

Duke University
Department of Computer Science

Senior Thesis

STAR-Tree

and

Protein Shape Matching

John Tran

Adviser: Pankaj K. Agarwal

April 24, 2002

Abstract

An interesting problem today in computer science and its applications is keeping track of moving data. That is, bounding groups of objects and keeping this bounding information up to date while they move over time. Motion and constant changes in the motion require algorithms that can efficiently update themselves in order to answer queries in a timely fashion.

This paper describes modifications made to the original STAR-Tree implementation that allow for a better and faster user interface due to changes in the graphics library used and simplification of the data.

Another problem in biology consists of trying to match proteins. Using the three-dimensional structure, we derive scores for the proteins using the intramolecular distances between carbon- α atoms in order to find the level of similarity or dissimilarity. This will then lead us to find the optimal alignment of the proteins, which will help us understand binding and protein-protein interactions.

Chapter 1

STAR-Tree

1.1 Introduction

An interesting problem today in computer science and its applications is keeping track of moving data. In simulating real-world environments, one must often monitor large sets of data as they move over time, such as in mobile communications, air-traffic control, or enzyme kinetics. Much work has already been done on developing structures that maintain themselves locally when changes are made to the data structure, but our desire is to maintain the entire structure so that we can answer various queries in an efficient manner.

Thus we can use the STAR-tree indexing technique designed by Procopiuc, Agarwal, and Har-Peled [3]. This *Spatio-Temporal self-Adjusting R-tree* builds on the R*-tree structure [2] and is also closely related to the index introduced by Saltenis [4]. The STAR-tree structure allows us to answer queries on continuously moving objects. This is accomplished by representing the position of the moving point with a function $f(t)$ that can be updated on the fly to account for change of velocity of an object. By representing the location as a function of time, we can monitor continuously moving data, as opposed to the much simpler case of discretely moving data.

There are three types of queries that we would like to answer. Figure 1.1(a) shows an example of a rectangular query at time t . This query wants to know what points of the entire set are in a rectangular bounding box at time t . In (b), we want to know how many points lie in that rectangle in a

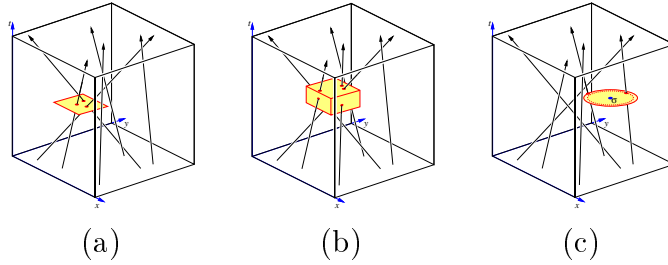


Figure 1.1: Three types of queries

certain time interval. Query (c) is known as the nearest-neighbor query, in which we want to know the closest point to the current point. STAR-Tree helps us answer all of these queries efficiently.

Each node in the STAR-tree keeps track of the bounding box that incorporates all of its children, and all of the data points are stored in the leaf nodes. As the data set changes, the difficult task is to keep track of the bounding boxes. In order to do this, a curve approximation is used that is discussed further in [3].

One of the applications of the STAR-tree is to monitor large sets of traffic data. We have produced data sets of cars moving along roads in the Raleigh-Durham, North Carolina area. Cars are represented as points to simplify calculations and visualization. Each car is given source and destination locations, and they precompute the optimal shortest path between these locations. This path can be modified at run time as well, in which case the STAR-tree would recalculate the new path and position function. The goal is to visualize the STAR-tree structure as the cars move, keeping track of the bounding boxes at each level of the tree structure and to watch the update events to see their effect on the structure of the tree.

1.2 Background

Procopiuc et al originally designed a visualization program for the STAR-tree structure using QT, a platform-independent API that contains functionality to produce graphical user interfaces. However, our desire is to produce a

much more sophisticated visualization suite. We would like to be able to zoom in and out of the road networks, ranging from views of a map of the entire U.S.'s major highways all the way down to the neighborhood-street level. We would also like to specify whether we wish to track a single point or a node in the tree as we wish. Since QT is not easily adaptable to these types of visualization, we turn to OpenGL and GLUT to perform our visualization.

1.3 Interface Implementation

The interface was developed on several SunSPARC machines, ranging from SPARC-5 to SPARC-20. Because of one of the I/O libraries used, it has to run on a UNIX-based system, although it can be ported to work with Linux. At run time, the program consumes memory proportional to the data sets that it gets as input. For most of our tests, this was approximately 300 megabytes of memory.

Cars in our data structure are stored as Point2d structs, which keep track of the x and y coordinates of the point. Roads are stored as a polygonal chain of points. These vertices are predetermined by a parser that goes through TIGER data from the US Bureau of the Census of all of the US road networks and filters the data so we only get the vertices of each road. That data is then simplified with adjustable parameters describing the number of roads and the number of legs to put in each optimal path. We can simplify a road up to any number of legs, meaning however many number of vertices we wish to use to describe the road. The tradeoff is between precision in describing the road and performance when optimizing paths through the network. This will be described more later.

An I/O efficient library called TPIE (Transparent Parallel I/O Environment) was used to minimize I/O activity. The specifics of cache and block sizes are included in the paper by Procopiuc [3] and the specifics of TPIE are described in [1].

In order to maximize efficiency, OpenGL display lists were used to store the road maps. One display map was used for each zoom level, where the number of zoom levels can be set by modifying one variable in the program. The zoom level can be chosen by either clicking the mouse to zoom in on a point or selecting the buttons on the side of the window which will keep the map centered at the same point but will change the zoom level to whichever

level is chosen.

We can also choose to look at a specific node by clicking on the box at the bottom of the screen. We can see what node we are on, what its children are, and who its parent is. Clicking on any of these choices displays that node and all of its children in the main screen.

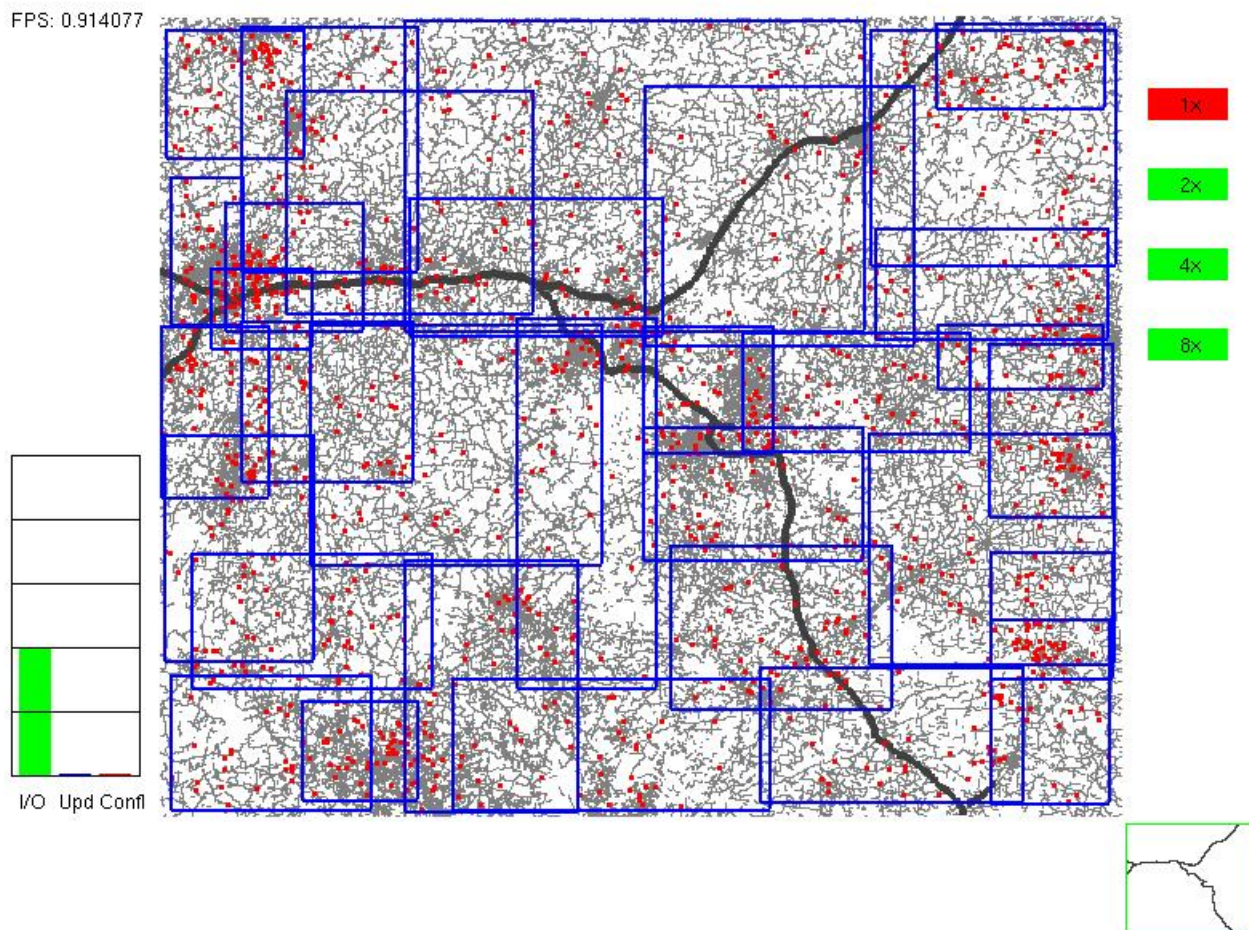


Figure 1.2: STAR-Tree interface

Figure 1.2 shows an image of the interface. Red dots represent the cars traveling along the road network and the blue boxes represent the bounding boxes of the nodes in the tree. On the left side of the interface, we can see

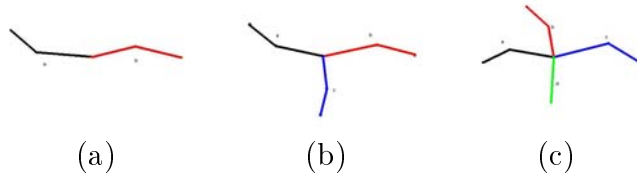


Figure 1.3: Different possible road endpoint connections

in real-time the number of I/O accesses, updates in the tree structure, and conflicts, which will lead to adjustments in the tree.

1.4 Road Simplification

We studied the statistical makeup of the road networks with the purpose of simplifying the road network into fewer roads. For the Raleigh-Durham map, we verified that most roads consisted of between three and six vertices. In addition, we concluded that most vertices in the network (86%) were part of only one road and that no vertices connected more than 6 roads. This allows us to perform more simplification on the road network as follows:

In order to minimize the number of roads while at the same time not losing any vertex information, we have to analyze the three cases for endpoints of roads as seen in figure .

In case *a*, a road ends where another road begins. We can simply join these two contiguous roads into one continuous road. In case *b*, three roads end at the same point. Here, we would like to join the two roads that create the largest angle, which would correlate to what we would desire in real life. This is easily done by using the law of cosines with each of the adjacent roads. In case *c*, we have a 4-way intersection, where we would like to join the opposing roads. This turns out to be the same case as *b* because we want to find two roads with the largest angle between them. Joining more than four roads with one vertex in common as that constitutes only 0.01% of the total roads, so we can ignore this case, although we handle it the same way as for *b*, finding the largest angle and combining those two roads.

The simplification makes the program run visibly faster.

1.5 Improvements

The current implementation of the road network classifies roads on two levels: a highway or a road. In order to produce many more zoom levels we would need to create more classifications. That is, we would need to create a hierarchy of road networks so we could easily filter out which roads to display at each zoom level. For efficiency concerns, we would also need to represent each road with a different level of detail for the different zoom levels. The final goal would be something similar to the maps found on popular online travel planner maps.

A major concern would also be to ensure that the cars travel on the paths of the roads. This problem arises because during the visualization, we have to filter out roads with fewer than a certain number of vertices (typically 3 to 5). This has to be done in order to achieve a reasonable map that is not too clustered. Even though this simplification could be taken care of in the different levels of detail of the roads, we would still have to force the cars to take the paths defined by the roads. This would involve each road knowing which cars are on it and each car knowing which road it is on. This would add a level of sophistication to the program that would allow us to add many more features, such as traffic pattern coloring or filtering out unused roads.

1.6 Conclusions

The visualization tool as it is successfully displays the correctness of the STAR-tree algorithm. We can view the number of queries, updates, and I/O operations in real time as well as view the bounding boxes to see how STAR-tree works.

However, since we desire a much more higher-level tool to visualize the STAR-tree structure, we would be better off creating a better road network simplification. This would not only improve the running time of the program, but would allow for simpler zooming in and out as we request different levels of detail.

1.7 References

1. Arge L, Barve R, Hutchinson D, Procopiuc O, Toma L, Vengroff DE, Wickeremeshinghe R. *TPIE User Manual and Reference (Edition 0.9.01b)*. Duke University, 1999. Manual and software distribution available online at <http://www.cs.duke.edu/TPIE>.
2. Beckman N, Kriegel H-P, Schneider R, and Seeger B. **The R*-tree: An efficient and robust access method for points and rectangles**, *Proc. ACM SIGMOD Conf. on Management of Data*, 1990, pp 322-331.
3. Procopiuc CM, Agarwal PK, and Har-Peled S. **STAR-Tree: An Efficient Index for Moving Objects**. Submitted for publication 2001.
4. Saltenis S, Jensen CS, Leutenegger ST, and Lopez MA. **Indexing positions of continuously moving objects**, *Proc. ACM SIGMOD International Conference on Management of Data*, 2000, pp331-342.

Chapter 2

Protein Shape Matching

2.1 Introduction

As computing power and resources increase, we are finding more ways of solving biological problems using computers. Problems that require many calculations and iterations are now becoming easier to understand as we apply faster computers and more efficient algorithms to them. An example is in sequencing the human genome. A goal that would have sounded outrageous 50 years ago is now attainable and is in fact attainable in more than one method, as we observed in the race between Celera and NIH. Another example is the recent explosion in the application of DNA microarray analysis, producing immense quantities of data researchers can use to analyze regulatory patterns of thousands of genes.

The large number of proteins being sequenced over the years has ignited much research in the field of protein matching. Having these protein sequences helps us to classify proteins in many ways, including by active sites, by three-dimensional structure, or by size. We can then derive computational models that we can use to simulate protein-protein interactions or just to observe the characteristics of the protein. One topic of research related to this is protein shape matching.

Shape matching involves looking at the 3-D structure of a protein and attempting to match it to another protein.

When classifying and comparing proteins, one must define whether the goal is to classify similarity or dissimilarity. As one can predict, similarity

scores how similar two proteins are, and produces higher scores for proteins that are more structurally similar. Dissimilarity produces higher scores for structurally dissimilar proteins. One would therefore like to maximize similarity or minimize dissimilarity in order to match proteins. For the purposes of this project, we examined dissimilarity, and the inverse algorithm will work for similarity.

Given as an input the alpha-carbon backbone of the proteins, each represented as a point in 3-space, two predominate methods of structural shape matching are used. Most research so far has been done on comparing intermolecular distances. In doing this, one must come up with the correct set of transformations on one protein to line it up with the other to minimize the intermolecular distances. The difficulty here arrives in coming up with these transformations, as there are many ways to attempt to line up two proteins.

Another method of shape matching involves examining intramolecular distances. Here, one looks at the distances between residues in a single protein and tries to come up with a scoring function to relate it to the other protein's intramolecular distances. In this case, one must derive a scoring function that will assign a single (or several) value(s) to an entire 3-D structure. Difficulty arises when we try to maintain all of the three dimensional information in coming up with this one dimensional score for the protein. Holm and Sandler[2] defined a similarity score as $(a - \frac{D}{\langle D \rangle}) \exp(-(\frac{\langle D \rangle}{b})^2)$ where $\langle D \rangle$ is the average of the two intramolecular distances and a and b are arbitrarily defined. Several others [4,5] used $\exp(-(\frac{D}{a})^2) \exp(-(\frac{S}{a})^2)$ where S takes into account the local neighbors for each pair of atoms. As described by Koehl [3], no one scoring function has been proven better than the rest.

Both cases involve effort to minimize the running time of the algorithm. It is fairly simple to observe that a naive algorithm operates in exponential time. Given two proteins, X and Y, with sizes m and n , one could enumerate every possible subsequence of X and match it to every possible subsequence of Y, creating 2^m subsequences to match with 2^n others. To overcome this exponential running time, we will use dynamic programming in order to reduce the running time to polynomial time.

2.2 Theory / Background

In order to come up with an algorithm to find dissimilarity in proteins, it may be useful to examine the problem of sequence-aligning two strands of DNA. Determining a Longest Common Subsequence for these two strands is a classic problem in dynamic programming.

Given two strings written over the finite set of the four base pairs {A, C, G, T}, we would like to know the longest subsequence that is common to both strings. In order to do this, we define all possible subproblems and derive a recursion for dissimilarity.

Theorem 15.1 in Cormen states the following:

Let $X = \{x_1, x_2, \dots, x_m\}$ and $Y = \{y_1, y_2, \dots, y_n\}$ be the sequences we are comparing and $Z = \{z_1, z_2, \dots, z_k\}$ be the LCS so far. Three cases can occur:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

This can be rewritten in the recursion (Equation 15.14 in Cormen):

$$c[i, j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i - 1, j - 1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\ \max(c[i - 1, j], c[i, j - 1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j \end{cases} \quad (2.1)$$

where $c[i, j]$ is the length of the LCS up to x_i and y_j .

To produce an algorithm that runs in $O(n^2)$ time, we construct a table of size $m * n$ and store the LCS of X_i and Y_j at $c[i, j]$. At each table entry, we also store which of the three cases we have used, indicating which "direction" to take in the back-traversal of the table to determine the LCS. Figure 2.1 shows an example of a dynamic programming table used to find the LCS of two sequences.

We can extend this algorithm to incorporate a cost function. In order to do this, we define an indel as follows. An indel represents an insertion or deletion of residues in a sequence. When aligning two sequences, we come across certain residues that we will decide will not match, namely, we would have to skip some residues in one protein in order to come up with

| j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|-----|-------|---|---|---|---|---|---|
| i | y_j | B | D | C | A | B | A |
| 0 | x_i | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | A | 0 | ↑ | ↑ | ↑ | ← | ← |
| 2 | B | 0 | 1 | ← | ← | ↑ | ← |
| 3 | C | 0 | ↑ | ↑ | 2 | ← | ↑ |
| 4 | B | 0 | ↑ | ↑ | ↑ | ↑ | ← |
| 5 | D | 0 | ↑ | 2 | 2 | 2 | ↑ |
| 6 | A | 0 | ↑ | ↑ | ↑ | ↑ | ↑ |
| 7 | B | 0 | ↑ | ↑ | ↑ | ↑ | ↑ |

Figure 2.1: Sample dynamic programming table

an optimal alignment overall. The two sequences can be made to match if we insert indels in the lacking sequence to match up with the excess residues in the first sequence. An example is as follows, with hyphens representing indels:

```

-GCTAGGATATAG---CT
  |||  ||| |||  ||
GGGT--GAT-TAGAACT

```

We can now define an indel function $g(k) = \alpha + \beta(k - 1)$, where α represents the cost of starting a new gap and β represents continuing an old one. Then, defining functions F and E , which represent the cost of producing a gap in the first and second protein, respectively, we can produce the final algorithm we will use (Theorem 9.6 in Waterman):

Let $g(k) = \alpha + \beta(k-1)$ for constants α and β . Set $E_{0,0} = F_{0,0} = D_{0,0} = 0$, $E_{i,0} = D_{i,0} = g(i)$, and $F_{0,j} = D_{0,j} = g(j)$. If $E_{i,j}$ and $F_{i,j}$ satisfy

$$E_{i,j} = \min\{D_{i,j-1} + \alpha, E_{i,j-1} + \beta\} \quad (2.2)$$

and

$$F_{i,j} = \min\{D_{i-1,j} + \alpha, F_{i-1,j} + \beta\} \quad (2.3)$$

then

$$D_{i,j} = \min\{D_{i-1,j-1} + \phi(i,j), E_{i,j}, F_{i,j}\} \quad (2.4)$$

The challenging tasks are to define the constants α and β and to come up with a cost function ϕ that correctly assigns high costs to incorrect matches.

2.3 Design / Implementation

The cost function is vital to determining the optimal alignment. We want to reward matches or near-matches and we want to pay a penalty for misalignments. In addition, we also want to maintain 3-dimensional information for the structure. In order to do this, we would like to derive a cost function that sums up the intramolecular distances from the current node to each of the previously matched nodes.

When comparing two residues, x_i and y_j , we want to compare the intramolecular distances to all of the previously matched residues in Z_k .

$$\phi(i,j) = \frac{\sum_{r=0}^k |d(x_i, x_r) - d(y_j, y_r)|}{k} \quad (2.5)$$

where $d(a,b)$ is the intramolecular distance between a and b . The summation is normalized over k to accommodate for the number of matches made so far.

2.4 Testing

For purposes of testing, it was only necessary to look at one dimension. Since we are only concerned with intramolecular distances, a one-dimensional

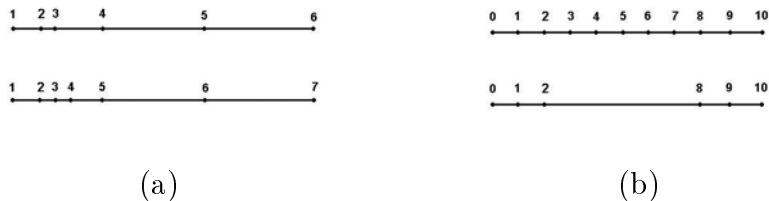


Figure 2.2: Test Cases

problem is sufficient. Any set of 3-D information is simply processed into a distance matrix, so we are reducing complexity for the test cases and not losing quality of the program for 3-D.

- Test Case 1: Aligning a protein with itself

This test case always worked as predicted. The program always aligns a protein with itself, matching every residue to the corresponding residue in the other protein.

- Test Case 2: Figure 2.2(a)

The points are distributed randomly along the x-axis, and the second protein is almost identical except that the second has an extra residue that does not match at residue 4. The algorithm successfully found this mismatch and skipped over residue 4 in the second protein when calculating the LCS.

Many modifications of this test case were also tested, including adding much more residues to the second protein and checking if the algorithm would ignore them, and extending the beginning of either protein to see if the algorithm would successfully align the beginnings. The algorithm worked as predicted in all of these test cases.

- Test Case 3: Figure 2.2(b)

The first protein has points uniformly distributed along the x-axis. The second protein has the same first three points and same last three points. Theoretically, the algorithm should match the first three, then

overlook the cost of skipping 5 in the first protein, and then match the last three residues in each respective protein. However, since we adopt a greedy dynamic programming approach to solve this problem, the algorithm does not find this match.

Let us label the first protein A and the second B. The problem comes with the perfect uniform distribution of the points in A. Because we can match B_{0-2} to any three sequential points in A, the algorithm tries to match it every iteration. Therefore, once we are comparing residues A_8 and B_8 , the algorithm looks back at A_7 and B_2 and thinks that it has found the perfect match. If α and β are small enough relative to the intramolecular distances, then the algorithm decides that it will take the penalty of not perfectly aligning A_{5-7} with B_{0-2} in order to maximize the match.

2.5 Conclusion / Future Work

The idea of this project was to extend some well-known algorithms for sequence alignment to 3-D structural alignment and determine the best constants to use in the cost functions. We have determined that although this algorithm may not always provide an optimal solution, it provides a good heuristic for finding a better algorithm. We have also discovered that determining the constants α and β is a difficult task. We have to find values that somehow relate to the intramolecular distances such that we are paying the appropriate penalty costs for skipping residues.

2.6 References

1. Cormen T, Leiserson C, Rivest R, and Stein C. **Introduction to Algorithms**. Cambridge, Massachusetts: MIT Press, 2001.
2. Holm L, Sander C: **Protein-structure comparison by alignment of distance matrices**. *J Mol Biol*, 233:123-138.
3. Koehl, P: **Protein Structure Similarities**. *Current Opinion in Structural Biology* 2001, 11:348-353.
4. Rossmann MG, Argos P: **Exploring structural homology of Proteins**. *J Mol Biol* 1976, 105:75-95.
5. Russell RB, Barton GJ: **Multiple protein-sequence alignment from tertiary structure comparison: assignment of global and residue confidence levels**. *Proteins* 1992, 14:309-323.
6. Waterman M: **Introduction to Computational Biology: Maps, Sequences and Genomes**. Chapman and Hall: CRC. 2000.

Acknowledgements

I would like to thank Magda Procopiuc for her help in understanding the STAR-Tree structure and the implementation, Sarel Har-Peled for helping me understand the road generator, and Tavi Procopiuc and many other graduate students at Duke for answering my many questions throughout the summer.

I would like to also thank Pankaj K Agarwal for continued support and guidance throughout this study.

Code

- `guigl.cc` - The OpenGL implementation of STAR-Tree. This calls on many other files which are related to the STAR-Tree structure which are not included here but are available upon request: `jdt@cs.duke.edu`
- `join.cc` - Road simplification program that merges connected roads so we have fewer total roads to work with
- `Protein.h` and `Protein.cpp` - Store a protein's vertices and calculates the distance matrix
- `main.cpp` - Performs the optimization and outputs the LCS