Empirical Analysis of New Methods for Computing Minimum Cost Paths with Turn Constraints

Ali Boroujerdi and **Jeffrey Uhlmann**
Information Technology Division
Naval Research Laboratory
Washington, D.C. 20375-5320

ABSTRACT

In this paper we describe a recently developed algorithm for computing least cost paths under turn angle constraints. If a graph representation of a two or three dimensional routing problem contains $|V|$ vertices and $|E|$ edges, then the new algorithm scales as $O(|E| \log |V|)$. This result is substantially better than $O(|E||V|)$ algorithms for the more general problem of routing with turn penalties, which cannot be applied to large scale graphs. We also describe an enhancement to the new algorithm that dramatically improves the performance in practice. We provide empirical results showing that the new algorithm can substantially reduce the computation time required for constrained vehicle routing. This performance is sufficient to allow for the dynamic re-routing of vehicles in uncertain or changing environments.

Keywords: Dijkstra's algorithm, least cost paths, range searching, routing, turn constraints.

1 Introduction

Dijkstra's algorithm[4] provides an efficient method for identifying least cost paths in graphs; however, the approach is not directly able to incorporate many types of constraints which arise in practical routing applications[2]. For example:

1. The steering mechanism of a car imposes a minimum turn radius that must be respected when computing a route for the vehicle.
2. When routing ground vehicles over free terrain – not on prepared roadways – restrictions often must be placed on changes in vertical angles in order to provide sufficient clearance for the front or back end of the vehicle.
3. The routing of commercial aircraft must consider the sharpness of turns both in terms of stress on the airframe and on the comfort level of passengers.
4. In robotic jointed arm applications it is necessary to compute shortest paths through a general parameter space defined by the arm's degrees of freedom. Sharp angular motions can cause stress on joints that can lead to material fatigue, especially for mechanisms that perform repetitive tasks.

In this paper we describe an algorithm, first presented in[1], for enforcing turn radius (and related) constraints that is both efficient and general. We also present a new practical enhancement that improves the average case performance of the algorithm. Finally, empirical results are provided to demonstrate that it is possible to use these new algorithms to efficiently compute minimum cost routes which respect turn constraints.

2 Problem Description

We assume that we are given a directed graph $G = (V, E)$, where V is a set of vertices $\{v_1, ..., v_{|V|}\}$ and E is a set of edges in which e_{ij} is an element if and only if there exists an edge from vertex v_i to v_j. (An undirected graph is treated as a directed graph in which $e_{ij} \in E$ implies $e_{ji} \in E$.) Associated with each edge $e_{ij} \in E$ is a cost c_{ij}, which represents the cost for a vehicle to travel directly from v_i to v_j. If the cost associated with the routing of a vehicle is simply proportional to the total distance it travels, then $c_{ij} = d(v_i, v_j)$, where $d(\cdot, \cdot)$ is Euclidean distance. In general, however, the cost associated with each edge may be any positive real value[1].

Dijkstra's algorithm can be used to identify a sequence of edges connecting a given source vertex v_s to a given terminal, or sink, vertex v_t such that the sum of the costs associated with the edges in the path is minimum. The algorithm scales as $O(|V| \log |V| + |E|)$ and so is very efficient. Unfortunately, many types of practical routing problems impose restrictions on sequences of edges whose traversal are beyond the physical limitations of the vehicle to be routed. For example, a path containing a pair of edges requiring a vehicle to make a 120° turn might be infeasible. Because Dijkstra's algorithm selects edges independently, such restrictions on turn angle cannot be respected.

Although Dijkstra's algorithm itself cannot directly enforce turn constraints, a suitable transformation of the underlying graph G can produce a graph G' in which illegal sequences of edges are structurally eliminated. In other words, the transformation ensures that the application of Dijkstra's algorithm to G' yields a least cost path in G that does not violate the turn constraints. In general, however, G' is larger than G, so the critical question is whether the size of G' significantly increases the computational complexity of the constrained least cost path problem over that of Dijkstra's algorithm for the unconstrained problem.

As will be shown, the transformation allows Dijkstra's algorithm to be used with costs defined on pairs of consecutive edges. Exploiting this generality leads to an $O(|E||V|)$ algorithm for a more general problem in which different penalties are associated with different turns. In the next two sections we show that least cost constrained paths in two or three dimensions can be computed in $O(|E| \log |V|)$ time, which is comparable to the complexity of Dijkstra's algorithm using a binary heap for the unconstrained problem, but asymptotically less efficient than an $O(|V| \log |V| + |E|)$ implementation of Dijkstra's algorithm using a Fibonacci heap[4]. Thus, the restriction of the problem from arbitrary turn penalties to hard turn constraints leads to a dramatic reduction in computational complexity.

3 The Transformation

As mentioned, the first step in enforcing turn constraints is the transformation of $G = (V, E)$ to $G' = (V', E')$. This is accomplished as follows:

1. For each edge $e_{ij} \in E$ we define a corresponding vertex $v'_{ij} \in V'$.

[1]The case of negative costs can be handled as well, but in order to avoid unnecessary diversions we will simply consider positive costs in this paper.

2. For each pair of edges $e_{ij}, e_{jk} \in E$ which satisfy the turn constraints we define a corresponding edge $e'_{ijk} \in E'$ from vertex v'_{ij} to v'_{jk}.

It is straightforward to verify that $|V'| = |E|$ and $|E'| \leq d_1 \cdot d'_1 + d_2 \cdot d'_2 + ... + d_{|V|} \cdot d'_{|V|}$, where d_i is the out-degree and d'_i is the in-degree of vertex v_i in G. In practice, however, G' does not need to be explicitly generated. In fact, the only auxiliary data structures that are required are range search structures, one corresponding to each vertex $v_j \in V$, for efficiently determining which edges e_{jk} are within the constrained angular ranges of a given edge e_{ij}. For vertices corresponding to points in two and three dimensions, these range search structures ensure that for each vertex $v'_{ij} \in V'$ it is possible to determine all $e'_{ijk} \in E'$ in $O(\log |V| + m)$ time, where m is the number of edges emenating from v'_{ij}, rather than $O(|V|)$. Specifically:

1. In two dimensions the required range search structure for vertex v_i can be a balanced binary tree of d_i angles, one for each vertex adjacent to v_i. These angles can be determined from an arbitrary zero angle about the spatial point corresponding to vertex v_i (with obvious care taken to accommodate angular query ranges that straddle the zero angle). Thus, the query time is $O(\log d_i + m)$, the preprocessing time is $O(d_i \log d_i)$, and the space requirement is $O(d_i)$.
2. In $k > 2$ dimensions the range search for vertex v_i is defined by constraints on $k-1$ angles. As will be discussed, such queries can be satisfied in $O(\log^{k-2} d_i + m)$ with $O(\log^{k-2} d_i)$ time required for the deletion of each retrieved element in the query range[3]. The preprocessing time and space requirement is $O(d_i \log^{k-2} d_i)$. Thus the time complexity in three dimensions is the same as for two dimensions.

The total preprocessing time required to support the implicit generation of G' in 2 dimensions is $O(\sum_{i=1}^{|V|} d_i \log d_i)$, which is $O(|E| \log |V|)$ since $E = \sum_{i=1}^{|V|} d_i$. For $k > 2$ dimensions the total preprocessing time is $O(|E| \log^{k-2} |V|)$. The space requirement in k dimensions is $O(\sum_{i=1}^{|V|} d_i \log^{k-2} d_i)$, which is $O(|E| \log^{k-2} |V|)$.

4 Dijkstra's Algorithm on the Transformed Graph

The following steps constitute the application of Dijkstra's algorithm to G':

// The algorithm assumes $v_s \neq v_t$, where v_s is the source and v_t is the sink

(1) $Q \leftarrow$ *null* *// Q is a priority queue*

(2) For each v_j adjacent to v_s, insert v'_{sj} into Q with priority c_{sj}

(3) While Q is not empty do:

- **(3.1)** Retrieve and delete minimum priority vertex v'_{ij} from Q; let p be the priority of the vertex
- **(3.2)** If v_j is the sink vertex in G, then stop and report p as the cost of the minimum cost path
- **(3.3)** For each vertex v'_{jk} in the adjacency list of v'_{ij} do: */* The adjacency list of v'_{ij} can be obtained by performing a range search in the adjacency list (range search structure) of v_j in G and identifying those edges $e_{jk} \in E$ that do not violate the turn constraints with respect to edge $e_{ij} \in E$ */*
  - **(3.3.1)** Insert v'_{jk} into Q with priority $p + c_{jk}$
  - **(3.3.2)** Delete v_k from the adjacency list of v_j in G */* No other path containing $e_{jk} \in G$, hence $v'_{jk} \in G'$, can have less cost */*

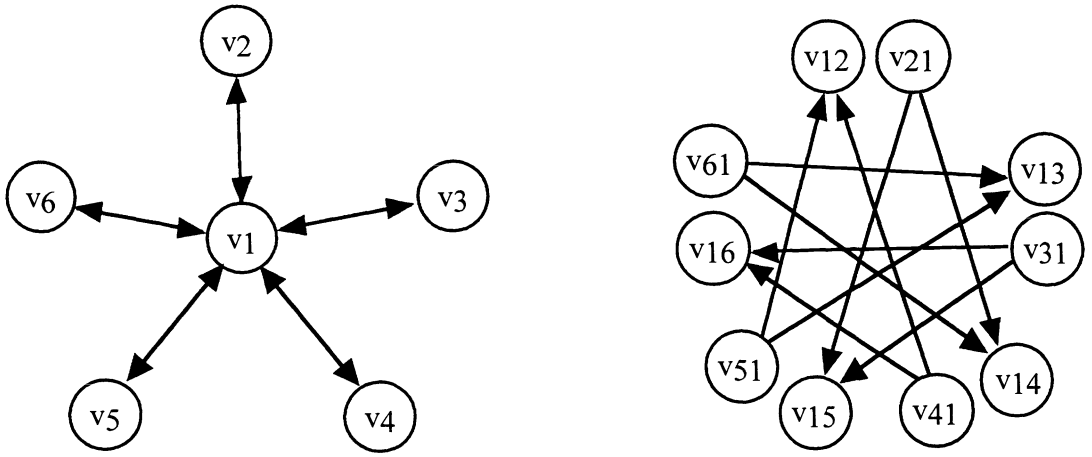

Figure 1: On the left is a subgraph of G. On the right is the corresponding subgraph of the transformed graph G'.

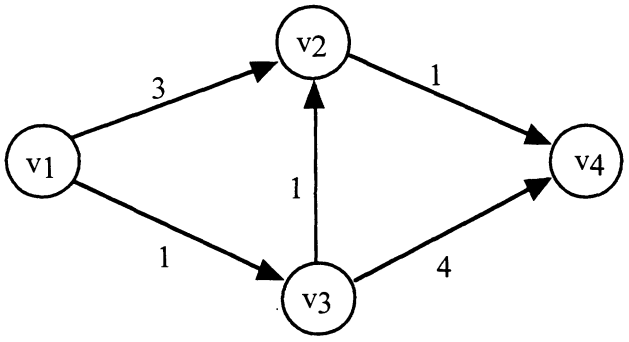

Figure 2: Routing under the constraint that turn angles must be less than90 degrees leads to aleast cost path consisting of v1 to v2 to v3. The least cost unconstrained path is v1 to v3 to v2 to v4.

Although the above steps only determine the cost of the least cost path, it is straightforward to obtain the path itself. The path in G can then be constructed by replacing each vertex v'_{ij} in the least cost path in G' with its corresponding edge $e_{ij} \in G$.

The computational complexity of the above steps can be evaluated as follows. The total number of times loop (3) can be executed is bounded by $|V'| = |E|$. Loop (3.3) is also executed at most $|E|$ times since an edge reported by a range search query is deleted before subsequent queries. Therefore:

- The overall complexity of the priority queue operations in steps (3.1) and (3.3.1) is $O(|E| \log |V|)$.
- The overall complexity of the range searching performed in step (3.3) is $O(|E| \log |V|)$ for two dimensions and $O(|E| \log^{k-2} |V|)$ for $k > 2$ dimensions.
- Each deletion in step (3.3.2) can be accomplished in $O(\log d_j + m)$ time for two dimensions (deletion from a 1-dimensional binary tree) for an overall complexity of $O(|E| \log |V|)$. In $k > 2$ dimensions the deletion complexity is $O(\log^{k-2} d_i)$ for an overall complexity of $O(|E| \log^{k-2} |V|)$ using the semi-dynamic data structure of[3] in which only deletions (no insertions) are supported. *It must be reiterated that k refers to the spatial dimension; the range searching occurs in $(k-1)$-dimensional angular space.*

From the above analysis we can conclude that the overall scaling of the application of Dijkstra's algorithm to G' solves the turn constrained routing problem in $O(|E| \log |V|)$ time for two and three dimensions and $O(|E| \log^{k-2} |V|)$ time for $k > 3$ dimensions. The key to achieving this complexity is the recognition that each range search data structure can be constructed in batch and then needs only to support deletion and query operations. This permits the use of a highly efficient semi-dynamic variant of the augmented range tree[3].

We note that although only angular constraints have been discussed, constraints on other variables, such as changes in speed, also can be accommodated within the same framework. The only caveats are (1) efficient range search structures only support query regions defined by independent ranges on each variable, and (2) each additional variable incurs an extra $\log |V|$ factor on the overall complexity.

5 Implementation

It is interesting that the physically realizable cases of two and three dimensions have the same complexity. Complexity analysis does not necessarily tell the whole story, of course, so we provide the following implementation description demonstrating that the algorithm can be implemented simply and efficiently without any explicit generation of the transformed graph G':

// Here v_s and v_t can be the same vertex

(1) $Q \leftarrow$ *null*; $T \leftarrow$ *null* *// Q is the priority queue and T is the shortest path tree*

(2) Insert s into Q with priority 0

(3) Insert s into T *// s is the root of T hence no parent*

(4) While Q is not empty do:

  (4.1) Retrieve and delete minimum priority vertex v_j from Q; let p be the priority of this vertex and v_i be its parent in T

  (4.2) If v_j is the sink, then the minimum cost path is obtained (in reverse order) by following the parent pointers from v_j to v_s in T; p is the cost of the path

(4.3) For each vertex v_k in the adjacency list (range search structure) of v_j such that the angle $v_i v_j v_k$ does not violate the turn constraint do:

(4.3.1) Insert v_k into Q with priority $p + c_{jk}$

(4.3.2) Insert v_k into T and set its parent pointer to v_j

(4.3.3) Delete v_k from the adjacency list of v_j

The seemingly large amount of bookkeeping suggested by the description of the constraint enforcement transformation is completely avoided in the above implementation. In particular, although each vertex v_i may appear multiple times in the priority queue (once for each incoming edge), the different occurences are uniquely distinguished in the shortest path tree. Even more surprising is the fact that the implementation is actually *simpler* than Dijkstra's algorithm because there are no priority queue decrease-key operations; thus, a simpler, more efficient heap structure can be used.

The only component of the algorithm that is not trivial to implement is the range search structure for three and higher dimensions. In two dimensions the 1-dimensional range search structure on angles is simply an ordinary binary tree. However, in three and higher dimensions the structure is somewhat more sophisticated[2]. Fortunately, each structure can be constructed in a batch process that involves little more than several sorting operations. Because the structure only needs to be semi-dynamic, the deletion algorithm performs very little maintenance on the data structure and is relatively straightforward.

6 A Practical Enhancement

The algorithm discussed thus far is efficient in the worst case, but there is some room for improving the average case performance. Recall that Dijkstra's algorithm successively expands its search from the vertex v on the priority queue whose path from the source has minimum cost. We will denote the cost of this path as $g(v)$. Now, if we can obtain a lower bound estimate on the cost of the path from v to the sink, which we will denote as $h(v)$, then it is possible to define a more efficient expansion criterion. Specifically, it can be proven that expanding the vertex v on the queue that minimizes the function $f(v) = g(v) + h(v)$ will yield an optimal solution as long as $h(v)$ satisfies the lower bound condition[5]. The important fact is that any $h(\cdot) > 0$ will reduce the number of vertices examined by Dijkstra's algorithm. The problem of course is to identify a suitable h-function.

In general applications of Dijkstra's algorithm it is impossible to obtain a nontrivial lower bound on the cost of the path from a given vertex to the sink. However, when Dijkstra's algorithm is applied to find constrained paths, as it is in the algorithm developed in this paper, it is always possible to use the cost of the unconstrained path as a lower bound. This simply requires a preprocessing step in which Dijkstra's algorithm is used to find the least cost unconstrained path from each vertex to the sink. The cost of the unconstrained path from each vertex v to the sink then becomes the value of $h(v)$.

The number of vertices examined during the course of the algorithm is strongly affected by the quality of the h-function estimates. If they are perfect, then the the number of vertices examined will be proportional to the number of edges in the minimum cost path. If $h(v) = 0$ is assumed for all v, which is implicitly what is done in conventional Dijkstra's algorithm, then almost every vertex will be examined. Thus, the h-function can be expected to provide less benefit for graphs in which least cost constrained paths deviate greatly from the least cost unconstrained paths.

[2]In most practical 3-dimensional routing problems the discretization of the space produces graphs with substantially fewer edges in the vertical plane than in the horizontal plane. If the discretization in the vertical plane is sufficiently low, it may be more efficient to simply perform a 1-dimensional range search on the horizontal angles and then return only those elements which also satisfy the vertical angular constraint.

7 Test Results

In this section we provide test results for several approaches to routing with turn constraints:

Version 0 *Dijkstra's algorithm*: This benchmark provides the time required by Dijkstra's algorithm when turn constraints are ignored. No optimal algorithm that respects the constraints can be faster, so this represents a lower bound baseline for comparison.

Version 1 *Ad hoc (greedy) checking of turn angles*: This version is simply Dijkstra's algorithm altered so that the shortest path tree is never expanded in such a way as to violate the turn constraints. This naive approach can yield highly suboptimal routes or may even fail to find a solution altogether. It is included to show the difference in computation times between commonly used greedy approaches and the algorithms described in this paper for generating optimal solutions.

Version 2 *Dijkstra's algorithm applied to the transformed graph G'*: This version enforces turn constraints and yields optimal solutions, but it does not use efficient range search data structures to bound the worst case computational complexity. It is included to show the efficiency of the unadorned application of Dijkstra's algorithm to G' to obtain optimal constrained routes.

Version 3 *Same as Version 2 but with efficient range searching*: This version enforces turn constraints optimally and is efficient in the worst case. It is included to show how the range search structures improve the performance of Version 2.

Version 4 *Same as version 2 but uses the h-function enhancement*: This version provides efficient average-case performance, but it does not use efficient range search data structures to bound the worst case computational complexity. It is included to show how the h-function enhancement improves the average case performance of Version 2.

Version 5 *Combination of Versions 3 and 4*: This version efficiently generates optimal constrained routes in both the average and worst cases. It is included to show the average case performance achieved by the combination of efficient range search structures and the h-function enhancement.

We perform a battery of tests in which we vary the following parameters:

1. **Grid size:** We examine the effects of problem size by varying the size of the graph/grid. Specifically, we consider 2D grids having 100 grid cells (or vertices) in width. The length of the grid is varied from 100 to 800 grid cells, with the source and sink vertices on opposite sides. Thus for Euclidean edge weights, the cost of the shortest path will be roughly the length of the grid.

   Each vertex is connected by edges to vertices within a 3 grid cell neighborhood. Redundant edges are not included, e.g., the edge connecting vertex (i, j) to vertex $(i+2, j+2)$ is not included because it is equivalent to a pair of smaller edges.

2. **Edge cost range:** One of the most flexible ways to model the difficulty of routing problems is to parameterize the edge costs to vary between values corresponding to Euclidean distance and values which are essentially random. This can be accomplished by randomly generating edge costs from the interval $[d, r \cdot d]$, where d is the Euclidean length of the edge and $r \geq 1$ is the parameter determining the "convolutedness" of the cost topography. Setting $r = 1$ yields minimum cost paths which are also minimum Euclidean distance paths, while setting $r \to \infty$ yields minimum cost paths that are essentially drawn at random from the set of all possible paths. We consider the values 2, 4, and 8 for r to model moderate, difficult, and very difficult routing problems, respectively.

3. **Maximum turn angle:** This is the critical constraint considered in this paper. We consider maximum turn angles of 30, 60, and 90 degrees.

In the following three sections we provide empirical results showing the performance of the constrained routing algorithms described in this paper on problems of varying difficulty. The first test considers the highly restrictive constraint requiring the router to find the least cost path having no turns greater than 30 degrees. The subsequent two tests then consider the less restrictive turn angle constraints of 60 and 90 degrees respectively.

7.1 Test 1 - Turn Constraint of 30 Degrees

The following three tables provide the computation times (in seconds) of the various routing modules applied to various sized graphs. The first table contains results for graphs in which the cost for each edge is drawn randomly and uniformly from the interval $[d, 2d]$, where d is the length of the edge. The subsequent two tables contain results for increasingly more difficult problems in which the edge costs are drawn from the intervals $[d, 4d]$ and $[d, 8d]$, respectively.

Table 1.1 - Cost range [d,2d] (Max turn angle 30 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.3
V.2: Dijkstra on G'	33	56	120	240
V.3: V.2 w/ range search	4.7	11	21	40
V.4: V.2 w/ h-function	0.3	1.4	17	100
V.5: Combo of V.3 & V.4	0.3	0.8	4.8	25

Table 1.2 - Cost range [d,4d] (Max turn angle 30 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	1.9
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.5
V.2: Dijkstra on G'	32	55	120	236
V.3: V.2 w/ range search	5.5	10	22	50
V.4: V.2 w/ h-function	0.7	7.2	54	187
V.5: Combo of V.3 & V.4	0.4	2.2	12	42

Table 1.3 - Cost range [d,8d] (Max turn angle 30 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.3	2.5
V.2: Dijkstra on G'	31.7	58	121	255
V.3: V.2 w/ range search	5.5	12	21	46
V.4: V.2 w/ h-function	1.3	19	80	209
V.5: Combo of V.3 & V.4	0.6	4.7	17	45

The principal conclusions that can be drawn from the above tables are:

1. Even for this case in which turns are constrained to be less than 30 degrees, the optimal constrained route can be found relatively efficiently. Specifically, the optimum constrained algorithm is only 20 to 25 times slower than (unconstrained) Dijkstra's algorithm on the largest and most difficult examples. On the smaller problems the difference is even less.

2. The h-function enhancement provides decreasing benefit with increasing grid size. This is to be expected because the number of possible paths increases geometrically with grid size.

3. The benefit of the h-function enhancement is also negatively impacted by increasing the edge cost range. This is because wide edge cost ranges tend to produce more circuitous constrained routes which deviate substantially from the optimum unconstrained route.

The real bottom line is that the combination of the optimal worst case algorithm and the practical enhancement provides solutions in 45 seconds for the most difficult problem. As is shown in the following sections, however, more realistic turn constraints lead to much greater computational efficiency.

7.2 Test 2 - Turn Constraint of 60 Degrees

The following three tables provide test results analogous to those of Test 1 but with the turn constraint relaxed from 30 degrees to 60 degrees.

Table 2.1 - Cost range [d,2d] (Max turn angle 60 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	1.9
V.1: Ad hoc / greedy	0.3	0.6	1.1	2.3
V.2: Dijkstra on G'	29	55	112	218
V.3: V.2 w/ range search	4.7	9.8	21	41
V.4: V.2 w/ h-function	0.2	0.4	1.0	2.5
V.5: Combo of V.3 & V.4	0.3	0.6	1.2	2.5

Table 2.2 - Cost range [d,4d] (Max turn angle 60 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.4
V.2: Dijkstra on G'	30	55	111	220
V.3: V.2 w/ range search	5.2	10	20	40
V.4: V.2 w/ h-function	0.2	0.7	7.8	41
V.5: Combo of V.3 & V.4	0.3	0.7	3.1	13

Table 2.3 - Cost range [d,8d] (Max turn angle 60 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.5
V.2: Dijkstra on G'	29	52	111	221
V.3: V.2 w/ range search	4.9	10	21	42
V.4: V.2 w/ h-function	0.2	3.4	32	127
V.5: Combo of V.3 & V.4	0.3	1.3	8.2	31

Most of the same conclusions can be drawn from the above results as were drawn from the results of Test 1. The main difference is that the absolute computation times have been reduced significantly.

7.3 Test 3 - Turn Constraint of 90 Degrees

The following three tables provide test results analogous to those of the previous tests but with the turn constraint relaxed to 90 degrees.

Table 3.1 - Cost range [d,2d] (Max turn angle 90 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	0.9	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.3
V.2: Dijkstra on G'	27	50	101	207
V.3: V.2 w/ range search	6.7	9.7	19	41
V.4: V.2 w/ h-function	0.2	0.4	0.9	1.8
V.5: Combo of V.3 & V.4	0.3	0.5	1.2	2.5

Table 3.2 - Cost range [d,4d] (Max turn angle 90 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.5
V.2: Dijkstra on G'	27	50	105	208
V.3: V.2 w/ range search	4.8	10	20	42
V.4: V.2 w/ h-function	0.2	0.5	1.0	1.9
V.5: Combo of V.3 & V.4	0.3	0.6	1.2	2.5

Table 3.3 - Cost range [d,8d] (Max turn angle 90 degrees)

	100x100	100x200	100x400	100x800
V.0: Straight Dijkstra	0.2	0.5	1.0	2.0
V.1: Ad hoc / greedy	0.3	0.6	1.2	2.3
V.2: Dijkstra on G'	26	50	110	211
V.3: V.2 w/ range search	5.3	11	21	41
V.4: V.2 w/ h-function	0.2	0.5	2.0	5.9
V.5: Combo of V.3 & V.4	0.3	0.6	1.6	3.7

The above results show many of the same scaling relationships among the different variables as were revealed in the previous test cases. The principal effect of allowing 90 degree turns is that the absolute computation times are significantly reduced. These results for maximum 90 degree turn angles are probably the most relevant to practical constrained routing applications.

8 Conclusions

We have described an algorithm for an important practical variant of the least cost path optimization problem. Specifically, we have presented an $O(|E| \log |V|)$ approach in two and three dimensions for computing least cost paths incorporating constraints on turn angles. We have also shown that the algorithm generalizes to $k > 3$ dimensions with complexity $O(|E| \log^{k-2} |V|)$.

The complexity in two and three dimensions is comparable to unconstrained Dijkstra's algorithm and is substantially better than the $O(|E||V|)$ algorithm for the more general problem of routing with turn penalties. More

specifically, in practical routing problems $|E|$ is $O(|V|)$, which implies that the complexity for respecting general turn penalty costs is $O(|V|^2)$, whereas our algorithm has complexity $O(|V| \log |V|)$. Much of the computation time for the constrained algorithm is spent on overhead for the construction of the search structures. In systems in which routing is performed repeatedly on the same graph, this computational overhead is incurred only once.

Because traditional constrained routing approaches scale quadratically with graph size, they can only be applied in off-line applications. The $O(|V| \log |V|)$ scaling of the new algorithm, however, reduces the computational time for constrained routing in large graphs from hours to seconds. This performance is sufficient to allow, e.g., an autonomous vehicle to dynamically re-route as it simultaneously senses its environment. Such a capability can provide dramatic improvements in the overall quality of the traversed routes, as well as permitting much greater tasking flexibility over vehicles operating in novel environments.

In addition to the improved computational complexity, we have also described a practical enhancement to the algorithm that leads to substantial reductions in computation time for realistic scenarios. More specifically, the enhancement will tend to improve performance in all but the most extreme cases in which the turn constraints lead to highly circuitous routes. The test results demonstrate the combined benefits of the improved algorithmic complexity of the new algorithm and the improved practical performance provided by the enhancement.

In summary, we believe we have developed a solution to a class of constrained routing problems which can be applied to very large scale graphs. This class of problems covers a wide variety of important land and air routing applications which previously could only be addressed using heuristic methods.

REFERENCES

[1] A. Boroujerdi and J. Uhlmann, "An Efficient Algorithm for Computing Least Cost Paths with Turn Constraints," (submitted to *Information Processing Letters*, 1997).

[2] A. Boroujerdi, J. Uhlmann, M. Zuniga, "Constrained Routing for Strike Aircraft," *NRL Review*, 1997.

[3] K. Mehlhorn and S. Naher, "Dynamical Fractional Cascading," *Algorithmica*, 5:215-241, 1990.

[4] B. Moret, H. Shapiro, *Algorithms from N to NP*, Benjamin/Cummings, US (1990).

[5] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, Prentice-Hall, 1995.