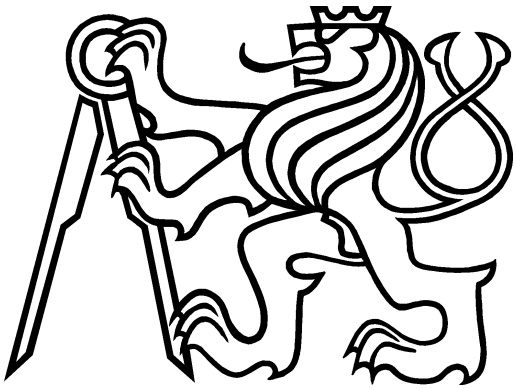
CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING
DEPARTMENT OF CYBERNETICS



Path planning in protein structures

Bachelor Thesis

Tom Jankovec

Thesis supervisor
Ing. Vojtěch Vonásek

Prague, May 2016

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Tom J a n k o v e c

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Path Planning in Protein Structures

Guidelines:

1. Study the problem of path planning for 3D objects [3], study sampling-based path planning methods like Rapidly Exploring Random Tree (RRT) [4], and Probabilistic Roadmaps (PRM) [5]. Implement both methods for the purpose of path planning of general 3D objects.
2. Get familiar with tunnel detection and path planning in protein structures [1,2,6].
3. Study methods for fast collision detection suitable for path planning in protein structures.
4. Adapt RRT or PRM for path planning in static protein structures. Select and implement another RRT-based or PRM-based method suitable for protein structures.
5. Compare the implemented methods on a dataset provided by advisor/oponent.

Bibliography/Sources:

[1] Cortés, Juan, et al. "A path planning approach for computing large-amplitude motions of flexible molecules."Bioinformatics 21.suppl 1 (2005): i116-i125.
[2] Chovancova, Eva, et al. "CAVER 3.0: a tool for the analysis of transport pathways in dynamic protein structures." (2012): e1002708.
[3] LaValle, Steven M - Planning algorithms - Cambridge university press, 2006.
[4] LaValle, Steven M., and James J. Kuffner Jr. - Rapidly-exploring random trees: Progress and prospects - (2000).
[5] Kavraki, Lydia E., et al. - Probabilistic roadmaps for path planning in high-dimensional configuration spaces - Robotics and Automation, IEEE Transactions on 12.4 (1996): 566-580.
[6] Introduction to Protein Structure, Carl Branden, John Tooze, Garland Science; 2 edition (January 3, 1999).

Bachelor Project Supervisor: Ing. Vojtěch Vonásek

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, November 5, 2015

Author statement for undergraduate thesis
I declare that the present work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date ....
signature

Acknowledgments
I'd like to express my gratitude to my bachelor thesis supervisor, Ing. Vojtěch Vonásek, for his endless support and enthusiasm and for dedicating such vast amounts of time to aiding me in the development of this thesis. I'd also like to thank my family for supporting me throughout the years and for never giving up on me. And last but not least, I'd like to thank my beloved orca for bearing with me through these arduous times, kedlubno.

Abstrakt
Problematika detekce dutin a tunelů v proteinových molekulových strukturách je intenzivně zkoumaným tématem jak z biochemického tak z informatického hlediska, s širokým spektrem aplikací, kterými jsou například návrh léků či proteinové inženýrství. Existuje řada softwarových nástrojů (příkladem je *fpocket* [31] a *CAVER* [14]), které umožňují detekci a analýzů proteinových tunelů. Cílem této bakalářské práce je vytvoření nové alternativní metody pro detekci proteinových tunelů, za využití algoritmů pocházejících z oblasti robotiky (například algoritmu Rapidly-exploring Random Trees). Nově navržená metoda využívá konceptu *zakázaných oblastí* pro ovlivnění způsobu růstu stromu, spolu s řadou dalších modifikací, které umožňují najít proteinové tunely v molekulách.

Abstract
The problem of protein cavity and tunnel detection in is a widely studied topic both from the biochemical and the computational standpoint, with a broad spectrum of applications, such as pharmaceutical design and protein engineering. A number of software tools currently exists, such as the *fpocket* [31] and *CAVER* [14], which allow for protein tunnel detection and analysis. The goal of this thesis is to propose a new alternative method for protein tunnel detection, which utilizes well established and analyzed algorithms originating in the field of robotics, namely the Rapidly-exploring Random Trees algorithm. The novel method utilizes a novel concept of *disabled areas* to guide the tree growth, plus a score of other modifications to allow the algorithm to find multiple protein tunnels.

Keywords
protein tunnel detection, path planning, computational geometry.

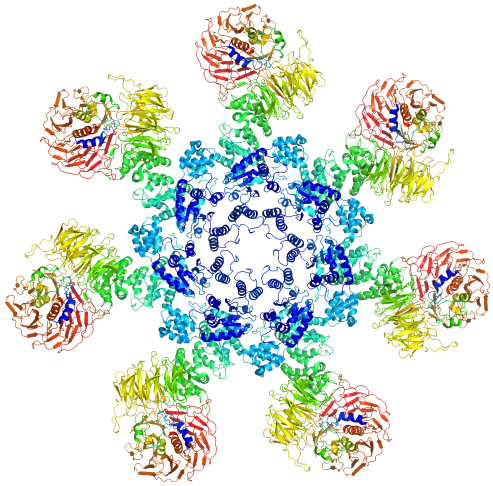
Contents

List of Algorithms

List of symbols

Symbol	Meaning	Name in the program
C	configuration space (the set of all possible states)	-
q	a random state $q \in C$	-
α_t	the largest tunnel width after which a tunnel is no longer considered to be a tunnel [Å]	*MAX_TUNNEL_WIDTH*
δ_t	a threshold value which denotes a differing protein tunnel [-]	*TUNNEL_DIFFERENCE_THRESHOLD*
γ	maximum tree growth speed [Å]	*MAX_TREE_GROWTH*
ϵ	interpolation resolution [Å]	*INTERPOLATION_RESOLUTION*
σ_o	sampling region offset [Å]	*SAMPLING_REGION_OFFSET*
τ_o	disabled area overlap [Å]	*DISABLED_AREA_OVERLAP*
ι	the number of optimizing iterations	*OPTIMIZING_ITERATIONS*
v	sampling probe radius [Å]	-

Table 1: A list of symbols used throughout this thesis

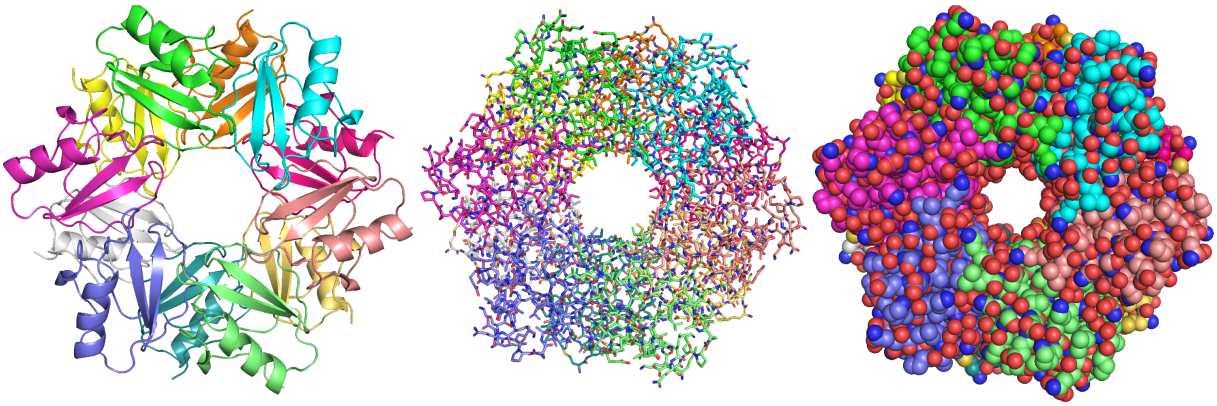


1 Introduction

This chapter aims to give the reader an insight into protein tunnel detection and the currently available state-of-the-art software tools.

1.1 Motivation

Proteins are large macromolecules consisting of long atomic chains (amino acids). The size of proteins in the human body can vary anywhere from 48 atoms (Thyrotropin-releasing hormone or TRH, which would be considered a peptide rather than a protein) [3] to more than 500000 atoms (Titin, the largest known protein) [4]. The individual atoms in the chains and their interactions during protein folding, which occurs after the molecule is synthesized, give rise to their molecular structure. See *Figure 1.1* for an example of protein molecular structure and its different representations.



a) b) c)

Figure 1.1: An example of visual representations of a protein molecule. Figure a) *is a cartoon representation highlighting secondary structures in the protein, figure* b) *is a stick representation highlighting chemical bonds between the atoms and figure* c) *is a sphere representation highlighting individual atoms. Due to its nature, figure* c) *is the most useful in our application*

This structure can reach varying degrees of structural complexity, depending on the size of the protein, and it is what allows proteins to perform their various important biological or industrial functions, primarily through so called molecular *tunnels* and

pores. Tunnels are empty spaces in the molecule, spanning from its surface to varying depths. Pores are tunnels connecting two locations on the molecular surface through the molecule (see *Figure 1.2* for depictions).

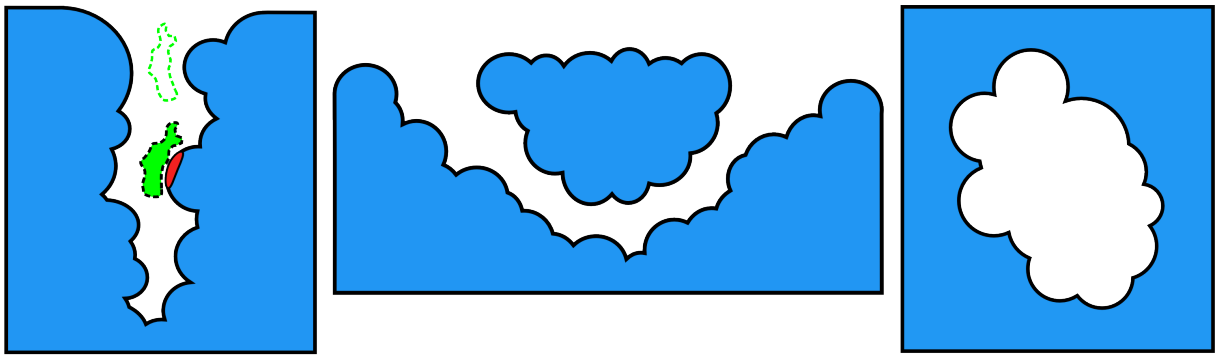


Figure 1.2: A diagram illustrating a protein (blue) tunnel a) *with a "solvent/ligand (green) accessible site (red)" or an "active site", a protein pore* b) *and a protein cavity* c)

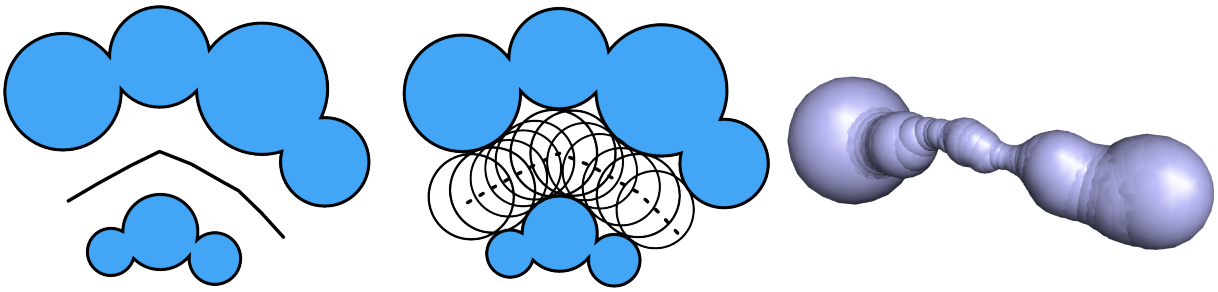
Proteins function by interacting with other, smaller molecules (ligands). After binding them into their structure at so called chemically *active sites* (or solvent/ligant accessible sites) (see *Figure 1.2*). Since these active sites are often located inside protein cavities, studying properties of these structures, especially the previously mentioned tunnels and pores, can give rise to new insights regarding functions of these proteins. For example, knowing whether a tunnel is large enough to accommodate a passage of a ligand molecule can help biochemists to determine whether the two molecules can react and bind together.

Protein tunnel detection and visualization has therefore been an important and active research field. Its applications can be found mainly in the area of pharmaceutical industry, such as structure-based drug design and protein engineering. Computer simulations and analyses of protein structure and their interactions with ligand molecules have recently allowed experts in the field of biochemistry to increase the rate of their research, while being significantly cheaper than laboratory tests.

1.2 Problem description

The main task of protein tunnel detection is to find (compute) a collision-free path for a ligand molecule, leading from an active site located inside the protein molecule to its surface. Protein tunnels can be represented as a sequence of spheres, with their radii equal to the largest non-colliding sphere able to fit in the center of the tunnel, as depicted in *Figure 1.3*.

Let us define the term protein *tunnel width* in a location $q \in C$ as a radius of the largest possible sphere which does not intersect any of the atoms of the protein with its center positioned in q. Similarly, let us define the term *tunnel bottleneck* as the radius of the tunnel's smallest sphere. See *Figure 1.4* for illustrations.



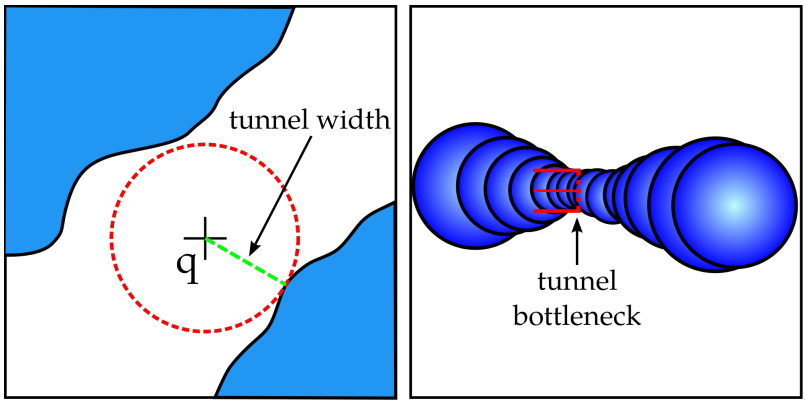
a) b) c)

Figure 1.3: A diagram illustrating tunnel visualization using spheres. a) *shows a found path in a protein tunnel. The path is interpolated into a set of points and every point is used as an origin for a largest possible non-colliding sphere, as depicted in* b). c) *shows the final result in the molecular graphics system* Pymol *[10]*

Among the tunnels' most important characteristics for the field of biochemistry are the size of the tunnel's bottleneck (the narrowest place in the tunnel) and the neighboring amino acids.

It is worth noting that finding the exact center points of the tunnel, i.e., the positions in which the tunnel would be able to accommodate a sequence of the largest possible spheres, is a non-trivial task (since we have to find a global minimum among a set of local minima) and is further discussed in *Section 3*.

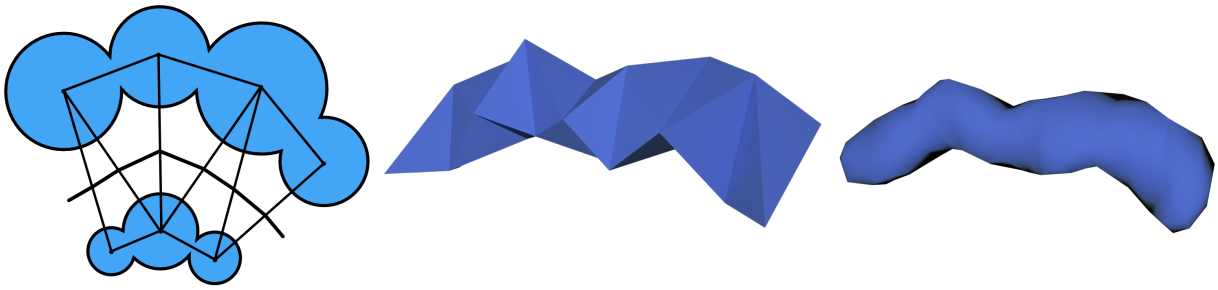
The spherical approach allows us to determine the size of a tunnel's bottleneck more easily, while being less suitable in the process of mesh generation and tunnel rendering [22].



a) b)

Figure 1.4: A diagram illustrating the term tunnel width *and* tunnel bottleneck*. Figure* a) *shows the interpretation of tunnel width at a location q as the radius (green) of the largest possible sphere able to fit in this location.* b) *shows the narrowest place in the tunnel, i.e., the tunnel's bottleneck.*

Another approach employed by tools utilizing Delaunay triangulation (see *Section 1.3.2*) consists of generating a mesh from the resulting 3D triangles, which is then smoothed using various techniques, as is depicted in *Figure 1.5*.



a) b) c)

Figure 1.5: A diagram illustrating tunnel visualization using Delaunay triangulation. Figure a) *shows a triangulated space inside the protein channel, which is later analyzed and extracted* b). *Additional smoothing provides us with the final tunnel shape* c).

Given the possible complexities of the found tunnels, the criteria telling us which solution we should utilize are unclear. Both approaches can suffer from a lack of precision, be it an offset from the tunnel center or smoothing which results in either a shape colliding with the protein's atoms or a shape too small and therefore leaving too much free space between the atoms and the sides of the tunnel.

A number of challenges arises when trying to solve the tunnel detection problem.

- The program has to solve this task in a feasible time framework (minutes/hours) even on average CPUs. Therefore, a trade-off between optimality (finding all existing tunnels), precision (finding tunnels with small or zero spatial tolerance) and time complexity has to reached.
- Since the protein and ligand molecules exist in a 3 dimensional Euclidean space, their configuration space C (the set of all possible states, in which the two objects can exist) is infinite. An approximation representing an incorporation of a trade-off between memory complexity and computational speed has to be created.
- We have to consider the size of the computer's, sadly finite, memory, while bearing in mind the usefulness of the chosen software structure. A less memory-intensive structure might be able to hold a much higher number of states, while being nearly useless in our task, because of its large access times. Similarly a structure, which would allow us to quickly find a solution, might not be able to hold a necessary amount of states in the computer's memory to actually solve the task.
- We need to only choose the relevant states, in which the objects we are studying can physically exist. Therefore, it is desirable for the sampling distributions to contain as few states, which denote colliding molecules, as possible.

1.2.1 Goals and outline of this thesis

The goal of this thesis is to investigate the feasibility of protein tunnel detection utilizing sampling-based motion planning methods and to propose a solution consisting

of a computer program. The program will, on an input of a protein molecule and the respective parameters (such as the probe radius), output a set of all found protein tunnels, as depicted in *Figure 1.6*. The parameters include 3D positions inside the protein (representing tunnel origins), and a probe (ligand) radius, representing the minimum tunnel width.

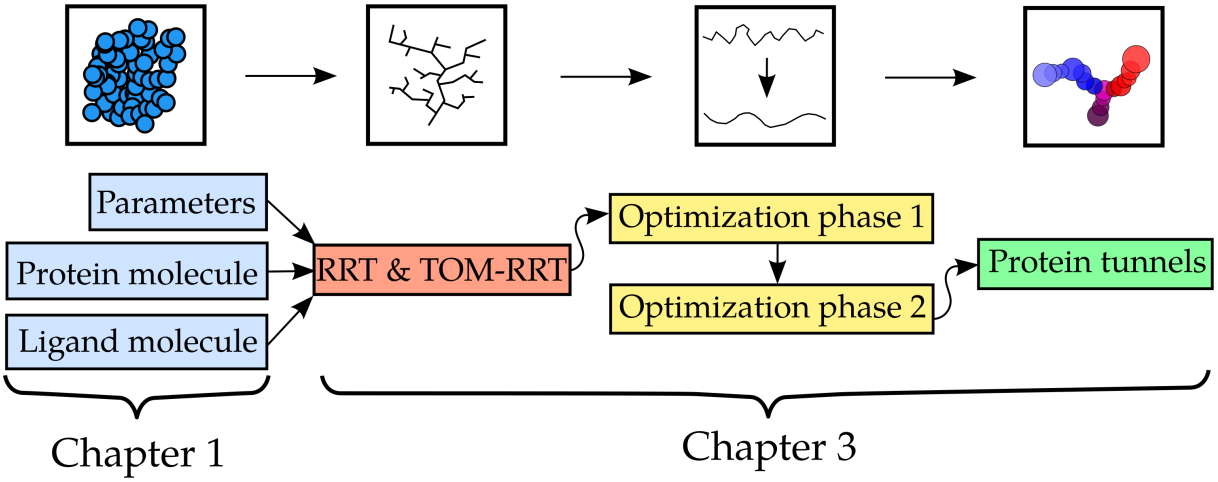


Figure 1.6: A diagram illustrating the organization of this thesis when related to the program's functionality.

Our task is concerned only with protein tunnel detection. The following biochemical analysis of the found tunnels is beyond the scope of this thesis, which mainly explores the computational and geometrical aspects.

In this work we utilize a simplification consisting of representing the ligand molecule as a spherical probe. The possibility of planning for an unsimplified ligand molecule is presented in *Chapter 4*.

The main advantages of this approach are the possibility of planning a path (and therefore detecting a protein tunnel) for a complex ligand molecule (as opposed to the spherical approximation utilized by the current solutions), the possibility to plan in a dynamic protein environment and lower memory requirements when computing [11].

Chapter 1 aims to give the reader an insight into protein tunnel detection and to describe some of the state-of-the-art tools. Some of the available solutions in the field of robotics and their applicability are described and discussed in *Chapter 2*. *Chapter 3* explains the modifications required to create the final proposed solution able to compute multiple tunnels in a single protein molecule for a spherical-based ligand approximation. Further modifications and a solution capable of computing a path through a protein molecule for a non-spherical ligand molecule is described in *Chapter 4*. *Chapter 5* presents and explains the achieved results of the final solution. *Chapter 6* discusses the results of the presented solution and proposes further research topics and improvements.

1.3 Related work

In the last decade, a number of tools aiming to provide protein tunnel analysis has become available [8]. They utilize a variety of methods, each of which has its own up and down-sides. Some of these methods are described in the following sections.

1.3.1 Grid-based tools

Algorithms of this type utilize the idea of approximating a protein molecule as a regularly distributed grid in a three dimensional space. The tunnels and solvent-accessible spaces (see *Figure 1.2*) are detected using well known graph-traversing algorithms, such as the Dijkstra's algorithm [16]. By representing protein atoms as spheres with radii equal to the atoms' Van der Waals radii, the grid-based method is able to determine which cells (representations of regions of space) are occupied and which are not, and fill the grid with nodes of the corresponding type. This grid is then used for tunnel detection. See *Figure 1.7* for illustrations.

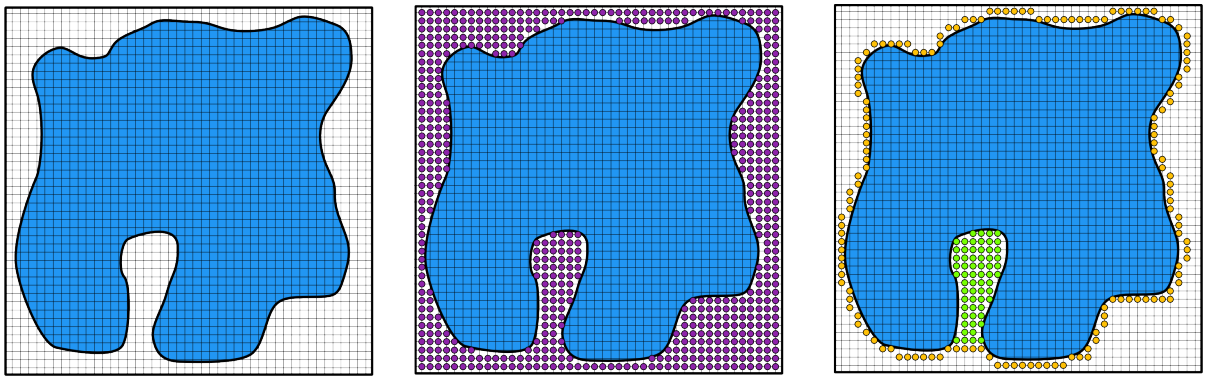


Figure 1.7: A diagram illustrating an approximation of a molecule using a grid-based solution. Figure a) *shows a protein molecule positioned in a discrete grid-like environment. The molecule is analyzed for a list of unoccupied nodes (purple dots in* b)*).* c) *shows the final result, with the molecule's* alpha shape *(see* Section 2.1*) colored in orange and a protein tunnel in green.*

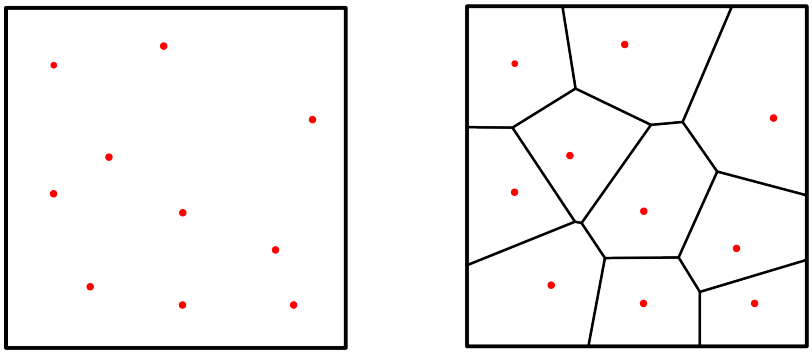
Despite being exhaustive (the finiteness of the grid guarantees that all eligible cells will be searched, compared to the infinite number of states in a Euclidean space utilized by sampling-based tools), this method has several disadvantages. Depending on the resolution of the grid (i.e., too large cells) and the position, or rather the rotation of the protein tunnel (i.e., the tunnel is positioned is such a way that the approximating algorithm considers the space occupied), the tool can fail to find available channels where other methods succeed [33]. The fact that the entire grid has to be sampled in order for the approach to work also poses another disadvantage — the necessity to hold the entire data structure in the computer's memory.

This grid-based approach is used in e.g. *POCKET* [24] and its further improvements, *LIGSITE* [18], *dxTuber* [27], *HOLLOW* [21], 3V [28] and other software tools [19].

The main advantages of tools of this type are the exhaustiveness of the algorithms, since the voxel grid is finite and the simplicity of implementing this approach. The main disadvantages are the lower probability of finding protein tunnels, because of the inherent imprecision, and the high memory requirements.

1.3.2 Tools based on Voronoi diagrams

Let us have a set of points in an n-plane. If we partition this n-plane into convex shapes, so that every shape contains precisely one original point and all points in the n-plane contained by the shape's boundary are closer to the original generating point than any of the other original points, we arrive at a *Voronoi Diagram* [36]. See *Figure 1.8* for illustrations.



a) b)

Figure 1.8: An example of a Voronoi diagram. Figure a) *shows a number of points in an empty plane.* b) *shows the Voronoi diagram of said points.*

A weighted Voronoi diagram is a special case of a regular Voronoi diagram, in which the distance function is defined using a specific set of parameters (see *Figure 1.9*). It is particularly useful in this scenario, since it allows us to take into account different atomic radii. "The Delaunay triangulation and Voronoi diagram in $\mathbb{R}^2$ are dual to each other" [34].

Generalized into three dimensions, it can be used in protein cavity detection. Protein molecules are triangulated using Delaunay tetrahedra and protein shapes are derived by removing empty regions. These shapes can later be analyzed and used to detect protein channels.

While this approach is more precise that the one previously described in *Section 1.3.1*, it also has its downsides. The ligand molecule can only be represented as a sphere. Since the spheres representing atoms have different radii (much like the atomic Van der Waals radii), a standard Voronoi diagram is inherently imprecise — the cell boundaries are used to represent equidistant points between the centers of two atoms and this doesn't apply when the atomic sizes don't match. See *Figure 1.10* for illustrations.

Some tools overcome this problem by using an approximation of a weighted Voronoi diagram (or Delaunay triangulation). For example, larger spheres can be represented by several small spheres, which allows for the usage of a standard Voronoi diagram [14,

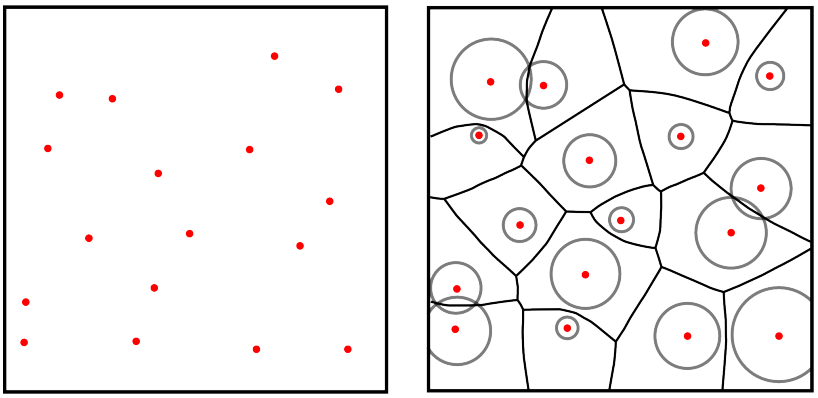


Figure 1.9: An example of a weighted Voronoi diagram. Figure a) *shows a number of points in an empty plane.* b) *shows a Weighted Voronoi diagram of said points with the points' weights shown as gray circles. The sizes of radii of the gray circles represent the values of weights of the individual points. Image courtesy of [5]*

26]. While this does indeed simulate the behavior of a weighted Voronoi diagram, it increases the time and memory complexity of the task and prolongs the calculation. It also introduces an inaccuracy into the value of a protein tunnel's bottleneck size, since the resulting cell boundaries are not located in the precise center points of the tunnel.

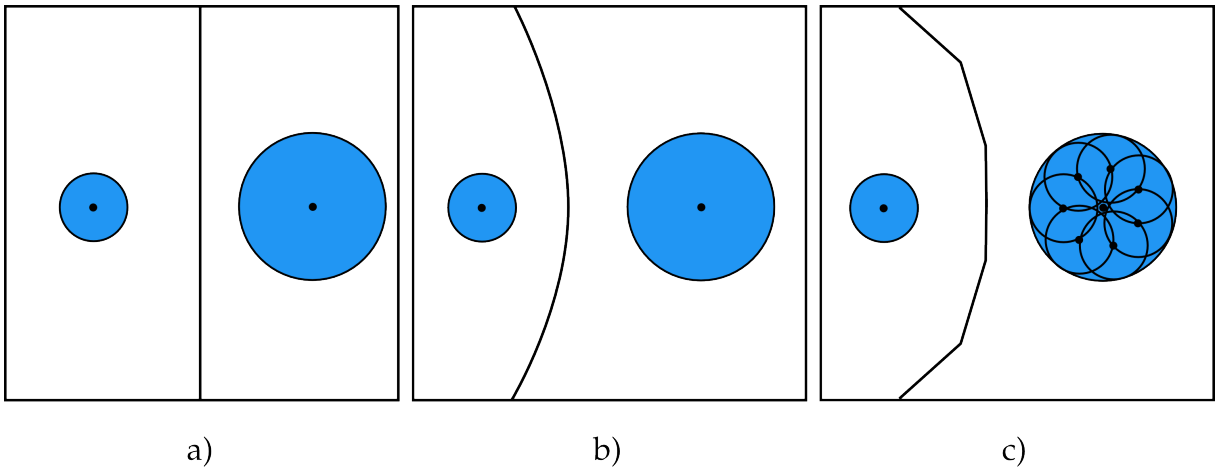


Figure 1.10: A diagram illustrating inaccuracy stemming from the usage of a standard Voronoi diagram. Figure a) *is a result of a standard Voronoi diagram and is clearly less than optimal; the nerve of the cell is not positioned equidistant from both atomic surfaces.* b) *is a result of a weighted Voronoi diagram, and is the optimal solution.* c) *illustrates an approximation employed by some of the solutions ([14], [26]), where the larger sphere is represented using a number of smaller spheres.*

Some of the tools utilizing Voronoi diagrams (or Delaunay triangulation) are e.g. *Mole* [26], *MolAxis* [13], *CAVER* [14] and others.

The main advantages of this type of tools are smaller memory requirements compared to grid based tools and the exhaustiveness of the programs, since the cell structure is created from a finite number of atoms. The main disadvantages are the inherent

imprecision when utilizing a weighted Voronoi diagram approximation and the necessity to use a spherical ligand molecule approximation.

1.3.3 Sampling-based motion planning tools

In recent years a number of tools, utilizing sampling-based algorithms developed in the field of robotics for protein tunnel detection, has emerged. These tools formulate the task of protein tunnel detection as a motion planning problem, where the goal is to find a feasible path between two given positions. The motion planning approach allows us to plan a path for an arbitrary shape (compared to the spherical approximation utilized by the other types of tools), to utilize a flexible ligand molecule and to plan in a time-variable environment. One of the most prominent being the French *MoMA-LigPath* web server [11].

It utilizes the Manhattan-like RRT algorithm (ML-RRT) to model protein molecule conformations. ML-RRT divides all variables into two groups, passive and active. Active variables denote parameters which are crucial for the motion planning task, whereas passive variables are only taken into account when necessary (e.g. when they hinder the task). The planner then proceeds considering the two groups as separate [9].

The nature of sampling-based motion planning algorithms conveys a number of computational challenges, most importantly the notion of an infinite configuration space. Therefore, if a solution doesn't exist, such algorithm may run forever, unless terminated by the user or a time-based constraint. However, a majority of said algorithms also allows for low memory complexity, compared to e.g. grid-based tools, since the whole configuration space doesn't have to be discretized. This topic is discussed in detail in *Chapter 2* and *Chapter 3*.

2 Path planning

This chapter aims to describe the available solutions in the field of robotics and to elucidate the integration of protein tunnel detection.

2.1 Protein tunnel detection and path planning from a robotics' standpoint

The task of path planning in the field of robotics has been an area of intensive research for many decades, with rapid progress in both the task-solving speed and the complexity of the tasks. Robots with many degrees of freedom, which would have previously required unfeasibly long computing times to plan even simple motions, can nowadays be dealt with in near real-time speed in both static and dynamic environments [20].

Assuming the paradigm of robotics, the problem of protein tunnel detection can be described rather similarly to a wide range of easily solvable problems in the field of path planning for robots. The protein molecule can be approximated as a static 3D environment consisting of differently sized spheres (according to the atomic Van der Waals radii). Similar representation is utilized for the ligand molecule. Since the ligand molecule is capable of translation and rotation in 3D space, it has 6 degrees of freedom. When given the protein's active sites (or the coordinates symbolizing the tunnel's origin point), the problem consists of finding a collision-free trajectory from an initial point (the active site) to an unspecified location outside of the environment (the protein molecule). If such a trajectory exists, we have found a tunnel, which we know the ligand molecule can pass through. Given these conditions, the already well established algorithms can, with some modifications, even be used for multiple tunnel detection (this topic is discussed later).

The assumption that the protein molecule is a static 3D environment does not, of course, apply in real cases. The dynamic nature of the positions of the protein molecule's atoms, caused by heat motions, means that the tunnel characteristics, namely the sizes of the tunnels' bottlenecks, are dynamic as well.

A configuration of a static 3D object in a 3D Euclidean space can be uniquely identified by its state (i.e., its position $P \in \mathbb{R}^3$ and its rotation $R \in SO(3)$, denoting the 3D rotation group). Given the nature of the $SO(3)$ rotation group, we can utilize many existing mathematical constructs to describe it. For the sake of simplicity and an ease of

use, we have utilized quaternions ($R = (a, b, c, d)$), which are hyper-complex numbers satisfying certain conditions [35].

Let us define a configuration space C as a space of all available states (object positions and rotations: $q \in (x, y, z, a, b, c, d)$). Let C_{free} be a subspace of all non-colliding states and let $Protein \subset \mathbb{R}^3$ represent a region of space occupied by the protein molecule:

$$C_{free} = \{q \in C | S(q) \cap Protein = \emptyset\},$$

where $S(q)$ represents a sphere with a chosen probe radius (v) in a position denoted by the vector q. Let us define the goal region

$$C_{goal} \subseteq C = \{q \in C | S(q) \cap Protein_{\alpha}\},$$

where $Protein_{\alpha} \subset \mathbb{R}^3$ represents a region of space bounded by an outer layer of the protein's alpha shape (see *Section 2.1*). It follows that C_{goal} is a subspace of C_{free}. See *Figure 2.1* for illustrations.

Our task therefore consists of finding a sequence of states s_i (a path T) for a defined object in its configuration space C between the initial $s_{init} \in C_{free}$ and the final $s_{final} \in C_{goal}$ states, so that the object doesn't collide with the environment (the trajectory $T \subset C_{free}$) for any of the found states or for any trajectory segments represented by two subsequent states $s_i \rightarrow s_{i+1}$ (and all states belonging in a subspace of C created by interpolating states s_i and s_{i+1}). We say that a state is free if $s \in C_{free}$, otherwise we say that the state is colliding. The path T can therefore be presented as:

$$s_i = (P, R) = ((x, y, z), (a, b, c, d))$$

$$T = \left(s_{init}, s_2, ..., s_{n-1}, s_{final}\right) : \left(s_i \in C_{free}; colSeg(s_i, s_{i+1}) = false; i = 1, 2, ..., n-1\right),$$

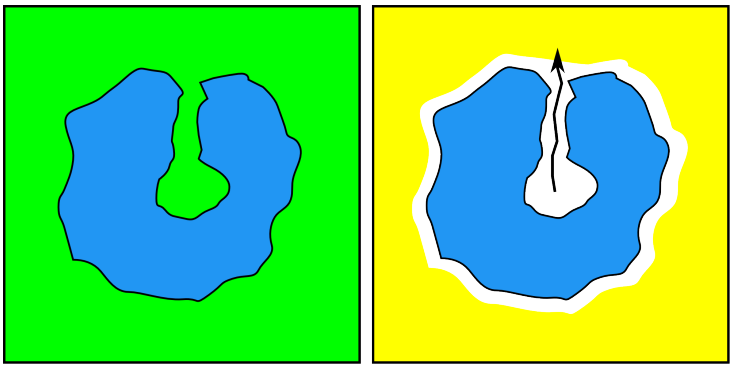
where n is the number of states in the final trajectory and $colSeg(s_i, s_{i+1})$ is a function detecting a collision between an object and the environment for the subset of C created by interpolating states s_i and s_{i+1} returning either *true* or $false$.

The distance between two configurations can be measured as:

$$\begin{aligned} d(s_1, s_2) = &\sqrt{(s_{1x} - s_{2x})^2 + (s_{1y} - s_{2y})^2 + (s_{1z} - s_{2z})^2} + \\ &+ 10 \cdot \sqrt{(s_{1a} - s_{2a})^2 + (s_{1b} - s_{2b})^2 + (s_{1c} - s_{2c})^2 + (s_{1d} - s_{2d})^2} \end{aligned}$$

where s_1 and s_2 are two states. Note that the indices x, y, z correspond to the object's position and a, b, c, d to the object's rotation.

The described configuration space is multidimensional and infinite. As is the case with the majority of computers used today, it is usually impossible to represent using a grid-based approach due to memory requirements. Random sampling-based algorithms therefore present excellent candidates for this type of task.



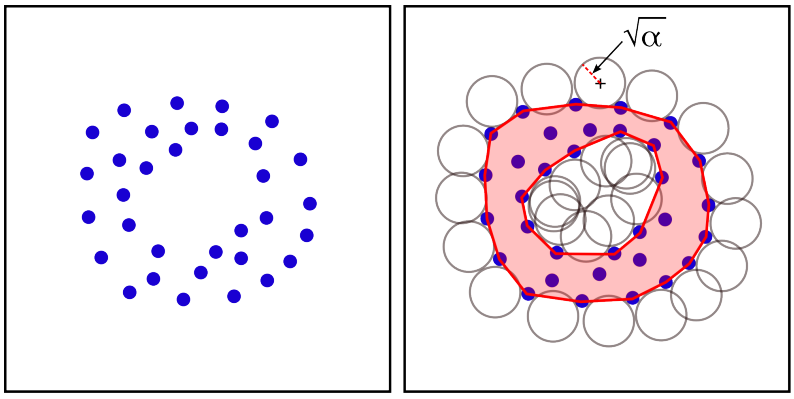
a) b)

Figure 2.1: An illustration of a free subspace and a goal region. Figure a) *displays a protein molecule (blue) and the free subspace C_{free} (green).* b) *displays a protein molecule (blue) and its goal region (yellow) C_{goal} as well as an example of a path from the inside of a molecule to the goal region.*

Surface detection

An important part of protein tunnel detection is the computation of a protein surface. This can be achieved using the concept of α-shapes.

Let us have a set of points in a 2 or 3-dimensional space and a parameter, α (which is the origin of the name, please note that this parameter is not identical to the utilized symbol α_t, denoting a particular tunnel radius value). If we position spheres in all the regions we can "reach" using a sphere with a radius equal to the square root of the mentioned α parameter, so that none of the said points lie in the these spheres and every sphere touches at least one point, and if we connect said touched points using a generated mesh, we have arrived at the structure's alpha shape (see *Figure 2.2*). If $\alpha = 0$, the shape degenerates to set of the points, if $\alpha \to \infty$, the shape converges to a convex hull of the set of points. Alpha shapes can be used to determine the border of a protein molecule (surface).



a) b)

Figure 2.2: Figure a) *shows a set of points (blue), for which an alpha shape (red) is constructed in* b)

As previously stated, shapes of proteins can reach immense complexities and a

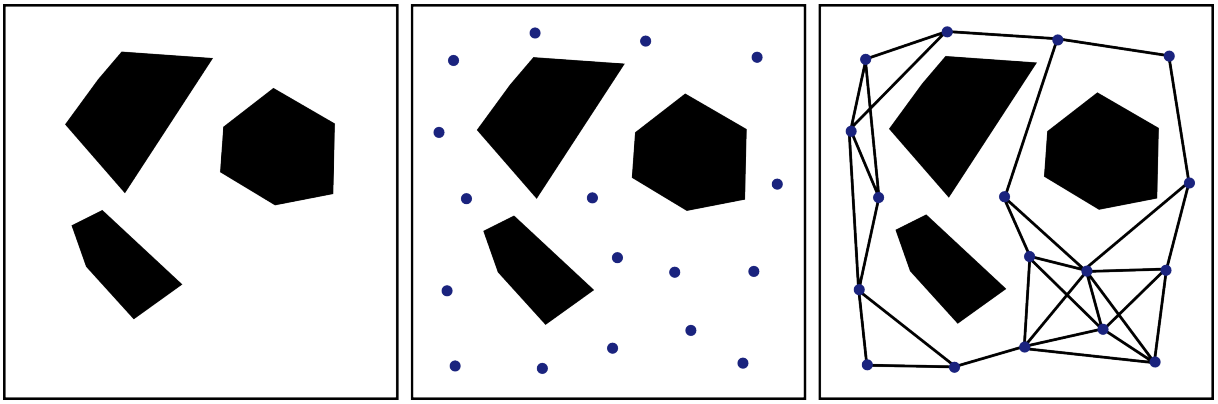
robust solution to provide their borders is necessary for tasks such as duplicate protein tunnel detection (sampling-based algorithms can find multiple tunnels in the same physical tunnel inside the protein molecule, which is undesirable). See *Section 3.3* for further details regarding multiple tunnel detection.

2.2 Sampling-based path planning

As previously mentioned, a wide variety of sampling-based algorithms already exists in the field of robotics. These algorithms randomly sample the provided configuration space to find relevant collision-free states which are then used to build a graph and ultimately find a path through the environment. The functionality of two currently arguably most commonly used algorithms is described below.

2.2.1 Probabilistic Roadmaps

The Probabilistic RoadMap (PRM) [12] algorithm samples the configuration space until a desired (previously specified) number of free states is created. The found free states are then stored in a list. Afterwards a list of n nearest free states is found for each of the previously stored states. The algorithm then tries to connect these states with undirected edges. If a trajectory created by interpolating a state and its neighbor is collision-free, the two states are connected regardless of whether they already are in a graph or not (and can therefore create new and unconnected graph components). These steps are repeated until both the initial and the goal state are in a single connected component of the graph [12]. This can be checked after every iteration with any of the commonly used graph-traversing algorithms, such as the A* algorithm. The resulting graph is displayed is *Figure 2.3*.



a) b) c)

Figure 2.3: A diagram showing the construction process of a PRM graph. A given configuration space with a set of obstacles in a) *is sampled until a set amount of free nodes (blue) is created in* b)*. Connections between these nodes are then created in* c)*.*

The algorithm therefore consists of two phases: the construction phase and the query phase. Let there be a graph consisting of nodes N and undirected edges E, an

initial seeding node $n_{init} \in N$ a function $neighbors(n)$, which returns a list of nearest neighbors for the node n from the list of previously sampled collision-free nodes, a function $collisionFree(n_i, n_j)$, which returns a boolean value if the trajectory between the two nodes is collision-free and a function $goalReached()$, which returns a boolean value if the goal node is reachable from the initial node. The pseudocode is listed in *Algorithm 1*.

```
1  Function buildPRM(n_init)
2      N ← n_init;
3      E ← ∅;
4      while ¬goalReached() do
5          q ←a randomly sampled collision-free node;
6          N ← N ∪ q;
7          N_q ← neighbors(q);
8          foreach n_i ∈ N_q do
9              if collisionFree(n_i, q) ∧ no connection between n_i and q then
10                 E ← E ∪ new graph edge formed between nodes n_i and q;
11             end
12         end
13     end
14 Function collisionFree(s_1, s_2)
15     count ← distance(s_1, s_2)/ε;
16     foreach i ∈ {0, 1, ..., count} do
17         Node n ← interpolateNodes(s_1, s_2, i/count);
18         if n.colliding then
19             return false;
20         end
21     end
22     return true;
```

Algorithm 1: A Probabilistic Roadmap construction phase algorithm

The function $interpolateNodes(n_1, n_2, alpha)$ in *Algorithm 1* interpolates nodes from n_1 to n_2 using a floating-point parameter $alpha \in < 0, 1 >$ to describe the ratio between the two nodes and returns a new node afterwards. If $alpha = 0$, the new node is equivalent to n_1 and if $\alpha = 1$, the new node is equivalent to n_2.

The function $goalReached()$ in *Algorithm 1* can easily be implemented using a graph-traversing algorithm, (e.g. the Dijkstra's algorithm or the A* algorithm). Some authors verify that all connected nodes always lie in a single-connected component before forming new edges, however, our results indicated that, in our case, the slowdown caused by the complexity of such verifications is worse than the slowdown caused by their absence.

Probabilistic roadmaps are known for their ability to quickly and easily traverse high-dimensional configuration spaces [12] and are therefore a suitable candidate for the task of tunnel detection.

2.2.2 Rapidly-exploring Random Trees

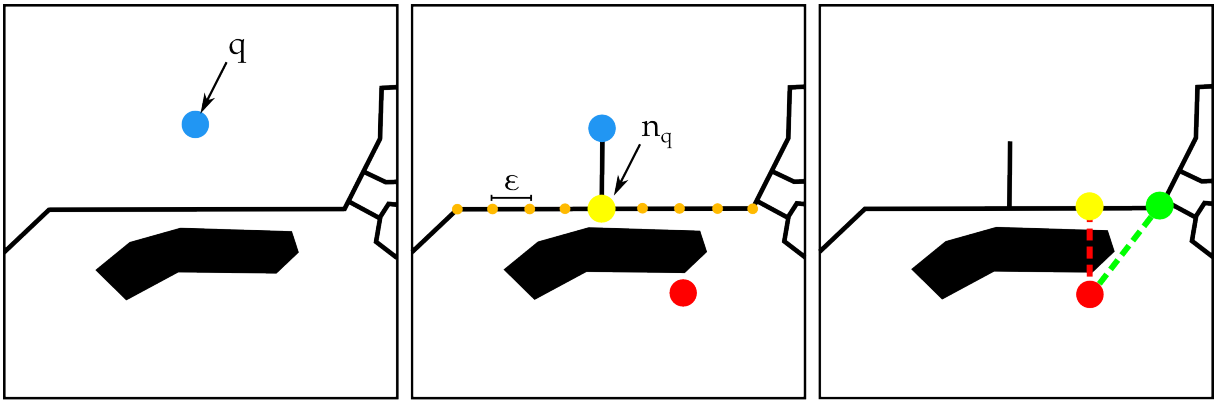


Figure 2.4: A diagram showing the working principle of the basic RRT algorithm. Figure a) *shows a newly sampled state (blue), which can be connected to the tree via a collision-free and shortest possible trajectory in* b). b) *also shows a newly sampled node (red) which can't be connected to the nearest state n_q, because the shortest possible trajectory would collide with an obstacle (red line in* c)). c) *shows one of the main differences between the RRT and other algorithms — a new node can be connected directly to the edge of the tree, whereas some other algorithms would try and connect the node to an existing node (green). ϵ denotes trajectory interpolation resolution (orange) which is discussed in Section 3.4.*

The Rapidly-exploring Random Trees (RRT) [25] algorithm builds a tree of sub-trajectories formed by connecting new sampled free states with previously found and connected states (i.e., with the tree).

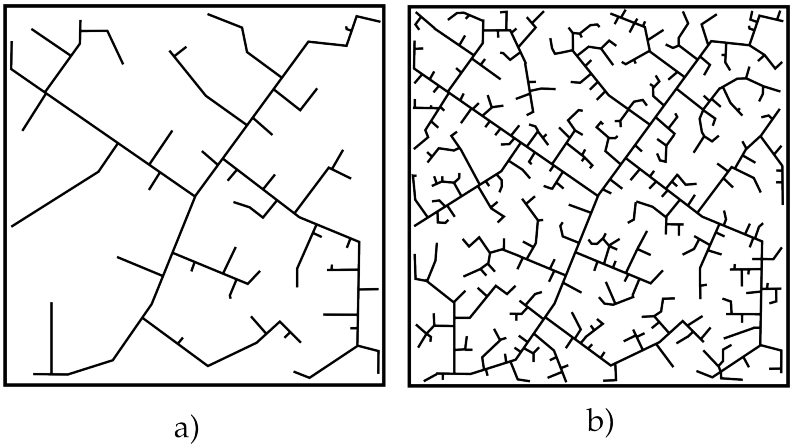


Figure 2.5: An example of an RRT graph. Fig. a) is the result of running the algorithm for 45 iterations, b) is the result after 390 iterations. Image courtesy of [2]

First, an initial seeding point s_{init} of the resulting tree is specified. The algorithm then randomly samples the supplied configuration space. If the created sample q is not free, it is discarded. Otherwise the closest state n_q contained in the tree is found and the algorithm tries to connect the two points by checking whether the trajectory created by

interpolating them is collision-free. If it is, the randomly sampled point is added to the tree and goal reachability is checked (we say that the goal is reachable if the trajectory formed by interpolating the newly added state and the goal state is collision-free). See *Figure 2.4* for illustrations. This process repeats until the goal state is reachable. See *Figure 2.5* for the results of running the algorithm. The nature of the algorithm ensures that the resulting graph is a tree, that is, a graph with no parallel edges or cycles.

Let there be a tree $T(N, E)$ consisting of nodes N and directed edges E, an initial node n_{init}, a function $nearestPoint(n)$ returning either a nearest node or a node lying in an existing edge, an functions $goalReached()$ and $collisionFree(n_i, n_j)$ with implementations similar to those in *Algorithm 1*. The pseudocode is listed in *Algorithm 2*.

```
1  Function buildRRT(n_init)
2      N ← n_init;
3      E ← ∅;
4      while ¬goalReached() do
5          q ←a randomly sampled collision-free node;
6          n_q ← nearestPoint(q);
7          if collisionFree(n_q, q) then
8              N ← q;
9              E ← E ∪ new edge formed from node n_q to node q;
10         end
11     end
```

Algorithm 2: A basic RRT construction algorithm

Rapidly-exploring Random Trees have shown the ability to swiftly traverse very complex environments and are therefore also (alongside PRM, as described in *Section 2.2.1*) a suitable candidate for the task of protein tunnel detection.

2.2.3 RRT-Path

RRT-Path is a type of a guided RRT algorithm. It speeds up by the path-finding process by first finding a simplified version of the final path for a smaller mobile object using the PRM algorithm. The found path is then used to alter the sampling distribution so that samples along the found path are preferred with probability $p \in < 0, 1 >$ [32].

The algorithm has been implemented and tested, however, since the PRM phase failed to find a path in any of the tested protein molecules because of their complexity, the approach had to have been abandoned.

2.2.4 Discussion

The described sampling-based algorithms RRT and PRM (see *Section 2.2.2* and *2.2.1*) utilize the notion of an infinite configuration space C. When presented with an initial $s_{init} \in C_{free}$ and final $s_{final} \in C_{goal}$ location, they strive to find a collision free trajectory between these two locations. As stated previously in *Section 1.3.3*, the utilization of

sampling-based algorithms brings a number of both advantages and disadvantages from the computational standpoint.

The main advantage is the concept of a configuration space approximation. Tools utilizing either a discrete space grid or Delaunay triangulation only have to hold a finite amount of elements in memory (regardless of whether such elements are nodes of a graph or Delaunay tetrahedra). This does, however, also convey a disadvantage, since the finite configuration space has to be kept in the memory in its entirety. Therefore, if we work with large protein molecules consisting of many atoms or we require highly precise results (and therefore use high resolution), these tools can be very memory demanding.

Sampling-based approaches convey significantly lower memory requirements — we don't have to store the entire configuration space in the computer's memory and can work only with the states we consider important. The main disadvantage is the fact that the configuration space is infinite — any two states can be interpolated to infinitely many other states. Consequently, the algorithms can never be exhaustive, unless suitably constrained (using e.g. time or resolution-based constrains).

Another large advantage of sampling-based algorithms is the fact that, unlike with the other groups of tools, ligand molecules don't have to be approximated using a circumscribed sphere. This brings up the possibility of discovering tunnels in some extreme cases, where current tools would fail (e.g. an extremely mono-dimensionally extended ligand molecule). This fact is further discussed in *Chapter 4*.

3 Proposed solution

The task of protein tunnel detection is to find all possible tunnels when presented with a protein molecule. Our goal is to utilize the Rapidly-exploring Random Tree algorithm [25], a sampling-based motion planning algorithm from the field of robotics. We aim to investigate the application of this algorithm, which was chosen because of its ability to quickly find paths for complex objects, while being easy to understand, implement and alter. The original RRT algorithm is not designed for the purpose of multiple path (tunnel) search and it therefore requires modifications in order to be used for the tunnel detection task. These modifications are described in the following sections.

The previous algorithms have been designed for the purpose of planning a path between two explicitly defined locations. This is unsuitable in our situation, since having a single defined goal location (outside the protein molecule) could significantly slow down the speed of the search. This also gives rise to a problem when trying to search for multiple tunnels. Due to its nature, the algorithm only allows us to search for a single path, which is undesirable, since we want to find all available protein tunnels. The algorithm therefore had to be modified to account for a goal region, rather than a location, and to allow a multiple tunnel search.

The modifications are described in the sections below. The resulting implementation is described in detail in *Section 3.4*.

3.1 Outline of the proposed algorithm

The RRT algorithm works by building a tree from the initial location to the final location by trying to connect free samples to the already existing structure (tree). Upon reaching this final location, it simply traverses the tree to the root while storing the states (see *Section 2.2.2* for greater detail).

To be able to detect multiple tunnels in a protein molecule at the level of current state-of-the-art tools we have to address a number of issues:

1. The choice of a proper sampling region which allows us to reach maximum efficiency

2. Tunnel end-point detection, to avoid slowing down the algorithm by searching to a particular point in space

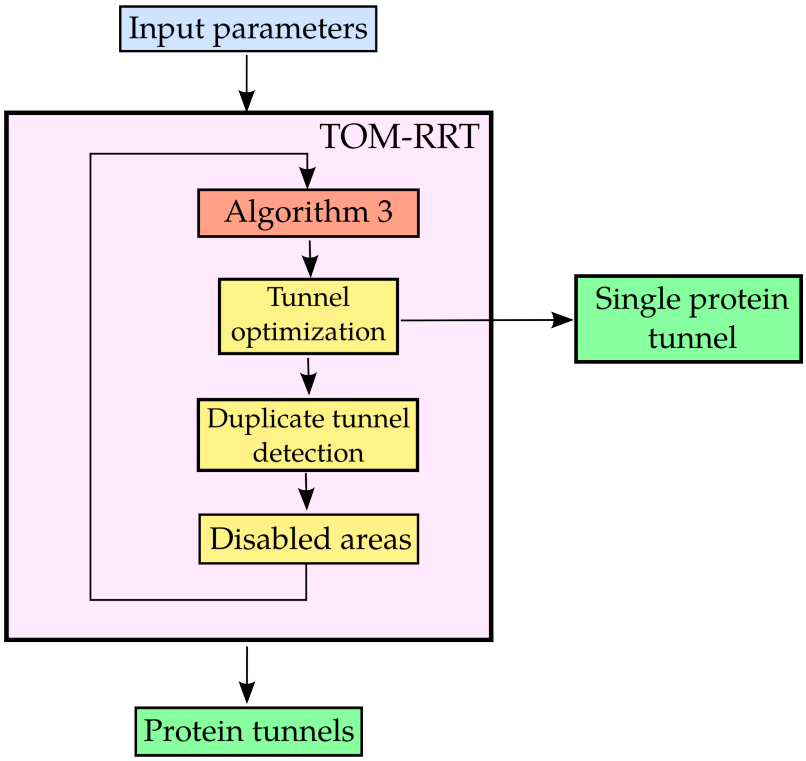


Figure 3.1: A diagram illustrating the functionality of the introduced TOM-RRT algorithm.

3. Restricting the algorithm from analyzing already searched areas (preventing it from finding tunnels which have already been found)

4. Analyzing the found tunnels to detect duplicates

5. Tunnel optimization in order to ensure that the found tunnel's characteristics correspond to those of the physical tunnel

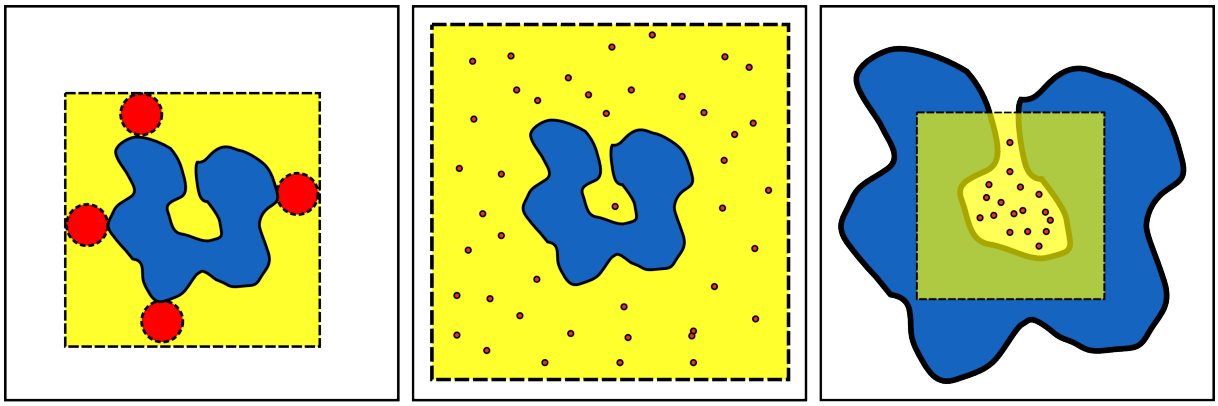
The modifications can therefore be summed into two goals, the first being the ability to find a protein tunnel, and the second the ability to find multiple tunnels. The proposed algorithm Tunnel-rOuting-Magical RRT (TOM-RRT) consists of a modified RRT algorithm running in a loop with an environment-altering routine, establishing the concept of "*disabled areas*" (see *Section 3.3* for more details and *Figure 3.1* for illustrations).

3.2 Single tunnel detection

This section describes the modifications required for the RRT algorithm to find a single tunnel in the protein molecule.

3.2.1 Sampling region

A sampling region is an area of space in which random samples are created. It is a subspace of the configuration space C. To ensure maximum efficiency, the ideal situation for which we strive is to have the sampling region equal to the free configuration space C_{free} and subsequently have the entirety of goal region C_{goal} contained in it.



a) b) c)

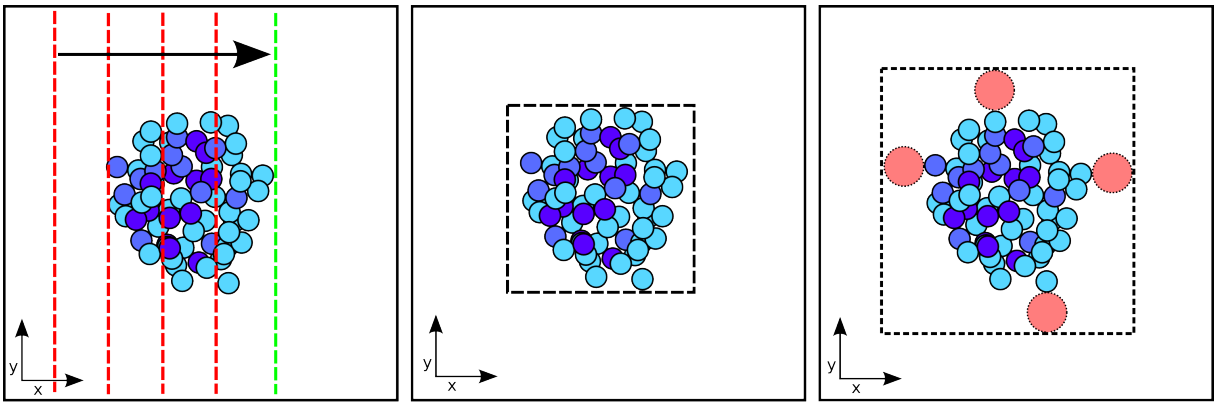
Figure 3.2: A diagram illustrating possible sampling region scenarios. Figure a) *shows an ideal sampling region (yellow), the entire protein molecule (blue) is contained with enough space around it for the ligand molecule (red) to fit in and therefore satisfy the goal condition.* b) *shows a sampling region which is too large, resulting in the majority of samples being positioned outside of the protein molecule and therefore useless for the task at hand.* c) *shows too small sampling region, which will never produce a sample satisfying the goal condition.*

The problem of sampling region designation is crucial for the entire task. If we utilize a sampling region which is too large, the majority of samples may lie in an area where they can't be connected to the initial point because of obstacles lying in the way and they may get discarded. On the other hand, if the sampling region is too small, a sample which would satisfy the goal condition may never be picked, as depicted in *Figure 3.2*.

To address the problem of choosing a sampling region for every protein molecule, a function which analyzes both the protein and the ligand molecule has been developed. It works by scanning the protein and ligand atoms along all three axes, choosing the ones positioned the furthest along each axis and using their coordinates as the basis for the sampling region, along with the size of the ligand molecule and the σ_o parameter in the source code. See *Figure 3.3* for illustrations.

We have noticed a significant performance improvement upon integrating this function, compared to the previous approach, which consisted of determining the sampling region manually. This is likely caused by a more desirable sampling distribution.

When a large amount of samples is generated in an environment as complex as this, a very high percentage of them gets discarded, since they are in collision with the environment. Choosing a distribution function which minimizes this percentage can therefore convey a large speed improvement in the final solution. An attempt was

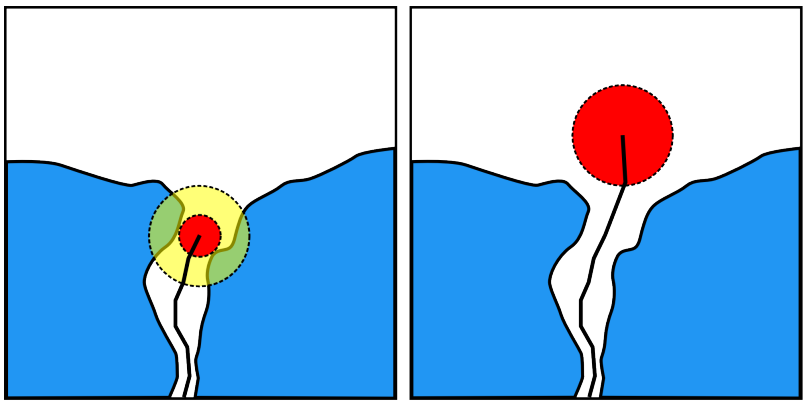


a) b) c)

Figure 3.3: A diagram illustrating sampling region identification. Figure a) *shows how one of the borders is moved (red) until the protein molecule (blue) lies entirely in the half-space designated by it (green).* b) *shows the result of said process and* c) *shows the final region, enlarged by the ligand molecule's size (red) and the σ_o parameter.*

made to generate a potential field from the static environment (the protein molecule's atoms) which would then be used as a distribution function.

A simple uniform distribution function was chosen, as well as a more sophisticated one, consisting of implementing the algorithm RRTPath [32], which is optimized to create samples in free configuration space (see *Section 2.2.3* for more details).



a) b)

Figure 3.4: A diagram illustrating the principle of goal region detection. In figure a) *a newly added node doesn't satisfy the goal condition, as its tunnel width (red) is lower than the value denoted by the parameter (yellow).* b) *shows how a node (red) located outside the protein molecule (blue) satisfies the goal condition.*

3.2.2 Tunnel end-point detection

Both of the algorithms described previously in *Section 2* were modified to avoid having to explicitly pick a goal location. A parameter denoting a maximum protein tunnel width (see *Section 3.3.1*), α_t has to be defined manually to ensure we only consider eligible cavities in the protein molecule as tunnels. This fact, along with the fact that the graph used to plan the ligand molecule's path always has to start in a tunnel, can be utilized.

Every time a new node is connected to the graph, the amount of free space surrounding it is checked. If the radius of a largest possible sphere which can be fit in this node's position is larger than the previously mentioned parameter α_t, the cavity is considered to be outside the protein molecule and the program has therefore succeeded in finding a tunnel. See *Section 3.3.1* for more details regarding the implementation of the tunnel width computing function and *Figure 3.4* for illustrations.

This approach approximates the currently established and computationally expensive protein surface detection using alpha shapes (see *Section 2.1*). The resulting pseudocode of the previous algorithm is described in *Algorithm 3*.

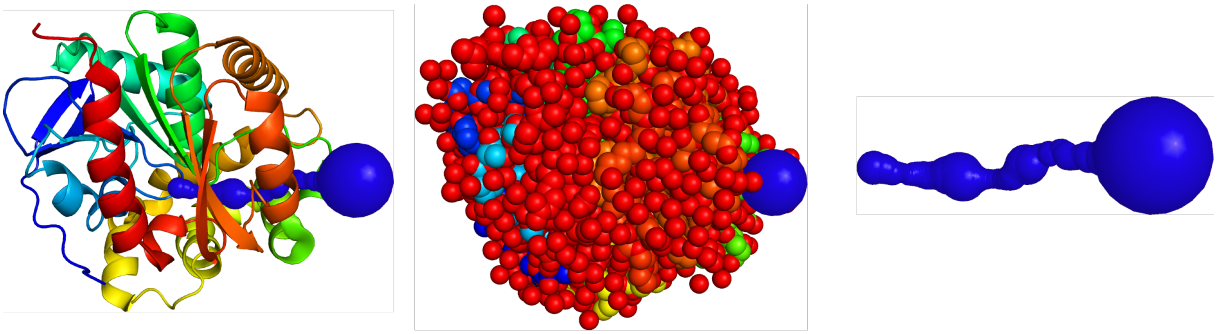
```
1  Function findSingleTunnel(s_init)
2      terminate ← false;
3      while !terminate do
4          terminate ← checkConstraints();
5          q ← createNonCollidingSample();
6          n_q ← findNearestNeighbor(q);
7          collisionFree ← validateTrajectory(q, n_q);
8          if !collisionFree and distance(q, n_q) > γ then
9              q ← moveSample(q, n_q);
10             collisionFree ← validateTrajectory(q, n_q);
11         end
12         if collisionFree then
13             connectNewState(q);
14             if checkGoalCondition(q) then
15                 terminate ← true;
16                 return createTunnelFrom(q);
17             end
18         end
19     end
```

Algorithm 3: Single tunnel detection pseudocode

The variables *terminate* and *collisionFree* in *Algorithm 3* are boolean variables, *checkConstraints*() is a boolean function returning true when time-based termination conditions are fulfilled *createNonCollidingSample*() is a function which returns a free sample in the specified sampling region, q is a randomly sampled state, $findNearestNeighbor(state\ s)$ is a function which returns a state in the tree nearest to the provided state s, n_q is a state nearest to the state q, $validateTrajectory(state\ s,\ state\ t)$

is a boolean function returning true if the trajectory formed between the two states is collision-free, analogous to the function described in *Algorithm 1*,
distance(*state s*, *state t*) is a function which returns a distance between the states s and t computed using the provided distance metric (see *Section 2.1*),
moveSample(*state s*, *state t*) is a procedure which moves the state s to the state t so that the distance between the two states is equal to the γ parameter,
connectNewState(*state s*) is a function which connects the provided sample to the tree and interpolates the formed trajectory, *checkGoalCondition*(*state s*) is a boolean function returning true when the provided sample satisfies the goal condition (see *Section 3.2.2*) and *createPath*(*state s*) is a function which returns a path formed by states in the tree created by traversing from the provided state to the root of the tree.

The described modifications allow the altered RRT algorithm to find a single tunnel in any protein molecule when provided with a right set of parameters. The result of the search is shown in *Figure 3.5*. *Section 3.3* describes a variety of other modifications which enable us to perform multiple protein tunnel detection.



a) b) c)

Figure 3.5: An example of a result of single tunnel detection. a) *shows the resulting tunnel in the cartoon representation of the protein molecule (1CQW),* b) *shows the same in the spherical representation.* c) *shows the tunnel without the molecule.*

3.3 Multiple tunnel detection

The vast majority of protein molecules contain more than one tunnel. The nature of the utilized algorithms does not, however, allow for multiple planned paths. We have therefore devised a number of modifications to *Algorithm 3*, which allow for multiple paths detection.

The method starts by finding a single tunnel. Once a tunnel is found, it is stored and a so called "*disabled area*" is created with its center in the last node of the trajectory. Its shape is a sphere with a radius equal to the *width* of the tunnel (see *Section 3.3.1*) plus a small overlap determined by the parameter τ_o. Its purpose is to prevent the tree from growing through the region contained by it. Any samples positioned there

are therefore discarded and any subtrajectories intersecting it are considered to be invalid. See *Figure 3.6* for illustrations of this concept and *Figure 3.7* for illustrations of the resulting algorithm's behavior.

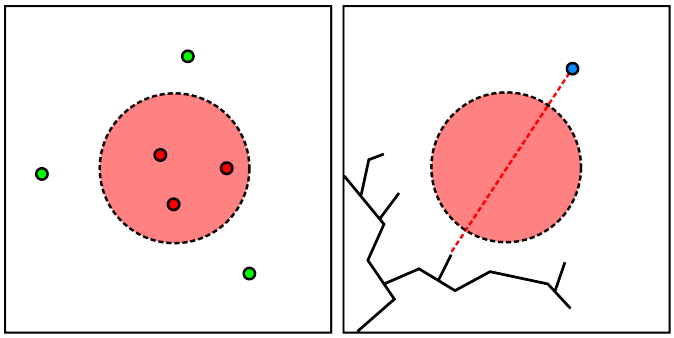


Figure 3.6: A diagram illustrating the functional principle of disabled areas. Figure a) *illustrates the sampling phase, free samples created outside of the disabled area (dotted red circle) are not affected (green), whereas free samples located inside the disabled area (red) are discarded.* b) *shows how an expansion of the tree (black structure in the bottom left corner) is prevented, since the sub-trajectory created by connecting the new sample (red) would intersect the disabled area.*

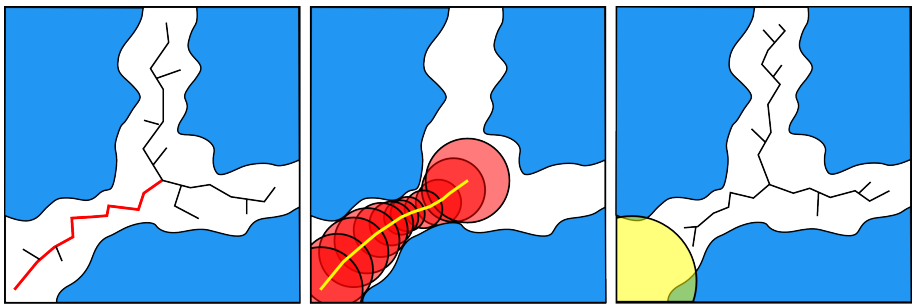


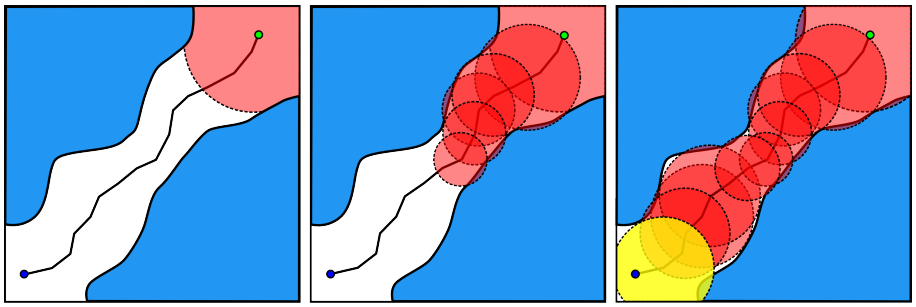
Figure 3.7: A diagram illustrating the functional principle of the TOM-RRT algorithm. a) *shows an initial found path (red) and a grown tree (black).* b) *shows an optimized path (yellow) and the resulting protein tunnel (red).* c) *shows the added disabled area (yellow) and how the newly created tree is prevented from growing into the disabled area.*

Afterwards, the tree created in the previous iteration is discarded in its entirety.

We have experimented with the notion or analyzing the tree and only removing parts of the tree which intersected disabled areas, but the performance was inferior compared to simply generating an entirely new tree, due to the speed of growth.

The algorithm is then restarted and run until a new path is found. If the found path is new and not a duplicate of a previously found path (see *Section 3.3.2* for details regarding duplicate tunnel detection), a new disabled area is created at the end of the newly found trajectory, as depicted in *Figure 3.8*.

New disabled areas are then created along the previously found respective trajectories, closer to the root of the tree and the ligand molecule's original position, but



a) b) c)

Figure 3.8: A diagram illustrating how disabled areas progress. Figure a) *shows how the first disabled area (red) is created at the surface of the protein (blue) after a trajectory is found. The blue node represents an initial state, the green node the final state (and the end of the trajectory).* b) *shows the creation of additional disabled areas after every iteration and the consequent filling of the entire tunnel.* c) *shows the termination of said process, when a new disabled area (yellow) would contain the initial node (blue) and therefore the root of the tree.*

not close enough so that the root of the tree would intersect one of the disabled areas. This results in a structure similar to that of the found tunnel. These steps are then repeated until the program is terminated by the user or until a time-based constraint is fulfilled. A rough summarization of the resulting algorithm's pseudocode is presented in *Algorithm 4*.

```
1 Function findTunnels(q_init)
2     while !terminate() do
3         tunnel ← findSingleTunnel(q_init);
4         tunnel ← optimizeTunnel(tunnel);
5         addNewDisabledAreas(tunnel);
6     end
```

Algorithm 4: A rough working principle of the TOM-RRT algorithm

3.3.1 Tunnel width analysis

Widths of the found tunnels are analyzed using a simple function which, when provided with a position, tries to place spheres with increasing radii in this position to see whether they collide with the environment or not. Once the sphere collides with the protein, the last increase in radius size is reverted and a smaller step is used, until even the smallest increase in size results in a collision with the environment. The resulting radius is then returned as the tunnel's width. See *Algorithm 5* for details of the implementation.

This approach is an approximation of searching for a point closest to the supplied sphere and computing distance between the two, which is significantly more computationally expensive.

```
1  Function CheckTunnelWidth(State s)
2      double i = 0.01;
3      while !isColliding(s, i) do
4          i+ = 1.0;
5      end
6      i− = 1.0;
7      while !isColliding(s, i) do
8          i+ = 0.1;
9      end
10     i− = 0.1;
11     while !isColliding(s, i) do
12         i+ = 0.01;
13     end
14     i− = 0.01;
15     return i;
```

Algorithm 5: The tunnel width analyzing function

The function *isColliding*(*State s*, *double i*) in *Algorithm 5* is a boolean function returning true if a sphere with a set radius position in the supplied state is colliding with the environment.

3.3.2 Tunnel similarity analysis

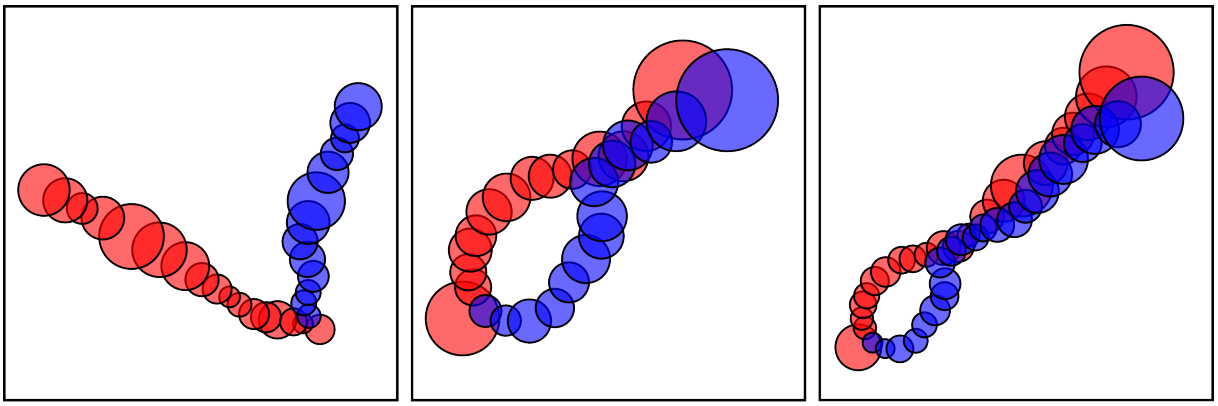
The functionality which allows us to search for multiple tunnels (and paths) gives rise to a new problem, which is the robust and accurate detection of tunnel duplicity, i.e., detecting when multiple paths have been created in the same tunnel in the protein molecule. While the concept of disabled areas should theoretically prevent such scenarios, the complex protein structure can cause the disabled area to fail to block the entirety of a tunnel and a tree can still grow around it.

The solution to this problem is not trivial, since the criterion deciding when two tunnels are identical has not been explicitly defined. We have therefore taken the liberty of creating our own criterion. We have utilized the existence of the α_t parameter, which denotes the maximum tunnel width of a tunnel, which is still considered to be a tunnel (rather than an open space or a cavity) and is set manually by the user.

Two conditions are taken into account when determining whether a new found tunnel is different or not: The first one consists of checking whether at least one of the new tunnel's nodes is located further than the value of the mentioned α_t parameter. The other criterion consists of computing the sum of Euclidean distances of all of the nodes in the new tunnel to their nearest counterparts (i.e., the nodes of previously found tunnels). This sum is then normalized by being divided with the Euclidean length of the new tunnel.

Let us have paths A and B consisting of k and l states:

$$A = \{s_1, s_2, ..., s_k\}; \quad B = \{t_1, t_2, ..., t_l\}$$



a) b) c)

Figure 3.9: A diagram illustrating tunnel duplicity. Figure a) *shows a typical example of two different tunnels.* b) *shows two tunnels that are still regarded as different, despite ending in the same location on the protein molecule's surface. The difference near the beginning of the tunnels is large enough to consider them as different, relative to their length.* c) *shows two tunnels, that are considered as duplicates of each other. Despite being exactly as different as the ones in the previous scenario, when related to their length, the difference between them is not significant enough.*

We can then compute the dissimilarity value of tunnel B as:

$$dissimilarity\ value(B) = \frac{\sum_{i=1}^{l} ||s_i - t'||}{\sum_{i=1}^{l-1} ||t_i - t_{i+1}||}$$

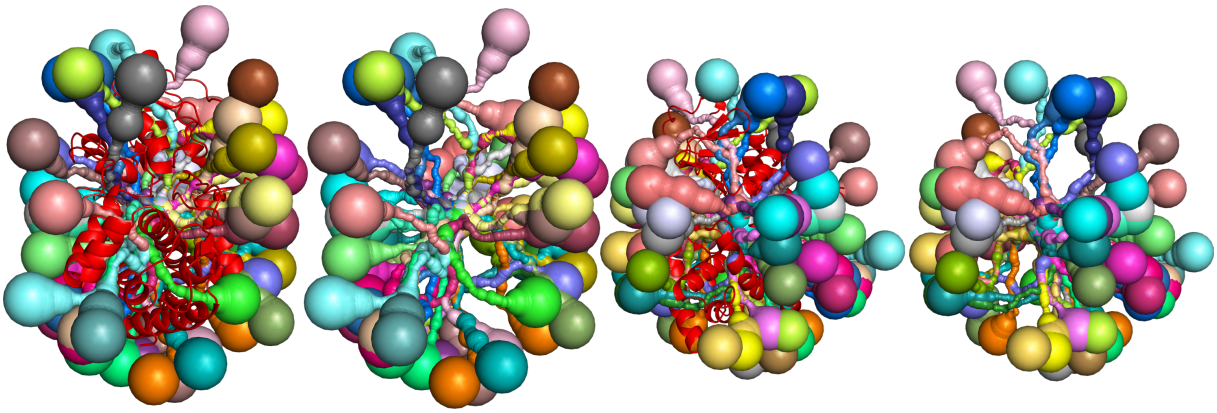
where $||q_i - q_j||$ is the value of the defined distance function between the two states and $t' \in B$ is the closest state to the state $s_i \in A$. Note that although the denominator in the equation could theoretically equate to zero, a tunnel with a zero length is physically impossible. This scenario is checked for in the program and can never occur.

The second condition consists of checking the *dissimilarity value* of a tunnel. If this dissimilarity value is higher than a previously empirically discovered and manually set threshold value in the parameter δ_t (see *Section 5.2* for details), this condition is considered to be satisfied, indicating that the tunnels are different.

The final check consists of a logical conjunction of these two described conditions. The tunnels are considered to be different if and only if the first and the second condition are both true. The resulting behavior is illustrated in *Figure 3.9*

The inherently complex structure of protein molecule's surface sometimes causes the RRT to grow along the surface of the protein, without satisfying the goal condition. As the algorithm is stochastic, a combination of samples can be created which makes the tree grow so that it closely hugs the protein surface. This causes an undesired "tail" at the end of the path, which could, in certain scenarios, result in increasing the tunnel's dissimilarity index.

This situation is mitigated by disregarding both the initial and the final parts of the path when checking the criteria. Since all of the tunnels start in the same position, slight variations in the initial segments of the paths are of no value to us. Similarly, some of the tunnels end in similar locations on the protein surface, while occupying



a) b)

Figure 3.10: A rendering of all protein tunnels found in the 1AKD molecule. Figures a) *and* b) *show the molecule from the front and back respectively, with the molecule visible in the left portions (red).*

different regions inside the molecule. Thus, the ending segment is also unrelated in our search. Nodes, which are closer than the α_t parameter to the first and the last nodes are therefore not taken into account when checking the conditions.

While this behavior does simulate the precision of alpha-shapes and Voronoi diagrams quite well, it can fail if the length of the portion of the path growing along the protein surface is large enough. Both criteria can be satisfied despite the tunnel being a duplicate of an existing one. Even though the likelihood of this scenario is exceedingly low, a manual check by the user of the provided results is still recommended.

That said however, the robustness of this solution was tested with excellent results. After manually checking all 70 tunnels found in the protein molecule *1AKD* (see *Figure 3.10* for illustrations), none were duplicates of each other.

3.4 TOM-RRT

The previous sections described some of the utilized modifications to the RRT algorithm. The resulting implementation employs these modifications, plus a handful of others, which enhance the performance in the protein molecule environment.

The standard RRT algorithm attempts to connect samples regardless of their distance. This can be undesirable, since a complex environment might be difficult to traverse with large steps and smaller steps would help the tree to spread more quickly. We have therefore employed a limiting parameter, which specifies the maximum length of a newly connected segment. The algorithm creates a new sample, finds the closest node in the tree and tries to connect these two. If the trajectory is collision-free, the sample is connected. If it is not and the distance between the two samples is larger, the sample is moved so that the distance between the samples is equal to the γ parameter. If this new short trajectory is collision-free, the moved sample is connected, otherwise it gets discarded. See *Figure 3.11* for illustrations of the resulting behavior.

The γ parameter has to be carefully set by the user to allow the tree to both grow in

complex environments and not let it be hindered by too low growth speed.

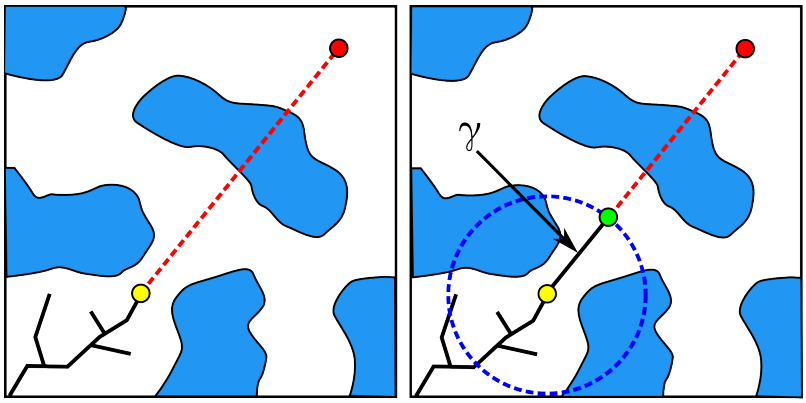


Figure 3.11: A diagram illustrating how a sample is moved when being connected to the tree. Figure a) *shows how unlimited tree growth can result in an undesirable behavior, since the tree wouldn't grow in this scenario.* b) *shows how the sample is moved to a more favorable position.*

Another approach consists of interpolating newly formed trajectories to a given resolution. The standard RRT algorithm connects new states directly to edges of a graph (see *Figure 3.12*). Since searching for the nearest node in the in the tree's connecting trajectories is computationally significantly more complex than searching for nearest nodes in the tree, any new trajectory is interpolated to a resolution set by the ϵ parameter. This creates a sufficient approximation of the desired behavior while keeping the computational complexity at low levels, thanks to the external library MPNN (see *Section 5.1.1* for details). The resulting detailed pseudocode of the TOM-RRT algorithm is listed in *Algorithm 6*.

```
1  Function TOM − RRT(s_init)
2      terminate ← false;
3      while !terminate do
4          resetPlanner();
5          terminate ← checkConstraints();
6          tunnel ← singleTunnelSearch(s_init);
7          tunnel ← optimizeTunnel(tunnel);
8          if isTunnelNew(tunnel) then
9              exportTunnel(tunnel);
10             addNewDisabledArea(tunnel);
11         end
12         moveDisabledAreas();
13     end
```

Algorithm 6: The resulting TOM-RRT pseudocode

The variable *terminate* in *Algorithm 6* is a boolean variable, *checkConstraints*() is a boolean function returning true when time-based termination conditions are fulfilled

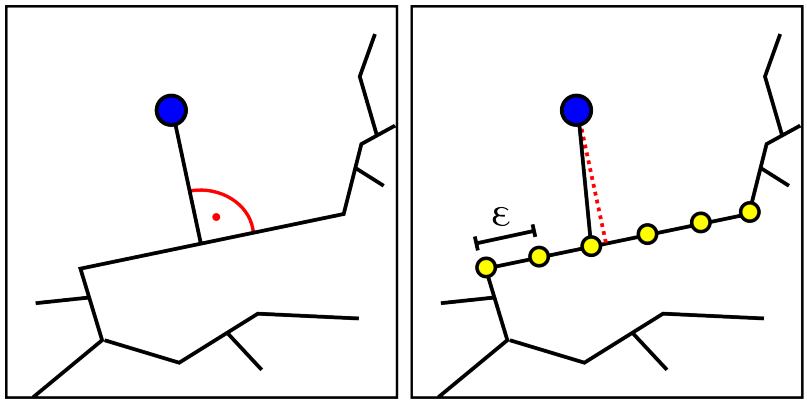


Figure 3.12: A diagram illustrating trajectory interpolation. Figure a) *the behavior of a standard RRT algorithm. The new trajectory is formed at precisely right angles to the tree's subtrajectories, therefore being the shortest possible.* b) *shows an approximation of this behavior, which is implemented using a nearest neighbor finding library at a fraction of the computational cost. The existing trajectories have been interpolated (yellow) and the new node can be connected only to the existing nodes, as opposed to the previous example, where the node has been connected directly to the trajectory. The approximation error is shown in red.*

$singleTunnelSearch(State\ s)$ is a function described in *Section 3.2.2* returning the first found tunnel, s_{init} is the initial state, $optimizeTunnel(Tunnel\ t)$ is a function which returns a smoothed-out tunnel with the largest widths possible, $isTunnelNew(Tunnel\ t)$ is a boolean function returning true when the tunnel has not previously found before, $exportTunnel(Tunnel\ t)$ is a function which exports the supplied tunnel for further use in external software, $addNewDisabledArea(Tunnel\ t)$ which stores the found tunnel so that it can be later used for the process of disabled areas generation and $moveDisabledAreas()$ is a function which generates new disabled areas from the stored tunnels.

3.4.1 Path optimization

The output of the utilized algorithms (in our implementation a sequence of states of the ligand molecule interpolated in an equidistant way, according to the ϵ parameter) is far from optimal. The main task of the utilized algorithms is to quickly find a path in an unknown and complex environment, regardless of any optimality. It can therefore contain a number of sections which can later be optimized or entirely eliminated.

Our optimization focuses on two criteria. In the first phase, we strive to achieve the shortest possible trajectory. This results in a path closely hugging the environment's atoms. This behavior is undesired, since the path is later used to analyze the tunnel's parameters and it would cause the program to report tunnel width deviating down from the actual value (tunnel width computed from a path located precisely in the center of a tunnel is higher than that of a path offset from said center). This is therefore corrected in the second phase, where the trajectory is moved to the middle of the tunnel

via a local minima search. See the sections below for a more detailed explanation.

Path length optimization

This phase utilizes two approaches. First, the path is interpolated to a manually preset resolution. Afterwards, *Algorithm 7* is run. The algorithm iterates over states in the found path and tries to connect them with the furthest possible state through a collision-free trajectory.

```
1 Function OptimizationPhaseOneAlgorithmOne(initialTunnel)
2     foreach i ∈ {0, 1, ..., initialPath.size() − 1} do
3         foreach j ∈ {0, 1, ..., initialPath.size() − 1} do
4             if collisionFree(initialPath[i], initialPath[j]) then
5                 initialPath ← deleteNodes(initialPath, i, j);
6             end
7         end
8     end
9     return initialPath;
```

Algorithm 7: The first algorithm of path optimization

The variable *initialPath* in *Algorithm 7* is an array containing a sequence of nodes (similar to those described previously), $collisionFree(Node\ init, Node\ final)$ is a function which returns true if a trajectory formed via interpolating the two supplied nodes is collision free and $deleteNodes(initialPath, i, j)$ is a function which deletes nodes positioned between indices i and j in the supplied array and returns the result. See *Figure 3.13* for a possibly more intuitive explanation.

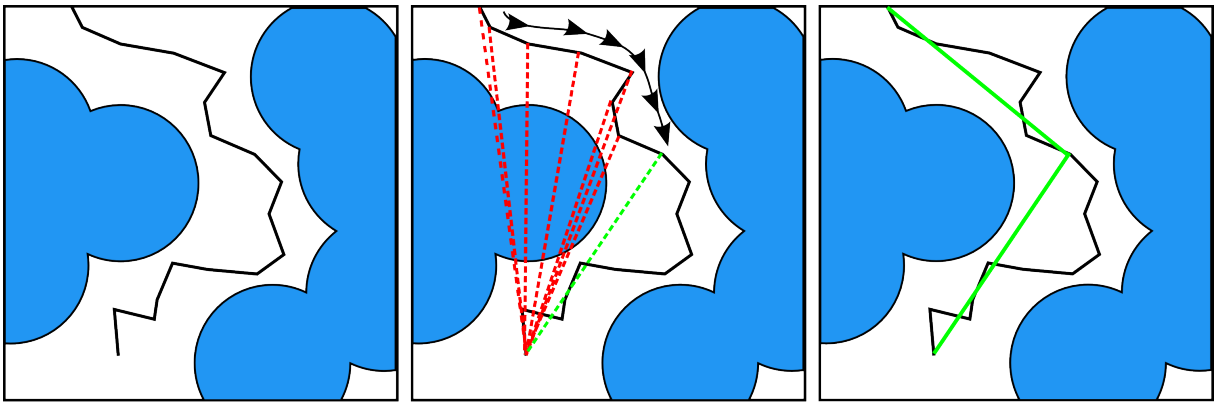


Figure 3.13: A diagram illustrating Algorithm 7. *Figure* a) *shows an initial found trajectory. Figure* b) *illustrates how the algorithm iterates over the nodes and tries to connect each pair. Figure* c) *shows the final result of this trajectory optimizing algorithm (green).*

This results in a significantly shorter, but a rather jagged path. The path is then interpolated to a resolution specified by the ϵ parameter followed by running the second optimization algorithm, *Algorithm 8*,which focuses more on smoothing out the path. This algorithm tries to connect nodes neighboring to a randomly selected node, thereby eliminating the "sharp" corners resulting from the previous algorithm while also shortening the path.

```
1 Function OptimizationPhaseOneAlgorithmTwo(initialPath)
2 |  i ← random integer ranging from 1 to initialPath.size() − 1;
3 |  if collisionFree(initialPath[i − 1], initialPath[i + 1]) then
4 |  |  initialPath ← deleteNodes(initialPath, i − 1, i + 1);
5 |  end
6 |  return initialPath;
```

Algorithm 8: The second algorithm of path optimization

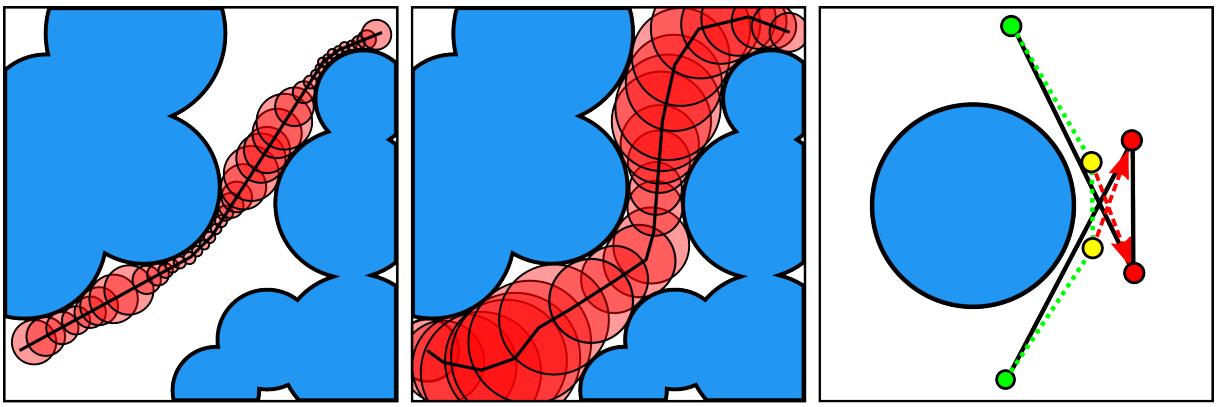
After the path is optimized, it is interpolated again using the ϵ parameter. These iterations are run multiple times, according to the ι parameter.

Path position optimization

The second phase of path optimization alters the path in a way which allows for later tunnel parameters analysis. As stated previously, the path is a sequence of nodes (subtrajectories). Iterating from the initial to the final node, a normal vector is computed from the currently optimized and the preceding node. We can use this normal vector to describe a plane perpendicular to said normal vector. It is our goal to only move the currently optimized node in this plane, to avoid creating unnecessary sections and loops in the path. See *Figure 3.14* for illustrations.

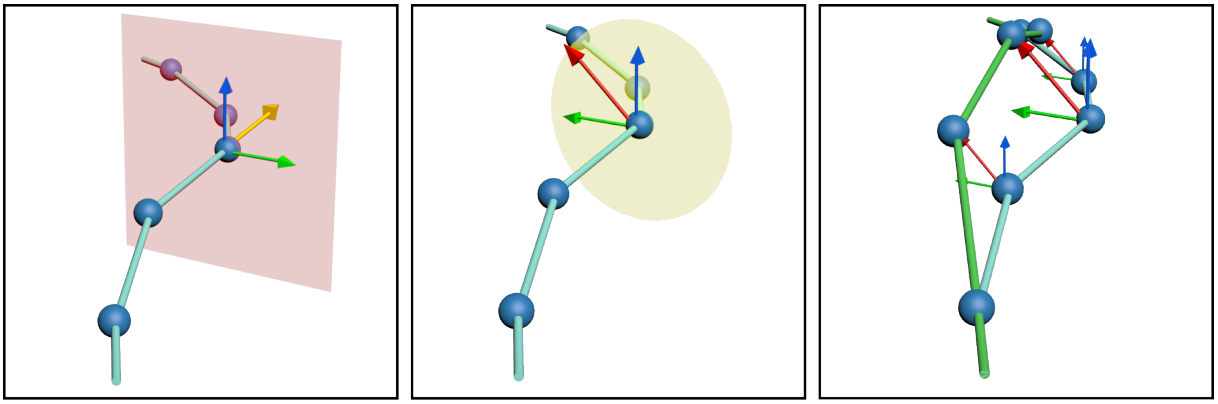
The said plane is then used to compute two perpendicular and linearly independent vectors lying in the plane. Thus, a combination of these vectors allows us to "reach" any point in the normal plane. We then utilize simulated annealing to try and position the nodes in their optimal positions (in the center of the tunnel).

A random angle is picked to create a vector on a unit circle. This vector is then decomposed into the sine and cosine coefficients in the said unit circle. These coefficients are then used to multiply the lengths of the previously mentioned perpendicular vectors, along with the final vector length coefficient, which decreases as the iterations progress (thereby ensuring than the nodes move by smaller distances as the iterations progress). After the node is moved using the previously created vectors, tunnel width is computed again. If its previous value was lower, the two nodes neighboring to the previously moved node are moved in the same direction (as depicted in *Figure 3.15*), with half of the final vector's length and are checked for collisions. If the neighboring nodes are free as well, all three nodes are kept in their positions, otherwise the entire operation is reverted. The position alteration of the neighboring nodes is required to prevent excessive jaggedness of the resulting path.



a) b) c)

Figure 3.14: A diagram illustrating how a position of a path affects the resulting tunnel's width. Figure a) *shows what would happen if we tried to analyze tunnel width (red circles) using a path optimized for shortness. Notice how the trajectory closely hugs the protein atoms (blue). Figure* b) *shows the desired optimal position of the path with the biggest possible resulting tunnel widths. Figure* c) *shows the result of moving nodes using only local minima searches without the mentioned normal plane. The yellow nodes, previously located in non-optimal positions were moved in the directions of the orange arrows, which resulted not only in a loop in the final path, but also in the lower optimality of the resulting trajectory (compared to the original green trajectory).*



a) b) c)

Figure 3.15: A diagram illustrating path position optimization. Figure a) *displays a normal vector (yellow) and the resulting plane (red) and its vectors (blue and green).* b) *shows how an angle is picked (red vector in a yellow circle) and how it is decomposed into the previous two vectors.* c) *shows the difference between the previous (red) trajectory and the final trajectory.*

4 Non-spherical ligand planning

As stated previously in *Chapter 2*, one of the main advantages of sampling-based path planning algorithms is the ability to plan for non-spherical objects (an option which is absent in tools utilizing discrete voxel grids, Voronoi diagrams or Delaunay triangulation). The possibility of detecting tunnels when taking into account the molecule's specific shape could allow us to both find new tunnels and to accommodate tunnels, which would have been discarded if we used a spherical probe.

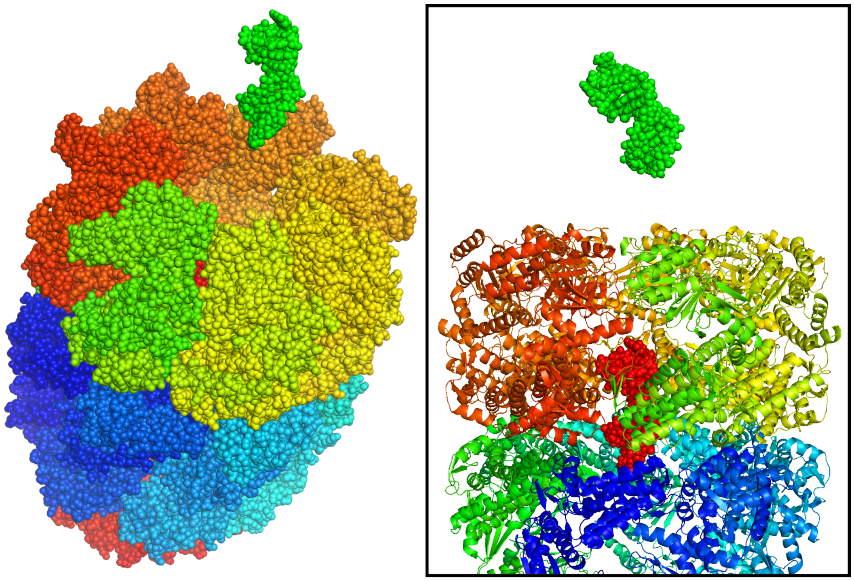
We have investigated this notion using a version of the RRT algorithm which utilizes some of the previously described modifications plus an improved version of collision detection. This resulted in a speed improvement when working with protein molecules, as described below.

4.1 Collision detection

As stated in *Section 5.1.3*, we have utilized an external library (*OZCollide*) for the purpose of collision detection. The final implementation of collision detection functionality consists of checking every atom (or rather its geometric primitive representation) of the object (ligand molecule) for collisions with the AABB tree of the environment (protein molecule). See *Figure 4.1* for an example of non-spherical ligand molecule. A count of the amount of previous collisions is kept for every atom's representation and every specified number of iterations the order of collision checking is altered to reflect it (the atoms with the highest collision counts are checked first). Since atoms on the border regions of the ligand molecule are most likely to collide with the environment, this implementation should result in a considerable speed improvement.

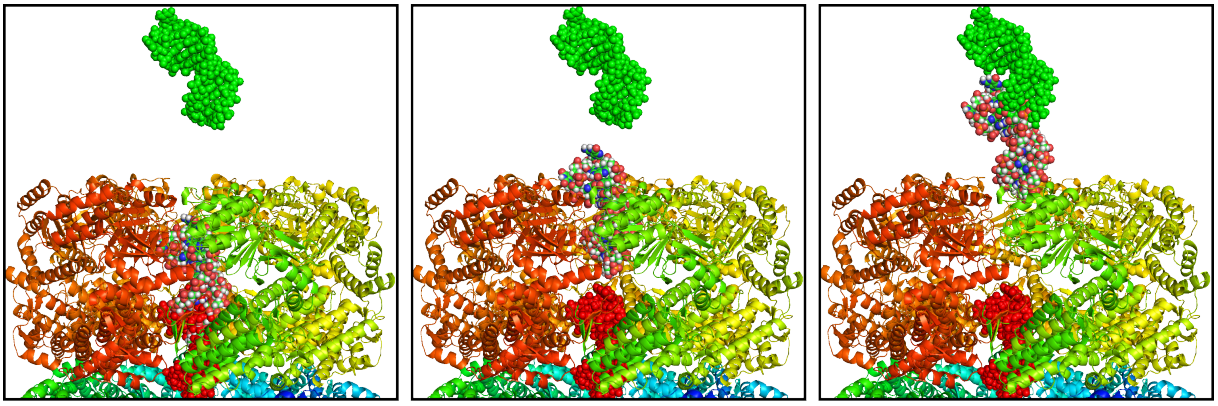
Figure 4.2 depicts the possibility of planning motions from the inside of a large molecule using ligand molecules with non-spherical structure. A movie generated from this motion sequence can be found on the attached disk.

The resulting implementation is able to find paths inside protein molecules for ligands of considerable complexities, however, due to a lack of time, it was only tested on simple molecules.



a) b)

Figure 4.1: An example of path planning for a non-spherical molecule. Figure a) *shows a spherical representation of a large protein molecule (1AON) with a large noticeable cavity and a spherical representation of a ligand molecule (2N7X).* b) *shows a cartoon representation of the protein molecule and the initial (green) and final (red) positions of the ligand molecule.*



a) b) c)

Figure 4.2: An example of a planned motion sequence for a ligand molecule. Figures a) *to* c) *show a sequence of snapshots from a planned trajectory of the ligand molecule.*

5 Experimental results

This chapter presents the results of the final algorithm, as well as a number of comparisons to the previously mentioned *CAVER 3.0* [14].

5.1 Implementation details

The previously described algorithm utilizes a number of high-level functions, which are non-trivial to efficiently implement in even low dimensional configuration spaces. In our case, the main performance drawbacks were the collision detection routine for complex objects and the problem of finding a nearest-neighbor node in n dimensions (in our case $n = 3$). Performance of the resulting protein tunnel-detecting solution relies heavily on speed and efficiency of these high-level functions. We have therefore decided to employ a multitude of external libraries in order to save time and avoid as many software bugs as possible. A number of available external libraries was integrated and tested for its performance, in order to ensure the highest speed possible in our application.

5.1.1 Nearest neighbor search

As the problem of finding a closest vector in an n-dimensional space is non-trivial for high-dimensional configuration spaces and performance of its implementation is crucial for overall speed, an external library was utilized. The *MPNN library* [6], which utilizes kD-trees and supports n-dimensional manifolds (Cartesian products of Euclidean one-space, circle, and three-dimensional rotation group, SO(3)), provided excellent performance and low memory-complexity while being extremely easy to integrate and use.

5.1.2 Collision detection

As previously stated, the atomic count in protein molecules can reach into the orders of hundreds of thousands of atoms (the "Titin" protein consists of more than 500 000 atoms). Since collision detection routines can be invoked in the orders of hundreds of millions of times during both the path finding and path optimizing phase, it is crucial

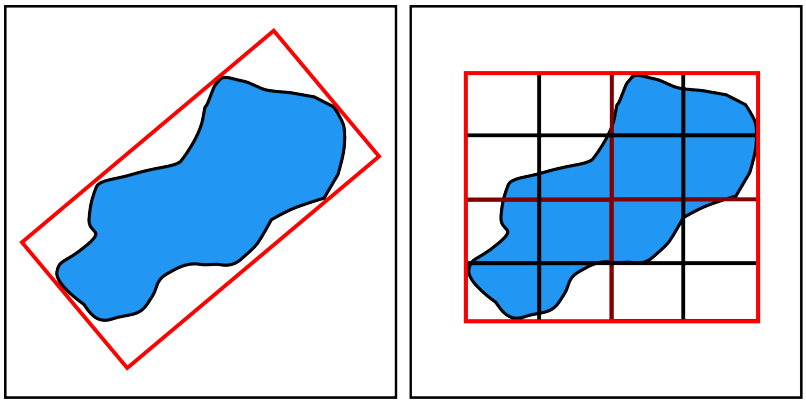
that the collision detection implementation is both quick, even with such complex objects and has low memory requirements, because of said complex object structure.

Individual atoms were represented using spheres with radii equivalent to the Van der Waals radii of the respective atoms. Collision libraries which allowed the usage of geometric primitives were therefore preferred, since libraries which lacked such functionality required a memory costly 200 polygon triangulation sphere approximations to be used instead.

OBB and AABB trees

Fast collision detection relies on dedicated data structures, such as oriented bounding boxes or axis-aligned bounding box trees (see *Figure 5.1* for illustrations of the concepts). Oriented bounding boxes (OBB) speed up collision detection by simplifying the detection process in some cases. A smallest possible 3D box is positioned around a more complex object in such a way that the object is contained by it completely. During collision detection the OBBs are then checked first to see if they are colliding. If they are not, the more complex objects inside them can not possibly be colliding either and the whole process is sped up immensely without the costly and complex checks of the said objects.

Axis-aligned bounding box trees utilize a similar process, the complex object is split into smaller segments, each of which is wrapped by an axis-aligned bounding box. The resulting boxes are then repeatedly wrapped in bigger bounding boxes, until all of the boxes are contained in a single big bounding box. During the collision detection phase, collisions between the boxes are checked according to their size — the largest boxes are checked first.



a) b)

Figure 5.1: A diagram of an oriented bounding box (red rectangle) around a protein molecule (blue) in figure a) *and an axis-aligned bounding box tree* b) *— the darker shades of red indicate a lower position in the tree.*

Bearing the above criteria in mind, several available collision detection libraries

were tested. A previously written report [17] regarding this issue was used to find the best looking candidates available. The resulting list with attributes is available below.

Naive implementation

Used mostly as a "sanity check" for other collision detection libraries, this implementation utilizes the simplest spherical collision detection possible. It compares the distance between origins of two spheres to the sum of their radii. If the distance is smaller that the sum, the spheres are colliding. In the worst case scenario, the implementation checks every sphere of one object for collisions with every sphere of the other object and therefore results in a performance much worse than that of better optimized implementations.

RAPID

The Robust and Accurate Polygon Interference Detection [29] library is a lightweight, simple to use intersection detection library built on an underlying structure of OBB trees. However, since it is almost 20 years old, it lacks some of the functionality offered by newer implementations, such as the support for geometric primitives. Instead, it relies on 3D triangulations to represent objects. While its performance was the best of all tested implementations in non-colliding scenarios, it was falling behind when the objects were colliding, despite using the available speed optimizations (stopping intersection detection functions on the first detected triangle intersection). Given these facts, combined with the lack of precision resulting from having to use spherical approximations, it was deemed unsuitable for our purposes.

Bullet Physics

The Bullet Physics [1] library is a very well optimized and maintained library created for the purpose of robust real-time physical simulation of both rigid and soft bodies. Despite its main focus, it does provide the functionality of simple polygon intersection detection. Given the purpose of its creation, it has several characteristics which make it difficult to use in our application (e.g. the robust and quick physical simulation functionality requires a non-zero collision detection tolerance to be set). This can, however, be manually disabled for the purposes of intersection detection. Its most important issue is the fact that it is very memory costly even for small objects. Despite the support for geometric primitives, it requires the creation of a static discrete collision environment virtual structure, which alone can take up to several hundreds megabytes' worth of RAM. The measured amount of required memory is provided in *Table 5.1*.

The "freeze" note indicates that the precise amount couldn't be determined, since the operating system lacked the available RAM space to function properly and stopped working. While being an excellent choice for real-time physical simulations, we deemed it unsuitable for our purposes.

Object atom count	Environment atom count	Required memory [MB]
250	25000	700
250	50000	1125
500	25000	1500
500	37500	2020
500	50000	2125
600	35000	1840
750	35000	>2300 (freeze)
600	50000	>2200 (freeze)

Table 5.1: Measured amount of memory taken up by the Bullet library while checking collisions 25 times.

OZCollide

OZCollide [23] is a recent, lightweight and easy-to-use collision detection library with the support of geometric primitives and many speed optimizations, such as the automatic AABB trees creation. While the advertised collision checking between two AABB trees appears to be lacking the described functionality (in the sense of not functioning at all), collision checking between a geometric primitive and an AABB tree provided excellent performance, surpassing all other implementations. We have, therefore, decided to utilize this implementation in our solution.

5.1.3 The resulting performances

Provided below are computing times of each implementation. Given the nature of the task and the optimizations, we have decided to test the implementations in colliding (the object was intersecting the environment) and non-colliding scenarios separately.

A modular programming solution was developed so that the collision detection libraries could be switched with ease. The individual implementations were tested using randomly generated environments and objects with a set number of atoms for both, in order to simulate real-life scenarios as closely as possible. The tests were run 1100 times for each set of graphs.

As is clearly visible, the performances differ significantly between colliding (*Figures 5.2, 5.3, 5.6, 5.7*) and non-colliding scenarios (*Figures 5.4, 5.5, 5.8, 5.9*). This is likely caused by the fact that available optimizations such as the AABB-trees can not be fully utilized when the objects are colliding and the implementations have to resort to slow polygonal intersection checking. While the performance of OZCollide is slightly inferior in non-colliding scenarios, it is still comparable to other implementations and, since the objects are colliding in most cases (given the complex nature of both the object and the environment), where its performance is superior, it is obviously the best candidate of all checked possibilities.

The final implementation is described in greater detail in *Section 4.1*.

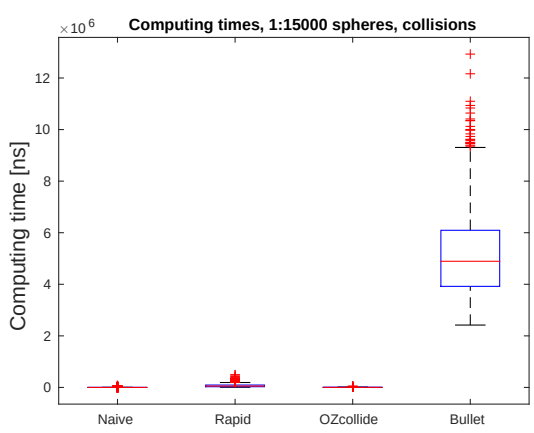


Figure 5.2: Computing times, 1:15000 spheres, colliding scenarios

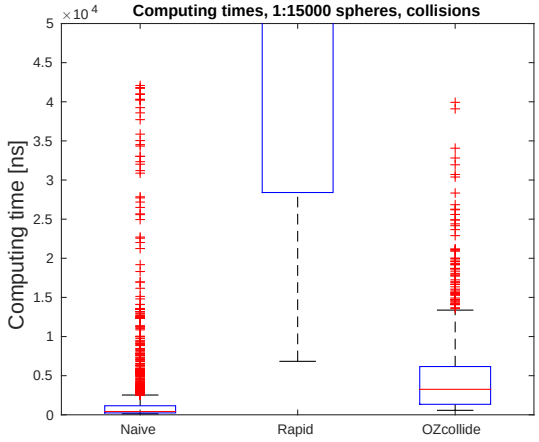


Figure 5.3: Close-up of computing times, 1:15000 spheres, colliding scenarios

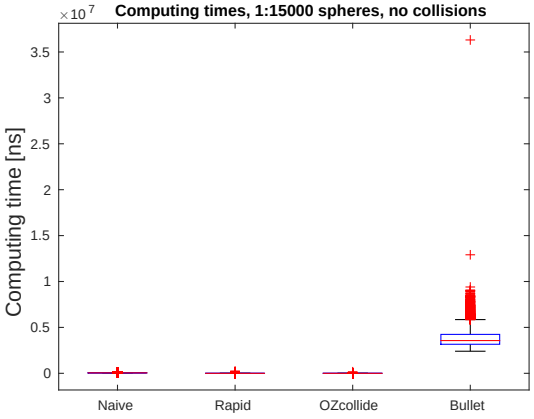


Figure 5.4: Computing times, 1:15000 spheres, non-colliding scenarios

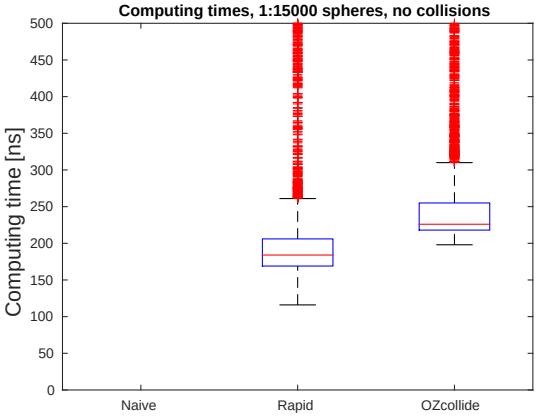


Figure 5.5: Close-up of computing times, 1:15000 spheres, non-colliding scenarios

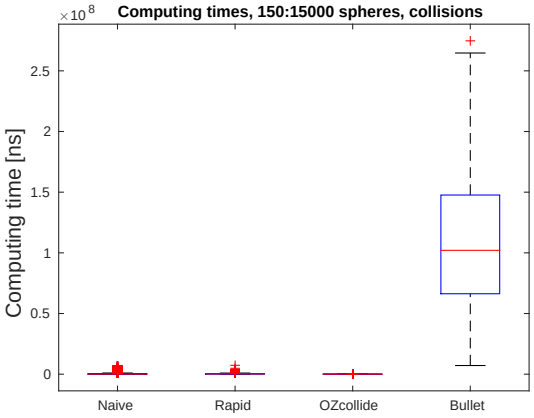


Figure 5.6: Computing times, 150:15000 spheres, colliding scenarios

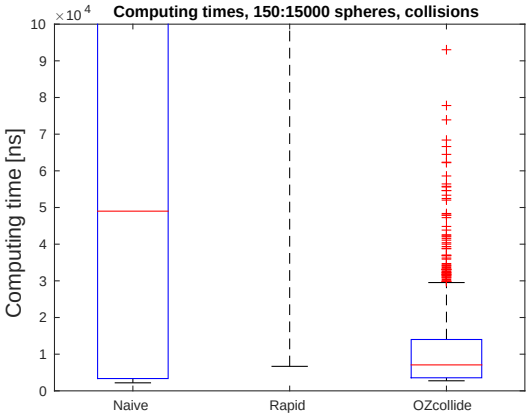


Figure 5.7: Close-up of computing times, 150:15000 spheres, colliding scenarios

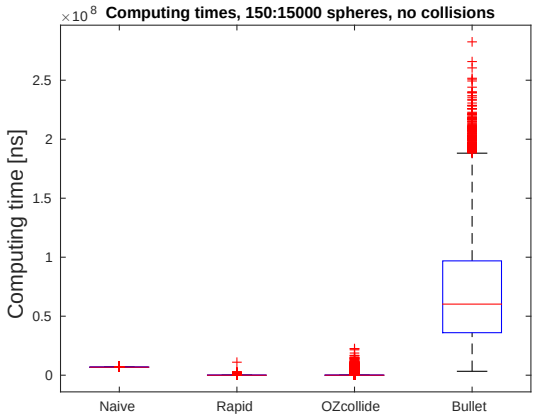


Figure 5.8: Computing times, 150:15000 spheres, non-colliding scenarios

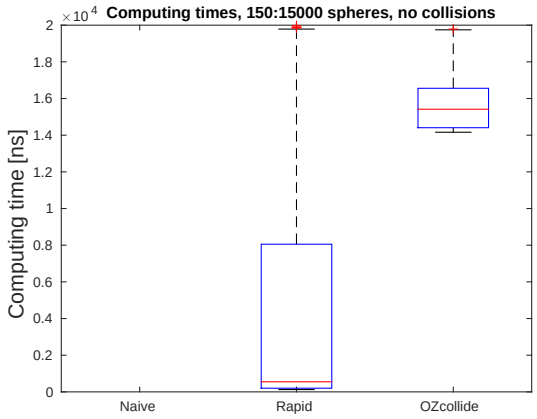


Figure 5.9: Close-up of computing times, 150:15000 spheres, non-colliding scenarios

5.2 Tunnel matching threshold

The dissimilarity metric introduced in *Section 3.3.2* requires a previously set threshold parameter δ_t to function. Using the *1CQW* and *1AKD* molecules an initial tunnel was found. Afterwards, the *dissimilarity value* was computed for each of the tunnels found next and these tunnels were manually inspected. If they differed from the initial tunnel, the lower bound was raised to that value. If the tunnel differed, the upper bound was lowered to that value. Using this process, the value of the threshold δ_t was empirically determined to be $9,0$. See *Figure 5.10* for depictions of the resulting behavior.

Please note that this value may differ for molecules of different size, as the tested molecules all occupied comparable regions of space.

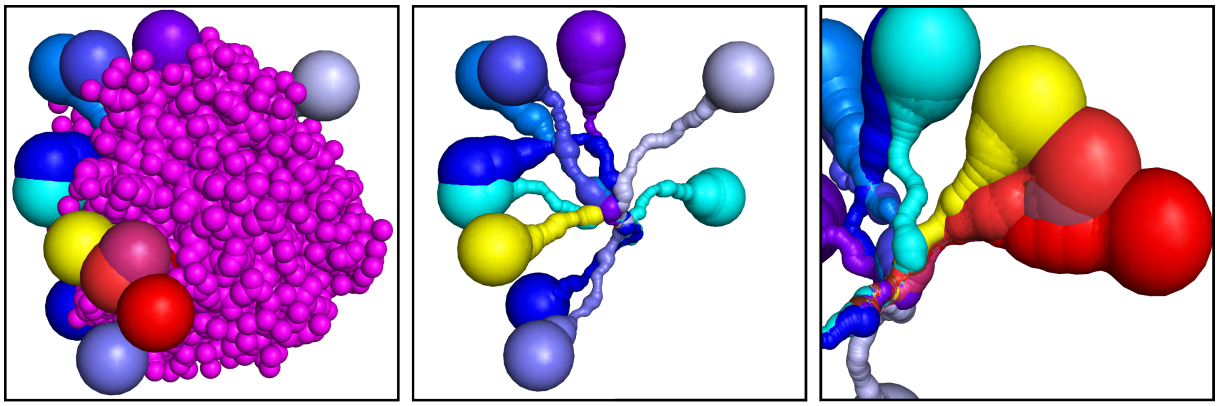
5.3 Comparison to CAVER

Below are the final results of the TOM-RRT algorithm. Note that tunnels found by *CAVER* 3.0 [14] and not found by our software are not listed. Each column contains the tunnel's index number, the length of the tunnel, the number of times the tunnel was found by our solution and the number of times the test was run.

The tests were run on two computers:

- *Computer 1* was an HP OptiPlex 755 desktop station with an Intel® Core™ 2 Duo E6550 CPU with 2 cores clocked at 2.33GHz and a 3GB RAM, running Ubuntu 14.04 LTS 64-bit operating system. The program was compiled using GNU Make 3.81 and gcc version 4.8.4.

- *Computer 2* was an HP ProBook 450 G1 laptop with an Intel® Core™ i5-3230M CPU with 4 cores clocked at 2.60GHz and an 8GB RAM, running Ubuntu 14.04 LTS 64-bit operating system. The program was compiled using GNU Make 3.81 and gcc version 4.8.4.



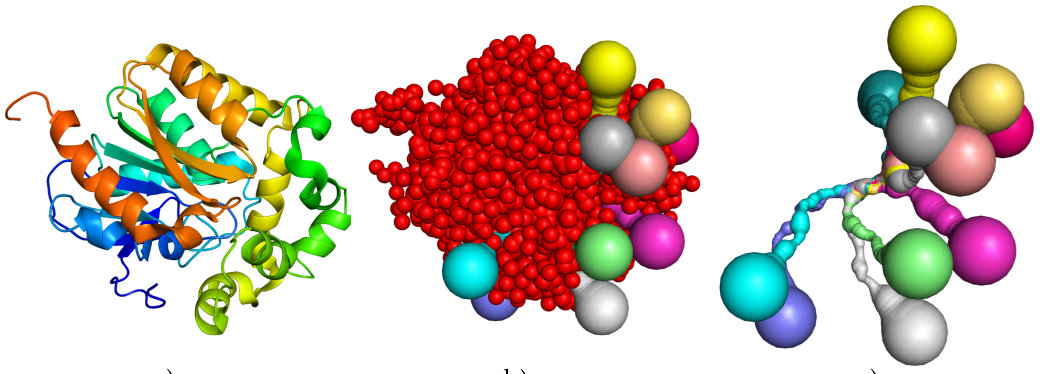
a) b) c)

Figure 5.10: A representation of tunnel difference detection shown in a). *After the threshold value was determined, an initial tunnel (yellow) was found in the 1CQW protein molecule (purple). Afterwards 40 more tunnels were found. Those which differed from the original tunnel are colored with shades of blue, those which did not are colored with shades of red. Please note that some tunnels were omitted for the purpose of better visibility.* b) *shows a close-up of the differing tunnels and* c) *shows a close-up of the duplicate tunnels.*

The algorithm was run utilizing 4 threads on *Computer 1* and 8 threads on *Computer 2*.

5.3.1 1CQW

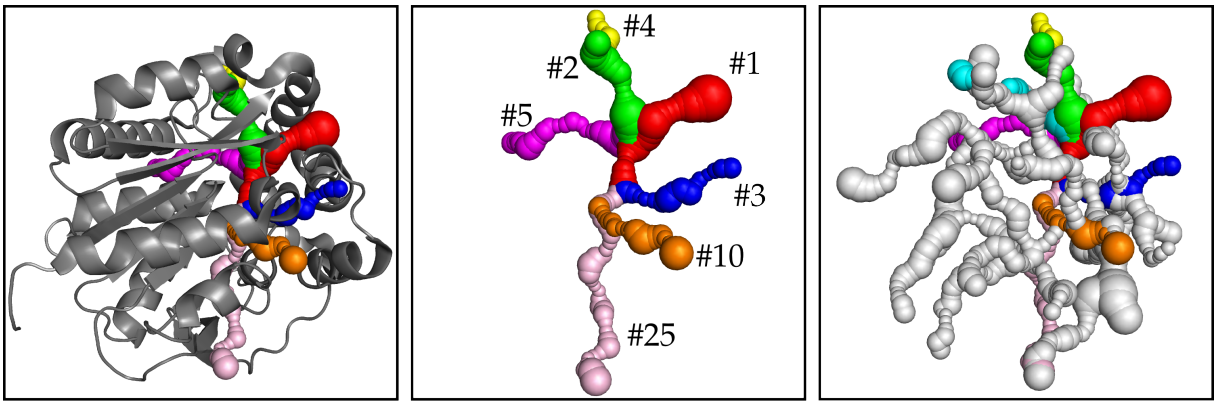
The structure of the *1CQW* protein molecule is depicted in *Figure 5.11* along with an example of the set of the found tunnels. The resulting performance of the TOM-RRT algorithm for the *1CQW* molecule is listed in *Table 5.2*. The test was performed on *Computer 1*.



a) b) c)

Figure 5.11: Details of the 1CQW molecule. a) *shows the cartoon representation,* b) *shows the spherical representation with the found tunnels and* c) *shows an example of the found tunnels.*

Upon analyzing the results in *Table 5.2* we're struck with an interesting artifact — tunnels rated by *CAVER* rather low in the list have a significantly higher probability of being found than the other low rated tunnels (these tunnels are depicted in *Figure 5.12*.



a) b) c)

Figure 5.12: Analysis of found tunnels in the 1CQW molecule. a) *shows the cartoon representation with some of the found tunnels,* b) *the found tunnels, #1 in red, #2 in green, #3 in blue, #4 in yellow, #5 in purple, # 10 in orange and #25 in pink.*c) *shows all protein tunnels.*

1CQW (2352 atoms, 2 500 000 iterations, 50 runs) Probe radius:												
0.6Å			0.7Å			0.8Å			0.9Å			
#1:	17,34	100%	#1:	17,34	100%	#1:	17,34	98%	#1:	17,34	68%
#2:	20,75	98%	#2:	20,75	90%	#2:	20,75	74%	#2:	20,75	2%
#3:	16,14	90%	#4:	28,10	45%	#3:	28,10	14%			
#5:	28,10	66%	#5:	28,25	2%						
#6:	28,14	12%	#13:	36,59	2%						
#7:	26,83	14%	#17:	35,54	6%						
#8:	28,61	8%									
#9:	19,67	4%									
#10:	21,84	92%									
#11:	23,73	2%									
#13:	23,57	2%									
#14:	26,61	12%									
#21:	36,59	24%									
#23:	30,96	22%									
#24:	39,72	2%									
#25:	30,89	56%									
#27:	30,56	2%									
#30:	37,98	2%									
CAVER tunnel count:												
30			23			12			2			

Table 5.2: Results for the 1CQW molecule, 2 500 000 iterations, 50 runs. Each column contains the CAVER*'s tunnel index, the length of the tunnel (in Ångstroms) and the probability of being found by the TOM-RRT algorithm.*

To try and explain this phenomenon we have utilized the Pymol visualization and analyzed the tunnel characteristics manually.

The high probability of finding tunnel #1 (depicted in a red) stems from its large width and short length. Given the characteristics of the RRT algorithm, this ensures that it is the one most likely found with the highest speed. Tunnels number #2 (depicted in green) and #3 (blue color) are analogous, since they share a large portion with tunnel #1. The reason tunnel #4 (yellow color) wasn't found is most likely due to the fact that it branches off tunnel #2 near the end and continues through a long section of the molecule. This possibly resulted in the tunnel being cut off by a disabled area before the RRT algorithm could progress through the long section. Tunnels #5 (purple color), #6 and the rest progress as expected, with the probabilities decreasing due to increasing lengths of the respective tunnels.

We can notice a rather significantly high probabilities of finding tunnels #10 and #25. This is presumably due to the fact that these are the only tunnels located in a rather large section of the molecule and due to the nature of the RRT algorithm, have an increased probability of being found. Furthermore, tunnel #10 (depicted in orange) is rather straight relative to the rest of the tunnels, which makes is a more suitable environment for the RRT algorithm. Tunnel #25 (pink color) has a lower probability of being found when compared to tunnel #10, since it is a branch of tunnel #10.

Figure 5.13 plots the median of the number of found tunnels against the used radius of the probe. The results are as expected, with a larger probe resulting in a lower found tunnel count. Furthermore, due to the stochastic nature of the RRT algorithm as the resulting TOM-RRT algorithm, the number of found tunnels using this method is always less than or equal to the number of found tunnels using *CAVER*, which is exhaustive and deterministic.

Figure 5.14 plots the median of the running times for 2 500 000 iterations against the used radius of the probe. Once again, the results are as expected. The higher number of tunnels due to the usage of a smaller probe results in longer running times. This is caused by an increased number of checks due to a higher number of disabled areas, located inside the tunnels.

Judging by the results, the TOM-RRT algorithm performs well in cases with a lower number of protein tunnels. This is supported by the number of found tunnels comparable to the results of *CAVER* (especially in cases with probe radius equal to 0,9 Å). Furthermore, overall speed of the algorithm in situations with a lower tunnel count is significantly improved (see *Figure 5.14*).

5.3.2 1AKD

The structure of the *1AKD* protein molecule is depicted in *Figure 5.15* along with an example of the set of the found tunnels. The resulting performance of the TOM-RRT algorithm for the *1AKD* molecule is listed in *Table 5.3*. The test was performed on *Computer 1*.

Upon analyzing the results in *Table 5.3* we're once again struck with an interesting artifact —21 tunnels have a probability of being found by the TOM-RRT algorithm over 50 %. None of the other molecules exhibited this behavior, which is most likely

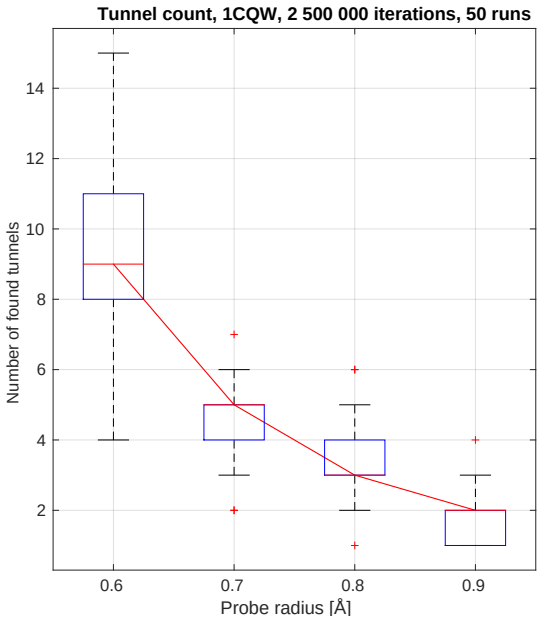


Figure 5.13: 1CQW — Number of found tunnels

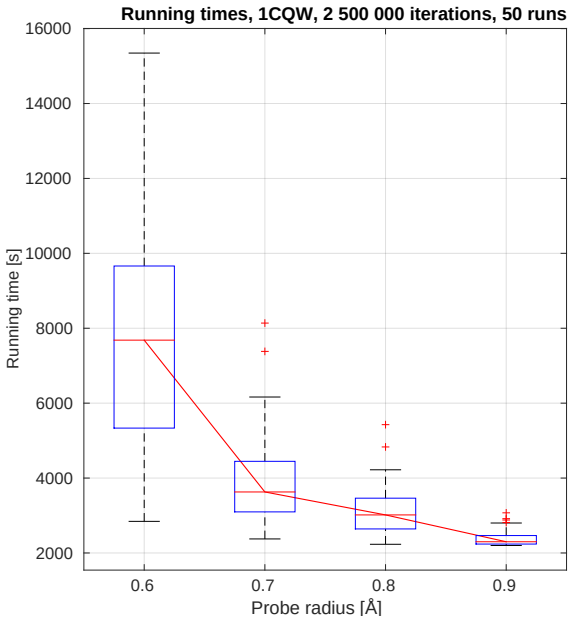


Figure 5.14: 1CQW — Running times

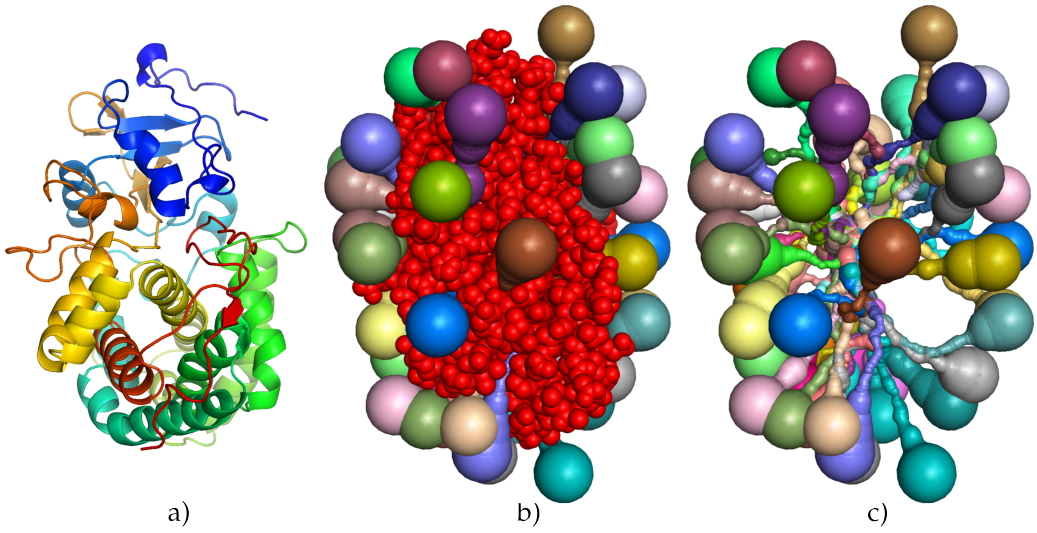


Figure 5.15: Details of the 1AKD molecule. a) *shows the cartoon representation,* b) *shows the spherical representation with the found tunnels and* c) *shows an example of the found tunnels.*

1AKD (3208 atoms, 2 500 000 iterations, 50 runs) Probe radius:												
0.6Å			0.7Å			0.8Å			0.9Å		
#1:	14,41	100%	#1:	16,96	32%	#1:	14,49	100%	#2:	24,51	100%
#2:	14,49	100%	#2:	14,49	100%	#2:	19,23	8%			
#3:	16,79	100%	#3:	16,79	46%	#5:	25,01	36%			
#4:	19,23	86%	#4:	19,23	68%	#6:	26,32	2%			
#5:	19,31	86%	#6:	25,01	34%	#7:	24,51	36%			
#6:	22,77	60%	#7:	26,45	100%	#9:	26,6	94%			
#7:	25,01	86%	#8:	24,51	98%	#12:	37,03	4%			
#8:	26,45	98%	#9:	22,67	8%						
#9:	24,51	98%	#10:	23,49	18%						
#10:	22,67	42%	#13:	29,51	46%						
#11:	22,80	98%	#17:	29,57	4%						
#13:	13,82	100%	#19:	33,16	2%						
#14:	24,52	100%	#20:	37,03	54%						
#15:	23,88	100%	#24:	39,26	2%						
#16:	27,17	84%	#27:	40,78	40%						
#17:	21,02	94%	#29:	42,98	70%						
#18:	25,73	46%	#30:	39,7	2%						
#19:	29,51	86%									
#20:	26,46	98%									
#25:	24,97	60%									
#27:	31,37	2%									
#29:	27,60	92%									
#31:	37,03	6%									
#34:	33,99	4%									
#35:	32,09	6%									
#36:	34,39	28%									
#37:	37,89	38%									
#38:	39,94	70%									
#41:	33,72	56%									
#46:	44,56	2%									
CAVER tunnel count:												
46			35			27			5		

Table 5.3: Results for the 1AKD molecule, 2 500 000 iterations, 50 runs. Each column contains the CAVER*'s tunnel index, the length of the tunnel (in Ångstroms) and the probability of being found by the TOM-RRT algorithm.*

caused by the extremely high number of rather straight protein tunnels located in the *1AKD* molecule (see *Figure 5.15*). This conjecture is supported by *Figure 5.17*, which shows an excellent performance improvement over the results for the *1CQW* molecule (*Figure 5.14*).

Results for *Figures 5.16 and 5.17* coincide with the results for the previous molecule and also with the physical characteristics of the molecule (smaller probe radius results in lower overall tunnel count). A noteworthy observation is the fact that performance in the *1AKD* molecule is faster than in the *1CQW* molecule despite the *1CQW* consisting from more atoms than *1AKD*.

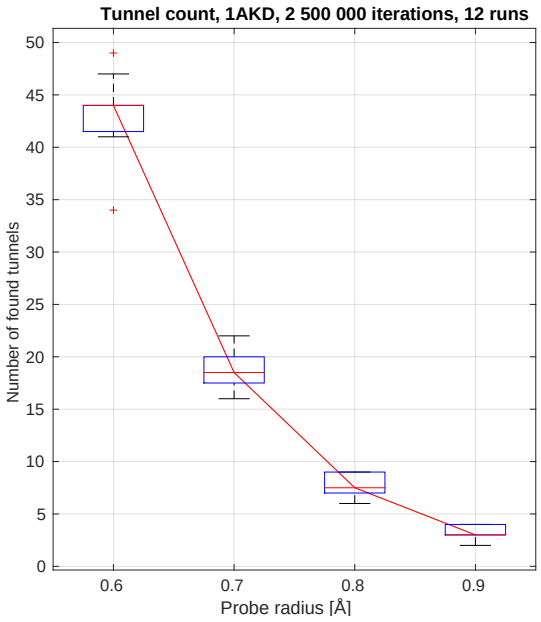


Figure 5.16: 1AKD — Number of found tunnels

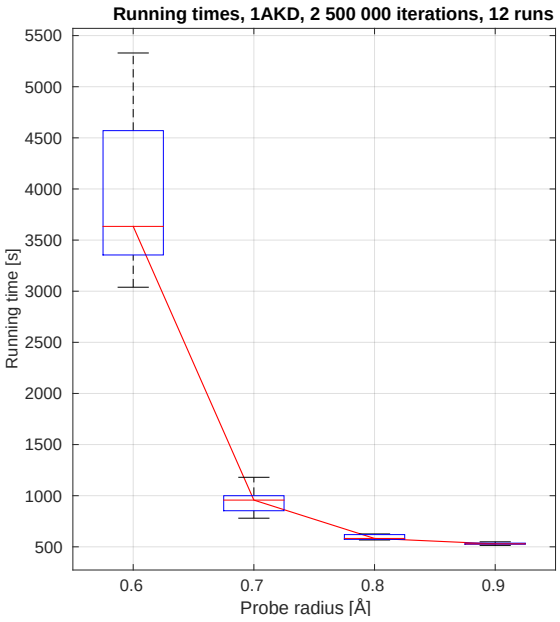


Figure 5.17: 1AKD — Running times

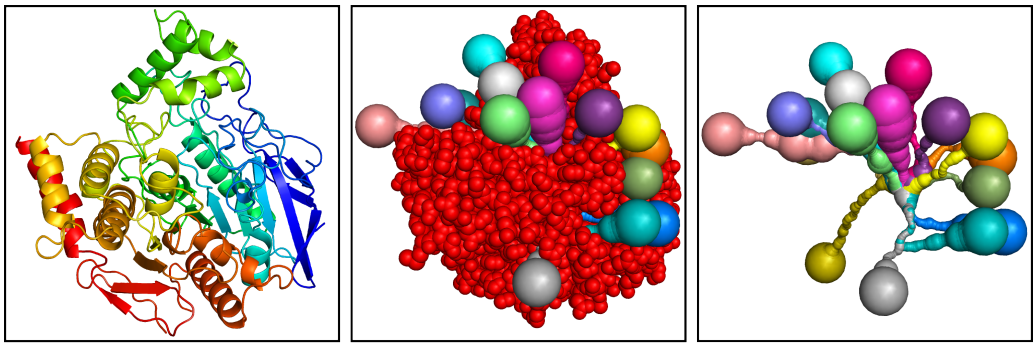
5.3.3 2ACE

The structure of the *2ACE* protein molecule is depicted in *Figure 5.18* along with an example of the set of the found tunnels. The resulting performance of the TOM-RRT algorithm for the *2ACE* molecule is listed in *Table 5.4*. The test was performed on *Computer 1*.

Once again, the results are as expected, with the curves connecting the median values in *Figures 5.19 and 5.20* following trends similar to those in previous molecules.

5.3.4 1MXTa

The structure of the *1MXTa* protein molecule is depicted in *Figure 5.21* along with an example of the set of the found tunnels. The resulting performance of the TOM-RRT algorithm for the *1MXTa* molecule is listed in *Table 5.5*. The test was performed on *Computer 1*.



a) b) c)

Figure 5.18: Details of the 2ACE molecule. a) *shows the cartoon representation,* b) *shows the spherical representation with the found tunnels and* c) *shows an example of the found tunnels.*

2ACE (4144 atoms, 1 000 000 iterations, 50 runs) Probe radius:												
0.6Å			0.7Å			0.8Å			0.9Å		
#1:	5,95	100%	#1:	5,95	98%	#1:	5,95	100%	#1:	5,95	96%
#2:	9,57	100%	#2:	9,57	100%	#2:	9,57	100%	#2:	9,57	82%
#3:	8,27	100%	#4:	18,96	6%	#3:	18,96	8%	#3:	18,96	60%
#4:	13,48	82%	#5:	18,06	2%						
#7:	18,06	6%	#8:	19,88	4%						
#8:	15,53	80%									
#9:	16,32	56%									
#10:	19,88	6%									
#11:	24,66	14%									
#28:	32,1	2%									
#31:	39,54	2%									
CAVER tunnel count:											
53			41			7			5		

Table 5.4: Results for the 2ACE molecule, 1 000 000 iterations, 50 runs. Each column contains the CAVER*'s tunnel index, the length of the tunnel (in Ångstroms) and the probability of being found by the TOM-RRT algorithm.*

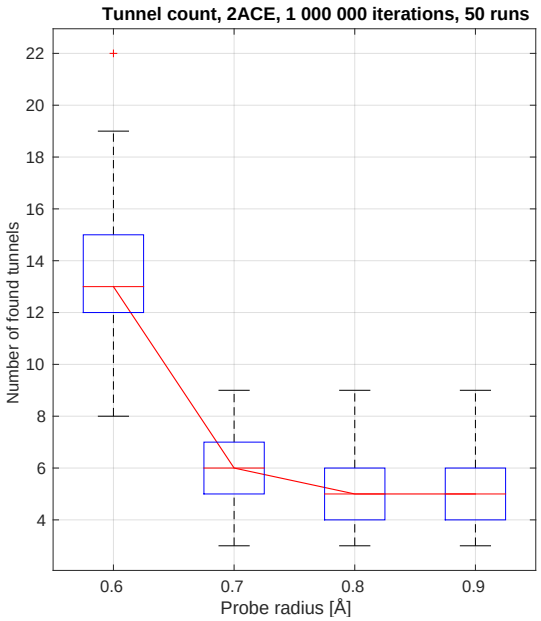


Figure 5.19: 2ACE — Number of found tunnels

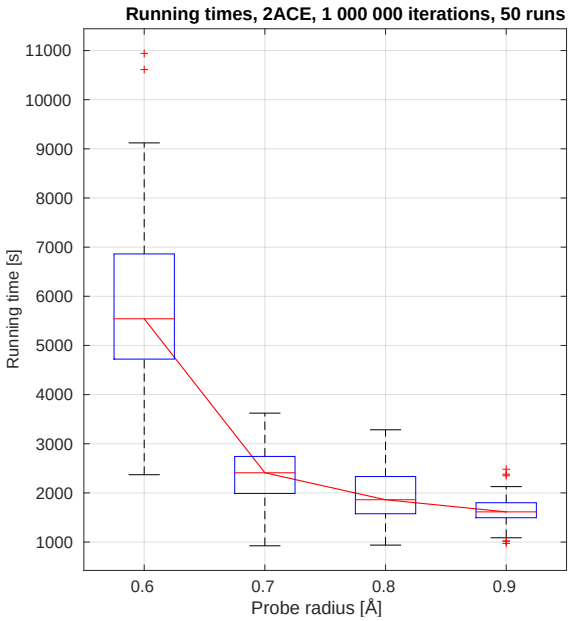
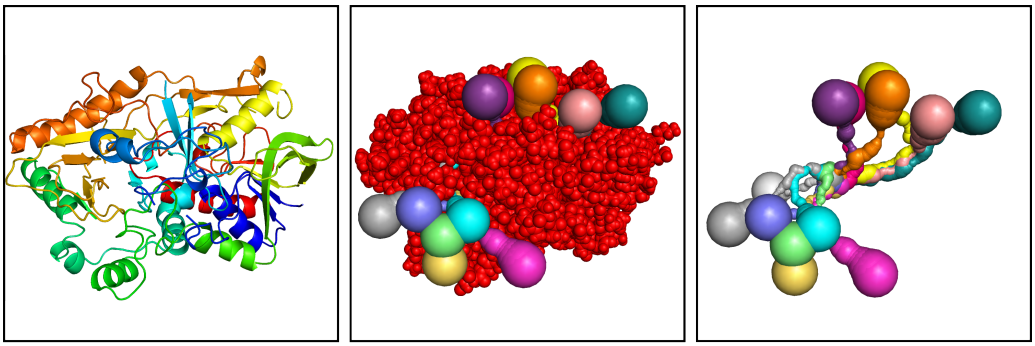


Figure 5.20: 2ACE — Running times



a) b) c)

Figure 5.21: Details of the 1MXTa molecule. a) *shows the cartoon representation,* b) *shows the spherical representation with the found tunnels and* c) *shows an example of the found tunnels.*

1MXTa (7519 atoms, 1 000 000 iterations, 50 runs) Probe radius:												
0.6Å			0.7Å			0.8Å			0.9Å		
#1:	15,14	100%	#1:	15,14	100%	#1:	15,14	100%	#1:	15,14	100%
#2:	14,79	100%	#2:	14,79	94%	#2:	14,79	76%	#2:	14,79	54%
#3:	33,60	100%	#3:	33,60	100%	#3:	33,60	100%	#:3	33,60	100%
#4:	20,85	4%	#5:	24,05	44%	#5:	24,04	22%			
#5:	24,04	78%									
#12:	38,90	6%									
#17:	29,29	4%									
#23:	45,24	2%									
CAVER tunnel count:											
27			8			7			4		

Table 5.5: Results for the 1MXTa molecule, 1 000 000 iterations, 50 runs. Each column contains the CAVER*'s tunnel index, the length of the tunnel (in Ångstroms) and the probability of being found by the TOM-RRT algorithm.*

Figure 5.23 once again reflects the physical properties of the molecule. Note the fact the the number of tunnels significantly decreases during the step from probe size $0,6$ to $0,7$ in the *CAVER* results as well as in the TOM-RRT results. The logical conclusion would be that this was caused by the molecular structure.

The *1MXTa* protein at 7519 atoms is the largest tested molecule. However, its size hasn't impacted the overall performance significantly, since it is on par with, or better than, performances in scenarios with molecules and similar tunnel counts (*2ACE*), as displayed in *Figure 5.23*. The conclusion one could draw from this is that the algorithm isn't hindered much by the protein's number of atoms but rather by the molecular structure and therefore the complexity of the environment the Rapidly-exploring Random Trees have to grow in.

5.3.5 1BL8

The structure of the *1BL8* protein molecule is depicted in *Figure 5.24* along with an example of the set of the found tunnels. The resulting performance of the TOM-RRT algorithm for the *1BL8* molecule is listed in *Table 5.6*. The test was performed on *Computer 1*.

Similarly to the *1AKD* molecule, a rather large number of tunnels have probabilities of being found higher than 50% (13 for probe size $0,6$ to be exact, with 7 tunnels being found every time the algorithm was run). Upon examining the example of found tunnels provided in *Figure 5.24* we once again see that the majority on them follows an almost straight line from the search origin point to the outside of the molecule. This fact allows the Rapidly-exploring Random Trees to grow more quickly, thus resulting in the observed characteristic.

Figure 5.25 exhibits behavior which hasn't been observer in any other tested molecule — the number of found tunnels doesn't follow the same type of curve seen in other molecules (a deformed hyperbolical curve). The number of found tunnels drops

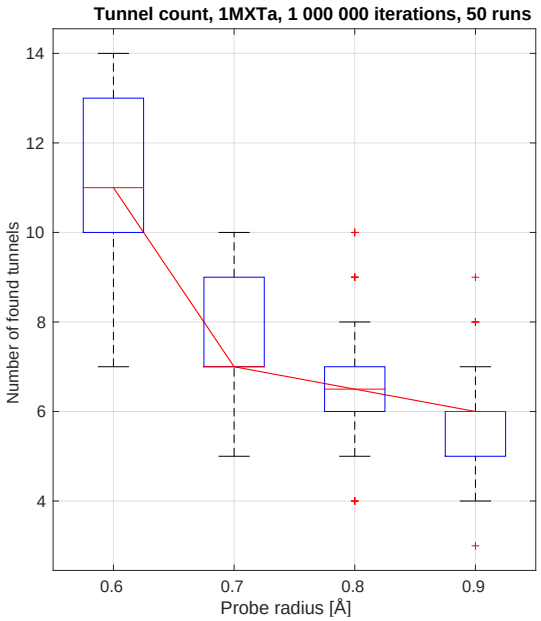


Figure 5.22: 1MXTa — Number of found tunnels

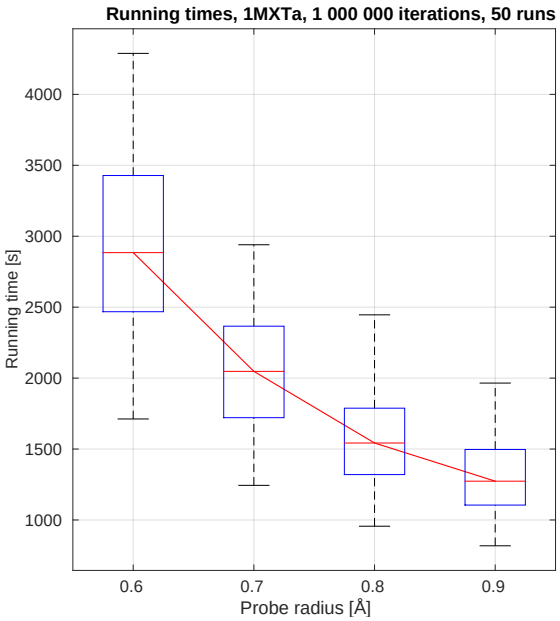
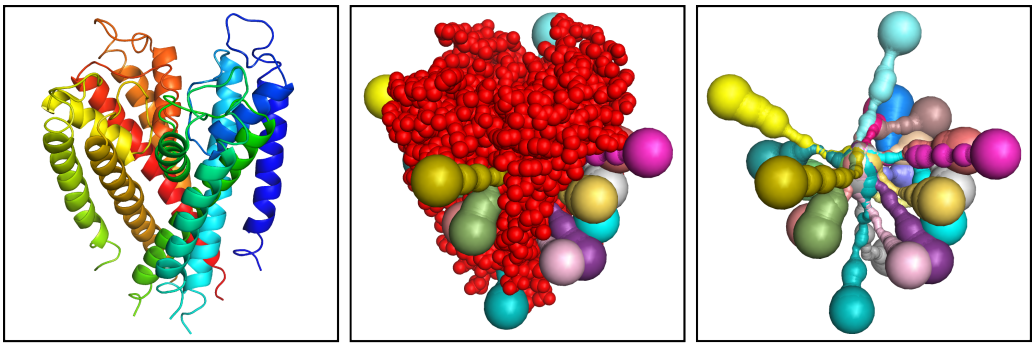


Figure 5.23: 1MXTa — Running times



a) b) c)

Figure 5.24: Details of the 1BL8 molecule. a) *shows the cartoon representation,* b) *shows the spherical representation with the found tunnels and* c) *shows an example of the found tunnels.*

1BL8 (2823 atoms, 1 000 000 iterations, 50 runs) Probe radius:												
0.6Å			0.7Å			0.8Å			0.9Å		
#1:	16,16	94%	#1:	16,16	98%	#1:	16,16	96%	#1:	16,16	98%
#2:	14,97	100%	#2:	14,97	98%	#2:	14,97	100%	#2:	14,97	98%
#3:	16	98%	#3:	16	100%	#3:	16	100%	#3:	16,00	100%
#4:	15,23	100%	#4:	15,23	100%	#4:	15,23	96%	#4:	15,23	100%
#5:	13,02	100%	#5:	13,02	100%	#5:	13,02	100%	#5:	13,02	10%
#6:	12,16	100%	#6:	12,16	100%	#6:	12,16	100%	#6:	12,27	18%
#7:	12,31	100%	#7:	12,31	100%	#7:	12,31	100%	#7:	12,31	8%
#8:	12,48	100%	#8:	12,48	100%	#8:	12,48	100%	#8:	12,48	18%
#10:	26,08	62%	#10:	26,08	74%	#10:	26,08	54%	#10:	26,08	54%
#13:	15,22	34%	#12:	15,22	8%	#18:	33,13	2%			
#14:	14,34	58%	#14:	14,34	8%						
#15:	15,19	62%	#15:	15,19	2%						
#16:	14,53	100%	#16:	21,07	4%						
#21:	26,1	2%	#19:	25,61	2%						
#26:	26,34	86%									
CAVER tunnel count:											
34			34			21			20		

Table 5.6: Results for the 1BL8 molecule, 1 000 000 iterations, 50 runs. Each column contains the CAVER*'s tunnel index, the length of the tunnel (in Ångstroms) and the probability of being found by the TOM-RRT algorithm.*

significantly with probe size equal to $0,9$. This fact can most likely be attributed to the stochastic characteristic of the TOM-RRT algorithm, which failed to discover tunnels lacking enough space to allow for the tree's growth steps to be performed and can therefore be attributed to the molecular structure.

Furthermore, *Figure 5.26* reveals that the algorithm's overall performance was by far the worst among all of the tested molecules, despite the protein's small size at 2823 atoms. Once again, this can be attributed to the large amount of found tunnels which resulted in a large number of disabled areas and required extensive checking by the algorithm.

5.3.6 Connection between the number of iterations and the number of found tunnels

Upon analyzing the previous data, several questions present themselves: Does the number of iterations (the length of the calculation) relate to the number of found tunnels? Does increasing the number of iterations yield more tunnels? Is the quality (and length) of tunnels improved after increasing the number of iterations?

The calculations were run with an increasing number of iterations. The results are provided below. The test was run on *Computer 2*, which resulted in running times in *Figure 5.28* differing from the rest of the tests.

The number of found tunnels seems to stabilize itself in a linear trend in *Figure 5.27*

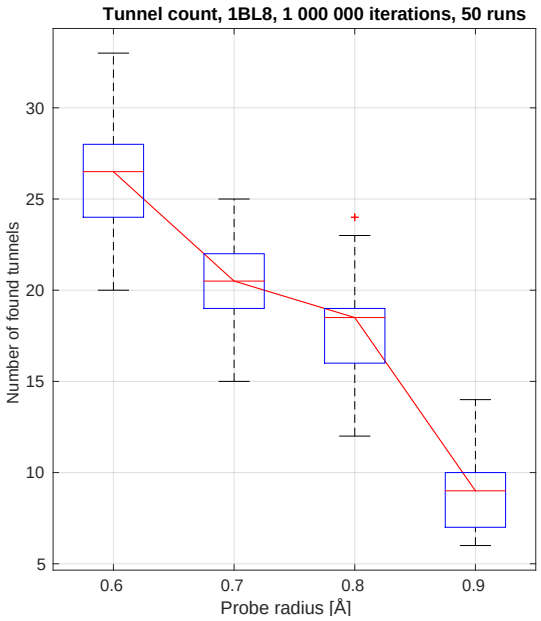


Figure 5.25: 1BL8 — Number of found tunnels

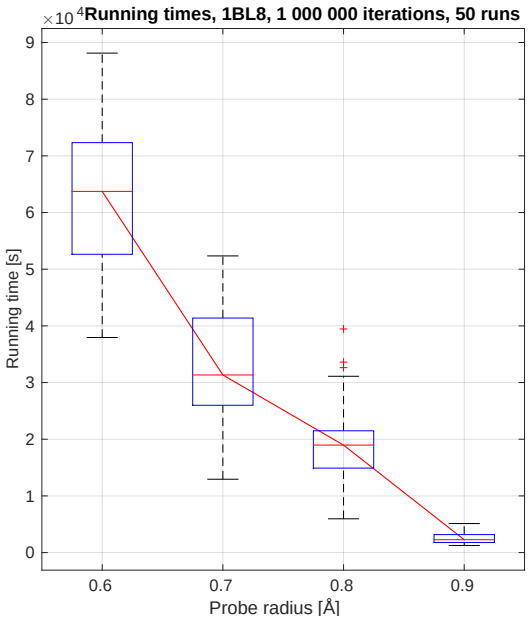


Figure 5.26: 1BL8 — Running times

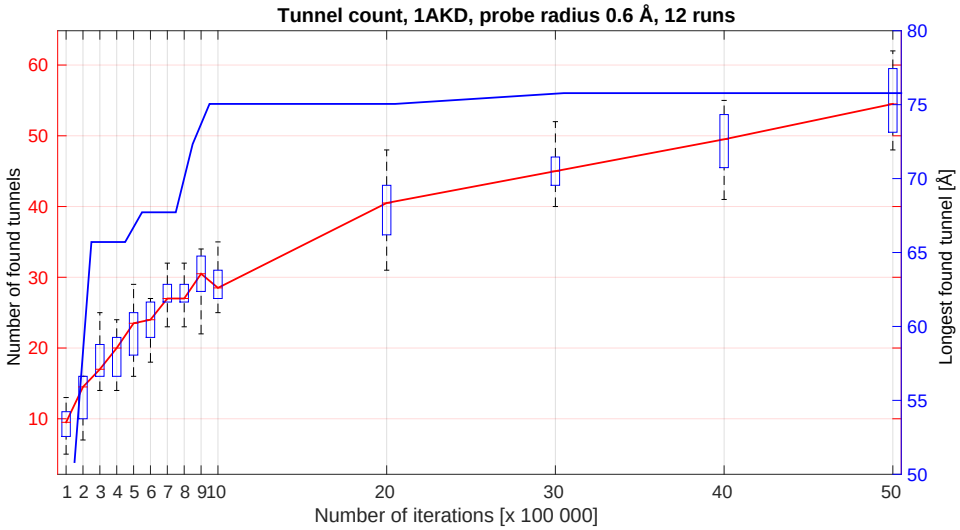


Figure 5.27: Number of found tunnels (red) and length of the longest tunnel found so far (blue) plotted against an increasing number of iterations. Please note that the data for this graph were collected on Computer 2

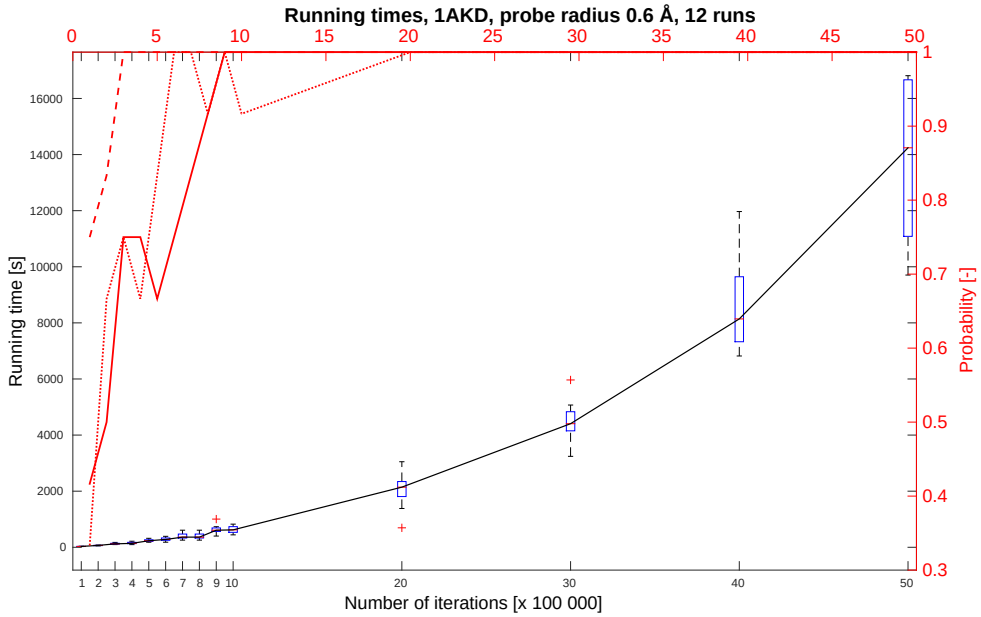


Figure 5.28: Running times (black) and the probability of finding the first 3 tunnels found by CAVER *(dotted, dashed and solid red lines) plotted against an increasing number of iterations. Please note that the data for this graph were collected on* Computer 2

— increasing the number of iterations could potentially yield an increase in the number of found tunnels (up to the saturation point where all the potential tunnels would be found). However, the required computation times in *Figure 5.28* increase with significant speed, the length of the longest found tunnel stagnates and the probabilities of finding the first 3 tunnels rated by *CAVER* as the best ones saturate long before reaching the highest tested number of iterations (at two million iterations).

5.3.7 Discussion of the results

The collected results of testing the algorithm for 5 different protein molecules have indicated that the algorithm's computational capabilities are worse than or nearing the level of current state-of-the-art tool *CAVER*. This applies mainly to situations with a low number of available tunnels, where the number of found tunnels was close to that of *CAVER*.

The results have also indicated that while the quantity of the found tunnels does increase with more time allowed for the search, the quality of the tunnels saturates and this method of search is therefore ineffective.

The algorithm has demonstrated the ability to search for protein tunnels utilizing non-spherical ligand molecules, a feature, which is lacking *CAVER* and the majority of currently established software solutions.

However, situations with a high amount of eligible tunnels have revealed that the TOM-RRT algorithm lacks in terms of filtering out the already found tunnels. While the duplicate tunnels were discarded, they were being rediscovered many times over. This

fact could signify a possible necessity to improve disabled area placement. Despite the fact that the current method does work, it either moves the disabled areas too slowly or fails to fill all the gaps in the tunnels, allowing the Rapidly-exploring Random Trees to grow around them. This area could present an excellent topic for further development, either to modify the placement of disabled areas or to improve the method of restricting tree growth by utilizing an entirely different concept.

Furthermore the algorithm lacks in terms of overall performance where the median time required to search an amount of tunnels nearing that of *CAVER* was over thirty times higher than the time required by *CAVER* (molecule *1BL8* for probe size $0,6$). This is most likely caused by the algorithm's stochastic nature, which doesn't guarantee the algorithm's exhaustiveness. This area could also present an excellent possibility for future improvements.

6 Conclusion

The problem of tunnel detection in protein molecules consists of detecting and analyzing tunnels in protein molecules on order to determine which ones might be relevant when considering chemical reactions with a given ligand molecule. It has numerous applications in e.g. pharmaceutical industry or protein engineering, as described in *Chapter 1*.

The majority of current methods for protein molecule analysis has a number of drawbacks, which can result in for example reporting data deviating from the actual values or the inability to perform some tasks (such as the planning for non-spherical ligand molecules). This thesis has therefore researched the feasibility of an application of sampling-based motion planning methods to protein molecule tunnel detection in *Chapter 2*.

The standard algorithms originating in the area of sampling-based motion planning are not directly usable for the task of protein tunnel detection because for example the inability to plan to a region of space located on the protein surface, rather than a fixed point in space, as is commonly the case and the inability to limit the Rapidly-exploring Random Tree's growth into certain areas of space, resulting in an inability to detect multiple protein tunnels.

We have therefore had to conceive and implement modifications, summarized in the resulting TOM-RRT algorithm, which allow the previously standard sampling-based motion planning algorithm to analyze the protein and ligand molecules, choose a fitting sampling region, efficiently and quickly grow the Rapidly-exploring Random Tree inside the molecule, detect when the tree has reached the surface of the molecule, optimize the resulting path for protein tunnel analysis and finally to analyze the resulting protein tunnel.

We have conceived a concept of *disabled areas* to restrict the tree's growth inside already detected tunnels in the protein molecules, thus enabling multiple tunnel detection and we have designed and implemented a metric to help us detect when a duplicate of a previously tunnel was found, as described in *Chapter 3*.

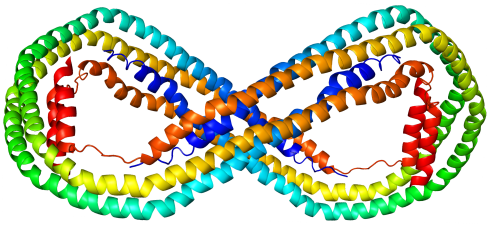
We have performed experiments to rate and pick the best performing external library to use in collision detecting functions of the algorithm. The resulting TOM-RRT algorithm's performance was tested on 5 protein molecules with sizes ranging from 2352 to 7519 atoms. The collected results have indicated that the detection capabilities of the algorithm are worse than or equal to those of the currently available state-of-the-art tool, *CAVER* [14], with the algorithm performing best in situations with low count

of protein tunnels. The algorithm's overall speed was however significantly slower than that of the *CAVER*, especially in situations with a high number of protein tunnels, as described in *Chapter 5*.

Furthermore, the algorithm was shown to be able to plan motions (and therefore detect protein tunnels) for complex non-spherical ligand molecules, an ability which isn't currently available in a majority of the already established tools, as described in *Chapter 4*.

Future research possibilities include improving the algorithm's performance, both in the terms of overall speed and the ability to detect all eligible protein tunnels, as discussed in *Chapter 5*. Another research topic could be the conception of an alternate solution to restrict the Rapidly-exploring Random Tree's growth in order to see if a solution able to surpass the performance of *disabled areas* can be conceived.

To summarize the previous statements, this thesis has presented a novel approach to protein tunnel detection and analysis. It has explored the topic from a computational standpoint, utilized well established algorithms originating in the field of robotics and presented a working solution, which upon a comparison with state-of-the-art tools, shows promising results. While it still falls behind in terms of overall speed and the number of delivered results, it could, with further development, serve as an alternative or a complement to the already existing solutions.



Bibliography

[1] Bullet Physics Library: A physics engine for soft and rigid body dynamics simulation. http://bulletphysics.org/. Accessed: 2015-11-15.

[2] RRT graph1 - Rapidly exploring random tree from Wikipedia the free encyclopedia. https://en.wikipedia.org/wiki/Rapidly_exploring_random_tree#/media/File:RRT_graph1.png. Accessed: 2016-03-05.

[3] Thyrotropin-releasing hormone. from Wikipedia the free encyclopedia. http://en.wikipedia.org/wiki/Thyrotropin-releasing_hormone. Accessed: 2016-02-28.

[4] Titin. from Wikipedia the free encyclopedia. https://en.wikipedia.org/wiki/Titin. Accessed: 2016-02-28.

[5] Voronoi diagram variety from kaiseh blog. http://d.hatena.ne.jp/kaiseh/20091010. Accessed: 2016-03-04.

[6] Yershova Anna and LaValle Steven M. MPNN: Nearest neighbor library for motion planning, 2006.

[7] Berglund, Camilla and Geelnard, Marcus. GLFW - An OpenGL library.

[8] Masood Talha Bin, Sandhya Sankaran, Chandra Nagasuma, and Natarajan Vijay. CHEXVIS: a tool for molecular channel extraction and visualization. *BMC bioinformatics*, 16(1):1, 2015.

[9] Juan Cortés, Duc Thanh Le, Romain Iehl, and Thierry Siméon. Simulating ligand-induced conformational changes in proteins using a mechanical disassembly method. *Physical Chemistry Chemical Physics*, 12(29):8268–8276, 2010.

[10] Warren L. DeLano. The PyMOL molecular graphics system. 2002.

[11] Didier Devaurs, Léa Bouard, Marc Vaisset, Christophe Zanon, Ibrahim Al-Bluwi, Romain Iehl, Thierry Siméon, and Juan Cortés. MoMA-LigPath: a web server to simulate protein–ligand unbinding. *Nucleic acids research*, page gkt380, 2013.

[12] Kavraki Lydia E, Švestka Petr, Latombe Jean-Claude, and Overmars Mark H. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *Robotics and Automation, IEEE Transactions on*, 12(4):566–580, 1996.

[13] Yaffe Eitan, Fishelovitch Dan, Wolfson Haim J., Halperin Dan, and Nussinov Ruth. MolAxis: efficient and accurate identification of channels in macromolecules. *Proteins: Structure, Function, and Bioinformatics*, 73(1):72–86, 2008.

[14] Petr Beneš Ondřej Strnad Jan Brezovský Barbora KozlíkovÁ Artur Gora Vilém Šustr Martin Klvaňa Petr Medek Lada Biedermannová Jiří Sochor Eva Chovancová, Antonín Pavelka and Jiří Damborský. CAVER 3.0: A Tool for the Analysis of Transport Pathways in Dynamic Protein Structures. *Pathways in Dynamic Protein Structures, PLoS Computational Biology 8: e1002708*, 2012.

[15] G-Truc Creation. OpenGL Mathematics.

[16] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., and Stein Clifford. Section 24.3: Dijkstra's algorithm. introduction to algorithms (second ed.). 29(7):595–601, 2001.

[17] S. Hoher, P.Neher, and Alexander Verl. Analysis and evaluation of collision detection libraries for collision monitoring of multi-channel control applications. 2013.

[18] Bingding Huang and Michael Schroeder. LIGSITE csc: predicting ligand binding sites using the connolly surface and degree of conservation. *BMC structural biology*, 6(1):1, 2006.

[19] Březovský Jan, Chovancová Eva, Gora Artur, Pavelka Antonín, Biedermannová Lada, and Damborský Jiří. Software tools for identification, visualization and analysis of protein tunnels and channels. *Biotechnology advances*, 31(1):38–49, 2013.

[20] Latombe Jean-Claude. Motion planning: A journey of robots, molecules, digital actors, and other artifacts. *The International Journal of Robotics Research*, 18(11):1119–1128, 1999.

[21] Ho Bosco K and Gruswitz Franz. HOLLOW: generating accurate representations of channel and interior surfaces in molecular structures. *BMC structural biology*, 8(1):49, 2008.

[22] Barbora Kozlíková, Filip Andres, and Jiří Sochor. Visualization of tunnels in protein molecules. In *Proceedings of the 5th international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*, pages 111–118. ACM, 2007.

[23] Igor Kravtchenko. Ozcollide: Advanced and fast collision detection library. `http://www.tsarevitch.org/ozcollide/`. Accessed: 2015-11-15.

[24] David G Levitt and Leonard J Banaszak. Pocket: a computer graphies method for identifying and displaying protein cavities and their surrounding amino acids. *Journal of molecular graphics*, 10(4):229–234, 1992.

[25] LaValle Steven M. and Kuffner Jr. James J. Rapidly-exploring random trees: Progress and prospects. 2000.

[26] Petřek Martin, Košinová Pavlína, Koča Jaroslav, and Otyepka Michal. MOLE: a voronoi diagram-based explorer of molecular channels, pores, and tunnels. *Structure*, 15(11):1357–1363, 2007.

[27] Raunest Martin and Kandt Christian. dxTuber: Detecting protein cavities, tunnels and clefts based on protein and solvent dynamics. *Journal of Molecular Graphics and Modelling*, 29(7):895–905, 2011.

[28] Voss Neil R. and Gerstein Mark. 3V: cavity, channel and cleft volume calculator and extractor. *Nucleic acids research*, 38(suppl 2):W555–W562, 2010.

[29] Gottschalk Stefan, Lin Ming C, and Manocha Dinesh. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 171–180. ACM, 1996.

[30] Stewart, Nigel and Ikits, Milan and Magallon, Marcelo. GLEW: The OpenGL Extension Wrangler Library.

[31] Le Guilloux Vincent, Schmidtke Peter, and Tuffery Pierre. Fpocket: an open source platform for ligand pocket detection. *BMC bioinformatics*, 10(1):168, 2009.

[32] Vojtěch Vonásek, Jan Faigl, Tomáš Krajník, and Libor Přeučil. RRT-Path–a guided rapidly exploring random tree. In *Robot Motion and Control 2009*, pages 307–316. Springer, 2009.

[33] Martin Weisel, Ewgenij Proschak, Gisbert Schneider, et al. PocketPicker: analysis of ligand binding-sites with shape descriptors. *Chem Cent J*, 1(7):1–17, 2007.

[34] Weisstein Eric W. Delaunay Triangulation. from MathWorld–a Wolfram Web Resource.

[35] Weisstein Eric W. Quaternion. from MathWorld–a Wolfram Web Resource.

[36] Weisstein Eric W. Voronoi Diagram. from MathWorld–a Wolfram Web Resource.

[37] Worldwide Protein Data Bank. wwPDB: File Format.

[38] Worldwide Protein Data Bank. wwPDB: File Format version 3.3: Coordinate Section.

6.1 Addendum 1: Contents of the attached disc

The attached disc contains:

- A generated pdf file with this thesis.
- Folder *Inputs* with the tested protein molecules, Cartesian coordinates of the origin points for the Rapidly-exploring Random Tree and a file with the used probe.
- Folder *Program* with source code of an implementation of the PRM algorithm, the RRT-Path algorithm and the resulting TOM-RRT algorithm, along with a template used to test collision detection libraries and a program utilized when comparing the detected protein tunnels with those found by *CAVER*. The folder also contains some of the utilized external libraries, such as the ANN library, the MPNN library, and the OZCollide library.
- Folders *Results_1* and *Results_2* with the generated data and a movie sequence planned for a non-spherical ligand molecule. The directory structure is as follows: <name-of-the-molecule>/<number-of-iterations>/<probe-radius>. The folder *ME* in the last directory contains numbered folders with tunnels generated by the TOM-RRT algorithm. The directory *CAVER* (which is not present in all cases) contains tunnels generated by *CAVER*.

6.1.1 Installation guide

The software was compiled using CMake 3.5.1, Make 4.1 and g++ 4.8.4 for Ubuntu 14.04. The compilation requires a number of external libraries to be installed. While we have tested the program with these specific versions, other versions may work as well:

- Boost C++ library version 1.58.0
- GLM library version 0.9.7.2-1
- GLEW library version 1.13
- FreeGLUT library version 3.0
- GLFW library version 3.0
- Eigen library version 3.0

Follow these steps to compile the program using terminal in the extracted folder on a Linux OS distribution:

```
mkdir build
cd build
cmake ../.
make
```

Figure 6.1: Compilation commands

6.2 Addendum 2: Further implementation details

6.2.1 PDB file parsing

The x-ray crystallography data, used as a reference for positions and types of atoms in the molecules, is provided in a text-based file format "Protein Data Bank" (*pdg.org*). This file contains a number of fixed-width columns and can easily be parsed, using a manual provided for the file format [37]. One section particularly interesting for our purposes is the detailed description of "ATOM" records, which provides all the required details for parsing molecular geometric data:

COLUMNS	DATA TYPE	FIELD	DEFINITION
1 - 6	Record name	"ATOM "	
7 - 11	Integer	serial	Atom serial number.
13 - 16	Atom	name	Atom name.
17	Character	altLoc	Alternate location indicator.
18 - 20	Residue name	resName	Residue name.
22	Character	chainID	Chain identifier.
23 - 26	Integer	resSeq	Residue sequence number.
27	AChar	iCode	Code for insertion of residues.
31 - 38	Real(8.3)	x	Orthogonal coordinates for X in Angstroms.
39 - 46	Real(8.3)	y	Orthogonal coordinates for Y in Angstroms.
47 - 54	Real(8.3)	z	Orthogonal coordinates for Z in Angstroms.
55 - 60	Real(6.2)	occupancy	Occupancy.
61 - 66	Real(6.2)	tempFactor	Temperature factor.
77 - 78	LString(2)	element	Element symbol, right-justified.
79 - 80	LString(2)	charge	Charge on the atom.

Figure 6.2: Details of the "ATOM" record format. Courtesy of Worldwide Protein Data Bank [38]

The final implementation is provided in class "*PDBParser*", located in file "*pdb_parser.cpp*".

6.2.2 Visualization

To provide a convenient and practical way of manual problem analysis, debugging and determining the initial and final positions of the molecules, a simple visualization tool was developed. Utilizing OpenGL and a variety of external libraries (GLM [15], GLEW [30] and GLFW [7]) a tool which, when provided with parsed atomic positions and type allows the user to manually pick initial and final positions of the ligand molecule and also allows for the display of the found trajectory. The final implementation is provided in class "*ProteinVisualizer* in file "*protein_visualization.cpp*" and an example of the visualization is depicted in *Figure 6.3*.

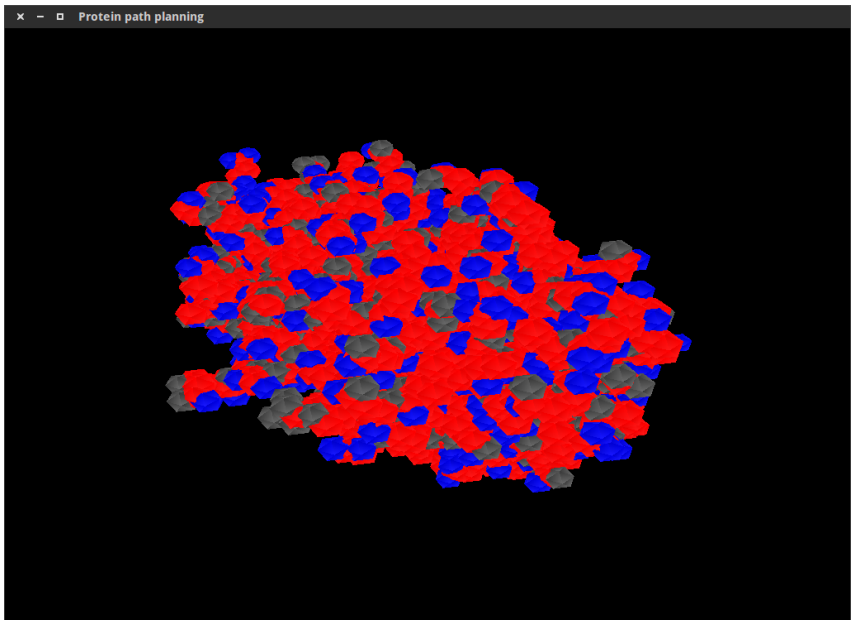


Figure 6.3: An example of the created molecule visualization.

6.2.3 Pymol exporting

As previously mentioned, *Pymol*, an existing tool for biochemical visualizations, mainly focused on protein molecules, exists. It is capable of visualizing both static and dynamic molecules, can switch between their different representations and much more. As it is arguably the currently most used tool, a feature which allows for molecules, their trajectories and the found protein tunnels to be exported to Pymol has been developed. The final implementation is provided in the class "*TrajectoryExporter*" in the file "*protein_trajectory_export.cpp*".

6.3 Addendum 3: Utilized Pymol commands

showspheres — turns on spherical atom representation
frame — switches to frame *n*
mviewstore — saves the camera position to the current animation sequence
mviewreinterpolate — interpolates saved camera positions in the current animation sequence
cmd.get_object_matrix(**object**, **state**) — return transformation matrix of the camera
setView() — set the transformation matrix of the camera
util.color_objs("**all**") -color each object with a different color
fetch — downloads the protein file and builds a biological unit from a protein fragment
png — saves an image file