

**A C++ IMPLEMENTATION OF A
POLYNOMIAL TIME APPROXIMATION SCHEME
FOR ALIGNING PROTEIN STRUCTURES**

BY

CHOR YUN JIA

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Perak Campus)

JAN 2013

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: A C++ IMPLEMENTATION OF A POLYNOMIAL TIME APPROXIMATION SCHEME FOR ALIGNING PROTEIN STRUCTURES

Academic Session: JANUARY 2013

I CHOR YUN JIA
(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in
Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

(Author's signature)

(Supervisor's signature)

Address:

74, TAMAN KODIANG,
06100 KODIANG,
KEDAH.

DR. NG YEN KAOW
Supervisor's name

Date: 8/4/2013

Date: _____

**A C++ IMPLEMENTATION OF A
POLYNOMIAL TIME APPROXIMATION SCHEME
FOR ALIGNING PROTEIN STRUCTURES**

BY

CHOR YUN JIA

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF COMPUTER SCIENCE (HONS)

Faculty of Information and Communication Technology

(Perak Campus)

JAN 2013

DECLARATION OF ORIGINALITY

I declare that this report entitled “**A C++ IMPLEMENTATION OF A POLYNOMIAL TIME APPROXIMATION SCHEME FOR ALIGNING PROTEIN STRUCTURES**”

is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature : _____

Name : CHOR YUN JIA

Date : 8/4/2013

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor for the opportunity to work on this project on protein structure alignment—my first step towards a career in bioinformatics. Thank you, Dr Ng.

This report is dedicated to my parents and my family for their love, support and continuous encouragement throughout the course.

When I asked for strength, God gave me more burdens to carry.

When I asked for love, God sent me people with problems.

When I asked for wisdom, God gave me more problems to solve.

I see that I did not get the things I asked for but I have been given all the things that I needed. Thank God.

ABSTRACT

This project studies and implements a polynomial time approximation scheme (PTAS) for aligning protein structures proposed by Li and Ng (Li and Ng, 2010). The algorithm addresses the alignment problem called the *LCP problem under bottleneck distance*. In the problem, each protein structure is modeled as a sequence of 3D points. Given two such sequences, the problem is to find a maximal subset of points from each protein and a bijection between the two subsets, under the restriction that there exists a superposition which brings each pair of points in the bijection to within a specified proximity. Whereas most current algorithms for the problem are heuristic in nature, the algorithm under study provides a solution with theoretical bounds for both runtime and accuracy.

Implementation of the algorithm is complicated by the difficult geometrical manipulations in the algorithm. The basic idea of the algorithm is as follows. Firstly, a rigid transformation is identified to superimpose the two structures. The transformation is searched in a discretized space of resolution δ – a theorem shows that the discretization will introduce discrepancies of at most 3δ . Each pair of points in Q will then be used as a rotation axis upon which P is rotated. During the rotation, the rotation angles where points in P and Q come within the required proximity are noted. At each angle, for each point in P we know a set of points in Q that it can be matched to. This set does not change for each rotation interval where no points come into or out of contact; hence the entire rotation interval can be considered similarly. For each such rotation interval, the problem is to find a bijection which matches the most points in P to Q. To solve this, a bipartite graph is constructed where the vertices are the points in P and Q, and an edge is placed between every two vertices that can be matched; a maximum bipartite matching of the graph gives us the optimal bijection. An exhaustive search of the optimal bijection for all rotation axes and rotation angles gives us the solution to the problem. The algorithm further exploits some geometrical properties as well as overlaps in the intermediate solutions to reduce runtime complexity.

This project gives a full implementation of the algorithm in C++, with very minimal use of external libraries.

TABLE OF CONTENT

TITLE	i
DECLARATION OF ORIGINALITY	ii
ACKNOWLEDGEMENTS	iii
ABSTRACT.....	iv
LIST OF FIGURES	vii
CHAPTER 1 INTRODUCTION.....	1
1.1 Motivation and Problem Statement.....	1
1.2 Project Objective and Contribution.....	2
1.3 Background information	3
CHAPTER 2 LITERATURE REVIEW.....	4
2.1 Literature Review.....	4
2.2 Polynomial Time Approximation Scheme to Implement	7
CHAPTER 3 METHODOLOGY AND TOOLS	11
3.1 Methodology and Tools	11
CHAPTER 4 ALGORITHM IMPLEMENTATION.....	12
4.1 Program Input and Output.....	12
4.2 Main Procedure of Program	13
4.2.1 Get user input.....	14
4.2.2 Read .pdb file.....	15
4.2.3 Get total length of common residue ID of two input structures (P and Q).....	16
4.2.4 Create Structure for the input structures.....	17
4.2.5 Identify radial pair candidates from P and Q	18
4.2.6 Transform structure Q so that it is along the y-axis	20
4.2.6.1 Translation	22
4.2.6.2 Rotation.....	23
4.2.6.3 Cross Product	24
4.2.6.4 Rotation axis	25
4.2.6.5 Rotation angle	28
4.2.7 Exhaustive search of positions for p_1 in discretized D_c sphere around q_1	31
4.2.8 Forming a sphere cap for p_2	34

4.2.9 Find angle that moves P in and out of contact of Q	41
4.2.10 Find the maximum bipartite matching.....	49
4.2.11 Output the maximum number of matching of both structures.....	51
4.3 Program Result.....	52
CHAPTER 5 DISCUSSION AND CONCLUSION.....	54
5.1 Discussion	54
5.1.1 Achievement.....	54
5.1.2 Implementation Issues and Challenges.....	54
5.1.2.1 Problem of Installing Cygwin.....	54
5.1.2.2 Early conceptual mistakes in implementing the algorithm.....	55
5.1.2.4 Lack of knowledge in circle-sphere intersection	57
5.1.2.4 Implementation of Hopcroft-Karp algorithm.....	57
5.1.3 Runtime Error and Memory Error	59
5.2 Conclusion.....	59
BIBLIOGRAPHY	60
APPENDIX A COMPLETE CODING OF THE WHOLE PROGRAM.....	1
A-1 main.cpp.....	A-1
A-2 pdb.h.....	A-4
A-3 pdb.cpp.....	A-5
A-4 Structure.h	A-7
A-5 Structure.cpp	A-8
A-5 MathOperation.h	A-11
A-6 MathsOperation.cpp.....	A-12
A-7 Transformation.h.....	A-15
A-8 Transformation.cpp	A-16
A-9 Graph.h.....	A-30
A-10 Graph.cpp.....	A-31
A-11 Example of .pdb file.....	A-34

LIST OF FIGURES

Figure Number	Title	Page
Figure 2.2.1	LCP Problem under bottleneck distance	7
Figure 2.2.2	Forming rotation axes with a radial pair within the discretized space	8
Figure 2.2.3	Rotation angles which move p into and out of contact with q	9
Figure 4.2.1.1	Code to Store User Input	14
Figure 4.2.2.1	Code of Function readFile	15
Figure 4.2.3.1	Code of Function matchPDB	16
Figure 4.2.4.1	Parts of codes in main function	17
Figure 4.2.5.1	Distance of P from Q must is at most D_c	18
Figure 4.2.5.2	Initial check for matching point p_1 to q_1 and p_2 to q_2	18
Figure 4.2.5.3	Code of function matchPoints	19
Figure 4.2.6.1	Translation of Q so that q_1 and q_2 are along y-axis	20
Figure 4.2.6.2	Code of function transformStruct	21
Figure 4.2.6.1.1	Code of function translateStruct	22
Figure 4.2.6.1.2	Code of function diff	22
Figure 4.2.6.2.1	Rotation of q_2	23
Figure 4.2.6.3.1	Cross Product	24
Figure 4.2.6.3.2	Code of function crossProduct	24
Figure 4.2.6.4.1	Rotation axis of q_2	25
Figure 4.2.6.4.2	Code of function rotateStruct	26
Figure 4.2.6.4.3	Code of function normalize	26
Figure 4.2.6.4.4	Rotation angle to rotates q_2 to y-axis	27
Figure 4.2.6.4.5	Law of Cosine	27
Figure 4.2.6.5.1	Calculation of rotation angle using Law of Cosine	28
Figure 4.2.6.5.2	Code of function rotateStruct	29
Figure 4.2.6.5.3	Code of function rotatesAboutVector	29

Figure 4.2.6.5.4	Code of function rotatesAboutArbLine	30
Figure 4.2.7.1	Cube for candidate positions of p_1	31
Figure 4.2.7.2	Discretization of cube of q_1	32
Figure 4.2.7.3	Sphere encapsulated in the cube	32
Figure 4.2.7.4	Code of function tryP1InGrid	33
Figure 4.2.8.1	Forming sphere cap with grid	34
Figure 4.2.8.2	Conditions leading to the formation of a sphere cap	35
Figure 4.2.8.3	Vector1 to rotate p_2	35
Figure 4.2.8.4	Code of function rotateVectorAbtVector	36
Figure 4.2.8.5	Method to form sphere cap	37
Figure 4.2.8.6	Diagram for formula of arc length	37
Figure 1.2.8.7	Code of function formSphereCap	38
Figure 4.2.8.8	Code of function formSphereCap	39
Figure 4.2.8.9	Code of function movePtoPlace	40
Figure 4.2.9.1	Angle that moves p in and out of contact with q	41
Figure 4.2.9.2	Code of function findAngleInOut	41
Figure 4.2.9.3	Intersection of circle and sphere	42
Figure 4.2.9.4	Intersection of a plane and circle	43
Figure 4.2.9.5	Intersection of two circles	44
Figure 4.2.9.7	Angle that moves p into contact of q	45
Figure 4.2.9.8	Angle that moves p out of contact of q	46
Figure 4.2.9.9	Code of function getIntersectPoint	47
Figure 4.2.9.10	Code of function getIntersectPoint	48
Figure 4.2.9.11	First θ_1 and θ_2 combination	49
Figure 4.2.9.12	Angle rearrangement	49
Figure 5.1.2.2.1	Wrong concept of finding radial pair of P to match Q	55
Figure 5.1.2.2.22	Wrong concept of forming sphere cap	56

CHAPTER 1 INTRODUCTION

1.1 Motivation and Problem Statement

Proteins are the most important molecules in all living organisms. It functions as enzymes, sensors, antibodies, transporters, among myriads of other roles. To perform their biological function, proteins will fold into one or more specific spatial conformations, driven by some non-covalent interactions. The functions performed are frequently determined by the molecular shape of a protein, and it has been observed that proteins of similar 3D structures are likely to function similarly. This allows us to predict the functions of a protein by comparing its three-dimensional structure to the proteins of known functions.

For this reason, there have been very intense researches on protein structure comparison. Consequently, a large number of approaches, algorithms and software tools have been developed to evaluate the similarity between the 3D proteins structures, under the topic name of Protein Structure Alignment (Li & Ng 2010).

Though there are many protein structure comparison methods, all these methods are heuristic in nature. That is, the principles which the methods are based on common-sense or past experiences, rather than strict mathematical results. For example, they may use readily accessible but loosely applicable information to increase the speed of finding a satisfactory solution. Heuristic methods provide no guarantee on their runtime or the accuracy of their outputs. Since the accuracy of the structure alignment is very important, there is a need to devise methods which guarantees a specified level of accuracy in its output. Such an algorithm to align the protein structures *under the bottleneck distance* for protein structures is given by Li and Ng (Li & Ng 2010). The algorithm looks for an optimal superposition of the two protein structures in a discretized space to approximate the solution, through the *radial pairs* (or *radial axes*) introduced in (Li, Bu, Xu & Li 2008).

1.2 Project Objective and Contribution

The implementation of the algorithm for the LCP problem under the bottleneck distance program will result in a first tool for aligning protein structures with a guarantee in its accuracy. This will help researchers determine such alignments accurately in cases where such accuracy is important – for example, when there is dispute with the results from two different alignment methods.

The technique of radial pair used in the technique has proven to be very useful, and has been successfully used to devise algorithms for so-called *model comparison* and *local continuous segments*. To correctly implement the techniques involved in the use of these radial pairs, e.g. the discretization and the rotation axis, is a non-trivial task which requires days, or even weeks, for a programmer. Hence, the implementation of radial pair library is important besides provide standard routines in the manipulation of the radial pairs.

The availability of the radial pair library will be very useful for the implementations of other algorithms which require them. The ease of programming radial pairs using the library will also help in testing new algorithms which makes use of them, spurring faster development of algorithms in structural alignment.

1.3 Background information

The main aim in the study of Protein Structure Alignment is to establish the similarities of two or more polymer structures according to their shapes and three-dimensional conformation. Two important problems studied are:

1. *Structure comparison*, which is to analyze the similarities and differences between two given structures, while
2. *Structure alignment*, which is to establish the equivalence between the individual atoms of two given structures.

Specifically, structure alignment requires no prior knowledge on the equivalence of the structures. Thus, structure alignment is very useful for evaluating structures that are predicted purely from protein sequences, that is, through *ab initio* protein structure prediction methods (Zhang & Skolnick 2005). Structure alignment is also very useful in finding the evolutionary relationship between the proteins which have only a small number of identical sequences, that is, where the evolutionary relationship is difficult to be determined by sequence alignment.

The output expected from a structural alignment typically includes (1) the equivalence between the atoms of the input structures, (2) a minimal root mean square deviation (RMSD) between the structures. The RMSD measures the similarity between the two structures under the best possible superposition of the two structures. The changes in domain relative orientation between two structures can unnaturally increase the RMSD. Hence, structural alignment can be complicated because of the multiple protein domains in one or more input structures.

CHAPTER 2 LITERATURE REVIEW

2.1 Literature Review

Many methods for comparing two protein structures are based on the three following steps. First, detect the common similarities of the two protein structures. Then, align the structures based on such similarities. Lastly, compute a statistical measure on the similarities. Most of these methods do well in recognizing observable similarities between the protein structures. However, it is difficult to align two or more structures. The accuracy may depend on the method used or what the user is about to achieve (Martin, Capriotti, Shindyalov & Bourne 2009). The protein structure alignment methods that are widely used and cited are DALI (Holm and Sander, 1993b), Combinatorial Extension (CE) (Shindyalov and Bourne, 1998), MAMMOTH (Ortiz, Strauss, and Olmea, 2002), RAPIDO (Mosca, Schneider TR.), SALIGN (Sali and Blundell, 1990) and SSAP (Orengo and Taylor, 1996). There are many other new methods that have been published and used. For instance, SABERTOOTH, TOPOFIT, SARF2 and ProBis are some known structure alignment methods in the world.

DALI, which is a distance alignment matrix method, represents each structure as a distance matrix. Proteins are transformed into 2D arrays of distance between all $C\alpha$ atoms (Martin, Capriotti, Shindyalov & Bourne 2009). The method breaks the structures into hexapeptide fragments and generates a comparison distance matrix by evaluating the contact patterns between those successive fragments to align the two structures (Holm & Sander 1996). The matrix diagonal consists of secondary structure features that involve contiguous regions in the sequence's residues. Other diagonals reflect spatial contacts between the residues that are far from each other in the sequence. The features they represent are parallel if these diagonals are parallel to the main diagonals. On the other hand, their features are anti-parallel whenever they are perpendicular. The features in the square matrix are symmetrical about the main diagonal.

Combinatorial extension (CE) is a method similar to DALI, which breaks structure into fragments before reassembling into a complete alignment. AFPs (aligned fragment pairs) which are a series of pair-wise combinations of fragments are used to identify a similarity matrix through which an optimal path is generated to determine the

final alignment. In order to reduce unnecessary search space and increase the efficiency, only AFPs which satisfy the criteria for local similarity are included in the matrix (Shindyalov & Bourne 1998). Some similarity metrics are possible. CE identifies most alike AFPs between the compared structures by using combinatorial extension algorithm. To increase the performance of the process, the size of each AFP must be within 30 residues distance to the current alignment ends (Martin, Capriotti, Shindyalov & Bourne 2009) and maximum gap size are set to empirically determined value of 8 (Shindyalov & Bourne 1998).

MAMMOTH (MAtching Molecular Models Obtained from THeory) transforms coordinates of the protein structure into a set of six unit-vectors which are computed from the $C\alpha$ trace of the consecutive heptamers. This method decomposes protein structures into short heptapeptides to compare with the heptapeptides of other proteins. It obtains a similarity score between the heptapeptides and store in a similarity matrix by using a unit-vector RMS, which is known as URMS method (Kedem, Chew & Elber 1999). With hybrid dynamic programming, it computes the optimal residue alignment over the similarity matrix. Lastly, identify the largest local structure alignment within a given RMSD threshold by using the non-constant heuristic that is implemented in MaxSub (Martin, Capriotti, Shindyalov & Bourne 2009).

RAPIDO, the Rapid Alignment of Proteins In terms of Domains, is actually a web server for crystal structures of different protein molecules in 3D alignment. It identifies fragments that have similar structure in two proteins based on the difference distance matrices. The Matching Fragment Pairs (MFPs) are represented as nodes in a graph. The MFPs are chained together to form an alignment through an algorithm to identify the longest path on a Directed Acyclic Graph (DAG). After finish the alignments of the two structures, the server identify conformational invariant regions by a genetic algorithm (Jogn & Wendy 2004).

SALIGN compares structure properties by calculating the 3D coordinates of two or more proteins. This method aligns the coordinates by dynamic programming. SALIGN represents proteins by a set of properties or features which is calculated from protein sequences and structures, or arbitrarily defined by the user. SALIGN can be applied to align three or more protein structures. There are two approaches of SALIGN, which are a

tree-based or a progressive alignment. Progressive alignment has less intensive than the tree-based alignment (Martin, Capriotti, Shindyalow & Bourne 2009).

The **SSAP** (Sequential Structure Alignment Program) algorithm uses a double dynamic programming optimizer to compare 3D structures based on the atom-to-atom vectors. Instead of using alpha carbons, SSAP uses the C_{β} atoms to generate a set of vectors for all residues except for glycine, a dummy C_{β} is used. A series of inter-residue distance vectors between every residue and its closest non-contiguous neighbours on a protein is constructed. Then, the matrices which consist of the vector differences between C_{β} vectors are constructed. By applying the dynamic programming on each resulting matrix, a set of selected matching position is defined. The final matrix is obtained by comparing vectors between C_{β} atoms at pairs of same protein to the C_{β} atoms from the selected matching position. Then, the final structural alignment is computed over the matrix of scores S by second dynamic programming (Martin, Capriotti, Shindyalow & Bourne 2009).

2.2 Polynomial Time Approximation Scheme to Implement

As stated before, the methods discussed previously are heuristic in nature. Since the accuracy of the structure alignment is very important, there is a need to devise methods which guarantees a specified level of accuracy in its output. Such an algorithm to align the protein structures *under the bottleneck distance* for protein structures is given by Li and Ng (Li & Ng 2010). The algorithm looks for an optimal superposition of the two protein structures in a discretized space to approximate the solution, through the *radial pairs* (or *radial axes*) introduced in (Li, Bu, Xu & Li 2008).

We first describe the alignment problem which the algorithm aims to solve. Given two protein structures, the problem is to find a subset of points from each protein and a bijective matching of the points between the two proteins, with the objective of maximizing the number of pairs of points. The resultant problem is called the largest common point set problem (LCP), which is formally stated in Figure 2.2.1. In the formulation of the problem, each residue in the protein structure is represented as a point in \mathbb{R}^3 .

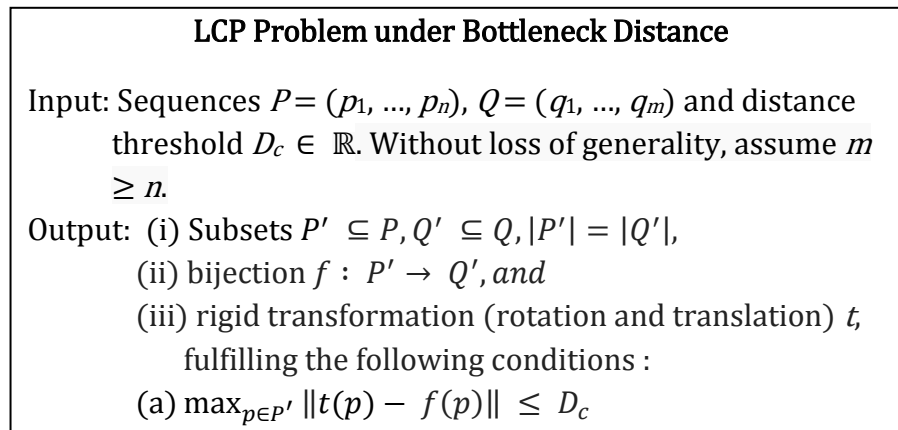


Figure 2.2.1: LCP Problem under bottleneck distance (Li & Ng 2010)

We let $\mathbf{P}, \mathbf{Q}, f, \mathbf{T}, S$ denote an optimal P', Q', f, t, S , respectively. Assume that a pair of points $p_1, p_2 \in \mathbf{P}$ is known for the problem. We consider \mathbf{T} as consisting of two parts: (1) an initial transformation T which transforms p_1, p_2 into their positions under \mathbf{T} , and (2) a rotation R about an axis through the points $T(p_1)$ and $T(p_2)$, such that $\forall p \in \mathbf{P}, R(T(p)) = \mathbf{T}(p)$.

Let P and Q be two given structures. A rigid transformation T is to be finding to superimpose the two protein structures.

Assume that two points $p_1, p_2 \in P$, and their corresponding point $q_1, q_2 \in Q$ are known. Let T define an axis where a rotation will complete the transformation T . All possible coordinates for p_1 and p_2 are examined in a discretized space in order to find T . T , however, may not be within the discretized space. Let T be a transformation within the discretized space which best approximates T . We will show that when p_1 and p_2 fulfills certain properties, the error introduced by T can be bounded by the resolution of the discretization.

Definition 1 (Radial pair (Li, Bu, Xu & Li 2008)). Given $p, p' \in P$. We call p and p' a *radial pair* iff p' is the furthest point from p among all the points in P , which is written as $\langle [p]p' \rangle_p$. Note that $\langle [p]p' \rangle_p$ does not imply $\langle [p']p \rangle_{p'}$.

Lemma 2 (Li, Bu, Xu & Li 2008). Given a set of points P , a rigid transformation T , T and radial pair $p_1, p_2 \in P$. If $\|T(p_1) - T(p_1)\| \leq \delta$ and $\|T(p_2) - T(p_2)\| \leq \delta$, then there exists a rotation R about an axis through the points $T(p_1)$ and $T(p_2)$, such that $\forall p \in P, \|R(T(p)) - T(p)\| \leq 3\delta$.

Hence an error of δ introduced by T on p_1 and p_2 with respect to T will at most introduce an error of 3δ on the rest of the points in P , and hence it suffices that we search for the transformation in a discretized space of resolution δ to obtain a solution of accuracy up to 3δ . We now discuss how to perform this search in the discretized space.

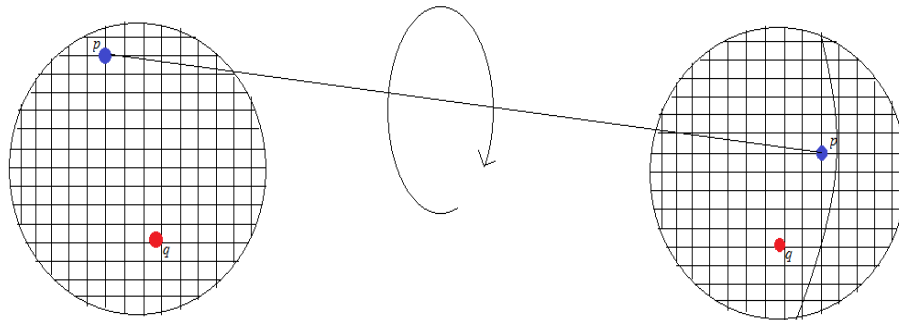


Figure 2.2.2: Forming rotation axes with a radial pair within the discretized space

Assume that a radial pair p_i, p_j of P match respectively to q_k, q_l of Q . Since we require that $\|p_i - p_j\| \leq D_c$ and $\|q_k - q_l\| \leq D_c$, it suffices that we examine the coordinates which fulfill these conditions. Then, the sphere of q_k is discretized using cubes of size length $\delta = \epsilon D_c / 3$. Every cube corresponds to a grid point in the discretized space. After fixing p_i at a grid point, all possible positions of p_j must be placed on a sphere cap at center p_i and radius $\|p_i - p_j\|$, as illustrated in Figure 2.2.2. There are $O(\frac{1}{\epsilon^3})$ grid points in the sphere and $O(\frac{1}{\epsilon^2})$ grid points in the sphere cap, giving us a total of $O(\frac{1}{\epsilon^5})$ rotation axes. That is,

Lemma 3 (Li, Bu, Xu & Li 2008). If it is known that $p_i, p_j \in P$ is a radial pair, and that $f(p_i) = q_k$ and $f(p_j) = q_l$ for $q_k, q_l \in Q$, then one only needs to search among $O(\frac{1}{\epsilon^5})$ transformations to find a transformation T which will result in at most ϵD_c difference to each $p \in P$ transformed by T , by way of Lemma 2.

With a rotation axis fixed, we find the angle $\theta \in [0, 2\pi]$ about the axis through p_i and p_j such that the score S is maximized. For a given $p \in P$ and $q \in Q$, we find the rotation angles that move p into and out a distance of $((1 + \epsilon)D_c)$ from q , as shown in Figure 3. The angle θ_1 moves p into within a distance of $((1 + \epsilon)D_c)$ from q , while the angle θ_2 moves p out from within a distance of $((1 + \epsilon)D_c)$ from q . Within the angles θ_1 and θ_2 , p may be matched to q while outside of the angles, p cannot be matched to q .

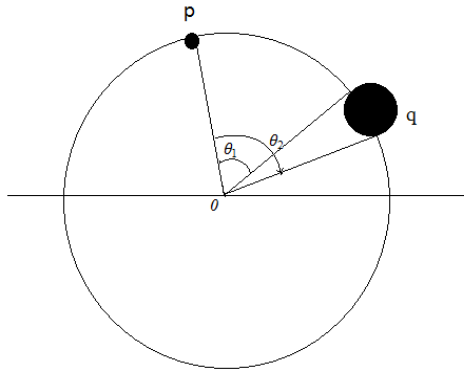


Figure 2.2.3: Rotation angles which move p into and out of contact with q

Given a rotation axis and a rotation angle, we want to find the maximum number of one-one matches between the points in P and Q . To do so we construct a graph $G = (V,$

E). Each node in the graph represents a point in either P or Q . We place an edge between two nodes if and only if at that angle, the point they represent can be matched. That is, an edge $(u, v) \in E$ iff $u \in P, v \in Q$ and $\|u - v\| \leq (1 + \epsilon) D_c$. It is clear that G is bipartite. To find the maximum one-one matches, it suffices that we find the maximum bipartite matching of G (Schneider 2002). It is clear that for any angle in $\theta \in [0, 2\pi]$ which does not correspond to an angle where a point p comes into, or out of contact with some point q , the bipartite graph created will be the same as one where the last change in point contact occurs. Hence, we only need to construct bipartite graphs for the angles where points come into, or out of contact. The angles which define these events divide $[0, 2\pi]$ into mn intervals. We sort the angles that define the rotation intervals in order. The bipartite graph constructed for each subsequent interval differs by at most an edge, and the bipartite matching for one graph can be used as an initial matching for the subsequent graph (Li & Ng 2010), allowing the matching to be performed in $O(1)$ time. The time required for the first bipartite matching, $O(m^{2.5})$, while each of the $O(mn)$ subsequent bipartite matching requires $O(mn)$ time, resulting in a total time complexity of $O(m^{2.5} + m^2n^2)$.

Hence, we are able to determine the maximum number of matches in $O(m^{2.5} + m^2n^2)$ time when a rotation axis is given. Given our discretization scheme, there are at most $O(\frac{1}{\epsilon^5})$ rotation axes given a radial pair (Lemma 3). However, we do not know which pair is a radial pair of P , nor do we know their matching points in Q . For this reason we exhaustively search all the possible m^2n^2 combinations of pairs of points in P and Q . This gives us the following result.

Theorem 1 (Li, Bu, Xu & Li 2008). There is an algorithm of time complexity $O((n^2m^{4.5} + n^4m^4)/\epsilon^5)$ for the LCP problem under bottleneck distance.

CHAPTER 3 METHODOLOGY AND TOOLS

3.1 Methodology and Tools

My choice of language to implement the algorithm is C++. The intermediate-level language feature is the reason I chosen C++. It comprises the combination of high-level and low-level language features (Herbert 1998).

For the present work, C++ is chosen based on consideration of performance and versatility. My first consideration is speed, since the algorithm has a runtime of a very high polynomial. Due to the intermediate level nature of the C++ language, compiled codes in C++ remain to date faster than those written in most other languages.

Part of the aim of the present work is to realize a library for the manipulation of radial pairs. A major consideration for library codes is the ability to make the codes run under different environments and software development platforms. C++ is perfect for this purpose. A number of interpreters of other programming languages are implemented in C++ language themselves.

To automate the experimentation, I choose to use the UNIX operating system for this project. UNIX also supports many system level programming calls for controlling processes, files, and access rights, such as those to prevent programs from simultaneously accessing the same resource. UNIX is also a natural candidate for program automation, due to its simplicity in the use of plain text and files for inter-process communication (IPC), as well as a wealth of software tools for processing textual data, which can be easily strung together through a command line interpreter using pipes. The use of UNIX shells naturally adds control structures for scripting the software tools. The Bash shell is chosen for this project.

CHAPTER 4 ALGORITHM IMPLEMENTATION

4.1 Program Input and Output

Input of the program:

- Protein Structure 1 file (.pdb)
- Protein Structure 2 file (.pdb)
- Value of D_c
- Value of ϵ

Output of the program:

- Maximum number of residue matched.

4.2 Main Procedure of Program

The main procedure of the program is as follows:

1. Get user input.
2. Read .pdb file.
3. Get total length of common residue ID of two input structures (P and Q).
4. Create Structure for the input structures.
5. Identify candidate radial pairs: p_1, p_2 from P , and corresponding q_1, q_2 from Q .
6. Transform structure Q so that q_1 and q_2 are along the y-axis.
7. Discretize a sphere of radius D_c centered at q_1 as candidate positions for p_1 .
8. Form discretized sphere cap centered at q_2 as candidate positions for p_2 .
9. For each p in P and each q in Q , find the angles that move p in and out of contact of q .
10. For each angular interval where there are no changes in the points in P and Q that may come into or out of contact, find for each point p in P the set of points in Q that it may be matched to.
11. Construct a bipartite graph out of the contact information obtained for the angular interval in (10).
12. Find the maximum bipartite matching of the bipartite graph in (11).
13. Repeat for each candidate radial pair.
14. Output the maximum number of matching of both structures.

The geometrical manipulations to achieve each of these steps are detailed in the following.

4.2.1 Get user input

```
Dc = atof(argv[3]);  
epsilon = atof(argv[4]);  
  
pdb* nativePdb = new pdb(argv[1]);  
pdb* modelPdb = new pdb(argv[2]);
```

Figure 4.2.1.1: Code to Store User Input

Input from the user is specified through command line arguments passed to the program. The first input argument is to be the native structure (structure Q); the second input argument is to be the model structure (structure P). The third input argument is to be the value of D_c while the last input argument is to be the value of ϵ .

The sample input:

```
a.pdb b.pdb 0.1 2
```


4.2.2 Read .pdb file

The protein structure file (.pdb) will be read. The total number of residues in the file, residue IDs and also $C\alpha$'s x, y and z coordinate will be read.

```

void pdb::readFile(){
    FILE *fInput=fopen(mPDBFile, "r");
    if(!fInput)
    {
        cerr << "Unsuccessfully open protein file " << mPDBFile << " !"
<< endl;
        exit(0);
    }

    char temp[80];
    double x,y,z;
    string residueName;

    mNumOfResidue = 0;

    while ( fgets ( temp, sizeof(temp), fInput ) != NULL )
    {
        string line;
        line = temp;
        if(line.substr(0, 6) == "ENDMDL")
            break;
        if(line.substr(0, 6) != "ATOM  ")
            continue;
        if(line.substr(12,4) == " CA " || line.substr(12,4) == "CA  ")
        || line.substr(12,4) == " CA")
        {
            x = toDouble(line.substr(30,8));
            y = toDouble(line.substr(38,8));
            z = toDouble(line.substr(46,8));

            mCAlpha[mNumOfResidue*3] = x;
            mCAlpha[mNumOfResidue*3+1] = y;
            mCAlpha[mNumOfResidue*3+2] = z;

            residueName = line.substr(17,3);
            mresidueID[mNumOfResidue] = toInt(line.substr(22,4));

            mNumOfResidue++;
        }
    }
    fclose(fInput);
}

```

Figure 4.2.2.1: Code of Function readFile

Only $C\alpha$ atom's x, y and z coordinates will be needed. Thus the corresponding x, y and z will be stored when the 12th–13th letters of each line is "CA". The residueID is taken from the 22nd character of each line.

4.2.3 Get total length of common residue ID of two input structures (*P* and *Q*)

```

int matchPDB(pdb* native, pdb* model)
{
    int natLength = native->mNumOfResidue;
    int modLength = model->mNumOfResidue;

    int* resNative = native->mresidueID;
    int* resModel = model->mresidueID;
    int j = 0, k = 0;
    for(int i = 0; i < natLength; i++)
    {
        for(; j < modLength; j++)
        {
            if(resNative[i] == resModel[j])
            {
                mIndexNative[k] = i;
                mIndexModel[k] = j;
                k++;
                break;
            }
            else if(resNative[i] < resModel[j])
                break;
        }
    }

    if(k == 0)
    {
        cout << "There is no common residues in the input Structures" <<
endl;
        exit(0);
    }

    return k;
}

```

Figure 4.2.3.1: Code of Function matchPDB

In this function, the total number of common residueID of both structures and the index of the common residue will be stored. The coordinates of the common residues will be copied to create two Structure objects as shown in Figure 4.2.4.1.

4.2.4 Create Structure for the input structures

```

int alignLength = matchPDB(nativePdb, modelPdb);
double* nativeCoord = new double[alignLength*3];
double* modelCoord = new double[alignLength*3];

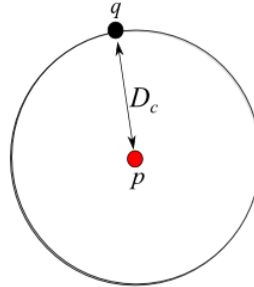
//copy coordinates of common residues.
for(int i = 0; i < alignLength; i++)
{
    //Native
    nativeCoord[i*3] = nativePdb->mCAlpha[mIndexNative[i]*3]; //x
    nativeCoord[i*3+1] = nativePdb->mCAlpha[mIndexNative[i]*3+1]; //y
    nativeCoord[i*3+2] = nativePdb->mCAlpha[mIndexNative[i]*3+2]; //z

    //Model
    modelCoord[i*3] = modelPdb->mCAlpha[mIndexModel[i]*3]; //x
    modelCoord[i*3+1] = modelPdb->mCAlpha[mIndexModel[i]*3+1]; //y
    modelCoord[i*3+2] = modelPdb->mCAlpha[mIndexModel[i]*3+2]; //z
}

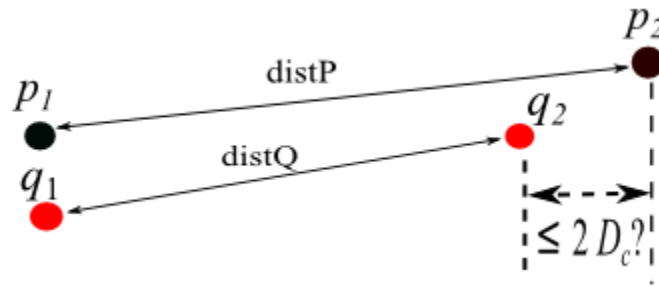
Structure* mNative = new Structure(nativeCoord, alignLength*3);
Structure* mModel = new Structure(modelCoord, alignLength*3);

```

Figure 4.2.4.1: Parts of codes in main function

4.2.5 Identify radial pair candidates from P and Q Figure 4.2.5.1: Distance of p from q must be at most D_c

The condition for a point $p \in P$ to be matched to $q \in Q$ is that the distance between p and q must be within D_c (as shown in Figure 4.2.5.1). Thus, a prerequisite for a pair $p_1, p_2 \in P$ to be matched to $q_1, q_2 \in Q$ is for $|\text{distP} - \text{distQ}| \leq 2D_c$, where $\text{distP} = \|p_1 - p_2\|$ and $\text{distQ} = \|q_1 - q_2\|$. In the program, $|\text{distP} - \text{distQ}| \leq 2D_c$ is checked (as shown in Figure 4.2.5.2) before the combination of matching point p_1 to q_1 and p_2 to q_2 proceeds to the next step.

Figure 4.2.5.2: Initial check for matching point p_1 to q_1 and p_2 to q_2

```

void Transformation::matchPoints(int iNative, int jNative, int iModel, int jModel)
{
    //Translate p1 and p2 to match q1 and q2.
    //iNative = q1    iModel = p1
    //jNative = q2    jModel = p2

    //-----Translation to match model to native-----
    -----
    double distP = calDist(mModel->mCoord + iModel, mModel->mCoord + jModel);
    double distQ = calDist(mNative->mCoord + iNative, mNative->mCoord + jNative);

    if(fabs(distP - distQ) <= 2*mDc)
    {

        //Transform Q structure such that q1 and q2 is along y-axis
        mNative->transformStruct(mNative->mCoord + iNative, mNative->mCoord +
jNative);

        //Discretize q1 and try p1 in the grid
        tryP1InGrid(mModel->mCoord + iModel, mNative->mCoord + iNative, mModel-
>mCoord + jModel, mNative->mCoord + jNative);

    }
}

```

Figure 4.2.5.3: Code of function matchPoints

4.2.6 Transform structure Q so that it is along the y-axis

The program now proceeds with the assumption that p_1 is matched to q_1 and p_2 is matched to q_2 . To simplify subsequent geometrical manipulations, I transform the structure Q so that q_1 and q_2 lies on the y-axis, and q_1 is located at the origin (as shown in Figure 4.2.6.1).

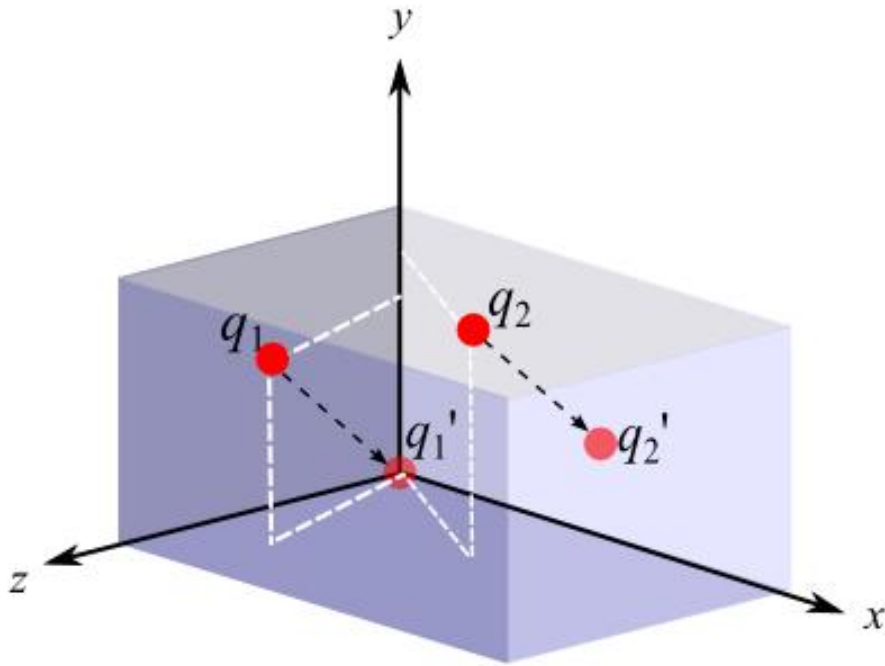


Figure 4.2.6.1: Translation of Q so that q_1 and q_2 are along the y-axis

```

/*
    Transform structure such that two points is along y-axis
*/
void Structure::transformStruct(double* q1, double* q2)
{
    double* transStep = new double[3];
    bool q2IsOrigin = false;

    //when one of the point is at origin, no need to do translation, just
perform rotation.
    if((q1[0] == 0 && q1[1] == 0 && q1[2] == 0))
    {
        ;
    }
    else if(q2[0] == 0 && q2[1] == 0 && q2[2] == 0)
    {
        q2IsOrigin = true;
    }
    else
    {
        //get the translation step of q1 to origin.
        transStep = q1;

        //translate whole structure so that q1 is at origin.
        translateStruct(transStep);
    }

    if(q2IsOrigin) //ONLY if q2 is origin, we rotate q1.
    {
        rotateStruct(q1);
    }
    else
    {
        //Rotates whole structure so that q2 is on y-axis.
        rotateStruct(q2);
    }
}

```

Figure 4.2.6.2: Code of function transformStruct

4.2.6.1 Translation

To perform the translation, I first obtain the translation which moves q_1 to the origin, and apply the translation on the remaining points of the structure. The procedure is skipped when q_1 or q_2 is already located at the origin.

```
/*  
    Translate the whole structure  
*/  
void Structure::translateStruct(double* transStep)  
{  
    for(int i = 0; i < mLength; i += 3)  
    {  
        diff(mCoord + i, transStep, mCoord + i);  
    }  
}
```

Figure 4.2.6.1.1: Code of function translateStruct

```
void diff(double* A, double* B, double *ans)  
{  
    ans[0] = A[0] - B[0];  
    ans[1] = A[1] - B[1];  
    ans[2] = A[2] - B[2];  
}
```

Figure 4.2.6.1.2: Code of function diff

4.2.6.2 Rotation

If q_2 is at origin, q_1 is rotated so that q_1 it lies on the y-axis. Otherwise, q_2 will be rotated so that it is on the y-axis as shown in Figure 4.2.6.2.1.

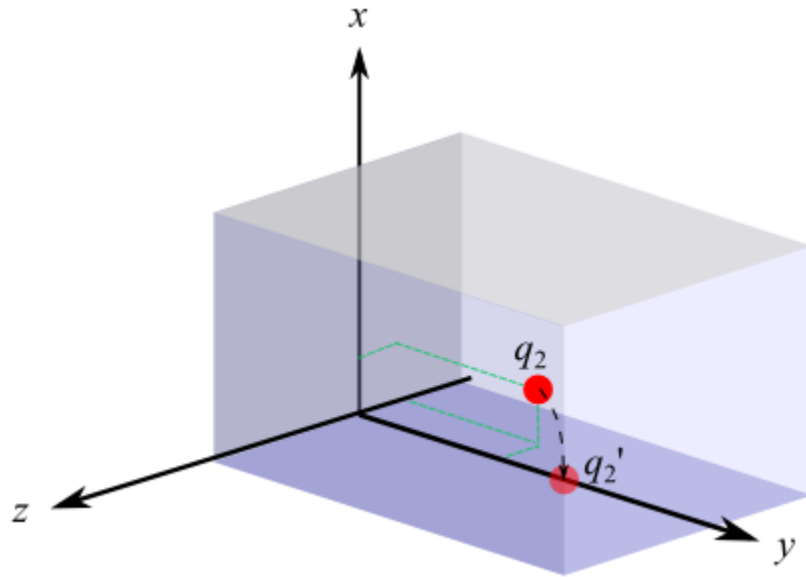


Figure 4.2.6.2.1: Rotation of q_2

To rotate q_2 to its position along the y-axis, the rotation axis must first be determined. This rotation axis can be found by using the cross product. The result of cross product is a vector that is perpendicular to both of the vectors that are being multiplied.

4.2.6.3 Cross Product

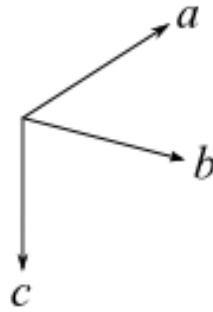


Figure 4.2.6.3.1: Cross Product

Formula of cross product:

$$\mathbf{A} = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

$$\mathbf{B} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

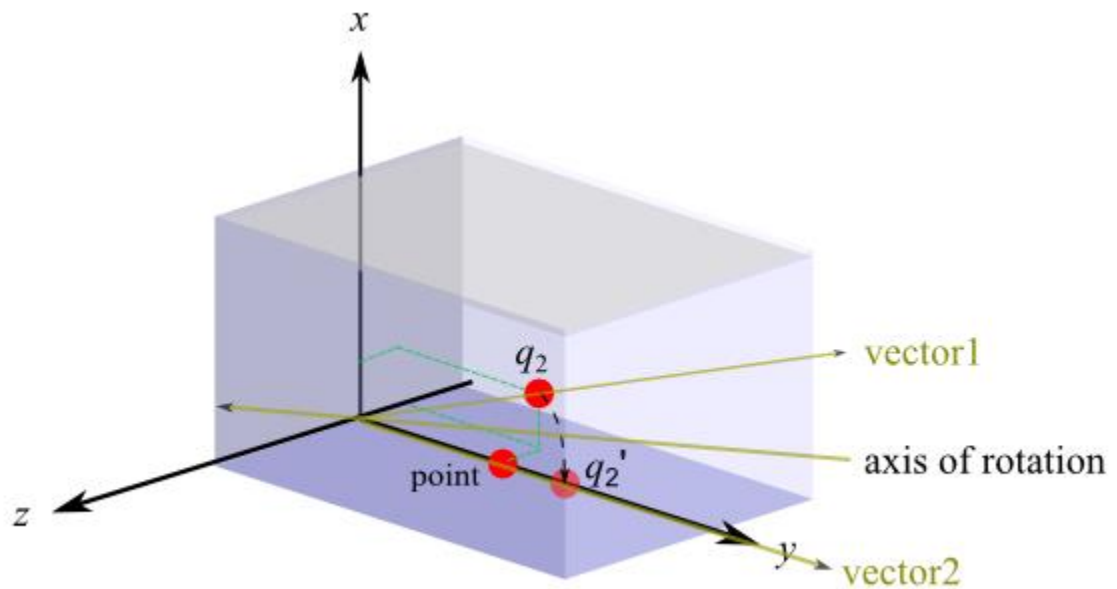
$$\mathbf{C} = \mathbf{A} \times \mathbf{B}$$

$$i.e. \begin{bmatrix} c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

```
void crossProduct(double* A, double* B, double* ans)
{
    ans[0] = A[1]*B[2] - A[2]*B[1];
    ans[1] = A[2]*B[0] - A[0]*B[2];
    ans[2] = A[0]*B[1] - A[1]*B[0];
}
```

Figure 4.2.6.3.2: Code of function crossProduct

4.2.6.4 Rotation axis

Figure 4.2.6.4.1: Rotation axis of q_2

To get the rotation axis, cross product is used. The rotation axis is a vector that is the cross product of *vector1* and *vector2*, where *vector1* is the vector of two points (q_2 and origin) and *vector2* is the vector of two points (*point* and origin). Note that *point* is a point on y-axis where its value of y is taken from q_2 . For example, if $q_2 = (3, 5, 2)$, then *point* = (0, 5, 0). After obtaining the cross product, *vector3*, I normalize the vector to make it a unit vector.

The next step is to find the rotation angle on our new rotation axis which will rotate q_2 into the y-axis.

```

/*
    Rotate structure Q so that q1 and q2 are along y-axis
*/
void Structure::rotateStruct(double* q2)
{
    double a, b, c;
    double origin[3] = {0, 0, 0};
    double cosC, theta;
    double point[3] = {0, q2[1], 0};
    double vector1[3], vector2[3], vector3[3];

    //get vector1 (a), vector2 (b), vector3 (c), where c = a x b cross
    product, c = axis of rotation

    diff(q2, origin, vector1); //a
    diff(point, origin, vector2); //b
    crossProduct(vector1, vector2, vector3); //c:axis of rotation

    normalize(vector3);

    //get angle of rotation using law of cosines
    a = calDist(q2, origin);
    b = calDist(point, origin);
    c = calDist(q2, point);

    cosC = (a*a + b*b - c*c) / (2*a*b);

    if(cosC > 1)
        cosC = 1;
    if(cosC < -1)
        cosC = -1;

    theta = acos(cosC);

    rotateStruct(origin, vector3, theta);
}

```

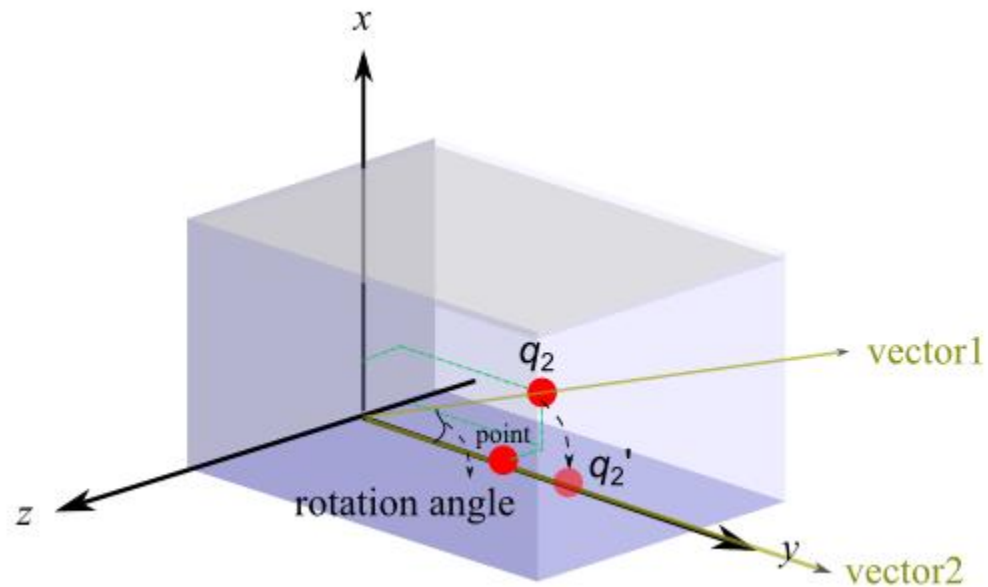
Figure 4.2.6.4.2: Code of function rotateStruct

```

void normalize(double* u)
{
    double norm = sqrt(u[0]*u[0] + u[1]*u[1] + u[2]*u[2]);
    u[0] = u[0]/norm;
    u[1] = u[1]/norm;
    u[2] = u[2]/norm;
}

```

Figure 4.2.6.4.3: Code of function normalize

Figure 4.2.6.4.4: Rotation angle to rotate q_2 to y-axis

To find this rotation angle as shown in Figure 4.2.6.4.4, the Law of Cosine (Weisstein, Eric, n.d.) is used. The Law of Cosine is shown in Figure 4.2.6.4.5.

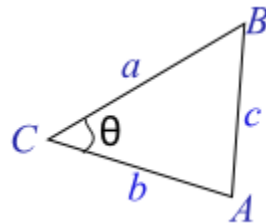


Figure 4.2.6.4.5: Law of Cosine

Formula of Law of Cosine:

$$\cos C = \frac{a^2 + b^2 - c^2}{2ab}$$

4.2.6.5 Rotation angle

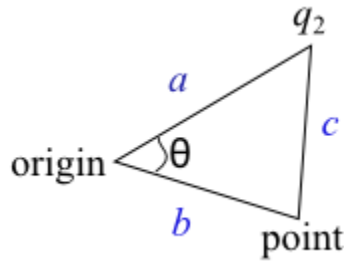


Figure 4.2.6.5.1: Calculation of rotation angle using Law of Cosine

In order to calculate the angle of rotation, I calculate the distance between the *origin* and *q₂* as *a*, the distance between *origin* and *point* as *b* and the distance between *point* and *q₂* as *c*. The angle of rotation can then be obtained from:

$$\cos C = \frac{a^2 + b^2 - c^2}{2ab}$$

$$\text{angle of rotation} = \cos^{-1}(\cos C)$$

Finally, *Q* is transformed using the determined rotation axis and rotation angle. The rotation is performed using a rotation matrix. That is, new coordinates for each point in *Q* is computed through multiplication with the matrix. The resultant coordinates can be written in the following functional form: For a point (*x*, *y*, *z*) about a line through (*a*, *b*, *c*) with direction unit vector $\langle u, v, w \rangle$, where $u + v + w = 1$, by the angle θ (Glenn 2011),

$f(x, y, z, a, b, c, u, v, w, \theta) =$

$$\begin{bmatrix} (a(v^2 + w^2) - u(bv + cw - ux - vy - wz))(1 - \cos\theta) + x\cos\theta + (-cv + bw - wy + vz)\sin\theta \\ (b(u^2 + w^2) - v(au + cw - ux - vy - wz))(1 - \cos\theta) + y\cos\theta + (cu - aw + wx - uz)\sin\theta \\ (c(u^2 + v^2) - w(au + bv - ux - vy - wz))(1 - \cos\theta) + z\cos\theta + (-bu + av - vx + uy)\sin\theta \end{bmatrix}$$

Applying the function to the current rotation axis and rotation angle,

(*x*, *y*, *z*) = points in the structure,

(*a*, *b*, *c*) would be the *origin*, so that the point pass through origin,

$\langle u, v, w \rangle$ would be the **vector3**, which is the rotation axis, and θ would be the rotation angle.

```

/*
    Rotates structure about an axis
    point: a point that the vector(axis) pass through
    axis : vector(axis of rotation)
    theta: rotation angle
*/
void Structure::rotateStruct(double* point, double* axis, double theta)
{
    for(int i = 0; i < mLength; i+=3)
    {
        rotatesAboutVector(mCoord+i, point, axis, theta, mCoord+i);
    }
}

```

Figure 4.2.6.5.2: Code of function rotateStruct

```

/*
    //rotatesAboutVector and rotatesAboutArbLine are the same
    //rotatesAboutVector is to convert double* into 3 double
    xyz : point to be rotate
    abc : a point that the rotation axis passes through
    uvw : direction vector (unit vector)
    theta : angle of rotation
    newCoord : the rotated point
*/
void rotatesAboutVector(double* xyz, double* abc, double* uvw, double theta,
double* newCoord)
{
    rotatesAboutArbLine(xyz[0], xyz[1], xyz[2], abc[0], abc[1], abc[2],
uvw[0], uvw[1], uvw[2], theta, newCoord);
}

```

Figure 4.2.6.5.3: Code of function rotatesAboutVector

```

void rotatesAboutArbLine(double x, double y, double z, double a, double b,
double c, double u, double v, double w, double theta, double* newCoord)
{
    double costheta = cos(theta);
    double sintheta = sin(theta);
    double oneMinusCosTheta = 1 - costheta;
    double v2 = v*v;
    double u2 = u*u;
    double w2 = w*w;
    newCoord[0] = (a*(v2 + w2) - u*(b*v + c*w - u*x - v*y - w*z)) *
oneMinusCosTheta + x*costheta + (-c*v + b*w - w*y + v*z)*sintheta;
    newCoord[1] = (b*(u2 + w2) - v*(a*u + c*w - u*x - v*y - w*z)) *
oneMinusCosTheta + y*costheta + (c*u - a*w + w*x - u*z)*sintheta;
    newCoord[2] = (c*(u2 + v2) - w*(a*u + b*v - u*x - v*y - w*z)) *
oneMinusCosTheta + z*costheta + (-b*u + a*v - v*x + u*y)*sintheta;
}

```

Figure 4.2.6.5.4: Code of function rotatesAboutArbLine

4.2.7 Exhaustive search of positions for p_1 in discretized D_c sphere around q_1

Suppose structure Q has been transformed so that q_1 and q_2 are along the y -axis. I want to discretize q_1 and q_2 with grids of size $\frac{\epsilon D_c}{3}$ and exhaustively examine p_1 and p_2 at each grid point. Now p_1 can be at most $(1 + \epsilon)D_c$ away from q_1 . To find the position for p_1 , I discretize a sphere of radius $(1 + \epsilon)D_c$ centered at q_1 . However, it is difficult to construct such a discretization with a sphere; to simplify this discretization a cube is used instead. The cube has the same width, length and depth, $2(1 + \epsilon)D_c$, as shown in Figure 4.2.7.1.

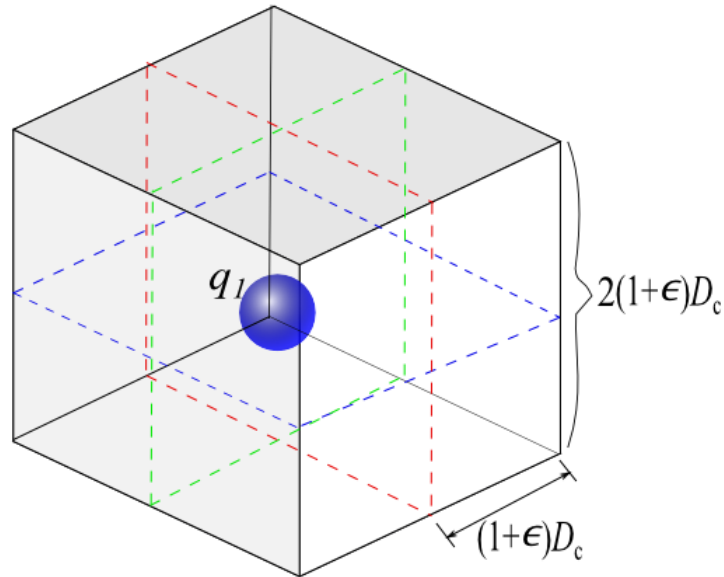
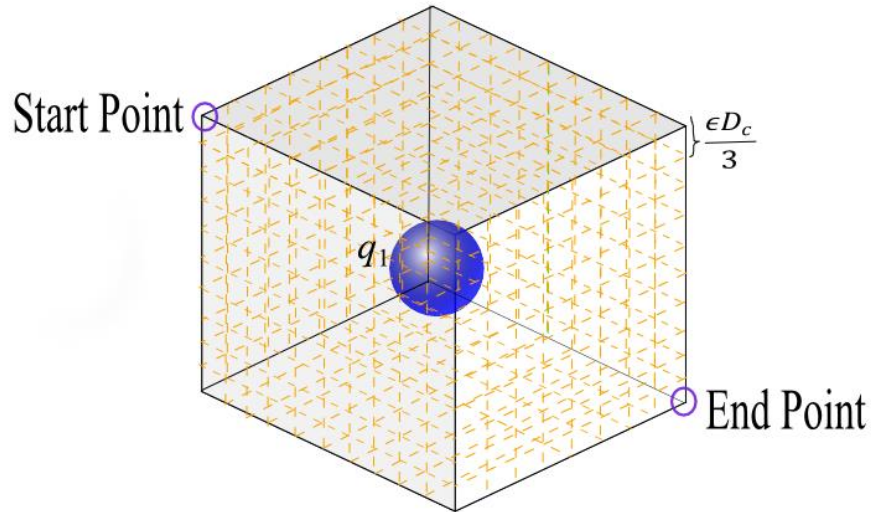


Figure 4.2.7.1: Cube for candidate positions of p_1

Then, the cube is discretized with grids of size $\frac{\epsilon D_c}{3}$. Positions for p_1 will be examined on each grid point on the cube and started with the start point and end point as shown in Figure 4.2.7.2. Start point is calculated by using x , y and z value of q_1 subtract $(1 + \epsilon)D_c$ while end point is calculated by using x , y and z value of q_1 add $(1 + \epsilon)D_c$, which is as follows:

$\text{Start} = (x - (1 + \epsilon)D_c, y - (1 + \epsilon)D_c, z - (1 + \epsilon)D_c)$ $\text{End} = (x + (1 + \epsilon)D_c, y + (1 + \epsilon)D_c, z + (1 + \epsilon)D_c)$

Figure 4.2.7.2: Discretization of cube of q_1

Beginning from the start point, I exhaustively examine p_1 on the grid. However, since this is a cube, the distance of each grid point to the center (q_1) may be greater than the radius of a sphere, which is $(1 + \epsilon)D_c$. Hence, I examine the distance from each grid point to the center of the sphere; when the distance is more than $(1 + \epsilon)D_c$, the grid point will not be considered further.

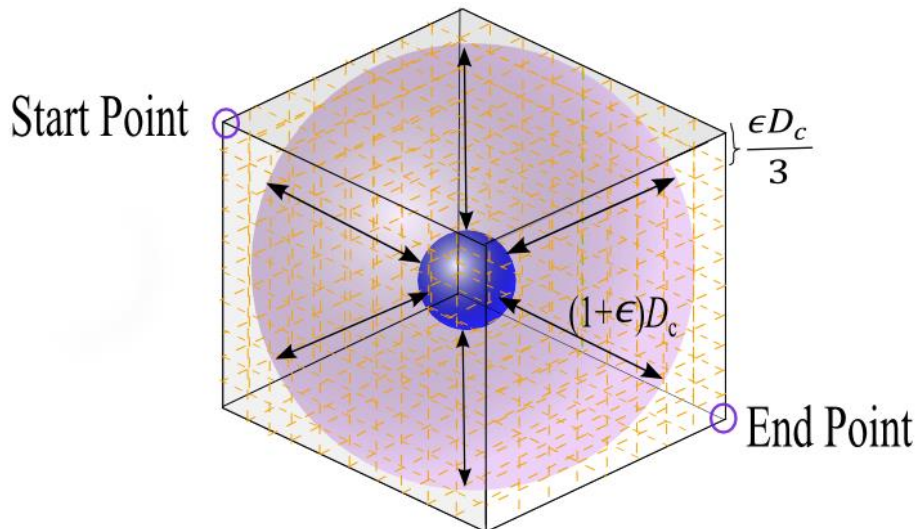


Figure 4.2.7.3: Sphere encapsulated in the cube

```

/*
    Examine p1 in the grid of q1 (all possible positions)
    p1 : coord of p1
    q1 : sphere center
    p2 : another point of structure P.
    q2 : point of structure Q.
*/
void Transformation::tryP1InGrid(double* p1, double* q1, double* p2, double* q2)
{
    double start[3] = {q1[0] - mThres, q1[1] - mThres, q1[2] - mThres}; //Store each
start point of each exist
    double end[3] = {q1[0] + mThres, q1[1] + mThres, q1[2] + mThres}; //Store each
end point of each exist
    double coord[3]; // coord = p1 on the grid
    double distQ = calDist(q1,q2); //distance between q1 and q2

    double radius = calDist(p1, p2);
    int count=1;
    for(double x = start[0]; x <= end[0]; x += mStepSize)
    {
        coord[0] = x;
        for(double y = start[1]; y <= end[1]; y += mStepSize)
        {
            coord[1] = y;
            for(double z = start[2]; z <= end[2]; z += mStepSize)
            {
                coord[2] = z;
                if(isInThres(q1, coord)) //is in sphere of q1
                {
                    //check whether distance(p1,q2) > mThres
                    if(calDist(coord,q2) > mThres)
                    {
                        //check whether p2 is in sphere of q2
                        if(radius >= (distQ - mThres) && radius <=
(distQ + mThres))
                        {
                            formSphereCap(p1, coord, p2, q2,
radius);
                            cout<<"next"<<endl;
                        }
                    }
                }
            }
        }
    }
}

```

Figure 4.2.7.4: Code of function tryP1InGrid

4.2.8 Forming a sphere cap for p_2

After fixing p_1 at a grid point, a sphere cap is to be formed for all possible positions of p_2 , centered at p_1 with radius $\|p_1 - p_2\|$, and the sphere cap must be inside the sphere of q_2 . The sphere cap is also discretized with grids of size $\frac{\epsilon D_c}{3}$. p_2 is then examined on each grid point of the sphere cap.

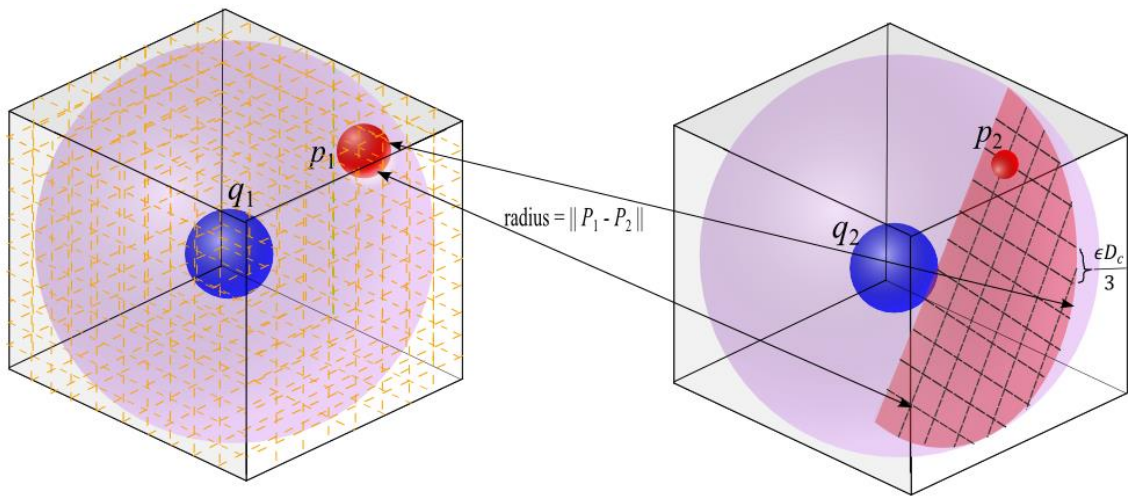


Figure 4.2.8.1: Forming sphere cap with grid

I check for the condition of whether the distance between the grid point and q_2 is greater than $(1 + \epsilon)D_c$. If the grid point is within $(1 + \epsilon)D_c$ from q_2 , they are approximately matched, and the formation of a sphere cap is not required. In order to form the sphere cap, another condition must be fulfilled. Namely, that the distance d between p_1 and p_2 must have

- $d \geq [\text{distance between } q_1 \text{ and } q_2] - (1 + \epsilon)D_c$, and
- $d \leq [\text{distance between } q_1 \text{ and } q_2] + (1 + \epsilon)D_c$, as shown in Figure 4.2.8.2.

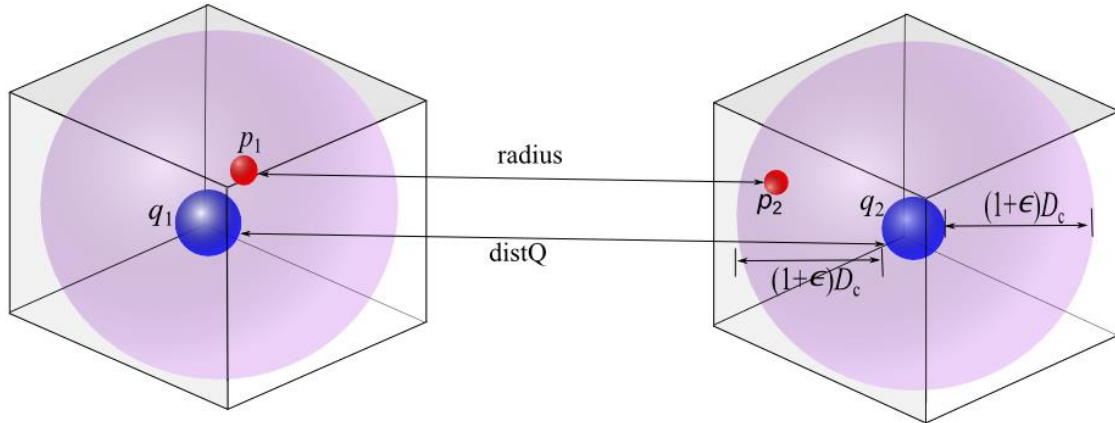


Figure 4.2.8.2: Conditions leading to the formation of a sphere cap

To fulfill the conditions, all the possible positions of p_2 form a sphere cap inside the sphere of q_2 . The sphere cap is generated in the program as follows: starting with a point that forms a straight line with p_1 and is parallel to the y -axis (Figure 4.2.8.3). I rotate the point upward, downward, leftward or rightward, until forming a circle, 2π . The step size of the movement is set according to the resolution of the discretization.

Now I want to find the rotation axis and rotation angle to perform the rotations in each of the four directions mentioned. The rotation axis is initialized with a vector that passes through p_1 and is parallel to the z -axis as shown in Figure 4.2.8.3.

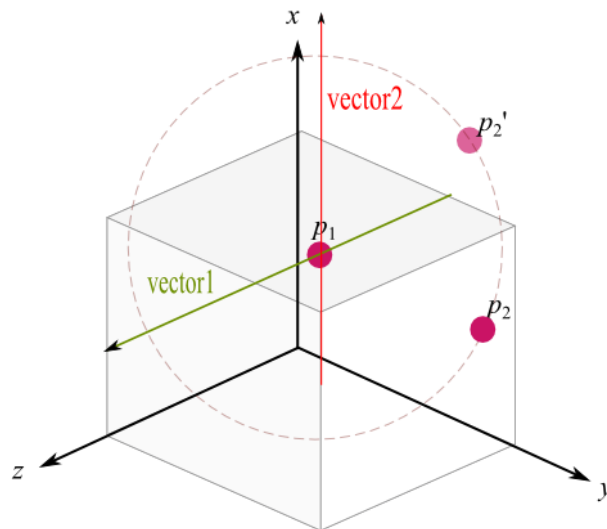


Figure 4.2.8.3: Vector1 to rotate p_2

The vector (*vector1*) then will be rotated around another vector (*vector2*) that passes through p_1 and is parallel to the x-axis, to be rotated to the left or right, corresponding to the leftward and rightward movement of the point. Each rotated point will be checked to see if it is still within the sphere of q_2 by checking whether the distance between the point and q_2 is within $(1 + \epsilon)D_c$. Otherwise, the rotation continues until 2π .

To rotate a vector about another vector, the following formula is used:

$$V = \{A - [(axis \cdot A)axis]cos\theta\} + [(A \times axis)sin\theta],$$

where A is the vector to be rotated, and $axis$ is the rotation axis (Rotating a vector around an arbitrary axis 2003).

```

/*
   A:Vector to be rotate
   axis:axis
   V = (A - ((axis.A)*axis)*costheta) + ((A X axis)*sintheta) +
   (axis.A)*axis
*/
//must use normalized vector
void rotateVectorAbtVector(double* A, double* axis, double theta,
double* ans)
{
   double cross[3], dotAxis[3], dotAxisCosTheta[3], ans1[3], ans2[3],
ans3[3];
   double dot;

   dot = dotProduct(A,axis);           //(axis.A)
   multiply(dot, axis, dotAxis);      //(axis.A)*axis
   multiply(cos(theta), dotAxis, dotAxisCosTheta);
   //((axis.A)*axis)*costheta

   crossProduct(A,axis,cross);        //(A X axis)

   diff(A, dotAxisCosTheta, ans1);    //(A -
   ((axis.A)*axis)*costheta)
   multiply(sin(theta), cross, ans2); //((A X axis)*sintheta)
   ans3[0] = dotAxis[0];              //(axis.A)*axis
   ans3[1] = dotAxis[1];
   ans3[2] = dotAxis[2];

   add(ans1, ans2, ans);
   add(ans, ans3, ans);
}

```

Figure 4.2.8.4: Code of function rotateVectorAbtVector

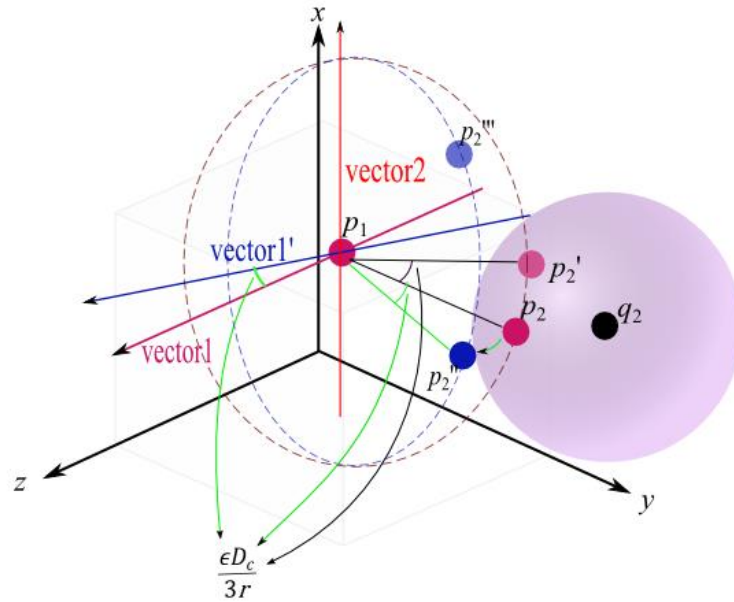


Figure 4.2.8.5: Method to form the sphere cap. Vector1 is the rotation axis for p_2 to rotate in the up or down direction. Vector2 is the vector for vector1 and point p_2 to rotate to the left or right. Each rotated point of p_2 will be checked to see if it is within the sphere of q_2 .

The rotation angle is calculated using the formula of arc length, $s = r\theta$. Hence, $\theta = \frac{s}{r}$. For the present purpose, $s = \frac{\epsilon D_c}{3}$, $r =$ distance between p_1 and p_2 . Thus θ is computed as

$$\frac{\epsilon D_c}{3\|p_1 - p_2\|}$$

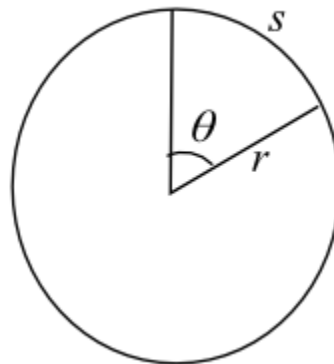


Figure 4.2.8.6: Diagram for formula of arc length

```

/*
    Form a sphere cap for p2 on q2 grid, try all coordinates on sphere cap
    oriP1: the original p1 before putting on grid
    p1 : center of the sphere cap
    p2 : point to form sphere cap
    q2 : center of sphere where the sphere cap is inside
    radius: distance between p1 and p2, radius of forming sphere cap
*/
void Transformation::formSphereCap(double* oriP1, double* p1, double* p2, double* q2,
double radius)
{
    double totalangle,totalAOut,PI2;           //totaltheta must be at most 360 to
stop the rotation.
    double coord[3];                          //point on sphere cap
(new p2)
    double middleCoord[3];                    //center point of each moving
up and down/left and right
    double oriCoord[3];                      //the original point before
any rotation
    double vector1[3], vector2[3];           //axis of rotation
    double angle = mStepSize/radius;        //rotation angle
    PI2 = 2*PI;
    bool isWithinQ2 = false;                //is the coordinate within sphere of
q2
    double rotAng;

    //start to form sphere cap with p1 point that form straight line parallel to y-
axis
    middleCoord[0] = p1[0];
    middleCoord[1] = p1[1] + radius;
    middleCoord[2] = p1[2];

    //copy the coordinates to prepare for rotation
    coord[0] = middleCoord[0];
    coord[1] = middleCoord[1];
    coord[2] = middleCoord[2];
    oriCoord[0] = middleCoord[0];
    oriCoord[1] = middleCoord[1];
    oriCoord[2] = middleCoord[2];

    //vector1 to rotate p2 on sphere cap (up and down): firstly parallel to z, will be
changed later
    vector1[0] = 0;           //parallel to z
    vector1[1] = 0;
    vector1[2] = 1;

    //point that passes through the vector1 and vector2(axis of rotation) = p1

    //vector2 to rotate p2 on sphere cap (left and right) : parallel to x-axis
    vector2[0] = 1;
    vector2[1] = 0;
    vector2[2] = 0;

    totalAOut = 0;
}

```

Figure 4.2.8.7: Code of function formSphereCap


```

do
{
    /*prepare to rotates up and down*/
    //going up and down

    totalangle = 0;

    //rotates p2 up and down around vector1
    do
    {
        rotatesAboutVector(coord, p1, vector1, angle, coord);    //going
up or down

        totalangle += angle;
        //check whether is in sphere of q2
        isWithinQ2 = isInThres(q2, coord);
        if(isWithinQ2)
        {
            //Move other P respectively
            movePtoPlace(p1, oriP1, coord, p2); //p1: new p1 on grid;
oriP1: p1 before putting on grid;

            //coord: p2 on sphere cap;    p2: p2 before putting on sphere cap
            //form rotation axis and rotates
            findAngleInOut();
            getMaxMatch(); //get most number of residue matched from
the maximum bipartite matching for each rotation angle.
        }

    } while(totalangle < PI2);

    //reset coord back to original coord before rotates up&down, to prepare to
move left/right

    coord[0] = middleCoord[0];
    coord[1] = middleCoord[1];
    coord[2] = middleCoord[2];
    //rotates p2 left and right and rotates vector1 together with same theta

    rotatesAboutVector(coord, p1, vector2, angle, coord);    //going
left&right

    //rotates also vector1
    rotateVectorAbtVector(vector1, vector2, angle, vector1);
    //normalize vector1
    normalize(vector1);
    totalAOut += angle;

    middleCoord[0] = coord[0];
    middleCoord[1] = coord[1];
    middleCoord[2] = coord[2];

    } while(totalAOut < PI2);
}

```

Figure 4.2.8.8: Code of function formSphereCap

Having moved p_1 and p_2 to a grid point around q_1 and q_2 respectively, all the points in P are to be transformed accordingly. To do so, they are first applied the translation step which moved p_1 from its original to its position on the grid point around

q_1 . Then, they are applied the rotation which moved p_2 to its position on the sphere cap around q_2 . The exact geometrical manipulation follows the same method used earlier to transform Q such that q_1 and q_2 lie along the y-axis. The rotation axis is determined by using cross product and rotation angle is calculated by using the Law of Cosine.

```

/*
    Move all points in structure P respectively after fixing p1 and p2
    newP1 : the point on grid(fixed)
    p1 : the old point before putting on grid
    newP2 : the point on sphere cap(fixed)
    p2 : the old point before putting on sphere cap
*/
void Transformation::movePtoPlace(double* newP1, double* p1, double* newP2,
double* p2)
{
    //get translation step to translate from old P1 to new P1.
    double transStep[3];
    double vector1[3];
    double vector2[3];
    double vector3[3];
    double theta, a, b, c, cosC;

    diff(p1, newP1, transStep);
    mModel->translateStruct(transStep);

    //get vector1 (a), vector2 (b), vector3 (c), where c = a x b cross
product, c = axis of rotation
    diff(newP1, p2, vector1); //a
    diff(newP1, newP2, vector2); //b
    crossProduct(vector1, vector2, vector3); //c:axis of rotation

    normalize(vector3);

    //get angle of rotation using law of cosines
    a = calDist(newP1, p2);
    b = calDist(newP1, newP2);
    c = calDist(p2, newP2);

    cosC = (a*a + b*b - c*c) / (2*a*b);

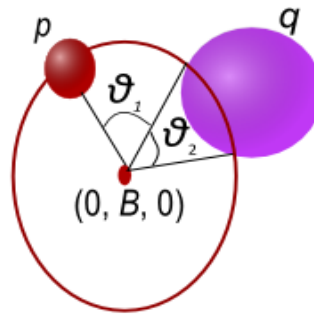
    if(cosC > 1)
        cosC = 1;
    if(cosC < -1)
        cosC = -1;

    theta = acos(cosC);

    mModel->rotateStruct(newP1, vector3, theta);
}

```

Figure 4.2.8.9: Code of function movePtoPlace

4.2.9 Find angle that moves P in and out of contact of Q Figure 4.2.9.1: Angle that moves p in and out of contact with q

The program proceeds with p_1 , p_2 and Q fixed. The remaining points in P are to be rotated about the axis formed by p_1 and p_2 . In the course of the rotation, points in P might come into contact with some points in Q (that is, within $(1 + \epsilon)D_c$ distance from these points), as well as going out of contact from the points in Q (that is, further than $(1 + \epsilon)D_c$ from these points) which they are originally in contact with.

The program now computes, for each p in P and each q in Q , the angle that moves p in and out of contact with q (as shown in Figure 4.2.9.1). Noting that the rotational path of p forms a circle, this can be computed by finding the intersection of a circle and a sphere.

```

/*
   Get the angle of rotation that moves P into and Out of Contact with Q,
   by using intersection between the sphere of Q and unit circle(path of p)
*/
void Transformation::findAngleInOut()
{
    int indexp, indexq;
    initializeMatch();

    for(int i = 0; i < mNative->mLength; i+=3)
    {
        indexq = i/3;
        for(int j = 0; j < mModel->mLength; j+=3)
        {
            indexp = j/3;

            getIntersectPoint(mModel->mCoord + j, mNative->mCoord +
i, indexp, indexq);
        }
    }
}

```

Figure 4.2.9.2: Code of function findAngleInOut

Initially, the distance between p and q is checked. p is approximately matched to q when the distance between p and q is within $(1 + \epsilon)D_c$. Since the p has been matched to q , no rotation is required for the specific p and thus no calculation on intersection of circle and sphere is required. θ_1 , the angle that moves p into q , is set to 0 since p is already matched to q while θ_2 , the angle that moves p out of q , is set to 2π .

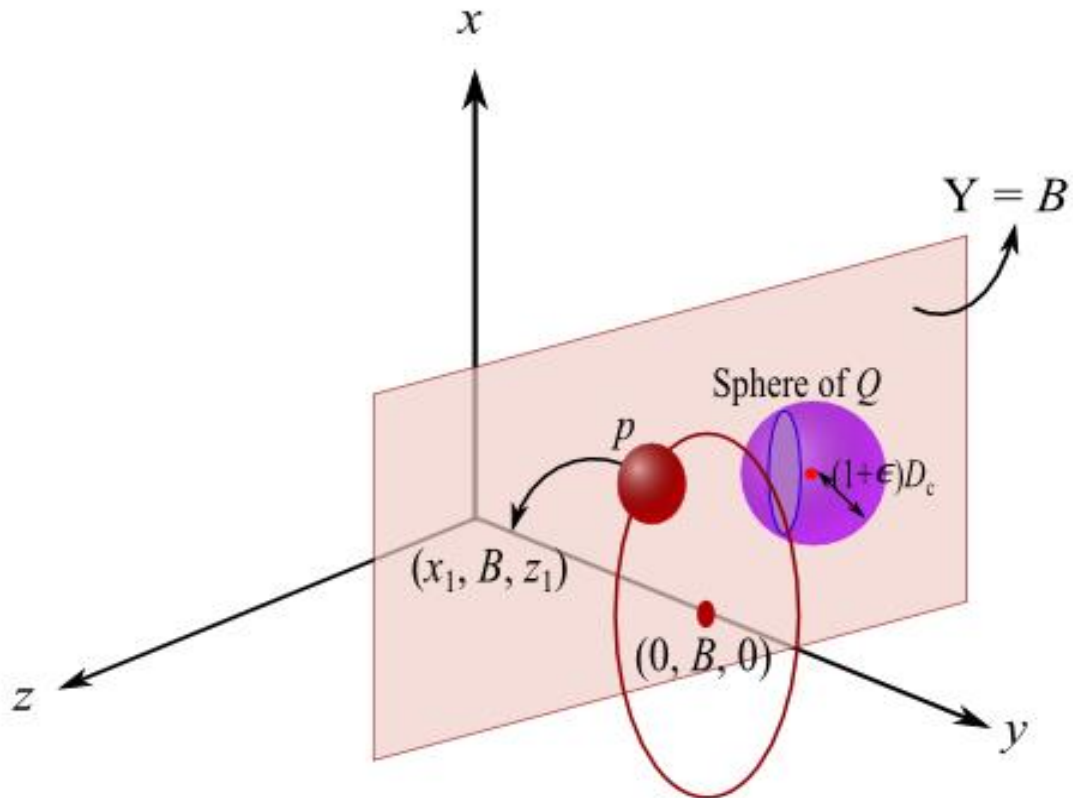


Figure 4.2.9.3: Intersection of circle and sphere

To find the intersection of a circle and a sphere, I transform the circle into a plane, then intersect the sphere with the plane. Each circle centered at $(0, B, 0)$ and point $P = (x_1, B, z_1)$. Hence, the plane equation is $Y = B$. The intersection of the plane $Y = B$ and the sphere of Q will be a circle as in Figure 4.2.9.4.

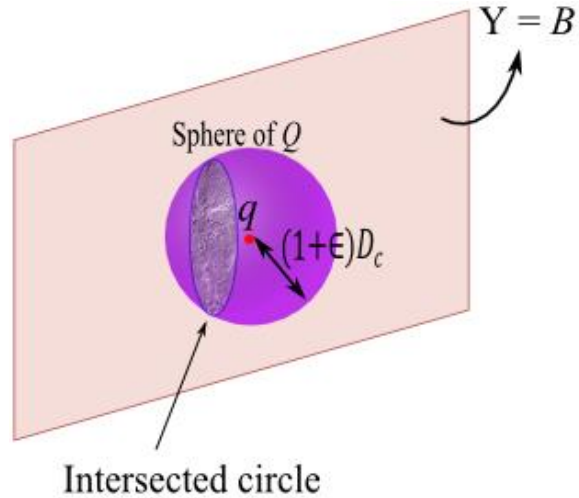


Figure 4.2.9.4: Intersection of a plane and circle

Equation of plane: $Y = B$

Equation of sphere: $(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = ((1 + \epsilon)D_c)^2$

The formula of equation of the intersected circle can be found in (AmBrSoft Quality Softwares, n.d.):

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2,$$

$$\text{where } r = \sqrt{R^2 - d^2},$$

$R = \text{radius of sphere},$

$$d = \frac{|Ax_0 + By_0 + Cz_0 + D|}{\sqrt{A^2 + B^2 + C^2}}$$

By applying the formula above,

$$d = y_0 - B,$$

$$r = \sqrt{((1 + \epsilon)D_c)^2 - (y_0 - B)^2},$$

which gives the equation:

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = ((1 + \epsilon)D_c)^2 - (y_0 - B)^2,$$

In order to find the intersection of the circle and the sphere, the intersection of the circle (formed by the rotational path of p) and the intersected circle (intersection of plane and sphere Q) is calculated as shown in Figure 4.2.9.5.

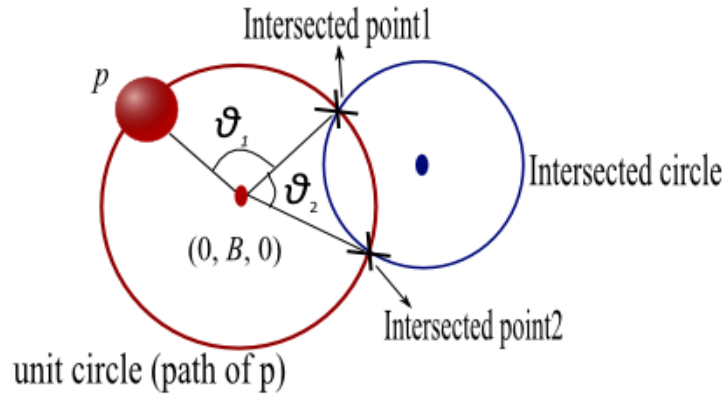


Figure 4.2.9.5: Intersection of Two Circles

θ_1 = Angle that moves p into contact with q

θ_2 = Angle that moves p out of contact with q

θ_1 and θ_2 can be calculated by using the Law of Cosine.

The intersection of two circles can be found analytically. Firstly, since the circles lies on the x - z plane where $y = B$, there are only two unknowns to solve.

Suppose the radius of the intersected circle is $\sqrt{((1 + \epsilon)D_c)^2 - (y_0 - B)^2}$ and the radius of the unit circle is $\sqrt{x_1^2 + z_1^2}$. To simplify the equation, let the radius of intersected circle be R and the radius of unit circle be r .

The equation of unit circle : $x^2 + z^2 = r^2$ ----- ①

The equation of the intersected circle : $(x - x_0)^2 + (z - z_0)^2 = R^2$ ----- ②

Expanding ②,

$x^2 - 2x_0x + x_0^2 + z^2 - 2z_0z + z_0^2 = R^2$ ----- ③

③—①,

$$-2x_0x + x_0^2 - 2z_0z + z_0^2 = R^2 - r^2$$

$$x = \frac{r^2 - R^2 + x_0^2 - 2z_0z + z_0^2}{2x_0} \quad \text{-----} \textcircled{4}$$

Substitute ④ into ① and $\times 4x_0^2$,

$$(4x_0^2 + 4z_0^2)z^2 + (-4r^2z_0 + 4R^2z_0 - 4x_0^2z_0 - 4z_0^3)z + (r^4 - 2x_0^2r^2 - 2R^2r^2 + 2z_0^2r^2 + R^4 - 2x_0^2R^2 - 2z_0^2R^2 + x_0^4 + 2x_0^2z_0^2 + z_0^4) = 0$$

Using quadratic formula,

$$z = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a},$$

where $a = (4x_0^2 + 4z_0^2)$,

$b = (-4r^2z_0 - 4R^2z_0 - 4x_0^2z_0 - 4z_0^3)$,

$c = (r^4 - 2x_0^2r^2 - 2R^2r^2 + 2z_0^2r^2 + R^4 - 2x_0^2R^2 - 2z_0^2R^2 + x_0^4 + 2x_0^2z_0^2 + z_0^4)$

Substitute Z values into ④,

$$x = \frac{r^2 - R^2 + x_0^2 - 2z_0z + z_0^2}{2x_0}$$

After getting the intersection points, the angles are calculated using Law of Cosine as shown in Figure 4.2.9.7 and Figure 4.2.9.8.

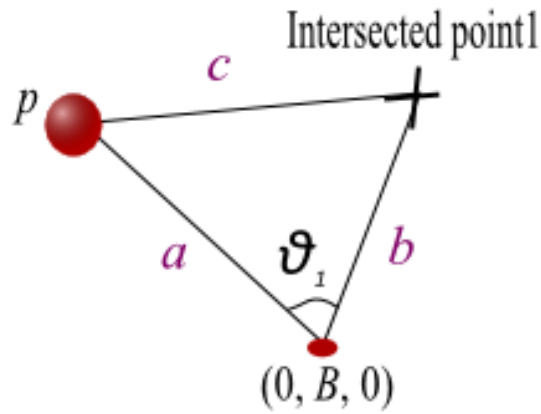


Figure 4.2.9.7: Angle that moves p into contact with q

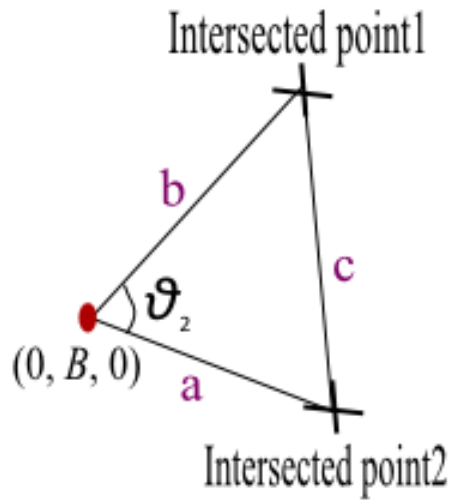


Figure 4.2.9.8: Angle that moves p out of contact of q


```

/*
   Get intersection point of circle centered at (0,B,0) and circle centered
   at (X0, B, Z0) - intersection circle of plane Y=B and sphere of q
   Circle 1: (intersection of plane and sphere) equation: (X-x0)2 + (Y-
   B)2 + (Z-z0)2 = ((1+eps)Dc)2 - (y0-B)2
   Circle 2: (path of p) equation: X2 + (Y-B)2 + Z2 = (x1)2 + (z1)2
   p : any point of p (x1,B,z1)
   q : any point of q (x0,y0,z0)
   B : plane to intersect with sphere.
*/
void Transformation::getIntersectPoint(double* p, double* q, int indexp,
int indexq)
{
    double B = p[1]; //y-coordinate
    double R = sqrt(mThres*mThres - (q[0]-B)*(q[0]-B)); //Radius of Circle1
    double r = sqrt(p[0]*p[0] + p[2]*p[2]); //radius of Circle2
    double R2, r2; //squared of R and squared of r.
    double c2center[3] = {0,B,0}; //center of circle 2

    R2 = R*R;
    r2 = r*r;

    double point1[3],point2[3]; //intersection points between two circles
    double a,b,c,cosC, angle1, angle2;

    double q02, q22; //squared of q[0] and squared of q[2].
    q02 = q[0]*q[0];
    q22 = q[2]*q[2];

    double dist = calDist(p, q); //distance between two centers of circle
    int yes;

    if(dist <= mThres) //p is approximately matched to q
    {
        angle1 = 0;
        angle2 = 2*PI;
    }
    else
    {
        //(-b +/- sqrt(b2-4ac)) / 2a
        a = 4*q02 + 4*q22;
        b = -4*r2*q[2] + 4*R2*q[2] - 4*q02*q[2] - 4*q22*q[2];
        c = r2*r2 - 2*q02*r2 - 2*R2*r2 + 2*r2*q22 + R2*R2 - 2*R2*q02 -
2*R2*q22 + q02*q02 + 2*q02*q22 + q22*q22;

        point1[2] = (-b + sqrt(b*b-4*a*c)) / 2*a; //z1
        point2[2] = (-b - sqrt(b*b-4*a*c)) / 2*a; //z2

        point1[1] = B; //y1
        point2[1] = B; //y2

        point1[0] = (q22 + q02 - 2*q[2]*point1[2] - R2 + r2) / 2*q[0];
        //x1:sub z1
        point2[0] = (q22 + q02 - 2*q[2]*point2[2] - R2 + r2) / 2*q[0];
        //x2:sub z2
    }
}

```

Figure 4.2.9.9: Code of function getIntersectPoint

```

//get angle using law of cosines
//angle that moves P into Q
a = calDist(c2center, p);
b = calDist(c2center, point1);
c = calDist(p, point1);

cosC = (a*a + b*b - c*c) / (2*a*b);

angle1 = acos(cosC);

//angle that moves P out of Q using law of cosines
a = calDist(c2center, point2);

c = calDist(point1, point2);

cosC = (a*a + b*b - c*c) / (2*a*b);

angle2 = acos(cosC);
}

setRotInterval(angle1, angle2, indexp, indexq);
}

```

Figure 4.2.9.10: Code of function getIntersectPoint

All the angles (θ_1 and θ_2) computed thus far are then to be sorted. Suppose this produces the sequence of increasing angles $\phi_1, \phi_2, \phi_3, \dots$. Then, each interval (ϕ_i, ϕ_{i+1}) represents a region where no two points from P and Q respectively come into or moves out of contact. That is, the ways to match points in P and Q are the same for all the angles in the interval; hence in finding the optimal way to match the points between P and Q , only one angle needs to be considered.

To sort the angles, I start with first angle of θ_1 . Note that θ_1 and θ_2 come in pair and θ_2 must be next to the θ_1 . For each angle, the matching of the respective p and q will be stored. For the first angle, θ_1 , there is no respective p and q to be matched, hence -1 is set as its respective index. In θ_2 , the respective p and q will be matched; the index combination will be set to 1.

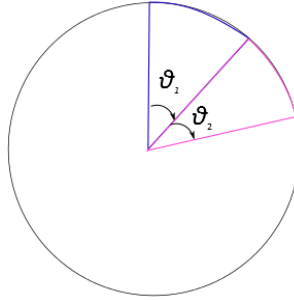


Figure 4.2.9.11: First θ_1 and θ_2 combination

When the next θ_1 is smaller than the first angle in the storage, there will be another angle created after the first angle in the storage as shown in left Figure 4.2.9.12. Then, its pair of θ_2 must be started right after θ_1 , as shown in right Figure 4.2.9.12. When the angle is greater than the next cumulative angles, there also will be a new angle created (as in Figure 4.2.9.12).

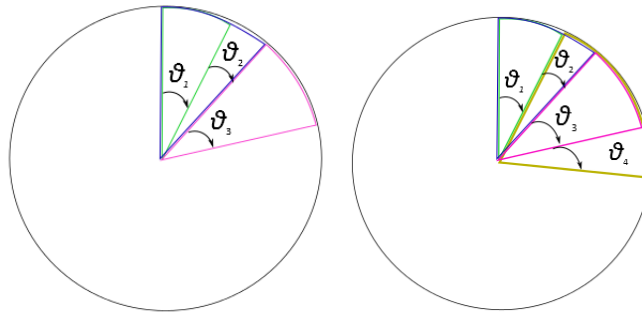


Figure 4.2.9.123: Angle rearrangement

4.2.10 Find the maximum bipartite matching

After arranging the angles in order and setting the matching for each angle, the maximum bipartite matching for the matching in each angle is to be determined. The Hopcroft-Karp algorithm (Hopcroft & Karp 1973) is used to obtain this matching.

The algorithm takes as input a bipartite graph, and produces as output a maximum cardinality matching, i.e. a set of maximum edges with the property that no two edges share an endpoint. This algorithm runs in $O(|E|\sqrt{V})$ time in the worst case, where E is

the set of edges in the graph and V is the set of vertices of the graph. The algorithm is able to work with a partial solution, improving on it to produce a maximal set of shortest augmenting paths in order to increase the size of the partial matching. This allows us to use the partial matching for the bipartite graph from an interval (ϕ_{i-1}, ϕ_i) as initial input to the problem instance for the interval (ϕ_i, ϕ_{i+1}) .

The pseudocode of Hopcroft-Karp algorithm is as below:

```

/*
G = G1  $\cup$  G2  $\cup$  {NIL}
where G1 and G2 are partition of graph and NIL is a special null vertex
*/

function BFS ()
  for v in G1
    if Pair_G1[v] == NIL
      Dist[v] = 0
      Enqueue(Q,v)
    else
      Dist[v] =  $\infty$ 
  Dist[NIL] =  $\infty$ 
  while Empty(Q) == false
    v = Dequeue(Q)
    if Dist[v] < Dist[NIL]
      for each u in Adj[v]
        if Dist[ Pair_G2[u] ] ==  $\infty$ 
          Dist[ Pair_G2[u] ] = Dist[v] + 1
          Enqueue(Q,Pair_G2[u])
  return Dist[NIL] !=  $\infty$ 

```

```

function DFS (v)
  if v != NIL
    for each u in Adj[v]
      if Dist[ Pair_G2[u] ] == Dist[v] + 1
        if DFS(Pair_G2[u]) == true
          Pair_G2[u] = v
          Pair_G1[v] = u
          return true
    Dist[v] = ∞
    return false
  return true

```

```

function Hopcroft-Karp
  for each v in G
    Pair_G1[v] = NIL
    Pair_G2[v] = NIL
  matching = 0
  while BFS() == true
    for each v in G1
      if Pair_G1[v] == NIL
        if DFS(v) == true
          matching = matching + 1
  return matching

```

4.2.11 Output the maximum number of matching of both structures.

```

    cout << "Maximum number of residue matched: " << maxMatch << endl;
    return 0;
}

```

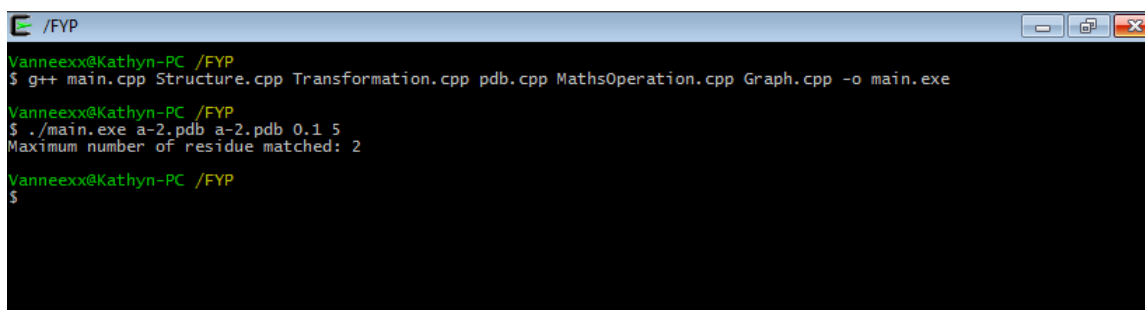
4.3 Program Result

Since the program runs for a long time, the program is tested with several .pdb file that has very less number of CA atoms to shorten the time of running. The results have shown the accuracy of the algorithm.

In order to test the accuracy of the program, the same .pdb file has been used for the input of the program. For instance, if protein structure file has two residues of CA atoms, the output of the program should also be two since the two files are the same.

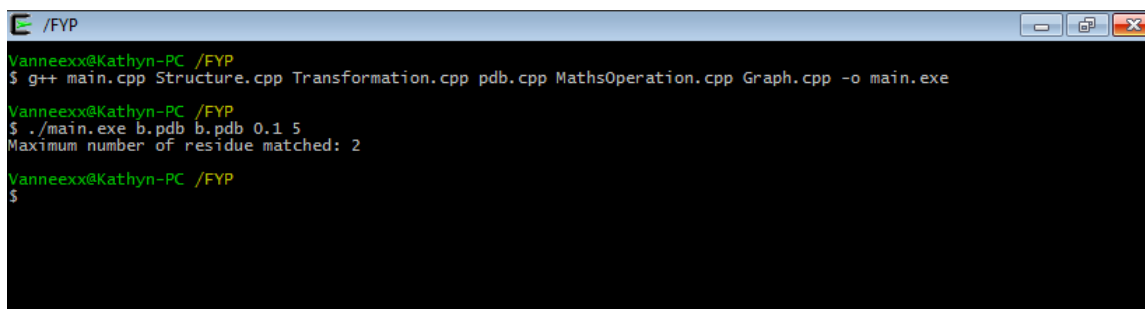
Some test cases of my program testing are as follows:

Test case 1: The program was tested with two same .pdb file, namely a.pdb that has two residues of CA atoms. The result showed that there are two residues matched.



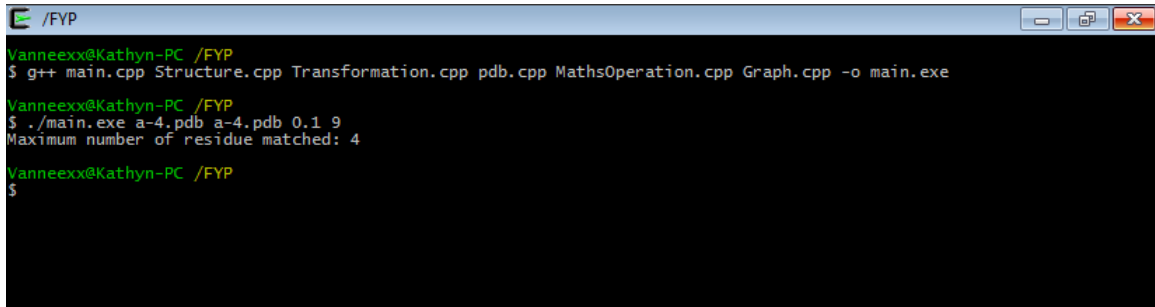
```
/FYP
Vanneexx@Kathyn-PC /FYP
$ g++ main.cpp Structure.cpp Transformation.cpp pdb.cpp MathsOperation.cpp Graph.cpp -o main.exe
Vanneexx@Kathyn-PC /FYP
$ ./main.exe a-2.pdb a-2.pdb 0.1 5
Maximum number of residue matched: 2
Vanneexx@Kathyn-PC /FYP
$
```

Test case 2: The program was then tested with another .pdb file, that is b.pdb that is also has two residues of CA atoms. The output of the program is two, which is the expected output.



```
/FYP
Vanneexx@Kathyn-PC /FYP
$ g++ main.cpp Structure.cpp Transformation.cpp pdb.cpp MathsOperation.cpp Graph.cpp -o main.exe
Vanneexx@Kathyn-PC /FYP
$ ./main.exe b.pdb b.pdb 0.1 5
Maximum number of residue matched: 2
Vanneexx@Kathyn-PC /FYP
$
```

Test case 3: The input protein structure files have four residues of CA atoms. The result of the program is 4.



```
/FYP
Vanneexx@Kathyn-PC ~/FYP
$ g++ main.cpp Structure.cpp Transformation.cpp pdb.cpp MathsOperation.cpp Graph.cpp -o main.exe
Vanneexx@Kathyn-PC ~/FYP
$ ./main.exe a-4.pdb a-4.pdb 0.1 9
Maximum number of residue matched: 4
Vanneexx@Kathyn-PC ~/FYP
$
```

CHAPTER 5 DISCUSSION AND CONCLUSION

5.1 Discussion

5.1.1 Achievement

I have successfully implemented the algorithm in 2.2 in C++. Due to the high runtime complexity of the algorithm, the program takes quite a long time to run. The discretization and the forming of sphere cap for pairs of point, which introduces the $O(\frac{1}{\epsilon^5})$ runtime, appears to be the bottleneck. Although this program takes a long time to run, it guarantees the accuracy of the matching of two structures and thus will help researchers to determine the alignments accurately.

The testing of this program has been carried out to ensure the correctness of the program.

However, the analysis of code in order to accelerate the program has not been carried out. The accuracy of its output compared to other methods has also not been examined because of the time taken to run the whole program. The packaging of the codes for handling radial pairs (into an easy to use library) has also not been carried out due to time constraint.

The unfinished work should be continued in the future in order to make the program more useful.

5.1.2 Implementation Issues and Challenges

5.1.2.1 Problem of Installing Cygwin

At the beginning of the project, Microsoft Visual Studio 2010 is used to run my code. However, in the Bioinformatics field, most of the people are using Unix operating system. Thus, I installed Cygwin to run the program. I found that it was difficult to use since it has no GUI. Installing Cygwin also requires some knowledge such as: in order to compile C++ codes, the GCC package has to be installed. The default installation of Cygwin did not include the package, and resulted in some time lost before the problem was identified.

After seeking advice from a friend that used Cygwin to run his code, I successfully run my code. He also showed me several commands to compile and run the codes, as well as obtained the packages for me from the internet.

5.1.2.2 Early conceptual mistakes in implementing the algorithm

- Find radial pair of structure P for matching structure Q .

In finding the radial pair of P , I made a few errors in coding the condition of “ $\|p_1 - q_1\| \leq D_c$ and $\|p_2 - q_2\| \leq D_c$ ”. I translated whole structure P so that p_1 match q_1 , and then checked the distance between p_2 and q_2 to see if it is within $2D_c$. This is an error for all the cases because the position of p_2 and q_2 might be within $2D_c$ after rotation. The checking of the distance between p_2 and q_2 after translation is not correct. After discussing with my supervisor, he showed me my mistake and I made the corrections.

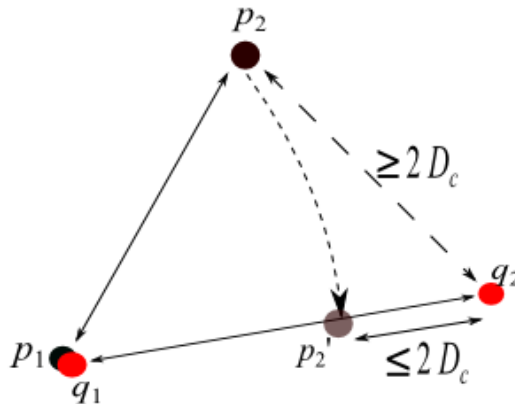


Figure 5.1.2.2.1: Wrong concept of finding radial pair of P to match Q

- Form sphere cap for p_2

I had a few conceptual misunderstanding regarding the forming of the sphere caps for p_2 . Initially, I first form grids on the cube of q_2 , and then only form the sphere cap by checking the distance between the grid and the cube center (q_2). However, this is wrong as when the sphere cap is not on the grid, the sphere cap could not be formed.

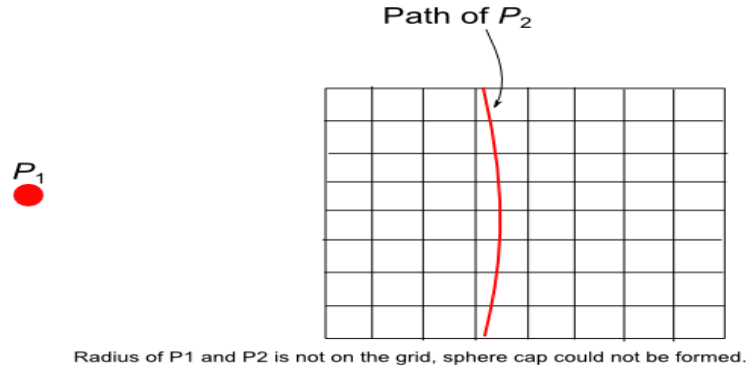


Figure 5.1.2.2.2: Wrong concept of forming sphere cap

- Rotation of P to match Q

To rotate structure P to match structure Q , I have to rotate structure P around an arbitrary axis of q_1 and q_2 . However, I did not know how to do this. I took advice from supervisor to translate structure Q so that q_1 and q_2 lies along the y -axis. I did not know how to use the rotation matrix to rotate a point about an arbitrary axis. I searched for the formula at Google for days. Finally, I found the formula and also the graphic visualization of the formula. I also did not know how to rotate a vector about an arbitrary axis, which is to change the direction of the rotation axis. I tried to come out with a formula for rotating a point about an arbitrary axis, but failed. I searched for the formula for days and finally found the formula on web. It took me more than a week to complete this step.

- Total Rotation Angle

In my initial tests of the program, the program did not complete even for very small problem instances. At first I suspected that this is a natural consequence of the high time complexity. However, after some debugging, I found that I had written an incorrect condition for enumerating the rotation angles. The enumeration should terminate when the angle is at most 2π , which I wrote 360° . This, however, caused the program to require ~100 times the actual time, since the rotation angle should be specified in radian.

5.1.2.4 Lack of knowledge in circle-sphere intersection

Another part of the program which I found difficult to code is the part where I am to find the intersection between a circle and a sphere. I searched for solutions to this problem on many forums on the internet, but was unable to find any information on the problem. There were only methods of finding plane-sphere intersection as well as circle-circle intersection. My supervisor told me to reduce the problem to one of finding circle-circle intersection, by changing the sphere into a circle via a plane-sphere intersection.

Thus, I extended one circle into a plane and cut the sphere with the plane to obtain a circle from the sphere. Then, the problem becomes one of circle-circle intersection.

I also did not know what the equation for a plane is. I discussed this with my supervisor again and finally came out with the complete solution.

5.1.2.4 Implementation of Hopcroft-Karp algorithm

I searched for graphic explanation of Hopcroft Karp algorithm on the internet. However, I could not get any explanation of the algorithm with graphics. I understood the step of finding augmenting path through several powerpoint slides that were available on web. The pseudocode of the Hopcroft-Karp algorithm is available on the internet and I found that someone has already implemented it in C++. The writer posted his code on Blogspot so that the code is able to help some people who need it (as shown below). I tried to run his code but failed to get the expected result since I did not know what the input of the code should be. The program also ran into an infinite loop at a specific line. After trying for days and asking the writer about the input, I managed to get the expected result.

Next, I began to change the input of the function into the form required by my program, namely, matching of the two structures P , Q at specific angle. The declaration method of the vector<int> type double array is inappropriate. I tried many declaration methods and resulted in error. An example of a wrong declaration is as follows:

```
vector<int> G[length*length+1];
```

I followed the method of declaration of original coding to declare a double array of size length^2+1 . But, this did not work as well as it requires a constant whereas length is the parameter that is passed into the function. I found a method to declare this `vector<int>` with the build-in function “resize”. I make the following declaration and resize in the same function and found that it is also wrong.

```
int hopcroft_karp(int length, int** matching)
{
    vector<vector<int> > G;
    G.resize(length+1);
}
```

After trying for few times, I found that the program will work when `vector<int>` is declared a global variable.

```
vector<vector<int> > G;
int hopcroft_karp(int length, int** matching)
{
    G.resize(length+1);
}
```

Nevertheless, I also faced the problem of two uninitialized variables: “dist” and “match”. The original code which I obtained declares them as global variable but I declared them locally. I did not notice this would make a difference. Only after debugging for some times did I realize that there are special meanings to these variable when they are set to zero; `dist[] = 0` means that all path is not defined yet and `match[] = 0` means that there is no initial matching for the specific combination of points. These details were not documented in the code which I obtained.

Original coding

```
int n, m, match[MAX], dist[MAX];
```

My coding

```
int hopcroft_karp(int length, int** matching)
{
    G.resize(length+1);
    int length2 = length*length;
    int *match = new int[length2 + 1];
    int *dist = new int[length2 + 1];
}
```

5.1.3 Runtime Error and Memory Error

There are many memory errors that occurred while testing my program. I have to print out some words for each section to find out which lines caused the memory problem. Typically, these are caused by inappropriate deletion and allocation of memory for the variables. Besides, pointing to a specific memory location that does not exist will also cause the error shown below.

segmentation fault (core dumped)

5.2 Conclusion

This project started out with the aim to implement an algorithm proposed by Li and Ng in (Li & Ng 2010) to help researchers in structural biology to carry out protein structure comparison. The algorithm solves the LCP problem under bottleneck distance and has a time complexity of $O((n^2m^{4.5} + n^4m^4)/\epsilon^5)$. This aim has been achieved – the algorithm is now a C++ program.

The program turned out to be fairly difficult to code. It involves the use of many mathematical operations and knowledge. It requires a strong foundation of mathematics to complete the program. Any translation, rotation and graph require application of addition, subtraction, vector formulas, rotation matrix, quadratic equation and mathematical logics. Although some other aims of the project, such as comparisons of the algorithm with other heuristic-based algorithms, has not been carried out, the main part of the project has been accomplished.

BIBLIOGRAPHY

- S. C. Li, Y. K. Ng, 2010, 'On protein structure alignment under distance constrain', *Journal of Theoretical Computer Science*, vol. 412, no.32, pp. 4187-4199
- S. C. Li, D. Bu, J. Xu, M. Li, 2008, 'Finding the largest well-predicted subset of protein structure models', *Journal of Combinatorial Pattern Matching*, vol. 5029, Springer-Verlag, pp. 44-45.
- Zhang Y, Skolnick J, 2005, 'The protein structure prediction problem could be solved using the current PDB library', *Journal of Proc Natl Acad Sci USA*, vol. 102 no. 4, pp. 1029-1034.
- Martin-Renom, Capriotti, Shindyalov and Bourne, 2009, Structure Comparison and alignment, *Journal of Structural Bioinformatics*, vo.l 44, pp. 397-412
- Holm L, Sander C, 1996, 'Mapping the protein universe', *Journal of Science*, vol. 273, no. 5275, pp. 595-603.
- Shindyalov, I.N., Bourne P.E, 1998, 'Protein structure alignment by incremental combinatorial extension (CE) of the optimal path', *Journal of Protein Engineering*, vol. 11, no.9, pp. 739–747.
- Kedem, K., Chew, L., and Elber, R., 1999, 'Unit-vector RMS (URMS) as a tool to analyze molecular dynamics trajectories', *Journal of Proteins*, vol. 37, no.4, pp. 554–564 .
- Schneider, 2002, 'A genetic algorithm for the identification of conformationally invariant regions in protein molecules', *Journal in Acta Crystallogr D Biol Crystallogr*, vol. 58, no. 2, pp. 195-208.
- Jogn M. Boyer, Wendy J. Myrvold, 2004, 'On the Cutting Egde: Simplified O(n) Planarity by Edge Addition', *Journal of Graph Algorithms and Applications* <http://jgaa.info/>, vol. 8, no. 3, pp. 241–273.

REFERENCES

- Herbert S., 1998, *C++ The Complete Reference*, 3rd edn. Osborne McGraw-Hill, Sydney.
- Weisstein, Eric W, n.d., *Law of Cosines*. Available from: <<http://mathworld.wolfram.com/LawofCosines.html>>. [3 March 2013]
- Glenn M., 2011, *Rotation About an Arbitrary Axis in 3 Dimensions*. Available from: <<http://inside.mines.edu/~gmurray/ArbitraryAxisRotation/#x1-10011>>. [20 Feb 2013]
- gamedev.net, 2003. Camera: Rotating a vector around an arbitrary axis. *gamedev.net*, *Maths and Physics*. Available from: <<http://www.gamedev.net/topic/183293-camera-rotating-a-vector-around-an-arbitrary-axis/>>
- AmBrSoft Quality Softwares, 2012, *Sphere and plane intersection*. Available from: <http://www.ambrsoft.com/TrigoCalc/Spher/SpherePlaneIntersection_.htm>. [20 Feb 2013].
- J.E. Hopcroft, R.M. Karp, 1973, 'An $n^{5/2}$ algorithm for maximum matchings in bipartite matchings in bipartite graphs', *Journal of SIAM J. Comput.*, vol.2, no. 4, pp. 225-231.

APPENDIX A COMPLETE CODING OF THE WHOLE PROGRAM

A-1 main.cpp

```

#include "pdb.h"
#include "Structure.h"
#include "Transformation.h"
#include <math.h>
#include <iostream>

using namespace std;

int mIndexNative[LONGEST_CHAIN];
int mIndexModel[LONGEST_CHAIN];

//Match both structure to get common residues.
int matchPDB(pdb* native, pdb* model)
{
    int natLength = native->mNumOfResidue;
    int modLength = model->mNumOfResidue;

    int* resNative = native->mresidueID;
    int* resModel = model->mresidueID;
    int j = 0, k = 0;
    for(int i = 0; i < natLength; i++)
    {
        for(; j < modLength; j++)
        {
            if(resNative[i] == resModel[j])
            {
                mIndexNative[k] = i;
                mIndexModel[k] = j;
                k++;
                break;
            }
            else if(resNative[i] < resModel[j])
                break;
        }
    }

    if(k == 0)
    {
        cout << "There is no common residues in the input
Structures" << endl;
        exit(0);
    }

    return k;
}

```


APPENDICES

```
int main(int argc, char** argv)
{
    if(argc !=5)
    {
        cout << "Usage: (native pdb) (model pdb) (distance
threshold) (epsilon)" << endl;
        exit(0);
    }

    double Dc, epsilon;

    Dc = atof(argv[3]);
    epsilon = atof(argv[4]);

    pdb* nativePdb = new pdb(argv[1]);
    pdb* modelPdb = new pdb(argv[2]);

    int alignLength = matchPDB(nativePdb, modelPdb);
    double* nativeCoord = new double[alignLength*3];
    double* modelCoord = new double[alignLength*3];

    //copy coordinates of common residues.
    for(int i = 0; i < alignLength; i++)
    {
        //Native
        nativeCoord[i*3] = nativePdb->mCAlpha[mIndexNative[i]*3];
        //x
        nativeCoord[i*3+1] = nativePdb-
>mCAlpha[mIndexNative[i]*3+1]; //y
        nativeCoord[i*3+2] = nativePdb-
>mCAlpha[mIndexNative[i]*3+2]; //z

        //Model
        modelCoord[i*3] = modelPdb->mCAlpha[mIndexModel[i]*3];
        //x
        modelCoord[i*3+1] = modelPdb->mCAlpha[mIndexModel[i]*3+1];
        //y
        modelCoord[i*3+2] = modelPdb->mCAlpha[mIndexModel[i]*3+2];
        //z
    }

    Structure* mNative = new Structure(nativeCoord, alignLength*3);
    Structure* mModel = new Structure(modelCoord, alignLength*3);

    Transformation* transf = new Transformation(mNative, mModel, Dc,
epsilon);

    int maxMatch = 0;
```

APPENDICES

```
//Match native to model
for(int iNative = 0; iNative < alignLength*3; iNative+=3)
{
    for(int jNative = 0; jNative < alignLength*3; jNative+=3)
    {
        for(int iModel = 0; iModel < alignLength*3; iModel+=3)
        {
            for(int jModel = 0; jModel < alignLength*3;
jModel+=3)
            {
                if(iNative != jNative && iModel != jModel)
                {
                    transf->matchPoints(iNative,
jNative, iModel, jModel);
                    maxMatch = transf->mMaxMatch;
                    if(maxMatch < transf->mMaxMatch)
                        maxMatch = transf->mMaxMatch;
                }
            }
        }
    }
}

cout << "Maximum number of residue matched: " << maxMatch << endl;

return 0;
}
```

A-2 pdb.h

```
#ifndef _PDB_H_
#define _PDB_H_
#include <iostream>
using namespace std;

#define LONGEST_CHAIN 2000

class pdb
{
public:
    char* mPDBFile;
    int mNumOfResidue;
    float* mCAlpha;
    int * mresidueID;
    char* mSeq;
    void readfile();
    void write(double** rot, double* shift);

    pdb(char* fileName);
    ~pdb();
};
#endif
```

A-3 pdb.cpp

```

#include <iostream>
#include <string.h>
#include <string>
#include <stdlib.h>
#include <math.h>
using namespace std;

#include "pdb.h"

double toDouble(string str)
{
    char* tempStr=new char[10];
    int index=0;
    double ans;

    for(int i=0; i<str.size(); i++){
        if(str[i]!=' ')
            tempStr[index++]=str[i];
    }
    tempStr[index]='\0';

    ans=atof(tempStr);

    delete [] tempStr;

    return ans;
}

int toInt(string str)
{
    char* tempStr=new char[10];
    int index=0,ans;

    for(int i=0; i<str.size(); i++){
        if(str[i]!=' ')
            tempStr[index++]=str[i];
    }

    tempStr[index]='\0';

    ans=atoi(tempStr);

    delete [] tempStr;

    return ans;
}

pdb::pdb(char* fileName){
    mPDBFile = fileName;
    mAlpha = new float[3*LONGEST_CHAIN];
    mresidueID = new int [LONGEST_CHAIN];
    readFile();
}

```

APPENDICES

```
pdb::~pdb()
{
    delete [] mCAlpha;
    delete [] mresidueID;
}

void pdb::readFile() {
    FILE *fInput=fopen(mPDBFile, "r");
    if(!fInput)
    {
        cerr << "Unsuccessfully open protein file " << mPDBFile <<
" !" << endl;
        exit(0);
    }

    char temp[80];
    double x,y,z;
    string residueName;

    mNumOfResidue = 0;

    while ( fgets ( temp, sizeof(temp), fInput ) != NULL )
    {
        string line;
        line = temp;
        if(line.substr(0, 6) == "ENDMDL")
            break;
        if(line.substr(0, 6) != "ATOM  ")
            continue;
        if(line.substr(12,4) == " CA " || line.substr(12,4) == "CA
" || line.substr(12,4) == " CA")
        {
            x = toDouble(line.substr(30,8));
            y = toDouble(line.substr(38,8));
            z = toDouble(line.substr(46,8));

            mCAlpha[mNumOfResidue*3] = x;
            mCAlpha[mNumOfResidue*3+1] = y;
            mCAlpha[mNumOfResidue*3+2] = z;

            residueName = line.substr(17,3);
            mresidueID[mNumOfResidue] = toInt(line.substr(22,4));

            mNumOfResidue++;
        }
    }

    fclose(fInput);
}
```

A-4 Structure.h

```
#ifndef _PDB_H_
#define _PDB_H_
#include <iostream>
using namespace std;

#define LONGEST_CHAIN 2000

class pdb
{
public:
    char* mPDBFile;
    int mNumOfResidue;
    float* mCAlpha;
    int * mresidueID;
    char* mSeq;
    void readfile();
    void write(double** rot, double* shift);

    pdb(char* fileName);
    ~pdb();
};
#endif
```

A-5 Structure.cpp

```

#include "Structure.h"
#include <iostream>

using namespace std;

Structure::Structure(double* coord, int length)
{
    mCoord = coord;           //coordinates for translation and
rotation
    mLength = length;
}

Structure::~Structure()
{
    delete mCoord;
}

/*
    Translate the whole structure
*/
void Structure::translateStruct(double* transStep)
{
    for(int i = 0; i < mLength; i += 3)
    {
        diff(mCoord + i, transStep, mCoord + i);
    }
}

/*
    Rotates structure about an axis
    point: a point that the vector(axis) pass through
    axis : vector(axis of rotation)
    theta: rotation angle
*/
void Structure::rotateStruct(double* point, double* axis, double theta)
{
    for(int i = 0; i < mLength; i+=3)
    {
        rotatesAboutVector(mCoord+i, point, axis, theta, mCoord+i);
    }
}

/*
    Transform structure such that two points is along y-axis
*/
void Structure::transformStruct(double* q1, double* q2)
{
    double* transStep = new double[3];
    bool q2IsOrigin = false;
}

```

```

//when one of the point is at origin, no need to do translation,
just perform rotation.
if((q1[0] == 0 && q1[1] == 0 && q1[2] == 0))
{
    ;
}

else if(q2[0] == 0 && q2[1] == 0 && q2[2] == 0)
{
    q2IsOrigin = true;
}
else
{
    //get the translation step of q1 to origin.
    transStep = q1;

    //translate whole structure so that q1 is at origin.
    translateStruct(transStep);
}

if(q2IsOrigin) //ONLY if q2 is origin, we rotate q1.
{
    rotateStruct(q1);
}
else
{
    //Rotates whole structure so that q2 is on y-axis.
    rotateStruct(q2);
}
}

/*
Rotate structure Q so that q1 and q2 are along y-axis
*/
void Structure::rotateStruct(double* q2)
{
    double a, b, c;
    double origin[3] = {0, 0, 0};
    double cosC, theta;
    double point[3] = {0, q2[1], 0};
    double vector1[3], vector2[3], vector3[3];

    //get vector1 (a), vector2 (b), vector3 (c), where c = a x b cross
product, c = axis of rotation

    diff(q2, origin, vector1); //a
    diff(point, origin, vector2); //b
    crossProduct(vector1, vector2, vector3); //c:axis of rotation

    normalize(vector3);

    //get angle of rotation using law of cosines
    a = calDist(q2, origin);
    b = calDist(point, origin);
    c = calDist(q2, point);

    cosC = (a*a + b*b - c*c) / (2*a*b);

```


APPENDICES

```
    if(cosC > 1)
        cosC = 1;
    if(cosC < -1)
        cosC = -1;

    theta = acos(cosC);

    rotateStruct(origin, vector3, theta);
}
```

A-5 MathOperation.h

```
#ifndef _MATHS_OPERATION_H_
#define _MATHS_OPERATION_H_
#include <math.h>
#define PI 3.14159265

double calDist(double* A, double* B);
void add(double* A, double* B, double* ans);
void diff(double* A, double* B, double *ans);
void rotatesAboutVector(double* xyz, double* abc, double* uvw, double
theta, double* newCoord);
void rotatesAboutArbLine(double x, double y, double z, double a, double
b, double c, double u, double v, double w, double theta, double*
newCoord);
void normalize(double* u);
double dotProduct(double* A, double* B);
void crossProduct(double* A, double* B, double* ans);
void multiply(double num, double* A, double* ans);
void rotateVectorAbtVector(double* A, double* axis, double theta,
double* ans);

#endif
```

A-6 MathsOperation.cpp

```

#include "MathsOperation.h"

double calDist(double* A, double* B)
{
    double sum = 0;
    sum += (A[0] - B[0]) * (A[0] - B[0]);
    sum += (A[1] - B[1]) * (A[1] - B[1]);
    sum += (A[2] - B[2]) * (A[2] - B[2]);

    return sqrt(sum);
}

void add(double* A, double* B, double* ans)
{
    ans[0] = A[0] + B[0];
    ans[1] = A[1] + B[1];
    ans[2] = A[2] + B[2];
}

void diff(double* A, double* B, double *ans)
{
    ans[0] = A[0] - B[0];
    ans[1] = A[1] - B[1];
    ans[2] = A[2] - B[2];
}

/*
    //rotatesAboutVector and rotatesAboutArbLine are the same
    //rotatesAboutVector is to convert double* into 3 double
    xyz : point to be rotate
    abc : a point that the rotation axis passes through
    uvw : direction vector (unit vector)
    theta : angle of rotation
    newCoord : the rotated point
*/
void rotatesAboutVector(double* xyz, double* abc, double* uvw, double
theta, double* newCoord)
{
    rotatesAboutArbLine(xyz[0], xyz[1], xyz[2], abc[0], abc[1],
abc[2], uvw[0], uvw[1], uvw[2], theta, newCoord);
}

```

APPENDICES

```
void rotatesAboutArbLine(double x, double y, double z, double a, double
b, double c, double u, double v, double w, double theta, double*
newCoord)
{
    double costheta = cos(theta);
    double sintheta = sin(theta);
    double oneMinusCosTheta = 1 - costheta;
    double v2 = v*v;
    double u2 = u*u;
    double w2 = w*w;
    newCoord[0] = (a*(v2 + w2) - u*(b*v + c*w - u*x - v*y - w*z)) *
oneMinusCosTheta + x*costheta + (-c*v + b*w - w*y + v*z)*sintheta;
    newCoord[1] = (b*(u2 + w2) - v*(a*u + c*w - u*x - v*y - w*z)) *
oneMinusCosTheta + y*costheta + (c*u - a*w + w*x - u*z)*sintheta;
    newCoord[2] = (c*(u2 + v2) - w*(a*u + b*v - u*x - v*y - w*z)) *
oneMinusCosTheta + z*costheta + (-b*u + a*v - v*x + u*y)*sintheta;
}

void normalize(double* u)
{
    double norm = sqrt(u[0]*u[0] + u[1]*u[1] + u[2]*u[2]);
    u[0] = u[0]/norm;
    u[1] = u[1]/norm;
    u[2] = u[2]/norm;
}

double dotProduct(double* A, double* B)
{
    return (A[0]*B[0] + A[1]*B[1] + A[2]*B[2]);
}

void crossProduct(double* A, double* B, double* ans)
{
    ans[0] = A[1]*B[2] - A[2]*B[1];
    ans[1] = A[2]*B[0] - A[0]*B[2];
    ans[2] = A[0]*B[1] - A[1]*B[0];
}

void multiply(double num, double* A, double* ans)
{
    ans[0] = num*A[0];
    ans[1] = num*A[1];
    ans[2] = num*A[2];
}
```

APPENDICES

```
/*
    A:Vector to be rotate
    axis:axis
    V = (A - ((axis.A)*axis)*costheta) + ((A X axis)*sintheta) +
    (axis.A)*axis
*/
//must use normalized vector
void rotateVectorAbtVector(double* A, double* axis, double theta,
double* ans)
{
    double cross[3], dotAxis[3], dotAxisCosTheta[3], ans1[3], ans2[3],
ans3[3];
    double dot;

    dot = dotProduct(A,axis);           //(axis.A)
    multiply(dot, axis, dotAxis);      //(axis.A)*axis
    multiply(cos(theta), dotAxis, dotAxisCosTheta);
//((axis.A)*axis)*costheta

    crossProduct(A,axis,cross);        //(A X axis)

    diff(A, dotAxisCosTheta, ans1);    //(A -
((axis.A)*axis)*costheta)
    multiply(sin(theta), cross, ans2); //((A X axis)*sintheta)
    ans3[0] = dotAxis[0];               //(axis.A)*axis
    ans3[1] = dotAxis[1];
    ans3[2] = dotAxis[2];

    add(ans1, ans2, ans);
    add(ans, ans3, ans);
}
```

A-7 Transformation.h

```

#ifndef _TRANSFORMATION_H_
#define _TRANSFORMATION_H_
#include "Structure.h"
#include "MathsOperation.h"
#include "Graph.h"

class Transformation
{
private:
    /*combine all thetas found*/
    int len;
    typedef struct
    {
        double angle;
        int** matching;           //adjacency matrix used to
store the graph of p and q matching
    }ROTATION;

public:
    Structure *mNative;
    Structure *mModel;
    double mDc, mEpsilon;
    double mThres, mStepSize;

    ROTATION* mRotInterval;           //rotation interval and
its maching.
    int mRotItvIndex;           //index of rotInterval
    int mMaxMatch;           //the maximum
number of matched for specific p1,p2 and q1,q2

    Transformation(Structure* native, Structure* model, double
Dc, double epsilon);
    ~ Transformation();

    void matchPoints(int iNative, int jNative, int iModel, int
jModel);

    bool isInThres(double* center, double* points);
    void tryP1InGrid(double* p1, double* center, double* p2,
double* q2);

    bool isRadius(double* center, double* coord, double radius);
    void formSphereCap(double* oriP1, double* p1, double* p2,
double* q2, double radius);
    void movePtoPlace(double* newP1, double* p1, double* newP2,
double* p2);
    void findAngleInOut();
    void getIntersectPoint(double* p, double* q, int j, int i);

    void copyMatching(int** A, int** B);

```

APPENDICES

```
tempArr2); void moveBackwards(int start, int end, double temp2, int**
angle); void insertTheta2(int start, int end, int i, int j, double
void setRotInterval(double theta1, double theta2, int i,
int j);
void getMaxMatch();
void initializeMatch();
};
#endif
```

A-8 Transformation.cpp

APPENDICES

```
#include "Transformation.h"
#include <iostream>

using namespace std;

Transformation::Transformation(Structure* native, Structure* model,
double Dc, double epsilon)
{
    mNative = native;
    mModel = model;
    mDc = Dc;
    mEpsilon = epsilon;
    mThres = (1 + mEpsilon)* mDc;
    mStepSize = mEpsilon * mDc / 3;
    len = mModel->mLength/3; //total points in mNative or mModel

    mRotInterval = new ROTATION[2*len*len];

    for(int i = 0; i < 2*len*len; i++)
    {
        mRotInterval[i].matching = new int*[len];

        for(int j = 0; j < len; j++)
        {
            mRotInterval[i].matching[j] = new int[len];

            for(int k = 0; k < len; k++)
            {
                mRotInterval[i].matching[j][k] = 0;
            }
        }
    }

    mRotItvIndex = 0;
    mMaxMatch = 0;
}

Transformation::~~ Transformation()
{
    delete[] mNative;
    delete[] mModel;
    delete[] mRotInterval;
}
```


APPENDICES

```

void Transformation::matchPoints(int iNative, int jNative, int iModel,
int jModel)
{
    //Translate p1 and p2 to match q1 and q2.
    //iNative = q1    iModel = p1
    //jNative = q2    jModel = p2

    //-----Translation to match model to native-----
    -----
    double distP = calDist(mModel->mCoord + iModel, mModel->mCoord +
jModel);
    double distQ = calDist(mNative->mCoord + iNative, mNative->mCoord
+ jNative);

    if(fabs(distP - distQ) <= 2*mDc)
    {

        //Transform Q structure such that q1 and q2 is along y-axis
        mNative->transformStruct(mNative->mCoord + iNative,
mNative->mCoord + jNative);

        //Discretize q1 and try p1 in the grid
        tryP1InGrid(mModel->mCoord + iModel, mNative->mCoord +
iNative, mModel->mCoord + jModel, mNative->mCoord + jNative);

    }
}

/*
    Examine p1 in the grid of q1 (all possible positions)
    p1 : coord of p1
    q1 : sphere center
    p2 : another point of structure P.
    q2 : point of structure Q.
*/
void Transformation::tryP1InGrid(double* p1, double* q1, double* p2,
double* q2)
{
    double start[3] = {q1[0] - mThres, q1[1] - mThres, q1[2] -
mThres}; //Store each start point of each exist
    double end[3] = {q1[0] + mThres, q1[1] + mThres, q1[2] + mThres};
//Store each end point of each exist
    double coord[3]; // coord = p1 on the grid
    double distQ = calDist(q1,q2); //distance between q1 and q2

    double radius = calDist(p1, p2);
    int count=1;
    for(double x = start[0]; x <= end[0]; x += mStepSize)
    {
        coord[0] = x;
        for(double y = start[1]; y <= end[1]; y += mStepSize)
        {
            coord[1] = y;
            for(double z = start[2]; z <= end[2]; z += mStepSize)
            {
                coord[2] = z;
            }
        }
    }
}

```

```

        if(isInThres(q1, coord)) //is in sphere of q1
        {
            //check whether distance(p1,q2) > mThres
            if(calDist(coord,q2) > mThres)
            {
                //check whether p2 is in sphere of
                if(radius >= (distQ - mThres) &&
                radius <= (distQ + mThres))
                {
                    formSphereCap(p1, coord, p2,
                    q2, radius);
                }
            }
        }
    }
}

/*
    Check whether the points (grid on cube) is within thres (sphere)
    center : sphere center
    point : point on the grid on cube
*/
bool Transformation::isInThres(double* center, double* point)
{
    if(calDist(center, point) <= mThres)
        return true;
    return false;
}

/*
    Form a sphere cap for p2 on q2 grid, try all coordinates on
    sphere cap
    oriP1: the original p1 before putting on grid
    p1 : center of the sphere cap
    p2 : point to form sphere cap
    q2 : center of sphere where the sphere cap is inside
    radius: distance between p1 and p2, radius of forming
    sphere cap
*/
void Transformation::formSphereCap(double* oriP1, double* p1, double*
p2, double* q2, double radius)
{
    double totalangle,totalAOut,PI2;           //totaltheta must be at
most 360 to stop the rotation.
    double coord[3];                           //point on sphere
cap (new p2)
    double middleCoord[3];                       //center point of
each moving up and down/left and right
    double oriCoord[3];                           //the original
point before any rotation
    double vector1[3], vector2[3];               //axis of rotation

```

APPENDICES

```
double angle = mStepSize/radius;    //rotation angle
PI2 = 2*PI;
bool isWithinQ2 = false;            //is the coordinate
within sphere of q2

//start to form sphere cap with p1 point that form straight line
parallel to y-axis
middleCoord[0] = p1[0];
middleCoord[1] = p1[1] + radius;
middleCoord[2] = p1[2];

//copy the coordinates to prepare for rotation
coord[0] = middleCoord[0];
coord[1] = middleCoord[1];
coord[2] = middleCoord[2];
oriCoord[0] = middleCoord[0];
oriCoord[1] = middleCoord[1];
oriCoord[2] = middleCoord[2];

//vector1 to rotate p2 on sphere cap (up and down): firstly parallel
to z, will be changed later
vector1[0] = 0;    //parallel to z
vector1[1] = 0;
vector1[2] = 1;

//point that passes through the vector1 and vector2(axis of rotation)
= p1

//vector2 to rotate p2 on sphere cap (left and right) : parallel to
x-axis
vector2[0] = 1;
vector2[1] = 0;
vector2[2] = 0;

totalAOut = 0;

do
{
    /*prepare to rotates up and down*/
    //going up and down

    totalangle = 0;

    //rotates p2 up and down around vector1
    do
    {

        rotatesAboutVector(coord, p1, vector1, angle, coord);
        //going up or down

        totalangle += angle;
        //check whether is in sphere of q2
        isWithinQ2 = isInThres(q2, coord);
        if(isWithinQ2)
        {
```

APPENDICES

```

//Move other P respectively
movePtoPlace(p1, oriP1, coord, p2); //p1: new
p1 on grid;          oriP1: p1 before putting on grid;

    //coord: p2 on sphere cap;    p2: p2 before putting on sphere cap
    //form rotation axis and rotates
    findAngleInOut();
    getMaxMatch(); //get most number of residue
matched from the maximum bipartite matching for each rotation angle.

    }

    } while(totalangle < PI2);

    //reset coord back to original coord before rotates up&down,
to prepare to move left/right

    coord[0] = middleCoord[0];
    coord[1] = middleCoord[1];
    coord[2] = middleCoord[2];
    //rotates p2 left and right and rotates vector1 together
with same theta

    rotatesAboutVector(coord, p1, vector2, angle, coord);
//going left&right
//rotates also vector1
rotateVectorAbtVector(vector1, vector2, angle, vector1);
//normalize vector1
normalize(vector1);
totalAOut += angle;

    middleCoord[0] = coord[0];
    middleCoord[1] = coord[1];
    middleCoord[2] = coord[2];

    } while(totalAOut < PI2);
}

/*
Move all points in structure P respectively after fixing p1 and
p2

    newP1 : the point on grid(fixed)
    p1 : the old point before putting on grid
    newP2 : the point on sphere cap(fixed)
    p2 : the old point before putting on sphere cap
*/
void Transformation::movePtoPlace(double* newP1, double* p1, double*
newP2, double* p2)
{
    //get translation step to translate from old P1 to new P1.
    double transStep[3];
    double vector1[3];
    double vector2[3];
    double vector3[3];
    double theta, a, b, c, cosC;

```

APPENDICES

```
diff(p1, newP1, transStep);
mModel->translateStruct(transStep);
//get vector1 (a), vector2 (b), vector3 (c), where c = a x b
cross product, c = axis of rotation
diff(newP1, p2, vector1); //a
diff(newP1, newP2, vector2); //b
crossProduct(vector1, vector2, vector3); //c:axis of rotation

normalize(vector3);

//get angle of rotation using law of cosines
a = calDist(newP1, p2);
b = calDist(newP1, newP2);
c = calDist(p2, newP2);

cosC = (a*a + b*b - c*c) / (2*a*b);

if(cosC > 1)
    cosC = 1;
if(cosC < -1)
    cosC = -1;

theta = acos(cosC);

mModel->rotateStruct(newP1, vector3, theta);
}

void Transformation::initializeMatch()
{
    for(int i = 0; i < 2*len*len; i++)
    {
        for(int j = 0; j < len; j++)
        {
            for(int k = 0; k < len; k++)
            {
                mRotInterval[i].matching[j][k] = 0;
            }
        }
    }

    mRotItvIndex = 0;
}

/*
    Get the angle of rotation that moves P into and Out of Contact
    with Q, by using intersection between the sphere of Q and unit
    circle(path of p)
*/
void Transformation::findAngleInOut()
{
    int indexp, indexq;
    mRotItvIndex = 0;

    for(int i = 0; i < mNative->mLength; i+=3)
```

APPENDICES

```

{
    indexq = i/3;
    for(int j = 0; j < mModel->mLength; j+=3)
    {
        indexp = j/3;

        getIntersectPoint(mModel->mCoord + j, mNative->mCoord
+ i, indexp, indexq);
    }
}

/*
    Get intersection point of circle centered at (0,B,0) and circle centered
    at (X0, B, Z0) - intersection circle of plane Y=B and sphere of q
    Circle 1: (intersection of plane and sphere) equation: (X-x0)2 + (Y-
B)2 + (Z-z0)2 = ((1+eps)Dc)2 - (y0-B)2
    Circle 2: (path of p) equation: X2 + (Y-B)2 + Z2 = (x1)2 + (z1)2
    p : any point of p (x1,B,z1)
    q : any point of q (x0,y0,z0)
    B : plane to intersect with sphere.
*/
void Transformation::getIntersectPoint(double* p, double* q, int indexp,
int indexq)
{
    double B = p[1]; //y-coordinate
    double R = sqrt(mThres*mThres - (q[0]-B)*(q[0]-B)); //Radius of Circle1
    double r = sqrt(p[0]*p[0] + p[2]*p[2]); //radius of Circle2
    double R2, r2; //squared of R and squared of r.
    double c2center[3] = {0,B,0}; //center of circle 2

    R2 = R*R;
    r2 = r*r;

    double point1[3],point2[3]; //intersection points between two circles
    double a,b,c,cosC, angle1, angle2;

    double q02, q22; //squared of q[0] and squared of q[2].
    q02 = q[0]*q[0];
    q22 = q[2]*q[2];

    double dist = calDist(p, q); //distance between two centers of circle
    int yes;

    if(dist <= mThres) //p is approximately matched to q
    {
        angle1 = 0;
        angle2 = 2*PI;
    }
    else
    {
        //(-b +- sqrt(b2-4ac)) / 2a
        a = 4*q02 + 4*q22;
        b = -4*r2*q[2] + 4*R2*q[2] - 4*q02*q[2] - 4*q22*q[2];

```

APPENDICES

```

        c = r2*r2 - 2*q02*r2 - 2*R2*r2 + 2*r2*q22 + R2*R2 - 2*R2*q02 -
        2*R2*q22 + q02*q02 + 2*q02*q22 + q22*q22;

        point1[2] = (-b + sqrt(b*b-4*a*c))/ 2*a; //z1
        point2[2] = (-b - sqrt(b*b-4*a*c))/ 2*a; //z2

        point1[1] = B; //y1
        point2[1] = B; //y2

        point1[0] = (q22 + q02 - 2*q[2]*point1[2] - R2 + r2) / 2*q[0];
//x1:sub z1
        point2[0] = (q22 + q02 - 2*q[2]*point2[2] - R2 + r2) / 2*q[0];
//x2:sub z2

        //get angle using law of cosines
        //angle that moves P into Q
        a = calDist(c2center, p);
        b = calDist(c2center, point1);
        c = calDist(p, point1);

        cosC = (a*a + b*b - c*c) / (2*a*b);

        angle1 = acos(cosC);

        //angle that moves P out of Q using law of cosines
        a = calDist(c2center, point2);
        b = calDist(point1, point2);
        c = calDist(c2center, point1);

        cosC = (a*a + b*b - c*c) / (2*a*b);

        angle2 = acos(cosC);
    }

    setRotInterval(angle1, angle2, indexp, indexq);
}

////////////////////////////////////Calculate
rotation
interval////////////////////////////////////

/*
    Copy matching matrix A to matrix B
    A: Matrix to be copied
    B: Copied Matrix
*/
void Transformation::copyMatching(int** A, int** B)
{
    for(int i = 0; i < len; i++)
    {
        for(int j = 0; j < len; j++)
        {
            B[i][j] = A[i][j];
        }
    }
}

```

APPENDICES

```
/*
    Move the elements in the array of mRotInterval backwards, based
    on start and end.
        Start: position to start shift backwards.
        End : mRotInterval array size - mRotItvIndex(usually).
    shift until the end of the array
        temp2 : temporary variable that holds the new value of
    angle[start].
        tempArr2: temporary array that holds the new value of
    matching[start].
*/
void Transformation::moveBackwards(int start, int end, double temp2,
int** tempArr2)
{
    double temp;
    int** tempArr;

    tempArr = new int*[len];
    for(int i = 0; i < len; i++)
        tempArr[i] = new int[len];

    for(int l = start; l < end; l++)
    {
        temp = mRotInterval[l].angle;
        mRotInterval[l].angle = temp2;
        temp2 = temp;

        copyMatching(mRotInterval[l].matching, tempArr);
        copyMatching(tempArr2, mRotInterval[l].matching);
        copyMatching(tempArr, tempArr2);

    }

    //destroy tempArr in memory address
    for(int i = 0; i < len; i++)
        delete[] tempArr[i];
    delete[] tempArr;
}

/*
    Put in mTheta[i][j]->mTheta2 value in the rotation interval -
    mRotInterval
        Start: starting position of mRotInterval to trace for
    theta2. (theta2 must always starts from one element behind theta1)
        End : mRotInterval array size - mRotItvIndex(usually).
    shift until the end of the array
        i      : index of p structure
        j      : index of q structure
        angle : mTheta[i][j]->mTheta2's value
*/
void Transformation::insertTheta2(int start, int end, int i, int j,
double angle)
{
    int** tempArr;

    tempArr = new int*[len];
```


APPENDICES

```

for(int x = 0; x < len; x++)
    tempArr[x] = new int[len];

double totalTheta = 0;
double temp, temp2;
bool isSet = false;

for(int k = start; k < end; k++)
{
    totalTheta += mRotInterval[k].angle;
    if(angle < totalTheta)
    {
        totalTheta = totalTheta - mRotInterval[k].angle;
//exclude angle[k]
        temp = mRotInterval[k].angle;
        mRotInterval[k].angle = angle - totalTheta;
        temp2 = mRotInterval[k+1].angle;
        mRotInterval[k+1].angle = temp - angle;

        copyMatching(mRotInterval[k+1].matching, tempArr);
        copyMatching(mRotInterval[k].matching,
mRotInterval[k+1].matching);

        //set matching of new matching[k]
        //set also matching for the angle before (for whole
totalTheta need to set)
        for(int x = start; x <= k; x++)
        {
            mRotInterval[x].matching[i][j] = 1;
        }
        mRotItvIndex++;

        //move backwards
        moveBackwards(k+2, mRotItvIndex, temp2, tempArr);
        isSet = true;

        break;
    }
    else if(angle == totalTheta)
    {
        for(int l = start; l <= k ; l++)
        {
            mRotInterval[l].matching[i][j] = 1;
        }
        isSet = true;
        break;
    }
}

// angle > totalTheta => add new theta at behind
if(!isSet)
{
    mRotInterval[mRotItvIndex].angle = angle - totalTheta;
    for(int l = start; l <= mRotItvIndex ; l++)
    {
        mRotInterval[l].matching[i][j] = 1;
    }
}

```

APPENDICES

```

        mRotItvIndex++;
    }
    //destroy tempArr in memory address
    for(int i = 0; i < len; i++)
        delete[] tempArr[i];
    delete[] tempArr;
}

/*
    Calculate rotation interval and matching of p and q in each
    rotation interval - to do bipartite matching(next step)
    thetal : angle that moves p to q
    theta2 : angle that moves p out of q
    i : index of p structure
    j : index of q structure
*/
void Transformation::setRotInterval(double thetal, double theta2, int i
int j)
{
    double totalTheta = 0;
    double temp, temp2;
    bool isSet = false;

    int** tempArr;
    /*initialize*/
    tempArr = new int*[len];

    for(int x = 0; x < len; x++)
        tempArr[x] = new int[len];

    totalTheta = 0;
    isSet = false;
    if(mRotItvIndex == 0) //first element, first theta to be insert into
array
    {
        if(thetal > 0)
        {
            mRotInterval[mRotItvIndex].angle = thetal;//angle that
moves p to q
            mRotInterval[mRotItvIndex].matching[i][j] = -1;
            mRotItvIndex++;
        }

        mRotInterval[mRotItvIndex].angle = theta2;//angle that moves p
out of q
        mRotInterval[mRotItvIndex].matching[i][j] = 1;
        mRotItvIndex++;
    }
    else
    {
        if(thetal == 0) //insert only theta2 when thetal = 0
        {
            insertTheta2(0, mRotItvIndex, i, j, theta2);
        }
        else
        {
            for(int k = 0; k < mRotItvIndex; k++)

```

```

        {
            totalTheta += mRotInterval[k].angle;

            if(theta1 < totalTheta)
            {
                totalTheta = totalTheta - mRotInterval[k].angle; //exclude angle[k]

                temp = mRotInterval[k].angle; //copy angle[k] to temp
                mRotInterval[k].angle = theta1 - totalTheta; //replace angle[k] with new theta
                temp2 = mRotInterval[k+1].angle; //copy angle[k+1] to temp2
                mRotInterval[k+1].angle = temp - theta1; //replace angle[k+1] with new theta

                copyMatching(mRotInterval[k+1].matching, tempArr); //copy matching[k+1] to tempArr
                copyMatching(mRotInterval[k].matching, mRotInterval[k+1].matching); //copy matching[k] to matching[k+1]
                //set matching of new matching[k]
                //set also matching for the angle before
                (for whole totalTheta also need to set)
                for(int x = 0; x <= k; x++)
                {
                    mRotInterval[x].matching[i][j] = -1;
                }

                mRotItvIndex++;
                //move backwards
                moveBackwards(k+2, mRotItvIndex, temp2, tempArr);

                //theta2 - must be the at position that
                next to the theta1
                insertTheta2(k+1, mRotItvIndex, i, j, theta2);

                isSet = true;
                break;
            }
            else if(theta1 == totalTheta)
            {
                for(int l = 0; l <= k ; l++)
                {
                    mRotInterval[l].matching[i][j] = -1;
                }

                //theta2 - must be the at position that
                next to the theta1
                insertTheta2(k+1, mRotItvIndex, i, j, theta2);

                isSet = true;
                break;
            }
        }

```

```

    }
    // theta1 > totalTheta => add new theta at behind
    if(!isSet)
    {
        mRotInterval[mRotItvIndex].angle = theta1 - totalTheta;

        for(int l = 0; l <= mRotItvIndex ; l++)
        {
            mRotInterval[l].matching[i][j] = -1;
        }
        mRotItvIndex++;

        //theta2 - must be at the position that next to
        insertTheta2(mRotItvIndex, mRotItvIndex, i, j,
        theta2);
    }
}

//destroy tempArr in memory address
for(int x = 0; x < len; x++)
    delete[] tempArr[x];
delete[] tempArr;
}

//////////////////////////////////////Get maximum
bipartite
matching//////////////////////////////////////
void Transformation::getMaxMatch()
{
    int max = 0;

    for(int i = 0; i < mRotItvIndex; i++)
    {
        max = hopcroft_karp(len, mRotInterval[i].matching);

        if(mMaxMatch < max)
            mMaxMatch = max;
    }
}

```

A-9 Graph.h

```
#ifndef _GRAPH_H_
#define _GRAPH_H_
#define NIL 0
#define INF (1<<28)

#include <iostream>
#include <queue>
using namespace std;

void insertMatch(int** matching, int length);
bool bfsearch(int n, int* match, int* dist);
bool dfsearch(int v, int* match, int* dist);
int hopcroft_karp(int length, int** matching);

#endif
```

A-10 Graph.cpp

```

/*
    Modified from code on web:
    http://zobayer.blogspot.com/2010/05/maximum-matching.html
*/
#include "Graph.h"
// nLeft: number of nodes on left side, nodes are numbered 1 to n,G1
//      (p structure)
// nRight: number of nodes on right side, nodes are numbered n+1 to
// n+m,G2      (q structure)
// G = NIL[0] U... G1[G[1---n]] U... G2[G[n+1---n+m]]

vector<vector<int>> > G;
void insertMatch(int** matching, int length)
{
    for(int i = 0; i < length; i++)
    {
        for(int j = 0; j < length; j++)
        {
            if(matching[i][j] == 1)
            {
                j += length;          //v += nLeft
                G[i+1].push_back(j+1); //G[u].push_back(v);
            }
        }
    }
}

bool bfsearch(int n, int* match, int* dist)
{
    int u, v, length;
    queue<int> Q;
    //for vertex in G1: from 1 to n
    for(int i = 1; i <= n; i++)
    {
        if(match[i] == NIL) //Pair_G1 = match
        {
            dist[i] = 0;
            Q.push(i);
        }
        else
        {
            dist[i] = INF;
        }
    }
    dist[NIL] = INF;

    while(!Q.empty())
    {
        v = Q.front();Q.pop();//vertex = Dequeue(Q), vertex in G1 =
u

        if(v != NIL)
        {
            length = G[v].size();

```

```

        //for each u in Adj[v]
        for(int i = 0; i < length; i++)
        {
            u = G[v][i];

            if(dist[match[u]] == INF)
            {
                dist[match[u]] = dist[v] + 1;
                Q.push(match[u]);
            }
        }
    }
    return (dist[NIL] != INF);
}

bool dfssearch(int v, int* match, int* dist)
{
    int u, length;

    if(v != NIL)
    {
        length = G[v].size();
        //for each u in Adj[v]
        for(int i = 0; i < length; i++)
        {
            u = G[v][i];
            if(dist[match[u]] == dist[v] + 1)
            {
                if(dfssearch(match[u], match, dist))
                {
                    match[u] = v;
                    match[v] = u;
                    return true;
                }
            }
        }
        dist[v] = INF;
        return false;
    }
    return true;
}

int hopcroft_karp(int length, int** matching)
{
    G.resize(length+1);
    int length2 = length*length;
    int *match = new int[length2 + 1];
    int *dist = new int[length2 + 1];

    for(int i = 0; i <= length2; i++)
    {
        match[i] = 0;
        dist[i] = 0;
    }
    insertMatch(matching, length);
}

```

APPENDICES

```
int totalMatch = 0;

while(bfsearch(length, match, dist))
{
    for(int i = 1; i <= length; i++)
    {
        if(match[i] == NIL)
        {
            if(dfsearch(i, match, dist))
            {
                totalMatch++;
            }
        }
    }
}

delete match;
delete dist;
return totalMatch;
}
```


A-11 Example of .pdb file

ATOM	1	N	MET	1	1.040	0.374	-0.952	1.00	0.00
ATOM	2	CA	MET	1	0.000	0.000	0.000	1.00	0.00
ATOM	3	C	MET	1	0.598	-0.376	1.349	1.00	0.00
ATOM	4	O	MET	1	-0.110	-0.827	2.249	1.00	0.00
ATOM	5	CB	MET	1	-0.835	-1.146	-0.550	1.00	0.00
ATOM	6	N	SER	2	1.907	-0.187	1.484	1.00	0.00
ATOM	7	CA	SER	2	2.623	-0.615	2.679	1.00	0.00
ATOM	8	C	SER	2	2.169	0.169	3.904	1.00	0.00
ATOM	9	O	SER	2	2.381	-0.258	5.039	1.00	0.00
ATOM	10	CB	SER	2	4.124	-0.467	2.479	1.00	0.00
ATOM	11	N	PHE	3	1.543	1.316	3.667	1.00	0.00
ATOM	12	CA	PHE	3	0.979	2.118	4.746	1.00	0.00
ATOM	13	C	PHE	3	-0.184	1.398	5.417	1.00	0.00
ATOM	14	O	PHE	3	-0.676	1.830	6.460	1.00	0.00
ATOM	15	CB	PHE	3	0.534	3.474	4.222	1.00	0.00
ATOM	16	N	ILE	4	-0.620	0.298	4.814	1.00	0.00
ATOM	17	CA	ILE	4	-1.688	-0.516	5.381	1.00	0.00
ATOM	18	C	ILE	4	-1.262	-1.139	6.704	1.00	0.00
ATOM	19	O	ILE	4	-2.042	-1.195	7.655	1.00	0.00
ATOM	20	CB	ILE	4	-2.113	-1.594	4.397	1.00	0.00
ATOM	21	N	GLU	5	-0.020	-1.607	6.759	1.00	0.00
ATOM	22	CA	GLU	5	0.522	-2.203	7.975	1.00	0.00
ATOM	23	C	GLU	5	0.414	-1.244	9.153	1.00	0.00
ATOM	24	O	GLU	5	-0.144	-1.586	10.195	1.00	0.00
ATOM	25	CB	GLU	5	1.968	-2.622	7.761	1.00	0.00
ATOM	26	N	LYS	6	0.951	-0.040	8.981	1.00	0.00
ATOM	27	CA	LYS	6	0.883	0.984	10.017	1.00	0.00
ATOM	28	C	LYS	6	-0.554	1.224	10.461	1.00	0.00
ATOM	29	O	LYS	6	-0.842	1.292	11.656	1.00	0.00
ATOM	30	CB	LYS	6	1.511	2.278	9.524	1.00	0.00
ATOM	31	N	MET	7	-1.454	1.351	9.492	1.00	0.00
ATOM	32	CA	MET	7	-2.867	1.562	9.781	1.00	0.00
ATOM	33	C	MET	7	-3.436	0.419	10.612	1.00	0.00
ATOM	34	O	MET	7	-4.200	0.642	11.552	1.00	0.00
ATOM	35	CB	MET	7	-3.655	1.724	8.490	1.00	0.00
ATOM	36	N	ILE	8	-3.060	-0.806	10.261	1.00	0.00
ATOM	37	CA	ILE	8	-3.501	-1.984	10.998	1.00	0.00
ATOM	38	C	ILE	8	-2.935	-1.993	12.412	1.00	0.00
ATOM	39	O	ILE	8	-3.627	-2.350	13.365	1.00	0.00
ATOM	40	CB	ILE	8	-3.103	-3.252	10.257	1.00	0.00
ATOM	41	N	GLY	9	-1.673	-1.598	12.541	1.00	0.00
ATOM	42	CA	GLY	9	-1.020	-1.534	13.843	1.00	0.00
ATOM	43	C	GLY	9	-1.722	-0.545	14.764	1.00	0.00
ATOM	44	O	GLY	9	-1.941	-0.827	15.942	1.00	0.00
ATOM	46	N	SER	10	-2.074	0.615	14.220	1.00	0.00
ATOM	47	CA	SER	10	-2.805	1.625	14.975	1.00	0.00
ATOM	48	C	SER	10	-4.147	1.089	15.456	1.00	0.00
ATOM	49	O	SER	10	-4.542	1.314	16.600	1.00	0.00
ATOM	50	CB	SER	10	-3.002	2.877	14.133	1.00	0.00
ATOM	51	N	LEU	11	-4.844	0.379	14.576	1.00	0.00
ATOM	52	CA	LEU	11	-6.149	-0.182	14.906	1.00	0.00
ATOM	53	C	LEU	11	-6.027	-1.282	15.953	1.00	0.00
ATOM	54	O	LEU	11	-6.921	-1.470	16.777	1.00	0.00
ATOM	55	CB	LEU	11	-6.830	-0.713	13.654	1.00	0.00

APPENDICES

ATOM	56	N	ASN	12	-4.914	-2.007	15.915	1.00	0.00
ATOM	57	CA	ASN	12	-4.711	-3.146	16.802	1.00	0.00
ATOM	58	C	ASN	12	-4.910	-2.752	18.259	1.00	0.00
ATOM	59	O	ASN	12	-5.491	-3.504	19.042	1.00	0.00
ATOM	60	CB	ASN	12	-3.326	-3.739	16.595	1.00	0.00
ATOM	61	N	ASP	13	-4.425	-1.568	18.618	1.00	0.00
ATOM	62	CA	ASP	13	-4.443	-1.122	20.006	1.00	0.00
ATOM	63	C	ASP	13	-5.557	-0.110	20.245	1.00	0.00
ATOM	64	O	ASP	13	-5.622	0.518	21.301	1.00	0.00
ATOM	65	CB	ASP	13	-3.096	-0.530	20.390	1.00	0.00
ATOM	66	N	LYS	14	-6.433	0.042	19.257	1.00	0.00
ATOM	67	CA	LYS	14	-7.539	0.986	19.354	1.00	0.00
ATOM	68	C	LYS	14	-8.480	0.618	20.494	1.00	0.00
ATOM	69	O	LYS	14	-8.594	-0.551	20.864	1.00	0.00
ATOM	70	CB	LYS	14	-8.298	1.049	18.037	1.00	0.00
ATOM	71	N	ARG	15	-9.152	1.621	21.047	1.00	0.00
ATOM	72	CA	ARG	15	-10.245	1.387	21.984	1.00	0.00
ATOM	73	C	ARG	15	-11.326	0.514	21.362	1.00	0.00
ATOM	74	O	ARG	15	-12.018	-0.227	22.061	1.00	0.00
ATOM	75	CB	ARG	15	-10.833	2.709	22.454	1.00	0.00
ATOM	76	N	GLU	16	-11.468	0.605	20.044	1.00	0.00
ATOM	77	CA	GLU	16	-12.479	-0.164	19.327	1.00	0.00
ATOM	78	C	GLU	16	-12.101	-1.638	19.256	1.00	0.00
ATOM	79	O	GLU	16	-12.968	-2.507	19.163	1.00	0.00
ATOM	80	CB	GLU	16	-12.682	0.402	17.930	1.00	0.00
ATOM	81	N	TRP	17	-10.802	-1.914	19.299	1.00	0.00
ATOM	82	CA	TRP	17	-10.305	-3.280	19.192	1.00	0.00
ATOM	83	C	TRP	17	-10.977	-4.192	20.210	1.00	0.00
ATOM	84	O	TRP	17	-11.401	-5.300	19.882	1.00	0.00
ATOM	85	CB	TRP	17	-8.795	-3.310	19.366	1.00	0.00
ATOM	86	N	LYS	18	-11.071	-3.720	21.449	1.00	0.00
ATOM	87	CA	LYS	18	-11.703	-4.486	22.515	1.00	0.00
ATOM	88	C	LYS	18	-13.158	-4.795	22.184	1.00	0.00
ATOM	89	O	LYS	18	-13.675	-5.853	22.542	1.00	0.00
ATOM	90	CB	LYS	18	-11.606	-3.738	23.836	1.00	0.00
ATOM	91	N	ALA	19	-13.813	-3.865	21.498	1.00	0.00
ATOM	92	CA	ALA	19	-15.197	-4.054	21.080	1.00	0.00
ATOM	93	C	ALA	19	-15.323	-5.210	20.096	1.00	0.00
ATOM	94	O	ALA	19	-16.298	-5.960	20.125	1.00	0.00
ATOM	95	CB	ALA	19	-15.746	-2.773	20.470	1.00	0.00
ATOM	96	N	MET	20	-14.330	-5.348	19.224	1.00	0.00
ATOM	97	CA	MET	20	-14.308	-6.436	18.254	1.00	0.00
ATOM	98	C	MET	20	-14.388	-7.792	18.945	1.00	0.00
ATOM	99	O	MET	20	-15.203	-8.638	18.580	1.00	0.00
ATOM	100	CB	MET	20	-13.059	-6.352	17.390	1.00	0.00
ATOM	101	N	GLU	21	-13.536	-7.991	19.945	1.00	0.00
ATOM	102	CA	GLU	21	-13.514	-9.242	20.694	1.00	0.00
ATOM	103	C	GLU	21	-14.851	-9.500	21.375	1.00	0.00