

A GENERIC FRAMEWORK FOR THE APPLICATION OF GRAPH THEORY TO  
IMAGE PROCESSING

by

Sonali Barua

A thesis submitted to the faculty of  
The University of North Carolina at Charlotte  
in partial fulfillment of the requirements  
for the degree of Master of Science in Computer Science

Charlotte

2007

Approved by:

---

Dr. Kalpathi R. Subramanian

---

Dr. William J. Tolone

---

Dr. Aidong Lu

©2007  
Sonali Barua  
ALL RIGHTS RESERVED

## ABSTRACT

SONALI BARUA. A Generic framework for application of Graph Theory to Image Processing.( Under the direction of DR. KALPATHI R. SUBRAMANIAN).

We report the development of a set of graph classes for the Insight Segmentation and Registration Toolkit. Graph theory has important applications in medical image processing such as centerline extraction; segmentation, skeletonization and distance transform calculation. Using generic programming and the philosophy of reusability and object-oriented design, a graph library has been designed and implemented. Classes have been designed for the Prim's minimum spanning tree, the depth first search algorithm, the Dijkstra's shortest path algorithm and the breadth first search algorithm. Supporting classes developed include the several classes that change an image to a graph and also classes that help determine whether a pixel is a node or not. The Prim's minimums spanning tree and the depth first search class have been applied for skeleton and centerline extraction within the Insight Registration and Segmentation Toolkit pipeline.

## TABLE OF CONTENTS

LIST OF FIGURES	vi
CHAPTER 1: INTRODUCTION	1
CHAPTER 2: BACKGROUND	7
2.1 Image Processing, Segmentation and Registration	7
2.2 Generic Programming	10
2.3 Graph theory and its application in image processing and analysis	11
2.3.1 Graph theory And Its Application To Segmentation.	12
2.3.2 Graph Theory And Its Application To Distance Transform	13
2.3.3 Graph Theory And Its Application To Registration	13
2.3.4 Graph Theory And Its Application To Centerline Extraction	14
2.3.5 Graph theory And Its Implementation Using Generic Programming	15
CHAPTER 3: GRAPH CLASSES	17
3.1 Process Objects in the Graph Library: the GraphToGraphFilter and ImageToGraphFilter General Design	17
3.2 Data Representation of Graphs	21
3.3 DataObject and ProcessObject Design that is Specific to Graph Algorithms	23
3.4 Graph Algorithms	25
3.4.1 Prim's Minimum Spanning Tree	25
3.4.2 Kruskal's Minimum SpanningTree	27
3.4.3 Dijkstra's Shortest Path Algorithm	28
3.4.4 Depth First Search Algorithm	30
3.4.5 Breadth First Search Algorithm	33

CHAPTER 4: EXPERIMENTATION AND APPLICATION	36
4.1 Experimentation	36
4.2 Application of the Graph Classes: Centerline Extraction	38
CHAPTER 5: CONCLUSIONS	46
REFERENCES	47
APPENDIX A: ITK ARCHITECTURE AND DESIGN.	49

## LIST OF FIGURES

FIGURE 2.1: Image Processing Pipeline	10
FIGURE 2.2: Graph	12
FIGURE 3.1: Graph Data Object and Process Object Inheritance Diagram	17
FIGURE 3.2: Graph Traits Inheritance and Relationship Diagram	21
FIGURE 3.3 Prim MST Graph Traits Inheritance and Relationship Diagram	23
FIGURE 3.4 Prim Minimum Spanning Tree Inheritance Diagram	24
FIGURE 3.5 Prim Minimum Spanning Tree Inheritance Diagram.	26
FIGURE 3.6 Dijkstra's Shortest Path Graph Traits Inheritance Diagram	29
FIGURE 3.7 Depth First Search Graph Traits Inheritance Diagram	31
FIGURE 3.8 Depth First Search Graph Filter Class Inheritance Diagram	32
FIGURE 3.9 Breadth First Search Graph Inheritance Diagram	34
FIGURE 3.10 Breadth First Search Graph Filter Inheritance Diagram	35
FIGURE 4.1 Skeleton of a Cylinder whose end points were predetermined	43
FIGURE 4.2 Skeleton of a Cylinder whose end points were calculated dynamically	44
FIGURE 4.3 Skeleton of an MRT data	45
FIGURE A.1. ITK Image Filter Data Pipeline	57
FIGURE A.2. Image DataObject and ProcessObject inheritance diagram(taken from the ITK Getting started series)	61
FIGURE A.3. Relationship between ProcessObject and DataObject	63
FIGURE A.4. Data Execution Pipeline	64

## CHAPTER 1: INTRODUCTION

Medical Imaging is the set of techniques and processes that are employed to create images of the human body for medical and clinical purposes [8, 9]. Its rapid development and proliferation has changed how medicine is practiced in the modern world. Medical imaging allows doctors and medical practitioners to non-invasively observe and analyze life saving information. Medical Imaging has gone beyond mere visualization and inspection of anatomic structures. Its application has expanded to include its use as a tool for surgical planning and simulation, intra-operative navigation, radiotherapy planning and for tracking the progress of a disease. For example, in radiotherapy, medical imaging allows a physician to deliver a necrotic dose of radiation to a tumor with very little collateral damage to the surrounding tissue.

Every year, millions of images of patients are taken which are in varying dimensions and sizes. Most of them are three-dimensional or four-dimensional images of patients taken in order to assist in diagnosis and therapy. There are many medical imaging modalities. The more commonly used ones are Computed Tomography (CT), Magnetic Resonance (MR), Ultrasounds (US), and Nuclear Medicine (NM) techniques such as Positron Emission Tomography (PET), as described in [8].

Although these modalities on their own have provided exceptional images of the internal anatomy, the application of computing technology to quantify and analyze the embedded structures within the image volumes with accuracy and efficiency has been

limited. In order to apply computing for biomedical investigations and clinical activities, accurate and repeatable quantitative data must be extracted efficiently. Unfortunately, at the current state, medical imaging has more qualitative features, with most observations of medical images done on two dimensional analogical supports such as films, typically on the basis of varying cross sections and a single modality at a time. Development of systems to support digital imaging of medical data has gained momentum over the past few years. It has lead to new and improved methodologies for storing medical images in digital formats on the production site and with the help of high bandwidth networks has open a new era with the potential for a much more efficient and powerful exploitation of images through image analysis and image processing, as has been explained in [8] and [9].

Image Analysis is the methodology by which information from images is extracted. Image analysis is mainly performed on digital images by using digital image processing techniques. Image processing consists of several techniques for feature extraction. Some of the images processing methodologies are segmentation, registration; dilating, eroding, skeletonization and distance transform calculation [7].

Interpreting quantitative data from two-dimensional analog cross sections is difficult and not very effective. In order to resolve that, digital image processing and analysis have introduced objective and three-dimensional measurements in the images. This allows precise extraction of data such as size, location and texture from three-dimensional anatomical and pathological structures. Visualization adds to the qualitative analysis of three-dimensional and four-dimensional images. All the above techniques are applicable for diagnosis of anatomical data. In order to apply image processing and analysis to



therapy, simulation is a methodology that helps in therapy. A virtual patient model helps in simulation of a particular therapy either with the help of a standard simulation of a patient or with the help of a template from the actual patient himself or herself although the latter will require a delicate intervention of that patient's anatomy [8, 9].

Image segmentation is one the earlier techniques applied to medical images, which are in the digital format. Segmentation is a technique by which an image is partitioned into physically meaningful regions. There are several approaches to segmentation. Many of these approaches have been adopted from various concepts such as mathematical morphology, energy minimization, partial differential equations, graph theoretic approach, or a combination of these approaches. There are no unique solutions and most segmentation problems are resolved using a combination of these techniques. Low-level image processing techniques for segmentation such as region growing, edge detection and mathematical morphology operations requires considerable amounts of expert interactive guidance. Deformable model based segmentation, on the other hand, can be applied without the need for slice editing and traditional image processing techniques and overcomes the latter's shortcomings that require the interactive guidance.

Following segmentation, the next image processing technique commonly used is image Registration. Image registration is the process of aligning images that are obtained, for example, at different times, from different sensors or from different viewpoints. Image registration can be performed intra- or inter- patient and between mono- or multi-modal three-dimensional images. This then leads to rigid and non-rigid image registration techniques as well as mono and multi-modal registration techniques. Some are feature based, using the result of a segmented image, for example, and some use raw image data.

There is a broad range of registration techniques that have been developed for different applications that use different mathematical concepts including correlation and sequential methods, Fourier transform methods, point mapping and model based mapping and graph theoretic based methods using prior information of the images for comparison and contrast for registration [8, 9].

Image processing also involves motion tracking of four-dimensional models as well. There are two objectives for motion tracking of four-dimensional models. First objective is the tracking of the boundaries of some anatomical structures to estimate the displacement field of a volume. Second objective is the quantification of the over all motion with a few objective and significant parameters. Deformable model based image processing techniques are commonly used for motion tracking [8].

Along with other techniques for medical data analysis, visualization of medical data is a common analysis tool that had gained some popularity before in terms of research contribution to the field. In general visualization requires a preliminary segmentation stage before any beneficial data maybe gathered from the technique. It also employs three-dimensional interactive graphics for the visualization of three-dimensional anatomical structure [9].

An image-processing tool that has seen wide applications in the medical imaging field is skeletonization. Skeletonization is the image processing technique that reduces the foreground in a binary image to a skeletal remnant that largely preserves the extent and connectivity of the original region while throwing away most of the original foreground pixels [7, 11, 12, and 13]. Skeletonization works by starting from the boundary of a

volume and traversing to a common mid point within the volume. Skeletonization is seen as connected centerlines, consisting of medial points of consecutive clusters [10].

This tool can be used for compact shape description, path planning and other applications such as motion tracking. Skeletonization of a three dimensional volume usually compacts the discrete objects within the volume and it provides an efficient method for visualization and analysis, such as feature extraction, feature tracking, surface generation, or automatic navigation [8, 10].

The image-processing techniques mentioned before can be improved using a graph theoretic approach. In segmentation, for example, graph theory has been successful applied by using graph search algorithms to find edge boundaries. Similarly, in registration, graph theory is applied to calculate the alignment of the pictures based on features of the images or mathematical constraints such as entropy by using the minimum spanning tree algorithm to calculate the minimum value of deviation and so on. Graph theory is also applicable in calculating the distance transform of the pixels in the images and the shortest path algorithm is applied to calculate the distance between the interior pixels and the contour pixels. With the help of distance transform calculation, the skeleton of a volume can be calculated along with the help of minimum spanning tree to calculate the minimum distance from the outer contours of the volume to the interior medial points [2, 3, 4, 5, 6, and 7].

As has been discussed above, graph theory has many applications in the image-processing pipeline. However, few image processing toolkits have implemented the graph algorithms in a generic and object oriented manner that would allow it to be applied for medical image processing. Although Boost [17] has a Graph library that is generic in

nature, it didn't design its software to be specific to image processing. The design proposed in this document for graph algorithms have been made keeping in mind the generic nature of the images used by medical images and also the specific nature of the data processed. Analyze [18] is another medical image processing toolkit that is object oriented in nature but not generic and hence has its own limitations. Analyze also uses Insight Registration and Segmentation Toolkit for some of its segmentation and registration calculations. Insight Registration and Segmentation Toolkit employs a generic and object oriented framework.

In this thesis we present this framework and extending it to include graph objects amongst its Data Objects, which uses an object factory design model, that can be easily generalized by a user, four graph algorithms have been proposed that has implemented these algorithms to be efficient and generalized for large multi-dimensional medical images. It has also been designed to be independent of the metadata of an image. The algorithms implemented include Prim's minimum spanning tree, Kruskal's minimum spanning tree, Dijkstra's shortest path, depth first search and breadth first search. Each is designed to be an In-place filter that is they use the same memory for the input and output graphs. Their outputs are usually parent node and child node map structure lists. Several supporting algorithms such as functors, which help determine the logic that is to be used for the application of the graph algorithms or for the conversion of images to their representational graphs, have been designed in order to make it an effective application in the generic framework. The functors follow the software design method commonly known as the visitor design. Other supporting classes that are specific to the implementation have also been made.

Tests have been conducted for the robustness and the efficiency of the application. On average the time it takes to execute a single pipeline from conversion of the image to graph to the assignment of an object the values of the map container containing the parent and child vertices of the graph, was around 12 to 24 milliseconds. An application of this framework on three-dimensional objects for skeletonization has shown its applicability to larger volumes and its relative time efficiency.

## CHAPTER 2: BACKGROUND

### 2.1 Image Processing, Segmentation and Registration.

*Image processing* is a form of information processing which is applied to images and video sequences. Image processing has the following steps, as it has been explained in [7]:

- *Image acquisition* involves using an image sensor and hardware capable of changing the signal data from the sensors to a digitized or analog form.
- *Preprocessing* typically deals with techniques that improve the data quality and success of subsequent processes in the image processing pipeline. For example, noise reduction, contrast enhancement and so on.
- During *Segmentation*, an input image is partitioned into its constituent parts or objects. The output image of this stage usually consists raw pixel data which either defines the boundary of a region or all points of the region in itself. Whether the data constitutes the data of the boundary of a region or the region is dependent upon the application of the image process.
- Also known as *Feature Selection, Description* is the stage where features are extracted that are of some quantitative information of interest or features within the image that are basic for differentiating one class of objects from another.

- *Recognition* is the process that assigns labels to an object based on the information provided by descriptors. Interpretation involves assigning a meaning to an ensemble of recognized objects or labeled entities.
- The *Knowledge Base* is responsible for guiding the operation of the image processing module and for controlling the interaction between modules. In the first case, the knowledge base holds information about a problem domain in the form of a database that is coded into the image processing system. The knowledge base can either as simple as detailing the regions of an image where the information of interest is located or it can be complex where a list of inter-related and possible defects in a materials inspection problem. In the second case, it depicts that the communication between two modules are dependent upon the prior knowledge of what the result should be for a processing module.
- *Image Registration* is the process module used during the reconstruction of three-dimensional objects from serial sections. It involves overlaying two images of the same scene by translating and rotating the objects in the images so that corresponding points of two images are coincident with one another.

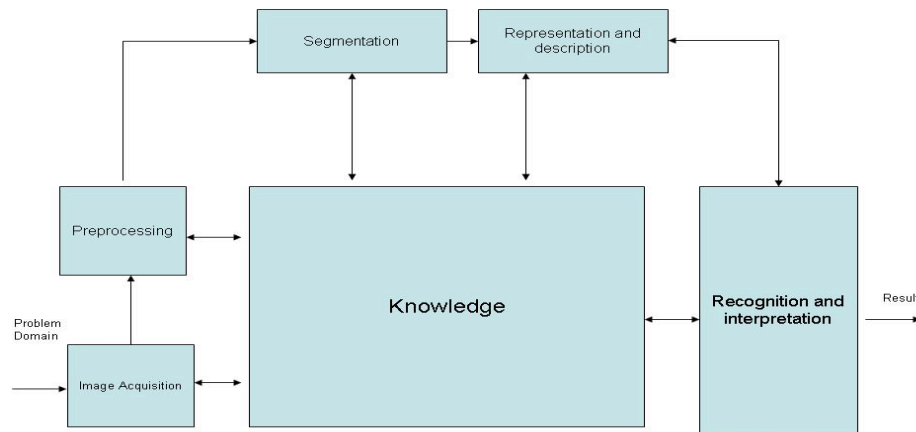


Figure 2.1 Image Processing Pipeline

## 2.2 Generic Programming.

Generic programming is a method of organizing libraries consisting of generic or reusable software components. Generic programming usually consist of containers that hold data, iterators to access the data and generic algorithms that use containers and iterators to create efficient fundamental algorithms such as sorting. It is implemented in C++ using template programming mechanism and the use of the STL (Standard template Library). C++ template programming is a programming technique that allows users to write software in terms of one or more unknown types  $T$ . A user of the software defines the type  $T$  in order to create executable code. The  $T$  may be a native type such as *float* or *int* or  $T$  may be a user-defined type (e.g. class). At compile time the compiler makes sure that the template types are compatible with the instantiated code and that the types are supported by the necessary methods and operators.



The advantage of using generic programming is that simply defining the appropriate template types supports an almost unlimited variety of data types. For example, in ITK it is possible to create images consisting of almost any type of pixel. The type resolution is done during compile time, so the compiler can optimize the code to deliver maximal performance. The disadvantage of generic programming is that many compilers usually do not support this high level of abstraction. Although some compilers may support generic programming, they may produce undecipherable code even for some of the simplest of errors.

### 2.3 Graph Theory And Its Application In Image Processing and Analysis.

A Graph  $G$  is said an ordered triple  $(V(G), E(G), \psi_G)$  where  $V(G)$  is a nonempty set of vertices,  $E(G)$  is a set of edges disjoint from  $V(G)$  and an incidence function  $\psi_G$  that associates with each edge of  $G$  an unordered pair of vertices of  $G$  which are not necessarily distinct. If  $e$  is an edge and  $u$  and  $v$  are vertices such that  $\psi_G(e) = uv$ , then  $e$  is said to join  $u$  and  $v$ ; the vertices  $u$  and  $v$  are called the ends of  $e$  [7, and 14]. For example,

$$G = (V(G), E(G), \psi_G)$$

where,  $V(G) = \{v_1, v_2, v_3, v_4, v_5\}$ ,

$$E(G) = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7, e_8\}.$$

And,  $\psi_G$  is defined as

$$\psi_G(e_1) = v_1v_2, \psi_G(e_2) = v_2v_3, \psi_G(e_3) = v_3v_3, \psi_G(e_4) = v_3v_4,$$

$$\psi_G(e_5) = v_4v_2, \psi_G(e_6) = v_6v_5, \psi_G(e_7) = v_2v_5, \psi_G(e_8) = v_2v_5.$$

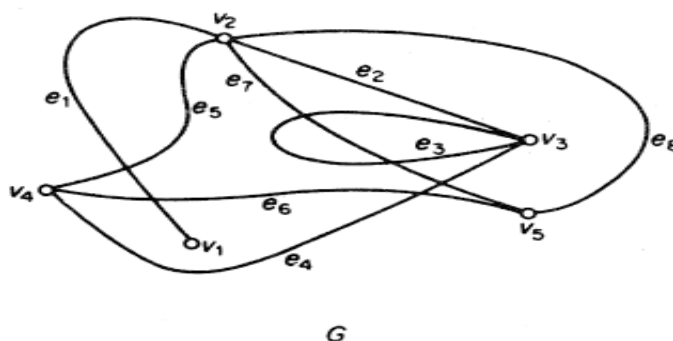


Figure 2.2 Graph

Graph theory has many applications in Image processing, especially in processes such as Segmentation and Registration. The following describes how graph theory is applied in segmentation, registration, distance mapping and skeletonization.

### 2.3.1 Graph theory And Its Application To Segmentation.

Graph theory can be used in segmentation of an image if the shape of the segmented region is dependent on the scanning direction. Graph theory is applied to the smoothed image. Segmented regions are given higher precise boundaries. In order to maintain the homogeneous property, a threshold value is induced to maintain the merging process of the graph theory. The following steps are used for application of graph theory on segmented images. 1) A Graph is created where each image pixel corresponds to a graph vertex and the vertex weight corresponds to the pixel value or the pixel intensity. Each vertex is linked to its adjacent vertices in 4 perpendicular directions. 2) The linked weight or the edge weight is determined by either the absolute difference in intensity value of each vertex or any other formula for comparison between the two vertices. 3) Merge two vertices by removing the link between them if the link weight is less than the threshold

value. The vertices are connected by the removed link are merged together with respect to the homogeneity criterion. The weight of the merged vertices will be replaced by their average value, therefore the link weights around these vertices have to be updated. 4) Region boundaries are created by different average values of the linked weights which are used to separate by the boundary links [14, 3, 7 and 8].

### 2.3.2. Graph Theory And Its Application To Distance Transform.

Graph theory can also be used to calculate the distance transform of an image using the classical shortest path algorithm in Graph Theory. The distance transform can be formulated as a graph theoretic problem by building a graph from the binary image whose distance transform we want to calculate and a neighborhood relationship between the pixels. The edges between the vertices are representing the desired distance metric. The tree roots are the external contour pixels of the objects. The shortest path from each pixel to the nearest root pixel is calculated. The distance image is the image which the pixel value is the path length to the nearest background pixel (root) [4, 9, 11, and 12].

### 2.3.3. Graph Theory And Its Application To Registration.

Graph theory can be used for image registration especially minimum spanning tree graph algorithm. To apply minimum spanning tree to image registration, one can use the fact that for perfectly aligned images their entropy value of the overlapped images would be minimum and hence the total length of the corresponding graph is the shortest amongst all the possible overlapping of the two images. Given the two noisy images  $I_1$  and  $I_2$ , to register these images, the following steps can be applied. 1) Feature vectors  $F_1$  and  $F_2$  are extracted from their respective images  $I_1$  and  $I_2$ . 2) Noise contamination

removal via  $k$  means MST method. 3) An Initial spatial transformation  $T$  is chosen. 4) Apply transformation  $T$  to  $F_1$  and merge the resulting feature vectors  $T(F_1)$  into  $F_2$ . 5) Calculate the minimum spanning tree of the graph generated from the bitmaps of the overlapped feature vectors. 6) Refine the transformation  $T$ . The steps 4 to 6 are repeated until the dissimilarity metric comes to its minima [5, 8, 9,7,11, and 12].

#### 2.3.4. Graph Theory And Its Application To Centerline Extraction.

Another application of graph theory is the extraction of the centerline and the skeletonization of a medical image volume. Blum et al introduced the concept of centerline or medial or symmetric axes [8]. In a tubular object such as the colon, for example, a single centerline usually spans the entire length of the volume. If an object is more complicated in shape then it may have several centerlines attaching each other through the object. The topology of such a group of centerlines usually forms something that looks similar to the skeleton of the object. Hence, the set of connected centerline of an object is also called skeleton and the process of extracting the skeleton is called skeletonization [11].

One of the applications of centerline extraction is Virtual Endoscopy. Some methods for finding the centerline in virtual endoscopy, for example, are distance mapping and topological thinning which uses graph theory to extract the centerlines. The other methods are manual and does not use graph theory [11]

Virtual endoscopy using topological thinning consists of three steps, as described in [11]. The first step is to process medical images and segment them to get structures of interest from the images. The second step consists of generating flight steps within these structures to obtain potential camera positions. The third and final step consists of

generating perspective views of the internal surfaces of such structures with the given flight paths and using conventional graphics rendering techniques. The calculation of the flight steps may be automatic or semi automatic. Most automatic centerline extraction algorithms make extensive use of graph theory to calculate the flight paths of the volumes processed. Some of the methods to calculate the centerline take into consideration the features of the images such as the Euclidean distance of the pixels from the boundary so as to keep the minimal paths of the flight paths centered. Another technique to calculate the minimal flight path is use the same Euclidean distance but formulated in a discrete setting. A discrete setting for such a methodology is to represent a coarse representation of the three dimensional volume as a graph. Each edge of the graph is assigned a weight, which is the combination of the Euclidean distance from the boundary object. The Dijkstra's shortest path algorithm is applied to this graph to extract the minimal flight path, which in turn represents the centerline of the volume.

Extracting the centerline in virtual endoscopy using distance mapping involves two phases. This method is also known to be one of the fastest methods for centerline extraction. The first phase consists of computing the distance from a user-specified or computed source point to each voxel inside the 3-D object, known as the distance from the starting point. After this distance is calculated, the second phase involves calculating and extracting the shortest path from any starting voxel to the source point by descending through the gradient of the distance map. If it is used to create the distance map The shortest path can be rapidly extracted by a simple back tracing algorithm, rather than the steepest descent [12].

### 2.3.5. Graph theory And Its Implementation Using Generic Programming.

There have been several designs used to implement graphs and their algorithms in a generic fashion. Most graph algorithms require some modification in order to make the data output meaningful. Hence, most graph implementation is done for a specific application. Some applications such as the Boost Library [18] have implemented graphs for a generic framework. However, they have not made it designed for medical image analysis specifically. Insight Segmentation and Registration Toolkit [13] implements a generic framework but lacks a comprehensive application of graph data objects in their library. Analyze [17], which is medical image analysis software, has been implemented using a procedural language and hence cannot take advantage of generic program designs. The graph framework proposed takes into consideration the generic capabilities of the toolkit while designing its functionalities.

## CHAPTER 3: GRAPH CLASSES USING A GENERIC FRAMEWORK

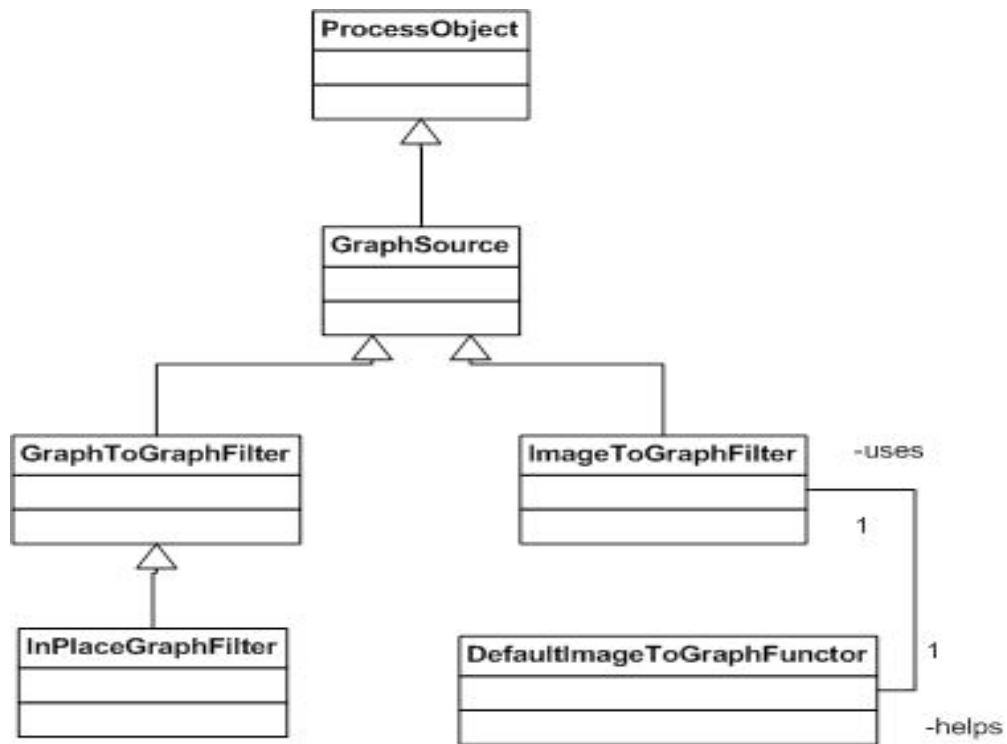


Figure 3.1 Graph Data Object and Process Object Inheritance Diagram

As seen in the earlier chapter, the image data object and its filters follow a hierarchy in terms of functionalities. To emulate the same functionalities, the same hierarchy was followed and was used for the design for the Graph library.

### 3.1 Process Objects In The Graph Library: The GraphToGraphFilter and ImageToGraphFilter General Design.

The proposed graph library follows similar design and architecture as that of the *Image* class and the *ImageToImageFilter* class. As we have seen in the case of the *Image* class and *ImageToImageFilter* design, all filter classes are inherited from *itk::ProcessObject*. The class called *GraphSource*, which has a design similar to that of *ImageSource* for the *ImageToImageFilter* inheritance diagram, is a base class for all process objects that output graph data. Specifically, this class defines the *GetOutput ()* method that returns a pointer to the output graph. The class also defines some internal private data members that are used to manage streaming of data. The *GraphSource* also has the definitions for *GraftOutput ()*. The *GraftOutput ()* function takes the specified *DataObject* and maps it onto the *ProcessObject*'s output. The method or function then grabs a handle to the specified *DataObject*'s bulk data to use as its output's bulk data. It also copies the *ivars* and the meta-data from the specified data object into the *GraphSource* object's output *DataObject* [2].

An *ImageToGraphFilter* is sub-classed from *GraphSource*. The *ImageToGraphFilter* converts a given image to a graph depending upon the specification determined by *DefaultImageToGraphFunctor*. A *DefaultImageToGraphFunctor* is a class that defines which pixels will be considered nodes and which will not be considered nodes and also to decide on the edges of the graph. The *DefaultImageToGraphFunctor* is a sub-class of the *ImageToGraphFunctor* class. The *ImageToGraphFunctor* has the methods *GetEdgeWeight ()*, *GetNodeWeight ()*, *IsPixelANode ()* and *NormalizeGraph ()* declared in this definition. The *GetEdgeWeight* is a virtual function that, when defined, is used to define the edge weight between two nodes. It accepts the indexes of the first and second pixels between which the nodes are defined. The *GetNodeWeight ()* method is a



virtual function that, when defined, is used to assign the node weight to a node. Usually it is the image pixel value but can be defined to represent data that is not related to the pixel value. The *IsPixelANode* () method is used to define the pixels that will be determined as a node. Usually it is easy to define the method when it involves a segmented image. *NormalizeGraph* lets the user make adjustments to the graph. *SetRadius* () is used to define the length of the radius of the *NeighborhoodIterator* that is used to determine the edge weights and the nodes that form edges. Several *itkGetMacro* and *itkSetMacro* functions are defined for the variables *ExcludeBackground*, which is a Boolean type, and *BackgroundValue*, which is a *PixelType*. *ExcludeBackground* is set to **true** if the output graph is constructed from a sub-region. *BackgroundValue* defines the pixels whose values are not considered as being part of the sub-region which would form the graph. The *DefaultImageToGraphFunctor* implements the methods *GetNodeWeight*, *GetEdgeWeight*, *IsPixelANode* and *SetRadius*[2].

The *ImageToGraphFilter* applies the logic for converting the image into a graph with the help of the functor classes mentioned above. The initial design proposed by [2] had the *ImageToGraphFuntor* class defined as a variable and the class took a *DefaultImageToGraphFunctor* variable that was not a template class. Changes have been made to the template parameters to allow an *ImageToGraphFunctor* class as a template parameter, thereby, allowing users to change the logic implemented for the graph while converting it from an image. The *ImageToGraphFilter* also defines the *SetInput* function of the base class *GraphSource* and defines the method to accept a variable of the *itk::Image* class.

Another class that inherits from *GraphSource* is the *GraphToGraphFilter* which contains the graph algorithm that acts on the Graph data object [2]. The *GraphToGraphFilter* takes in an *itkGraph* template parameter as an input and another class definition of the *itkGraph* class as an output. The *GraphToGraphFilter* dedicates two separate data object variables for the input and output graphs. For large images this could mean a lot of space in terms of memory and execution time. As the *ImageToGraphFilter* defines the *SetInput* method to accept an *itk::Image* class, the *GraphToGraphFilter* defines the *SetInput* method to accept an *itk::Graph* class. This class is used as a base class for all filters that accept an *itk::Graph* as an input and outputs an *itk::Graph* as well.

*InPlaceGraphFilter* is a sub class of *GraphToGraphFilter*, which has only one template parameter and is implemented if both the input and output have the same definition for the Graph class as a template parameter and if the memory of the input data is shared by the output data. Otherwise, the definition of *GraphToGraphFilter* is used instead, if the input and output definitions for the Graph data object are different. This class saves memory as most medical images generally large in size.

### 3.2 Data Representation of Graphs.

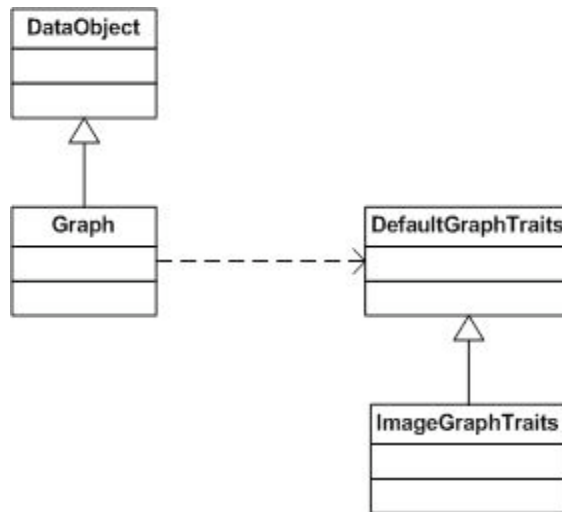


Figure 3.2 Graph Traits Inheritance and Relationship Diagram

*Itk::Graph* is a class that is sub classed from *DataObject* class like the *Image DataObject*. *Itk::Graph* references *DefaultGraphTraits* which defines the Graph Node and Graph edge structure types. *ImageGraphTraits* inherits from *DefaultGraphTraits* and defines the properties of the Node and Edge structure specifically for an image by storing the pixel data as well [2].

The definition of the nodes and edges is provided by template parameters for *itk::Graph* which are sub classes of *DefaultGraphTraits*. The Node structure contains a unique number or key that is the Identifier of the node, a vector container that holds all the outgoing edge indices, a vector container that holds all the incoming edges and the

node weight. The Edge structure consists of the unique edge key or Identifier, the Identifier of the source node and the Identifier of the edge node. An identifier for any corresponding reverse edges is also stored, and the final field is the edge weight. The data type for both the edge and node weight is determined by a template parameter that is used to define the *DefaultGraphTraits* class. . *ImageGraphTraits* adds the Index of the pixel that is being represented by the node to the node structure as an *itk::Index* variable along with the other fields [2].

*Itk::Graph* consists of containers that hold the nodes and edges. These containers are vectors. *CreateNewEdge ()* and *CreateNewNode ()* are defined to create nodes and their corresponding edges. These classes are also overloaded to define several other possibilities in input parameters. *GetNode ()* and *GetEdge ()* return the corresponding node and edge. Variations of these methods have also been defined. *GetNodePointer* and *GetEdgePointer* returns a pointer variable of the requested node and edge respectively. *SetNodeContainer* and *SetEdgeContainer* sets the node and edge containers whereas the *GetNodeContainer* and *GetEdgeContainer* allows access to the node and edge containers. There variations of the *GetNodeWeight* and *GetEdgeWeight* methods that return the node and edge weights respectively. There are methods to change the weight of a node and edge as well. Most methods take the identifier of the node as a parameter. The *itk::Graph* also has two friend classes defined. They are the *NodeIterator* and the *EdgeIterator*. The *NodeIterator* is an Iterator class. An Iterator class is a class that iterates through the data object element by element and has several operators overloaded. The *NodeIterator* class iterates over the nodes if the *itk::Graph* class. The *EdgeIterator* class iterates through the

edges of the graph. Both Iterators have a *GoToBegin ()* and *IsAtEnd ()* classes. The *GoToBegin ()* method goes to the start of the container holding the nodes or edges. The *IsAtEnd ()* method checks whether the Iterator has reached the end of the container. The *GetPointer()* method returns a pointer to the node or edge that the Iterator is currently pointing to. *Get ()* returns the node or edge that the Iterator is pointing to. *GetIdentifier ()* just returns the Index of the node container in the list [2].

### 3.3 DataObject and ProcessObject Design That Is Specific To Graph Algorithms

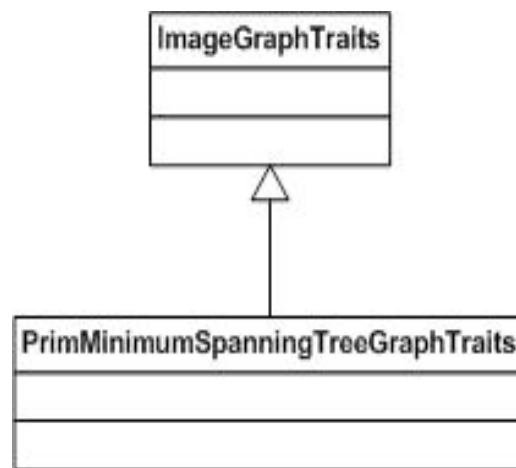


Figure 3.3 Prim's MST Graph Traits Inheritance and Relationship Diagram

For the proposed graph algorithm, the *ImageGraphTraits* class is further sub classed to be more specific for each graph algorithm. For the Prim's Minimum Spanning Tree Graph algorithm, the *PrimMinimumSpanningTreeGraphTraits* has been defined. This is sent as a template for the *PrimMinimumSpanningTreeGraphFilter*. The *PrimMinimumSpanningTreeGraphTraits* has a node structure which is similar to that of *ImageGraphTraits* but has some additional fields. In addition to the Identifier, the two

containers of edges for outgoing edges and incoming edges, and the node weight, the node structure also has fields that describe the color of each node and the key weight of each node or the corresponding distance weight at each node. The *color* field is an enumerate type named *ColorType* which has three colors defined: White, Gray and Black. The *KeyWeight* field is of type *EdgeWeight* whose data type is defined by the template parameter of the *ImageGraphTraits* class.

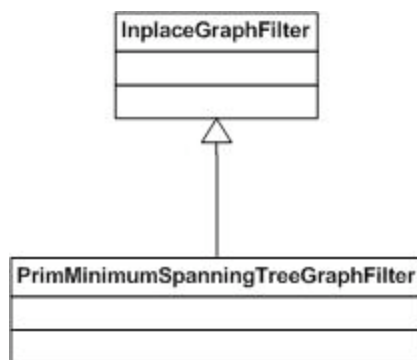


Figure 3.4 Prim Minimum Spanning Tree Inheritance Diagram

For Example, the *InPlaceGraphFilter* and the *DefaultImageToGraphFuncion* in turn are inherited by *PrimMinimumSpanningTreeGraphFilter* and *PrimMinimumSpanningTreeImageToGraphFuncion* class respectively. The *PrimMinimumSpanningTreeGraphFilter* implements the Prim Minimum Spanning Tree algorithm while maintaining the interface from *InPlaceGraphFilter*. *PrimMinimumSpanningTreeImageToGraphFuncion* defines the logic by which a node is determined from a pixel or not and whether or not a particular relation between two nodes can be considered an edge.

### 3.4 Graph Algorithms

The next few sections will discuss the algorithms that have been developed for the Insight Segmentation and Registration Toolkit. The first section is about the Prim's minimum spanning tree algorithm and its implementation details. The second section is about the Dijkstra's shortest path algorithm following that is the breadth first search algorithm and the depth first search algorithms.

#### 3.4.1 Prim's Minimum Spanning Tree.

```

Prim(Graph G, Node root)

Initialize parent;
Initialize PriorityQueue;
DistanceWeight = 0;

for each node in V(G)
    node.color=WHITE;
    node.key=INFINITY;

root.color=GRAY;
root.key=0;
parent [root] =root;
insert(PriorityQueue, root);

while (! PriorityQueue.empty())
    Node u=PriorityQueue.min ();
    for edge in E(G, u)
        Node v=DestinationNode(edge);
        if (edge.Weight < v.key)
            parent[v]=u;
            v.key=edge.Weight;
            if(v.color=WHITE)
                v.color=GRAY
                insert(PriorityQueue, v);
            else if(v.color=GRAY)
                Update(PriorityQueue, v);
    u.color=BLACK;
    DistanceWeight=DistanceWeight + u.key;

```

Generic minimum spanning tree algorithm is a greedy algorithm that grows a minimum spanning tree one edge at a time. Prim's Minimum Spanning Tree algorithm is

a generic minimum spanning tree algorithm [14]. The Prim's algorithm has the property that the edges of a set  $A$  that forms the minimum spanning tree forms a single tree. The tree starts from an arbitrary root vertex  $r$  and grows until the tree spans all vertices in  $V$ . At each step, a light edge is added to the minimum spanning tree  $A$  whose one vertex is in  $A$  and the other is in  $V-A$ . This strategy is said to be greedy as the tree augmented at every step adds an edge that has a minimum amount possible to the tree's weight. During the execution of the algorithm, all vertices that are not in the tree reside in the Priority queue based on the key field. For each vertex  $v$ ,  $\text{key}[v]$  is the minimum weight of any edge connecting  $v$  to a vertex in the tree. By convention, the key value of a vertex  $v$  is equal to infinity if such a vertex does not exist. The parent of a node  $v$  stores the source node of the edge that is part of the minimum spanning tree and is the parent of the node  $v$ . The algorithm terminates when the priority queue is empty.

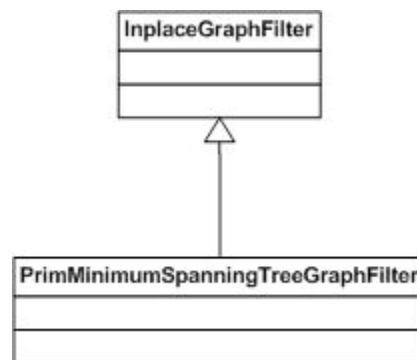


Figure 3.5 Prim Minimum Spanning Tree Inheritance Diagram

The `itk::PrimMinimumSpanningTreeGraphFilter` is a subclass of `InPlaceGraphFilter` that implements the Prim's Minimum Spanning Tree algorithm on an



*itk::Graph* Data Object. It takes in *itk::Graph* as a template parameter and also as an Input. It outputs an *itk::Graph* Data Object that is similar to the Input except that it now contains the distance weight or key weight for each node and also the updated color sequence. In order to access the parent node of each node that forms the minimum spanning tree, the function call *GetMinimumSpanningTree ()* returns a *ParentNodeListType* that is an *itk::MapContainer* type. The *itk::MapContainer* takes a *NodeIdentifierType* which stores the child node's identifier variable as the Element identifier and the Element of the *itk::MapContainer* is another *NodeIdentifierType* which stores the parent's node identifier. The *itk::PrimMinimumSpanningTreeGraphFilter* also calculates the total weight of the Minimum Spanning tree and that weight can be accessed via the *GetDistanceWeight ()* method call which returns a double value type. In order to be able to assign a different node as the starting node of the algorithm, the class has a *SetRootNodeIdentifier ()* method call which accepts a *NodeIdentifierType* as a parameter. The identifier is that of the node which is to be the starting node of the algorithm.

### 3.4.2 Kruskal's Minimum Spanning Tree Algorithm

Kruskal's algorithm as is described in [14], is a minimum spanning tree algorithm that is greedy by nature. In the Kruskal's minimum spanning tree algorithm, the edges of the graph are sorted by their weight. Two separate trees are maintained. One of them is the growing minimum spanning tree and the other contains the rest of the nodes of the graph. As long as there are edges that are still part of the graph and not part of the minimum spanning tree, an edge is added to the minimum spanning tree that is the minimum in the current graph and whose nodes aren't part of just one of the sub-graphs

but has one node in the growing tree sub-graph and one of the node in another. The tree is complete when all the nodes are in the minimum spanning tree sub graph.

*Itk::KruskalsMinimumSpanningTreeGraphFilter* implements the Kruskal's minimum spanning tree Algorithm. Like the *itk::PrimMinimumSpanningTreeGraphFilter*, it is a sub class of the *InPlaceGraphFilter* and accepts as a template parameter an *itk::Graph* class type that describes the input and output graph types. The output graph of this class just returns the original graph with the updated node weights. The filter also has a *GetMinimumSpanningTree ()* method call that returns a *ParentNodeListType* which is a *std::map* container type and contains the nodes of the minimum spanning tree. The key type for the *std::map* is a *NodeIdentifierType* which is usually an unsigned integer, and the value type of the *std::map* is also a *NodeIdentifierType*. The key type stores the child node whereas the value type stores the parent of the child node that is part of the minimum spanning tree of the graph.

### 3.4.3 Dijkstra's Shortest Path Algorithm

```

DijkstrasShortestPath(Graph G, node root)
PriorityQueue=NULL;
For node in V(G)
    DistanceNode[node]=INFINITY;
    Insert(PriorityQueue , node);
DistanceNode[root]=0;
Color[root]=GRAY;
Parent[root]=root;
While( !PriorityQueue.empty)
    U=PriorityQueue.min
    Set(u);
    For edge in E(G, u)
        v=DestinationNode[e];
        if(DistanceNode[u]+e.Weight <DistanceNode[v])
            DistanceNode[v]=DistanceNode[u]+e.Weight;
            Parent[v]=u;
    Color[u]=BLACK;

```

Dijkstra's algorithm, as explained in [14], tries to calculate the single shortest path between two vertices or nodes on a weighted directed graph for the case in which all edge weights are non-negative. Dijkstra's algorithm maintains a set  $S$  of vertices whose final shortest path weights from the source  $s$  has been determined. The algorithm repeatedly selects a vertex  $u$  which is not in set  $S$  with the minimum shortest path estimate, inserts the vertex  $u$  into  $S$ , and relaxes or decreases the value of all edges leaving  $u$  or for which  $u$  is the source of the edge. A priority queue holds all the vertices that are not in the set  $S$  keyed by their  $d$  values, where  $d$  is defined as the minimum distance of a node. Dijkstra's algorithm, like Prim's minimum spanning tree algorithm, is a greedy algorithm as it chooses the closest or the lightest edge vertex amongst the vertices that are not in  $S$  to be added to  $S$ .

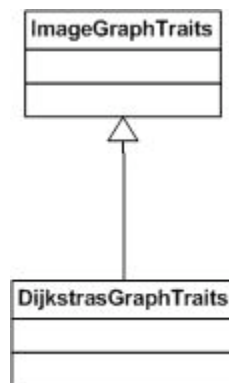


Figure 3.6 Dijkstra's Shortest Path Graph Traits Inheritance Diagram

*Itk::DijkstrasShortestPathGraphFilter* implements the Dijkstra's shortest path algorithm. Like the *itk::PrimMinimumSpanningTreeGraphFilter*, it is a sub class of the *InPlaceGraphFilter* and accepts as a template parameter an *itk::Graph* class type that describes the input and output graph types. The output graph of this class just returns the original graph with the updated node weights. The filter also has a *GetShortestPath ()* method call that returns a *ParentNodeListType* which is a *std::map* container type and contains the nodes of the shortest path. The key type for the *std::map* is a *NodePointerType* which is a *NodeType* pointer, and the value type of the *std::map* is also a *NodePointerType*. The key type stores the child node whereas the value type stores the parent of the child node that is part of the shortest path of the graph. The *GetDistanceNode ()* method call returns the distance associated with each edge in the shortest path. The *DistanceNode* variable is of the *std::map* type. The key type of the *std::map* is a *NodePointerType* and the value type is that of an *EdgeWeightType* which is defined by *itk::GraphTraits*. The function call *SetRootNode ()* takes a *NodePointerType* that can let the user define any other node as the root or start node of the Dijkstras Shortest Path. The *GetRootNode* returns the *RootNode* as a *NodePointerType*.

### 3.4.4 Depth First Search Graph Algorithm

```

DepthFirstSearch (Graph G, Node root)
for node in V (G)
  color [node]=WHITE;
  parent[v]=NULL;
time=0;
DepthFirstSearch_Visit (root);
For node in V (G)
  If (color [node]=WHITE)
    DepthFirstSearch_Visit (node);

DepthFirstSearch_Visit (node u)
Color[u] =GRAY;
DiscoveredTime[u] =time+1;
Time=time+1;

For e in E (G, u)
  v=DestinationNode[e];
  if (color[v] =WHITE)
    DepthFirstSearch_Visit (v);
Color[u] =BLACK;
FinishTime[u] =time+1;
Time=time+1;

```

Depth first search algorithm is a graph search algorithm that explores edges from a recently discovered vertex  $v$  that still has unexplored edges leaving it, [14]. When all of the edges of a vertex have been explored, the algorithm backtracks to explore edges leaving the vertex from which the vertex  $v$  was discovered. This continues until all the vertices from the source or root vertex or node are reached. If any vertex is left undiscovered, then one of them is selected as a new source and the search is repeated from that source. This process is repeated until all vertices are discovered. The predecessor sub-graph of a depth first search is therefore defined slightly differently from that of breath first search algorithm or minimum spanning tree predecessor sub-graphs. The predecessor sub-graph of a depth first search algorithm does not form just a tree but a forest and is called a depth first forest that is composed of several depth first trees. Each

vertex has a time when it is discovered and when it is grayed, and a time when it is finished, and when it is blackened. This technique makes sure that each vertex belongs to a single tree in the depth first forest. These times are helpful in understanding the behavior of the depth first search.

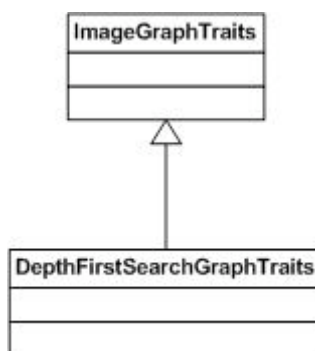


Figure 3.7 Depth First Search Graph Traits Inheritance Diagram

*Itk::DepthFirstSearchGraphFilter* implements the Depth First Search Graph Algorithm. This class is a sub class of *InPlaceGraphFilter* and takes an *itk::Graph* template class as a parameter for the input and output. The algorithm produces a *ParentNodeListType* of `std::map` container type that can be accessed by the *GetDepthFirstSearch*. The key value holds the *NodePointerType* of the child node and the value type of the `std::map` container is a *NodePointerType* and is the parent node. The *GetDiscoveredTime* and the *GetFinishedTime* are method calls that each returns a *TimeNodeType*. *TimeNodeType* is a `std::map` container that consists of *NodePointerType* as the key type and a double as the value type. *SetRootNodeIdentifier* allows a user to set

the start node of the algorithm to any other node besides the default node. *GetTime* will return the total time of the Depth First Search algorithm and it returns a double value.

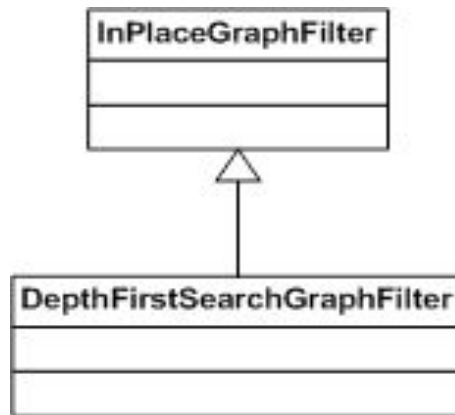


Figure 3.8 Depth First Search Graph Filter Class Inheritance Diagram

### 3.4.5 Breadth First Search Graph Algorithm

```

BreadthFirstSearch(Graph G, Node root)
For node in V(G)
    Node.color=WHITE;
    DistanceNode[node]=INFINITY;
DistanceNode[root]=0;
Root.color=GRAY;
Parent[root]=root;
Insert(Queue,root);

While(!Queue.empty)
    U=Queue.pop;
    For edge in E(G,u)
        V=DestinationNode[e];
        If(v.color=WHITE)
            v.color=GRAY;
            DistanceNode[v]=DistanceNode[u]+1;
            Parent[v]=u;
        Queue.push(v);
    u.color=BLACK;

```

Breadth first search is a simple graph search algorithm, [14]. Given a graph and a source or root node, breadth-first search systematically explores the edges of the graph to discover every vertex that is reachable from  $s$ . It computes the distance from  $s$  to all such reachable vertices. For any vertex  $v$  reachable from  $s$ , the path in the breadth-first tree from  $s$  to  $v$  corresponds to a “shortest path” from  $s$  to  $v$  in  $G$ , that is, a path containing the fewest number of edges. The algorithm works on both directed and undirected graph. To keep track of all the nodes that the algorithm traverses and the progress of the algorithm, all vertices are colored white, gray and black. All vertices start out white and become gray and black if the vertex is discovered while traversing the graph. All vertices which are black have been discovered. Vertices that are gray may have some undiscovered white vertices and they are a frontier between discovered and undiscovered vertices. Each discovered vertex has a single parent node as it is discovered at most once.



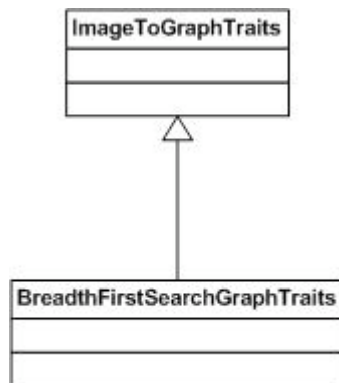


Figure 3.9 Breadth First Search Graph Inheritance Diagram

*Itk::BreadthFirstSearchGraphFilter* implements the Breadth First Search Graph Algorithm. This class is a sub class of *InPlaceGraphFilter* and takes an *itk::Graph* template class as a parameter for the input and output. The Algorithm produces a *ParentNodeListType* of `std::map` container type that can be accessed by the *GetBreadthFirstSearch*. The key value holds the *NodePointerType* of the child node and the value type of the `std::map` container is a *NodePointerType* and is the parent node. *SetRootNode* allows a user to set the start node of the algorithm to anything other node besides the default node by taking in as a parameter a *NodeIdentifierType*. *GetDistanceWeight* () returns the distance weight, which is double value. *GetDistanceNode* () returns a `std::map` container called *DistanceNodeType* where the key type is a *NodePointerType* and the value type is an *EdgeWeightType* whose type is defined by *itk::GraphTraits*.

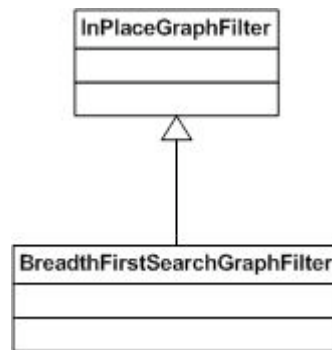


Figure 3.10 Breadth First Search Graph Filter Inheritance Diagram

### 3.5 Application of Functors

The Application of Functors, which adopts a visitor design pattern, helps the design of the graph algorithms to be more flexible. An application of this flexibility is shown by `itk::BaseBreadthFirstSearchFunctor`. This class has the basic methods for the distance calculation per vertex defined when the algorithm traverses the graph and the comparison function for the heap calculation of the weights of the vertices. The `DefaultBreadthFirstSearchFunctor` class defines the logic for those classes. The `BreadthFirstSearchFunctor` defines the interfaces that accept an outside function and class to define the logic for distance calculation and comparison for the heap function. It takes as a template value, a class that has the distance and comparison functions defined. Then

it accepts the definitions of the functions as pointers and those function definitions are used for the overall logic of the class.

## CHAPTER 4: EXPERIMENTATION AND APPLICATION

### 4.1 Experimentation.

In order to test the algorithms first an adjacency matrix was designed with graphs of 7 to 10 nodes each. That adjacency matrix was stored as an *itk::Image* class with the Pixel type as an integer and the dimension as 2. A modified *ImageToGraphFunctor* was used to determine what would be considered as a node and what was not considered as a node. Any pixel with the value of 0 was discarded. An integer value of a pixel was assigned as the Edge weight and the row and column was considered as the nodes. Using this method, a graph was made and the graph was assigned as the input for all the graph filters. The output was tested to see if it gave the correct output for the given start node. In order to test for robustness, the amount of nodes was increased and tested using the same method as it was used for smaller number of nodes.

The algorithms were then tested using images for robustness and accuracy. An *itk::ImageReader* class was used to read in the image and pass it to the *itk::ImageToGraphFilter* that also took in a modified *ImageToGraphFunctor* class that was application specific. All algorithms managed to process more than 100,000 nodes.

For each of the classes, the examples in [14] were used for the initial test. Later larger graphs were used and then images. These graphs were either taken from random examples in books or made by the author. Some adjustments such as memory management of Standard template library had been made, since *stl::map* objects occupy

too much memory and doesn't release them upon the end of its scope, unlike other container architecture. For large images and volumes, that could considerably slow down the speed. Initially, separate *stl::map* container classes were designated for the color and the distance for each node, but it took up too much space and also caused several segmentation faults. In order to avoid that, an enumeration type field called *ColorType* was declared as part of the node structure. Also, the distance field has been replaced for all the classes by a double field.

For the heap of *itk::PrimMinimumSpanningTreeGraphFilter*, the *stl::set* map class was used. So instead a vector class was used. This design was then used for the Dijkstra's shortest path and the Kruskal's minimum Spanning tree algorithm as well.

The breadth first search algorithm was tested with its functor class. First, it was tested with the default class *itk::DefaultBreadthFirstSearchFunctor*. Then with the sub class *itk::BreadthFirstSearchFunctor* was used to test the algorithm for the small examples and then for larger examples. To test its function pointers, a demo class was defined and the same test was repeated, to test for accuracy. The demo class contained the same functionalities as the *itk::BreadthFirstSearchFunctor* except it was passed on as a function pointer to the *itk::BreadthFirstSearchFunctor* class. And its functions were passed on as function pointers as well.

Some of the classes had their *stl::map* containers replaced by ITK's definition called *itk::MapContainer* for reusability of ITK code and as well as better integration of the code.

#### 4.2 Application of the Graph Classes: Centerline Extraction.

In order to show the application of an ITK graph class for practical use, the algorithm to calculate robust centerline extraction for skeletonization has been used. As it has been discussed before, the skeletonization of an object requires the calculation of the distance transform of an image and this can either be achieved by using the shortest path graph algorithm or the minimum spanning tree graph algorithm can achieve this. The graph contains all the object voxels as vertices or nodes of the graph, and the edges of the graph are considered as the connection between two adjacent pixels. The edge weights are inverse ratio of the distance metrics of each pixel. Alternately slight modification of the breadth first search algorithm may work for the calculation of the centerline as well.

As it has been explained in Chapter 1, as well as Chapter 2, distance transform is an image processing technique that calculates the relative distance of a voxel of an object from the boundary voxels of that object.

The extraction of centerlines of an object is obtained from the skeleton of an object. Extraction of centerlines is a useful application in medical image analysis and processing, especially from images of lungs, bronchia, blood vessels and colon. This technique relies on various features of the segmented image such as the accuracy by which the image was segmented which is dependent upon the noise level of the images. Distance field based method was used because of two strong points, one, where outside of the distance field calculation, centerline extraction algorithm is itself quite efficient, and, secondly, the centerline is guaranteed to be inside the structure. However, most distance field based algorithms are dependent upon segmented images as well which needs to be highly accurate, which can be challenging as most images are noisy and can also have

interfering organ objects. In order to overcome these shortcomings, a Gaussian type probability model is applied to compute the modified distance field, and after that standard distance field algorithms are then applied to extract the centerline.

As majority of the images in medical analysis where skeletonization is applicable, have a tubular structure such as blood vessels; their properties were taken into consideration while implementing the algorithm. The first property of these images is the use of the second order derivative that can be defined by a Hessian matrix. The second property of multi scale analysis, or scale space theory, relates scale to derivatives. In order to apply probabilistic methods of determining the centerline of a volume object, the following steps have been taken.

- Volume Preprocessing: The input volume is roughly segmented into object voxels and background voxels. Region growing or thresholding is used to isolate the object of interest. An *itk::Graph* object was created at this stage that held all the object voxels as nodes that met the threshold criteria. The threshold a criterion was implemented using a specialized *itk::DefaultImageToGraphFunctor* called *itk::SkeletonImageToGraphFunctor* and passed as a template parameter into the *ImageToGraphFilter* class. The *ImageToGraphFilter* class then outputted the *itk::Graph* object.
- Most medical images are band limited which is caused by the nature of their acquisition and reconstruction and hence it can be concluded that the medical structures do not have a step boundary but are blurred by a Gaussian factor. The goal of the application proposed by [10] was to define a probability function across the object boundary.

- The next step is to estimate a constant  $K$ , which would help in determining the probability function of the object voxels that are close to the object boundary. In order to determine the constant  $K$ , with each boundary voxel as a starting point, the tracking direction is determined which would lead to local maxima. The tracking direction is usually along the gradient direction. Increase in the gradient magnitude usually indicates that the gradient is towards the boundary and the decrease indicates that the gradient is moving away from the boundary. With the help of the local maxima that is determined by the gradient magnitude and direction, the constant  $K$  is calculated.
- Once the constant  $K$  is calculated, the probability values are assigned to the voxels that are close to the boundary. A starting point is determined to calculate the probabilities of the boundary voxels using the gradient direction and magnitude. Decisions regarding the value of the voxels are determined by the two approximate and tentative values of  $K$ , which is,  $K_1$  and  $K_2$ .
- The next step, all the other non-boundary voxels are assigned probabilities for the remaining voxels. As the pre-segmentation roughly classifies all voxels as either background or object voxels, these initial values are used as a starting probability and local neighborhood, usually 26 connected, operations are performed to get more accurate values. The probabilities are calculated against a background threshold value and an object threshold value by using the average probability value. This part of the code of assigning voxels their probability value is handled by the *ImageToGraphFilter* and the *SkeletonImageToGraphFunctor* which



assigns the probability value to the voxel as the weight of the node which is assigned to the voxel.

- The distance fields of all the voxels are calculated. The boundary voxels have non-zero values as their distance from the boundary value unlike former algorithms that had applied similar techniques. For a node A and B, the distances are calculated as:

$$D_A = D_B + P_A D(B, A)$$

$$D_B = D_A + P_B D(A, B)$$

Where,  $D(B, A)$  is the distance that was scaled between the points B and A to the probability of the point being on the boundary. This value is added by using the *SkeletonImageToGraphFunctor* and the *SetEdgeWeight()* function which is designed to add the edge weight according to the values of  $P_A$  and  $P_B$ . The inverse of the values of these fields are assigned to the voxel edge weights.

- Exact voxel distances are used to assign the distance value to each node. Once the distance transform has been calculated for each object voxel, the pixel with the largest distance from boundary is calculated for the initial start node or root node of the minimum spanning tree that would traverse through the object nodes. The seed or the root node id is calculated. This id is of the *NodeIdentifierType* of the *itk::Graph* class. The largest geodesic distance from the start node is also noted which is used to lead towards the root point, via the parent nodes of the Minimum Spanning Tree container. This was implemented using the *itkBreadthFirstSearchGraphFilter* class, which used a heap for the queue to retrieve the nodes. These nodes are sorted by distance by default. But for the

extraction of the centerline, it was sorted by their node weights, which was equal to the inverse of the pixels distance from the boundary value. This was set by the *SetNodeWeight ()* function of *itkSkeletonImageToGraphFunctor* class. The breadth first search distance was calculated as it traversed the nodes of the volume. For the specialized calculation of the distance while traversing the volume, the *itkBreadthFirstSearchFunctor* defined the methods for comparing the weights between two nodes and calculating the distance with the help of a third class.

The results of the code were tested against the original application for accuracy using various synthetic and medical datasets. The following images are a demonstration of the application of the *itkBreadthFirstSearchGraphFilter* class. Figure 4.1 is a cylinder that has ending and starting points which are predetermined. Figure 4.2 is an image of a cylinder that has had its tip detection done dynamically. Figure 4.3 is an image of an MRT scan with segmentation threshold value of 80.

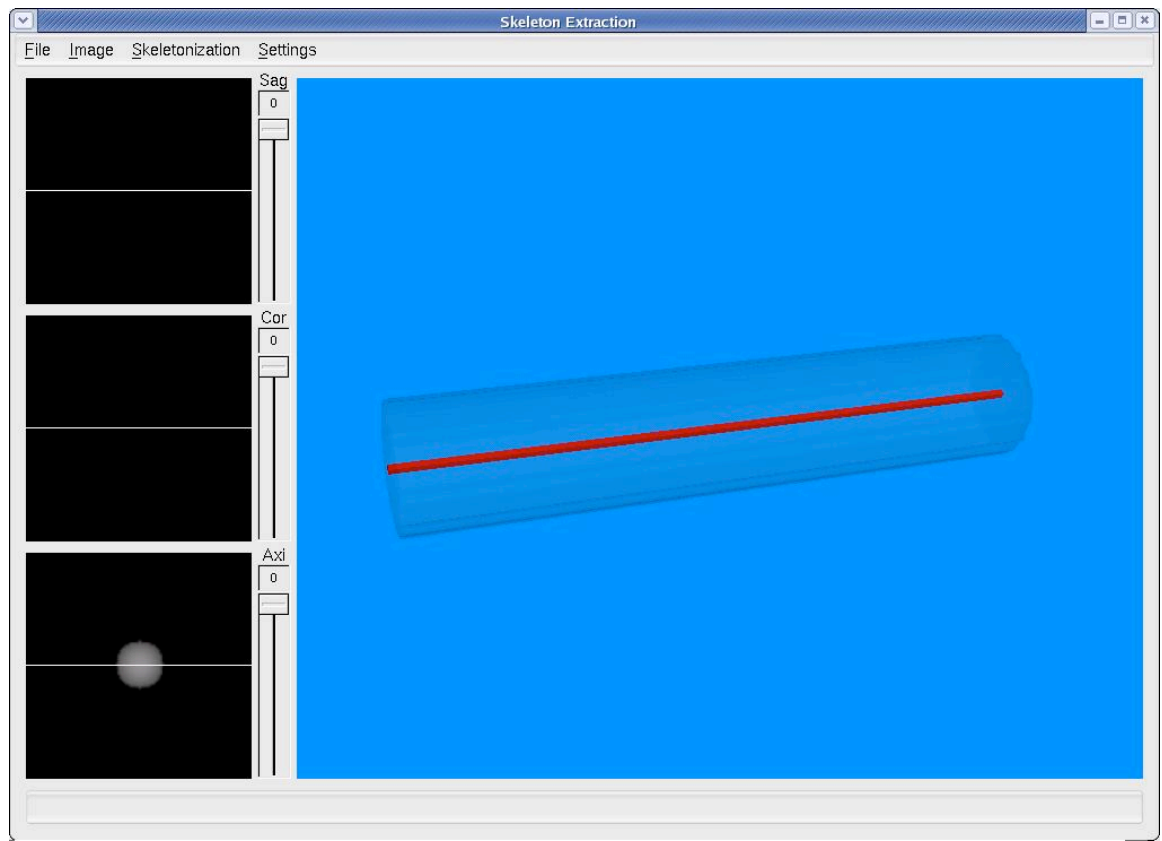


Figure 4.1 Skeleton of a Cylinder whose end points were predetermined.

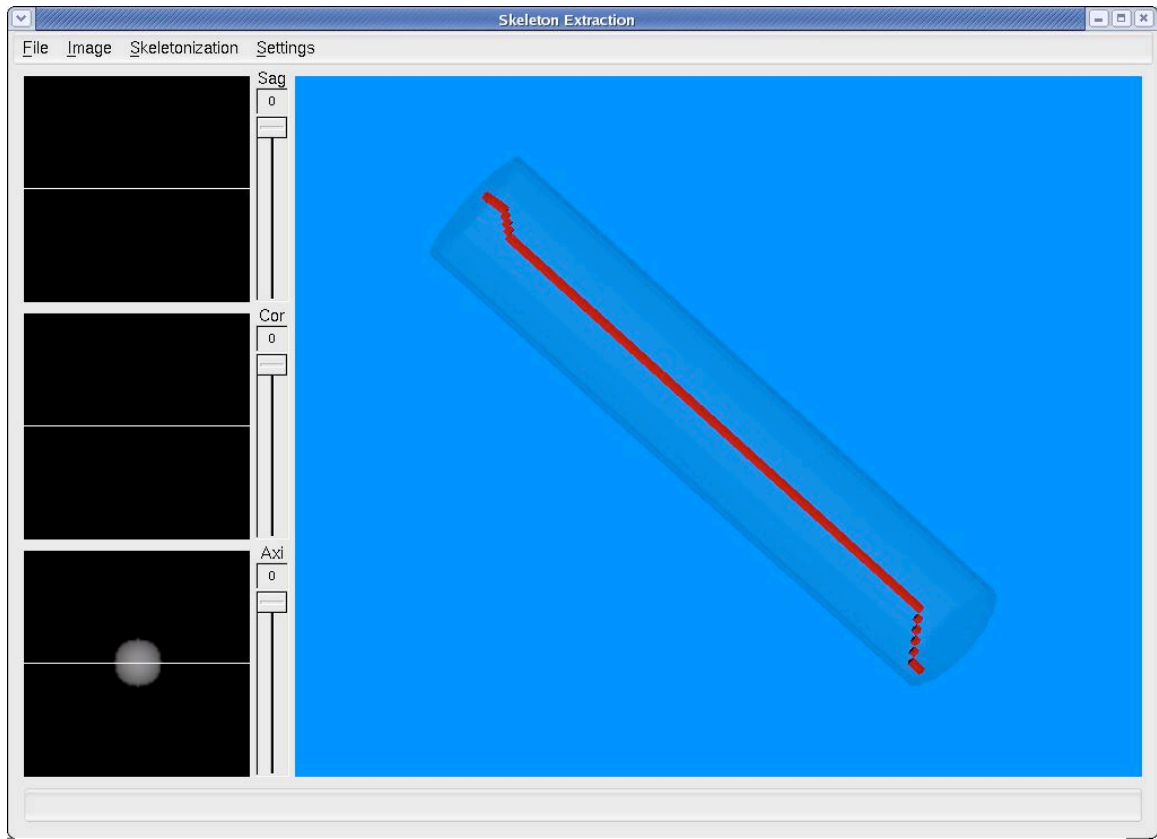


Figure 4.2 Skeleton of a Cylinder whose end points were calculated dynamically.

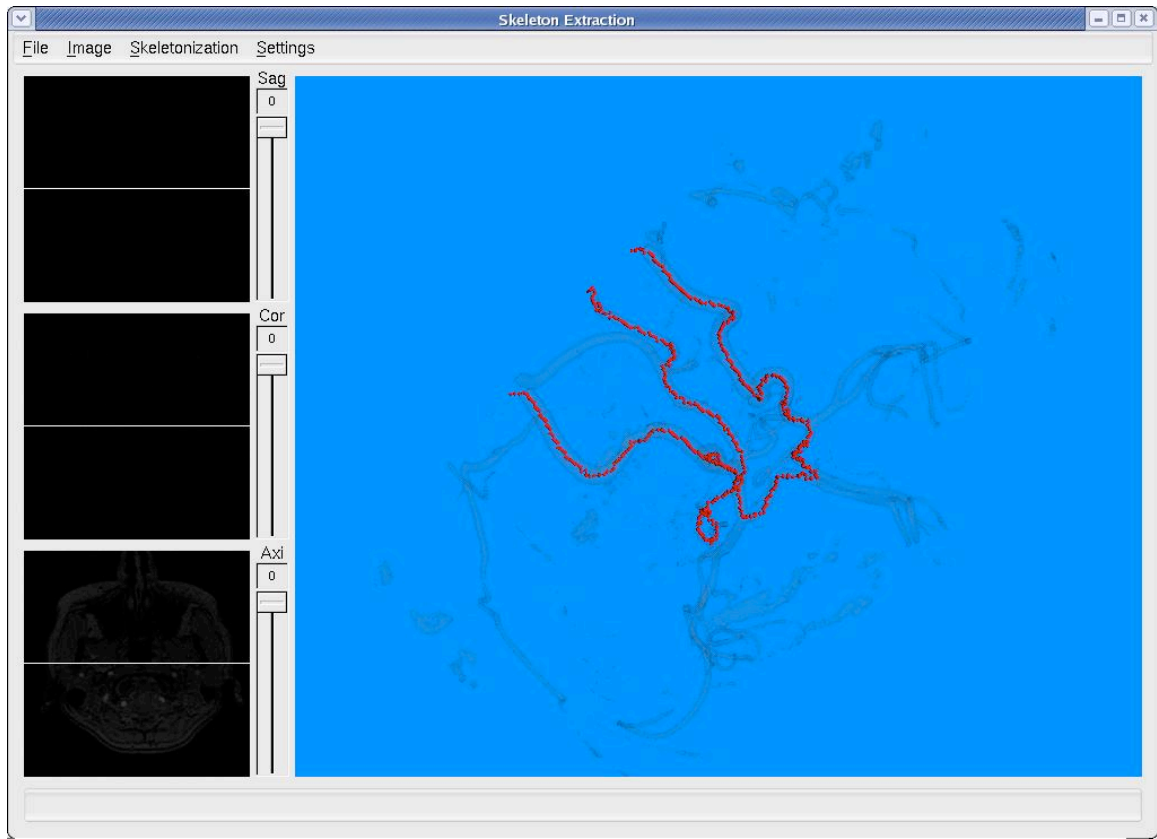


Figure 4.3 Skeleton of an MRT data.

## CHAPTER 5: CONCLUSIONS

As, it has been shown the application of Graph Theory and its algorithms in Image Processing and especially in the area of Medical Image Analysis makes the development of a Graph Theory library for the ITK library a necessary and useful addition for accurate and effective processing and analysis of images. The example of the skeleton extraction has shown that it can be applied for major medical analysis applications. And it can also be used for general calculations. The implementation has been made flexible in order to allow it to be applied to varying problems.

Further definitions of functor classes for Prim's minimum spanning tree, depth first search, Dijkstra's shortest path algorithm and Kruskal's minimum spanning tree will be designed as future work. Also, designs for the graph traits classes will be made more generic and user defined. This way the application of all the graph classes will be truly generic and graph theory can be applied easily for image analysis.

## REFERENCES

- [1] *Insight Segmentation and Registration (ITK) Software Guide*. Luis Ibanez, Will Schroeder, Lydia Ng, Josh Cates and the *Insight Software Consortium*.
- [2] *Graph Cuts and Energy Minimization*. Nicholas J. Tustison, Paul A. Yushkevich, Zhuang Song, and James C. Gee.
- [3] *Segmentation on Edge Preserving Smoothing Image based on Graph Theory*. S. Chitwong, F. Cheevasuvit, K. Dejhan, S.Mitatha, C. Nokyoo, and T. Paungma, Faculty of Engineering, King Mongkut's Institute of Technology Lakrabang, Bangkok 10520, Thailand.
- [4] *Fast Euclidean Distance Transform using a Graph-Search Algorithm*. Roberto A. Lotufo, Alexandre A. Falcao, Francisco A. Zampiroli. Faculdade de Engenharia Eletrica de Computacao, Brasil.
- [5] *Image Registration with Minimum Spanning Tree Algorithm*. Bing Ma, Alfred Hero, Department of EECS, University of Michigan, Ann Arbor, MI 49109, John Gorman, ERIM International Ann Arbor , Olivier Michel, ENS-Lyon, France.
- [6] *Breadth First Search and its Application to Image Processing Problems*. Jaime Silvela, Javier Portillo. IEEE Transactions on Image Processing, Vol. 10, No. 8, August 2001.
- [7] *Digital Image processing*. Rafael C. Gonzalez, Richard E. Woods, Addison Wesley, 1992.
- [8] *Deformable models in Medical Imaging*. Ajit Singh, Dmitry Goldgof, Demetri Terzopoulos, IEEE Computer Society.
- [9] *Medical Image Analysis and Simulation*. Nicholas Ayache, INRIA-EPIDAURE Project. Sophia-Antipolis, France.
- [10] *Robust Centerline Extraction from Tubular Structures in Medical Images*. Jianfei Liu, Kalpathi Subramanian. University of North Carolina, Charlotte, USA.
- [11] *Flux Driven Automatic Centerline Extraction*. Sylvain Bouix, Kaleem Siddiqi, and Allen Tannenbaum. Harvard Medical School, Georgia Institute of Technology, USA and McGill University, Canada.
- [12] *Automatic Centerline Extraction for virtual Colonoscopy*. Ming Wan, Zhengrong Liang, Qi Ke, Licjan Hong, Ingmar Bitter and Arie Kaufman. IEEE Transaction on medical Imaging, Vol. 21, No, 12, December 2002.
- [13] [www.itk.org](http://www.itk.org).

[14] *Introduction to Algorithms*. Thomas Cormen, Charles Leiserson, Ronald Rivest and Clifford Stein. Published by McGraw-Hill.

[15] *Introduction to the special section on graph algorithms in computer vision*. S. Dickinson, M. Pelillo, and R. Zabih. IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 23, no. 10, pp. 1049-1052, October 2001.

[16] *Curve-Skeleton Applications*. N. Cornea, D. Silver and P. Min. Proceedings of IEEE Visualization 2005, pp. 95-102, 2005.

[17] Boost Graph Library. [www.boost.org](http://www.boost.org).

[18] Analyzer Image Processing Tool kit. [www.analyzer.org](http://www.analyzer.org).



## APPENDIX A

## A.1. Insight Segmentation and Registration Tool Kit: Introduction.

Insight Segmentation and Registration Tool Kit (ITK), as explained in [1], is an open source, object oriented software system for image processing, segmentation and registration. The Insight Segmentation and Registration Toolkit were funded by the United States National Library of Medicine of National Institute of Health in 1999. In 2002, the first public release of ITK was made available [14 and 1].

ITK is made of several subsystems. Some design concepts that it utilizes in order to implement a simple and flexible system are generic programming, smart pointers for memory management and object factories for adaptable object instantiation, event management using the command/observer design paradigm, and multithreading support. For Numeric purposes, ITK uses VXL's VNL library which are an easy to use C++ wrappers around *Netlib Fortran* numerical analysis routines. For data representation, ITK has several classes defined. Two of the principal ones are `itk::Image` and `itk::Mesh`. Classes called Iterators and Containers are used to traverse and hold the data. ITK also has classes called Filters that are organized into Dataflow pipelines. These pipelines maintain state and therefore execute only when necessary. Pipeline also supports multithreading and is capable of streaming. The Data Processing Pipeline consists of sources, filters that initiate the pipeline and mappers, filters that terminate the pipeline. The standard example for source and mapper would be readers and writers. A reader class takes in input data usually from a file, and writers output data from the pipeline onto either the screen or another image class or file. In ITK, geometric objects are represented

using the spatial objects hierarchy. These classes usually are used for modeling anatomical structures. They use a common basic interface for representing regions of space in a variety of different ways. For example, mesh structures may be used as the underlying representation scheme. Spatial objects are a natural data structure for communicating the results of segmentation methods and for introducing anatomical priors in both segmentation and registration methods, as it has been described in [1].

ITK supports a framework for four types of Registrations: Image, Multi-resolution, PDE-Based and Finite Element method (FEM) registration. It also supports a framework for general FEM problems in particular non-rigid registration. This package also includes mesh definition (such as nodes and elements), loads and boundary conditions. Level set framework is another set of classes used for creating filters to solve partial differentiation equations on images using iterative, finite difference update scheme. The framework consists of finite difference solvers including a sparse level set solver and several specific subclasses including threshold Canny and Laplace's algorithm based Methods.

ITK used a unique and powerful system for producing interfaces (i.e. "wrappers") to interpreted languages such as TCL and Python. It also includes tools such as GCC\_XML to produce an XML description of arbitrarily complex C++ code. ITK also has several auxiliary subsystems such as calculators which are classes that perform specialized operations in support filters, partial DICOM parser and interfaces to the Visualization Toolkit (VTK) system [1].

## A.2. Generic Programming.

Generic programming is a method of organizing libraries consisting of generic or reusable software components. Generic Programming usually consist of containers that hold data, iterators to access the data and generic algorithms that use containers and iterators to create efficient fundamental algorithms such as sorting. It is implemented in C++ using template programming mechanism and the use of the STL (Standard template Library). C++ template programming is a programming technique which allows users to write software in terms of one or more unknown types T. A user of the software defines the type T in order to create executable code. The T may be a native type such as *float* or *int* or T may be a user defined type (e.g. class). At compile time the compiler makes sure that the templated types are compatible with the instantiated code and that the types are supported by the necessary methods and operators.

The advantage of using generic programming for ITK is that an almost unlimited variety of data types are supported simply by defining the appropriate template types. For example, in ITK it is possible to create images consisting of almost any type of pixel. The type resolution is done during compile time, so the compiler can optimize the code to deliver maximal performance. The disadvantage of generic programming is that many compilers usually do not support this high level of abstraction and hence cannot compile ITK. Although some compilers may support generic programming, they may produce undecipherable code even for some of the simplest of errors.

### A.3. Include files and Class definitions

Each class in ITK is defined by two files: a header file ending with *.h* and an implementation file- *.cxx* if a non-templated class, and a *.txx* if a templated class. The header files contain class declarations.

### A.4. Object Factories

In ITK, most of the classes are instantiated through an object factory mechanism. Instances of ITK classes are created using the static class *New ()* method. The class's constructor and destructor are protected and hence are not generally possible to construct ITK classes on the heap.

The object factory allows run time instantiation of classes by registering one or more factories with *itk::ObjectfactoryBase*. These registered factories support the method *CreateInstance* (class name) which takes as input the name of a class to create. The factory can choose to create the class based on a number of factors including the computer system configuration and environment variables. By using the object factory, it is possible to replace a particular ITK Filter class with a custom class. Usually the ITK object factories are used by the ITK input/output (IO) classes.

### A.5. Smart pointers.

ITK implements memory management via reference counting. A count of the number of references to each instance is kept. When the reference count is zero, the

object destroys itself. Reference counting deletes memory immediately, unlike garbage collection which does it at random times.

*Reference counting* is implemented through *Register ()/Delete ()* member function interface. All instances of an ITK object have a *Register ()* method invoked in them by any other object that references them. The *Register ()* function increments the object instance's reference count. When the reference to the object is removed a *Delete ()* method is invoked on the instance that decrements the reference count. When the reference count reaches the count of zero, the object instance is destroyed. This whole protocol is implemented in a helper class called *itk::SmartPointer*. Although it acts like a normal pointer it also updates the reference count by performing *Register ()* when referring to an instance. When an instance is de-referenced, *UnRegister ()* is called. The Smart Pointer can be allocated on the program stack and is automatically deleted when the scope of the method in which the Smart Pointer was declared in, is closed.

#### A.6. Error handling and Exceptions

ITK uses exception handling to manage errors during program execution. Exception handling is a standard part of the C++ language and generally takes the form of a try and catch statement where the catch statement has an argument of type Exception. In ITK the exception class used to catch errors is *itk::ExceptionObject*. *Itk::ExceptionObject* may be sub-classed to customize exception handling.

### A.7. Event handling

ITK also supports Event Handling using the Subject/Observer design pattern where the objects indicate that they are watching for a particular event- invoked by a particular instance that they are watching. Objects that are registered with the event are notified when the event occurs. The notification is usually a command such as a callback function or method invocation that is specified during the registration process. All registered observers are notified by the invocation of the *Command::Execute ()* method.

### A.8. Multi Threading

ITK supports a multithreading by implementing a high level design abstraction. This design provides portable multithreading and hides the complexity of differing thread implementations on the many systems supported by ITK. The Class that implements Multithreading in ITK is called *itk::MultiThreader*. Multithreading is used by an algorithm during its execution phase. A Multi-thread class can be used to execute a single method on multiple threads, or to specify a method per thread.

### A.9. Data Representation.

There are two principal types of data represented in ITK. They are images and meshes. In ITK, they are represented by the classes, *itk::Image* and *itk::Mesh*. Data

classes in ITK are all sub classed from the object class called *itk::DataObject*. These data objects can be passed around in a system and can participate in a data flow pipeline.

#### A.9.1. Image representation in Insight Segmentation and Registration Tool Kit

*Itk::Image* represents an n dimensional, regular sampling of data. The sampling direction is parallel to each of the coordinate axes, and the origin of the sampling, inter-pixel spacing, and the number of samples in each direction (i.e., image dimension) can be specified.

A sample or pixel can be arbitrary; a template parameter *TPixel* specifies the type upon template instantiation. Along with the pixel type, the dimensionality of the image must also be specified during the instantiation of the image class. One factor of the pixel type that must be taken into consideration is that it must support certain operations such as addition and subtraction. Such a factor is necessary for the code to run in all perceivable cases. C++ simple types such as integer and float or pre-defined pixel types are usually used.

Another concept of data representation that is important amongst ITK concepts is Region. Region is rectangular and continuous pieces of an image. They are used to specify which part of an image to process or which part to hold in memory [1].

There are three types of regions:

- *LargestPossibleRegion* – the image in its entirety.

- *BufferedRegion* – the part of the image that is stored in the memory.
- *RequestedRegion* – the part of the region requested by a filter or other class when operating on the image.

#### A.8.2. Mesh representation in Insight Segmentation and Registration Tool Kit.

*Itk::Mesh* class represents an n- dimensional, unstructured grid. The topology of the mesh is represented by a set of cells defined by a type and connectivity list. The connectivity list references points. These points constitute the n-dimensional geometry of a mesh in combination with associated cell interpolation functions.

The mesh is defined in terms of three template parameters:

- A pixel type associated with the points, cells, and cell boundaries;
- The dimension of the points (which in turn limits the maximum dimension of the cells);
- A “mesh traits” template parameter that specifies the types of the containers and identifiers used to access the points, cells and boundaries.

Mesh is a sub class of *itk::PointSet*. The *PointSet* class allows a user to represent data either as point clouds or randomly distributed landmarks, etc. The *PointSet* class has no associated topology [1].



## A.10. Data Pipeline.

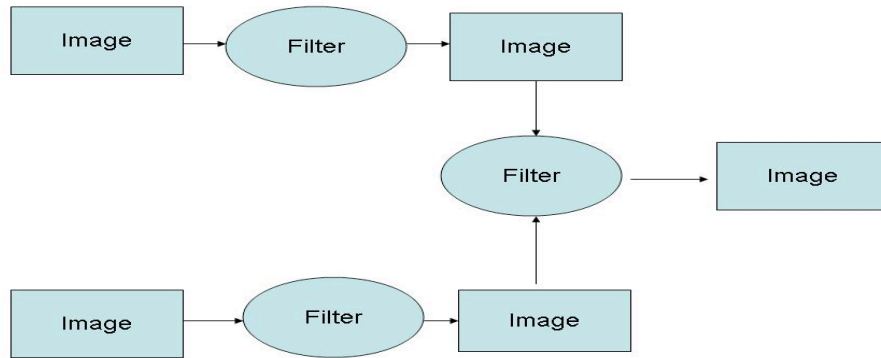


Figure A.1. ITK Image Filter Data Pipeline

Data Processing Pipeline consists of data object and Process object interaction. Data Objects are usually Image and Mesh. Process Objects consist of sources, filter objects or mappers. Process Objects operate on data objects and may produce new data objects. Sources (such as readers) produce data, filter objects take in data and process it to produce new data, and mappers accept data for output either to a file or some other system[1 and 13].

## A.11. Basic software design for *ProcessObject* in ITK.

Some of the key design aspects of an ITK filter are, as it has been described in Chapter 2 of [1]:

- They all inherit the properties of *itk::ProcessObject* which in turn is inherited from *itk::Object*.

- A base class which inherits *itk::ProcessObject* called *itk::ImageToImageFilter* is used for most advanced filters. The *itk::ImageToImageFilter* takes in as a template an Image class description that describes the input and another one that describes the output.
- The base class *ProcessObject* describes the methods *SetInput ()*, *GetInput ()*, *SetOutput ()* and *GetOutput ()*. These methods usually take in *DataObject* sub classes such as *itk::Image* and *itk::Mesh*. The sub classes of *ProcessObject* describe the behavior of the filter by defining the virtual classes declared by the base class.
- All filters are part of the namespace *itk*.
- All private variable members are accessed using macros described by ITK. These macros are called *itkSetMacro ()* and *itkGetMacro ()*. The first argument of these macros accepts the variable name while the second argument accepts the type of the variable. The macros however fail to work for any data type that is of a templated array type.
- The **typedef** definitions of ITK for base class and the classes own data type definition are usually declared public in order to allow users of the class to definition and describe their own data types for their classes.
- One of the pre defined methods of *ProcessObject* is called *GenerateData()* which is declared protected in a filter class. The filter algorithm is defined in this method.

- Update function defined in `ProcessObject` is the only way a user can run the algorithm contained in the `GenerateData ()` method of a class.
- ITK restricts the number of algorithms that can be supported by a single filter to one.
- A composite class can be built provided it is a pipeline and not three distinct algorithms.

There are several types of macros and traits that need to be part of the definition of an ITK filter.

- *ItkNewMacro*: This macro defines the `New ()` function of an ITK filter. This function is needed in order to instantiate an instance of the filter class in ITK.
- *ItkSetMacro* and *itkGetMacro* are the macros that define the storage and retrieval of private data types of the filter class.
- *itkTypeMacro*: Notifies the compiler about the association of the base class and the sub class.
- *SmartPointer*: A pointer description of the filter class is made which allows a user of the filter to declare a pointer variable of the filter class. For example,

```
typedef SmartPointer<Self> Pointer;  
typedef SmartPointer<const Self> ConstPointer;
```

Here, *Self* is a typedef of the filter class's name.

- *PrintSelf ()* is a method declared within a filter class that may be defined in order to know the state of the variables defined by the class.

#### A.12. Template programming in C++.

Some of the concepts that involve template programming in C++ are as follows:

- Every class starts with the template word. For example, `template<class TPixel, class TSize>`.
- **typedef** is an important keyword in template programming. It allows users who define sub classes to maintain constancy between base class and sub classes.
- Template class inheritance is different from usual inheritance. *Export* is used before template to let the subclasses use the inline functions of the base class. ITK has its own `ITK_EXPORT` that has the same functionality as *export* in ANSI C++.
- Subclass template parameters must be used to define the base class unless the base class has a default type defined for any of its parameters. All the base class parameters must be mentioned while defining the sub class.

**typename** is another keyword that is a part of generic programming. *typename* is used to enforce a type defined by the class or its sub class on a variable or for defining a

user-defined type. It indicates to the compiler that a user is aware of a variable's type but is not sure what kind and where it is defined exactly

### A.13. Design of ITK Image inheritance and Data Processing Pipeline.

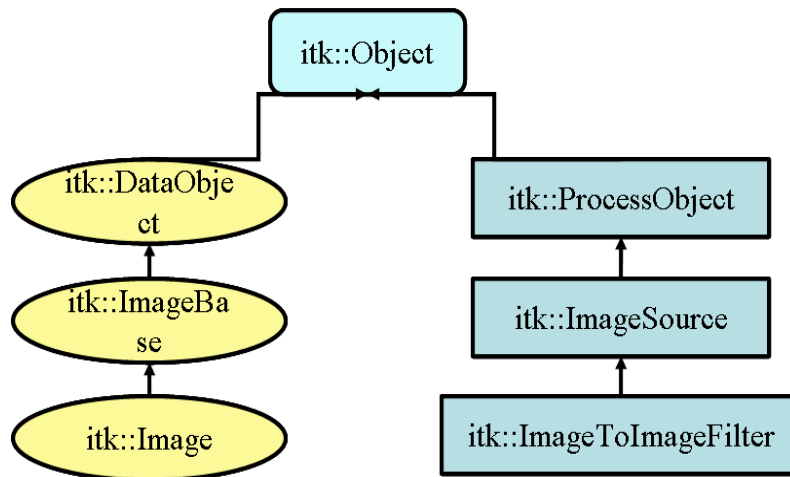


Figure A.2. Image DataObject and ProcessObject inheritance diagram(taken from the ITK Getting started series)

The *itk::Image* and *itk::ImageToImageFilter* which are two important class definition of the ITK data processing pipeline constitutes the following inheritance specification. Both *DataObject* and *ProcessObject* are sub-classed from *itk::Object* class. The *itk::DataObject* class is in turn sub classed to *itk::ImageBase* which is sub classed by *itk::Image*. *Itk::ProcessObject* in turn is inherited by *itk::ImageSource* which is inherited by *itk::ImageToImageFilter* [13].

*itk::DataObject* is the base class that is inherited by all classes that are used to represent and provide access to data. ITK has its classes, *itk::Image* and *itk::Mesh*, that

are inherited from this base class. *Itk::DataObject* is an inherited class of *itk::Object* which is the base class for all factory object classes in ITK. It defines the input and output of a *DataObject* class, and also the regions within the data object [1 and 13].

*ImageBase* is the base class for the templated Image classes. *ImageBase* is templated over the dimension of the image. It provides the API and the variables that depend solely on the dimension of the image. *ImageBase* does not store the image or pixel data. That is done by its sub-classes such as *itk::Image* [1 and 13].

*Itk::Image* is a templated n dimensional class. Images are templated over a pixel type and a dimension. The container that stores the pixel data is *ImportImageContainer*. Within the Container, images are modeled as arrays, defined by a start index and a size. Pixels can be accessed using *SetPixel ()* and *GetPixel ()* methods or can be accessed via iterators. *Begin ()* creates an *Iterator* that can walk a specified region of a buffer. The data in an image is arranged in a 1D array as slice then row then column with the column index varying the most rapidly. The Index type reverses the order so that with Index [0] = column, Index [1] = row, Index [2] = slice.

*Itk::ProcessObject* is the base class for all the filters. It defines the filter input and output interfaces and the interface for *GenerateData* and *UpdateData*.

*Itk::ImageSource* is the base class for all process objects that output image data. Specifically, this class defines the *GetOutput ()* method that returns a pointer to the output image. The class also defines some internal private data members that are used to manage streaming of data. Memory management in *itk::ImageSource* is slightly different than that of *itk::ProcessObject*. While in *ProcessObject* the bulk data associated with the output is always released with their output prior to the *GenerateData()* being called, in

*itk::ImageSource*, the default is not to release the bulk data in case that particular memory block is large enough to hold the new output values. This avoids unnecessary de-allocation sequences. *ImageSource* can be made to have a similar memory management behavior such as that of *ProcessObject* by calling the *ProcessObject::ReleaseDataBeforeUpdateFlagOn ()* [1].

*Itk::ImageToImageFilter* is the base class for all ITK filters that take in an image type as inputs and outputs. This class specifically defines *SetInput ()*. It also provides infrastructure for supporting multithreaded processing of images. If a filter provides implementation of *GenerateData ()*, a single threaded implementation of the algorithm is handled. If on the other hand, *ThreadedGenerateData ()* is called, then the image is divided into number of pieces and a number of threads are spawned and each thread will handle *ThreadedGenerateData ()* separately. This class also provides the implementation of *GenerateInputRequestedRegion ()*. The base assumption at this point in the hierarchy is that a process object would ask for the largest possible region on input in order to produce any output.



Figure A.3. Relationship between ProcessObject and DataObject

The typical ITK filter pipeline execution performs the following important functions along with other functions:

- It determines which filters in the pipeline need to be executed.
- Initializes the filters output data objects and prepares them for new data. It also determines how much memory each filter must allocate for its output and allocates that memory for the filter.
- The execution process determines how much data a filter must process in order to produce an output of sufficient size for downstream filters; it also takes into account any limits on memory or special filter requirements. Other factors include the size of the data processing kernels, which affect how much data input data (extra padding) is required.
- It subdivides data into sub-pieces for multithreading.
- It may free or release output data if filters no longer need it to compute, and the user requests that data be released.

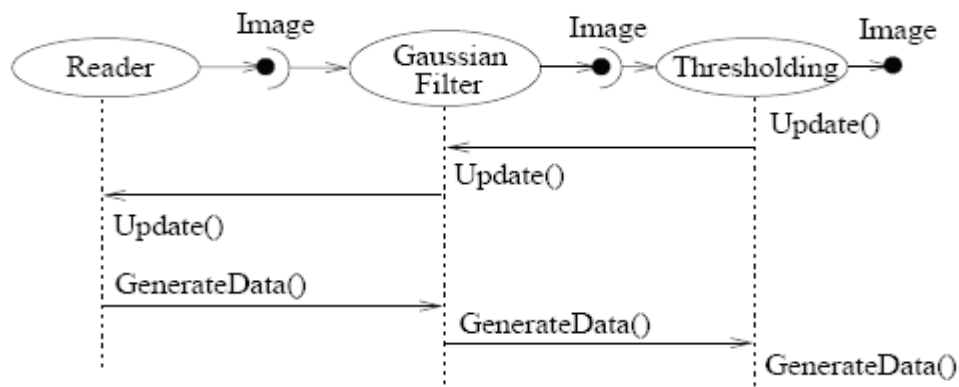


Figure A.4. Data Execution Pipeline

A pipeline execution is initiated when a process object receives the `ProcessObject::Update ()` method invocation. This method is simply delegated to the



output of the filter, invoking the *DataObject::Update ()* method. This behavior is typical between *ProcessObject* and *DataObject*. In this way data request from the pipeline is propagated upstream, initiating data flow that returns downstream. The *DataObject::Update ()* method in turn invokes three other methods:

- *DataObject::UpdateOutputInformation()*
- *DataObject::PropagateRequestedRegion()*
- *DataObject::UpdateOutputData ()*

The *DataObject::UpdateOutputInformation ()* method determines the pipeline modified time. It may set the *RequestedRegion* and the *LargestPossibleRegion* depending on how the filters are configured. The *UpdateOutputInformation ()* propagates upstream through the entire pipeline and terminates at the sources.

The *DataObject::PropagateRequestedRegion ()* call propagates upwards to satisfy a data request. In a typical application this request is usually the *LargestPossibleRegion* of the output. The function of this method is, given a request for data; propagate upstream configuring the filter's input and output process object's to the correct size. Eventually, this means configuring the Buffered Region, that is the amount of data actually allocated.

*DataObject::UpdateOutputData ()* is the final method as a result of the Update method call. The purpose of this method is to determine whether a particular filter needs to execute in order to bring its output up to date (A filter executes when it's *GenerateData ()* method is invoked.) Filter execution occurs when a) the filter is modified as a result of modifying an instance variable; b) the input to the filter changes; c) the input data has been released; or d) an invalid *RequestedRegion* was set previously and the filter did not produce data, Filters execute in order in the downstream direction. Once a filter executes, all filters

downstream of it must also execute. This method is delegated to the *DataObject*'s source only if the *DataObject* needs to be updated.