Finding Shapes in a Set of Points

Kenneth A. Ross, David Vespe, David Hessing, Pranay Jain
kar@cs.columbia.edu, {djv, djh44, pj2130}@columbia.edu

Columbia University*

Abstract

We present a tool for querying a set of points for geometric shapes. This tool was developed as part of a larger project studying the architecture of 13th century French churches. We present a query language for specifying shapes. We describe an implementation of the tool that optimizes and executes shape queries. We describe the performance of the tool, and show examples of how it can be used to analyze building floorplans. The tool is available for download over the internet.

1 Introduction

In 2003, a team of computer scientists and art historians at Columbia began an interdisciplinary collaboration. The project aimed to use computers in novel ways to help historians better understand the architecture of Romanesque churches in the Bourbonnais region of central France. The tools developed during the course of the project could also be applied to other similar endeavors elsewhere. For an overview of the churches studied, see [1].

In this article, we describe one of these tools, called the "Shape Query" tool, whose purpose is to look for "interesting shapes" in the buildings' designs. Since written records of the construction of these 13th century churches have been lost, the only remaining source of information is the buildings themselves. During a number of visits to the region, members of the team carefully measured over one hundred churches using modern instruments, and created digital representations of them. The shape query tool operates on two-dimensional sections of these representations. A commonly used section is the horizontal section that describes the floorplan of the church. Each section is annotated (either by a domain specialist or by a person posing a query) with architecturally meaningful points, such as the corners of the building, the centers of the columns within the building, the apex of a crossing overhead, etc.

It is sometimes possible to deduce some of the building practices used by understanding the geometric shapes found in the layout of the building [2]. For example, consider the rectangle formed between four columns in a grid-like arrangement. A rectangle in which the ratio of length to width is $\sqrt{2}$, is suggestive of the use of a string aligned with the diagonal of a unit square, with the string rotated to form the long side of the rectangle. (This was a commonly used way of reliably obtaining certain measurements using the technology of the day.) Other special ratios have analogous interpretations.

The shape recognition tool aims to help identify the underlying geometric patterns within and among the various churches. Prior to such tools, art historians would typically look for patterns by hand, a process that is not only time consuming, but subject to possible subjective judgements.

The Shape Query tool required a *shape query language*. We describe a language that allows a user to specify a collection of points with constraints on distances and angles, to be enforced within a given tolerance[1] level. We then describe how the Shape Query tool processes shape queries, including a query optimization phase. We outline how the query processing engine is packaged so that it can be used either in a stand-alone manner (with an appropriate user interface), or as a component of a larger system. Finally, we describe some initial results obtained from using our system on the church datasets.

*This work was funded in part by a grant from the Andrew W. Mellon Foundation.

[1] A tolerance accounts for building settling, measurement error in construction, and measurement error in data acquisition.

2 Point Set

Each underlying data set is a set of labeled points in a two-dimensional plane. Point-sets are transferrred between the shape query engine and external sources using an XML representation. A *pointset* contains various instances of the *point* entity. Each *point* entity stores information about its x and y coordinates. An optional *id* associated with the point is unique in the point dataset and can later be helpful to interpret the results quickly. A point may have any number of *label* values. The *label* attribute allows us to provide extra information related to that point (e.g., that point marks a corner of the church) that is useful for writing queries.

Example 2.1: A point set containing 2 points.

```
<pointset name="two-points">
  <point id="l1">
    <x>10</x>
    <y>20</y>
    <label>corner</label>
  </point>
  <point id="l2">
    <x>50</x>
    <y>100</y>
    <label>column</label>
  </point>
</pointset>
```

3 Query Language

A query is specified as follows:

- A number k of points. We denote the points by P_1 through P_k. Each query point is required to map to a distinct point in the data.
- A list E_1 through E_m of edges. An edge is specified as a pair (P_i, P_j) representing the directed line segment from P_i to P_j .
- A list A_1 through A_p of angles. An angle is specified as a pair (E_i, E_j) representing the anticlockwise angle (in degrees) from E_i to E_j.[2]
- A list of constraints on the edges $E_1, \cdots, E_m$ and angles $A_1, \cdots, A_p$ expressed in a constraint language L.

[2]The angle between edges makes sense even if the edges do not share a common point.

Note that constraints do not apply to single points; since the coordinate system used is assumed to be arbitrary, there is nothing interesting one can say about single points. All semantically interesting queries relate points to other points. Edge lengths and angles are independent of the coordinate system.[3] Our constraint language L allows equalities and inequalities of mathematical expressions involving edges (E_i implicitly represents its length), angles, and numeric constants. Points may also be constrained to have a certain label. For example, if we are only interested in shapes whose vertices are church columns, we would constrain the points accordingly.

Example 3.1: Square

Points	4
Edges	$E_1 : (P_1, P_2), E_2 : (P_2, P_3), E_3 : (P_3, P_4)$
Angles	$A_1 : (E_2, E_1), A_2 : (E_3, E_2)$
Constraints	$E_1 = E_2 = E_3, A_1 = A_2 = 90$

There are many equivalent alternative ways to specify a square. Because of symmetry, a subset of points may match a query in more than one way.

The following example shows one way to specify a square in which an absolute tolerance of δ is allowed for edge lengths, and an absolute tolerance of ϵ is allowed for angles. (Both absolute and relative tolerances are supported by the system.)

Example 3.2: Approximate Square

Points	4
Edges	$E_1 : (P_1, P_2), E_2 : (P_2, P_3), E_3 : (P_3, P_4)$
Angles	$A_1 : (E_2, E_1), A_2 : (E_3, E_2)$
Constraints	$\mid E2 - E1 \mid < \delta, \mid E3 - E1 \mid < \delta,$ $\mid A1 - 90 \mid < \epsilon, \mid A2 - 90 \mid < \epsilon$

The tolerance parameters in this query are applied in an asymmetric way. It is possible that the distance between P_4 and P_1 is more than δ away from E_1 because it is constrained only indirectly via the other constraints. Similarly, it is possible that the angle $P_3P_4P_1$ is more than ϵ away from 90 degrees, and that angles A_1 and A_2 are more than ϵ away from each other.

Example 3.3: Rectangle of Columns

Points	4
Edges	$E_1 : (P_1, P_2), E_2 : (P_2, P_3), E_3 : (P_3, P_4)$
Angles	$A_1 : (E_2, E_1), A_2 : (E_3, E_2)$
Constraints	$E_1 = E_3, A_1 = A_2 = 90,$ $\text{label}(P_1) = \text{label}(P_2) = \text{label}(P_3) =$ $\text{label}(P_4) =$ `column`

[3]Some care is needed to avoid artifacts caused by angle wrapping at 360 degrees.

Queries are accepted by the shape query engine in an XML representation that implements the query language described above.

Given a point dataset containing n points, a k-point query can be answered in time $O(n^k)$ by simply checking all $n(n-1)\cdots(n-k+1)$ combinations of distinct points to see if the constraints are satisfied. Some queries have inherently high-complexity. For example, a k point query with no constraints has $\Theta(n^k)$ answers.

Fortunately, such queries are likely to be uncommon in practice, analogous to cartesian product queries over relational databases. Points are added to queries typically with a contraint relating the point to previously defined points. That constraint is typically quite selective. (For example, a query for a ninety degree angle with a 1.5 degree tolerance would have a selectivity of $\frac{3}{360}$ on a uniform dataset.)

Nevertheless, many queries can take $\Omega(n^2)$ time for the following simple reason: Consider the first two points that are specified in the query. Since constraints must be independent of the coordinate system, the only constraint that can be placed on this pair of points is on the distance between them. However, we are often querying for shapes independent of scale, so this distance will, in general, be unconstrained. Thus, all possible pairs of data points are candidates for these first two query points, requiring $\Omega(n^2)$ operations.

4 Query Processing

We initially implemented the shape query engine using a commercially available SQL database to store the point and edge data, with appropriate indexes created on combinations of point-id, angle and distance. However, we found that the performance of the SQL database was poor, and that the plans chosen by the commercial optimizer did not match the plans that we believed were optimal. When we reimplemented the system using a specially tailored optimizer (described below), we observed an order of magnitude speedup. For our problem domain, the point and edge data sets are likely to fit in RAM. Our cost estimates and query processing algorithms assume in-memory structures for the source data and intermediate results.

In preparation for running queries, we build index structures based on all the edges in the point set. Since every pair of distinct points defines an edge in each direction, each index structure contains $n(n-1)$ edges, where n is the number of points. The system assigns each point a unqiue point-id, that is unrelated to the user-visible *id* attribute. We build the following four index structures on the edge relation. Angle is measured relative to some arbitrary coordinate system.

1. Index on (first-point-id, angle)
2. Index on (first-point-id, distance)
3. Index on (angle, distance)
4. Index on (distance, angle)

Since the point sets are static, each index is implemented as an array of pointers to edges. The order of elements in the index is determined by the lexicographic order of edges according to the specified pairs of attributes. To speed up in-memory access to the indexes, we implemented CSS Trees [3] that reduce the number of cache misses needed to search a sorted array.

4.1 Building a Query Plan

The process of building a query plan uses dynamic programming, and is analogous to the process of choosing a join ordering for a relational query plan [4]. Plans for subqueries involving a small number of query points/edges are initially computed, and their costs are calculated. For each combination of query points, the plan with the best cost is retained, and used to determine candidate plans for larger subsets of points/edges. The cardinality of the sub-plan, i.e., the number of solutions, is also estimated.

A subplan can be combined with a new point/edge using the analog of either a nested-loop join, or an index-nested-loop join. Where possible, index-nested-loop joins are preferable because of their lower selectivity. New points can be added based on an angle or distance constraint relative to existing points, using the id/angle or id/distance indexes respectively. New pairs of points can also be added using an angle or distance constraint relative to existing points, using the angle/distance or distance/angle indexes.

Once new points have been added, there may be additional constraints that can be simply checked before finding additional points. Such checks are performed as soon as possible to reduce the size of the intermediate results.

The dynamic programming approach allows query optimization to work in reasonable time for small queries, but it still becomes expensive for longer queries. For this reason, when the amount of

work exceeds a certain level, we revert to a greedy approach in which we look at a fixed number of plan alternatives for a set of query points. In practice, we set the maximum number of plans per sub-query at 10, since this gave us a balance of speed in finding plans and speed in their execution.

We illustrate query optimization by studying the square query of Example 3.1 and illustrated in Figure 1. Point P_1 corresponds to the lower-left corner, and points are numbered anticlockwise.

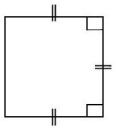

Figure 1: Constraints defining a square

One plan, illustrated in Figure 2 starts out by taking the set of all possible edges and labeling each in turn as P_1P_2. Then, for each candidate edge P_1P_2 the plan uses the length index to find other edges that could correspond to P_3P_4 by virtue of having the same length (within the required tolerance) as P_1P_2. Now that we have all four points, the remaining contraints may simply be verified in some order, with non-matches being eliminated from the result set. In Figure 2, we first check that the length of P_1P_2 matches the length of P_2P_3, then that angle $P_1P_2P_3$ is within the specified tolerance (1.5 degrees) of 90°, and finally that angle $P_2P_3P_4$ is within the specified tolerance of 90°.

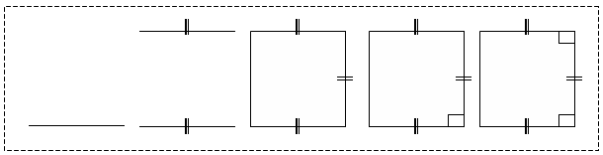

Figure 2: Query Plan 1 to identify a square

A second plan, illustrated in Figure 3 starts out by taking the set of edges and labeling each in turn as the edge P_1P_2. For each such candidate edge, the plan uses the (first-point-id,angle) index to find other edges (representing the edge P_2P_3) whose angle relative to $P_1P_2$is within the specified tolerance of 90°. At this point, we can verify that the length of P_1P_2 is within the specified tolerance of the length of P_2P_3. We again use the (first-point-id,angle) index to find other edges (representing the edge P_3P_4) whose angle relative to P_2P_3 is within the specified tolerance of 90°. Finally, we verify that the length of P_3P_4 is within the specified tolerance of the length of P_1P_2.

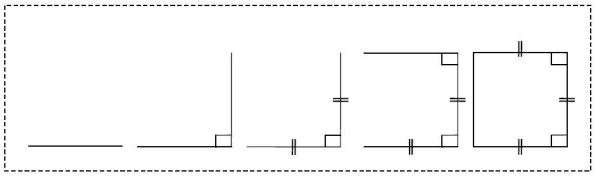

Figure 3: Query Plan 2 to identify a square

To obtain cost estimates for query plans, we need to estimate the work done for one point/edge to look up the appropriate index structure. We also need to estimate the cardinality of intermediate results, because that determines how many index lookups will be performed on subsequent steps. As for relational query optimization, it is not essential to get these estimates exactly right, particularly if very accurate estimates would require excessive computation time or sophisticated geometric reasoning.

If n is the number of points, then our initial point-pair will have work proportional to n^2, and cardinality $n(n-1)$. We assume all indexes are accessed in time logarithmic in the number of edges. Cardinality estimates are based on the selectivities of the constraints on the edge relation.

For angle constraints, the selectivity is estimated as $\frac{2\epsilon}{360}$ where ϵ is the tolerance in degrees. For (point,angle) constraints, the selectivity is $\frac{2\epsilon}{360n}$. These estimates are based on a model in which edge angles are uniformly distributed.

In some query plans, such as the first plan above, we use a constraint on distances to allow access to edges via an index. The distribution of edge lengths is not uniform on any range. To obtain a selectivity estimate for a constraint of the form $|E_1 - f(E_2)| < \epsilon$ we proceed as follows. Three random edges are chosen from the edge table as candidates for E_2. For each edge, the number of edges with E_1 satisfying $|E_1 - f(E_2)| < \epsilon$ is determined by consulting the index on distance; matching edges will be in a single range within the index. The selectivity estimate is the average proportion of matching E_1 values over the three random E_2 values.

For other selection conditions, the actual selectivities may vary from one query to another. Based on empirical observations with realistic queries, we estimate a selectivity of 0.05 for all such constraints.

Comparing the two plans above, our cost model correctly identifies the second plan as superior to the first. The primary reason for this superiority is that more partial solutions are pruned early, before candidates for P_4 are looked up.

#Points	#Results	Index Time	Plan Time	Exec Time
50	0	0.016	0.016	0.062
100	8	0.062	0.016	0.188
250	230	0.328	0.062	1.58
500	3968	1.75	0.078	9.63
750	17505	4.14	0.094	29.9
1000	61103	8.42	0.156	71.1
1250	156972	13.7	0.187	130

Table 1: Query speed versus number of points.

Plans are executed in a pipelined fashion, analogous to a left-deep plan in a relational system. In a pipelined plan, at each level, an edge passes completeley through the pipeline before the next edge at that level is considered. This is particularly important in our in-memory setting because intermediate results can be very large, and could exceed the physical memory capacity.

The result set generated by a query is reported back to the main system in an XML format. The final result set may have duplicates because a particular set of points can match a query in more than one way; this occurs frequently when the shape that is the target of the search has any sort of symmetry. For example, in an equilateral triangle query, three points that are all mutually equidistant will match the query in six different ways. Users are given the option of eliminating duplicates, meaning that a given set of data points is reported as an answer exactly once. Duplicate elimination is implemented as a final step on the computed query result.

5 Query Speed

The system was implemented in Java, and executed on a 3.4 GHz Pentium 4 system with 3 GB of RAM. We illustrate the performance of the system using a square query over a dataset containing a variable number of points randomly distributed in a unit square. Figure 4 illustrates a square found in a collection of 100 random points. For our target application, any single church would typically be annotated with fewer than 50 points. Nevertheless, we investigate how well query time scales with the number of data points. Table 1 shows the results, where time is measured in seconds. "Index time" refers to the time taken to construct the appropriate indexes, a process that is done once per session with a given data set. "Plan time" refers to the time spend in query optimization.

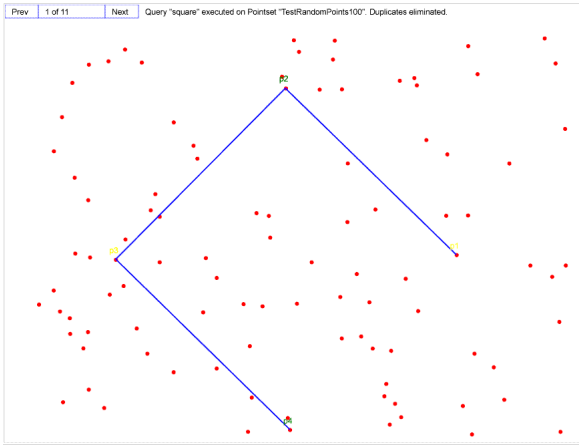

Figure 4: A square found in 100 random points

The run time is increasing at a rate proportional to approximately $n^{2.7}$, consistent with our earlier discussion that the complexity would be $\Omega(n^2)$ but less than $O(n^4)$, the complexity of trying all possible combinations of data points.

The benefit of using CSS Trees relative to binary search amounted to about 4% of the execution time. This benefit is small probably due to the various interpreter overheads of Java.

6 Patterns Identified

For our church analysis application, it is not sufficient to simply find all matching shapes and analyze them. The results must be scanned by the user to eliminate spurious answers. For example, suppose that the art historian is attempting to understand whether there is an overabundance of rectangular shapes with side ratio of $\sqrt{2}$, in order to determine whether certain design methods were used in building the church. The art historian expects that rectangles whose sides are parallel to the church walls are likely to reflect active design. On the other hand, rectangles that diagonally traverse the church are likely to be coincidental; given enough marked points and a wide enough tolerance, such coincidental shape matches will arise just by chance.

Some simple patterns were apparent. For example, when looking at the "smallest" rectangles formed by adjacent columns, the shorter edge was consistently between 3m and 6m long. See Figure 7. These bounds probably represent structural constraints. Columns have to be far enough apart to define a useful space, but close enough together to adequately support the arches overhead.

To begin to study the question about design methods used for these churches, we specified queries that looked for rectangles, and analyze the side length ratio of the resulting answers (with spurious rectangles eliminated). Figure 5 shows a rectangle found for the church in Besson.

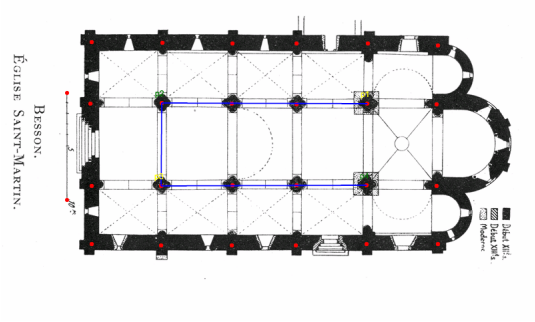

Figure 5: A rectangle found in "Besson"

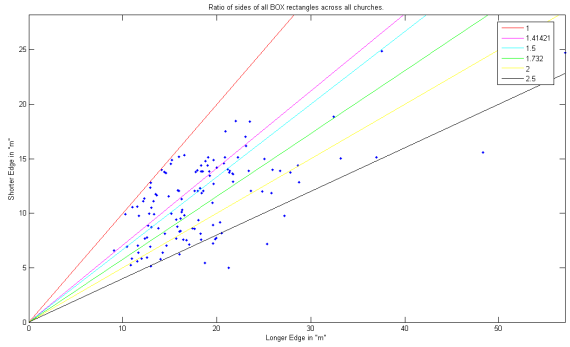

Figure 6: Longer side by shorter side rectangles for entire church body.

We further refined the analysis to look only at rectangles formed by the main body of the church, in which the corners of the rectangle are the corners of the church itself. These rectangles are summarized in Figure 6. Some well-known ratios such as $\sqrt{2}$ are shown on the diagram for reference.

We also refined the analysis to look only at rectangles formed by adjacent columns within the church, in which the corners of the rectangle are columns of the church.[4] These rectangles are summarized in Figure 7.

[4]This query cannot be expressed in the query language provided by our system, because there is no way to distinguish adjacent columns from nonadjacent columns. To perform this analysis, we used our query engine to find all rectangles with columns as corners, and in a post-processing step only retained one rectangle for each church, namely the rectangle with smallest area.

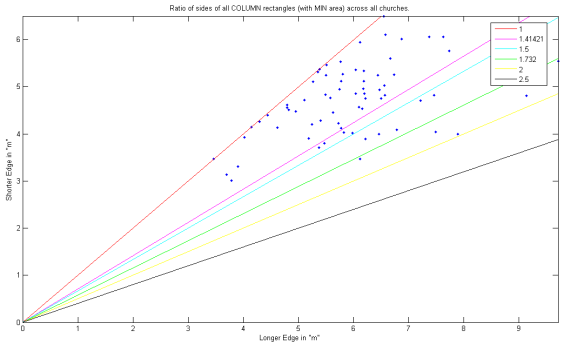

Figure 7: Longer side by shorter side for rectangles formed by adjacent columns.

7 Conclusion

We have designed and implemented a system for finding user-defined shapes among a set of points. Our application domain focused on finding shapes among marked points on building floorplans or cross-sections. Nevertheless, our system could be useful in other domains, such as finding constellations of stars in the night sky.

Our query language might need to be generalized for other applications. Derived points that are not explicitly present in the database might be useful. For example, the center of a rectangle could be used to define distance constraints on other points. At present, there is also no facility for dealing with curved shapes.

The source code and executable for the system can be downloaded at `http://www.cs.columbia.edu/~kar/software.html`.

References

[1] http://www.learn.columbia.edu/bourbonnais/

[2] F. Bucher, Medieval Architectural Design Methods, 800–1560, Gesta, 11, 1972, 37-51.

[3] J. Rao and K. A. Ross, *Cache Conscious Indexing for Decision-Support in Main Memory,* Proceedings of the 1999 VLDB Conference, September 1999.

[4] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. Access Path Selection in a Relational Database Management System. SIGMOD 1979: 23–34.