ABSTRACT

JASPER, MICAH NATHANAEL. Development and Application of the DIRECT Algorithm for Leak Detection in Water Distribution Systems. (Under the direction of Dr. Kumar Mahinthakumar).

The Dividing Rectangles (DIRECT) search is a deterministic, global search algorithm for bound constrained problems. The algorithm searches for the global minimum by recursive space partitioning, essentially grouping similar regions in the decision space and selecting a sample from each group. The DIRECT algorithm was initially designed for continuous problems, but has since been modified to allow for integer variable types.

This research extends the DIRECT algorithm to use a mix of continuous and discrete variables, including connected graph nodes. The algorithm is first tested with standard test functions and then applied to leak detection problems in water distribution systems (WDSs), which involve both discrete network nodes and continuous magnitudes. In addition, the DIRECT algorithm is parallelized using a master-worker paradigm and tested using cluster resources for a moderate number of processors.

The extension of DIRECT algorithm to problems involving discrete network connected nodes will enable the application of DIRECT to our wider class of problems than previously possible.
Development and Application of the DIRECT Algorithm for Leak Detection in Water Distribution Systems

by
Micah Nathanael Jasper

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:

Dr. Gnanamanikam Mahinthakumar
Committee Chair

Dr. Sanmugavadivel Ranjithan
Dr. Earl Downey Brill, Jr.
DEDICATION

I dedicate this thesis to my wife, my bestest best friend, and something more,

Christian Ashley Jasper. C:-

I love you. Thank you for your love and support, and encouraging me to keep going.
Micah Nathanael Jasper grew up in Burlington, NC. After graduating from high school, he wanted to get as far away from home as possible, so he went to Duke University (only about 35 miles away). At Duke, he met his best friend and soon to be wife, and they were married on May 17, 2008, just after her graduation, and in between his junior and senior year. After graduating from Duke with a BS in Computer Science, he worked for a year in web development in Raleigh. His wife had worked on a Master's of Public Health at UNC and went on to medical school at Wake Forest. They had always had dreams of helping in developing nations together, her as a doctor and him as an engineer. So Micah went back to school for Civil Engineering at NC State. He hopes to continue on to do PhD research in industrial waste water treatment and combining mineral extraction with desalination. Then go on to get his professional license and organize trips to help others have access to clean water.
ACKNOWLEDGMENTS

First and foremost, I would like to thank God from whom all blessings flow, such as life, breath, a beating heart, and many other things that I take for granted (James 1:17). It is through our Lord Jesus that all things are possible (Mark 9:23) and it is He who is able to do immeasurably more than all we ask or imagine (Ephesians 3:20).

I would like to thank my advisor, Prof. Kumar for his help with the C coding, EPANET, and MPI. He is very knowledgeable about high-performance computing and inverse modeling, and he greatly aided this research. I would like to thank Prof. Ranjithan for encouraging me to go to graduate school when I was still taking undergraduate classes at NC State. I would also like to thank Prof. Brill for his encouragement through this research and for giving me the opportunity to TA his Engineering Economics class, probably my favorite engineering class. I would also like to thank my committee members for their feedback and discussion in the group meetings over the past two years. Thank you for your encouragement and advice.

I would also like to thank Jörg Gablonsky, Daniel Finkel, and Prof. C. T. Kelly who were students and faculty here at NC State, whom I have never met, but whose research on the DIRECT method greatly helped my understanding of the algorithm.

Thank you to the National Science Foundation and the Civil Engineering Department at NC State for funding this research.

And finally thank you to my wife, for all of her encouragement and support.

May God bless you all.
This research is supported in part by the National Science Foundation under grant number CMMI-1100458. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.
# TABLE OF CONTENTS

LIST OF TABLES ................................................................................................................... ix  
LIST OF FIGURES ................................................................................................................ x  

Chapter 1 Introduction .............................................................................................................. 1  
  1.1 The DIRECT Algorithm ................................................................................................. 1  
  1.2 Leak Detection in Water Distribution Systems .............................................................. 3  
  1.3 Overview ......................................................................................................................... 3  

Chapter 2 The Original DIRECT Algorithm ............................................................................ 4  
  2.1 Definition of Lipschitz Continuous ................................................................................ 4  
     2.1.1 Examples of Lipschitz continuous functions ........................................................... 5  
     2.1.2 Examples of Continuous Functions that are not Lipschitz Continuous ................... 5  
  2.2 Lipschitz Optimization in One Dimension ..................................................................... 6  
     2.2.1 Algorithm Steps ....................................................................................................... 7  
     2.2.2 One Dimensional Example ...................................................................................... 7  
     2.2.3 Disadvantages of Lipschitz Optimization ................................................................. 9  
  2.3 DIRECT Algorithm in One Dimension .......................................................................... 9  
     2.3.1 Division in One Dimension ..................................................................................... 9  
     2.3.2 Potentially Optimum Intervals ............................................................................... 11  
     2.3.3 $\varepsilon$ Parameter .................................................................................................. 14  
     2.3.4 Algorithm Steps ..................................................................................................... 15  
     2.3.5 One Dimensional Example .................................................................................... 17  
  2.4 DIRECT Algorithm in Two or More Dimensions ........................................................ 19  
     2.4.1 Normalization to a Unit Hypercube ....................................................................... 19  
     2.4.2 Division in Two or More Dimensions ................................................................... 19  
     2.4.3 Rectangle Size ........................................................................................................ 21  
     2.4.4 Algorithm Steps ..................................................................................................... 22  
     2.4.5 Two Dimensional Example .................................................................................... 23  
  2.5 Changes to the DIRECT Algorithm from Prior Research ............................................ 24  
     2.5.1 Aggressive DIRECT .............................................................................................. 24  
     2.5.2 Dividing Only One Dimension .............................................................................. 25  
     2.5.3 Generalized Definition of Rectangle Side Length and Radius .............................. 26
LIST OF TABLES

Table 3.1 - General Variable Interface ................................................................. 33
Table 3.2 - Possible Division Procedures for Various Variable Types .................... 34
Table 3.3 - General Rectangle Interface ............................................................... 36
Table 3.4 - Function Evaluations to find Global Minimum for Test Functions .......... 39
Table 5.1 - Comparison of DIRECT Options for Low Dimensional Problems in Terms of the Number of Function Evaluations to Find the True Leak Case ............... 51
Table 5.2 - Comparison of Aggressive DIRECT Options for Low Dimensional Problems in Terms of the Number of Function Evaluations to Find the True Leak Case ...... 51
Table 5.3 - Comparison of Various $\epsilon$ Parameters using Method 1 for Low Dimensions in Terms of the Number of Function Evaluations to Find the True Leak Case ...... 53
Table 5.4 - Comparison of Various $\epsilon$ Parameters using Method 2 for Low Dimensions in Terms of the Number of Function Evaluations to Find the True Leak Case ...... 53
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Lipschitz continuous function, with Lipschitz constant $K$, such that for any point within the domain, the slope is within $\pm K$</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Functions that are Lipschitz continuous over the real domain</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Functions that are not Lipschitz continuous over the real domain</td>
<td>5</td>
</tr>
<tr>
<td>2.4</td>
<td>First Iteration of Lipschitz Optimization with the Objective Function Estimate, $B$, shown for the Interval $[a, b]$</td>
<td>6</td>
</tr>
<tr>
<td>2.5</td>
<td>One Dimensional Example of Lipschitz Optimization</td>
<td>8</td>
</tr>
<tr>
<td>2.6</td>
<td>Comparison of Lipschitz and DIRECT Divisions</td>
<td>10</td>
</tr>
<tr>
<td>2.7</td>
<td>Graphical interpretation of &quot;potentially optimal&quot; interval selection, with intervals represented as points, and the &quot;potentially optimal&quot; intervals along the delineated convex hull</td>
<td>11</td>
</tr>
<tr>
<td>2.8</td>
<td>Graphical interpretation of Lipschitz interval selection, with the Lipschitz constant, $K$, represented as a line, and the selected interval as a point along this line</td>
<td>13</td>
</tr>
<tr>
<td>2.9</td>
<td>Graphical interpretation of DIRECT interval selection with $\varepsilon$ Parameter illustrated as a slope and the &quot;potentially optimal&quot; intervals delineated along the modified convex hull</td>
<td>15</td>
</tr>
<tr>
<td>2.10</td>
<td>Rectangle Storage for Potentially Optimal Determination</td>
<td>16</td>
</tr>
<tr>
<td>2.11</td>
<td>One Dimensional Example of DIRECT Optimization</td>
<td>18</td>
</tr>
<tr>
<td>2.12</td>
<td>DIRECT sampling and division in two dimensions with the objective function value shown above each point</td>
<td>20</td>
</tr>
<tr>
<td>2.13</td>
<td>Objective Function for Two Dimensional DIRECT Example</td>
<td>23</td>
</tr>
<tr>
<td>2.14</td>
<td>Two Dimensional Example of DIRECT Optimization</td>
<td>24</td>
</tr>
<tr>
<td>2.15</td>
<td>Graphical interpretation of the rectangle selection using Aggressive DIRECT with most optimal rectangle of each size group selected and delineated in the plot</td>
<td>25</td>
</tr>
<tr>
<td>4.1</td>
<td>The WDS Simulator</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>The Simulation Optimization Approach</td>
<td>43</td>
</tr>
<tr>
<td>4.3</td>
<td>The Simulation Optimization Approach with Decision Variables and Objective Function</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Recursively Trisected WDS Nodes</td>
<td>45</td>
</tr>
<tr>
<td>5.1</td>
<td>Actual and Candidate Leak Locations for the Small Network (Net3)</td>
<td>48</td>
</tr>
</tbody>
</table>
Chapter 1 Introduction

The Dividing Rectangles (DIRECT) search is a deterministic, global search algorithm for bound constrained problems [1]. This method is able to locate areas around local optima in relatively few function evaluations, but it takes many more function evaluations to converge around a global optimum [2].

The DIRECT method was originally designed for continuous problems. The algorithm has since been modified to allow for integer problems [3]. However, the DIRECT method has not been applied to other discrete variable types, such as graph nodes. Part of this research is generalizing the DIRECT algorithm to allow for all possible variable types as long as the decision space can be intelligently grouped or divided, and as long as each group or section can be representatively sampled.

Furthermore, the DIRECT method has not previously been applied to problems involving a water distribution system (WDS). A second aim of this research is evaluate the feasibility of solving for the location and magnitude of leaks in a water network using the DIRECT search.

1.1 The DIRECT Algorithm

The DIRECT algorithm was developed to be a deterministic global search algorithm with few parameters to specify. Since it is deterministic, and not stochastic, there is no need for multiple runs. Furthermore, DIRECT does not have many algorithmic parameters, so that fine tuning is minimized [1]. In the original DIRECT algorithm, there is only one parameter to manipulate. However, subsequent modifications to DIRECT allow different configuration options, but there is still only one real-valued parameter to adjust.
The DIRECT method is typically used for problems with bound constraints (also called box constraints), meaning that each of the decision variables has a minimum and maximum values, but no other constraint equations. However, the DIRECT algorithm has been later modified to allow for other kinds of constraints [2] [3].

Furthermore, the DIRECT method is considered to be a sampling algorithm, or a derivative-free method, because it does not use the derivative of the objective function. This is particularly useful in engineering applications and other difficult problems where information about the function, such as the smoothness, continuity, and differentiability, is not available. Often times in engineering problems, the objective may be the product of a simulation or some other iterative method. This also means that it can be computationally expensive to calculate the objective, taking many minutes, hours, or even days. Therefore, it is preferable in many engineering problems to use an algorithm that takes few function evaluations and is easily parallelizable.

The DIRECT algorithm takes relatively few function evaluations when compared to other methods [1], and can be parallelized [2] [4] [5] [6], though the number of function evaluations does change per iteration. It has also been noted that the DIRECT method does not scale well in higher dimensions [3] [7]. This is a problem for some engineering applications, including the leak detection problem solved in this research.
1.2 Leak Detection in Water Distribution Systems

Water distribution systems (WDSs) are critical in transporting safe, clean drinking water to the public. Yet, these systems age and degrade, and are thus vulnerable to leaks. It has been estimated that up to 50 percent of water in a WDS is lost to leaks [8].

There are many current ways of detecting leaks. Equipment called acoustic listening devices that can detect the sound of water flowing through a crack in the pipe, like a stethoscope for pipes. Other methods include inverse transient analysis, infrared imaging, and tracer gas analysis. Most of the current methods are time and labor intensive. The goal of this research is to aid in the location of suspect leak areas by using measurements that can be routinely taken, such as the pressure and chlorine concentration at specific nodes, and the flows through certain pipes. These measurements can be used in an inverse modeling approach, specifically a simulation-optimization approach, using EPANET a WDS simulator, to solve for the location and magnitude of leaks.

1.3 Overview

Chapter 2 describes and derives the DIRECT algorithm from the original Lipschitz optimization algorithm. Chapter 3 details the implementation of the generalized DIRECT method for all variable types. Chapter 4 describes an implementation of the DIRECT method to solve the leak detection problem. Chapter 5 provides the results of the leak detection trails. Chapter 6 details the parallel implementation and performance. Finally, Chapter 7 gives the conclusions and potential future work.
Chapter 2 The Original DIRECT Algorithm

The DIRECT algorithm is considered a Lipschitzian optimization algorithm, even though it bears only a slight resemblance to the original Lipschitz optimization. It was originally designed to overcome some of the pitfalls of traditional Lipshitz optimization [1]. This chapter details the origin of the DIRECT algorithm and some modifications since.

2.1 Definition of Lipschitz Continuous

A function is considered Lipschitz continuous if the slope of the function is not greater in magnitude at any point in the domain than a finite constant. More formally, a function is Lipschitz continuous if and only if there is a positive constant $K$, called the Lipschitz constant, such that

$$ |f(x_1) - f(x_2)| \leq K|x_1 - x_2|, \text{ for all } x_1 \text{ and } x_2 \text{ in the domain } X $$

A function that is Lipschitz continuous over a specified domain is shown in Figure 2.1.

![Figure 2.1 - Lipschitz continuous function, with Lipschitz constant K, such that for any point within the domain, the slope is within ±K](image)
2.1.1 Examples of Lipschitz continuous functions

- \( f(x) = |x| \) is not differentiable at \( x = 0 \), but is continuous, and has a Lipschitz constant of \( K = 1 \).
- \( f(x) = \sin(x) \) is Lipschitz continuous because its slope, the function's derivative \( f'(x) = \cos(x) \), is bounded by 1. So the Lipschitz constant is \( K = 1 \).
- \( f(x) = \sqrt{x^2 + 5} \) is differentiable, and the absolute value of the derivative is bounded by 1. So \( K = 1 \).

![Functions](image)

Figure 2.2 - Functions that are Lipschitz continuous over the real domain

2.1.2 Examples of Continuous Functions that are not Lipschitz Continuous

- \( f(x) = \sqrt{x} \) is continuous, but not Lipschitz continuous, because the slope becomes infinitely steep as \( x \to 0 \).
- \( f(x) = e^x \) is also continuous, but the slope becomes infinitely steep as \( x \to +\infty \).
- \( f(x) = x^2 \) is continuous along the domain of all real numbers, but is not Lipschitz continuous, because the slope becomes infinitely steep as \( x \to \pm\infty \).

![Functions](image)

Figure 2.3 - Functions that are not Lipschitz continuous over the real domain
2.2 Lipschitz Optimization in One Dimension

Lipschitz optimization requires the knowledge of the Lipschitz constant $K$. Finding the true Lipschitz constant is as hard as solving the optimization problem itself [9]. The Lipschitz optimization algorithm was developed independently by Piyawskii [10] and Shubert [11]. It is based on the assumption that the Lipschitz constant $K$ is known. Shubert's algorithm is described here, with a brief overview, followed by detailed steps and an illustrative example.

The Shubert algorithm finds the intersection point $B$ of the lines with slopes of $-K$ and $+K$, originating at the start and end points of the interval respectively, as shown in Figure 2.4 above. It assumes that this intersection point is a lower bound of the objective function. This assumption is true as long as the estimated Lipschitz constant $K$ used is not less than the true Lipschitz constant. Then the algorithm evaluates the objective function at this intersection point $(X, B)$ and divides the interval into two sub intervals, and determines the objective function estimate, $B$, for each new sub interval. Then the interval with the lowest $B$ value is further divided. The algorithm stops when the estimated objective function value, $B$, is within a specified tolerance, $E$, of the true objective function value, $f$, at any point. The algorithm steps below and the example given illustrate how the algorithm works.
2.2.1 Algorithm Steps

First, define $B(a, b)$ (Equation 2.1) as the estimate of the best objective function value contained in interval $[a, b]$. Also define $X(a, b)$ (Equation 2.2) as the x value at the estimate for the best objective function value contained in interval $[a, b]$.

\[
B(a, b, f, K) = \frac{f(a) + f(b)}{2} - K \cdot \frac{b - a}{2} \tag{2.1}
\]

\[
X(a, b, f, K) = \frac{a + b}{2} + \frac{(f(a) - f(b))}{2K} \tag{2.2}
\]

Initialize

1. Evaluate the function at the bounds, $f(a)$ and $f(b)$. Update $f_{\text{min}} = \min(f(a), f(b))$
2. Calculate $B$ for the entire domain interval $[a, b]$ (Equation 2.1).
3. Add this interval to a list of intervals $L$.

Iterate

1. Remove the interval from the list $L$ with the least $B$ value.
2. Evaluate the objective function at point $X$ (Equation 2.2).
   Update $f_{\text{min}} = \min(f_{\text{min}}, f(X))$.
3. Split this interval at point $X$ to yield two intervals $[a, X]$ and $[X, b]$
4. Calculate $B$ for each new interval.
5. Add each new interval to the list of intervals $L$.
6. Stop when the difference between the best objective function value $f_{\text{min}}$, and the minimum $B$ value, are within a predefined tolerance $E$ (stop when $f_{\text{min}} - B_{\text{min}} < E$).

2.2.2 One Dimensional Example

The following example illustrates the Shubert algorithm to optimize the objective function $f(x) = \frac{(x - 8)^2}{16} + 16$ on the interval $[0, 24]$. Assume the Lipschitz constant is $K = 2$ and stopping tolerance is $E = 0.5$. 
1. Start by assuming the correct $K$ value is known. To initialize the algorithm, evaluate the objective function at the endpoints $a$ and $b$. Then calculate the point $(X(a,b,f,K), B(a,b,f,K))$. These three points create a rudimentary estimation of the objective function, as seen in Figure 2.5a to the right. Finally, add the interval $[a, b]$ to the list of intervals $L$.

$$a = 0, f(a) = 20, \quad b = 24, f(b) = 32, \quad K = 2 \quad \{[0, 24]\}$$

$$X_0(0, 24, f, 2) = 9, \quad B_0(0, 24, f, 2) = 2$$

2. Pick the interval from the list $L$ with the least $B$ value. In this case, there is only one interval. Evaluate the objective function at the $X$ point for that interval. Divide the interval into two intervals $[a, X]$, $[X, b]$. Calculate the point $(X, B)$ for each of the new intervals. Then add both of new intervals to the list of intervals.

$$X_0 = 9, f(X_0) = 16.0625 \quad \{[0, 9], [9, 24]\}$$

$$X_1(0, 9, f, 2) = 5.48, \quad B_1(0, 9, f, 2) = 9.03$$

$$X_2(9, 24, f, 2) = 12.51, \quad B_2(9, 24, f, 2) = 9.03$$

3. Again, pick the interval with the least $B$ value. Now there are two intervals with the same $B$ value. Pick the interval with the least $a$ value. Evaluate $(X, f(X))$ for the selected interval. Divide the interval into two new intervals. Calculate the point $(X, B)$ for the new intervals. Then add the new intervals to the list of intervals.

$$X_1 = 5.48, f(X_1) = 16.39 \quad \{[0, 5.48], [5.48, 9], [9, 24]\}$$

$$X_3(0, 5.48, f, 2) = 5.48, \quad B_3(0, 5.48, f, 2) = 12.71$$

$$X_4(5.48, 9, f, 2) = 12.51, \quad B_4(5.48, 9, f, 2) = 12.71$$

$$\cdots$$

4. In the final iteration, the evaluated points and estimation points have indeed become a better estimate of the true objective function as seen to the right in Figure 2.5d. The algorithm stops when the minimum estimation point $B_{\min}$ is within the preset tolerance $E$ of the minimum objective function evaluated so far $f_{\min}$.

(Stop when $f_{\min} - B_{\min} < \epsilon$).

$$f_{\min} = 16.004, \quad B_{\min} = 15.595$$

$$f_{\min} - B_{\min} = 0.419 < E = 0.5$$
2.2.3 Disadvantages of Lipschitz Optimization

Lipschitz optimization has two main disadvantages:

1. Initializing the algorithm requires the storage and evaluation of the objective function at the corners of the search domain, which requires $2^N$ function evaluations in the first iteration, where $N$ is the number of dimensions. This many function evaluations makes the initialization very time consuming in high dimensional problems. Furthermore, for some problems, a single evaluation of the objective function takes a very long time to evaluate. This is especially true if the objective function uses iterative methods or simulations.

2. In many applications, the Lipschitz constant is unknown. Since finding the correct Lipschitz constant is as difficult as solving the optimization problem itself [9], estimates of the Lipschitz constant are often used. However, an incorrect estimate of the Lipschitz constant may lead to extremely slow convergence [12].

2.3 DIRECT Algorithm in One Dimension

The DIRECT algorithm was introduced by Jones, Perttunen, and Stuckman in their paper "Lipschitzian Optimization without the Lipschitz Constant" [1] specifically to overcome the shortcomings of Lipschitz optimization mentioned previously. This section describes the one-dimensional DIRECT algorithm, and how it differs from the one-dimensional Lipschitz optimization.

2.3.1 Division in One Dimension

Given an initial search interval, the DIRECT algorithm first samples the objective function in the middle of the interval, and then to the left and right of the midpoint at equal
distances of one third of the interval length, ±ℓ/3. These sample points will be the centers for new sub intervals that are one third of the size of the parent interval, as shown below in Figure 2.6.

Lipschitz Optimization

DIRECT Algorithm

![Comparison of Lipschitz and DIRECT Divisions](image)

As shown in Figure 2.6, one of the main differences between the DIRECT algorithm and Lipschitz optimization is that DIRECT divides the specified interval into three subintervals of equal length, instead of two subintervals of non-equal length as in Lipschitz optimization. A major advantage of DIRECT is that only one function evaluation is needed at the center of the hyperrectangle, instead of having to evaluate the endpoints. This makes the DIRECT algorithm easier to extend into higher dimensions. Furthermore, the sampling of the midpoint, and always dividing into thirds, eliminates the use of the Lipschitz constant, which, as stated earlier, can be hard or impossible to determine in some cases [1] [9].
2.3.2 Potentially Optimum Intervals

When choosing the interval to further divide, the DIRECT algorithm does not simply pick the interval with the most optimal function evaluation. The DIRECT method chooses multiple "potentially optimal" intervals each iteration based on both the objective function value and the size of the interval. This section will first describe and illustrate the selection process, and then afterward give the formal definition of "potentially optimal intervals."

To illustrate how the DIRECT algorithm identifies potentially optimal intervals, first the intervals are plotted with the size (in this case the interval length) along the x-axis, and the objective function value is plotted along the y-axis. Then by examining this graph, all of the intervals (represented by points) along the lower right of the convex hull of these data points are chosen as potentially optimal, as shown in Figure 2.7 below.

![Graphical interpretation of "potentially optimal" interval selection](figure2.7.png)

Figure 2.7 - Graphical interpretation of "potentially optimal" interval selection, with intervals represented as points, and the "potentially optimal" intervals along the delineated convex hull
Formally stated, suppose that the initial interval is divided into \( m \) sub intervals \([a_i, b_i]\) with midpoints \( c_i \), for \( i = 1, 2, \ldots, m \). Define \( f_i \) as the objective function value evaluated at the midpoint of interval \( i \), so that \( f_i = f(c_i) \), and define \( r_i \) as the distance from the center point \( c_i \) of the interval to one of the end points (\( a_i \) or \( b_i \)) of interval \( i \), which is the length of interval divided by 2, so that \( r_i = (b_i - a_i)/2 \) (\( r_i \) is called the radius or the size of the interval). The interval \( j \) is said to be "potentially optimal" if there exists some rate-of-change constant \( K' > 0 \) such that:

\[
f_j - K' d_j \geq f_i - K' r_i, \quad \text{for all } i = 1, 2, \ldots, m
\]

(2.3)

Again, this formal statement simply makes sure that the potentially optimal intervals are on the lower right convex hull of the objective function value vs. interval size graph.

Here are some generalizations about selecting the potentially optimal intervals as described so far:

1. If all of the intervals are the same size at a given iteration, the DIRECT algorithm will pick the interval(s) with the best objective function value.

2. If there are different interval sizes at a given iteration, the DIRECT algorithm will always pick at least the interval(s) with the best objective function value overall, as well as the interval(s) that have the best objective function value among the intervals with the longest length.

The second point, shows that the DIRECT method is trying to balance local search, by exploiting the intervals with the best objective function value overall, as well as global search, by choosing to further divide intervals with the longest length so that more area is explored.
2.3.2.1 Comparison of Interval Selection Methods for Lipschitz Optimization and the DIRECT Algorithm

The original DIRECT paper [1] shows that Lipschitz optimization implicitly does this same kind of "potentially optimal interval" selection, by positioning a line with a fixed slope of the Lipschitz constant, $K$, below all of the data points, then shifting this line upwards until it touches one of the points. The algorithm then divides the interval represented by this point. This is illustrated below in Figure 2.8. The similarity in the two algorithms for one dimension is paraphrased in the original DIRECT paper [1] as follows: "The one dimensional DIRECT algorithm is essentially Shubert's algorithm modified to use center-point sampling and to sample all potentially optimal intervals during an iteration."

Indeed, some of the main characteristics of Lipschitz optimization—the recursive division, and the incorporation of global and local search—are retained in the DIRECT method, even though many other properties of the algorithm have changed.

![Graphical interpretation of Lipschitz interval selection](image)

Figure 2.8 - Graphical interpretation of Lipschitz interval selection, with the Lipschitz constant, $K$, represented as a line, and the selected interval as a point along this line
2.3.3 $\varepsilon$ Parameter

A purely local search would only divide the intervals with the best objective function value. A purely global search would divide only the longest intervals until none of that size group is left, and then continue to divide the next size group until none of that size group is left. The global approach would be very similar to a uniform division of the search space. The DIRECT algorithm tries to balance local and global searching by dividing intervals of various size groups in the same iteration.

The $\varepsilon$ parameter pushes the DIRECT algorithm more towards a global search by stopping the algorithm from dividing intervals that are close in objective function value to the best function value evaluated so far. In the objective function value vs. interval size plot illustrated earlier in Figure 2.7, this parameter would make sure that the slope $K'$ between two convex hull points is greater than $\varepsilon$. If the slope is not greater than $\varepsilon$, then this interval will not be further divided in the current iteration. This can be seen in Figure 2.9 below, where the lowest point on the bottom left is excluded from the "potentially optimal" intervals. It is excluded, even though it is on the convex hull, because the slope between this point and the next potentially optimal interval on the convex hull is not greater than the $\varepsilon$ parameter.
The introduction of the $\varepsilon$ parameter modifies the previous formal definition of potentially optimal intervals, by adding another condition. In addition to the definitions given above, further define $f_{\text{min}}$ as the minimum objective function value evaluated so far, and define $\varepsilon$ as a positive constant. The interval $j$ is said to be "potentially optimal" if there exists some rate-of-change constant $K' > 0$ such that:

$$f_j - K' r_j \geq f_i - K' r_i, \quad \text{for all } i = 1, 2, \ldots, m \quad (2.3)$$

$$f_j - K' r_j \geq f_{\text{min}} - \varepsilon |f_{\text{min}}| \quad (2.4)$$

### 2.3.4 Algorithm Steps

First, a discussion of how to store the rectangles (or intervals in this case) will be given. Then the steps will be given for the one-dimensional DIRECT algorithm, followed by an example in Section 2.3.5.
2.3.4.1 Rectangle Storage

The process of determining the potentially optimal rectangles requires a specialized data structure to be algorithmically efficient. All of the rectangles must be organized by size first, then by objective function value. He et al. [7] describes a data structure to store the potentially optimal rectangles. For this research, the rectangles are organized as a list of lists. The primary list contains the secondary lists and it is indexed by rectangle size. More specifically, the number of times, $T$, the rectangle has been divided can be used as an integer index for the secondary lists, since $T$ is directly proportional to the rectangle size. The secondary lists (referred to as size groups) contain the rectangles of the same size and are sorted by objective function value evaluated at the midpoint. This type of data structure is shown in Figure 2.10 below.

![Rectangle Storage for Potentially Optimal Determination](image)

Now, only the rectangles with the best objective function values per size group need to be checked against Conditions 2.3 and 2.4 for being potentially optimal. However, it is possible for multiple rectangles of the same size to have the same objective function value. Therefore, all of the rectangles in a size group with the best objective function value must be checked. Since, the secondary lists are sorted, all of the rectangles with the best objective function value will be at the front of the list.
2.3.4.2 DIRECT One Dimensional Steps

Given a data structure, similar to the one shown in Figure 2.10, to store the intervals (called SizeRects or "the list of all rectangles"), here are the steps for the one-dimensional DIRECT algorithm.

Initialize

1. Sample the midpoint of the entire search space.
2. Add the initial interval to SizeRects.

Iterate

1. Determine the Set $P$ of all potentially optimal intervals in SizeRects. Remove Set $P$ from SizeRects.
2. For each interval $I$ in Set $P$:
   a. Evaluate points ±1/3 the length of interval $I$ from the midpoint of $I$. Increment the function counter $m$ by 2.
   b. Divide $I$ into thirds, such that the points evaluated previously are now the centers of these new intervals.
   c. Add the new intervals into a temporary Set $R$.
4. Stop when the evaluation counter $m$ is greater than the maximum number of function evaluations $m_{\text{max}}$, or when the iteration count is greater than the maximum number of iterations $i_{\text{max}}$ (Stop when $i > i_{\text{max}}$ or $m > m_{\text{max}}$).

2.3.5 One Dimensional Example

The following example shows the first four iterations of the one-dimensional DIRECT algorithm to optimize the objective function $f(x) = (x - 8)^2/16 + 16$ on the interval $[0, 27]$. Assume $\epsilon = 0.001$. The objective function value vs. interval size graphs show the size in log scale, and the size has been normalized to $3^{-k}/2$, where for one-dimension $k$ is the
number times the interval has been divided. Since the graph has a log scale, the straight lines on the convex hull are not accurate, but serve as illustrations.

1. Sample the middle of the search space. Add the entire search interval to the list of all rectangles.

   \[ a_0 = 0, \quad b_0 = 27 \]
   \[ c_0 = \frac{(27 - 0)}{2} = 13.5 \]
   \[ f(c_0) = 17.89 \]
   \[ \{[0, 27]\} \]

2. Pick the potentially optimal intervals, (there is only one interval at this point), then sample to the left and right, then divide. Add these sub intervals to the list.

   \[ c_1 = 4.5, \quad f(c_1) = 16.78 \]
   \[ c_2 = 22.5, \quad f(c_2) = 29.14 \]
   \[ \{[0, 9], [9, 18], [18, 27]\} \]

3. Pick the potentially optimal intervals, there is only one size group. Pick the best intervals in this size group. Sample, divide, and add to list.

   \[ c_3 = 1.5, \quad f(c_3) = 18.64 \]
   \[ c_4 = 7.5, \quad f(c_4) = 16.02 \]
   \[ \{[9, 18], [18, 27]\} \]
   \[ \{[6, 9], [3, 6], [0, 3]\} \]

4. Pick the potentially optimal intervals, checking the conditions. The smallest size group is ignored because of \( \varepsilon \) and Condition 2.4.

   \[ c_5 = 6.5, \quad f(c_5) = 16.14 \]
   \[ c_6 = 8.5, \quad f(c_6) = 16.02 \]
   \[ c_7 = 10.5, \quad f(c_7) = 16.39 \]
   \[ c_8 = 16.5, \quad f(c_8) = 20.52 \]

   \[ \{[18, 27]\} \]
   \[ \{[9,12],[3,6],[12,15],[0,3],[15,18]\} \]
   \[ \{[8, 9], [7, 8], [6, 7]\} \]

---

Figure 2.11 - One Dimensional Example of DIRECT Optimization
2.4 DIRECT Algorithm in Two or More Dimensions

The main difference between the one-dimensional DIRECT algorithm and the multi-dimensional DIRECT algorithm is how the potentially optimal hyperrectangles are divided. There are two other small differences. One difference is that, in the initialization of the algorithm, the search space is normalized to a unit hypercube. The second difference is the definition of rectangle size. These details are explained in this section.

2.4.1 Normalization to a Unit Hypercube

Normalization is performed to convert all of the variable ranges from their original upper and lower bounds to a range between 0 and 1. Conversion back to the original values only needs to be performed when calling the objective function. Normalizing the variables means that the side lengths of each rectangle will all be multiples of 1/3. This step is supposed to make the division calculations simpler.

2.4.2 Division in Two or More Dimensions

2.4.2.1 Division of a Hypercube

The original DIRECT algorithm divides the hypercube into thirds along every dimension around the center point. It also divides the hypercube along all of the dimensions in a specific order in an attempt to avoid premature, local convergence.

First, the algorithm samples the objective function at points around the center point, plus and minus one-third of the cube's side length in every dimension. Formally stated, it samples points \( \mathbf{c} \pm \delta \mathbf{u}_i \) for all \( i = 1, 2, \ldots, n \), where \( \mathbf{c} \) is the center point of the hypercube, \( \delta \) is one-third the side length of the hypercube, \( n \) is the number of dimensions, and \( \mathbf{u}_i \) is a unit vector in the \( i \)th dimension (i.e. a vector with all zeros, except for a one in the \( i \)th dimension).
Now with all of these samples evaluated, DIRECT sorts the dimensions based on the best out of the two samples per dimension. Then, the algorithm divides the hypercube in thirds along the dimension with the best objective function value first, and then it recursively divides the center rectangle along the subsequent dimensions in order of their best function value. This means that the dimension with the best objective function value will have the largest rectangles. The divisions are performed in this order to allow the area with the best objective function value to be searched more. If the dimensions were divided in the reverse order, the dimension with the best objective value would have the smallest rectangle, and this could lead to premature convergence at a local optimum. Figure 2.12 shows a hypercube sampled and divided in two dimensions. The objective function values are shown above the sample point.

![Hypercube Division](image)

**Figure 2.12 - DIRECT sampling and division in two dimensions with the objective function value shown above each point**

### 2.4.2.2 Division of a Hyperrectangle

Once the initial hypercube is divided, there will be many hyperrectangles as well as smaller hypercubes. The hyperrectangles are divided in a similar manor as the hypercube, except that only the longest sides of the rectangle are sampled, sorted, and divided.
2.4.2.3 Division Steps

The steps for dividing potentially optimal hyperrectangles are described below.

These include the hypercube as a special case of hyperrectangle.

1. Identify the set $I$ of dimensions for the potentially optimal hyperrectangle with the longest side length. Define $\delta$ as one-third of the longest side length.

2. Evaluate the objective function at the points $c \pm \delta u_i$ for each dimension $i$ in the set $I$, where $c$ is the center point of the potentially optimal hyperrectangle and $u_i$ is the unit vector in the $i$th dimension.

3. Sort the dimensions in set $I$ by the lowest objective function value $v_i$ for each dimension, from the least to the greatest.

   \[ v_i = \min\{f(c + \delta u_i), f(c - \delta u_i)\} \]

4. Divide the potentially optimal rectangle along each of the dimensions in set $I$ in the sorted order from the least function value $v_i$ to the greatest, such that the dimension with the least objective function value has the largest sub rectangles.

2.4.3 Rectangle Size

In the one-dimensional DIRECT, the interval is essentially the one-dimensional rectangle, and its size is measured by the distance from the center point to the end point of the interval $i$, which is the same as the length of the interval divided by 2, $r_i = (b_i - a_i)/2$.

In the multi-dimensional DIRECT, the definition of the size of a rectangle is slightly modified to be the distance from the center point of rectangle $i$ to any one of the corner points of the rectangle, and the rectangle size will still be represented by $r_i$. 

---

21
2.4.4 Algorithm Steps

The algorithm steps have not significantly changed from the one dimension algorithm except for the addition of normalization in the initialization, and a more complicated division strategy. The procedure for determining potentially optimal rectangles is still the same, only with a slightly modified definition of rectangle size. The objective function value vs. rectangle size plot and lower right convex hull strategy still holds. Below are the steps for the multi-dimensional DIRECT algorithm, given a data structure for organizing rectangles called SizeRects as described earlier in Section 2.3.4.1.

Initialize

1. Normalize all variables to the unit hypercube.
2. Sample the midpoint of the entire search space.
3. Add the initial rectangle to SizeRects

Iterate

1. Determine the Set $P$ of all potentially optimal intervals in SizeRects. Remove Set $P$ from SizeRects.
2. For each Rectangle $R$ in Set $P$:
   a. Sample and evaluate points $c \pm \delta u_i$ for all dimensions $i = 1, 2, \ldots n$ Increment the function counter $m$ by $2 \cdot n$.
   b. Divide Rectangle $R$ into thirds along every dimension, such that the points evaluated previously are now the centers of these new rectangles.
   c. Add the new rectangles into a temporary Set $T$
4. Stop when the evaluation counter $m$ is greater than the maximum number of function evaluations $m_{max}$, or when the iteration count is greater than the maximum number of iterations $i_{max}$ (Stop when $i > i_{max}$ or $m > m_{max}$).
2.4.5 Two Dimensional Example

This two dimensional example shows the first three iterations of the DIRECT algorithm to optimize the objective function \( f(x) = (x_1 - 8)^2/16 + (x_2 - 6)^2/16 + 16 \) in the domain \( 0 \leq x_i \leq 27 \) for \( i \in \{1,2\} \). This function is quadratic, and is graphed in Figure 2.13. When displaying the SizeRects structure, the rectangles are represented by their center point. Also, the grayed rectangles indicate that they will be picked as potentially optimal in the next iteration.

\[
f(x) = (x_1 - 8)^2/16 + (x_2 - 6)^2/16 + 16
\]

Figure 2.13 - Objective Function for Two Dimensional DIRECT Example
1. Sample the middle of the search space. Add the initial hypercube to the list of all rectangles.

\{(13.5,13.5)\}

![Figure 2.14a](image)

2. Pick the potentially optimal intervals (there is currently only one rectangle), then sample $c = \delta u_i$, then divide. Add these sub intervals to the list of all rectangles.

\{(13.5,4.5),(13.5,22.5)\}
\{(4.5, 13.5),(13.5,13.5),(22.5,13.5)\}

![Figure 2.14c](image)

![Figure 2.14d](image)

3. Pick the potentially optimal intervals, there is only one size group. Pick the best intervals in this size group. Sample, divide, and add to list.

\{(13.5,22.5)\}
\{(4.5,4.5),(13.5,4.5),(4.5, 13.5), (13.5,13.5),(22.5,4.5), (22.5, 13.5)\}

![Figure 2.14e](image)

![Figure 2.14f](image)

Figure 2.14 - Two Dimensional Example of DIRECT Optimization

2.5 Changes to the DIRECT Algorithm from Prior Research

There have been many changes to the DIRECT method reported in the literature to improve efficiency or to adapt DIRECT to a specific application. Modifications to the DIRECT algorithm relevant to this research are discussed here.

2.5.1 Aggressive DIRECT

The original DIRECT algorithm does not divide the same number of rectangles every iteration. Usually the number of rectangles considered "potentially optimal" grows each
iteration. This makes DIRECT difficult to scale for parallel performance. The Aggressive DIRECT algorithm (Aggressive DIRECT) was introduced by Baker et al. [13] specifically to improve parallel performance. It does this by changing the definition of potentially optimal. Instead of using the convex hull strategy, Aggressive DIRECT divides all of the rectangles that have the best objective function value for each size group, as shown below in Figure 2.15. The number of rectangles divided still grows with each iteration, but the number of rectangles (and thus the number of function evaluations) is very large except in the early iterations. Therefore, the Aggressive DIRECT algorithm becomes almost embarrassingly parallel, and can yield good parallel performance scalability.

![Graphical interpretation of the rectangle selection using Aggressive DIRECT](image)

**Figure 2.15** - Graphical interpretation of the rectangle selection using Aggressive DIRECT with most optimal rectangle of each size group selected and delineated in the plot

### 2.5.2 Dividing Only One Dimension

In a later article [3], Jones suggested, instead of the more complicated division strategy in the original DIRECT algorithm, to simply divide the hyperrectangle along only
one of the longest sides. Apparently, there is little to no improvement gained by recursively dividing all of the dimensions and using the dimension sorting procedure described in Section 2.4.2.3. Furthermore, dividing only one dimension at a time saves function evaluations and time used in sorting the dimensions. In the words of Jones: "Experience has since shown, however, that the robustness benefit is small and that trisecting on a single long side (as here) accelerates convergence in higher dimensions."

Jones also gives a procedure for breaking ties among the rectangles longer sides. He suggests keeping a counter $t_i$ of how often each dimension $i$ has been divided in the entire search (the counter is for the division of any rectangle along dimension $i$). To decide between the longer sides of the hyperrectangle choose the side with the lowest division count $t_i$, and if two of the long sides have the same division count, then choose the side with the lowest dimension index $i$.

### 2.5.3 Generalized Definition of Rectangle Side Length and Radius

A generalized version of the rectangle side length and the rectangle radius (distance from the center point one of the vertices) is first described in the Jones article [3]. It is further detailed in a thesis by Gablonsky [2]. The generalized definitions for rectangle side length and radius are described below, as well as how the radius definition affects the selection of "potentially optimal" rectangles.

#### 2.5.3.1 Side Length

Every time that a rectangle is trisected, the length of that side is reduced by a factor of 3 (i.e. multiplied by a factor of $1/3$). Since all of the variables are normalized to a unit hypercube at the beginning of the algorithm, the side length and center-point-to-vertex
distance (the radius) can be generalized. First, define $T$ as the total number of times a particular rectangle has been trisected, and define $N$ as the total number of dimensions. Then define the level as $k = \lceil T / N \rceil$ and the stage as $j = \text{mod}(T, N)$. It can be shown that $T = k \cdot N + j$. Before explaining the meaning of the level $k$ and stage $j$, it is important to note that for any given rectangle used in the DIRECT method, there are only two side lengths, a longer side length and a shorter side length. This is because the DIRECT method always divides one of the longer sides. The longer side length can be represented as $\ell = 3^k$, and the shorter side length will be $\ell_s = 3^{(k+1)}$. Conceptually, the level, $k$, is the number of times the longest side has been trisected, and the stage, $j$, is a count of the shorter sides for that particular rectangle.

### 2.5.3.2 Radius

The radius of the rectangle can also be abstracted. Using simple algebra, the radius $r$ (Equation 2.8) is determined from the level $k$ (Equation 2.5) and the stage $j$ (Equation 2.6) as shown below:

\begin{align*}
  k &= \lceil T / N \rceil \\
  j &= \text{mod}(T, N) \\
  T &= k \cdot N + j
\end{align*}

\[ r = \frac{3^{-k}}{2} \left( \frac{1}{9} + N - j \right)^{0.5} = \frac{3^{-k}}{2} \sqrt{N - \frac{8j}{9}} \]

These equations mean that the length of the longest and shortest sides, as well as the rectangle radius, can all be determined from the number of dimensions in the problem $N$, and the number of times the rectangle has been divided $T$. 

27
2.5.3.3 Rectangle Size Definition

In the original DIRECT algorithm, the rectangle size is synonymous with the rectangle radius $r$ (i.e. the center-to-vertex distance), which directly corresponds to the number of times the rectangle has been divided $T$. However, in later research by Gablonsky [2], this size definition was changed to be the level $k$ of the rectangle. The level of the rectangle was defined in the previous section to be the number of times the longest side of the rectangle has been divided $k = \lfloor T / N \rfloor$ (Equation 2.5). The definition of rectangle size was changed in order to group more rectangles in a size group, so that there would be fewer potentially optimal rectangles chosen per iteration. Gablonsky took the opposite approach to the Aggressive DIRECT algorithm, in that he was trying to reduce the number of potentially optimal rectangles per iteration. The change in size definition was found to reduce the total number of function evaluations needed to find the optimal solution for certain test functions. However, it did increase the number of iterations.

2.5.4 Only One Minimum Rectangle per Size Group

If two or more rectangles have the same size and the same objective function value, they will occupy the same point on the objective function value vs. rectangle size graph. And if this point is on the convex hull, the original DIRECT algorithm would divide all of these rectangles. Gablonsky suggested picking only one of the rectangles per point as potentially optimal. This may decrease the number of function evaluations taken to find the global minimum.
2.5.5 Integer Variables

In the same article where Jones suggests dividing only one dimension and generalizes the definition of rectangle radius and side length [3], he also modified the algorithm to handle integers and inequality constraints. Jones mentions two minor changes that have to be made to make DIRECT work with integers: one is in the way the midpoint of the rectangle is defined, and the second is in the trisect routine.

2.5.5.1 Integer Midpoint

Redefining the midpoint is trivial. Jones defines it as the floor of algebraic average for each dimension. For example, for the interval [1, 8], the integer midpoint cannot be 4.5, which is \((b - a)/2\) as previously defined. So the integer midpoint is defined as \(\lfloor (b - a)/2 \rfloor\) for the interval \([a, b]\) (in this example 4). Therefore, the integer midpoint for multiple dimensions is the floor of algebraic average for the interval of each dimension.

2.5.5.2 Trisecting Integers

Jones gives a specific procedure for dividing the integer dimensions. First, compute \(\Delta = \lfloor (b - a + 1)/3 \rfloor\). If \(\Delta > 1\), then the first sub interval will be \([a, a + \Delta - 1]\), the center sub interval will be \([a + \Delta, b - \Delta]\), and the third sub interval will be \([b - \Delta + 1, b]\). If \(\Delta = 0\), then this particular integer dimension must have a range of only two integers. In this case, the center sub interval will be \([a, a]\) and the third sub interval will be \([b, b]\), and the first sub interval will be empty and can be ignored. This is to keep with the convention that the midpoint of the parent rectangle be the midpoint of the center child rectangle.
2.5.5.3 Resolving Complications

Using integer variables yields three more complications, which Jones addresses in the article:

1. Because the midpoint of the integer dimension will not truly be at the geometric center, the center-to-vertex distance is not the same from the midpoint to all of the vertices. This is resolved by ignoring the actual center-to-vertex distance, and instead using the radius formula (Equation 2.8) given above, which is based solely on the number of dimensions and the number of times the rectangle has been divided.

2. The floored integer division procedure yields side lengths that are not precisely in the $\ell = 3^k$ format. This means that sides that have been divided the same number of times will not necessarily be the same length. The side length is mainly important in determining which sides are the longest and choosing the dimension to divide. This problem is solved by relaxing the definition of longest sides to mean those sides that have been divided the least number of times. Furthermore, if a side has a length of 0, meaning it only has one integer within that side's range $[a, a]$, it will not be further divided.

3. Finally, if all of the variables are integers, then a rectangle could be reduced to a single point. In this case, the rectangle will be ignored from the "potentially optimal" selection process and will not be divided anymore.

2.5.6 Adaptive $\varepsilon$ Parameter

Finkel [12] introduced a version of DIRECT that adaptively changes the $\varepsilon$ value. It starts with a value of $\varepsilon = 0$, which is searching locally as well as globally. If the DIRECT method starts to show no improvement, then it means that the algorithm is probably stuck in
a local optimum, and the algorithm will change the $\varepsilon$ parameter to a higher value to induce a more global search. If there is no improvement, with the higher $\varepsilon$ value, then the algorithm switches back to a more local search with $\varepsilon = 0$.

The adaptive $\varepsilon$ parameter is not currently incorporated into this research.
Chapter 3 The DIRECT Algorithm for the Generic Variable Type

The motivation for an abstract variable type is to be able to use other discrete variable types with the DIRECT method, specifically to use nodes in a water distribution system (WDS), which are essentially graph nodes. To be able to use DIRECT with graph nodes, more abstraction and generalization is needed, in the same manner that Jones abstracted the DIRECT method for integer variables, and mixed real and integer problems. This chapter details the generalizations that are made.

3.1 Generalized Variable Type

This section develops an interface for the generic variable type. An interface is used in computer programming as a class with abstract functions that are defined by the implementing classes, like a header file in C or C++. The interface is a template that needs the specific implementations to be further defined by the child objects. Such an interface is given in Table 3.1.

As Jones noted when changing the DIRECT method to handle integer types, the two main considerations are how to handle the division and how to select a midpoint of the variable range. These will be the first two generalizations discussed in this section.

Ideally, this abstraction would allow for the use of any variable type, as long as the members can be intelligently divided or grouped, and a representative sample taken. It is even possible that a binary variable type could be used. If all the variables are binary, however, the algorithm would essentially be bisecting a binary search tree and the algorithm would not be expected to perform well. The binary variable had not been implemented in this research.
It is important to note that a side of the rectangle is a dimension of the rectangle, and both represent a variable being search by the DIRECT algorithm. Therefore the words variable, side, and dimension are essentially synonymous in this discussion.

Table 3.1 - General Variable Interface

<table>
<thead>
<tr>
<th>Interface</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>interface Variable</td>
</tr>
<tr>
<td>2:</td>
<td>properties</td>
</tr>
<tr>
<td>3:</td>
<td>int</td>
</tr>
<tr>
<td>4:</td>
<td>end properties</td>
</tr>
<tr>
<td>5:</td>
<td>methods</td>
</tr>
<tr>
<td>6:</td>
<td>Variable[]</td>
</tr>
<tr>
<td>7:</td>
<td>float</td>
</tr>
<tr>
<td>8:</td>
<td>boolean</td>
</tr>
<tr>
<td>9:</td>
<td>Variable</td>
</tr>
<tr>
<td>10:</td>
<td>end methods</td>
</tr>
<tr>
<td>11:</td>
<td>end interface</td>
</tr>
</tbody>
</table>

3.1.1 Division

To generalize the division process, every variable type must have its own trisect method which returns three new child variables that are subdivisions of the original variable. The DIRECT method does not depend on how the variable division is taking place. Table 3.2 shows examples of how the division can take place for different variable types. If the variable range does not contain three points (as can be the case with discrete variable types), then it can return the two new variables and leave the third as NULL, as described for integers earlier. Again, the purpose of the interface is to make the DIRECT optimization algorithm independent from the division process for specific variable types, and therefore the variable division procedures are not hard-coded as a part of the DIRECT algorithm.
Each time a variable is divided, the divcount counter should be incremented in each of its child variables. That way this counter is an accurate count of the number of times a variable has been divided.

Table 3.2 - Possible Division Procedures for Various Variable Types

<table>
<thead>
<tr>
<th>Variable Type</th>
<th>Possible Division Procedures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real</td>
<td>A real variable range ([a, b]) can be divided into three child variables ([a, a + \Delta], [a + \Delta, b - \Delta], [b - \Delta, b]), where (\Delta = (b - a)/3)</td>
</tr>
<tr>
<td>Integer</td>
<td>An integer variable range ([a, b]) can be divided into three child variables ([a, a + \Delta - 1], [a + \Delta, b - \Delta], [b - \Delta + 1, b]), where (\Delta = \lfloor (b - a + 1)/3 \rfloor) (see Section 2.5.5)</td>
</tr>
<tr>
<td>Discrete Set</td>
<td>A discrete set of sorted integer or real numbers can be divided into subsets where each subset has approximately the same number of members per subset. For example, ({1, 2, 4, 8, 16, 32, 64}) can be divided into three subsets ({1, 2}, {4, 8, 16}, {32, 64}).</td>
</tr>
<tr>
<td>Binary</td>
<td>A binary variable can be trisected, so that the third child is NULL. ({0, 1} \rightarrow {0}, {1}, \text{NULL})</td>
</tr>
<tr>
<td>Cartesian Points</td>
<td>A set of Cartesian points can be divided into three subsets based on either the x-coordinate or the y-coordinate, or it can alternate between the dimension it divides each time the variable is divided.</td>
</tr>
<tr>
<td>Graph Nodes</td>
<td>A set of graph nodes can be partitioned based on adjacency information, such as using recursive minimum cuts, or using clustering algorithms.</td>
</tr>
<tr>
<td>Water Distribution System (WDS) Nodes</td>
<td>A set of WDS nodes can be recursively skeletonized, using network skeletonization algorithms, starting from a very coarse skeletonization and then refining the skeletonization as the set is further divided.</td>
</tr>
</tbody>
</table>
3.1.2 Midpoint

   Each Variable also has a midpoint method that returns the midpoint of the variable's range. This method returns a floating-point type, but if the variable is an integer type it can still return an integer cast as a floating point, and if the variable type is a graph node, it can return the node's index (an integer) cast as a floating point.

   This midpoint method is called when the objective function value of the rectangle midpoint of the rectangle is evaluated. The objective function is responsible for casting the floating-point midpoint before use. Since the variables of a rectangle are ordered, the objective function casts and uses each floating-point midpoint number appropriately. For example, if the variable is an integer type variable, the floating-point midpoint will be cast to an integer.

3.1.3 Singularity

   As Jones pointed out in [3], when dealing with integer variables, care must be taken when the size of the variable’s range reduces to below three. Specifically, if there is only one item in the variable’s range, then it should not be divided any more. The issingular method returns a boolean that is true if the variable only contains one item, and it returns false otherwise. When a variable is trisected with only two items, one of the child variables will be NULL.

3.1.4 Copy

   When dividing along one dimension of the rectangle, one variable will be divided in the child rectangles, while the other variables will stay the same. The copy method gives that ability. When trisecting a rectangle, one variable can be trisected into three new variables.
The three child rectangles will each receive one of the new child variables respectively, as well as copies of the other variables that were not divided.

3.2 Generalized Rectangle

The rectangle interface given below in Table 3.3 standardizes the methods available and so that interaction with the various variable types is made clear.

<table>
<thead>
<tr>
<th>Interface</th>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>interface Rectangle</td>
</tr>
<tr>
<td>2:</td>
<td>properties</td>
</tr>
<tr>
<td>3:</td>
<td>int divcount</td>
</tr>
<tr>
<td>4:</td>
<td>int ndims</td>
</tr>
<tr>
<td>5:</td>
<td>end properties</td>
</tr>
<tr>
<td>6:</td>
<td>methods</td>
</tr>
<tr>
<td>7:</td>
<td>Rectangle[] trisect(int n)</td>
</tr>
<tr>
<td>8:</td>
<td>float[] midpoint()</td>
</tr>
<tr>
<td>9:</td>
<td>boolean issingular()</td>
</tr>
<tr>
<td>10:</td>
<td>float radius()</td>
</tr>
<tr>
<td>11:</td>
<td>end methods</td>
</tr>
<tr>
<td>12:</td>
<td>end interface</td>
</tr>
</tbody>
</table>

3.2.1 Division

3.2.1.1 Trisect

The rectangle trisect method divides the current rectangle along the $n$th dimension and returns three child rectangles. As said earlier, when dividing one dimension of the rectangle, the divided dimension will change in the child rectangles, but the other variables will stay the same in the child rectangles. So when trisecting the rectangle, one variable will be trisected into three new variables. The three child rectangles will each receive one of the new child variables respectively, and the rest of the variables will simply be copied.
3.2.2 Midpoint

The midpoint method returns the midpoint of the rectangle. This midpoint can be thought of as a "central" or representative sample of the decision space contained within the rectangle. This midpoint is not necessarily geometrically central, because some of the variable types solved for in the problem may be discrete. The method returns an array of floating-point numbers that is the midpoint of this rectangle. The array is made up of the midpoints of each of the variables that comprise this rectangle. For example, the first element of the array will be the midpoint of the first variable in the rectangle, and the second element will be the midpoint of the second variable, etc. This midpoint is the solution that is evaluated in the objective function, and represents this rectangle when determining the potentially optimal rectangles.

3.2.3 Singularity

The rectangle is considered singular if all of its variables are singular, as previously defined in Section 3.1.3. If the rectangle is singular, it means that there is only one solution contained within the rectangle, and that the rectangle will not be considered for division any longer. However, the objective function value of the singular rectangle will still be considered when determining the best solution found.

3.2.4 Radius

The radius method returns the rectangle radius, \( r \), given in Equation 2.8 which is directly dependent on the dimensionality of the problem, \( N \text{(Rectangle.ndims)} \), and the number of times the rectangle has been divided, \( T \text{(Rectangle.divcount)} \). The rectangle
radius is the measure of rectangle size used when determining the potentially optimal rectangles. It is the x-axis in the objective function value vs. rectangle size plot.

3.3 Test Functions

In the original DIRECT paper by Jones et al. [1], there are numerous test functions given to gauge the performance of DIRECT compared to other algorithms. In the thesis by Gablonsky [2], he also used these test functions to compare his modifications to the original DIRECT algorithm. Table 3.4 below presents some of these test functions. The number of function evaluations to find the known global minimum is given for the original application of the DIRECT method, for Gablonsky's results, and for the results of this research. In this research, the version of the DIRECT algorithm used to solve these test functions divided only one dimension at a time, sorted the rectangles by the longest side (level), and only considered one of the minimum rectangles per size group as potentially optimal, unless noted otherwise.

Table 3.4 shows that the version of DIRECT used in this research was able to outperform both the original DIRECT algorithm and Gablonsky’s version of DIRECT in some cases (indicated in bold). In few cases, this version of DIRECT only outperformed the original DIRECT method, but did not outperform Gablonsky’s version. Finally, this version of DIRECT underperformed both the original DIRECT algorithm and Gablonsky’s DIRECT for the Shekel functions, which have a relatively flat objective space with many local minimum in very sharp spires. The underperformance is most likely due to the differences in division schemes. For instance, the version of DIRECT used in this research divides only one dimension at a time, whereas the original DIRECT algorithm and Gablonsky's DIRECT
divide all the dimensions of a rectangle in one iteration. For most cases, dividing only one
dimension at a time reduces the number of function evaluations, but not in every situation.

Table 3.4 - Function Evaluations to find Global Minimum for Test Functions

<table>
<thead>
<tr>
<th>N Dims</th>
<th>Range</th>
<th>Global Min</th>
<th>1991 Jones et al.</th>
<th>2001 Gablonsky</th>
<th>This Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>2</td>
<td>$[0,1]^2$</td>
<td>0</td>
<td>475</td>
<td>173</td>
</tr>
<tr>
<td>Quadratic</td>
<td>2</td>
<td>$[0,10]^2$</td>
<td>10</td>
<td>139</td>
<td>65</td>
</tr>
<tr>
<td>Branin</td>
<td>2</td>
<td>$[-5,10] \times [0,15]$</td>
<td>0.397887</td>
<td>195</td>
<td>159</td>
</tr>
<tr>
<td>Shekel 5</td>
<td>4</td>
<td>$[0,10]^4$</td>
<td>-10.153</td>
<td>155</td>
<td>147</td>
</tr>
<tr>
<td>Shekel 7</td>
<td>4</td>
<td>$[0,10]^4$</td>
<td>-10.403</td>
<td>145</td>
<td>141</td>
</tr>
<tr>
<td>Shekel 10</td>
<td>4</td>
<td>$[0,10]^4$</td>
<td>-10.536</td>
<td>145</td>
<td>139</td>
</tr>
<tr>
<td>Hartman 3</td>
<td>3</td>
<td>$[0,1]^3$</td>
<td>-3.863</td>
<td>199</td>
<td>111</td>
</tr>
<tr>
<td>Hartman 6</td>
<td>6</td>
<td>$[0,1]^6$</td>
<td>-3.322</td>
<td>571</td>
<td>295</td>
</tr>
<tr>
<td>Goldstein Price</td>
<td>2</td>
<td>$[-2,2]^2$</td>
<td>3</td>
<td>191</td>
<td>115</td>
</tr>
<tr>
<td>Six-hump Camel Back</td>
<td>2</td>
<td>$[-3,3] \times [-2,2]$</td>
<td>-1.03163</td>
<td>285</td>
<td>191</td>
</tr>
<tr>
<td>Shubert</td>
<td>2</td>
<td>$[-10,10]^2$</td>
<td>-186.731</td>
<td>2967</td>
<td>2043</td>
</tr>
</tbody>
</table>

* Stops when the best value is within 0.01% of the known global minimum
** Considered the rectangle size by the radius instead of the longest side length
Chapter 4 Leak Detection in Water Distribution Systems (WDS) using the Dividing Rectangles (DIRECT) Search

This chapter introduces the leak detection problem in WDS (Section 4.1), describes WDS simulation and optimization (Section 4.2), and details the specific problem representations used in this research (Section 4.3).

4.1 Introduction to Leak Detection

4.1.1 Motivation

Water distribution systems are a vital part of modern infrastructure, bringing safe clean drinking water to the public. Yet these systems are susceptible to leaks and contaminant intrusion. High pressures, freezing water, corrosion, and aging can cause cracks in the distribution pipes. It has been estimated that anywhere between 3 and 50 percent of water is lost to leaks in a WDS: three percent in well-maintained systems, and fifty percent in aging systems and in developing countries [8]. Large leaks are usually easy to locate because they can cause significant property damage and flooding, but small leaks gradually lose water into the soil and can be difficult to locate. Leaks cause the pressure to drop in the system, which requires more pumping to maintain the required pressure. If there is negative pressure in the pipe, a leak can become a contaminant intrusion point by leaching chemicals from soil into the water.

4.1.2 Current Approaches

Utilities typically monitor locations that are prone to leaks, based on a history of previous leakage or the age of the pipes. A leak can be detected, for example, by using acoustic listening devices that "hear" the sound of water escaping from the pipe. There are
other field methods, such as thermal imaging and tracer gas analysis. There are also simulation-based approaches, the most common of which is called Inverse Transient Analysis (ITA). However, ITA requires the use of induced pulses (e.g. opening and closing a fire hydrant), and then backward calculation to determine the leak location. Also, ITA is usually used for locating leaks along a straight long pipe, not in a water network.

These current methods, however, are usually expensive and time consuming. This research seeks to use measurements that can be routinely collected, such as the pressure and chlorine concentration at sensor nodes, and the flow through certain metered pipes. In some more progressive utilities, these measurements can be obtained in real time. These measurements can carry a signature that will help identify the leak location and magnitude by using an inverse-modeling approach, potentially reducing the time and expense of leak detection.

4.2 Water Distribution Simulator

EPANET is an open source water distribution system (WDS) simulator that is widely used and freely available from the US EPA [14]. It uses the network topology, physical properties of the links and nodes, and demands to determine many of the hydraulic and quality values at each timestep, including the pressure heads and water quality (usually chlorine concentration) at each node, and the flow through each pipe. EPANET divides the total duration time of the simulation into hydraulic timesteps and quality timesteps. At each timestep, it solves many nonlinear equations simultaneously using an iterative method called "Gradient Method" [15]. EPANET uses properties, like the current tank level, as an initial condition for the next timestep. If a leak configuration is known, it can be modeled as an
input to the water distribution simulator and it will affect the pressure, quality, and flow values, as illustrated in Figure 4.1.

![Figure 4.1 - The WDS Simulator](image)

### 4.2.1 Leak Representation

Leaks are modeled in EPANET as emitters, such as a sprinkler or a fire hydrant. The distinction between an emitter and a demand, is that a demand is a flow out of a node that is known and fixed at each timestep, but an emitter is a flow that depends on the pressure at that node at that time. The equation for the emitter flow is given below (Equation 4.1).

\[
f_{n,t} = c_n p_{n,t}^\gamma
\]  

(4.1)

where \( n \) is the emitter node (i.e. leak node), \( t \) is the current timestep, \( \gamma \) is called the pressure exponent and is fixed at \( \gamma = 0.5 \), \( p \) is the pressure at node \( n \) at time \( t \), \( f \) is the emitter flow (i.e. flow through the leak) for node \( n \) at time \( t \), and \( c \) is the emitter coefficient for node \( n \).

The variables in this equation when modeling leaks are the leak location (the leak node \( n \)), and the leak magnitude (the emitter coefficient \( c_n \)).
4.2.2 Inverse Modeling Approach

It is not a straightforward operation to invert complex models in EPANET to obtain the inputs, such as leak location and magnitude, from the outputted pressure, flow, and water quality values. However, this problem can be formulated as a parameter estimation problem, and an inverse modeling approach (specifically, a simulation-optimization approach) can be used to determine the leak locations and leak magnitudes. In this approach, the leak parameters can be iteratively estimated by using the WDS simulator as the objective function that returns the simulated pressure, quality, and flow values. Then the difference (or error) between the simulated and the measured values can be calculated. By minimizing this error, a better estimation of the leak parameters is made. This is an optimization problem minimizing the difference between the simulated and measured sensor values for pressure, quality, and flow. Figure 4.2 shows the simulated and measured values being compared to make better estimates of the leak parameters.
4.2.3 Objective Function

As previously stated, the goal of this inverse modeling approach is to find the leak parameter values (location and magnitudes) that minimize the error between the simulated values and the measured values. Since there are three different type of measurements with different units and magnitudes (for pressure [ft], quality [mg/L], and flow [gpm]), they are normalized to be incorporated into a single objective function. Equation 4.2 below gives the normalized objective function, which is minimized for this leak detection problem.

\[
\frac{\sum_n \sum_t (p_{nt} - p_{nt}^0)^2}{\sum_n \sum_t (p_{nt}^0)^2} + \frac{\sum_n \sum_t (q_{nt} - q_{nt}^0)^2}{\sum_n \sum_t (q_{nt}^0)^2} + \frac{\sum_l \sum_t (f_{lt} - f_{lt}^0)^2}{\sum_l \sum_t (f_{lt}^0)^2}
\]  

(4.2)

where \( p \) represents the pressure values, \( q \) the quality values, \( f \) the flow values, \( t \) the current timestep, \( n \) the sensor node, \( l \) the sensor link (pipe), and \( 0 \) the measured values (observed, instead of simulated).

Figure 4.3 below shows the objective function as used in this simulation-optimization approach, along with the decision variables.

Figure 4.3 - The Simulation Optimization Approach with Decision Variables and Objective Function
4.3 Dividing the Discrete Network Nodes

In this research, the WDS nodes were considered to be a set of Cartesian points. When the DIRECT algorithm searches among these points for the leak node, the points are first trisected along the x-coordinate. Then the second time the points are divided, each set of points is trisected along the y-coordinate. The algorithm continues to alternate the coordinate dimension that is used to divide the set of points. This process is similar to how a k-d tree is divided [16], and will yield an approximately equal number of nodes in each child rectangle. Figure 4.4 shows WDS nodes recursively trisected four times. The first trisection occurs along the x-axis, then each of the three child rectangles are trisected along on the y-axis.

The midpoint is selected as the node closest to the spacial center of a rectangle tightly enclosing the points contained in the variable.

![Figure 4.4 - Recursively Trisected WDS Nodes](image)
4.4 Problem Representation

The leak locations (leak nodes, \( n \)) and leak magnitudes (emitter coefficients, \( c_n \), as shown in Equation 4.1), can be found in two practical ways:

**Method 1** For a fixed number of leaks \( N \), find the location \( n \) (discrete) and magnitude \( c \) (continuous) for each leak. This means that there will be a mix of discrete and continuous variables. The decision vector will have the following form:

\[
(n_1, c_1), (n_2, c_2), \ldots (n_N, c_N)
\]

with two decision variables per leak. This formulation does not have the flexibility to solve for an arbitrary number of leaks. If the number of leaks is unknown, the number of leaks could be incrementally increased until the solutions converge.

**Method 2** For a fixed subset of "candidate" nodes, find the magnitude \( c \) (continuous) for each node. The decision vector will be:

\[
c_1, c_2, c_3, c_4, \ldots c_N
\]

where \( N \) is the total number of candidate nodes. This means there are only continuous variables, and there is only one decision variable per candidate node.

When a candidate node \( n \) has no leak, the corresponding magnitude \( c_n \) should be zero. This approach has the flexibility to solve for any number of leaks and can be useful if there are already suspect leak areas or nodes. However, to incorporate all of the nodes in the WDS, it would take as many decision variables as there are nodes. This could mean tens of thousands of decision variables for large networks.

Results using these two methods are compared in Chapter 5.
Chapter 5 Leak Detection Results

The DIRECT algorithm was used to solve for leak locations and magnitudes on a small and a large test network. The small network is introduced in Section 5.1 and the large network is described in Section 5.2. Tests were performed on the small network to optimize the options and parameters of DIRECT. Results for these tests are given for a comparison of various DIRECT options in Section 5.1.1 and different values of the $\varepsilon$ parameter in Section 5.1.2. The dimensional scalability of the DIRECT algorithm is tested for both problem formations, that is Method 1 and Method 2 described previously in Sections 5.1.3 and 5.1.4 respectively. The dimensional scalability is also tested for the large network for both problem formations in Sections 5.2.1 and 5.2.2 respectively. Finally, the dimensional scalability of the DIRECT algorithm is compared to a genetic algorithm in Section 5.3.

In all of these test cases, the measured values are generated, by introducing artificial leaks in the network and then using these simulated values as the measured values. Also the leak magnitude (modeled as an emitter coefficient), which is continuous, was discretized at increments of 0.1 between 0 and 10.

5.1 Small Network

The small network used for the following tests is the Net3 network provided as an example network with EPANET. It contains 92 junctions (nodes), 2 reservoirs, 3 tanks, 117 pipes, and 2 pumps. The total flow through the network is 13,158 gpm, with demands ranging from 5 to 200 gpm.

In Figure 5.1 below, the small network is shown with the actual leak locations used when tested. The sensor nodes where pressure and quality are measured are enclosed in blue.
squares. The sensor links where the flow is measured are immediately after the reservoirs and tanks. The nodes where actual leaks are placed are enclosed in red circles and numbered. The ordering of the leak nodes indicates the location of the actual leak nodes. For example, when there is one actual leak, it is at node 1, when there are two actual leaks, they are at nodes 1 and 2, and so on. When using Method 2 to solve for the leak magnitudes at selected candidate nodes, the numbering also indicates which nodes are the candidate nodes. For example, if there are 4 candidate nodes and 2 actual leaks, the candidate nodes will be at nodes 1, 2, 3, and 4 and the actual leaks will be at nodes 1 and 2.

Figure 5.1 - Actual and Candidate Leak Locations for the Small Network (Net3)
5.1.1 DIRECT Options Tests

In Table 5.1 and Table 5.2 below, the number of function evaluations required to solve the leak detection problem using the various DIRECT options described in Section 3.3.2 are compared. Table 5.1 modifies the original DIRECT algorithm (Option 1 = Off), while Table 5.2 modifies the Aggressive DIRECT algorithm (Option 1 = On). The known global optimum for the leak detection problem is zero, indicating that there will be no difference between the simulated and measured values when the algorithm has found the correct leak scenario. For lower dimensions, the algorithm should be able to find the correct leak configuration for most option combinations. If the algorithm is able to find the actual leak scenario within the maximum of 500,000 function evaluations, the number of evaluations is given. If the algorithm did not find the true leak case with the function evaluation limit, the final objective function value is given in parentheses.

The column headings for Table 5.1 and Table 5.2 indicate what options were turned on (1) or off (0). The options described in these tables are the ones listed in Section 3.3.2. "Divide only 1 Side" is Option 2, "Size by Level" is Option 3, and "Only 1 Minimum" is Option 4. For example, the heading "1 0 1" for Table 5.1 indicates that Aggressive DIRECT was not used (Option 1 = 0), only one dimension/side is divided at a time (Option 2 = 1), the rectangle size is determined by the radius instead of the longest side (level) (Option 3 = 0), and only one minimum rectangle is divided per size group (Option 4 = 1).

The row headings indicate the number of leaks searched for, first, and the number of actual leaks, second. For example, when looking for both location and magnitude per leak (Method 1), the heading "2 1" means that two leaks were search for (4 total decision variables), but there was only one actual leak. The algorithm will ideally give a solution with
one non-zero magnitude at the actual leak node and one zero magnitude at an arbitrary leak node. When searching for only the magnitude at selected candidate nodes (Method 2), the heading "4 2" indicates that 4 leak magnitudes were searched for, but there was only two actual leaks. This means that the solution should return two non-zero magnitudes at the actual leak nodes and two zero magnitudes at the other candidate nodes.

When comparing the numbers between Table 5.1 and Table 5.2, it is obvious that Aggressive DIRECT takes more functions evaluations, and in many cases does not find the actual leak scenario. Dividing only one dimension at a time seemed to reduce the number of function evaluations in almost all cases where there were actual leaks (e.g. in Table 5.1, row "2 2" for Candidate Nodes, compare the options in columns "0 0 0" to "1 0 0"), but not in the cases where there are no actual leaks (e.g. in Table 5.1, row "2 0" for Candidate Nodes). Dividing only one dimension at a time reduces the number of function evaluations per iteration, giving the algorithm time to select other potentially optimal rectangles and only divide one of the child rectangle at a time. Determining the size of the rectangle by level increased the number of function evaluations in most cases (e.g. in Table 5.1, row "2 2" for Candidate Nodes, compare the options in columns "0 0 0" to "0 1 0"). This means that when solving the leak detection problem the DIRECT algorithm benefits from dividing many size groups in each iteration. And finally, dividing only one of the minimum rectangles per size group (Option 4) reduces the number of function evaluations slightly, in most cases (e.g. in Table 5.1, row "4 4" for Candidate Nodes, compare the options in columns "0 0 0" to "0 0 1"), but seemed to help the algorithm converge to the actual leak case when solving for both the leak location and magnitude (Method 1).
Table 5.1 - Comparison of DIRECT Options for Low Dimensional Problems in Terms of the Number of Function Evaluations to Find the True Leak Case (Option 1 = Off)

If the true leak case is not found, the final objective function value is shown in parenthesis.

<table>
<thead>
<tr>
<th>Option Configurations (0 = On, 1 = Off)</th>
<th>Opt. 2 Divide only 1 Side</th>
<th>Opt. 3 Size by Level</th>
<th>Opt. 4 Only 1 Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>89</td>
<td>89</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>46861</td>
<td>256331</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(0.0003)</td>
<td>280050</td>
</tr>
</tbody>
</table>

| Method 2 Location and Magnitude | 2 | 0 | 145 | 145 | 87 | 87 | 79 | 79 | 49 | 49 |
| Method 2 Magnitude at Candidate Nodes | 2 | 2 | 91 | 91 | 129 | 127 | 65 | 65 | 41 | 41 |
| 4 | 0 | 1339 | 1335 | 595 | 711 | 807 | 807 | 199 | 231 |
| 4 | 2 | 3453 | 3417 | 1231 | 1153 | 2749 | 2849 | (0.00051) | 19795 |
| 4 | 4 | 374211 | 172939 | (0.00106) | 194351 | 2685 | 2599 | 2253 | 1879 |
| Total | 965108 | 716417 | 1692649 | 774542 | 129181 | 102538 | 1503250 | 745922 |

Table 5.2 - Comparison of Aggressive DIRECT Options for Low Dimensional Problems in Terms of the Number of Function Evaluations to Find the True Leak Case (Option 1 = On)

If the true leak case is not found, the final objective function value is shown in parenthesis.

<table>
<thead>
<tr>
<th>Option Configurations (0 = On, 1 = Off)</th>
<th>Opt. 2 Divide only 1 Side</th>
<th>Opt. 3 Size by Level</th>
<th>Opt. 4 Only 1 Minimum</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>67</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>283</td>
<td>279</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>(0.000248)</td>
<td>(3.54e-5)</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>(0.000557)</td>
<td>(0.000624)</td>
</tr>
<tr>
<td>Method 2 Magnitude at Candidate Nodes</td>
<td>2</td>
<td>0</td>
<td>233</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>125</td>
<td>125</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4489</td>
<td>2115</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>14755</td>
<td>4357</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>(0.00026)</td>
<td>(0.0002)</td>
</tr>
<tr>
<td>Total</td>
<td>2019952</td>
<td>1522005</td>
<td>2501305</td>
</tr>
</tbody>
</table>
The optimum configuration of options was chosen based on the sum of function evaluations taken to find the actual leak scenario, in the row labeled "Total" in the tables above. The option configuration that used the least number of function evaluations only divided one dimension at a time, and only divided one of the minimum rectangles per size group. It did not use Aggressive DIRECT, and it did not considering rectangle size by level only. This optimal configuration is Column "1 0 1" in Table 5.1, and is indicated in bold.

5.1.2 $\varepsilon$ Parameter Tests

The balancing parameter $\varepsilon$ essentially controls the emphasis on local or global search (described in Section 2.3.3). A value of $\varepsilon = 0$ induces more emphasis on local search, since it will always divide the rectangles with the minimum objective function value. A larger $\varepsilon$ value will emphasize a more global search by ignoring the rectangles with the minimum objective function value if they do not meet Condition 2.4:

$$f_j - K\cdot d_j \geq f_{\min} - \varepsilon|f_{\min}|$$

Table 5.3 and Table 5.4 below show the number of function evaluations taken to find the actual leak scenario for various $\varepsilon$ values for Method 1 and Method 2 respectively. The options used are those determined to be the best configuration in the previous Section 5.1.1; it only divides one dimension at a time, and only divides one of the minimum rectangles per size group. The vertical axis labels and data values use the same notation as Table 5.1 and Table 5.2 described previously. The maximum function evaluations limit is 500,000.
Table 5.3 - Comparison of Various $\varepsilon$ Parameters using Method 1 for Low Dimensions in Terms of the Number of Function Evaluations to Find the True Leak Case

If the true leak case is not found, the final objective function value is shown in parenthesis.

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>0.01</th>
<th>0.001</th>
<th>0.0001</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>85</td>
<td>77</td>
<td>77</td>
<td>77</td>
</tr>
<tr>
<td>1 1</td>
<td>214</td>
<td>224</td>
<td>228</td>
<td>232</td>
</tr>
<tr>
<td>2 0</td>
<td>379</td>
<td>285</td>
<td>297</td>
<td>299</td>
</tr>
<tr>
<td>2 1</td>
<td>30490</td>
<td>30206</td>
<td>29842</td>
<td>55925</td>
</tr>
<tr>
<td>2 2</td>
<td>64971</td>
<td>62585</td>
<td>64255</td>
<td>(0.00014)</td>
</tr>
<tr>
<td>3 0</td>
<td>25709</td>
<td>877</td>
<td>80680</td>
<td>1417</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>96139</td>
<td>93377</td>
<td>94699</td>
<td>556533</td>
</tr>
</tbody>
</table>

Table 5.4 - Comparison of Various $\varepsilon$ Parameters using Method 2 for Low Dimensions in Terms of the Number of Function Evaluations to Find the True Leak Case

If the true leak case is not found, the final objective function value is shown in parenthesis.

<table>
<thead>
<tr>
<th>$\varepsilon$</th>
<th>0.01</th>
<th>0.001</th>
<th>0.0001</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 0</td>
<td>79</td>
<td>79</td>
<td>79</td>
<td>79</td>
</tr>
<tr>
<td>2 2</td>
<td>65</td>
<td>69</td>
<td>69</td>
<td>69</td>
</tr>
<tr>
<td>4 0</td>
<td>807</td>
<td>789</td>
<td>789</td>
<td>807</td>
</tr>
<tr>
<td>4 2</td>
<td>2849</td>
<td>2955</td>
<td>2965</td>
<td>3105</td>
</tr>
<tr>
<td>4 4</td>
<td>2599</td>
<td>2773</td>
<td>2881</td>
<td>2883</td>
</tr>
<tr>
<td>6 0</td>
<td>(0.00139)</td>
<td>(0.00139)</td>
<td>(0.00139)</td>
<td>26733</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>506399</td>
<td>506665</td>
<td>506783</td>
<td>33676</td>
</tr>
</tbody>
</table>

The results for the $\varepsilon$ parameter tests show that different $\varepsilon$ values are best for Methods 1 and 2. The sums of function evaluations for all trials (row "Total") in Table 5.3 show that $\varepsilon = 0.001$ is best for Method 1, whereas the total function evaluations in Table 5.4 show that $\varepsilon = 0$ is best for Method 2. This means that more emphasis on local searching is beneficial when formulating the problem with magnitudes only (Method 2). However, more emphasis on global search is needed when searching for both discrete network nodes and magnitudes (Method 1).
5.1.3 Location and Magnitude for Small Network (Method 1)

Figure 5.2 below shows the final objective function value after a maximum of 100,000 function evaluations when using the DIRECT method to solve for the leak location and magnitude. The number of leaks solved for is gradually increased, and the number of actual leaks is gradually increased. The leak detection problem becomes harder to solve as the number of leaks solved for increased. This is because the dimensionality of the problem is increasing linearly and the hypervolume of the search domain is increasing exponentially. Therefore it is expected that the algorithm will take many more rectangle divisions (and thus function evaluations) to converge on the global optimum.

The horizontal axis for Figure 5.2 indicates the number of leaks searched for, first, and the number of actual leaks, second. For example, the label "2 1" means that two leaks were search for (4 total decision variables), but there was only one actual leak. Ideally, one leak will have a non-zero magnitude at the actual leak node, and the other leak will have a zero magnitude at an arbitrary node.

![Figure 5.2 - Final Objective Function Value for Various Leak Configurations for the Small Network using Method 1](image-url)
The graph in Figure 5.2 shows that it does indeed become more difficult to find the actual leak scenario as the number of leaks solved for increases. However, the graph also shows that it becomes increasing difficult as the number of actual leaks increases. This is because as the number of actual leaks increases the number of nodes with a non-zero magnitude increases. When the magnitude is zero, it doesn't matter what node the algorithm has chosen. Therefore, increasing the number of actual leaks essentially increases the number of "active" decision variables.

Figure 5.3 and Figure 5.4 below show some selected examples of the solutions returned by the DIRECT algorithm when solving for the leak location and magnitude (Method 1) at the end of the maximum function evaluations.

Figure 5.3 - DIRECT Solution for 3 Leaks Searched For and 1 Actual Leaks "3 1"

Figure 5.4 - DIRECT Solution for 3 Leaks Searched For and 3 Actual Leaks "3 3"
In Figure 5.3, the DIRECT solution has one larger leak and one smaller leak at adjacent nodes to the actual leak node, and one very small leak farther away from the actual leak node. This example shows that multiple smaller leaks can have a similar objective function value as a single leak. In various tests performed, it has been found that solutions that have similar total leak magnitudes (related to the total water loss) yield very similar objective function values.

The DIRECT solution shown in Figure 5.4 is not very different from the actual leak scenario. The leak nodes found were adjacent to the actual leak nodes, and the magnitudes are very similar. Ironically, the solution in Figure 5.4, which appears closer to the actual leak scenario, has a greater error than the solution in Figure 5.3.

5.1.4 Magnitude at Candidate Nodes for Small Network (Method 2)

Figure 5.5 below shows the final objective function value after a maximum of 100,000 function evaluations when solving for the magnitude at candidate locations. The number of candidate leak nodes is gradually increased, and the number of actual leaks is gradually increased. As stated earlier, the leak detection problem becomes harder to solve as the dimensionality increased. Therefore it is expected that the algorithm will take an increasing amount of function evaluations to find the actual leak scenario as the number of candidate nodes increases.

The horizontal axis for Figure 5.5 has a similar notation to Figure 5.2. The number of candidate nodes is first, and the number of actual leaks is second. For example, the label "4 2" means that there are four candidate nodes, but only two actual leaks. Ideally, two magnitudes would be non-zero at the actual leaks nodes, and the other two magnitudes would be zero.
The graph in Figure 5.5 shows that it becomes more difficult to find the actual leak scenario as the number of candidate leaks increases. However, the graph also shows a curved pattern when comparing final objective values for leaks scenarios with the same number of candidate leaks. The algorithm converges to a solution closer to the true optimum solution when the number of actual leaks is close to zero or close to the number of candidate nodes. The algorithm diverges with leak scenarios in between those cases. This is because the leak magnitude has been discretized, and the number of combinations of solutions that yield similar objective function values (the solutions that have similar total magnitudes) is less when the number of actual leaks is close to zero or close to the number of candidate nodes. Also the magnitude of all the actual leaks is 1.0. If the magnitudes had been different for each leak, then the trend of increasing difficulty with an increasing number of actual leaks would have been seen, as in Figure 5.2.
Comparing the magnitudes of the errors in Figure 5.5 and Figure 5.2, it can be seen that Method 2 yields lower error values. This is because Method 2 reduces some of the complexity of the leak detection problem by preselecting nodes.

Figure 5.6 and Figure 5.7 below show some selected examples of the solutions returned by the DIRECT algorithm at the end of the maximum function evaluations when solving for the magnitudes at candidate nodes (Method 2).

In Figure 5.6, the solution returned is very similar to the actual leak scenario. There are only small magnitudes at nodes that do not actually have leaks. The DIRECT solution in Figure 5.7 is also very similar to the actual leak scenario. There are only very small differences in magnitudes. The solution shown in Figure 5.7 has a smaller error than the solution in Figure 5.6 as shown in the graph in Figure 5.5.
5.2 Large Network

The large network tested in this research is the Network_2 network given in the Battle of the Water Sensor Network (BWSN) competition [16]. It contains 12,523 junctions (nodes), 2 reservoirs, 2 tanks, 14822 pipes, 4 pumps, and 5 valves. The total flow through the network is 11,061 gpm, with almost all demands under 25 gpm.

In Figure 5.8, the large network is shown with the actual leak locations used when tested. The notation is the same as in Figure 5.1. The sensor nodes are enclosed in blue squares. The sensor links where the flow is measured are immediately after the reservoirs and tanks. The nodes where actual leaks are placed are enclosed in red circles and numbered. As in Figure 5.1, the ordering of the leak nodes indicates the location of the actual leak nodes. For example, when there is one actual leak, it is at node 1, when there are two actual leaks, they are at nodes 1 and 2, and so on. When using the Method 2 to solve for the leak magnitude at selected candidate nodes, the numbering also indicates which nodes are the candidate nodes.
Figure 5.8 - Actual and Candidate Leak Locations for the Large Network (Network_2)
5.2.1 Location and Magnitude for Large Network (Method 1)

Figure 5.9 below shows the final objective function value after a maximum of 10,000 function evaluations when using the DIRECT method to solve for the leak location and magnitude on the large network. The maximum function evaluations limit is set lower for the large network, because the simulations took much longer to run for the large network. The graph shows that the dimensional scalability trends for the large network are essentially the same for the smaller network. Increasing number the number of leaks searched for, increases the difficulty of converging on the actual leak scenario. Increasing the number of actual leaks increases the difficulty as well.

![Graph showing final objective function value comparison for the large network using Method 1.](image)

**Figure 5.9 - Final Objective Function Value Comparison for the Large Network using Method 1**

In Figure 5.10 below is the leak scenario where three leaks were search for and there are three actual leaks. The leak nodes found are far away from the actual leak nodes. This is due to few things. First, much more than 10,000 function evaluations would be needed to
converge near the actual leak nodes. Second, because the network is so vast, the 20 sensors do not give the resolution to easily distinguish between nodes. Leaks at different points in the network can have similar sensor readings, because the sensors are sparse in very dense and vast network.

Figure 5.10 - DIRECT Solution when Searching for 3 Leaks with 3 Actual Leaks "3 3"
5.2.2 Magnitude at Candidate Nodes for the Large Network (Method 2)

Figure 5.11 below shows the final objective function value after a maximum of 10,000 function evaluations when using the DIRECT method to solve for the magnitude at candidate locations the large network. Again, the dimensional scalability trends for the large network for Method 2 are essentially the same for the smaller network. Increasing number the number of leaks searched for increases the difficulty of converging on the actual leak scenario. Also, it more quickly converges to the actual leak scenario when the actual number of leaks is close to zero or close to the number of candidate nodes.

![Figure 5.11 - Final Objective Function Value Comparison for the Large Network using Method 2](image)

Comparing the magnitudes of Figure 5.11 with Figures 5.9, 5.5, and 5.2 it should be noted that the magnitude of the errors using Method 2 are smaller than the errors when using Method 1. Again, this is because the complexity of the problem is reduced by preselecting candidate nodes. Also, it is important to note that the errors for the large network are much
smaller than the errors for the small network. This means that leaks of the same magnitude have less of an effect on the large network measurement values than on the small network. It does not mean that the solutions returned by the algorithm are any closer to the actual leak scenario. In fact, the solutions shown in Figure 5.9 and Figure 5.11 are farther from the actual leak scenario, than the solutions shown for the small network.

In Figure 5.12 below is the leak scenario where six candidate nodes and there are six actual leaks. The magnitudes at the candidate nodes are very similar to the actual leak scenario for four out of six of the candidate nodes.
5.3 Comparison of the DIRECT algorithm and a Genetic Algorithm

The four graphs on the next page give a comparison of the results of the DIRECT method to those obtained using a basic genetic algorithm (GA) for Method 1 and Method 2 and for the small and large networks. The GA used a mutation rate of 0.02 and a crossover rate of 0.8. It used a uniform mutation and a uniform integer crossover for both the leak node and the leak magnitude. The leak magnitude was discretized in increments of 0.1 between 0 and 10. For the small network, the GA had a population size of 2000 and ran for 50 generations (100,000 total function evaluations). For the large network, the GA had a population size of 500 and ran for 20 generations (10,000 total function evaluations). The GA was run for 10 trials and the best and average values are shown below.

For the GA runs performed for Method 1, the best and average final objective function values were generally lower than the final objective function values when using the DIRECT method. This means that the GA was better able to reduce the error than the DIRECT method within the function evaluation limit. However, the same general pattern of increasing difficulty with increasing dimensions is seen in the GA solutions as well as the DIRECT solutions. For example, in Figure 5.13 solving the "4 4" leak case with DIRECT yields a greater objective function value than the average of the GA trials.

For the tests runs performed for Method 2, the DIRECT algorithm performed similarly to the GA. The solutions returned by DIRECT had objective values mostly in between the average and best solutions return by the GA. For example, in Figure 5.14 solving the "8 8" leak case with DIRECT yields a smaller objective function value than the average of the GA trials.
Figure 5.13 - GA Comparison for Small Network using Method 1

Figure 5.14 - GA Comparison for Small Network using Method 2

Figure 5.15 - GA Comparison for Large Network using Method 1

Figure 5.16 - GA Comparison for Large Network using Method 2
Chapter 6 Parallel Performance

A secondary goal of this research was to provide a high performance computing (HPC) framework for the DIRECT algorithm to solve the leak detection problem.

6.1 Parallel Organization

The DIRECT algorithm has been parallelized before [2] [4] [5] [6] and as noted in Gablonsky's work [2], there are two main places that parallel functionality can be added: parallelizing the independent function evaluations, or parallelizing the sampling and evaluating of each of the rectangles. Gablonsky parallelizes the sampling and evaluating of the rectangles in his implementation of DIRECT. This research, however, parallelizes only the function evaluations, in order to keep the optimization algorithm separate from the leak simulator. By keeping these parts separate, another optimization algorithm can easily be interchanged to solve the leak detection problem, and this implementation of the DIRECT algorithm can easily be used to solve other problems.

In this research, the parallelism is incorporated in two locations. First, an independent middleware was developed so that the DIRECT algorithm can be used with any objective function. This was used for each of the test functions in Section 3.4. Second, parallelism is built into the leak simulator and can be optionally used. This was done so that multiple leak detection simulations can be run in parallel without using optimization. Figure 6.1 illustrates the simple parallel implementation.
6.2 Computer Architecture

The parallel performance tests were run on a local Linux cluster in the Civil Engineering Department at NCSU. It is a six node cluster, with each node having four Opteron processors with eight cores each, a total of 192 cores. The hardware specifications are given below:

- 4 processors per node
  
  AMD "Magny-Cours" Opteron 6128MS 2.0 GHz, 8 Core CPU

- 64 GB total memory at 1333Mhz

- 4 TB HD (Primary node)

- 500 GB HD (Secondary nodes)

- 6 nodes connected with a Gigabit switch

The Message Passing Interface version 2 (MPI) [17] was used to distribute the WDS simulations. The master node handles all of the DIRECT optimization and ran no WDS simulations. The master processor then distributes the different leak scenarios generated by the DIRECT algorithm to the worker processors using MPI_Send and MPI_Recv. The master processor uses the MPI_ANY_SOURCE keyword to allow load balancing among the...
processors. All of the processors are initially sent one simulation, and then the remaining simulations are sent to the processors as they finish their current simulation. So as the processors finish their simulation, they are sent a new simulation. This more effectively balances the work load among the worker processors. If one simulation or processor is running slower than the others, it will not hold up the other simulations from being completed.

6.3 Speedup Curves for Different DIRECT Options

The original DIRECT algorithm is not expected to yield very good parallel performance since the number of function evaluations can change from iteration to iteration. As is shown in Sections 2.3 and 2.4, the DIRECT algorithm chooses multiple "potentially optimal" rectangles to divide per iteration, and the number of rectangles chosen to divide changes per iteration. Therefore, it is difficult to equally balance the load per processor and amortize the communication and optimization costs. The purpose of Aggressive DIRECT [13] is to increase the number of function evaluations per iteration, and thus better amortize the costs, yielding more effective parallel performance.

The scalability curves shown below in Figure 6.2 are the time to run 100,000 function evaluations (WDS simulations), while optimizing the three leak problem for the small network, where there are two actual leaks. There is a limit of 100,000 function evaluations.
The scalability tests show that the Aggressive DIRECT algorithm scales the best. This is because it does many more function evaluation per iteration, and does not have as much optimization overhead per iteration. However, Aggressive DIRECT will take more function evaluations to optimize a problem than other versions of DIRECT, so care must be given in determining which options are best when using DIRECT, especially if the objective function is very time consuming to compute.

The original DIRECT algorithm performed almost as well as the Aggressive DIRECT algorithm for this number of function evaluations. Both the original DIRECT algorithm and Aggressive DIRECT algorithm plateaued from 8 to 32 worker processors. This is because the time saved by distributing the objective function evaluations is lost in communicating between processors and nodes.
The version of DIRECT used in this research to solve the leak detection problem did not scale as well as the original DIRECT algorithm or Aggressive DIRECT. This is because it performs significantly less function evaluations per iteration, and is therefore not as efficiently parallelized. Both the original DIRECT algorithm and Aggressive DIRECT divide all of the dimensions of a rectangle in each iteration, whereas this implementation of DIRECT only divides one dimension of a rectangle in each iteration.

For applications on massively parallel architectures, Aggressive DIRECT is the better algorithm to use. Even though it performs many more function evaluations per iteration, it will still converge in the same number of iterations, or potentially less. It is able to utilize more processors and not waste as much processing power as the original DIRECT algorithm, because it evaluates the objective function more times in each iteration.
Chapter 7 Conclusions and Future Work

7.1 Summary

In this thesis, the original DIRECT algorithm has been described and modified to allow for generic variable types, including discrete types. This was done by abstracting the properties and division process for rectangles and different variable types. This allowed the DIRECT algorithm to be used for finding the leak locations and magnitudes in a water distribution system, a mixed discrete and continuous variable problem.

It was shown that this version of DIRECT performed similarly to previous implementations of DIRECT for test functions. It was also shown that the DIRECT method can be used to solve for a small number of leaks in a small or large WDS. Two methods of formulating the leak detection problem were introduced, and both methods scaled similarly. As the number of leaks increases, the dimensionality of the problem increases, and thus the decision space increases exponentially. Because of the high dimensionality of decision space in the leak detection problem, the DIRECT algorithm requires many more partitions to get close to the global optimum, and thus requires many function evaluations.

The DIRECT method was compared to a GA for solving the leak detection problem with a small number of leaks, and the DIRECT algorithm performed similarly to the GA. When solving for both leak location and magnitude (Method 1), the DIRECT method slightly underperforms the GA, but when solving for the leak magnitude at fixed candidate locations (Method 2), the DIRECT algorithm slightly outperforms the GA.

Finally, the DIRECT method used in this research was parallelized and the scalability was compared to the original DIRECT algorithm and the Aggressive DIRECT algorithm.
The version of DIRECT used in this research does not scale as well as either the original DIRECT method or Aggressive DIRECT, but uses fewer function evaluations to converge. When using massively parallel architectures, the Aggressive DIRECT algorithm should be used.

**7.2 Future Work**

First, the adaptive ε parameter modification of DIRECT (described in Section 2.5.6 and introduced in Finkel's thesis [12]) seems to reduce the number of function evaluations needed, but this was not explored in this research. Implementing this modification could make the DIRECT algorithm more useful for highly dimensional problems, like leak detection in water distribution systems.

Second, the introduction of a way to generalize the DIRECT algorithm for different variable types as presented here gives great flexibility for this algorithm to be applied to many new ways. In this research, the WDS nodes were considered to be Cartesian points and were divided along their X-Y coordinates. However, other division schemes can be used. The WDS nodes can be considered graph nodes and divided using their adjacency information. The weight of each of the graph links can be the length of the pipe, the volume of the pipe, or the average hydraulic residence time. Considering the WDS nodes as graph nodes could mean using minimum cuts, k-way partitioning, or various other clustering and partitioning algorithms as the division scheme. The WDS nodes can also be recursively skeletonized, starting with a coarse skeleton, and refining the skeleton of the region to be divided. WDS nodes can also be divided based on pressures, which is essentially like
identifying and dividing pressure zones. Furthermore, the binary variable type, which has not been explored in this research, may be helpful in WDS applications.

The generalizations presented in this research will provide a basis for the DIRECT algorithm to be applied to many different types of problems and to be used in new ways previously not possible.
REFERENCES


Appendix A Source Code

The source code for the DIRECT method in C, as well as the Leak Detection framework using EPANET will be made public. It is available under the terms of the GNU General Public License (GPL).

The source code will be hosted at this public address:
http://sourceforge.net/projects/leaksim/

The code is also currently hosted at this address (as of April 9, 2014), but may be taken down in the future:
http://d-mnjasper-01.ce.ncsu.edu/svn/leak/

Finally, anyone interested in using this code or asking questions about DIRECT or leak detection using WDS software, like EPANET, may contact the author at:

mnjasper@ncsu.edu
micahnjasper@gmail.com