ABSTRACT

DYER, TRISTAN. An Interface for Subdomain Modeling using a Novel Range Search Algorithm for Extracting Arbitrary Shapes. (Under the direction of John Baugh.)

ADCIRC is a finite element program that is widely used by modelers to simulate storm surge in order to predict the effects of hurricanes on coastal areas. Subdomain modeling in ADCIRC is a technique used to reduce the total runtime and storage requirement of a series of hurricane storm surge simulations performed for a particular storm on multiple, alternative topographies in a geographic region of interest. This thesis presents a graphical user interface for the subdomain modeling approach, called the ADCIRC Subdomain Modeling Tool (SMT), that is meant to streamline pre- and post-processing requirements by allowing modelers to extract subdomains from ADCIRC meshes interactively and to organize subdomain runs on the file system. Several shape selection tools are included in SMT that allow modelers to select areas of interest of an ADCIRC mesh by drawing the shape on top of a plan view of the mesh. Because ADCIRC meshes can often be extremely large, a novel search algorithm, called the generalized range search algorithm, is used to provide a sufficient level of responsiveness from the tool when a modeler is creating subdomains. The algorithm, which operates on a quadtree data structure that contains an ADCIRC mesh, works by avoiding the need to search through regions that are *guaranteed* to fall either inside or outside of the shape. It is possible to apply this algorithm to essentially any 2-dimensional geometric shape, with the only requirements being that there exists a test that determines if a point falls inside of the shape and there exists a test that determines if a line segment intersects with the shape. This flexibility makes the algorithm highly versatile and easily extensible. The upper and lower bounds on performance of the algorithm are derived, and test cases show that the actual performance mirrors the expected performance. Two case studies demonstrating the proper use of the core features of SMT are presented.

An Interface for Subdomain Modeling using a Novel Range Search Algorithm for Extracting Arbitrary Shapes

by
Tristan Dyer

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Civil Engineering

Raleigh, North Carolina

2014

APPROVED BY:

Downey Brill Jr.	Kumar Mahinthakumar

John Baugh
Chair of Advisory Committee

DEDICATION

To my family.

BIOGRAPHY

Tristan Dyer was born in 1988 in Fayetteville, North Carolina. He completed high school in 2007 and graduated from North Carolina State University in 2011 with a degree in civil engineering before beginning the Master of Science program in 2012. He currently resides in Raleigh, NC, and spends his free time in Ironman training.

ACKNOWLEDGMENTS

I would like to thank Dr. John Baugh for his support and guidance throughout my study at NCSU. I would also like to thank Dr. Brill and Dr. Kumar for their support and for taking the time to serve on my committee.

Additionally, I would like to thank Rick Luettich, Brian Blanton, and Casey Dietrich for providing the base files used in this study, including the NC FEMA meshes and wind files, as well as Alper Altuntas and Lake Trask for their help in developing and testing SMT.

This material is based upon work supported by the Coastal Hazards Center of Excellence, a US Department of Homeland Security Science and Technology Center of Excellence under Award Number 2008-ST-061-ND 0001.

The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the US Department of Homeland Security.

TABLE OF CONTENTS

LIST OF TABLES . . . . . **vii**

LIST OF FIGURES . . . . . **viii**

LIST OF ALGORITHMS . . . . . **x**

Chapter 1 Introduction **1**
1.1 ADCIRC 1
1.2 Subdomain Modeling in ADCIRC 2
1.3 Introduction to SMT 3

Chapter 2 Design and Development of SMT **4**
2.1 Design Goals 4
2.2 Identification of Required Functionality 5
2.3 Characterization of Core Features 6
2.3.1 GUI 7
2.3.2 Projects 7
2.3.3 Backend 7
2.4 Development of SMT 8
2.4.1 GUI Development 8
2.4.2 Developing ADCIRC Mesh Interactivity 9

Chapter 3 Search Algorithm **10**
3.1 Introduction to the Algorithm 10
3.2 Data Structure 11
3.2.1 Quadtree Basics 11
3.2.2 Storing a Finite Element Mesh in a Quadtree 12
3.3 Search Algorithm 16
3.3.1 Theory 17
3.3.2 Shape 17
3.3.3 The Algorithm 18
3.3.4 Explanation 20
3.4 Expected Performance 26
3.4.1 Theory 26
3.4.2 Verification 36
3.4.3 Effect of Non-Full Leaves 39
3.5 Implementation 41
3.5.1 The `QuadtreeSearch` Class 43
3.5.2 The `CircleSearch` Class 43
3.5.3 The `RectangleSearch` Class 44
3.5.4 The `PolygonSearch` Class 45
3.5.5 The `PointSearch` Class 47

3.6 Performance Results . . . 48
3.6.1 Comparison with Exhaustive Search . . . 48
3.6.2 Analysis of Comparison with Exhaustive Search . . . 54
3.6.3 Effect of Bin Size on Performance . . . 55
3.7 Optimization . . . 59

Chapter 4 Case Studies . . . 64
4.1 Quarter Annular . . . 64
4.2 Coastal North Carolina . . . 74

Chapter 5 Conclusions . . . 82

References . . . 84

Appendices . . . 87
Appendix A Data Type Definitions . . . 88
Appendix B `QuadtreeSearch` Class Code . . . 91
Appendix C `CircleSearch` Class Code . . . 99
Appendix D `RectangleSearch` Class Code . . . 103
Appendix E `PolygonSearch` Class Code . . . 107
Appendix F `PointSearch` Class Code . . . 111
Appendix G `ConvexCircleSearch` Class Code . . . 116

LIST OF TABLES

Table 3.1 Effects of Conditions Met During the Search Algorithm 25
Table 3.2 Assembly Time For Various Bin Sizes 58

LIST OF FIGURES

Figure 1.1 A Portion of an Unstructured Mesh 2

Figure 3.1 Drawing and Selecting From a Circle 10
Figure 3.2 A Simple Quadtree with Two Branches 11
Figure 3.3 The Empty Quadtree with the Root Branch Highlighted 13
Figure 3.4 The Empty Quadtree with the Leaves Highlighted 13
Figure 3.5 Visualization of the Quarter-Annular Case 15
Figure 3.6 Visualization of Quadtree Containing the Quarter-Annular Case 15
Figure 3.7 The Branch or Leaf Contains the Entire Shape 21
Figure 3.8 The Shape Contains the Entire Branch or Leaf 21
Figure 3.9 The Branch or Leaf is Fully Outside of the Shape 22
Figure 3.10 A Simple Filled Quadtree 26
Figure 3.11 6 Edges Followed, 2 Leaves Found 28
Figure 3.12 3 Edges Followed, 2 Leaves Found 29
Figure 3.13 1 Edge Followed, 1 Leaf Found 30
Figure 3.14 6 Edges Followed, 1 Leaf Found 30
Figure 3.15 Plot for a Solution Size of 1 31
Figure 3.16 Performance Plot for All Solution Sizes 32
Figure 3.17 Upper and Lower Bounds of the Sample Quadtree Performance 32
Figure 3.18 A Quadtree with 4 Leaves 34
Figure 3.19 A Quadtree with 7 Leaves 34
Figure 3.20 A Quadtree with 10 Leaves 35
Figure 3.21 Upper and Lower Bounds for Any Quadtree 36
Figure 3.22 Actual vs. Expected Performance 37
Figure 3.23 Actual vs. Expected Performance (Log Axes) 38
Figure 3.24 Actual vs. Expected Performance (Log Axes, Zoomed) 38
Figure 3.25 Effect of Non-Full Leaves on Performance 40
Figure 3.26 Partial Class Diagram of Shape Search Classes 42
Figure 3.27 Orientations Being Determined for Two Line Segments 46
Figure 3.28 Circle Search Performance Results 49
Figure 3.29 Rectangle Search Performance Results 50
Figure 3.30 10-Edge Polygon Search Performance Results 51
Figure 3.31 30-Edge Polygon Search Performance Results 52
Figure 3.32 Point Search Performance Results 53
Figure 3.33 Comparison of Search Performance Using Varying Bin Sizes 56
Figure 3.34 Comparison of Quadtree Build Time With Average Search Time 57
Figure 3.35 Assembly Time For Various Bin Sizes 59
Figure 3.36 Performance of Algorithm Modified for Convex Shapes 63

Figure 4.1 The Initial SMT Window 65
Figure 4.2 The New Project Dialog 66
Figure 4.3 The New Quarter Annular Project 67

Figure 4.4 The Project Layout . . . 68
Figure 4.5 The Create New Subdomain Module . . . 69
Figure 4.6 Selecting a Polygon in the Quarter Annular Project . . . 70
Figure 4.7 The Elements Selected Using the Polygon Tool . . . 70
Figure 4.8 The Create New Subdomain Dialog . . . 71
Figure 4.9 The Polygon Subdomain of the Quarter Annular Project . . . 72
Figure 4.10 The ADCIRC Module . . . 73
Figure 4.11 The Run ADCIRC Dialog . . . 73
Figure 4.12 Preparing to Run the Quarter Annular Project Subdomains . . . 74
Figure 4.13 The Open ADCIRC Project Dialog . . . 75
Figure 4.14 The North Carolina Coast Project Opened . . . 76
Figure 4.15 Viewing the Oregon Inlet Subdomain . . . 76
Figure 4.16 The Display Options Window . . . 77
Figure 4.17 Map of Camp Lejeune . . . 78
Figure 4.18 Camp Lejeune in the Full Domain . . . 79
Figure 4.19 Selecting the Camp Lejeune Subdomain . . . 79
Figure 4.20 The Camp Lejeune Subdomain . . . 80
Figure 4.21 Running Three of the North Carolina Coast Subdomains . . . 81

LIST OF ALGORITHMS

Algorithm 3.1 Add a Node to a Branch 14
Algorithm 3.2 Add an Element to a Branch 16
Algorithm 3.3 Node Search Algorithm (Branch Search) 19
Algorithm 3.4 Node Search Algorithm (Leaf Search) 20
Algorithm 3.5 Point Inside Square Test 23
Algorithm 3.6 Search Algorithm for Convex Shapes (branch) 61
Algorithm 3.7 Search Algorithm for Convex Shapes (leaf) 62

Chapter 1

Introduction

This thesis is divided into five chapters. Following this introductory chapter, the second describes the design and development of a GUI tool, called the ADCIRC Subdomain Modeling Tool (or SMT), that aims to provide a front-end for the pre- and post-processing requirements of subdomain modeling. The third chapter describes the development of a novel range search algorithm that allows modelers to have seamless interaction with an ADCIRC mesh in SMT. The fourth chapter presents two case studies that illustrate the usage of SMT, and finally, the fifth chapter includes concluding remarks and anticipated future work.

1.1 ADCIRC

ADCIRC is a program that uses finite element methods over unstructured grids to solve time dependent, free surface circulation and transport problems [1]. It is widely used by scientists and engineers to analyze the effects of hurricane storm surge on coastal regions [2]. ADCIRC can solve both two-dimensional depth integrated (2DDI) and three dimensional problems and can be run in serial or in parallel.

An ADCIRC run is performed over an unstructured mesh. An unstructured mesh is composed of nodes and triangular elements. A node represents a point in three-dimensional space and an element describes the connectivity of three nodes. Figure 1.1 shows a small portion of an

unstructured mesh that contains 7 nodes and 6 elements.

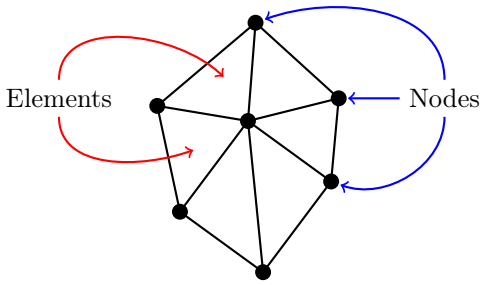


Figure 1.1: A Portion of an Unstructured Mesh

For storm surge modeling, meshes that represent the topography of a geographic region are used, and typically need to be on the order of millions of nodes and elements in order to provide sufficient land coverage as well as adequate refinement for areas of interest. Depending on the size of the mesh being used, the amount of time being simulated, the number of processors being used, and a host of other factors, an ADCIRC run can take anywhere from seconds to weeks to complete. Additionally, large problems are capable of producing an enormous amount of output with files on the order of tens of gigabytes in size.

1.2 Subdomain Modeling in ADCIRC

The subdomain modeling approach was developed with the goal of reducing the total runtime of a series of hurricane storm surge simulations performed for a particular storm on multiple, alternative topographies in a geograpic region of interest [2, 3]. This is done by recording data at the boundaries of the region of interest during the simulation of the storm event over the entire geographic region (called a full domain). The recorded data are then used to enforce the boundary values during a subdomain run, producing a solution that is nearly identical to the full domain run in a fraction of the time. With this method, a modeler is able to see the effects

of localized changes in a region of interest without having to perform multiple full domain runs, which can be very costly.

1.3 Introduction to SMT

Development of the subdomain modeling approach centered around ensuring the correctness of the solution provided by a subdomain run. The functionality needed for pre- and post-processing of data for subdomain modeling is currently provided as a command-line interface and is implemented in C++ . The goal of this thesis is to provide an interactive graphical user interface explicitly for the purpose of streamlining the pre- and post-processing requirements of the subdomain techinique as well as organizing and managing data. The design and development of this tool, called the ADCIRC Subdomain Modeling Tool (or SMT), is described in Chapter 2. Two example use cases of SMT are provided in Chapter 4.

Chapter 2

Design and Development of SMT

The development of the ADCIRC Subdomain Modeling Tool has been inspired by the need to streamline the pre- and post-processing needs of the subdomain modeling approach in order to provide an interactive environment for modelers to use and to promote the use of the subdomain approach. The following sections describe the development process from initial design decisions and goals to specific implementation details.

2.1 Design Goals

The design choices made in creating the ADCIRC Subdomain Modeling Tool were guided by the following three ideals:

Speed

An average PC should be able to run SMT without any significant speed limitations. A very large ADCIRC mesh (for example, the western north-Atlantic mesh, which contains approximately 1.2 million elements) should be able to load in a reasonable amount of time, and computations that provide the backend for all interactivity features should perform well enough that they go unnoticed by the user. As benchmarks, unit tasks such as loading a file should take no more than 10 seconds [4] and graphical processes such as zooming, panning, and selection should take

no longer than 100 ms, which is considered to be the perceptual processing time constant [4, 5], or the largest amount of time between an action and a reaction that can go unnoticed.

Interactivity

The user interface should provide both an interactive experience with ADCIRC meshes, as well as a logical and intuitive layout that allows even the most inexperienced modelers to create and run a subdomain with minimal instruction.

Extensibility

The coding should be intuitive, portable, and well documented so that developers can quickly and easily add features to the core program as they are needed by modelers.

2.2 Identification of Required Functionality

The design of the SMT begins with the identification of the functionality it will provide. This functionality is determined by stepping through the workflow of subdomain modeling.

Targeting Regions of Interest

The first task a modeler must complete when using the subdomain approach is locating and selecting the region of interest in the full domain. In order to complete this task, SMT must provide a visual representation of the full domain mesh with which the user can interact. ADCIRC meshes can have wide range of refinement, so the ability to zoom and pan is required to be able to visualize every part of the mesh with appropriate accuracy. Location of areas of interest may be based on the topography of a region for some modelers, while for others it may be based on geographical coordinates, so SMT must provide a way to visualize these as well. A number of selection tools that allow the modeler to select elements in the subdomain also need to be provided.

Handling Multiple Subdomains

The subdomain modeling technique allows a modeler to create multiple subdomains under a single full domain. The boundary conditions for each of these subdomains is recorded during a single full domain run. SMT needs to provide a way to manage multiple subdomains as well keep different sets of full/subdomains separate and organized.

File Management and Backend Operations

Once an area of interest has been targeted, the modeler needs to create a number of files and place them in specific directories in order for the subdomain modeling approach to work correctly. SMT should automate this process with minimal input from the user. Additionally, after a full domain run, the boundary condition data for each subdomain must be extracted. Depending on the length of the run and the number of boundary nodes, this process can be very time consuming. SMT must provide a strong backend that performs these operations as quickly as possible.

Mesh Modification

Being able to run multiple small domains with localized changes is one of the features that draws users to subdomain modeling, so SMT must provide a way of making localized changes to a subdomain as well as quantifying or visualizing the changes in the solution that result from localized changes to the mesh.

2.3 Characterization of Core Features

With the required functionality defined, a characterization of the core features of SMT is listed.

2.3.1 GUI

SMT provides a graphical user interface in which all subdomain modeling operations are performed. The GUI is partitioned in a way such that the mesh visualization takes up the largest portion of the display and the modules window fills the remainder of the available space.

Mesh Visualization

The largest portion of the GUI is dedicated to visualizing the mesh because it is expected that most of the modeling time will be spent interacting with the mesh or visualizing data on the mesh.

Modules

The modules portion of the GUI provides an interface for all non-visual operations and is laid out in such a way that the work flow of subdomain modeling is mirrored. It contains modules for managing projects, creating subdomains, editing features in subdomains, running ADCIRC, and analyzing results.

2.3.2 Projects

SMT organizes subdomains into groups based on the full domain from which they were created. A full domain and all of its subdomains are considered an ADCIRC Subdomain Project. A project can be saved using the specified ADCIRC Subdomain Project File format (.spf). When a project is opened in SMT, the structure of the project is displayed in the Project Explorer module. This structure is similar to that of a file-tree.

2.3.3 Backend

SMT uses C++ file I/O to provide a high performance backend for boundary data extraction, which is a time consuming process that almost exclusively involves reading and writing large

files. The visualization backend is implemented using OpenGL [6] and the computationally intensive interactivity features (i.e., node and element selection) use a novel range search algorithm developed specifically for this application, that is described in Chapter 3.

2.4 Development of SMT

SMT is written entirely in C++ because of its speed [7] and because it supports object-oriented programming and has a wealth of options for free, cross-platform GUI toolkits [8, 9, 10]. Additionally, because of the requirement for a fast, I/O based backend, a C based language is an obvious choice. Java and Python were considered as alternatives but initial tests of the visualization libraries [11, 12, 13] did not perform well when interacting with or animating very lage meshes.

Qt [8] is used as the GUI framework because it is free[1], cross-platform, extensively documented and well supported, and provides a native OpenGL context. Additionally, Qt's event handling and signal/slot mechanism are well documented and very easy to understand and implement, providing for furter extensibility and ease of development.

OpenGL is used for all graphics rendering because it is open-source, cross-platform, high performance, and enjoys very widespread use. Direct3D was also considered, but because it is exclusive to Microsoft operating systems, OpenGL was chosen as the more portable option.

2.4.1 GUI Development

The layout of the SMT user interface is a result of numerous iterations in which many different layouts were tested. Early iterations focused more on visualization rather than a strong coupling of visualization and subdomain modeling. Typically, more options for modifying how the mesh appeared were visible to the user while tools for subdomain modeling were hidden in menus. As the importance of these tools became apparent, these roles were reversed. Now, the subdomain

[1] Qt is available under the GNU GPL (version 3). See `http://qt-project.org/doc/qt-5.1/qtdoc/gpl.html`

modeling tools are more organized and more visible to the user, which is more in line with the goal of providing a streamlined user interface specifically for use in subdomain modeling.

2.4.2 Developing ADCIRC Mesh Interactivity

Providing interactivity with an ADCIRC mesh is one of the most vital features for subdomain modeling. OpenGL provides sufficient performance to display very large meshes as well as navigate within them by zooming and panning. However, the ability to select a subset of elements from the mesh in an interactive way is a more difficult problem that must be solved by SMT. Additionally, the goal was to provide a number of different ways for modelers to select elements so that they would not be limited by the shape of their subdomains. Early versions of SMT used the exhaustive search method, in which each node or element is tested individually in order to determine if it falls inside of the desired shape, but it became clear very quickly that this method could not provide sufficient performance in order to maintain an appropriate level of interactivity with very large meshes.

Research showed that the only algorithms specifically made to find a subset of spatial data from a larger set, called 'range search' algorithms [14], are only capable of finding data in a set range, which in the case of 2-dimensional data, means a rectangle. As a result, we have developed an algorithm, called the generalized range search algorithm, that is capable of finding a subset of data that falls within a range defined by any arbitrary 2-dimensional shape. The design and development of this algorithm is described in depth in Part 3.

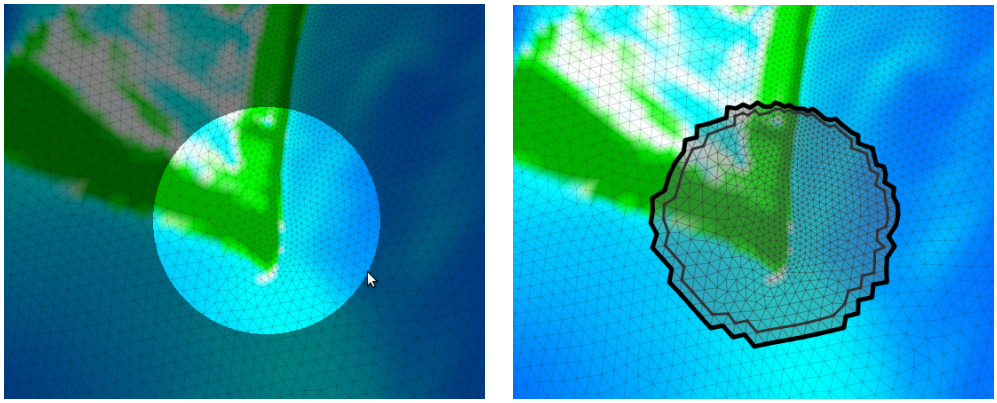
Using the generalized range search algorithm as the backend, the element selection tools, which currently include a point search, circle search, rectangle search, and polygon search, provide the level of responsiveness required to create an interactive user interface.

Chapter 3

Search Algorithm

3.1 Introduction to the Algorithm

The generalized range search algorithm developed for the ADCIRC Subdomain Modeling Tool is essential to providing responsive interactivity with the ADCIRC mesh. Node and element selections in SMT are made by using one of the shape selection tools to draw a shape on a plan view of the ADCIRC mesh. For example, the images in Figure 3.1 show a circle being drawn over a mesh and the elements that are selected after the circle is 'dropped'.



(a) Drawing the Circle (b) The Selected Elements

Figure 3.1: Drawing and Selecting From a Circle

The goal of this algorithm is to provide a very fast method for selecting nodes or elements that fall within any arbitrary shape. The algorithm combines many existing methods in order to produce the desired result.

3.2 Data Structure

The key to this algorithm's success is the underlying data structure, a quadtree. More specifically, the physical properties of the quadtree as they relate to the mesh that it contains are what make the algorithm possible.

3.2.1 Quadtree Basics

A quadtree is a hierarchical data structure in which each branch contains exactly four children [15, 16]. Each of these children can be either a leaf or another branch. The top branch is called the root branch. Figure 3.2 shows an example of a simple quadtree with two branches.

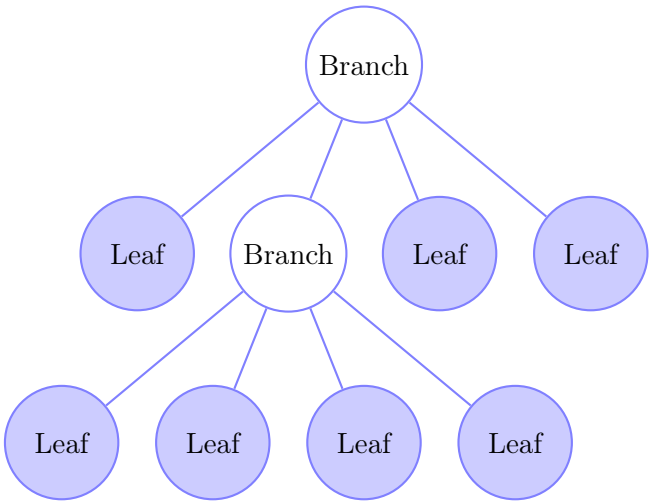


Figure 3.2: A Simple Quadtree with Two Branches

This simple data structure can be used to decompose a two-dimensional finite element mesh into smaller, more manageable groups based on the physical location of nodes or elements. Many types of quadtrees exist, each with their own benefits and drawbacks. This algorithm uses a simple *bucket PR quadtree*, where PR stands for point-region [14]. PR quadtrees have been used successfully in many GIS applications [17, 18] and they contain the properties required for this algorithm to work properly.

PR quadtrees are partitioned into equal-sized regions rather than based on data point location, as in point quadtrees. This partitioning is done recursively until each leaf contains only a single data point. Because tree structures often use pointers, excessive page faults can lead to slow memory accesses and decreased performance [14]. However, because region partitioning is used, we are able to use bucket methods. Bucket methods aim to reduce page faults and ensure efficient memory accesses [14]. They do this by allowing leaf nodes to contain more than one data point. In essence, the bucket method groups data points that are physically close to each other, which is the property that will be leveraged in this algorithm.

3.2.2 Storing a Finite Element Mesh in a Quadtree

Before a search can be performed over the finite element mesh, the nodal and elemental data need to be stored in the quadtree. The following data are needed to populate the quadtree:

- A list of x- and y- coordinates for all nodes
- A list of elements
- The minimum and maximum x- and y- nodal coordinates
- The maximum allowable number of nodes in a leaf (i.e., the bucket size)

Starting with this set of data, an empty quadtree is created. This empty quadtree consists of a root branch that has four empty leaves. The root branch is represented physically as a rectangle with bounds $(x_{min}, x_{max}), (y_{min}, y_{max})$. Splitting the root branch down the middle

along both the x- and y-coordinates gives the bounds of the four leaves. Figure 3.3 shows the root branch of an empty quadtree and Figure 3.4 shows the four leaves of an empty quadtree.

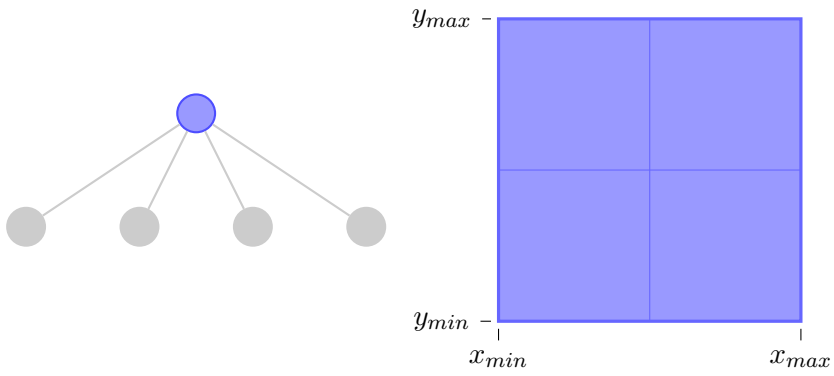


Figure 3.3: The Empty Quadtree with the Root Branch Highlighted

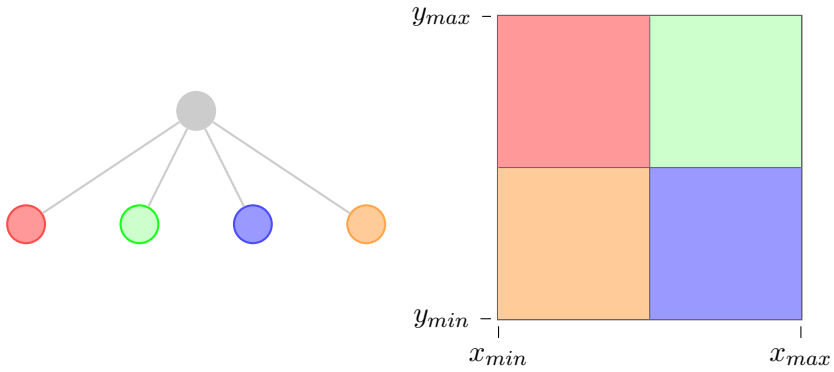


Figure 3.4: The Empty Quadtree with the Leaves Highlighted

Adding Nodes to the Quadtree

Once the empty quadtree has been created, the nodes are added to the quadtree using the recursive algorithm defined in Algorithm 3.1 that is based on the insertion algorithm for PR

quadtrees [14] but modified to accomodate the bucket method.

Algorithm 3.1: Add a Node to a Branch

```
   /* Branch: The branch to add the node to                          */
   /* Node: The node to add to Branch                                */
   /* BinSize: The maximum number of nodes in a leaf                 */
 1 Algorithm AddNode(Branch, Node, BinSize)
 2     for subLeaf in Branch do
 3         if Node is inside subLeaf then
 4             Add Node to subLeaf;
 5             if subLeaf.nodeCount > BinSize then
 6                 subLeaf → subBranch;
 7                 for subNode in subLeaf do
 8                     AddNode(subBranch, subNode, binSize);
 9             return;
10     for subBranch in Branch do
11         if Node is inside subBranch then
12             AddNode(subBranch, Node, binSize);
13             return;
```

This `AddNode` function is called for each node with the root branch and bucket size as the two additional parameters. The first `for` loop determines if the node falls into any of the four leaves that the target branch contains. If it does, the node is added to the leaf and a test is performed to determine if the number of nodes in that leaf now exceeds the bucket size. If it does, the leaf is converted to a branch and all of the nodes from that leaf are added to this new branch using the `AddNode` function. Otherwise, the function returns.

If, however, the node did not fall into any leaves of the target branch, the second `for` loop determines if the node falls into any of the branches that the target branch contains. If it does, the function recurses, using the appropriate sub-branch as a parameter.

Figure 3.6 shows how the nodes of the small finite element mesh in Figure 3.5 are stored within a quadtree of size two.

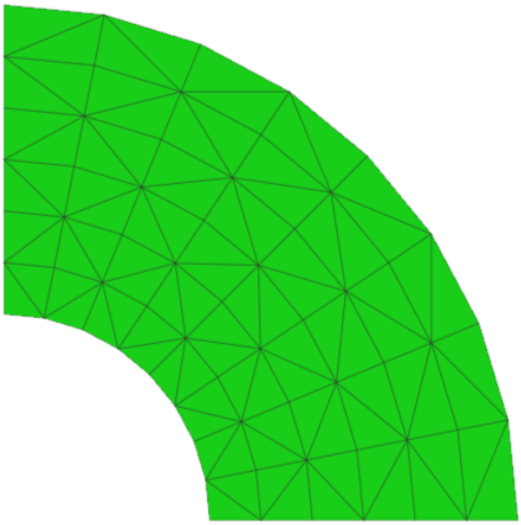


Figure 3.5: Visualization of the Quarter-Annular Case

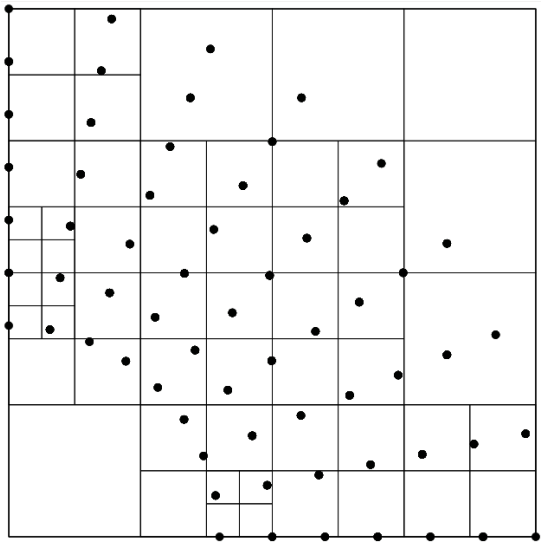


Figure 3.6: Visualization of Quadtree Containing the Quarter-Annular Case

Adding Elements to the Quadtree

Once the nodes have been added to the quadtree, adding the elements is very simple. At this point, the structure of the quadtree is finalized, so adding elements is merely a matter of finding the correct leaf (or leaves) to add each element to. The recursive procedure presented in Algorithm 3.2 is used.

Algorithm 3.2: Add an Element to a Branch

```
   /* Branch: The branch to add the element to                    */
   /* Element: The element to add to Branch                       */
1  Algorithm AddElement(Branch, Element)
2  |  for subLeaf in Branch do
3  |  |  if Element is inside subLeaf then
4  |  |  |  Add Element to subLeaf;
5  |  for subBranch in Branch do
6  |  |  if Element is inside subBranch then
7  |  |  |  AddElement(subBranch, Element);
```

This is the same procedure used to add nodes to the quadtree with the exception that there is no check against a bucket size. In order for the search algorithm to find all elements when performing a search, an element must be considered to be inside of a leaf or branch if any of its three nodes fall within the leaf or branch. This approach has the added side-effect of allowing an element to be placed into multiple leaves. Therefore, the structure of the quadtree is determined by the node partitioning in order to simplify the creation of the quadtree.

3.3 Search Algorithm

The goal of the search algorithm is to overlay a 2-D finite element mesh with any arbitrary shape and extract the set of nodes or elements that fall inside of the shape. Numerous search algorithms, called range search algorithms, exist to search quadtrees for data points that fall within a specified region [14, 15], but to our knowledge none exist in publication that specifically seek to leverage the properties of the bucket methods and apply them to a generalized shape.

The search algorithm uses recursive methods, as is common in hierarchical data structures [14]. It also bears some resemblance to R-Tree search algorithms [19] in that a 'pruning' of sorts is performed on the children of a branch that is known to not intersect with the range of interest in any way.

3.3.1 Theory

There are three basic concepts that allow the algorithm to perform quickly and correctly:

- If a branch or leaf of the quadtree falls completely within an arbitrary shape, all of the nodes and elements contained by that branch or leaf are also guaranteed to fall within the shape.
- If an edge of a branch or leaf of the quadtree intersects with an edge of the arbitrary shape, the nodes and elements contained by that branch or leaf *might* fall within the shape.
- If a branch or leaf of the quadtree does not fall completely within or intersect with an arbitrary shape, all of the nodes and elements contained by that branch or leaf are guaranteed to not fall within the shape.

Using these properties, it is possible to traverse the quadtree in a depth-first search manner and very quickly drastically reduce the size of the exhaustive search that is used to find the set of nodes or elements that fall inside of an arbitrary shape.

3.3.2 Shape

In order to provide a method that is very generalized and can be applied to as many arbitrary shapes as possible, the search algorithm requires that the shape meet only three simple standards:

- Knowledge of the location of any single arbitrary point on the shape's edge
- A test exists to determine if a node is inside of the shape

- A test exists to determine if a line segment intersects with the shape

These two tests are used extensively throughout the algorithm, so the more complex these tests become, the longer the search will take.

3.3.3 The Algorithm

This section describes the search algorithm in depth. A node search is used in the description, but note that an element search is identical with two minor exceptions. First, the exhaustive search performed once the algorithm has completed is performed on a list of elements insead of a list of nodes. Second, the list of elements returned by the algorithm may contain duplicates because of the manner in which an element is determined to fall inside of a branch or leaf. Therefore, these duplicates must be removed.

Before beginning the search, the following is required:

- An empty list, `finalNodes`, that will contain the final list of nodes that fall within the shape
- An empty list, `partialNodes`, that will contain nodes that might fall within the shape
- A quadtree root, `root`, as an entry point to begin the search
- A shape, `shape`, that has the following variable and two functions:
  - `shape.testPoint` - any single point that falls on the boundary of the shape
  - `shape.pointIsInside(Point p1)` - a function that returns true if the point is inside the shape, false otherwise
  - `shape.intersects(Branch b)` and `shape.intersects(Leaf l)` - an overloaded function returns true if the shape intersects with the edges of the branch or leaf, false otherwise

The algorithm, defined in Algorithm 3.3, is an overloaded function that performs the search, accepting either a branch or a leaf as its parameter. An overloaded helper function is provided, `AddAll()`, that takes a list and a branch or leaf as parameters. If a branch is passed as a parameter, all nodes in every leaf below that branch will be added to the list. If a leaf is passed as a parameter, all nodes in that leaf will be added to the list. Additionally, the `EarlyExit()` function indicates that the search can be terminated.

Algorithm 3.3: Node Search Algorithm (Branch Search)

```
1  Algorithm SearchNodes(Branch currBranch)
2      if shape.intersects(currBranch) then
           /* The current branch overlaps the shape                    */
3          for subBranch in currBranch do
4              SearchNodes(subBranch);
5          end
6          for subLeaf in currBranch do
7              SearchNodes(subLeaf);
8          end
9      else
10         if currBranch.contains(shape.testPoint) then
               /* The shape is inside of the current branch            */
11             for subBranch in currBranch do
12                 SearchNodes(subBranch);
13             end
14             for subLeaf in currBranch do
15                 SearchNodes(subLeaf);
16             end
17             EarlyExit();
           end
18         if shape.pointIsInside(currBranch.NorthEast) then
               /* The current branch is inside of the shape            */
19             AddAll(currBranch, finalNodes);
20         else
               /* The current branch is outside of the shape           */
21             return;
           end
       end
```

The search algorithm applied to a leaf, defined in Algorithm 3.4, is almost identical to the

algorithm applied to a branch, with the exception that the leaf terminates the recursion process.

Algorithm 3.4: Node Search Algorithm (Leaf Search)

```
1  Algorithm SearchNodes(Leaf currLeaf)
2      if shape.intersects(currLeaf) then
           /* The current leaf overlaps the shape                    */
3          AddAll(currLeaf, partialNodes);
4      else
5          if currLeaf.contains(shape.testPoint) then
               /* The shape is inside of the current leaf            */
6              AddAll(currLeaf, partialNodes);
7              EarlyExit();
           end
8          if shape.pointIsInside(currLeaf.NorthEast) then
               /* The current leaf is inside of the shape            */
9              AddAll(currLeaf, finalNodes);
10         else
               /* The current leaf is outside of the shape           */
11             return;
           end
       end
```

3.3.4 Explanation

The algorithm steps into the quadtree recursively checking for very specific relations between the current branch or leaf and the target shape. At each recursion step a number of tests are performed in order to determine the physical relationship between the shape and the branch or leaf being tested. The order in which these tests are performed is very important. From these relationships, it is possible to make deductions about the location of every node contained in the quadtree with relation to the shape being tested.

First, a test is performed to determine if any of the four edges of the leaf or branch intersects with the edge of the shape. If there is an intersection, the only conclusions that can be drawn are that any nodes the leaf or branch contains *may* fall inside of the shape. Therefore, if the test was performed on a branch, the algorithm recurses on that branch, and if the test was performed on

a leaf, the nodes inside of that leaf are added to the list of nodes that could potentially fall inside of the shape. The actual test for an intersection is performed using the function supplied with the shape. Because this test is guaranteed to be performed at each step of recursion, choosing a fast and efficient method is crucial to the performance of the algorithm.

If it is determined that there are no edge intersections between the shape and the current leaf or node, one of three possible deductions can be made. Either the branch or leaf contains the entire shape, as shown in Figure 3.7, the shape contains the entire branch or leaf, as shown in Figure 3.8, or the branch or leaf is fully outside of the shape, as shown in Figure 3.9.

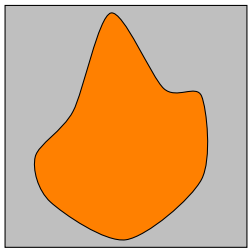


Figure 3.7: The Branch or Leaf Contains the Entire Shape

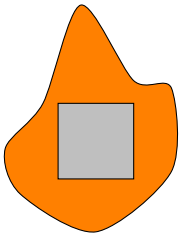


Figure 3.8: The Shape Contains the Entire Branch or Leaf

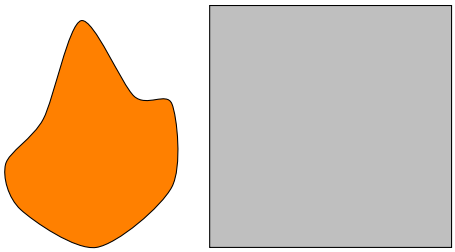


Figure 3.9: The Branch or Leaf is Fully Outside of the Shape

Testing to determine which of these three conditions is satisfied can be performed using only two tests.

Branch or Leaf Contains Shape

The test to determine if the branch or leaf contains the entire shape is performed first. It consists of a point-in-rectangle test and is always the same, regardless of the shape being tested. This test is typically no worse computationally than the test for a point inside of the shape, which is used to test for the other two conditions. Additionally, large portions of the quadtree can be removed from searching when this condition is found to be true, so performing this test first is typically beneficial.

Because the algorithm has already determined that there are no edge intersections, if any arbitrary point that falls on the *edge* of the shape is contained inside of the branch or leaf being tested, it can be deduced that the entire shape must also be contained inside of the branch or leaf. Note that an interior point of the shape being contained inside of the branch or leaf *does not* guarantee that the entire shape is inside of the branch or leaf. Therefore, the test is performed using the required edge point provided with the shape.

The node in branch or leaf test can be performed a number of ways. Many generalized point-in-polygon tests exist, including winding number and ray-casting, both of which are $O(n)$ [20]. The winding number algorithm relies heavily on trigonometric floating point operations, making it significantly slower than ray-casting [21]. The ray-casting algorithm as implemented in [20]

relies on dozens of floating point operations and logical comparisons.

However, because all branches and leaves are guaranteed to be axis-aligned rectangles, a simple bounds test can be performed. In this test, defined in Algorithm 3.5, a maximum of four equality comparisons are performed, making it extremely fast.

Algorithm 3.5: Point Inside Square Test

```
bool PointIsInsideSquare(Point p, Branch branch)
    if p.x ≥ branch.leftBound and
       p.x ≤ branch.rightBound and
       p.y ≥ branch.lowerBound and
       p.y ≤ branch.upperBound then
        return true;
    else
        return false;
    end
```

If a branch is being tested and it is determined that the branch contains the entire shape, the algorithm recurses over the branch's children. If a leaf is being tested and it is determined that it contains the entire shape, the only deduction that can be made is that any node contained by the leaf *may* fall inside of the shape. Therefore, all of the nodes in the leaf are added to the list of nodes that might fall inside of the shape. In either case, the location of the entire shape is now known and any searching that remains to be done at or above the current level of the quadtree can be eliminated.

Branch or Leaf Contained Within Shape or Fully Outside Shape

If the branch or leaf does not contain the shape, determining which of the two remaining possible conditions holds true can be done using a single test. If one is determined to be true, the other must be false, and similarly, if one is determined to be false, the other must be true.

Because it has been determined that there are no edge intersections, if any point on the edge of the branch or leaf is inside of the shape, it can be deduced that the entire branch or leaf must fall inside of the shape. For simplicity, this test is performed by determining if one of the randomly chosen corner points of the branch or leaf falls inside of the shape (the northeast

corner is used in Algorithms 3 and 4). The test itself is performed using the function that is provided with the shape. This test function is also used to perform the exhaustive search over nodes that may fall into the shape, so choosing a fast and efficient method is crucial to the performance of the algorithm.

If a branch is being tested and it is determined that the branch falls entirely inside of the shape, it can be deduced that every child of the branch also falls entirely inside of the shape. Therefore, every node contained in every leaf below the branch is added to the list of nodes that are guaranteed to fall inside of the shape. On the other hand, if it is determined that the branch falls entirely outside of the shape, the branch and every node contained in every leaf below the branch are ignored during the rest of the search because they are guaranteed to outside of the shape.

Similarly, if a leaf is being tested and it is determined that it falls entirely inside of the shape, all of the nodes contained in the leaf are added to the list of nodes that are guaranteed to fall inside of the shape, and if it is determined that the leaf falls entirely outside of the shape, all of the nodes contained by the leaf are ignored.

Summary

Table 3.1 summarizes the conditions being checked and the result that occurs if the condition is met for both a branch and a leaf.

Table 3.1: Effects of Conditions Met During the Search Algorithm

Condition	**Branch**	**Leaf**
Has intersection with shape	Recurse on children	Add all nodes to list of nodes that could possibly be inside of the shape
A point on the shape edge is inside bounds	Recurse on children and then terminate search	Add all nodes to list of nodes that could possibly be inside of the shape and terminate search
A corner is inside of the shape	Add the nodes of all children to the list of nodes that are guaranteed to be inside of the shape	Add all nodes to the list nodes that are guaranteed to be inside of the shape
Else	None of the nodes contained are inside of the shape	

Exhaustive Search

Once the algorithm completes, the two lists will have finished populating and each node in the the list of nodes that might fall inside of the shape needs to be tested individually. If the node falls inside of the shape it is added to the list of nodes guaranteed to fall inside of the shape, otherwise it is discarded. The exhaustive search tests are performed using the point inside shape test provided with the shape. In the worst case, every node in the quadtree will need to be tested using the exhaustive search, so choosing a very fast method of performing this test is important.

3.4 Expected Performance

The following three sections describe how the algorithm is expected to perform in any given scenario.

3.4.1 Theory

Determining the expected performance of the search algorithm analytically is difficult because it is not possible to know a priori the subset of the quadtree that will actually be reached by the algorithm for any given search. However, for any given quadtree, it is possible to quantify both an upper and lower bound on the expected performance for any given search. These definitions of these bounds are derived using the simple quadtree shown in Figure 3.10.

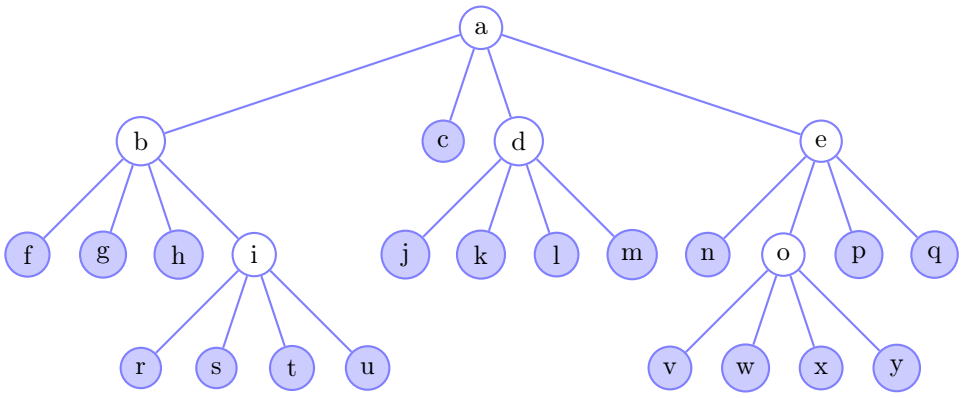


Figure 3.10: A Simple Filled Quadtree

The terminology used in this example is as follows:

- A circle in the quadtree is called a *quadtree element*. Quadtree elements will be written in bold italics (e.g., $\boldsymbol{a}$)
- Every quadtree element is either a *branch* or a *leaf*. Branches are white and always have four children. Leaves are blue and have no children.

- A line connecting two quadtree elements is called an *edge*. Edges will be written in the form $\overline{\boldsymbol{ab}}$, where the first letter indicates the parent branch and the second indicates the child quadtree element.
- The *solution* is the set of leaves that are found by the algorithm.
- The *solution size* is the number of leaves in the solution.

Additionally, the following assumptions are made:

- The search will always begin at branch $\boldsymbol{a}$.
- The only expensive operation being performed is tree traversal (i.e., following an edge from a branch to a child of that branch).
- The cost of following an edge is constant.

At its core, the search algorithm simply determines which of the four edges of any given branch to follow. Independent of any implementation specifics that define how these decisions are made, the number of edges followed once a branch has been reached can always be 0, 1, 2, 3, or 4. For example, should the algorithm choose to follow the edge $\overline{\boldsymbol{ab}}$, it can then choose to follow any, or none, of the edges $\overline{\boldsymbol{bf}}$, $\overline{\boldsymbol{bg}}$, $\overline{\boldsymbol{bh}}$, or $\overline{\boldsymbol{bi}}$. This process is repeated recursively until either the entire quadtree has been traversed or the algorithm decides not to follow any more edges. Once the algorithm has completed, the solution is the set of leaves that have been reached. This solution can be any possible subset of the quadtree, and is determined by how the edge-following decisions are made (e.g., In the SMT implementation, the user supplies an arbitrary shape and the decisions are made based on the physical relationship between the shape and the branch).

Because of the assumption that following an edge is the only expensive operation being performed, the metric used to evaluate the performance of the algorithm for any given solution needs to be the total number of edges that were followed in order to find that particular solution. As an example, the solution in Figure 3.11 was found by following six edges (bold red edges indicate edges followed during a solution).

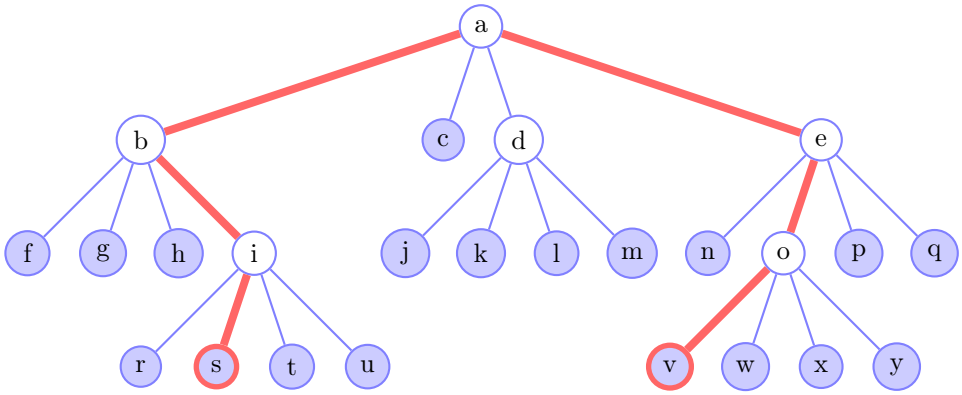


Figure 3.11: 6 Edges Followed, 2 Leaves Found

Therefore, the performance of this solution can be calculated to be $6 \times c_e$, where c_e is the cost of following an edge.

Additionally, because solutions can come from any subset of the full quadtree, it is possible that there are many distinct solutions with the same solution size. As such, the performance of the algorithm for a given solution size can vary. For example, the solution in Figure 3.12 also contains two leaves, but the solution was found by following only 3 edges. Therefore, the performance of this solution is only $3 \times c_e$, meaning it was found twice as fast as the solution in Figure 3.11.

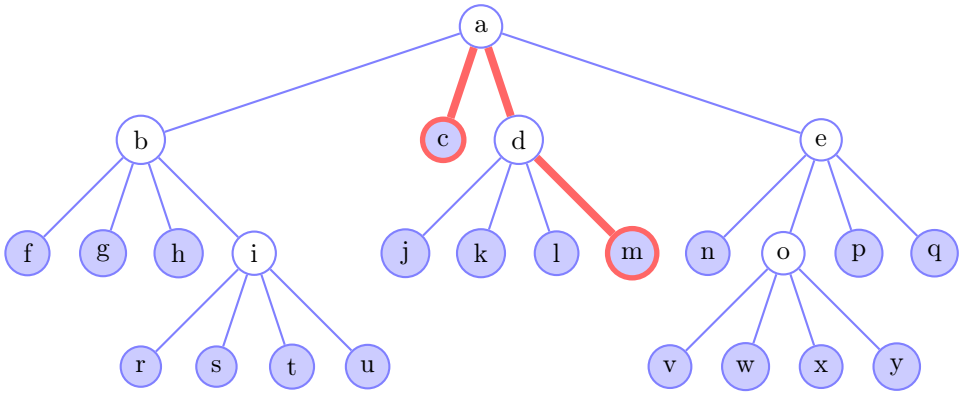


Figure 3.12: 3 Edges Followed, 2 Leaves Found

As a result, in order to determine the upper and lower bounds of the expected performance of the algorithm as a whole, the best and worst possible performance for every possible solution size must be found. Plotting these values reveals a graph that shows how the algorithm will perform for any given solution size.

Producing the upper and lower bound solutions for a small quadtree such as the one in Figure 3.10 is a very simple process that can be performed visually. As solutions are produced for each solution size, they are plotted on a graph where the x-axis is the solution size (ranging from 0 to n, where n is the total number of leaves in the quadtree), and the y-axis is the number of edges followed in order to produce the solution (ranging from 0 to e, where e is the total number of edges in the quadtree). For example, Figure 3.13 shows the minimum solution size for which a single leaf is found.

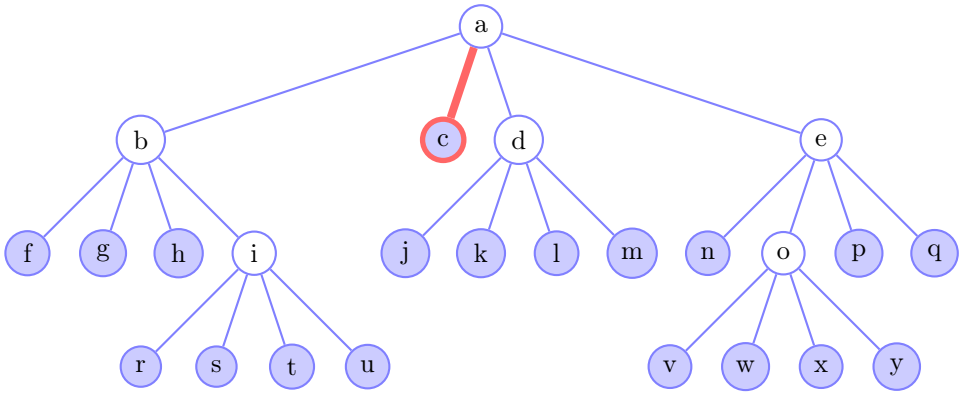


Figure 3.13: 1 Edge Followed, 1 Leaf Found

This solution followed one edge and found one leaf, so the point (1, 1) would be plotted. Similarly, Figure 3.14 shows the maximum solution size for which a single leaf is found.

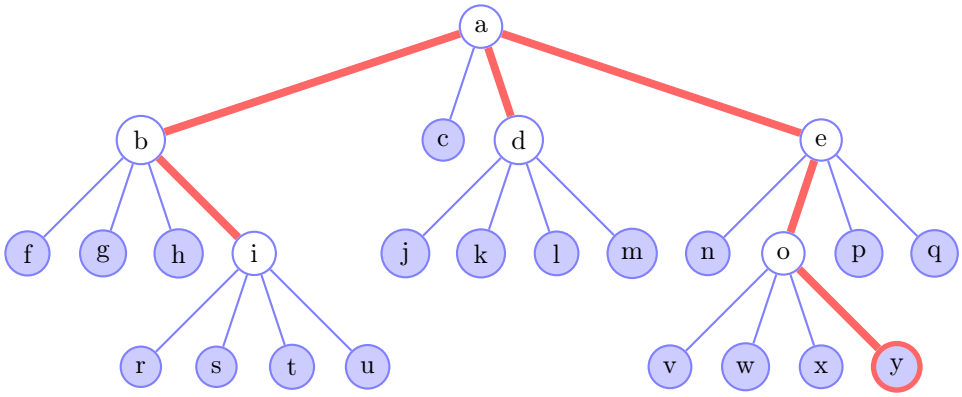


Figure 3.14: 6 Edges Followed, 1 Leaf Found

This solution followed 6 edges and found one leaf, so the point (1, 6) would be plotted, producing the graph in Figure 3.15, which now contains the minimum and maximum possible performance for a solution size of one.

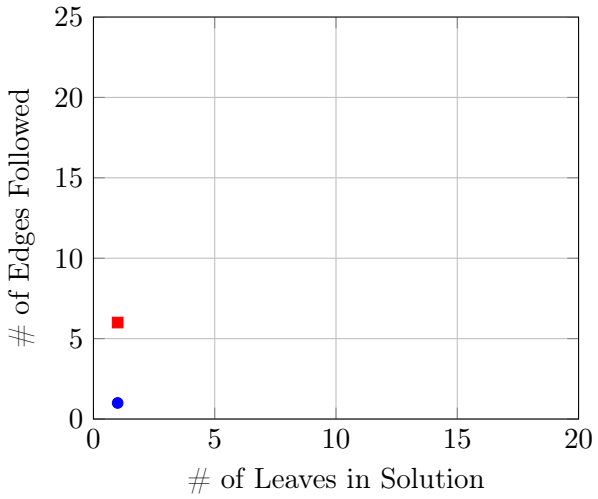


Figure 3.15: Plot for a Solution Size of 1

Because these values represent the best and worst performance for a solution size of one, it can be deduced the performance for any random solution containing one leaf is guaranteed to fall in the range $[1, 6]$. Therefore, repeating this process for every possible solution size will produce a plot that shows the performance of the algorithm for any solution size, as can be seen in Figure 3.16.

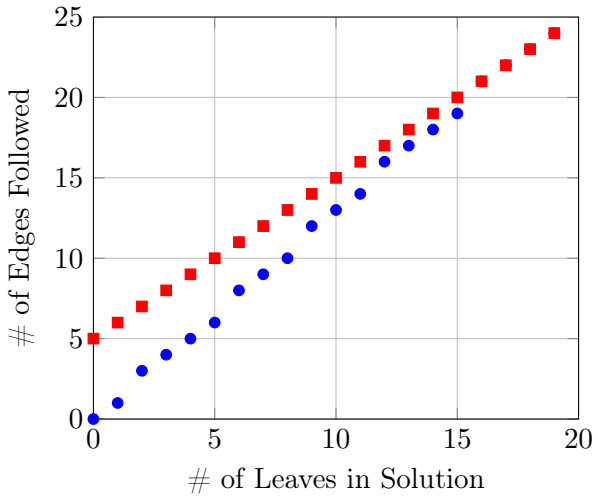


Figure 3.16: Performance Plot for All Solution Sizes

Connecting these points produces the exact upper and lower bounds of the performance of the algorithm, as seen in Figure 3.17.

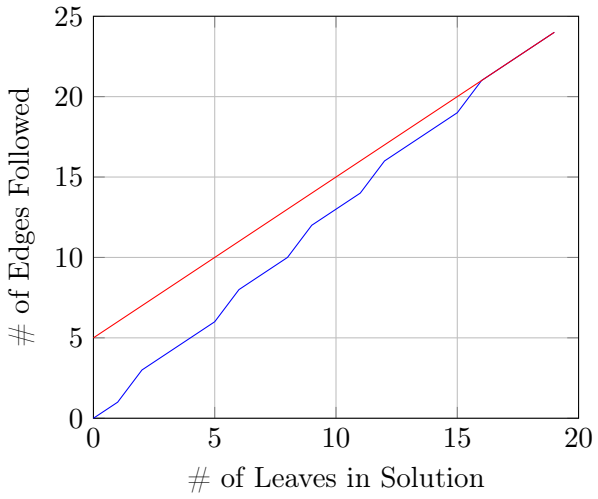


Figure 3.17: Upper and Lower Bounds of the Sample Quadtree Performance

A number of deductions can be made from this plot. First, the worst performance for any given problem size will always be a result of visiting every branch in the quadtree before finding any leaves (e.g., the solution in Figure 3.14). Once every branch in the quadtree has been visited, following any remaining edge in the quadtree will result in finding a single leaf, which is why the plot of the upper bound is linear. This is a property that is always true for the search algorithm, independent of the quadtree shape or size, so the following can be said about the worst possible performance of the algorithm:

The upper bound of the algorithm performance is

$$O(n) = b + n$$

where n is the solution size and b is the total number of branches below the root of the quadtree.

The best performance, on the other hand, will always come from the smallest breadth first search solution for a given solution size. This solution is found when the algorithm always chooses the leaf of a branch that has already been visited instead of visiting another branch in order to find a leaf. For example, in Figure 3.13 the algorithm chose to follow edge $\overline{\boldsymbol{ac}}$ to find a leaf instead of first visiting one of the branches $\boldsymbol{b}$, $\boldsymbol{d}$, or $\boldsymbol{e}$ and then finding a leaf. Because the number of leaves for any given branch will vary between quadtrees, it can be concluded that the lower bound of the performance of the algorithm is dependent on the size and shape of the quadtree. However, an absolute lower bound of the algorithm performance as applied to any quadtree does exist. Consider the quadtree presented in Figure 3.18.

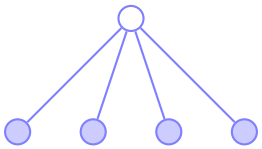


Figure 3.18: A Quadtree with 4 Leaves

The best performance for solution sizes of 1, 2, 3, and 4 will always be 1, 2, 3, and 4, respectively. For any quadtree with more than four leaves, however, at least one of the leaves will have to be turned into a branch. So, adding four more leaves produces the quadtree in Figure 3.19.

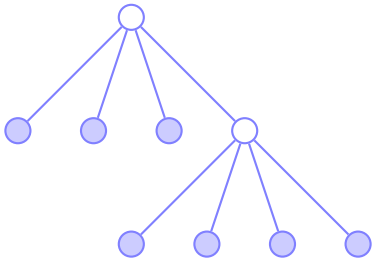


Figure 3.19: A Quadtree with 7 Leaves

Now the best possible performance for a solution size of 4 is 5 because an edge has to be followed in order to reach the fourth leaf.

Adding four more leaves to the quadtree will result in another leaf being turned into a branch. If one of the three remaining leaves from the original quadtree is made a branch, the best performance for a solution size of 3 will become 4. This can be avoided by converting one of the four lowest level leaves into a branch instead, resulting in the quadtree in Figure 3.20.

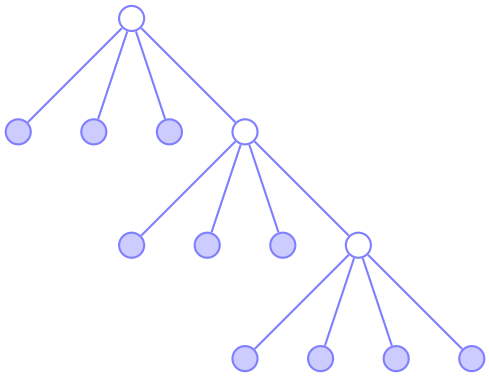


Figure 3.20: A Quadtree with 10 Leaves

This process of always converting one of the lowest level leaves into a branch continues until the desired quadtree size is reached. The quadtree produced using this method will always contain the solution with the absolute best performance possible from the algorithm for any quadtree of equal or greater size. The following function, which describes the behavior of this pattern, can be used to define the absolute lower bound on algorithm performance:

The lower bound of the algorithm performance is

$$O(n) = \begin{cases} 0 & n = 0 \\ n + \frac{(n-1)-(n-1) \bmod 3}{3} & 0 < n < l \\ b + n & n = l \end{cases}$$

where n is the solution size, l is the number of leaves in the quadtree, and b is the number of branches below the root of the quadtree.

Finally, plotting these two bounding functions gives the area in which algorithm performance must always fall, as shown in Figure 3.21.

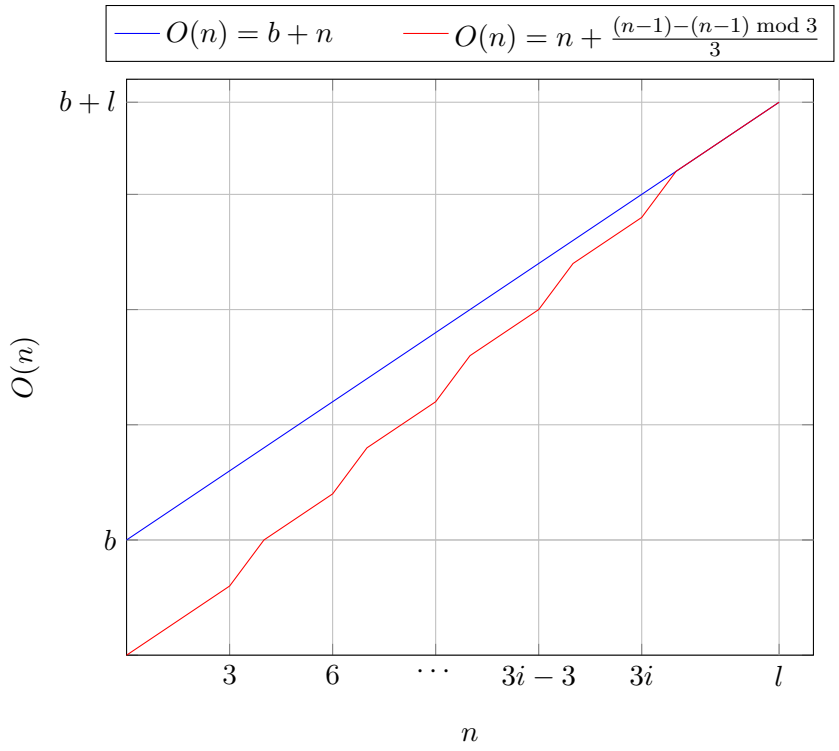


Figure 3.21: Upper and Lower Bounds for Any Quadtree

3.4.2 Verification

Bounds Validation

Validation of the aforementioned limits is done using an ADCIRC mesh with approximately 30,000 nodes and 60,000 elements. Five thousand random circles are produced and the circle shape search implementation of the search algorithm is applied using each circle. As the algorithm runs, the number of leaves reached and the number of edges followed are counted. The results are plotted in Figure 3.22.

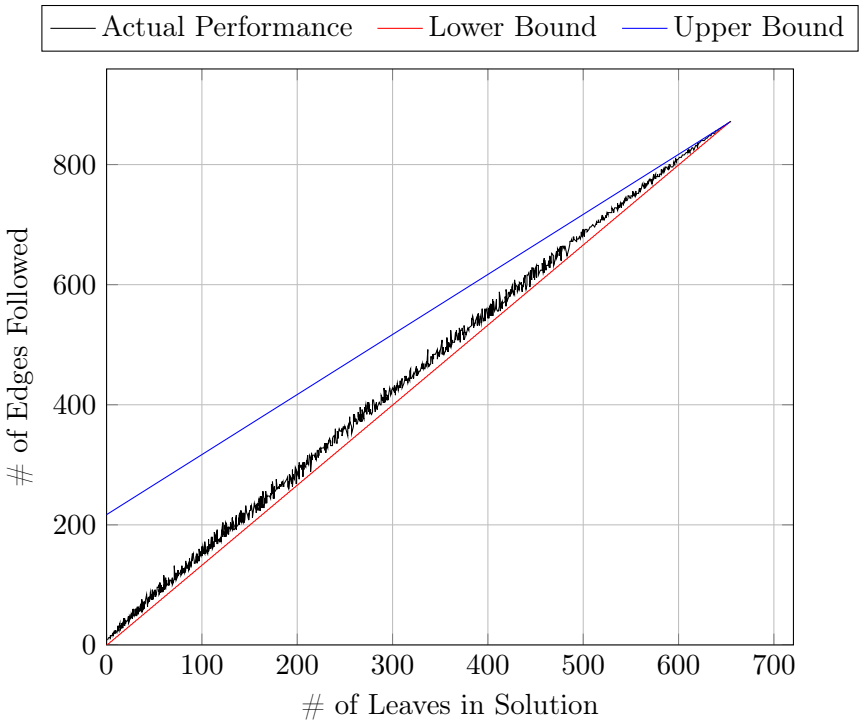


Figure 3.22: Actual vs. Expected Performance

Although it can be seen relatively clearly that the algorithm is performing as expected, plotting results on a log-log axis can be helpful in visualizing the results, especially when a very large solution size is possible. The same results are plotted on a log-log axis in Figure 3.23. Zooming into the higher end of the performance bounds of the log-log plot reveals that the algorithm has stayed within the expected bounds for every case, as shown in Figure 3.24.

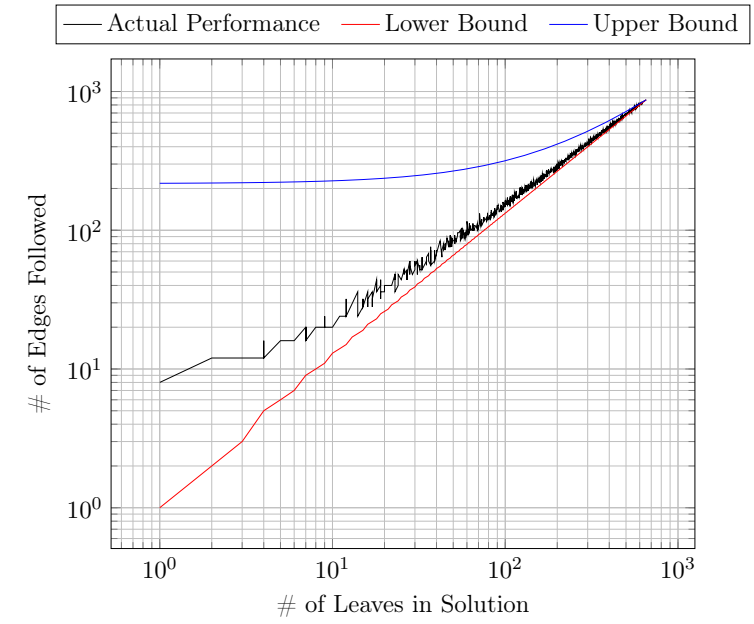


Figure 3.23: Actual vs. Expected Performance (Log Axes)

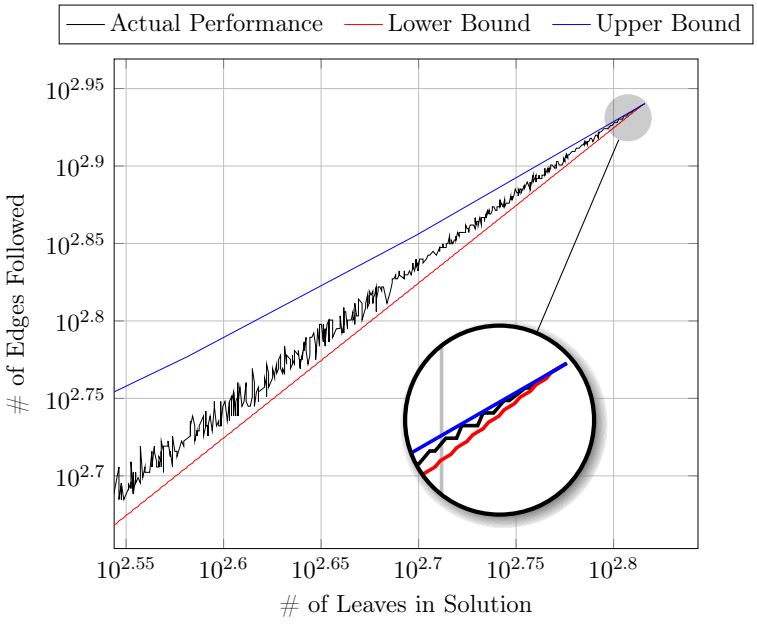


Figure 3.24: Actual vs. Expected Performance (Log Axes, Zoomed)

Additionally, it is clear that the results specific to the shape search implementation tend towards the lower bound of the expected performance. Recall that the upper bound on performance is limited by the case in which every branch is visited before finding any leaves. The shape search algorithm is implemented in a way such that if a branch is visited, at least one of the children of that branch will always be visited. As a result, the performance curve remains separated from the upper bound until visiting every branch is required to achieve the given solution size.

Timing Verification

The performance bounds presented are based on the assumptions that the only cost is traversing an edge and that the cost of traversing an edge is constant. In the real world, this will not be the case. However, if it is the case that traversing an edge is the *most expensive* operation, then it can be assumed that the general shape of the actual performance curve will follow the shape of the theoretical curve. The performance results plotting in Section 3.6 show that this assumption holds true for the shape search implementation of the algorithm.

3.4.3 Effect of Non-Full Leaves

The theoretical bounds presented are also based on the assumption that reaching a leaf adds a value of one to the size of the solution. This may not be true for every implementation of the algorithm, as is the case in the shape search algorithm. In the shape search, the size of the solution can increase by *at most* the bucket size of the quadtree. For example, if a quadtree has 250 nodes inside of a leaf, the algorithm may visit that leaf and determine that only 50 of the nodes fall within the circle, increasing the solution size by only 50. Therefore, it is possible that the algorithm can visit all branches *and* some nodes before reaching a given solution size. This results in a vertical shift of the upper bound.

This effect can be easily seen with an example problem that contains leaves with 1 or 0 nodes. A random search is performed over a mesh containing 3,060 nodes and 5,900 elements.

The quadtree containing the mesh data has a bucket size of 1, and there are 3,060 leaves that contain a node and 880 that do not. If the algorithm reaches a leaf that contains a node, the node is always added to the solution. The results of the test are plotted alongside the original expected upper bound as shown in Figure 3.25.

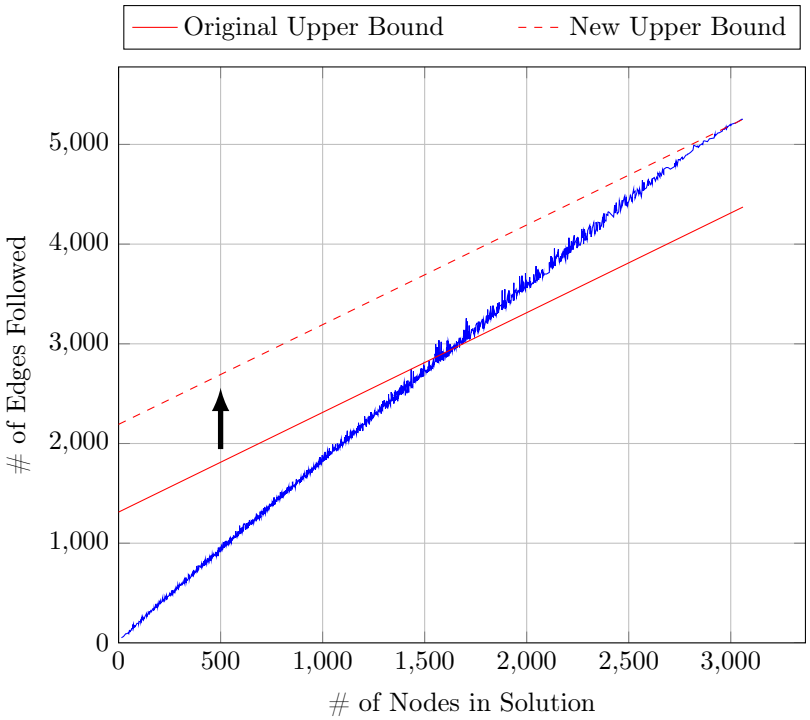


Figure 3.25: Effect of Non-Full Leaves on Performance

Because every leaf can contain a maximum of one node, the worst performance for every solution size will occur when the algorithm visits every branch and empty leaf before finding a leaf with a node. This results in a linear upward shift of the original upper bound, equal to the total number of empty leaves, which in this case is 880.

The shift experienced by the upper bound as a result of non-full leaves will not always be linear. As the bucket size changes and the total number of nodes within a leaf varies, so will the

rate at which the shift occurs.

3.5 Implementation

Consistent with the design goals, the C++ code used to implement the algorithm takes advantage of polymorphism in order to provide a simple codeset that can be easily extended by developers to fit their specific needs. The search algorithm functionality is provided as a virtual class called `QuadtreeSearch`. Subclasses of `QuadtreeSearch` are then made for each individual type of shape for which a search will be performed. Three virtual functions from the `QuadtreeSearch` parent class, corresponding to the required functions outlined in Section 3.3.3, are defined in each subclass. In addition, the point that falls on the edge of the shape is defined as a class variable in `QuadtreeSearch` and it must be set by the subclass before a search begins. This typically occurs when defining attributes of the shape, so this implementation defines a unique `SetShapeAttributes()` function in each subclass. The partial class diagram in Figure 3.26 shows these relationships using the four shapes provided in this implementation (helper functions and data types are hidden for clarity; the full source code for each of these classes can be found in the appendices).

Using this framework is beneficial from a development standpoint because the search algorithm code is only written a single time and the specific search types (i.e., shapes) are separated into smaller and more useful and portable classes. A drawback, however, is that the algorithm itself cannot be modified without changing the behavior of all subclasses. This prevents optimizations that could be made to the algorithm for specific classes of shape (such as convex shapes, as outlined in Section 3.7). However, all non-public functions are implemented as protected functions should a developer choose to rewrite any of the search algorithm components in order to accomodate for these optimizations.

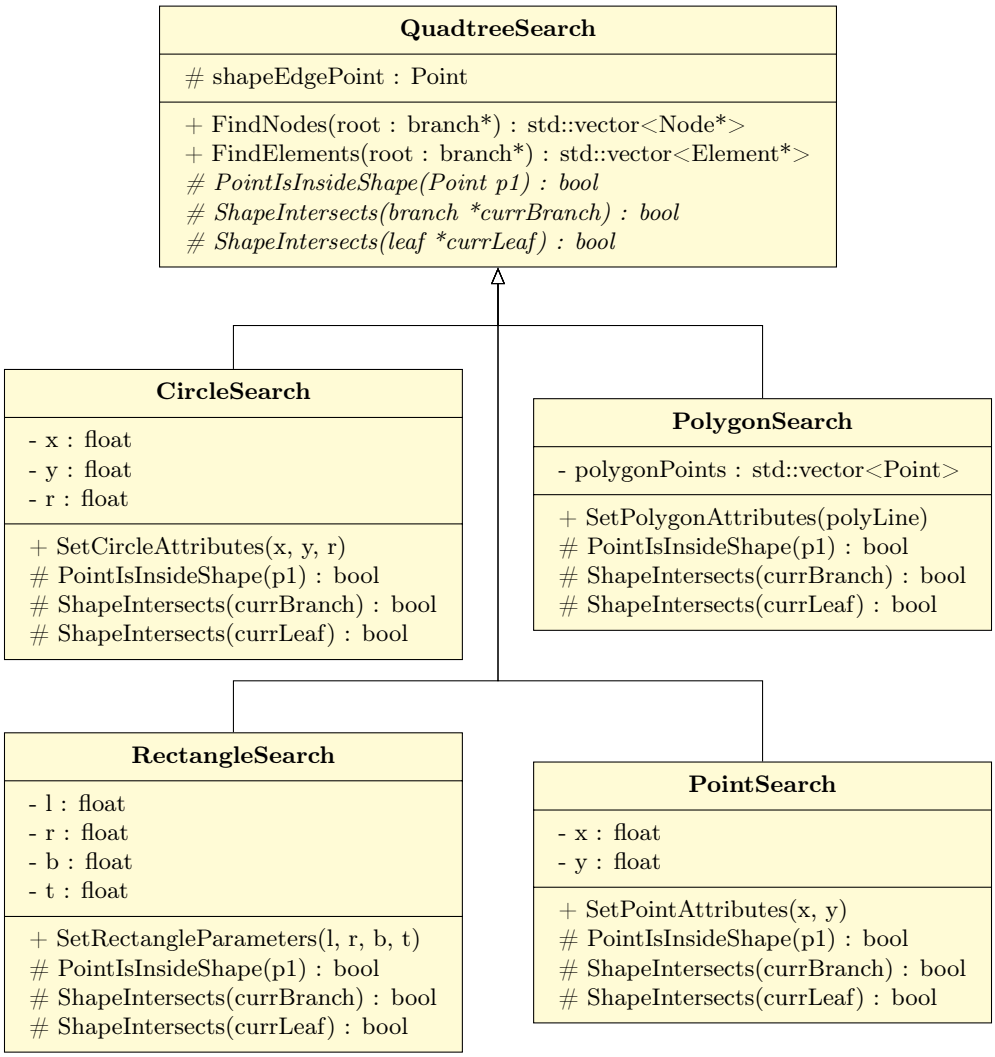


Figure 3.26: Partial Class Diagram of Shape Search Classes

3.5.1 The `QuadtreeSearch` Class

See Appendix B for source code

The `QuadtreeSearch` class is a virtual class that provides the functionality of the search algorithm. Specifically, it has only two public functions, called `FindNodes(branch*)` and `FindElements(branch*)`, that can be called to initiate a search for either a set of nodes or a set of elements, respectively. When either of these functions is called, the recursive search begins with a call to either `SearchNodes(branch*)` or `SearchElements(branch*)`, depending on what is being searched for. These private functions of the `QuadtreeSearch` class are the direct implementation of the search algorithm described in Section 3.3.3.

Three protected virtual functions, `PointIsInsideShape(Point)`, `ShapeIntersects(branch*)`, and `ShapeIntersects(leaf*)` are declared, making the class virtual. The purpose of these three virtual functions is outlined in Section 3.3.3, and they must be implemented by every subclass.

Finally, a protected class variable called `shapeEdgePoint` of type `Point`[1] is defined but never assigned a value. This is the point that falls on the edge of the shape, and it is up to the subclass to define it before beginning the search.

3.5.2 The `CircleSearch` Class

See Appendix C for source code

The `CircleSearch` class is a subclass of the `QuadtreeSearch` class that can search for nodes or elements that fall within a circle. In order to define the circle properties and set the value of the point that falls on the circle edge before a search begins, the `SetCircleParameters(float x, float y, float z)` function is called.

A circle used to perform a search in the `CircleSearch` class is defined by a center point with x- and y- coordinates and a radius. The `PointIsInsideShape` function uses the Pythagorean

[1] See Appendix A for data type definitions

Theorem to determine if a point falls inside of the circle. To avoid calculating square roots, which can be computationally costly, the following condition is tested:

$$(x - x_{center})^2 + (y - y_{center})^2 < r^2$$

where x_{center} and y_{center} are the x- and y- coordinates of the circle center, r is the circle radius, and x and y are the x- and y- coordinates of the point being tested. If this condition holds true, the point lies inside of the circle.

The following procedure is used in the `ShapeIntersects` functions to determine if the circle has an intersection with a branch or leaf. For each edge of the branch or leaf, determine the point on the infinite line created by the segment that is closest to the circle center by solving for u in the parameterized equation of a line:

$$\mathbf{P_1} + u\,(\mathbf{P_2} - \mathbf{P_1})$$

where $\mathbf{P_1}$ and $\mathbf{P_2}$ are the endpoints of the edge. If this value is in the range $[0, 1]$, then the point falls on the line segment and it is the point on the segment that is closest to the circle center. Otherwise, one of the end points of the line segment is the point closest to the circle center, so the two distances are computed in order to determine which is closest. Finally, it can be concluded that if the distance from the circle center to the point on the line segment that is closest to the circle center is less than the circle radius, the circle and line segment intersect.

3.5.3 The `RectangleSearch` Class

See Appendix D for source code

The `RectangleSearch` class is a subclass of the `QuadtreeSearch` class that can search for nodes or elements that fall within an axis-aligned rectangle. In order to define the rectangle properties and set the value of the point that falls on the edge of the rectangle before a search begins, the `SetRectangleParameters(float l, float r, float b, float t)` function is called.

A rectangle used to perform a search in the `RectangleSearch` class must be axis-aligned and is defined by four bounding values, the left, right, bottom, and top bounds. The `PointIsInsideShape` function simply compares the points with the bounds in order to determine if it falls inside of the shape. If the following is true, the point is inside of the rectangle:

$$[l \leq x \leq r] \text{ and } [b \leq y \leq t]$$

where l, r, b, and t correspond to the left, right, bottom, and top bounds, and x and y are the coordinates of the point.

Because branches, leaves, and the rectangle are all axis aligned, an initial test is performed using simple logic for the `ShapeIntersects` functions. If the right side of the rectangle is to the left of the left bound of the branch/leaf, no intersections are possible, and if the left side of the rectangle is to the right of the right bound of the branch/leaf, no intersections are possible. Similarly, if the top of the rectangle is below the lower bound of the branch/leaf, no intersections are possible, and if the bottom of the rectangle is above the upper bound of the branch/leaf, no intersections are possible. If all of these conditions are met, no possible edge-edge intersections are possible. However, two additional cases must be considered. If the rectangle is fully inside of the branch/leaf or if the branch/leaf is fully inside of the rectangle, the logic determines that there is an intersection. Because of the ordering of the algorithm, this is not desirable and these cases must indicate that there is no intersection for the algorithm to work as expected. As such, a reordering of the algorithm to test for these cases before testing for intersections could produce some performance benefits, and is discussed further in Section 3.7.

3.5.4 The `PolygonSearch` Class

See Appendix E for source code

The `PolygonSearch` class is a subclass of the `QuadtreeSearch` class that can search for nodes or elements that fall within a polygon with three or more edges. A polygon used in the polygon

search is defined as a list of three or more points in which every pair of adjacent points defines an edge of the polygon. The `SetPolygonParameters(std::vector<Point> polyLine)` function is called before a search is performed in order to define the polygon and set the value of the point that falls on the edge of the polygon.

There are a number of existing algorithms for determining if a point falls inside of a polygon, most of which rely on the *even-odd rule* or *winding number* concepts [22]. This implementation uses the PNPOLY algorithm developed by Franklin [23] for the `PointIsInsideShape` function because it is compact (only 7 lines) and performs very well [22].

In order to determine if the polygon intersects with a branch or leaf for the `ShapeIntersects` functions, each segment of the polygon is tested against each edge of the branch or leaf. This test is performed by determining if the quadrilateral formed by the four end points is one in which the vertices alternate between the two segments. Specifically, the two line segments $\overline{ab}$ and $\overline{cd}$ intersect if and only if only one of the two triples a, b, c and a, b, d is in counterclockwise order [24], as illustrated in Figure 3.27.

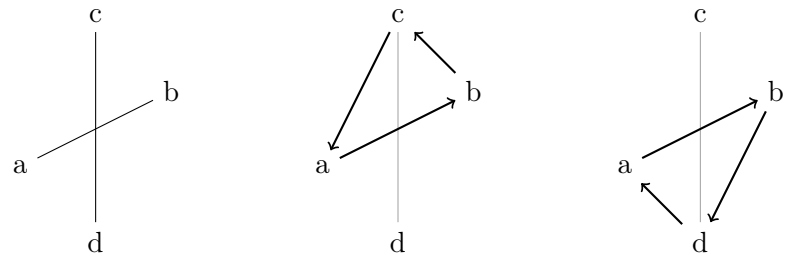


Figure 3.27: Orientations Being Determined for Two Line Segments

3.5.5 The `PointSearch` Class

See Appendix F for source code

The `PointSearch` class has a slightly different functionality from the rest of the shape classes, and therefore requires that a few functions the search algorithm uses for tests be slightly modified. Nonetheless, the algorithm itself remains the same.

The goal when performing a node search in this class is to find the node that is closest to a provided point. This is a well-known and heavily researched problem, typically called the *nearest neighbor search*, for which there are a number of very efficient solutions. Many variations of the solution can perform the search in $O(\log n)$ time, including implementations that use Voronoi diagrams [20] and others that use k-d trees [25, 26]. These solutions typically require their own preprocessing so they are not ideal considering our space has already been partitioned into a quadtree. Other fast solutions exist that avoid preprocessing by simply "walking through" the mesh [27]. However, using the algorithm described in this thesis is both simple and provides adequate performance.

The goal when performing an element search in this class is to find the element that contains the provided point. This is the same problem being solved in the `PolygonSearch` class but because this search is looking for an element, which is always a triangle, a simpler test can be used. In this implementation, the barycentric coordinates of the provided point are calculated in order to determine if the point falls within the triangle created by the element.

Searching for both nodes and elements is performed the same way. Because a single point is provided, the algorithm only needs to find the single leaf that contains the point. Once that leaf has been found, a specialized exhaustive search is performed on the nodes or elements in that leaf, depending on what is being searched for.

The `PointInsideShape` function must always return false, and the `ShapeIntersects` functions return true or false depending on if the point falls inside the bounds of the branch or leaf. When the search algorithm is run, this has the effect of finding the leaf that contains the point and storing all of the nodes or elements from that leaf in the list of nodes

or elements that might fall inside of the shape.

The nodes and elements in this list are then searched using a specialized search that determines either the closest node or the element that contains the point. The node search calculates the distance between the point and every node, adding the closest node to the list of nodes that are guaranteed to fall within the shape. This list, which contains only a single node, is returned as the solution. The element search steps through the list of candidates, and once it finds an element that contains the point, adds that element to the list of elements that are guaranteed to fall within the shape and stops searching. This list, which contains only a single element, is returned as the solution.

3.6 Performance Results

The following sections demonstrate the performance achieved using the algorithm defined in Section 3.3.3 for various shapes.

3.6.1 Comparison with Exhaustive Search

In this section, the performance of the search algorithm is compared against the performance of the exhaustive search for each shape in which every node or element in the mesh is tested to determine whether or not if falls within the provided shape. This comparison is made in order to show the algorithm performance using a number of different shapes. For each shape, it may be possible to create a search algorithm that performs better than the algorithm presented, but the goal here is to demonstrate consistent performance across a variety of shapes. To our knowledge, there are no other algorithms in publication that search for the set of points inside of an arbitrary shape, and so exhaustive search is the only alternative with which a valid comparison can be made.

The system on which all tests in this thesis are performed has an Intel Core i7 (2.8 GHz) CPU with 8 GB of RAM running Ubuntu 12.04. Each shape is tested using an ADCIRC mesh with approximately 5 million nodes and 10 million elements, and a quadtree bucket size of 250

nodes is used. 1000 random variations of the shape are generated and the search algorithm and exhaustive search methods are tested using each variation of the shape for both the node and element search. Each individual search is timed and the results are plotted. The x-axis contains the number of nodes or elements found in any given search (i.e., the solution size), and the y-axis contains the time in seconds required to find the given solution size. Results are plotted on log-log axes in order to more effectively show performance results.

Circle Search

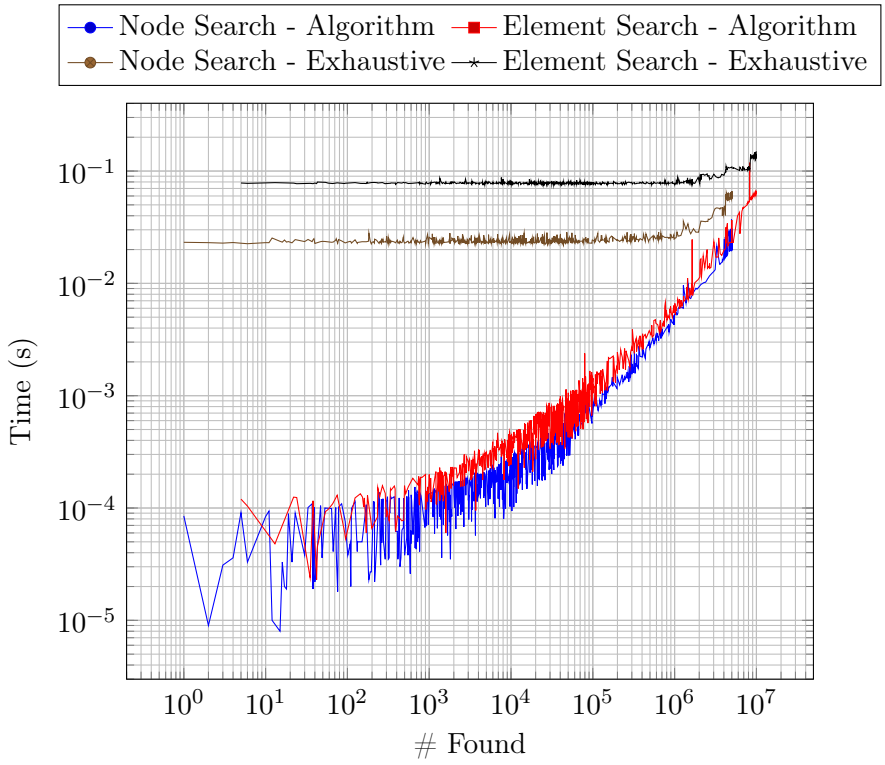


Figure 3.28: Circle Search Performance Results

As expected, the results plotted in Figure 3.28 show that the algorithm performs very well when a small number of nodes or elements are being searched for. Additionally, it can be seen that the

exhaustive search always takes approximately the same amount of time to complete a search, independent of the number of nodes or elements being searched for. As the number of nodes or elements being searched for increases, the performance of the algorithm decreases, approaching the performance of the exhaustive search. These results closely mirror the predicted performance curve outlined in Section 3.4.

Rectangle Search

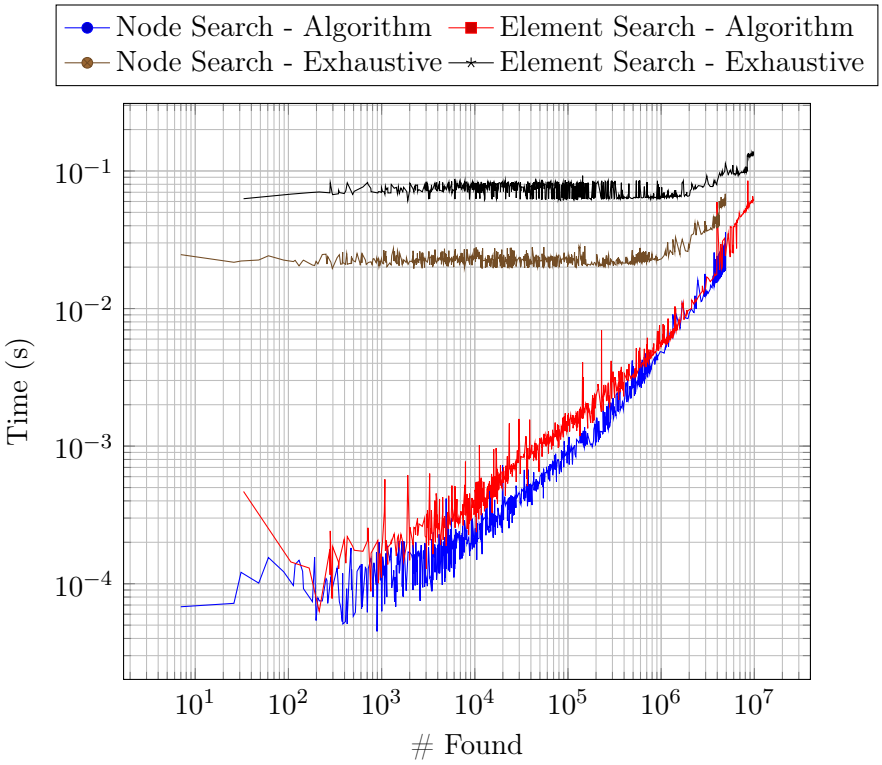


Figure 3.29: Rectangle Search Performance Results

The rectangle search performs in a similar manner to the circle search. The results plotted in Figure 3.29 show that the search completes very quickly when searching for a small number of nodes or elements but performance approaches the level achieved by the exhaustive search as

the number of nodes or elements being searched for increases. While the results also mirror the expected performance outlined in Section 3.4, the shape of the performance curve is slightly different from that of the circle search. This difference can be attributed to variations in the complexity of the test functions supplied with each of the shapes.

Polygon Search

The polygon performance benchmarking is performed using two different polygon sizes, one containing 10 edges and one containing 30 edges. The results of the tests performed using a 10-edge polygon are plotted in Figure 3.30 and the results of the tests performed using a 30-edge polygon are plotted in Figure 3.31.

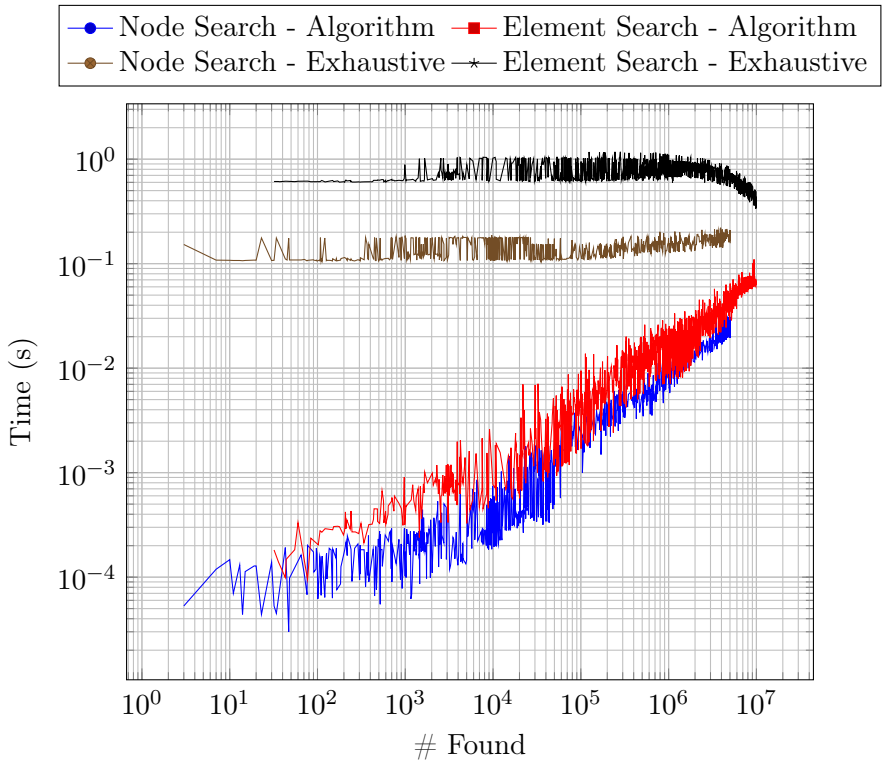


Figure 3.30: 10-Edge Polygon Search Performance Results

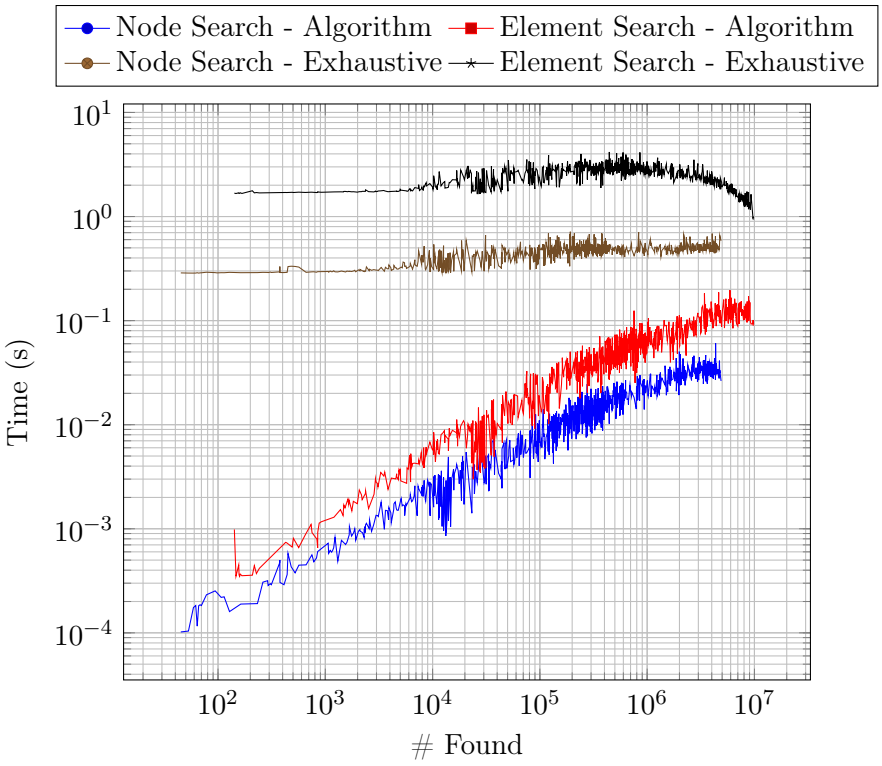


Figure 3.31: 30-Edge Polygon Search Performance Results

Again, in both cases the algorithm performs very well when a small number of nodes or elements is selected, with performance decreasing as the number of nodes or elements selected increases. The performance of the algorithm in the 30-edge case decreases much more rapidly than in the 10-edge case. This is due to the increased cost of performing tests on a 30-edge polygon over a 10-edge polygon.

Point Search

The point search is performed slightly differently than in the testing of the other shapes. In this test, every individual node or element is searched for specifically by using either the exact coordinates (in the case of a node search) or the center point (in the case of an element search) as the point being searched for. The exhaustive search is performed by iterating through a list

that contains all nodes or elements that is sorted by node or element number (as defined in the `fort.14` file). In the exhaustive node search, the distance between the point and all nodes is calculated and the closest node is chosen. In the element search, a test is performed to determine if the point falls inside of each element, starting at the beginning of the list containing the elements and proceeding down the list until the appropriate element is found. Once the element is found, the search is terminated. The results of both searches are plotted in Figure 3.32 with the node search results on the left and the element search results on the right.

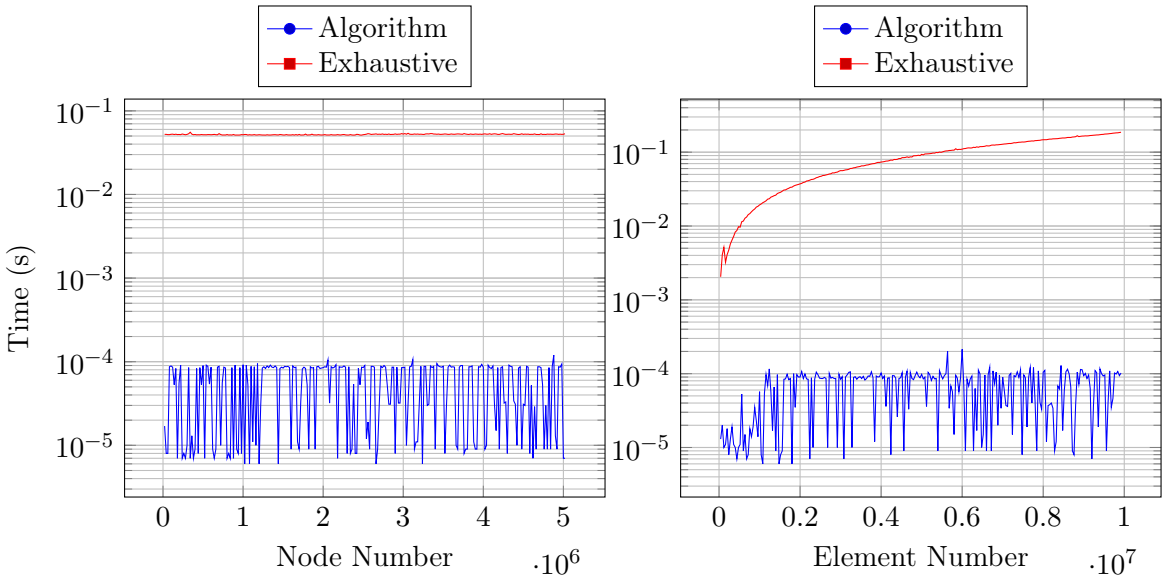


Figure 3.32: Point Search Performance Results

The x-axis in this plot represents the node or element number, and is therefore a good graphical representation of the entire ordered list of nodes or elements. Only every 20,000$^{\text{th}}$ node or element is plotted to increase clarity.

3.6.2 Analysis of Comparison with Exhaustive Search

Recall from Section 2.1 that the target completion time for a node or element search is less than 100 milliseconds (10^{-1} seconds). The completion time for the exhaustive search in both the circle search and rectangle search are both very close to this limit and in both polygon searches, this limit is exceeded. The exhaustive search is exceptionally slow in the polygon element search with some completion times exceeding 3 seconds. The completion time for the search algorithm in all cases, however, is far below this limit for a small number of nodes or elements, only approaching this limit, and rarely surpassing it, as the solution size approaches the number of nodes or elements in the entire domain. This is an acceptable tradeoff in the context of subdomain modeling because typically only smaller numbers of nodes and elements need to be found.

The decrease in performance corresponding to an increase in the number of nodes or elements being searched for in all cases except the point search is a result of using a quadtree to store the mesh data. As the search shape increases in size, the number of branches and leaves that can be completely excluded from the search decreases, resulting in a decreased performance caused by recursion, as described in Section 3.4. Additionally, the shape of the curves produced by these tests closely mirrors the expected shape of the expected upper and lower bounds on performance outlined that section. Differences between the actual and expected performance in each test case, as well as differences between the individual test cases, can be attributed to additional computational overhead. Every time a branch or leaf is recursed upon, there is overhead associated with determining the position of that branch or leaf in relation to the shape. The magnitude of this additional cost depends on the shape being tested. For very complex shapes, the overhead associated with these tests can become very large. Variations in performance can also come from the overhead associated with building the two lists used in the algorithm. Copies of the nodes or elements contained in the leaves are put into these lists in order to maintain the integrity of the quadtree data for future searches. The lists are implemented as C++ standard library vectors and the copying is done using range operations,

which is considered best practice [28]. Nonetheless, there is still overhead, and the accumulative cost of this overhead increases as the total number of nodes or elements included in the search increases. Finally, as the size of the shape increases, the number of nodes or elements that need to be tested using exhaustive search also increases, which can produce a significant reduction in performance, especially if the point inside shape test is costly.

The performance benefit in both the node and element point search cases is clear. The exhaustive node search takes constant time because every node has to be tested in each search. The algorithm node search also performs an exhaustive search at the completion of recursion, taking approximately constant time, but the number of nodes being tested is the quadtree bucket size instead of the mesh size. The exhaustive element search is linear in the distance of the element from the beginning of the list, as can be seen in the plot. The algorithm element search on the other hand, performs in almost constant time based on the quadtree bucket size.

3.6.3 Effect of Bin Size on Performance

The effect of the quadtree bin size on search performance is tested using four different bin sizes in the node circle search over an ADCIRC mesh with approximately 5 million nodes and 10 million elements. For each bin size, 500 random circles are generated and the search algorithm is timed. The results are plotted in Figure 3.33 with the number of nodes found in the x-axis and the time required to complete the search on the y-axis. Two plots of the same data are provided, one with logarithmic axes and one with linear axes.

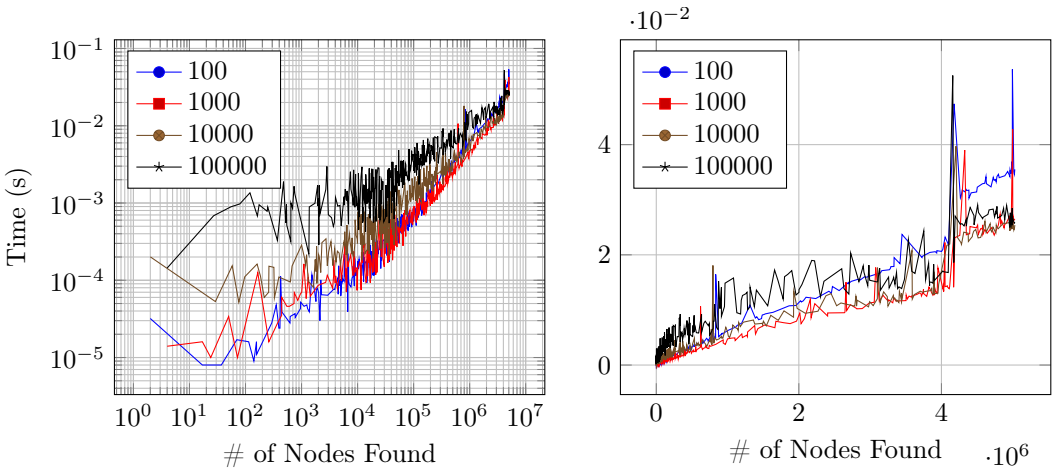


Figure 3.33: Comparison of Search Performance Using Varying Bin Sizes

As the plot shows, the time required for the algorithm to complete is dependent on both the bin size and the solution size. When the solution size is very small, a smaller bin size is preferable. This can be seen in the logarithmic plot where the bin size of 100 performs better than the bin size of 1,000, the bin size of 1,000 performs better than the bin size of 10,000, and the bin size of 10,000 performs better than the bin size of 100,000. The reduced performance in increasing bin sizes aligns with the theory developed in Section 3.4.3 that more non-full leaves in the quadtree will result in an increased upper bound on performance.

On the other hand, as the solution size increases, the larger bin sizes begin to perform better and the smaller bin sizes perform worse, as can be seen in the linear plot. This effect can be attributed to the cost of performing a recursion and list expansions. When very large bin sizes are used, fewer recursions and fewer list expansions are needed to reach a large solution size, as opposed to when very small bin sizes are used, in which case a large number of recursions and list expansions are required in order to reach the same solution size.

Because ADCIRC subdomain modeling typically involves selecting very small numbers of nodes or elements from a mesh, a smaller bin size seems preferable. However, there is a tradeoff that occurs with the preprocessing time required to build the quadtree that contains the mesh

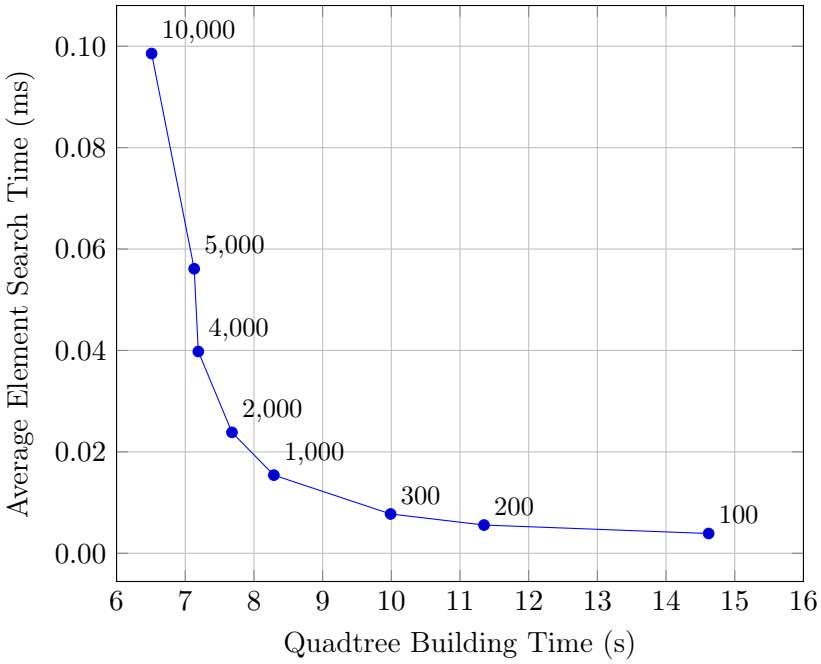


Figure 3.34: Comparison of Quadtree Build Time With Average Search Time

data.

To demonstrate this tradeoff, Figure 3.34 shows the average time it takes to perform a single element search compared with the time it takes to build the quadtree on which the search is being performed. In order to create this plot, eight different quadtrees were created from the same 5 million node, 10 million element mesh. In each of these quadtrees, 10,000 individual element searches were performed, and an average search time was determined by dividing the total time it took to perform the searches by 10,000. In the figure, each data point represents a different quadtree size (as noted by the value in black next to each data point).

The tradeoff between preprocessing of mesh data and real-time mesh data selection is clearly seen in Figure 3.34, and one can see how the choice of quadtree bin size could vary between applications. From the user's perspective in an application such as SMT, it may be benefitial to remove a few seconds of loading time at the expense of a few milliseconds of responsiveness during element selection (which could go unnoticed by the user), whereas if a user will be

programatically performing millions of point searches over a quadtree, a few extra seconds of preprocessing may be benefitial in reducing overall run time.

The assembly of the quadtree is a highly recursive process. Nodes are added to leaves one at a time, and when the number of nodes in a leaf exceeds the bin size, the leaf is converted into a branch, which requires that every node in the leaf be re-added to the new branch. A smaller bin size results in a larger number of leaf to branch conversions, which is a time consuming process. Table 3.2 and Figure 3.35 show the time required to build five different quadtrees for the 5 million node, 10 million element mesh. As outlined in Section 2.1, a maximum time of 10 seconds is used as a benchmark for unit tasks. Assembling the quadtree typically is included in the process of opening an ADCIRC mesh, which also includes reading the `fort.14` file. Therefore, a quadtree size needs to be chosen, based on the largest expected mesh size, such that the time required to read the data from file and assemble it into a quadtree is less than 10 seconds.

Table 3.2: Assembly Time For Various Bin Sizes

Bin Size	Time (s)
10	433.750
100	14.620
1000	8.290
10000	6.510
100000	5.410

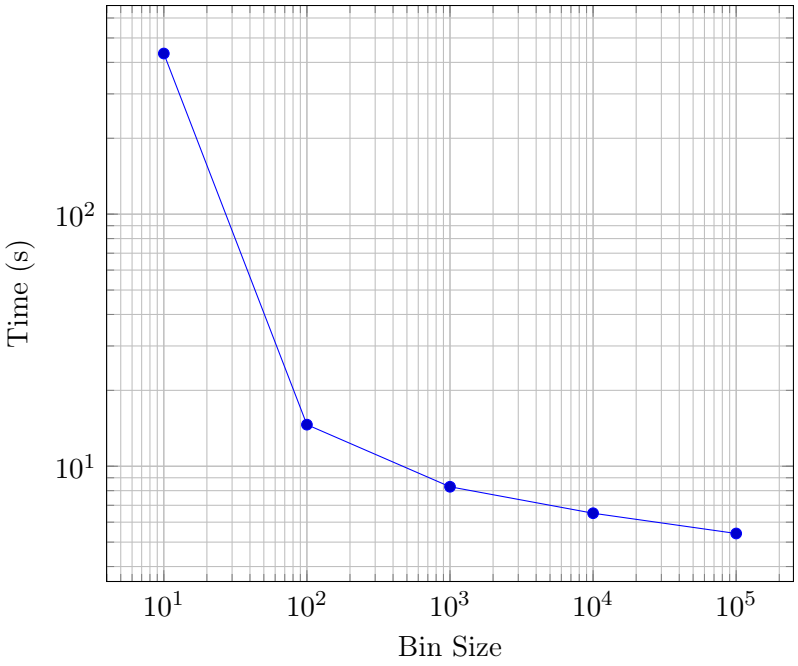


Figure 3.35: Assembly Time For Various Bin Sizes

3.7 Optimization

Optimizing the algorithm typically involves attempting to reduce the amount of overhead caused by the operations performed within the algorithm itself in order to force the performance curve to approach the theoretical best performance curve.

Convex Shapes

This optimization attempts to alleviate some of the calculation overhead by removing some of the general applicability of the algorithm, allowing a search to be performed only on convex shapes. The logic behind the algorithm remains essentially untouched, but modifying certain minor attributes could produce performance benefits for specific classes of shapes. This is achieved by reordering the tests that are performed to determine the location of a branch or leaf in relation to the shape such that more expensive tests come later in the algorithm, with the intent that

fewer of them need to be performed.

The optimization of the algorithm for convex shapes relies on the fact that if all four corners of a leaf or branch (which is rectangular) fall inside of the shape, there can be no edge-edge intersections and the entire leaf or branch is guaranteed to fall inside of the shape. This means that the first test performed in the algorithm can be to determine if the leaf or branch falls inside of the shape, which for many convex shapes is a much cheaper operation than finding edge-edge intersections.

Next, if the number of corners of the leaf or branch inside of the convex shape is equal to one, two or three, then an edge-edge intersection is guaranteed. Again, this removes potentially costly operations. If none of the corners of the leaf or branch are inside of the convex shape, a test is performed to determine if the entire shape falls inside of the branch or leaf. If it does, the algorithm recurses and performs an early exit once that recursion completes. Otherwise, a final edge-edge intersection test is performed. If there is an edge-edge intersection, the recursion process continues, otherise the branch or leaf is entirely outside of the shape.

This modified version of the algorithm is outlined in Algorithm 3.6 and Algorithm 3.7.

Algorithm 3.6: Search Algorithm for Convex Shapes (branch)

```
1  Algorithm SearchNodes(Branch currBranch)
2      cornerInShapeCount = 0;
3      if shape.pointIsInside(currBranch.NorthEast) then
4          cornerInShapeCount += 1;
5      if shape.pointIsInside(currBranch.SouthEast) then
6          cornerInShapeCount += 1;
7      if shape.pointIsInside(currBranch.NorthWest) then
8          cornerInShapeCount += 1;
9      if shape.pointIsInside(currBranch.SouthWest) then
10         cornerInShapeCount += 1;
11     if cornerInShapeCount == 4 then
12         AddAll(currBranch, finalNodes);
13     else if cornerInShapeCount != 0 then
14         for subBranch in currBranch do
15             SearchNodes(subBranch);
           end
16         for subLeaf in currBranch do
17             SearchNodes(subLeaf);
           end
18     else if currBranch.contains(shape.testPoint) then
19         for subBranch in currBranch do
20             SearchNodes(subBranch);
           end
21         for subLeaf in currBranch do
22             SearchNodes(subLeaf);
           end
23         EarlyExit();
24     else if shape.intersects(currLeaf) then
25         for subBranch in currBranch do
26             SearchNodes(subBranch);
           end
27         for subLeaf in currBranch do
28             SearchNodes(subLeaf);
           end
       end
```

Algorithm 3.7: Search Algorithm for Convex Shapes (leaf)

```
1  Algorithm SearchNodes(Branch currLeaf)
2      cornerInShapeCount = 0;
3      if shape.pointIsInside(currLeaf.NorthEast) then
4          cornerInShapeCount += 1;
5      if shape.pointIsInside(currLeaf.SouthEast) then
6          cornerInShapeCount += 1;
7      if shape.pointIsInside(currLeaf.NorthWest) then
8          cornerInShapeCount += 1;
9      if shape.pointIsInside(currLeaf.SouthWest) then
10         cornerInShapeCount += 1;
11     if cornerInShapeCount == 4 then
12         AddAll(currLeaf, finalNodes);
13     else if cornerInShapeCount != 0 then
14         AddAll(currLeaf, partialNodes);
15     else if currLeaf.contains(shape.testPoint) then
16         AddAll(currLeaf, partialNodes);
17         EarlyExit();
18     else if shape.intersects(currLeaf) then
19         AddAll(currLeaf, partialNodes);
       end
```

The modified code used to benchmark this method can be found in Appendix G. It is implemented as a subclass of the `CircleSearch` class only for the purposes of this test. An implementation requiring general applicability to all convex shapes should be subclassed from the `QuadtreeSearch` class. Benchmarking of the modified algorithm is performed on the same 5 million node, 10 million element ADCIRC mesh. One thousand random circles are generated and the time required to find the elements is measured and plotted in Figure 3.36.

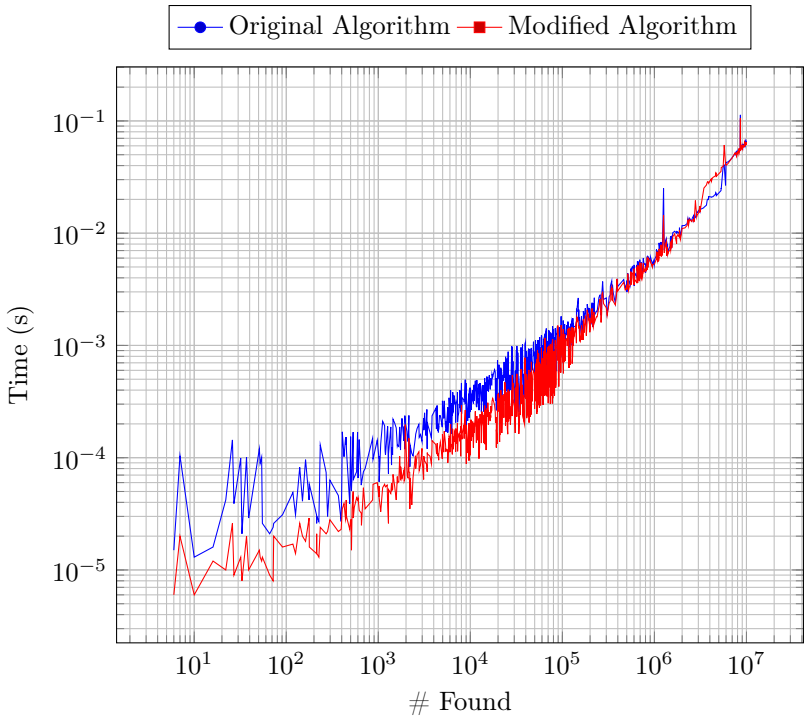


Figure 3.36: Performance of Algorithm Modified for Convex Shapes

In general, it appears that the modified version of the code performs better than the original. The magnitude of this performance benefit in this case is fairly small, so for the purposes of this implementation the specialized code is not needed. However, the modified algorithm may be desirable for convex shapes that have very expensive edge-edge intersection tests.

Chapter 4

Case Studies

The following two case studies aim to present SMT by demonstrating how it is used to perform the most basic functions that are essential to subdomain modeling.

4.1 Quarter Annular

The Quarter Annular case study is used to present the user interface layout and to demonstrate the workflow of the features of SMT that have been implemented at the time of this writing. This includes creating a project, creating subdomains, and running both full and subdomains in ADCIRC. Editing subdomains and analyzing results are both features that continue to undergo development.

Upon starting the SMT program, the user is presented with the window in Figure 4.1. The main window itself is split into two main regions. The region on the left contains the work modules (Project Explorer, Create New Subdomain, Edit Subdomain, ADCIRC, and Analyze Results) which are ordered from top to bottom based on the order in which they typically will be used. Above the Project Explorer is a toolbar consisting of six icons.

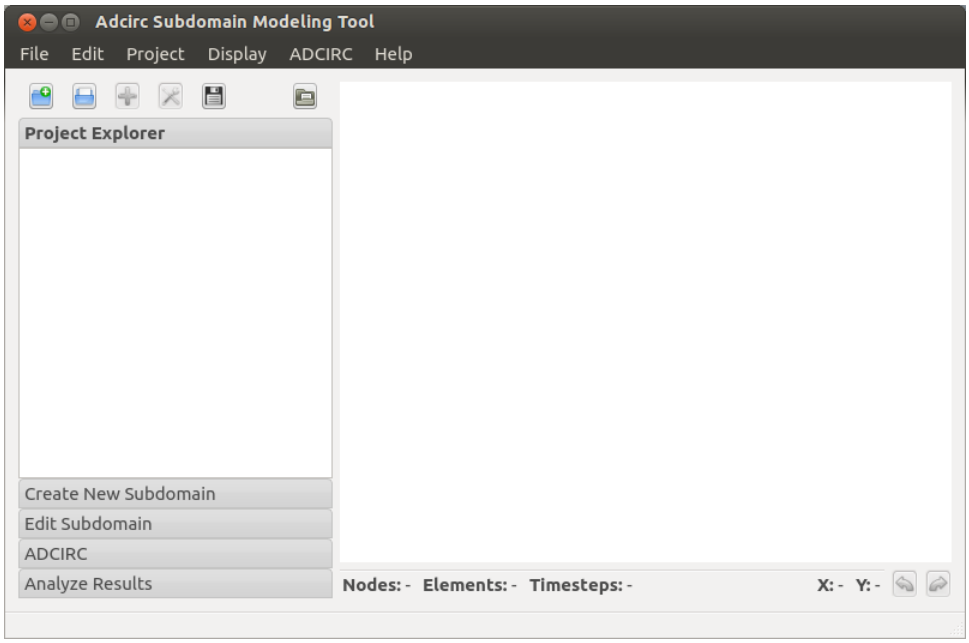


Figure 4.1: The Initial SMT Window

These icons, from left to right, are:

- **New Project** - Create a new project.
- **Open Project** - Open an existing project.
- **Add File** - Add a file to the current project.
- **Edit Preferences** - Open the project preferences window.
- **Save Project** - Save the current project.
- **Open File** - Open a file for viewing (the file will not be added to the current project). If the file is an ADCIRC mesh file, it will be displayed as a mesh in the display window.

The region on the right contains the display window. This is where the user will interact with an ADCIRC mesh. Below the display window is a toolbar that displays the properties of the currently visible mesh (number of nodes, number of elements, number of timesteps if the

domain has been run) on the left. On the right, the toolbar shows the current coordinates of the mouse pointer in the coordinate system of the currently visible mesh, as well as an undo and a redo button.

The top of the main window contains a menu bar with File, Edit, Project, Display, ADCIRC, and Help menus available. These menus contain all of the same functionality of the modules and tool buttons already presented as well as additional supplemental features. The entire process outlined in this case study can be accomplished without accessing any of these menus, so they will not be referenced.

To create a new project, the **New Project** menu button is pressed, presenting the dialog in Figure 4.2. In this dialog, the user chooses a name for the new project, as well as the directory in

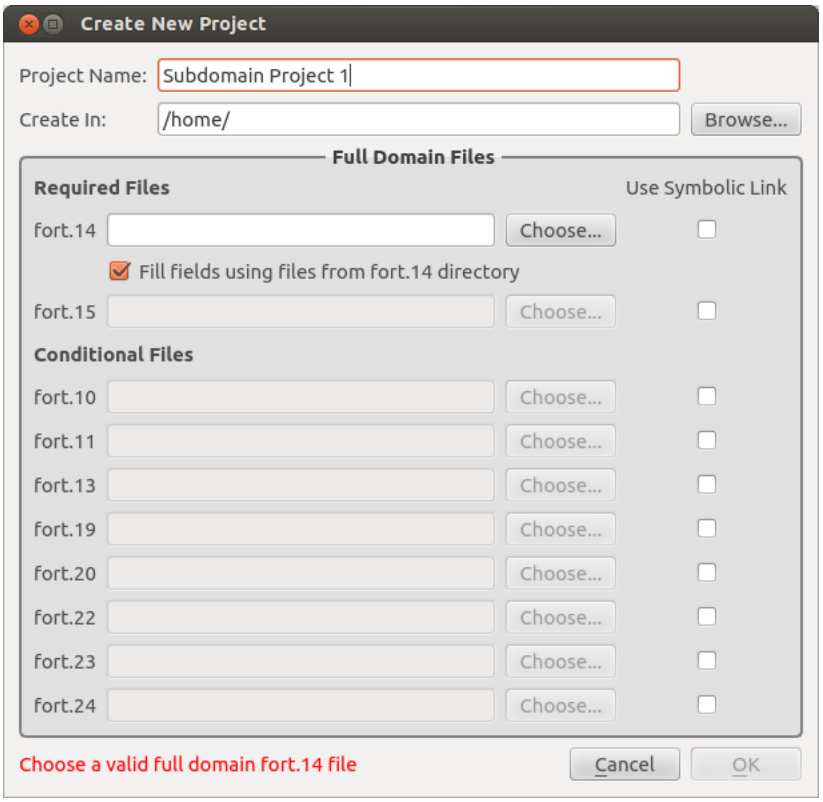


Figure 4.2: The New Project Dialog

which it will be created. Because a single full domain is associated with each SMT project, the required full domain files must be chosen. The only two files required to make a subdomain are full domain `fort.14` and `fort.15` files. If the "Fill fields using files from fort.14 directory" option is selected, the dialog will automatically populate the rest of the fields with the appropriate files, if they exist in the same directory as the chosen `fort.14` file, otherwise the user is free to choose the files individually. The user also has the option of using symbolic links, which can be helpful in reducing redundant data in the file system, especially when using large files, such as ADCIRC meteorological files. If a symbolic link is not used, the file will be copied into the new project directory.

Upon creating a new project, which in this case study has been named "Quarter Annular", the project is displayed as shown in Figure 4.3. The display window now contains an interactive

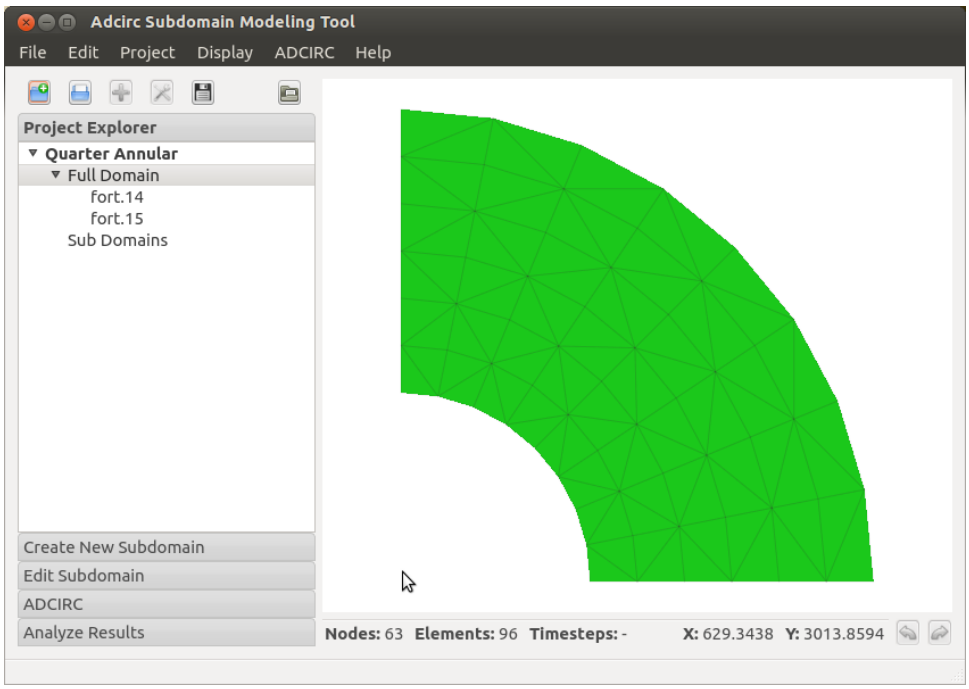


Figure 4.3: The New Quarter Annular Project

image of the full domain. The user can zoom by scrolling and pan by left-clicking and dragging. Additionally, the toolbar below the display window and the Project Explorer module have been populated with new data. The toolbar shows that the full domain contains 63 nodes and 96 elements, and the mouse pointer is located at the coordinates (629.3438, 3013.8594). The Project Explorer shows the tree layout of the current state of the project. This data will always be presented as shown in the tree in Figure 4.4. The current project does not have any

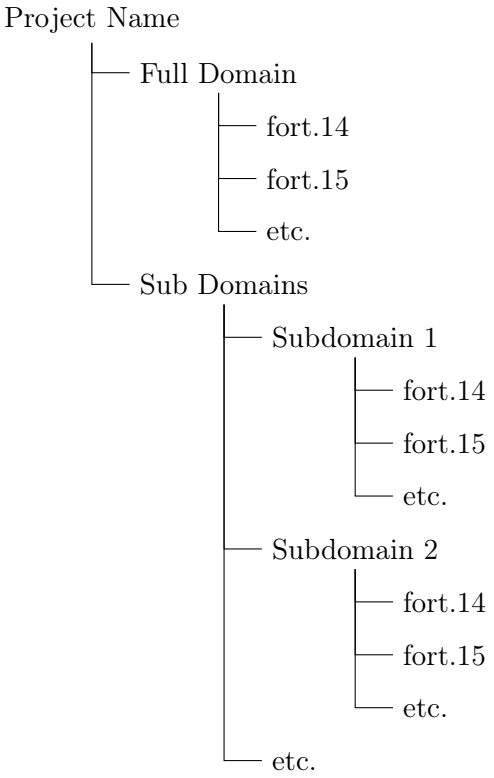


Figure 4.4: The Project Layout

subdomains (because none has been created yet), so the "Sub Domains" branch of the tree contains no data. Next, in order to create a new subdomain, the user clicks on the "Create New Subdomain" module, displaying the module as shown in Figure 4.5.

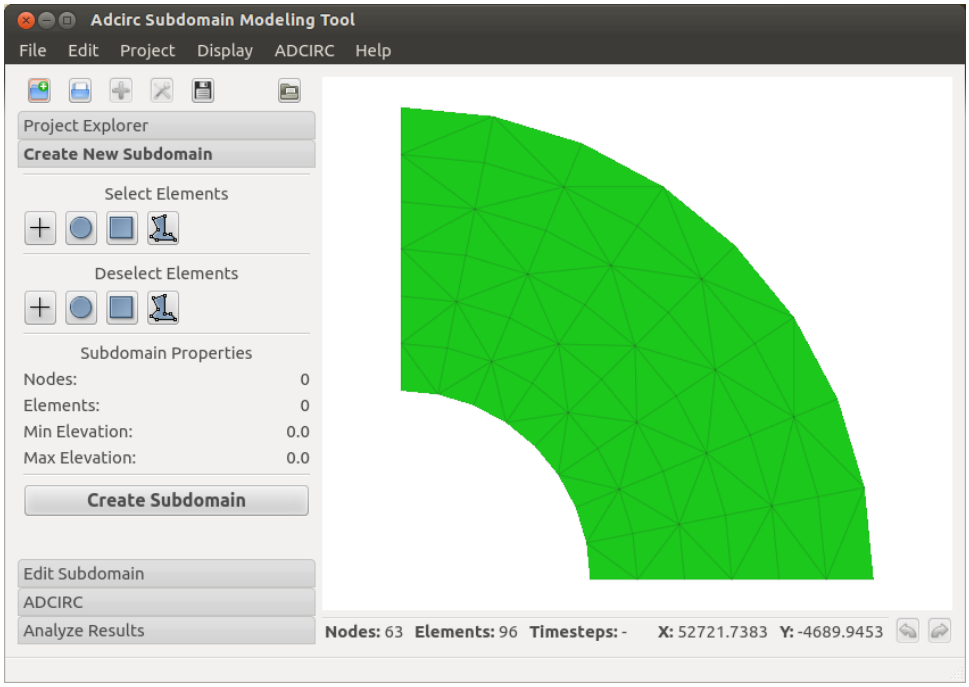


Figure 4.5: The Create New Subdomain Module

The "Create New Subdomain" module presents the user with four tools that can be used to either select or deselect elements:

-  - Select or deselect individual elements
-  - Select or deselect elements by drawing a circle
-  - Select or deselect elements by drawing a square
-  - Select or deselect elements by drawing a polygon

Elements for this subdomain will be selected using the polygon tool. When the polygon tool is selected, every left click on the display window will drop a vertex of the polygon. A double-click will drop the final vertex and select the elements that fall inside of the polygon, as shown in Figures 4.6 and 4.7.

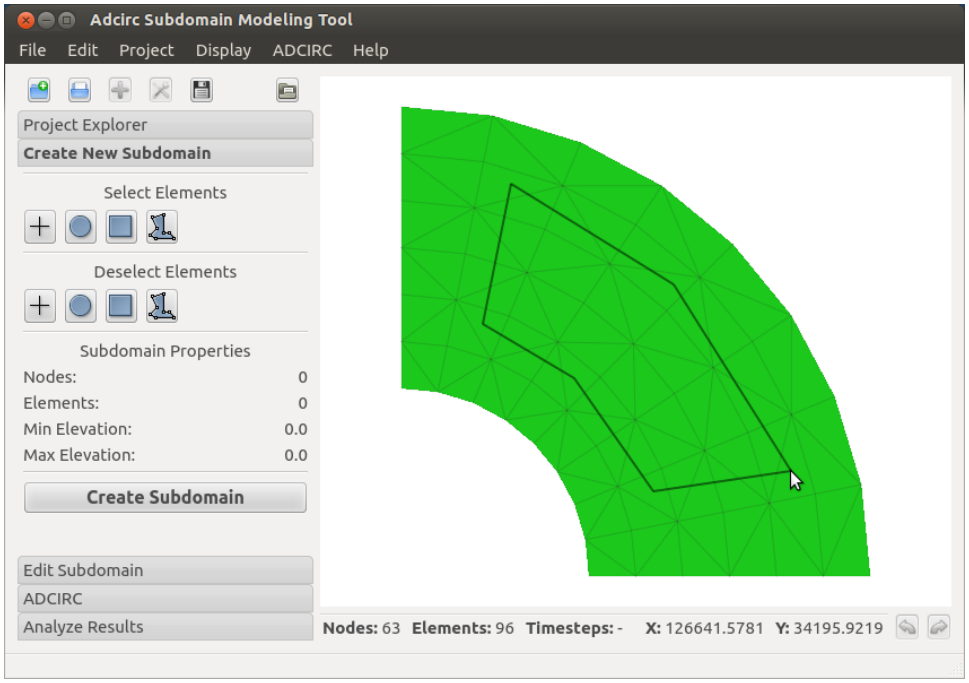


Figure 4.6: Selecting a Polygon in the Quarter Annular Project

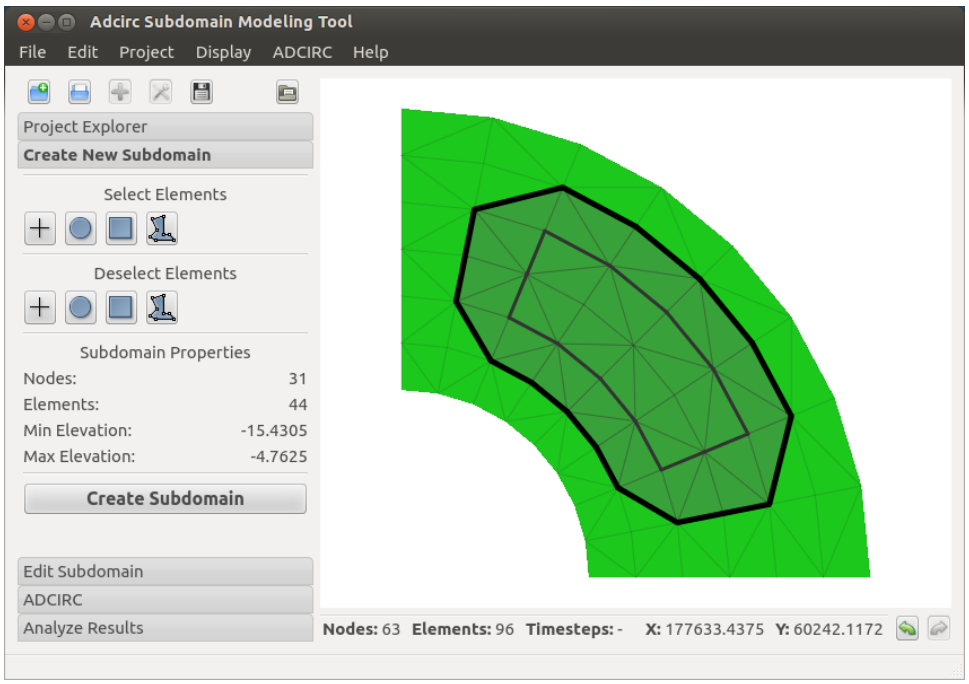


Figure 4.7: The Elements Selected Using the Polygon Tool

Every element that has at least one node inside of the polygon is now selected and the outer and inner boundaries of the subdomain are displayed as bold lines. Notice that the properties of the subdomain are now listed below the selection tool buttons. These properties are updated as elements are selected and deselected by the user. Additionally, the undo button below the display window is now clickable and clicking it will undo the previous selection or deselection as well as activate the redo button.

Now that the appropriate elements have been selected, the user presses the "Create Subdomain" button in the "Create New Subdomain" module, displaying the dialog shown in Figure 4.8. In this dialog, the user chooses a display name for the subdomain, the directory in which

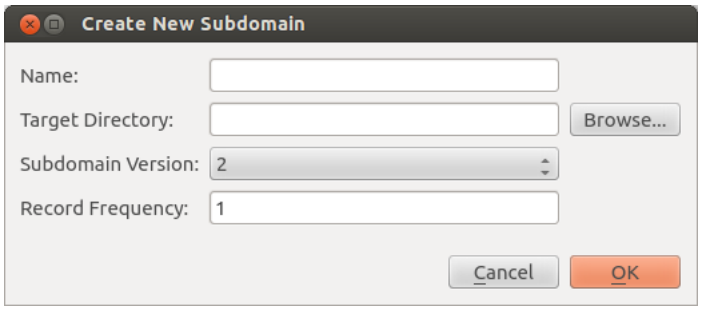


Figure 4.8: The Create New Subdomain Dialog

to store the subdomain, the version of the subdomain method to use (a second, newer version of the subdomain method which produces more accurate results is being developed alongside SMT [29]), and the recording frequency (see thesis by Altuntas [2] for a definition). Note that the subdomain version and record frequency will only be options for the first subdomain created in a project. This is because these values must match for every full domain - subdomain pair in the project. Once these fields have been populated, the user presses the OK button to create the new subdomain. In this case, the name chosen for the subdomain is "Polygon". When the new subdomain is created, it will automatically be selected in the Project Explorer and displayed in the display window, as seen in Figure 4.9. At this point, the user is free to either run the full

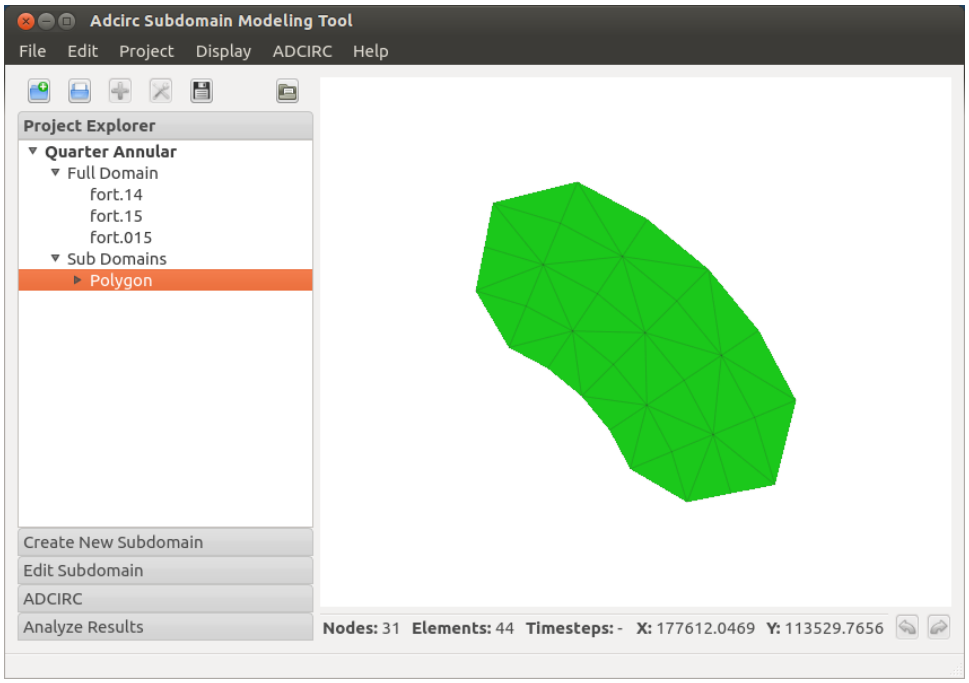


Figure 4.9: The Polygon Subdomain of the Quarter Annular Project

domain to record boundary conditions, edit the subdomain to make localized changes (a feature currently under development), or create more subdomains.

In order to run the full domain, the user clicks on the "ADCIRC" module, displaying the module as shown in Figure 4.10. The "ADCIRC" module contains two buttons and a list. Only the "Run Full Domain" button will be active until the full domain has successfully completed. To run the full domain, press the "Run Full Domain" button to display the dialog shown in Figure 4.11. The user chooses the location of the ADCIRC executable and whether they would like to open a separate shell window in which to run ADCIRC (this allows the user to continue working in or to exit SMT), or to run ADCIRC directly from SMT (which prevents SMT from performing any actions until the run is complete, and is not recommended). Clicking OK begins the full domain run.

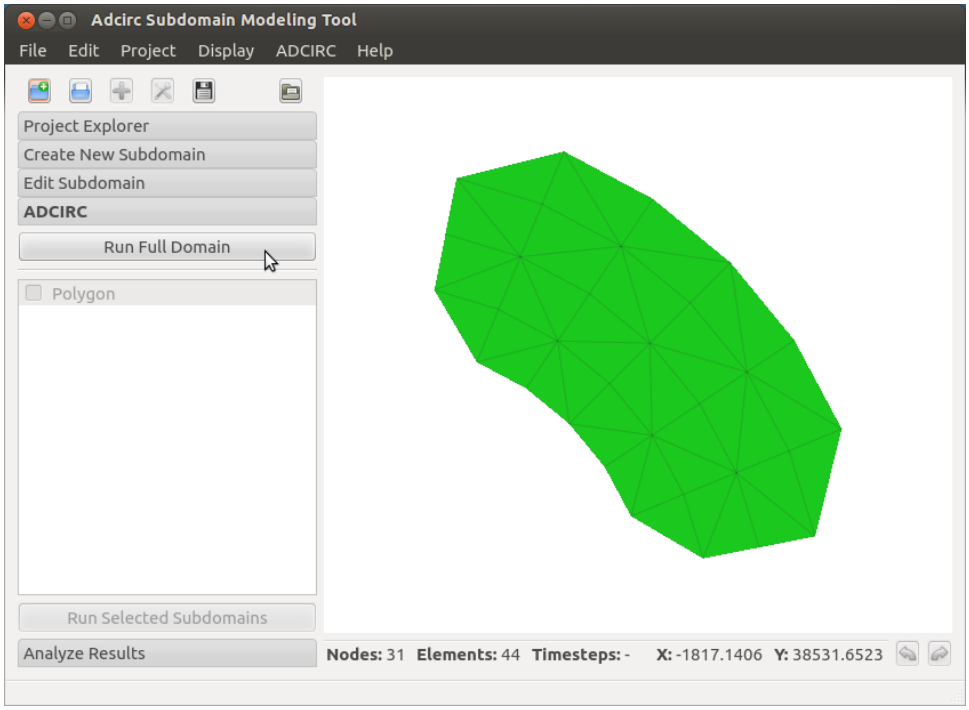


Figure 4.10: The ADCIRC Module

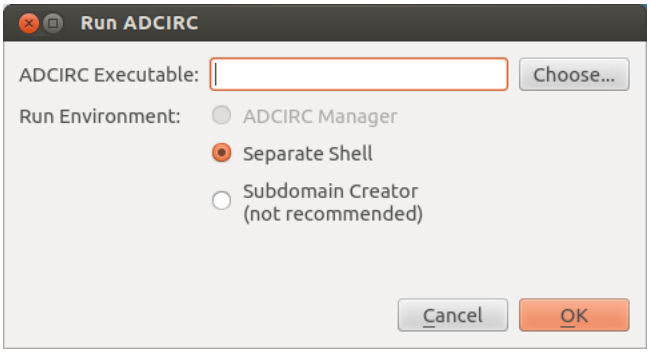


Figure 4.11: The Run ADCIRC Dialog

Once the full domain run is complete, the list of subdomains in the project and the "Run Selected Subdomains" button will become active, as seen in Figure 4.12. From this point, running any number of subdomains is as simple as selecting them from the list and clicking the "Run Selected Subdomains" button. The Run ADCIRC dialog will open again, providing the same options as for the full domain run.

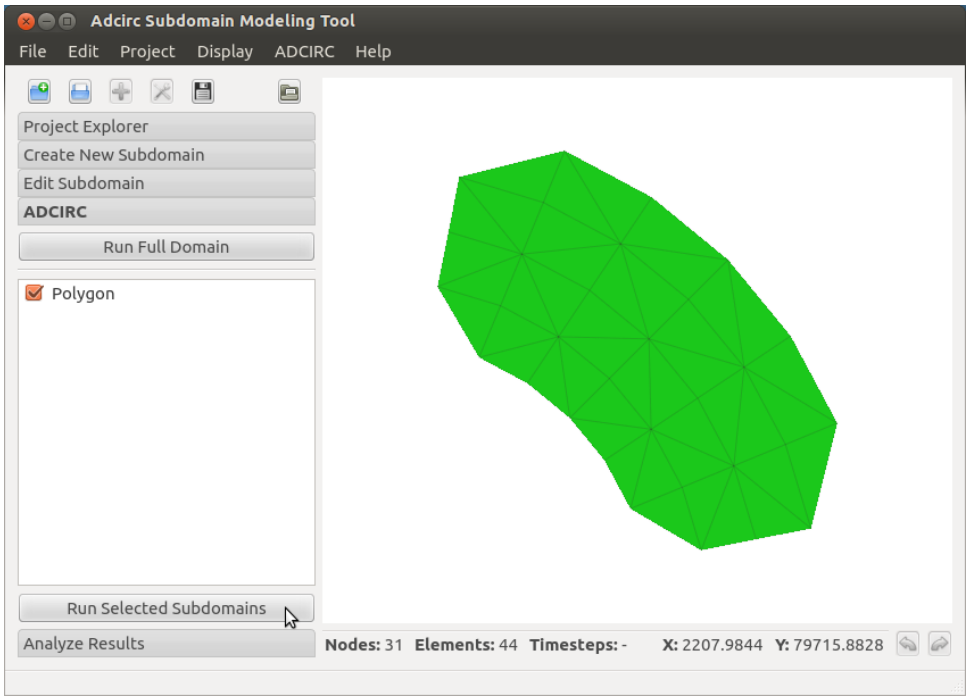


Figure 4.12: Preparing to Run the Quarter Annular Project Subdomains

4.2 Coastal North Carolina

The Coastal North Carolina case studied is used to demonstrate how multiple subdomains are managed within an SMT project. Starting with an existing project that already contains multiple subdomains, a new subdomain for a region of interest is created, and the full and multiple subdomain ADCIRC runs are completed.

First, the user opens SMT and clicks the "Open Project" button. A file dialog is displayed and the user navigates to the directory containing the North Carolina Coast project, as demonstrated in Figure 4.13. This directory contains the SMT project file and all of the directories that contain

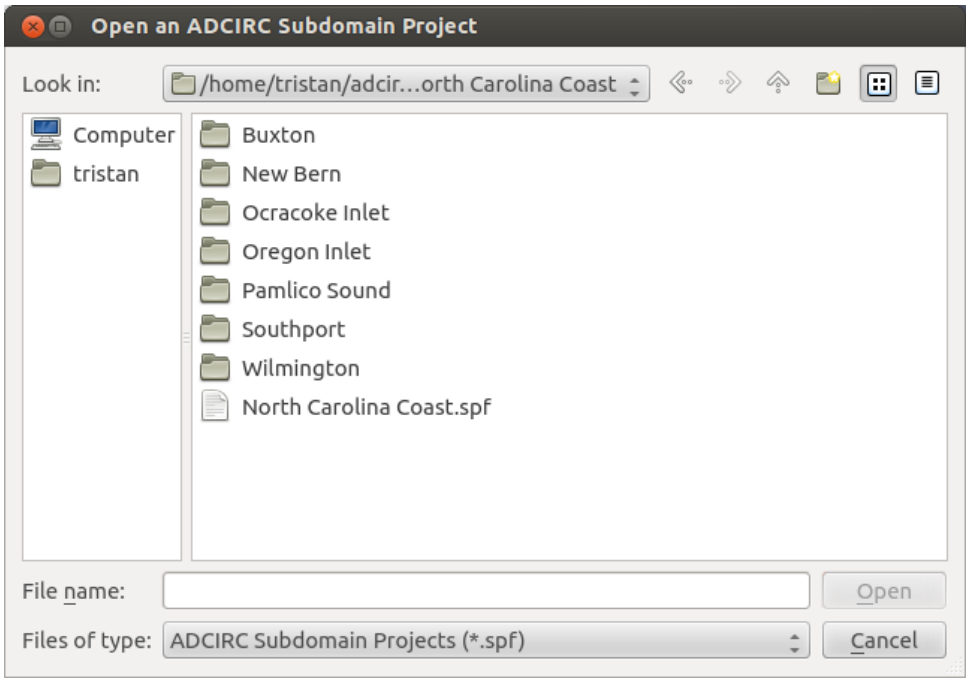


Figure 4.13: The Open ADCIRC Project Dialog

the subdomains in this project. Note that the project file extension is `.spf`, for *subdomain project file*. The full domain ADCIRC files (`fort.14`, `fort.15`, etc.) are also in this directory, but are hidden from view in this dialog.

Opening the `North Carolina Coast.spf` file opens the project in SMT. Upon opening the project, the full domain is displayed and the project contents are listed in the Project Explorer, as shown in Figure 4.14. This project currently contains seven subdomains, any of which can be viewed by clicking on the subdomain name in the Project Explorer. For example, the Figure 4.15 shows the view of the Oregon Inlet subdomain.

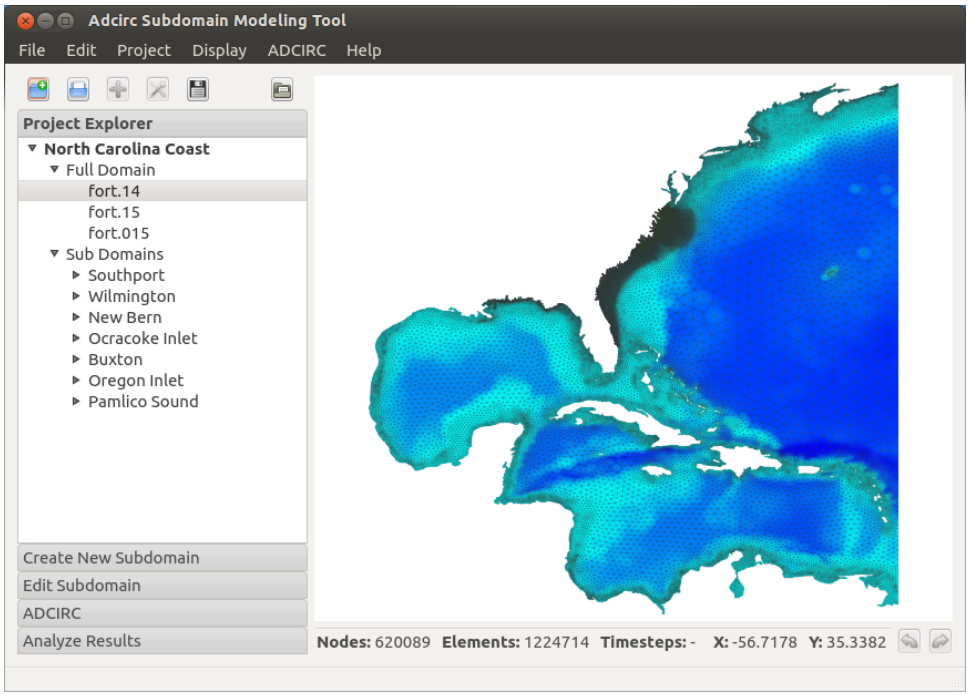


Figure 4.14: The North Carolina Coast Project Opened

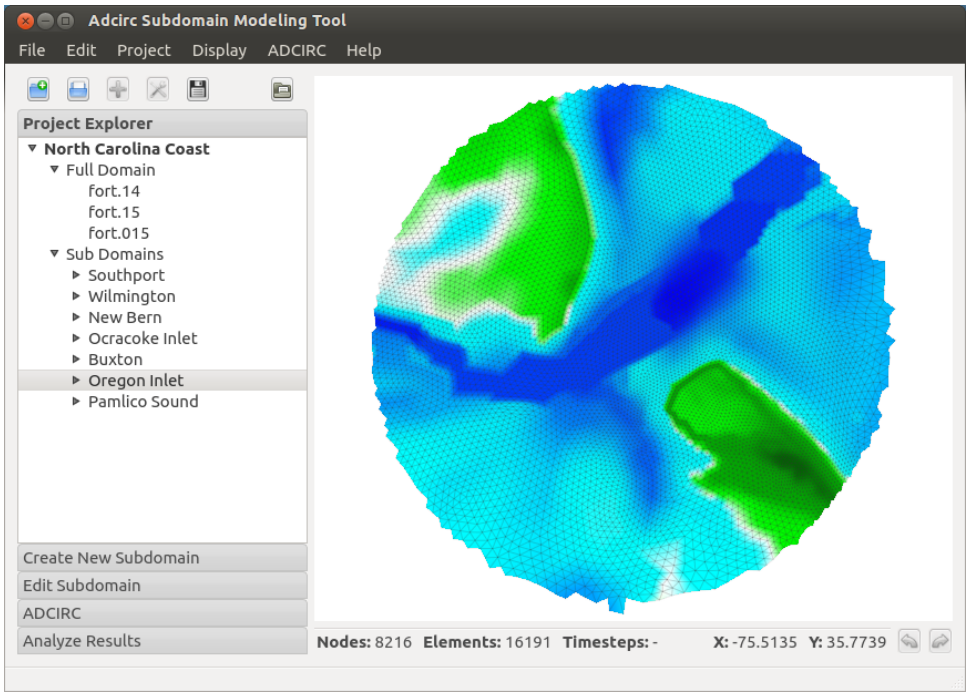


Figure 4.15: Viewing the Oregon Inlet Subdomain

In this project, the user has already defined specific color options in order to display the mesh in a more usable manner. Choosing Display > Display Options... will open the display options window shown in Figure 4.16. This options window allows the user to interactively choose how

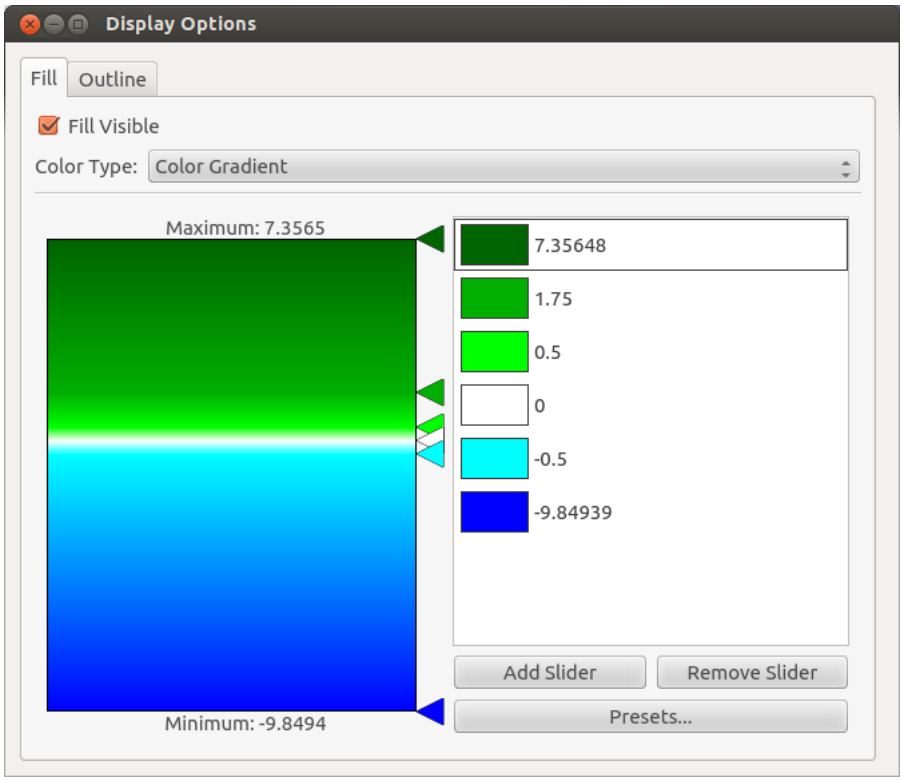


Figure 4.16: The Display Options Window

to color the mesh, meaning that as options are changed, the mesh display is updated accordingly. There are two tabs at the top of the window, allowing the user to choose color options for the fill or the outline of elements. Currently, the user may choose either a solid color or a color gradient that is based on the mesh elevation values. This user has chosen a color gradient. The gradient is chosen by adding sliders, each of which has an elevation value and a color. Any number of sliders can be used and elevation values can be changed by either moving the sliders themselves up and down or by double-clicking the slider in the list on the right to edit the value.

The user now wishes to create a new subdomain around Camp Lejeune, which is located in Jacksonville, North Carolina, on the north coast of the New River Inlet as seen in Figure 4.17. The user displays the full domain by clicking on 'Full Domain' in the Project Explorer and then

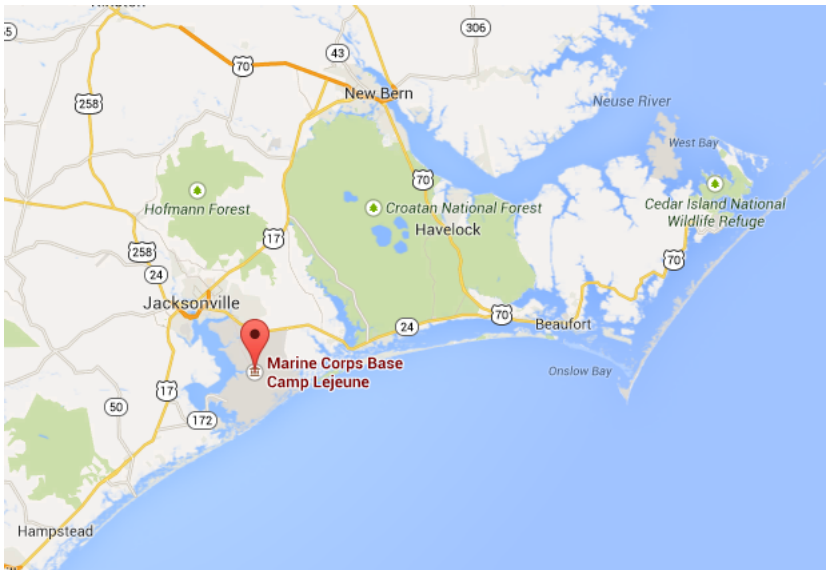


Figure 4.17: Map of Camp Lejeune

zooms into the area of interest, as shown in Figure 4.18.

The user is only interested in the results on the eastern coast of the New River Inlet, so they select the subdomain using the polygon tool, only including the areas of interest, as shown in Figure 4.19. Once satisfied with the size and coverage of the subdomain, the user clicks the Create Subdomain button to create the new subdomain, which will be called "Camp Lejeune".

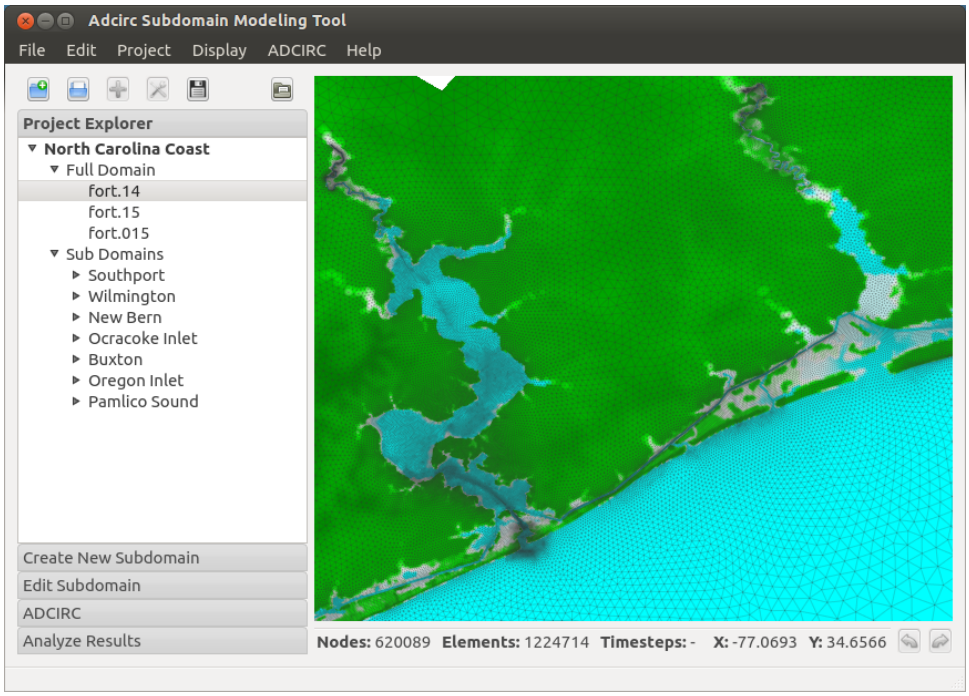


Figure 4.18: Camp Lejeune in the Full Domain

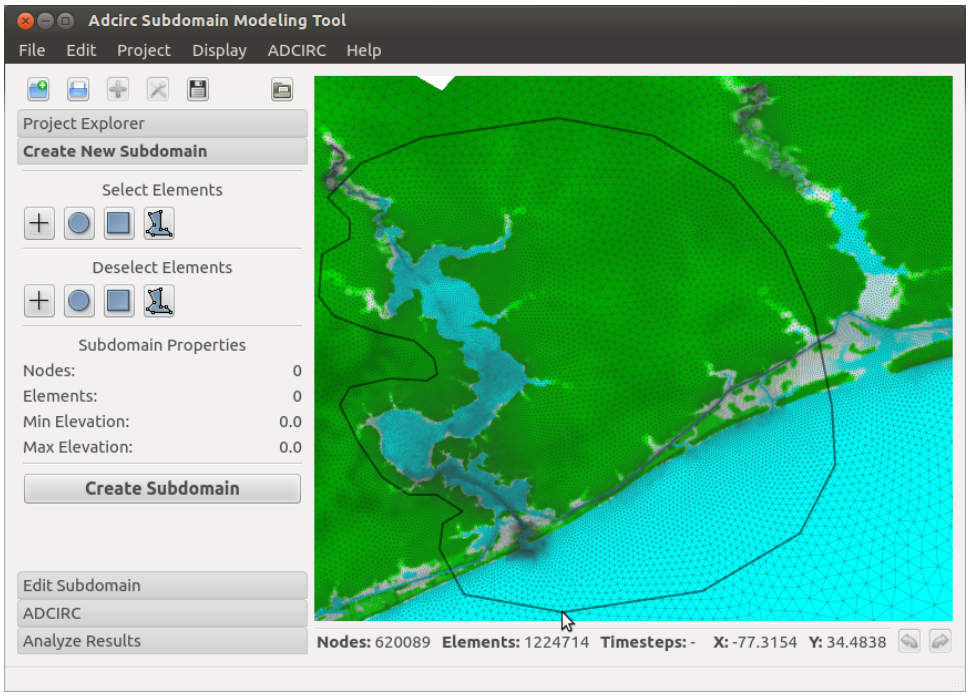


Figure 4.19: Selecting the Camp Lejeune Subdomain

Once the subdomain is created, the Project Explorer will be updated and the new mesh will be displayed, as shown in Figure 4.20.

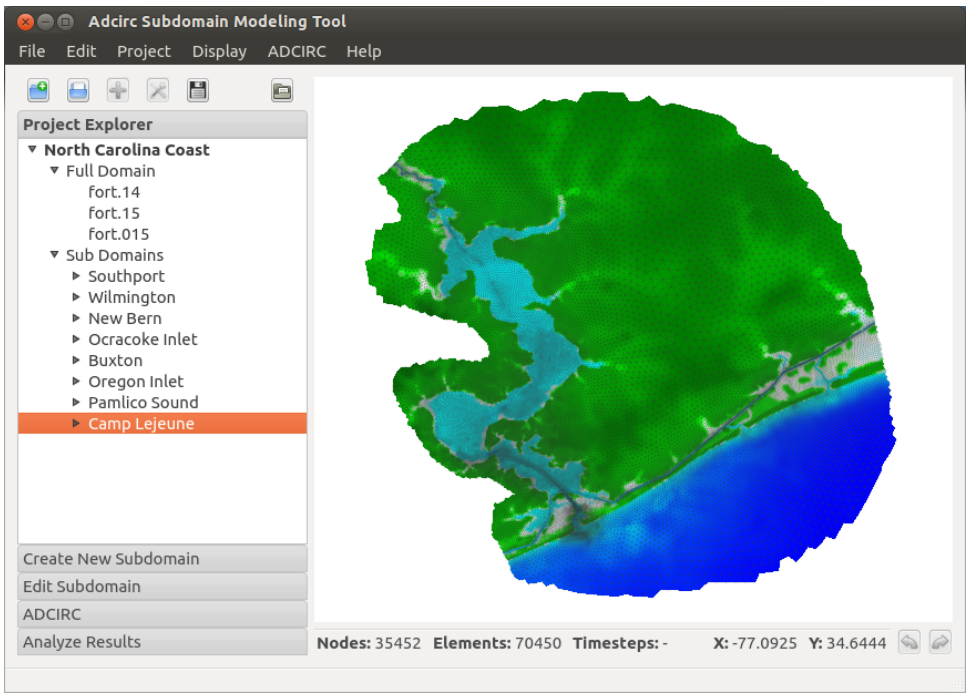


Figure 4.20: The Camp Lejeune Subdomain

Now that the user has all of the needed subdomain for this project, they run the full domain. Because the full domain is very large, this is done on a multi-node computing cluster. All of the necessary files have already been created, so this is as simple as copying the files from the full domain directory to the compute cluster and performing an ADCIRC run as normal. Once the full domain run has completed, the subdomain output files need to be copied back into the local full domain directory. Once this has been done, all of the subdomains are ready to be run, as shown in Figure 4.21.

The user only wishes to run the Southport, Wilmington, and Camp Lejeune subdomains at this time, so they select them from the list in the "ADCIRC" module and click the "Run Selected

Subdomains" button.

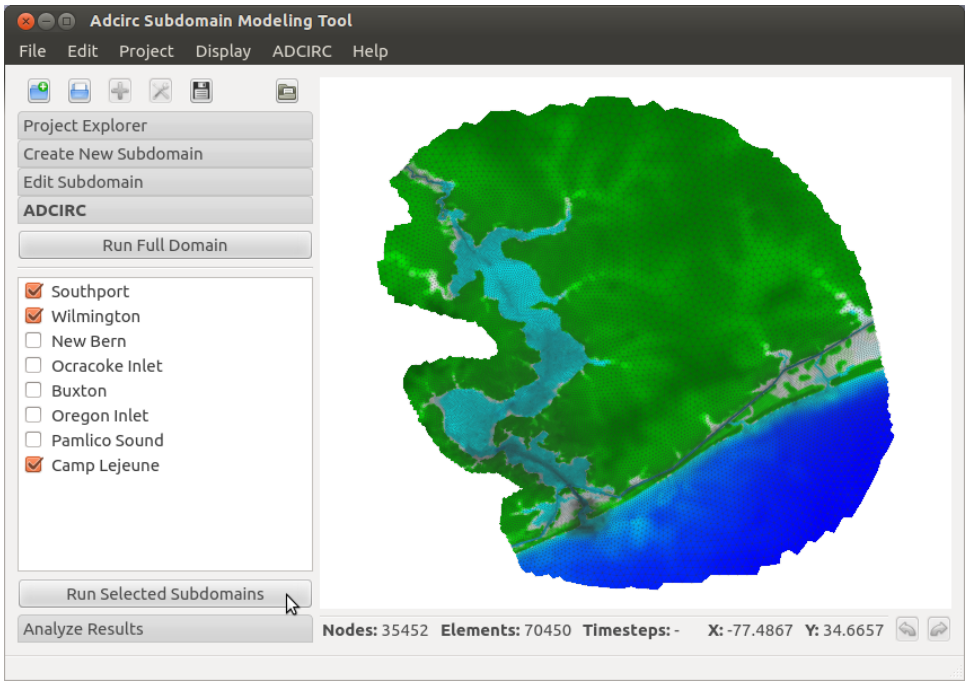


Figure 4.21: Running Three of the North Carolina Coast Subdomains

Chapter 5

Conclusions

Subdomain modeling as it exists requires extensive pre- and post-processing of ADCIRC mesh data. This thesis presents an interactive graphical user interface, called SMT, that streamlines the subdomain modeling process in a fast, portable, and reliable way. Several shape selection tools are included in SMT that allow the user to select elements of an ADCIRC mesh by drawing the shape on top of a plan view of the mesh. These tools use a newly developed search algorithm, called the generalized range search algorithm, to provide sufficient performance for effective levels of interactivity by performing the actual search in order to determine the elements selected by the user. The algorithm, which operates on a quadtree data structure that contains an ADCIRC mesh, works on the principal that if the container of a set of elements does not fall inside the shape, none of the elements that it contains will fall inside the shape either. Likewise, if the container falls entirely inside the shape, all of the elements that it contains will also fall inside the shape. The derivations of the upper and lower bounds on performance of this algorithm are presented, and test cases show that the actual performance falls in line with the expected performance. Additionally, the algorithm performs best when selecting small sets of elements so it is well suited for subdomain modeling. Finally, two case studies that show the ease and speed with which a modeler can now use the subdomain modeling technique, are been presented.

Future Work

Directions for future work with SMT are numerous. Work is underway on all features described in this thesis that have not yet been implemented, such as mesh modification and visualization of results. Methods of providing a map overlay feature that allows users to identify road networks, geographic features, and points of interest are being explored. Additionally, because full domain runs are often performed on high performance computing clusters, a feature that allows remote connectivity is being developed. Future work on the generalized range search algorithm includes research into the effects of tree balancing on performance, as well as methods for improving performance for large solution sizes.

REFERENCES

[1] ADCIRC, http://www.adcirc.org.

[2] A. Altuntas, Downscaling storm surge models for engineering applications, Master's thesis, North Carolina State University, Raleigh, NC (2012).

[3] J. S. Simon, A computational approach for local storm surge modeling, Master's thesis, North Carolina State University, Raleigh, NC (2011).

[4] S. K. Card, G. G. Robertson, J. D. Mackinlay, The information visualizer, an information workspace, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '91, ACM, New York, NY, USA, 1991, pp. 181–186. doi:10.1145/108844.108874. URL http://doi.acm.org/10.1145/108844.108874

[5] S. K. Card, The psychology of human-computer interaction, L. Erlbaum Associates, Hillsdale, N.J., 1983.

[6] R. S. Wright Jr., N. Haemel, G. Sellers, B. Lipchak, OpenGL Superbible, 5th Edition, Addison-Wesley, Upper Saddle River, NJ, 2011.

[7] L. Goldthwaite, Technical report on C++ performance, ISO/IEC PDTR 18015.

[8] Qt Project, http://qt-project.org/doc/qt-5.1/qtdoc/index.html.

[9] gtkmm, http://www.gtkmm.org/en/documentation.html.

[10] wxWidgets, http://docs.wxwidgets.org/stable/.

[11] Java 2d API specification, http://docs.oracle.com/javase/7/docs/technotes/guides/2d/spec.html.

[12] Java binding for the openGL API, http://jogamp.org/jogl/www/.

[13] PyOpenGL – the python openGL binding, http://pyopengl.sourceforge.net/.

[14] H. Samet, The Design And Analysis of Spatial Data Structures, Addison-Wesley, Reading, Massachusetts, 1990.

[15] H. Samet, Applications of Spatial Data Structures, Addison-Wesley, Reading, Massachusetts, 1990.

[16] S. Har-Peled, Geometric Approximation Algorithms, Vol. 173 of Mathematical Surveys and Monographs, American Mathematical Society, Providence, R.I., 2011.

[17] H. Samet, A. Rosenfeld, Application of hierarchical data structures to geographical information systems, Tech. rep., University of Maryland, College Park, MD (September 1983).

[18] R. K. V. Kothuri, S. Ravada, D. Abugov, Quadtree and r-tree indexes in oracle spatial: A comparison using gis data, in: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data, SIGMOD '02, ACM, New York, NY, USA, 2002, pp. 546–557. doi:10.1145/564691.564755.
URL http://doi.acm.org/10.1145/564691.564755

[19] A. Guttman, R-trees: A dynamic index structure for spatial searching, SIGMOD Rec. 14 (2) (1984) 47–57. doi:10.1145/971697.602266.
URL http://doi.acm.org/10.1145/971697.602266

[20] J. O'Rourke, Computational Geometry in C, 2nd Edition, Cambridge Press University, New York, 1998.

[21] E. Haines, Point in polygon strategies, Graphics Gems IV 994 (1994) 24–26.

[22] K. Hormann, A. Agathos, The point in polygon problem for arbitrary polygons, Computational Geometry 20 (3) (2001) 131–144. doi:http://dx.doi.org/10.1016/S0925-7721(01)00012-8.
URL http://www.sciencedirect.com/science/article/pii/S0925772101000128

[23] W. R. Franklin, PNPOLY - Point Inclusion in Polygon Test, http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html.

[24] J. Erickson, Non-lecture f: Line segment intersections, http://www.cs.uiuc.edu/~jeffe/teaching/algorithms/.

[25] J. H. Friedman, J. L. Bentley, R. A. Finkel, An algorithm for finding best matches in logarithmic expected time, ACM Transactions on Mathematical Software 3 (3) (1977) 209–226.

[26] I. Al-Bluwi, A. Elnagar, A logarithmic-complexity algorithm for nearest neighbor classification using layered range trees, Intelligent Information Management 4 (2) (2012) 39–43. URL http://proxying.lib.ncsu.edu/index.php?url=http://search.ebscohost.com/login.aspx?direct=true&db=iih&AN=75059225&site=ehost-live&scope=site

[27] E. P. Mucke, I. Saias, B. Zhu, Fast randomized point location without preprocessing in two- and three-dimensional Delaunay triangulation, Computational Geometry 12 (1999) 63–83.

[28] H. Sutter, A. Alexandrescu, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Addison-Wesley, Boston, 2005.

[29] J. W. Baugh Jr., A. Altuntas, J. S. Simon, Subdomain modeling of storm surge with engineering applications, submitted.

APPENDICES

Appendix A

Data Type Definitions

adcData.h

```
#ifndef ADCDATA_H
#define ADCDATA_H


struct Node
{
                unsigned int nodeNumber;
                float x;
                float y;
                float z;
                float normX;
                float normY;
                float normZ;
};


struct Element
{
                unsigned int elementNumber;
                Node*   n1;
                Node*   n2;
                Node*   n3;
};


struct Point
{
                float x;
                float y;
                Point() : x(0.0), y(0.0) {}
                Point(float a, float b) : x(a), y(b) {}
};


#endif // ADCDATA_H
```

QuadtreeData.h

```
#ifndef QUADTREEDATA_H
#define QUADTREEDATA_H

#include "adcData.h"
#include <vector>


struct leaf
{
                float                   bounds[4];
                std::vector<Node*>      nodes;
                std::vector<Element*>   elements;

                bool contains(Point p)
                {
                        return (p.x >= bounds[0] &&
                                p.x <= bounds[1] &&
                                p.y >= bounds[2] &&
                                p.y <= bounds[3]);
                }
};


typedef struct branch {
                float           bounds[4];
                leaf            *leaves[4];
                struct branch   *branches[4];

                bool contains(Point p)
                {
                        return (p.x >= bounds[0] &&
                                p.x <= bounds[1] &&
                                p.y >= bounds[2] &&
                                p.y <= bounds[3]);
                }
} branch;


#endif // QUADTREEDATA_H
```

Appendix B

`QuadtreeSearch` Class Code

QuadtreeSearch.h

```
#ifndef QUADTREESEARCH_H
#define QUADTREESEARCH_H

#include "adcData.h"
#include "QuadtreeData.h"

class QuadtreeSearch
{
        public:
                QuadtreeSearch();

                std::vector<Node*>      FindNodes(branch *root);
                std::vector<Element*>   FindElements(branch *root);

        protected:

                Point           shapeEdgePoint;

                virtual bool    PointIsInsideShape(Point p1) = 0;
                virtual bool    ShapeIntersects(branch *currBranch) = 0;
                virtual bool    ShapeIntersects(leaf *currLeaf) = 0;

                std::vector<Node*>      finalNodes;
                std::vector<Node*>      partialNodes;
                std::vector<Element*>   finalElements;
                std::vector<Element*>   partialElements;

                int     SearchNodes(branch *currBranch);
                int     SearchNodes(leaf *currLeaf);
                int     SearchElements(branch *currBranch);
                int     SearchElements(leaf *currLeaf);
                void    BruteForceNodes();
                void    BruteForceElements();
                void    AddAll(branch *currBranch, std::vector<Node*>* nodeList);
                void    AddAll(leaf *currLeaf, std::vector<Node*>* nodeList);
                void    AddAll(branch *currBranch, std::vector<Element*>* elementList);
                void    AddAll(leaf *currLeaf, std::vector<Element*>* elementList);
};

#endif
```

QuadtreeSearch.cpp

```cpp
#include "QuadtreeSearch.h"

QuadtreeSearch::QuadtreeSearch()
{
}


std::vector<Node*> QuadtreeSearch::FindNodes(branch *root)
{
	finalNodes.clear();
	partialNodes.clear();

	SearchNodes(root);
	BruteForceNodes();

	return finalNodes;
}


std::vector<Element*> QuadtreeSearch::FindElements(branch *root)
{
	finalElements.clear();
	partialElements.clear();

	SearchElements(root);
	BruteForceElements();

	return finalElements;
}


std::vector< std::vector<Node*>*> QuadtreeSearch::FindNodesLists(branch *root)
{
	finalNodesLists.clear();
	partialNodesLists.clear();

	SearchNodesPtrs(root);
	BruteForceNodesLists();

	return finalNodesLists;
}


std::vector< std::vector<Element*>*> QuadtreeSearch::FindElementsLists(branch *root)
{
	finalElementsLists.clear();
	partialElementsLists.clear();

	SearchElementsPtrs(root);
	BruteForceElementsLists();

	return finalElementsLists;
}
```

```
54
55
56  int QuadtreeSearch::SearchNodes(branch *currBranch)
57  {
58          if (ShapeIntersects(currBranch))
59          {
60                  for (int i=0; i<4; ++i)
61                  {
62                          if (currBranch->branches[i])
63                          {
64                                  if (SearchNodes(currBranch->branches[i]))
65                                          return 1;
66                          }
67                  }
68                  for (int i=0; i<4; ++i)
69                  {
70                          if (currBranch->leaves[i])
71                          {
72                                  if (SearchNodes(currBranch->leaves[i]))
73                                          return 1;
74                          }
75                  }
76          }
77          else if (currBranch->contains(shapeEdgePoint))
78          {
79                  for (int i=0; i<4; ++i)
80                  {
81                          if (currBranch->branches[i])
82                          {
83                                  if (SearchNodes(currBranch->branches[i]))
84                                          return 1;
85                          }
86                  }
87                  for (int i=0; i<4; ++i)
88                  {
89                          if (currBranch->leaves[i])
90                          {
91                                  if (SearchNodes(currBranch->leaves[i]))
92                                          return 1;
93                          }
94                  }
95                  return 1;
96          }
97          else
98          {
99                  if (PointIsInsideShape(Point(currBranch->bounds[1], currBranch->bounds[3])))
100                 {
101                         AddAll(currBranch, &finalNodes);
102                 }
103         }
104         return 0;
105 }
106
107
```

```
108 int QuadtreeSearch::SearchNodes(leaf *currLeaf)
109 {
110         if (ShapeIntersects(currLeaf))
111         {
112                 AddAll(currLeaf, &partialNodes);
113         }
114         else if (currLeaf->contains(shapeEdgePoint))
115         {
116                 AddAll(currLeaf, &partialNodes);
117                 return 1;
118         }
119         else if (PointIsInsideShape(Point(currLeaf->bounds[1], currLeaf->bounds[3])))
120         {
121                 AddAll(currLeaf, &finalNodes);
122         }
123         return 0;
124 }
125
126
127 int QuadtreeSearch::SearchElements(branch *currBranch)
128 {
129         if (ShapeIntersects(currBranch))
130         {
131                 for (int i=0; i<4; ++i)
132                 {
133                         if (currBranch->branches[i])
134                         {
135                                 if (SearchElements(currBranch->branches[i]))
136                                         return 1;
137                         }
138                 }
139                 for (int i=0; i<4; ++i)
140                 {
141                         if (currBranch->leaves[i])
142                         {
143                                 if (SearchElements(currBranch->leaves[i]))
144                                         return 1;
145                         }
146                 }
147         }
148         else if (currBranch->contains(shapeEdgePoint))
149         {
150                 for (int i=0; i<4; ++i)
151                 {
152                         if (currBranch->branches[i])
153                         {
154                                 if (SearchElements(currBranch->branches[i]))
155                                         return 1;
156                         }
157                 }
158                 for (int i=0; i<4; ++i)
159                 {
160                         if (currBranch->leaves[i])
161                         {
```

```
162                                 if (SearchElements(currBranch->leaves[i]))
163                                         return 1;
164                         }
165                 }
166                 return 1;
167         }
168         else if (PointIsInsideShape(Point(currBranch->bounds[1], currBranch->bounds[3])))
169         {
170                 AddAll(currBranch, &finalElements);
171         }
172         return 0;
173 }
174
175
176 int QuadtreeSearch::SearchElements(leaf *currLeaf)
177 {
178         if (ShapeIntersects(currLeaf))
179         {
180                 AddAll(currLeaf, &partialElements);
181         }
182         else if (currLeaf->contains(shapeEdgePoint))
183         {
184                 AddAll(currLeaf, &partialElements);
185                 return 1;
186         }
187         else if (PointIsInsideShape(Point(currLeaf->bounds[1], currLeaf->bounds[3])))
188         {
189                 AddAll(currLeaf, &finalElements);
190         }
191         return 0;
192 }
193
194
195 void QuadtreeSearch::BruteForceNodes()
196 {
197         for (std::vector<Node*>::iterator currNode = partialNodes.begin();
198              currNode != partialNodes.end();
199              ++currNode)
200         {
201                 if (PointIsInsideShape(Point((*currNode)->normX, (*currNode)->normY)))
202                 {
203                         finalNodes.push_back(*currNode);
204                 }
205         }
206 }
207
208
209 void QuadtreeSearch::BruteForceElements()
210 {
211         for (std::vector<Element*>::iterator currElement = partialElements.begin();
212              currElement != partialElements.end();
213              ++currElement)
214         {
215                 if (PointIsInsideShape(Point((*currElement)->n1->normX,
```

```
                                                (*currElement)->n1->normY)) ||
                    PointIsInsideShape(Point((*currElement)->n2->normX,
                                                (*currElement)->n2->normY)) ||
                    PointIsInsideShape(Point((*currElement)->n3->normX,
                                                (*currElement)->n3->normY)))
                {
                        finalElements.push_back(*currElement);
                }
        }
}


void QuadtreeSearch::AddAll(branch *currBranch, std::vector<Node*>* nodeList)
{
        for (int i=0; i<4; ++i)
        {
                if (currBranch->branches[i])
                {
                        AddAll(currBranch->branches[i], nodeList);
                }
                if (currBranch->leaves[i])
                {
                        AddAll(currBranch->leaves[i], nodeList);
                }
        }
}


void QuadtreeSearch::AddAll(leaf *currLeaf, std::vector<Node*>* nodeList)
{
        if (currLeaf->nodes.size() > 0)
        {
                nodeList->insert(nodeList->end(),
                                currLeaf->nodes.begin(),
                                currLeaf->nodes.end());
        }
}


void QuadtreeSearch::AddAll(branch *currBranch, std::vector<Element*>* elementList)
{
        for (int i=0; i<4; ++i)
        {
                if (currBranch->branches[i])
                {
                        AddAll(currBranch->branches[i], elementList);
                }
                if (currBranch->leaves[i])
                {
                        AddAll(currBranch->leaves[i], elementList);
                }
        }
}

```

```
270
271 void QuadtreeSearch::AddAll(leaf *currLeaf, std::vector<Element*>* elementList)
272 {
273         if (currLeaf->elements.size() > 0)
274                 elementList->insert(elementList->end(),
275                                     currLeaf->elements.begin(),
276                                     currLeaf->elements.end());
277 }
```

Appendix C

`CircleSearch` Class Code

CircleSearch.h

```
#ifndef CIRCLESEARCH_H
#define CIRCLESEARCH_H

#include "QuadtreeSearch.h"

class CircleSearch : public QuadtreeSearch
{
        public:
                CircleSearch();

                void    SetCircleParameters(float x, float y, float radius);

        protected:

                virtual bool    PointIsInsideShape(Point p1);
                virtual bool    ShapeIntersects(branch *b);
                virtual bool    ShapeIntersects(leaf *l);

        private:

                float   x, y, r;

                bool    EdgeIntersectsCircle(Point p1, Point p2);
                float   DistanceSquared(float x1, float y1, float x2, float y2);
};

#endif // CIRCLESEARCH_H
```

CircleSearch.cpp

```cpp
#include "CircleSearch.h"

CircleSearch::CircleSearch()
{
}


void CircleSearch::SetCircleParameters(float x, float y, float radius)
{
	this->x = x;
	this->y = y;
	this->r = radius;
	shapeEdgePoint.x = x+radius;
	shapeEdgePoint.y = y;
}


bool CircleSearch::PointIsInsideShape(Point p1)
{
	return DistanceSquared(x, y, p1.x, p1.y) < r*r;
}


bool CircleSearch::ShapeIntersects(branch *b)
{
	Point BottomLeft(b->bounds[0], b->bounds[2]);
	Point BottomRight(b->bounds[1], b->bounds[2]);
	Point TopLeft(b->bounds[0], b->bounds[3]);
	Point TopRight(b->bounds[1], b->bounds[3]);

	return EdgeIntersectsCircle(BottomLeft, BottomRight) ||
		EdgeIntersectsCircle(BottomRight, TopRight) ||
		EdgeIntersectsCircle(TopRight, TopLeft) ||
		EdgeIntersectsCircle(TopLeft, BottomLeft);
}


bool CircleSearch::ShapeIntersects(leaf *l)
{
	Point BottomLeft(l->bounds[0], l->bounds[2]);
	Point BottomRight(l->bounds[1], l->bounds[2]);
	Point TopLeft(l->bounds[0], l->bounds[3]);
	Point TopRight(l->bounds[1], l->bounds[3]);

	return EdgeIntersectsCircle(BottomLeft, BottomRight) ||
		EdgeIntersectsCircle(BottomRight, TopRight) ||
		EdgeIntersectsCircle(TopRight, TopLeft) ||
		EdgeIntersectsCircle(TopLeft, BottomLeft);
}


bool CircleSearch::EdgeIntersectsCircle(Point p1, Point p2)
{
```

```
54          float dist1 = DistanceSquared(x, y, p1.x, p1.y);
55          float dist2 = DistanceSquared(x, y, p2.x, p2.y);
56          bool p1Inside = dist1 <= r*r;
57          bool p2Inside = dist2 <= r*r;
58          if (p1Inside && p2Inside)
59          {
60                  return false;
61          }
62          else if (p1Inside != p2Inside)
63          {
64                  return true;
65          }
66
67          float xDiff = p2.x - p1.x;
68          float yDiff = p2.y - p1.y;
69          float d2 = xDiff*xDiff + yDiff*yDiff;
70          float u = (xDiff * (x - p1.x) + yDiff * (y - p1.y)) / d2;
71          if (u >= 0.0 && u <= 1.0)
72          {
73                  float closestX = p1.x + u*xDiff;
74                  float closestY = p1.y + u*yDiff;
75                  return DistanceSquared(x, y, closestX, closestY) <= r*r;
76          } else {
77                  return (DistanceSquared(x, y, p1.x, p1.y) <= r*r ||
78                          DistanceSquared(x, y, p2.x, p2.y) <= r*r);
79          }
80  }
81
82
83  float CircleSearch::DistanceSquared(float x1, float y1, float x2, float y2)
84  {
85          return (x1 - x2)*(x1 - x2) + (y1 - y2)*(y1 - y2);
86  }
```

Appendix D

`RectangleSearch` Class Code

RectangleSearch.h

```
#ifndef RECTANGLESEARCH_H
#define RECTANGLESEARCH_H

#include "QuadtreeSearch.h"
class RectangleSearch : public QuadtreeSearch
{
        public:
                RectangleSearch();

                void    SetRectangleParameters(float l, float r, float b, float t);

        protected:

                virtual bool    PointIsInsideShape(Point p1);
                virtual bool    ShapeIntersects(branch *currBranch);
                virtual bool    ShapeIntersects(leaf *currLeaf);

        private:

                float l, r, b, t;


};

#endif // RECTANGLESEARCH_H
```

RectangleSearch.cpp

```cpp
#include "RectangleSearch.h"

RectangleSearch::RectangleSearch()
{
}


void RectangleSearch::SetRectangleParameters(float l, float r, float b, float t)
{
        this->l = l;
        this->r = r;
        this->b = b;
        this->t = t;
        shapeEdgePoint.x = l;
        shapeEdgePoint.y = b;
}


bool RectangleSearch::PointIsInsideShape(Point p1)
{
        return (p1.x <= r &&
                p1.x >= l &&
                p1.y <= t &&
                p1.y >= b);
}

bool RectangleSearch::ShapeIntersects(branch *currBranch)
{
        if (!(l >= currBranch->bounds[1] ||
              r <= currBranch->bounds[0] ||
              b >= currBranch->bounds[3] ||
              t <= currBranch->bounds[2]))
        {
             Point BottomLeft(currBranch->bounds[0], currBranch->bounds[2]);
             Point BottomRight(currBranch->bounds[1], currBranch->bounds[2]);
             Point TopLeft(currBranch->bounds[0], currBranch->bounds[3]);
             Point TopRight(currBranch->bounds[1], currBranch->bounds[3]);
             if (PointIsInsideShape(BottomLeft) &&
                 PointIsInsideShape(BottomRight) &&
                 PointIsInsideShape(TopLeft) &&
                 PointIsInsideShape(TopRight))
                  return false;
             if (currBranch->contains(Point(b, l)) &&
                 currBranch->contains(Point(b, r)) &&
                 currBranch->contains(Point(t, l)) &&
                 currBranch->contains(Point(t, r)))
                  return false;
             return true;
        }
        return false;
}

bool RectangleSearch::ShapeIntersects(leaf *currLeaf)
```

```
{
        if (!(l >= currLeaf->bounds[1] ||
              r <= currLeaf->bounds[0] ||
              b >= currLeaf->bounds[3] ||
              t <= currLeaf->bounds[2]))
        {
             Point BottomLeft(currLeaf->bounds[0], currLeaf->bounds[2]);
             Point BottomRight(currLeaf->bounds[1], currLeaf->bounds[2]);
             Point TopLeft(currLeaf->bounds[0], currLeaf->bounds[3]);
             Point TopRight(currLeaf->bounds[1], currLeaf->bounds[3]);
             if (PointIsInsideShape(BottomLeft) &&
                 PointIsInsideShape(BottomRight) &&
                 PointIsInsideShape(TopLeft) &&
                 PointIsInsideShape(TopRight))
                  return false;
             if (currLeaf->contains(Point(b, l)) &&
                 currLeaf->contains(Point(b, r)) &&
                 currLeaf->contains(Point(t, l)) &&
                 currLeaf->contains(Point(t, r)))
                  return false;
             return true;
        }
        return false;
}
```

Appendix E

`PolygonSearch` Class Code

PolygonSearch.h

```
#ifndef POLYGONSEARCH_H
#define POLYGONSEARCH_H

#include "QuadtreeSearch.h"

class PolygonSearch : public QuadtreeSearch
{
        public:
                PolygonSearch();

                void    SetPolygonParameters(std::vector<Point> polyLine);

        protected:

                virtual bool    PointIsInsideShape(Point p1);
                virtual bool    ShapeIntersects(branch *currBranch);
                virtual bool    ShapeIntersects(leaf *currLeaf);

        private:

                std::vector<Point>      polygonPoints;

                bool    EdgesIntersect(Point pointA, Point pointB, Point pointC, Point pointD);
                bool    IsCCW(Point A, Point B, Point C);
};

#endif // POLYGONSEARCH_H
```

PolygonSearch.cpp

```cpp
#include "PolygonSearch.h"

PolygonSearch::PolygonSearch()
{
}

void PolygonSearch::SetPolygonParameters(std::vector<Point> polyLine)
{
        polygonPoints = polyLine;
        shapeEdgePoint = polyLine[0];
}


bool PolygonSearch::PointIsInsideShape(Point p1)
{
        bool pointIsInside = false;
        unsigned int i, j = 0;
        for (i=0, j=polygonPoints.size()-1; i<polygonPoints.size(); j = i++)
        {
                if (((polygonPoints[i].y > p1.y) != (polygonPoints[j].y > p1.y)) &&
                    (p1.x < (polygonPoints[j].x-polygonPoints[i].x) *
                    (p1.y-polygonPoints[i].y) / (polygonPoints[j].y-polygonPoints[i].y) +
                    polygonPoints[i].x))
                        pointIsInside = !pointIsInside;
        }

        return pointIsInside;
}


bool PolygonSearch::ShapeIntersects(branch *currBranch)
{
        Point point1(currBranch->bounds[0], currBranch->bounds[2]);     /* Bottom Left */
        Point point2(currBranch->bounds[1], currBranch->bounds[2]);     /* Bottom Right */
        Point point3(currBranch->bounds[0], currBranch->bounds[3]);     /* Top Left */
        Point point4(currBranch->bounds[1], currBranch->bounds[3]);     /* Top Right */
        unsigned int pointCount = polygonPoints.size();

        for (unsigned int i=0; i<pointCount-1; ++i)
                if (EdgesIntersect(point1, point2, polygonPoints[i], polygonPoints[i+1]) ||
                    EdgesIntersect(point1, point3, polygonPoints[i], polygonPoints[i+1]) ||
                    EdgesIntersect(point3, point4, polygonPoints[i], polygonPoints[i+1]) ||
                    EdgesIntersect(point2, point4, polygonPoints[i], polygonPoints[i+1]))
                        return true;
        if (EdgesIntersect(point1, point2, polygonPoints[0], polygonPoints[pointCount-1]) ||
            EdgesIntersect(point1, point3, polygonPoints[0], polygonPoints[pointCount-1]) ||
            EdgesIntersect(point3, point4, polygonPoints[0], polygonPoints[pointCount-1]) ||
            EdgesIntersect(point2, point4, polygonPoints[0], polygonPoints[pointCount-1]))
                return true;
        return false;
}


```

```
54  bool PolygonSearch::ShapeIntersects(leaf *currLeaf)
55  {
56          Point point1(currLeaf->bounds[0], currLeaf->bounds[2]); /* Bottom Left */
57          Point point2(currLeaf->bounds[1], currLeaf->bounds[2]); /* Bottom Right */
58          Point point3(currLeaf->bounds[0], currLeaf->bounds[3]); /* Top Left */
59          Point point4(currLeaf->bounds[1], currLeaf->bounds[3]); /* Top Right */
60          unsigned int pointCount = polygonPoints.size();
61
62          for (unsigned int i=0; i<pointCount-1; ++i)
63                  if (EdgesIntersect(point1, point2, polygonPoints[i], polygonPoints[i+1]) ||
64                      EdgesIntersect(point1, point3, polygonPoints[i], polygonPoints[i+1]) ||
65                      EdgesIntersect(point3, point4, polygonPoints[i], polygonPoints[i+1]) ||
66                      EdgesIntersect(point2, point4, polygonPoints[i], polygonPoints[i+1]))
67                          return true;
68          if (EdgesIntersect(point1, point2, polygonPoints[0], polygonPoints[pointCount-1]) ||
69              EdgesIntersect(point1, point3, polygonPoints[0], polygonPoints[pointCount-1]) ||
70              EdgesIntersect(point3, point4, polygonPoints[0], polygonPoints[pointCount-1]) ||
71              EdgesIntersect(point2, point4, polygonPoints[0], polygonPoints[pointCount-1]))
72                  return true;
73          return false;
74  }
75
76
77  bool PolygonSearch::EdgesIntersect(Point pointA, Point pointB, Point pointC, Point pointD)
78  {
79          if (IsCCW(pointA, pointC, pointD) == IsCCW(pointB, pointC, pointD))
80          {
81                  return false;
82          }
83          else if (IsCCW(pointA, pointB, pointC) == IsCCW(pointA, pointB, pointD))
84          {
85                  return false;
86          } else {
87                  return true;
88          }
89  }
90
91
92  bool PolygonSearch::IsCCW(Point A, Point B, Point C)
93  {
94          if ((C.y-A.y) * (B.x-A.x) > (B.y-A.y) * (C.x-A.x))
95                  return true;
96          return false;
97  }
```

Appendix F

`PointSearch` Class Code

PointSearch.h

```
#ifndef POINTSEARCH_H
#define POINTSEARCH_H

#include "QuadtreeSearch.h"

class PointSearch : public QuadtreeSearch
{
        public:
                PointSearch();

                void                    SetPointParameters(float x, float y);
                std::vector<Node*>      FindNodes(branch *root);
                std::vector<Element*>   FindElements(branch *root);

        protected:

                virtual bool    PointIsInsideShape(Point p1);
                virtual bool    ShapeIntersects(branch *currBranch);
                virtual bool    ShapeIntersects(leaf *currLeaf);

        private:

                float x, y;

                void            CustomBruteForceNodes();
                void            CustomBruteForceElements();
                float           DistanceSquared(float nodeX, float nodeY);
                bool            PointIsInsideElement(Element *currElement);
};

#endif // POINTSEARCH_H
```

PointSearch.cpp

```cpp
#include "PointSearch.h"

PointSearch::PointSearch()
{
}


std::vector<Node*> PointSearch::FindNodes(branch *root)
{
	finalNodes.clear();
	partialNodes.clear();

	SearchNodes(root);
	CustomBruteForceNodes();

	return finalNodes;
}


std::vector<Element*> PointSearch::FindElements(branch *root)
{
	finalElements.clear();
	partialElements.clear();

	SearchElements(root);
	CustomBruteForceElements();

	return finalElements;
}


void PointSearch::SetPointParameters(float x, float y)
{
	this->x = x;
	this->y = y;
	shapeEdgePoint.x = x;
	shapeEdgePoint.y = y;
}


bool PointSearch::PointIsInsideShape(Point p1)
{
	return false;
}


bool PointSearch::ShapeIntersects(branch *currBranch)
{
	return  (x >= currBranch->bounds[0] &&
		 x <= currBranch->bounds[1] &&
		 y >= currBranch->bounds[2] &&
		 y <= currBranch->bounds[3]);
}
```

```
54
55
56  bool PointSearch::ShapeIntersects(leaf *currLeaf)
57  {
58          return  (x >= currLeaf->bounds[0] &&
59                   x <= currLeaf->bounds[1] &&
60                   y >= currLeaf->bounds[2] &&
61                   y <= currLeaf->bounds[3]);
62  }
63
64
65  void PointSearch::CustomBruteForceNodes()
66  {
67          if (partialNodes.size())
68          {
69                  Node *currClosest = partialNodes[0];
70                  float currDistance = DistanceSquared(currClosest->normX, currClosest->normY);
71                  for (std::vector<Node*>::iterator currNode = partialNodes.begin() + 1;
72                       currNode != partialNodes.end();
73                       ++currNode)
74                  {
75                          float newDistance = DistanceSquared((*currNode)->normX, (*currNode)->normY);
76                          if (newDistance < currDistance)
77                          {
78                                  currClosest = *currNode;
79                                  currDistance = newDistance;
80                          }
81                  }
82                  finalNodes.push_back(currClosest);
83          }
84  }
85
86
87
88  void PointSearch::CustomBruteForceElements()
89  {
90          if (partialElements.size())
91          {
92                  for (std::vector<Element*>::iterator currElement = partialElements.begin();
93                       currElement != partialElements.end();
94                       ++currElement)
95                  {
96                          if (PointIsInsideElement(*currElement))
97                          {
98                                  finalElements.push_back(*currElement);
99                                  return;
100                         }
101                 }
102         }
103 }
104
105
106 float PointSearch::DistanceSquared(float nodeX, float nodeY)
107 {
```

```
108         return ((nodeX-x)*(nodeX-x) + (nodeY-y)*(nodeY-y));
109 }
110
111
112 bool PointSearch::PointIsInsideElement(Element *currElement)
113 {
114         Node *n1 = currElement->n1;
115         Node *n2 = currElement->n2;
116         Node *n3 = currElement->n3;
117         float a = (n2->normX - n1->normX)*(y - n1->normY) - (x - n1->normX)*(n2->normY - n1->normY);
118         float b = (n3->normX - n2->normX)*(y - n2->normY) - (x - n2->normX)*(n3->normY - n2->normY);
119         float c = (n1->normX - n3->normX)*(y - n3->normY) - (x - n3->normX)*(n1->normY - n3->normY);
120
121
122         if ((a > 0 && b > 0 && c > 0) || (a < 0 && b < 0 && c < 0))
123                 return true;
124         return false;
125 }
```

Appendix G

`ConvexCircleSearch` Class Code

ConvexCircleSearch.h

```
#ifndef CONVEXCIRCLESEARCH_H
#define CONVEXCIRCLESEARCH_H

#include "CircleSearchNew.h"

class ConvexCircleSearch : public CircleSearch
{
        public:
                ConvexCircleSearch();

        protected:

                int     SearchElements(branch *currBranch);
                int     SearchElements(leaf *currLeaf);
};

#endif // CONVEXCIRCLESEARCH_H
```

ConvexCircleSearch.cpp

```cpp
#include "ConvexCircleSearch.h"

ConvexCircleSearch::ConvexCircleSearch()
{
}


int ConvexCircleSearch::SearchElements(branch *currBranch)
{
	int cornerInShapeCount = 0;
	if (PointIsInsideShape(Point(currBranch->bounds[0], currBranch->bounds[3])))
		++cornerInShapeCount;
	if (PointIsInsideShape(Point(currBranch->bounds[0], currBranch->bounds[2])))
		++cornerInShapeCount;
	if (PointIsInsideShape(Point(currBranch->bounds[1], currBranch->bounds[3])))
		++cornerInShapeCount;
	if (PointIsInsideShape(Point(currBranch->bounds[1], currBranch->bounds[2])))
		++cornerInShapeCount;

	if (cornerInShapeCount == 4)
	{
		AddAll(currBranch, &finalNodes);
	}
	else if (cornerInShapeCount != 0)
	{
		for (int i=0; i<4; ++i)
		{
			if (currBranch->branches[i])
			{
				if (SearchElements(currBranch->branches[i]))
					return 1;
			}
		}
		for (int i=0; i<4; ++i)
		{
			if (currBranch->leaves[i])
			{
				if (SearchElements(currBranch->leaves[i]))
					return 1;
			}
		}
	}
	else if (currBranch->contains(shapeEdgePoint))
	{
		for (int i=0; i<4; ++i)
		{
			if (currBranch->branches[i])
			{
				if (SearchElements(currBranch->branches[i]))
					return 1;
			}
		}
		for (int i=0; i<4; ++i)
```

```
54                  {
55                          if (currBranch->leaves[i])
56                          {
57                                  if (SearchElements(currBranch->leaves[i]))
58                                          return 1;
59                          }
60                  }
61                  return 1;
62          }
63          else if (ShapeIntersects(currBranch))
64          {
65                  for (int i=0; i<4; ++i)
66                  {
67                          if (currBranch->branches[i])
68                          {
69                                  if (SearchElements(currBranch->branches[i]))
70                                          return 1;
71                          }
72                  }
73                  for (int i=0; i<4; ++i)
74                  {
75                          if (currBranch->leaves[i])
76                          {
77                                  if (SearchElements(currBranch->leaves[i]))
78                                          return 1;
79                          }
80                  }
81          }
82          return 0;
83  }
84
85  int ConvexCircleSearch::SearchElements(leaf *currLeaf)
86  {
87          int cornerInShapeCount = 0;
88          if (PointIsInsideShape(Point(currLeaf->bounds[0], currLeaf->bounds[3])))
89                  ++cornerInShapeCount;
90          if (PointIsInsideShape(Point(currLeaf->bounds[0], currLeaf->bounds[2])))
91                  ++cornerInShapeCount;
92          if (PointIsInsideShape(Point(currLeaf->bounds[1], currLeaf->bounds[3])))
93                  ++cornerInShapeCount;
94          if (PointIsInsideShape(Point(currLeaf->bounds[1], currLeaf->bounds[2])))
95                  ++cornerInShapeCount;
96
97          if (cornerInShapeCount == 4)
98          {
99                  AddAll(currLeaf, &finalNodes);
100         }
101         else if (cornerInShapeCount != 0)
102         {
103                 AddAll(currLeaf, &partialNodes);
104         }
105         else if (currLeaf->contains(shapeEdgePoint))
106         {
107                 AddAll(currLeaf, &partialNodes);
```

```
108                     return 1;
109             }
110             else if (ShapeIntersects(currLeaf))
111             {
112                     AddAll(currLeaf, &partialNodes);
113             }
114             return 0;
115     }
```