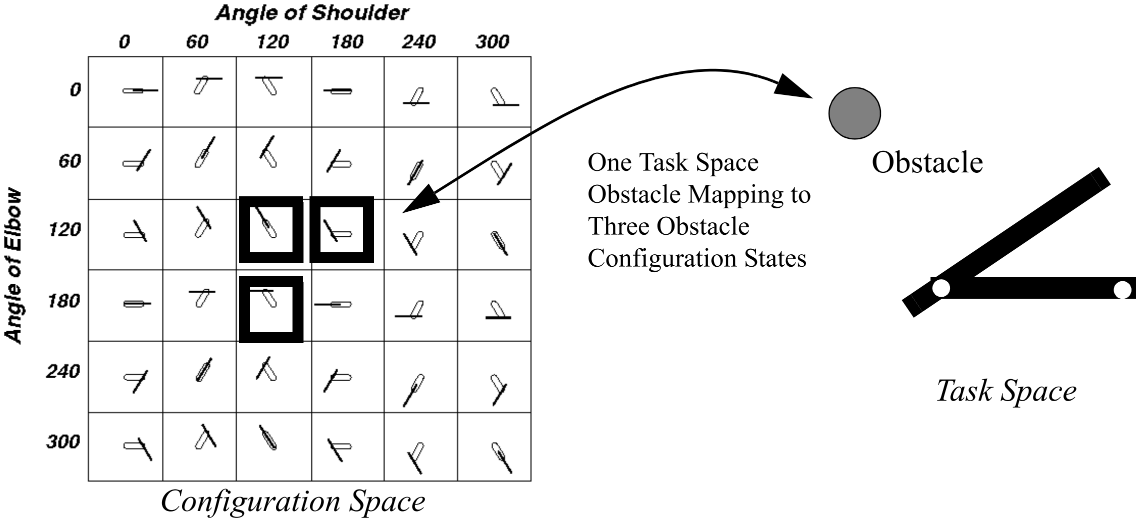


Figure 22: Obstacle Transformation

Start and Goal States

Goal states are a specification of acceptable target locations. These may be supplied by a higher level task planner [25,26] or by user specification through the user interface (pointing to a grid on a map for example). In the robot example the goal position could be the pose represented by the (180°, 180°) state.

The *starting state* must be known in order to issue commands to the controller. It is not required for the planning process unless a heuristic requiring the starting state is employed.

For either the start or goal pose of the robot, it is likely that the specification will not be in the form of a set of joint angles, but rather that a more general specification is given. For example the required pose is to place the end effector (or gripper) at a specific location. The joint angles to achieve this pose (possibly mapping to one or more goal states) can be computed using standard methods of *inverse kinematics* [2].

3.5.1 A* to Compute Costs and Paths

Now that the configuration space has been mapped into a graph with a start and goal, costs for transitions, a possible heuristic, a definition of successors in the neighborhood, and forbidden regions, then all the elements of the framework are in place. The cost-weighted neighborhood is simultaneously propagated from all goal states through the configuration space in an A*, least-cost-first manner until the space is filled (or at least until the start state has been reached).

Figure 24 shows a 64 by 64 grid of interconnected nodes which represent the same range of joint motion as Figure 17, but is a more finely discretized configuration space. For clarity, the delineation of nodes is removed. Each node is connected to 16 local neighbors surrounding it as shown in Figure 23. As before, the topology allows both vertical and horizontal wrap-around.

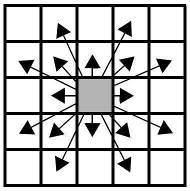
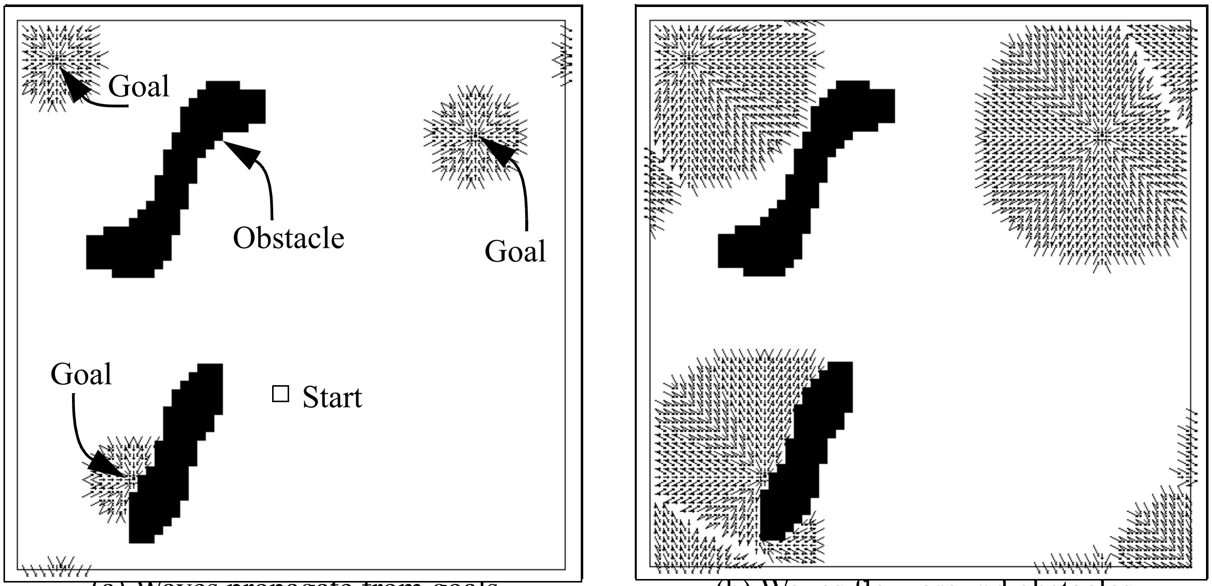


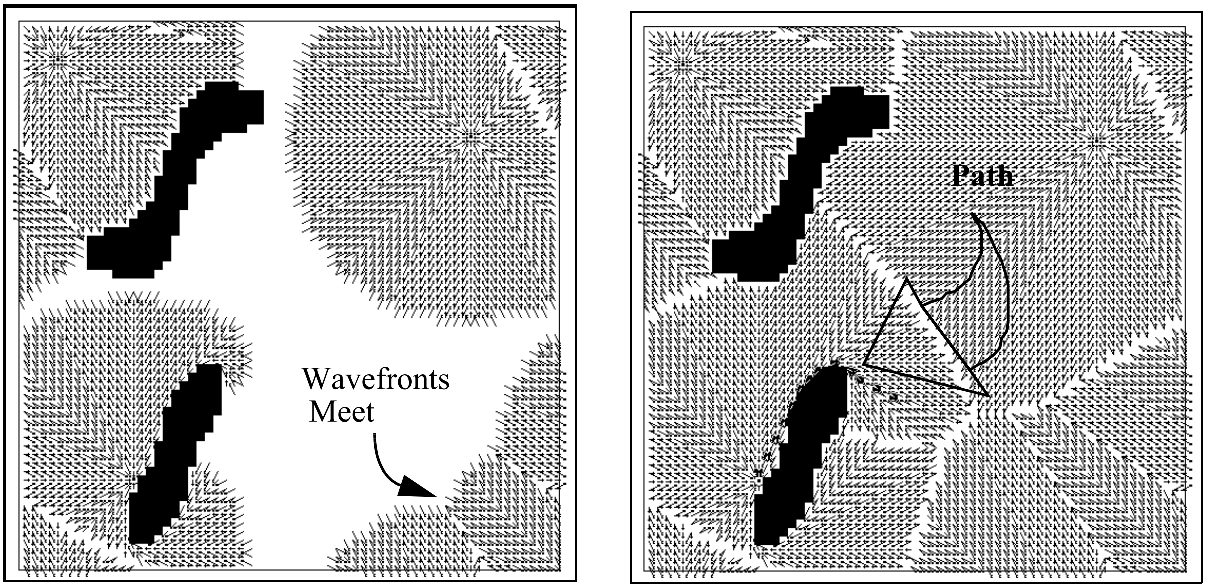
Figure 23: 16-Connected Neighborhood.

Using A* and a zero heuristic with three goal nodes and two forbidden regions, snapshots of the intermediate searching are shown in Figure 24(a-c). The path between a start and a goal is determined in Figure 24(d) by tracing from the start to the goal via the arrows. The intermediate nodes are set-points (subgoals) along the way. Only the resulting pointers of the explicated graph are shown, not the original graph G_0. In examples like this, the processing of the graph by A* is reminiscent of waves propagating through an area, and is often referred to as wave propagation [12].

A* marks each state with the cost to the goal and a pointer toward the neighboring state leading to the nearest goal. Because pointers are followed from node to node, it will be assumed that any pointers from a node n to n' will be stored in the node n. Pictorially, the cost waves cover the configuration space graph as in Figure 24. In practice, for this two-link example with a relatively coarse discretization, A* is so fast that the full space can be computed quickly without a heuristic, since the computation of the heuristic adds time. Whenever there are a large number of states in the configuration space a heuristic has greater impact on the time to compute. This is to be expected when generating high dimensional spaces or using fine discretization, since the purpose of the heuristic is to reduce the number of states explored.



(a) Waves propagate from goals (b) Waves flow around obstacles



(c) Wavefronts meet and stop (d) Final pattern and a path

Figure 24: Wave-front Snapshots. Waves Move Around Black Areas of Infinite Cost.

3.5.2 Planned Solutions

The result of A* is a globally optimal path from every reachable state to the nearest goal. The total solution is an explication graph with pointers and is shown in Figure 24. From every state there is a motion indicated to reach the next feasible state. In addition, the state also contains the total cost remaining to reach the goal. This information alone is useful, since a decision could be made in advance about whether to attempt the goal based on the commitment to the cost of the path. By following from state to state, the goal is ultimately reached.

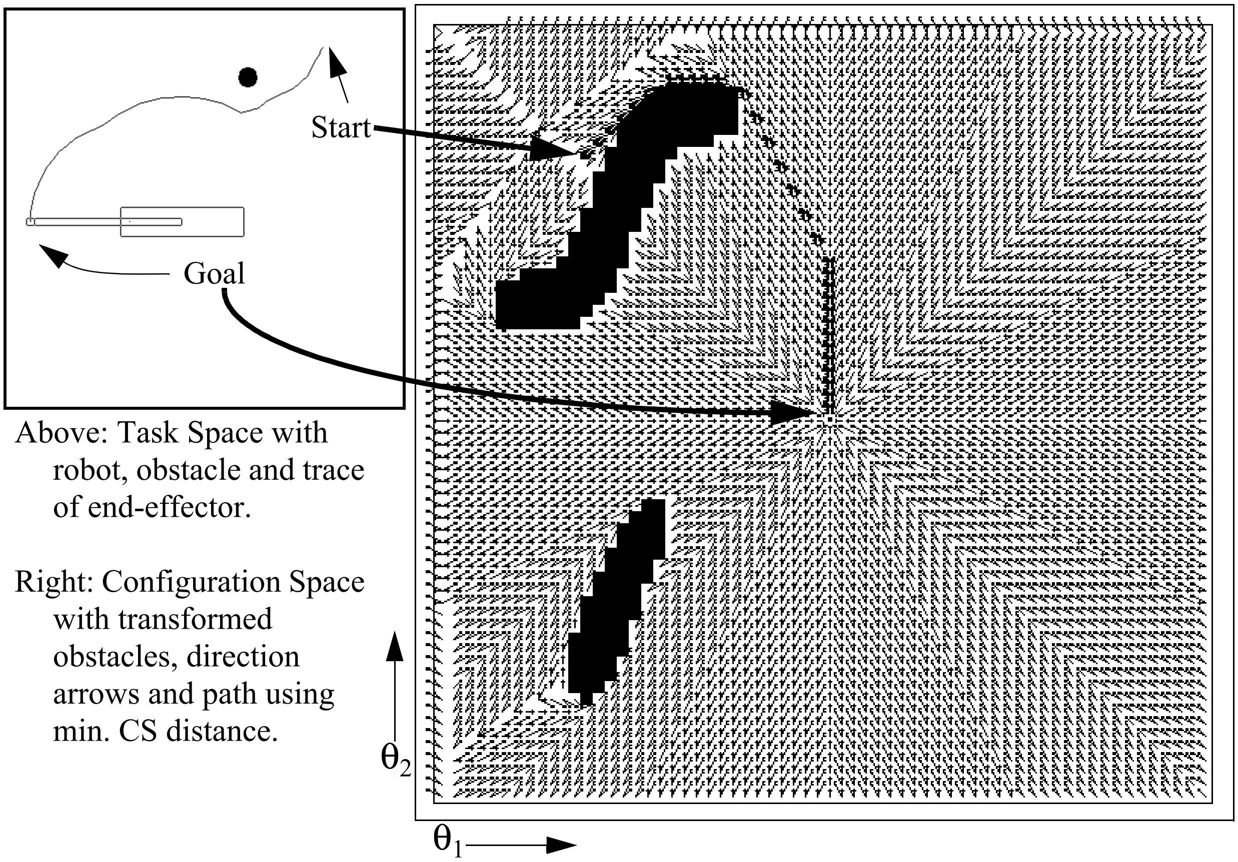
3.5.3 Variations: Robot Examples using Different Cost Criteria

This framework handles four optimality criteria (cost measures) to guide the behavior of the robot, namely least communication, straightest path, least effort, and fastest motion. A* allows an efficient expansion of the states regardless of the cost measure. The cost waves are generated according to the cost. Each of the examples below uses a different measure of optimality. In all, the permissible motions are described by a 16-connected local neighborhood. In all of these examples the obstacle is in the same position so that the resulting paths can be more clearly compared. For each state in the configuration space there is at least one arrow that shows the next pose (setpoint) that the robot should achieve. By following these setpoints the robot will move optimally and purposefully to the goal. Note that the imperfections in the end-effector travel are due in part to the motion required for obstacle avoidance, but also to discretization error.

Least Communication: Minimum Distance in Configuration Space

The example in Figure 25 minimizes the distance in configuration space. This corresponds to minimizing the number of setpoints that must be sent to the reflex controller. This can be useful if communication to the robot is the system's bottleneck. The cost measure is the Euclidean distance which is given in at the bottom of the figure as a function of two nodes α and β. The cost measure as a function of two poses of the robot $\alpha = (\theta_1,\theta_2)$ and $\beta =(\theta_1 + \delta\theta_1, \theta_2+ \delta\theta_2)$. The transformation of the task space on the left is shown in the configuration space shown on the right. In the graphics of the implementation, as each state is read-out, the robot animation leaves a trace of the end-effector.

The path found using a zero heuristic is shown in Figure 25. This covers the entire space, because no starting node was selected. The utility in this complete navigation map can now be seen, because for every starting state, a path can be constructed to the goal. For applications where the goal is likely to be static, but the start may vary, this is an effective method to avoid recomputation of similar problems.



Above: Task Space with robot, obstacle and trace of end-effector.

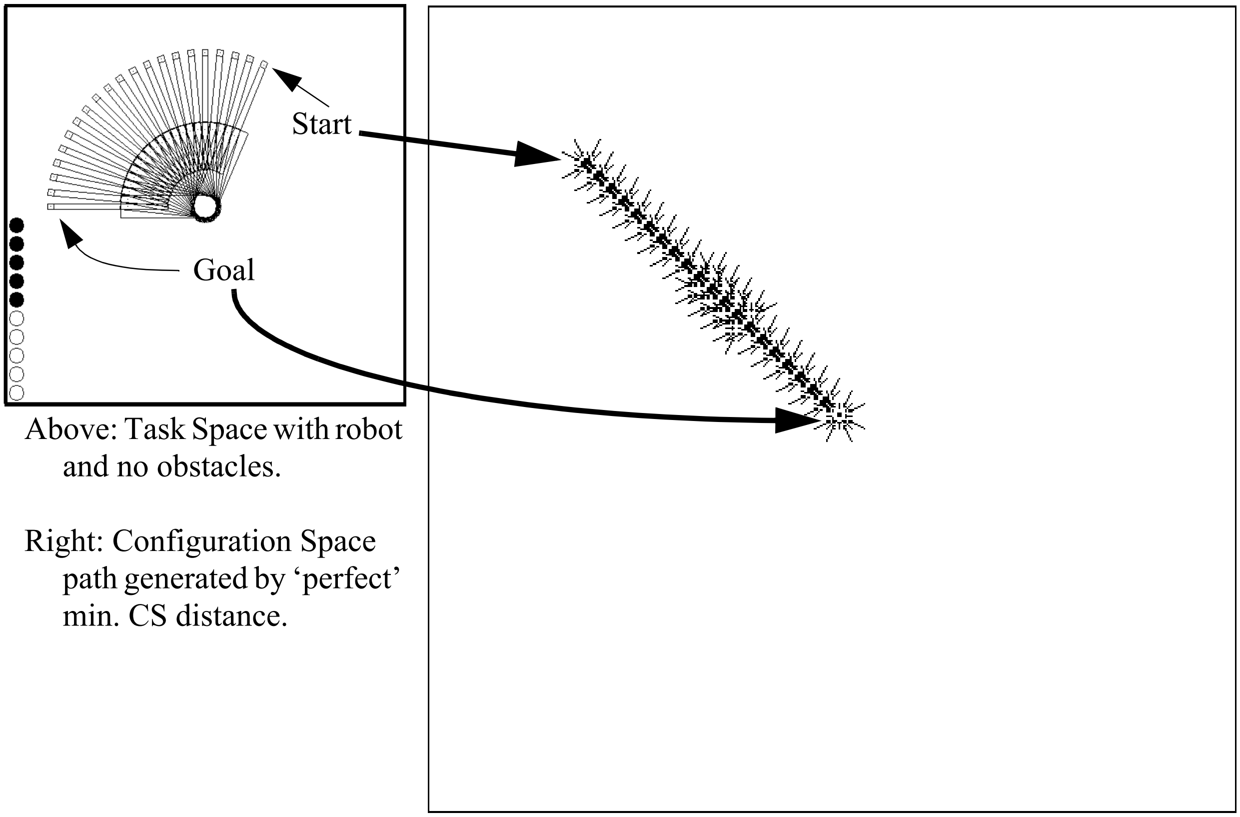
Right: Configuration Space with transformed obstacles, direction arrows and path using min. CS distance.

$$C(\alpha_{\theta1},\alpha_{\theta2},\beta_{\theta1},\beta_{\theta2}) = \sqrt{(\delta\theta_1)^2 + (\delta\theta_2)^2}$$

Figure 25: Least Communication.

Least Communication: Minimum Distance using an Admissible Heuristic

In Figure 26, the same cost criterion is used, except an admissible heuristic is used. The heuristic gives the straight line (Euclidean) measure $h(n_{\theta 1},n_{\theta 2},s_{\theta 1},s_{\theta 2}) = \sqrt{(\delta\theta_1)^2 + (\delta\theta_2)^2}$, where the change in angle is measured between the current node and the start. This produces a straight line from the goal configuration state to the start. Notice the dramatic reduction in the number of nodes opened to find the solution. Even though there are fewer nodes, there is more calculation required for the heuristic than for local computation of nearby neighborhoods. The local neighborhoods, because they are consistent, can be pre-computed once for each combination of $\delta\theta_1$ and $\delta\theta_2$ and placed in a look-up table, whereas this is not possible for every combination of the current node and start. The area explored is then an elliptic shape which is bounded by nodes for which c(g,n)+c(n,s) is constant. The better the heuristic, the smaller the ellipse.



Above: Task Space with robot and no obstacles.

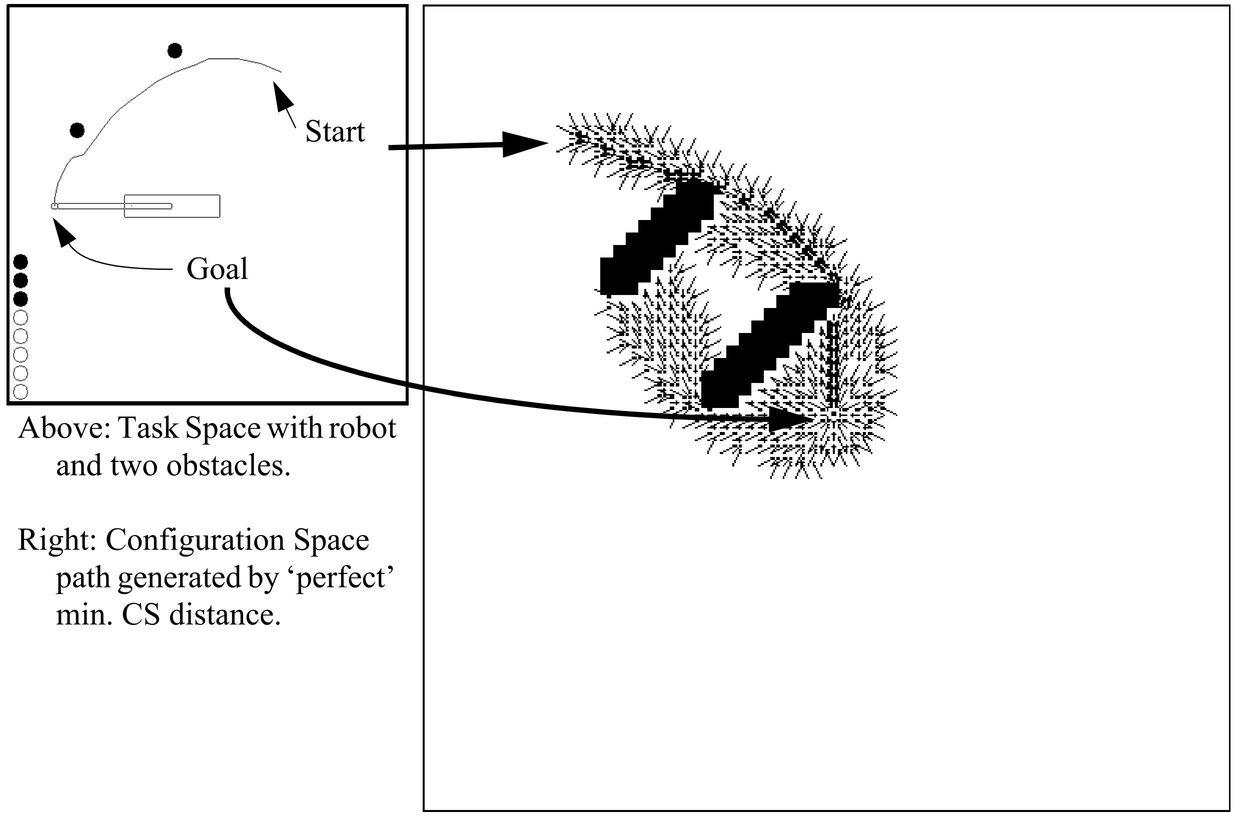
Right: Configuration Space path generated by 'perfect' min. CS distance.

$$H(N_{\theta 1},N_{\theta 2},S_{\theta 1},S_{\theta 2}) = \sqrt{(\delta\theta_1)^2 + (\delta\theta_2)^2}$$

Figure 26: Euclidean Distance using a 'Perfect' Admissible Heuristic.

Least Communication: Minimum Distance &Admissible Heuristic for Two Obstacles

In the example of Figure 27, there are two obstacles, each transforming to one blob each. The heuristic causes the widening as it contacts the obstacle nearer the goal, and then works around the two ends toward the farther obstacle. By minor differences in symmetry, the path is found around one side before the other.



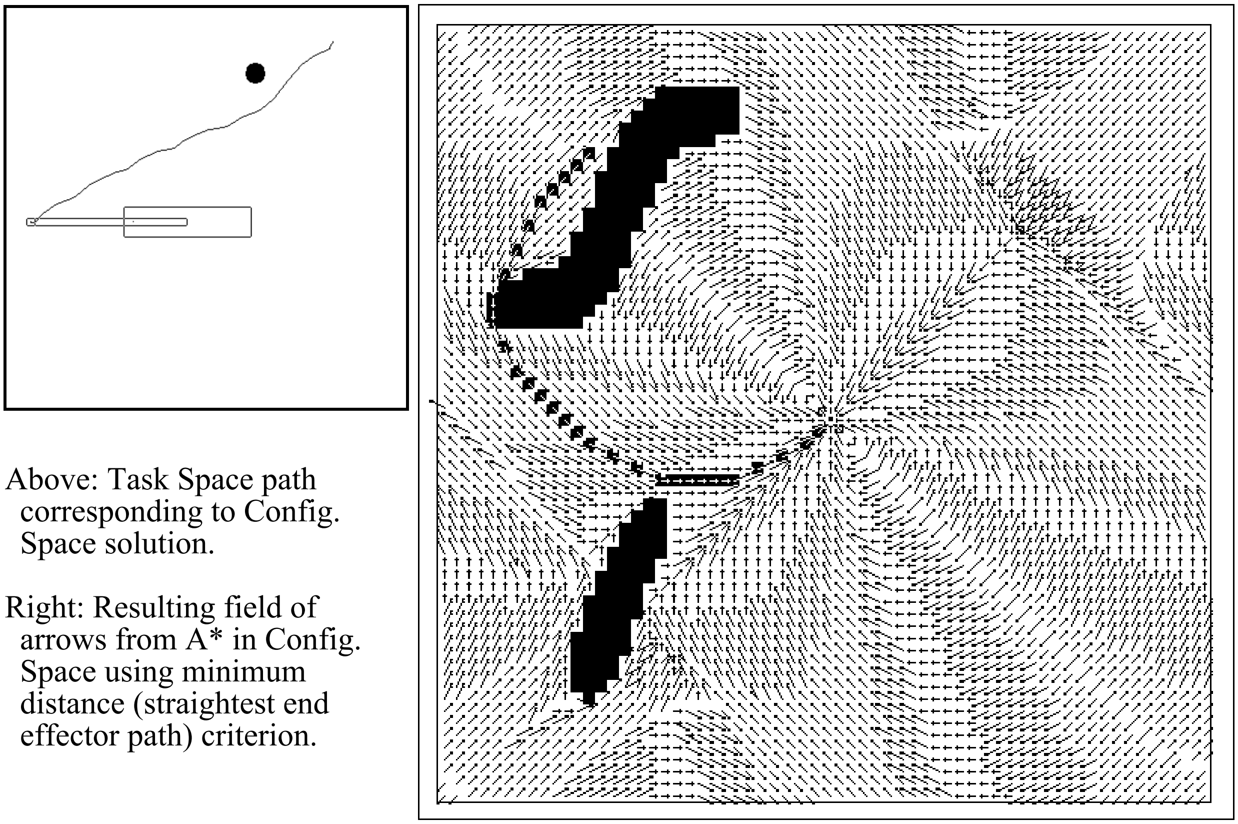
Above: Task Space with robot and two obstacles.

Right: Configuration Space path generated by ‘perfect’ min. CS distance.

Figure 27: Euclidean Distance & Admissible Heuristic for Two Obstacles.

Straightest Path: Least Travel for the End-Effector

The example in Figure 28 minimizes end-effector travel. Least end-effector motion will reduce the distance that the tip of the robot will move, yielding the straightest possible, most direct motions. The same cost is given previously in Figure 21, but is restated for convenience. L_1 and L_2 are the lengths of the upper arm and lower arm. L_1 is measured from the shoulder to the pivot of the elbow and L_2 connects the elbow to the end effector.



Above: Task Space path corresponding to Config. Space solution.

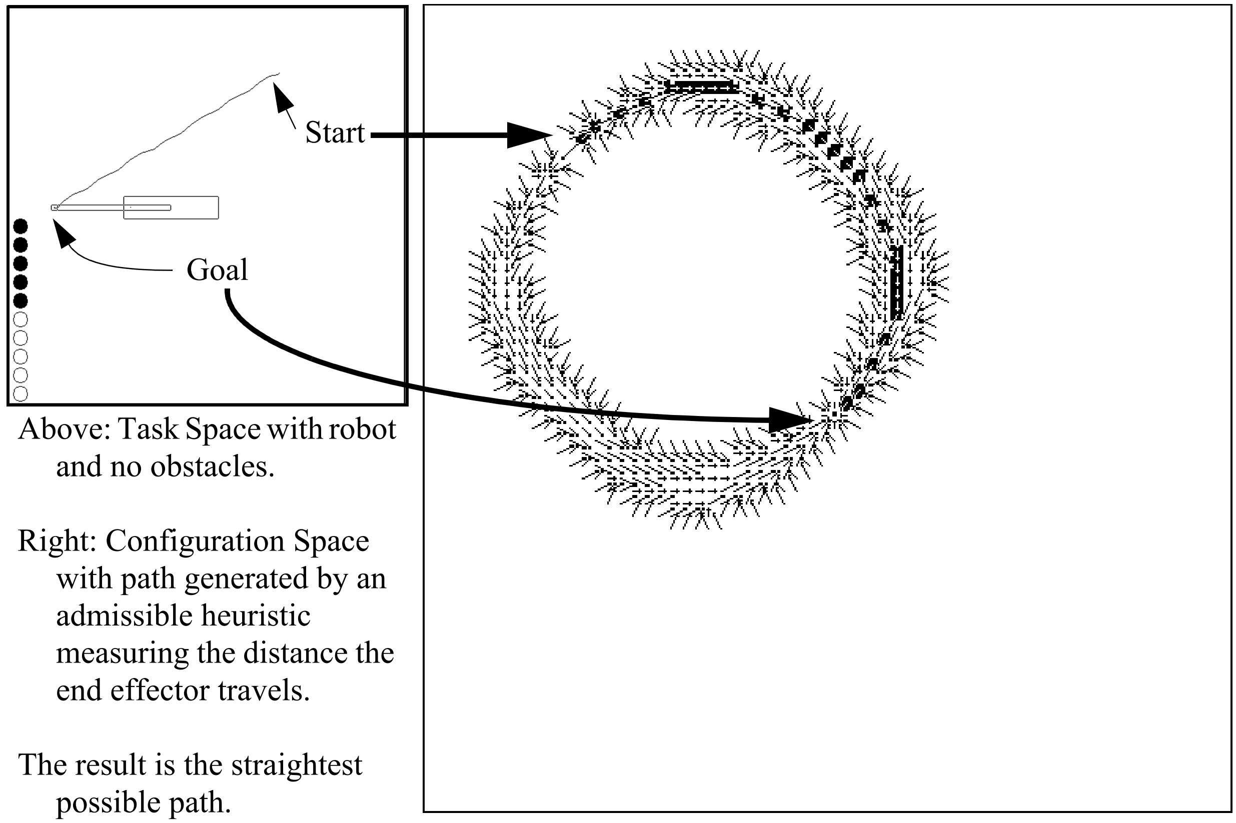
Right: Resulting field of arrows from A* in Config. Space using minimum distance (straightest end effector path) criterion.

$$C(\theta_1,\theta_2,\delta\theta_1,\delta\theta_2) = \sqrt{(L_1\delta\theta_1)^2 + (L_2\delta\theta_2)^2 + 2L_1\delta\theta_1 L_2\delta\theta_2\cos(\theta_1-\theta_2)}$$

Figure 28: Straightest Path

Straightest Path: Least Travel for the End-Effector using an Admissible Heuristic

In Figure 29, the heuristic function is the cost estimate given in Figure 28, except that the cost is from a node α to the starting state. The heuristic weights the left and right handed symmetric configurations equally, therefore the search explores each half ring at the same rate. Only by chance does the right configuration reach the starting node before the other.



Above: Task Space with robot and no obstacles.

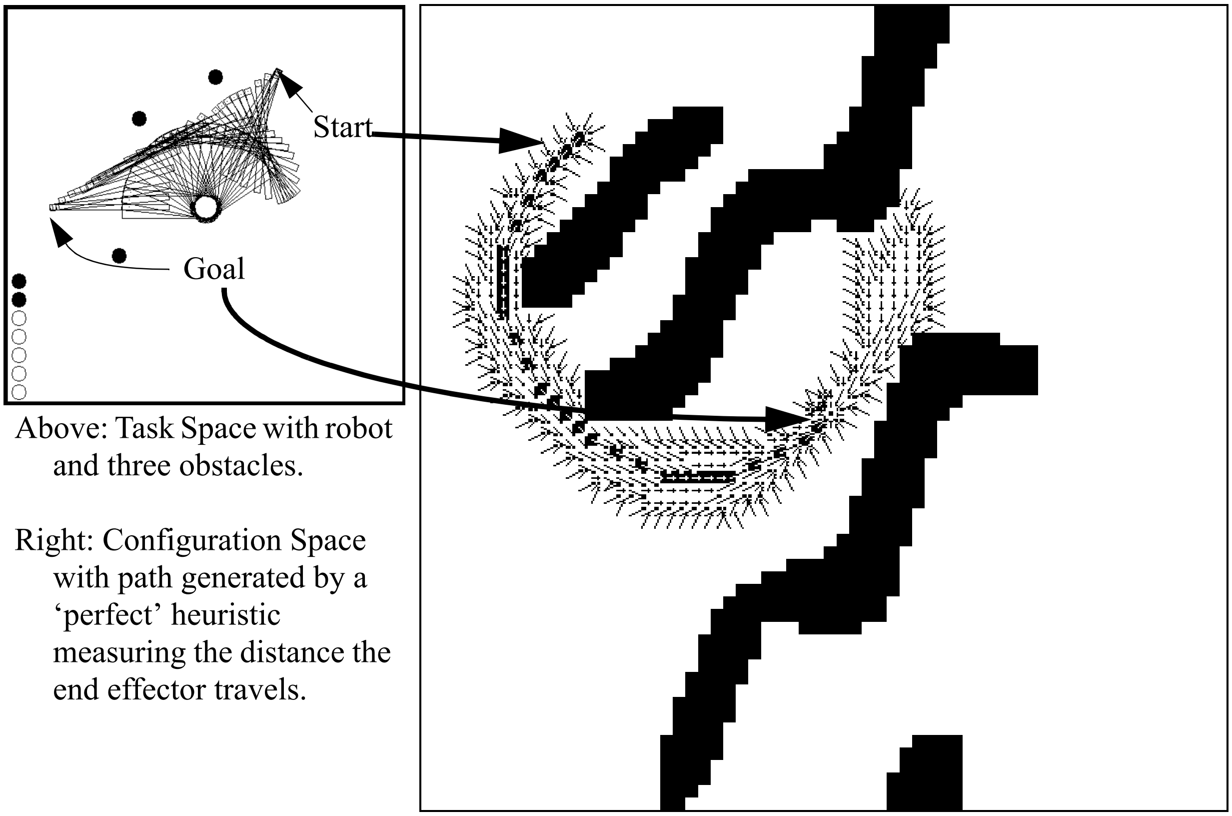
Right: Configuration Space with path generated by an admissible heuristic measuring the distance the end effector travels.

The result is the straightest possible path.

Figure 29: Straightest Path using an Admissible Heuristic.

Straightest Path: Using an Admissible Heuristic for Three Obstacles

In Figure 30, the search attempts to reach around one side, only to find that path blocked. The lesser cost path eventually becomes the left branch as the right branch becomes more expensive to explore.



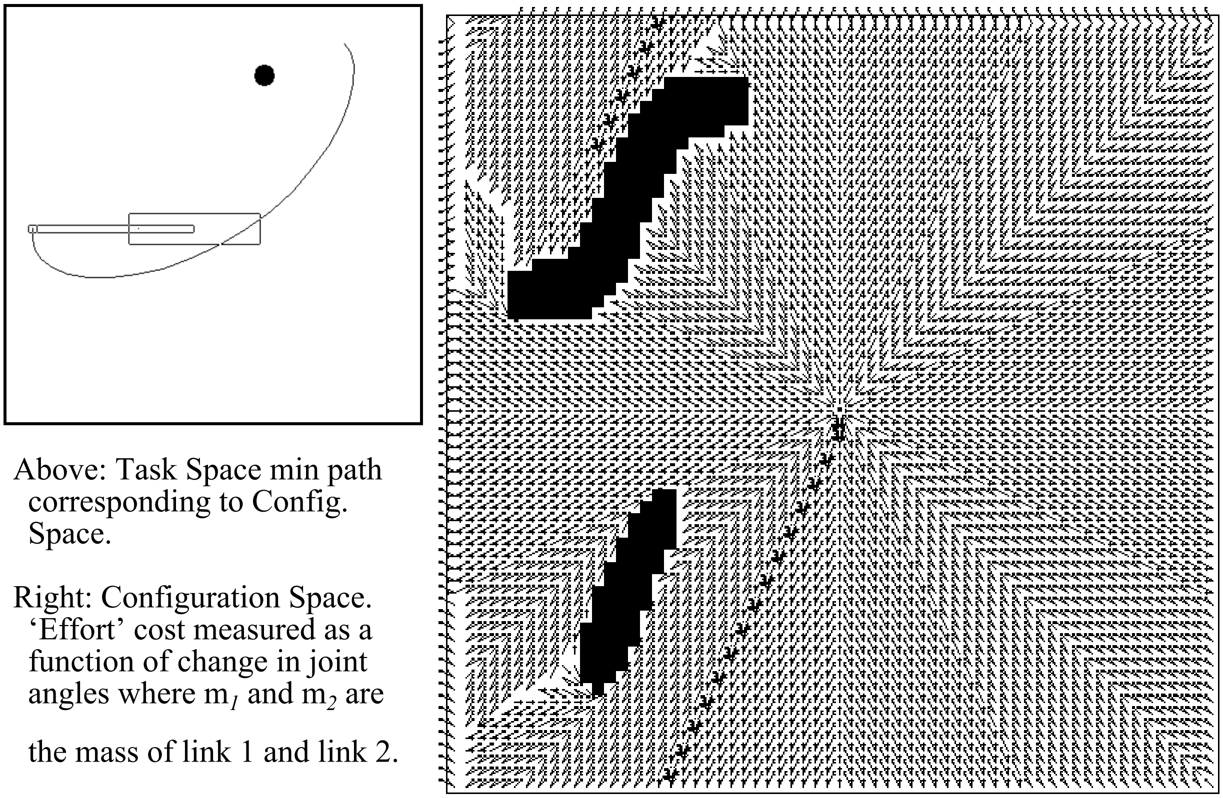
Above: Task Space with robot and three obstacles.

Right: Configuration Space with path generated by a 'perfect' heuristic measuring the distance the end effector travels.

Figure 30: Least Travel for End Effector through Three Obstacles using an Admissible

Least Effort: Grace with Least power consumption

The example in Figure 31 illustrates the least effort criterion which minimizes the use of high mass robot links to reduce power requirements. For the example, the first link is 5 times the mass of the second link. This causes a preference to move the lighter link. The result is also a graceful motion. The cost measure *C* is given in the formula within the figure. In this case, the cost is represented as the change in the respective angles of the shoulder and elbow since the cost is independent of the current location. The masses of the first and second links are m_1 and m_2 respectively.



Above: Task Space min path corresponding to Config. Space.

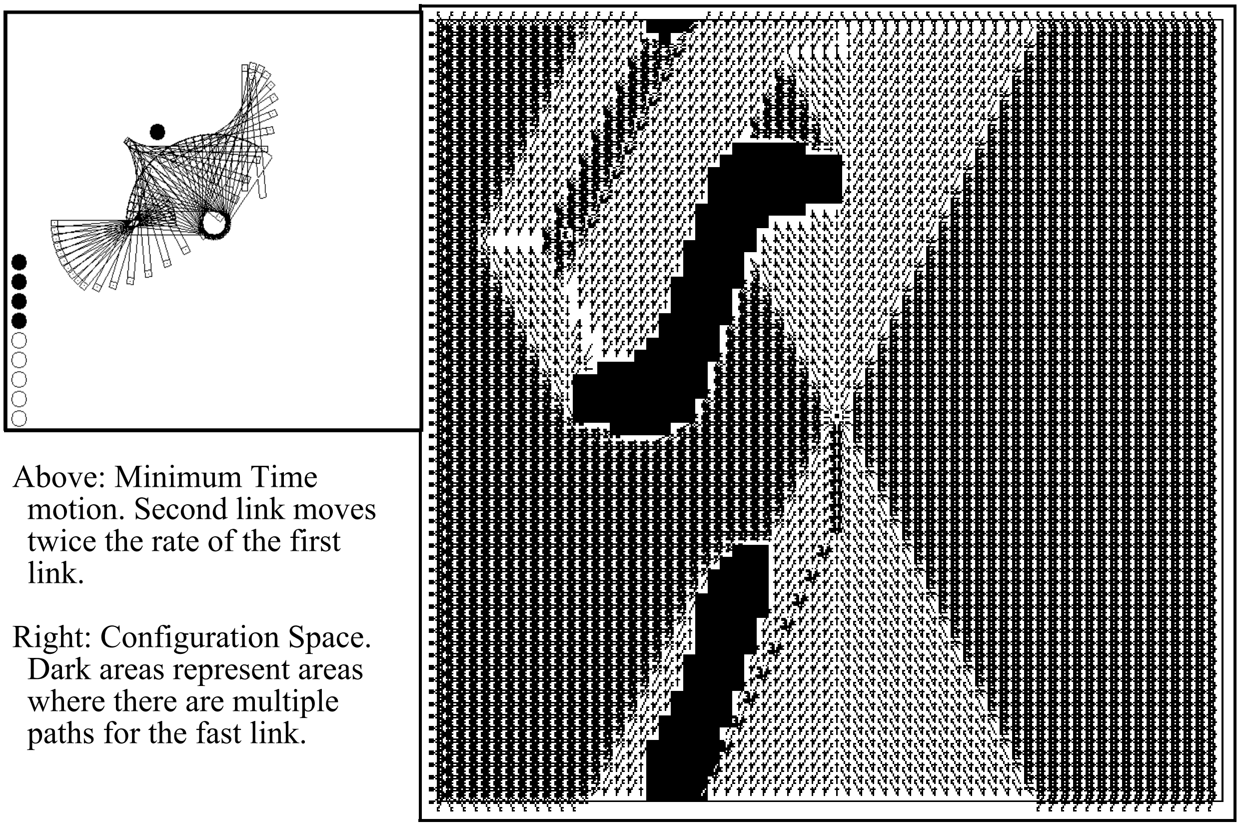
Right: Configuration Space. 'Effort' cost measured as a function of change in joint angles where m_1 and m_2 are the mass of link 1 and link 2.

$$C(\delta\theta_1, \delta\theta_2) = \sqrt{(m_1\delta\theta_1)^2 + (m_2\delta\theta_2)^2}$$

Figure 31: Least Effort

Fastest Motion: Minimum Time

The example in Figure 32 shows the fastest path possible for a robot where the second (smaller) link can move at twice the speed (v_2) of the first link (v_1). This is more precisely described by the figure's included formula. This causes an interesting pattern to emerge in the configuration space. The pattern shows a darker grey area which results from many arrows pointing in the general direction of the goal, but having a great deal of latitude for the second link motion until the first link crosses the boundary into the less grey area. Because the second link is fast and controlled independently from the first link, there are many configurations where the second link can easily reach the goal in the (longer) time required by the slower first link.



Above: Minimum Time motion. Second link moves twice the rate of the first link.

Right: Configuration Space. Dark areas represent areas where there are multiple paths for the fast link.

$$C(\delta\theta_1,\delta\theta_2) = max \left\{ \left|\frac{\delta\theta_1}{v_1}\right|, \left|\frac{\delta\theta_2}{v_2}\right| \right\}$$

Figure 32: Minimum Time

3.6 Performance Issues vs. Quality of Paths

3.6.1 Representational Issues in Configuration Space

Earlier in the framework discussion, the system parameters were discretized into a configuration space structure. The immediate question is then how to determine the proper level of discretization so that a sufficiently accurate result is produced. Another factor that effects the accuracy however, is the number of neighbors that each node has. The issue is how to trade off between the level of discretization and the number of neighbors to achieve acceptable results within the required amount of time.

From Formula 1, of Section 2.5.1:

$$\boxed{\text{Time} = (\gamma+1)\, b \sum_{t=1}^{T} \log_2 \Omega(t) + T(M(G+C+H) + K(\gamma+1))}$$

it follows that the computation time is a function of the number of OPEN nodes, and the total number of iterations (T) required to find the solution. Both of these are related to the discretization which determines the number of nodes in the graph between the start and goal. To reduce computation time, this number should be minimized.

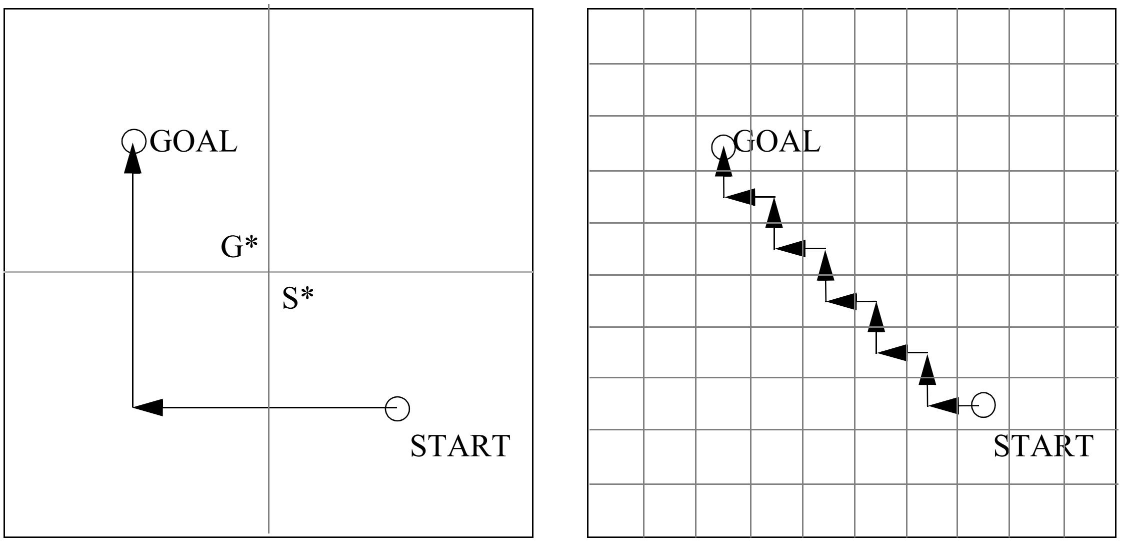
The number of neighbors affects the constant M in the equation. Assuming the average M << T, then increasing the number of neighbors has generally little impact on the computation time.

Increasing Discretization

Intuitively, increasing the discretization would seem to improve the accuracy of a straight line solution, because visually, the line appears to move more directly from start to goal. Figure 33 shows a coarse and fine discretization of an obstacle-free configuration space. By inspection, we can now see that the total distance by traversing in larger steps vs. smaller steps is equal. This is because the sum of the small x steps on the right must equal the large step in x on the left (i.e. the net distance is the same); the same is true for y. Therefore, for any discretization dividing the space into (1..n) discrete pieces will generate the same cost. But there is still some utility in increasing the discretization.

The previous discussion had to do with the sum of distances being equal regardless of discretization, but it did not address the fact that the key states of interest, such as the start, goal(s) and obstacles are also affected by the discretization. Suppose the true (continuous) start, denoted S*, is not located nominally in the center of the state as drawn, but rather in the upper left corner of that coarse state, and the true goal, denoted G* is located in the lower right of its nominal state. In this case, the naive measure from the centers of the respective states will cause a significant error compared to the true cost.

Another factor is that the transformed obstacles will label a state as ‘illegal’ if any part of the state has the possibility of a collision. In a coarse configuration space, not only will entire regions be avoided unnecessarily, but it is possible that an existing solution (reaching between two nearby obstacles) will not be found at all. The possibility that no solution is found, is therefore the overriding concern when defining the configuration space discretization.



Left: Very Coarse Discretization. Total Cost is 2.

Right: More Finely Discretized Transitions have Same Total Cost as Coarse Transitions.

Figure 33: Coarse vs. Fine Discretization.

Neighborhood Variation & Density

A neighborhood defines the transitions that are permitted between one state and others. It is possible to have a rich variation in the contribution and number of neighbors. In a situation where a 4-connected neighborhood (e.g. North, East, South, West) is used, a more valuable neighborhood that includes the diagonals (NE,NW,SE,SW) would enable a more direct and precise path to be generated to a diagonally located node. A 16-connected neighborhood, including ‘knights moves’, such as NNE, would improve the path still further. In each case the neighbors each contribute a unique direction that cannot be more efficiently created by other neighbor combinations. Thus a neighbor that is EE, would not be beneficial. The more uniquely contributing neighbors there are, the more improved the resulting calculation.

The amount of error for Euclidean distances in 8-connected and 16-connected neighborhoods has been studied by Dorst [11] to determine accurate unbiased distance measurements on a grid. In this case the grid of 8-connected neighborhoods is formed from the horizontal, vertical and diagonal transitions in configuration space. A 16-connected neighborhood includes the 8 ‘knights-moves'. By using 8 connected neighbors in a discretized Euclidean distance there is a 4% error compared to the optimal solution. By using 16 connected neighborhoods there is a 1.4% error due to finer discretization of orientations. Therefore increasing the neighborhood can dramatically improve the accuracy of the costs determined during the search by providing a more precise path.

3.6.2 Memory Storage

Storage in contemporary computers is not typically difficult for problems described in this chapter. Frequently however, commercial autonomous and planning (e.g. [42]) systems are pressured to reduce the total cost of the system, including memory and computer hardware. It is therefore valuable to grasp the impact of certain representation decisions in terms of the memory space required.

The primary storage requirements for the A* algorithm are derived from the size of the configuration space and the heap management system. The configuration space requirements are the number of nodes times the size of a single configuration space 'node' structure. For a robot, the dimensions of a configuration space are defined by the number of degrees of freedom. In each dimension, the range of motion defines the limits of the configuration space, but the discretization of the range multiplies the size of the configuration space. In a non-specific example where there are R nodes of resolution in each of D dimensions, there are R^D nodes in configuration space. For a robot with six *degrees of freedom* (DOF), typically meaning the machine has six joints, with 100 discretized states per range, there are one trillion (10^{12}) states in configuration space. Therefore judicious discretization of the space is required. Although no longer perfectly optimal, it is also possible to decouple the problem into two sub-problems of 3 DOF each, typically dividing the first joints, which provide larger (transportation) motions and last joints, which involve the gripper and finer motion. Using the previous numbers, it is clear that $2x10^3$, is a preferable problem to solve.

As implemented in the two degree of freedom robot examples, the node structure is one character (1 byte) used as a flag, one integer to store the direction arrows (4 bytes) plus the cost as either float or integer (4 bytes). The configuration space for the two-link robot arm is 64x64 nodes, giving a total size of about 37K bytes.

The heap is the other major use of space. The heap must be large enough to store the largest number of OPEN nodes at one time. Each OPEN node, because it stores only the indices of the configuration state requires 2 short integers (4 bytes total). In the above examples the heap requires storage of about 4,000 nodes or about 16K total.

Even though the prime concern throughout this thesis it the time performance of the method, the relatively large storage requirements must also be of concern. On Sun machines and those with memory management allowing for paging of memory, the large size can be accommodated. Even still, the larger the memory requirements, the more paging is required. On (DOS/Windows) PCs, however, the memory size has historically been a problem, limiting the type of problems that will work there.

3.6.3 Sifted Heap

In the implementation of the A* algorithm, it was suggested in Section 2.4.1 that an indirectly addressed heap of nodes would be more efficient than storing each node and cost, possibly several times, in OPEN. The sifted heap permits each node to be stored in the heap only once, with the value of the node updated in the configuration space as the A* algorithm progresses.

To implement this, it was also observed that if a value changes in the configuration space, then the most proper action is to find the node in the heap, and reposition it in the heap. The difficulty is in providing an efficient mechanism to achieve this. In practice this requires a pointer from the configuration space to the associated reference in the heap. This adds O(N) storage locations. In addition, it requires that the heap management system update these pointers with every 'swap' of a node.

If the sifted heap is used without the repositioning overhead, this reduces the storage requirements for the maximum heap size, and therefore also reduces the time for each insertion and deletion within the heap. As an engineering choice, the savings in space and time must be weighed however with the number of nodes that are produced out of order, and the significance of those out of order nodes.

Experiments using the Euclidean cost measure have shown that using the sifted heap mechanism, approximately 5% of nodes are retrieved out of order, however unless the nodes happen to be adjacent, there will be no affect (one side of the growing circle will grow first on one side, then the other). If they are adjacent, the nodes will be recalculated with correct values, causing the recalculation of neighbors as well. In the Euclidean (H=0) example however, there were no recalculated nodes, making this a fast and efficient method. Depending on factors such as the neighborhood size and the overall maximum heap size, the sifted heap may or may not be a useful option.

3.6.4 Representation Issues Affect Timing

The worst case timing for this algorithm, not counting path following, is O(N log N) for N states in the configuration space. As seen in Chapter 2, A* is a one pass method with no computation for obstacle states. Therefore, obstacle strewn environments give faster results than those with no obstacles as is shown in Table 1. For robot examples, the number of states is determined by the resolution R (fineness of discretization which is assumed to be equal in each dimension), and the number of degrees of freedom D, so that $N=R^D$. For m neighbors, Time $\propto mR^D \log R$. Even though this is exponential, many practical problems have still been solved in acceptable time by accommodating the most essential factors of each specific problem. Some key factors that improve computation time are described next.

The nature of this method makes it more efficient for spaces with more obstacles, which are also typically problems that are the most difficult to perform by inspection. The resolution R is a tunable parameter that can be adjusted for regions of greater sensitivity, and has even been adjusted dynamically by Featherstone [15] and Verwer [61]. For robotic applications, D may be adjusted by observing that most machines have the greatest volume of motion in only 3 or 4 degrees of freedom, whereby the remaining degrees of freedom may be treated in a fixed configuration until the final manipulation is required.

For machines with larger numbers of degrees of freedom, major improvements can be made by decoupling the spheres of motion of the robot. Motions can be grouped according to the span of motion - smaller motions grouped first. This will allow separation of the problem into the major moving parts first (treating the smaller motion members together) with subsequent computing for the motion of the small motion members. In many machines, only a few degrees of freedom cause most of the motion, allowing this method to work. However the solution will not be globally optimal.

In real-time problems, the environment is not static but ever changing. Obstacles and goal opportunities arrive and depart over time. To keep pace with the changing environment, the planner must adapt the plan as quickly as possible. In the most current hardware, a 10,000 (100 x 100) state configuration space with an 8-connected neighborhood with no obstacles (worst case computationally) can be computed from scratch in 595 ms (~1/2 second) on a Sun-4 (SPARCstation IPX). A similarly connected 90,000 (300 x 300) state configuration space can be computed in 6177ms. Higher connected neighborhoods require more time, as will larger numbers of states. Even though these times may seem fast, more complex problems are easy to create.

The trade-off between the time, space and accuracy must be made separately for each implementation. In Table 1 there are timings for some of the above examples with different numbers of obstacles. The units are in seconds and were made on a Sun 3/260. With the more current IPX equipment, similar computations are virtually instantaneous (for example, 270 ms for the Min. with zero obstacles) but still follow the same pattern showing an improvement in efficiency as the number of obstacles increases.

Table 1: Timing for Some Computations (in seconds)

Cost Measure	Zero Obs	1 Obs	5 Obs
Min. Communication	2.3	2.1	1.3
Straightest	2.9	2.9	1.7
Min. Time	3.8	3.6	2.2

3.6.5 Time Bounds

A specific formula was proposed in Section 2.5.1 to describe the detailed behavior of various actions in the algorithm. If some of the values are quantified, they can be used to compute lower and upper time bounds for the application of the algorithm and the error. The upper bound will assume a zero heuristic such that all nodes are expanded. The lower bound assumes an admissible heuristic (Section 2.2.1) leading to the farthest node. Repeating the time expressed in Formula 1 of Section 2.5.1,

$$\text{Time} = (\gamma+1)\, b \sum_{t=1}^{T} \log_2 \Omega(t) + T(M(G+C+H) + K(\gamma+1))$$

then for a particular problem, the formula is a function of T, $\Omega(t)$ and many constants. The values for γ, and M are known or can be accurately estimated. The number of open nodes as a function of iteration ($\Omega(t)$), the constant for managing the heap (b), and total CPU time can be measured for a particular computer and application (configuration space, cost measure, etc.). This permits verification of Formula 1 and so, implicitly, of the assumptions it was based on. It also allows the accurate computation of the other constants (G+C+H). The time plot can then be generated for a variety of T's using the zero and admissible heuristics as bounds for a variety of examples of different cost criteria and heuristics.

The time measurement is now used in each of the following examples. The number of open nodes $\Omega(t)$, and newly opened nodes $m(t)$ are collected for each iteration t. The average of $m(1..t) = \gamma(t)$. The data collected for the actual times were at approximately equal intervals, and between seven and ten data points are graphed.

3.6.6 Euclidean Cost Measure (Min. Communication), Without Obstacles

Euclidean, H=0

Figure 34(a) illustrates the number of open nodes, $\Omega(t)$ (written |open(t)| on the plot) for every iteration t. This shows the regular growth of the number of nodes until approximately iteration number 2800, and then it decreases rapidly to 0. Because no specific start is given, all nodes are expanded.

The explanation of this is as follows. The search grows consistently from the center of configuration space toward the edges which affects the size of the heap. The heap contains the OPEN nodes which are those along the circumference of this ever growing circle. The iteration number gives the total number of states in the area computed in the configuration space. The circumference (i.e. number of OPEN nodes at iteration t) might be expected to be about 2*sqrt(OPEN(t)*π), but as we see in Figure 34(a), the value is a little over twice as high. This indicates that the 'thickness' of the number of OPEN nodes at the edge of the search is also slightly over two. This is sensible because the radius of the 16-connected neighborhood is slightly over two at its widest (i.e. sqrt(5)). The search continues smoothly until it begins to wrap around the space. This occurs when the radius is about 30. At 30, the number of nodes in the heap should be 900π, or about 2800.

The model also requires knowledge about the underlying γ value which describes the number of newly opened nodes at each iteration t. In particular, it must be determined if γ can be treated as a constant. The plot in Figure 34(b) shows that the average number (from 1..T) of newly opened nodes per iteration is between 1 and 2 for most values of t. Because the value is nearly constant for most of the range of T, it will be treated as a constant. The variation can be explained from the growth of the circle's circumference. For the geometry of the growing circle, the larger the radius, the fewer new nodes along the radius are uninitialized. As the radius becomes large, the curvature at the edge of the circle becomes almost flat. Therefore the new nodes required approach 1 to advance the edge forward.

The area where the model is predicted to be accurate is when P is small and given by:

$$\boxed{P \cong \frac{\gamma^2}{(\gamma + 1)\Omega(t) \bullet \log_e \Omega(t)}} \qquad (1)$$

For the Euclidean, H=0 computation, if P is about 0.02% (due to dropped terms), and γ is assumed to be about 1.5, then the formula will hold when the number of open nodes is 100 or greater.

Given the expression for time above, we can evaluate the time as predicted by the model compared to the experimental time. In this example, a goal is placed in the center of a 64x64 (4096) state configuration space.

The number of neighbor nodes is constant and known to be 16. For this case then, M, the determination of the m(t) successor nodes is the number of neighbors. The other values, G,C,H and K are constants since they perform a specific fixed computation.

Let the following represent constants:
$A = M(G+C+H) + K(\gamma+1)$
$C = (\gamma+1)\,b$

so that the computation time is expressed as the following equation:

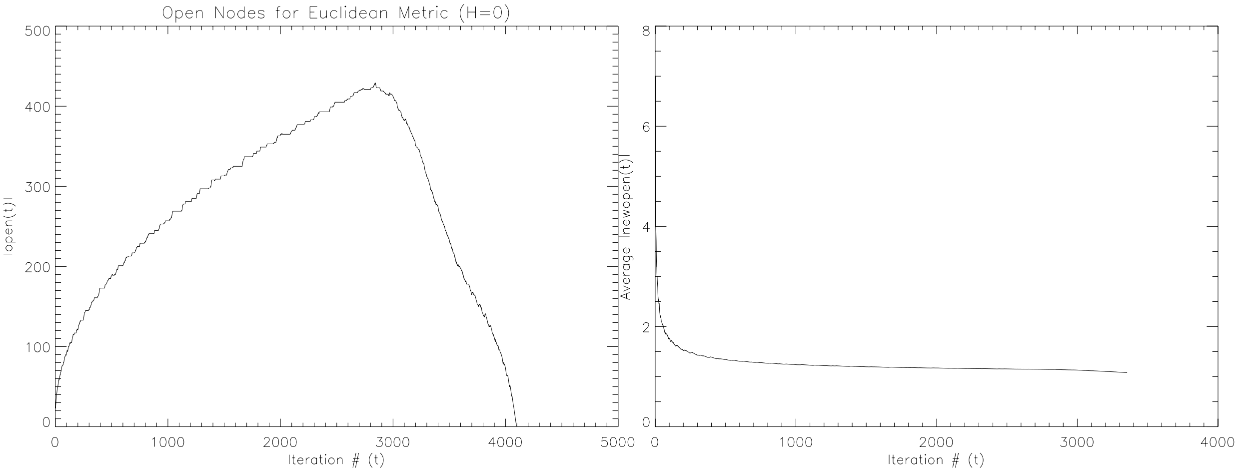
$$\text{Time} = AT + C\,f(t) \tag{2}$$

and $f(t) = \sum_{k=1}^{t} \log_2 \Omega(k)$

By performing several experiments, the values for the sum of logs, and A and C can be found for this specific environment including constraints, neighborhood, and cost measures. We fit the model using the measured points according to a standard method [35] (pages 523-524) to determine where the model is valid. The two values A and C then define the predicted time for all t. Based on this, the predicted time according to the model and actual time are plotted as in Figure 34(c).

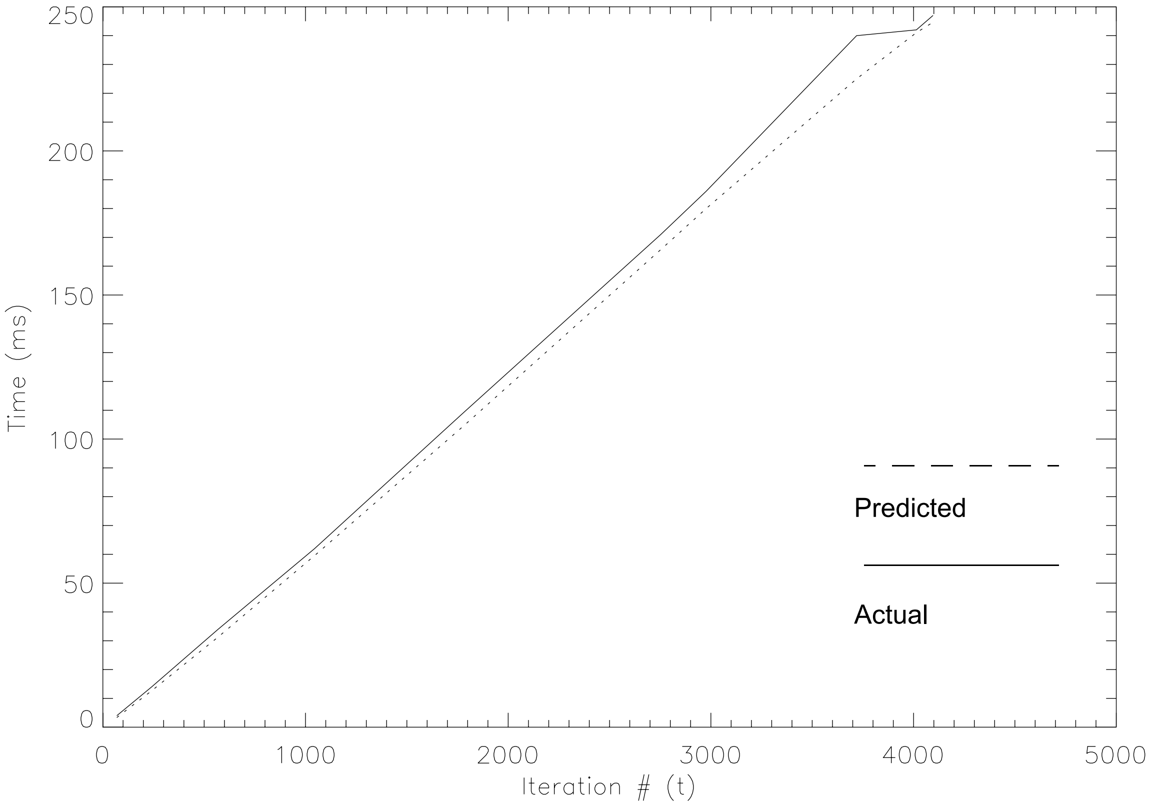
Figure 34(c) is a reasonable, but not perfect fit. One source of this error is believed to be the error in collecting the CPU time on a Unix system. Six trials were run to discover a sample variation in the system performance time. The deviation varied between 5% and 40% of the average time, however over several trials (typically 8-10), a ‘mode’ average could be obtained to minimize this affect. This was the data then used as representative data in the calculation of A and C.

It is also important to observe that although the order of the algorithm is O(N log N) time, the plots are straight. This is somewhat surprising, but indicates that the sum of the constant times greatly outweigh the heap management overhead for this size problem (4096 states) on a SUN IPX machine with 32 megabytes of RAM. As the size of the problem grows to larger than the memory size, we expect the value of C to increase as memory management overhead increases.



(a) Open Nodes, Ω(t)

(b) Average of Newly Opened Nodes, γ(t)



(c) Predicted and Actual Time. A = 20 ms. C = 5 ms.

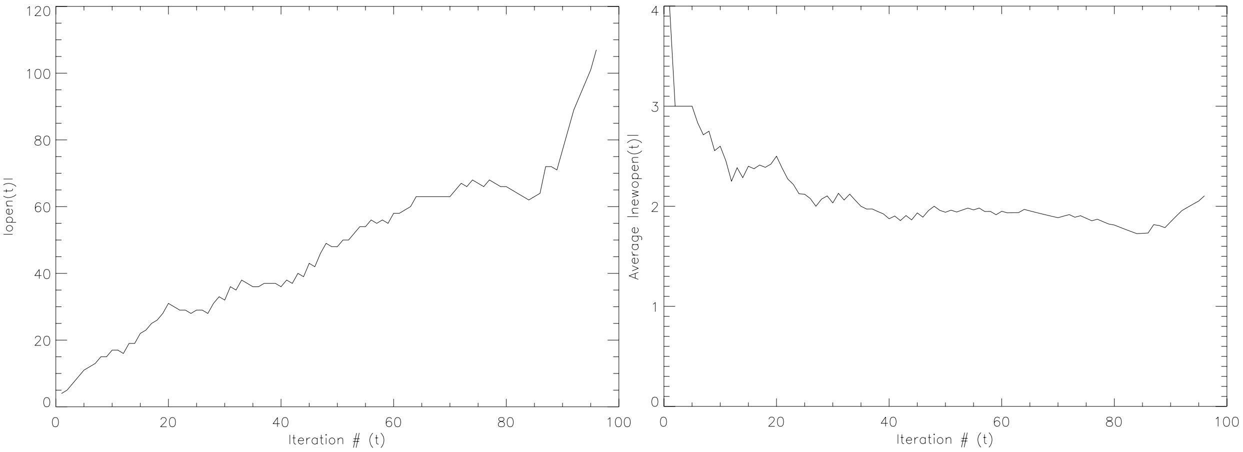
Figure 34: Euclidean Cost Measure (Min. Communication) H = Zero Heuristic.

Euclidean, H = Admissible

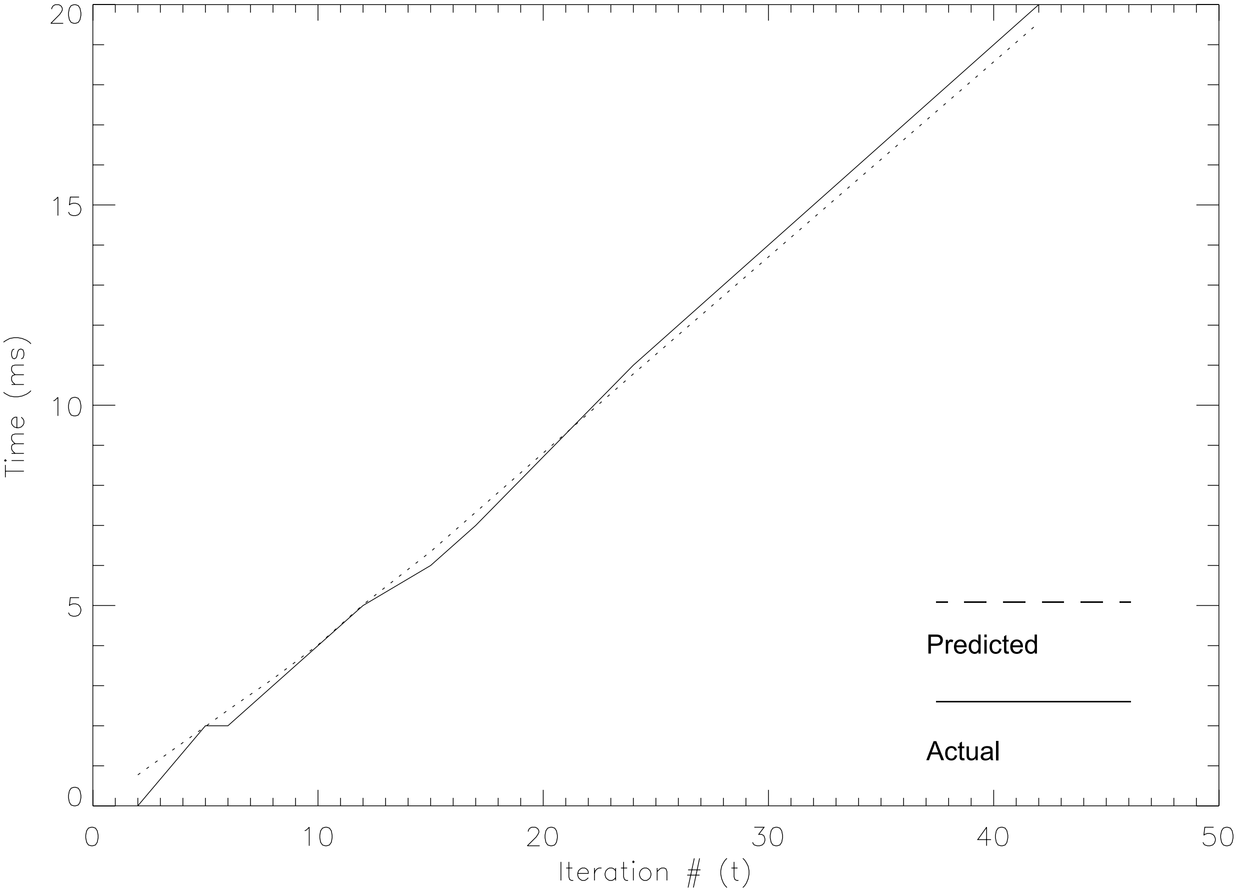
For the Euclidean cost measure in configuration space, with the admissible heuristic H, we see in Figure 35(a) the relatively steady growth of open nodes, with a drop and sudden rise just before the start is found. This is likely due to a series of unproductive nodes (shown in the middle of the path), and a final reach to the start.

The number of newly opened nodes, γ, is about 2 as seen in Figure 35(b), but the raw data (not shown) has large variations that are averaged out. The average is used as a representative value to give an approximate error for large t, however if the worst case is desired, then a value of about $\gamma=4$ would double each of the error estimates below. Because the number of open nodes is small, the error is more significant than in the h=0 case. When the error for P is about 6%, the model is expected to be valid for open nodes greater than 10, and 2% for the number of open nodes greater than 20.

The Euclidean cost measure using the admissible heuristic can also be predicted and then plotted with the actual time. These results are shown in Figure 35(c). The variation in actual time is fairly small, but because it is quantized in milliseconds, the discretization error appears larger than it might otherwise. That is, the measurement is coarse for this brief computation.



(a) Open Nodes, $\Omega(t)$ (b) Average of Newly Opened Nodes, $\gamma(t)$



(c) Predicted and Actual Time. A = 370 ms. C = 15 ms.

Figure 35: Euclidean Cost Measure (Min. Communication) H = Admissible Heuristic.

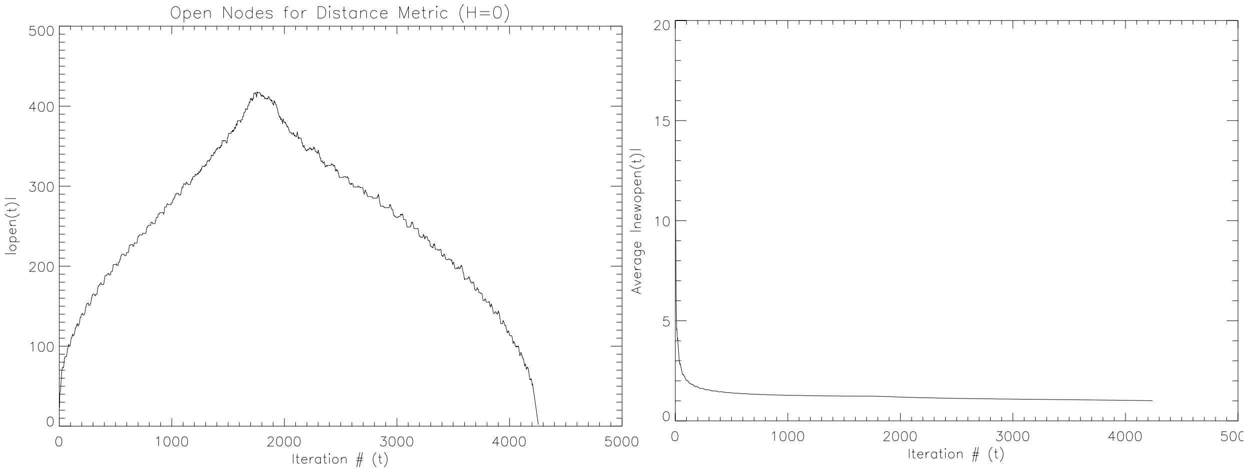
3.6.7 Distance Cost Measure (Straightest), Without Obstacles

Distance, H=0

For the distance (of the end effector) cost measure in Figure 36(a & b), the absolute number and average number of new nodes is similar to the Euclidean version. The perimeter of the field of arrows in Figure 36(a) however, grows faster earlier, and in a rectangular shape (covering more area than the growing circle of the Euclidean cost measure.

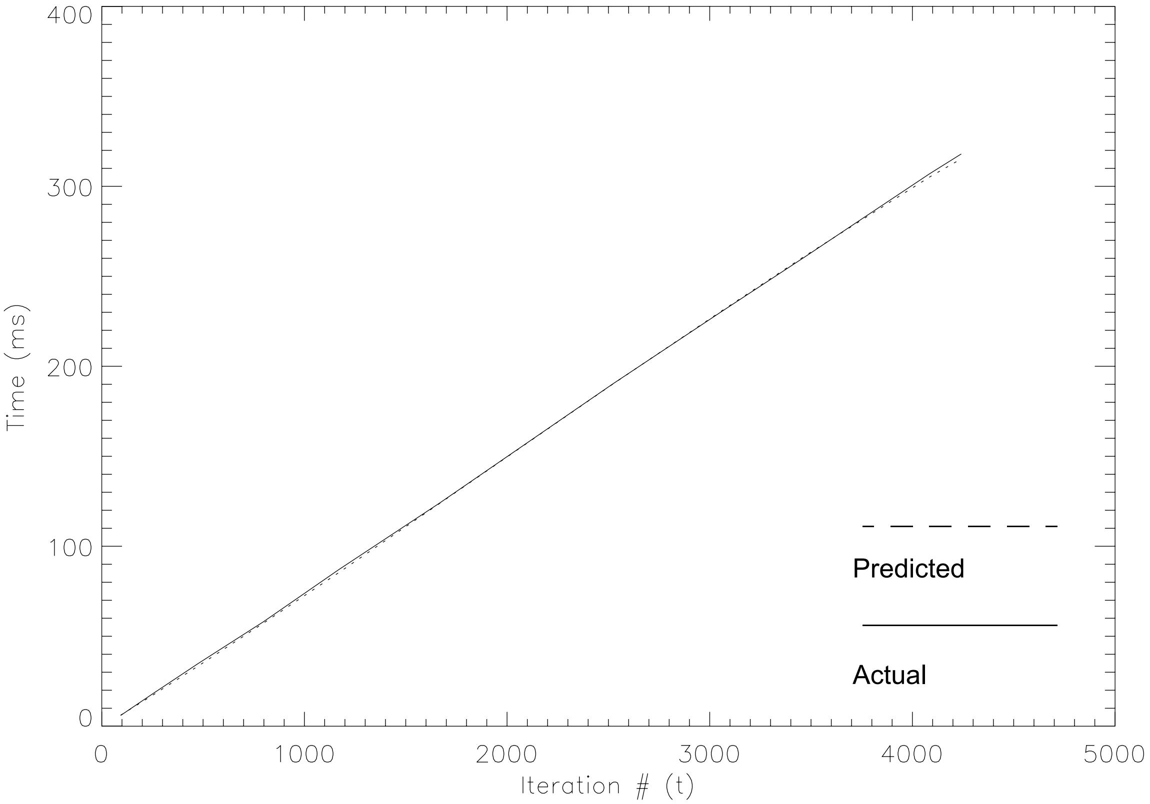
The number of newly opened nodes Figure 36(b) is found to compute the error. There is a small spike of new open nodes just before the t=2000 region when the transition cost is nearly zero, causing a |newopen(t)| of about 5, however this is barely noticeable in the average. Using 2, the error is the same as the Euclidean H = admissible case in Section 3.6.6.

Figure 36(c) shows that the model with constants A = 35 ms. and C = 5 ms. can very accurately predict the actual experimental time, even when the cost measure and expansion behavior is significantly different than the Euclidean case.



(a) Open Nodes, Ω(t)

(b) Average of Newly Opened Nodes, γ(t)



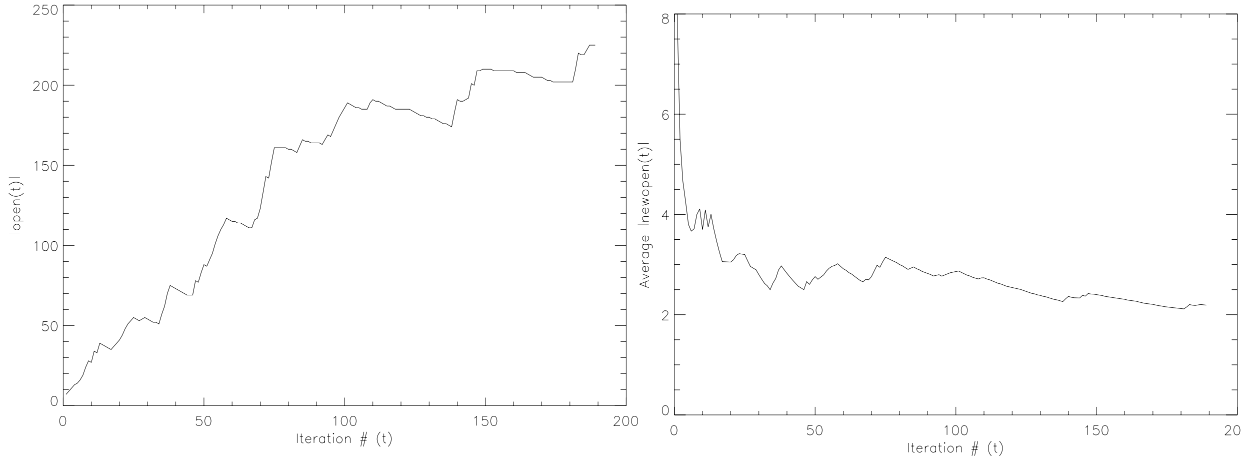
(c) Predicted and Actual Time. A = 35 ms., C = 5 ms.

Figure 36: Distance Cost Measure (Straightest Path) H = Zero Heuristic.

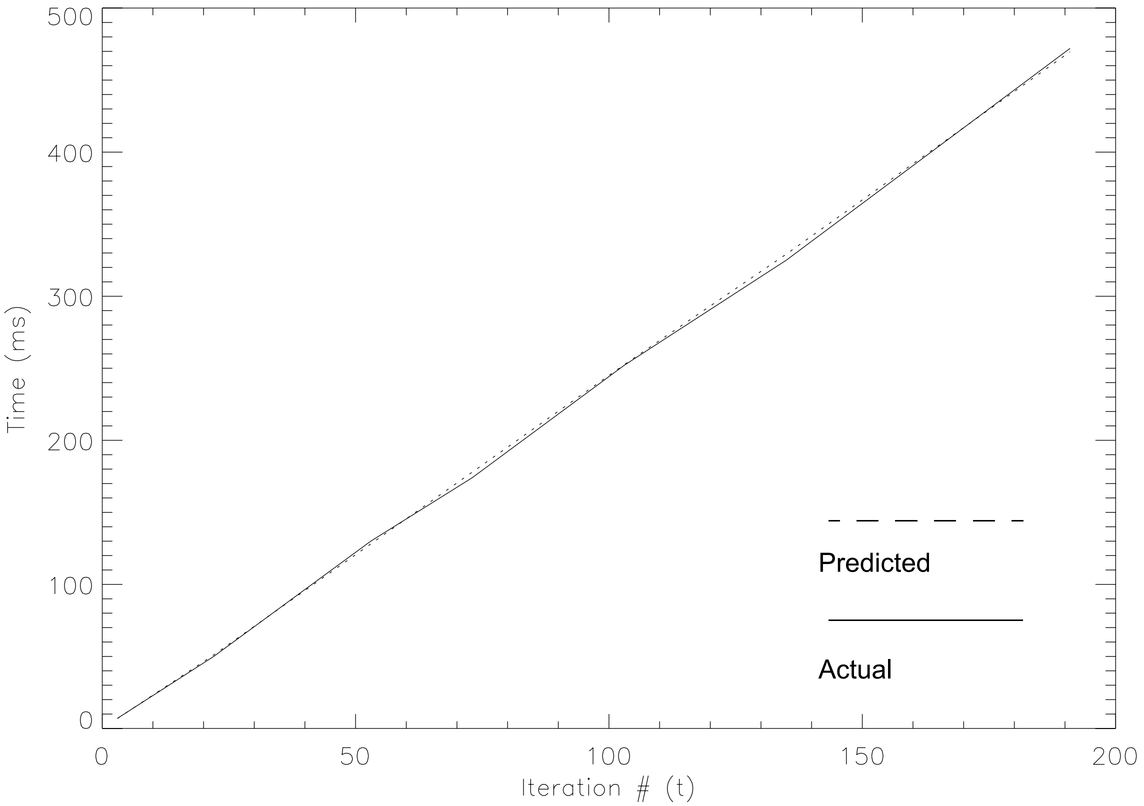
Distance, H = Admissible

This example uses the same starting node as Figure 36 but with an admissible heuristic. In Figure 37(a), the number of open nodes grows somewhat more erratically but with a clear increasing trend until the start is found. The number of newly opened nodes γ in Figure 37(b) is very erratic, but still seems to average out to about 2.5 for larger t. The error is then about 2.9% for nodes (t) over 20.

Figure 37(c) shows that the model with calculated constants A=225 ms. and C = 28.8 ms. can very accurately predict the actual experimental time.



(a) Open Nodes, Ω(t) (b) Average of Newly Opened Nodes, γ(t)



(c) Predicted and Actual Time. A = 225 ms. C = 28.8 ms.

Figure 37: Distance Cost Measure (Straightest Path) H = Admissible Heuristic.

3.6.8 Observations about the Model

The constants of A and C have a general trend and form a pattern. Both A and C are very small numbers, which should be expected since they are the overhead constants for the implementation.

Further, we can see that the A and C values for both of the h=0 are about the same. This is likely since there are the same number of neighbors in both cases, precomputed integer cost-measures, and similarly sized graphs. C is not surprisingly the same, since it is strictly a function of b, the overhead for heap management and γ, the number of newly opened nodes.

The admissible heuristic versions have significantly larger values of A than their h=0 counterparts. Both values of C are larger because the additional computation required in heap management, which must perform arithmetic operations for each swap of the heap to compute g+h = f. This is because the value of g can be changed in the configuration space, and not be reflected immediately in the heap (in the current implementation). The significantly higher value for A is due to the significantly larger (and differing) overhead for computing the function g in a non-precomputed way. The Euclidean measure has a smaller value for A than the Distance measure, because the straight line calculation requires two squares and a square-root. The distance measure, requires several trigonometric functions for each calculation of the function g.

3.7 Related Work

A standard algorithm for general path planning is Dijkstra's algorithm [10]. It is a general graph search algorithm that computes a *single-source shortest path* method that computes the path from a source vertex s to every vertex v. The method begins by labelling each vertex with a high (infinite) cost, successively expanding the least cost vertex until all vertices have been expanded. The description of the algorithm assumes that the graph G is represented by adjacency lists. If the least cost is determined by the use of a priority queue in the form of a linear array, then each extraction requires O(V), for V total vertices, performed V, times, or $O(V^2)$ operations.

There are many other methods and structures however for robot path planning, and each has different limitations. A selection is presented here, including the piano-movers problem, a visibility graph based configuration space, the quad-tree concept, a multi-resolution search based on orthogonal projections, swept bubbles, the variable cost measure, the constrained distance transform (CDT) and potential fields.

The *Piano Mover's Problem* as defined by Schwartz and Sharir [37,38] computes the *continuous* collision-free path that a piano must take in a two dimensional polygonal environment, or determines that no motion exists. Based on the number of walls (n), the algorithm requires polynomial time (i.e. $O(n^5)$). The method finds the set of possible motions, called critical curves outlined by the configurations causing collisions, and then defines rules when these curves may be applied. A search of the possible combinations of rules determines if a solution exists. This thesis is focused on computing in *discretized* spaces, which are more directly suitable for computer calculations since they can be mapped into conventional data structures (e.g. arrays, etc.).

The concept of a configuration space is essential to many of the planning applications. A survey of configuration spaces was presented by Lozano-Perez [24]. He used a 'visibility graph' to create the configuration space. This graph maps a graph of vertices that represent the corners of obstacles, between which travel is possible in a straight line. Fixed costs are allocated to each arc, and the graph is searched between the source and destination. One disadvantage to the visibility graph is that it must be re-computed each time the source or destination changes. It has been shown that this is an relatively expensive ($O(N^2)$) recalculation for a likely occurrence in robotics.

An alternative structure to configuration space is a *quad-tree*. This is a hierarchical, multi-resolution representation of two-dimensional silhouette images described by Samet [36]. This structure is obtained by recursively subdividing non-homogeneous regions into finer regions of equal size. The region is initially considered as a single node at the top of a tree. If it is non-homogeneous, it is subdivided into four regions, which become children of the node. Each region is subdivided until all the regions are homogeneous or a predefined resolution is reached. The leaf nodes are either filled (black) or empty (white). The non-terminal nodes are called mixed (grey). The complexity of a quadtree representation described by Hunter and Steiglitz [19], expressed as the total number of nodes in the tree, is O(p+q), where p is the total perimeter of the object(s) and q is the depth of the tree. An octree is a 3-D generalization of the 2-D quadtree concept, where the regions are recursively divided into eight equal voxels (3-D volume-pixels). In this case the complexity measure of p is the total surface of the objects measured by the surface area of the smallest cubes. This method is useful if the computation can be made in the task space as opposed to the configuration space of the problem.

Another *multi-resolution search based on orthogonal projections* of the (task-space) workspace is described by Wong and Fu [64]. This is a fast robotics path planning method to address the difficulty of the discretization of the obstacles. A 3-D collision test is decomposed into three 2-D collision tests, without and explicit reconstruction of the 3-D representation. A path is searched in a breadth first-fashion on multiple levels of the search space simultaneously. This method could benefit by substituting an A* search for the breadth-first search.

Swept bubbles are an improvement that can be used with the presented framework to create a more concise representation of freespace by breaking regions based on obstacles. The swept bubble concept described by Featherstone and Verwer [15,61] can divide recursively smaller spaces according to obstacle occupancy, thus interleaving obstacle computation and planning in such a way that the obstacle transformation is governed by a planning heuristic. Swept bubbles also provide a uniform geometric representation of the machine and the obstacle environment which provides quick intersection tests required for high-dimensional planning. This also provides a way to scale the problem to the amount of time available, because a coarse solution can first be computed, and then subsequently refined according to the time remaining.

Verwer [59,60] introduces a *variable cost measure*, or metric, which increases Euclidean costs by integer values in the region surrounding obstacles. This causes the path to not only avoid obstacles, but also avoid the suspense as the path leads near them. Because the method increases the cost of motion near obstacles, practical, safe paths can be obtained. This reinforces the motivation for using cost measures as a tool to create intelligent motion. Verwer's implementation, however, uses a bucket sort [9] which can only be used if the values of the cost measure are integers, and do not span a large range. Because this is the case for the integer filters upon which the method is originally based, it is an efficient (linear time) alternative for these types of problems.

The *constrained distance transform* (CDT) was originally used in the field of image processing, where relatively simple computations can be performed efficiently in hardware. In addition, the techniques described by Dorst and Verbeek [13] can be used in the field of path planning using a local masking operation to traverse each of the nodes in a regular grid so that the minimum of the locally accumulated costs can be found. The method requires a mask giving the cost difference for all local 'neighbors' to be swept repeatedly over the entire space until no change is made in the last pass, indicating that the solution has converged. The advantage of this method is that it can be processed in parallel in hardware. Using conventional computers however, the computational order for an n by n matrix of (N total) nodes is about $O(N^2)$.

The *potential field* technique is analogous to the magnetic property in classical physics, but has been adapted as a computer algorithm for path finding by Khatib [20]. This method does not provide a path that is globally optimal, nor does it always finds a solution if there is one, because it can get stuck in local minima. The method works by continuously controlling each link's closest point to the obstacle. Potential fields do have an advantage however in that they are fast and have the ability to work on local information only. Because it is fast, it also provides a way to quickly respond to changes, such as moving obstacles. In many ways, the potential field technique is a nice complement to the global path planning technique because it can operate more responsively at a level nearer the hardware such as for reflex control. Finally, potential fields become more important when the knowledge of the workspace is incomplete, because it provides a quick and relatively informed response compared with simple servo rules.

3.8 Chapter Summary

A* planning in configuration space solves for a globally optimal path that minimizes cost without entering illegal states (such as obstacles). It will also find a solution if one exists. In addition, if there is no solution to get to the goal, then this is immediately known because there are no direction arrows, and a higher control authority can be summoned. Unlike the potential field technique, there are optimal-direction arrows at all reachable states. This is useful when controlling a device or issuing commands that are not carried out perfectly. For machines in the field, unexpected slippery terrain or failing gear mechanisms could cause such problems. In this event, if the controlled device falls off the optimal path, then a new path can be found from any state without further computation.

As we have seen, there are several elements that are required in the framework: a problem configuration (state) space, a neighborhood, a cost measure on the neighborhood, a transformation of the obstacles/forbidden regions, and a goal set.

We transform the problem into an easier space in which it can be solved. The problem can be encapsulated into the kinematic control parameters (usually the joint angles) and the geometry separately. The planning takes place using the joint angles so that what is computed is a path describing a series of setpoints for the joint angles. The solution is then transformed back to the original ‘task’ space where it is carried out.

The resulting traversal of all reachable configuration states yields pointers directing motion toward the goal. This is particularly useful for robots because the planning may not have taken into account dynamics or the robot may not be able to follow a predicted path due to mechanical wear. If the entire space is covered and the robot fails to track perfectly, then it can begin from any current state, and try to move optimally toward the goal. In addition, if there are no direction arrows at a starting state, then the path to the goal is blocked.

For each newly added obstacle configuration, a new wave propagation using A* is required. Although this can be performed quickly (< 2 seconds on a Sun 3/260 for the example of Figure 25) for two degrees of freedom (DOF), the fact that the algorithm is exponential in the number of degrees of freedom motivated us to find improvements that are also a function of the degrees of freedom. One such improvement is specifically targeted at adaptation to changes in the robot environment. We call it the Differential A* Method, and it is the subject of Chapter 5.

The time required for a given A* algorithm on a specific machine and graph is characterized by a formula: Time = AT+C f(t), where the constants A and C can be estimated based on previous experience. The values for A and C are can be used to predict the time required for similar problems based on the graph structure, the number of successors, and the computation style (i.e. table driven or computed on demand) for functions. The admissible and zero-heuristic versions of the solution differ in the relative speed of computation, particularly because the admissible heuristic computes the heuristic on demand, whereas the zero-heuristic relies only on precomputed (table) entries.

The total time is a function of the number of nodes that must be opened before the solution is found. There are far fewer open nodes for the admissible heuristic than the zero heuristic version. We also show that the number of OPEN nodes is related to the surface area of the growing space (with h=0). This surface area is related to the radius of the 16-connected neighborhood which is sqrt(5). We may also expect other A* computations with the same neighborhood size to have sqrt(5)*surface_area states in OPEN, regardless of the cost measure or heuristic.

Finally, the framework described in this chapter is applicable to other complex robots such as a car. The car has a different set of permissible motions and a more complicated representation of the vehicle is required. The method of treatment for this type of machine, including a description of an experiment to control it is given next in Chapter 4.

Chapter 4

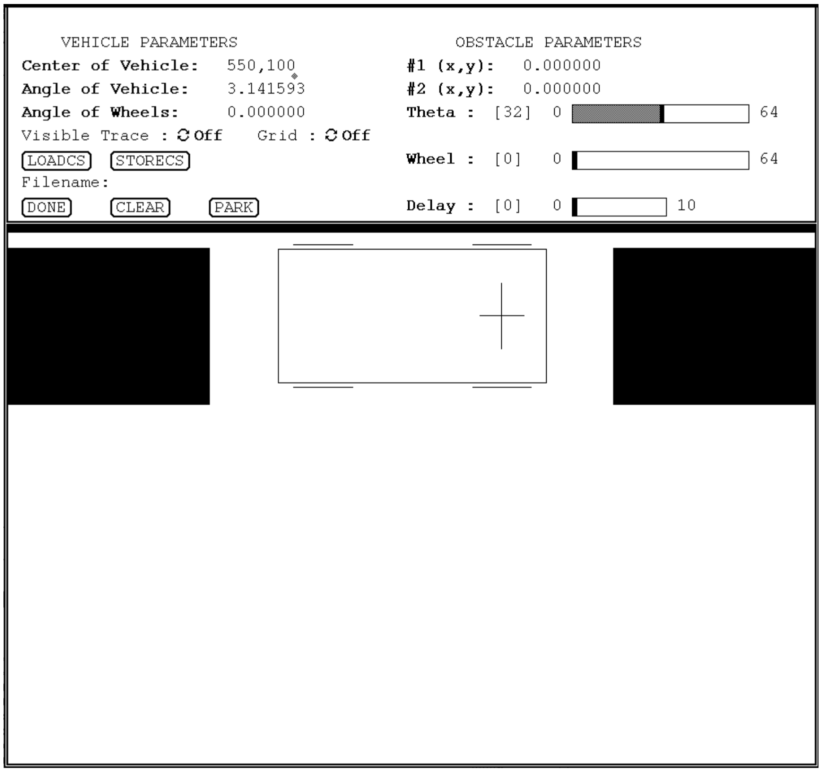
Autonomous Vehicle Maneuvering

The framework described previously was applied to a two degree of freedom robot arm (in Section 3.5). To show the breadth of this method, another type of moving machine, a car, is addressed using the same framework. The key challenges are to properly identify the configuration space graph, the neighborhood of the car, and a fast way to transform typical obstacles. Issues connecting the planning results to the controller are raised as the theory is put into practice, as a radio controlled car is modeled and controlled. Finally, the method is applied to a bulldozer and used for high-speed vehicle control.

Figure 38 shows the full snapshot of an early version of the user interface which illustrates the car and its environment and provides a mechanism to specify the goal and obstacle positions. A simulated vehicle is placed inside a predefined rectangle corresponding to the actual testbed. Obstacles (parked cars) can be reshaped in the corners, and the motion is constrained to the limits of view. As the vehicle moves, the configuration data is reported, and the user may present the motion as an animation, or display the trace of all motion. The goal position (and orientation) of the car is set by a button, and then the car is dragged and rotated to the desired starting position. The starting pose can be reset to see the identical action several times.

4.1 Path Planning for a Car - Vehicle Maneuvering

The path of the vehicle, called a maneuver [32,48,49,50,52] can be determined by using the A* framework given in Chapter 3. The dimensions and steering capability of the vehicle must be known in advance. An example vehicle is given in Figure 39. To use the framework, each of the components must be properly constructed. A more elaborate description specific to the vehicle maneuvering implementation follows.



```
    VEHICLE PARAMETERS                        OBSTACLE PARAMETERS
Center of Vehicle:  550,100          #1 (x,y):   0.000000
Angle of Vehicle:   3.141593         #2 (x,y):   0.000000
Angle of Wheels:    0.000000         Theta :  [32]  0        64
Visible Trace : Off    Grid : Off
                                     Wheel :  [0]   0        64
Filename:
                                     Delay :  [0]   0    10
```

Figure 38: Simulation Interface.

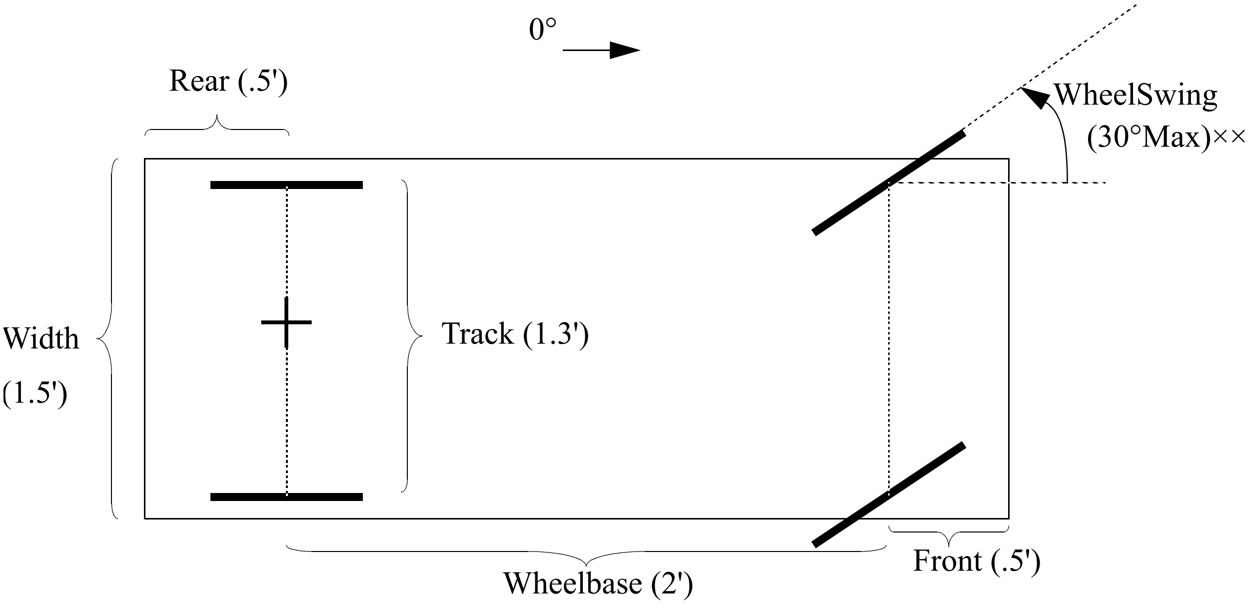


Figure 39: Example Vehicle Dimensions.

Configuration Space

Although the vehicle is *controlled* by the orientation of the steering wheels and the transmission direction, the *configuration space* is defined by a different range of parameters (as discussed in Section 3.2.1). The three parameters minimally necessary for describing the state of a vehicle are the x-y position and the angle. This is different from the control parameters which cannot uniquely determine the *position* (status) *of the car*. For a vehicle such as an ordinary front steering car, the center of the rear axle (shown by the cross-hair (+) in Figure 39) is used to define the x,y position and angle θ relative to a world frame of reference. This location is selected because this is the position where the center line of the car is tangent to the circle formed by the vehicle's turning radius. It is helpful in the implementation because the change in angle is symmetric for both forward and reverse directions and provides the widest angle changes as motion progresses. The x, y, θ parameters span a 3 dimensional configuration space. The maximum x-y positions of interest (i.e. the permitted area for the car to drive) define the ranges of the configuration space in x and y. Since a vehicle commonly has the ability to be oriented in any direction, the range is 0 to 2π, and can clearly can wrap around periodically. An example of the topology of such a configuration space is shown in Figure 40, although it is also often illustrated as a cube where the axis θ is assumed to wrap around (as in Figure 43 and Figure 46). For this example, there are 36 x 24 x 64 states for each of the x,y, θ dimensions respectively, for a total of 55296 states in the configuration space. The actual size of the simulated space is 9'(2.74 meters) long by 6' (1.82 meters) deep, and is discretized into 3 inch (7.6 centimeter.) squares. The angle is discretized into approximately 0.1 radians per state.

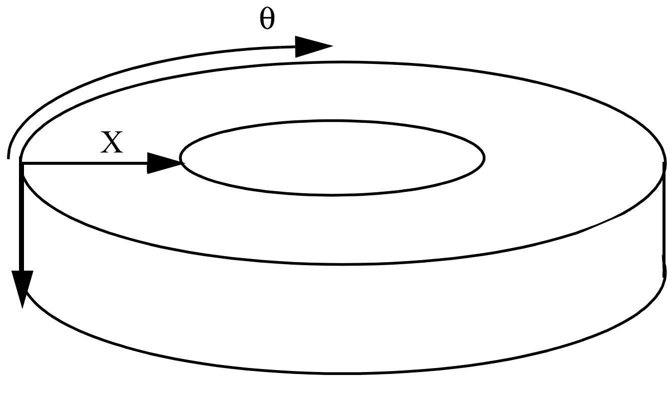


Figure 40: Topology of the 3-Dimensional Configuration Space.

As an implementation detail, for problems with configuration spaces larger than the RAM memory size, it is important to arrange the configuration space with indices in the order that they are likely to be accessed. For example, since the transformation is computed 'slice by slice', the indices should be [θ][x][y], and not [x][y][θ]. This seemingly simple change reduced the transformation time (on a Sun 3/160) from about 4 hours to the current few seconds.

Neighborhood

The *neighborhood* represents the feasible motions assuming no obstacles. The car is kinematically constrained and not able to move arbitrarily in any of the 3 dimensions (i.e. it cannot jump sideways), but its total motion can be described as a two dimensional shape as in Figure 43. The method for generating this general shape follows.

In a simplified example, a vehicle could be considered to have only hard right, hard left, and straight steering positions for the front wheels in either forward or reverse for a total of six possible elementary motions. As other studies [18] have shown, optimal motion control in an obstacle free environment is achieved with only these extreme motions. This is also called *bang-bang* control, where the vehicle quickly switches from one steering position to another. Therefore this subset of six primary directions for discretized neighbors will be selected. For more natural travel, where the excitement of rapid trajectory changes is undesirable, more neighbors can be used, with a higher cost for the more extreme motions.

By examining the local range of motions for a vehicle, the trace of the measured point (mid-point of the rear axle) would give a symmetric bow-tie shape (as in Figure 42), if traced on the ground. Since the orientation θ of the vehicle changes, the neighborhood in the 3-D configuration space is actually represented as the *twisted bow-tie* shown in Figure 43. The individual neighbors are discrete locations along each of the six directions. Note that the ‘twisted bow-tie’ shape results from the change in angle per unit change in x and y. The shape of the neighborhood remains consistent regardless of the orientation, however for illustration purposes, the labelling of the x, y,θ axes in Figure 42-Figure 44 is shown specifically for θ=0°. The discretized neighborhood is a function of the track, wheelbase and maximum steering angle. The specific method to create this neighborhood is given in Appendix B. The mathematical relationship between x, y, and θ is elaborated for the continuous case in [14]. These are relatively simple when computing changes from the origin for example.

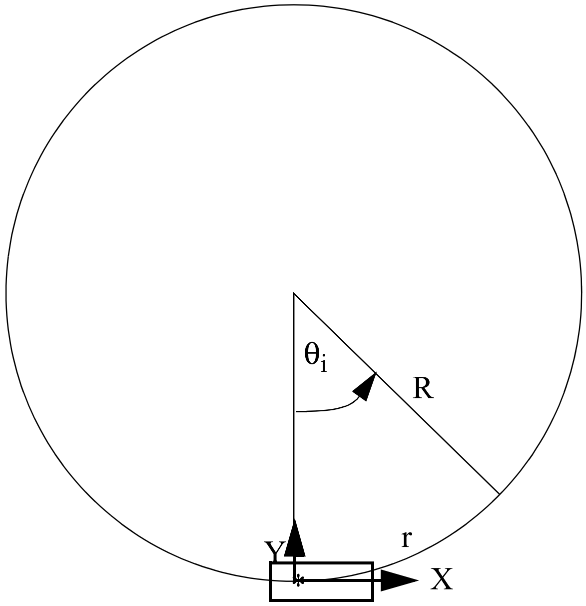


*Figure 41***:** *Fundamental Motion of a Vehicle.*

In this case, the control variables are:

The angle of the steering wheels Θ
and
The velocity v, which will be constant

When the vehicle moves along the circle, the traveled distance $r = |vt_i| = R\theta_r$

For $\Theta \neq 0$, the new position, x_i, y_i, θ_i can be calculated according to the formulas:

$X_i = R \sin \theta_i$

$Y_i = R (1 - \cos \theta_i)$

$$\theta_i = \frac{vt_i}{R}$$

The turning radius, R is:

$$R = \frac{track}{2} + \frac{wheelbase}{|\tan(\Theta)|}$$

For $\Theta = 0$, the new position, x_i, y_i, θ_i can be calculated according to the formulas:

$X_i = vt_i$

$Y_i = 0$
$\theta_i = 0$

and (distance) cost measure = $r = R\,\theta_i = vt_i$

Assume that the steering wheels have three positions: $-\Theta$, 0, $+\Theta$, and the velocity is constant in a positive or negative direction, we obtain the *bowtie* shape within a range for X_i where,
$-X_{max} < X_i < X_{max}$

These equations also form the basis for Appendix B.

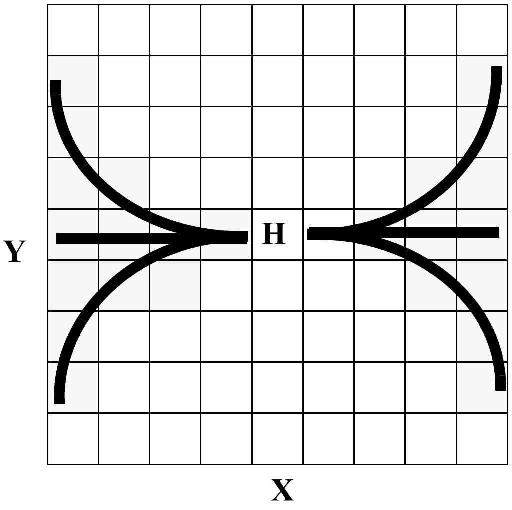


Figure 42: Example Trace of Neighborhood.

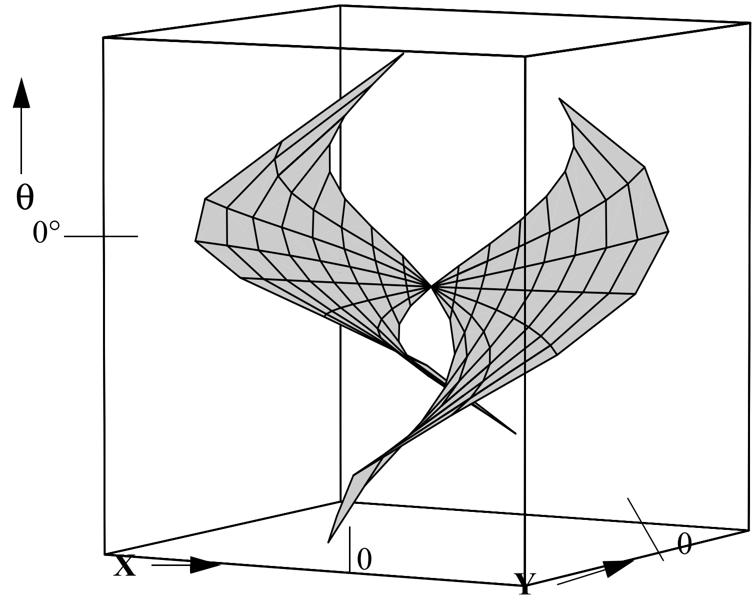


Figure 43: Example Neighborhood in 3-D Configuration Space.

The size of the neighborhood must be chosen to be sufficiently large so that changes in configuration parameters (x, y, and angle) can be seen for all discretized angles of the car. If the turning radius is large, such as for a bus, the neighborhood must also be large, so that incremental parameter changes occur in the neighborhood. The neighborhood must be at least large enough to cause changes, but even larger neighborhoods can be used to cover the entire helix (in 3 space) of possible neighbors. As observed in Section 3.6.5, the size of the neighborhood will increase the number of OPEN nodes at the edge of the search. This will not only increase the number of nodes in the heap (increasing the time for insertion of any state) it will change the order in which the space is computed. For example, if the start is precisely 1/2 the turning radius from the goal, the full helix will find the solution with the first expansion. The shorter neighborhood (with a zero heuristic) will first compute all states with lower distances than the start, but will expand more nodes per unit time because of the ‘thinner’ edge of OPEN nodes. If all states in the configuration space are to be expanded, then the most efficient structure is the smallest neighborhood for which changes in each of x,y,θ occur (for every car angle θ). For the vehicle in the simplified example of Figure 47, we have chosen a 9 x 7 x 90° (X x Y x Angle) neighborhood (for car angle θ = 0°), which is rotated and discretized for other angles.

Cost Measure

A simple optimization criterion is the minimum distance traveled in task space. In this case, the length of the arc or straight line for each neighbor in the neighborhood represents the cost of the motion, as measured at the cross-hair. The specific method to compute the distance is given in Appendix B. This distance is different from the cost from center to center distance, because these centers imply a different turning radius. The cost distance to each neighboring configuration state will therefore be accurate. A more complex criterion might add a cost penalty for any x,y,θ configuration state which places the vehicle in a danger zone as a delay cost on the node, such as the street area when parallel parking. Alternatively, a penalty for turning the wheels can be imposed. The current example will only consider minimum distance traveled. If a neighborhood is provided so that each neighbor points to a different discretized state, then it will be called non-redundant. A sample, non-redundant neighborhood and its associated costs are given in Figure 44. It is shown as a projection for the vehicle orientation, θ = 0°.

Neighbor #	Cost
1,11	.25
2,12	.50
3,13	.75
4,14	1.0
5,8,15,18	.502
6,9,16,19	.757
7,10,17,20	1.017

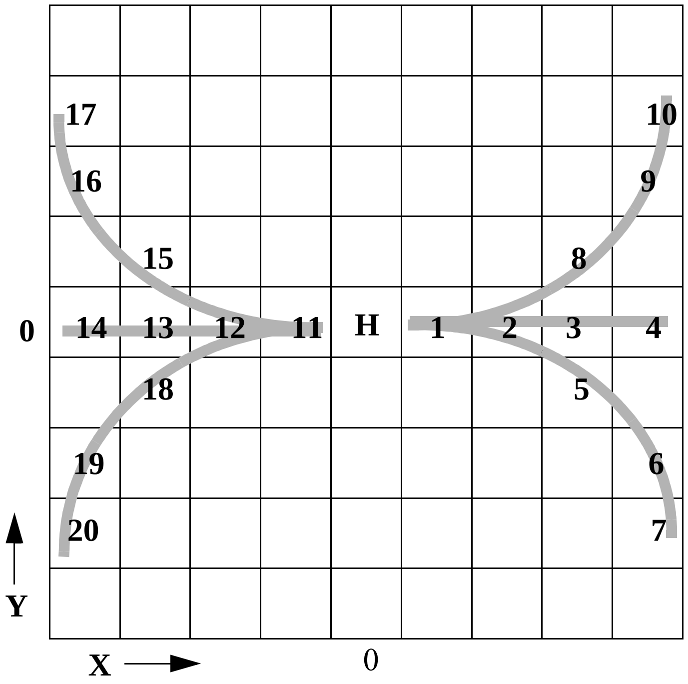


Figure 44: Example Neighborhood and Associated Costs θ=0°.

Constraint Transformation

The transformation of 3-D obstacles for vehicle maneuvering can be simplified if it is assumed that the downward projection of any obstacle to the 2-D floor from the maximum height of the vehicle determines the illegal zone. In the examples given below, obstacles are first enclosed by a rectangular area and then transformed. The controlled vehicle itself is also approximated by a rectangle. In Figure 45, two parked cars and the curb (assumed along the top) are transformed based on the dimensions of the controlled vehicle and is a function of the angle. This is achieved by determining the outer region where the body of the controlled vehicle would intersect the parked cars or the curb. Since the position of the vehicle is measured by the center of the rear axle, the region is forbidden if this center would cause a collision. An outline of the forbidden region can be generated by ‘tracing’ along the convex and concave corners of the obstacles with the shape of the controlled vehicle. This can be done by simple geometry, and is most efficiently implemented by separately computing the illegal region of x,y for each possible angle. The forbidden region may then be completely filled for each angle. The transformation for one angle is a slice of the full transformation for all angles. Figure 46 shows the full transformation of the configuration space, which is computed fairly quickly (50ms on a Sun IPX).

In the simulation, it was found that it is sufficient to fill the configuration space region with a thickness of three states to the inside of the transformed shape, rather than take the time to fill the entire region. Once the vehicle cannot cross this boundary it is irrelevant if it is filled. This is because of the precedence of neighborhood blocking discussed previously in Section 3.4.1 and illustrated later for this example in Figure 47. It is <u>not</u> sufficient to leave only a single state thickness however, since the neighborhood is shaped so that it can penetrate the hollow obstacle at an angle (i.e. between the slices). The three state thickness ensures sufficient overlap between one angle transformation and the next.

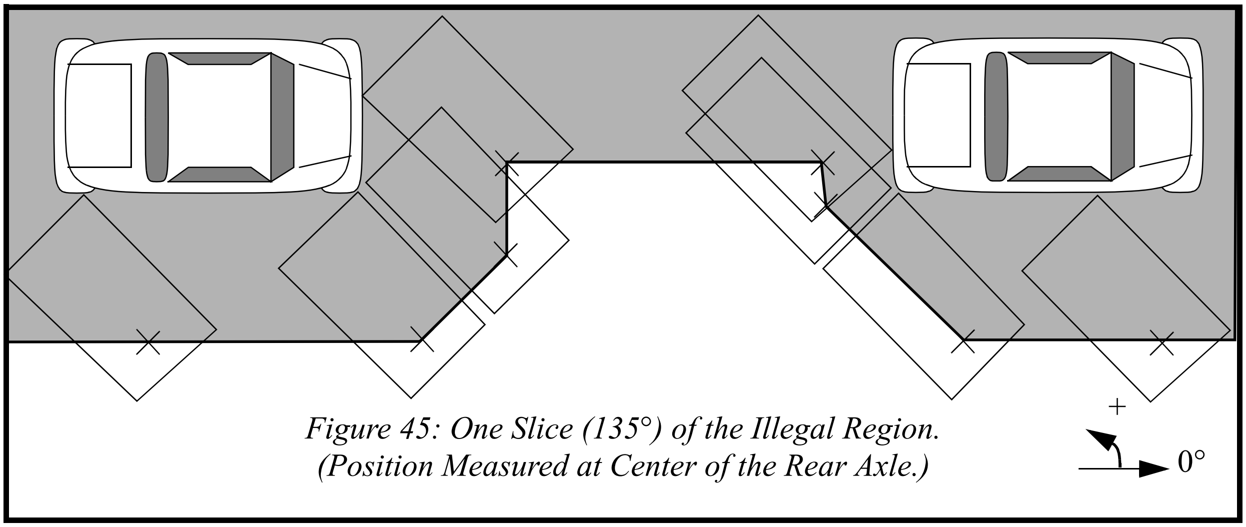


Figure 45: One Slice (135°) of the Illegal Region. (Position Measured at Center of the Rear Axle.)

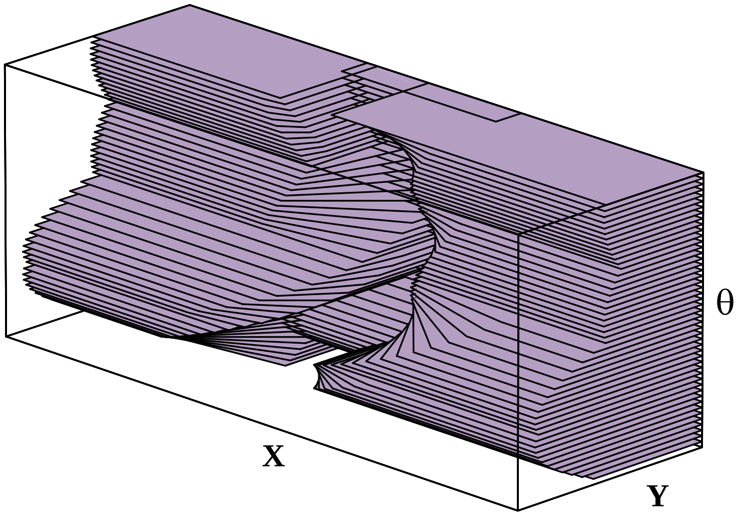


*Figure 46***:** *Full Transformation of Parked Cars in 3-D Configuration*

Start and Goal States

The *goal position* is usually simple to transform. If the goal is to be parallel parked between the two parked cars, in a specific orientation (e.g. parked in the same direction as they are) then the transform of the goal x,y coordinate of the car and orientation can be given directly. If there are several parking spots to choose from, then each could contain at least one goal. In the current example, only one goal is used.

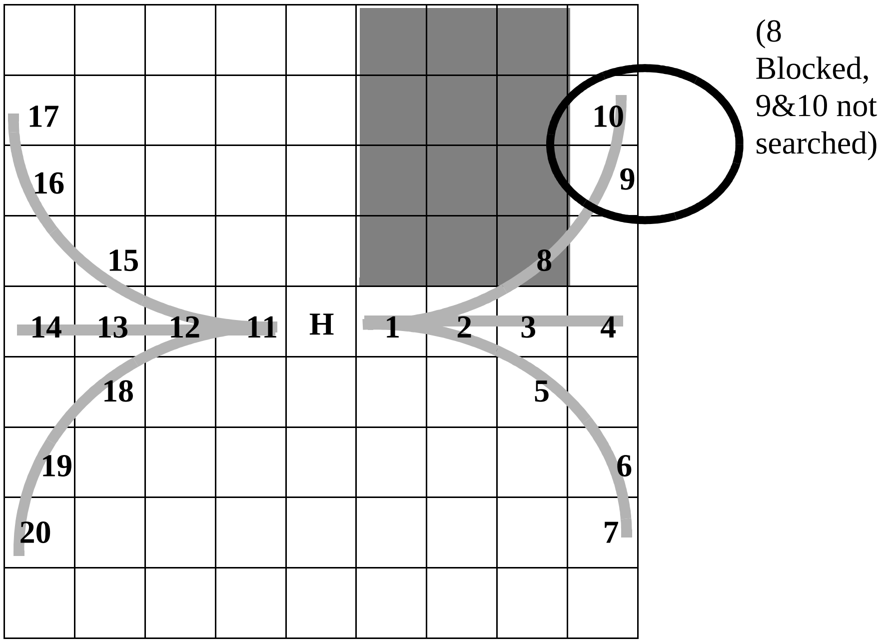


Figure 47: Later Neighbors Not Searched if Prior Neighbor Obstructed.

4.1.1 A* to Compute Costs and Paths

Once the elements are provided, the configuration space, obstacle transformation, neighborhood cost criterion, and start and goal are known, the A* method can be used with the use of precedence in the neighborhood as introduced in Section 3.4.1. In the vehicle maneuvering example, the neighborhood has a precedence and must be evaluated so that the nearer neighbors along a particular direction are evaluated first. If a near neighbor is blocked by an obstacle state, then the remainder of the neighbors are not explored (see Figure 47). This feature is necessary to avoid collisions with convex corners such as the one shown in grey. If precedence is not used and neighbors 9 or 10 are searched and selected because they are unobstructed, then this leads to an incorrect implication that a safe path between to the original home (H) position is possible (and thus the path would be incorrect).

While it is difficult to show the A* search expansion (similar to Figure 24) in 3 dimensions, the beginning of the search can be shown as a projection. Six sequential expansions (centered at home locations for H_1-H_6) starting at the goal G and using the neighborhood of Figure 44 are illustrated in Figure 48. By expanding the nodes least cost first, the covered area grows faster in the X direction than either the Y direction or angle. For each new expansion, a neighborhood is selected that is a function of the new angle. The figure shows only the continuous character of the underlying neighborhood before discretization.

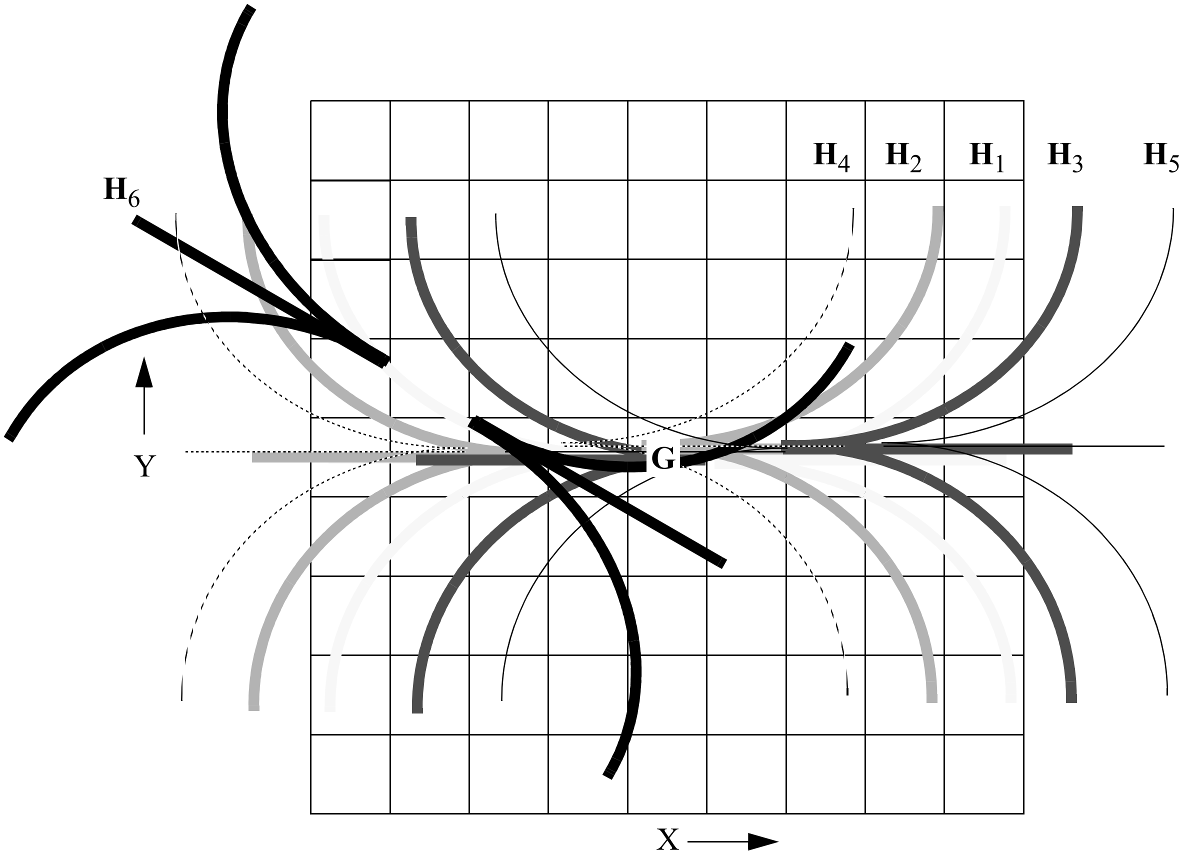


Figure 48: A Expansion for Vehicle Example.*

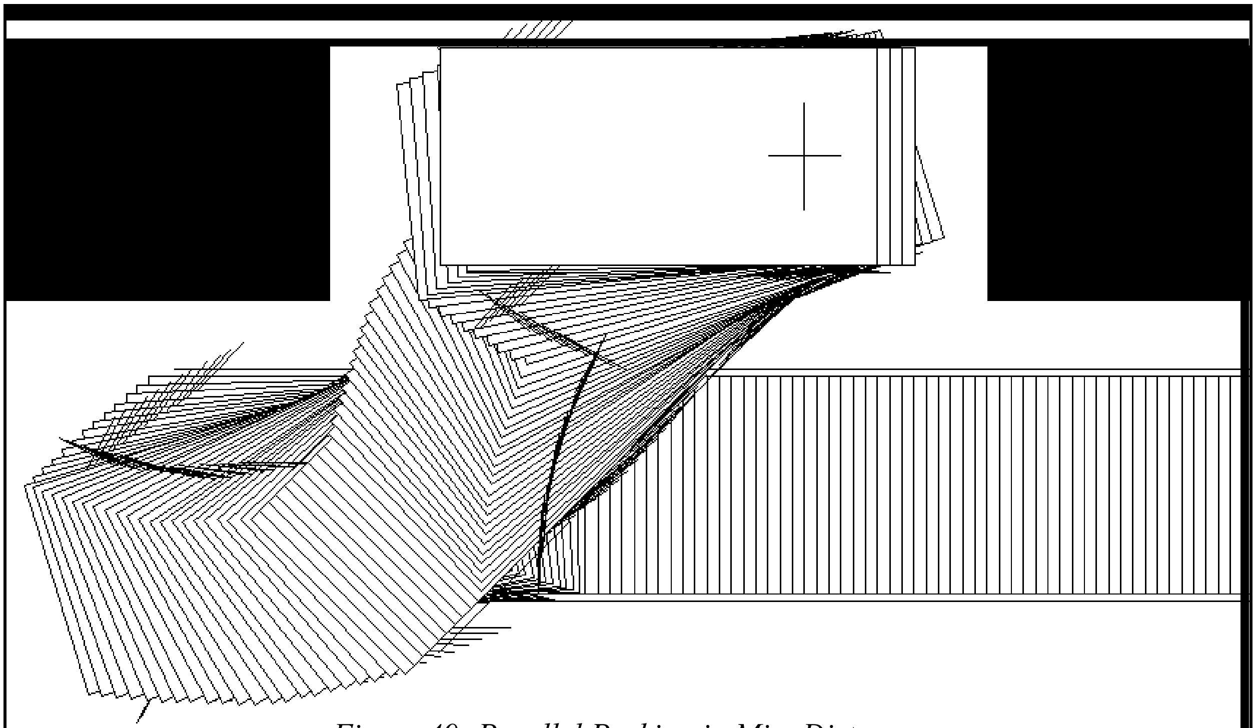


*Figure 49**:** Parallel Parking in Min. Distance.*

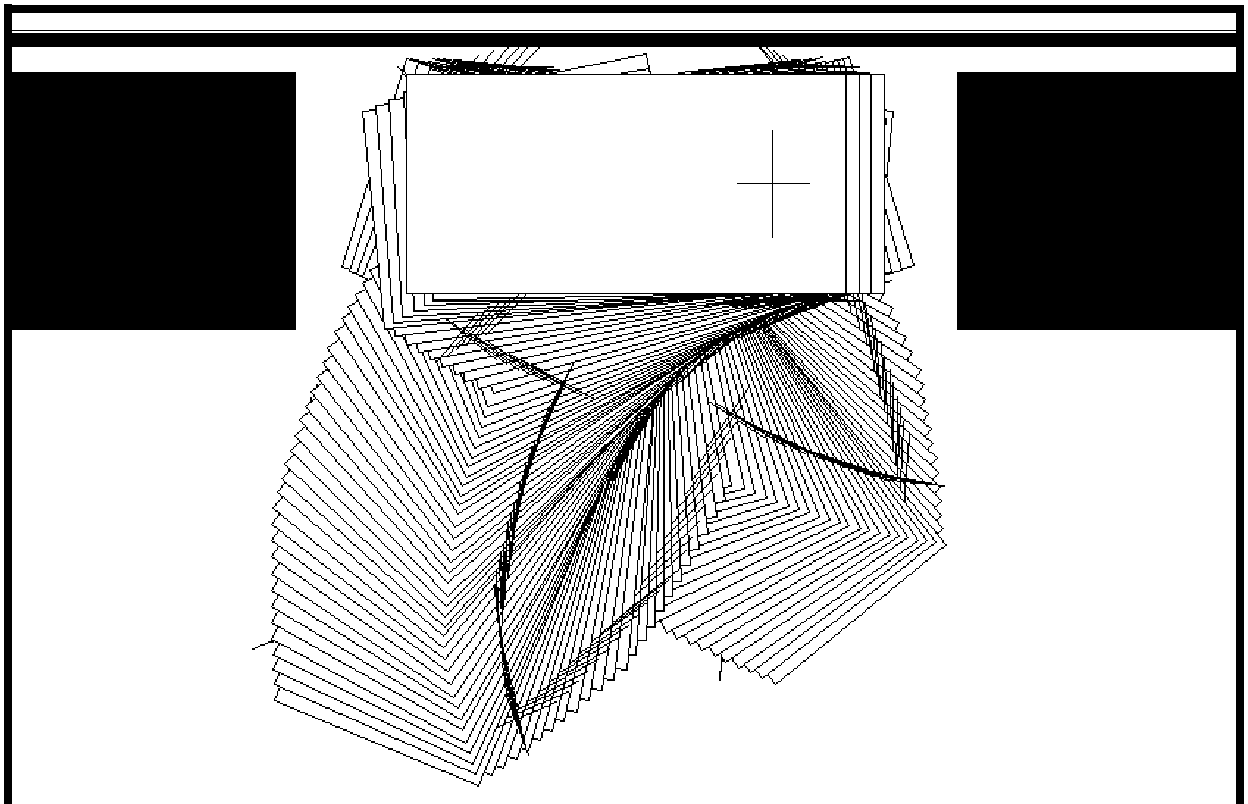


Figure 50: Starting Reversed.

4.1.2 Planned Solutions

The result from the A* method is a direction arrow in each reachable configuration state indicating the steering wheel orientation and transmission direction (i.e. forward or reverse) for the vehicle. The vehicle does not have any dynamics factored into the plan, and it is assumed to move at a fixed speed with instantaneous acceleration. By following the arrows from any starting state, the minimum distance path will be traveled to the goal. Figure 49 shows the path resulting from a minimum distance criterion starting from a position parallel to the rear parked car. For animation purposes, we interpolate linearly between state transitions to produce a smooth motion between longer neighborhood ‘setpoints’. Finer discretization could not resolve this, since the longer neighbors of the neighborhood would still require interpolation. This introduces some error which could be removed if the interpolation would match the curvature of the transition, but this animation does not. There are between 4 and 13 interpolated images for every neighbor (pointer) that is traced, producing plausible animated behavior with a minimum of computation.

Of course parallel parking is not the only maneuver that can be computed. From any starting state a path to the goal can be traced. Figure 50 shows the maneuver necessary if the controlled vehicle is started in the parking spot backwards.

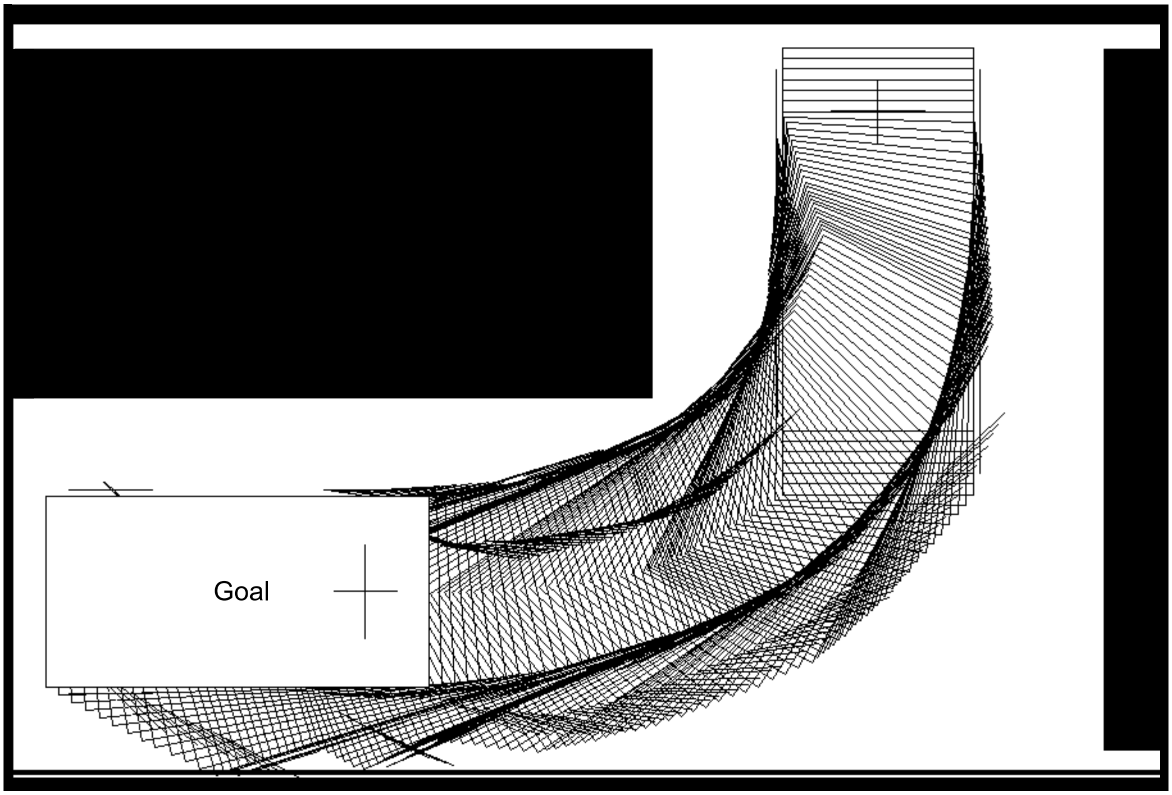


Figure 51: A Right Turn with Forward-Only Constraint.

4.1.3 Variation: Forward-Only Motion

If the above methods were used to make a right turn, it would likely lead to a maneuver with several transmission changes, because it is the actual minimum distance solution. This is obviously undesirable in ordinary traffic. This could be solved by greatly penalizing changes of transmission, although in this case it is simpler to eliminate the neighbors corresponding to the reverse direction entirely. This has the additional benefit of speeding A*, since there are fewer neighbors and because in a limited space, fewer locations are actually reachable. All other aspects of the setup and control are the same. Figure 51 shows a simple right hand turn constrained to allow only forward motions. The goal position must be set by the higher level planner (the simulation user in this case) along with the selection of this forward-only behavior.

4.1.4 Computation Times for Above Examples

The cost criterion for the vehicle's travel is distance. Direct computation of straight-line distance between locations is not a good measure without considering orientation. This is because the distance estimate does not embody the complex motion required to move in horizontal direction, for example. Because we have not found any admissible heuristic estimating distance traveled for the method, H( ) = 0 was used in all examples.

There are two times that can be reported for any specific situation. The shorter time is for computing the solution until the starting state is reached, i.e. delayed termination. This computes all states closer to the goal than the original starting position. The longer time is for filling the entire configuration space so that solutions are computed for all starting positions (even those farther than the original start). This has the benefit that when the vehicle is controlled, no additional planning time is required unless the obstacle configuration is changed. Reading the directions from the configuration space is virtually instantaneous, and the car can be repositioned and driven to the goal without delay.

The times given in Figure 52 below are in seconds of clock-time (not CPU-time), as computed on a relatively slow Sun 3/60. Experience with other A* demonstrations on a Sun SPARC IPX indicate that these times will decrease by about a factor of 10 on the IPX.

Figure	To Start	All States
Figure 49 (Parallel Parking)	45	90
Figure 50 (Reversal)	80	90
Figure 51 (Right Turn)	10	35

Figure 52: Seconds of Clock Time to Compute the Vehicle Configuration Space.

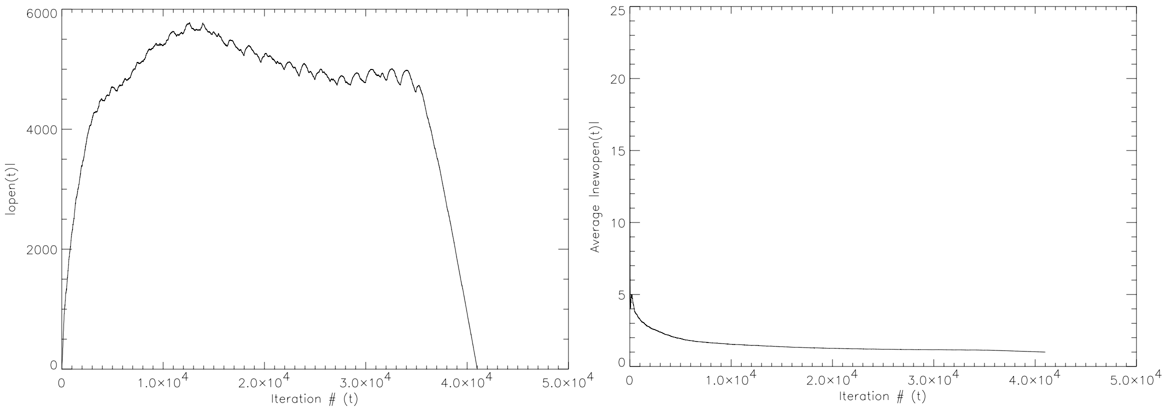
4.1.5 Time Constants for Model

Similar to the timing related to the number of OPEN nodes in previous examples (e.g. Section 3.6.6), the performance of the vehicle example will be explored. The vehicle environment is typically filled with obstacles, particularly near the goal state. To reduce the effect of the obstacles, the goal was placed in the center, and obstacles, except the (fixed) curb at the top of the image, were removed. This location was the area most unencumbered by obstacles. Even so, the bounds of the window cause an obstacle area.

Time constants (A and C) for the time model can be computed for the vehicle example. For a vehicle that is bounded to have the cross-hair remain in the window, we can compute the constants for A and C in equation (2) on page 64: Time = A T + C f(t). For this example, a zero heuristic was used and the full region was filled. The number of open nodes in Figure 53(a) grows dramatically to almost 6,000 at its peak. The total number of configuration states is large (36*24*64 = 55,296), and about 41,000 are unconstrained by what amounts to the four walls.

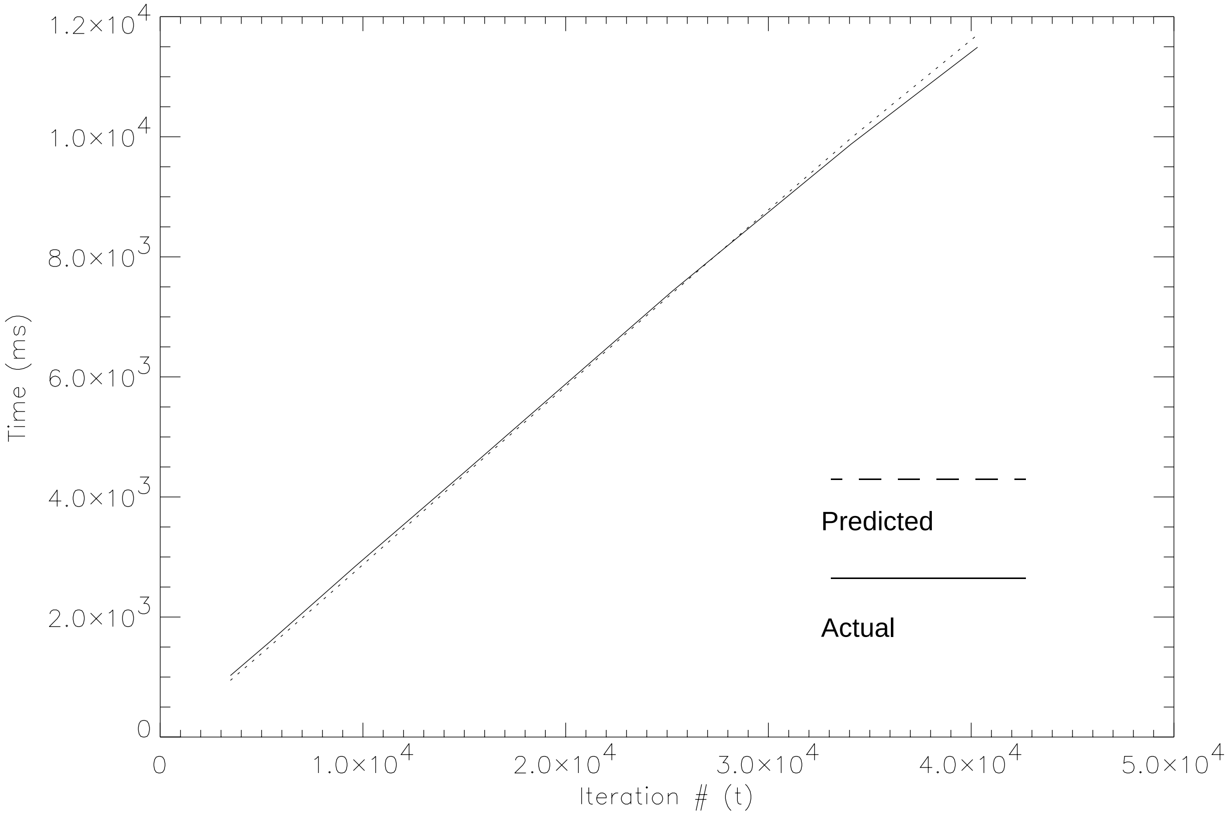
Figure 53(b) shows the trend from 1.5 average new nodes at about iteration 11,000 to approximately 1 node by iteration 30,000. Early in the computation however, the number of new nodes drops rapidly from 22 to about 4. The error due to dropped terms in the model is computed according to equation (1) on page 63. For the vehicle example, with a zero heuristic, if P is about .02% (due to dropped terms), and γ is assumed to be about 1.5, then the time formula will hold when the number of open nodes is 100 or greater.

Figure 53(c) shows a slight error between the predicted and actual computation time using A = 24 ms. and C = 22 ms. The value of A is small because the neighborhood and associated costs are pre-computed, and the number of newly opened nodes is also relatively small. The value of C is still small, but about four times greater than the previous robotic (H=0) examples most likely because of the heap overhead for storing, manipulating and accessing indices and data for the third dimension.



(a) Open Nodes, Ω(t)

(b) Average of Newly Opened Nodes, γ(t)



(c) Predicted and Actual Computation Time. A= 24 ms. C= 022 ms.

*Figure 53**:** Vehicle Computation.*

4.2 A Radio-Controlled Implementation

In order to find the limitations of the planned paths when they are conveyed to a controller, a vehicle testbed was designed and constructed. The goal is to show that the planning theory can be carried out in practice. The vehicle is controlled in a 6’x8’ testbed as shown in Figure 54. The simulated paths of the vehicle shown in Figure 49, Figure 50 and Figure 51 execute in the testbed environment shown in Figure 54. In addition to these, many more complex maneuvers were accomplished.

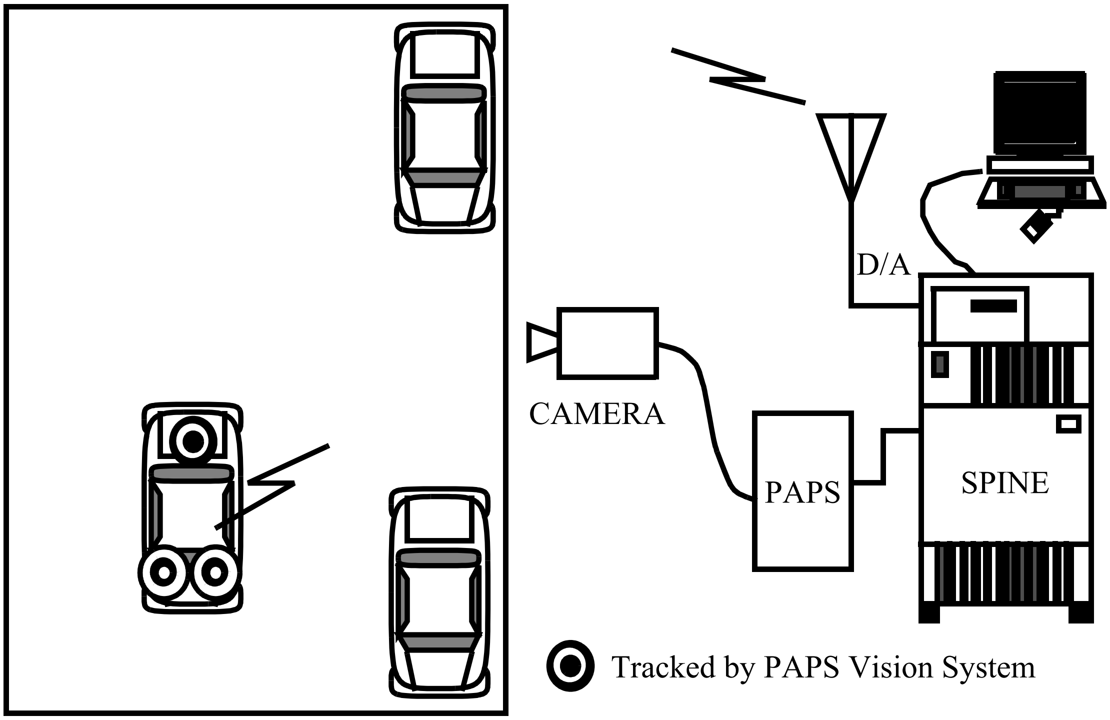


Figure 54: The Testbed Environment.

4.2.1 Overview of the Autonomous System

Recalling the general autonomous systems architecture of Figure 9 of Section 3.1, the *path planner* receives higher level instructions such as the goal and optimality criterion from the *task planner*, and then must produce a plan to move from the current location to the goal. In this case the task level instructions are created by the user. To perform the planning however, other information is required such as the location of the obstacles. Although the obstacles could be detected by the image acquisition (vision) system and then input into the system, in this case they are also given manually through the user interface. This is because the vision system may not perfectly identify the location of the obstacle, and because the system under test is the planner and not a computer vision system.

Once the information is in place, the planner produces a full navigation map (using H=0) which is used as a basis to control the vehicle. The vision system is used to track the position and orientation of the vehicle. The reflex controller and servo controller, which reside together in the same piece of software, use this information to control it. The reflex controller does sense the error between the vehicle and the desired track and uses that as input for the servo system. It does not sense and respond to unforeseen obstacles, since this was not the objective of the testbed. It then works with the servo part of the software to continually steer the wheels toward each setpoint. Naturally, the controlled *machine* is the car.

4.2.2 Real Time Operating System and Hardware

To coordinate and control the activities of the car, a multiprocessor system called SPINE (Structured Processor Interactive Networked Environment) [62] was used. Designed for real-time experiments, it consists of several Motorola 68020 processors running at 16.7 MHz with 1Mbyte of RAM each on a VME back-plane, plus a 4Mbyte independent common memory board accessible from all boards, a Digital to Analog board, and Digital I/O board form the multi-processor system.

A stock 1/10 scale RC-10 radio controlled car is used as the controlled vehicle. The body dimensions, structure and steering ability are used as input for planning the motions. The body is 20” long and 9.75” wide. The center of the rear wheels is 4.5” from the rear, and the front wheels are 5” from the front. The steering wheels can rotate up to 25° to the right and left. Therefore the wheelbase is 10.5”, and the track was 9.75”. The testbed also contains two other car bodies (obstacles) with similar body dimensions.

4.2.3 Sensory Tracking System

The position and orientation of the RC-10 are determined by an infra-red camera, mounted above, viewing 3 infra-red LEDs mounted on the top of the car body. Two LEDs are positioned about .1” behind and parallel to the rear axle. Ideally they would be placed over the rear axle, but mechanical restrictions prevented this. The third LED was placed in the nose of the vehicle on the longitudinal axis. This positioning provided a fairly compact way to compute the x,y location, and gave sufficient distance to accurately define the angle.

The lens is a wide angle corrected lens, that requires calibration to match the pixel locations to the x,y location and orientation, but fortunately no distortion correction was needed. The camera transfers the image to a Philips PAPS (Picture Acquisition and Processing System) where the image is thresholded, refining the points, and the position and orientation are reported to the local memory area of the M68020 processor controlling the car. This computation provides the location every 25 ms (40Hz).

4.2.4 Control System

The RC-10 is radio controlled from a stock FM radio-control transmitter. The transmitter is computer controlled by the digital to analog board (D/A) which then ties into the transmitter circuitry. One circuit controls steering angle while the other controls the transmission (forward/reverse). The drive-train of the RC-10 was altered so that it has a geared down rear axle and smaller motor. This was done both to reduce the speed of the vehicle and to extend the life of the on-board battery to about 4 hours.

To control the motions of a given device, the current position of the device is available with a delay of 25 ms. The direction arrows found in the corresponding configuration state determine the optimal motion for the device. By concatenating the optimal motions from the neighborhood, an optimal path to the goal is produced. In practice however, the path is followed piece by piece in a repeated move-select sequence. This allows for on-line correction in the event that the device ‘falls off-track’ either due to mechanical error, low-order unmodelled behavior or operator intervention.

From the common memory area, the controller can read the direction arrows indicating the proper control of the vehicle from any currently sensed position. In practice, an arc is struck from the current active position through the next x,y,θ setpoint and the vehicle is controlled using a proportional integral derivative (PID) method[1] [41] through that point. This feedback control helps correct for an imperfect mechanical system or any sliding on the surface. The controller continually monitors the active position in comparison with the desired setpoint. If the current position is wildly out of track with the setpoint, such as if we reach in and flip the car around, the controller can abort the motion and continue from a newly determined starting position.

Smooth Control

In order to control the vehicle, the discretized configuration space path must be converted to primitives for the controller[51]. These primitives are an arc curvature (based on a center and radius) and direction (forward or reverse). This conversion is necessary to provide smooth control of the car. Because the shape of the neighborhood causes severely discontinuous behavior between the directions for nodes in close proximity to one other within the navigation map, the elementary control strategy of simply sensing the current position and following that control directive is not sufficient. For example, a minor change in the perceived orientation of the vehicle might cause opposing control directives to be issued. By this method, the vehicle will have many oscillations of fits and starts while attempting motion to the goal. This is clearly undesirable.

The fact that control cannot be achieved reliably by localized directions means that instantaneous error recovery is also impossible. Any minor mechanical error or outside intervention cannot be

---

[1] PID. Proportional: Control response is proportional to the error. Integral: Integrates the error over time (useful for slow or stop control). Derivative: Control proportional to the derivative of position. In other words, if the error is large, and velocity small, apply large force; if the error is small, and velocity high, apply a negative force. The derivative causes damping in the system.

immediately detected and acted upon. The following enhancements cause a smooth motion to be reconciled with reasonable error recovery.

The first conversion is to give longer setpoints. Since neighbors are arcs of limited length, any sequence of like-kind neighbors are concatenated to determine the next setpoint for control. For example, if the path contains two directives that both result in forward motion with wheels at 45 degrees, then the setpoint should be at the end of the farthest 45 degree neighbor. This allows some smoothing of the discretization errors that necessarily occur when planning a continuous problem in a discrete space.

The second conversion is to control to the actual arc, rather than the planned arc. This is achieved by striking an arc from the current world position through the next x,y,θ setpoint. The vehicle is PID controlled using a smooth arc through that point. The direction of motion is determined by the selected neighbor. This helps correct for an imperfect mechanical system, any variation in the discretized starting state or minor sliding on the surface.

The third conversion is to define a *stopping criterion* that defines the moment when the vehicle is considered to have reached the setpoint. If this is not used, then if any control or mechanical failure occurs, and the vehicle does not arrive perfectly at the desired setpoint, then the PID control will force the car to try indefinitely to achieve a possibly small (and usually horizontal) motion. The vehicle has reached its stopping position when it crosses the line that runs through the setpoint and perpendicular to the setpoint orientation. This line allows for some discretization, mechanical or frictional error, while still ensuring a fairly robust hand off between setpoints. This means that the vehicle has achieved its setpoint even though the planned distance and actual distance are likely to vary.

Finally, once the vehicle has crossed the line defining the stopping criterion, an evaluation is made. If the current location is within reasonable range of the setpoint, then control resumes with the next neighbor in the path, otherwise the path is determined anew from the current location. As an alternative to this, an acceptability test can be made while control is underway. If the current position is wildly out of track with the setpoint, such as if we had reached in and flipped the car around, the controller can abort the motion and continue from a newly determined starting position.

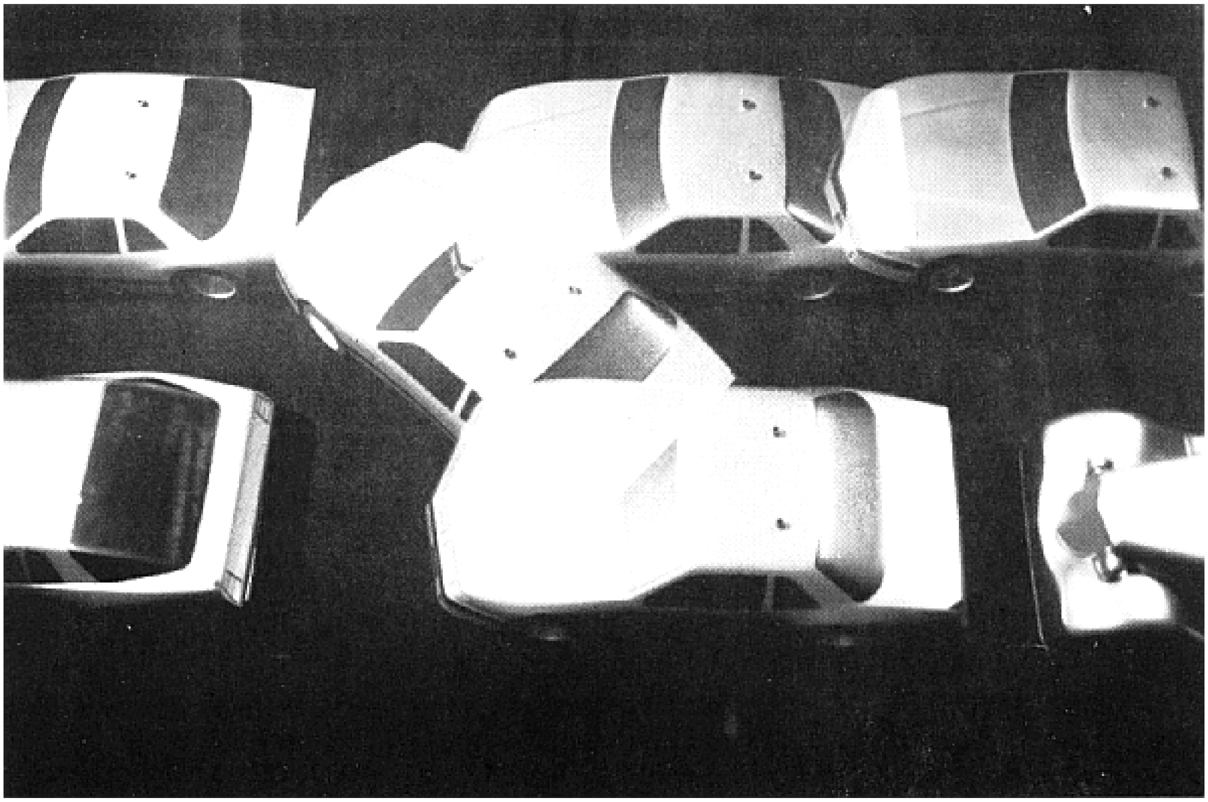


Figure 55: The Car in Stop-Action.

4.2.5 User Interface Software

The setup of the parked cars, goal location and optimality criterion is input via a graphical user interface (GUI) on a Sun 3/160 as shown in Figure 38. In a future implementation, the location of the parked cars and other obstacles could be determined by the sensing system. Once the configuration space is computed, it is downloaded using the SPINE communication mechanisms to the 4Megabytes of common memory RAM in SPINE.

4.2.6 Driving

To use the testbed once the system is loaded, the RC-10 is placed anywhere within the physical boundaries, and a ‘start button’ is pushed. This is an input to the digital board which produces a signal to begin. The car then drives to the goal. Figure 55 shows the car in stop-action[1].

---

[1] A videotape in either PAL or NTSC is available which shows the vehicle maneuvering testbed. It is available on request through the author at Philips Research, 345 Scarborough Rd., Briarcliff Manor, NY 10510 or via e-mail: kit@philabs.research. philips.com .

4.3 Other Applications

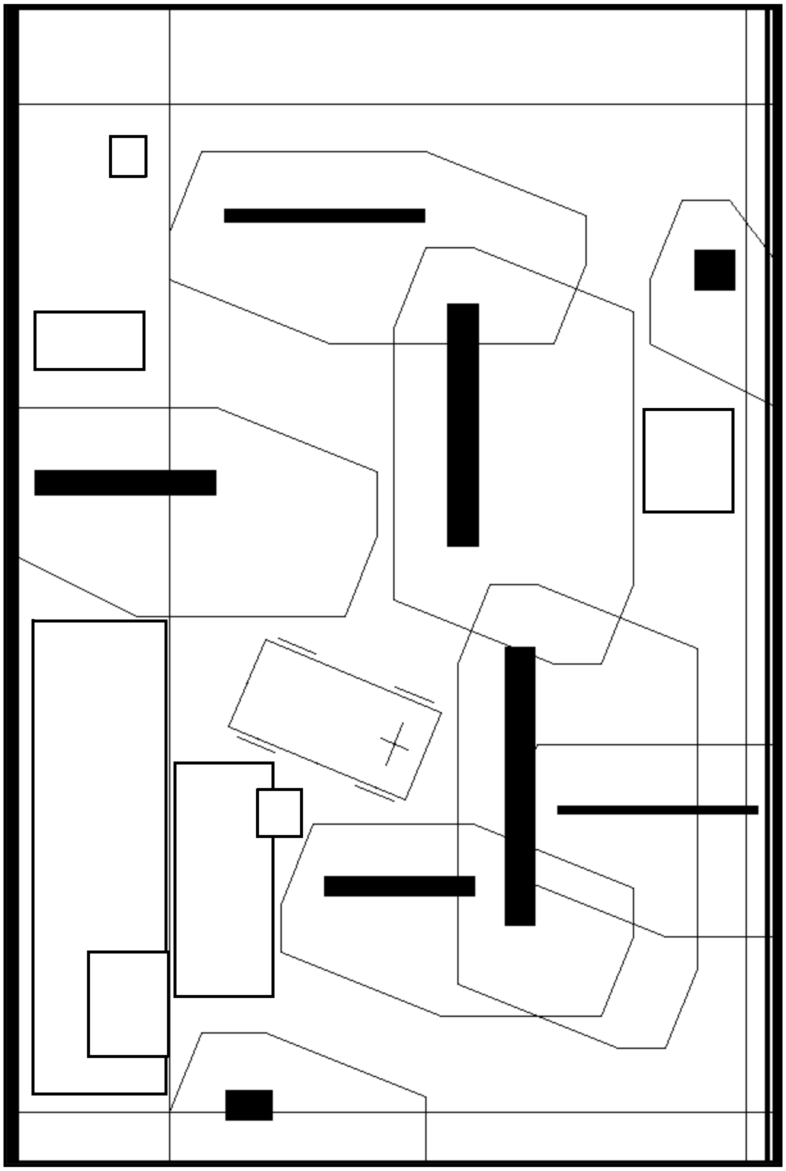


Figure 56: Transform of 8 Obstacles and 4 Walls for a Car at 160°.

4.3.1 Use of the System for Complex Maneuvers and Larger Areas

The setup and computation of complex maneuvers and larger areas follow directly from the previous example, and can be used to test that the solution can be scaled. Given the vehicle's dimensions and steering capability, the neighborhood can be precomputed automatically. The cost used is the distance the vehicle *position* (as defined in Section 4.1) travels. If the obstacles and vehicle are approximated by rectangles, then the transform resulting from a single angle of the vehicle is as shown in Figure 56. The region around an obstacle represents the closest that the position of the vehicle can get to the obstacle at the current orientation of the vehicle. The illegal region in configuration space is determined by computing the transformation at each of the convex and concave corners and filling in the forbidden region. More complex shapes are certainly possible. For each possible angle of the car, a transform is computed. The transforms for all angles of the car require about 500 ms (on a Sun IPX) for the walls, obstacles and curb of Figure 56.The optimal path is generated by performing A* from the goal (or goals). A* avoids the transformed obstacle regions, and eventually fills the reachable 3-D configuration space. The A* computation time for this large area is about 18 seconds (on the same Sun IPX). As mentioned before, the more obstacles that are present, the faster the solution is computed. By following the steering and transmission commands from location to location the path in Figure 57 is found.

Complex maneuvers are required when there are many perhaps odd shaped obstacles or a restricted space. The problem is set up and computed in the same way as for simpler arrangements, however the sensing problem is more difficult. In the case where the vehicle is performing anything much more than parallel parking, inexpensive sensors will not suffice. This is particularly true if complex obstacles must be sensed, or a large area is used for maneuvering. Even if the environment is known, the position and orientation of the vehicle must be reported in real time. This is perhaps the most difficult problem, since open-loop control is only accurate for a few motions. Only if the vehicle is performing in a restricted location, such as warehouse, airport, ship, or commercial establishment, can fixed sensors be placed efficiently to track or locate the vehicle. Alternatively, the vehicle can use beacons (bar-codes) or landmarks to find its position. Global positioning with differential correction may be of use in some outdoor applications, although this is an area for future research. Other than the sensing problem, there is no difficulty in computing an optimal path around numerous obstacles.

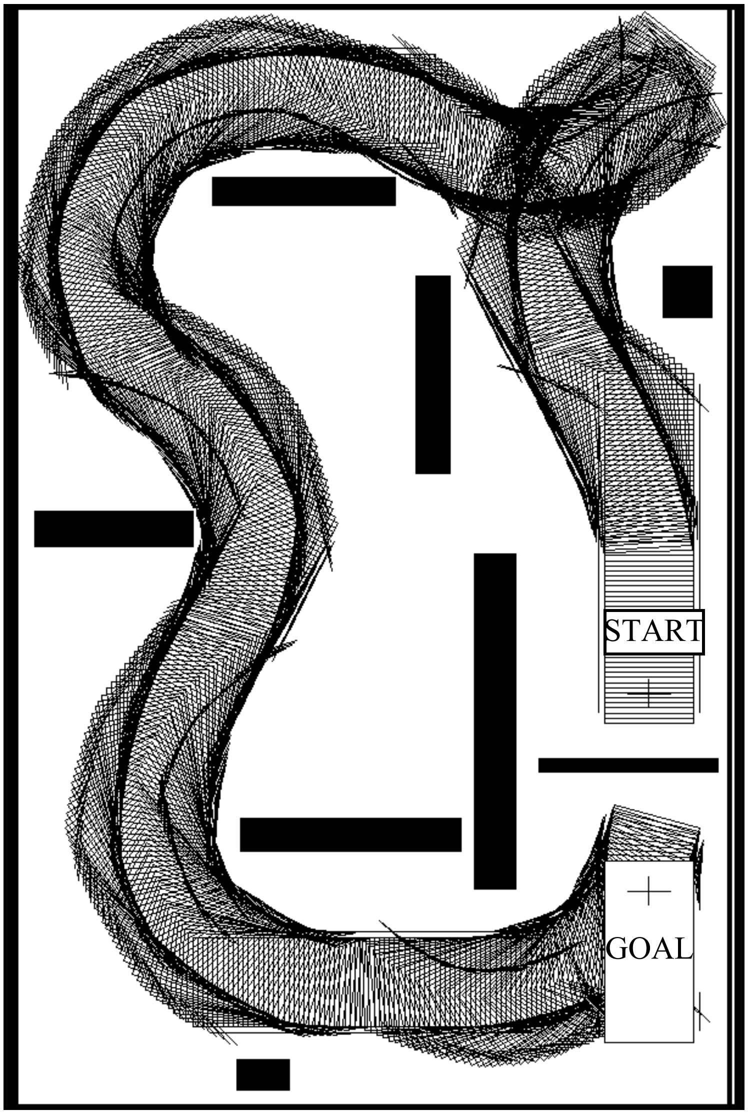


Figure 57: Minimizing Distance traveled While Avoiding Obstacles.

4.4 Bulldozer Maneuvering

Treaded vehicles (such as a bulldozer) are a simpler kinematic case than cars. Using the same essential configuration space of the car (of Section 4.1) that has 48 states (in X) by 36 states (in Y) by 64 states (in θ), the position and orientation of the bulldozer can be described. As simply modeled, they may only rotate or drive (forward or reverse) based upon the current orientation. Therefore, the turning radius is zero since it can turn about its axis, decoupling the rotation and drive controls. The problem then degenerates into a simpler point-to-point routing problem in configuration space. The neighborhood has two fundamental directions, as shown in both two and three dimensions in Figure 58. For each angle θ, the neighborhood is calculated to contain four neighbors in each fundamental direction, for a total of 16. A smaller neighborhood could have been provided, but since the angle θ of the neighborhood varies, less discretization error (as discussed in Section 3.6.1) will occur with the larger neighborhood This gives the changes in x and y as the bulldozer moves in a straight line (for the current orientation) in configuration space. The cost measure is the distance traveled, (not time) with a zero cost for rotation. Using a zero heuristic, and the same essential obstacle transforms as the car, a bulldozer maneuver can be computed in the prior environment of the car. Because the optimal path is the shortest Euclidean distance in configuration space, an admissible heuristic could be provided.

The strength of the method is seen by the modularity of the changes. The only difference between the car and bulldozer is that the neighborhood (and associated costs), and the dimensions of the vehicle where changed. The transformation, planning, and most of the user interface, was preserved.

In Figure 59, the bulldozer avoids an obstacle, and backs into the parking place.

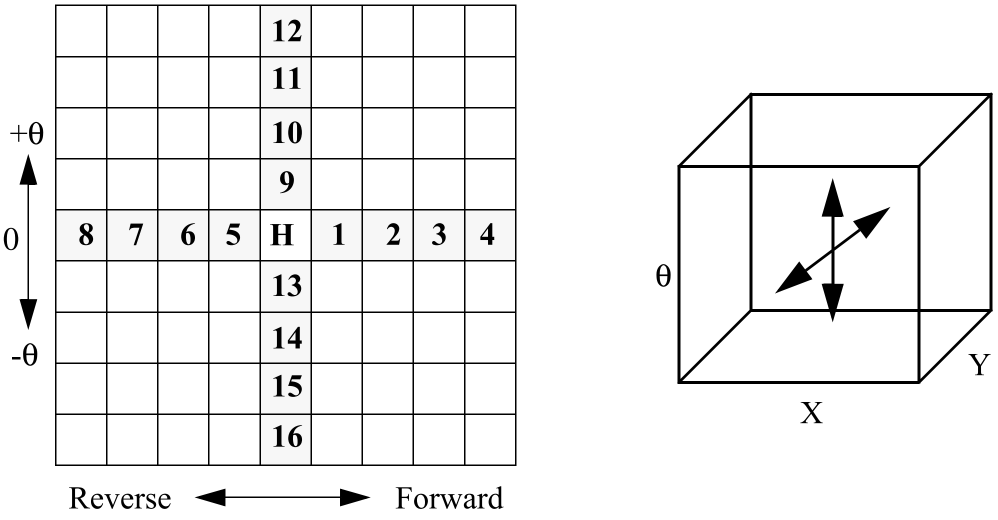


Figure 58: Bulldozer Neighborhood in 2 and 3 Dimensions.

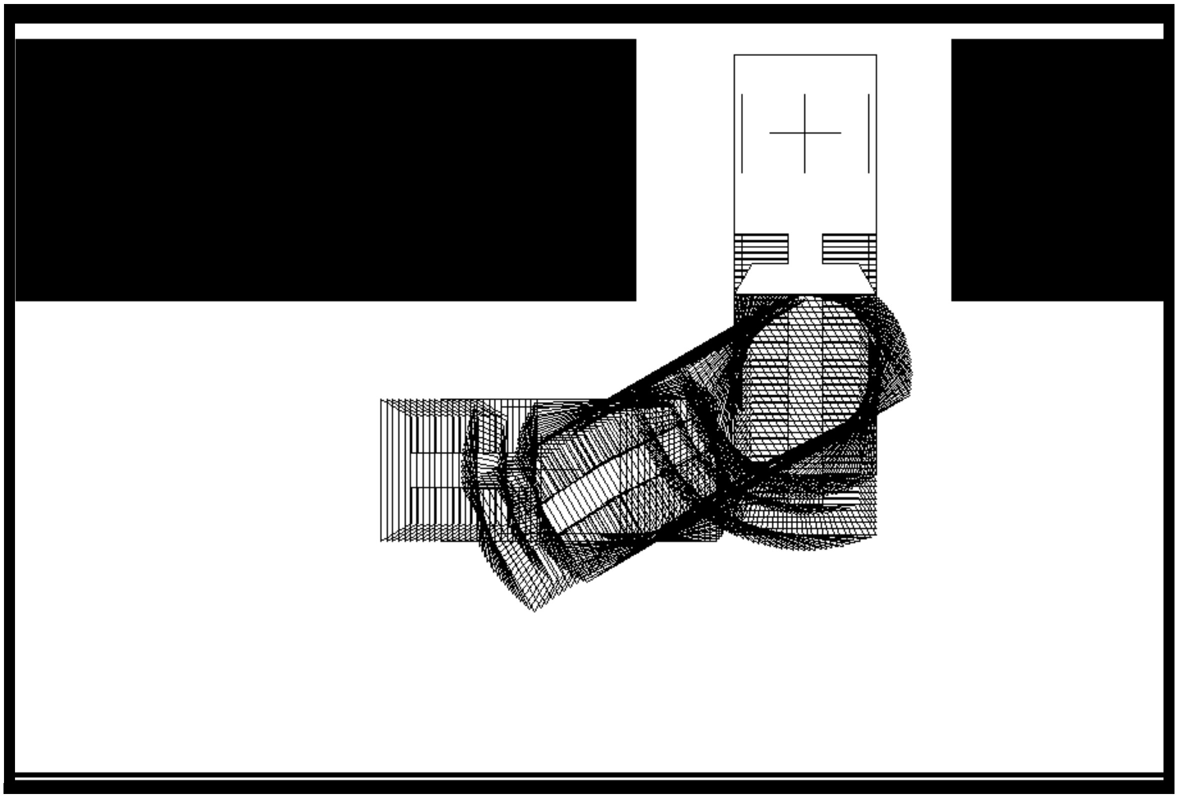


Figure 59: Bulldozer Maneuvering.

4.5 High Speed Vehicle Maneuvering - Managing Relative Motion

Initially, the planning for a scenario any larger than that of the complex region in the previous section would seem impractical, not only because of the real-time sensing issues, but also because the planning would be too slow and require too much memory. Since maneuver planning determines the proper reaction in the sensing-reaction control loop, it is important to be able to perform this quickly to get adequate ‘reaction time’. Fortunately, quick planning in a relatively small space can be performed for this problem [55].

Configuration Space

In the situation where a vehicle is moving along a highway, the configuration space should be considered as the space moving relative to some other object or marker. Previously, all configuration spaces were considered to be relative to a fixed origin. Using this method, the configuration space is two dimensional rather than three, and is therefore much quicker to calculate. The discretization of the space should be coarse in the x direction (e.g. 3 lanes wide), because it is impolite to weave through traffic without fully entering or leaving a lane. The discretization in y (road distance) should be chosen in a trade-off with the computation time because it is a component in defining the reaction time, along with the sensing capabilities.

Neighborhood

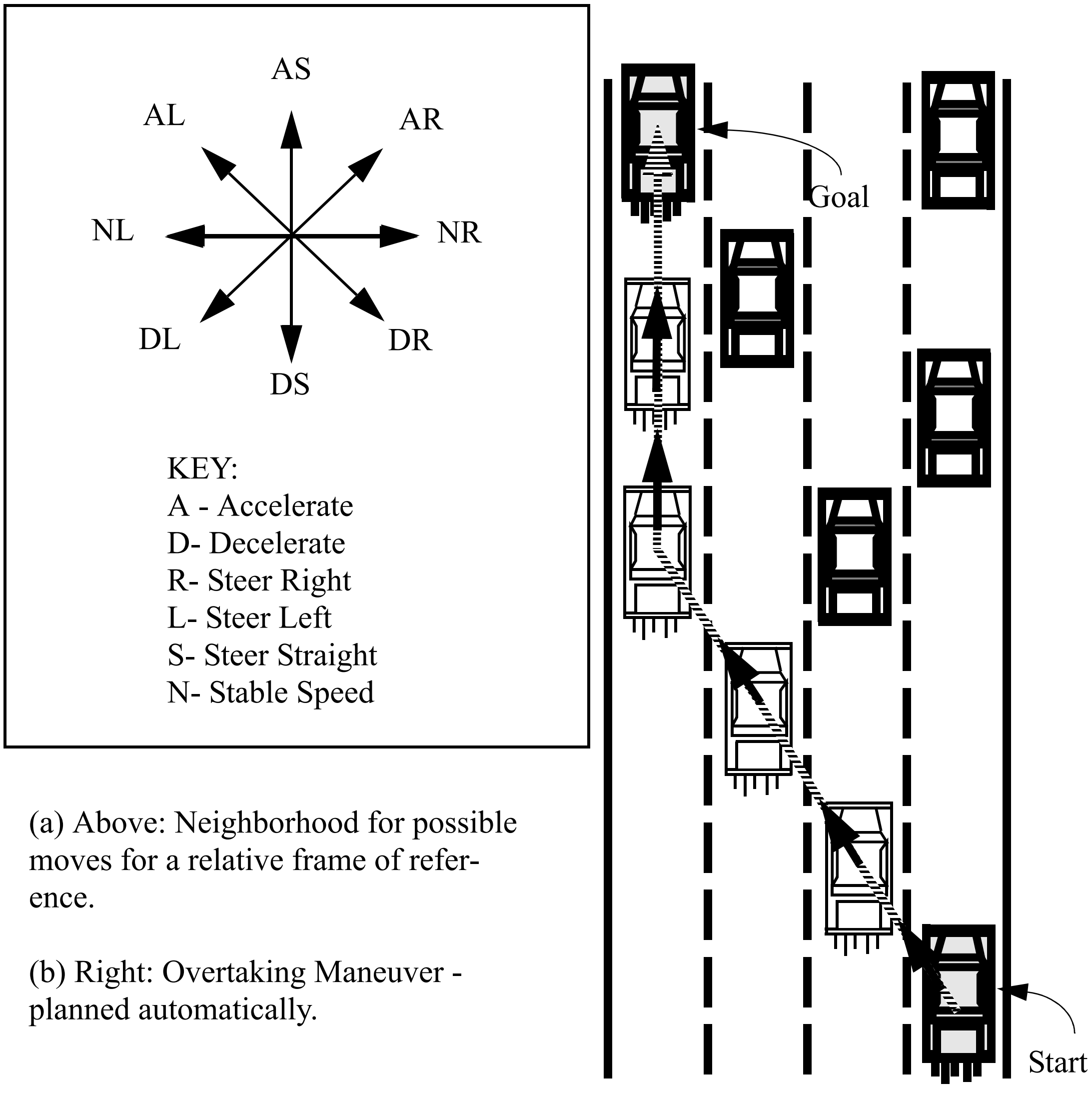
Looking at the ‘neighborhood of permissible motions’ for a high speed vehicle, it is discovered that the vehicle can in fact move sideways, relative to other vehicles, simply by steering. Also, forward and backward motion can be achieved by accelerating or decelerating. Speed limits can be incorporated naturally by changing the neighborhood as a function of the current speed. This is a delightfully simple neighborhood for planning purposes, as shown in Figure 60(a). Figure 60(b) gives the results of straightforward path planning in a relative space for vehicles moving at high speeds.

Cost Measure

The cost measure is the Euclidean distance in the configuration space.

Constraint Transformation

The obstacles are the vehicles to be avoided or passed, and the edge of the road. These are transformed in the same manner as the vehicle transformation of Section 4.1, which ‘grows’ the obstacles by the dimension of the controlled vehicle for the orientations of interest. In this case the orientation is parallel to the edge of the road which can be treated as a fixed axis at each moment in time. Therefore the transformation is straightforward and requires only one slice of the previous transformation. It is also possible to lengthen the obstacle to the front or rear to allow buffer distances for proper passing.



(a) Above: Neighborhood for possible moves for a relative frame of reference.

(b) Right: Overtaking Maneuver - planned automatically.

Figure 60: High Speed Vehicle Maneuvering.

Start and Goal States

The goal can be set as a position relative to the other vehicles. If the goal is to simply pass the other cars, then there are goals covering each lane ahead of the vehicles. The start is the current location.

4.5.1 A* to Compute Costs and Paths

A* is used to compute the path around the surrounding cars. At each state there are directives such as “accelerate and move one lane to the left”. Since the space is two dimensional, and possibly relatively small, and the transformation is fast, the solution can be found quickly.

4.5.2 Planned Solution

Figure 60(b) shows the resulting proposed path for a car to overtake other vehicles. If there is no path, then the current relative position should be maintained. Any sensed change in the relative positions of the other cars can be accommodated in a short time because the planner is fast. For high speed maneuvering the time to update the configuration space is clearly critical.

4.6 Concluding Remarks

The framework using A* and configuration space which uses a neighborhood with costs has been shown to provide a flexible, yet powerful mechanism in which to determine optimal autonomous motions for a vehicle [48]. The clear conceptual components provide a mechanism to encapsulate each part of the planning problem in a manageable form. The A* computation can then automatically generate the required navigation map, which may be highly complex. It is simple to use the navigation map to produce optimal, collision-free motions.

For automatic maneuver planning for a car, the fundamental maneuvers are more complicated than for the two link robot since the kinematic restrictions must be taken into account. Even though the machine operates in a three dimensional configuration space, the vehicles capabilities restrict it to a subset of two dimensions. These type of constraints is also called non-holonomic[17] constraints. This is because the control parameters are different from the planning parameters.

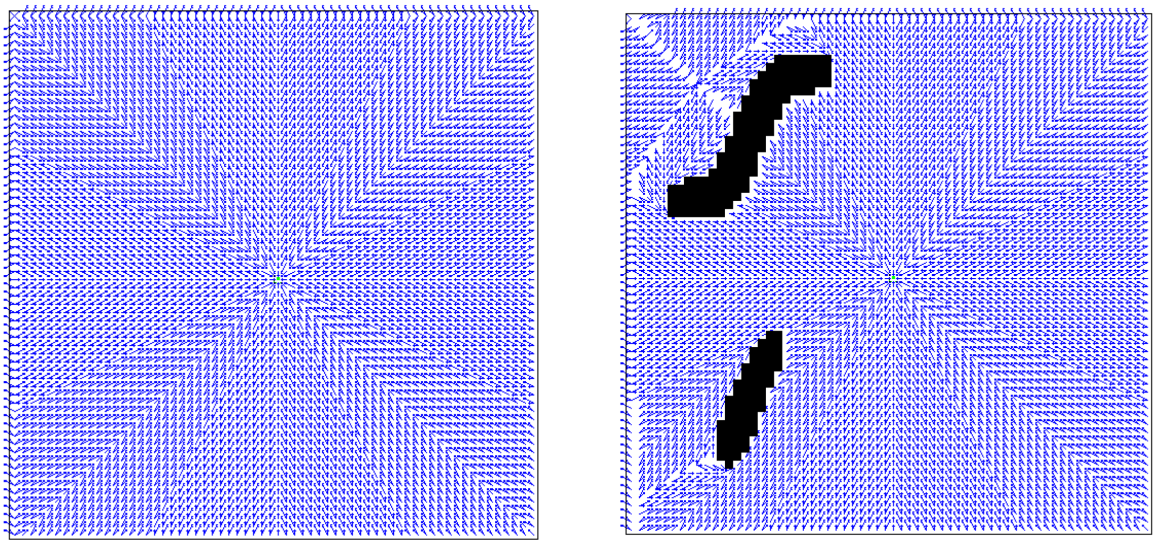
A few methods exist to compute maneuvering solutions that begin to address this class of problem. One example is Volkswagen’s parallel parking Futura which involves imitating the human rules (tricks) to achieve automatic control for specific maneuvers. A similar method [40] allows a user to select from a collection of monotonic maneuvers (i.e. S-shapes, and arcs but no zig-zags), which are tuned to the local parking environment, and then executed on a vehicle. Another method uses rules based on locally optimal maneuvers [43]. Some success has been achieved using well known dynamic programming techniques for localized animated car parking [57], and by using potential fields in configuration space[3]. Many methods avoid the problems associated with the kinematic constraint altogether by planning only for vehicles that rotate about their axis.

The method presented in this chapter provides a systematic method to compute globally (not just locally) optimal, collision-free paths for both a car and treaded vehicles. This method computes paths determined by the inherent kinematic limitations of each of the machines, and is not based on quantified rules taught to the system. This provides a way for the machine to be ‘self taught’, that is, it has an automatic, general way for the machine to determine the optimal path autonomously.

Chapter 5

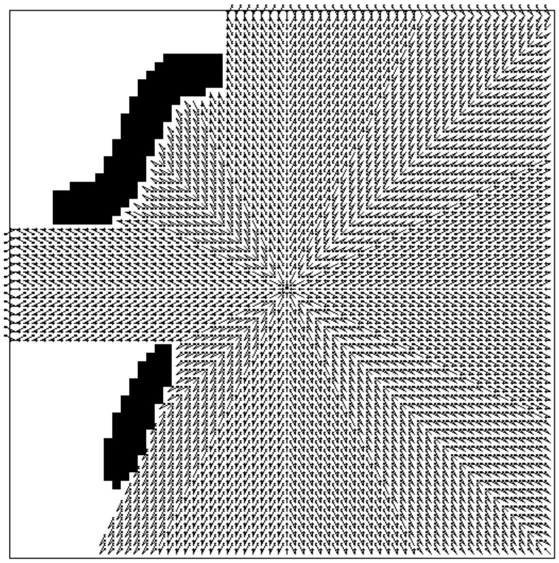
Differential A* (^{1}A*) for Graphs

In the A* method, any change in the cost of nodes or transitions in a graph G, forces complete recalculation to obtain the explicated (solution) graph. Depending on the problem domain, such changes may occur quite often. In many cases, however, the pointers in the explicated graph differ only partially after recalculation.



(a) Original graph G_e

(b) Revised graph G_e'



(c): Region affected by obstacle

Figure 61: A Small Change Affecting a Small Region.

For example, Figure 61(a - b) shows two explicated configuration space graphs taken from robotics for the 2-jointed robot shown in Figure 15 of Chapter 3. Figure 61(a) shows the arrow pattern of the resulting explicated graph G_e once the search is completed. The graph in Figure 61(b) shows an obstacle that has been introduced, and the revised G_e. The entire region has been recomputed, but only a portion of the resulting G_e has changed. If the environment started as in Figure 61(a) and received the obstacle states all at once, then perhaps it would be preferable to change only the area affected by the obstacle itself rather than recompute the full region. To highlight the portion of the configuration space affected in this example, Figure 61(c) has those nodes erased that are affected. The research topic of this chapter determines that this area can be identified and recomputed, so that in many cases less computation time is required.

We designed the "Differential A*" algorithm as a method that builds on the A* approach to adapt quickly to changes in a graph by determining and updating the localized regions affected by those changes rather than regenerating the entire space. Changes in node cost, transition cost, and starting state selection all affect the explicated graph G_e.

5.1 Outline of the Method

The explicated graph is implicitly determined by the properties of G: transitions (and their costs), delay costs, and the start and goal nodes of the search. Therefore if any of these change, they may impact the explicated graph. These elements are the smallest instruments of change to the graph, and are essential input to the Differential A* structure outlined in Figure 62. First, new notation specific to Differential A* will be introduced including a node n and its relationship to a parent node, active and passive transitions, and pointers. In addition to these objects, several functions that act on them will be presented. With this, each of the different types of changes are analyzed, giving rise to an efficient method for generating a correct explicated graph without full recomputation.

Figure 62 shows a block diagram of the principal components of the Differential A* graph search method along with the interfaces and results. The changed transitions and nodes as well as the nodes that were the OPEN nodes of any previous A* search are used as input drivers to the difference engine. The difference engine determines the impact that each of these changes has on the current state of G_e. This impact and the previous OPEN nodes are used to determine systematically the minimum number of nodes required as seed (or OPEN) nodes. These nodes become the source of the new A* search.

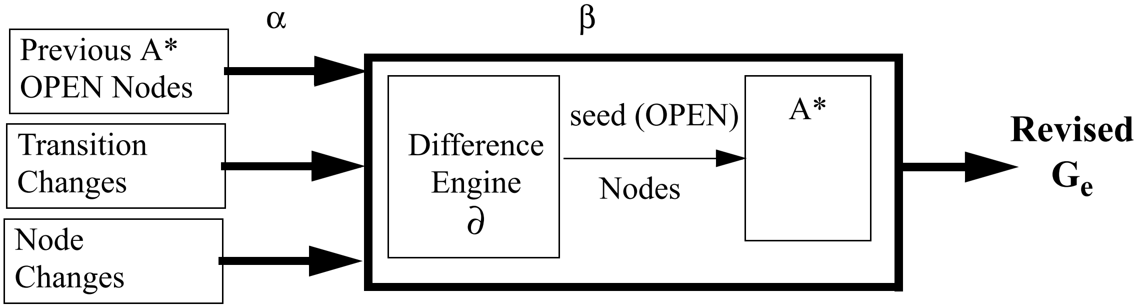


Figure 62: Differential A - Outline.*

5.2 Notation

This method will vary only slightly from the definitions of A* in Chapter 2. In the same way that for the robotics applications of Chapter 3 we chose to view the goal state as the 'starting node' of the A* search, this method will assume the same direction of search. This causes the successor function for any given node to be equivalent to the 'neighborhood' function defined in the robotics section Section 3.2.1. Pointers indicate the optimal transition from a current node to the next nearest node to the goal, and the direction of pointer is the same as the transitions, rather than opposite as in the original A* method of Section 2.3. In practice, this provides a more intuitive use of the transitions which then represent permissible motions from state to state. In the event that the search originates with the starting state rather than the goal state as proposed, the same method can apply, taking care that the direction of computation affects the cost measures and successor functions.

The notation will be divided into three categories. The first is general, the second affects the graph G and the third affects the explicated graph G_e.

5.2.1 General Notation

- A *node n* represents a node which is a point of reference for a particular discussion.
- An *uninitialized node* ϕ is a node that has not been on OPEN during the computation. These nodes have the property that the delay cost and total cost are not determined and there are no pointers leading out of ϕ. $\phi + K \Rightarrow \phi$, where K is any value.
- Φ denotes the set of ϕ nodes.
- *Seed nodes* are the nodes sought by the Difference Engine. These nodes may become the OPEN nodes for the subsequent A* search.

5.2.2 Notation Related to Graph G

- A *transition t* from any node n_1 to a second node n_2 implies a cost, greater than zero, that is incurred by moving from n_1 to n_2. The transition is written $t(n_1,n_2)$. The cost is written

$c(n_1,n_2)$

- SCS(n) is a successor function of a given node n. It returns the set of all those nodes that have transitions in the general graph G from each (successor) node n' to the node n.
- scs(n) represents a single successor node of n.
- PRED(n) is a predecessor function of a given node n. It returns the set of those nodes that have transitions in the general graph G from the node n to other nodes n', and is related to the inverse of the successor function.
- PRED(t) is a predecessor function of a given transition t. Even though this function is overloaded by accepting either a node or transition, it seems sufficiently intuitive so that we will keep the same name. It returns the single node which is the destination of the transition. In other words if a transition t_1 points from n to n', then n' is the predecessor of t_1.
- pred(n) represents a single predecessor node of n or t.

5.2.3 Notation Related to Graph G_e

- A *parent node* of n is a member of the explicated graph G_e that is pointed to by a given node n. An individual parent node will be denoted as parent(n). If there is a pointer from the node n to a parent, then n ∈ SCS(parent), n being one of many possible successors of parent, but being at least one of those nodes now in the explicated graph $G_{e.}$.
- PARENTS(n) is a function returning a set of all parents of n.
- A *child node* of n is a member of the explicated graph G_e that points to a given node n. An individual child node will be denoted in upper case as CHILD(n).
- CHILDREN(n) is a function returning a set of all possible children of n. The node n can be considered a child node of parent, so that n ∈ CHILDREN(parent).
- *An active transition* $t(n_1,n_2)$ of G has a corresponding pointer from the node n_1 to the node n_2 in G_e. This represents the lowest cost path from n_1, through n_2 to the starting node.
- *A passive transition* $t(n_3,n_4)$ of G does not have a corresponding pointer between n_3 and n_4 in G_e.
- *Perimeter nodes* are nodes that surround Φ and are defined as the set of all nodes that are $\{pred(\phi) \setminus \Phi \mid \phi \in \Phi\}$. These nodes are candidate nodes (described next).
- *Candidate nodes* are the combination of two sets of nodes, the nodes left on OPEN when the previous A* computation terminated and nodes identified as candidate nodes by the difference functions.

5.2.4 Example

An example that uses the basic definitions is given in Figure 63. Transitions represent permissible changes between nodes in the graph, and are therefore elements of G. They are drawn in this and later figures with dashed arrows. Active transitions that are elements of the explicated subgraph G_e are optimal and have corresponding solid pointers. A pointer path, PP, is obtained by following from node to node according to the pointers. In this figure, the only pointer is between n_1 and its parent n_0. It can also be said that n_1 is a child of n_0. All non-active transitions are passive, such as between n_3 and n_2. Key functions that were described earlier were the successor (SCS) and predecessor functions (PRED). In this instance, SCS(n_1) = {n_3,n_4} because these are transitions into n. PRED(n_3) = {n_1,n_2} because these transitions lead out of n_3.

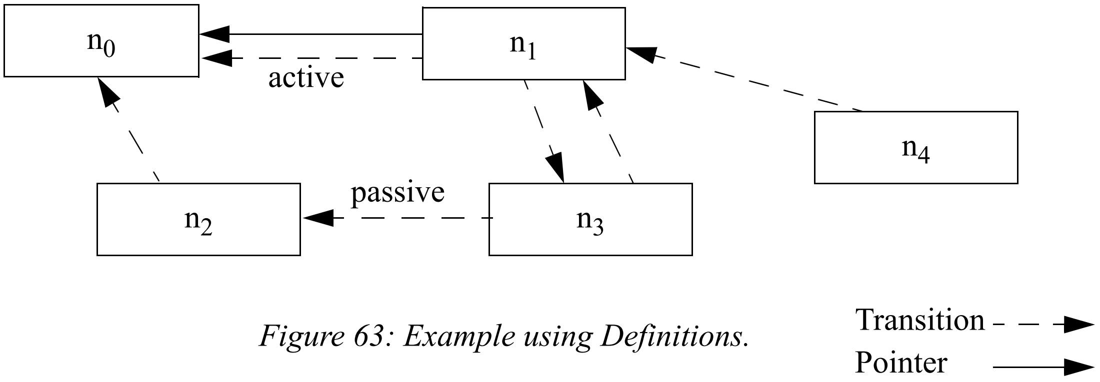


Figure 63: Example using Definitions.

5.2.5 Fundamental Operations

- The *Clear Influence* function χ (n) is the key function in the difference engine. Given changes in the nodes and transitions, all nodes must be identified that will require an increased cost, so that they may be recomputed. The χ (n) function pinpoints these nodes by adding n and recursively all children of n into Φ and resets them to their uninitialized state ϕ. In addition it finds the nodes that are adjacent to (perimeter of) those that are cleared. This are the source of OPEN nodes, in addition to any OPEN nodes left from the previous search, used to *recompute* Φ using A*.
- *Individual Node Recomputation* is a generic function for all nodes. A node n may need to have the value of f(n) updated. The cause may be the change in the incurred cost within the node itself in G, such as delay cost. Alternatively the cause may be a change in the cost of a transition in G on which the node depends. Ordinarily this does not change the pointer of the node in G_e. However, in some graphs G_e, the node n may have more than one equivalent cost parent in G_e. In such a case, there will be several pointers leading out of n, so n has several parents. If the cost of the node is decreased by a transition to one of the parents, then clearly the alternate pointer to the (now higher cost) parent must be deleted.
- *Add to Candidates* is a repository for all candidate nodes that will be reviewed as possible seed (OPEN) nodes.

5.3 Difference Engine

This section details the objective for the difference engine which has been introduced previously in [6, 45, and 46]. Once the objective is defined, a method comprised of several cases is outlined to achieve the objective. Finally, each case is analyzed and a summary method is proposed.

The objective of the Difference Engine is to efficiently manage the changes that might affect a graph in such a way as to minimize the A* recomputation required. An outline of the Difference Engine is shown in Figure 64. The recomputation then produces a revised (correct) graph. This is achieved by reinitializing some parts of the graph and then using specially selected ‘seed nodes’ to recompute the graph.

Transition and node changes are the primary types of change. These changes may be their creation, deletion, increase or decrease (in cost values, recalling that f(n) = g(n) + h(n)). In addition, transitions that are active and passive are also evaluated as separate cases. Nodes may also be changed into or out of the special categories of ‘start’ or ‘goal’. For each of these changes, the case is analyzed to identify the difference calculations required to recompute the graph. It will be assumed that updating the transition and node values according to the specified change will be performed as a matter of course.

The overall objective, however, is to determine the ‘seed’ or OPEN nodes that will be used in the subsequent A* computation. The transition and node changes may cause some nodes to be considered as candidate nodes. In addition to candidate nodes that are found due to changes, OPEN nodes in the preceding A* search must also be considered as candidate nodes. The order of candidate generation varies Φ, and different affected areas may overlap. This means that the final determination of *seed nodes* (OPEN nodes) for the next call to A* cannot occur until all affected areas are found.

These candidate nodes will be re-evaluated just prior to the A* recomputation to ensure that they are not duplicates or are disqualified by other difference calculations which set the node to uninitialized. By adding only the nodes necessary, computation is minimized. The qualified candidate nodes will then be considered the OPEN or ‘seed’ nodes for A*.

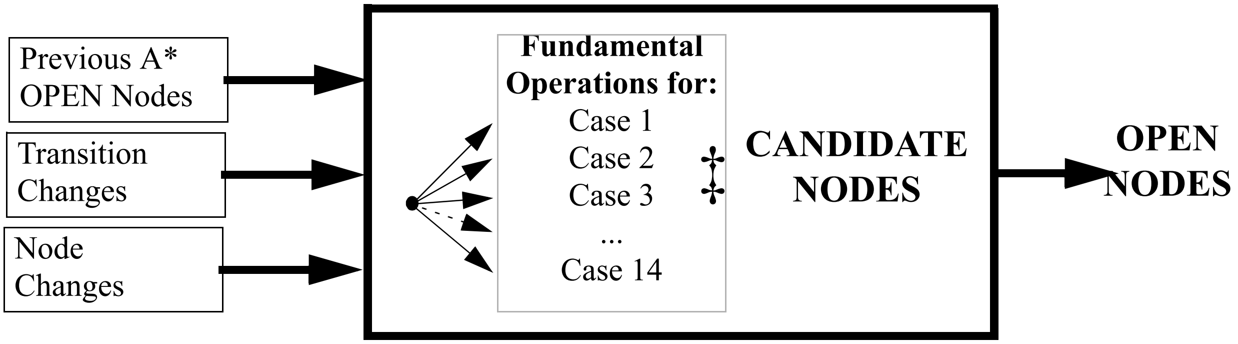


Figure 64: Difference Engine ∂ .

5.4 Case Analysis for Difference Engine

To determine the minimum number of relevant cases, each variety must first be evaluated. The analysis is broken down into the possible cases (Figure 65). The parts of the graph that can change are transitions and nodes. Transitions are further broken down into their more specific cases of passive and active. The types of change that can affect these parts of the graph are increases or decreases in the cost, specifically transition cost or node delay cost, or the creation or deletion of a new part. Each of these types of change is discussed as a separate case. Beyond these fundamental cases, two additional cases are also considered. Case 13 and Case 14 manage the change in classification of a node to or from a 'start' node.

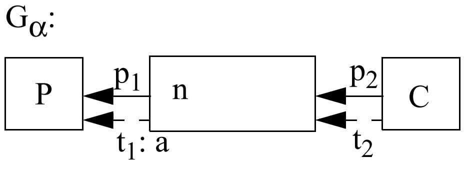
	Transitions		Nodes	Start Node
	Passive	Active		
> (Increase)	Case 3	Case 1	Case 9	
< (Decrease)	Case 7	Case 5	Case 10	
Creation	Case 8	Case 6	Case 11	Case 13
Deletion	Case 4	Case 2	Case 12	Case 14

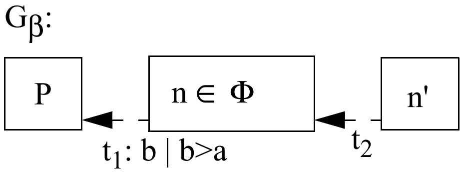
Figure 65: Table of Cases.

For each of the cases, the objective is to determine the effect of the node or transition changes. Some recursively affect child nodes in addition to their own values. This process will use the 'clear influence' function of Section 5.2.5. The cases will be listed first for changes to transitions, next for changes to nodes, and last for starting state changes.

Each case will be illustrated showing the graph (including G_e and G features) at time α, before the difference engine and then the graph at time β after the difference function; these will be denoted as G_α and G_β, respectively. Costs for nodes or transitions will be preceded by a colon i.e. (t_1: b | b>a). There is typically a node or transition which is the focus of the case; these will be denoted by n and t, respectively. If there is any parent or child node, it will be denoted by a P or C, typifying all PARENTS(n) and CHILDREN(n). Actions performed on a typical node will be assumed for the others.

Case 1: Increase in Cost of Active Transitions





In this case, the value of the transition t1 is increased. In G_α, p_1 indicates that n optimally points to P. In G_β, the pointer of n may now be impacted because another PRED(n) may be the (optimal) new parent of n. Not only is the pointer affected, but the cost will now be increased unless the pointer happens to result in an equivalent cost via the new parent. This recomputation to find the new parent of n can be achieved by examining all of the predecessors of n, determining the minimum f(n), and assigning the corresponding pointer to the parent that leads to the lowest f(n). The predecessor computation and pointer revision must then be followed by the recomputation of all children of n which must also be updated.

We discuss this process here in more detail, but it is in fact a detailed example preparing and using the clear influence function χ of Section 5.2.5.

The transition cost change affects n, but it must also be reflected in the children of n. It is not sufficient to simply propagate the increased cost to the children, because each child may have other predecessors from which to take a more optimal cost (a child of n may have other parents than n'.). Therefore, for each child, the cost must be recomputed based upon the possible predecessors of the child similar to the recomputation of n.

This process would lead to the recomputation of all of the children in the order they are found, which, depending upon the algorithm used for tracing them (breadth first, depth first, etc.), may not result in an efficient recomputation. In addition, it is often the case that several simultaneous changes are made to the nodes or transitions of a graph.

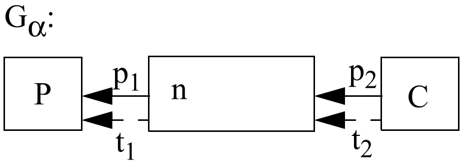
More efficient management of the possibly multiple simultaneous changes can be achieved by a two step process. The process first identifies a collection of nodes as ‘uninitialized’ for each of the perhaps overlapping changes, and then prior to performing the A* recomputation finds the predecessors of the union of these uninitialized nodes and defines them as *candidates*.

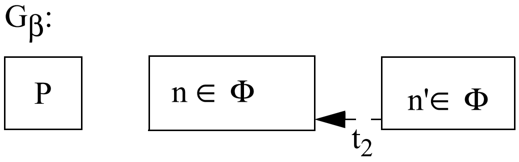
By marking all child nodes that must be recomputed (as ϕ), the predecessors of these children become *candidates*. These seeds are considered OPEN nodes for an A* search if at the end of all the changes, they have not themselves become uninitialized. If the predecessor does become marked as uninitialized, this can only occur if *its* predecessor is a seed, providing the propagation to the uninitialized nodes. The function of identifying and marking the children and adding each of their predecessors to the candidates is the function that χ ( ) provides.

Rather than recompute n based upon its predecessors, it seems more direct to use χ (n) to cause the recalculation of all the nodes related to n in a homogeneous way.

Required Difference Calculation: perform χ (n).

Case 2: Deletion of Active Transition

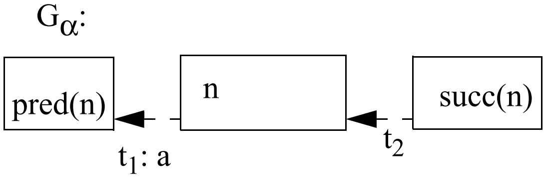


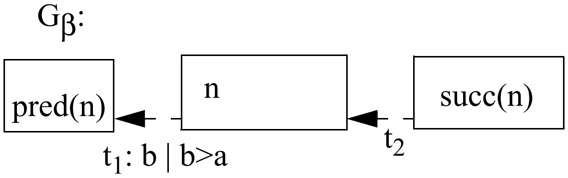


In this case, t_1 is deleted. In G_α, p_1 indicates that n optimally points to P. In G_β, this will be impacted because the transition no longer exists. Therefore, another pred(n) will have to be optimal parent of n if n has a pointer, so pred(n) is required as part of the perimeter for this case. In addition, the value of g(n) will now need to be changed and updated based upon the best transition between n and PRED(n) and the pointers pointing out of n (i.e. p_1) must be deleted[1], i.e. n goes into Φ. In addition, all of the recursive children of n must be reset. This is achieved by χ (n).

Required Difference Calculation: perform χ (n).

Case 3: Increase in Cost of Passive Transition





In this case, the value of t_1 is increased, but in G_α there is no pointer indicating that n optimally points to this predecessor of n, which means the predecessor is not a parent of n. The node n may be pointing to other predecessors who are parents, however. Since this transition was not selected to have a corresponding pointer, it is not in the explicated graph G_e. By increasing the cost of this transition, the desirability of this transition is further reduced, but causes no change to the explicated graph. Therefore, there are no fundamental operations required. The only action is to note the transition cost change.

Required Difference Calculations: None.

---

[1] Note that the same results for deleting a transition can be achieved by setting the cost of $t_1 = \infty$. This means increasing the cost of the transition to a value that is daunting for the problem, resulting in a G_e without this transition. It is therefore not surprising that Case 2 is treated the same as Case 1 which manages increased cost transitions.

Case 4: Deletion of Passive Transition

In this case, the t_1 is deleted, and in G_α there is no pointer indicating that n optimally points to this predecessor of n. The node n may be pointing to other predecessors who are parents, however. Since this transition was not selected to have a corresponding pointer, it is not in the explicated graph G_e. By deleting the transition, there is no change to the explicated graph. Therefore, there are no fundamental operations required.

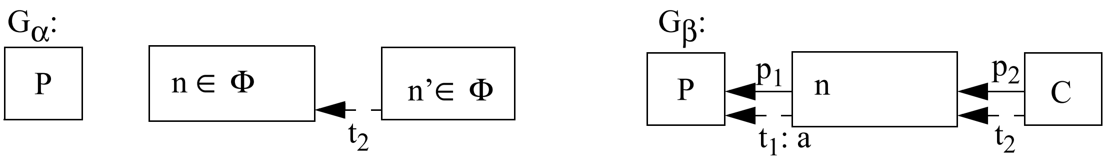
Required Difference Calculations: None.

Case 5: Decrease in Cost of an Active Transition

In this case, the value of transition t_1 is decreased. In G_α, p_1 indicates that n optimally points to P. In G_β, the optimality will not change the existence of pointer p_1, because t_1 is even more desirable than before. Because this transition can only affect the node n, followed by its children, it is sufficient to recompute (as in Section 5.2.5) n using the new lower cost of t1 and add n to the set of candidates. The subsequent invocation of A* will manage the propagation of this lower cost throughout any children and possibly other nodes in the graph.

Required Difference Calculations: 1) Recompute n, 2) Add n as a candidate.

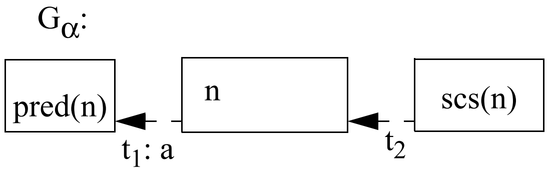
Case 6: Insertion of an Active Transition

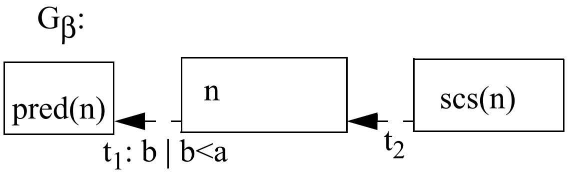


This case can not occur, but is included for completeness. The only changes permitted are in G, so while it is possible to insert passive transitions (since they are in G), whether a transition is active is only determined after A* is performed (i.e. active transitions are in G_e). The difference engine cannot tell a priori that a transition will be active. (Insertion of passive transitions is discussed later in Case 8.)

Required Difference Calculations: None.

Case 7: Decrease in Cost of a Passive Transition

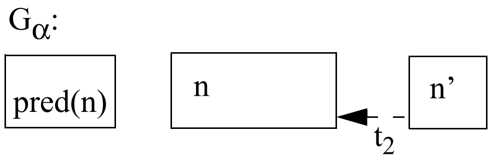


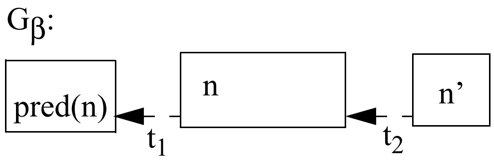


In this case, the value of t_1 has decreased; therefore, it is possible that this may cause it to become active. For this computation to be made, either the node n must be recomputed with the new value of the transition with a comparison to the current value and new pointers set, or more simply, the predecessor may be added to the candidates, whereby A* in normal course performs the computations above.

Required Difference Calculations: Add the predecessor corresponding to $t_1$to the set of candidates.

Case 8: Insertion of a Passive Transition

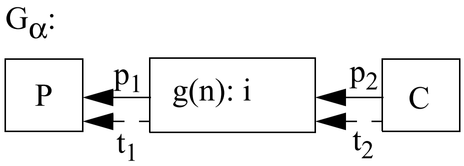


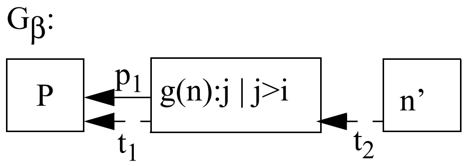


In this case, the transition t_1 is inserted. As stated in Case 6, it cannot be known in advance whether this transition will be active or passive, so all transitions are assumed to be passive until shown otherwise. The affected node is therefore node n. The value of n could be recomputed, compared with the current value, and new pointers set just as in Case 7, but again it is more efficient to have the predecessor of n added to the set of candidate nodes. (It makes sense that the actions required to manage a new transition are analogous to the situation where a transition decreases from infinity to a finite value i.e. Case 7.)

Required Difference Calculations: Add the predecessor corresponding to t_1 to the set of candidate nodes.

Case 9: Increase in Node Delay Cost



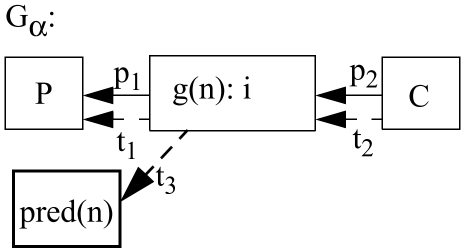


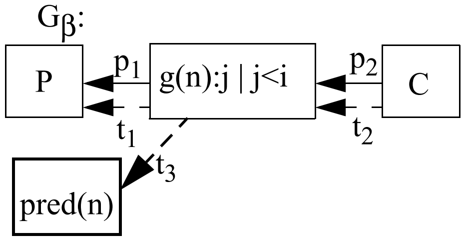
In this case, cost g(n) is increased (from i to j). This means that first, the cost of the node should be recomputed with the new cost. Next, the node will be added to the set of candidate nodes. The pointer p_1 in G_α is not going to change because the transition cost has not changed. The pointer p_2 in G_α will have to be deleted, because the child node C may have a lower cost imparted on it by a potentially different parent node than n. Generally, therefore, for all $C \in$ CHILDREN(n), PRED(C) must be added to the perimeter. This is because the cost of g(n) has increased; therefore, all nodes that depend on this node, that is, recursively all children of n, must be recomputed.

This case could then best be managed by recomputing n and then performing χ (CHILDREN(n)). However, similar to the argument in Case 1, a more homogeneous treatment is to perform χ (n). It is slightly less ideal, since the pointer to the node n will not change, but this is a minor overhead for maintaining a more consistent functional structure.

Required Difference Calculation: χ (n).

Case 10: Decrease in Node Delay Cost

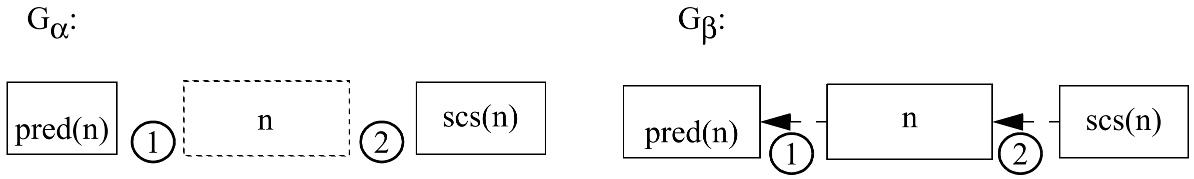




In this case, cost g(n) is decreased (from i to j). Since g(n) will change by a constant value, the t_1 transition will remain active and the t_3 transition will remain passive. Since no change to G_e occurs, then it is sufficient to recompute g(n) and add n to the set of candidate nodes. Because A* will manage the decrease in cost through all of the recursive children, no further action is required.

Required Difference Calculations: 1) recompute n, 2) add n to the set of candidate nodes.

Case 11: Insertion of Node n



The insertion of a node has two steps. The first step is to manage the insertion of (passive) transitions leading into and out of n. The second step is to insert the node itself.

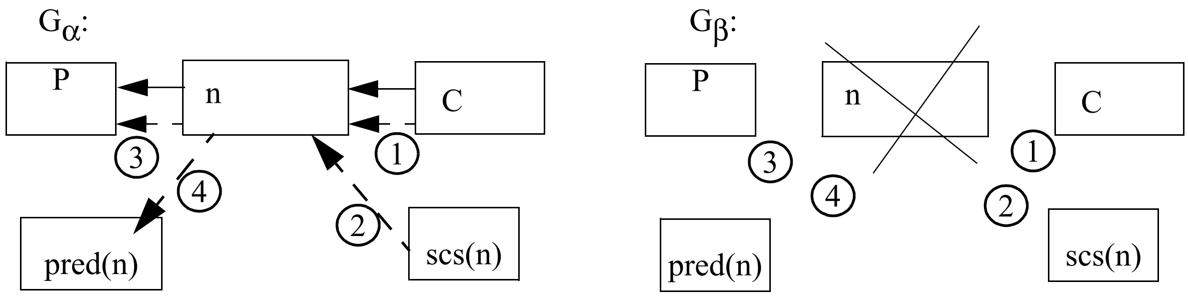
Situation 1 shows a transition being inserted between the new node n and a predecessor. This requires the predecessor corresponding to the transition to be added to the set of candidate nodes. A* performs the remaining computations.

Situation 2 shows a transition leading from a successor to n. In this case, n should be added to the set of candidate nodes.

Taken together, these actions overlap. If 1 and 2 are both added transitions, then it should be noted that it is sufficient to add all predecessors (i.e. PRED(n)) to the set of candidate nodes because the successor(s) will be expanded in due course through A*. If there are no predecessors of n and only successors, then n is not actually connected to G.

Required Difference Calculations: Add all predecessors, PRED(n) to the set of candidate nodes.

Case 12: Deletion of Node n



The deletion of a node has two steps. The first step is to manage the deletion of the active and passive transitions leading into and out of n. These four situations are shown in the diagram above. The second step is to delete the node itself. Because these elements may affect one another, each will be examined with an eye toward the consequences.

Situation 1 shows an active transition with the associated pointer that leads to n from one of its children. The result of deleting the transition according to Case 2 means that PRED(C), in the new situation where n $\notin$ PRED(C), will be added to the perimeter when χ(C) occurs. This will ensure that another parent (if possible) will be found for all C.

Situation 2 shows an incoming passive transition from a successor of n leading to n. The result of deleting this transition according to Case 4 requires no particular action other than the deletion of the transition.

Situation 3 shows an active transition with the associated pointer that leads from n to P. The result of deleting an active transition according to Case 2 is that χ(n) would clear the influence of n, and the recursive children of n. One important note is that since the χ function uses both the successor and predecessor functions, the χ(n) function must be computed before the SCS(n) and PRED(n) functions are invalidated, which could happen as a result of deleting n in the second step.

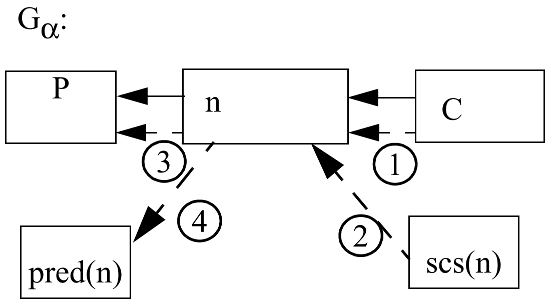
Situation 4 shows a passive transition leading from n to a predecessor of n. The result of deleting this passive transition according to Case 4 is once again that no action is required for n.

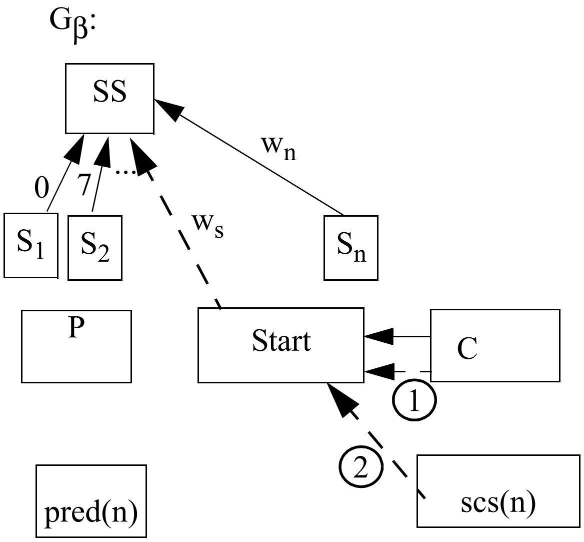
Even though these four situations could be managed separately by ensuring the proper order of computation, two improvements can be obtained by the same recommended change. It is recognized that situation 1 and situation 3 contain duplicate activities. Situation 1 clears the influence starting with the children of n. Situation 3 clears the influence starting with n, which therefore includes n plus its children. Taking these together, a more efficient solution is to add the PRED(n) directly to the set of candidate nodes and then perform χ(C). This also eliminates the complication that PRED(n) and SCS(n) must be valid even if n is not.

The node n is then removed. PRED(n) and CHILDREN(n) operations must be computed before (or at least remain valid after) the node n is removed.

Required Difference Calculations: 1) add PRED(n) to the set of candidate nodes, 2) perform χ(CHILDREN(n)). PRED(n) and CHILDREN(n) operations must be computed before (or at least remain valid after) the node n is removed.

Case 13: Conversion of a Node to a 'Start'

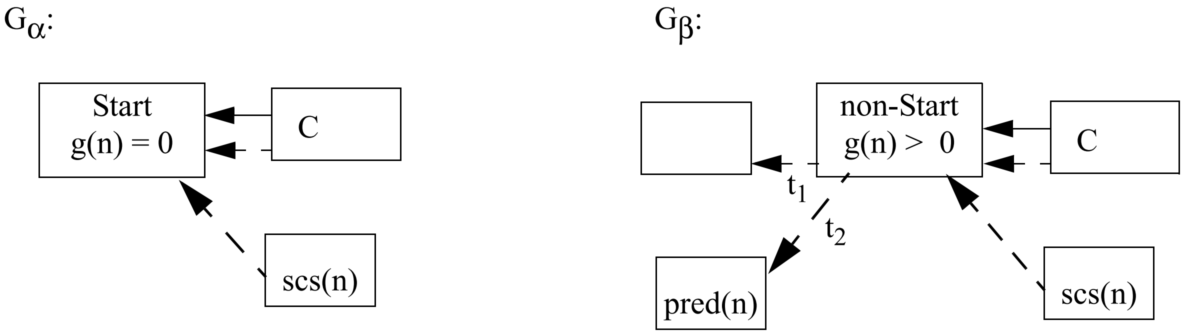




In this case, a node is converted in status to one of the special 'start' nodes. If the node does not exist, then it must first be created as in Case 11. The search begins from this start node. In G_α, the node n has passive and active transitions pointing in and out of it. If the node is labeled 'start', the implication is that the system that generates this node must also set the value of the start, g(n), to some value (usually zero) which is the value that in most instances is a decrease in value to that node. If the start is a descendent of a *super start* (or *dummy node* as in Appendix A) which contains zero cost references to all true starts, then this should cause the actions of Case 10, but also possibly Case 8 to manage transitions w_s (that may be non-zero). In addition, the generating system must delete any transitions that lead out of the starting node taking the actions of Case 2 and Case 4 because these transitions become nonsensical, but will not be active since it is unlikely that a cost will be lower than the start. Also, PARENTS(start) = {} and PRED(start) = {}. This node therefore requires no special treatment, other than to determine which of the relevant cases apply.

Required Difference Calculations: Delete transitions as needed (Case 2 and Case 4). Change costs as required (Case 10). Define PRED(start)={}. Add transitions (Case 8).

Case 14: Conversion of a start node to a non-start node



When a node is no longer in use as a start node, then the cost will be changed either by increasing the transition cost (or removing the transition altogether) between the 'super start' and the start node or by increasing the cost of the node as shown in the diagram. It is also likely that transitions out of the node are then added, such as t_1 and t_2, which had previously been meaningless since the node was the starting node. The changes caused by these transitions are carried out in Case 8. If the transition between the super start and the start is removed, then the delete transition actions of Case 2 is used (because the transition can be thought of as active). If the node delay cost, g(n) is increased, then this is managed in Case 9. Either way, the resulting cost change will affect any successors or child nodes.

Required Difference Calculations: Redefine PRED(start).

5.5 Sequence of Fundamental Operations and Final Actions in the Method

Now that all cases have been analyzed, they have been summarized in Figure 66. This shows that some of these cases have identical actions, and the similarities allow them to be grouped as in Figure 67. This table can be used to design the algorithm that manages each of these changes.

Although within each case, functions may be carried out in any order without any undesirable effects, it may matter in which order the cases are calculated because they can force a different sequence of operations on the same node. Specifically, Groups A and E clear nodes which may be added to the candidates seeds, or nodes that have been recomputed. By examining the combinations of these actions, it can be determined in what order they can be computed.

5.5.1 Recomputed then Added to Candidates

Recomputation of the node and subsequent addition to the candidate seeds ensures that the value added to the candidates is correct at the moment it occurs. Therefore these fundamental operations are valid in this order.

5.5.2 Added to Candidates then Recomputed

This can occur in Group C. If a node is added to candidates and then recomputed, then the candidate's value must be updated as well. Assuming that all costs are referenced indirectly, this occurs automatically. Therefore, these fundamental operations are valid in this order.

5.5.3 Recomputed then Cleared

If a node is recomputed and later cleared, the recomputation is then generally unnecessary, because the value must be recomputed anyway. The only time that this is not true is when the recomputation is a result of Case 13, a starting node insertion. When a start is added, the value of the current node, now deemed the start uses Case 10 to set its node delay cost set to zero, or it may be a descendent of a Super Start node, having the node cost updated to zero indirectly by the Case 8 insertion of a passive node. Whether the node is updated directly or indirectly, the start value (0) must persist, and the node must be added to OPEN. Therefore these fundamental operations are valid in this order other than for operations caused by start insertion.

5.5.4 Cleared then Recomputed

If a node is cleared, and then is attempted for recomputation, then the value of the node must remain ϕ. This is the reason that the uncosted value ϕ plus any value must remain ϕ. The only way to change the value ϕ is to assign a value that is not a function of ϕ. The value of the node is set correctly for the next A* operation. Therefore, these fundamental operations are valid in this order.

5.5.5 Added to Candidates then Cleared

If the node is added to the candidate list and later cleared, then the node will be rejected when the final seeds are determined. Therefore, these fundamental operations are valid in this order.

5.5.6 Cleared then Added to Candidates

If a node is cleared, and is subsequently added to candidates, then the value is being added when it will never be selected as a seed. Although this is a slightly wasteful operation, it is harmless to the result, because at some point, the value will be checked and the node rejected. Therefore, these fundamental operations are valid in this order.

5.5.7 Summary

By examining the cases requiring these fundamental operations we discover that they can be performed in any order, except for operations relating to new start nodes. The starting value (0) must be assigned, and the starting node must be added to the list of candidates, which is then added to OPEN. Any algorithm performing these fundamental operations in any order will accomplish the Differential A* objective. The solution is to process all new starting nodes of Case 13 after all other cases, followed by any related cases.

5.5.8 Final Actions

The objective is to determine the final seed (OPEN) nodes from the candidate nodes. The candidate nodes must include all of the:

- nodes previously on OPEN when the last A* terminated,
- nodes identified by the various cases as candidates,
- nodes identified as the perimeter resulting from the χ function.

It can occur that a candidate may be added to the list by more than one of the three mechanisms above, or, for example, by different calls to $\chi(\ )$. The final seed nodes which will be OPEN in the next step (A*) can be created with the following function:

$$\{\text{OPEN nodes}\} = \{\text{candidate nodes}\} \setminus \Phi.$$

This ensures that there is only one copy of each of the candidate nodes. It also ensures that no nodes that were once candidates, but were later set to ϕ by a subsequent call to χ, are removed from the final seed nodes.

Case #: Type	Recompute	Add to candidates	χ ()
Case 1: Increase in Cost of Active Transitions			n
Case 2: Deletion of Active Transition			n
Case 3: Increase in Cost of Passive Transition	NONE		
Case 4: Deletion of Passive Transition	NONE		
Case 5: Decrease in Cost of an Active Transition	n	n	
Case 6: Insertion of an Active Transition	NONE		
Case 7: Decrease in Cost of a Passive Transition		PRED(t)	
Case 8: Insertion of a Passive Transition		PRED(t)	
Case 9: Increase in Node Delay Cost			n
Case 10: Decrease in Node Delay Cost	n	n	
Case 11: Insertion of Node n		PRED(n)	
Case 12: Deletion of Node n[a]		PRED(n)	CHILDREN(n)
Case 13: Conversion of a Node to a ‘Start’[b]	NONE		
Case 14: Conversion of a start node to a non-start node[c]	NONE		

[a] Operations must be computed before (or at least remain valid after) the node n is removed.

[b] For new start, define Super Start $\in$ PRED(start), g(start)=0, but involves a number of other cases.

[c] For non-start node, define Super Start $\notin$ PRED(start), g(start) = ϕ, but involves a number of other cases.

Figure 66: Required Functions for Differential A Cases.*

Group Case #: Type	Recompute	Add to list of candidate seeds	$\chi()$
Group A Case 1: Increase in Cost of Active Transitions Case 2: Deletion of Active Transition Case 9: Increase in Node Delay Cost			n
Group B Case 3: Increase in Cost of Passive Transition Case 4: Deletion of Passive Transition Case 6: Insertion of an Active Transition Case 13: Conversion of a Node to a 'Start'[a] Case 14: Conversion of a start node to a non-start node[b]	NONE		
Group C Case 5: Decrease in Cost of an Active Transition Case 10: Decrease in Node Delay Cost	n	n	
Group D Case 7: Decrease in Cost of a Passive Transition Case 8: Insertion of a Passive Transition Case 11: Insertion of Node n		 PRED(t) PRED(n)	
Group E Case 12: Deletion of Node n[c]		PRED(n)	CHILDREN(n)

[a] For new start, define Super Start $\in$ PRED(start). g(start)=0, but involves a number of other cases.

[b] For non-start node, define Super Start $\notin$ PRED(start). g(start)=ϕ, but involves a number of other cases.

[c] Operations must be computed before (or at least remain valid after) the node n is removed.

Figure 67: Grouped Differential A Cases.*

5.6 Analysis of the Differential A* Method

Functionally, both A* and Differential A* achieve the same results; they compute G_e, but with different levels of efficiency. Obviously, the issue is: when is Differential A* preferable to A*. The relative efficiencies are addressed in this section.

In Differential A*, there are two general steps. The first is the use of the difference engine to analyze the changed nodes. The second is to compute A* beginning with those changed nodes rather than from a starting state. The three functions used in the difference engine will be analyzed and then their impact on the number of nodes will be computed. The functions are (i) recompute, (ii) add to candidates, and (iii) $\chi()$. A* is then computed conventionally, except that it begins with the seed nodes rather than the starting nodes.

5.6.1 Recompute

To perform the *recompute* function for a single node (required in cases Case 5 and Case 10 of Group C), the only values required are those of potential parent nodes. Therefore the predecessors PRED(n) of the node must be found, and the best cost found. Assuming that the number of predecessors is equal to the number of successors m (from Section 2.5.1), the time required to find the successors is M. Assuming the time for recomputing the cost is R, then $M + mR$ represents the total recomputation time. Often the number of successors and cost computation can be treated as a fixed cost, or O(1) per node.

5.6.2 Add to Candidates

Adding to the candidates is required in the cases of Groups C, D and E, and is performed in constant time, or O(1). This is because adding to a list can be performed in fixed time.

5.6.3 $\chi()$

To perform $\chi()$ in G_e, which occurs in Cases 1, 2, 9 and 12, all nodes recursively dependent on a given node are reset to ϕ and then the perimeter $\mathrm{pred}(\Phi)\backslash\,\Phi$ are added to the candidates. In advance, it may be difficult to estimate the number of nodes that will be affected by the χ function. The best case is that there are no successors to a node. The worst case is when all nodes are dependent on a given node. Typically, only a portion of the graph is affected as will be seen in Figure 70, and in other examples in Chapter 6. The χ function also finds the perimeter by using the predecessor function performed on each of the final set of cleared nodes, at a computation time of P.

5.6.4 Impact of Groups

Now that the three main functions have been discussed, the separate Groups of Figure 67 will be examined. We can observe that the recompute function and the adding to the candidates list are not expensive in terms of time. It is clear that the groups that involve χ ( ) that will be the most impactful in terms of time. In each group it will be determined when it is better to use Differential A* and when it is better to fully recompute the graph using A*.

Group A

Group A consists of nodes and transitions that cause a ripple effect of increased costs to the node and its children. Because these are increased costs, the affected area must be cleared and recomputed because A* can only serve to lower node costs. The clearing of the affected area is achieved with χ ( ).

If this is the precise region that is required to complete the next A* computation, then the primary overhead for Differential A*, is the time required for χ ( ) and setting up the perimeter in OPEN.

Often, a cleared region (denoted n_χ) represents only a portion of the given graph. Compared to complete recalculation with A*, there will likely be some number of nodes that are saved from recalculation. The number of these nodes is $(N - n_\chi)$. When the overhead for clearing the nodes using χ ( ) is less than recomputing the potentially saved nodes, then Differential A* will be more effective.

It is not guaranteed however that the cleared portion is either recomputed, or the only area that is affected by the change. For example, if we remove the last transition connecting one part of the graph to another, then the later graph will be cleared but never recomputed. The cleared region may be large, but the actual repropagation area is negligible. This is an extreme case. It is also possible that in the same computation, some transitions will be added while others are removed. In this event it is possible for the cleared nodes to be few, but cause a major recalculation for A*.

In this group of changes, the number of nodes cleared will vary according to the varied characteristics of the graph, therefore the ability to estimate the number of cleared nodes for particular situations and some knowledge of the frequency that they occur can help predict whether Differential A* or A* is preferable. This can be difficult, but for some applications, (with an example in the next chapter) it may be estimated.

Group B

Group B consists of transitions that have no immediate impact on the graph G_e, and the relabeling of some nodes as start or non-start. The start/non-start relabeling itself has no impact, but the likely case is that the changes that occur as a result, such as decreasing the cost of the node, or changing the associated transitions, could cause changes in the graph G_e. These changes are managed independently in the respective groups, however, so we will not consider them here.

The greatest gain in using Differential A* occurs in the situations when the only changes are in Group B transitions. This is because the transitions have no effect on the graph G_e; therefore, no computations are required. If this was not observed, then these changes would ordinarily cause the complete and unnecessary recomputation of the graph.

In this group of changes, Differential A* is always more effective because it saves all A* operations.

Group C

Group C consists of nodes and transitions that have decreased cost which will require the recomputation of the node itself and its children, the total number of nodes defining n_r. No clearing is required because a lower cost can be propagated correctly to the children with A*. The overhead for recomputing a node and adding it to the candidate nodes is a constant, very small, and would also need to have been performed by A*. If n_r happens to equal the number of nodes computed by A* from scratch, then the overhead for using Differential A* is essentially zero. Often however, the number of nodes n_r represents only a portion of the total graph. Therefore, the savings occur in the number of nodes $n_s = N - n_r$ that were saved from unnecessary recomputation. According to Section 2.5.2, this would have taken $O(n_s \log n_s)$ time.

In this group, Differential A* is at least as effective as A* because it saves all $O(n_s \log_2 n_s)$ operations. At the worst, it is the same as A*.

Group D

Group D consists of passive transitions that have decreased cost, and the insertion of a new node with associated new transitions. In these cases it is unclear whether the change will affect the graph in a large or small way. A passive transition in itself indicates that there is some other, lower cost transition that more preferably leads from the current node toward the node(s) representing the start of the search. Therefore, to lower the cost of a passive transition, there is no way to know a priori if the cost is sufficiently low to force it to become the new (lower cost) active transition. If it is sufficiently low, then through A*, this would require the recomputation of the node itself and its children. If it is not low enough, then no other computations would be required.

Accordingly, the nodes or transitions in this group will fall either into Group B or Group C, depending upon how much the costs decrease with respect to the other transitions. Consequently, some of the same discussion as in Groups B and C also applies here. If the costs are not sufficiently low to force the propagation of new costs through to children, then this change can save all operations as it does in Group B. Even when cost propagation is required, as in Group C, if this is the precise region of n_r nodes that is required to complete the next A* computation, then the overhead for using Differential A* is essentially zero. However, it may be that this computed region represents only a portion of the given graph. The overall savings occurs in the number of nodes n_x that were not unnecessarily recomputed.

In this group, Differential A* either saves all $O(N \log_2 N)$ operations or $O(n_s \log_2 n_s)$ operations depending on whether it results in Group B or C actions. It is always preferable to A*.

Group E

Group E consists of the deletion of a node. This forces the removal of the node and its associated transitions. Removal of a passive transition has no effect in situations 2 and 4 of Case 12 (Group B). However, when the node is completely disconnected from the graph, the removal of the node may have an effect such as in situations 1 or 3 (of Case 12). This has an effect similar to the removal of an active transition in Group A. The main difference is that the node itself is removed; therefore, the influence must be cleared starting with the children, and the predecessors of the deleted node will be the candidates to originate the new search.

Therefore, this node will have the effect of Group D, which in turn results in either Group B or C, and similar actions to Group A. The effect of Groups B and C are that Differential A* has virtually no overhead, and may save some time. The affect of performing χ (CHILDREN(n)) as in Group A, however may result in either A* or Differential A* being more effective.

In this group, as in Group A, no general statement of Differential A* effectiveness can be made.

5.6.5 Impact of Admissible Heuristics

Whether a graph is computed with a heuristic or not, the same fundamental operations apply to each group of changes described in Section 5.6.4. Nothing specific to obstacles or transitions with this type of change is different. Influenced areas will be cleared, but they will be smaller, in accordance with the focus of the heuristic. So, Differential A* is able to manage changes in nodes and transitions in a straightforward fashion.

Still, the heuristic itself may change because it is a function of the starting state (goal node). Although not the core concern of this thesis, this is another type of global change that can affect the graph. When an admissible heuristic function changes, which can happen if the goal node moves, then the proper operations vary depending upon the current situation. Initially, it may be said that this is in essence a change in cost, forcing full recomputation with the new heuristic. Alternatively, we may observe other options based on the new location of the goal node relative to the previous goal node.

If the new goal node is within the existing G_e, that is to say, the goal has a value and pointers as a result of the prior calculation, and is not part of OPEN, then the values of g() and pointers will be correct to connect between the start and this new goal. In this situation, no further calculations are required.

If the new goal node is in OPEN however, then this indicates that not all computations have been completed to assure that *delayed termination*[1] has occurred (for the cost of this new goal), and that an optimal path is not guaranteed. Because the node is already in OPEN, it is likely that with only a few node expansions, a solution can be found. This provides a way to build on the previous work performed by A*. In this situation, it is best to recompute the open nodes with the new heuristic

---

[1] Recall that delayed termination forces computation to continue until the cost of newly expanding OPEN node is greater than the cost of the goal node.

value and continue the A* calculation until delayed termination is achieved. Note that any previously calculated nodes not in OPEN, will have an incorrect heuristic value (if it is stored with the node). Still, a correct path is found with minimal work, and the computed values of the g function are correct.

If the node is in neither within G_e or OPEN, then the only way to guarantee that a correct path is found is to recompute the OPEN nodes with the new heuristic and calculate until delayed termination occurs. This may require substantial overhead with little return, such as if the start is nearer to the new goal than any OPEN node. Naturally, it depends on the graph, heuristic, and the locations of the start and goal. In this situation, it may be more conservative to completely recompute the graph with the new heuristic.

5.7 Summary Comparison between A* and Differential A*

It is clear that Differential A* is guaranteed to provide a dramatic improvement over A* for Group B, C and D changes, since there is virtually no overhead, and fewer nodes must be recomputed than if the complete graph is recomputed. Groups A and E however, require overhead for the χ function to identify and clear dependent nodes, and find the perimeter from which to begin A*, plus the A* computation for this area.

These similar groups (A and E) may also provide a strong improvement, but must be compared to full recomputation with A*. The key Differential A* computation time (Time_δ) is then, the sum of the time to clear the nodes with χ, the time to move nodes from the perimeter into Candidates, plus the time for the A* recomputation based on these nodes and others in OPEN. This gives a way to define the situations when Differential A* is preferable.

$$\mathrm{Time}_\delta = \mathrm{Time}(\chi) + \mathrm{Time}(\mathrm{Perim}) + A^*(\mathrm{OPEN})$$

Depending upon the situation, and particularly the number of nodes affected by groups A and E, Differential A* can provide drastic improvement, or cause slight overhead. It must be determined in each practical situation the frequency that each group will occur, an estimate of the impact on nodes, and the affect on the overall objective, i.e. whether the task can tolerate the worst case.

If for example, all costs in the graph were to change, perhaps due to a global change in robot's cost criterion (e.g. from the *least communication* depicted in Figure 25 to the *straightest path* depicted Figure 28, both from Chapter 3) then it is certain that many node costs and pointers will be changed. In this situation it is clearly preferable to compute the graph from scratch, not using Differential A*.

Because Groups A and E are pivotal, they are a major focus of Chapter 6 with specific timing gathered in Section 6.6. Specifically, the comparative utility for each method is quantified, including an estimate of the number of nodes affected by the function χ for a given obstacle.

Chapter 6

Mapping Differential A* to Robotics

6.1 Environment Changes in Robotics

Robotics is a field which exemplifies intelligent autonomous systems. Robots can be used in situations that require the ability to change the planned motion as quickly as possible. The objective is to modify a navigation map, perhaps already containing a plan accounting for essential constraints (joint limits, fixed known obstacles, etc.) of the workcell, as rapidly as possible so that motion can proceed. An important attribute of a partially recomputed space, is that the current motion can continue unless directly affected by the change.

We wish to have efficient responses for as many common changes as possible. A list of typical types of robotic changes includes:

I) The appearance of new obstacles in a largely known and stable environment. A quick response can be an efficient mechanism for 'learning' about objects at runtime.

II) Obstacles that move.

III) Change of start.

IV) A change of goals (inserted, moved, or deleted).

V) A change in optimality criterion. With the same obstacle layout, time may be valuable for one task, then energy for the next task.

A quick mechanism to adapt to all of these changes makes the robotic system more productive.

6.2 Differential A*: Managing Changes

6.2.1 Implementation Assumptions

In this robotics application, we prefer to maintain the configuration space in a regularized discretized *grid* of nodes representing configuration space, and change the delay costs to force some regions to be forbidden. The reason to maintain the grid is that obstacles often move in the robot domain, potentially causing frequent allocation and de-allocation of nodes. For example, rather than physically removing a node when an obstacle obstructs it, increasing the delay cost to infinity can cause the same change. When the same node is reintroduced, memory allocation routines can be avoided by changing (decreasing) the delay cost from infinity to *uninitialized*.

Differential A* can be used to respond to all of the changes listed in Section 6.1, by focusing on the changes that occur in a typical robotics application, such as for the two link robot of Figure 15. The frequent changes are the insertion, movement and deletion of obstacles and goals. Common changes are possible, but infrequent in our application, so this chapter will focus on changes to nodes.

6.2.2 Representative Cases for Changes

From Figure 67, we can select representative cases describing the robotic changes. Once the cases are found, they become the basis for a Differential A* algorithm used for robotics.

“Type I Changes”: New and Removed Obstacles

Typical robots have standard, ever-present obstacles and limitations in the workcell. The robot has certain joint limits, and a fixed work environment including walls, floors, and other forbidden interactions such as robot links that must not intersect with other robot links. Ideally, this standard environment can be captured once, and then be adapted to new or smaller changes.

A new obstacle state is either inserted or deleted (from possibly overlapping transformed obstacles) if the overall result is that the obstacle state is newly inserted or deleted. The inserted obstacles do not cause states to be deleted physically from the configuration space, but rather they are assigned an infinite ∞ (unattainable and high) delay, as in Case 9. Deleted obstacles lower this cost from infinite to uninitialized, or in other words, reset the state to ϕ (as defined in Section 5.2.1). This corresponds to decreasing the node delay cost of Case 10.

In robotics, often the environment in which the robot moves may have moving obstacles or there may be incorrect position information about the obstacles. If a robot follows an optimal path based on knowledge of obstacles that was assumed to be correct and complete, yet detects (perhaps with a proximity sensor such as skin or sonar) an unexpected obstacle, it is preferable to be able to incorporate this incremental information quickly so that a more informed path can be followed.

In an interactive environment these types of change may be common. The shape and motion of obstacles may be discovered only at run time, and the robot must *learn* about them and react appropriately. The faster the planner can sense and react, the better the overall response can be. But for fastest response, lower-level reflex control may be required. Reflex control typically performs instantaneous and pre-programmed responses, such as ‘reverse direction’, or ‘when in danger, stop’. Ideally, a fast planner can keep the objective (goal) in focus, while deviating from the current plan appropriately.

If the changes (to configuration space) are small, then Differential A* is an effective way to adapt the navigation map quickly. As the changes become very large however, it may be preferable to fully recompute the space. The trade-off between A* and Differential A* will be discussed in Section 6.6.

In Differential A*, new and removed obstacles are computed by Cases 9 and 10.

“Type II Changes”: (Known) Moving Obstacles

Moving Obstacles can be thought of as a combination of removing and adding obstacle states. Conveyor belts may bring in new parts - some which enter the workspace of the robot. Some obstacles are created by the robot, as a workpiece is assembled, for instance. This is a situation where some obstacle states are deleted and others are added as the part moves through task space. Because moving obstacles typically leave some configuration states while adding others, the total changes are often small, and can be computed quickly. Naturally, the faster the planner and sensing system are, the fewer changes to the configuration space will be performed each time.

Moving obstacles are computed by Cases 9 and 10, so these are actually the same operations to adapt to changes that are found in “Type I Changes”.

“Type III Changes”: Change of Start

In Section 3.3, the goal *states*[1] were mapped to the starting *nodes* of standard A*, and a zero heuristic is used so that a path is provided from every possible starting state leading to the goal. This provides an efficient mechanism to recover from unplanned changes in the current (starting) position, without recomputation.

Differential A* is not required, because the change is managed by the existing A* framework.

“Type IV Changes”: Goal Changes

It is clear that as a robot performs its tasks, the goal location must change. While it is possible to set a few known goal locations which are standard, and maintain several configuration spaces simultaneously, seemingly arbitrary goals may be selected by the task planner.

Goals are likely to change according to the instructions of the task planner. Inserting a new goal requires changing a state to a goal state and is achieved by identifying it as a ‘starting node’ for the search and decreasing the delay cost to zero. This corresponds to Case 13 and Case 10. Deletion of a goal requires changing the goal state to an uninitialized state and is achieved by converting it to a non-start node, and increasing the delay cost to uninitialized, or ϕ. This corresponds to Case 14 and Case 9.

We should observe however that if a single goal is moved, that is, deleted and added elsewhere, then the entire configuration space must be first cleared and then regenerated from the new location. This clear inefficiency should prevent the use of Differential A* if the primary use of the

---

[1] Recall that *states* refer to configuration space and *nodes* refer to general graphs.

robot is to constantly plan to single new goals. If there is more than one goal, then some of the configuration space is likely to be preserved, and again, Differential A* is applicable.

“Type V Changes”: Change in Cost Criterion

No matter how the obstacles or goals change, if the cost measure is changed throughout the entire configuration space, then all nodes must be recomputed. This is covered by changes in transition and node costs, covering Cases 3, 7, 9 and 10. Clearly it is preferable to avoid any potential overhead of Differential A*, by simply using A* to recompute the space.

For this case, A* is preferable over Differential A*.

6.2.3 Selecting a General Application

Two types of changes cover the desired actions. “Type I Changes” which adapt the space to obstacles and “Type IV changes” which adapt it to goals can be combined, requiring cases 9, 10, 13 and 14. Type II changes are actually the same operations as Type I changes. Changes to the start (Type III) are already managed by a zero heuristic, which will be assumed if this type of change is likely to be needed for the system. The resulting algorithm that covers all of the Type I-IV changes above, is introduced here. We define an algorithm which first computes Group A changes, and then Group C changes, recalling from Section 5.5 that new goal states (starting nodes) are managed last.

In addition, the performance of Group A (specifically Case 9) can be used as an example to test the efficiency of Differential A*.

6.3 Differential A* Algorithm for Robotics

The Differential A* algorithm [6] begins with the current configuration space, detects changed obstacles and goals, performs the fundamental operations of the difference engine and finally performs A* on the OPEN nodes to recompute the space. This is shown in block diagram form in Figure 68. It corresponds to the original Differential A* outline of Figure 62. The previous configuration space (G and G_e) structure, with all remaining OPEN nodes, is one component needed for the *difference engine* and changes to obstacles and goals is the other. The output of the difference engine is a selection of states to be used as OPEN nodes for the subsequent A* calculation.

6.3.1 Differential A* - Graphical Motivation and Robotic Implementation

In the difference engine, We first treat all cases that require the function χ, perform the clearing function of χ, followed by the other operations. The order is not critical other than finishing the algorithm by inserting the goals (recalling that they are starting nodes for the search). To implement the algorithm, we provide corresponding application pseudo-code in Figure 69.

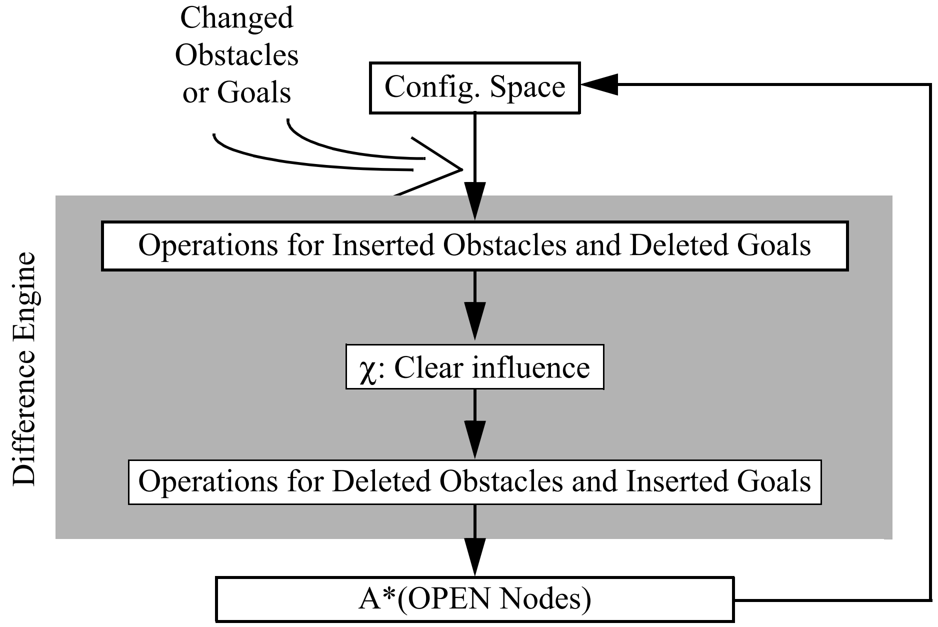


Figure 68: Differential A Flowchart for Robotics Application.*

It helps to view the algorithm results graphically before embarking on the details, so that the objectives are more clear. Figure 70 illustrates the series of steps required to perform Differential A* to adapt to the sudden arrival of an obstacle. In practice, sudden arrivals should be unusual, because the system typically senses the obstacles's arrival more gradually as it comes into view. The Differential A* operations are matched to the results of complete calculation with A* alone.

Figure 70(a) shows an example of a configuration space graph. In this example there are no OPEN nodes, because the entire space is filled. The initial configuration space will be considered a stable graph from which changes are defined.

The first action of the algorithm is to detect obstacle or goal changes in the configuration space. In Figure 70(b) the obstacle transformed from task space creates two forbidden regions. Each state in the region is assigned an increased cost of infinity (∞).

The second action is for the Difference Engine to perform the required fundamental operations on these nodes, specifically for Case 9. This means that for each of the newly forbidden states, the function χ must be performed. Because obstacles in robotics tend be filled regions, it is useful to treat them together, rather than completing the operations node by node. By performing this action on all nodes simultaneously, the non-obstacle predecessors of the obstacle are added to the candidates, and all non-obstacle successors are the nodes to be cleared. If there had been a removed goal in this example, the cost and pointers would be reset by Case 14 and the effect

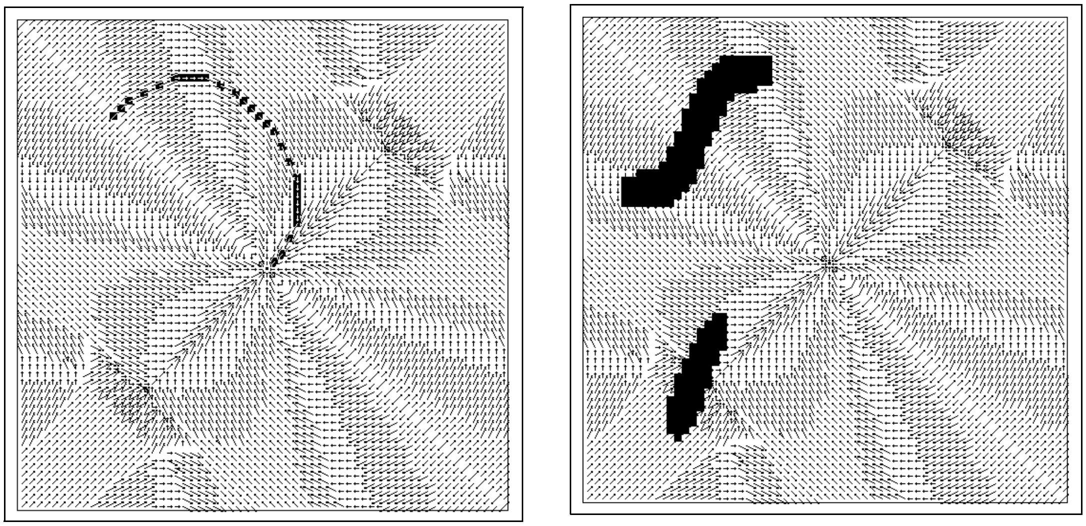
Definitions

O: Set of all obstacles, nodes with ∞ delay
C: Set of Candidate Nodes
ϕ: Uninitialized (Cleared) State (no pointers)
Φ: Set of all ϕ
OPEN: Set of nodes to be calculated
Z: Set of Cleared Nodes that Originate further Clearing
∞: Highest representable cost, otherwise unreachable.

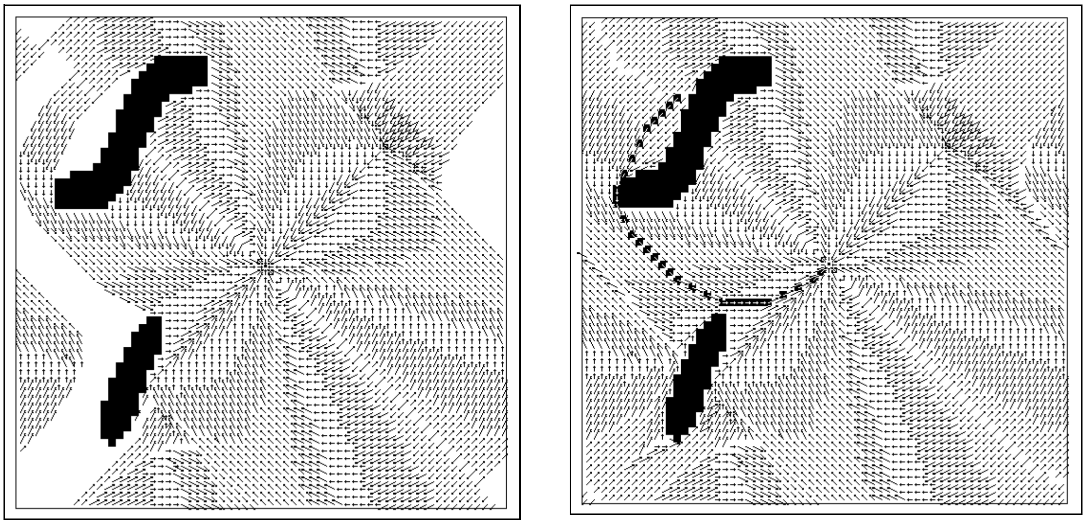
Pseudo-Code

```
For each newly inserted obstacle state $o_a$ having node delay = $\infty$
        if ($pred(o_a) \notin O$)
                $C \Leftarrow pred(o_a)$
        if ($scs(o_a) \notin O$)
                $Z \Leftarrow scs(o_a)$

For each removed goal state $g_r$
        $Z \Leftarrow g_r$

For each state n $\in Z$
    for each $scs(n), n'$
        if $\exists$ any n' s.t. ($n' \in parent$(n) and (($n' \in O$) or ($n' \in \Phi$)))
                n $= \phi$,   $Z \Leftarrow scs(n)$
        otherwise
                $C \Leftarrow$ n

For each newly deleted obstacle state $o_r$ with g($o_r$)= $\phi$
    for each $pred(o_r), n'$
        if ($n' \notin \Phi$)
            $C \Leftarrow n'$

For each inserted goal state $g_a$ having g($g_a$)=0
        $C \Leftarrow g_a$

For each state $c \in C$
        if ($c \notin \Phi$) and ($c \notin$ O)
            $OPEN \Leftarrow c$

A*($OPEN$)
```

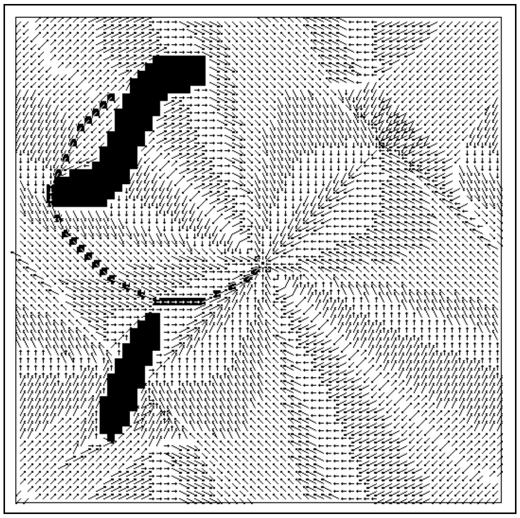
Figure 69: Differential A Pseudo-Code for Robotics.*



(a) Stable Solution Graph (b) New Obstacles (many states)



(c) χ (Obstacles) Produce Affected Area (d) Arrow Pattern Adjusted



(e) Recomputation with A* (from Scratch)

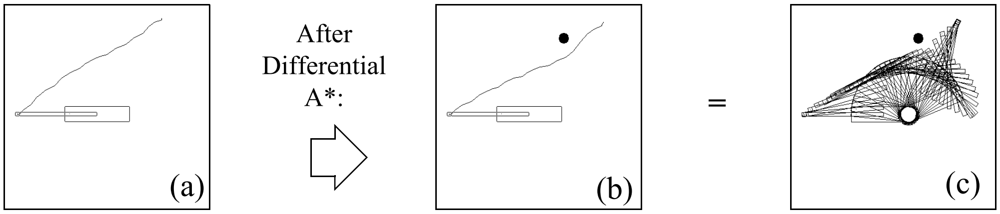
Figure 70: Differential A Modifications for New Obstacle States in Configuration Space Compared to Recomputation with A*. (Example of a "Type* I *Change")*

would be managed by Case 9. The node delay cost would be increased, and the function χ performed on the removed goal. This case is illustrated later in Figure 76.

The third action carries out the clearing function χ for all affected nodes as part of the Difference Engine. This re-initializes every area affected by the new obstacles and (possible) removed goals all at once. It re-initializes all pointers and costs of affected nodes in G_e and G respectively by tracing those nodes whose paths would lead through one of the changed nodes. Nodes just beyond those affected nodes are ultimately identified as being in the *perimeter* of the *affected area*. The affected area is shown in white in Figure 70(c), with the perimeter being the surrounding nodes.

The fourth action is also part of the Difference Engine and performs required operations on deleted obstacles and inserted goals. In this implementation, deleted obstacles are treated as Case 11, adding the predecessors of the node to the list of candidates, with the delay cost set to ϕ from infinity. If a goal is added, then its cost is set to zero, and as in Case 10, adds it to the list of candidates.

After these four actions, Differential A* all of the candidate nodes are examined to ensure they are valued (i.e. not ϕ) and then added to OPEN.



(a) TS corresponding to CS in Figure 70(a).
(b) TS with new obstacle corresponding to Figure 70(d).
(c) Swept arm motion of Figure 71(b)

Figure 71: Task Space Corresponding to Figure 70.

A* is initiated from the lowest cost node in OPEN. When the graph search is complete, the new solution graph G_e, will give the local directions at each node to generate globally minimum path(s) to the goal. The resulting configuration space is shown in Figure 70(d) with the pointers corrected. To demonstrate the correctness of the solution, the same setup is used to compute with A* from scratch. Detailed comparison has shown that all pointers are indeed correct. In the figure, note that the path runs past the bottom edge of the upper obstacle, not through it[1].

The obstacle that was transformed into Figure 70 is shown in its native *task space* in Figure 71. The figure shows the original environment in (a), and then after the introduction of the obstacle, and the Differential A* computation, a revised path is found in (b). The path shows the trace of

---

[1] In the computer simulation, color is used to differentiate the obstacle in black and the path in red.

the end effector in (b) and a ‘swept arm’ in (c). Figure 71(c) shows that the obstacle was indeed properly transformed, ensuring that no part of the robot will collide when the path is carried out.

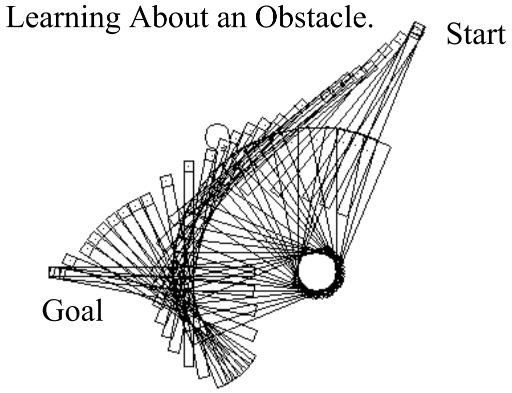
6.4 Various Examples of Changes

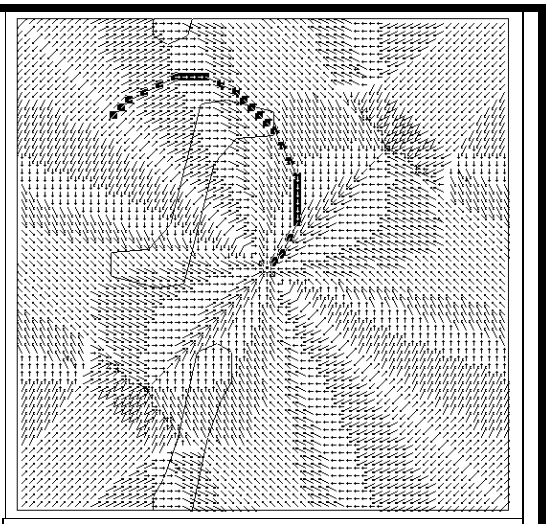
This section gives 3 examples covering changes of type “I, II and IV” above. Type III is not presented because it is solved using a zero heuristic, and is seen throughout this thesis. Type V is not presented since it is a change that is better processed by using A* rather than Differential A*.

6.4.1 Application to Path Planning with Obstacle Discovery (“Type I Changes”)

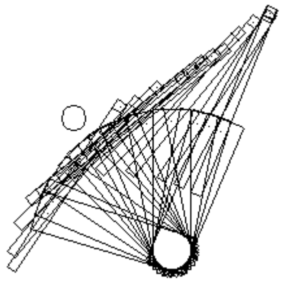
In Figure 72(a), an obstacle that the robot does not know about is introduced into the environment. The obstacle in Figure 72, it is a black edged white circle.

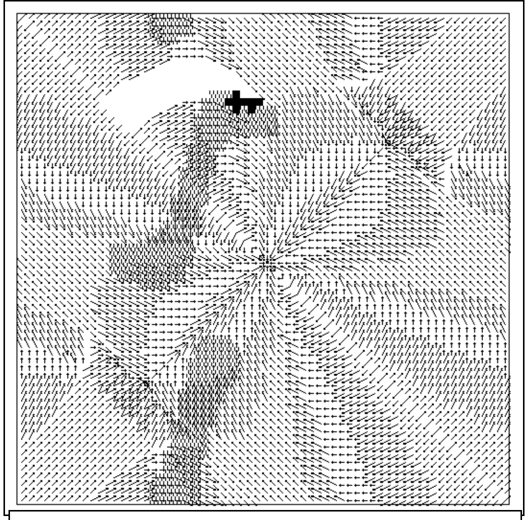
The robot discovers the obstacle only when it is hit, and it is incorporated incrementally as the robot senses the edge of the obstacle state by state. Since only a small amount of obstacle information is discovered and incorporated at a time, the Differential A* method provides a quick adaptation. Figure 72(a-e) shows the configuration space change by change, with the discovered obstacle states represented as small black states. This transformation is performed in the most naive fashion. That is, only the configurations that cause a ‘hit’ are forbidden. In practice, when the robot encounters an obstacle, the sensed task-space obstacle location should be transformed to the possibly numerous obstacle states in configuration space. In this case, that would lead to the creation of a thin S shape.



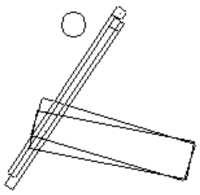


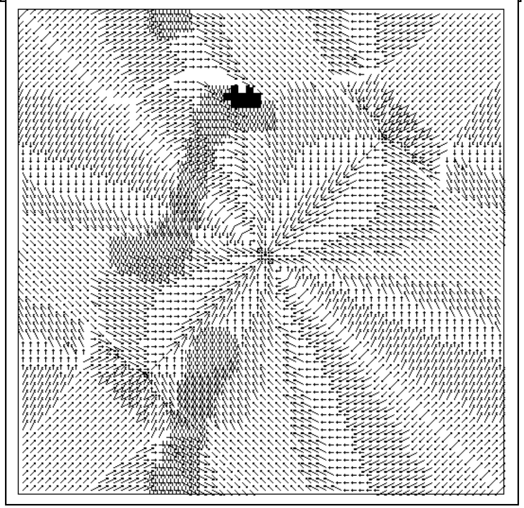
(a) Proposed path with an unforeseen obstacle. Right: CS





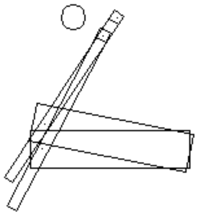
(b) Robot first senses the obstacle. Right: Influenced CS Area is cleared.



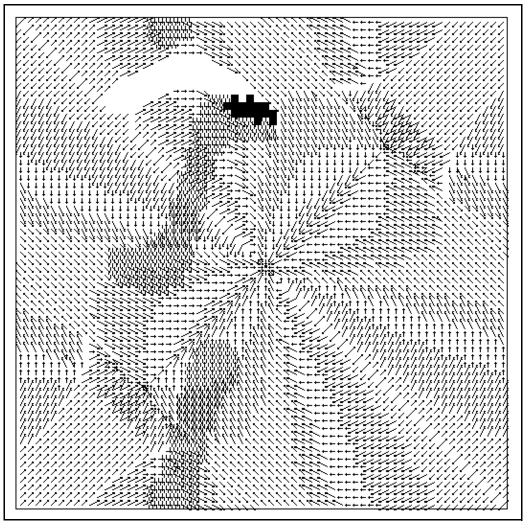


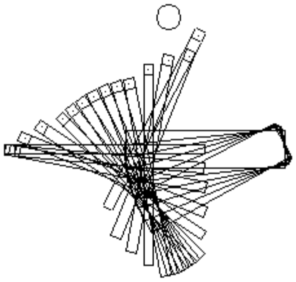
(c) After Differential A*, Robot attempts to move, but hits again. Right: Influenced CS Area is cleared.

Figure 72: Learning about an Obstacle. (Example of a "Type I *Change")*



(d) After Differential A*, Robot attempts to move, but hits again.





(e) After Differential A*, robot is clear of obstacle to goal. Right: Final path piece.

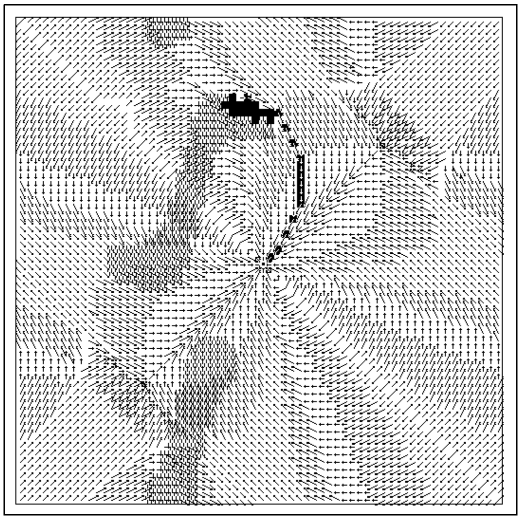


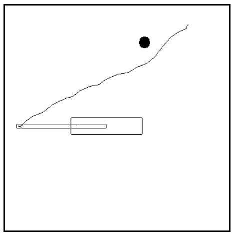
Figure 72 (Continued)

6.4.2 Inserted, Moved and Deleted Obstacles (“Type I and II Changes”)

This section shows an example of the Differential A* technique of Section 6.3 applied to “Type I Changes” for the two-link robot (of Figure 15) in an environment where: a round black obstacle in the environment suddenly appears. The stepwise incorporation into the task space is shown in Figure 71. The round obstacles are transformed into two blobs, (one is an S shape) in configuration space. Once the ‘NOW’ button on the user interface is selected, the changes in goal and obstacle states is determined. Following Figure 68, ‘Operations for Inserted Obstacles and Deleted Goals’ are performed next. The white areas of Figure 70(c) are the relatively small regions affected by the obstacle’s sudden presence and were computed using the χ function. Operations for ‘deleted obstacles and inserted goals’ is then performed (but there are no actions for this example), and then A* is carried out to fill the region shown inFigure 70(d). This configuration space is used as the initial configuration space for the next change shown in Figure 73, when this obstacle moves.

Figure 73 (top) shows the obstacle moved closer to the pivot location of the robot arm reflecting a “Type II change”. Because it is positioned to block some shoulder positions, a complete range of shoulder motions (in configuration space) is transformed into illegal regions and the result of the χ function is again shown as the white area in the configuration space. When the A* computation recomputes this area, the solution shown at the bottom is generated. The task space path related to the solution is somewhat surprising at first. It contains a straight line to the center, where the shoulder can rotate freely with the end effector over the shoulder, for a cost ($\delta > 0$).

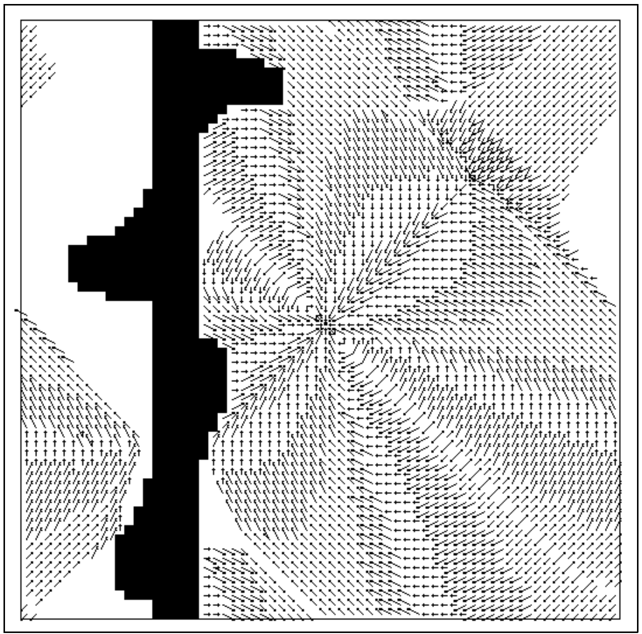
The obstacle of Figure 73 (bottom) is then completely removed (i.e. disappears from the reach of the robot) in Figure 74 (top). By removing the obstacle, no computation is required for the ‘Operations for Inserted Obstacles and Deleted Goals’, therefore there are no nodes for the χ function. The prior obstacle states require actions in ‘Operations for Deleted Obstacles and Inserted Goals’, which then find the perimeter of the removed states for OPEN. The deleted obstacle states are reset to ϕ and appear as the white region. When A* propagates from the OPEN nodes, the space is filled as shown in Figure 74 (bottom), which is the same status as Figure 70(a).

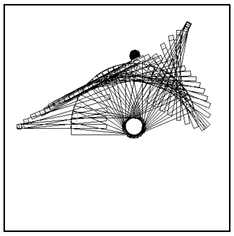


Above: Prior Path from Figure 71(b).

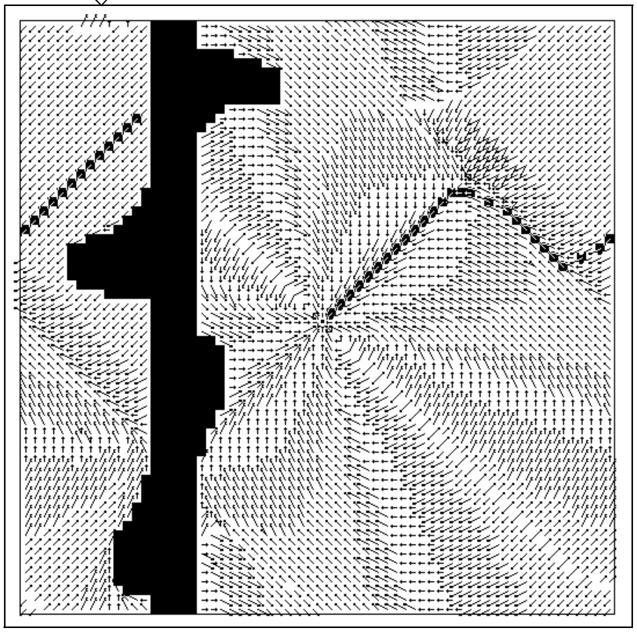
Below: Arm Sweep showing obs. moved into prior path.

Right: CS after clear-influence step for new obstacle moved from location in Figure 70(d)





Below: CS after Differential A*.



Right: Solution Space determined by Differential A*.

Below: Task Space path corresponding to new CS path.

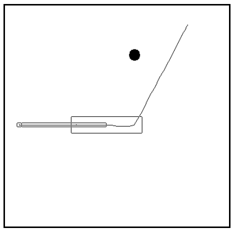
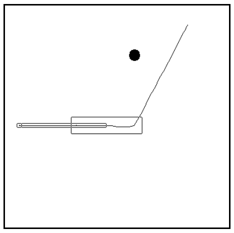
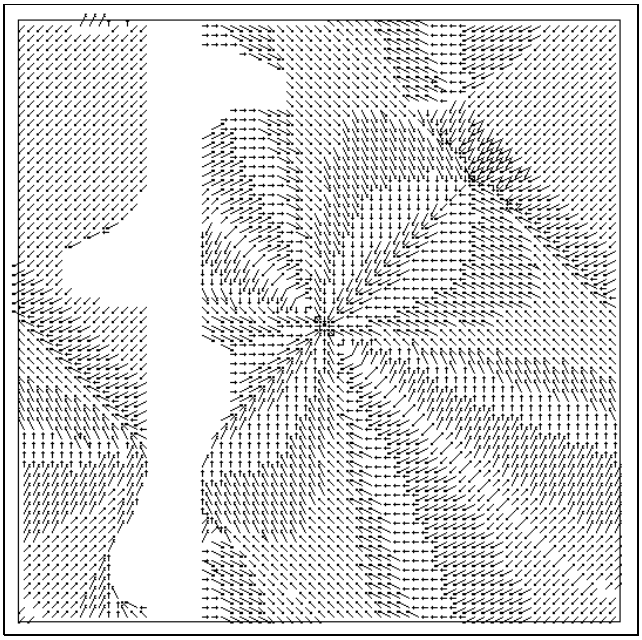


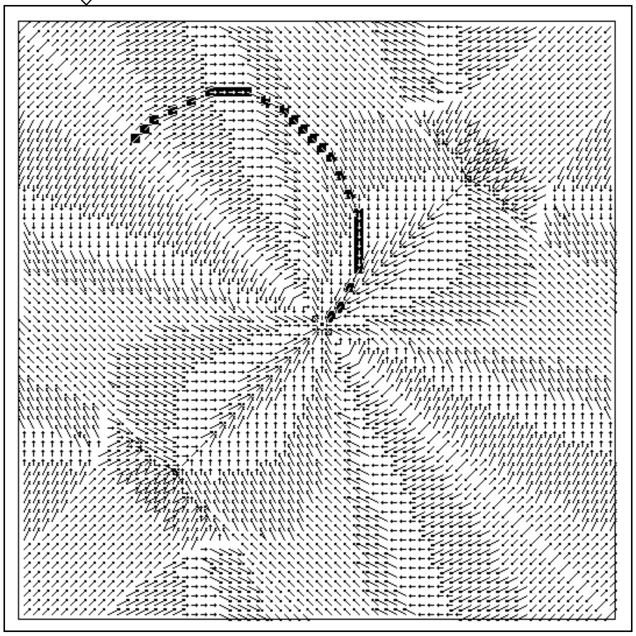
Figure 73: Obstacle Moves.
(Example of a "Type II Change")





Above: Path with obstacle. When obstacle is removed from reach, the obstacle states disappear from Config. Space (right).

Below: CS after Differential A*.



Below: Task Space path corresponding to new CS path (right).

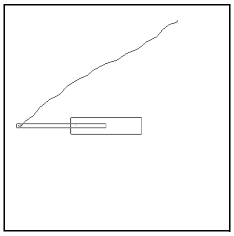
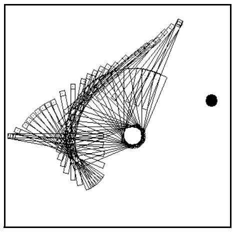


Figure 74: Obstacle Moves Out of Reach (Disappears).
(Example of a “Type II *Change”)*

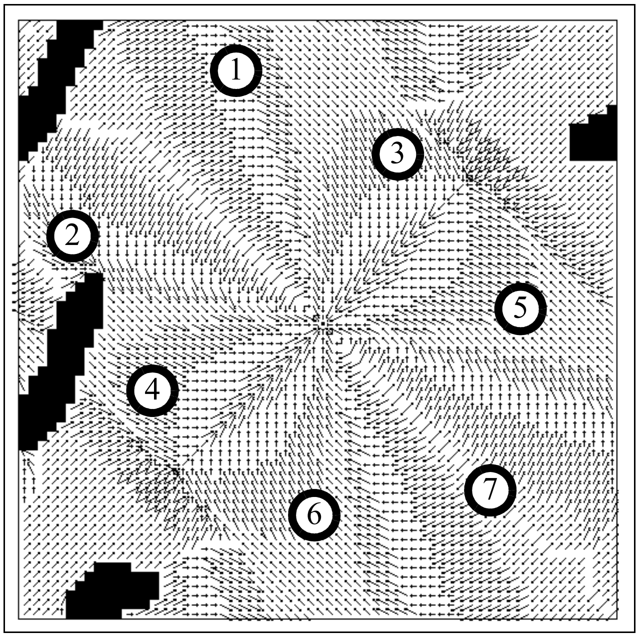
6.4.3 Inserted and Deleted Goals ("Type IV Changes")

Figure 75 shows seven added goal states. Because the goals are specified in task space, there are generally right and left handed configurations that can be represented symmetrically in configuration space. The addition of new goals is a one step process. Each of the new goals produces reduced costs, managed by Case 10, and becomes a starting node for the A* search. Since none of the transitions is deleted or increases in cost, there is no affected area. Removal of one of these goals is managed by Case 9 and is shown in Figure 76. This situation will increase the cost of transitions, producing the affected area, in white. It is filled in by the subsequent A* computation.



Above: Task Space with obstacle and 1 goal.

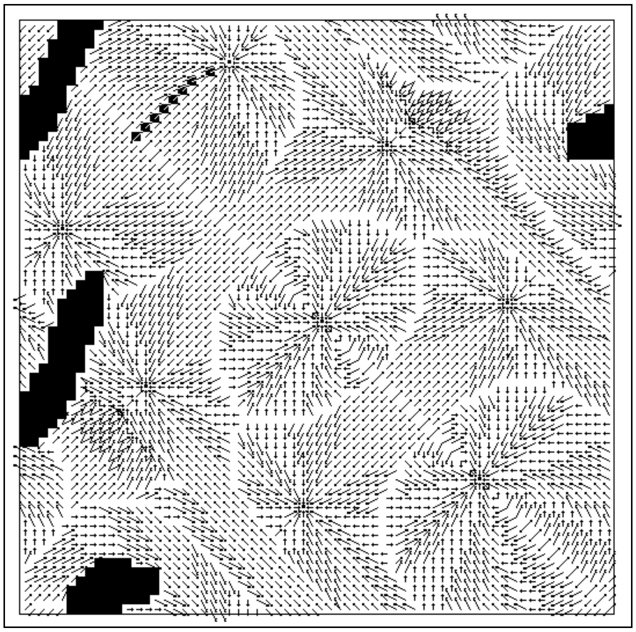
Right: CS with 7 new goals before Differential A*. Six of the goals arise from three right and left handed (equivalent) goals in task space.



Below: CS after Differential A*.

Right: Computed Configuration Space.

Below: new path in task space to nearer goal.



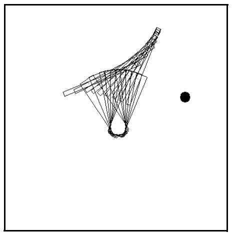
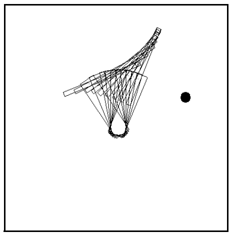
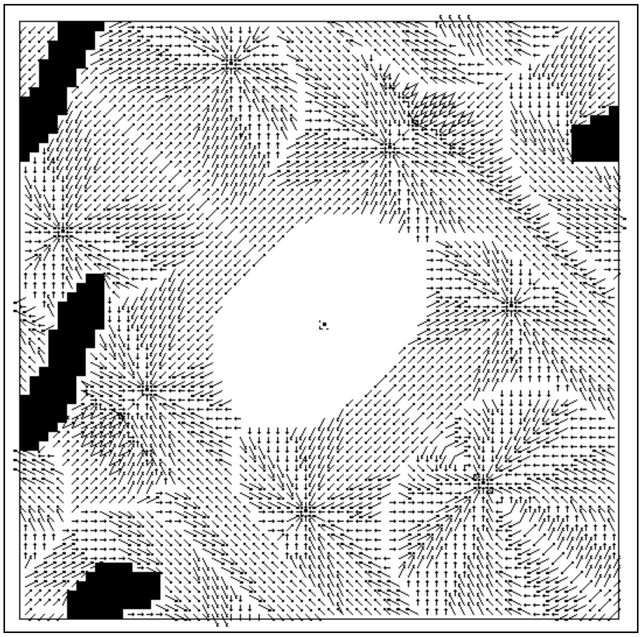


Figure 75: Seven New Goal States in C.S.
(From 3 in Task Space) Plus the Center Goal. (Example of a "Type IV *Change").*

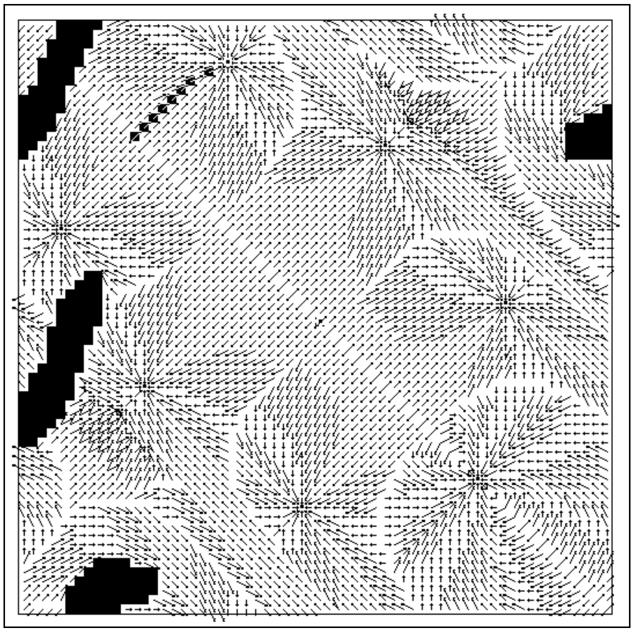


Above: Task Space with path from Figure 75. Right: CS with center goal removed.



Below: CS after Differential A*.

Right: Updated CS.
Below: original path in task space; goal removal caused changes distant from prior path.



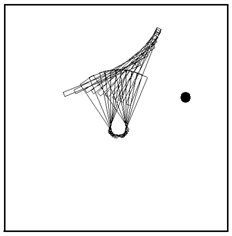


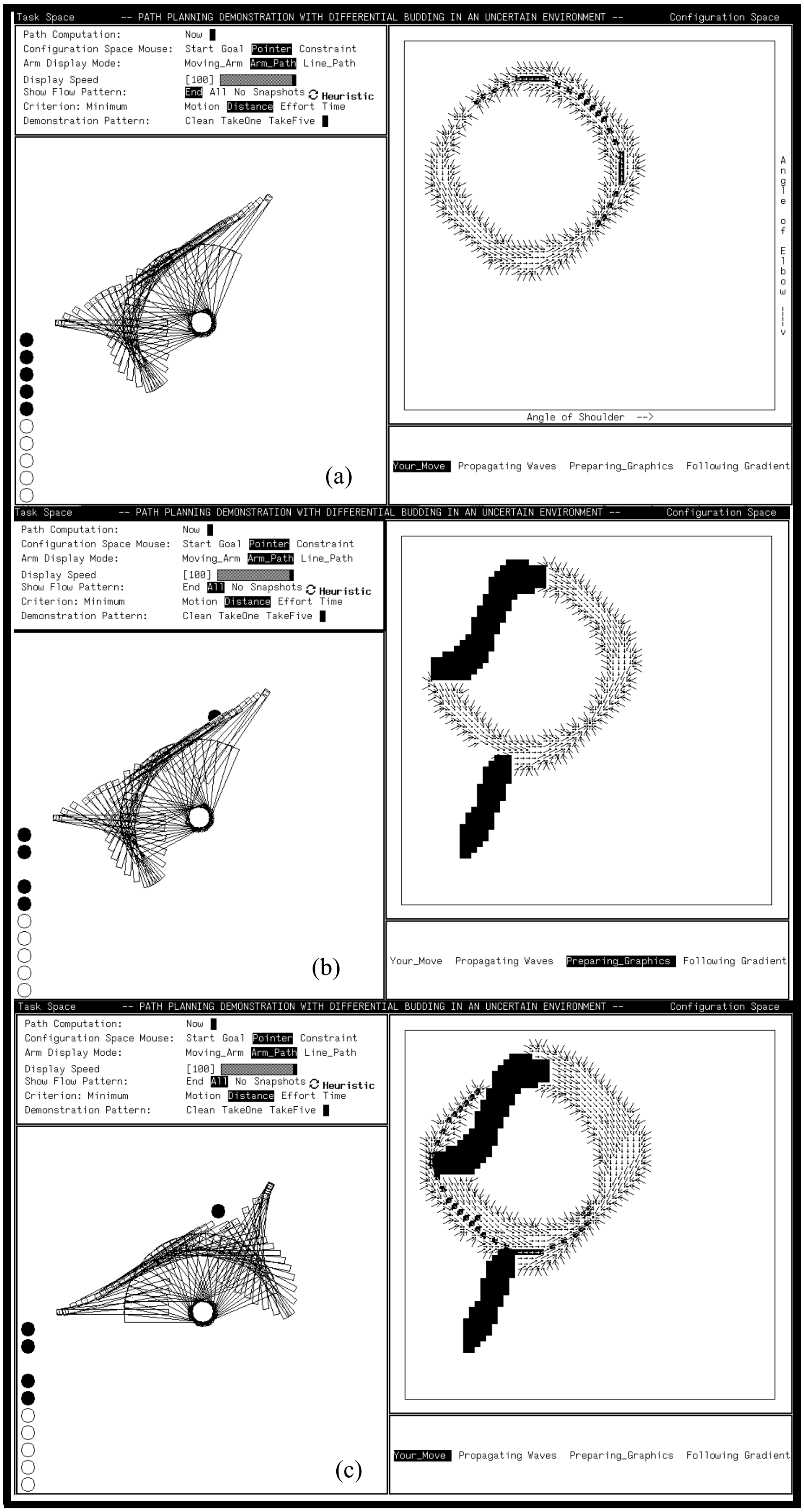
Figure 76: Removed Center Goal. Old path Unaffected.
(Example of a "Type IV *Change")*

6.5 Differential A* with an Admissible Heuristic

Just as in the previous examples, changes to a configuration space that has been calculated with a heuristic are not any different than one created with h=0, unless the heuristic is dependent on a starting state that has changed (described in Section 5.6.5). Therefore, changes to the obstacles or goals will follow the same Differential A* algorithm. Figure 77(a) shows the 'straightest path' found using the same transition costs and heuristic introduced in Figure 29 and Figure 30 . When the obstacle is inserted into the task space (on the left side) of Figure 77(b), the same algorithm is used to clear all pointers behind the obstacle, and find the perimeter. The result is shown in the configuration space (on the right side) of Figure 77(b).

In this case the perimeter of the cleared area is the set of states at the edges of the two obstacle. These are added to the set of OPEN nodes which includes those from all the edges of the computed region. As the A* search proceeds, the area searched widens in response to seeking a path around the obstacle blocking the original path. As seen in Figure 77(c), the path is found around one side before the other. The swept path is shown on the left as evidence that the arm would not hit the obstacle.

Just as in the h=0 implementation, if there is a single goal that is moved (or removed), then all pointers will be removed, and it is better to fully recompute with A*.



```
Path Computation:            Now
Configuration Space Mouse:   Start Goal Pointer Constraint
Arm Display Mode:            Moving_Arm Arm_Path Line_Path
Display Speed                [100]
Show Flow Pattern:           End All No Snapshots
Criterion: Minimum           Motion Distance Effort Time
Demonstration Pattern:       Clean TakeOne TakeFive
```

Figure 77: Admissible Heuristic Guides Search and Adapts to a New Obstacle.

(Example of a "Type I *Change")*

6.6 Quantified Results: A* vs. Differential A*

From Section 5.6 it is clear that the Differential A* method is more efficient in groups B, C, and D. Groups A and E have similar overhead required with $\chi()$, the clear influence function, making these the situations where A* may be preferable over Differential A*. The next section evaluates the robotic algorithm using an optimality criterion that minimizes Euclidean distance in configuration space and a non-informed, but admissible heuristic, h=0 to define the applicability of the two methods.

We will use group A, specifically inserted obstacles which increase the total cost, to represent the class of problems in groups A and E. In the robotics domain, the increase in the node delay cost is achieved by adding an obstacle, so that the node delay cost becomes essentially infinity. This is similar to deleting the node as in group E. Similarly, the deletion of a goal increases the current node cost to a value above zero. These mechanisms are used interchangeably to determine when A* and Differential A* should be used.

The time to clear nodes with the function χ (including the computation of the perimeter) is the largest cost directly attributable to the Differential A* method. To complete the calculation however, the perimeter must be added to the sifted heap of OPEN nodes, and then A* must be performed. Section 5.7 identifies the time required to recompute the space with Differential A* as:

$$\text{Time}_{\delta A*} = \text{Time}(\chi) + \text{Time}(\text{Perim}) + \text{Time}(\text{A*}(\text{OPEN}))$$

When the total time for these functions is less than full recomputation for A*, Differential A* is preferable.

By measuring the time to perform χ on a given set of nodes, including the time to add the elements to the perimeter, and the time to perform A* on the perimeter, a comparison can be made to the time required to fully recompute the space using A*. The time was measured as a function of the number of cleared nodes, based on an initially obstacle-free configuration space, to which new obstacles were added. The obstacles never covered the goal, nor did they cut off sections of the space. The obstacles varied in size, number and location throughout the configuration space. Figure 78 shows the amount of time required for each function.

Multiple obstacles in combination can cause more than 2300 states to be cleared. The number of cleared states is typically low when the size of the new obstacle states are small or far from the goal. Depending upon the overlap between obstacles and their positions relative to one another and the goal, different numbers of cleared nodes are produced.

As seen from the plotted Differential A* timing experiments of Figure 78, the time for computing the function χ, adding to the perimeter and A*, vary depending upon the number of cleared nodes. The variety of obstacle number, size and distance (cost measure distance relative to the goal) varies this data somewhat, but it is stable over a large range of the number of cleared values. Generally, A* time to recompute the cleared nodes from the OPEN nodes is about twice the clearing time of $\chi()$. The time to add nodes to the perimeter is about 5% of the clearing time. If the assumed objective is to regenerate the complete navigation map, the cost of updating the differences in the space must be compared to the recomputation of the entire space. For this configuration space, with Euclidean cost measure and h=0, complete recomputation on a Sparc IPX requires 250 ms. Therefore from Figure 78, when the number of cleared nodes is under about 2300, Differential A* will complete before A*.

Accordingly, for this cost measure and heuristic, if the number of cleared nodes is less than 55% of all the nodes that are likely to be computed, then it can be determined that Differential A* is preferable to A*. For more compute intensive cost measures, or higher numbers of neighbors this percentage is likely to be somewhat higher, because clearing a node will be even more efficient than recomputing. In higher dimensional configuration spaces, for a similar cost measure, heuristic, and neighborhood, we expect that when less than 55% of the nodes are affected, Differential A* will also perform well. This is because the fundamental operations do not depend on the dimensionality of the space.

The configuration space of the experiment is 64x64=4096 states large. It can be shown that the number of cleared nodes for a single, round obstacle can never exceed 2300 states. This is because of the behavior of the transformed S-shaped obstacle, assuming the obstacle does not cover the goal (whereupon the complete space is cleared, but no A* occurs). Figure 78 shows that Differential A* is preferable to A* below this number of cleared nodes. So, with a single obstacle in the two dimensional case, Differential A* should always be used.

Consequently, the key is to understand any problem sufficiently well to be able to estimate this percentage. For insight into the two dimensional problem for general objects, it would be helpful to understand the relative impact between the location of the obstacle, the size, and the resulting cleared region. Figure 80 shows the results of an experiment that uses a single obstacle in configuration space, not generated from the robot example, but rather generated as a single rectangle situated diagonally at +45°, having a width that varies, and depth 3 units thick, placed symmetrically about the -135° axis. The size never exceeded the diagonal space, thus it never ‘wrapped around’. These obstacles represent a class of obstacles in this periodic configuration space that are relatively perpendicular to the nominal pointers, positioned so that the cleared area is maximized (because the entire corner area is cleared).

A diagram of the different obstacle distances and widths are shown in Figure 79. By varying the width along the diagonal axis, the chart can be used to estimate the clearing affect for similar obstacles at similar distances. For ‘thicker’ obstacles, we estimate that the additional thickness of the obstacle would have otherwise been in the region of the cleared nodes. Therefore, the number of cleared nodes is decreased by the additional number of nodes in the obstacle. Because the space is limited either by the periodicity (where the pointers begin to point in the opposite direction) or for other examples where the space has specific bounds (joint limits, for example) the farther the obstacle is from the goal, the fewer nodes remain to be cleared by the function χ. For non-diagonal (i.e. vertically) located obstacles, the chart in Figure 80 can be used as an upper bound. By taking the sum of the obstacle affects, an estimate can be made for multiple obstacles in this problem.

We expect a similar trend for other problems. For a method to estimate the affect of obstacles in a particular (possibly higher dimension) configuration space and cost measure, experiments analogous to Figure 78 and Figure 80 should be performed.

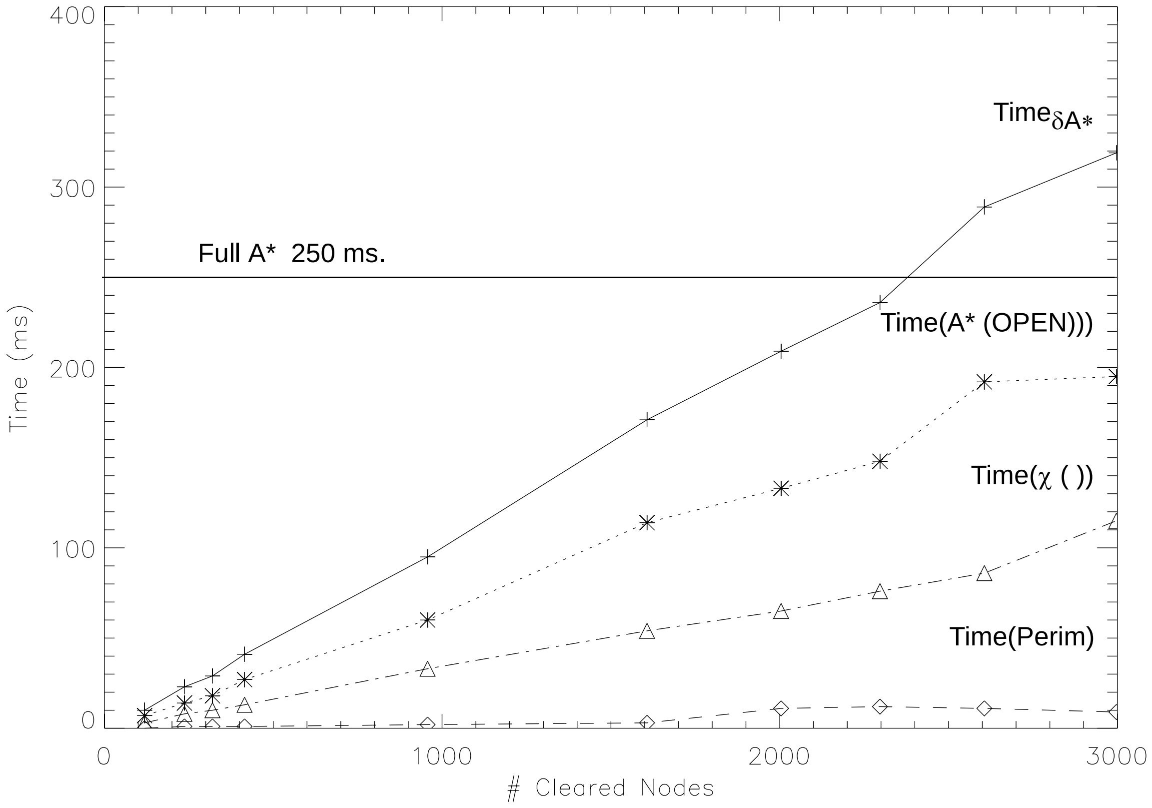


Figure 78: Differential A vs. A* Timing .*

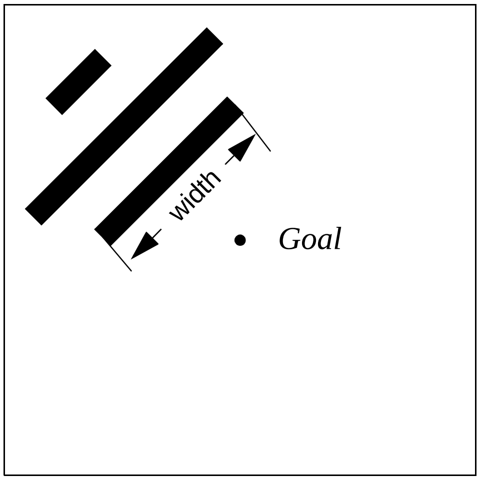


Figure 79: Experimental Setup for Test of Obstacle Affect.

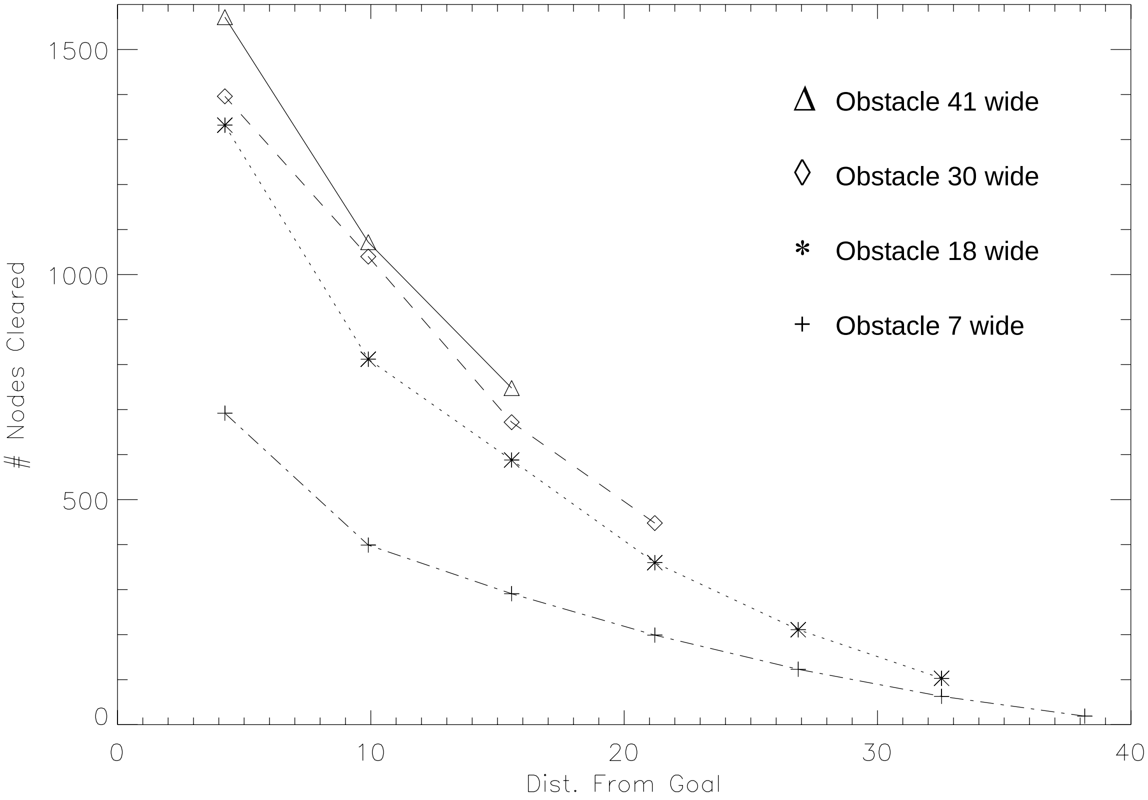


Figure 80: Obstacle Effects.

6.7 Memory Requirements

Since the clear-influence state stores the surface of the cleared region, the additional memory requirement for Differential A* beyond A*, is that amount necessary to store a surface in one less dimension than the problem. Usually the surface is quite small, particularly if heuristics are used. The remainder of the procedure uses the same memory space.

6.8 Alternative Approaches

Other techniques are also available to update changes in the configuration space. One method in particular [5], identifies regions that are swept out by the occurrence of obstacles and identifies the shadow areas from the vertices of the obstacle. It relies on another method that computes ‘cost waves’ using the Constrained Distance Transform (CDT) [4,22,23]. It also inherits some of the limitations of the CDT, namely, the inability to manage non-Euclidean Distance cost measures or irregular ‘neighborhoods’.

6.9 Other Practical Aspects of Differential A^* in Path Planning.

In the case of path planning, it is essential for a general planning algorithm to be able to handle multi-dimensional (and hence large) configuration spaces, since a robot having many degrees of freedom will require it. Often, the optimal path does not need to be completely recomputed when a change affects only a small part of the space [45,47].

In a practical robotics problem, a useful artifact of this process is that the robot may be able to carry out an action while the space is being recalculated. Although the robot is not guaranteed to follow an optimal solution (since the space is being improved), occasionally it is better to make a slightly sub-optimal move sooner, rather than to wait for the calculated optimum. This is possible with the Differential A* method if the starting state is not in the affected area (shown as white in Figure 70(c)).

6.10 Chapter Conclusions

Differential A* can adapt pre-existing stable solution-graph(s) to new changes and produce a new configuration space that is identical to one that would have been generated from complete initialization and A*. The essence of the algorithm is to clear and subsequently revisit the area influenced by the changes. The purpose of “clearing” some of the identified areas is to reset the node values to *uninitialized* so that subsequent A* computation can correct the area with a new, higher cost. This is necessary since A* is only used to initialize or lower costs during searching.

Fortunately, for many robotics problems, the changes in the configuration space are relatively small, and can be accommodated quickly without full recomputation. This allows the robot to continue moving if the change is not immediately affecting the current path, but also gives the robot a quicker response, even if the current path is affected. For the two dimensional case, when less than 55% of the nodes are cleared, Differential A* is preferable.

Finally, the Differential A* method can identify the effect of arbitrary obstacle shapes, with arbitrary cost measures that may be used in A*, in high dimensional spaces.

Chapter 7

Discussion

7.1 Contributions

The achievements of this work include a framework for solving the path planning problem for autonomous systems, including the concept of a neighborhood, and the introduction of Differential A*. Some preliminary work toward extending the framework toward the task level, called ‘rendezvous planning’, is also introduced as a subject for further research at the end of this chapter and in Appendix C.

The modular framework for A* planning in discrete configuration spaces was described for planning motions for autonomous systems. A variety of machines were fairly easily modeled, producing apparent intelligence through autonomous behaviour. The neighborhood is shown to be a powerful new concept in the framework, since it characterizes the fundamental motion of a specific machine. The concept is the key to embodying:

- the simple motions for a floor based industrial robot or emergency guidance system,
- the joint coordination of a robot arm,
- the kinematically complex coordination of the steering and drive wheels of a car,
- the simplicity of a treaded vehicle,
- and the relative motion of highway speed cars.

Once the neighborhood is known, the cost of a specific motion can be assigned based on the cost criterion selected. By changing the neighborhood or cost criterion, a machine will behave differently. These modular changes provide a clear structure giving greater flexibility for defining the overall desired behaviour of a machine. The wide variety of machines and situations addressed with this framework is an indication of its power.

Differential A* was introduced as a new graph method that quickly adapts existing A* plans to changes. The situations when Differential A* is likely to be more efficient than full recomputation with A* has been analyzed. Although not always more efficient, it can be useful in dynamic environments that incorporate newly sensed changes. The mechanism is especially powerful when small incremental changes occur, because not only is the recomputation quick, but under some (space filling) conditions, the mechanism may resume optimal motion while the calculation is still underway. This provides a quick reaction, improving the perceived intelligent behaviour.

This framework and the techniques described in this thesis can be applied in practice to control many types of autonomous machines. The key obstacles to commercializing autonomous applications however, are the cost of sensing apparatus and legal liability. In some applications however, such as the intelligent emergency exits with standard sensors, or more generic (non-machine) Differential A* graph planning problems, immediate technical improvements are possible. The legal climate remains a business challenge in some countries, most acutely affecting self-driving vehicles. Once the demand for autonomous vehicles to help otherwise non-mobile people [16] exceeds the legal barriers, an opportunity exists. We speculate however, various intelligent features will be gradually integrated into vehicles, decreasing the minimum ability required for save transportation.

In this chapter, we discuss some insights into the advantages and disadvantages of the methods and applications presented in the thesis. Some insights reflect what was learned, while others suggest possible directions for future research.

7.2 Insights into Planning Algorithms

7.2.1 A* for Path Planning

The modular framework that uses A* to actual applications in intelligent autonomous systems has been shown to be a powerful and useful tool to create optimal motion plans. Part of the power comes from the ability to simply substitute a new neighborhood or cost measure to produce an entirely different, but optimal, behaviour of a system. With this, a wide variety of mechanisms have been modeled or controlled using this framework.

A useful property of the planner is that knowledge that a path does not exist is sometimes as important as finding the path. A mechanism using this planner can then originate a request to the calling task planner at the earliest possible moment. Other mechanisms may not even realize they are trapped.

The planner performs best when the configuration space is most cluttered. The more obstacles that forbid computation, the faster the plan is found. So, problems that appear difficult are actually easier to solve.

We find that it is useful to compute the complete configuration space for situations where the environment is likely to maintain the same goal (set), obstacles and cost measure, while the start varies. This is commonly the case where the computed path is used as a sequence of setpoints to a controller, since it may fail to follow the prescribed path. Rather than recompute upon each failing, we can compute the space once, and then have a more robust option for recovery by following the omni-present directions.

One difficulty with the planner, is that it assumes that all forbidden regions are known in advance. This may not be the case, and the plan must be recomputed whenever they are ‘discovered’.

7.2.2 Differential A*

The Differential A* algorithm can provide a quick way to update the graphs. This is useful if small changes occur, which is seen most poignantly with a dynamically sensed obstacle. The shape of the obstacle can be ‘learned’ at run-time with minimal affect on overall performance.

While this is a generally positive attribute, a system should check periodically that assumed obstacles still remain.

Differential A* is more efficient than A* for the example problems where obstacles affect less than 55% of the configuration space nodes. In the 2-D robot arm example, this could only happen when there was more than one obstacle since a single node cannot cover more than 50% of the space at a time. The number of nodes affected is a function of their size and inversely distance (in terms of the cost measure) from the goal, and must be determined for each environment. If there are multiple goals distributed evenly throughout the space, it is likely that the number of affected nodes is small, and that Differential A* will work well.

Differential A* can manage changes, but while the adaptation is fast, it still relies on external sensing systems. Therefore, the response time is larger than the longer of the sensing cycle time and planning time. Sensor technology continues to improve in speed and accuracy, but is still far from being a commodity.

7.3 Insights into Planning for Autonomous Systems

7.3.1 Robot Application

The two dimensional robot application is a good carrier problem to identify and illustrate the breadth of A* and Differential A* applications possible. Our implementation provides the ability to see the immediate results of the different optimality criteria, both in the configuration space and verification in task space. It provides a mechanism to examine and change the environment by adding goals and obstacles in a simple way. The new environments sometimes produce very surprising (but correct) paths, each time giving greater insight into the effect of modules in the fundamental method. Without the user interface to observe the changes in configuration space, it would be difficult to have identified the opportunity for Differential A*.

The size of some graphs, caused by large numbers of degrees of freedom in robotics, affect the likely time performance of the algorithm. One of the practical concerns for the method, is the time and space required for computation. Particularly for robotics, the number of states grows according to the number of dimensions (i.e. degrees of freedom (dof)) of configuration space. For large (6-8 dof) problems careful choices about discretization and possible decoupling of a large problem are some options to provide solutions on today's computers. Fortunately, the planner can perform best when the configuration space is most cluttered. The more obstacles forbid computation, the faster the plan is found. Because the graphs can grow rapidly with finely discretized configuration spaces, at current processing speeds, large problems are intractable with the current method(s). With equal ease, computations to minimize time, distance, effort or time can be accommodated. Each change produces a change in optimal behaviour, which sometimes appeared surprising at first.

The Differential A* method can be applied to manage arbitrary obstacle shapes and their affect regardless of their original cost measures or the dimension of the configuration space.

Finally, the robot could be controlled [29] in practice by providing the components for the planning framework. By incorporating the environment and optimization criteria a path was generated. This discretized path was then communicated to the reflex controller to servo to the given setpoints.

7.3.2 Vehicle Application

The vehicle maneuvering application provided a challenge to the framework, since the neighborhood and configuration space are three dimensional. Because the data is not easy to view, as in the robot example, it required careful attention to the specification of each framework component.

Because the neighborhood is the encapsulation of the basic vehicle motions, it can be calculated by geometry. The obstacle transformation is also straightforward geometry, when viewed one angle slice at a time rather than as a set of interconnecting helices.

We were able to generate a solution in a three dimensional configuration space without the benefit of any (obviously desirable) graphical insight, because the framework was succinct. The three dimensional neighborhood and obstacle transform could be clearly incorporated into the framework to produce remarkably complicated maneuvers with relative ease.

The proof that plans can be created and carried out in the vehicle maneuvering testbed for a mechanism as complicated as a car, shows the power of the overall method. Just as the task space must be converted to configuration space for optimal path planning, the results of the plan must be transformed to the primitives for controlling the car. It is not always as simple as a feedback loop using the configuration states directly.

Maneuvers such as parallel parking, reversal in a constrained location, and simple forward-only motion can be realized with the same framework. The parallel parking and reversal are achieved by changing the starting point. By computing the space once, these and other interesting maneuvers can be seen. The forward-only motion is achieved by a change in the neighborhood. This simple change produces a radical behaviour change in the vehicle, and severely limits the maneuvers possible.

Differential A* was not implemented for the vehicle maneuvering example because almost all nodes would be affected at each calculation. The two parked cars (obstacles) are on opposite sides of the goal, in addition to the curb which covers a third side. Any change would affect the majority of the space, therefore Differential A* was not implemented for this problem.

In the implementation, the simple arrangement of the configuration space data structure (i.e. the ordering of the indices) can cause a change in transformation time by several orders of magnitude. This is because the order in which the transform is calculated results in a ‘plane by plane’ access. Therefore, these indices change most rapidly. By placing them in locations where they are nearer each other (i.e. [angle][x][y] rather than [x][y][angle]), can cause smaller jumps, which in the Sun 3/160 is significantly more efficient.

Finally, we challenge the concern that the vehicle maneuver planning would be prohibitively slow for any highway applications. By viewing the vehicle motion in a relative rather than absolute fashion, we observe that the kinematic response of a car is much simpler at highways speeds. Specifically the car can move sideways relative to other cars. This simplification allows the use of small configuration spaces and neighborhoods that are fast to compute.

7.4 Directions for Research

There are four immediate areas brought up in this thesis that require further research. Specifically they are in obstacle transformation, discretization, and statistical uncertainty. The framework for the planner has been shown to be useful as it incorporates some planning attributes of the control level. Extending the framework toward the task planning level has been begun with a method for rendezvous planning, but has not been rigorously analyzed or broadly applied.

- In the robotic domain, a general transformation for obstacles from the task space to the configuration space is an unsolved problem. For example, even in a three dimensional space, there is no generally applicable algorithm to transform arbitrarily shaped obstacles for an arbitrarily shaped and jointed robot. Therefore the problems that are chosen, are usually carefully defined with narrowly defined robot or obstacle shapes. One possibility for machines with local obstacle detection systems (i.e. robot skin designed by Cheung and Lumelsky [8]) is for the robot to progress systematically through all configurations, testing for self intersection and external obstacle collisions. This gives a baseline for a the robot in its normal environment. Other obstacles can then be modifications of the baseline and can be incorporated into a given plan using Differential A*. Unfortunately ‘robot skin’ is not yet a commodity, but is currently a research topic in labs. Still, other mechanisms to ‘exercise’ the machine configurations are also possible, for instance in CAD simulations.

- It would be desirable to have a systematic way to characterize and balance the time, space, functionality and aesthetics of discretization, particularly as it is impacted by obstacles. This is currently an art, guided by empirical feedback. That is, the resolution is adapted until the computation provides ‘acceptable’ levels of solution. This is determined more by basic functionality and aesthetics than by analytic insight. For example, coarse discretization may provide a solution, but the robot does not seem to move near enough to an obstacle to seem plausible as an ‘optimal solution’.

- Again in the robotic domain, but also applicable to others is a challenge to incorporate statistical uncertainty in the assumptions of the graph or configuration space. For example, if the robot is to travel through a given building, but there is one door that is locked 10% of the time, and open the rest of the time. Is it preferable to refuse a path that might lead through this door? How can this risk be incorporated into the plan? What additional parameters must be provided to define the acceptable level of *risk*?

- Finally, the planner can be extended to address some questions that ordinarily exist at the task planning level. Some introductory work is provided in Appendix C specifically in an area we call *rendezvous planning* (initially described in [53,54]). This is a method to describe and algebraically manipulate the results of several planned paths. The objective is to combine various constraints, defined as a boolean expression, on the problem that cannot be ordinarily represented in a single planned result. A simplified example with two truck 'agents' shows the mutually acceptable locations when one agent is minimizing fuel and the other is minimizing time. The trucks begin at different locations and have some subset of locations that fit within ranges defined by each. Naturally, the coordination of activities can be computed for multiple agents with many more constraints (e.g. the first truck must minimize fuel AND arrive before the other truck). The brief introduction in Appendix C gives a flavor of the research direction.

Summary

This thesis presents a powerful, modular framework that uses the A* algorithm to plan motions for an autonomous system. We also introduce Differential A*, a new and complementary method that can be used in the framework for quickly addressing changes in the environment.

The same framework plans motions for mobile robots, emergency exits, robot arms, cars, bulldozers, and highway vehicles. Higher dimensional spaces are accommodated by the consistency of the method. Changing the *neighborhood* is the key to changing the behaviour and personality of a machine. Definitions of optimality can change the behaviour of a machine as well. Each of these can be easily incorporated into a module, and the solution computed quickly.

Adapting to changes in the machine's environment can be accomplished quickly by the use of Differential A*. Because the machine can adapt, it can operate in less structured environments and learn its surroundings. An analysis is also provided which indicates the environments when Differential A* or A* should be used to provide the fastest recalculation time. When sensing systems are fast and accurate, the machines may be able to respond faster and more precisely than even humans, the sensing and planning champions.

Finally, we identify an area for future research called *rendezvous planning* that extends the planning framework to the task planning level. The goal of rendezvous planning is to determine the options for coordinating the actions of multiple machines (called actors) while taking their respective constraints and optimization criteria into consideration.

Chapter 1 gives an introduction to the problem domain of optimal path planning as part of an autonomous system. It also describes some related work and provides an overview of the rest of the thesis.

In **Chapter 2**, the A* algorithm is reviewed, which efficiently computes the globally optimal path planning solution. In the A* algorithm, the search is guided by admissible heuristics which direct the search in the most likely direction of the solution, and reduce the amount of searching required. The notion of an *uninitialized* node and the specific use of a *sifted heap* structure for OPEN node management is introduced to improve the performance of the A* implementation. The worst case time of A* is O(N log N); a more detailed time and error analysis is given showing the dependence on the number of OPEN nodes.

In **Chapter 3**, the A* method is applied to path planning within autonomous systems. In autonomous systems, a higher level task planner defines the overall goal for the planned motion, and reflex and servo level controllers carry out the planned motion safely. The objective is to determine the optimal, collision free path to the nearest goal. The information required for the planner includes a description of the set of possible configurations (poses) for the robot. This forms the *configuration space*. The *neighborhood* of possible motions permitted from each node to other (usually local) nodes is also required. Based on this neighborhood, a cost can be associated with each neighborhood transition that reflects the criterion (least time, distance, etc.). Next, the *forbidden regions*, typically obstacles, are set as prohibited states in the configuration space. These

states can be assigned infinite cost to prohibit a search from considering these states as viable candidates for any optimal path. The *goal* state is assigned by the task planner, and the *start* is provided by the current state of the system. From this state, the sequence of transitions are generated until the goal is reached. The sequence forms setpoints for the (reflex) control system.

Only optimal paths are generated and the search is guided by admissible heuristics. In the examples in chapter 3, each application can use the underlying optimality criterion as the optimistic estimate (i.e. admissible heuristic) to guide the search.

A two degree of freedom robot arm is controlled by the method. The configuration space contains the set of permissible arm locations determined by the joint angles. This robot is unusual in that it has two joints that are both fully revolute. The configuration space provides configurations that wrap around the range from 0 to 2π for both the shoulder and elbow. The basic control of the robot is encapsulated in the neighborhood of motions, which have assigned costs for each motion. The cost for making a motion in the neighborhood is defined by the optimality criterion and an admissible heuristic (which can be zero). Several criteria are illustrated, including minimum communication, minimum distance of the end-effector and minimum time. The two link robot application uses these different cost criteria and heuristics to test the time bounds formula. With this, we validate the formula and identify constants for the individual applications. Finally, we observe that the planner works fastest when the configuration space is most cluttered and filled with obstacle states. The more obstacles forbid a computation, the faster a path is found.

The way in which the configuration space (graph) is represented greatly affects the speed and accuracy of the computation. While intuitively we might believe that more fine discretization will provide a more accurate solution, this is actually not the case. The effect of a finer discretization is that obstacles can be represented more accurately, since a state is an obstacle if any part of the robot would cause a collision. The representation of the neighborhood also plays an important role. Increasing the variety of neighbors (rather than strictly the number of neighbors) in a neighborhood can dramatically increase the quality of the resulting plan. For example, doubling the unique neighbors (from 8 to 16) for the two-link robot neighborhood gives a 285% gain in accuracy. This is a true gain that requires much less computation overhead than an attempt to double the discretization.

One of the practical concerns for the method however, is the time and space required for computation. Particularly for robotics, the number of states grows exponentially with the number of dimensions (i.e. degrees of freedom (dof)) of configuration space. For large (6-8 dof) problems careful choices about discretization and possible decoupling of a large problem are some options to provide solutions on today's computers.

Finally, the method for defining the time bound (proposed in chapter 2) is applied to the robotic examples of chapter 3. We see that the overhead time is relatively low for cost measures that use look-up tables and a zero heuristic, but are significantly higher for computed heuristics. This higher time may actually cause the user to prefer non-heuristically computed solutions for small problems.

In **Chapter 4**, paths are planned and carried out for a kinematically more complicated autonomous system, a car. The neighborhood encapsulates the possible local maneuvers for the car. This is derived from the position of the driving and steering wheels. The optimal path is defined as the minimum driving distance with the possibility for moving in forward or reverse directions. By fairly straightforward geometry, the obstacles can be transformed for each angle pose of the car. The layering of each possible car angle provides the full obstacle transformation. The configuration is specified through the user interface, and with a zero heuristic, the maneuver from all starting states is computed. With this, quite complex maneuvers can be computed and carried out. For example, the car can be placed in the parking spot reversed, pull itself out, and reverse itself. As an alternative cost measure, forward-only motion can be used to plan maneuvers so that the car can behave like any other in traffic. The time for the computation is measured using the same mechanism specified in Chapter 2.

To test the planned paths, the method is applied to a stock, 1/10 scale radio control car. The vehicle is contained in a plexi-glass walled, rectangular testbed with two other vehicles used as obstacles. The dimensions of the testbed define the limits of the possible locations of the vehicle, and therefore the size of the configuration space. The computed configuration space provides all possible optimal paths to the goal. It is loaded to common memory where it is available for the control system. A vision system is integrated with the control system to carry out the paths. Several conversions are required by the controller to achieve smooth reliable maneuvers. These smooth the discretization error by determining longer setpoints where possible, by recomputing the controlled arc to match the world coordinates rather than the discretized (and ideal) planned coordinates, and by defining stopping criterion that allows for some discretization, control or mechanical error.

Other vehicles and environments are also presented in chapter 4. A simulation of a much larger driving environment was created to show that the solution scales properly. In this environment, many more complex maneuvers can be illustrated. To change the behaviour of the car to a treaded (bulldozer) vehicle, the neighborhood and basic costs are all that must be changed. The bulldozer has a simpler kinematic model than the car, since it can rotate about its axis. Finally, we challenge the concern that the planning would be prohibitively slow for any highway applications. By viewing the vehicle motion in a relative rather than absolute fashion, we observe that the kinematic response of a car is much simpler at highways speeds. Specifically the car <u>can</u> move sideways relative to other cars and is always parallel to the other cars. This simplification allows the use of small configuration spaces and neighborhoods that are fast to compute. With fast sensing equipment, it may be easier to control a car on the highway than to park it.

In **Chapter 5**, Differential A* is introduced as a general, fast method for updating solution spaces generated by A* when changes to the underlying connections occur. The added, removed, increased or decreased transitions or nodes are identified, and are input into the *difference engine*. This recomputes nodes and clears dependent nodes to define the OPEN nodes required to regenerate the changed portions of the graph. The changes often affect only a small fraction of the entire solution graph that is ordinarily calculated with A*. Each type of change and the required actions are defined. Some cases are identified where changes require little or no computation. These are where the obvious time savings occur over full recomputation with A*.

The Differential A* algorithm is analyzed for each of the major operations: recomputation, clearing influence and adding a node to the candidates, highlighting the primary overhead of Differential A* as the χ function. The extent to which applications are likely to need this clearing function provides a way to identify when it pays to use Differential A* rather than completely recompute with A*.

Many of the characteristics of A* are also inherited by Differential A*, namely: computation of optimal cost transitions and topology, ability to handle multiple dimensions, and timing that is also a function of the total number of expanded (OPEN) nodes. The primary advantage of Differential A* is rapid adaptation in environments that have changes affecting only part of the solution-space.

In **Chapter 6**, a trade-off between Differential A* and A* is investigated. The Differential A* algorithm is applied to path planning in robotics when unexpected changes occur. With this new process, the robot 1) continues movement because the previous solution is usually not affected, 2) encounters obstacles on the fly and incorporates this knowledge, and 3) makes the best possible maneuver based on all knowledge possessed. This makes the system ‘learn’ and make the best decisions based on the current model of the world.

The timing of the method is tested in a zero heuristic situation. In this case the portion of time spent on the three fundamental operations is compared to the time required to fully recompute. The situation indicates that when less than 55% of the nodes are cleared, Differential A* will be more efficient than A*.

Differential A* is applicable for situations that currently use conventional A*, and can often considerably increase the speed of the solution. Good candidate problems are those with transition cost changes and unchanging goal.

Chapter 7 discusses the insights gained from this work, and a gives suggestions for future research directions. This section refers to Appendix C where an introduction to a new method for rendezvous planning is given.

Samenvatting

Dit proefschrift betreft een krachtig modulair raamwerk voor het plannen van bewegingen voor een autonoom systeem. De methode wordt beschreven en geanalyseerd in termen van het algemene A* algoritme, en een nieuwe en complementaire methode voor het behandelen van veranderingen, differentieel A*, wordt geïntroduceerd.

Een en hetzelfde raamwerk kan bewegingen plannen voor mobiele robots, nooduitgangen, robotarmen, auto's, bulldozers en voertuigen op de snelweg. Hoger-dimensionale ruimten worden ingepast door de consistentie van de methode. De sleutel tot de verandering van het gedrag en de 'persoonlijkheid' van een machine is de verandering van de zgn. *nabuurschap* ('neighborhood').

Het machinegedrag kan ook worden beïnvloed door definities van wat optimaal is. Elk van deze veranderingen kan gemakkelijk worden opgenomen in een module, en de oplossing kan snel worden berekend.

Het aanpassen aan veranderingen in de werkomgeving van de machine kan snel worden verricht met behulp van *differentieel A**. Omdat de machine zich kan aanpassen, kan het in minder gestructureerde werkomgevingen werken, en die werkomgevingen ook leren kennen. Een analyse wordt gegeven die aangeeft voor welke werkomgevingen A* dan wel differentieel A* zou moeten worden toegepast om de bewegingen zo snel mogelijk te herberekenen. Als de waarnemingssystemen snel en nauwkeurig zijn, dan zullen de machines sneller en precieser kunnen reageren dan mensen - nu nog steeds de kampioenen in waarnemen en plannen.

Tenslotte bespreken we een gebied voor toekomstig onderzoekmogelijkheid voor een toepassing die het raamwerk voor bewegingsplanning uit zou kunnen breiden naar het niveau van taak-planning: *rendez-vous plannen*. Het doel van rendez-vous planning is om de mogelijkheden te bepalen voor het coördineren van de acties van meerdere machines te coordineren, rekening houdend met hun respectivelijke beperkingen en
optimaliseringscriteria.

Hoofdstuk 1 introduceert het probleem domein van de planning van optimale paden als deel van het funktioneren van een autonoom systeem. Het beschrijft ook gerelateerd werk en geeft een overzicht van dit proefschrift.

Hoofdstuk 2 bespreekt het bekende A* algoritme, dat efficiënt de globaal optimale oplossing van het padplan probleem berekent. In het A* algoritme wordt het zoeken geleid door toegelaten heuristieken ('admissible heuristics') die het zoeken concentreren in de meest waarschijnlijke richting van de oplossing, en daardoor de zoektijd reduceren. Het begrip van een *ongeïnitialiseerde node* en het specifieke gebruik van een gesorteerde 'heap' structuur voor het behandelen van de OPEN nodes wordt geïntroduceerd om de prestatie van de A* implementatie te verbeteren. Een analyse wordt gegeven die aantoont voor welke werkomgevingen gewoon A*, en voor welke differentieel A* zou moeten worden toegepast om de bewegingen zo snel mogelijk te herberekenen.

In **hoofdstuk 3** wordt de A* methode toegepast op pad planning binnen autonome systemen. In autonome systemen definieert een planner van een hoger niveau het totale doel van de te plannen beweging, en reflex- en servo-regelaars voeren de geplande beweging veilig uit. Het doel is om de optimale, botsingsvrije beweging naar het dichtsbijzijnde doel te bepalen. De informatie die nodig is voor de planner omvat een beschrijving van de verzameling van toegestane configuraties voor de robot. Deze vormen samen de *configuratieruimte*. De *nabuurschap* ('neighborhood') van mogelijk toegestane bewegingen tussen twee nodes is ook nodig. Met behulp van deze nabuurschap kunnen aan iedere overgang kosten worden toegekend, die het optimaliteitscriterium representeren (kortste tijd, afstand, etc.). Vervolgens worden de *verboden gebieden* (typisch overeenkomend met de obstakels) aangegeven in de configuratieruimte. Aan deze toestanden kunnen oneindige kosten worden toegekend om te voorkomen dat de zoekprocedure deze toestanden als mogelijke kandidaten voor een optimaal pad beschouwt. *De doeltoestand* wordt bepaald door de taakplanner, en de *begintoestand* wordt verschaft door de huidige toestand van het systeem. Vanuit deze toestand wordt een serie van overgangen bepaald tot het doel is bereikt. Deze reeks vormt geleide-punten ('setpoints') voor het (reflexieve) controle systeem.

Slechts optimale paden worden gegenereerd en het zoeken wordt geleid door toegestane heuristieken. In de toepassingsvoorbeelden van hoofdstuk 3 kan iedere applicatie het onderliggende optimalisatie criteria gebruiken als de optimistische schatting (d.w.z. de toegestane heuristiek) voor het zoeken.

Een robot-arm met twee graden van vrijheid wordt door de methode bestuurd. De configuratieruimte bestaat uit de verzameling van toegestane houdingen van de arm, gekarakteriseerd door de gewrichtshoeken. De besproken robot is lichtelijk ongebruikelijk omdat beide gewrichten volledig draaibaar zijn. De configuratieruimte geeft dan configuraties voor zowel schouder als elleboog in het gebied van 0 tot 2 π, en periodiek overlappend. De besturing van de robot is bevat in de nabuurschap van bewegingen, die voor iedere beweging met kosten gepaard gaan. De kosten voor het maken van een beweging in de nabuurschap wordt bepaald door het optimaliteitscriterium en een toegstane heuristiek (mogelijkerwijs gelijk aan nul). Verscheidene criteria worden geïllustreerd, inclusief minimale communicatie, minimale afstand van de eindeffector, en minimale tijdsduur. De toepassing op de robot met twee gewrichten benut deze uiteenlopende criteria en heuristieken om de formule voor tijdsduur te testen. Hiermee valideren we de formule en identificeren we constanten voor de verschillende toepassingen. Tenslotte merken we op dat de planner het snelst werkt als de configuratieruimte het drukst bezet is met obstakeltoestanden. Hoe meer obstakels er zijn die een berekening in de weg staan, des te sneller wordt een pad gevonden.

De wijze van representatie van de configuratieruimte (een graaf) beïnvloedt de snelheid en precisie van de berekening in sterke mate. Intuïtief lijkt een fijnere discretisatie tot een nauwkeuriger oplossing te leiden, maar dit is niet altijd het geval. Het effect van een fijnere discretisatie is dat de obstakels nauwkeuriger kunnen worden gerepresenteerd, omdat een toestand een obstakel is als een willekeurig deel van de robot in één van de overeenkomstige poses een botsing zou veroorzaken. De representatie van de nabuurschap speelt ook een belangrijke rol. Toename van de variatie in het type buren (meer dan het aantal) in een nabuurschap kan de kwaliteit van het resulterende plan aanmerkeljk vergroten. Het verdubbelen van het aantal buren (van 8 tot 16) voor de robot met twee gewrichten leidt tot een verbetering in nauwkeurigheid van 285%. Dit is een echte verbetering die veel minder overhead in berekening kost dan een poging de discretisatie simpelweg te verdubbelen.

Eén van de praktische bezwaren van de methode is echter de tijd en ruimte nodig voor berekeningen. Met name voor toepassingen in de robotica groeit het aantal toestanden exponentieel met het aantal dimensies van de configuratieruimte (dus met het aantal vrijheidsgraden van de robot). Voor grote problemen (6 tot 8 vrijheidsgraden) moet men zorgvuldig gebruik maken van discretisatie en ontkoppeling om ze op hedendaagse computers te implementeren.

Tenslotte wordt de methode om het tijdsverbruik te bepalen uit hoofdstuk 2 toegepast op de robotica voorbeelden van hoofdstuk 3. Het blijkt dat de tijd relatief laag is voor kostmaten die opzoektabellen en een heuristiek van nul gebruiken, maar significant hoger voor berekende heuristieken. Deze langere tijd kan ertoe leiden dat de gebruiker de niet-heuristische berekening van oplossingen prefereert.

In **hoofdstuk 4** worden paden gepland en uitgevoerd voor een kinematisch meer gecompliceerd autonoom systeem: een auto. De nabuurschap bevat nu de mogelijke locale bewegingen van de auto. Deze worden afgeleid van de postie van de aandrijfwielen en de stuurwielen. Het optimale pad wordt gedefinieerd als the minimale rijafstand, met de mogelijkheid voor- of achteruit te bewegen. De obstakels worden rechtstreeks geometrisch getransformeerd voor iedere oriëntatie van de auto. Het stapelen van alle uitkomsten levert de totale obstakeltransformatie. De doelconfiguratie wordt met het user-interface gespecificeerd, en met heuristiek 0 (nul) wordt de manoeuvre voor alle mogelijke begintoestanden berekend. Zo worden behoorlijk gecompliceerde manoeuvres berekend en uitgevoerd. Een voorbeeld is een auto die verkeerd om in een parkeerplaats staat: hij rijdt eruit, doet zijn 'straatje keren' en een parkeersteek. Als alternatieve kostprijs kan men 'alleen vooruit' bewegingen opleggen, om gewone verkeersmanoeuvres te plannen. De rekentijd wordt gemeten met een hetzelfde mechanisme als in hoofdstuk 2.

Om de geplande paden te testen werd de methode toegepast op een standaard radio-bestuurd autootje, schaal 1:10. Het wagentje was bevat binnen een rechthoekige testopstelling, met twee andere voertuigen als obstakels. De afmeting van de testopstelling bepalen de begrenzingen van de mogelijke posities, en daarmee de grootte van de configuratieruimte. De berekende configuratieruimte bevat alle mogelijke optimale paden naar het doel. Dit wordt in 'common memory' geladen, voor gebruik door het controle systeem. Een 'vision' systeem is geïntegreerd met het controle systeem om de paden uit te voeren. Er zijn enige conversies nodig door de controller om soepele betrouwbare bewegingen mogelijk te maken. Deze smeren de discretisatiefout uit door verder weg liggende streefpunten te bepalen, door de te regelen curve aan te passen aan de wereldcoordinaten i.p.v. de gediscretiseerde (en geïdealiseerde) geplande coordinaten, en door een stopconditie te definiëren die onvolkomenheden in discretisatie, besturing en mechanisme ondervangt.

Ook andere voertuigen en omgevingen worden behandeld in hoofdstuk 4. Omdat de testopstelling nogal klein was werd een simulatie van een grotere werkomgeving gemaakt om aan te tonen dat de oplossing goed schaalbaar is. In die simulatie kunnen meer complexe manoeuvres worden gedemonstreerd. Om het gedrag van de auto te veranderen in dat van een rupsvoertuig hoeven alleen de nabuurschap en de basiskosten worden veranderd. Een rupsvoertuig heeft een eenvoudiger kinematica dan een auto, omdat hij om zijn as kan draaien. En we ondervangen ook het bezwaar dat het plannen te langzaam zou zijn voor toepassingen op de snelweg. Door de beweging van het voertuig relatief i.p.v. absoluut te beschouwen laten we zien dat de kinematica van een auto bij grote snelheid veel eenvoudiger is - i.h.b. kan de auto nu *zijwaards* bewegen t.o.v. andere auto's, en is altijd aan de andere evenwijdig. Deze vereenvoudiging staat ons toe om kleine configuratieruimten te gebruiken, en die zijn snel te berekenen. Met snelle sensoren kan het gemakkelijker zijn een auto op een snelweg te besturen, dan er één te parkeren.

Hoofdstuk 5 introduceert differentieel A* als een algemene en snelle methode om door A* gegenereerde oplossingruimtes aan te passen als er veranderingen plaatsvinden in de onderliggende overgangen. De toegevoegde, verwijderde, verhoogde of afgenomen overgangen of nodes worden geïdentificeerd, en doorgegeven aan een *verschil-machine* ('difference engine'). Deze berekent nodes en wist afhankelijke nodes uit, daarmee de OPEN nodes bepalend die nodig zijn om de veranderde delen van de graaf opnieuw te genereren. Vaak beïnvloeden deze veranderingen slechts een klein deel van de totale oplossingsgraaf die door A* berekend zou worden. Elk type verandering en de benodigde acties worden gedefinieerd; ook worden gevallen geïdentificeerd waarvoor geen actie nodig is; dit soort gevallen geven de meest duidelijke besparing t.o.v. volle herberekening met A*.

Het differentieel A* algoritme wordt geanalyseerd voor ieder van de belangrijkste acties: herberekening, uitwissen van invloed, en het toevoegen van een node aan de kandidaten; dit geeft aan dat de belangrijkste overhead van differentieel A* de uitwisfunctie is. De mate waarin toepassingen deze uitwisfunctie nodig hebben wordt daarmee het middel om te bepalen wanneer gebruik van differentieel A* voordeliger is dan volledig herberekenen met A*.

Veel van de karakteristieken van A* worden overgeërfd door differentieel A*,namelijk: de berekening van optimale overgangen en topologie, de mogelijkheid om meerdere dimensies te behandelen, en een tijdsduur die hoofdzakelijk een functie is van het aantal geëxpandeerde (OPEN) nodes. Het voornaamste voordeel van differentieel A* is de snelle aanpassing in werkomgevingen waarin de veranderingen slechts een deel van de oplossingsruimte beïnvloeden.

In **hoofdstuk 6** wordt de trade-off tussen differentieel A* en A* onderzocht. Het differentieel A* algoritme wordt toegepast op een padplanprobleem in de robotica waarin onverwachte veranderingen optreden. In dit nieuwe gebruik kan de robot 1) blijven bewegen omdat de vorige oplossing meestal niet verandert, 2) obstakels op het pad ter plekke in zijn kennis opnemen en 3) de beste manoeuvre maken op grond van de alle aanwezige kennis. Dit stelt het systeem in staat te 'leren' en de beste beslissing te nemen gebaseerd op de meest recente kennis van de wereld.

De tijdsduur van de methode wordt getest in een situatie met heuristiek 0 (nul). De tijd voor de drie elementaire operaties wordt vergeleken met de tijd voor volledige herberkening. Er wordt gevonden dat als minder dan 55% van de nodes worden gewist, differentieel A* meer efficient is dan A*.

Differentieel A* is toepasbaar in situaties waarin thans gebruik gemaakt wordt van conventionele A*, en kan de oplossingsnelheid vaak aanzienlijk vergroten. Goede kandidaatproblemen zijn problemen met veranderingen in de kostprijs van overgangen, en hoofdzakelijk stabiele doelen met wisselende beginpunten.

Hoofdstuk 7 bespreekt de inzichten die door dit onderzoek zijn verkregen, en geeft suggesties voor toekomstig onderzoek. Dit hoofdstuk verwijst naar appendix C, waarin een inleiding wordt gegeven in een nieuwe methode voor rendez-vous planning.

References

[1] Avnaim F., Boissonnat J.D. and Faverjon B., *A practical exact motion planning algorithm for polygonal objects amidst polygonal obstacles*, Proceedings IEEE International conference on Robotics and Automation, pp. 1656-1661, 1988.

[2]Asada H. and Slotine J.-J.E., Robot Analysis and Control, John Wiley and Sons, 1986.

[3]Barraquand J. and Latombe J., *On Nonholonomic Mobile Robots and Optimal Maneuvering*, IEEE International Symposium on Intelligent Control, September 1989.

[4]Borgefors G., *Distance Transformations in Arbitrary Dimensions*, Computer Vision, Graphics and Image Processing, Vol. 27, pp. 321-345, 1984.

[5] Boult T., *Updating Distance Maps when Objects Move,* SPIE Conference on Mobile Robots, 1987.

[6] Bourbakis N.G., Applications of Learning and Planning Methods, Invited Chapter: *Differential A*: An Adaptive Search Method Illustrated with Robot Path Planning for Moving Obstacles & Goals and an Uncertain Environment,* Trovato K., World Scientific Publishing Co., 1991.

[7] Cavalieri S., Di Stefano A. and Mirabella O., *Optimal Path Determination in a Graph by Hopfield Neural Network*, Neural Netorks, Vol. 7, No. 2, pp. 397-404, Elsevier Science, Ltd, 1994.

[8] Cheung E. and Lumelsky V., *Motion Planning for Robot Arm Manipulators with Proximity Sensing*, IEEE Int. Conference on Robotics and Automation, Vol 2, 1988.

[9]Corman, Leiserson & Rives, Introduction to Algorithms, MIT Press, 1990.

[10] Dijkstra E.W., *A Note on Two Problems in Connection with Graphs*, Numerische Mathematik, 1959.

[11]Dorst L., *Discrete Straight Line Segments: Parameters, Primitives and Properties*, Ph.D. Thesis, Delft University of Technology, 1986.

[12]Dorst L. and Trovato K., *Optimal Path Planning by Cost Wave Propagation in Metric Configuration Space,* SPIE Conference on Advances in Intelligent Robotics Systems, November 1988.

[13] Dorst L. and Verbeek P., *The Constrained Distance Transformation: A Pseudo-Euclidean, Recursive Implementation of the Lee-Algorithm*, Proceedings of the European Signal Processing Conference, The Hague, 1986.

[14] Dorst L., Mandhyan I. and Trovato K., *The Geometrical Representation of Path Planning Problems*, Robotics and Autonomous Systems 7, pp. 181-195. North-Holland Publishing, 1991.

[15]Featherstone R., *Swept Bubbles: A Method of Representing Swept Volume and Space Occupancy*, Philips Technical Document No. MS-90-069, 1990.

[16] Fijalkowski B. and Trovato K., *A Concept for a Mechatronically Controlled Full-Time 4WD x 4WB x 4WA x 4WS Intelligent Vehicle for Drivers with Special Needs*, International Symposium on Automotive Technology and Automation, September 1994.

[17]Greenwood D.T., *Principles of Dynamics*, Prentice-Hall, 1965.

[18] Hendriks A.J., *Control and Tracking of Car Maneuvers*, Philips Internal Document, TN-90-061, 1990.

[19] Hunter G.M. and Steiglitz K., *Operations on Images Using Quad Trees*, IEEE Trans. on Patern Analysis and Machine Intelligence, Vol. 1, 1979.

[20]Khatib O., *Real Time Obstacle Avoidance for Manipulators and Mobile Robots*, International Journal of Robotics Research, Vol 5 No. 1, pp. 90-98, Spring 1986.

[21] Latombe J.-C., *Robot Motion Planning*, Kluwer Academic Pub., 1991.

[22] Lee C.Y., *An Algorithm for Path Connections and Its Applications*, IRE Transactions on Electronic Computers, September, pp. 346-365, 1961.

[23]Lozano-Pérez T. and Wesley M. A., *An Algorithm for Planning Collision-Free Paths Among Polyhedral Obstacles*, Communications of the ACM, No. 10, 1979.

[24] Lozano-Perez T., *Spatial Planning: A Configuration Space Approach*, IEEE Trans. Computers, Vol. 32, No. 2, 1983.

[25]Lyons D., *RS: A Formal Model of Distributed Computation For Sensory-Based Robot Control*, Ph.D. Thesis, University of Massachusetts, September 1986.

[26] Lyons D., *When is Reactivity Important in Task Planning for Automated Manufacturing?,* IEEE International Conference on Robotics & Automation, May 1993.

[27] Microsoft, Inc., *Microsoft Project*, Redmond, Washington, 1995.

[28] Newman W., *The Optimal Control of Balanced Manipulators, Robotics: Theory and Applications*, The American Society of Mechanical Engineers, 1986.

[29]Newman W., *High-Speed Robot Control in complex Environments,* Ph.D. Thesis, MIT, October 1987.

[30] Nilsson N.J., Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, 1971.

[31] Nilsson N., *Principles of Artificial Intelligence*, Tioga, 1980.

[32] Nwagboso C., Advanced Vehicle and Infrastructure Systems: Computer Application, Control and Automation, contributed chapter: *Collision free Maneuvering and Control for an Autonomous Vehicle*, Trovato K., John Wiley and Sons Ltd., 1996.

[33] Pearl J. and Kim J.H., *Studies in Semi-Admissible Heuristics*, IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol 4, No. 4, July 1982.

[34] Pearl J., *Heuristics: Intelligent Search Strategies for Computer Problem Solving,* Addison Wesley, Reading, Mass, 1984.

[35] Press W., Flannery B., Teukolsky S. and Vetterling W., Numerical Recipes in C, Press Syndicate of the University of Cambridge, 1988.

[36] Samet H., *The Quadtree and Related Hierarchical Data Structures*, Computing Surveys, Vol. 16, 1984.

[37] Schwartz J. and Sharir M., *On the 'Piano Movers' Problem: I. The Case of a Two-dimensional Rigid Polygonal Body Moving Amidst Polygonal Barriers*, New York University Report No. 39, October 1981.

[38] Schwartz J. and Sharir M., *On the 'Piano Movers' Problem: II. General Techniques for Computing Topological Properties of Real Algebraic Manifolds*, New York University Report No. 41, February 1982.

[39] Sedgewick R., Algorithms, Addison-Wesley, Reading, Mass, 1988.

[40] Shyu J.and Chuang C., *Automatic Parking Device For Automobile*, U.S. Patent No. 4,931,930, June 5, 1990.

[41] Snyder W.E., Industrial Robots: Computer Interfacing and Control, Prentice-Hall, 1985.

[42] Thoone M.L.G, Driessen L.M.H.E., Hermus C.A.C.M. and van der Valk K., *The Car Information and Navigation System CARIN*, Proceedings Society of Automotive Engineers International Congress and Exposition, 1987.

[43] Tournassoud P. and Jehl O., *Motion Planning for a Mobile Robot with a Kinematic Constraint*, IEEE Conf. Robotics & Automation, pp. 1785-1790, 1988.

[44] Trovato K. and Dorst L., *Method and Apparatus for Path Planning*, U.S. Patent Pending, 1987.

[45] Trovato K. and Dorst L., *Differential Budding: Method and Apparatus for Path Planning with Moving Obstacles and Goals*, U.S. Patent No. 4,949,277, August 14, 1990.

[46] Trovato K., *Differential A*: An Adaptive Search Method Illustrated with Robot Path Planning for Moving Obstacles & Goals and an Uncertain Environment,* International Journal of Pattern Recognition and Artificial Intelligence - Vol. 4, No. 2 June 1990.

[47]Trovato K. and Dorst L., *Path Planning with Transition Changes*, U.S. Patent No. 5,083,256, January 21, 1992.

[48] Trovato K. and Dorst L., *Method and Apparatus for Controlling Maneuvers of a Vehicle*, U.S. Patent Pending, 1988.

[49]Trovato K., *Autonomous Planning of Vehicle Maneuvers*, Intelligent Autonomous Systems-II, Amsterdam, The Netherlands, 1989.

[50] Trovato K., A*utonomous Vehicle Maneuvering*, SPIE Advances in Intelligent Robotics Systems Conference, 1991.

[51]Trovato K. and Dorst L., *Method and Apparatus for Smooth Control of a Vehicle with Automatic Recovery from Interference*, U.S. Patent Pending, 1992.

[52] Trovato K., *Autonomous Vehicle Parking and Maneuvering*, International Symposium on Automotive Technology & Automation, Aachen, September 1993.

[53]Trovato K., *Method and Apparatus for Identifying or Controlling Travel to a Rendezvous*, U.S. Patent Pending, 1993.

[54] Trovato K., *General Method for Strategic Assessment and Planning Problems*, Invited paper, Sixth Joint Service Data Fusion Symposium, 1993.

[55] Trovato K. and Mehta S., *Method and Apparatus for Controlling High Speed Vehicles*, U.S. Patent No. 5,220,497, June 15, 1993.

[56]Unimation, Inc. *User's guide to VAL*, Danbury, CT 1979.

[57] van de Panne M., Fiume E. and Vranesic Z., *Reusable Motion Synthesis Using State-Space Controllers*, Computer Graphics SIGGRAPH '90 Conference Proceedings, Vol. 24, No. 4, pp. 225-234, ACM Press, 1990.

[58] Verbeek P., Dorst L., Verwer B. and Groen F.C.A., *Collision Avoidance and Path Finding through Constrained Distrance Transformation in Robot State Space*, Proc. Intelligent Autonomous Systems, Amsterdam, The Netherlands, 1986.

[59] Verwer B., *Distance Transforms*, Ph.D. Thesis, Technische Universiteit Delft, 1991.

[60] Verwer B., *Heuristic Search in Robot Configuration Space using Variable Metric,* Third Conference on Artificial Intelligence for Space Applications, NASA Conference Publication 2492, pp. 153-154, November 1987.

[61]Verwer B., *A MultiResolution Work Space, MultiResolution Configuration Space Approach to Solve the Path Planning Problem*, Proceedings of the IEEE International Conference on Robotics and Automation, 1990.

[62]Warmerdam T., *The SPINE User's Guide,* Philips Technical Document No. TR-88-014, 1988.

[63] Winston P., Artificial Intelligence, Addison-Wesley Publishing Company, 1977.

[64] Wong E.K. and Fu K.S., *A Hierarchical Orthogonal Space Approach to Three-Dimensional Path Planning*, IEEE Journal of Robotics and Automation, Vol. 2., 1986.

Appendix A: A* Notation

$\mathbf{P}_{ni\text{-}nj}$: The set of all paths going from n_i to n_j , if node n_j is accessible from node n_i.

$\mathbf{P}_{n\text{-}\Gamma}$: The set of all paths going from n to the set Γ.

$\mathbf{P}_{n\text{-}\gamma}$: Any path going from n to a goal γ, therefore $\mathbf{P}_{n\text{-}\gamma} \in \mathbf{P}_{n\text{-}\Gamma}$.

$\mathbf{P}^*_{ni\text{-}nj}$: The set of <u>cheapest</u> paths going from n_i to n_j. The cost of such a path is denoted as $k(n_i,n_j)$ (following Nilsson [30]).

g*(n): The cheapest cost of paths going from s to some node n; thus $g^*(n) = k(s,n)$

h*(n): The cheapest cost of paths going from n to Γ, thus $h^*(n) = \min_{\gamma \in \Gamma} k(n,\gamma)$

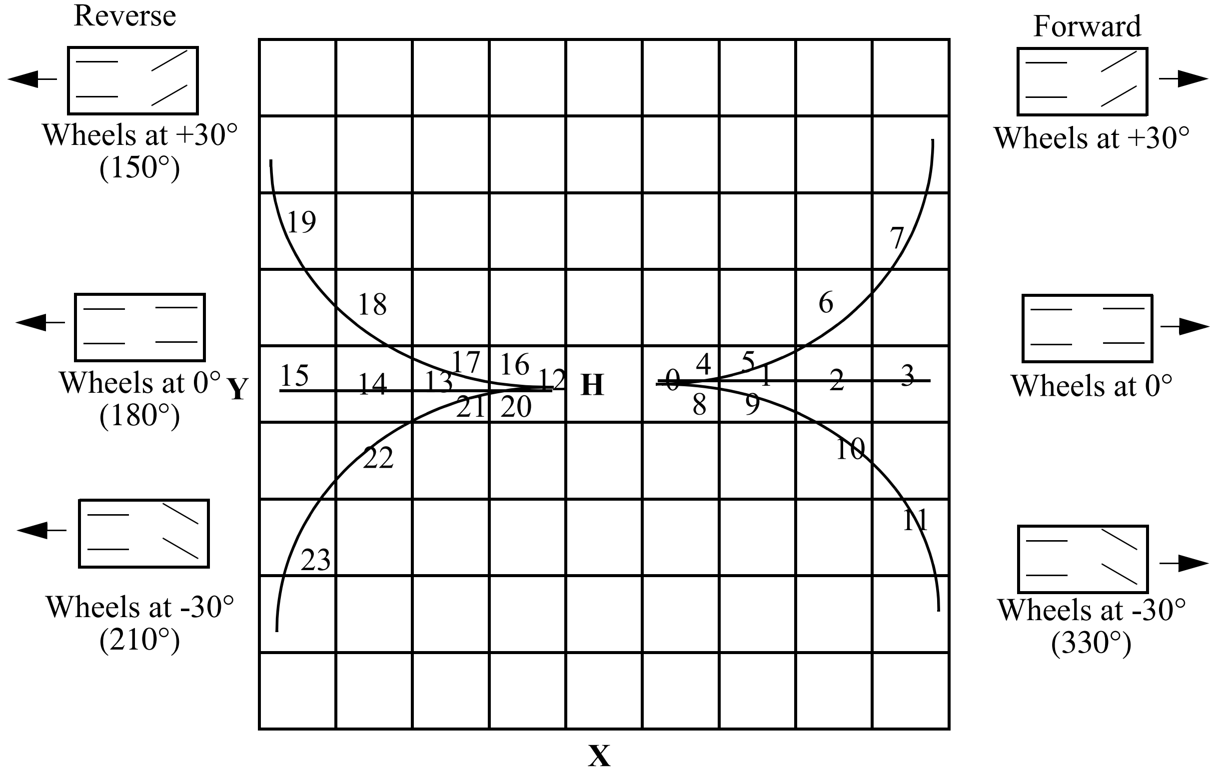
C*: Cheapest cost paths going from s to Γ, ie. $C^* - h^*(s)$

Γ^*: Set of *optimal* goals, i.e. the subset of goal nodes accessible from s by a path or paths that cost C*

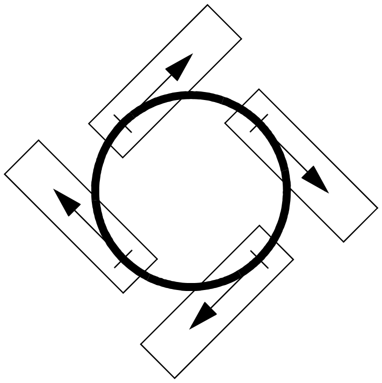
Multiple starting nodes are accomodated in [34] (pg. 74), by the addition of a dummy node, having zero cost ‘transitions’ to the set of actual starting nodes.

Appendix B: Discrete Neighborhood Calculation

This appendix describes the method to generate a neighborhood in D directions with N_D neighbors in each of the D directions. In the example below there are six directions, combinations of forward and reverse with the wheels straight, left and right. All of the actual neighbors are shown, unlike the non-redundant neighborhood of Figure 44. The non-redundant neighborhood removes the duplicated entries and selects the lowest cost representative neighbor. It then must create a precedence ordering of the nodes.



The image above is only a portion of the full circle (helix in the 3-D configuration space) that describes the extreme turning motion of the car. The full circle is shown below for steering wheels at -30° (i.e. hard right), moving in the forward direction. The position where the centerline of the car is tangent to the circle is midway between the rear wheels.



In an ideal way, the neighborhood would be comprised of only neighbors that best characterize the path from that state to the home (H) position. In this implementation however, all neighbors were retained to have consistency in the determination of neighbors, that is, no orientation specific code. In addition, this provides a straightforward way to define neighbors used in the precedence of later neighbors. As an alternative, a tree-like structure could be created in which later neighbors are not searched if ‘parent’ neighbors are blocked by forbidden states. The loss due to this mechanism however, is that several neighbors are computed that will never become part of an optimal path (for example 4 or 8).

The neighborhood is fixed for each orientation of the vehicle, therefore it can be computed once as part of the setup of the program. Given a specific neighbor (having an implied drive-wheel angle Θ and transmission setting of forward or reverse) and current car angle θ, a value for the change in x, y, θ, and distance traveled can be computed. The result must account for changes in the vehicle’s orientation θ, as well as discretization factors.

First, the neighborhood is reduced to two fundamental sets of neighbors, those that are straight and analogous to neighbors 0-3, and the those that are curved and analogous to 4-7. For each actual neighbor, there is a positive or negative affect on the x, y or θ based on the location of the neighbor. Therefore there is an x-factor, y-factor or θ-factor that must be determined according to:

x-factor:	-1	if $90° < \Theta < 270°$
	1	otherwise
y-factor	-1	if $\Theta > 180°$
	1	otherwise
θ-factor	-1	if $(90° < \Theta < 180°)$ or $(270° < \Theta < 360°)$
	1	otherwise

The turning radius R is then computed as:

$$R = \text{track}/2 + \text{Wheelbase}/|\tan(\Theta)|$$

To determine at least nominal distances along the x axis (dx), values equal to the discretization factor are used.

Therefore, dx for neighbors 0, 1, 2, and 3 might be 0.25, 0.5, 0.75 and 1.0. From this, the value of $d\theta$ can be computed.

$$d\theta = dx/R$$

$$dy = (dx * (d\theta)^2)/ 2$$

The true distance moved as a result of this neighbor transition is then:

$$\text{distance} = |dx/\cos(d\theta/ 2)|$$

Now the dx and dy values are corrected for the angular translation, and are subsequently discretized so that:

$$\Delta x = \text{discretized\_state}(x\text{-factor}*dx*\cos(\theta) - y\text{-factor}*dy*\sin(\theta))$$
$$\Delta y = \text{discretized\_state}(x\text{-factor}*dx*\sin(\theta) - y\text{-factor}*dy*\cos(\theta))$$
$$\Delta\theta = \text{radians\_to\_state}(\theta\text{-factor} * d\theta)$$

These values are stored in a table so they can be determined by lookup rather than computed on the fly each time. A portion of such a table is given below for the neighborhood of the simulation. Each direction (dir) is an index into the table of neighbors for each of the discretized car angles.

<u>First Four Neighborhoods for the Vehicle Simulation</u>

dir : dist dx dy dθ

carangle 0	**carangle 1**	**carangle 2**	**carangle 3**
dir 0: 0.250 1 0 0	dir 0: 0.250 0 0 0	dir 0: 0.250 0 0 0	dir 0: 0.250 0 0 0
dir 1: 0.500 2 0 0	dir 1: 0.500 1 0 0	dir 1: 0.500 1 0 0	dir 1: 0.500 1 0 0
dir 2: 0.750 3 0 0	dir 2: 0.750 2 0 0	dir 2: 0.750 2 0 0	dir 2: 0.750 2 0 0
dir 3: 1.000 4 0 0	dir 3: 1.000 3 0 0	dir 3: 1.000 3 0 0	dir 3: 1.000 3 1 0
dir 4: 0.250 1 0 0	dir 4: 0.250 0 0 0	dir 4: 0.250 0 0 0	dir 4: 0.250 0 0 0
dir 5: 0.502 2 0 1	dir 5: 0.502 1 0 1	dir 5: 0.502 1 0 1	dir 5: 0.502 1 0 1
dir 6: 0.757 3 0 2	dir 6: 0.757 2 0 2	dir 6: 0.757 2 0 2	dir 6: 0.757 2 0 2
dir 7: 1.017 4 0 3	dir 7: 1.017 3 0 3	dir 7: 1.017 3 1 3	dir 7: 1.017 3 1 3
dir 8: 0.250 1 0 0	dir 8: 0.250 0 0 0	dir 8: 0.250 0 0 0	dir 8: 0.250 0 0 0
dir 9: 0.502 2 0 -1	dir 9: 0.502 1 0 -1	dir 9: 0.502 1 0 -1	dir 9: 0.502 1 0 -1
dir 10: 0.757 3 0 -2	dir 10: 0.757 2 0 -2	dir 10: 0.757 2 0 -2	dir 10: 0.757 2 0 -2
dir 11: 1.017 4 0 -3	dir 11: 1.017 4 0 -3	dir 11: 1.017 3 0 -3	dir 11: 1.017 3 0 -3
dir 12: 0.250 -1 0 0	dir 12: 0.250 0 0 0	dir 12: 0.250 0 0 0	dir 12: 0.250 0 0 0
dir 13: 0.500 -2 0 0	dir 13: 0.500 -1 0 0	dir 13: 0.500 -1 0 0	dir 13: 0.500 -1 0 0
dir 14: 0.750 -3 0 0	dir 14: 0.750 -2 0 0	dir 14: 0.750 -2 0 0	dir 14: 0.750 -2 0 0
dir 15: 1.000 -4 0 0	dir 15: 1.000 -3 0 0	dir 15: 1.000 -3 0 0	dir 15: 1.000 -3 -1 0
dir 16: 0.250 -1 0 0	dir 16: 0.250 0 0 0	dir 16: 0.250 0 0 0	dir 16: 0.250 0 0 0
dir 17: 0.502 -2 0 -1	dir 17: 0.502 -1 0 -1	dir 17: 0.502 -1 0 -1	dir 17: 0.502 -1 0 -1
dir 18: 0.757 -3 0 -2	dir 18: 0.757 -2 0 -2	dir 18: 0.757 -2 0 -2	dir 18: 0.757 -2 0 -2
dir 19: 1.017 -4 0 -3	dir 19: 1.017 -4 0 -3	dir 19: 1.017 -3 0 -3	dir 19: 1.017 -3 0 -3
dir 20: 0.250 -1 0 0	dir 20: 0.250 0 0 0	dir 20: 0.250 0 0 0	dir 20: 0.250 0 0 0
dir 21: 0.502 -2 0 1	dir 21: 0.502 -1 0 1	dir 21: 0.502 -1 0 1	dir 21: 0.502 -1 0 1
dir 22: 0.757 -3 0 2	dir 22: 0.757 -2 0 2	dir 22: 0.757 -2 0 2	dir 22: 0.757 -2 0 2
dir 23: 1.017 -4 0 3	dir 23: 1.017 -3 0 3	dir 23: 1.017 -3 -1 3	dir 23: 1.017 -3 -1 3

Appendix C: Rendezvous Planning

This is an introduction to the composite algebra enabling multiple machines or 'actors' to achieve possible objectives of interaction while maintaining final goals. This rendezvous planning [54,53] is a task level planning method that allows a variety of cost measures to be satisfied for multiple actors. The set of possible rendezvous states that satisfy all actor's criteria is computed, and a preferred option may be selected based on yet another criteria or randomly.

1. Introduction

Diverse types of planning problems share a number of attributes. They often have a goal or set of goal states, a current state, some measure of success, and limits on how the system can progress from the starting state to any other state. The objective is to provide an optimal path to the nearest goal while avoiding the limits, which can be hard obstacles, movement based on rules of the game or even kinematic constraints. This appendix describes a framework that can be used to represent such problems, and then gives examples where it has been applied.

2. Planning the Coordination of Multiple Actors - Synergistic Planning

In the previous examples, only single machines are under the control of the planning system without regard to the requirements of other machines or to the collective synergy that can be obtained. The same general planning framework that has been outlined previously may be used to find locations for rendezvous or simultaneous requirements of multiple machines or actors. Each actor may have different cost measures for efficiency, different neighborhood constraints, and different starting and ending requirements. These may all be factored into a single problem so that predictive, optimal motions can be computed.

There are several steps required to perform this task which are listed in Figure 81. The first step is to specify the problem in terms of scenarios for each of the actors. A scenario can be as simple as the information in Figure 82. Next, the obstacle layout is sensed.

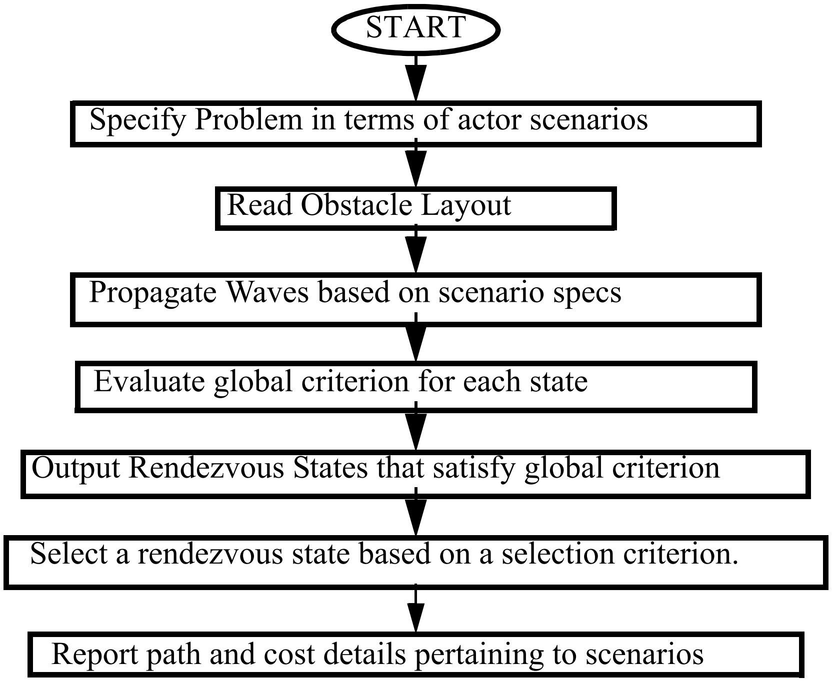


Figure 81: Rendezvous Planning.

```
Scenario a1                  /* actor a: scenario 1 */
Start                /* tells whether to propagate away or towards the 'source' */
                     3,2     /*equivalent locations that the actor could start from */
                     1,3     /* coordinates are listed <row>,<column> */
Cost measure
                     Min_distance /* for Neighborhood, each move is cost=1 */
                             /* horiz and vert. moves only */
```

Figure 82: A Scenario.

At least one of two fuel trucks and an airplane must rendezvous but:

- The plane (fuel cost cost measured in scenario A) must still have at least 100 gallons of fuel in its tank,
- The first fuel truck (distance cost measured in scenario F1) must travel less than 50 miles, and
- The second fuel truck (time cost measured in scenario F2) can only meet between 2pm and 4pm.

The global criterion defines locations that select only one fuel truck for rendezvous with the plane. Specifically, the global criterion above might look like:

G=(A>100) and ((F1<50) xor ((F2 > 1400) and (F2 < 1600)))

Figure 83: Global Criterion.

This information is used to set up the configuration spaces of the respective scenarios. Cost waves are then propagated based on the setup information for each independent scenario. The setup defines a start or goal 'source' based on the measurement needed between the reachable states and source. Naturally, these configuration spaces could be computed in parallel. At each state there is a cost_to_source and direction arrows leading to the next state toward the source. The global criterion is then evaluated for each task state. An example of a global criterion is in Figure 83, where it is first written in terse english, and then translated into the boolean expression required for the global criterion.

The task states are the regions where any possible meeting might take place between the actors. They may naturally map or transform into more than one configuration state in each of the configuration spaces. By evaluating the global criterion at each of these transformed task states, a set of satisfying (rendezvous) configuration states results. Since many states can be workable for the rendezvous, a 'best choice' can be selected from the set, based on a selection criterion. For example, of the rendezvous points possible, choose the one that will cost the least in terms of total dollars spent by all of the actors.

The specific movement/action details for each actor can then be read out from the respective scenario-based configuration spaces. This gives coordinated motion for all the actors to achieve a shared mission.

3. Example Coordination Problem

The following is a problem involving the coordination of two actors, A and B. Candidate rendezvous points are to satisfy the following global criteria:

1) the distance A travels from the start to the rendezvous state must be between (2,8],
AND
2) the distance A travels in total must be precisely 7,
AND
3) the time B travels from the start to the rendezvous state must be less than 4,
AND
4) The fuel spent by B from the rendezvous state to the goal must be at least 4,
AND
5) the sum of the (distance traveled by A + the fuel spent by B) must be less than 9,
AND
6) the sum of the (distance traveled by A + the fuel spent by B) must be greater than 6.

Scenarios representing the needed measurements for each actor can be enumerated as follows:

A: distance traveled away from the potential starting states
A: distance traveled from the rendezvous to the potential goal states
B: time traveled from the potential starting states
B: Fuel spent from the rendezvous to the potential goal states

For each of these four requirements, scenarios can be written for the specific start and goals. They are as follows:

Scenario A1 /* actor A: scenario 1 */ Start 3,2 1,3 Cost Measure Min_distance /* H&V moves. Each move is cost=1 */	Scenario B1 /* actor B: scenario 1 */ Start 4,8 4,9 Cost Measure Min_time /* H&V moves. Each move is cost=1 */
Scenario A2 /* actor A: scenario 2 */ Goal 4,8 / 1,7 Cost Measure Min_distance /* H&V moves. Each move is cost=1 */	Scenario B2 /* actor B: scenario 2 */ Goal 3,2 1,3 Cost Measure Min_fuel /* H&V moves. Each move is cost=1 */

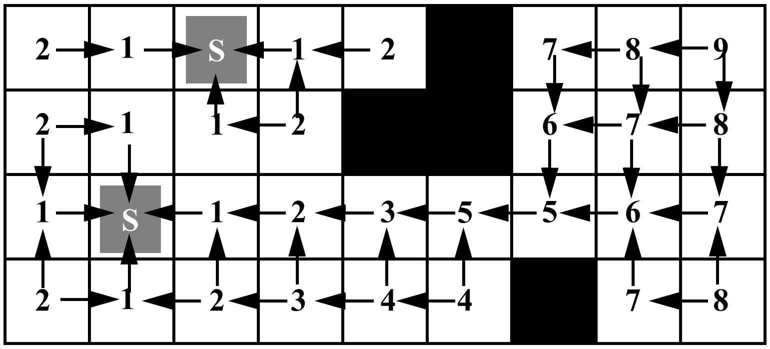
The boolean expression corresponding to these constraints is:

(A1 > 2) and (A1 <= 8) and (A1+A2 = 7) and (B1 < 4) and (B2 > 4) and
(A2+B2 < 9) and (A2 + B2 > 6)

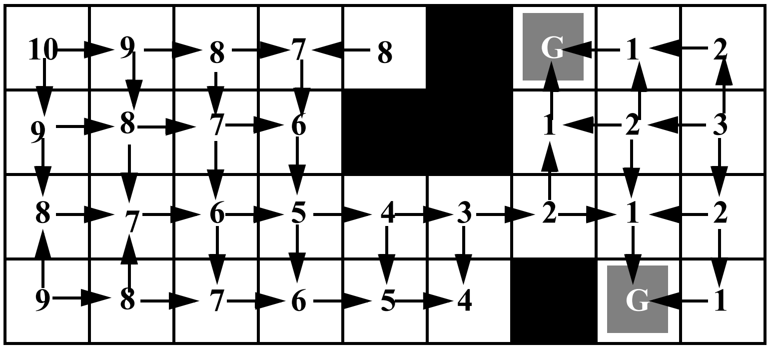
Each of the above scenarios correspond to an A* calculation in configuration space. For this example, the obstacle layouts are assumed to be the same for each actor-scenario. The four resulting spaces might be as in Figure 84.

Once these spaces are completed, the global criterion is evaluated for each state these actor-scenarios have in common (all in this case). The candidate rendezvous states that meet all the constraints will have a ‘true’ boolean result. For this example, the states are highlighted in Figure 85.

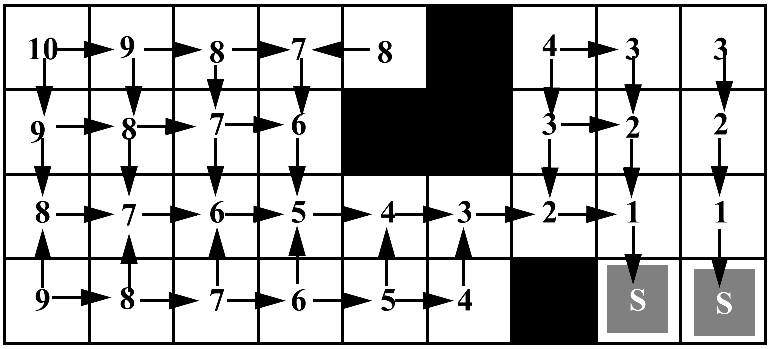
Scenario A1: Paths generated to optimize distance from start.



Scenario A2: Paths generated to optimize distance to goal.



Scenario B1: Paths generated to optimize time from start.



Scenario B2: Paths generated to optimize fuel use to goal.

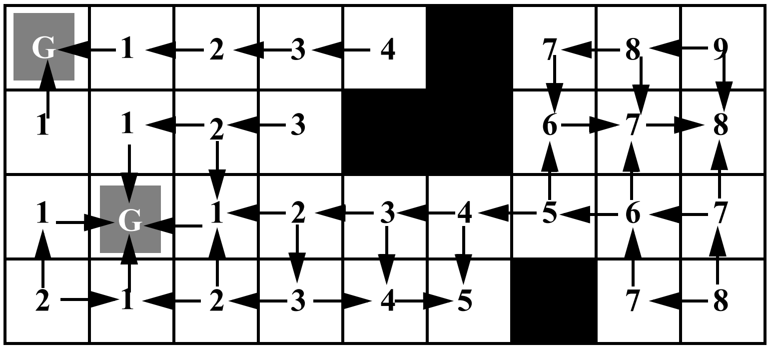


Figure 84: Paths for Two Actors - Two Scenarios Each.

Candidate rendezvous states are shown in cross-hatch. The selected rendezvous state is marked with an asterisk.

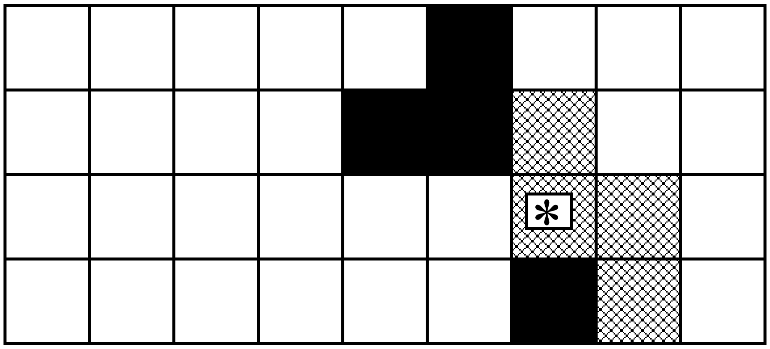


Figure 85: Candidate Rendezvous States.

Note: This is one of the two equivalent paths to different goals. Both would go through the rendezvous state.

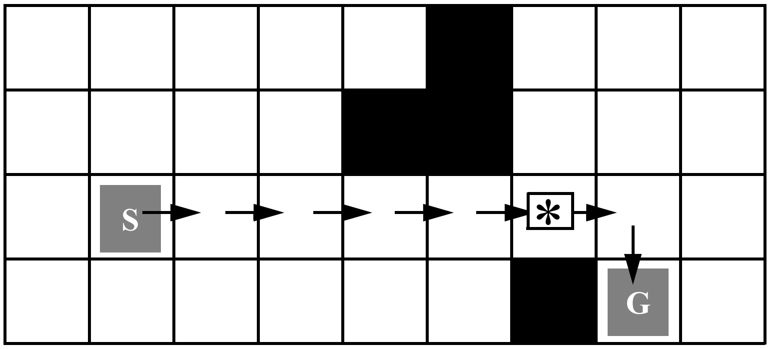


Figure 86: Path for Actor A from Start, through Rendezvous, to Goal.

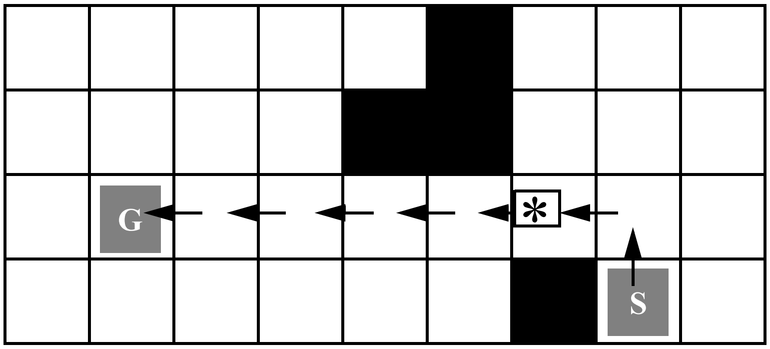


Figure 87: Path for Actor B from Start, through Rendezvous, to Goal.

If it is assumed that any candidate rendezvous state is acceptable, then a state may be selected at random for implementation. It is often the case however that one of the candidates is preferable to the others. In this case a selection criterion can be used to find the best candidate, for example one with the least implementation cost.

Once a state is selected, then the moves for each of the actors can be read from the respective scenario-based configuration spaces. For example if the state marked with the asterisk (*) is selected, then Actor A would proceed as in Figure 86 and Actor B would proceed as in Figure 87.

4. Coordination of Fleet Trucks

The above synergistic planning method is general in that it is equally applicable to many domains. To show the breadth of the types of problems that can be handled, I will enumerate a few using a sample scenario.

Suppose we own a national fleet of delivery trucks which have irregular delivery sites. Some trucks haul double-trailers and some haul single trailers. Some of the truckers are independent and have licenses to haul only specific cargo in specific states. Some trucks have limits on the weight they can haul. We can then compute answers for the following types of questions:

- What locations are feasible for exchanges between specific trucks?
- How shall they be coordinated so that all of the deliveries arrive on time?
- How can certain truck overhead be minimized by selecting the most time-direct routes for each?
- If a central depot for a regular trucking schedule is to be established, where should it be so as to be central to all trucks given their limitations?
- If a road is closed due to weather, traffic, construction, etc., how can coordination take place to manage the exchange between two trucks elsewhere?

5. Brief Analysis

While most methods would require a computation for a product of the state spaces, this method uses the sum of the state spaces by simply overlapping them with a global and selection criterion.

6. Other Applications

The military is another area where coordination of forces is critical. In a defensive or offensive strategy, one may place opposing actors into the equation, assuming that they will use the most opportune strategy available. This computational tool identifies the weak areas of each strategic side, and the timing of a strategic arrangement with the most favorable outcome (e.g. fewest casualties). This is also described in a previous paper [54].

Multiple machines or actors can be coordinated to satisfy their individual and collective constraints. Given this general framework, more efficient trucking distribution can be planned, as well as problems in other domains such as strategic defense. It is possible to evaluate the scope of threats and determine the appropriate temporal response.

Acknowledgment

I would like to thank my promotor, Dr. Frans Groen, for his enthusiasm and remarkable ability to view technology succinctly at many levels. His critical comments have led to many improvements in this thesis.

My co-promotor and colleague while at Philips, Dr. Leo Dorst, has completely changed my career from the moment he interviewed in 1985. He has passion for technical understanding and a contagious enthusiasm that affects his own work and those around him. I would like to thank him for opening this area to me and for our many hours of delightful, challenging and fruitful discussions. I am fortunate to have been both his colleague and student.

I would also like to thank many people at Philips Research. The work was first supported as part of the Robotics Research Department lead by Dr. Ernie Kent and Dr. Mark Rochkind. Their focus on long-term challenges provided the freedom to push the boundaries of the original methods. Since then, even with commercial pressures that have come to most research organizations, Dr. Peter Bingham, Dr. Mark Hoffberg, and Fred Teger have provided the much needed support and patience to allow me to finalize this work. I am also grateful for the patience of my current colleagues, Frankie, Dan and Susan. I also owe more than thanks to people who each contributed their expertise to build the testbeds. Julie Harazduk-Braunhut and Sandeep Mehta contributed their indispensible expertise in Sunview to help build the user interfaces of the car and robot arm. Dr. Wyatt Newman bravely provided his robot arm for experiments, and later contributed his expertise for the low-level control of the vehicle. Teun Hendriks donated his time and talent with the PAPS vision system to recommend the best overall sensing strategy and create the software to make it happen. Thom Warmerdam devoted numerous technical support hours to ensure his SPINE real-time operating system was delivering superb results, whether it required extensions or user education. Frank Guida provided expertise in system and device electronics for the robot arm and car, and mechanical design for both the original vehicle testbed and a subsequent portable version. In addition, his strong sense of organization saved innumerable hours and his sense of humor saved my sanity. Ken Nawyn contributed his detailed knowledge of radio-controlled cars. Because the testbeds required the integration of many different technologies, the (usually) calm and professional interaction, focused on solid results, resulted in the building of two novel, robust intelligent autonomous systems. It could not have been done without these talented, cooperative people, who were often working outside the responsibility of their own projects.

I would also like to thank the several reviewers of this thesis, particularly Dr. Charles Carman, Dr. Nevenka Dimitrova, and Anne Eisenberg. Their careful review and thoughtful comments have improved this thesis greatly. In addition, Diane McCoach, Vicki Diehl-Holm have provided carefully designed graphics found throughout the thesis.

I am grateful to my parents, Charles and Barbara Schroeder, who first gave me an appreciation for science and technology, and the tenacity required to be different.

Finally, and most importantly, I want to thank my husband, Steve Trovato. I am deeply indebted to him for his understanding and support during this endeavor. He somehow managed to provide the needed care for our home and our son Mark, and still be there for me.

Curriculum Vitae

Karen Irene Schroeder Trovato was born in Cold Spring, New York, on October 9th 1957. She received her first Amateur Radio License in 1966, and currently is a Technician class (WA2CVU). After graduating from Arlington High School in 1975, she attended Dutchess Community College, graduating in 1977 with an Associate of Arts degree in Mathematics. She attended the State University of New York at Stony Brook, and graduated in 1979 with a Bachelor of Science degree with dual majors in Applied Math & Statistics and Computer Science. She then joined Philips Research to work on higher level language (Modula) development. Within a year she began work on generic automated instrumentation initially focused on research prototypes of X-Ray Diffraction equipment, and built a higher-level language and small operating system to control low level electronic devices that can be reconfigured and interchangeably connected to a central system, called LABICS (Laboratory Instrumentation Control System). Almost concurrently with her employment with Philips, she entered the New York University graduate program and in 1982 graduated with a Master of Science in Computer Science. In 1985, she transferred to the Robotics Department headed by Dr. Ernest Kent. Working with Leo Dorst, she made inroads into the theory and practice of path planning for robotic systems. Karen has contributed to five issued patents, has eight more pending, and is a registered U.S. patent agent.

Karen is currently the project leader for the Geographic Information Systems project of the Software and Services Department. Current responsibilities include the prototyping and development of electronic-map based products and services that are distributed by product divisions such as Philips Media and Philips Home Services.