NTNU
Norwegian University of
Science and Technology

Fast Seeded Region Growing in a 3D Grid

Erlend Andreas Lorentzen

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Richard E. Blake, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

1 Problem Description

There is increasing interest in the use of 3D grids of voxels as a tool to model the real world in both graphics and computer vision. A number of standard procedures, for example, contour/surface tracing, region exploration and synthesis of a region from a given set of parameters, are familiar in the 2D world and need to be carried across into the 3D grid. Performance becomes particularly important when processing 3D grids since there is an 8 fold increase in data for each doubling in resolution. This vast amount of data makes it useful to examine ways to optimize the algorithms operating on this data as much as possible. This thesis will examine which data structures are best suited to represent the priority queue in the Seeded Region Growing (SRG) algorithm on consumer computing hardware when segmenting 3D grids. It will also look for ways to modify these data structures to improve performance as well as investigate simple ways of automating a version of the SRG algorithm for 3D grids.

Assignment given: 27. January 2011
Supervisor: Professor Richard E. Blake

Fast Seeded Region Growing in a 3D Grid

Erlend A. Lorentzen

June 2011

2 Abstract

The purpose of this thesis was to examine ways to adapt common 2D segmentation techniques to work with 3D grids. The focus of the thesis became how to automate and improve the performance of region growing in 3D grids. After examining relevant literature and developing a tool to run experiments, a simple automatic region grower for 3D grids was developed. Quantitative performance measures and qualitative analysis of the segmentation results were performed. This algorithm was then used as a baseline for comparison when developing a more advanced region grower for 3D grids based on the seeded region grower (SRG) for 2D grids. This new algorithm was then modified to improve its speed and later extended to allow fully automatic operation by automating the placement of starting seeds. It was found that for the algorithms that were extended to a 3D grid, the main challenge was the resources needed by these algorithms when operating on high resolution grids. It was found that even though there have been steady and rapid improvements in consumer hardware since the original region growing algorithms were used on 2D grids, the very large amounts of data resulting from an extension from surface grids to volume grids requires that special attention is paid to handling resources effectively. It was further revealed that what was considered the best data structures and algorithms for the SRG algorithm when it was first introduced, is not necessarily the best choice on todays computing hardware. Also, the conclusion is drawn that with regards to performance, it is now possible to segment volumes approximately as fast as surfaces were segmented in the early 1990s.

3 Acknowledgements

I would like to thank my supervisor Professor Richard E. Blake at the Department of Computer and Information Science, Norwegian University of Science and Technology for giving me the opportunity to write this thesis and for giving me the freedom to explore the subject matter and related subjects in an unrestricted fashion. I am also grateful to the Stack Overflow online community for their help with specific topics as well as general advice. There have been several long lasting technical problems during the writing of this thesis and I would like to thank Marc and Travis at WiTopia support as well as Kyrre Liaaen and Kristoffer Nes Langemyhr at the NTNU Orakel support desk for making it possible to access needed information during my long stay abroad. Last but not least, I would like to thank my wife Ling Zhang for her support and for unburdening me throughout the process of writing this thesis.

Contents

List of Tables

List of Figures

List of Algorithms

4 Introduction

The main focus of this thesis will be to examine ways of improving the performance of Seeded Region Growing (SRG) in 3D grids. Some attention will also be paid to segmentation quality as well as techniques for automating the SRG algorithm. Most literature on SRG focuses on segmentation quality rather than performance, as well as 2D grids instead of 3D grids. It was therefore viewed as valuable to examine and improve the performance of SRG in 3D grids, especially when taking into consideration that the large increase in data when going from 2D to 3D grids is pushing modern consumer computing hardware to its limits.

It is assumed that the reader has some basic knowledge about common data structures and algorithms, as well as computer graphics and segmentation. The groundwork for understanding the later parts of the thesis will be explained in early chapters, so most people with some basic knowledge in the mentioned areas should be able to use this thesis as a standalone text.

Chapter 5 will define segmentation and describe the principals of region growing. This chapter can be skipped by those who are familiar with these topics. Chapter 6 will examine relevant literature related to Seeded Region Growing and performance. This chapter points out important findings that will be studied in more detail in later chapters. Then, chapter 7 will state which problems the thesis will try to solve, followed by chapter 8 which will describe the tools that were developed and used in order to do experiments. This chapter also explains the methodology used throughout the thesis when measuring performance and segmentation quality. Chapter 9 describes, tests and discusses a simple 3D region grower, which will serve as a baseline for comparisons in later chapters. This is followed by chapter 10 which modifies the original SRG algorithm to operate on 3D grids and examines its performance and segmentation quality. Chapter 11 builds on the previous chapter by looking at ways to improve the performance of the SRG3D algorithm and chapter 12 looks at simple ways of automating the SRG3D algorithm. Finally, chapter 13 summarizes the conclusions of previous chapters and discusses possible avenues for further research.

5 Region Growing

Region growing is a segmentation technique and is the basis for the Seeded Region Growing technique that will be discussed throughout this thesis. Before explaining region growing, let us first define segmentation. One of the earliest attempts at doing so can be found in a paper by Zucker[2] from 1976, called *Region Growing: Childhood and Adolescence*. Zucker defined segmentation as dividing a picture into

regions, where a region is an area whose pixels have a common property, or more formally, a segmentation satisfies the following conditions:

1. $\bigcup_{i=1}^{N} X_i = X$
2. $X_i,\ i = 1, 2, \ldots,\ N$ *is connected.*
3. $P(X_i) = TRUE\ for\ i = 1, 2, \ldots,\ N.$
4. $P(X_i \bigcup X_j) = FALSE\ for\ i \neq j,\ where\ X_i\ and\ X_j\ are\ adjacent.$

In this definition, X represents the whole image, while X_1, X_2, ..., up to X_n represent non-overlapping segments and P is a logical predicate which is true when applied to all picture elements in a given X_i. The first condition states that the segmentation must be complete, i. e. all picture elements belong to a region, here stated by the fact that the union of all segments will result in a complete image. The second condition states that all picture elements within the same region must be connected. The third condition states that the similarity measure is true when applied to all picture elements within the same segment and finally the fourth condition states that the similarity measure will not be true for two different regions at the same time.

There are many ways to categorize segmentation techniques. Adams and Bischof[5] divide segmentation techniques into thresholding, boundary, region and hybrid techniques. Thresholding is a simple technique which works by grouping pixels globally by intensity. Boundary methods look for rapid changes in intensity between regions, while region based methods group pixels that are similar and connected. Fan, Zeng, Body and Hacid[14] use these same categories, but adds a separate category for clustering techniques, which group pixels by features (like intensity and position) in an n-dimensional feature space. A more recent and more broad categorization[10] is to view segmentation techniques as either structural, stochastic or hybrid approaches. Segmentation techniques can also be thought of as bottom-up approaches that start with individual pixels and build regions from these and top-to-bottom approaches that start looking at the picture as a whole and create regions by repeatedly splitting the image into smaller pieces[3]. Looking at these categories, one can view region growing as region-based method which is also a local, bottom-up, structural and stochastic method. It is bottom-up in the sense that it starts with the smallest components (pixels) and gradually builds larger regions from these. It is structural in the sense that it looks at the connectedness of pixels and it also has some stochastic qualities since it can use mean, variance or other statistical calculations to measure similarity.

Region growing is often used in combination with other techniques, or as one of several steps in a segmentation pipeline[10]. It can be seen as a form of thresholding that uses the structure of image segments to confine the thresholding process. Whereas thresholding uses global information in the form of similarities of pixel or voxel intensities throughout an image, region growing uses local information by requiring that each pixel or voxel that belong to the same segment are connected.

The main idea of region growing is to start with a single picture element and gradually grow a region (segment) by including all neighboring picture elements that are deemed similar enough to the starting element. There are many variations of region growing, but the basic steps that are common to all region growing methods are as follows. First, one or more seeds need to be placed in the image that is to be segmented (at least one seed for each segment). The purpose of these seeds is to serve as starting points for the segmentation process. These seeds can be placed manually by the user, or automatically by an algorithm. Automatic seed placement is usually preferred, but automatic placement is not always reliable (or even possible). Each seed is made from one or a small number of connected pixels (or voxels if segmenting a volume). Once the seeds are placed, the growing process can begin. To grow each seed into a region, the region growing process starts by examining each of the pixels or voxels that constitute the neighbors of a given seed, to determine whether these neighbors should be added to the region. For simplicity, the following discussion assumes that each seed consists of one picture element. The definition of a neighbor depends on whether the image is a surface or a volume.

Regular 2D images consisting of a raster of rectangular pixels have two common definitions of neighborhoods. The first is to view each pixel that shares an edge with a given seed as a neighbor. This gives each seed four neighbors. The other common way to define pixel neighbors is to view each pixel that shares an edge or a vertex as a neighbor. This definition gives each seed 8 neighbors (4 edges + 4 vertices). If the seed in question is a voxel placed in a volume, there are at least three possible definitions of a neighborhood, where two of these are in common use. The simplest definition would be to view each voxel that shares a surface with the seed voxel as a neighbor. This results in 6 neighbors, one for each surface of the cube representing the seed voxel. Another way to define the voxel neighborhood is to view each voxel that shares a surface, an edge or a vertex with the seed as a neighbor. This results in 26 neighbors (6 surfaces + 12 edges + 8 vertices).

Now that we have defined the neighborhood where we look for candidate pixels to include in the seeds region, we need to define the predicate that determines whether a pixel should be included or not. This predicate is usually based on some form of statistic calculated from the intensity values of the region and the candidate pixel

and is used to determine how similar the candidate pixel is to the current region. Information that is commonly used to check for similarity includes grey level and color intensities (threshold, mean, variance) as well as more high level properties like texture and geometry (region shape)[18, 12]. The simplest form of predicate is to calculate the difference between the seed pixels intensity and the neighboring candidate pixels intensity. If the difference is within a predetermined threshold, the pixel is included in the region[15]. A predicate that uses more information and is likely to make more intelligent decisions, is to calculate the regions mean intensity and compare the candidate pixel to this mean. When the only pixel in the region is the seed pixel, the regions mean intensity will of course give the same value as the seed intensity. But as more pixels are included in the region, the mean intensity will provide a much more robust form of inclusion criterion than comparing single pixel intensities directly. An even more advanced predicate is to calculate both the mean and the variance of the region[12]. The simplest forms of similarity measures have the advantage that they are fast to compute and simple to implement. The more high level properties use more computation time, are harder to implement and require more information to make good decisions. Some properties like geometry may require a priori knowledge about the image being processed. The advantage of these high level methods is that they can potentially make much more intelligent segmentation decisions.

The basic building blocks of all region growing algorithms are as described above. One or more seeds need to be placed and the neighborhood of these seeds are then checked for candidate pixels to include in the region. Neighbors are included in the current region if they satisfy a given similarity measure. For each new pixel that is added to the region, the step of checking its neighborhood for candidates to include in the region is repeated. This process continues until all pixels have been added to a region.

6 Seeded Region Growing, Volumes and Performance

In 1994, Adams and Bischof introduced a new region growing algorithm called *Seeded Region Growing* (SRG)[5]. This algorithm did not require any tuning, was resistant to noise and was efficient. The algorithm needed to be initialized with a set of seeds, each of which encapsulated what was a feature of interest. Each seed was the first pixel in a set which would grow to become one complete segment. Each iteration of the algorithm consisted of examining each of the unallocated pixels in the image

that bordered at least one of these sets. The pixel that was most similar to its bordering set was added to this set. If a pixel bordered more than one set, it would be added to the most similar set. This process would continue until all pixels had been assigned to a set. This way, regions that shared borders with very similar pixels would grow while regions bordering less similar pixels would stop growing. This gave the SRG algorithm a more global quality than many previous region growers since it always looked at all border pixels in the whole image for each iteration and would grow in the direction of highest similarity. This resulted in an unbiased algorithm that would absorb noise encountered during segmentation and that was also easily extendible to any number of dimensions. Adams and Bischof also suggested ways to automate the seed selection process to make the algorithm fully automatic. One of their suggestions in this regard was to use another automatic segmentation technique as a first step and then use the SRG algorithm to correct the starting segmentation. When measuring the performance of the SRG algorithm, Adams and Bischof found that about 4 seconds was needed to segment an 8-bit image consisting of 256^2 pixels on a DECstation 5000/200.

In 1997, Mehnert and Jackway developed an improved version of the SRG algorithm which they named *Improved Seeded Region Growing* (ISRG)[7]. The reason for developing the ISRG algorithm was that order dependencies were discovered in the SRG algorithm when Mehnert and Jackway tried to use it to segment chromatin within images of cell nuclei. The low resolution of these images combined with a high degree of similarity between adjacent chromatin clumps revealed different results depending on the order in which pixels were processed. One of these order dependencies manifested itself whenever several border pixels had the same minimum delta value. In this case, whichever border pixel was labeled first would be equally correct according to the algorithm. However, the selection of pixel did affect the mean of the region it was assigned to which further influenced the final segmentation. This order dependency was solved by processing all pixels with the same delta value in parallel. Another order dependency the ISRG authors discovered occurred whenever the pixel with lowest delta value bordered two or more regions that were equally similar to the selected pixel. This problem was solved by labeling the pixel as tied and reexamining all these tied pixels after all other pixels had been labeled. The ISRG algorithm was shown to provide more accurate segmentation than the original SRG algorithm, but because the ISRG algorithm was more complex, it required longer processing times. The algorithm used a binary search tree to represent the data structure that stored border pixels according to delta value, but it still required 15 seconds to segment an 8-bit image consisting of 256^2 pixels on a DEC3000 workstation. Based on a paper by Breen and Monro from 1994[6], Mehnert and Jackway suggested that their

algorithm could be optimized by using a splay queue data structure.

In 1998, Grinias and Tziritas developed a system for motion segmentation and tracking based around the SRG algorithm[8]. To reach their objectives, they modified the SRG algorithm and used it to separate a moving object from its background based on a set of images. To improve the performance of the SRG algorithm they used an AVL tree. The average computation time for each image was about 25 seconds on an ULTRA Sparc workstation.

In 2000, Lin, Jin and Talbot created an algorithm called *unseeded region growing* (URG) used for segmenting 3D images[11]. This algorithm was automatic and integrated region-based segmentation with techniques based on adaptive anisotropic diffusion filtering. It was automatic in the sense that it did not require manual placement of starting seeds. Instead, the algorithm would start growing a single region and gradually start new regions if it encountered a pixel that was significantly different from the current region. Adaptive anisotropic diffusion filtering was used to deal with noise. To avoid the order dependencies of the original SRG algorithm, pixels were not ordered according to delta value in a priority queue. This would make it necessary to reorder the pixels when changes were made. Instead the authors relied on a technique from 1999 by Talbot and Beare[9] that did not order the the pixels once and for all, but instead inserted them into a structure that allowed rapid search for the element with the smallest delta value. The original SRG algorithm spent most of its time finding the correct position to insert a border pixel in its data structure and provided rapid access to the the smallest element. URG on the other hand had very rapid insertion of border pixels and instead spent most of its time finding the pixel with the smallest delta value. By doing so, Lin, Jin and Talbot managed to combine the speed of the original SRG algorithm with the improved segmentation results of the slower ISRG algorithm. To achieve rapid access to the pixel with smallest delta, they used a combination of a splay queue and a heap. It should be noted that even though the algorithm was automatic with regards to seed placement, it still required that a user defined threshold was provided so that the algorithm would know when to start a new region. The justification for requiring this manual threshold was that the currently available algorithms for threshold selection were not good enough. It should also be noted that the algorithm was biased towards regions that were discovered early in the process since only the statistics of the region discovered so far were available to make a decision.

In 2010, Hendriks wrote an article called *Revisiting Priority Queues for Image Analysis*[21]. The purpose of this article was to examine which data structures and algorithms were best suited for image analysis on modern computing hardware. Most previous articles looking at priority queues focused on numerical simulation rather

than image analysis. Also, these articles were written more than ten years ago when CPU caches were smaller and the cache management logic was less advanced. Hendriks performed both synthetic and real world tests (grey-wighted distance transform) to compare a variety of data structures. Binary search trees, self balancing trees (like AVL trees, red-black trees and splay trees) as well as heaps were examined. The conclusion drawn from these experiments was that on modern computing hardware, the implicit heap was in most cases faster than all the self balancing trees and also had more consistent behavior across architectures. This contradicted the findings by Breen and Monro[6] 16 years earlier which concluded that the AVL tree and the splay tree performed better than the heap as representations of the priority queue for image analysis algorithms. Hendriks claimed that the reason for these very different conclusions were the advances in CPUs since Breen and Monro did their experiments, particularly the logic that predicts which parts of a program should be in the CPU cache at any given moment.

7 Problem Statement

This thesis will examine which data structure is best suited to represent the priority queue (SSL) in the SRG algorithm by Adams and Bischof[5] on todays consumer computing hardware when segmenting 3D images. It will also look for ways to modify these data structures to improve performance. Further, simple ways of automating the SRG algorithm will be examined.

As shown in chapter 6, most research look at more general cases, they look at synthetic situations or they look at other specific cases that are not directly comparable to the SRG algorithm. Therefore, it would be interesting to see how the findings of chapter 6 relate to the SRG algorithm in particular when segmenting volumes. Performance becomes very important when examining 3D images since there is an 8 fold increase in data for each doubling in resolution. This vast amount of data makes it useful to examine ways to optimize the algorithms operating on this data as much as possible. Looking at the performance of typical current 3D segmentation techniques, they take from 10 seconds to 30 minutes, depending on the algorithms, resolutions and computing hardware used[16, 17, 13].

The next chapter will introduce the tools, data and testing methodology used throughout this thesis.

8 Tools and Methodology

8.1 Segmentation and Visualization Tool

To facilitate testing and visualization of volume segmentation techniques, a tool was developed using C++ as programming language and using the C++ Standard Template Library (STL) and OpenGL as supporting libraries. This tool is a rewritten and significantly enhanced version of a tool developed for an earlier project[19]. Since this tool needed to be able to visualize and modify volume models interactively on mid-range computing hardware, considerable focus was directed towards improving its performance.

The tool stores a 3D grid of voxels by using three STL vectors. Each voxel is represented by a bit field consisting of 32 bits, which can be allocated in different ways. In the following experiments, the bit field was divided into two parts, 16 bits were used for a label and 16 bits for an intensity value. The main point of using the bit field was so that new fields could be added to or removed from the voxel type by redistributing the bits in different ways without changing the size of the voxel. Using bit fields made it simple to modify the voxel format without changing the memory footprint of the data set. A number of techniques were also used to improve the rendering speed of the tool. Firstly it should be mentioned that the tool does not use volume rendering techniques, but instead uses traditional surface rendering, where each voxel is represented by a cube built from 6 square surfaces. To reduce the number of surfaces the tool would need to render, a view mode was added that represents each voxel by only one surface whose surface normal is always pointing towards the camera. This mode is useful once the viewer is a certain distance from the model. To reduce the number of voxels to render, a voxel is only rendered if it has at least one surface that is not blocked by other voxels (this mode should be turned of if transparency is used). To remove the overhead of checking which voxels to render each frame, a list of only visible voxels is built when a model is first created. The tool keeps rendering voxels based on this list until modifications occur, which will cause the list to be rebuilt. Finally, all compiler optimizations were turned on. Combined, all these improvements reduced the time it took to render each frame to about 1/3 of the original rendering time.

8.2 Measurements

To measure the processing times of algorithms, ftime() was used (no longer part of the Single Unix Specification(SUS), but is available on many systems through

"sys/timeb.h"). The measurements have millisecond precision and the average of three runs is used as the final result for each processing time measurement.

8.3 Test System Specifications

All performance tests were run on an Apple Macbook Pro running OS X 10.5.8 with an Intel Core 2 Duo processor clocked at 2.4 GHz and with 4 GB of DDR2 SDRAM clocked at 667 MHz. The graphics card in the test system was an nVidia GeForce 8600M GT GPU with 256 MB of VRAM.

8.4 Methodology

To simplify testing, the tool developed for testing purposes was extended with the ability to generate primitive shapes like cubes and spheres that could be scaled and translated with input parameters. These shapes could then further be combined to create more complex shapes and an optional noise level could be set to simulate the presence of noise and artifacts.

For each segmentation algorithm that was tested, two main areas were of interest. One of these was the performance of the algorithm. Performance was measured as running time in milliseconds and where relevant, memory usage was measured as well. Each algorithm was tested at five different resolutions to see how they scaled and to make it possible to extrapolate what the performance would be at higher resolutions that the currently used computing hardware was unable to handle. Even though the lowest resolutions used in these tests are not used much for actual real-world segmentation tasks, they still provide us with more data points the can be used when doing regression analysis.

The second area of interest when testing segmentation algorithms was the quality of the segmentation result. Only the results of segmenting the synthetic volumes generated by the previously mentioned tool were verified in detail. The reason is that these generated synthetic models have a "gold standard" segmentation that is considered correct. When examining the segmentation results, the segmentation as a whole was looked at as well as how the algorithm handled noise and wether any form of segmentation bias was present. Mathematical evaluation methods like the Hausdorff-distance or the Dice-coefficient[20] were not used in the following segmentation tests. There were two reasons for this decision. Firstly, the volume models used in these tests were mostly very simple. This made it easy to visually verify the segmentation results by using the test tools pseudo color mode to distinguish one segment from another. Secondly, the main focus of the following tests was the

segmentation speed of each algorithm, not the segmentation quality itself.

9 S3DRG: Simple 3D Region Grower

This chapter describes a simple 3D region grower that will serve as a baseline when examining more advanced region growers that will be described in later chapters. The main goal while developing this region grower was to create an algorithm that was as simple as possible while at the same time ensuring that no manual interaction from the user was needed in the form of seed placement or other a priori information. An identical version of the algorithm that only works on 2D grids was also developed in order to perform comparisons. The first section will describe the algorithm in detail, while the second and third section will test the implementation and discuss the results respectively.

9.1 Description

To make seed placement fully automatic and at the same time simple, a single starting seed is always placed in the bottom left front corner (0, 0, 0) of the volume that is to be segmented. The seeds needed to grow the other regions of the volume that are not connected to this initial voxel are discovered as the algorithm progresses. Two different neighborhoods can be used to look for similar voxels while growing a region. The default neighborhood is regarded as each voxel that is surface-connected (6-connected) to the voxel that is currently being processed. As an alternative, the algorithm lets the user specify a surface-, edge- and vertex-connected (26-connected) neighborhood if desired. The predicate used to determine voxel similarity simply calculates the difference between the seeds intensity value and the intensity value of the voxel that is currently being examined. If this intensity value is within a given threshold, the voxels are labeled as belonging to the same segment. The 2D version of the algorithm, used to do comparisons on processing times and memory use, is identical to the 3D version with regards to seed placement (bottom left corner) and similarity predicate. The only difference is that it has a default neighborhood of edge-connected pixels (4-connected) and provides the possibility of specifying an edge- and vertex-connected (8-connected) neighborhood as an option.

Taking a high level view, the algorithm consists of four smaller algorithms. Algorithm 1 iterates through each voxel in the volume and sequentially starts the growing process of each region. Algorithm 2 grows the current region. Algorithm 3 is used to label voxels and update the queue containing voxels that are candidates to join the current region. And finally, algorithm 4 tests the similarity of voxels. Condensed

and simplified pseudo-code versions of each of these algorithms are shown in the following code listings.

Algorithm 1 [*segmentVolumeS3DRG()*] This algorithm iterates through each voxel in the volume and sequentially starts the growing process of each region. It is responsible for calling the function that grows a specific region once it has found an unlabeled voxel.

```
for x = 0 to gridsize − 1 do
  for y = 0 to gridsize − 1 do
    for z = 0 to gridsize − 1 do
      if voxels[x][y][z].getLabel() = unlabeled then
        intensity ← voxel[x][y][z].getIntensity()
        growRegionS3DRG(x, y, z, intensity)
      end if
    end for
  end for
end for
```

The algorithm starts by first placing a seed in the bottom left front corner of the volume (coordinate 0, 0 ,0 in the currently used implementation). It labels this voxel with the first available label (1). A queue is then created to temporarily store each voxel that matches the similarity predicate, so that the neighbors of these voxels can be examined. The seed voxel is placed in the queue and the loop that grows the current region is started. This loop will continue as long as there are candidate voxels in the queue. The loop starts by removing the seed voxel from the queue and examining all its neighbors. If none of the neighbors match the similarity predicate, the main region growing function reaches its end with an empty candidate queue and the current region will consist of only the seed voxel. If on the other hand any of the neighbors match the similarity predicate, these voxels are labeled with the current label and then added to the candidate queue so that their neighbors can be examined. Each iteration of the loop removes the last element from the queue and examines its neighbors. This process continues until the candidate queue is empty which means that the current region is complete i.e. the algorithm has discovered its first complete segment. When this happens the current label is incremented by 1 and the algorithm jumps to the function which is responsible for placing the next seed that will start the growing of the next region. This next seed is placed simply by incrementing the X index of the volume (1, 0, 0) and examining the voxel in this position. If this voxel is already labeled, the X index is incremented again (2, 0, 0) and the next voxel is examined. This continues until an unlabeled voxel is found.

Algorithm 2 [*growRegionS3DRG(x, y, z, intensity)*] This algorithm grows a specific region starting with the seed voxel at coordinate [x, y, z].

$voxels[x][y][z].setLabel(currentLabel)$
$queue.initialize()$
$queue \leftarrow createIndex(x, y, z))$
while not $queue.empty()$ **do**
  $voxindex \leftarrow queue.getLastElement()$
  $queue.removeLastElement()$
  $checkNeighbor(voxindex[x] + 1, voxindex[y], voxindex[z], queue, intensity)$
  $checkNeighbor(voxindex[x] - 1, voxindex[y], voxindex[z], queue, intensity)$
  $checkNeighbor(voxindex[x], voxindex[y] + 1, voxindex[z], queue, intensity)$
  $checkNeighbor(voxindex[x], voxindex[y] - 1, voxindex[z], queue, intensity)$
  $checkNeighbor(voxindex[x], voxindex[y], voxindex[z] + 1, queue, intensity)$
  $checkNeighbor(voxindex[x], voxindex[y], voxindex[z] - 1, queue, intensity)$
end while
$currentLabel \leftarrow currentLabel + 1$

Algorithm 3 [*checkNeighbor(x, y, z, queue, intensity)*] This algorithm labels the current voxel and updates the queue containing voxels that are candidates to join the current region.

if ($x < gridsize$ **and** $y < gridsize$ **and** $z < gridsize$ **and** $x >= 0$ **and** $y >= 0$ **and** $z >= 0$) **then**
  $voxelIntensity \leftarrow voxels[x][y][z].getIntensity()$
  $voxelLabel \leftarrow voxels[x][y][z].getLabel()$
  if (**not** $intensitiesSimilar(voxelIntensity, seedIntensity)$ **or** $voxelLabel = currentLabel$) **then**
    return false
  else
    $volume[x][y][z].setLabel(currentLabel)$
    $queue \leftarrow createIndex(x, y, z)$
    return true
  end if
end if

Algorithm 4 [*intensitiesSimilar(in1, in2)*] This predicate is used to test the similarity of the intensity of two voxels.

```
margin ← 0.02
diff ← fabs(in1 − in2)
if (diff > margin) then
  return false
else
  return true
end if
```

Once an unlabeled voxel is found, this voxel becomes the seed of a new region and this new region is grown in the same way as described above and each of its voxels labeled with the current label (2). This process continues until all the voxels in the volume have been systematically stepped through and labeled.

To provide a simple way of removing noise and to facilitate post-processing of the segmented volume, some additional steps were added to the region grower. Each time a voxel is labeled as belonging to a region, the index of this voxel is added to a list data structure that represents the region that is currently being processed. Once the processing of the region is finished, this list contains the indices of all the voxels in the region. The size of this list is then examined, and if this size is below a given threshold, each voxel in the list is relabeled as noise and the current label number is not incremented so that it can be used by the next segment. It is reasonable to assume that any region that is below a certain size (for example 5 voxels) would be too small to be considered as a separate segment by a human analyzing the results of the segmentation. For each region that is grown, the list with all the voxel indices that represent that specific region is added to another list data structure. This list represents the completed segmentation of the whole volume, where each of its elements represent a single segment. Storing all this data requires more memory, but it saves CPU time for some tasks since any segment can be looked up in the list structure without performing a search on the labeled voxels in the volume. This speeds up selective visualization of segments as well as classification and other post-processing tasks considerably. The additional data structure that stores each segment for later use (bottom layer in figure 1) is not used during the following tests that analyze the processing times and memory use of the S3DRG algorithm. Only the structure that temporarily stores the indices of the voxels of the current region which is used to uphold the minimum allowed region size (noise filter) is activated. The reason is that the additional data structure for storing each region for later use is not necessary to segment the volume (label each voxel), it is just an optional piece of

functionality that simplifies selective visualization and other post-processing tasks.

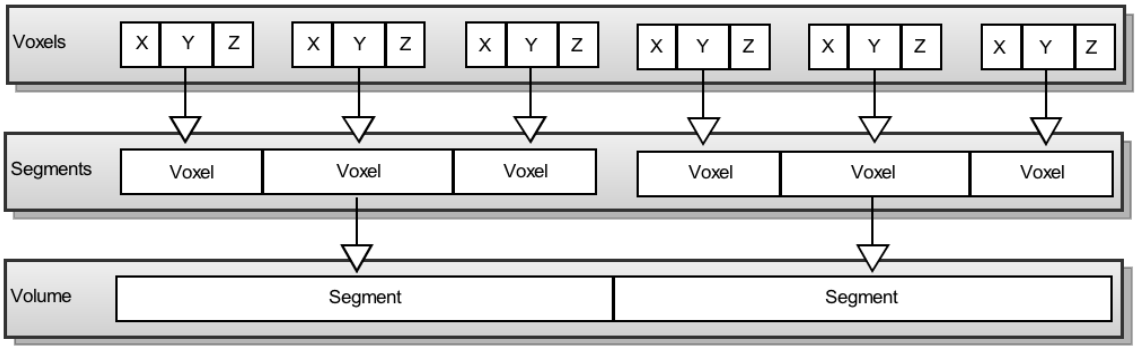

Figure 1: List data structure representation of segments

9.2 Results

An implementation of the S3DRG algorithm and its 2D equivalent (S2DRG) will now be run on various test data. First, the implementation will be run on some very simple low resolution test data to show how pseudo-color mode visualizes the segmented regions and to demonstrate how the 6- and 26-connected neighborhoods for the S3DRG algorithm differ from the 4- and 8-connected neighborhoods of its 2D counterpart. Following this are comparisons running the S3DRG implementation on two sets of similar data, but with different resolutions, then running the S2DRG algorithm on the same data. The purpose of this is to see how the algorithm scales for both 3D and 2D grids respectively. Finally, the S3DRG implementation will be run on a medical data set to see how it behaves on real-world data that is acquired from a 3D scanner.

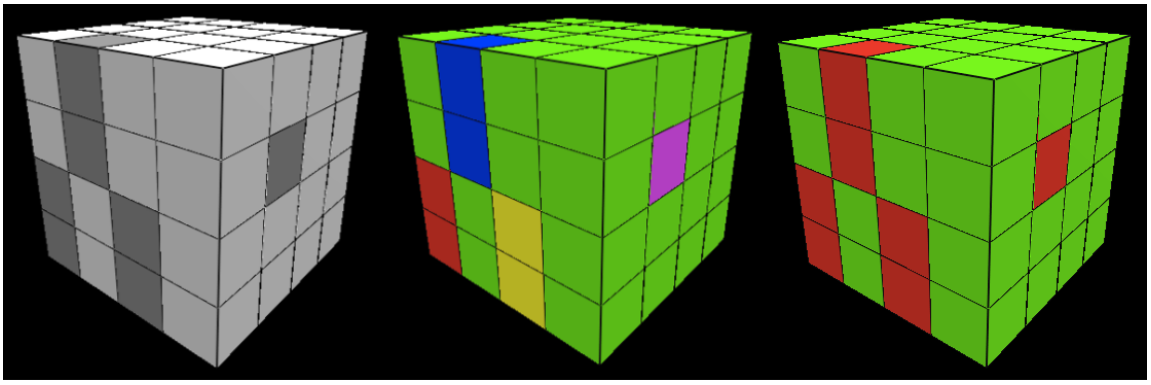

Figure 2: S3DRG segmentation of a simple low resolution volume. The leftmost image shows the original volume before segmentation while the middle and right images show the result of segmenting the volume with 6- and 26-connected neighborhoods respectively.

The first test of the S3DRG algorithm shows the result of running the implementation on a 4^3 volume with only two different grey-level intensities. Figure 2 (middle) shows what happens when using a 6-connected neighborhood. The color-coded volume of voxels shows that with a 6-connected neighborhood, the result is a volume consisting of five separate segments (red, green, blue, yellow and purple). Figure 2 (right) also shows that when using a 26-connected neighborhood, the result is a volume consisting of only two separate segments (red and green), since all the voxels belonging to each of the two different intensities are connected by either a surface, edge or vertex.

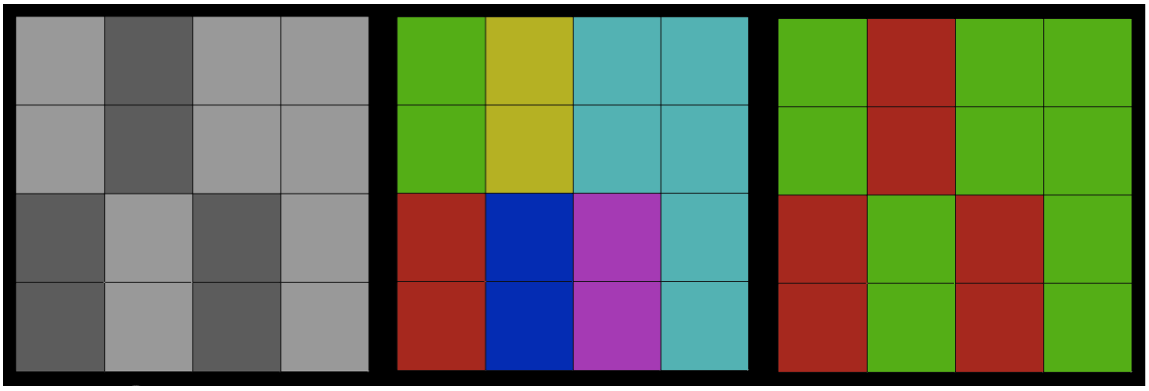

Figure 3: S2DRG segmentation of a simple low resolution surface. The leftmost image shows the original image before segmentation while the middle and right images shows the result of segmenting the image with 4- and 8-connected neighborhoods respectively.

Extracting the first slice of the 4^3 volume shown in figure 2 and running an implementation of the S2DRG algorithm on this slice gives us the results shown in figure 3. Using a 4-connected neighborhood divides the surface into 6 separate segments (red, green, blue, yellow, purple and cyan), and using an 8-connected neighborhood divides the surface into two separate segments (red and green).

Now that we have confirmed that both the S3DRG and S2DRG algorithms give the expected results when segmenting simple low resolution volumes and surfaces, higher resolution volumes and surfaces will be used to test the performance of these algorithms. Both the S3DRG algorithm and the S2DRG algorithm will be run using two different neighborhoods and 5 different resolutions.

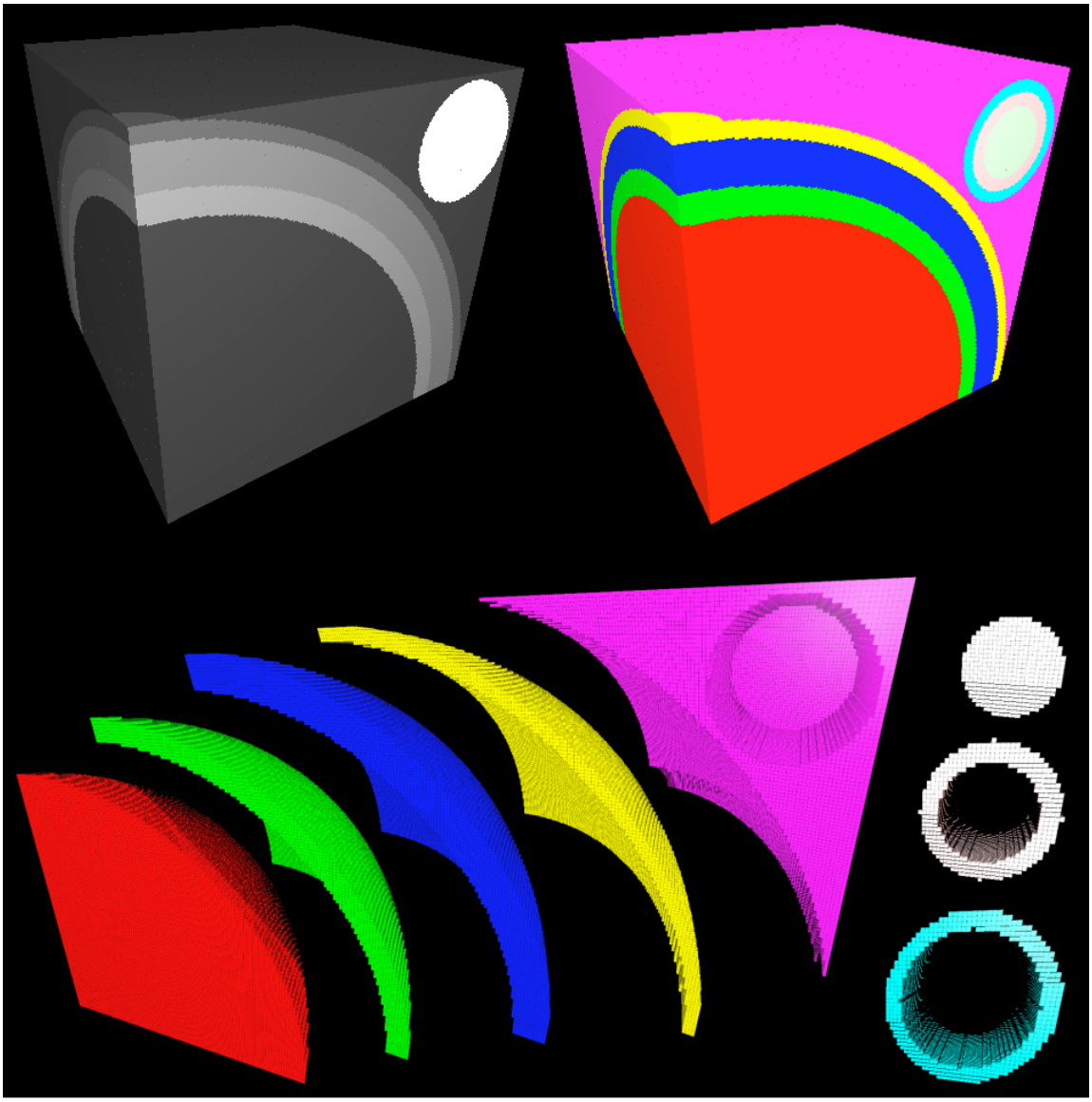

Figure 4: S3DRG segmentation of a simple high resolution volume of 256^3 voxels. The upper left volume shows the original image before segmentation while the upper right volume shows the result of segmenting the volume with a 6-connected neighborhood.

The right part of figure 4 shows the result of running S3DRG with a 6-connected neighborhood on the volume in the left part of the figure. In this case, both the

6-connected and the 26-connected neighborhoods would give the same segmentation. Figure 5 shows the result of running S2DRG on the first slice of the volume in figure 4 with a 4-connected neighborhood. The results of the two algorithms are in this case identical for the first slice of the volume.

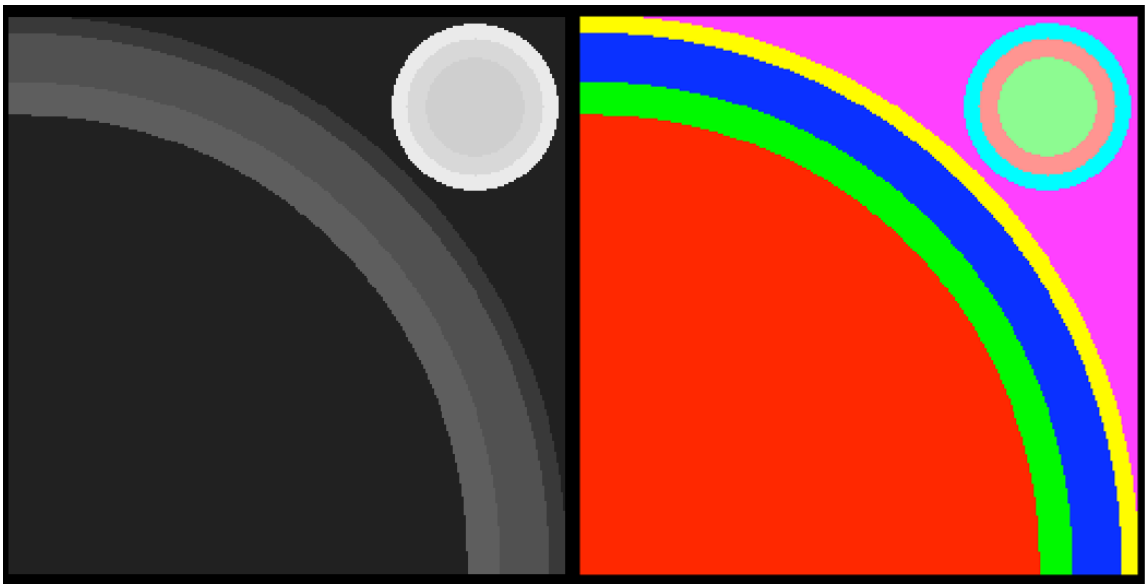

Figure 5: S2DRG segmentation of a simple high resolution image of 256^2 pixels. The left image shows the original image before segmentation while the right image shows the result of segmenting the image with a 4-connected neighborhood.

	Resolution				
Algorithm and neighborhood	32^3	64^3	128^3	256^3	512^3
S3DRG (6-connected)	47 ms	358 ms	2660 ms	20946 ms	183093 ms
S3DRG (26-connected)	79 ms	543 ms	4190 ms	33847 ms	234079 ms
	Resolution				
Algorithm and neighborhood	32^2	64^2	128^2	256^2	512^2
S2DRG (4-connected)	5 ms	20 ms	56 ms	220 ms	934 ms
S2DRG (8-connected)	5 ms	21 ms	59 ms	228 ms	968 ms

Table 1: Running times of S3DRG and S2DRG with two different neighborhoods each and at five different resolutions.

Table 1 shows the processing times when running the S3DRG and S2DRG algorithms on the same model at five different resolutions and with two different neighborhoods. Each of these tests were run three times, and the average of the three runs was used as the actual result of each measurement. Looking at the S3DRG algorithm with a 6-connected neighborhood, for each doubling of voxel resolution, the processing time required to complete the segmentation is multiplied by about eight. Going from a processing time of just 47 ms at a resolution of 32^3 to a processing time of more than 3 minutes with a resolution of 512^3 is a very noticeable difference, especially if the segmentation needs to be done in realtime. However, this increase in processing time is as expected since each doubling in voxel resolution means there are 8 times more voxels to process. This is confirmed by looking at the left graph in figure 6 which clearly shows that the processing time of the S3DRG algorithm grows linearly in step with the growth in number of voxels. Looking at both table 1 and the graph in figure 6 it is interesting to note that using a larger neighborhood for the S3DRG algorithm, results in a very significant increase in processing times. Checking 20 extra neighbors for each region voxel increases the processing time by a factor of approximately 1.6. The results of the S3DRG algorithm are mirrored to some extent in the results of the S2DRG algorithm, since its processing time also grows linearly with respect to the number of pixels it needs to process, and increasing the size of its neighborhood increases its processing time. Since the S2DRG algorithm processes a surface instead of a volume, a doubling of resolution only leads to a four fold increase in pixels versus an eight fold increase for voxels. Since the S2DRG algorithm has less data to process and has a smaller neighborhood, it uses much less processing time than the S3DRG algorithm as shown in table 1 and the right graph in figure 6. When comparing resolutions that give the two algorithms an equal amount of data to process, their processing times are within the same order of magnitude.

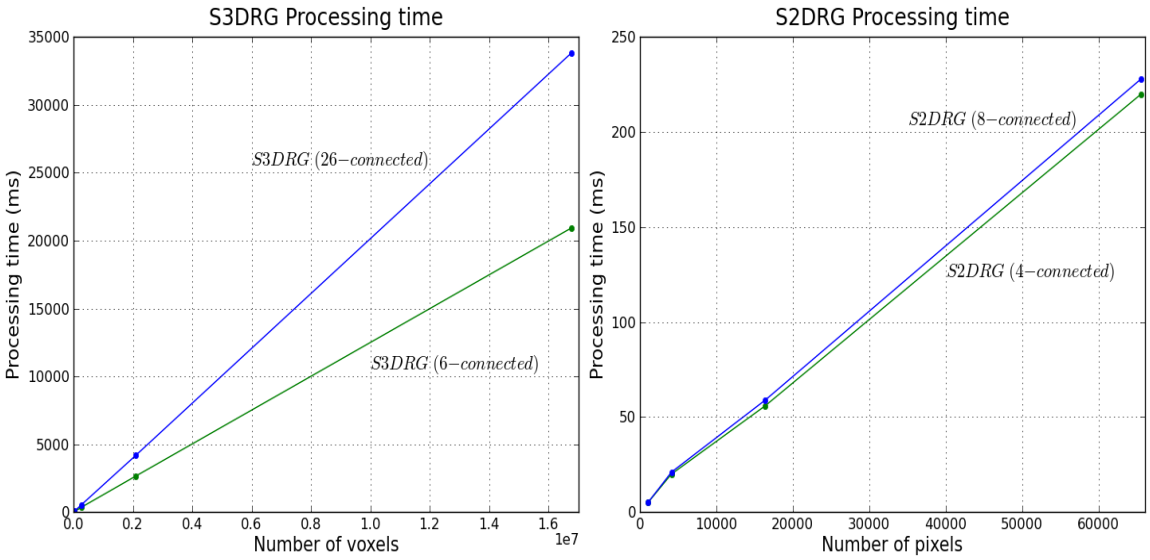

Figure 6: The left graph shows how the processing time of the S3DRG algorithm increases in step with the number of voxels. The right graph shows how the processing time of the S2DRG algorithm increases in step with the number of pixels. Both graphs were created from the data in table 1.

Table 2 shows the memory used by the S3DRG algorithm during segmentation of the volume in figure 4 with two different neighborhoods and at four different resolutions. The table shows the memory used by the data structures of the algorithm itself, including the data structure storing the indices of the voxels in the current region that is used for noise filtering. The memory used to store the volume itself or the memory used to store segments for later use as illustrated in the bottom layer of figure 1 is not shown. The main source of memory allocations in the S3DRG algorithm is the list data structure that stores the indices of voxels that constitute the current region, which is used to filter noise. The size of this list will increase as a region is grown and large regions will naturally result in a large list. Looking at table 2 and figure 7 it is clear that the memory used by the algorithm increases faster than the increase in data caused by an increase in volume resolution. Using regression analysis, the best fit for the growth in memory for the S3DRG algorithm when compared to the increase in data is to use a quadratic model. Extrapolating from the measurements made when using a resolution of 256^3, the memory requirement when using a 6-connected neighborhood and a resolution of 512^3 should be about 3.7 GB and when using a 26-connected neighborhood with the same resolution, the

memory requirement of the algorithm is about 9.9 GB. Considering that the memory requirements of the S3DRG algorithm (when using the noise filter) quickly outgrow the memory requirements of storing the volume itself, it is clear that using the algorithm with the noise filter does not scale well with regards to memory. The memory requirements of the S2DRG algorithm have been omitted from table 2 and the graph in figure 7 since its memory requirements are negligible at such low resolutions (256^2 and lower). The same is true for the S3DRG algorithm when the noise filter is turned off.

Algorithm and neighborhood	Resolution			
	32^3	64^3	128^3	256^3
S3DRG (6-connected)	$<$ 1 MB	$<$ 1 MB	5 MB	72 MB
S3DRG (26-connected)	$<$ 1 MB	1 MB	7 MB	176 MB

Table 2: Memory used by the S3DRG algorithm with two different neighborhoods and at four different resolutions when using the algorithms noise filter.

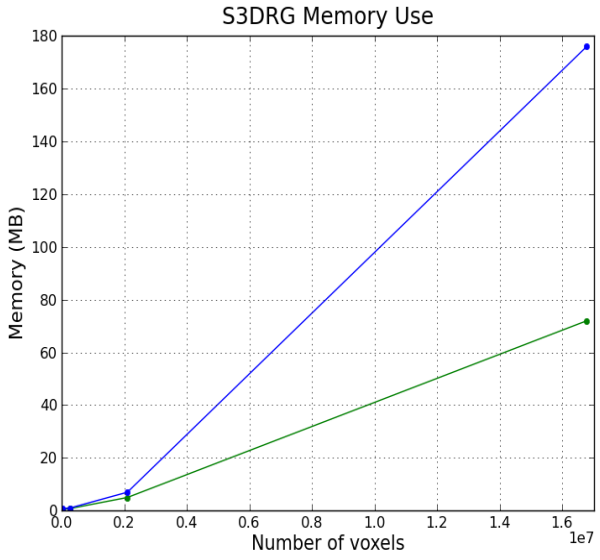

Figure 7: The graph shows how the memory use of the S3DRG algorithm increases compared to the number of voxels when using the noise filter. The graph was created from the data in table 2.

The volumes that have been segmented so far have been symmetric and regular and can be categorized as manufactured objects[12]. The next segmentation tests will use medical objects, which can be categorized as natural objects. Natural objects are created by processes that create a great deal of variety and tend to be less predictable and less constrained than manufactured objects. Because of the nature of such objects, they are often more difficult to segment, because the transitions between regions may be less distinct and it could be less clear what would constitute a correct segmentation of such objects. Another distinguishing factor with the following medical data is that it was acquired from a physical scanner, so the resulting volume could contain noise and artifacts that were not present in the previous volumes. The volumes and segments shown in figure 8 come from a medical scanner and depict a brain. Thresholding was applied to the volume before doing region growing to reduce the amount of data the region grower would need to process.

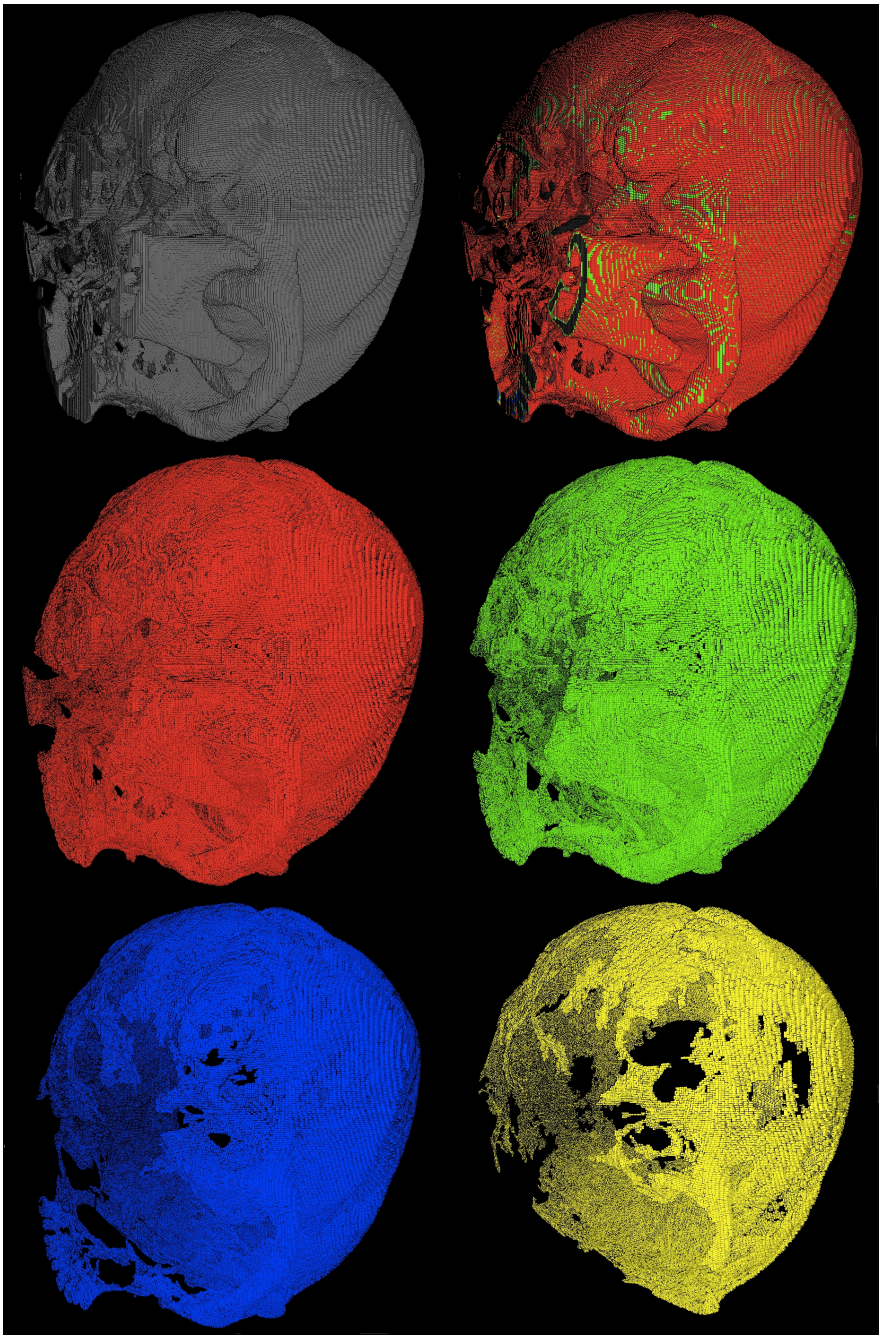

Figure 8: These images show the result of segmenting a 3D scan of a brain with S3DRG with a 26-connected neighborhood. The image in the upper left corner shows the original volume before segmentation. The image in the upper right corner shows the segmented version of the original volume. The next four images show the four largest segments individually without the other parts of the volume.

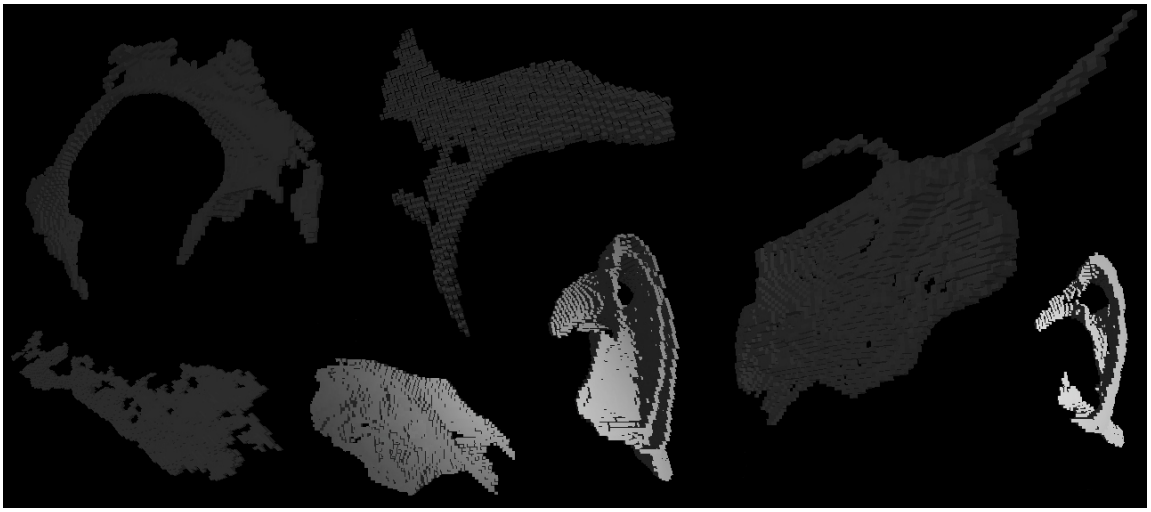

Figure 9: These images show various small segments resulting from the segmentation of the volume in figure 8.

9.3 Conclusions

Looking at the results in the previous section, it is clear that the S3DRG algorithm has the ability to satisfy the four conditions required for a volume to be a segmentation, as described in the chapter on region growing principals. However, whether it satisfies these conditions depends on whether the minimum region size is set to 1 or greater than 1. If the minimum region size is set to 1, regions of size 1 are allowed which will result in all voxels being part of a normal region and no voxels will be labeled as noise. In this case, all the conditions for a volume to be a segmentation are satisfied, since the union of all the segments equals the whole volume, all voxels that are part of the same region are connected and satisfy the similarity predicate, and in addition to this, the union of two different but adjacent regions will not satisfy the similarity predicate. If on the other hand the minimum region size is greater than 1, this could result in some groups of voxels being labeled as noise. These groups of voxels are not necessarily connected and do not necessarily satisfy the similarity predicate, although they do have the same label (noise) and in that sense belong to the same segment.

One of the most important advantages to the S3DRG algorithm is that it bypasses the need to place seeds either manually or by finding suitable seed placement locations by doing any kind of preprocessing on the volume. Placing seeds manually

requires a tool that has this capability and it is also a major administrative burden if there are many regions as well as requiring a user with some specific domain knowledge. Creating an interface to place seeds in a volume is much less straight forward than an interface for placing seeds on a surface. When placing a seed pixel in a 2D image, the whole image is visible making the process relatively simple for the user. However, when placing a seed voxel in a volume, most of the voxels in the volume will be obscured by other voxels at any given time, since most displays only give a 2-dimensional projection of the 3-dimensional data. This can be solved by using transparency, slice-by-slice viewing of the volume and/or other techniques, but the result is still likely to be less intuitive for the end user than it is in the 2-dimensional case. For this reason, automatic seed placement can be said to be even more important in 3D images than it is in 2D images.

The S3DRG algorithm itself is very simple and operates on the most simple kind of volume representation (compared to octrees etc.), namely a 3D grid of voxels. Since it treats the volume as one entity and keeps all the volume data in memory, it does not need to handle shuffling data back and forth between primary and secondary storage. This simplifies the implementation of the algorithm and increases its processing speed considerably. Another important factor when processing the whole volume as one entity instead of looking at each slice individually is that the region growing techniques used for 2D images can be applied directly to 3D images by just redefining the algorithms neighborhood to include the additional neighbors that are the result of having one extra dimension. Techniques that process each slice of the volume separately needs intelligent ways of propagating the seed from one slice to the next since regions that are disconnected in 2 dimensions could be connected in 3 dimensions[20].

Since the S3DRG algorithm uses a queue data structure instead of the system stack to store the indices of voxels that are candidates for addition to the current region, it is not sensitive to the problems of using the stack. Using the system stack would in some ways result in a simpler algorithm, but would also require that the algorithm was aware of the maximum size of the candidate voxel index data in advance, or continually resizes the stack when needed to prevent a stack overflow during segmentation. This problem is much more significant when processing 3D images than when processing 2D images, because of the 8 fold increase in data for each doubling of volume resolution. When using a similar algorithm to S3DRG, resolutions above 42^3 have been shown to cause a stack overflow when using a default stack size on two commonly used operating systems[19]. A stack based algorithm would also be slower[19] and as will be explained later, it would have an impact on the amount of data available to the similarity predicate when using a more complex

predicate than a hard threshold.

As shown in table 1 and in the left graph in figure 6 the processing time of the S3DRG algorithm grows linearly with the growth in data, so it scales acceptably with regards to processing time. Looking at the processing times across many resolutions, part of what is limiting the processing time of the algorithm is that it has a very simple similarity predicate. The time it takes to calculate the similarity predicate is not affected by the size of the region being processed. It should also be mentioned that the rapid increase in computing power predicted by Moore[1] makes it possible to run the S3DRG algorithm in a reasonable amount of time on consumer hardware today, whereas it a decade or more ago would have required a very expensive workstation or even special purpose hardware.

Regarding the disadvantages of the S3DRG algorithm, there are two major problems that prevent this algorithm from being a viable option for general purpose segmentation of volumes. One of these is the bias introduced by growing each region completely before moving to the next region[18]. Suppose a volume consists of three candidate regions, A, B and C, where region B is situated between regions A and C. Lets further suppose that region B has an intensity that is similar enough to be a candidate for inclusion in both regions A and C, while region A and C themselves are too different to satisfy the similarity predicate and will be regarded as two separate regions. In this scenario, it becomes clear that the region that is grown first will dominate the growth process. If the first seed happens to be placed in region A, then region A will absorb region B completely. If on the other hand the first seed happens to be placed in region C, then region C will completely absorb region B. In other words, when performing segmentation with the S3DRG algorithm, the result depends on the order in which each region is grown as shown in figure 10. If the first seed is always placed in the bottom left front corner of the volume (which is the case for the current implementation of the S3DRG algorithm), the segments in this area of the volume will dominate the growth process. This is clearly a very undesirable property. The problem is further compounded by the fact that the algorithm operates in a 3D grid, which has very large amounts of data at reasonable resolutions giving more possible candidates for seed placement, as well as there being more directions to grow from/to.

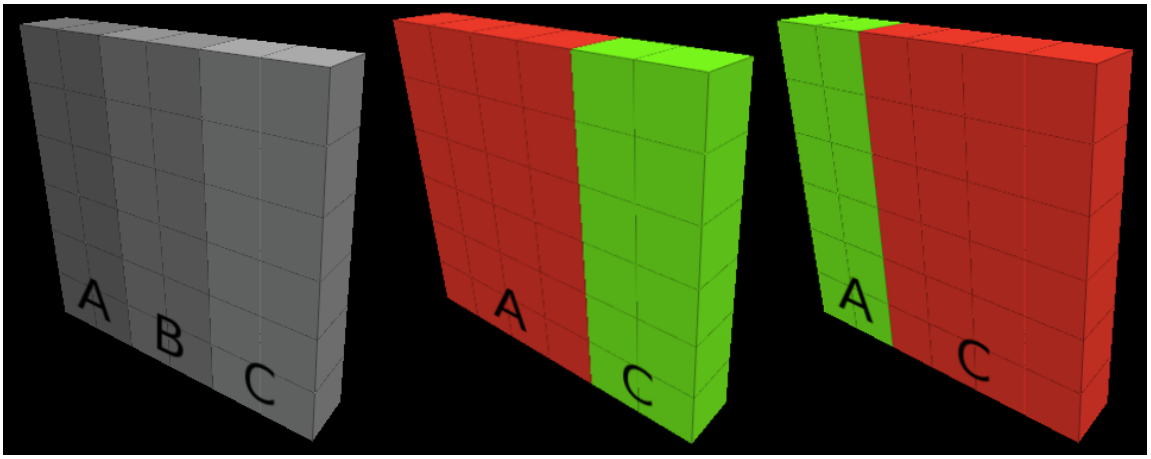

Figure 10: The leftmost image shows the original volume before segmentation with regions A, B and C. The middle and rightmost images show two segmentations, biased towards A and C respectively.

Another major problem with the S3DRG algorithm is that its memory consumption grows quadratically with respect to the growth in volume data when using the algorithms noise filter as shown in table 2. The very high memory requirements make the noise filter mostly unusable on consumer hardware at resolutions of 512^3 and above. Since a resolution of 512^3 is perhaps currently one of the most commonly used resolution in the real world (lower resolutions retain too little information, while higher resolutions require too much memory), the usefulness of the S3DRG algorithm drops considerably when using the noise filter unless some form of preprocessing is performed on the volume. To be able to segment the medical volumes shown in figure 8 and 9 with a resolution of 512^3 voxels, the volume was thresholded before the S3DRG algorithm was applied to reduce the amount of data. Even though the S3DRG algorithms memory requirements could be acceptable in a 2D grid, these requirements quickly become very hard to accommodate when using a 3D grid because of the massive amounts of data such grids contain, even at relatively low resolutions. The setting controlling the minimum region size can of course be turned off, which will allow high resolution volumes to be segmented with very little memory use. This will however frequently lead to many tiny undesirable segments because of the algorithms simple similarity predicate which is based on a hard threshold.

Although the simplicity of the predicate used by the S3DRG algorithm was mentioned as an advantage because of the very low computational cost of calculating it (as well as it being unaffected by region size), this predicate may prove to be too sim-

ple for many segmentation tasks. Although this predicate can be used on real world data[15], there are situations where a more powerful and flexible predicate would be needed. Replacing the currently used predicate with one that uses a different similarity measure, for instance the difference between the voxel under examination and the mean of the current region, would be very simple since all the information is already available. All that would be needed would be the intensity values and the number of voxels in the current region, both of which are available in the list structure that stores the indices of the voxels belonging to the current region. If this predicate were implemented, it would however reveal a potential weakness with the S3DRG algorithm related to the queue that stores the voxels whose neighbors are candidates for addition to the region. As mentioned earlier, the S3DRG algorithm uses a queue instead of the system stack to prevent running out of stack space during execution. For simplicity, this queue data structure uses a stack (LIFO) implementation internally, but since it has access to main memory like the rest of the algorithm, it will not run out of stack space until the algorithm runs out of main memory. Since this queue behaves like a stack, the last element inserted is the first element to be removed, which results in a growth pattern as shown in figure 11.

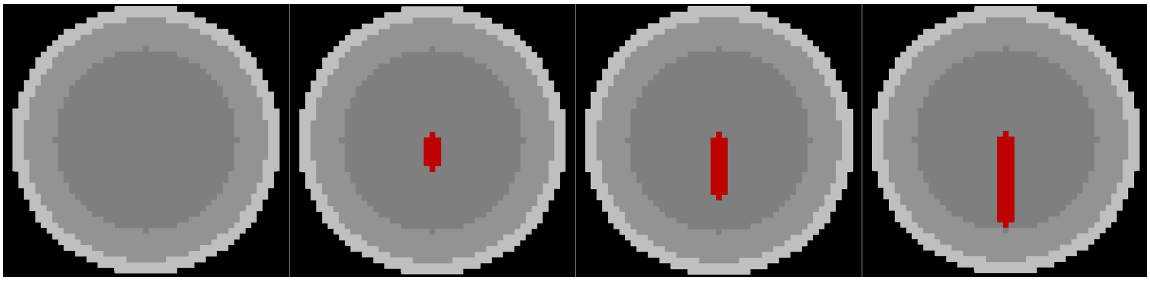

Figure 11: Shows the gradual growth of a region when the S3DRG algorithm uses a LIFO queue.

If on the other hand the LIFO queue is replaced with a FIFO queue, where the first element inserted is the first element to be removed, the growth pattern is as shown in figure 12. Comparing the growth pattern of the LIFO queue to the growth pattern of the FIFO queue, it becomes apparent that the LIFO queues growth stretches out in one direction, while the FIFO queue has a more "compact growth" and grows equally in all directions. Using the similarity predicate as described above where each new candidate voxels intensity is compared with the intensity of the seed voxel, these differing growth patterns have no impact on the final segmentation of the volume. However, if the similarity predicate is replaced with a predicate that

uses the mean intensity of all the voxels in the current region to make a decision, the growth patterns will to a certain extent influence the predicates similarity measure. The reason is simply that the currently used LIFO queue results in growth that stretches out in one direction, which tends to have the effect of coming into contact with less similar voxels that may belong to other regions more quickly than the FIFO queue. Comparing figures 11 and 12 it is clear that when growing a region with a LIFO queue, there are relatively few voxels that have been added to the region by the time the algorithm comes into contact with voxels that are less similar and may belong to another region. However, when using a FIFO queue it is clear that by the time the predicate needs to calculate the similarity of a voxel that may belong to another region, it will have a relatively large number of voxel intensities to base its decision on. The examples shown in figures 11 and 12 are ideal cases and the results will of course depend on the shape of the region, but in general it would be true that when starting in the center of a region, an algorithm that grows primarily in one direction will tend to encounter other regions earlier than an algorithm that distributes its growth equally in all directions.

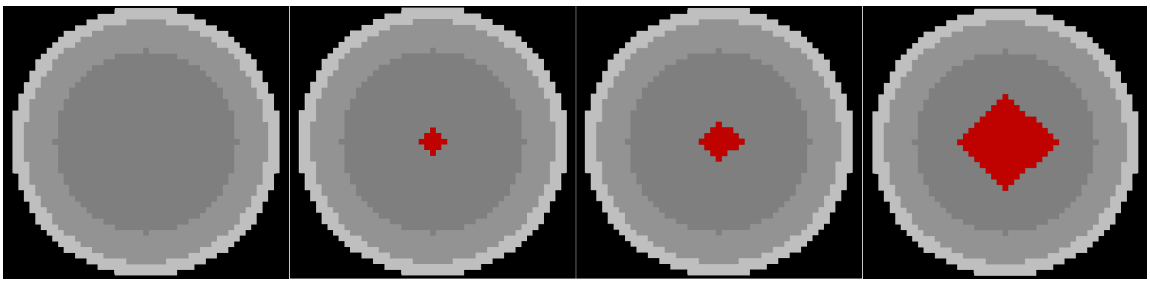

Figure 12: Shows the gradual growth of a region when the S3DRG algorithm uses a FIFO queue.

Finally it should be noted, that even though the S3DRG algorithm scales well with regards to processing time, it still uses a considerable amount of time when segmenting high resolution volumes. As shown in table 1, the S3DRG algorithm needs almost four minutes to completely segment a volume of resolution 512^3 when using a 26-connected neighborhood if no preprocessing was done on the volume to reduce the amount of data. This clearly rules out any form of interactive segmentation, relegating the algorithm to batch oriented segmentation tasks for high resolutions.

10 SRG3D: Seeded Region Growing 3D

The previous chapter examined a simple and fully automatic 3D region grower called S3DRG. Although this algorithm presumably performed well on most simple low resolution volumes, it had two very undesirable properties. Firstly, since growing each region was done sequentially and independently of all other regions, the segmentation results were clearly biased in favor of the region(s) grown first. Secondly, the algorithms memory requirements when using its noise filter were shown to grow quadratically with the growth of data, making it practically unusable for volume resolutions above 256^3. This chapter will describe, test and discuss an implementation of the Seeded Region Growing (SRG) algorithm[5] for 3D grids in an attempt to address the problems revealed by the S3DRG algorithm. The following section describes a specific implementation of the SRG algorithm called SRG3D, which has been adapted to run on a 3D grid. For a general description of the SRG algorithm, se chapter 6.

10.1 Description

When calling the SRG3D algorithm on a volume, a list of seed voxels need to be provided as input. Each seed voxel is labeled with a unique label based on its position in the input list, and their intensity values are added to a list that collects the sums of all the intensities for each region. There is also a list that stores how many voxels belong to each region. The intensity sums and numbers of voxels for each region are collected in order to do fast calculations of each region mean when checking the similarity between regions and new candidate voxels. Once the seeds have been labeled and their intensities added to their respective regions intensity sum, an algorithm that adds each seeds neighbors to a sequentially sorted list (SSL) is called. The SSL stores all the border voxels of all regions sorted by delta value (low to high). The delta value is a measure of how different a candidate voxel is from a prospective region, measured by the absolute value of the difference between the voxels intensity and the regions mean intensity.

Once the seeds neighbor voxels have been added to the SSL, the algorithm enters a loop which continues as long as there are still more voxels in the SSL to process. This loop starts by removing the first voxel in the SSL, since this is the voxel that has the lowest delta value and therefore is the least different from its neighbor region(s). Since this is the voxel among all the current border voxels that is most similar to its connected region(s), this voxel should be added to its connected region first. If all the labeled neighbors of this voxel have the same label, this voxel is labeled with this

Algorithm 5 [*SRG3D(seeds)*]

```
intensitySums
numIntensities
ssl
for i = 0 to seeds.size() − 1 do
  currentSeedVoxel ← voxels[seeds[i].x][seeds[i].y][seeds[i].z]
  currentSeedVoxel.setLabel(i + 1)
  intensitySums.add(currentSeedVoxel.getIntensity())
  numIntensities.add(1)
  add6NeighborsToSSL(seeds[i], ssl, currentSeedVoxel.getIntensity())
end for
while not ssl.empty() do
  sslElement ← ssl.front()
  ssl.popFront()
  neighborLabels ← get6NeighborLabels(sslElement)
  if neighborLabels.size() = 1 then
    label ← neighborLabels.begin()
    voxels[elem.x()][elem.y()][elem.z()].setLabel(label)
    intensity ← voxels[elem.x()][elem.y()][elem.z()].getIntensity()
    intensitySums[label − 1] ← intensitySums[label − 1] + intensity
    numIntensities[label − 1] ← numIntensities[label − 1] + 1
    voxelIndex ← createIndex(elem.x(), elem.y(), elem.z())
    mean ← intensitySums[label − 1]/numIntensities[label − 1]
    add6NeighborsToSSL(voxelIndex, ssl, mean)
  else
    intensity ← voxels[elem.x()][elem.y()][elem.z()].getIntensity()
    label ← neighborLabels.begin()
    mean ← intensitySums[label − 1]/numIntensities[label − 1]
    delta ← getDelta(intensity, mean)
    for iterator ← neighborLabels.begin() to iterator = neighborLabels.end()
    do
      itLabel ← iterator.getValue()
      mean ← intensitySums[itLabel − 1]/numIntensities[itLabel − 1]
      itDelta ← getDelta(intensity, mean)
      if itDelta <= delta then
        label ← itLabel
        delta ← itDelta
      end if
    end for
    voxels[elem.x()][elem.y()][elem.z()].setLabel(label)
    intensitySums[label − 1] ← intensitySums[label − 1] + intensity
    numIntensities[label − 1] ← numIntensities[label − 1] + 1
  end if
end while
```

label. If there is more than one label among its labeled neighbors, the voxel is labeled with the label of the most similar region (the region with which it has the lowest delta value). Each time a voxel is removed from the SSL and labeled, the region border changes and the change in the set of border voxels is updated by adding all of the neighbors of the newly labeled voxel to the SSL. Voxels that have already been labeled or that are already in the SSL are not added. To avoid duplicates in the SSL, each SSL element is labeled as SSL-ELEM and keeps this label until it is relabeled as belonging to a region. Since only unlabeled voxels are added to the SSL, the same voxel will not be added twice. Once all the voxels have been removed from the SSL and processed, all the voxels in the volume have been labeled and the algorithm is complete.

Algorithm 5 shows the details of the SRG3D algorithm. The functions that add neighbors to the SSL (add6NeighborsToSSL), finds the labels of a voxels neighbors (get6neighborLabels) and the similarity predicate that calculates the delta value (getDelta) have been abstracted away. This listing shows a version of the algorithm that uses a 6-connected neighborhoood.

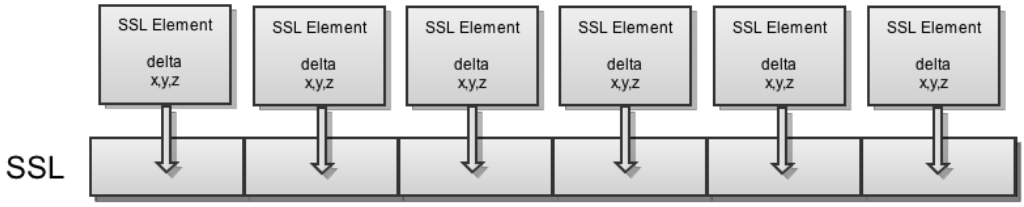

Figure 13: Illustration of the SSL data structure and its elements.

The data structure used for the SSL is a doubly-linked list. The reason is that such a list allows fast (constant time) insertion of data at any point in the list, which is crucial since the SRG3D algorithm needs to insert voxels in various places in the SSL, depending on their delta value. Removal of the first element in the list for each iteration is also fast and does not require any reallocation of memory. However, finding the right position to insert each voxel is very time consuming since the SSL will grow large at high resolutions and each insertion requires iteration on the SSL. If the range of delta values are spread out equally in the range 0 to 1, then the average insertion would require iterating through half of the elements in the SSL. Insertion by using a binary search would be much faster and should be considered to improve performance. To improve the speed of the algorithm slightly, the average delta value

of the SSL is calculated before each iteration. If the voxel to be inserted has a delta value less than or equal to this average, the list is iterated from beginning to end and if the delta value is above this average, the list is iterated in reverse. This decreases the processing time somewhat since elements with small delta values are closer to the beginning of the list and elements with large delta values are closer to the end of the list.

10.2 Segmentation Bias

As discussed in section 9.3 and illustrated in figure 10, the S3DRG algorithm has a clear segmentation bias leading it to favor regions that are grown first compared to regions grown at a later stage. Figure 10 showed that even if region B is more similar to region A than it is to region C, it may still be absorbed by region C if region C is grown first. This order dependence is solved by the Seeded Region Growing (SRG) algorithm[5]. Instead of growing one region at a time, it grows all regions simultaneously, but at different speeds. Regions that have very similar neighbors grow faster than regions with less similar neighbors, leading to a segmentation that is independent of seed ordering and where each region is more homogenous than when segmented with the S3DRG algorithm.

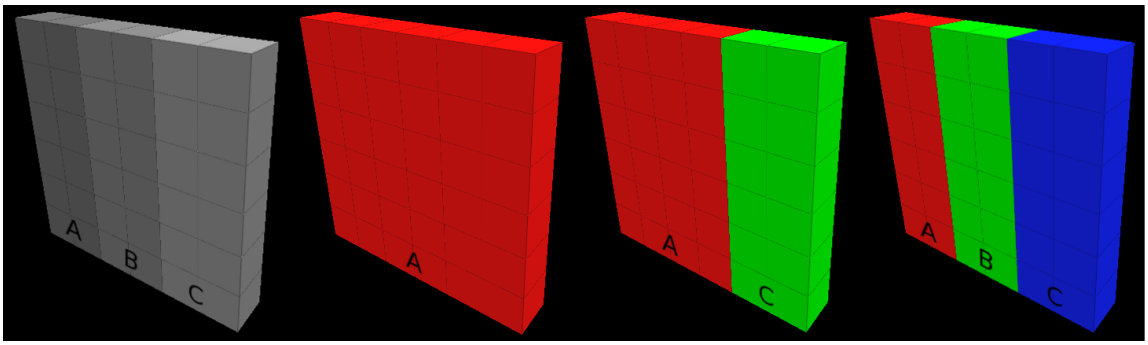

Figure 14: Result of segmenting a simple volume with the SRG3D algorithm. Illustrations two, three and four show the results of using one, two and three seeds respectively.

Figure 14 shows the result of segmenting a simple volume with the SRG3D algorithm. The second illustration in figure 14 shows that if only one seed is placed, the region represented by this seed will absorb the whole volume since there are no other regions to compete with. The third illustration shows what happens when one

seed is placed in area A and one seed is placed in area C. Since the intensity of area B is closer to the intensity of area A in this case than to the intensity of area C, area B is absorbed by area A. It is important to note that area B is absorbed by area A regardless of the order in which the SRG3D algorithm receives the seeds. This is probably the single most important aspect when comparing the SRG3D algorithm to the more primitive S3DRG algorithm. Finally the fourth illustration shows what happens when one seed is placed in each of the three areas A, B and C. In this case we end up with three separate regions, which we could only achieve with the S3DRG algorithm by tweaking the intensity threshold until we arrived at the desired result.

It should be noted that even though the SRG algorithm is not biased towards earlier segmented regions as is the case with the S3DRG algorithm, it still has some order dependencies as described by Mehnert and Jackway[7]. Mehnert and Jackway discovered that if a region B is equally similar to a bordering region A and bordering region C, which region it will be absorbed by is arbitrary and depends on the order in which the pixels (or voxels) happen to be processed by the SRG algorithm. For more details, se chapter 6 and [7].

10.3 Noise

As described and discussed in chapter 9, the S3DRG algorithm is very sensitive to noise. If the algorithms noise filter is not used, the segmentation of the volume will tend to have many miniscule segments that are the results of noise or artifacts being interpreted as distinctive regions. If the algorithms noise filter is used, this will prevent noise from being labeled as separate regions and instead label all such miniscule regions as noise. However, while this makes it possible to identify noise, it results in an incomplete segmentation of the volume. Figure 15 demonstrates what happens when the S3DRG algorithm segments a noisy volume. Illustration two shows that when the noise filter is turned on, each noise element is not regarded as a separate segment, but is instead labeled as noise. Illustration three shows that when the noise filter is turned off, each noise element is regarded as a separate segment, leading to many miniscule segments.

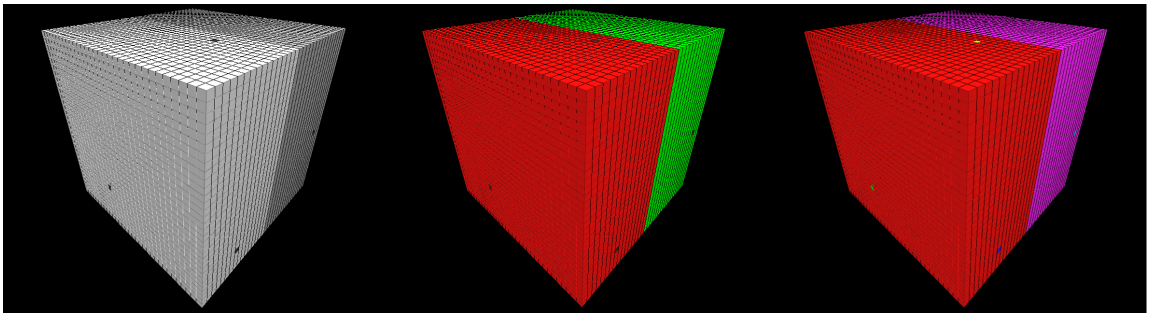

Figure 15: Result of segmenting a simple noisy volume with the S3DRG algorithm. Illustration one shows the original volume while illustration two and three demonstrate the results of having the noise filter switched on and switched off, respectively.

A better solution would be to let the region that is most similar to the noise (and connected to the noise) absorb the noise. This is exactly what the SRG3D algorithm does. Illustration two in figure 16 shows what happens when the SRG3D algorithm segments a noisy volume. As the illustration demonstrates, the noise is absorbed completely, resulting in a volume that is noiseless and completely segmented. However, even though the SRG3D algorithm handles noise much more gracefully than the S3DRG algorithm, it is important that the selected seed voxels are representative of the regions of interest. If a seed is placed in a noisy area, the resulting segmentation is likely to be very far from ideal as shown in illustration three in figure 16. To avoid this situation, Adams and Bischof[5] recommended using small clusters of pixels (or voxels in this case) as seeds instead of individual pixels when segmenting images with high levels of noise. Since the region mean is used to calculate the delta value when considering candidates for inclusion in the region, a cluster of seed pixels will reduce the effects such noisy outliers have on the final segmentation.

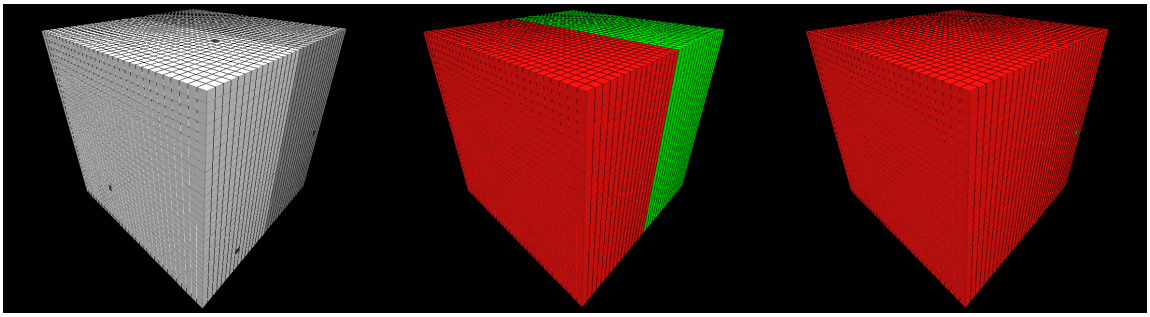

Figure 16: Result of segmenting a simple noisy volume with the SRG3D algorithm. Illustration one shows the original volume while illustration two and three demonstrate the effect of correct and incorrect seed placement, respectively.

10.4 Memory Usage

In chapter 9 it was revealed that the S3DRG algorithm uses very much memory. When the S3DRG algorithm has its noise filter turned on, its memory consumption grows quadratically compared to the the growth in number of voxels as the resolution increases. When segmenting the volume in figure 4, the SRG3D algorithm has a much lower memory use than the S3DRG algorithm as shown in table 3.

Algorithm and neighborhood	Resolution				
	32^3	64^3	128^3	256^3	512^3([1])
SRG3D (6-connected)	$<$ 1 MB	$<$ 1 MB	5 MB	41 MB	328 MB
SRG3D (26-connected)	$<$ 1 MB	2 MB	5 MB	60 MB	720 MB

Table 3: Memory used by the SRG3D algorithm with two different neighborhoods and at five different resolutions when segmenting the volume shown in figure 4.

The reason for this significant difference in memory usage between the two algorithms is that the S3DRG algorithm stores the index of each voxel in the region it is currently processing and later uses these indices to relabel small regions as noise. For large regions, this list of indices takes up very much space. The SRG3D algorithm on the other hand has no need for such a data structure and the only significant

[1]The memory use for this resolution is an estimate extrapolated from the measurements of the memory use at lower resolutions. Linear growth (compared to the growth in number of voxels) was assumed to be the best model.

amount of memory it uses is allocated by the SSL. The size of the SSL will grow with increased resolutions, but since this structure only stores the border voxels (the surface) of each region and not each regions interior, its size stays relatively low for the duration of the algorithms execution.

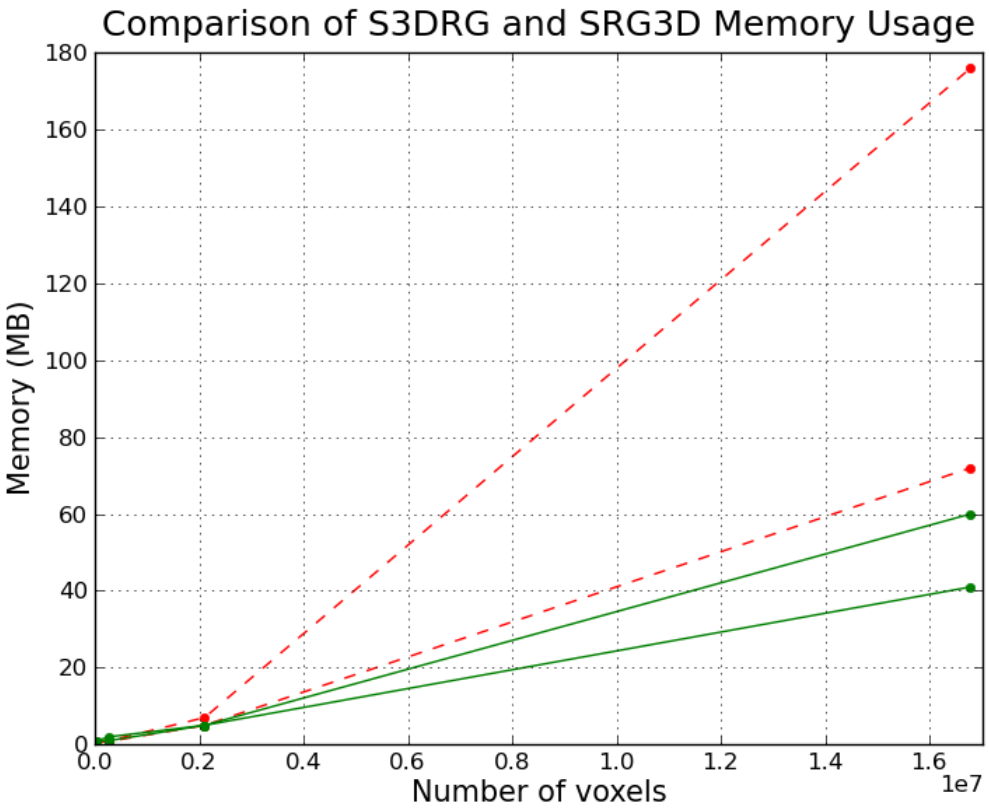

Figure 17: S3DRG and SRG3D memory use. The red dotted lines show the memory used by the S3DRG algorithm using 6-connected (bottom line) and 26-connected (top line) neighborhoods, while the green lines show the memory used by the SRG3D algorithm using 6-connected (bottom line) and 26-connected (top line) neighborhoods.

10.5 Processing time

In section 9.3 the conclusion was drawn that the processing times of the S3DRG algorithm were acceptable. This is not the case however with the SRG3D algorithm,

which quickly becomes unusable at high resolutions since it requires very much processing time. This becomes clear by examining the measurements displayed in table 4.

Algorithm and neighborhood	Resolution				
	32^3	64^3	128^3	256^3(2)	512^3(2)
SRG3D (6-connected)	89 ms	4644 ms	226128 ms	11406400 ms	570320000 ms
SRG3D (26-connected)	152 ms	7416 ms	286913 ms	14345650 ms	717282500 ms

Table 4: Processing time used by the SRG3D algorithm with two different neighborhoods and at five different resolutions when segmenting the volume shown in figure 4.

As is shown in the table, the SRG3D algorithm needs several minutes to segment a volume of only 128^3 voxels, a task that is completed in less than 3 seconds using the 6-connected version of the S3DRG algorithm. The reason for this considerable difference in processing times is simply that each iteration of the SRG3D algorithm requires searching in a large data structure, while each iteration of the S3DRG algorithm only requires the comparison of two values. Each time a new candidate voxel is added to the SRG3D algorithms SSL, this voxel needs to be inserted in the correct position which depends on its delta value. As the SSL grows large, this search will take considerable time. The S3DRG algorithm on the other hand does not need to perform this search, it only needs to do one comparison before its insertion. Figure 18 compares the processing times of these two algorithms as the number of voxels increases.

[2]The processing times for these resolutions are estimates extrapolated from the measurements of the processing times at lower resolutions. Linear growth (compared to the growth in number of voxels) was assumed to be the best model.

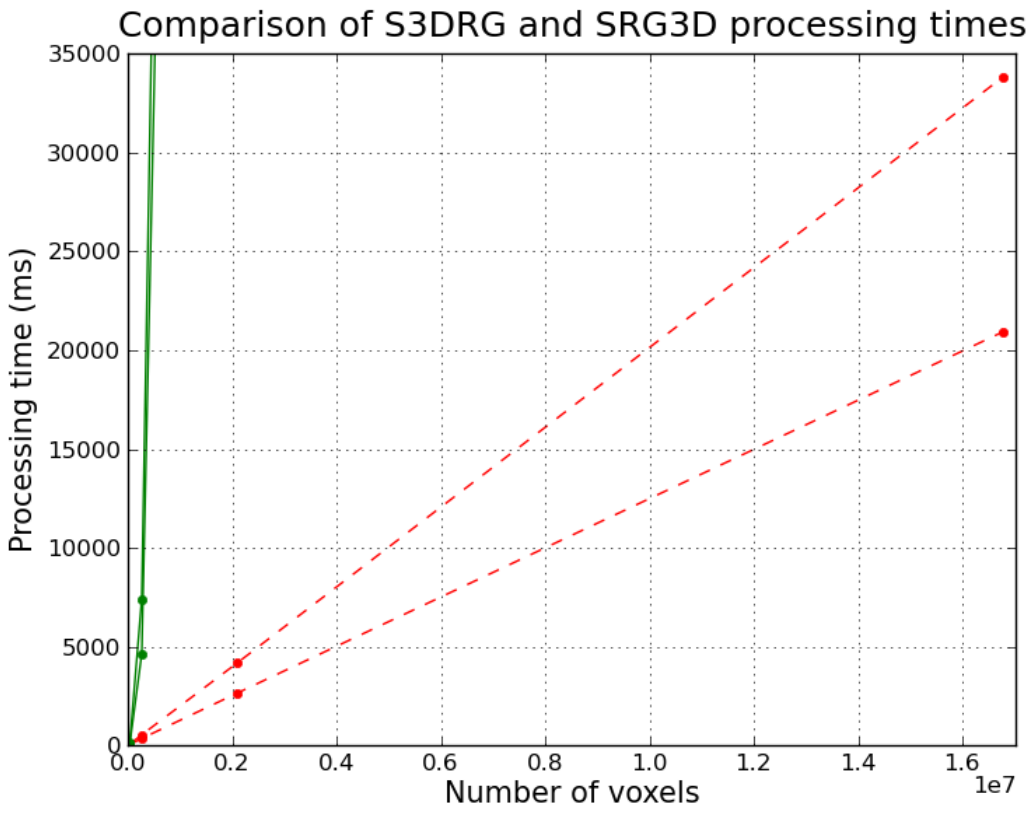

Figure 18: S3DRG and SRG3D processing times. The red dotted lines show the processing times used by the S3DRG algorithm using 6-connected (bottom line) and 26-connected (top line) neighborhoods, while the green lines show the processing times used by the SRG3D algorithm using 6-connected (bottom line) and 26-connected (top line) neighborhoods.

10.6 Conclusions

In section 9.3 it was mentioned that the S3DRG algorithm satisfies the conditions required to call the result of running the algorithm a segmentation. However, it only satisfies these conditions completely if the noise filter is turned off. The SRG3D algorithm on the other hand always satisfies all these conditions.

Section 9.3 also discussed the fact that in its current implementation, the S3DRG algorithms growth is dependent on the data structure used to store candidate voxels

for inclusion in the current region when certain similarity predicates are used. The SRG3D algorithm is less strongly coupled to its data structure. As long as each iteration removes the element with the smallest delta value from the list of candidate voxels, the underlying representation will not influence the final segmentation result, only the time required to reach the final result.

Perhaps the single most important advantage the SRG3D algorithm has over the S3DRG algorithm is that it looks at all regions at the same time and grows in the direction of smallest delta value. The SRG3D algorithm has a more global view than the S3DRG algorithm when growing each region since it has a common data structure for storing all region boundries. It can make decisions that are best for the segmentation of the volume as a whole, not just what is best for a single region at any point in time. This prevents it from biasing its segmentation in favor of the regions that are segmented first.

Another important advantage of he SRG3D algorithm is that it handles noisy volumes very well. Whereas the S3DRG algorithm was able to detect and label noise, it required very much memory and it did not really solve the noise issue, it just postponed it and made it easier to handle noise in a potential post-processing step. The SRG3D algorithm absorbs all noise as long as the seed voxels are not placed in a noisy area, which will give an unpredictable (and most likely an undesirable) result.

The SRG3D algorithm also has the advantage that it uses much less memory than the S3DRG algorithm when the S3DRG algorithms noise filter is turned on. While the S3DRG algorithm runs out of memory on most consumer hardware without preprocessing the volume when using high resolutions, the SRG3D algorithm has relatively modest memory needs, compared to the amount of data being processed, at these same resolutions.

Although the quality of the final segmentation provided by the SRG3D algorithm is far superior to that of the S3DRG algorithm, some problems still remain. As noted by Mehnert and Jackway[7], the SRG algorithm does not update the delta values of of older SSL elements as each region increases in size. Also, the algorithm handles border elements that have the same delta values rather arbitrarily (depending on the order in which elements are processed). Both of these were found to have an impact on the final segmentation result and showed that there were still some order dependencies in the SRG algorithm.

Another disadvantage with the SRG3D algorithm is that it relies on manual seed placement. There are many ways to automate the process of finding good starting seeds, with varying strengths and weaknesses. What they all have in common is that they remove the need for manual intervention by increasing the complexity and

required processing time of the algorithm. The success of such methods will depend on the method used and the properties of the data they are used on.

Although the issues with the SRG3D algorithm that were just mentioned are noticeable problems, there is one problem that overshadows all the other problems in the current implementation. Looking at the test results in this section it becomes very clear that the biggest problem with the current implementation of the SRG3D algorithm is its very long processing times even on volumes with relatively low resolutions. The next chapter will examine ways to improve the performance of the SRG3D algorithm.

11 Increasing the Speed of the SRG3D Algorithm

As demonstrated in the previous chapter, in the currently used implementation, the SRG3D algorithms biggest problem is the long time it needs to process a volume at high resolutions. This chapter will examine ways to improve the processing speed of the SRG3D algorithm. Each section will demonstrate and discuss a specific improvement and compare the original algorithm with this new improved version.

11.1 SRG3D With a Red-Black Tree

When processing volume data, a simple doubling of resolution increases the amount of data to process by a factor of eight. This is particularly troubling since each iteration of the SRG3D algorithm requires searching in a data structure whose size increases in step with the amount of data to process. The original SRG algorithm developed by Adams and Bischof in 1994[5] was originally used on 2D grids of pixels. The processing times reported for running the original SRG algorithm on a 2D image of 256^2 pixels was 4 seconds. If this same algorithm was run on the same hardware with a 3D grid of 256^3 voxels, the increase in data alone would be 256 times the original amount, which would give a processing time of about 17 minutes (assuming linear growth). Comparing this to the processing times used by the SRG3D algorithm (about 190 minutes at the same resolution), one can only conclude that the authors of the original SRG algorithm used some way to improve the SSL insertion speed of their algorithm. Their data structure is described as "just a linked list of objects", but it is very likely that the linked list they used was a low level representation of some other data structure. For instance, using a binary search tree would change the time complexity of the insertion from linear to logarithmic in the number of elements. Using a binary search tree as a data structure for the SRG algorithm is not a new idea. Mehnert and Jackway[7] used an unspecified type of binary search tree in their

ISRG algorithm, Grinias and Tziritas[8] used an AVL binary search tree in their version of the SRG algorithm and Lin, Jin and Talbot[11] used a Splay tree in their unseeded region growing algorithm. An AVL binary search has the advantage that it is self balancing. This becomes a very important quality for the SSL, since the SSL is a very dynamic structure that keeps changing (because of insertions and removals) throughout the lifetime of the SRG algorithm. If no form of balancing is performed on the binary search tree, the insertions will get slower as the tree grows deeper. The Splay tree is another form of self-balancing tree that is also self optimizing since it moves more frequently accessed elements nearer the root of the tree and in doing so provides faster access to these elements.

To improve the SRG3D algorithms speed, a red-black tree was chosen. The red-black tree is a type of self balancing binary search tree. Since it is self balancing, it is similar to an AVL tree. An AVL tree is more rigidly balanced than a red-black tree which allows for slightly faster retrievals. However, because of the more rigid balancing, insertion and removal are slightly slower when using an AVL tree[4]. The red-black tree on the other hand is faster at inserting and removing elements, which makes it very well suited to represent the SSL data structure because this structure is constantly changing by performing insertions and removals. A Splay tree should also be suited to the task. However, while a Splay tree is self optimizing, it still has a worst case time complexity that is worse (linear). Its self optimizing quality is likely to compensate for this drawback in practice, but this depends on how homogenous the volume data is. If more often than not, the next candidate for insertion is similar to previous candidates, the Splay tree is likely to be a good solution as well.

Table 5 and figure 19 show how much processing time is needed by the SRG3D algorithm when a red-black binary search tree with a worst case time complexity of O(log n) is used instead of the original linked list with a worst case time complexity of O(n).

	Resolution				
Algorithm and neighborhood	32^3	64^3	128^3	256^3	512^3
SRG3D (6-connected)	36 ms	307 ms	2646 ms	23264 ms	202015 ms
SRG3D (26-connected)	75 ms	635 ms	5318 ms	44759 ms	367133 ms

Table 5: Processing time used by the SRG3D algorithm with two different neighborhoods and at five different resolutions when using a red-black binary search tree while segmenting the volume shown in figure 4.

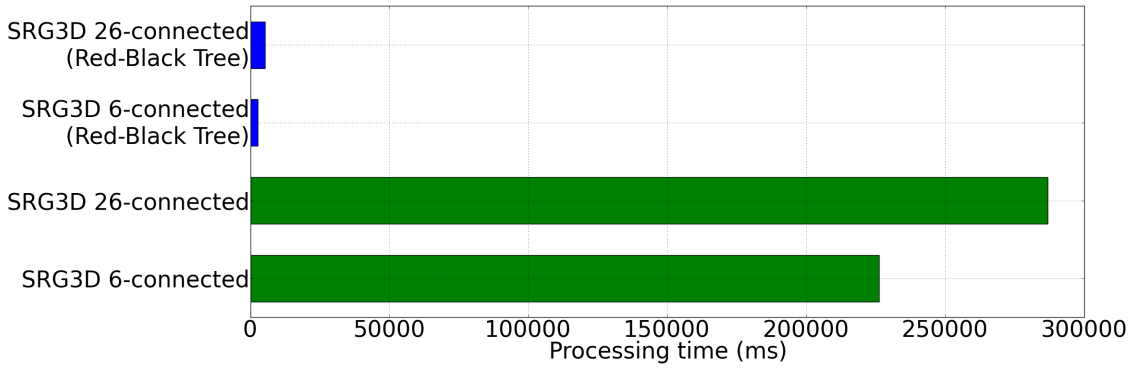

Figure 19: Comparison of SRG3D with and without an SSL based on a binary search tree at a resolution of 128^3. The bottom two bars show the original SRG3D algorithms processing times and the top two bars show the new version of the SRG3D algorithm that uses a red-black tree to represent its SSL.

Looking at figure 19, it is clear that using a red-black tree to represent the algorithms SSL data structure has a very noticeable impact on performance. If the old version of the algorithm had an SSL of size n = 100000, it would in the worst case need to do 100000 (n) comparisons to find the correct SSL position. The average case would be much better, but still very large at 50000 (n/2). When using a red-black tree however, in the worst case (and average case) the number of comparisons would be 5 (log n). This represents a qualitative improvement in the algorithm in the sense that it can now be used interactively for some resolutions that could previously only be processed as batch jobs.

11.2 Heuristic for Improving Search in Red-Black Tree

If a volume is relatively homogenous, it will often be the case that the next voxel to be inserted in the SSL has the same (or similar) delta value as the previously inserted voxel. This is something that can be taken advantage of in the SRG3D algorithm when using a red-black tree (or other binary search tree). If the red-black tree returns the index of the position of the previously inserted voxel and keeps this index cached until the next insertion, it can use this index as a heuristic for the insertion algorithm. The results of running the SRG3D algorithm when this heuristic is in use is showed in table 6.

Algorithm and neighborhood	Resolution				
	32^3	64^3	128^3	256^3	512^3
SRG3D (6-connected)	35 ms	294 ms	2476 ms	21520 ms	183602 ms
SRG3D (26-connected)	73 ms	615 ms	5144 ms	42582 ms	351773 ms

Table 6: Processing time used by the SRG3D algorithm with two different neighborhoods and at five different resolutions when using a red-black binary search tree with a search heuristic while segmenting the volume shown in figure 4.

When using the search heuristic, the only difference is that an attempt is made to insert the voxel in the position right behind the previous voxel. If this position is correct, the insertion is completed with only 1 comparison operation. If this position turns out to be wrong, a regular search is performed to find the correct position. Comparing the results from table 6 with the results in table 5 where a search heuristic was not used, there is a small, but noticeable improvement.

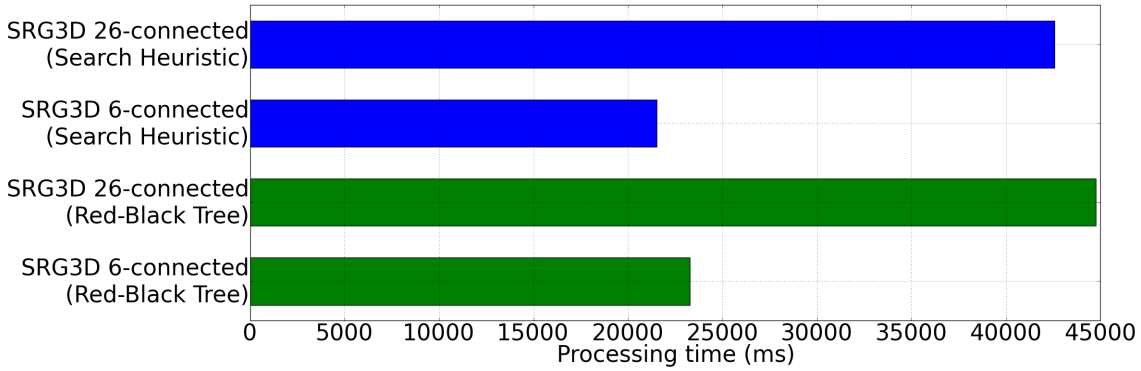

Figure 20: Comparison of the SRG3D algorithm when using a binary search tree and when using a binary search tree with a search heuristic.

11.3 SRG3D With a Heap-Based SSL

According to Breen and Monro[6], self-balancing binary search trees like splay trees, AVL trees and red-black trees provide better performance than the heap as a priority queue representation for image analysis applications. This view was later contested by Hendriks[21] who concluded that heaps are more efficient representations of priority queues on newer computing hardware. This section will examine the performance

of the SRG3D algorithm when using a heap data structure to represent the SSL and compare these results to the red-black tree representation from the two previous sections.

Algorithm and neighborhood	Resolution				
	32^3	64^3	128^3	256^3	512^3
SRG3D (6-connected)	30 ms	257 ms	2308 ms	21524 ms	n/a([3])
SRG3D (26-connected)	68 ms	582 ms	4982 ms	42800 ms	n/a([3])

Table 7: Processing time used by the SRG3D algorithm with two different neighborhoods and at five different resolutions when using a heap-based SSL to segmenting the volume shown in figure 4.

Looking at the results in table 7 we see that with a heap-based priority queue, the trend of steady improvements across all resolutions has changed. Whereas the search heuristic for the red-black tree caused a reduction in processing times at all the tested resolutions, the heap-based version of the SRG3D algorithm performs noticeably better than the previous best result at low resolutions, but the improvement tapers off as the resolution (and amount of data) increases. At a resolution of 64^3, the heap-based version is clearly faster than the red-black tree with search heuristic, but at a resolution of 256^3 they are practically equal. The reason that the heap does better at low resolutions and about the same or slightly worse than the red-black tree with a search heuristic at higher resolutions is probably because self-balancing trees like red-black trees have some overhead attached to staying balanced. The red-black tree is a more complicated data structure than the heap and is only worth the extra overhead when the amount of data gets above a certain size. The graphs in figure 21 and 22 show how the red-black tree, red-black tree with search heuristic and heap compare at lower resolutions and higher resolutions respectively.

[3]This measurement is not available since the underlying data structure used to run this test needs to allocate all memory contiguously. At this resolution, there is too much data for all of it to fit in one contiguous area in the memory of the test system. Extrapolation from earlier data points would give an approximation, but was decided against because of the small gap in performance between the heap-based algorithm and the red-black tree algorithm with search heuristic. The error margin of the extrapolation would make it hard to arrive at any definitive conclusions at this resolution.

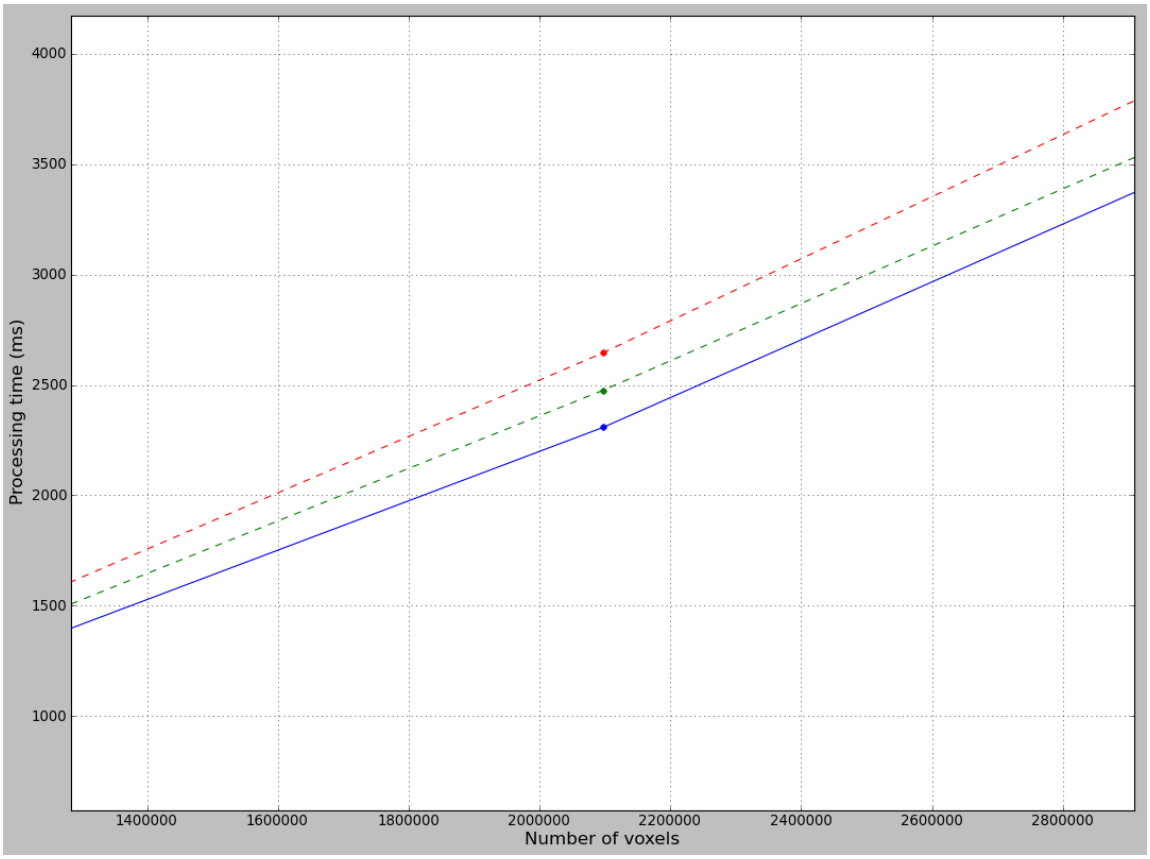

Figure 21: Change in processing time at low resolutions. The red dotted line shows an SSL based on a red-black tree, the green dotted line shows an SSL based on a red-black tree with a search heuristic while the blue line shows an SSL based on a heap.

As can be seen in the graphs, the red-black tree that is provided with a search heuristic and the heap data structure both consistently perform better than the red-black tree without a search heuristic. At low resolutions the heap is the best performer and at higher resolutions the heap and the red-black tree with a search heuristic are tied. Looking at very low resolutions (32^3 and lower) however, there is practically no difference between the data structures. The reason for this is probably that when the SSL gets below a certain size, any search algorithm will be fast enough since the total execution time of the SRG3D algorithm is no longer limited by the SSL insertion part of the algorithm. This is supported by the fact that the linear

complexity of the linked-list implementation is approaching the performance of the tree data structures at such low resolutions.

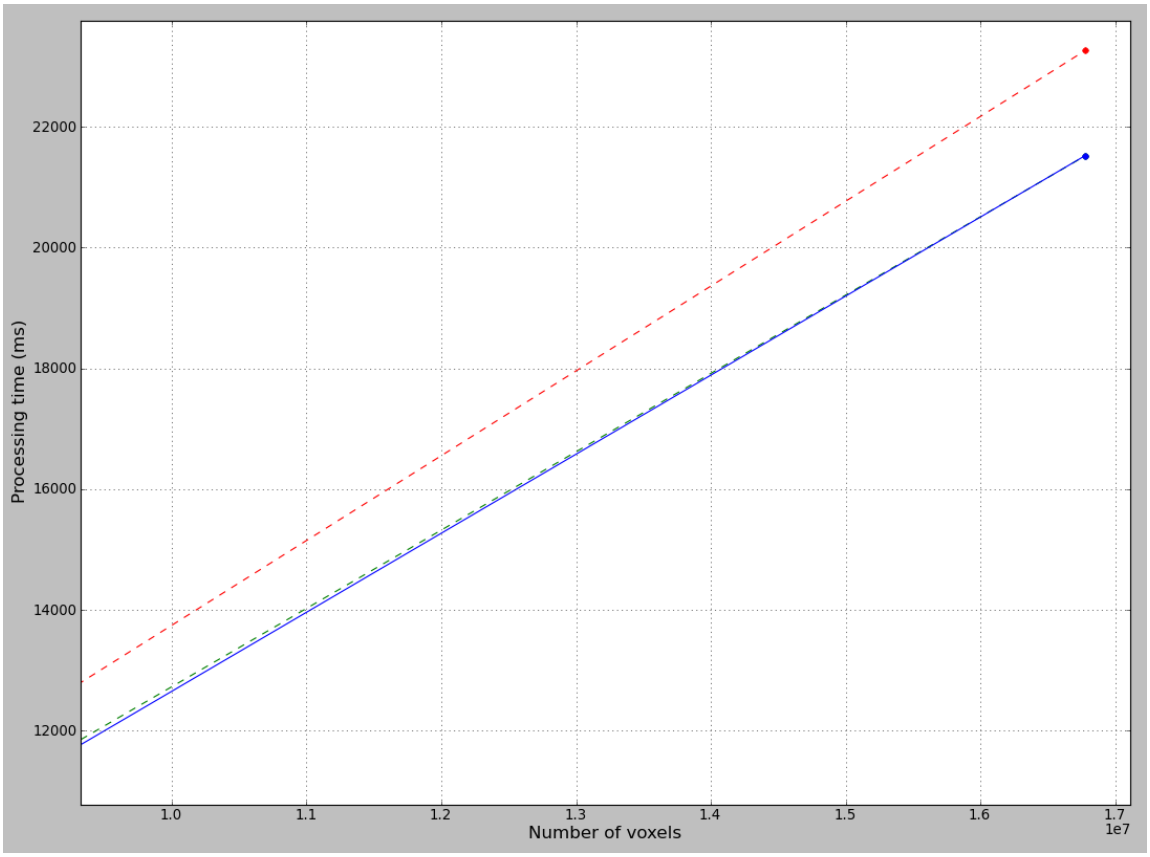

Figure 22: Change in processing time at high resolutions. The red dotted line shows an SSL based on a red-black tree, the green dotted line shows an SSL based on a red-black tree with a search heuristic while the blue line shows an SSL based on a heap.

11.4 Conclusions

The bar charts in figures 23, 24, 25 and 26 compare the Simple 3D Region Grower (S3DRG) from chapter 9 with the improved variants of the Seeded Region Growing 3D (SRG3D) algorithm that use a red-black tree, red-black tree with a search heuristic or a heap to improve the insertion speed of the SRG3D algorithms priority queue (SSL). Looking at the three lowest resolutions, we observe that the heap-based

version of the SRG3D algorithm performs best in terms of processing speed. It is even faster than the S3DRG algorithm at these resolutions when using a 6-connected neighborhood. The reason for this is probably that even though the S3DRG algorithm does not need to perform a search during each iteration, it visits a subset of the volumes voxels more than once. The heap-based SRG3D algorithm has very little overhead because of the simplicity of the heap data structure combined with the facts that it visits each voxel only once and the complexity of its search is close to O(1) when the SSL is very small as is the case at such low resolutions.

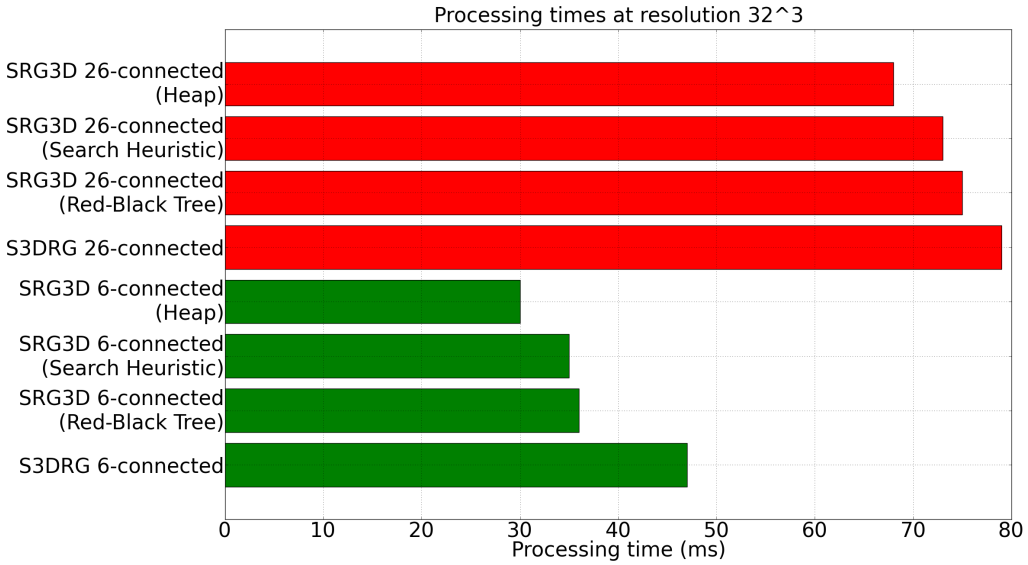

Figure 23: Comparison of processing times at resolution 32^3.

The performance of the SRG3D algorithm with a red-black tree is between the heap-based SRG3D algorithm and the S3DRG algorithms at the tree lowest resolutions when using a 6-connected neighborhood. This can be explained by the fact that the red-black tree version of the algorithm also visits each voxel in the volume only once and the complexity of its search should be close to O(1) at low resolutions, but the red-black tree also has slightly more overhead related to keeping itself balanced. This overhead constitutes a negligible part of the total processing time at higher resolutions, but at lower resolutions, the simplicity of the heap is more advantageous than the potential improvements provided by the extra complexity of the red-black tree.

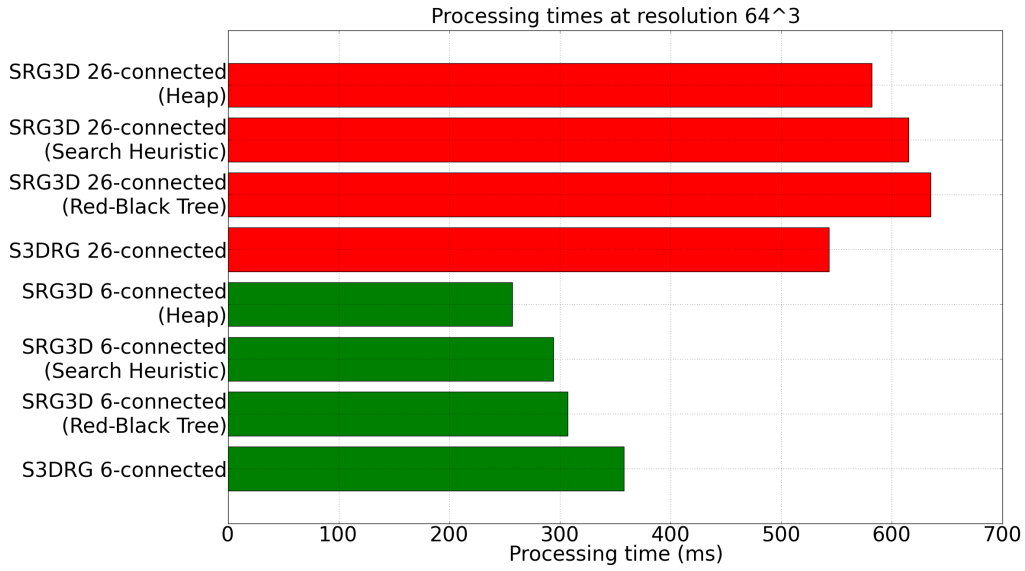

Figure 24: Comparison of processing times at resolution 64^3.

Looking at the highest resolution data for the 6-connected neighborhood in the bar chart in figure 26, the situation has changed. At this resolution, the S3DRG algorithms simplicity makes it the fastest algorithm. The amount of data gets very large because of the cubic growth in data in relation to the increase in resolution. With this drastic increase in data, the SSL grows to a considerable size which means that the SRG3D algorithms insertion speed becomes very important. At this resolution, the simplicity of the heap-based SSL still clearly outperforms the red-black tree, but the red-black tree with a search heuristic is now practically equal with the heap in terms of performance.

When examining the results of using a 26-connected neighborhood, the results mirror the findings of the 6-connected neighborhood to a large degree. As expected, examining more neighbors requires more time which can be seen across all resolutions. The only significant difference that stands out when using a 26-connected neighborhood is that the simplicity of the S3DRG algorithm starts outperforming the SRG3D algorithm at an earlier stage.

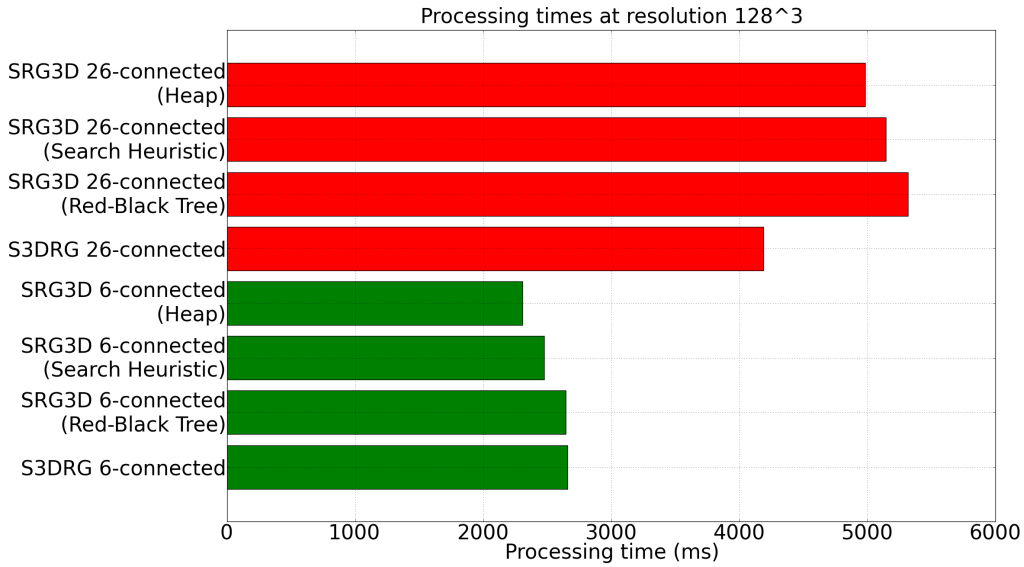

Figure 25: Comparison of processing times at resolution 128^3.

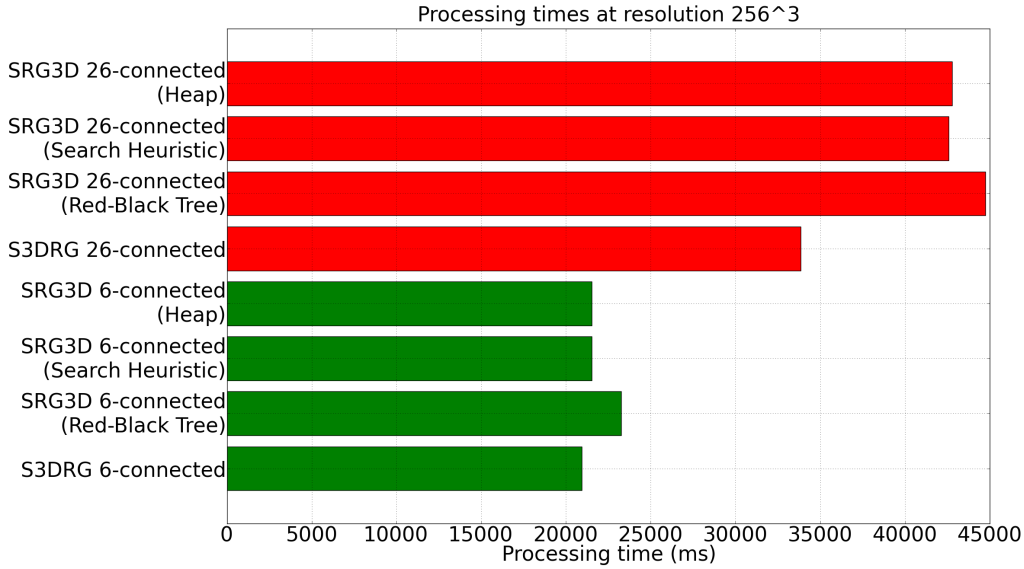

Figure 26: Comparison of processing times at resolution 256^3.

12 ASRG3D: Automatic Seeded Region Growing in a 3D Grid

When examining the S3DRG algorithm it was pointed out that it was biased, and even though it was able to distinguish noise from other data, it was not able to absorb this noise. The SRG3D algorithm was able to absorb noise and was not biased, but it was limited by the fact that it required manual input of seeds. This chapter will attempt to combine the S3DRG algorithm with the fastest version of the SRG3D algorithm to create an algorithm that retains the best features of both.

12.1 Description

The ASRG3D algorithm starts by running the S3DRG algorithm, which builds a vector that stores each segment that it has discovered. The first voxel in each of these segments is the S3DRG algorithms seed points and these are extracted and stored in a seed vector. The other results from running the S3DRG algorithm are discarded and the voxels in the volume are relabeled with the UNLABELED tag. The seed vector is then used as input when running the SRG3D algorithm, which in turn completes the segmentation of the volume. The pseudo code in algorithm 6 shows a high level view of the ASRG3D algorithm.

Algorithm 6 [*ASRG3D(seeds)*] Automatic Seeded Region Growing 3D

$segments \leftarrow S3DRG(neighborhood)$
$seeds \leftarrow findSeeds(segments)$
$SRG3D(seeds)$

12.2 Results

Figures 27 and 28 show the results of using the ASRG3D algorithm on synthetic and real-world data respectively. The first volume from the right in figure 27 shows how the ASRG3D algorithm segments the volume that was previously segmented by the S3DRG and SRG3D algorithms when testing the performance of these algorithms. The second and third volume shows segmentation of noisy synthetic volumes.

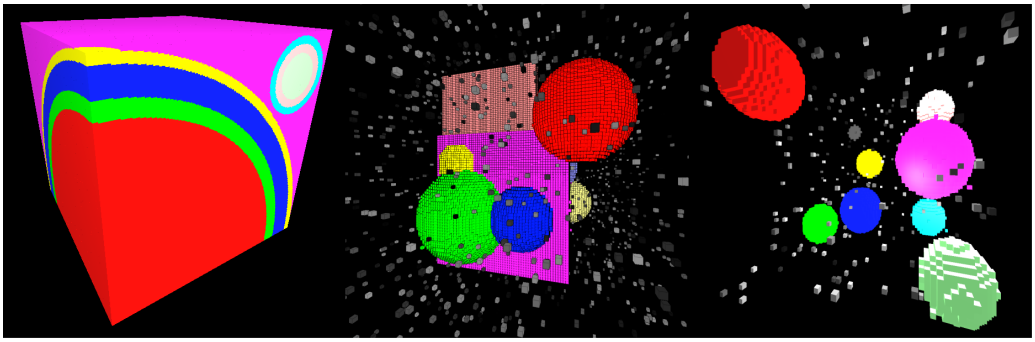

Figure 27: ASRG3D segmentation of synthetic volumes.

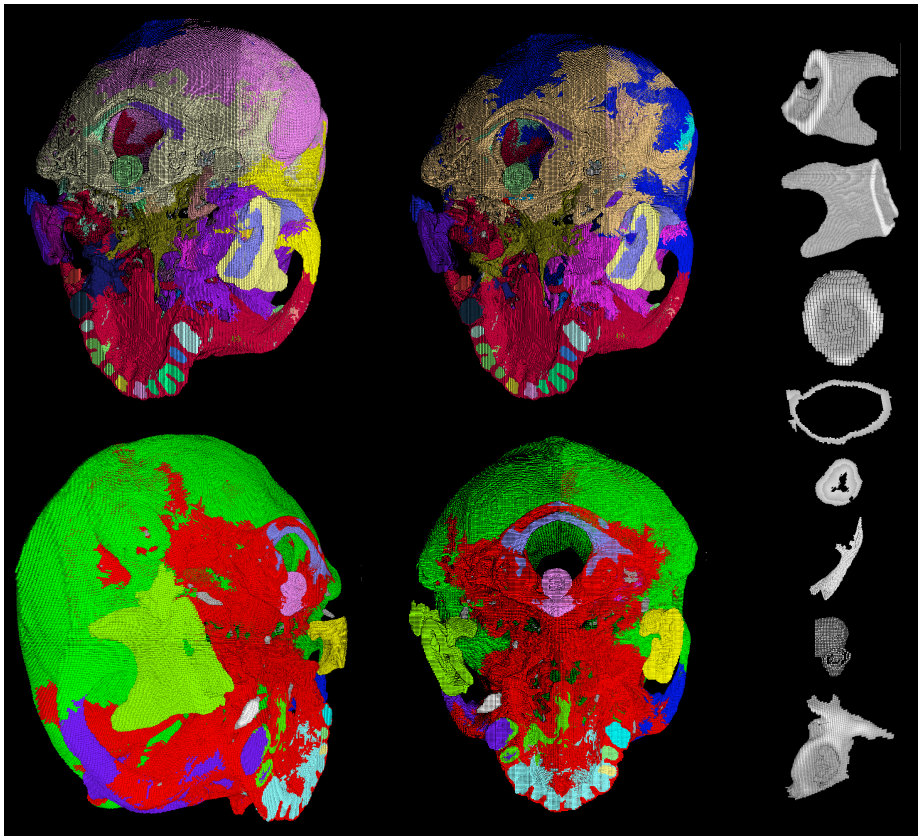

Figure 28: ASRG3D segmentation of medical data

12.3 Conclusions

The ASRG3D algorithm is very resistant to noise since it solves the noise related issues with both the S3DRG algorithm and the SRG3D algorithm. The S3DRG algorithm was able to detect noise and label it as such, but this would still result in an incomplete segmentation of the volume. The SRG3D algorithm was very noise resistant if the seed voxels were not placed on noisy voxels. If the seeds were placed in a noisy area, this would almost certainly result in an undesirable segmentation of the volume. The ASRG3D algorithm avoids these problems. Since the noise discovered by the S3DRG algorithm in the first stage of the ASRG3D algorithm, is excluded from the final segmentation result, the seeds that are provided to the SRG3D algorithm in the second stage of the ASRG3D algorithm will not be noisy. Instead, the seeds will be placed in a more representative area of the volume and noise will be absorbed when it is encountered.

The ASRG3D algorithm should also be able to avoid the clear segmentation bias that is present in the S3DRG algorithm. Manual seed placement would still be superior in many cases, but as long as each seed location discovered by the S3DRG algorithm is representative of its region, it should be able to segment the volume without introducing any form of segmentation bias.

The ASRG3D algorithm is slower than the previous algorithms, since it requires that two other algorithms are run sequentially in addition to the overhead resulting from the extraction of seeds and resetting of volume labels. However, when segmenting the medical volume in figure 28, the ASRG3D algorithm used less than 40 seconds which is reasonable when taking into account that it has the qualities of the SRG3D algorithm and in addition to this does not require manual seed placement. It should be noted however that the algorithm is still limited by the hard threshold used in the S3DRG algorithm, so some tweaking of this threshold will be necessary to get good results.

13 Conclusions and Further Research

This thesis has examined ways to improve the performance of volume segmentation algorithms. Specifically, the focus has been on using Seeded Region Growing (SRG) for segmentation of 3D grids of voxels and how to improve the speed of SRG on mid-range computing hardware. It was concluded that the insertion speed of the SRG algorithms priority queue (SSL) was the most significant bottleneck in terms of execution speed. Two important bodies of work that have looked at priority queues for image analysis were studied and these arrived at different conclusions. A paper by Breen and Monro[6] concluded that self-balancing binary search trees like the AVL tree and the splay tree perform better as the underlying data structure of a priority queue than the heap data structure. This conclusion was contested by Hendriks[21] who arrived at the opposite conclusion, namely that the heap was the best performing representation of a priority queue for image analysis tasks. Hendriks claimed that the reason for this discrepancy was that the cache prediction circuitry of modern CPUs was significantly improved in the years between the two studies and that this was the cause of the more simple heap data structures improved performance. To examine the claims of these studies, a version of the SRG algorithm that operates on volumes was developed called SRG3D. This algorithm was then tested with various data structures representing its priority queue. When comparing a version of the SRG3D algorithm that had a priority queue based on a red-black tree (self-balancing binary search tree), with a version of the algorithm that had a priority queue based on a heap, it was found that the heap consistently performed slightly better than the red-black tree implementation. This result confirms the conclusions of Hendriks, but it was also found that when adding a simple heuristic to guide the red-black trees search capability, the red-black tree performed equal to or slightly better than the heap at higher resolutions.

Using a self-balancing binary search tree or a heap drastically improves the processing speed of the SRG3D algorithm to the extent that there is probably not very much to gain from trying to improve the priority queue further and more attention should be paid to other parts of the algorithm. There is however one further modification of the priority queue that would be interesting to examine in the future, especially for high resolution data which results in a very large priority queue. As computing hardware keeps getting more and more parallel processing capabilities, both through multi-core CPUs and GPUs, finding ways to utilize these capabilities becomes important. One way to do this in the SRG3D algorithm would be to divide the priority queue into several parts, for example one for each thread of execution. Dividing the priority queue into parts has several advantages. If we assume that the

original priority queue is of size N and that we divide the queue into K parts, the insertion of each element will be faster, since the complexity of each insertion will now be O(log N/K) instead of O(log N). More importantly, when the priority queue is divided into K parts, we can perform K insertions in parallel by using K threads of execution. There is a slight overhead with this approach, as it would require that each time the algorithm needed to remove the smallest element from its priority queue, it would have to compare the first element from each part and remove the smallest one. As long as K is small, this overhead should however be negligible. If we for instance use two queues, we could perform two insertions simultaneously at cost O(log N/2) whereas we normally would need to perform two insertions sequentially at a cost of O(log N). The overhead would in this case simply be one comparison for each extraction of the smallest queue element (comparing the smallest element in queue 1 with the smallest element in queue 2 and removing the smallest of these). The reason this would work for a priority queue is that we are only interested in the smallest element at any point in time, we do not care about the order of the rest of the elements. Because of this, the queues do not need to be contiguous and can function completely independently. Achieving this type of parallelism by using a single priority queue would be much more challenging.

Another topic under examination in this thesis was how to automate the seed placement of the SRG3D algorithm. This was done by developing a two-pass algorithm called ASRG3D. The first pass of this algorithm performs a fast segmentation of the volume that also detects and labels potential noise. This first pass is biased and can not be defined as a complete segmentation, but it is in most cases adequate as a starting point for further processing. From each of the segments resulting from this first pass, one seed voxel is extracted that is believed to be a representative voxel for the segment. These seeds are then used as input to the second pass of the algorithm, which is the SRG3D algorithm. The result is an automatic version of the SRG algorithm that operates on volumes and that is very resistant to noise and that does not show any segmentation bias if the automatically selected seeds are representative of their respective regions.

There are two parts of the ASRG3D algorithm that would benefit considerably from further research. One of these is to find ways to automatically determine the threshold in the first pass of the algorithm. This threshold is now provided manually and is in its current state likely to need some tweaking when moving from one type of volume data to another. Another important factor to examine further is the seed extraction mechanism. At the moment, the seeds discovered by the first pass of the algorithm are used as input to the second pass. It is likely that it would be beneficial to examine the segments from the first part in more detail and extract a seed that

is closer to the center of each segment.

Finally, it is worth mentioning that when looking at the original papers examining the SRG algorithm and its performance on 2D images, we see that with the computing hardware and algorithms of today, it is now possible to segment 3D images in approximately the same time on a laptop computer as it would take to segment 2D images on a workstation computer in the early to mid 1990s. The ever increasing speed of computing hardware is opening up new areas of use for volume processing algorithms making further research in this area more important than ever before.

14 Bibliography

(In chronological order)

References

[1] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics, Volume 38, Number 8, 1965

[2] S. W. Zucker, *Region growing: Childhood and adolescence*, Comput. Graph. Image Process. vol. 5, pp. 382-399, 1976

[3] Fu and Mui, *A Survey on Image Segmentation*, Pattern Recognition Vol. 13, Pergamon Press Ltd., 1981

[4] D. D. Sleator and R. E. Tarjan, *Self-Adjusting Binary Search Trees*, Journal of the Association for Computing Machinery, Vol. 32, No. 3, pp. 652-686, 1985

[5] R. Adams and L Bischof, *Seeded region growing*, IEEE Transactions on pattern analysis and machine intelligence, Vol. 16, NO. 6, pp. 641-647, 1994

[6] E. Breen and D. Monro, *An evaluation of priority queues for mathematical morphology*, Mathematical Morphology and its Applications to Image Processing, Kluwer Academic Publishers, Dordrecht, pp. 249-256, 1994

[7] Mehnert, Jackway, *An improved seeded region growing algorithm*, Pattern Recognition Letters, Vol. 18, Issue 10, Oct. 1997

[8] I. Grinias and G. Tziritas, *Motion Segmentation and Tracking Using a Seeded Region Growing Method*, Proceedings of European Signal Processing Conference, 1998

[9] , R. Beare and H. Talbot, *Exact seeded region growing for image segmentation*, In Proceedings DICTA, 5th Biennial Conference Australian Pattern Recognition Society, pp. 132-137, 1999

[10] Lakare, *3D Segmentation Techniques for Medical Volumes*, State University of New York at Stony Brook, 2000

[11] Z. Lin, J. Jin, H. Talbot, *Unseeded region growing for 3D image segmentation* Proceeding, VIP '00 Selected papers from the Pan-Sydney workshop on Visualisation, Volume 2, 2000 (X)

[12] L. G. Shapiro, G. C. Stockman, *Computer Vision*, Prentice-Hall, 2001

[13] Zhen, Zhongmin and Cheng-Chang, *A Fast 3D Region Growing Approach for CT Angiography Applications*, Proc. of SPIE, Vol. 5370, 2004

[14] Fan, Zeng, Body and Hacid, *Seeded region growing: an extensive and comparative study*, Pattern Recognition Letters, Volume 26, Issue 8, June 2005

[15] Gonzalez and Woods, *Digital Image Processing*, Third Edition, Pearson / Prentice-Hall, 2008

[16] Lenkiewicz, Pereira, Freire and Fernandes, *Accelerating 3D Medical Image Segmentation with High Performance Computing*, Image Processing Theory, Tools & Applications, IEEE, 2008

[17] Qi, Xiong, Leow, Tian, Zhou, Liu, Han, Venkatesh, Wang, *Semi-automatic Segmentation of Liver Tumors from CT SCans Using Bayesian Rule-based 3D Region Growing*, The MIDAS Journal - Grand Challange Liver Tumor Segmentation, MICCAI Workshop, 2008

[18] Dougherty, *Digital Image Processing for Medical Applications*, Cambridge, 2009

[19] Lorentzen, *Extension of Standard Graphics Algorithms Into a 3D Grid*, TDT4500, NTNU IME, 2010

[20] Birkfellner, Figl and Hummel, *Applied Medical Image Processing*, CRC Press, 2010

[21] C. L. Luengo Hendriks, *Revisiting priority queues for image analysis*, Pattern Recognition, Volume 43, Issue 9, 2010