High Performance State Lattice Planning Using Heuristic Look-Up Tables

Ross A. Knepper and Alonzo Kelly

Robotics Institute
Carnegie Mellon University
Pittsburgh, PA, USA

rak@ri.cmu.edu, alonzo@ri.cmu.edu

***Abstract* - This paper presents a solution to the problem of finding an effective yet admissible heuristic function for A* by precomputing a look-up table of solutions. This is necessary because traditional heuristic functions such as Euclidean distance often produce poor performance for certain problems. In this case, the technique is applied to the state lattice, which is used for full state space motion planning. However, the approach is applicable to many applications of heuristic search algorithms. The look-up table is demonstrated to be feasible to generate and store. A principled technique is presented for selecting which queries belong in the table. Finally, the results are validated through testing on a variety of path planning problems.**

Index Terms – motion planning, state lattice, heuristic, nonholonomic, mobile robot

I. INTRODUCTION

The state lattice planner [1] efficiently encapsulates vehicle constraints such that they need not be considered during planning. The state lattice is a graph constructed from edges that represent continuous motions connecting discrete state space nodes. The control set which corresponds to these edges is generated according to the dynamic constraints of a particular vehicle.

A. Motivation

The state lattice planner derives its efficiency from several sources. First, infeasible motions are culled out prior to the search process. Second, the graph structure of the search space allows for elimination of redundant search steps.

Additionally, a heuristic search algorithm such as A* is needed in order to efficiently find the best path between start and goal (known as a *query*). Such algorithms require a heuristic cost estimate function, which estimates the true cost of a path. Planning performance depends on the quality of the heuristic. For some graph search problems, a simple function such as Euclidean distance is adequate.

However, no good closed-form heuristic for the state lattice is known. This is a hard problem because the planner operates in full state space, taking into account a vehicle's differential constraints. The Euclidean distance function has no knowledge of vehicle capabilities, and is thus poor heuristic because it underestimates actual path cost by violating differential constraints. Consequently, some of the most difficult queries to solve put the start and goal are in close proximity but misaligned, so that much maneuvering is necessary. The classic example of this situation is parallel parking. Another such example is shown in **Figure 1**.

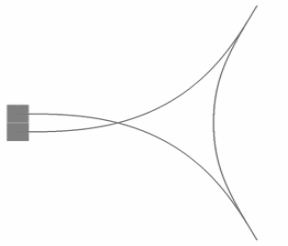

Figure 1: A Difficult Search Problem. When planning in full configuration space on a vehicle with dynamic constraints, Euclidean distance is a poor indicator of resulting path length. Here, an Ackerman-steered vehicle is asked to move sideways and turn around. The length of the solution path is approximately forty times what a Euclidean heuristic would estimate.

B. Problem Statement

Given ample computational resources, a straightforward and effective way to predict path cost is to pre-compute and store the actual costs that the planner will need, using the planner itself. Such a Heuristic Look-Up Table (HLUT) can be imagined as a large multi-dimensional array of real-valued query costs. Assuming that such a table could be generated, the invocation of the heuristic function becomes a simple table dereference. Like other heuristic cost functions, the HLUT cannot take into account obstacles or other terrain variation when precomputing queries. Since it is implausible to encode all possible worlds in a table, an obstacle-free policy must be assumed instead. For simplicity, the HLUT discussed here is designed for an Ackerman-steered vehicle. The techniques are applicable to any control set for any type of vehicle with any kind of constraints, but the need for the look-up table is much reduced in some problems. Below, the issues of generating, storing and using the HLUT for practical planning problems are explored.

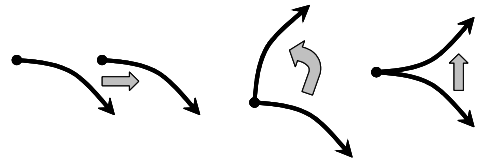

Figure 2: Symmetries of the State Lattice. Many different lattice edges and combinations of edges are equivalent under a small set of transformations. Exploitation of these properties dramatically reduces the total size of the HLUT. Symmetries shown from left to right: translation, rotation, and reflection.

C. Prior Work

An early approach to robot navigation applied gradient descent to a potential function in which the goal is a global minimum and hazards are represented by local maxima. Many such potential functions were explored, but most proved incomplete due to the problem of local minima. In [5], it was demonstrated that a potential function free of local minima, known as a navigation function, could be constructed. The HLUT is an example of such a navigation function.

Another approach to navigation involves the discretization of space so that search in the continuum can be approximated via graph search. Graph search, including heuristic search methods, has been thoroughly studied by the artificial intelligence community. The concept that a search could guarantee an optimal result while doing less work than a brute force search was introduced in [3], with the A* algorithm. Many variations on that theme have since been proposed. The connection between heuristics accuracy and efficient search has long been known. Only in recent years however has the effect of the heuristic on search complexity been understood in detail, described in works such as [7]. The problem of specifying an effective, admissible heuristic function remains a challenge, and most good heuristics are application-specific. Perhaps the most significant advance in heuristic search during the last decade was the concept of the pattern database, described in [1]. The pattern database is a look-up table indexed by a subset of the state and containing a precomputed heuristic value that reflects the cost of solving the corresponding subproblem. Many enhancements to pattern databases were proposed, including [2], [4], and [6], which each describe methods of combining multiple pattern databases to improve performance. Each of these works discusses heuristic search in the context of combinatorial puzzles like the Fifteen Puzzle and Rubik's Cube. Such problems have an intractably large but finite search space.

Reference [8] introduces the state lattice, which is an enhancement of discretized robot planning via graph search. That paper first introduced the HLUT, which is a type of pattern database in which the table is indexed by the full state. Unlike the problems that ushered in the pattern database however, the state lattice is infinite in extent, yet it can be represented effectively by a finite look-up table.

II. Constructing the Heuristic Look-Up Table

The lattice planner accommodates differential constraints, resulting in increased path complexity. In this environment, Euclidean distance proved a poor heuristic. The need for more effective heuristics was the impetus for this work.

The notion of an HLUT suggests several design considerations, including: the utility of a table of reasonable size; the amount of time required to generate it; and the policy for handling heuristic queries that are not in the table.

A. Space

The issue of limited memory can be addressed in most cases merely by exploiting symmetries in the lattice control set to eliminate duplicate information. Those symmetries include translation, rotation, and reflection (**Figure 2**). Because the discretization of the lattice is regular in position, all queries can be treated as if they originate from the origin. Only a subset of possible initial headings needs to be generated. For example, if the control set exhibits 8-axis symmetry, it is sufficient to precompute only $1/8^{th} + 1$ discrete initial headings. Finally, many goal states are redundant. For instance, with an initial heading of zero, there is symmetry about the *x*-axis. With all of these massive space savings, a useable size HLUT can be stored in 200,000 entries, consuming perhaps 2.5MB, approximately 0.001% of the size before symmetry is considered.

B. Time

To generate such a table requires solving many A* queries. An average A* lattice planning query with the Euclidean distance heuristic takes about 0.2 seconds on average (much more in the worst case), so generating 200,000 entries by the most naïve method would require more than ten hours. Fortunately, several properties of A* allow the table to be populated much more efficiently. When a search is performed, much more can be learned about the graph than just the final path length. Every time a state is expanded and put on A*'s CLOSED list, the lowest cost path to that state is known from each state along the path, so several more optimal query costs can be inserted into the HLUT. Secondarily, it is wasteful to delete the A* lists and start from scratch between queries, as these lists contain valuable data. The CLOSED list can be reused as-is, but the OPEN list is sorted according to distance estimates to the old goal state. Depending on the size of the OPEN list, it may be cheaper to delete it and begin anew or to recompute each estimate according to the new goal location. For a particular implementation, this cut-off was found to occur at approximately 150,000 OPEN states, so that several queries can be processed in a row before the lists need to be expunged. Finally, the order in which queries are populated in the HLUT has a significant effect, because the exact solutions of earlier queries can be used as a heuristic in later queries. The issue of population of the table is considered below.

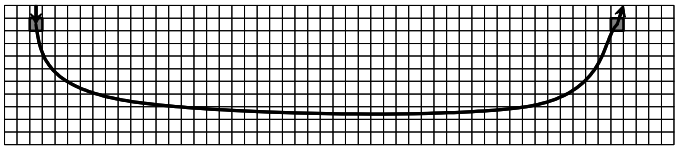

Figure 3: Euclidean Distance is a Good Approximation for Distant Queries. As two states are separated by a greater distance, parameters like curvature and heading have an increasingly insignificant effect on the overall path length.

C. *Inclusion of Queries for the Table*

The look-up table cannot be infinite in size. Therefore, some queries will occur for which no entry exists in the table. An alternate *backup heuristic* such as Euclidean distance must, of course, already be in use for purposes of generating the HLUT. It can also be used to satisfy queries missing from the table during planning. Doing so is easily justified by the fact that the Euclidean heuristic gives better approximations on more distant queries (**Figure 3**).

When generating the table, there must be some termination condition. Such a condition should be automatic, principled, and tunable according to the needs of various applications. This issue really breaks down into two separate questions. First, which queries should be included or excluded? Second, how many entries should be included?

To answer the first question, it is helpful to define the *trim ratio* as the ratio of the backup heuristic's estimate for a particular query to the true path cost which the HLUT would include for that query. There are two considerations for including a particular entry in the HLUT. As described in [4], it is more important to avoid low heuristic estimates than to retain large ones. Thus, queries with low trim ratios should be included because the backup heuristic does a poor job of estimating them (hence, they should be "trimmed" last.). Secondly, frequently needed queries should be included for the sake of efficiency, since failing over to the backup heuristic results in additional processing. **Figure 5** shows trim ratios in a slice of the HLUT where initial and final headings are zero for each query. Note that paths into dark regions (of low trim ratio) would correspond to compositions of maneuvers that result in ultimate sideways motion. These are the queries whose costs are most valuable to have stored in the HLUT.

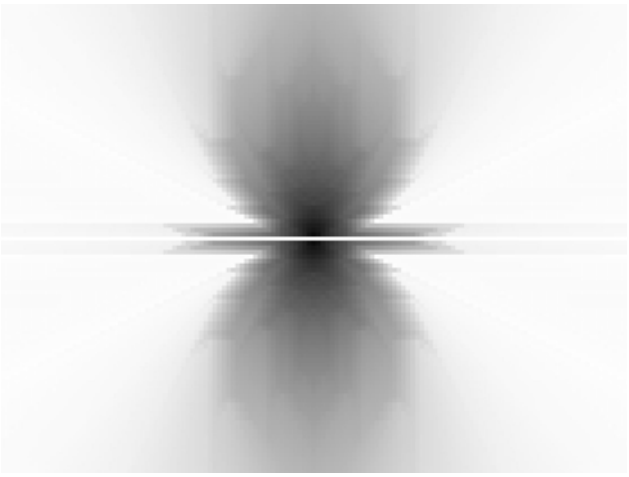

Figure 5: Visualization of HLUT. This cross-section of an example 4D HLUT shows a slice of paths from the origin to various x, y positions in which $\theta_0=\theta_f=0$. Brightness represents trim ratio, the ratio of Euclidean distance / nonholonomic path length. Dark regions correspond to low ratios.

III. Populating the Heuristic Look-Up Table

When performing A* searches to populate the look-up table, an important consideration is the order in which queries are performed. Ordering affects the ultimate selection of queries due to the inclusion of all states on the CLOSED list. In addition, the ability to reuse previously computed state varies considerably with ordering of queries. Three different approaches to HLUT population are considered here.

The simplest method of look-up table population is the naïve approach, which iterates through each entry in the table in a raster-scan fashion. The result of A* when run on each query is inserted in order. This is the slowest of the methods that were tested. It does not share significant state between sequential queries, and the most difficult queries are presented to the planner first, when no HLUT data have yet been built up to improve performance.

Parts of the table can be populated much faster by the use of Dijkstra's algorithm. In this approach, the search is run in such a way that the next node expanded in the graph is always the unexpanded node closest in cost to zero. This algorithm very quickly populates many HLUT entries, but those states are the easiest ones to reach, meaning that they generally have higher trim levels (**Figure 4**). This method produces queries that are monotone in distance from the start, but queries that are monotone in trim ratio are preferred, as discussed above. The Dijkstra's search can be terminated at an arbitrary time.

The *horizon method* is more principled in that it selects first those queries with the lowest trim levels. This algorithm maintains a HORIZON list of queries sorted by trim level, similar to the way in which A*'s OPEN list is sorted by cost. The HORIZON list is initially populated with queries in which both states are at the origin in every combination of initial and final heading (because these are the most difficult queries). Each is assigned a trim ratio of 0.0, since that is the Euclidean distance between overlapping points in a plane. During each iteration, the lowest-valued query is popped from the HORIZON list and presented to A*. Each neighboring state in the x,y-plane (with the same orientation) is considered a child of the popped state. Each child state is pushed onto the HORIZON list and sorted according to its parent's trim ratio. The ratio of the parent state is used since the exact cost of the

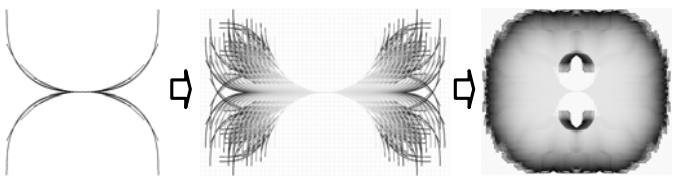

Figure 4: Populating the HLUT Using Dijkstra's Algorithm. Expansion of the lowest cost unexpanded state in the tree is an efficient way of finding the shortest distance to many states at once.

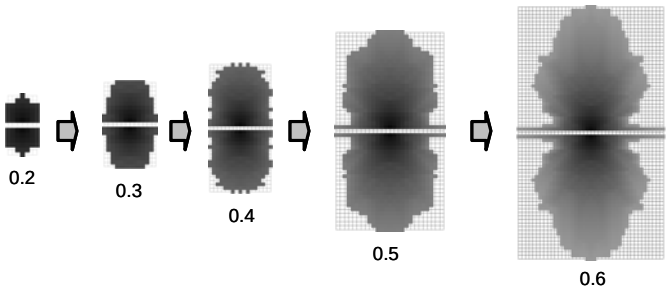

Figure 6: Growth of HLUT in the Horizon Method. As the HORIZON grows outward, queries of higher trim ratio are incorporated into the HLUT.

query is required to compute trim ratio, and that is not yet known for the child. The horizon algorithm terminates when a desired trim level is reached. The process is depicted graphically in **Figure 6**; note that symmetry can be considered to speed up the algorithm. In order for this method to find all states below a given trim level, it must be possible to draw a path outward from the origin through an arbitrary state such that the trim ratio increases monotonically along the path (**Figure 7**). Horizon is a faster algorithm than the naïve approach because it makes better use of precomputed state, but it is slower than Dijkstra's algorithm because it requires A* list resets.

The Dijkstra's search algorithms may leave gaps in the HLUT, as shown in **Figure 4**, while the two slower techniques produce dense results. A gap occurs in the table when an entry is absent but is surrounded by neighboring entries that are included. Gaps are undesirable because they result in less predictable search time among potential A* queries using the heuristic. Furthermore, a major underestimate resulting from falling back to a backup heuristic due to a gap causes a false lead for A*, which then expends a lot of unnecessary search time.

In order to populate the HLUT quickly while retaining

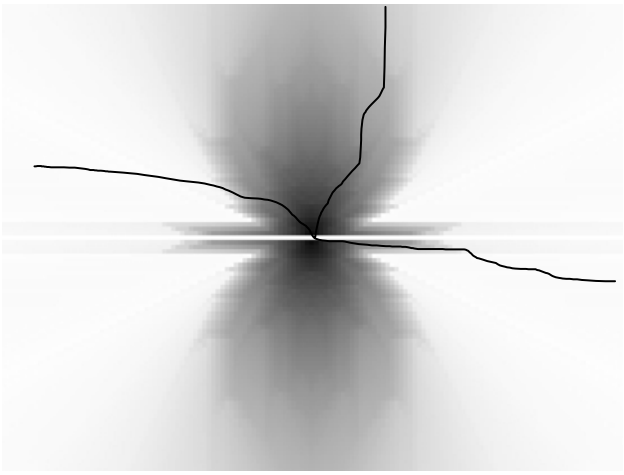

Figure 7: Monotonicity of Trim Level. Several example paths are shown in the *x*,*y*-plane, which are monotonic in trim level. Every point in the plane must have at least one such path that passes through it in order for the horizon algorithm to populate the HLUT with all queries below a certain trim level.

Trim ratio	HLUT entries	Generation time (mm:ss)
0.6	202,338	01:15
0.7	365,345	03:28
0.8	648,877	13:33

Table 1: Generating the HLUT. Through a combination of techniques, sizeable heuristic look-up tables can be generated efficiently. The runs shown here were performed on a 3 GHz Pentium 4.

desired properties, a combination of methods can be used. First, Dijkstra's algorithm fills in the majority of the HLUT. Then the gaps are filled in using the horizon method. In this case, the horizon method skips over those queries that were previously filled in to avoid duplication of effort.

Taken together, these techniques are remarkably efficient. **Table 1** shows computation time required to generate several different sizes of HLUT on an ordinary desktop computer. Each size look-up table was produced by performing Dijkstra's algorithm out to some size (ranging from 70 to 90 cells depending on the desired trim ratio), and then running the horizon algorithm to fill in the gaps.

IV. Benchmarking the Heuristic Look-Up Table

Thus far, the HLUT has been discussed in theoretical terms. The remainder of this paper is devoted to an examination of more practical considerations. Several points are demonstrated empirically through extensive testing. First, the HLUT produces a speedup in the state lattice when compared to other heuristics. Second, there is an optimal HLUT size when considering trade-offs such as memory and processor time.

A. Experimental Setup

A representative lattice control set was used in all tests. Its state space consisted of the 2D translational coordinates, heading and curvature (x, y, θ, and κ). For the sake of simplicity, curvature was constrained to be zero at each discrete state. This control set is depicted in **Figure 8**.

For these tests, a list of 10,000 random queries was generated, consisting of an initial and final state, also expressed as position, heading, and curvature. The set of queries was generated with the intent to require the planner to produce paths ranging from simple (nearly straight) to complex (e.g. parallel parking or n-point turn maneuvers) among obstacles. Each query was tested with a variety of configurations, including different obstacle fields and various start-goal relationships. Metrics for performance included time and memory consumption.

Start and goal states were produced with a random number generator that provides evenly distributed real values in a requested range. Initial positions were selected at random from the free space in order to produce the maximum possible variety of queries. The goal position was then specified by a randomly selected radius and angle specified in polar

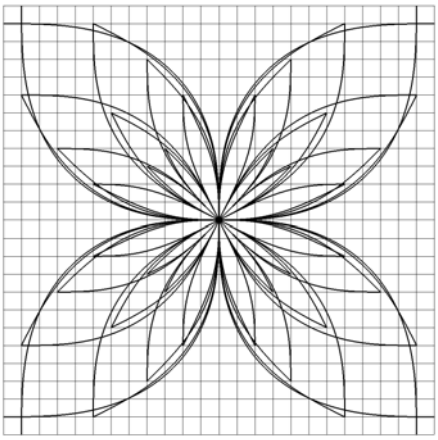

Figure 8: Lattice Control Set Used in Experiments. The state lattice control set selected for testing has 16 discrete headings, a maximum curvature of 1/8 of cell, and an average outdegree of 12, for a total of 192 curves. The straight edges cannot be seen because they are obscured by the longer curved edges.

coordinates with respect to the start, repeating this step as necessary to ensure that the goal is also in the free space. Initial and final headings were randomly selected, and curvature at end-points was constrained to be zero. Goal states were constrained to be no more than ten minimum turning radii (80 cells) from their corresponding start states when projected onto *x*, *y* space.

Each query was tested in two worlds. In both worlds, cost to traverse free space was held constant at 1 unit per cell of free space. Hence, path cost was equal to the distance traveled. The baseline case was an obstacle-free world, meaning that the HLUT was a perfect heuristic. Results were also obtained using a world with randomly placed point obstacles, shown in **Figure 9** with paths generated by two different planners. These obstacles are the size of one map cell, which in this control set corresponds to $1/8^{th}$ of the minimum turning radius. Points were generated with uniform distribution and 5% density in the plane.

B. Performance

When comparing heuristic performance, it is important to compare problems of similar complexity. In order to quantify

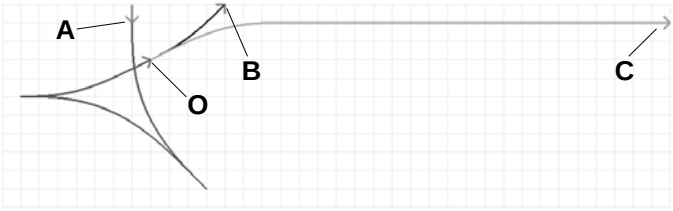

Figure 10: Absolute and Relative Query Difficulty. The difficulty of a query can be quantified in two dimensions. Each path, **A**, **B**, and **C**, starts at **O**. Query **A** is high in absolute difficulty as well as relative difficulty because it is long and has multiple cusps. **B** is simple in both measures. Query C has the same absolute difficulty as **A** (same length), but the same relative difficulty as **B** (nearly a straight line).

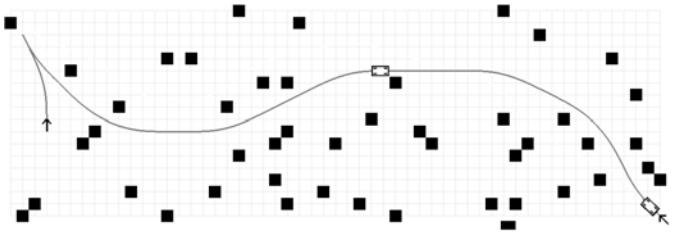

Figure 9: World with Obstacles. A portion of the world with point obstacles used in the experiments is shown here. The size of the vehicle is shown for scale.

the complexity of a particular query, the distinction is made between absolute difficulty, which is proportional to path length, and relative difficulty, which reflects how much the resulting path deviates from a straight line. **Figure 10** illustrates the two concepts. Both factors contribute to the overall resource requirements for a particular planning problem. Absolute difficulty is measured here by the path length. In the case of relative difficulty, the ratio of Euclidean distance / nonholonomic distance was used. In this scale, values near one indicate that the resulting path is nearly a straight line, while those values nearest to zero indicate that much maneuvering is necessary to reach the final pose. In the analysis below, results are shown for queries with absolute difficulty of 40 cells. This number was chosen arbitrarily for clarity of presentation. Any other absolute difficulty would have conveyed similar results.

A performance breakdown of the HLUT versus Euclidean distance heuristic is shown in **Figure 12**. Across the entire range of relative difficulty (the straightness of the solution path), the HLUT outperforms the simpler heuristic. It may seem counterintuitive that performance for the HLUT is best on those problems which are considered the hardest. This phenomenon is explained by the fact that the control set's straighter edges are shorter than the very curvy edges – a situation which naturally arises from the control set generation process described in [1]. In each query, the search using the HLUT performs no more expansions than are necessary, but more expansions are needed with the straighter paths. Since these expansions are compulsory, searches with the HLUT must be at least as fast as Euclidean in every case.

C. Sizing

The heuristic look-up table has been shown to have some benefits over simple heuristics, but that experiment relied on an HLUT with approximately 2.3 million entries consuming over 28MB. This table was generated by first using Dijkstra's method to a distance of 100 cells, followed by the horizon method with a trim ratio of 0.9. Even though this amount of memory is insignificant by today's standards, it would be desirable to devote fewer resources to the heuristic function if possible. Therefore, the impact on the planner of using smaller HLUTs was examined.

There is a basic trade-off involving HLUT size, which is defined according to its backup heuristic (Euclidean distance). An HLUT with maximum trim ratio of 0.0 is tantamount to a

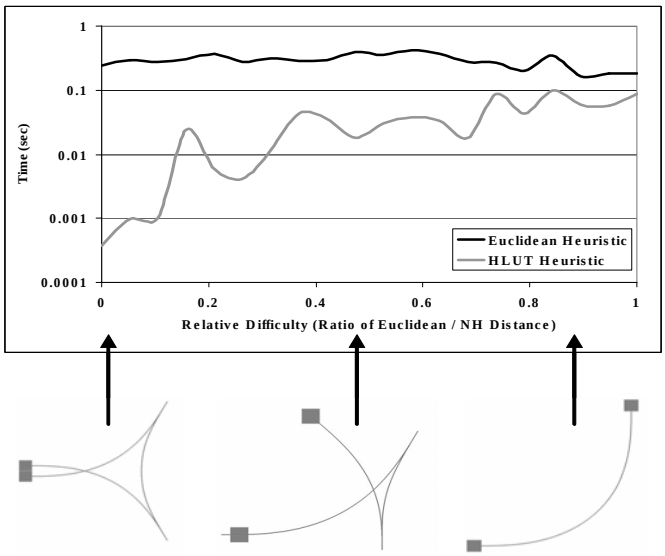

Figure 12: Performance Comparison of Heuristics. The HLUT consistently outperforms the Euclidean heuristic across all types of queries. The improvement varies from 2 to 1000x.

simple Euclidean distance heuristic function, which we have seen to perform poorly on many lattice queries because it performs unnecessary A* expansion steps. Conversely, if the HLUT is very large, few expansions are performed at the expense of increased demands for memory to store the HLUT itself. The ideal HLUT is large enough to solve queries efficiently without being so large that gains in memory saved in exploration are consumed by a monstrous HLUT.

The effect of trim level on average processor requirement can be observed by aggregating all queries for a given trim level and examining the effect of varying the HLUT size on CPU consumption. In **Figure 13**, the effect of trim level on computation time is shown. There is a clear knee in the curves both with and without obstacles, which occurs at trim level 0.8, corresponding to an HLUT size of approximately 2.5MB.

In **Figure 11**, the same data are examined from the perspective of memory usage. Here it is apparent that there is a trade-off between the amounts of memory consumed by the HLUT and because of the graph search itself. Once again, the optimal size is a trim level of 0.8.

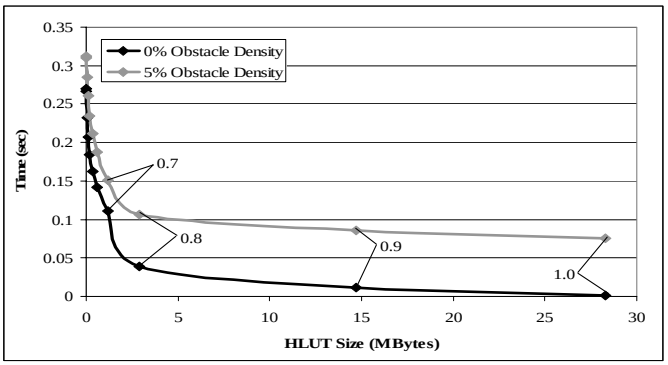

Figure 13: Time Comparison of HLUT Sizes. Upper trim levels are annotated. A* search time varies inversely with HLUT size, but there is a clear knee to the curve, with limited gains at trim levels above 0.8.

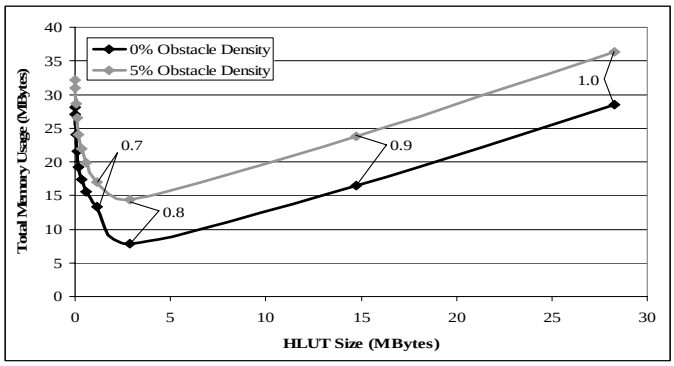

Figure 11: Memory Comparison of HLUT Sizes. Upper trim levels are annotated. Memory consumption varies with HLUT size, but a minimum occurs at trim level 0.8.

When using this new look-up table as a heuristic rather than the original one (which was ten times larger), the reduction in performance of the planner was shown to be quite insignificant, since only large-trim-ratio queries were removed. The size of the HLUT can be easily tuned as desired for any application, simply by selecting the desired trim ratio.

V. Conclusions

The state lattice combined with a heuristic look-up table has been shown to be an efficient means of generating path plans in full configuration space. The look-up table can be generated efficiently, and a useful HLUT can easily within the size of modern computer memory. Using the concept of trim ratio, entries in the HLUT may be selected for inclusion in a principled manner. This notion provides a useful knob for adjusting the trade-off between performance and space savings. While an HLUT is generated with respect to a particular vehicle's control set, the techniques are generally applicable to the creation of heuristic functions for A*.

Acknowledgment

This research was conducted at the Robotics Institute of Carnegie Mellon University under contract to NASA/JPL as part of the Mars Technology Program.

References

[1] Culberson, J.C., and J. Schaeffer, "Pattern Databases," *Computational Intelligence*, Vol. 14, No. 3, pp. 318-334, 1998.

[2] Felner, A., et al., "Compressing Pattern Databases," *National Conference on Artificial Intelligence*, 2004.

[3] Hart, P.E., N.J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, Vol. SSC-4, No. 2, July 1968.

[4] Holte, R.C., et al., "Multiple Pattern Databases," *International Conference on Automated Planning and Scheduling*, 2004.

[5] Koditschek, D.E., "Exact Robot Navigation by Means of Potential Functions: Some Topological Considerations," *International Conference on Robotics and Automation*, 1987.

[6] Korf, R.E., and A. Felner, "Disjoint Pattern Database Heuristics," *Artificial Intelligence*, Vol. 134, Issues 1-2, pp. 9-22, January 2002.

[7] Korf, R.E., and M. Reid, "Complexity Analysis of Admissible Heuristic Search," *National Conference on Artificial Intelligence*, 1998.

[8] Pivtoraiko, M., and A. Kelly, "Efficient Constrained Path Planning Via Search in State Lattices," *International Symposium on Artificial Intelligence, Robotics, and Automation in Space*, 2005.