Geometric Similarity Metrics for Case-Based Reasoning

Karen Zita Haigh
khaigh@cs.cmu.edu
(412) 268-7670

Jonathan Richard Shewchuk
jrs@cs.cmu.edu
(412) 268-3778

School of Computer Science
Carnegie Mellon University
Pittsburgh PA 15213-3891

Abstract

Case-based reasoning is a problem solving method that uses stored solutions to problems to aid in solving similar new problems. One of the difficulties of case-based reasoning is identifying cases that are relevant to a problem. If the problem is defined on a geometric domain — for instance, planning a route using a city map — it becomes possible to take advantage of the geometry to simplify the task of finding appropriate cases. We propose a methodology for determining a set of cases which *collectively* forms a good basis for a new plan, and may include *partial* cases, unlike most existing similarity metrics. This methodology is applicable in continuous-valued domains, where one cannot rely on the traditional method of simple role-substitution and matching. The problem of identifying relevant cases is transformed into a geometric problem with an exact solution. We construct two similar algorithms for solving the geometric problem. The first algorithm returns a correct solution, but is prohibitively slow. The second algorithm, based on the use of a Delaunay triangulation as a heuristic to model the case library, is fast, and returns an approximate solution that is within a constant factor of optimum. Both algorithms return a good set of cases for geometric planning. We have implemented the second algorithm within a real-world robotics path planning domain.

Introduction

Our wish list for the ideal path planner for real-world navigation includes several key properties: speed; the capacity to reuse knowledge learned while forming and executing plans; and the ability to react to changing conditions.

Case-based reasoning (CBR) [12; 10] is a planning method that seems appropriate for path planning, because it enables a planner to reuse *cases* — solutions to previous, similar problems — to solve new problems. A CBR planning system has to identify cases that may be appropriate for reuse and then modify them to solve the new problem. Identifying relevant cases is done by the use of a *similarity metric*, which estimates the similarity of cases to the problem at hand. An ideal metric might:

- take into account the relative desirability of different cases;
- suggest how multiple cases may be ordered in a single new solution; and
- identify which part(s) of a case are likely to be relevant.

Finding a similarity metric that is both effective and fast is a difficult task for the researcher. It is sufficiently difficult that many existing CBR systems identify neither multiple cases nor partial cases at all. This is unfortunate, because in many domains, and path planning in particular, the transfer rate of past experience is considerably reduced if only complete plans can be reused.

The route-planning system most similar to ours, ROUTER (developed by Goel *et. al* [7]), cannot reuse partial cases: it can find *partially matched* cases, but then the only modification performed is to add new steps at the beginning or the end of the case; it does not remove any unnecessary steps.

The domain of path planning offers both simplifications and added difficulties to the CBR researcher. On the one hand, geometric information can be used to simplify the task of identifying good cases. On the other hand, path planning occurs in a continuous-valued state space, and the traditional methods of role-substitution and matching on literals are not easily applied.

As a heuristic to exploit geometric information and confront the pitfalls of continuous domains, we transform the path planning problem into a mathematically precise geometric problem whose solution is expected to produce a good set of cases for planning. This method is not restricted to inherently geometric domains, but is applicable in any domain which can be reduced to a continuous two-dimensional representation. For instance, Broos and Branting [1] determine the desirability of alternative states in a state space by geometrically composing multiple training examples. However, their method only composes entire examples, and does not permit partial information reuse.

The geometric problem is to find the route of lowest cost between two points in an idealized map that reflects the relative desirability of different cases. We have developed two algorithms to solve this problem. The first finds a route of minimum cost, but may take a prohibitive amount of time to execute. The more practical second algorithm finds an approximation to the optimum route, and quickly generates a good set of cases for the planner. Both algorithms have the advantage that their internal representation of the problem can be quickly updated and reused as learning occurs and conditions change.

After a set of cases have been found, a planner constructs a new path with help from the cases identified by the geometric algorithm. These cases are returned with an ordering, and it may be possible for the planner to begin execution of the early part of the path before the remainder of the path is planned. Once the plan is complete, it can be subdivided into new cases which are added to the library. The geometric representation is also updated accordingly.

We have implemented the faster geometric algorithm within the context of the PRODIGY planning and learning system [3]. Unfortunately, the usual similarity metric used by PRODIGY is not appropriate for the continuous path planning domain. Although domain independent, the metric relies on role-substitution and matching, assuming that specific constants can be parameterized and rematched to other instances at retrieval time. However, in domains with continuous values (in which literals describe distance, time, or cost), it is hard to find a representation that satisfies this assumption.

The geometric similarity metric we propose herein overcomes these difficulties, and PRODIGY is well-suited to deal with the resulting information, which may include multiple and partial cases, as well as a suggested ordering for them. It is the planner's job to knit this information together into a plan, taking into account details such as one-way streets and illegal turns that cannot be resolved by the geometric algorithm.

Preliminary results show that our similarity metric markedly improves plan quality and reduces total planning time [9]. We illustrate the use of the algorithm within the domain of robotics path planning using a complete map of Pittsburgh containing over 18,000 intersections and 5,000 streets comprised of 25,000 segments [2].

Problem Representation

When PRODIGY generates a plan, the solution trace is either stored as a case, or broken into several pieces and stored as a series of cases. The representation of each case includes a detailed description of the situations encountered at execution time, including explanations of any errors that occurred and all replanning that was done to correct the problems.

For our path planning domain, each case is also approximated by a line segment in a two-dimensional graph, and line segments are allowed to intersect only at their endpoints. This graph acts as the index into the case library. When PRODIGY generates a plan that intersects existing segments, the plan and the cases it intersects are broken into smaller cases at the intersection points to maintain these constraints. The resulting graph, which we call the *case graph*[1], is illustrated in Figure 1.

Figure 1a is a map. Figure 1b shows the abstract manner in which paths are stored in the CBR indexing file. Note that Case 20 oversimplifies the path, but the bend in the road would not change the final routing, so this abstraction is acceptable. If any of these segments are needed for reuse, the similarity metric will estimate the best place *within* each segment to start using the case. (Note that ROUTER is unable to do this).

[1] The case graph is identical to what computational geometers call a *planar straight line graph.*

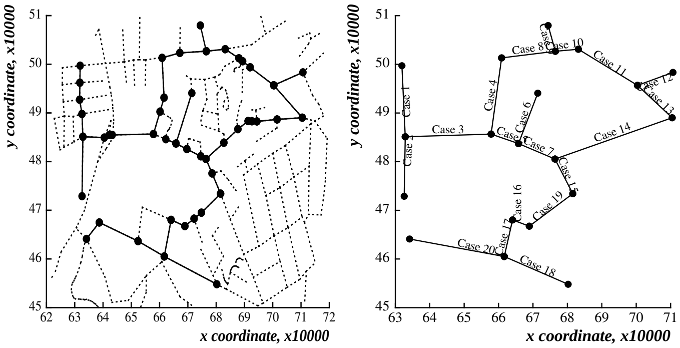

Figure 1: *(a) Map. Solid line segments are previously visited streets; dotted segments are unvisited streets. (b) Case graph representation of map. Straight line approximations are used to create the representation by cases. Several case segments may together describe one route, and several streets may be contained in one case segment.*

We call the segments of the case graph *case segments*. If the case graph contains intersecting segments, then these segments must be split at their intersection points, and a vertex must be added at each intersection. This process can be done incrementally when each case is added to the library.

The Geometric Problem

Suppose we undertake to plan a route on our map from some initial location i to some goal location g. Although we want to reuse cases, we prefer unexplored territory to long, meandering routes. It is important to find a reasonable compromise between staying on old routes and finding new ones. Hence, we assign each case an *efficiency* value β, which is a rough measure of how much a known case should be preferred to unexplored areas. In the map domain, the efficiency of a particular case might depend on such factors as road conditions or traffic. The efficiency satisfies $\beta \geq 0$, and may vary from case to case; low values of β correspond to more desirable streets. Values of $\beta > 1$ correspond to undesirable streets.

We assume that the *cost* of traversing an unknown region of the plane is equal to the distance travelled, while the cost of traversing a known case is equal to β times the distance travelled. Define a *route* to be a continuous simple path in the plane. A route may include several case segments (or parts thereof), and may also traverse unexplored regions. Assign each route a cost which is the sum of the costs of its parts.

The problem of finding a good set of cases is reduced to a geometric algorithm in which one finds an *optimum route* (that is, a route with the lowest cost) from the initial vertex i to the goal vertex g in the case graph. The case segments found in the shortest route are returned to the planner, which creates a detailed plan using the cases for guidance.

Note that it is not appropriate to use this geometric method at the finer-grained level (the entire map) for several reasons. For a detailed discussion of this point, see Haigh and Veloso [9]. In particular:

- The number of streets is much larger than the number of cases, and would create substantially more work;
- Anomalies such as one-way streets, illegal turning direc-

tions, and overpasses are not easy to represent geometrically;
- Use of large cases improves efficiency by taking advantage of prior experience; and
- We wish to interleave planning with execution.

Planning with PRODIGY/ANALOGY

Once identified, the cases are returned to the case reuse algorithm described by Veloso [14] within the framework of PRODIGY. PRODIGY/ANALOGY is one of the few systems that allow flexible reuse of partial cases, as well as the merging of multiple cases.

In its normal state of execution, PRODIGY uses a nonlinear planner to find solutions by examining and modifying its domain knowledge, encoded in the form of operators and state description predicates. When the CBR module of the system is included in the problem solving cycle, PRODIGY uses a case library to guide the search phase, and attempts to reuse prior successful decisions and avoid failed ones.

PRODIGY/ANALOGY's first action before making any decisions about a new problem is to request a set of cases from the case library. At each decision point in the new problem, it examines the decision taken in the case(s), and if the justifications for that action hold in the present context, makes the same decision. When old justifications do not hold, it uses domain knowledge to insert or ignore decisions.

For example, if a case indicated that the next move was to cross a bridge, PRODIGY/ANALOGY would check that it believes the bridge is still functional before committing to the same move. If the bridge was unusable, it would find an alternate route over the river. As a second example, if a case had subgoaled on finding a key to a locked door, but the key were already available in the new situation, then PRODIGY/ANALOGY would ignore all the steps in the case that were directly related to finding the key.

At the end of any problem solving episode, successful solutions are added to the case library. Associated with each case is a solution trace, which provides a detailed description of the plan and justifications for all decisions made.

If the case library is empty or there are no cases applicable in the new situation, the system will use its current domain knowledge to construct a viable solution, and incrementally expand the case library. Similarly, in the rare situation that PRODIGY/ANALOGY fails to generate a solution based on some set of cases, it can backtrack over those decisions and use domain knowledge to find a viable solution for the new problem if one exists.

An Exact Algorithm to Find an Optimum Route

To solve the geometric problem, we transform it into a graph problem. Define G to be the complete graph (i.e., every pair of vertices is connected by an edge) whose vertices are the vertices of the case graph (i.e., the endpoints of the cases), plus the initial and goal points (i and g). Each edge (u, v) of G is assigned a cost which represents the cost of a good simple route between u and v. Given this graph, a slow but exact algorithm is to apply Dijkstra's shortest path algorithm [5] to find the lowest-cost path from i to g in G.

The difficult part of the algorithm is to compute each edge cost. Where two vertices are connected by a case segment, the edge cost is simply β times the Euclidean distance between the vertices. The calculation is not always so simple. To see why, consider Figure 2. Imagine that $\overline{yz}$ is a highway,

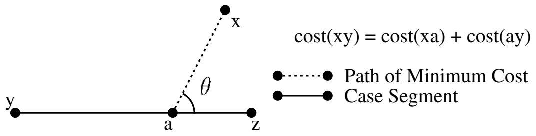

Figure 2: *Finding the route of minimum cost.*

and it is faster to take the highway for part of the route than to go directly from x to y. The best place to merge with the highway is at point a. (Note that there may not be an on-ramp at a in the real world; it is PRODIGY's responsibility to find some legal merge point close to a.)

In this figure, $\overline{yz}$ is a case segment. The optimum route between x and y is $\langle \overline{xa}, \overline{ay} \rangle$, for some point a that depends on the value of β. The cost of the optimum route is equal to length($\overline{xa}$)$+\beta\times$length($\overline{ay}$). Let θ represent the angle $\angle xaz$; the position of a is computed from the fact that $\theta = \cos^{-1}\beta$. In the limiting case where $\beta = 0$, $\overline{xa}$ is perpendicular to $\overline{yz}$.

We assign a cost to each edge (x, y) of G by considering a number of possible routes between x and y, and taking the route with minimum cost. The first route we consider is a straight line between x and y. Then, for each case segment, we consider the optimum route that uses the segment. Several examples are demonstrated in Figure 3 for shortest routes between x and y. The first three examples are all resolved by the formula $\theta = \cos^{-1}\beta$. Examples 2 and 3 are notable because they show that a segment does not need to be connected to x or y to provide a low-cost route. Example 4 shows that a segment between x and y does not always improve the optimum route.

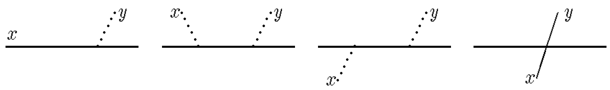

Figure 3: *Four examples of shortest routes. Other cases are possible. Points where dotted lines join the solid lines (cases) depend on the cost factor β of the case.*

For each edge (u, v) in G, we record which simple route from u to v had the minimum cost so that the entire route (and set of cases) can be reconstructed later. Even though we consider only a few of the infinite number of possible routes for each edge, it can be shown that finding the lowest cost path from i to g in G is equivalent to finding the optimum route from i to g in the case graph.

Unfortunately, the cost of this algorithm is prohibitive because we need to connect every pair of vertices, even if they are distant from each other. (Though i and g may be far apart, the optimal route between them might be a straight line.) The graph G has $O(n^2)$ edges, where n is the number of vertices. It takes $O(m)$ time to determine the cost of each edge, where m is the number of cases. Hence, the overall complexity is $O(m^3)$. However, we can develop from this a much faster algorithm that finds a good approximation to an optimal route.

A Fast Algorithm for Finding a Good Route

To derive a fast approximate algorithm, we take advantage of *locality*, the principle that one is most likely to travel from vertices to other nearby vertices. This saves computational effort because we can ignore interactions between distant vertices and segments. The graph G becomes a planar graph called a *Delaunay triangulation*. Delaunay triangulations have several desirable properties:

- They provide a structure that makes it possible to quickly determine the edge weights of G;
- Local modifications of the triangulation can be easily made; and
- They form a good approximation of which vertices are closest to each other. Take for example Figure 4. This

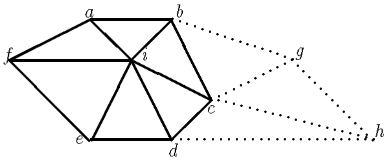

Figure 4: *A set of points and their triangulation.*

figure shows a small set of points and the Delaunay triangulation of those points. Imagine that each of the points a through h are endpoints of case segments, and that i is the initial point. In each direction around i, the triangulation indicates what case is closest to i. Any cases outside the hexagon centered at i (such as those involving g and h) are further away, and are far less likely to be directly connected to i in the final solution path. It is these more distant cases that we ignore in our heuristic.

Our algorithm consists of several steps. First, we form a Delaunay triangulation of the vertices of the case graph. Then we calculate the edge costs and use Dijkstra's shortest path algorithm to find a set of cases.

Step 1: Conforming Delaunay Triangulations

The Delaunay triangulation of a set of points is a triangulation in which there are no points inside any triangle's circumcircle. (The circumcircle of a triangle is the circle that passes through all its vertices.)

Triangulating the case graph makes it possible to determine the most likely cases that a vertex will be connected to, and allows us to reduce the number of cases examined for each pair of vertices. Note that this act is a heuristic: in very unusual situations, some cases are ignored that should not have been.

To represent the case graph, we use a *conforming Delaunay triangulation*. The vertices of the case graph (as in Figure 1a)) are triangulated, then additional points are added to ensure that all of the cases appear as edges in the triangulation (see Figure 5) and that other conditions are satisfied that, due to space limitations, we cannot enumerate here.

Points can be added incrementally to a Delaunay triangulation [15; 8], making it possible for the case library to grow without having to retriangulate the entire graph.

Inserting a vertex is a two-step process: first delete any triangles whose circumcircles contain the new vertex (Figure 6a). Then, add edges to connect the new vertex to its surrounding vertices (Figure 6b). Under ordinary circumstances, when a new point is added, only the nearby edges need change, and so a point can be added in constant time.

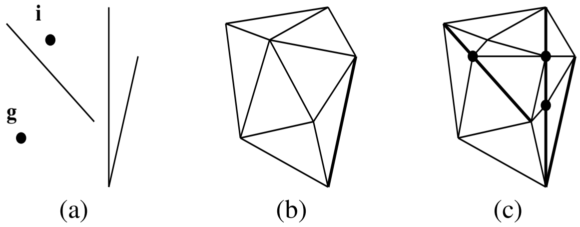

Figure 5: *(a) Case graph. (b) Delaunay triangulation of the case graph. (c) Conforming Delaunay triangulation of the case graph.*

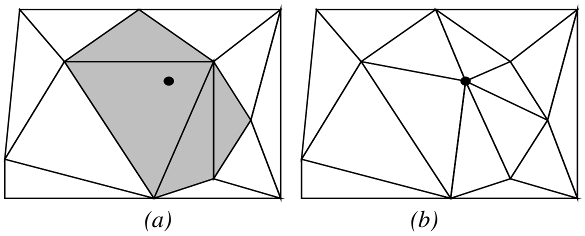

Figure 6: *(a) Inserting a new point into a Delaunay triangulation. The shaded triangles are deleted because the new point is inside their circumcircles. (b) The new point is connected to its neighbors.*

The conforming Delaunay triangulation of the set of cases from Figure 1b is shown in Figure 7.

Step 2: Edge Costs

The cost calculation for an edge (x, z) is similar to that of the optimal algorithm, but instead of examining every case segment, we examine only the edges of the triangles in the vicinity of (x, z). It can be shown that the only segments that need to be examined are those which intersect the interior of the diametral circle of (x, z).

Usually, we need only examine the two triangles adjacent to (x, z). (The exceptions are for a few cases like Examples 2 and 3 of Figure 3. A good heuristic is to ignore these examples and examine only two triangles.) For two triangles,

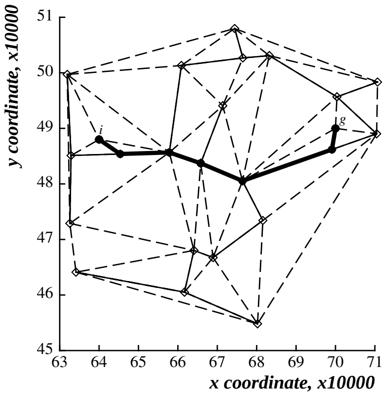

Figure 7: *A Conforming Delaunay Triangulation and the path found by Dijkstra's algorithm for some set of βs associated with each edge, and the labelled initial and goal points. Solid lines are cases; dashed lines are triangulation edges.*

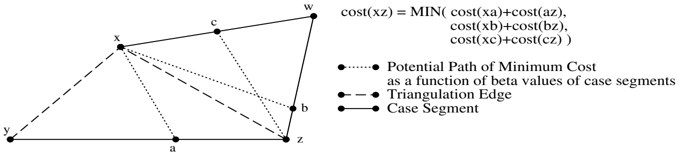

Figure 8: *Cost Calculation for Two Triangles*

Figure 8 illustrates some of the possibilities. To determine the cost of edge (x, z), only a small number of alternative routes need be considered, taking only constant time per edge.

Step 3: Dijkstra's Algorithm

Once the triangulation and cost assignment steps are complete, we use Dijkstra's shortest paths algorithm to find the optimum route, as before. It can be shown that in the worst case the cost of this route is within less than a factor of 3.5 of optimal; we conjecture that this constant can be improved to $\frac{\pi}{2}$. In practice, our algorithm does much better; we have never found a route worse than 1.3 times longer than the optimum route. The path shown in Figure 7 shows a path chosen by Dijkstra's algorithm, given the shown initial and goal points and some β value assigned to each edge.

The case segments on the route returned by Dijkstra's algorithm are sent to the planner, which transforms the coarse-grained plan into a fine-grained plan usable by a robot.

Complexity Analysis

All the steps in the method described above have costs that are based on the number of cases in the case graph (m) rather than on the size of the map. Since the cases are abstractions of the map, we expect m to be much smaller than the size of the map, which contains 25,000 street segments.

Conforming Delaunay Triangulation. Delaunay triangulations can be formed in $O(n \lg n)$ time, where n is the number of points in the triangulation [8]. Points occur at the endpoints of cases and at the initial and goal points. The number of additional points added to split segments is linear in practice[2]. Hence, $n \in O(m)$ and this step takes $O(m \lg m)$ time. Since the triangulation is planar, the number of edges E in the triangulation is $O(m)$.

Edge Costs. As mentioned before, the edge costs can be determined in constant time per edge. Hence, the total time to compute these is $O(m)$.

Dijkstra. Dijkstra's algorithm typically runs in $O(n^2)$ time, but it is possible to achieve a running time of $O(n \lg n + E)$ by implementing a priority queue with Fibonacci heaps [4; 6]. This step therefore runs in $O(m \lg m)$ time.

The total run time is $O(m \lg m)$, which is optimal[3]. Furthermore, once planning is complete, we can quickly update the triangulation to reflect any newly learned cases. The cost of this will depend on how much of the triangulation the route covers. Virtually all the Delaunay triangulations that arise in practice have the property that a point can be inserted in expected constant time, if the location of the point in the triangulation is known[4]. Therefore, we can expect to update the triangulation in time proportional to the number of vertices on the route, and we only need to update the costs of those edges that are near the route. This incremental behaviour is one of the benefits of using Delaunay triangulations.

For a problem with multiple goals, there is only one minor change to the above algorithm. Once we have built the triangulation and assigned edge costs, we run Dijkstra's algorithm several times, using one of the goal points as the source each time. The complete set of cases is then given to PRODIGY, which solves the problem. This change does not affect the asymptotic running time of the similarity metric since the number of goals is constant and small compared to the search space. PRODIGY/ANALOGY will find the best final ordering of the goals, implicitly solving the Travelling Salesperson Problem.

Results

For the purposes of this paper, we created a set of case files by randomly selecting streets and segments from the original map of 25,000 segments. Table 1 shows some of the timings we obtained for the similarity metric running on a SPARCstation ELC workstation. Note that when used within a CBR system, the triangulation is updated incrementally, and hence the Delaunay triangulation time is much smaller than shown in the table.

	11 cases	31 cases	492 cases
Delaunay	20 msec	85 msec	340 msec
Edge Costs	10 msec	10 msec	80 msec
Dijkstra	10 msec	15 msec	70 msec
Total	40 msec	110 msec	490 msec

Table 1: *Timings of parts of our algorithm. Dijkstra's algorithm is used for a single source and a single goal.*

Table 2 shows a sample of the distance values obtained by our algorithm when run with one particular initial point and several different goals.

In the graph used to generate these numbers, all case segments were assigned a β value of 1; hence the shortest route between two points is always a straight line. The table shows that the Delaunay distance, although slightly longer than the Euclidean distance, is well within the predicted range. As mentioned above, the worst ratio we have seen was 1.3.

We do not show a table in which the β values differ from 1 because we have not yet implemented the exact algorithm for comparison. We predict that as β decreases, the ratio between the route found by our algorithm and the true lowest cost route will be even better. It suffices to say that the

[2] Pathological cases can be found where an asymptotically larger number of points must be added, but these are of theoretical interest and do not occur in realistic maps.

[3] We show this by pointing out that finding a route through a one-dimensional sequence of cases is equivalent to sorting. A two-dimensional problem can only be worse.

[4] However, if the location of a point in a triangulation is not known, finding it typically takes $O(\lg m)$ time, like searching through a sorted list. This is why we cannot form a Delaunay triangulation in $O(m)$ time.

(goal)	Euclidean	Delaunay	Ratio
(586244.0, 507260.0)	13801.5	13801.5	1.0000
(619759.0, 512250.0)	46789.4	47685.5	1.0191
(622042.0, 514523.0)	49178.1	49178.1	1.0000
(707573.0, 408678.0)	169121.3	183481.2	1.0849
(594682.0, 417310.0)	96255.5	97538.3	1.0133
(595227.0, 421376.0)	92428.3	93936.8	1.0163

Table 2: *Results from the case-graph containing 11 cases, using (572984.0, 511088.0) as initial point. Ratio = Delaunay distance / Euclidean distance.*

algorithm consistently finds low cost paths containing cases, rather than choosing a straight line from the initial point to the goal.

Discussion

Unaided planning of a solution to the problems we have discussed could require exponential search time [9]. If a similarity metric can magically return a perfect cover-set of cases (one that covers every step of the entire low-level plan) in the correct order, the planning problem becomes linear in solution-length. Although we do not expect (or even want) the case library to contain a perfect cover-set of cases for all problems, having a good similarity metric can be profitable.

The practical method described in this paper provides the planner with a good set of cases quickly. Unlike ROUTER's *neighbourhood* heirarchy), our algorithm requires neither the cases nor their endpoints to be extremely close to the new problem, and thus allows much greater flexibility to the system as a whole. The method is applicable in any CBR planning domain with continuous values: some examples might include economics, physics, and chemistry.

Incremental behaviour, such as that of our case library, is a feature desirable in any learning system. Although the work involved in finding a good set of cases will grow with the size of the library, it is still only $O(m \lg m)$, where m is the size of the case library and is much smaller than the number of street segments that the planner is working with. Because the set of cases returned by the similarity metric will cover more and more of the entire solution as the library grows, the additional work in finding the similar cases will be more than made up for by the time saved in planning. The tradeoff between retrieval and search costs has been discussed at length elsewhere [11; 13].

The method described above is designed with a feedback loop in mind: the costs (β) on the segments are dynamic and not fixed. After each execution of a path, the costs can be modified and updated. The more a particular street is used, the more closely the cost will reflect the real cost. If the expense grows, then the likelihood that it will be found by the similarity metric drops. If the expense drops, then the likelihood that it will be found increases. In this way, the similarity metric will improve its behaviour with use and return a better and better set of cases.

We have integrated the similarity metric into PRODIGY/ANALOGY. We are currently investigating the improvement in plan quality and the reduction in total planning time, using the entire map as the basis. The entire planning system is being used with the simulator for one of Carnegie Mellon's robots. We look forward to seeing the system work with a robot, and to showing that indeed, a robot can learn.

References

[1] Patrick Broos and Karl Branting. Compositional instance-based acquisition of preference predicates. In *Case-Based Reasoning: Papers from the 1993 Workshop*, pages 70–75, Washington, D.C., July 1993. AAAI Press. Available as Technical Report WS-93-01.

[2] Bernd Bruegge, Jim Blythe, Jeff Jackson, and Jeff Shufelt. Object-oriented system modeling with OMT. In *Proceedings of the OOPSLA '92 Conference*, pages 359–376. ACM Press, October 1992.

[3] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: An integrated architecture for planning and learning. In K. VanLehn, editor, *Architectures for Intelligence*. Erlbaum, Hillsdale, NJ, 1990. Available as Carnegie Mellon Technical Report CMU-CS-89-189.

[4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press, Cambridge, Massachusetts, 1990.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[6] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(1):209–221, 1985.

[7] Ashok Goel and Todd J. Callantine. A control architecture for run-time method selection and integration. In *AAAI Workshop on Cooperation Among Heterogeneous Intelligent Agents*, July 1991.

[8] Leonidas Guibas and Jorge Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985.

[9] Karen Haigh and Manuela Veloso. Combining search and analogical reasoning in path planning from road maps. In *Case-Based Reasoning: Papers from the 1993 Workshop*, pages 79–85, Washington, D.C., July 1993. AAAI Press. Available as Technical Report WS-93-01.

[10] Kristian J. Hammond. Case-based planning: A framework for planning from experience. *Cognitive Science*, 14:385–443, 1990.

[11] M. T. Harandi and S. Bhansali. Program derivation using analogy. In *Proceedings of IJCAI-89*, pages 389–394, 1989.

[12] Janet L. Kolodner. *Case-Based Reasoning*. Morgan-Kaufmann Publishers, Inc., San Mateo, CA, 1993.

[13] Manuela M. Veloso. Variable-precision case retrieval in analogical problem solving. In *Proceedings of the 1991 DARPA Workshop on Case-Based Reasoning*. Morgan Kaufmann, May 1991.

[14] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992. Available as technical report CMU-CS-92-174.

[15] D. F. Watson. Computing the n-dimensional Delaunay tesselation with applications to Voronoi polytopes. *The Computer Journal*, 24(2):167–172, 1981.