Automating the neuron integration in connectome construction

Kalina Petrova

under the direction of
Prof. Sebastian Seung
Department of Brain and Cognitive Sciences, MIT

Research Science Institute
July 31, 2012

Abstract

The paper deals with automating the process of neuron integration when constructing a *connectome*, i.e., a comprehensive model of the neuron connections. To decrease the previously required amount of manual segment merging, algorithms for the following operations have been developed: (i) determining the most appropriate expansion of the current neuron state (*selection*) based on a dynamically changeable affinity threshold; (ii) growing the selection until a certain volume restriction is reached; (iii) removing portions of the selection based on the order in which the segments have been added; (iv) maintaining a global size threshold — a restriction on the size of the *neuron batches*. The complexity of the first three algorithms is $O(n)$, whereas the complexity of the method that handles the resetting of the size threshold is $O(n \times \log_2(n))$ where n is the number of segments. In addition, when implemented, the suggested algorithms yielded a 27.4% overall improvement of the users' time performance.

Summary

Constructing a *connectome* — a comprehensive map of a neural network — is a crucial step in modeling the brain's functions. An important stage in this process is obtaining the neurons' models by merging smaller segments together. To facilitate the manual procedure involved in this task, several algorithms have been developed and implemented. Instead of manually adding all the segments one by one in order to get the whole neuron, it is now possible to just trace the result of the algorithms and check for mistakes. The experimental testing of the innovative technique shows an average of 27.4% time improvement when performing a particular task.

1 Introduction

Computational neuroscience is an interdisciplinary field that uses programming techniques to gather information about the mechanisms of the brain to achieve better results in medicine and psychology. One of the major tasks in contemporary neuroscience is creating a model of brain dynamics. Constructing a *connectome* — a comprehensive map of a neural network — is a crucial step in modeling the brain's functions (Figure 1). This problem is the core of a specific subfield of neuroscience called *connectomics*. Finding more effective ways to create connectomes has been an important research goal for the last 40 years [1]. This paper focuses on the methodology of building connectomes. The techniques we develop are tested with the neuron connections in the retina but the proposed methods and their implementations are general and can be applied to any type of connectome [2].

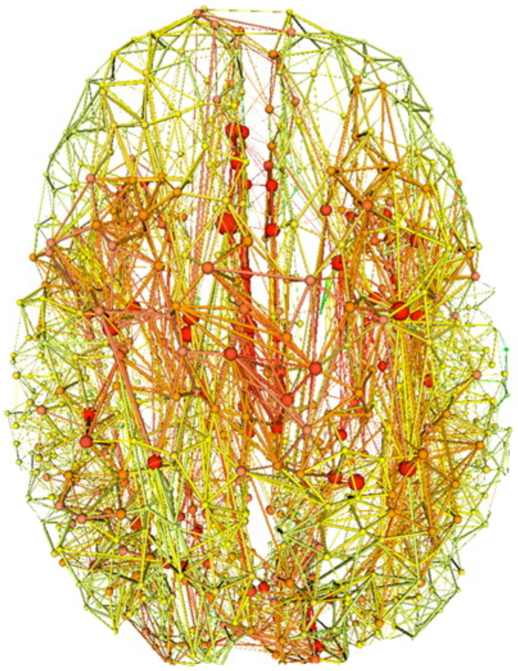


Figure 1: A connectome of a brain represented by a graph. Each vertex represents a brain region and the edges are commissural fibers connecting the vertices they are incident with.

1.1 The Process of Building a Connectome

Recreating the neuron organization of a brain or of parts of a brain requires deriving information from electron or light micrographs of individual slices of brain. The three-dimensional models are constructed of multiple two-dimensional images of such individual slices.

After obtaining the three-dimensional model of the neural network, neuron integration and synapses detection are performed. This paper investigates neuron integration, which involves determining the exact position of each neuron and the boundaries between different neurons. The process of locating the boundaries between neurons has three stages — the *watershed transformation*, from which separate unit blocks are obtained, the merging of these blocks to get segments, and finally the merging of the segments into neurons [3]. Each stage involves different algorithms and manual manipulations, the combination of which successfully handles the problem of neuron integration.

The watershed transformation is the process of partitioning a set of data in morphologically consistent blocks. In the case of neuron integration, a three-dimensional watershed transformation is required. The blocks that are obtained from the watershed transformation are called *supervoxels* — clusters of unit volume elements, *voxels*, that are known to belong to the same neuron.

Since the standard algorithms for performing the watershed transform do not yield satisfactory results, *convolutional networks* are used to partition the model into supervoxels [4]. The convolutional networks are trained by the Maximin Affinity Learning of Image Segmentation and the Boundary Learning by Optimization with Topological Constraints methods for learning [5, 6].

Next, the supervoxels have to be merged to form segments. The following actions are executed repeatedly until a threshold of uncertainty is reached. For each pair of supervoxels the algorithm checks whether they belong to the same neuron depending on some feature conditions. If these conditions are met the supervoxels are merged together into segments.

The criteria for that decision are size, orientation, boundary surface and information about the boundary between them in the three-dimensional model. Larger segments are ultimately obtained from this procedure.

The subsequent step in reconstructing the neuron network consists of merging the unit segments to form neurons. Part of that process is performed automatically — some of the segments are merged into *batches* and after that each batch is regarded as an inseparable unit. For brevity we will refer to these segment batches as segments unless both the terms *batch* and *segment* are touched upon at the same time. The whole neuron cannot be integrated automatically because boundaries between neurons often appear obscure on the images. Therefore manual merging of the batches is required.

1.2 Goals

Merging the different pieces of a neuron manually is often a time-consuming process. This motivates us to extend the set of algorithms managing the connectome construction by developing predictive algorithms that help automate that routine as much as possible.

The first algorithm is aimed at obtaining the neuron fast and with less effort on behalf of the people who are constructing it. When presented with a starting segment, our method branches the neuron out by adding to the starting segment other segments that are likely to be a part of the same neuron. Whether two segments are likely to belong to the same neuron is determined on the base of the brightness of the boundary between them. The growth of the neuron is implemented so that the output of the algorithm can be corrected at any time. We also support a data structure that allows for removing whole branches or tags of branches of a neuron.

A good restriction on the propagation of the neuron is its volume. We introduce a size threshold which is a maximum size restriction on the segments. This added feature is beneficial for controlling the automatic merging of segments into batches.

The described algorithms aim at speeding up the process of manual segment merging.

2 Notation

The set of segments is represented as an undirected weighted graph $G = (V, E)$ (Appendix A), where V is the set of segments and E is the set of unordered pairs (v, w), v and w — segments which share a boundary (Figure 2). The weight of the edge between two vertices is a feature called *affinity* — the probability that the respective segments belong to the same neuron. The affinity is calculated based on the brightness of the boundary between the segments. We call the edges with higher/lower affinity heavier/lighter respectively. For the subsequent merging operations to be executed using only the edges with the highest affinity, we need to perform all the operations on the *maximum spanning tree* of the constructed graph. The maximum spanning tree is computed by negating the weights of the edges and finding the minimum spanning tree via Kruskal's algorithm. A threshold affinity is chosen depending on the type of cell that is explored. All the edges heavier than this threshold are used to construct the graph of connectivity between segments. Each component of connectivity corresponds to a batch of segments. When the affinity threshold is set to a certain value different portions of segments in the neuron are put together. This is the automatic part of the final stage of the process of neuron integration. When the threshold is shifted up/down, some components of the graph are separated/merged, respectively. This is simulated by maintaining a data structure called a *splay tree*, a self-adjusting form of a binary search tree [7, 8].

We obtain the complete neuron by starting from a particular segment and gradually adding other segments to the construction. We call a *selection* the set of segments that are currently indicated as a part of the neuron which is being built. The set of vertices that correspond to the segments in the selection is denoted by Sel.

The explored area of brain is divided into cubical volumes and each volume is processed

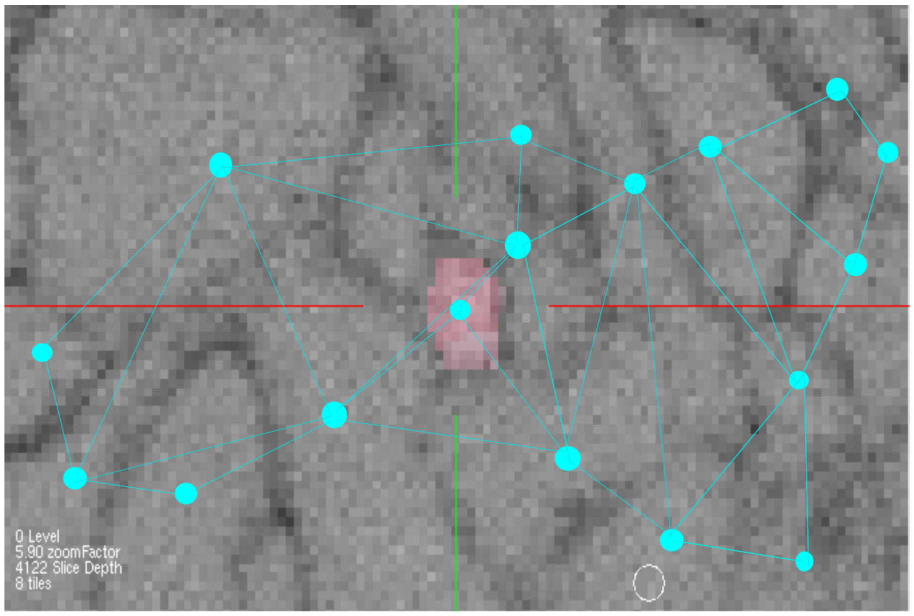


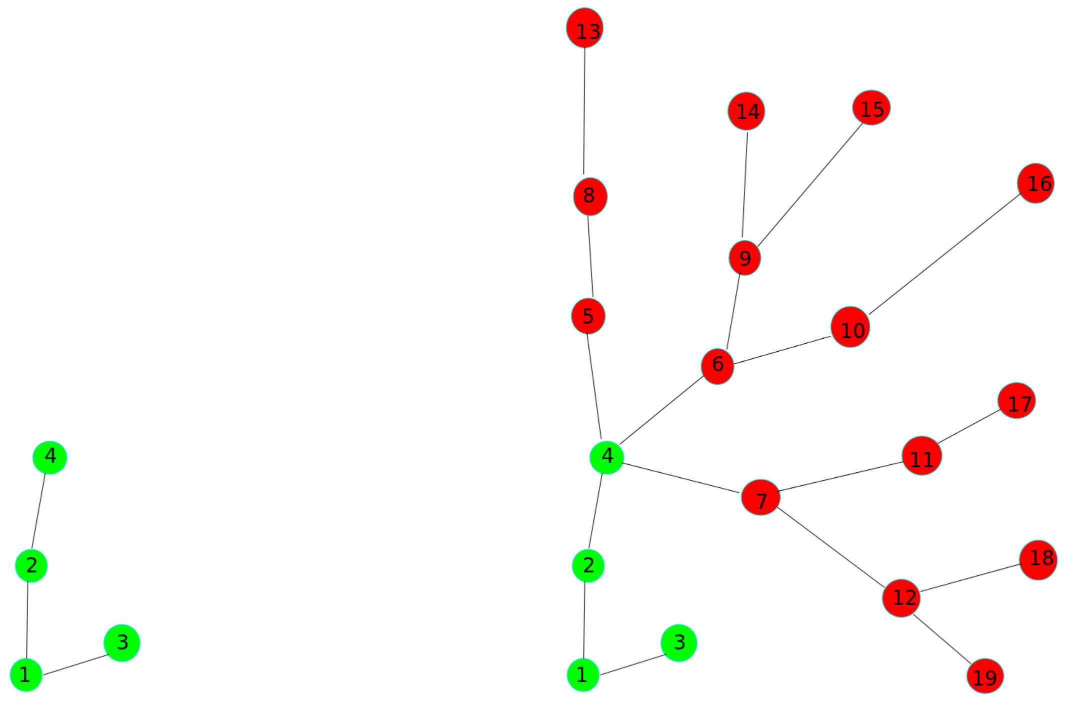
Figure 2: Constructing the graph based on the presence of boundaries between segments. The blue circles represent the vertices and the lines that connect them represent the edges. Each grey area constitutes a segment and the black lines are the boundaries between the segments.

individually in order to speed up the calculations.

3 Automatic Growth of the Selection

By introducing a new dynamically changeable variable, *LocalThreshold*, which reflects the specifics of the explored region, we develop an algorithm that adds multiple adjacent segments to the selection at once. The local threshold is a restriction on the process of adding vertices to Sel, i.e., only the edges that are heavier than *LocalThreshold* are considered in the algorithm. The input of the algorithm is a particular vertex $StartingVertex$ from which the addition begins. A vertex v is defined to be *reachable* from another vertex w if there exists a path $w = v_1, v_2, \ldots, v_k = v$ such that $v_i \in S$ for each $1 \leq i \leq k$. The algorithm is performed locally, meaning that all the added vertices are reachable from $StartingVertex$.

The results from performing the Automatic Growth of the Selection algorithm are displayed in Figure 3. The initial state of the selection is depicted in 3a. The algorithm is initiated with starting vertex 4 and the state of the selection after the addition of the new vertices is shown in Figure 3b.



(a) Stage 1 (b) Stage 2

Figure 3: The effect of the Automatic Growth algorithm in terms of the graph representation of the selection. The vertices are enumerated according to the order in which they are added to the selection when implementing the algorithm.

Let $IncEdges$ be an adjacency list with $IncEdges[0], IncEdges[1], \ldots, IncEdges[m-1]$ being its elements. An element $IncEdges[i]$ is a list of the edges incident with vertex i, denoted by $IncEdges[i][0]$, $IncEdges[i][1], \ldots, IncEdges[i][|N_G(v)|]$. Each element $IncEdges[i][j]$ of that list of edges has two fields. The first field is the weight of the edge $IncEdges[i][j].Weight$. The other field is the vertex $IncEdges[i][j].Neighbour$ that vertex i is connected to via the edge $IncEdges[i][j]$. In the queue q we store the vertices that are already added to the selection. By $q.Front$ we denote a function that returns and deletes the first element in the queue and by $q.Push(v)$ we refer to a function that pushes vertex v at the end of the queue. Algorithm 1 is used for automating the growth of the selection.

Algorithm 1 Automatic Growth of the Selection

Input: $IncEdges$ — a list of all the edges incident with each vertex; $LocalThreshold$ — a local affinity threshold; $StartingVertex$ — a vertex from which to start.

```
Initialize $q = \{StartingVertex\}$

while $q$ is not empty do
    Set $v = q.Front$
    Add $v$ to $Sel$

    for $i = 0 \to |N_G(v)| - 1$ do

        if $IncEdges[v][i].Neighbour$ has not been traversed yet and
        $IncEdges[v][i].Weight > LocalThreshold$ then
            $q.Push(IncEdges[v][i].Neighbour)$
            Mark $IncEdges[v][i].Neighbour$ as traversed
        end if
    end for
end while
```

The worst case complexity of the proposed algorithm is $O(n + m)$, where $n = |V|$ and $m = |E|$, since every vertex and every edge cannot be traversed more than once. Since we are working with the maximum spanning tree of the graph, $m = n - 1$ and the complexity is $O(n)$.

We develop a version of the algorithm with an automatically changing threshold. Let v denote the vertex that represents the selected segment and let $u \in N_G(v)$ satisfy $w(e(v, u)) \geq w(e(v, t))$ for each $t \in N_G(v)$. Then an edge (v, t) is used in the algorithm only if $w(e(v, t)) \geq w(e(v, u)) - c$, where c is a preliminarily selected threshold value.

4 Automatic Trimming of the Selection

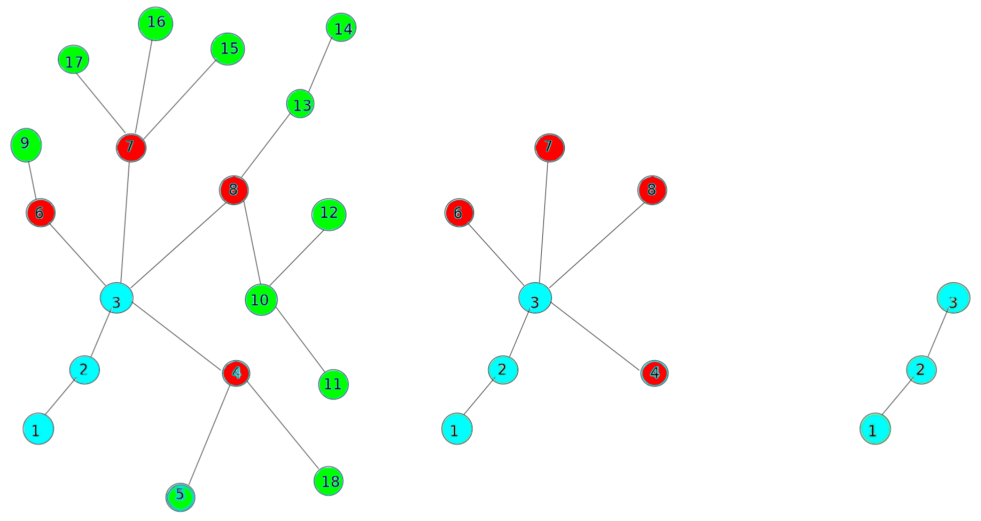
The next algorithm we develop serves for removing the unnecessary segments all at once. The input of the algorithm is a particular segment $StartingVertex$ from which the trimming is initiated. All branches growing out of that segment that have been added after the segment itself are removed from Sel. The algorithm is performed in such a way that all removed vertices are reachable from $StartingVertex$.

Let $Order(v)$ be an injective function on the vertices of the graph such that $Order(w) < Order(v)$ if and only if the segment corresponding to vertex v has been added to Sel after the segment corresponding to vertex w. We implement the algorithm initiating from $StartingVertex$ with the restriction that a vertex w is considered in the search if and only if $Order(StartingVertex) < Order(w)$.

Figure 4 illustrates the result from the algorithm. The initial condition of Sel is depicted in Figure 4a. The Automatic Trimming algorithm is applied several consecutive times with starting vertices $4, 6, 7$ and 8 to obtain the state displayed in 4b. The algorithm is applied a second time with starting vertex 3 (Figure 4c).

The value $Order(v)$ is calculated when vertex v is added to Sel. For this purpose, a global variable $NumberOfVertices$ keeps track of how many vertices have been added and enumerates them. When a new vertex v is added, the value of $NumberOfVertices$ is increased by one and $Order(v)$ is assigned the new value of $NumberOfVertices$. Algorithm 2 describes the method we use for trimming.

The worst case scenario complexity of the algorithm is again $O(n + m)$, where $n = |V|$ and $m = |E|$, since a vertex cannot be removed from the selection more than once. For the aforementioned reasons that simplifies to $O(n)$.



(a) Stage 1 (b) Stage 2 (c) Stage 3

Figure 4: The effect of trimming in terms of the graph representation of the selection. Every vertex of the graph is labeled with its serial number assigned when adding the segment.

Algorithm 2 Automatic Trimming of the selection

Input: $IncEdges$ — a list of all the edges incident with each vertex; $StartingVertex$ — a vertex from which to start.

```
Initialize $q = \{StartingVertex\}$

while $q$ is not empty do
    Set $v = q.Front$

    for $i = 0 \to IncEdges[v].Size - 1$ do

        if $IncEdges[v][i].Neighbour$ belongs to the selection and
        $Order(v) < Order(IncEdges[v][i].Neighbour)$ then
            $q.Push(IncEdges[v][i].Neighbour)$
            Remove $IncEdges[v][i].Neighbour$ from the selection
        end if
    end for
end while
```

5 Introducing a Global Size Threshold

We integrate the option to restrict the volume of the batches of segments in order to improve the control over the automatic merging of segments. Until now, that automatic merging depended only on the global threshold but this second restriction allows for more precise segmentation when loading and when tuning thresholds. We introduce a size threshold indicating the maximum size that a batch of segments can have. The size threshold can be changed dynamically so that respective changes in the batching of all the segments in the volume are reflected.

We call a *root* of a batch a segment belonging to that batch that serves for distinguishing it. We use $SizeThreshold$ to denote the size threshold and $GlobalThreshold$ to denote the overall affinity threshold. Let $Edges$ be the list of all the edges in the maximum spanning tree. Let $Edges[i]$ denote the i^{th} element in the list $Edges$. An element in $Edges$ has the fields v_1 and v_2, which are the vertices that the edge is incident with, and w, which is the weight of the edge. The function $Edges.Size$ returns the number of elements in the list $Edges$. The procedure $Sort(Edges)$ sorts all elements in $Edges$ in decreasing order of their affinities. By $Size(v)$ we denote the total volume of all segments in the batch that vertex v currently belongs to; $Root(v)$ returns the root of the batch that v belongs to. The function $Join(v_1, v_2)$ merges the two batches that vertices v_1 and v_2 belong to into one batch; $Split(edge)$ separates the batch to which $Edge.v_1$ and $Edge.v_2$ belong into two batches by deselecting the edge so that $Edge.v_1$ and $Edge.v_2$ are no longer in the same batch.

When the size threshold is increased, the actions described in Algorithm 3 are implemented. As can be seen from the description, the edges with higher affinity are used first because the higher the affinity of an edge, the higher the probability that the two vertices it connects belong to the same neuron. This method guarantees that the two thresholds are observed.

Algorithm 3 Increasing the size threshold

Input: *Edges* — a list of the edges in the graph; *SizeThreshold* — the new size threshold value; *GlobalThreshold* — the overall affinity threshold.

```
Sort(Edges)

for $i = 0 \to Edges.Size - 1$ do

    if $Edges[i].w \leq GlobalThreshold$ then
        Terminate the process
    end if
    $vertex_1 = Edges[i].v_1$
    $vertex_2 = Edges[i].v_2$

    if $Size(vertex_1) + Size(vertex_2) \leq SizeThreshold$ then
        $Join(vertex_1, vertex_2)$
    end if
end for
```

Whenever the size threshold is increased, all $n-1$ edges, where $n = |V|$, in the maximum spanning tree are traversed. The operation *Join* is implemented with complexity $O(\log_2(n))$ using the data structure slay trees. That makes the overall complexity of the algorithm $O(n \times \log_2(n))$.

An analogous approach is applied when the size threshold is decreased. The method is described in Algorithm 4. Here, the edges are traversed from the lightest to the heaviest so that the ones that are less likely to connect two segments belonging to the same neuron are removed first.

Decreasing the size threshold also leads to traversing all the edges in the maximum spanning tree. Similarly, the operation *Split* in the splay tree is implemented with complexity $O(\log_2(n))$, where $n = |V|$. Again, the overall complexity is $O(n \times \log_2(n))$.

We regard as *optimal* such a distribution of the segments into batches that obeys the size and affinity restrictions and the number of batches is the least possible. It is important to note that the method of choosing which edges to use when joining and splitting batches is *greedy*.

Algorithm 4 Decreasing the size threshold

Input: *Edges* — a list of the edges in the graph; *SizeThreshold* — the new size threshold value.

```
Sort(Edges)

for $i = Edges.size - 1 \to 0$ do
    $Vertex_1 = Edges[i].v_1$
    $Vertex_2 = Edges[i].v_2$

    if $Root(Vertex_1) = Root(Vertex_2)$ and $Size(Vertex_1) > SizeThreshold$ then
        $Split(Edges[i])$
    end if
end for
```

A greedy approach is an approach that, at each step of constructing the solution makes the locally optimal choice. The solution obtained thus is not necessarily the optimal solution, but it is *close to* the optimal. In this case this means that there might exist a selection of edges that leads to a distribution with less batches. In practice this is not a problem because the tasks are completed manually. In addition, finding the optimal distribution would lead to a much higher time complexity.

6 Introducing a Local Size Threshold

We develop an algorithm that, given a local size threshold (denoted by the variable *LST*), finds an affinity threshold such that if the selection started growing automatically using that threshold, it would grow as much as possible without exceeding *LST* in volume.

The most intuitive approach for finding an appropriate threshold is testing all possible values for it and checking which one leads to the best results. However, the affinity threshold is a real number between 0 and 1. Depending on how many digits after the decimal point we regard as significant, we denote by k the number of all possible values for the affinity threshold. We have to iterate through all of them and for each of them to execute the

Automatic Growth of the Selection algorithm to see how big the selection would grow. The complexity of the Automatic Growth algorithm is $O(n+m)$, where $n = |V|$ and $m = |E|$, since each vertex is traversed exactly once and each edge is processed exactly twice (once per both its incident vertices). In this particular case $m = n - 1$ since we are working with the maximum spanning tree of the graph. This makes the complexity $O(k \times n)$. We use $k = 10000$ since we experimentally found it most appropriate to regard the first four digits after the decimal point as significant. Thus the complexity becomes $O(n)$ with a large constant, hence this approach is too slow for the current purposes (the number of vertices in most volumes is approximately a million).

All operations should be performed starting from a segment that is received at the input of the algorithm $StartingVertex$. Therefore, that segment should be a parameter of all functions regarding these operations. In the following lines we assume that the starting segment (respectively vertex) is one and the same and omit it for brevity. Let $size(LAT)$ be the size of Sel after the implementation of the Automatic Growth function with a local affinity threshold (denoted by the LAT variable). In terms of this notation, we need the biggest $size(LAT)$ such that $size(LAT) \leq LST$, and we need an approach to do that in less than linear time.

What helps us here is the fact that the function $size$ is a monotonically decreasing function. To prove that we prove that the bigger the affinity threshold is, the smaller $size(LAT)$ is. Let LAT_1 and LAT_2 be two values for the affinity threshold such that $LAT_1 < LAT_2$. Let $S(LAT)$ denote the set of all segments and edges that belong to the selection obtained by running the Automatic Growth function with a local threshold LAT. An edge $e(v_1, v_2) \in S(LAT_2) \Rightarrow e(v_1, v_2) \in S(LAT_1)$ because $e_w \geq LAT_2 \Rightarrow e_w \geq LAT_1$ where e_w denotes the weight of the edge e. An edge $e(v_1, v_2) \in S(x) \Rightarrow v_1, v_2 \in S(x)$ for any x. In addition, there are not any isolated vertices in a selection obtained by the Automatic Growth algorithm ($v \in S(x) \Rightarrow$ there exists an edge $e(v, v') \in S(x)$). That means that $v \in S(LAT_2) \Rightarrow$

there exists an edge $e(v, v') \in S(LAT_2) \Rightarrow e(v, v') \in S(LAT_1) \Rightarrow v \in S(LAT_1)$ for any vertex v. Since $size(x)$ is the number of vertices in $S(x)$, $size(LAT_1) \geq size(LAT_2)$. $\square$

Because $size$ is monotonically decreasing, there exists a value LAT such that $size(x) \leq LST$ for each $x \geq LAT$ and $size(x) > LST$ for each $x < LAT$ and we can use the technique binary search (Appendix B) to find that LAT.

Calculating $size(i)$ for each i is implemented by simulating the Automatic Growth algorithm without actually adding any vertices to Sel. Thus the complexity of the method becomes $O((n + m) \times \log_2(k)) \subset O(n \times \log_2(k))$, where $n = |V|$ and $m = |E|$, which is a significant improvement over the complexity of consecutive searching. After finding the appropriate value for LAT, the Automatic Growth algorithm is implemented so that the newly added vertices can be added to the selection.

7 Application of the Algorithms Developed

The manual stage of the process of neuron integration is performed via a piece of software called Omni. The software visualizes the volume that the user is working on in two different ways. The first way is by displaying three two-dimensional sections of the volume, one for each combination of two axes — X and Y, X and Z, Y and Z. The second method of visualization is by presenting a three-dimensional model of the selection. Figure 5 shows the user interface of the software. The dark lines in the two-dimensional images represent the boundaries between different neurons and each integral grey area represents a separate segment. If a segment is colored as opposed to left gray, that indicates that this segment belongs to the selection. Figure 6 displays a branch of a neuron seen in the 3D window of the graphical interface. The segments that constitute the branch are colored in different colors so that the segmentation can be seen.

The user interface also includes four different boxes, one for each threshold — the global

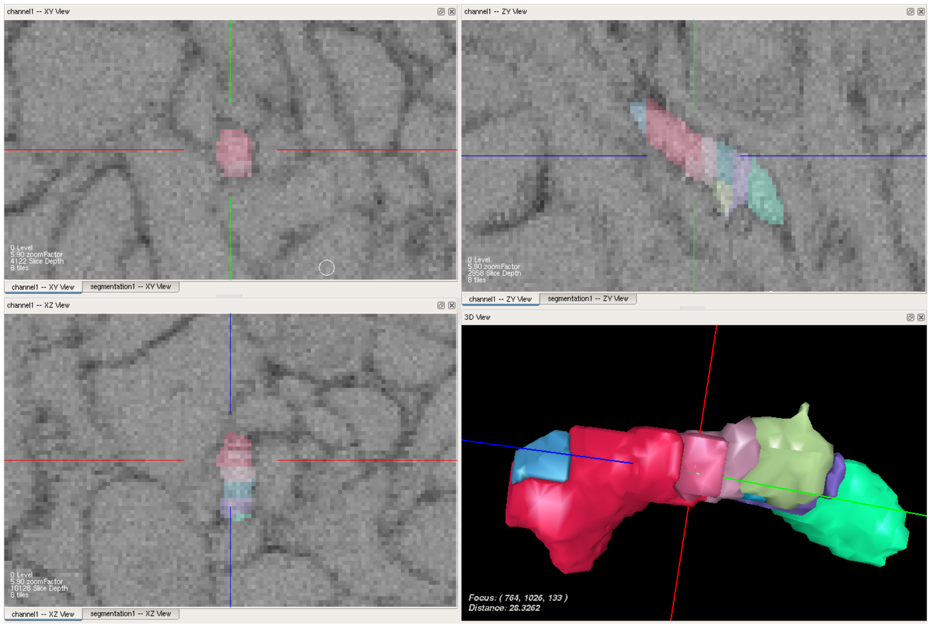


Figure 5: User interface of the software that handles merging the segments. The two visuals on the left and the top one on the right display two-dimensional sections of the volume and the bottom right window presents a three-dimensional model of the selection.

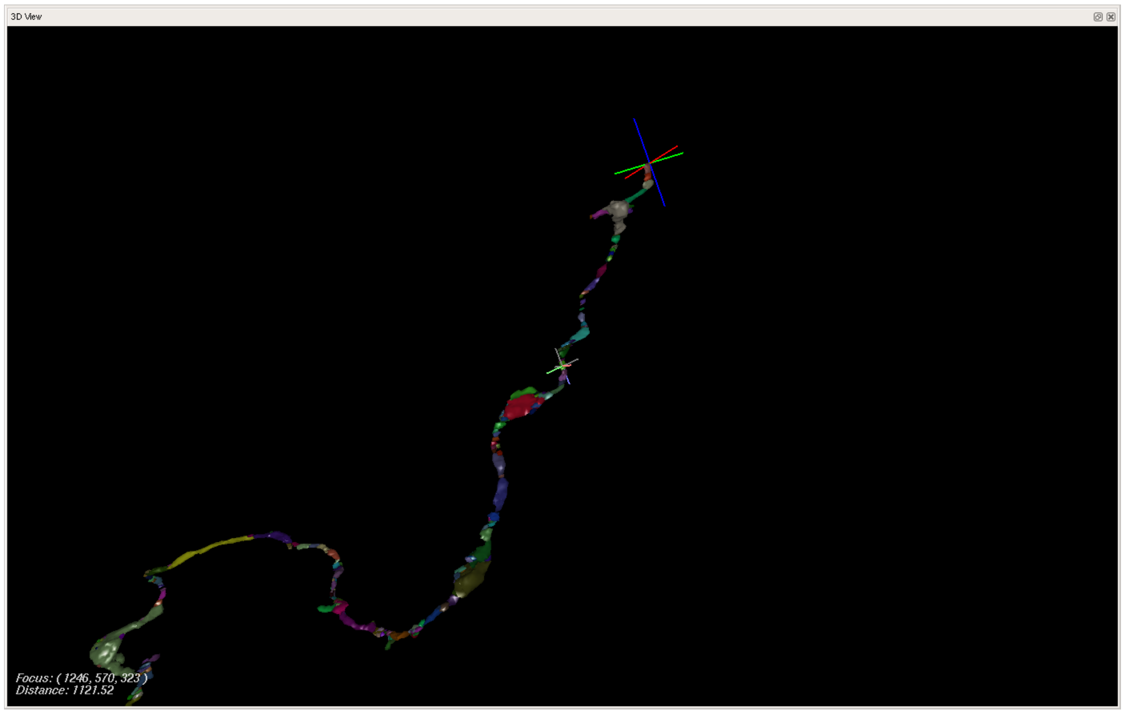


Figure 6: 3D view of a branch in the user graphical interface

and local affinity thresholds and the global and local size thresholds. When entering Omni the global affinity threshold is automatically set at such a value that all segments are separated. The affinity values are in the interval $[0, 1]$, so 1 is an appropriate value for this.

The users usually start from a segment that serves as a base of a neuron and gradually add other segments to it so that in the end the whole neuron is obtained. The user interface has several different modes with multiple tools in each of them.

In order to make the algorithms practically useful we integrate our methods into the existing software. The code of the software is written in C++ using object-oriented programming. We create an add-on to the software which requires establishing all the links between the existing objects and making sense of how to access all the data structures. We also alter the way the graph is stored since our algorithms are more efficient when using an adjacency list rather than when using a list of the edges. In addition, we create a way for the user to access our methods through the user interface, which involves passing the methods through all the levels of data representation in order to get to the rendering and input/output machinery. This integration is essential for the real-time testing of the algorithms.

We create *Automatic mode* — a new mode for the user interface, and develop tools for it. Each tool corresponds to one of the described algorithms. For each of the algorithms that initiate from a particular vertex, the user has to set a value for $StartingVertex$ by selecting a segment.

7.1 Automatic Growth of the Selection

When the Automatic Growth of the Selection algorithm is implemented, the newly added to the selection segments appear colored on the screen. Figure 7 visualizes the result from performing the Automatic Growth of the Selection algorithm.

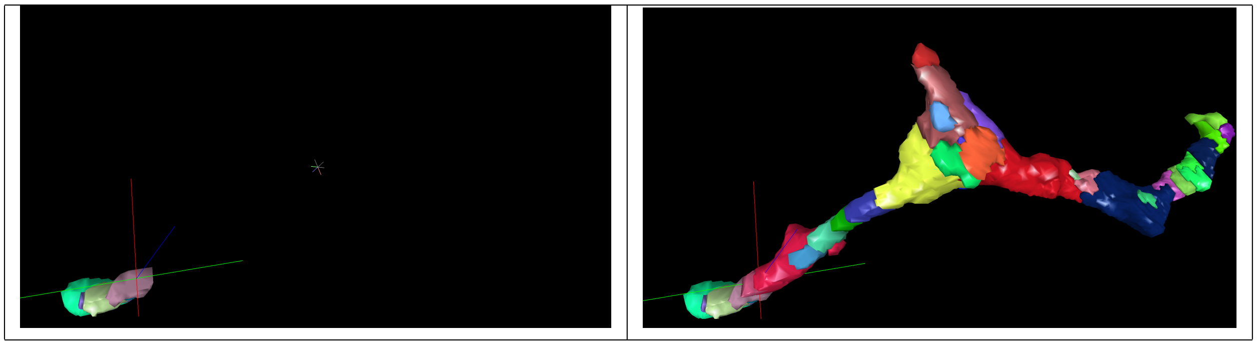


Figure 7: The effect of the Automatic Growth of the Selection algorithm.

7.2 Automatic Trimming of the Selection

The purpose of this algorithm is to allow the users to remove a whole branch at once if they realize that it is actually not part of the neuron after they have added it to the selection. Moreover, if a user accidentally presses some button and unwillingly adds a multitude of segments to the selection, this tool would be useful to fix that mistake.

The changes after the implementation of the Automatic Trimming of the Selection algorithm are displayed in Figure 8.

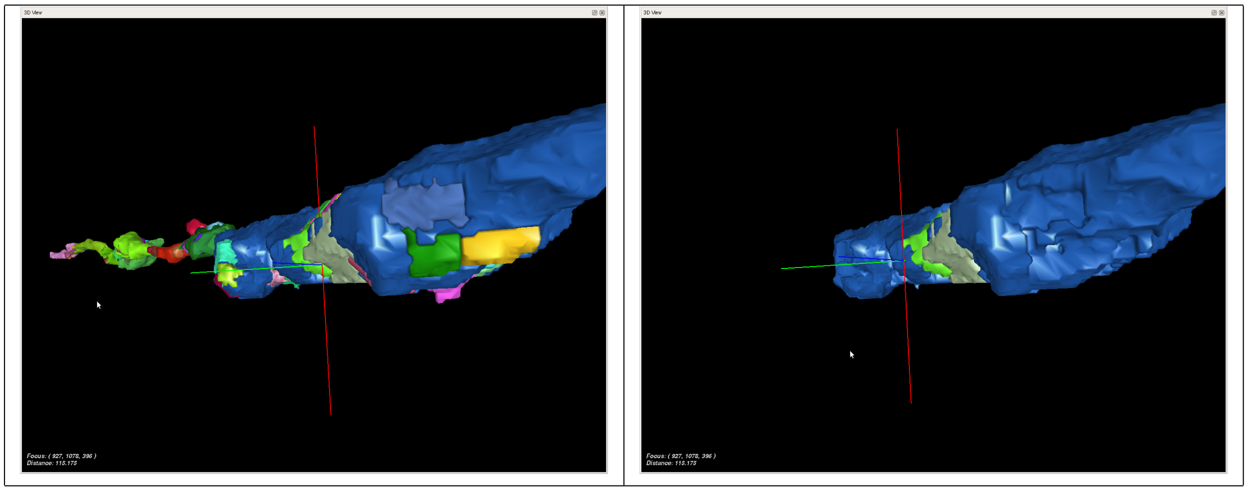


Figure 8: The effect of the Automatic Trimming algorithm.

7.3 Global Size Threshold

When loading a particular volume the size threshold is not defined so there is no restriction on the size of the batches of segments. The user can set/reset the size threshold at any time, which leads to redistributing the segments into batches, recalculating and visualizing the selection.

7.4 Local Size Threshold

The affinities between segments that belong to one and the same neuron vary depending on the type of cell that is explored. On that account it is convenient for the users to not have to think about finding the most appropriate affinity threshold for a particular segment. Instead, they are usually able to give a good approximation for the size of the neuron they are working on. For that reason we develop an algorithm which finds an affinity threshold such that if the selection starts growing automatically using that threshold, it grows as much as possible without exceeding the size threshold set by the user. Running the Local Size Threshold algorithm involves implementing the binary search to find the most appropriate value for the local affinity threshold, then implementing the Automatic Growth of the Selection algorithm with this local affinity threshold and rendering the newly calculated selection on the screen.

8 Discussion

To test the effectiveness of the developed algorithms we have two people working on sample portions of a branch. We use 9 different tasks, each of which are performed with and without the new methods. In order for the results to not be influenced by the difference between the performances of the two executors, each user performs half of the tasks using the new methods and the other half without using them. The results from the study are displayed in table 9.

Task number	Time required to complete the task without the new methods	Time required to complete the task using the new methods
1	2 minutes	2 minutes
2	2 minutes	1 minute
3	7 minutes	6 minutes
4	9 minutes	5 minutes
5	6 minutes	6 minutes
7	3 minutes	1.5 minutes
8	1.5 minutes	1 minute
9	5 minutes	4 minutes

Figure 9: Test data for the performances of two users executing 9 tasks. Comparison of the efficiency of the work with and without the new algorithms in terms of required time.

Applying the new algorithms brings about an overall reduction of the time that completing tasks requires. To estimate the difference that using the new algorithms makes, we calculate the relative decrease in time observed as a result of the methods. We obtain that using the new algorithms the tasks take 72.6% of the time they take when the new methods are not applied. This means we have a 27.4% time improvement. It is important to note that the users are not yet comfortable with working with the new tools, which suggests that in the future the improvement will be greater.

9 Future Development

One of the major challenges in the development of Omni is speeding up the initial loading as well as the loading when one of the thresholds is changed. A good perspective for doing so is dividing each volume into subvolumes. The data (lists of vertices and edges) for each subvolume will be stored separately. An individual splay tree forest will be maintained for every subvolume. Keeping track of the connections between segments from different subvolumes will have to be done in a different data structure outside of the splay tree forests. The type of this data structure will be determined after careful evaluation of the complexity and analysis of experimental data. This approach will speed up the program execution because when a local command is to be implemented, only the subvolumes in which changes occur will be considered. Moreover, the partition of the segments will allow for parallel computing which can decrease the running time between 2 and 16 times depending on the number of cores of the computer processor.

Another idea is investigating different possibilities for branching out from the current segment and presenting several predictions for how the branch should be continued. This algorithm will be implemented by direct exploration of a certain volume around the branch and finding the maximum affinity points.

10 Conclusion

This paper introduces an innovative approach for neuron integration. We have developed a set of predictive algorithms for automatic growth of the selection restricted via a dynamically changeable local threshold — two algorithms using an affinity threshold and one algorithm using a size threshold. We have implemented a linear time solution backtracking algorithm that takes into account both the automatic addition and the addition performed by the user. We have also introduced an overall size threshold restricting the volume of the batches of segments. We have studied the change in performance of users working with the new version of the software. A 27.4% time improvement has been registered.

11 Acknowledgments

I would like to thank my mentor Dr. Sebastian Seung, professor of Computational Neuroscience at the Department of Brain and Cognitive Sciences, Massachusetts Institute of Technology, Aleksandar Zlateski, Michael Purcaro, Matthew Balkam, and Jinseop Kim for providing me with the necessary information and supervising the research. I thank Rachel Prentki, Lindsey Trawinski, Marissa Sorek and Alex Bost for helping me with the software and providing me with test data. I would like to acknowledge my tutor Joseph Dexter for the constructive criticism and the useful advice for the paper. I would also like to thank Dominik Rabiej for the helpful comments on my final draft. I would like to thank the Center for Excellence in Education for sponsoring and the Massachusetts Institute of Technology for hosting the 2012 Research Science Institute, as well as Dr. Andrew Charman, the director of RSI, and Joann P. DiGennaro, the president of CEE, for making it possible for me to conduct this research. I would like to acknowledge St. Cyril and St. Methodius International Foundation and America for Bulgaria organization for sponsoring my participation in RSI.

References

[1] J. G. White, E. Southgate, J. N. Thomson, and S. Brenner. The structure of the nervous system of the nematode caenorhabditis elegans. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 1986.

[2] R. H. Masland. The fundamental plan of the retina. *Nature Publishing Group*, 4:877–886, 2001.

[3] J. B. T. M. Roerdink and A. Meijster. The watershed transform: definitions, algorithms and parallelization strategies. *Fundamenta Informaticae IOS Press*, 41, 2001.

[4] S. C. Turaga, J. F. Murray, V. Jain, F. Roth, M. Helmstaedter, K. Briggman, W. Denk, and H. S. Seung. Convolutional networks can learn to generate affinity graphs for image segmentation. *Neural computation*, 22:511–538, 2010.

[5] S. C. Turaga, K. L. Briggman, M. Helmstaedter, W. Denk, and H. S. Seung. Maximin affinity learning of image segmentation.

[6] V. Jain, B. Bollmann, , M. Richardson, D. R. Berger, M. N. Helmstaedter, K. L. Briggman, W. Denk, J. B. Bowden, J. M. Mendenhall, W. C. Abraham, K. M. Harris, N. Kasthuri, K. J. Hayworth, R. Schalek, J. C. Tapia, J. W. Lichtman, and H. S. Seung. Boundary learning by optimization with topological constraints.

[7] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *AT&T Bell Laboratories, Murray Hall, New Jersey*, 07974, 1982.

[8] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *Bell Laboratories, Murray Hill, New Jersey.*

Appendix A Graphs

A *graph* is an ordered pair $G = (V, E)$ where V is a set of *vertices* and $E \subseteq V \times V$ is a set of unordered pairs (v, w) called *edges*. The edge $e = (v, w)$ is called *incident* with the vertices v and w; v and w are also said to be incident with e. The vertices v and w are *neighbours* if there is an edge $(v, w) \in E$. $N_G(v)$ denotes the set of vertices w such that $(v, w) \in E$. A *weighted* graph is a graph $G(V, E, w : E -> R)$ where every edge e has a *weight* $w(e)$.

An ordered set of vertices $v_1, v_2, ..., v_k$ such that $(v_i, v_{i+1}) \in E$ for each $i = 1, 2, ..., k-1$ is called a path. The *length* of a path is the number k of vertices in it. A graph G is *connected* if for every pair of vertices v, w there exists a path $v_1 = v, v_2, ..., v_k = w$ in G. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if $V' \subseteq V, E' \subseteq E$, and the elements of E' are incident with vertices from V' only. A *connected component* of a graph G is a maximal connected subgraph of G. The connected components partition the vertices of G. A path $v_1, v_2, ..., v_k$ for which $v_1 = v_k$ is called a *cycle*. A graph without cycles if said to be *acyclic*. An acyclic connected graph is called a *tree*. Let $S(E)$ denote the sums of the weights of all the edges in E. A *minimum spanning tree* of the graph $G = (V, E)$ is a tree that is a subgraph $G' = (V, E')$ of G and there does not exist a tree $G''(V, E'')|S(E'') < S(E')$.

Appendix B Binary search

A *binary search* is a search technique that can be applied whenever the search field is a monotonically non-decreasing function of one parameter $f(i)$ such that for each i we have access to $f(i)$. We might be searching for a particular value of $f(i)$ or for the biggest/smallest i for which $f(i)$ satisfies a certain condition. For clarity purposes we will always consider finding the smallest i such that $f(i) \geq p$, where p is a parameter of the binary search. If a specific situation requires finding something else, it can be easily transformed to this pattern. The algorithm proceeds in the following manner:

1. Set boundaries for the parameter i of the function l and r so that $f(l+1)$ and $f(r)$ are respectively the smallest and the biggest values that can be the value we are looking for.

2. If the difference between l and r is less than some predetermined value a which denotes the needed accuracy, terminate the process

3. Find the middle possible value of the parameter i — $m = \frac{l+r}{2}$.

4. If $f(i) \geq p$, set $r = m$, else set $l = m$

5. Go back to step 2

After the execution of the algorithm the answer is kept in r and $f(r)$ is the sought value. The complexity of the algorithm is $O(\log_2 n)$ where n is the number of possible values for the parameter of the function i. The reason for that is that on each iteration the search space is reduced by a factor of 2.