DEGREE PROJECT, IN COMPUTER SCIENCE, FIRST LEVEL
STOCKHOLM, SWEDEN 2016

A comparison study of Kd-tree, Vp-tree and Octree for storing neuronal morphology data with respect to performance

MARCUS ADAMSSON
ALEKSANDAR VORKAPIC

KTH Computer Science
and Communication

A comparison study of Kd-tree, Vp-tree and Octree for storing neuronal morphology data with respect to performance

MARCUS ADAMSSON
ALEKSANDAR VORKAPIC

Degree Project in Computer Science, DD143X
Supervisor: Alexander Kozlov
Examiner: Örjan Ekeberg

CSC, KTH 2016-05-11

Abstract

In this thesis we investigated performance of Kd-tree,Vp-tree and Octree for storing neuronal morphology data. Two naive list structures were implemented to compare with the space-partition data structures. The performance was measured with different sizes of neuronal networks and different types of test cases. A comparison with focus on cache misses, average search time and memory usage was made. Furthermore, measurements gathered quantitative data about each data structure. The results showed significant difference in performance of each data structure. It was concluded that Vp-tree is more suitable for searches in smaller populations of neurons and for specific nodes in larger populations, while Kd-tree is better for volume searches in larger populations. Octree had highest average search time and memory requirement.

Sammanfattning

I denna rapport har vi undersökt prestanda av tre datastrukturer, Vp-tree, Kd-tree och Octree, för lagring av neurala morfologidata. Två naiva liststrukturer implementerades, för att kunna jämföras med tre datastrukturer. Prestanda mättes med olika storlekar av neurala nätverket och med olika typer av testfall. En jämförelse med fokus på cachemissar, genomsnittlig söktid och minnesanvändning utfördes. Dessutom, samlade mätningarna kvantitativ data om varje datastruktur. Resultatet visade signifikant skillnad i prestanda mellan de implementerade datastrukturerna. Det konstaterades att Vp-tree är bättre för sökning i mindre populationer av neuroner samt för sökning av specifika noder i större populationer, medan Kd-tree är bättre för volymsökning i större populationer. Octree hade högst medelsöktid och minnesanvändning.

Contents

1 Introduction

Intensive brain research during the last decade has been stimulated by introduction of large-scale, high-precision experimental methods of neural data acquisition. Which has resulted in development of tools for realistic brain simulations, in order to understand the brain functions and diseases [1]. Thus, new technologies are providing a flood of data about, e.g. genes, connection between cells and fiber tracts that connect different regions of the brain [1]. Exploiting this data is possible, by using supercomputer technology to integrate multi-level brain models [1].

A brain consist of up to 100 billion neurons, which are nerve cells that are responsible for reception and transmission of nerve impulses [2]. The amount of impulses per second in the brain is about 10^{16}, which is beyond the performance of modern supercomputers [3]. Furthermore, each neuron has around 1000 inputs that fires at an average rate of 10 Hz, resulting in 10^{15} connections per second, and each connection requires around 10 instructions [3].

Real-time simulation of human brain with reasonably realistic neuronal models, is not yet possible [3]. Terascale computers have already allowed EPFL's Blue Brain Project to make the leap from simulations of single neurons to cellular level simulations of neuronal microcircuits [1]. However, current supercomputers have limitations such as processor frequency, memory size, communication bandwidth, physical scale, energy consumption and hardware reliability [4]. Which constraints researchers to simulate a single neuron or small neural networks, in order to determine their functions in preparation for larger-scale simulations. Several studies in this area exist, such as reconstruction and simulation of neocortical microcircuitry [5], connectomic reconstruction of the inner plexiform layer in the mouse retina [6] and reconstructing the three-dimensional GABAergic microcircuit of the striatum [7].

A typical research workflow consist of three major phases, single-cell morphology reconstruction, building local network and simulation [8]. However, the software and data structures for some parts of the simulation workflow are missing [8]. This thesis will consider applicability of different spatial data structures for storing geometric data of neuronal morphology and compare their performance for diverse spatial queries.

Space-partitioning data structures are divided in two subgroups, space-driven and data-driven [9]. The most common space-driven data structure in three dimensional space, is Octree [9]. Space-partitioning structures such as Kd-tree and Vp-tree, are used for fast searches in higher dimensional spaces [10]. Previous research comparing Kd-tree and Vp-tree with other data structures, has concluded that Kd-tree and Vp-tree outperform other data structures in different search methods [10] [11]. However, research comparing Vp-tree, Kd-tree and Octree in three dimensional space was not found, and therefore we decided to investigate how these three data structures differ in performance while storing and retrieving neuronal morphology data.

1.1 Problem statement

The aim of this project is to compare three space-partitioning data structures, Octree, Vp-tree and Kd-tree with respect to performance. The performance of these data structures will be measured in terms of CPU (Central Processing Unit), cache misses and RAM (Random Access Memory) usage. These measurement will occur during typical scenarios of data-driven reconstruction of a neural network's connectivity. Thus, this thesis aims to investigate the following:

- How do Octree, Vp-tree and Kd-tree differ in performance while storing and retrieving neuronal morphology data?

1.2 Scope

This thesis will focus on three data structures, Vp-tree, Kd-tree and Octree, as it would not be possible to cover all sort of spatial data structures. Since efficient data structure is a vital part of the neuronal reconstruction, this thesis will primarily focus on required CPU time and amount of cache misses while searching for a specific node or volume. Furthermore, the required amount of RAM will be measured during the building of respective data structure. However, this thesis will perform all measurements on a personal computer and due to project's time frame not all measurements will be possible.

1.3 Terminology

Terms used in this thesis.

Population	- A set of neurons; a network of interconnected neurons
Node/Point	- Single measurement of a neuron's morphology, including spatial coordinates and other features
Morphology	- The study of organism's shape and structure
Space	- A dimensional volume, mostly three dimensional in this thesis
D/R value	- D is the half-length of a cube's side and R is the radius of a sphere
Program	- A executable code, same as software

2 Background

2.1 Neuronal morphology

A neuron is a nerve cell with tree-structure, consisting of dendrites, soma and axon, as shown in Figure 1*.* Dendrite is where the neuron receives connection from other neurons. The cell body, soma, contains the nucleus which maintains the nervous system and the cellular functions. Axon's purpose is to transmit information from soma to connected neurons, in shape of electrical impulses along the axon. The terminal region of the axon is called the synapse and it's where other neurons connect, as shown in Figure 1 [12] [13].

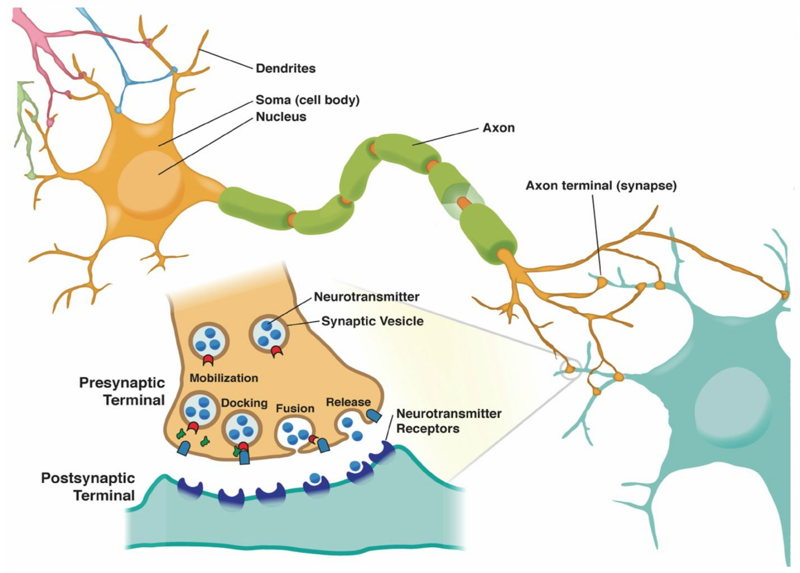

Figure 1: Structure of a biological neuron [2].

However, a neuron can connect up to 10,000 neighboring neurons. These connections are combined into a network, which supports the memory, emotions and reflexes. Only the human brain contains approximately 100 billion neurons which relay, process and store information [13].

Neurons are classified into four main groups based on their structure [14]. These groups are multipolar, unipolar, bipolar and pseudo-unipolar [15]. Neuronal morphologies can be classified based on the layer containing their somata, and their anatomical and electrical features. A brain is divided into six different cortical layers, which contain different types of neurons. Anatomical diversity of interneurons within cortical layer II/III is illustrated in Figure 2. Morphological type of neuron is related to its function and also affects connections of the cell within the local circuit [5].

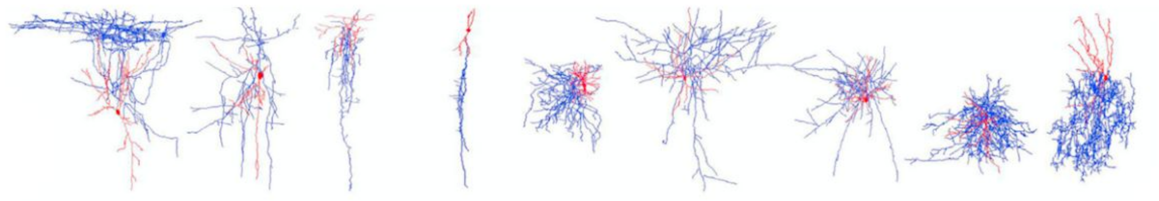

Figure 2: Interneurons in the cortical layer II/III [12].

2.2 Phases for reconstructing neuronal network

Reconstruction of a neuronal network, requires three major phases in a basic modeling process. The three major phases consists of making single cells, building a local network and performing a simulation using different tools [5]. This section is based on the latest study “Reconstruction and Simulation of Neocortical Microcircuitry” which is the most detailed presentation of the procedure and the largest reconstruction, published by Markram et al. [5].

2.2.1 Morphological diversity

This phase consists of several steps, where each step has its own purpose to increase and identify the morphological diversity of neurons [5]. The different steps of first phase includes reconstruction of morphologies, classification of morphologies, unraveling morphologies, repairing morphologies, cloning morphologies, object classification and validation of cloning and morphological structural analysis [5]. All parts deal with the single-cell models or populations of individual isolated cells [5]. Numerous morphometric software tools are currently available for academic use, e.g. NeuroM [16], btmorph [17], TREES toolbox [18], etc. However, NeuroM uses a list structure to store all nodes of a neuron [16]. Btmorph stores all nodes of the neuron in a tree shaped linked list [17] and TREES toolbox uses a two dimensional adjacency matrix to store connected nodes in a neuron [18].

2.2.2 Reconstructing microcircuit connectivity

The network building phase is computationally more intensive than the single-cell analysis since it involves spatial calculations among the large number of densely packed cells [5]. Software solutions are often tailored to particular tasks of limited size and does not scale well [12].

The second phase, consists of touch detection and touch filtering. The touch detection step is performed, after the placement of morphologies in the three dimensional space. Several software tools are used to generate structural circuit by detecting zones of geometric overlap called touches [5]. Furthermore, touch filtering is performed to analyze and process potential touches, according to biological and geometrical constraints [5].

2.2.3 Simulation

The reconstructed microcircuit is simulated using different software tools such as NEURON, GENESIS, BRIAN, IBM Neural Tissue Simulator and SPLIT [19] [20]. These software solutions endows computational neuroscience with tools. Thus, improvement of computational power would allow simulations of neuronal networks with high complexity, ranging from detailed biophysical models of single cells up to large-scale neural networks [19].

To enable simulation, analysis and visualization of neuronal network, researchers are relying on developing and integrating the network with various software tools. There is a large set of software tools, as mentioned in previous paragraph. While the development of such applications is increasing by dozens of contributors, researchers are required to create a comprehensive development environment [5].This contributes to deciding the right software for the right task, while analyzing computational power, accuracy, availability, features and efficiency of the software [19].

Compute intensive workflow phases, such as network building and simulation, requires advanced hardware performance [5]. Therefore, HPCs (High-Performance Computing) are used to execute these phases. HPCs are supercomputers with high-level computational capacity, which makes them fastest and most powerful currently available computers [5]. IBM Blue Gene/P and IBM Blue Gene/Q are examples of HPC computers and there exist plenty more [5].

2.3 Performance

The performance of data structures and algorithms are mainly measured in two ways, time and memory measurement [9]. Existing tools can measure execution time of a program, the percentage of miss rate in cache, memory usage, etc. [21].

As CPU cores become faster and more numerous, the limiting factor for execution of programs is memory access [21]. Since memory is the bottleneck of fast execution, modern CPUs have cache memory which is often called CPU memory [21]. Cache memory is faster than RAM, but smaller because it’s more expensive. Therefore, caches are mostly used to store frequently referenced instructions and the data that are used by the executing program [21]. Modern computers have different types and layers of cache, to describe how close and accessible the cache is to the CPU. Level 1 cache is small, but very fast and is usually embedded in the processor. However, the number of cache levels depends on the CPU and higher level caches have larger sizes and takes longer to access [22].

Since the cache memory can only store frequent instructions, RAM is used as main memory [21]. RAM is slower than cache, but can store data from several programs that are being executed. During recent years RAM has been improved in both speed and size. The size increase led to less swapping between main memory and secondary memory, which improves execution time of programs by less readings from the secondary memory [21].

However, measuring these major factors in performance is possible in different ways and by different tools. Execution time of a program can be measured by using unix command time [23] or by using available libraries in programming languages. In C++ CPU time can be measured using the the library <ctime> [24]. The time is measured by amount of clock cycles a CPU requires for executing a program, which can be converted into seconds [24].

Cache can be measured with valgrind in C/C++ [25] or with the command *pref stat* in linux terminal [23]. Furthermore, cachegrind is a valgrind option that simulates how a program interacts with the computer’s cache hierarchy. It simulates a computer with independent first-level instruction and data caches (I1 and D1), backed by a unified second-level cache (L2). Modern computers often have up to three or four levels of cache, but for these computers valgrind can detect these caches and simulate it as first-level and last-level caches [25].

RAM consumption for a program during execution can be measured with different tools. The most common tool is Valgrind in C/C++ [26]. Valgrind contains an option called Massif that measures memory consumption of the program being executed. Massif can measure the useful memory, detect memory leaks and indicate which part of the program that is responsible for allocating memory [26].

2.4 Space-partitioning data structures

Space-partitioning data structures tends to be divided into two subgroups, namely, space-driven and data-driven structures [9]. Space-driven structures partitions the embedded space containing the data points, while data-driven structures partitions the data points themselves [9].

2.4.1 Octree

One commonly used space-driven data structure for three dimensional space is Octree [9]. Octree is based on the principle of recursive decomposition and is constructed in the following manner [9]. The first step is to divide the image or object into eight congruent disjoint cubes which are called octants, as shown in Figure 3. Each octant is marked occupied or free according to whether there is any object occupying that location in the environment of the representation [9]. The process continues by dividing each of the eight octants into another level of eight octants and it continues until sufficient resolution has been achieved [9].

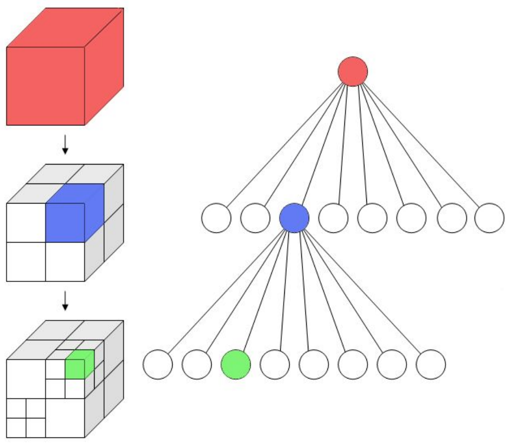

Figure 3: A demonstration of an Octree.

2.4.2 Vantage-point tree

Vantage-point Tree also called Vp-tree is a data-driven structure that supports multi dimension for storing spatial coordinates [11]. Vp-tree is constructed in the following manner. Consider a finite set S of N data nodes. A data object from the finite set is randomly chosen as vantage point of the tree. The green point in Figure 4, illustrates the vantage point of the tree. Let that point be the root of the tree called v. Let u be the the median distance from v to all other points in S [11]. The circle's radius is the median distance, as illustrated in Figure 4. All points within the circle, the blue points, will be in the left subtree and the points outside the circle, the red points, will be in the right subtree.

Left subtree = { s ∈ S | distance(s,v) < u}
Right subtree = { s ∈ S | distance(s,v) ≥ u}, where distance(a,b) is the distance between the points a and b in three dimensional space.
The same procedure is applied on left and and right subtree recursively [11].

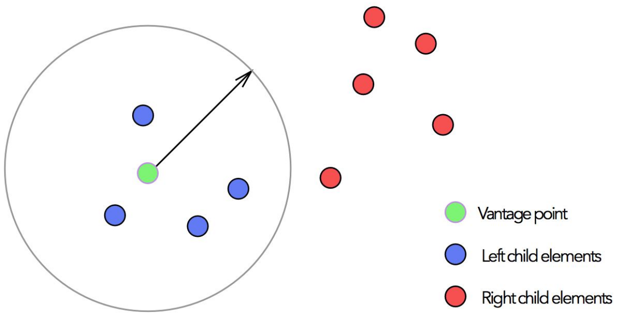

Figure 4: A demonstration of a Vp-tree in two dimensional space.

2.4.3 K-dimensional tree

Kd-tree is a data-driven structure, known as a k-dimensional binary tree where k is the size of the dimension [10]. Kd-tree can be used for range searches, nearest neighbor searches and space-partitioning [9]. A non-leaf node in the Kd-tree divides the space into two parts, called half-spaces. The two half-spaces represents a left and a right subtree [10]. In a two dimensional space, the Kd-tree's root would have a x-aligned plane, the root's children would have y-aligned planes, the root's grandchildren would have x-aligned planes and so on, as shown in Figure 5 [10].

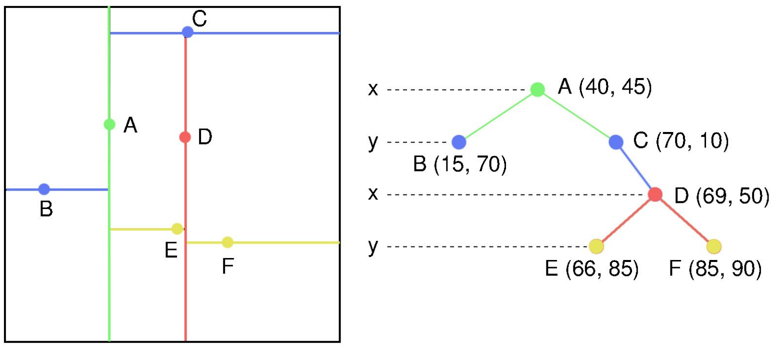

Figure 5: A demonstration of a Kd-tree in two dimensional space.

2.5 Related research

Previous research by Ashraf [10] compared Metric-trees, such as Vp-tree with Kd-trees. He compared how Metric-trees and Kd-trees differ in performance, while using nearest neighbor search in different dimensions. Kd-tree performed better in tests with lower dimensions such as d < 16, while Metric-tree performed better in higher dimensions. The different data structures were tested with a data size that varied between 1000 and 100k points [10]. Furthermore, Kd-tree was compared with Cover tree. Cover tree is a data structure specifically used for nearest neighbor search. Thus, Ashraf stated that Kd-tree was better than Cover tree for searches in lower dimension [10].

Juhong et al. [27] compared Octree with R-tree in three dimensional space with respect to runtime performance. The performance were tested with insertion, deletion and retrieval algorithms on the space-partitioning structures. They concluded that CPU performance of R-tree is better than Octree, because of the different data structures. R-tree is a data-driven structure were Octree is a space-driven structure was the main reason of the result [27].

Research made by Ada et al. [11] compared Vp-tree, R*-tree and M-tree with nearest neighbor search algorithm. They used a Vp-tree to compute distance between objects in multidimensional space. Different tests varied the dimension between 2 and 16, while data size varied between 10k and 100k points. However, they conclude that Vp-tree is more efficient than R*-tree and M-tree, on the basis of the results from their tests [11].

3 Methods

We implemented Kd-tree, Vp-tree and Octree to store and retrieve neuronal morphology data, in a typical scenario of constructing a neural network based on morphology of constituting neurons. We used a SWC file that is a standard file format for representing neurons and consisting of spatial coordinates [28]. The SWC file we used included a single neuron of a rat brain, from which we created two neuronal networks of different sizes.

During the network building phase within a population of neurons, a massive search for every node and node’s nearest neighbors is performed, as mentioned in section 2.2.2. For big populations this task is very heavy and requires efficient tools to be completed quickly. Thus, the highest speed is achieved, when all nearest neighbors of a requested node within required distance can be stored in L1 cache. This way data will be quickly located using minimal number of CPU cycles. To enable this, data should be partitioned in space and stored in efficient data structures. Therefore, we tested three data structures on a population of neurons, positioned beside each other with high density, to emulate the most difficult scenario of the network building. Furthermore, we measure L1 cache misses, CPU time and RAM usage to quantify efficiency of the selected tools. Software tools such as NeuroM, TREES toolbox and btmorph, as mentioned in 2.2.1, use different list structures to store SWC files. Thus, we decided to implement list structures, to compare with Vp-tree, Kd-tree and Octree.

In order to see how the chosen data structures, Octree, Kd-tree and Vp-tree differ in performance we constructed different test cases where we stored three dimensional coordinates together with different R/D values. The R/D represents the radius for a sphere and the length of a half side of a cube. Furthermore, we tested searching for a specific node and within a cube/sphere to detect how they differ in performance.

3.1 Computer specification

During this project we have used a Macbook Pro (Retina, 15-inch Early 2013) running OS X (64 bits). It is equipped with a 2.4 GHz Quad-Core Intel Core i7 processor (Turbo Boost up to 3.4 GHz). It has 32 KB instruction cache, 32 KB data cache and 6 MB shared L3-cache. Furthermore it has 8GB of DDR3 RAM which is operating at 1600Mhz [29].

3.2 Data structure and algorithm construction

The implemented data structures for comparison were Octree, Vp-tree and Kd-tree, as well as two naive implementations.

3.2.1 Kd-tree

Kd-tree is constructed using a list of nodes and a depth. The current dimension decides which axis should be split into two hyperplanes. The list of nodes is sorted by axis of the current dimension. To balance Kd-tree a median index is calculated to generate two subtrees of approximately equal sizes. The node with median index is parent and nodes with lower index than median are in the left child, and nodes with higher index than median are in the right child. As we go further down in the tree, the depth increases by one. The pseudo code below explains the construction of the Kd-tree in more detail.

Kd-tree

```
build_tree(nodeList, depth)
        if(size of nodeList == 0) return null
        if(sizeof(nodeList) == 1) return leaf(pointList)
        current_dimension ← depth % 3
        median ← sort list by current_dimension and choose median
        node ← median

        node.left ←  build_tree(points in nodeList before median , depth+1)
        node.right ← build_tree(points in nodeList after median, depth+1)
        return node
```

The search algorithm takes a node and a value d where the coordinates of the node is the center of the cube and each side of the cube has length 2*d. The algorithm searches for all nodes within the given cube by investigating if the cube is in the right or left hyperplane, according to current dimension. By exploring the hyperplane that consists of coordinates within the given cube, all nodes can be found and stored in a list.

3.2.2 Vp-tree

Vp-tree is constructed from a sorted list of nodes. A random node is chosen from the list and a median index is calculated, to generate two subtrees of approximately equal sizes. The nodes are sorted based on their distance to the random node in increasing order. The random node will be the center of a sphere and all nodes that have lower index than median will go into the left child and all nodes with greater index than median will go into right child. The threshold of the random node is calculated as the distance from the random node to the node with median index. The pseudo code below explains the construction of the Vp-tree in more detail.

Vp-tree
```
build_tree(nodeList)
    if(size of nodeList == 0) return null
    if(size of nodeList == 1) return leaf(nodeList)
    node ← choose an arbitrary node in nodeList (random node)
    median ← index of median node in nodeList
    pointList ← sort point around median distance where center is node

    node.threshold ← distance between first element and  median element
    node.data ← data of first element in nodeList
    node.left ←  build_tree(nodes in nodeList before median)
    node.right ←  build_tree(nodes in nodeList after median)
    return node
```

The search algorithm takes a node and a value r which is the radius of the sphere. The algorithm searches for all nodes within the given sphere. From the root it goes through the Vp-tree and calculates the distance from the wanted node and current node. However, the current node has a threshold distance. All nodes within threshold distance are in the left child and all outside threshold distance are in the right child. We explore the children containing our wanted sphere based on the threshold, if both children contains part of the sphere we explore both. All nodes within a given sphere are stored in a list.

3.2.3 Octree

As previous data structures, Octree is constructed from a list of nodes. Octree constructs a cube by given axis sizes, e.g. size of cube is x-size*y-size*z-size. Furthermore, if a cube already contains a node, the second node will split the big cube in 8 equal sized smaller cubes and place the first and second node in the cube that contains their coordinates. If nodes are placed in the same cube again, the current cube will split again in 8 equal sized cubes and this continues until each cube contains only a single node. This procedure is done to the whole list of nodes and will not balance the tree. Thus, pseudo code below explains the construction in more detail.

Octree
```
build_tree(nodeList)
    octree ← new Octree
    For each node in nodeList
        Insert(octree, node)

insert(octree, point)
    if(octree is leaf)
```

```
            if(octree.data == NULL)
                octree.data = point
                return
            else
                oldnode = octree.data
                octree ← create 8 equal childs of current octree

                for each child in octree
                    if(child contains node)
                        child.data = point
                    if(child contains oldnode)
                        child.data = oldnode
        else
          for each child in octree
            if(child contains node)
                insert(child,node)
```

However, by given cube/node a search in Octree starts at the root and investigates which of the 8 children contains the wanted cube/node. Each non-leaf node contains a set of coordinates within the current octant. Thus, children that contains the wanted node/cube are explored. This continues until a leaf that is the wanted node, or all nodes within the wanted cube are reached. All nodes within a given cube is collected and stored in a list.

3.2.4 Naive

The naive implementation consist of two data structures, Vp-naive and naive. Both implementations have linear search time with the main difference that Vp-naive uses a sphere as searching algorithm and naive uses a cube. Thus, the pseudo codes below explain the difference in further detail.

Vp-naive

```
linear_search(nodeList, wanted_node, dist, resultList)
        for each node in nodeList
                if( dist >= distance(node, wanted_node) )  resultList ← node
        return resultList

distance(node n1, node n2)
        return distance between n1 and n2
```

Naive

```
naive(nodeList, cube, wanted_node, resultList)
        for each node in nodeList
                if( nodeInsideBox(cube, wanted_node) ) resultList ← node
        return resultList

nodeInsideBox(cube, node)
```

```
        if(node inside cube) return true
        return false
```

3.4 Creation of neuronal networks from a SWC file

To construct a neuronal network, we used a single neuron, reconstructed from a rat brain. The reconstruction was made of a SWC file format containing 76810 nodes, which corresponds to a detailed geometrical model of a neuron.

Each time the SWC file was read, we cloned the rat neuron and incremented the (x,y,z) coordinates by (x + dx, y + dy, z + dz) in order to shift the neuron in space. This procedure was made with 10 and 100 loops, which constructed a network of 10 and 100 neurons, respectively. However, the small values we used on delta x, delta y and delta z, were chosen because we wanted a dense neuronal network. Thus, the values on delta x, delta y and delta z, were 0µm for a network consisting of the rat neuron. The delta values were incremented by 1 micron in each loop for neuronal networks consisting of 10 and 100 neurons.

3.5 Test cases

We constructed test cases which included three dimensional coordinates together with three different D/R values. As mentioned before, the D/R represents the radius for a sphere and the length of a half side of a cube. The test case coordinates were chosen randomly from the neuronal networks. For example in Vp-tree if D/R is d, it means that all nodes within the sphere with radius dµm from the current node will be found. The same values in Kd-tree and Octree where current node is the center and D/R is d, means that all nodes within the cube, which has length of the sides 2*d.

The D/R value was generated randomly for each test case. Thus, three ranges of D/R values were used, 0µm which is the exact coordinates for a specific node, 0-5µm which may have a value from 0-5 and 5-10µm which may have a value from 5-10. However, the 0-5µm and 5-10µm cases require usage of spherical or cubical search algorithms depending on data structure. These search strategies were chosen because of the time frame of the project. Searching for a specific node tests the performance of a data structure for searching specific coordinate in a three dimensional space. Furthermore, searching for bigger volumes was decided in order to see how they differ in performance, when performing searches for bigger volumes in the neuronal populations. Due to limited time and resources we could not perform searches for bigger search volumes.

For each neural network size, 5 different test cases were created with 10, 1.000, 10.000, 100.000 and 1.000.000 number of random search nodes. These values were chosen to test cache usage of the data structures, because same nodes and cubes/spheres were searched several times during the execution. Furthermore, each of these test files included D/R ranges as mentioned in the previous paragraph. The total amount of test cases during this thesis were 45.

3.6 Measuring strategies

We measured the average search time and cache misses while performing searches for a specific node or all nodes within a given cube/sphere. A time library in C++ was used to measure CPU time, since it measures amount of clock cycles the CPU requires to perform a specific task [24]. Thus, the amount of clock cycles can be converted into seconds. The measuring tool was easy to use and it measured time with sub-second precision which was sufficient [24]. Furthermore, we measured RAM usage during the construction of the different neuronal networks in each data structure. The measuring tool we used was Valgrind, an easy to use, and an efficient tool for measuring RAM usage, cache misses, etc. in C++ [25] [26]. Lastly, we measured the cache misses during construction and searches in respective data structure, in order to see how each data structure made use of data in cache memory. However, as mentioned in section 2.3, CPU, RAM and cache are significant parts of fast execution and good performance.

4 Results

4.1 CPU performance

Figure 6 and 7 shows the average search time for each data structure, including searches for a specific set of test nodes and spheres/cubes. The time is measured in milliseconds and it’s the average search time to find a single node or sphere/cube within a single neuron or neuronal network. The number of test nodes and spheres/cubes vary between 10-1000k, and the data structures are represented with specific colors.

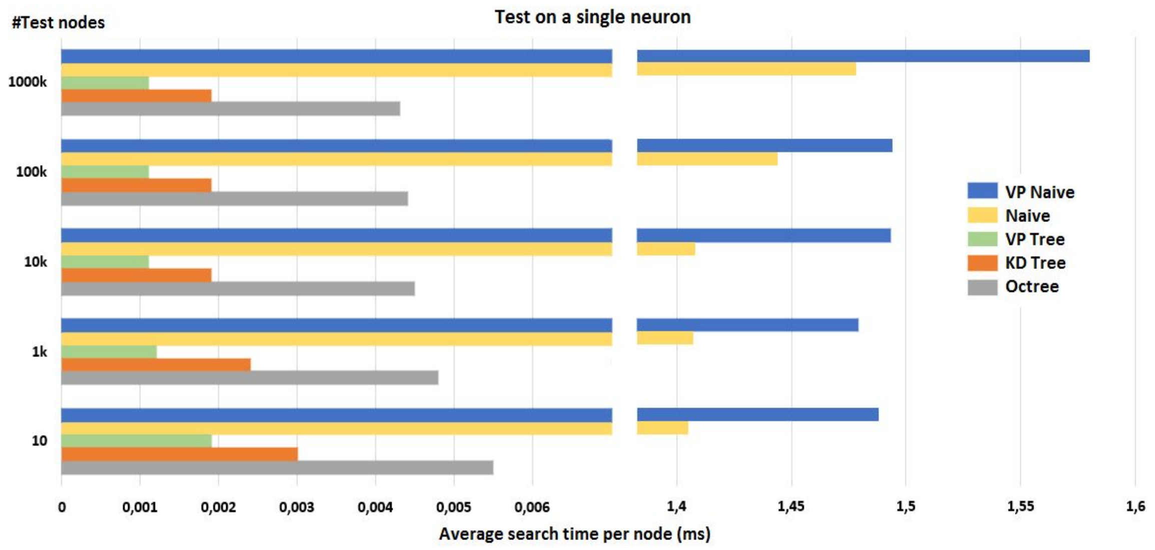

Figure 6: CPU performance of searches on a specific node.

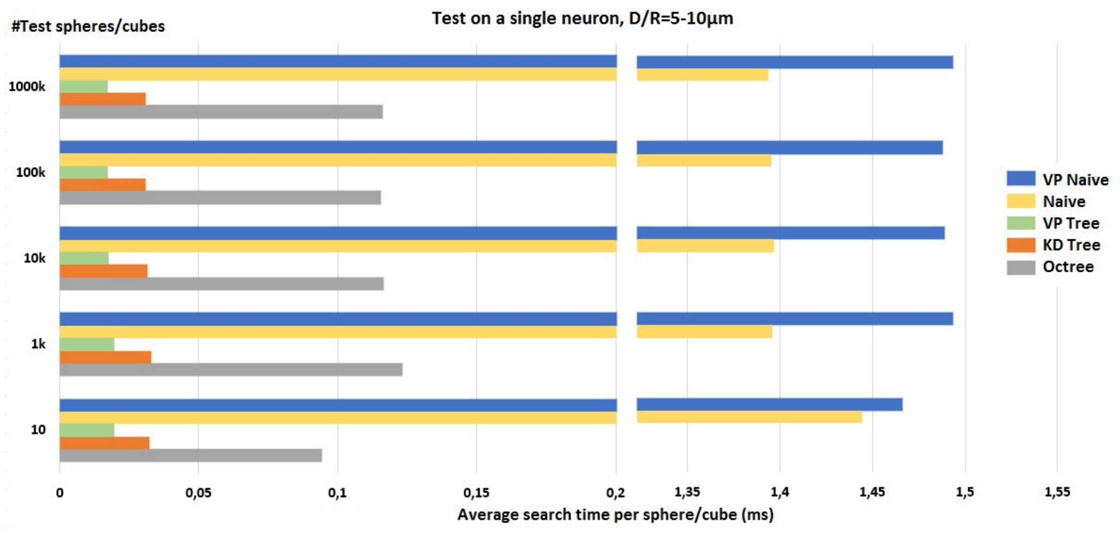

Figure 7: CPU performance of searches within a cube/sphere with D/R=5-10µm.

Vp-naive and naive implementations have the highest average search time per node in a single neuron, as shown in Figure 6. Vp-tree has the lowest average search time per node. As the number of searched nodes increase, the average search time decreases for Vp-tree, Kd-tree and Octree. When the number of searched nodes increase in naive and Vp-naive, the average search time per node increase as well. The lowest average search time per node is approximately 0,001 milliseconds and highest average search time is 1,57-1,58 milliseconds.

Figure 7 shows the result of average search time required to find all nodes within a given cube or sphere in a single neuron. Vp-tree and Vp-naive are searching for all nodes within a sphere, while Octree, Kd-tree and naive are searching for all nodes within a cube.

The average search time for a sphere/cube in Kd-tree, Octree and Vp-tree is higher than for a specific node in a single neuron, see Figure 7. The naive implementation has lower average search time than Vp-naive. Vp-tree has the lowest average search time, followed by Kd-tree and Octree.

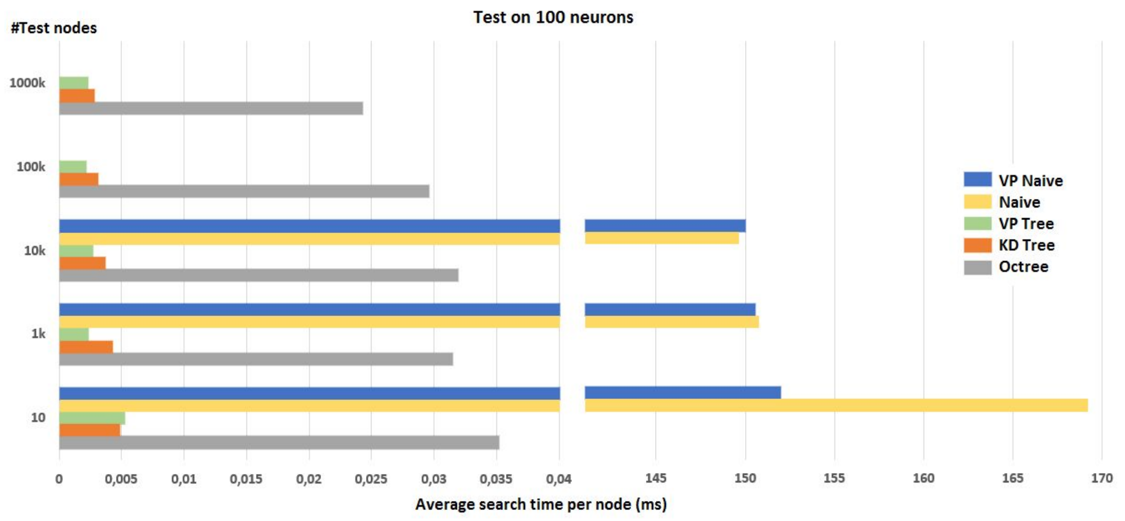

Figure 8: CPU performance of searches on a specific node.

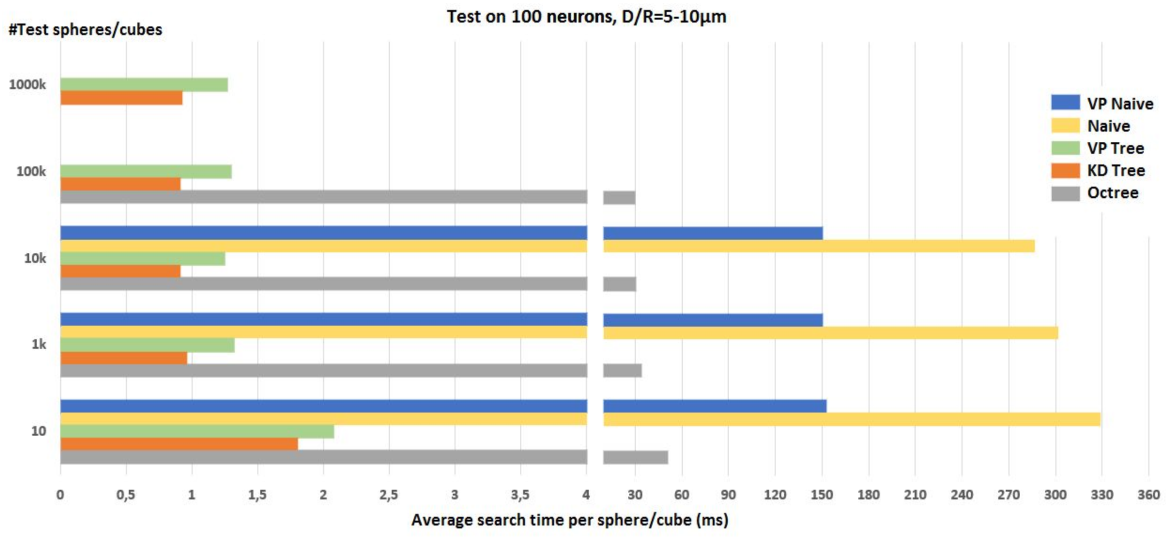

Figure 9: CPU performance of searches within a cube/sphere with D/R=5-10µm.

The naive implementation has the highest average search time per node and sphere/cube, except for test with 10k nodes, see Figure 8 and 9. Vp-tree has lowest average search time while searching for a specific node and while searching for spheres/cubes, Kd-tree has the lowest average search time, see Figure 9. Octree has the third highest average search time. When searching for a specific node or sphere/cube, the naive and Vp-naive implementations are not measured for 100k-1000k nodes and spheres/cubes. Octree is not measured for 1000k test with spheres/cubes, in the population of 100 neurons due to long execution time.

4.2 RAM performance

The result of RAM usage of each data structure during the building phase for 1, 10 and 100 neurons, see Figure 10. The naive consist of both naive and Vp-naive implementation, since both store the morphological data in a list.

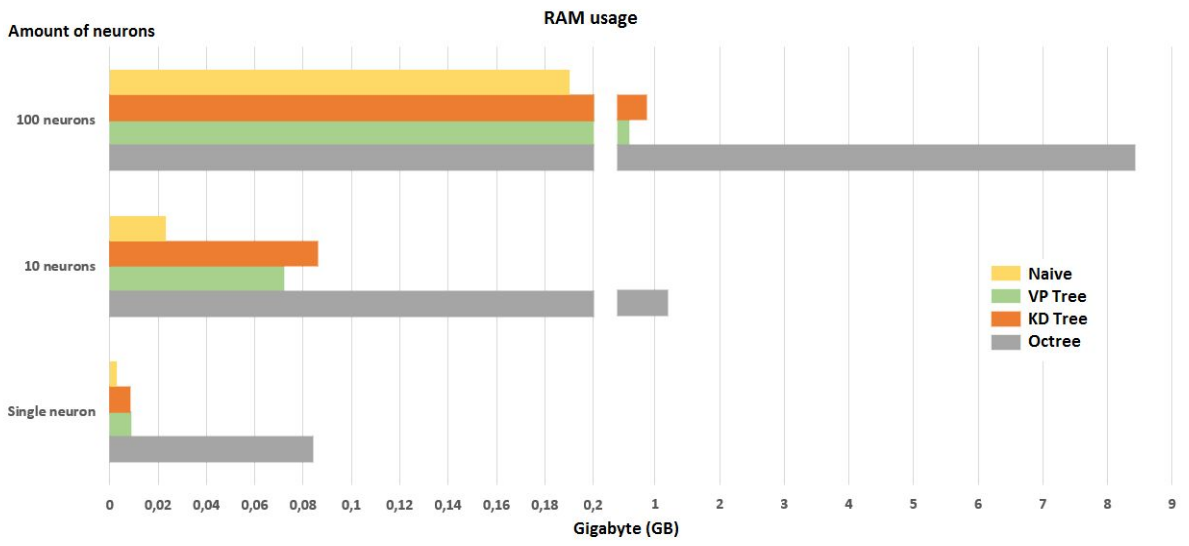

Figure 10: RAM usage measured in gigabyte, while building neuronal network.

As shown in Figure 10 building an Octree of 100 neurons has a memory usage above 8GB. Vp-tree and Kd-tree has about the same memory usage while the naive implementations have lower RAM usage than all three other data structures.

4.3 Cache performance

Results of cache misses in the different data structures are shown in Figures 11 and 12. The number of misses are measured in percentage and it’s the total percentage of searching for a specific set of nodes and spheres/cubes in a single neuron and neuronal network. The data structures are represented by different colors and the number of nodes and cubes/spheres vary from 10 up to 1000k, see Figure 11. The number of misses are measured on level one cache, and the number of data structures measured vary depending on test size. The tests are performed on a single neuron and on a network of 100 neurons. The different tests also vary from searching for a single node, as shown in Figure 11 and 13, and searching for a cube/sphere, as shown in Figure 12.

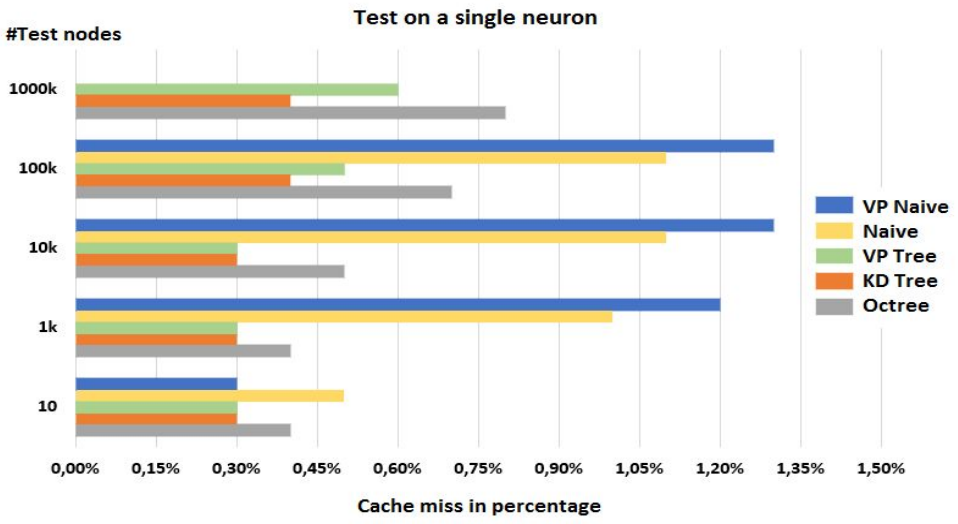

Figure 11: Cache performance of searches on a specific node.

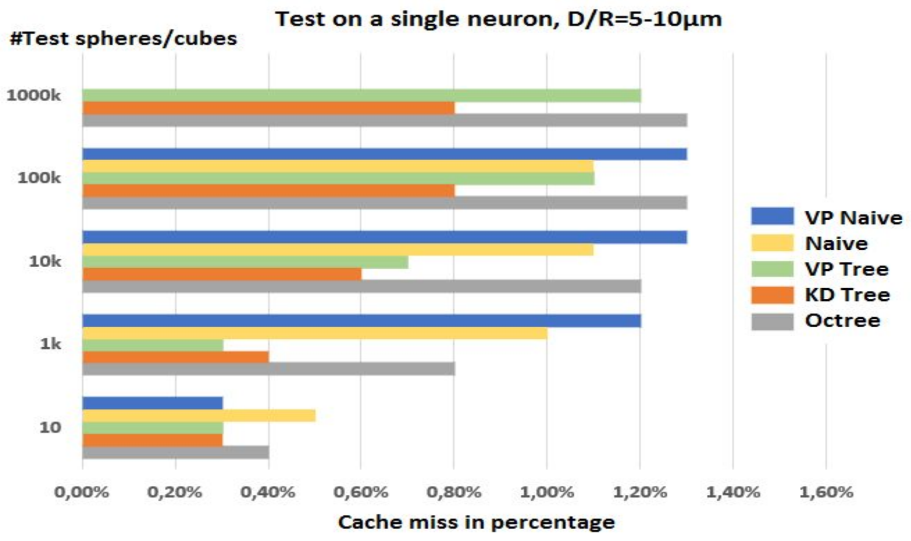

Figure 12: Cache performance of searches within a cube/sphere with D/R=5-10µm.

Searching for a specific node in a single neuron, has lowest cache miss percentage with the Kd-tree, as shown in Figure 11. As the number of searched nodes increases the percentage differs from 0.30% to 0.35% in the Kd-tree. Vp-naive implementation has the highest cache miss percentage of 1.33%, while searching for 100k nodes in a single neuron. The data structure that has the second lowest percentage in cache misses is the Vp-tree followed by Octree and naive.

When searching for a sphere/cube in a single neuron with D/R as 5-10μm, Kd-tree has the lowest cache miss percentage, as shown in Figure 12. Octree and Vp-naive implementations have the same percentage in the test with 100k spheres/cubes, while Naive implementation has lower percentage. The naive implementation has the same percentage as Kd-tree when searching for 100k spheres/cubes.

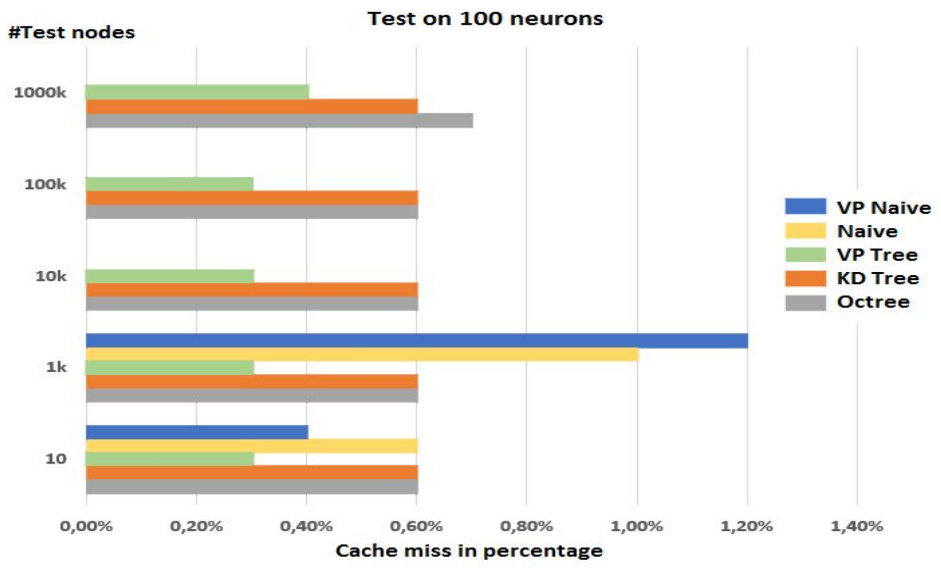

Figure 13: Cache performance of searches on a specific node.

Searching for a specific nodes in a network of 100 neurons, Vp-tree has the lowest percentage in all test cases, as shown in Figure 13. The naive implementations, Vp-naive and naive, is not measured for 10k-1000k. Octree has the highest percentage of the three implemented data structures, while searching for 1000k nodes.

5 Discussion

The circuit reconstruction phase in a biologically detailed neuronal network simulation requires efficient data structures and search methods. Therefore, usage of the most suitable data structure is important. In this study we used neuron morphology data to construct populations of neurons. We stored the population in the implemented space-partitioning data structures, Kd-tree, Vp-tree and Octree, to examine how they differ in performance.

5.1 Result analysis

We investigated how performance differ between Vp-tree, Kd-tree and Octree. The results show that there is a significant difference in performance between the data structures. Vp-tree has the lowest average search time in all test cases followed by Kd-tree and lastly Octree, as shown in Figure 6 and 7. Octree has a higher average search time compared to Vp-tree and Kd-tree. This result can be explained by the fact, that Octree is not a balanced tree as the other two structures, as mentioned in section 3.2. Therefore, the worst search time in Octree is significantly higher than the worst search time in Vp- and Kd-tree. Furthermore, the difference between Vp-tree and Vp-naive is big, thus, searching for a specific node in a single neuron is approximately 1150 times faster in Vp-tree. However, searching for a sphere in the naive implementation, has approximately the same average search time as searching for a node, while for Kd-tree, Vp-tree and Octree the average search time increases, see Figure 6 and 7. When searching for a volume in the binary trees such as Vp-tree, Kd-tree and Octree, requires exploring many branches in order to find the nodes within the given volume. Same search in a list structure such as Vp-naive and naive, doesn't increase the number of explored nodes, since the organization of the elements in the list doesn't change depending on the Vp-naive or naive.

In Figure 8, Vp-tree has still the lowest average search time followed by Kd-tree. Kd-tree has the lowest average search time in all test cases when searching for a cube in 100 neurons, as shown in Figure 9. Vp-tree tends to have lower average search time in smaller networks and in searches for specific nodes in larger networks, while Kd-tree is better for volume searches in larger networks.

We also intended to investigate how much RAM usage each implemented data structure requires. The result of RAM usage shows that Octree requires significantly more RAM than Vp-tree and Kd-tree, as shown in Figure 10. Thus, Octree does not store data in non-leaf nodes and therefore more nodes will be used, as described in section 3.2.3. As the population grows bigger, the more unused nodes will Octree contain. However, by analyzing the result of Figure 10, we can distinguish that Octree uses approximately 7 times more RAM memory than Vp- and Kd-tree, which indicates that Octree has many unused nodes and therefore requires more memory.

Kd-tree had lowest miss percentage in level one cache in most test cases, when performing different searches in a single neuron, followed by Vp-tree, as shown in Figures 11 and 12. Octree had in a few test cases higher miss percentage than the naive implementations. As shown in Figure 11, Octree differs in tests between 10 and 1000k with a percentage of approximately 0,4% compared to Kd-tree which only differs with around 0,1%. This shows that Kd-tree uses already stored data in level one cache, significantly more than Octree.

The unexpected result was that Vp-tree had lower miss percentage in level one cache than Kd-tree, when performing searches for a specific node in a populations of 100 neurons. Kd-tree had lowest miss percentage in level one cache in all tests with a single neuron. A reasonable conclusion is that we measured cache misses of storing the population together with the different searches. Thus, the majority of the cache misses, in tests with a population of 100 neurons, is from constructing the network. However, in order get more accurate results of cache misses during searches, we should have measured cache misses of searches separately.

As the volume and the number of neurons increase, more data is accessed from the level one cache. Kd-tree has lowest miss percentage in level one cache in the first two tests. As mentioned in the previous paragraph, majority of the cache misses in the test cases with 100 neurons, occur during the constructing phase. Thus, Kd-tree has the lowest average search time in larger populations and an explanation could be that, it makes better use of the data in level one cache memory.

In a related study by Juhong et al. [27] it was clear that Octree, a space-driven structure, have higher CPU time, when performing different searches compared to R-tree, data-driven structure. As mentioned in section 2.5, Vp-tree and Kd-tree are data-driven structures. Therefore, our results are similar to theirs. Thus, they compared Octree to R-tree, which does not partition the data in space the same way as Vp-tree and Kd-tree.

Previous studies [10] have concluded that Kd-tree has lower CPU time when performing nearest neighbor searches in lower dimensions than Vp-tree, such as dimensions lower than 16. Thus, our results show that Vp-tree is better in majority of the tests, when measuring searches in three dimensions. This result may be explained by the fact that our results are not from nearest neighbor searches and therefore may differ.

5.2 Method discussion

During the measurement of the chosen data structures, we have had some limitations, such as time frame of the project, cache size, RAM size, etc. All these factors could have affected the results with higher or lower values.

First thing that affected our measurements is the computer performance, more specifically size of the cache, RAM size and CPU performance. These restrictions must be taken in account during analyzation of the results, because as mentioned in section 2.3, bigger cache size results in faster search time, since more data can be stored and faster accessed. As mentioned in section 2.3, cache memory is faster than RAM, and much faster than secondary memory. Furthermore, higher RAM size would have resulted in less swapping from the secondary memory [21]. Lastly CPU performance depends on the configuration, amount of cores and clock frequency. During this measurement all tests were performed on a single core, one threaded execution. This will not be the case during the real simulation, because supercomputer contains thousands of cores and all must be used for best performance. Thus, these values are good start points to already at single core see the difference, with multi-core usage performance will increase and result in lower values. Furthermore, the speed of the CPU is important in managing amount of instructions per clock cycle, which can decrease the execution time.

The time frame of the project was another limiting factor, which resulted in smaller amount of tests and smaller neuronal networks. This restriction was caused by the required time to measure miss percentage in caches. Thus, we didn’t get all data while measuring cache misses in a populations of 100 neurons, like we did for the CPU tests. Even with 100 neurons, few values are missing for cache results because it exceeded our time limit. Furthermore, the cache miss values for bigger networks have been measured only once, which makes it less reliable. The average search time values were measured multiple times, since the measurement without using valgrind was significantly faster. An average value was computed from the multiple measurements, in order to gather more reliable data. However, researches comparing Kd-tree and Vp-tree were found, as well as comparison of Octree with other space-partitioning structures. Although, researches comparing Kd-tree, Vp-tree and Octree were not found, which makes it difficult to validate our data.

6 Suggestions for future research and Conclusion

6.1 Future research

The next step in this research is to construct bigger populations of neurons, that can better illustrate the real amount of neurons which will be used during the brain simulation. Diverse data structures exist, which should be taken in account for finding the most efficient one. Cache measurements should be made separately for creation of the neuronal network and searching. Thus, a more advanced computer with bigger cache and higher computing power is needed, to measure bigger populations. Therefore, a high performance computer should be used for the measurement. In that case the measurement would be more precise, since it would be measured on a similar hardware as the simulations are performed.

6.2 Conclusion

This study has investigated performance of three space-partitioning data structures, Octree, Vp-tree and Kd-tree when storing and retrieving morphological data. The results show that Vp-tree is most suitable for searches in smaller networks and for specific nodes in larger networks, while Kd-tree is better for searching within a cube in larger networks. The memory usage indicates that Vp-tree and Kd-tree are more appropriate than Octree for usage on computers with memory limitations. Thus, more tests are needed on larger populations of neurons and bigger search volumes on different space-partitioning data structures, in order to find the most suitable one for whole brain simulation.

References

1. Markram H, Meier K, Lippert T, Grillner S, Frackowiak R, Dehaene S, et al. Introducing the Human Brain Project. Procedia Computer Science. Dec. 2011. 7:39-42.
2. Byrne JH. Introduction to Neurons and Neuronal Networks [Internet]. Houston: Department of Neurobiology and Anatomy; [Date unknown]. [cited 24th February 2016]. Available from: http://neuroscience.uth.tmc.edu/s1/introduction.html
3. Furber S, Temple S, Brown A. High-Performance Computing for Systems of Spiking Neurons [Internet]. Manchester, Hampshire: The University of Manchester; [Date unknown]. [cited 10th March 2016]. Available from: http://apt.cs.manchester.ac.uk/ftp/pub/apt/papers/SBFaisb06.pdf
4. Britt AH, Humble ST. High-Performance Computing with Quantum Processing Units. arXiv:1511.04386 [cs.ET]. Nov. 2015. Available from: http://arxiv.org/pdf/1511.04386v1.pdf
5. Markram H, et al. Reconstruction and Simulation of Neocortical Microcircuitry. CELL. 2015 Oct: 163(2):456-492.
6. Helmstaedter M, et al. Connectomic reconstruction of the inner plexiform layer in the mouse retina. Nature. 2013 Aug: 500(7461):168-174.
7. Humphries MD, Wood R, Gurney K. Reconstructing the Three-Dimensional GABAergic Microcircuit of the Striatum [Internet]. PLOS Computational Biology; 2010 [cited 5 May 2016]. Available from: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1001011
8. Reimann MW, King JG, Muller EB, Ramaswamy S, Markram H, “An algorithm to predict the connectome of neural microcircuits”, *Front Comput Neurosci.*, Oct. 2015.
9. Mehta DP, Sahni S. Handbook of DATA STRUCTURES and APPLICATIONS [Internet]. CHAPMAN & HALL/CRC COMPUTER and INFORMATION SCIENCE SERIES; 2005 [cited 10th April 2016]. Available from: http://tinyurl.com/zvyesxy
10. Kibriya AM. Fast Algorithms for Nearest Neighbor Search [Master thesis]. Hamilton, The University of Waikato; 2007.
11. Ada Wai-chee Fu, Polly Mei-shuen Chan, Yin-Ling Cheung, Yiu Sang Moon. “Dynamic vp-tree indexing for n-nearest neighbor search given pair-wise distances”. The VLDB Journal. 2000 Jan; 9:154–173.
12. Cannon RC, Turner DA, Pyapali Gk, Wheal HV, “An on-line archive of reconstructed hippocampal neurons”, J Neurosci Methods. 1998 Oct; 84(1-2):49-54.
13. NeuroMorpho.org: A Central Resource for Neuronal Morphologies [Internet]. *George Mason University*; 2016 [cited 4th April 2016]. Available from: http://Neuromorpho.org
14. Szymik B. A Nervous Journey [Internet]. ASU - Ask A Biologist; 2011 [cited 9th February 2016]. Available from: https://askabiologist.asu.edu/neuron-anatomy
15. Jabr F. Know Your Neurons: How to Classify Different Types of Neurons in the Brain's Forest [Internet]. Scientific American: Ferris Jabr; 2012 [cited 2 May 2016]. Available from: http://tinyurl.com/nsqrwqs

16. NeuronM 0.0.16 [Internet]. [Place unknown]: [Publisher unknown]; 2015 [cited 18th April 2016]. Available from: http://neurom.readthedocs.org
17. Torben-Nielsen B. Welcome to btmorph's documentation! [Internet]. [Place unknown]: [Publisher unknown]; 2014 [cited 18th April 2016]. Available from: http://btmorph.readthedocs.org
18. Cuntz H, Forstner F, Borst A, Häusser M. One rule to grow them all: A general theory of neuronal branching and its practical application [internet]. PLOS Computational Biology; 2010 [cited 18th April 2016]. Available from: http://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1000877
19. Brette R, Rudolph M, Carnevale T, et al. Simulation of networks of spiking neurons: A review of tools and strategies [Internet]. Springer Science, Business Media; 2007 [cited 3rd March 2016]. Available from: http://www.cse.unr.edu/~fredh/papers/journal/19-sonosnarotas/paper.pdf
20. Pearn J. IBM Neural Tissue Simulator [Internet]. Munich: Artificial Brains; 2012. [cited 4th March 2016]. Available from: http://www.artificialbrains.com/ibm-neural-tissue-simulator
21. Drepper U. What Every Programmer Should Know About Memory [Internet]. Red Hat, Inc; 2007. [cited 25th March 2016]. Available from: https://www.akkadia.org/drepper/cpumemory.pdf
22. Rouse M. Cache memory [Internet]. Techtarget; 2014. [cited 26th March 2016]. Available from: http://searchstorage.techtarget.com/definition/cache-memory
23. Kuse M. C++ Performance Analysis & Profiling Tools [Internet]. Hong Kong: Kuse; 2012 [cited 15th March 2016]. Available from: http://tinyurl.com/jjqdhal
24. C Time Library [Internet]. Cplusplus.com; 2016. [cited 2nd May 2016]. Available from: http://www.cplusplus.com/reference/ctime/
25. Cachegrind: a cache and branch-prediction profiler [Internet]. Valgrind.com; 2015. [cited 20 February 2016]. Available from: http://valgrind.org/docs/manual/cg-manual.html
26. Massif: a heap profiler [Internet]. Valgrind.com; 2015. [cited 20 February 2016]. Available from: http://valgrind.org/docs/manual/ms-manual.html
27. Liu J, Panajoti G, Xiao H. Indexing [Project report]. Chicago: The University of Illinois at Chicago; [Date unknown] [cited 23rd April 2016]. Available from: https://www.cs.uic.edu/~hxiao/courses/cs581-project.pdf
28. NeuroMorpho.org: A Central Resource for Neuronal Morphologies [Internet]. [Updated June 2015, cited 9th February 2016]. Available from: http://Neuromorpho.org
29. MacBook Pro (Retina, 15-inch, Early 2013) - Technical Specifications [Internet]. Apple Inc. [Updated 28 July 2014; cited 27th April 2016]. Available from: https://support.apple.com/kb/SP669?locale=sv_SE&viewlocale=en_US

Appendix

CPU graphs

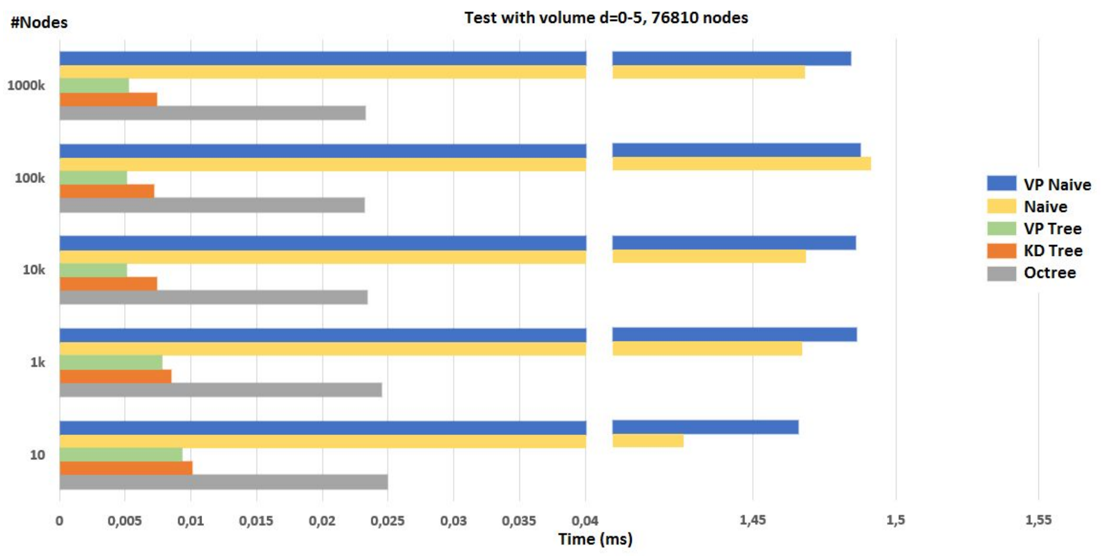

Figure 14: CPU performance of searches within a cube/sphere with D/R=0-5µm.

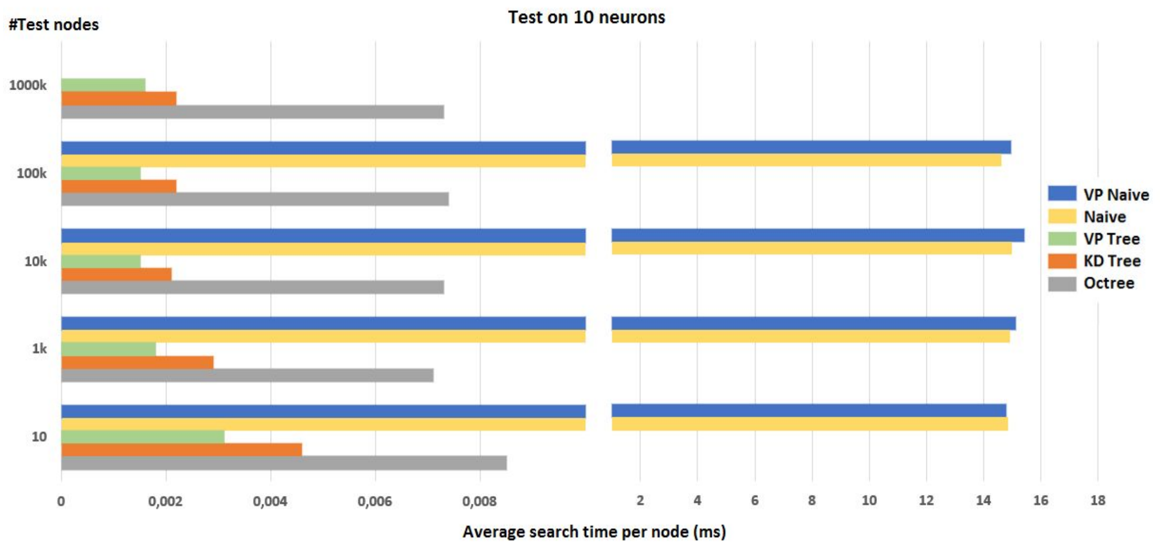

Figure 15: CPU performance of searches for a specific node in a network of 10 neurons.

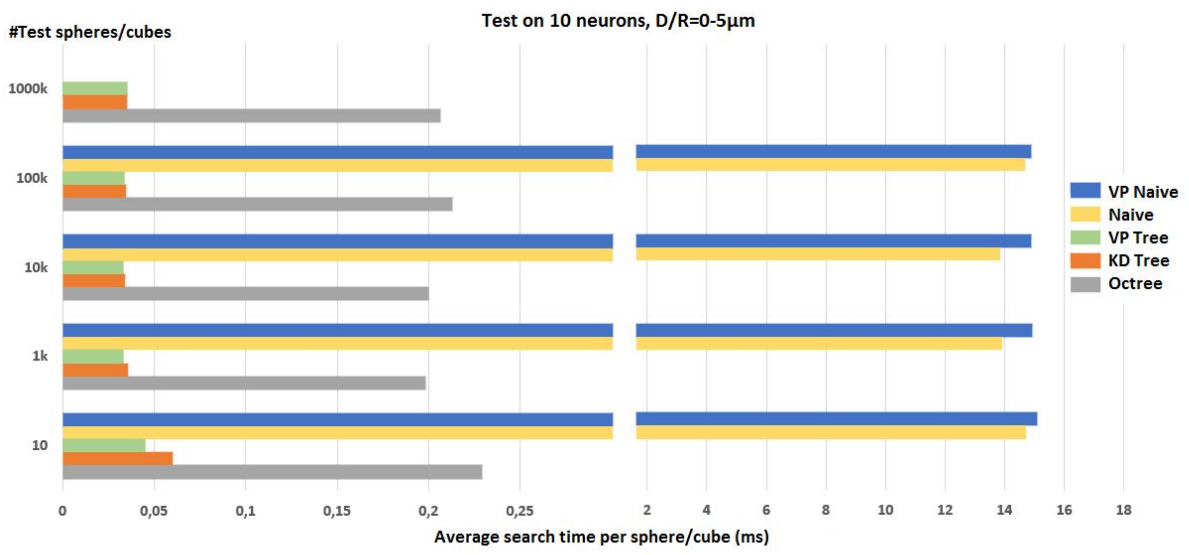

Figure 16: CPU performance of searches within a cube/sphere with D/R=0-5μm, in a network of 10 neurons.

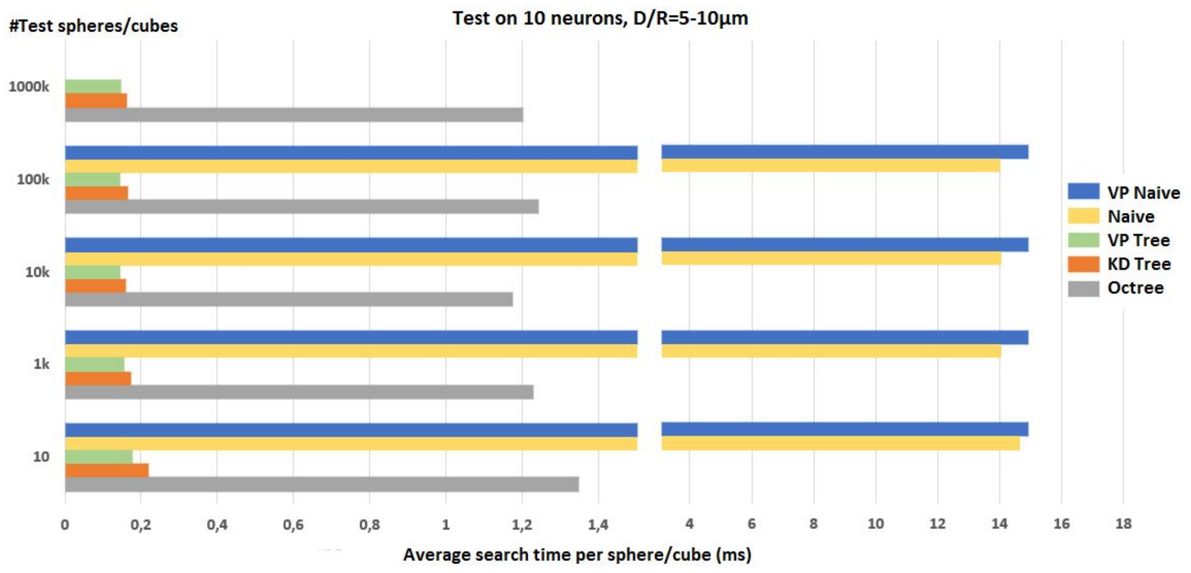

Figure 17: CPU performance of searches within a cube/sphere with D/R=5-10μm, in a network of 10 neurons.

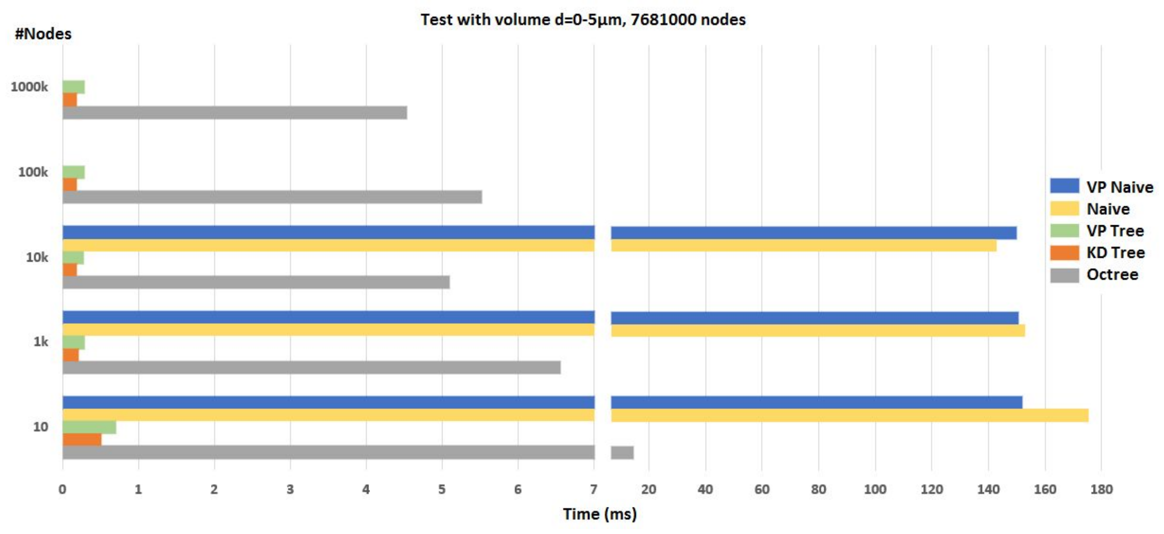

Figure 18: CPU performance of searches within a cube/sphere with D/R=0-5µm, in a network of 100 neurons.

Cache graphs

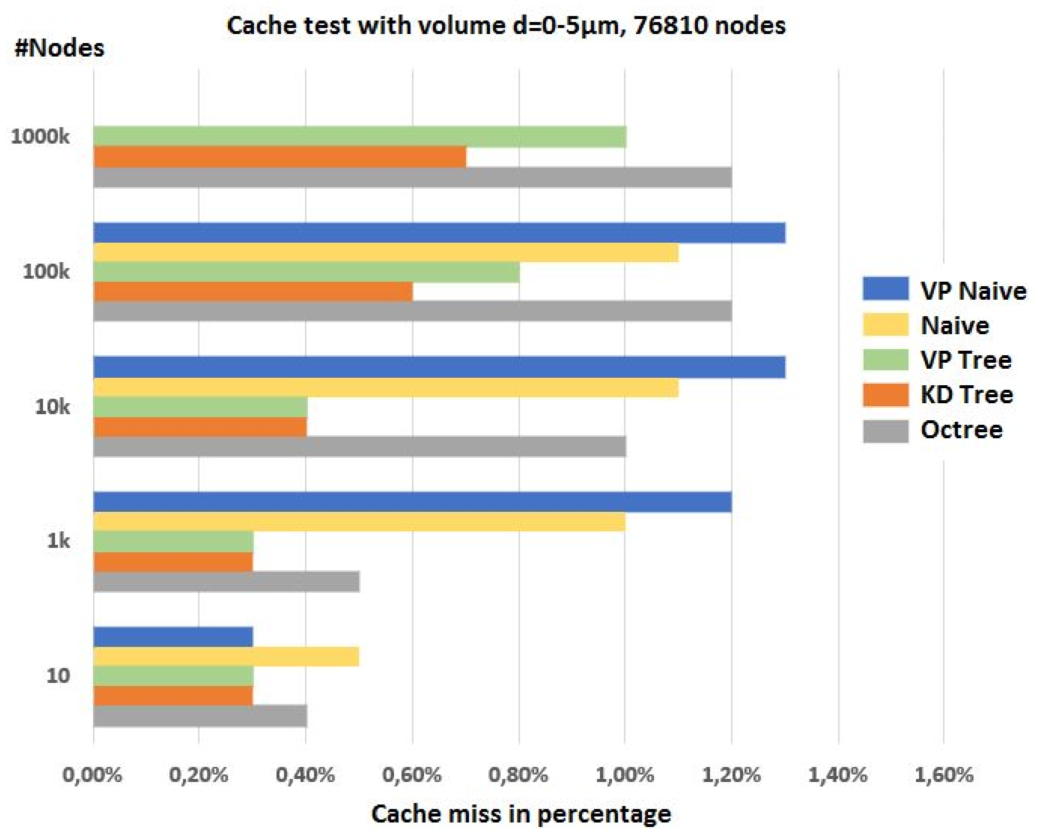

Figure 19: Cache performance of searches within a cube/sphere with D/R=0-5µm.

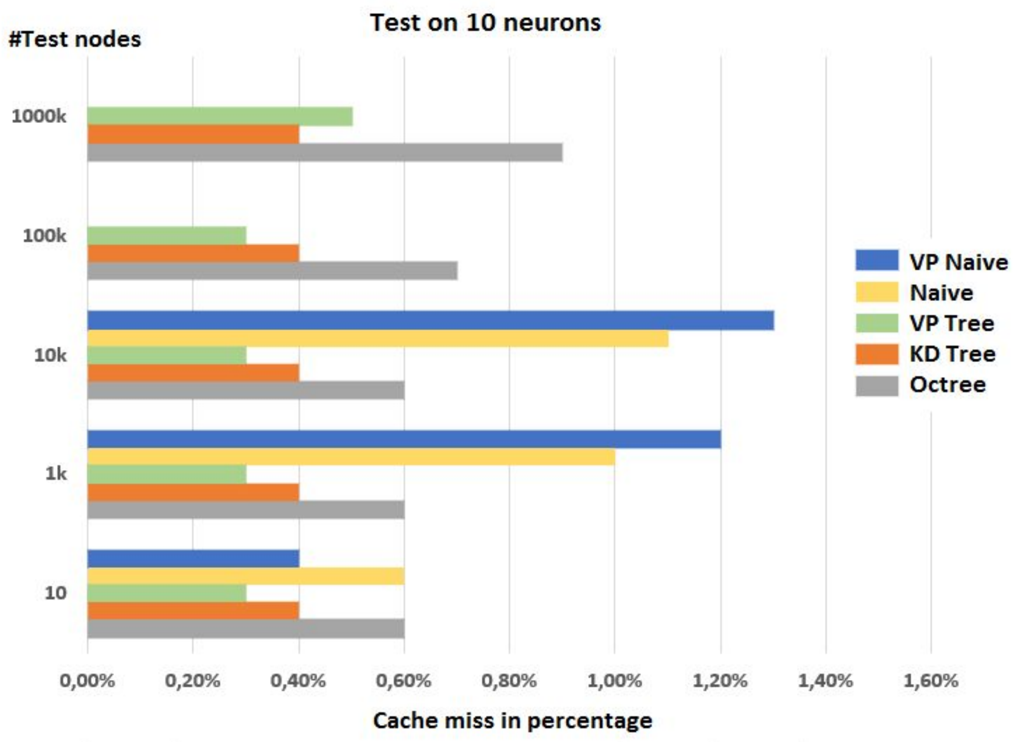

Figure 20: Cache performance of searches for a specific node, in a network of 10 neurons.

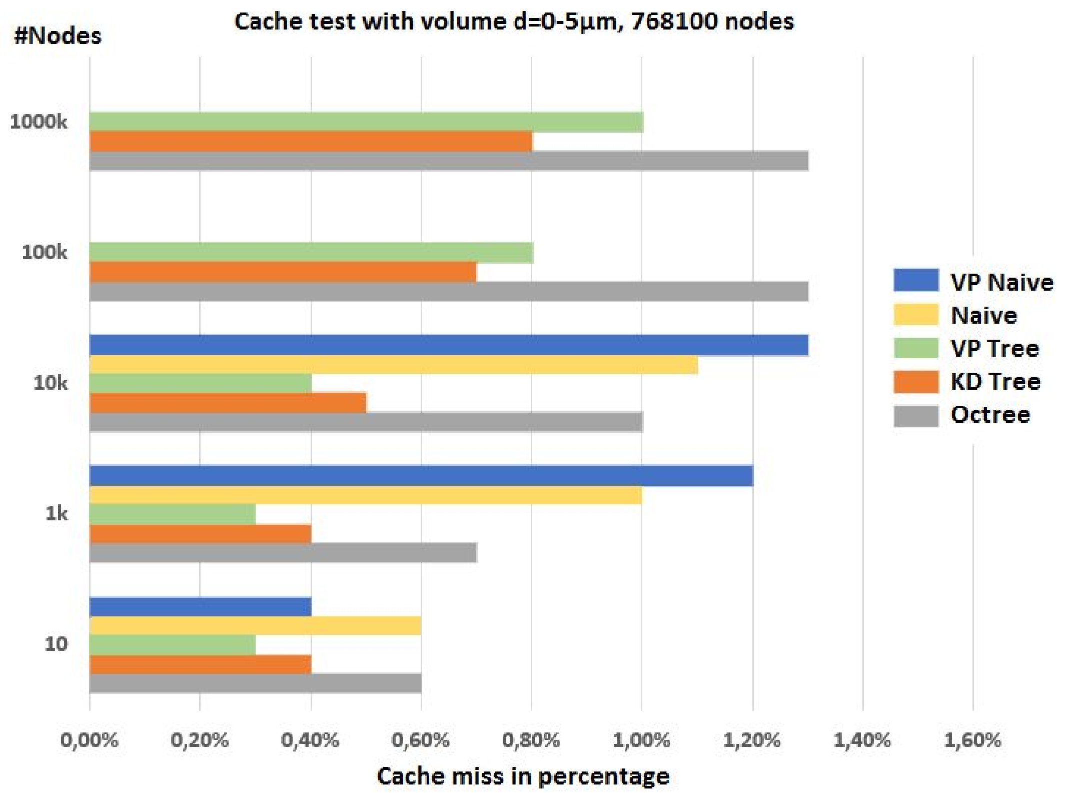

Figure 21: Cache performance of searches within a cube/sphere with D/R=0-5µm, in a network of 10 neurons.

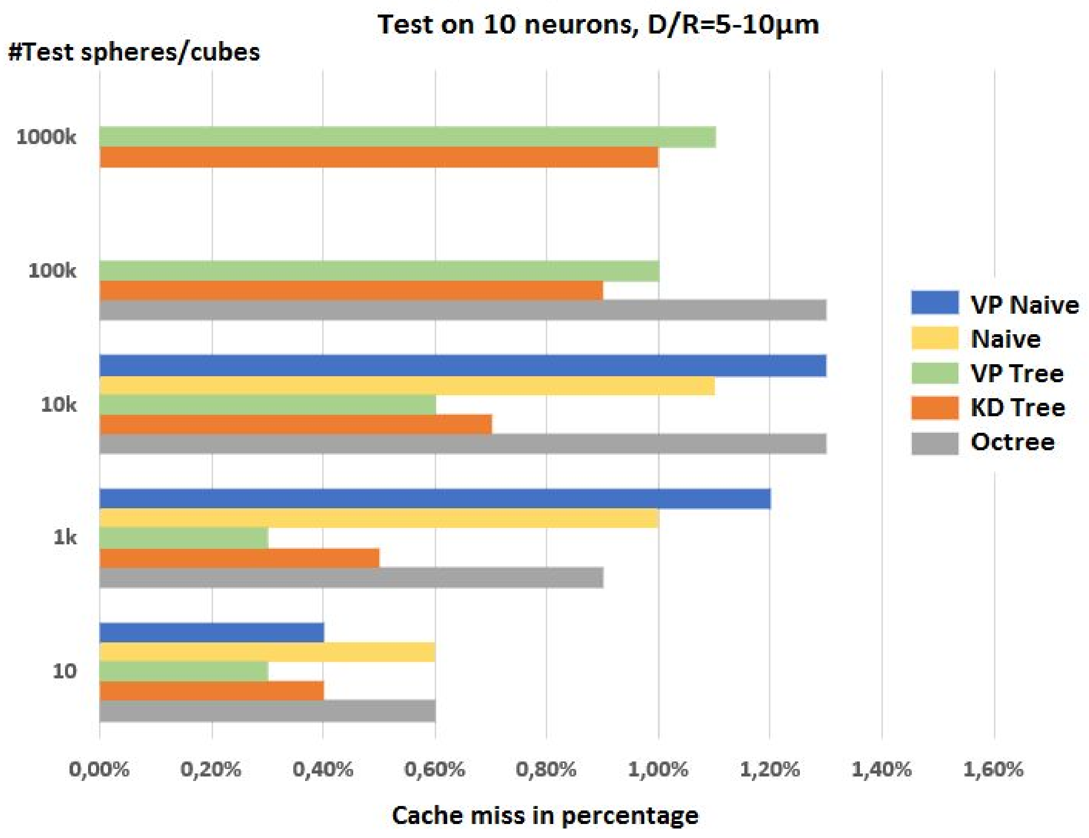

Figure 22: Cache performance of searches within a cube/sphere with D/R=5-10µm, in a network of 10 neurons.