College of William and Mary
W&M ScholarWorks

Undergraduate Honors Theses | Theses, Dissertations, & Master Projects

5-2009

Software Engineering of a Direct Search Package for Nonlinear Optimization

Charles Michael Liarakos
College of William and Mary

Follow this and additional works at: https://scholarworks.wm.edu/honorstheses

Recommended Citation
Liarakos, Charles Michael, "Software Engineering of a Direct Search Package for Nonlinear Optimization" (2009). *Undergraduate Honors Theses.* Paper 304.
https://scholarworks.wm.edu/honorstheses/304

This Honors Thesis is brought to you for free and open access by the Theses, Dissertations, & Master Projects at W&M ScholarWorks. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of W&M ScholarWorks. For more information, please contact scholarworks@wm.edu.

Software Engineering of a Direct Search Package for Nonlinear Optimization

A thesis submitted in partial fulfillment of the requirement
for the degree of Bachelor of Science in Computer Science from
The College of William & Mary

by

Michael Liarakos

Accepted for ______________________________
(Honors, High Honors, Highest Honors)

____
Virginia Torczon, Director

____
Robert Michael Lewis

____
Andreas Stathopoulos

Williamsburg, VA
April 27, 2009

Abstract

The DiSCOTech software package implements a direct search method with heuristic improvement techniques for solving unconstrained and linearly constrained nonlinear optimization problems. By implementing various heuristic improvement techniques DiSCOTech seeks to improve the performance of direct search methods. Currently, the development of DiSCOTech is ongoing and is the subject of this research.

First, an object-oriented restructuring of the codebase is discussed. Next, the issue of designing an effective caching mechanism for DiSCOTech is addressed. Issues with proposed cache designs are investigated and an alternative design is implemented and tested. Finally, the relative merits of the heuristic improvement techniques implemented by DiSCOTech are tested to determine which heuristics reliably provide improved performance.

Acknowledgements

First and foremost I would like to thank Virginia Torczon and Michael Lewis for their guidance and patience. Dr. Torczon provided my first introduction to direct search and encouraged me to pursue summer research and, ultimately, an honors thesis. Dr. Lewis was always on hand to answer my questions about DiSCOTech and provide thought-provoking conversation. I would like to acknowledge Andreas Stathopoulos for agreeing to be on my honors committee. I would also like to thank the faculty and students who participated in CSUMS 2008 for an amazing summer research experience. This research was funded by the National Science Foundation under Grant DMS-0703532. Finally, I am grateful to Robert Staubs for his moral support and for proofreading my thesis.

Contents

1 Introduction **3**

1.1 Direct Search Overview . . . 3

1.2 Improving the Performance of Direct Search . . . 4

1.2.1 Exploratory Moves . . . 4

1.2.2 Modified Local Searches . . . 5

1.3 Goals . . . 6

2 Code Restructuring **6**

2.1 Code Overview . . . 6

2.2 Object-Oriented Restructuring . . . 7

3 Cache **10**

3.1 Motivation . . . 10

3.2 Designing a Cache . . . 10

3.2.1 Binary Search Tree . . . 11

3.2.2 AVL Tree . . . 13

3.2.3 Generalized Binary Search Tree . . . 15

3.3 Importance of a Total Order . . . 16

3.3.1 Total Order and the ϵ-metric . . . 16

3.3.2 Subtle Mistakes . . . 16

3.4 Revisiting Alternatives . . . 17

3.4.1 Selecting a New Design . . . 17

3.4.2 Testing . . . 18

3.4.3 Results . . . 19

4 Comparing Heuristic Improvement Techniques **23**

4.1 Comparison Setup . . . 23

4.2 Results . . . 23

4.2.1 Local Search Algorithm: Sequential Search vs. Polar Search . 28

4.2.2 Local Search Method: Local Search vs. Active Set Local Search 28

4.2.3 Trial Steps . . . 29

5 Conclusion **29**

6 Cache Source Code **31**

References **37**

1 Introduction

Consider solving non-linear optimization problems of the form

$$\begin{array}{ll} \text{minimize} & f(x) \\ \text{subject to} & x \in S \subseteq \mathbb{R}^n \end{array}$$

without explicit knowledge of the derivative. This means that traditional methods that rely on first and second derivative information cannot be used. A new class of derivative-free optimization methods are required to tackle these problems. Currently, such derivative-free methods with strong convergence guarantees exist. The Direct Search Constrained Optimization Techniques (DiSCOTech) package is a prerelease software package that implements a particular type of derivative-free optimization method called *direct search* for unconstrained and linearly constrained nonlinear optimization problems.

DiSCOTech began as a testbed for techniques to improve the performance of direct search methods. The ongoing development of DiSCOTech eventually reached a point where the codebase needed to be prepared for release. This involved the restructuring of code to a more clear and extensible format, implementing a caching mechanism, and evaluating the effectiveness of various heuristic improvement techniques. In order to better understand these issues, a brief introduction to direct search methods is required.

1.1 Direct Search Overview

Direct search methods attempt to minimize the value of an objective function f by iteratively searching for improvement in the value of the objective function at various sets of test points. Given a feasible iterate x_k, direct search methods take steps of length $\Delta_k \, ||d_k||$ (where Δ_k is called the step-length) from x_k in each direction d_k in a set of search direction $\mathcal{D}_k$, looking for a test point $x_k + \Delta_k d_k$ that reduces on the value of the objective function. If such a point is found, the iteration is deemed *successful* and the test point is accepted as the next iterate. If no such point is found, the iteration is deemed *unsuccessful* and Δ_k is reduced. This process continues until Δ_k falls below a certain tolerance Δ_{tol}. Improvement in the objective can be a simple improvement in the value of the objective, i.e. $f(x_k + \Delta_k d_k) < f(x_k)$, or a sufficient decrease enforced via a forcing function ρ that depends on the current value of Δ_k, i.e. $f(x_k + \Delta_k d_k) < f(x_k) - \rho(\Delta_k)$. A commonly used forcing function is $\rho(\Delta) = \alpha \Delta^2$ with $\alpha > 0$. Assume that improvement in the value of the objective requires sufficient decrease for the remainder of this discussion of direct search. This basic algorithm handles the unconstrained case and is summarized in Figure 1.

By properly constructing $\mathcal{D}_k$ it is possible to ensure that at least one direction d_k is within 90° of the direction of steepest decent and is therefore a descent direction. This can be accomplished, for example, be making $\mathcal{D}_k$ a *positive basis* of $\mathbb{R}^n$. A positive basis is a set of vectors where no vector is a nonnegative combination of the others and whose positive span is $\mathbb{R}^n$ [9].

Initialization:

Let $f : \mathbb{R}^n \to \mathbb{R}$ be given.

Let $x_0 \in \mathbb{R}^n$ be the initial guess.

Let $\Delta_{\text{tol}} > 0$ be the step-length convergence tolerance.

Let $\Delta_0 > \Delta_{\text{tol}}$ be the initial value of the step-length control parameter.

Let $\rho(\Delta) = \alpha\Delta^2$ be the forcing function, with $\alpha > 0$.

Algorithm: For each iteration $k = 1, 2, \ldots$

Step 1. Generate the set of search directions $\mathcal{D}_k$

Step 2. If there exists a $d_k \in \mathcal{D}_k$ such that $f(x_k + \Delta_k d_k) < f(x_k) - \rho(\Delta_k)$, then:

- Set $x_{k+1} = x_k + \Delta_k d_k$.
- Set $\Delta_{k+1} = \Delta_k$.

Step 3. Otherwise, $f(x_k + \Delta_k d_k) \geq f(x_k) - \rho(\Delta_k)$ for all $d_k \in \mathcal{D}_k$, then:

- Set $x_{k+1} = x_k$.
- Set $\Delta_{k+1} = \frac{1}{2}\Delta_k$.
- If $\Delta_{k+1} < \Delta_{\text{tol}}$, then terminate.

Figure 1: *Basic direct search algorithm*

The algorithm discussed thus far deals primarily with unconstrained problems, but can be generalized to constrained problems as well. DiSCOTech uses a more robust version of this algorithm for constrained and unconstrained problems, as discussed in [7]. However for the purposes of a basic overview, this simpler case is sufficient.

1.2 Improving the Performance of Direct Search

The basic algorithm and the more robust version used by DiSCOTech have multiple avenues for attempting to improve a search's performance. DiSCOTech implements two such approaches: *exploratory moves* and *modified local searches*.

1.2.1 Exploratory Moves

As previous discussed, direct search methods require that there be a sufficient number of properly chosen search directions $\mathcal{D}_k$ to ensure that at least one is a descent

direction. However, the search directions are not limited to those that are necessary to ensure convergence. Additional search directions derived from various heuristics can be included in an attempt to more rapidly find a solution [7]. These additional directions are called *trial* or *exploratory moves*.

DiSCOTech implements two types of exploratory moves: *pattern steps* and *active set steps*. Pattern steps operate on the assumption that if the direction from x_{k-1} to x_k yield improvement, then taking an additional step in that direction might yield continued improvement. Therefore a test point x_p is defined such that

$$x_p = x_k + (x_k - x_{k-1}).$$

A search is performed around the point x_p to find a point x_+ that yields sufficient improvement in the objective. In addition to being simple to compute, pattern steps allow for steps longer than Δ_k, providing the possibility of making greater strides toward a stationary point.

Active set steps, detailed in [8], apply to linearly constrained optimization problems, and therefore require a brief explanation of how linear constraints are handled. For linearly constrained optimization problems DiSCOTech uses information about the geometry of the constraints near the current iterate to build $\mathcal{D}_k$. To determine which constraints matter, DiSCOTech constructs a *working set* of constraints that are either active at x_k or near x_k. For a more detailed discussion of constraints, see [7]. An active set step assumes that all these constraints might be active at the solution, and attempts a step to the face of the constraint polytope defined by these constraints. If this step yields sufficient improvement in the objective, then it is accepted as the next iterate.

There are other active set strategies that can be applied at other stages in the search process. For a detailed discussion of these strategies, see [8]. Three such strategies will be briefly discussed. First, the search directions $\mathcal{D}_k$ can be oriented to point toward constraints that might become active at the solution. Second, after an unsuccessful search around the current iterate an active set step can be attempted to find a new iterate before searching all the directions in $\mathcal{D}_k$ again. Finally, active set steps can also be combined with pattern steps. In this case the active set step is treated as the pattern step and a search is performed around the point defined by the active set step.

1.2.2 Modified Local Searches

The search around x_k in the directions $\mathcal{D}_k$ can be executed in multiple ways. The simplest way is a *sequential search* of each direction accepting the test point the yields the best sufficient improvement. DiSCOTech provides an additional method called a *polar search* which favors search directions that are close to the direction from the last iterate to the current iterate. Call this direction d_{prev}. In a polar search d_{prev} is used to order the set of search directions so that directions with the smallest angles between themselves and d_{prev} are tested first. However, unlike a sequential search, the search direction that yields the best sufficient improvement is not necessarily the direction selected by a polar search. Let $x_k + \Delta_k d_i$ be the first direction found by the

polar search the yields sufficient improvement in the value of the objective. All the following directions must then yield sufficient improvement compared to the objective value at $x_k + \Delta_k d_i$. This means that a search direction d_j will be selected over d_i only if $f(x_k + \Delta_k d_j) < f(x_k + \Delta_k d_i) - \rho(\Delta_k)$. In this way a polar search will favor the search directions close to d_{prev}

There is one additional caveat: if a search direction does not yield sufficient improvement, the search direction that best points in the opposite direction is checked next. This allows search directions opposite d_{prev} a chance to be checked early if the search directions close to d_{prev} are not yielding sufficient improvement.

1.3 Goals

As has been discussed, DiSCOTech incorporates promising techniques for improving the performance of direct search methods. However, the codebase had outdated options, complicated code paths, and was not designed to be highly extensible. The first goal of my research was to streamline the DiSCOTech codebase to remove unneeded options and standardize how operations were performed using an object-oriented design. Next, I investigated issues regarding proposed designs for $\langle x, f(x) \rangle$ caches and designed such a cache for use in DiSCOTech. Finally, I investigated the effectiveness of the various heuristic improvement techniques discussed thus far on a set of standard nonlinear optimization problems.

2 Code Restructuring

2.1 Code Overview

DiSCOTech implements a direct search method with heuristic improvement techniques for solving unconstrained and linearly constrained nonlinear optimization problems. In addition to the features outlined in Section 1, DiSCOTech provides options to project infeasible initial points, normalize search directions, scale variables, limit the total number of iterations, and limit the total number of objective function evaluations. DiSCOTech is coded in the programming language of the numerical computing environment MATLAB to take advantage of MATLAB's standard library of advanced mathematical functions and powerful syntax for matrix and vector operations. The original code consisted of a set of methods that operated on global data structures containing the problem parameters and working variables. The code was invoked with the optimization problem parameters and proceeded to run a loop trying an exploratory move and, if the exploratory move failed, falling back on a variant of the direct search algorithm around the current iterate. A note on nomenclature: the code refers to a direct search as a *local search* and an exploratory step as a *trial step*.

Determining which heuristic improvement techniques (Pattern Step vs. Active Set Step, Sequential Search vs. Polar Search) were used was handled by checking the values of feature flags at various steps of the computation. This allowed DiSCOTech to provide a wide range of options in solving optimization problems, but led to large

logic structures to deal with the combination of different options. This increased the size of methods, meant that it was difficult to add new functionality without having to add additional logic, and increased the chances of unforeseen option interactions. Additionally, as features were added and improved certain flags and features became obsolete but remained in the codebase.

2.2 Object-Oriented Restructuring

In order to address the issues present in the current code, I decided to reimplement DiSCOTech using an object-oriented design pattern. The structure of the DiSCOTech code lent itself very well to such an organization. The concept of different trial points and local search methods translated naturally to hierarchically-related classes. In addition, separating functionality into classes provided the opportunity to group flags with their related features within a class, rather than having the flags be global. Additionally, during this process of moving to an object-oriented design, outdated flags could be identified and removed. The hierarchical relationship among features is summarized as:

- **Local Search Method**
  - Standard local search
  - Local search with search directions oriented towards constraints that are possibly active at the solution
  - Local search with an active set step after an unsuccessful iteration
- **Local Search Algorithm**
  - Sequential Search
  - Polar Search
- **Exploratory Steps**
  - Pattern Step
  - Active Set Step
  - Combination Pattern Step and Active Set Step

Using these relationships the DiSCOTech codebase was carefully restructured while maintaining the underlying algorithm. The resulting design is shown in the UML diagram in Figure 2.

The `DiSCOTechSolver` class acts as the driver class for solving optimization problems. It requires an instance of the `OptimizationProblem` class that defines problem parameters such as the objective function, initial point, and linear constraints. The `OptimizationProblem` class provides some basic data checking to ensure that the constraints are logical and sets default values for parameters such as the stopping

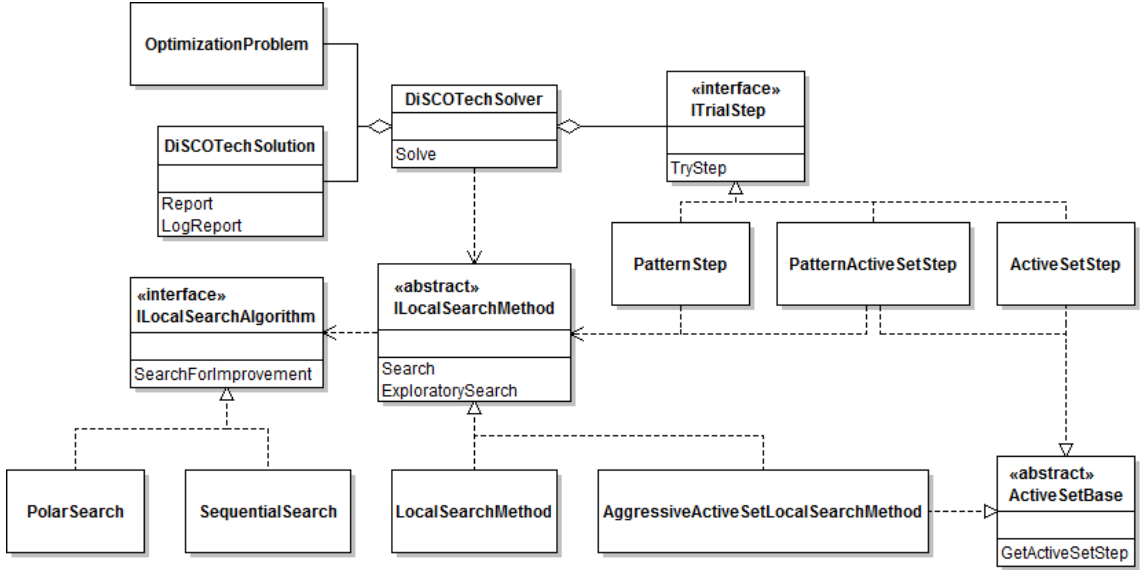


Figure 2: *UML diagram of the restructured, object-oriented DiSCOTech code*

tolerance, iteration limit, and objective evaluation budget. An optimization computation is started by invoking the `Solve` method of the `DiSCOTechSolver` class. This method requires an `OptimizationProblem`, a local search method that implements the `ILocalSearchMethod` abstract class, and an optional trial step that implements the `ITrialStep` interface. The `Solve` method follows the same procedure as the original code: attempt a trial step and, if it is unsuccessful, perform a local search.

Classes that implement the `ILocalSearchMethod` abstract class perform a variant of direct search similar to that described in Section 1.1. DiSCOTech provides the `LocalSearchMethod` class for standard local searches and the `AggressiveActiveSetLocalSearchMethod` class for local searches that orient search directions toward possible active constraints and attempt active set trial steps after unsuccessful iterations. These classes provide two methods: `Search` and `ExploratorySearch`. The `Search` method performs the usual direct search, decreasing Δ_k after an unsuccessful iteration. The `ExploratorySearch` method is used by trial steps to search around a trial point without decreasing Δ_k. The `ILocalSearchMethod` abstract class relies on a class that implements the `ILocalSearchAlgorithm` interface to actually evaluate test points at each of the current search directions. This is handled through the `SearchForImprovement` method. DiSCOTech provides the `SequentialSearch` and `PolarSearch` classes.

Trial steps are handled through the `ITrialStep` interface. Classes that implement this interface provide a `TryStep` method that attempts a trial step. DiSCOTech provides the `PatternStep`, `ActiveSetStep`, and `PatternActiveSetStep` classes. The pattern step classes `PatternStep` and `PatternActiveSetStep` also rely on the `ILocalSearchMethod` to perform searches around the trial points they generate. The active set based trial steps `ActiveSetStep` and `PatternActiveSetStep`, in

addition to the active set based local search `AggressiveActiveSetLocalSearchMethod`, rely on the abstract `AcitveSetBase` class to calculate active set steps.

A new logging system was implemented to track the progress of DiSCOTech as it iterated. Almost every method provides a log object back to its caller providing details about the execution of the method. DiSCOTech provides two log objects, `LogEntry` and `IterationLogEntry` that both implement the `ILogEntry` interface. The `ILogEntry` interface specifies a method to output in a human-readable form the data contained in the log object. The `IterationLogEntry` class is used by the `DiSCOTechSolver` to record the outcome of one iteration of the `Solver` method. The `IterationLogEntry` class contains information such as the success of an iteration and the best objective value found during that iteration. In addition, it contains a list of `LogEntry` objects describing the outcome of any trial steps and/or local searches performed during that iteration. The logging objects' relations with other classes are shown in the UML diagram in Figure 3.

DiSCOTech returns a `DiSCOTechSolution` object once one of the stopping criteria is satisfied. This object that contains the calculated optimal objective value, the point at which that value occurs, information about the parameters of the optimization problem that was solved, and the series of log objects generated by the `DiSCOTechSolver` object. The `DiSCOTechSolution` class provides the methods `Report` and `LogReport` to output in a human-readable format the results of the optimization run. The `Report` method outputs general summary information while the `LogReport` method outputs the contents of the log objects.

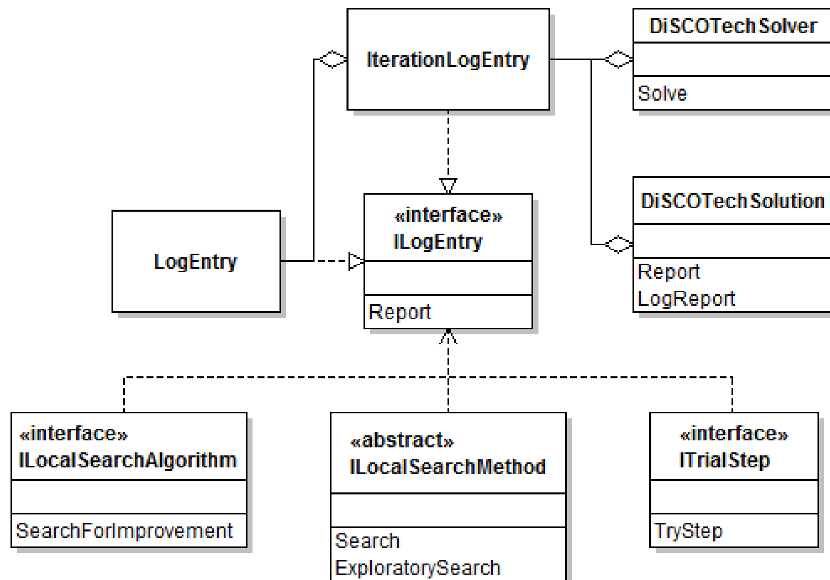


Figure 3: *UML diagram of the object-oriented logging system used by the restructured, object-oriented DiSCOTech code*

Using this object-oriented approach attempts to solves the issues discussed in Section 2.1. By separating the local search method, local search algorithm, and trial

point functionality, new features can be easily be added using the provided interfaces. Additional, the choice of which features to use during a run of DiSCOTech is handled by providing the appropriate objects. The logic for determining which features to enable is processed only once before the start of the optimization run. Once the appropriate objects have been created there is no need to evaluate this logic again to decide which features to use.

3 Cache

3.1 Motivation

One of the primary areas where direct search methods are useful is simulation-based optimization [6]. Many times the derivatives of the function underlying a simulation cannot be computed because of the design of the simulation software. Direct search methods provide a way to optimize the parameters of such simulations by simply evaluating the function, i.e. running the simulation. The down-side to this approach is that direct search methods must repeatedly sample the output of the simulation. This can amount to a significant number of runs of the simulation and require a great deal of time. A natural strategy to minimize this limitation is to store previously computed values of the objective so that they might be used again during later iterations. This establishes the need for an efficient data structure to store and recover test points and their associated objective function values.

3.2 Designing a Cache

An initial complication in designing a cache is that all computations take place in floating-point arithmetic. The floating-point errors that can accrue over the course of a computation require a relaxed notion of equivalence in order to match points in a cache. To handle this, the ϵ-metric proposed in [4] is used. This metric states that $x, y \in \mathbb{R}^n$ are ϵ-equal if

$$|x_i - y_i| \leq \epsilon \quad \text{for } i = 1, \ldots, n$$

and that x is considered ϵ-less than y if there exists an index j such that

$$|x_i - y_i| \leq \epsilon \quad \text{for } i = 1, \ldots, j-1 \qquad \text{and} \qquad y_j - x_j > \epsilon.$$

This notion of order is introduced in [4] to make use of the ϵ-metric in a *splay tree* [11].

The idea of using a splay tree or other binary search tree (BST) data structure seems immediately appealing. Binary search trees are a staple data structure for providing efficient solutions to various data storage and searching problems. Compared to list-based solutions, if managed in an appropriate fashion binary search trees average $O(\log N)$ time complexity for insertion, retrieval, and deletion. By imposing additional conditions, binary search trees can guarantee even better performance. *AVL trees* [1], which enforce balancing conditions, and *red-black trees* [2, 5], which

enforce coloring conditions, are two examples of binary search trees that guarantee $O(\log N)$ time complexity for insertion, retrieval, and deletion.

However, initial tests of BST-based caches revealed unexpected searching behavior. Occasionally, a search would yield a false negative, i.e. claim that a point in the tree was not actually present. A critical requirement of binary search trees is that there is a total order defined on the value of the nodes in the tree. The ϵ-metric does not fulfill this requirement and therefore provides no guarantees of proper search behavior. Unfortunately, ϵ-metric based BST caches can operate successfully while subtle errors appear in the tree structure. In order to understand the subtlety and danger of this behavior, it is illustrative to consider several examples.

3.2.1 Binary Search Tree

Using the ϵ-metric with $\epsilon = 1$, insert the points

$$\begin{bmatrix} 8 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \begin{bmatrix} 6 \\ 9 \end{bmatrix}, \begin{bmatrix} 7 \\ 2 \end{bmatrix}, \begin{bmatrix} 22 \\ 1 \end{bmatrix}, \begin{bmatrix} 3 \\ 12 \end{bmatrix} \tag{1}$$

in the order given into a binary search tree. The resulting tree (Figure 4), although constructed correctly, is not a BST and does not operate as expected. There are

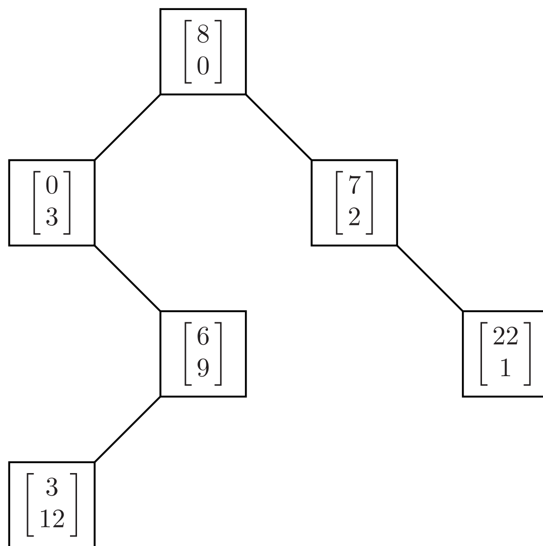


Figure 4: *Tree resulting from the sequence of insertions in (1)*

two problems with the tree's expected behavior. The first problem is that the binary search tree insertion and retrieval operations are no longer guaranteed to function as expected because the tree is no longer a binary search tree. This problem is caused by the relative positions of the nodes containing $\begin{bmatrix} 8 & 0 \end{bmatrix}^T$, $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ and $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$. Assuming that there are no duplicate nodes, a BST is defined by the following rules:

- Each node has a value

- There is a *total order* defined on the values of the nodes
- A node's left sub-tree contains only values less than the value of the node
- A node's right sub-tree contains only values greater than the value of the node

The positions of these three nodes violate these properties. The nodes containing $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ and $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ are positioned correctly in relation to the root $\begin{bmatrix} 8 & 0 \end{bmatrix}^T$ ($\begin{bmatrix} 6 & 9 \end{bmatrix}^T < \begin{bmatrix} 8 & 0 \end{bmatrix}^T$ meaning it should be in the left sub-tree and $\begin{bmatrix} 7 & 2 \end{bmatrix}^T > \begin{bmatrix} 8 & 0 \end{bmatrix}^T$ meaning that it should be in the right sub-tree). However, this implies that

$$\begin{bmatrix} 6 \\ 9 \end{bmatrix} < \begin{bmatrix} 7 \\ 2 \end{bmatrix}$$

which is not the case according to the ϵ-metric.

The second problem is that the resulting tree does not satisfy the requirement of a cache, i.e. that points which are similar according to the ϵ-metric match. Consider finding the point $\begin{bmatrix} 7 & 9 \end{bmatrix}^T$ in the tree. According to the ϵ-metric,

$$\begin{bmatrix} 7 \\ 9 \end{bmatrix} = \begin{bmatrix} 6 \\ 9 \end{bmatrix},$$

so the search of the tree should return $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$. However, this is not the case. The first step of the search compares $\begin{bmatrix} 8 & 0 \end{bmatrix}^T$ and $\begin{bmatrix} 7 & 9 \end{bmatrix}^T$ and moves to the right sub-tree because

$$\begin{bmatrix} 7 \\ 9 \end{bmatrix} > \begin{bmatrix} 8 \\ 0 \end{bmatrix}.$$

Because $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ is in the left sub-tree of $\begin{bmatrix} 8 & 0 \end{bmatrix}^T$, it is obvious that the search will never reach $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$. Ultimately, the search follows the path

$$\begin{bmatrix} 8 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 7 \\ 2 \end{bmatrix} \rightarrow \begin{bmatrix} 22 \\ 1 \end{bmatrix} \rightarrow \texttt{null}$$

concluding that $\begin{bmatrix} 7 & 9 \end{bmatrix}^T$, or any ϵ-equal point, is not in the tree. This means that a binary search tree using the ϵ-metric cannot be guaranteed to match points considered equal under the metric.

However, the binary search tree will correctly match points that are exactly equal (i.e. $x_i = y_i$ for $i = 1, \ldots, n$). This is guaranteed because the tree structure is never rearranged, meaning that the relative positions of nodes never change. The search of the tree for a particular point will always follow the same path. However, in dealing with trees that are rearranged under a series of insertions a more serious problem can arise, as illustrated by considering the use of the ϵ-metric with an AVL tree.

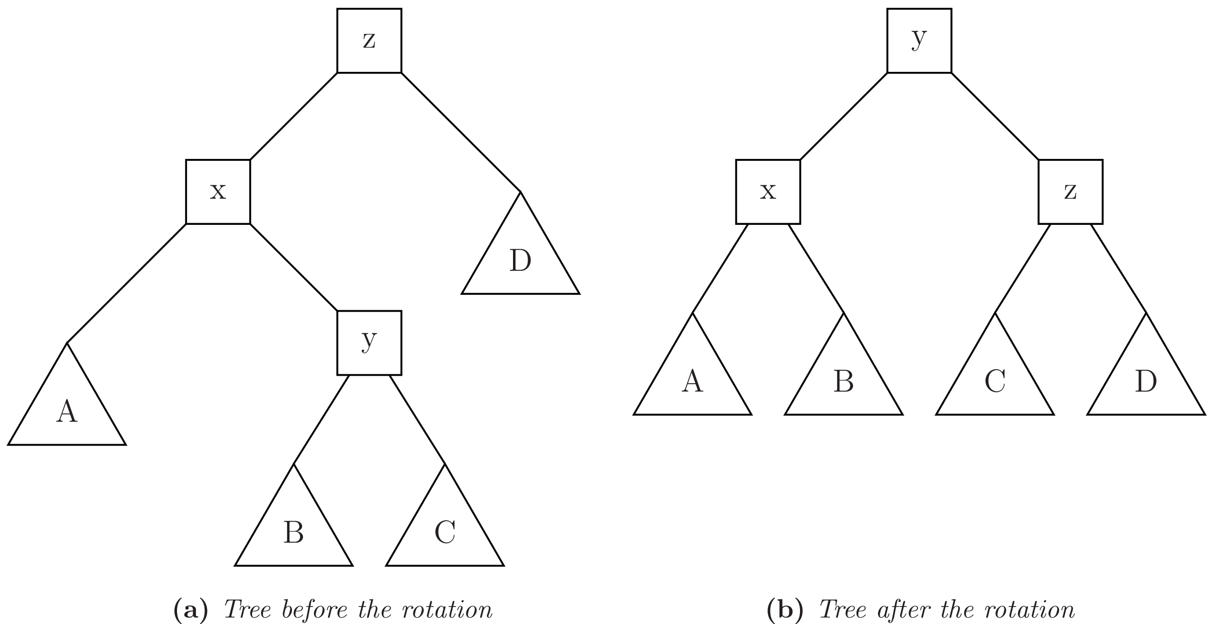
3.2.2 AVL Tree

An AVL tree [1] is a binary search tree that ensures better time complexities for insertion, retrieval, and deletion by enforcing a balancing condition. As new elements are inserted an AVL tree is rearranged, when appropriate, to maintain balance. This rearrangement is accomplished using tree rotations.

Tree rotations are a standard means of rearranging a binary search tree while preserving both its structural and ordering properties. One type of tree rotation is illustrated in Figure 5. The AVL tree contains three distinguished nodes x, y, and z, and four possibly empty sub-trees A, B, C, and D. Assuming no duplicate nodes the total order property implies that

$$\forall\, a \in A,\ b \in B,\ c \in C,\ d \in D \qquad a < x < b < y < c < z < d. \tag{2}$$

The rotation alters the structure of the tree to move the node containing y to the root while preserving (2). The ability to preserve the ordering properties under a



(a) *Tree before the rotation* **(b)** *Tree after the rotation*

Figure 5: *Binary search tree rotation that preserves (2)*

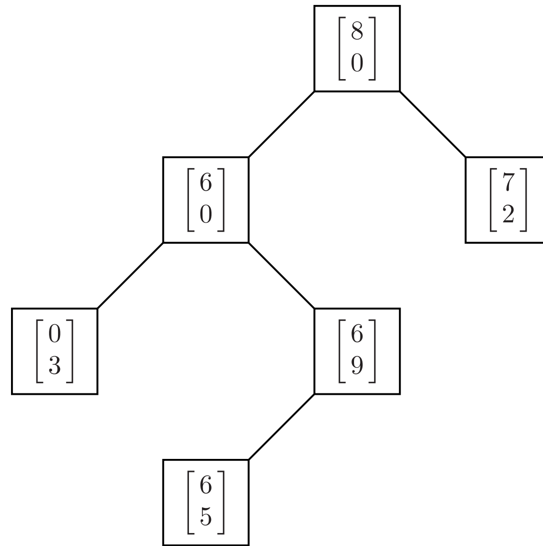
sequence of rotations depends in a critical way on the assumption of a total order, as is demonstrated in the following example.

Using the ϵ-metric with $\epsilon = 1$, insert the points

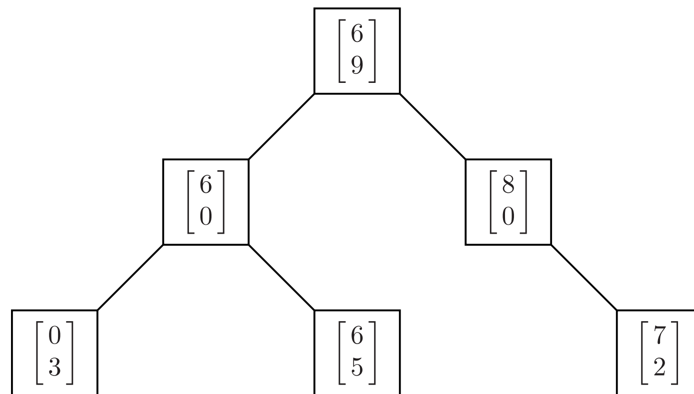
$$\begin{bmatrix}8\\0\end{bmatrix}, \begin{bmatrix}6\\0\end{bmatrix}, \begin{bmatrix}7\\2\end{bmatrix}, \begin{bmatrix}0\\3\end{bmatrix}, \begin{bmatrix}6\\9\end{bmatrix}, \begin{bmatrix}6\\5\end{bmatrix} \tag{3}$$

in the order given into an AVL tree. The order of insertion is such that only the final insertion results in a rotation. The tree before and after the rotation is illustrated

in Figure 6. Although the problems that arose with a regular binary search can still occur in an AVL tree, the focus here is on a new problem that arises once rotations are introduced: the inability to find even *exact* matches.



(a) *Tree before the rotation*



(b) *Tree after the rotation*

Figure 6: *Tree resulting from the sequence of insertions in (3)*

Recall the discussion of the three nodes containing $\begin{bmatrix} 8 & 0 \end{bmatrix}^T$, $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ and $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ from Section 3.2.1. The tree illustrated in Figure 6a contains a similar positioning of these nodes. However, the rotation alters the relative position of these nodes so that the node containing $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ becomes an ancestor of $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ with $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ being in the right sub-tree of $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$. The resulting tree (Figure 6b) is not a binary search tree because this structure implies that $\begin{bmatrix} 7 & 2 \end{bmatrix}^T > \begin{bmatrix} 6 & 9 \end{bmatrix}^T$ which is not true according to

the ϵ-metric. Unlike the discussion of binary search trees in Section 3.2.1, this tree violates the requirement that all the values in a node's right sub-tree are greater than the value of the node. The result of this is that the point $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ cannot be found. The search follows the path

$$\begin{bmatrix} 6 \\ 9 \end{bmatrix} \rightarrow \begin{bmatrix} 6 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 6 \\ 5 \end{bmatrix} \rightarrow \texttt{null}$$

concluding that $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ is not in the tree. The current configuration of the tree, with $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ as an ancestor of $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$, prevents the point from being found because $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$ directly influences the search path. All searches for $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ must pass through $\begin{bmatrix} 6 & 9 \end{bmatrix}^T$, but because $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$ is in the incorrect sub-tree, no search will ever find it.

This problem is a consequence of applying the rotation on a tree that did not fulfill the total order requirement for binary search trees. Any search of the tree before the rotation follows the path

$$\begin{bmatrix} 8 \\ 0 \end{bmatrix} \rightarrow \begin{bmatrix} 7 \\ 2 \end{bmatrix}$$

and correctly finds $\begin{bmatrix} 7 & 2 \end{bmatrix}^T$. Only after the rotation is the point unreachable.

Additionally, because rotations are fundamental to managing many other specialized binary search trees such as red-black trees [2, 5] and splay trees [11] those trees are also susceptible to this type of error when using the ϵ-metric.

3.2.3 Generalized Binary Search Tree

The binary search tree and AVL tree examples show some of the problems that arise when using the ϵ-metric. Because the root cause involves the violation of the total order property, all binary trees are susceptible to these type of errors. To illustrate this consider a very simple binary search tree (Figure 7) comprised only of the points

$$\begin{bmatrix} 8 \\ 0 \end{bmatrix}, \begin{bmatrix} 6 \\ 9 \end{bmatrix}, \begin{bmatrix} 7 \\ 2 \end{bmatrix} \tag{4}$$

inserted in the order given using the ϵ-metric with $\epsilon = 1$. Even with only three nodes,

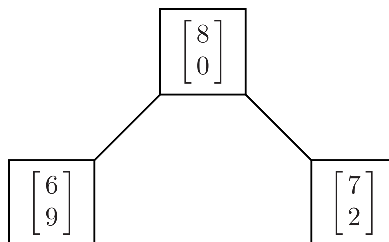


Figure 7: *Simple tree resulting from the insertion sequence (4)*

the tree is not a valid binary search tree. Because this simple three node tree is a

general form for many different types of binary search trees, it serves as an example of an invalid tree for many different types of binary search trees. Additionally, it shows the ease with which errors can occur in very general conditions.

3.3 Importance of a Total Order

3.3.1 Total Order and the ϵ-metric

For a set $\mathcal{S}$ to be totally ordered under $<$, *transitivity* must hold for all $a, b, c \in \mathcal{S}$. Specifically,

$$\text{If } a < b \text{ and } b < c \text{ then } a < c.$$

The fundamental difficulty with the ϵ-metric is that it is not transitive. The simple tree discussed in Section 3.2.3 (Figure 7) provides a straightforward illustration. Although $\begin{bmatrix}6 & 9\end{bmatrix}^T < \begin{bmatrix}8 & 0\end{bmatrix}^T$ and $\begin{bmatrix}8 & 0\end{bmatrix}^T < \begin{bmatrix}7 & 2\end{bmatrix}^T$, $\begin{bmatrix}6 & 9\end{bmatrix}^T > \begin{bmatrix}7 & 2\end{bmatrix}^T$. The ϵ-metric is not transitive and is therefore not a total order. This lack of transitivity causes all the unexpected behavior in the binary search trees because the relationship between the values of a node's left and right sub-trees derives from the transitive property of a total order.

3.3.2 Subtle Mistakes

Although a binary search tree constructed using the ϵ-metric cannot be guaranteed to preserve order, it can still operate successfully in many cases. The examples provided so far have been intentionally constructed to produce unexpected behavior rapidly, but that is not always the case. The chance that errors will be introduced depends on the choice of ϵ and the scale of the points being inserted in the tree. The lexicographic metric without the concept of an ϵ is a total order, so the longer the ϵ-metric mimics the behavior of a strict lexicographic metric, the longer the tree will function without errors. If ϵ is chosen to be relatively small compared to the distance between corresponding elements of the points in the tree, then two points are less likely to be considered ϵ-equal. If no points are ϵ-equal then the tree is constructed in the same manner as using the lexicographic metric and will function properly. Under these conditions a tree can function properly for some time while still remaining a binary search tree.

The previous examples highlight unexpected behavior when searching for specific points. However, not all searches for points in a faulty tree will fail. In the case of the regular binary search tree (Figure 4), the vector $\begin{bmatrix}8 & 2\end{bmatrix}^T$ is matched correctly to $\begin{bmatrix}7 & 2\end{bmatrix}^T$ even though $\begin{bmatrix}7 & 9\end{bmatrix}^T$ could not be matched to $\begin{bmatrix}6 & 9\end{bmatrix}^T$. In the case of the AVL tree (Figure 6), $\begin{bmatrix}7 & 2\end{bmatrix}^T$ is the only point that cannot be found by an exact search. In all cases, the possibility for errors depends on the current configuration of the tree, which depends on the prior sequence of insertions. An example of this is the tree illustrated in Figure 8. This tree contains the same points as the tree in Figure 6,

but in a different configuration given by the insertion sequence

$$\begin{bmatrix}7\\2\end{bmatrix}, \begin{bmatrix}6\\0\end{bmatrix}, \begin{bmatrix}6\\9\end{bmatrix}, \begin{bmatrix}0\\3\end{bmatrix}, \begin{bmatrix}8\\0\end{bmatrix}, \begin{bmatrix}6\\5\end{bmatrix}. \quad (5)$$

It is straightforward to verify that exact searches for any of these points will execute correctly.

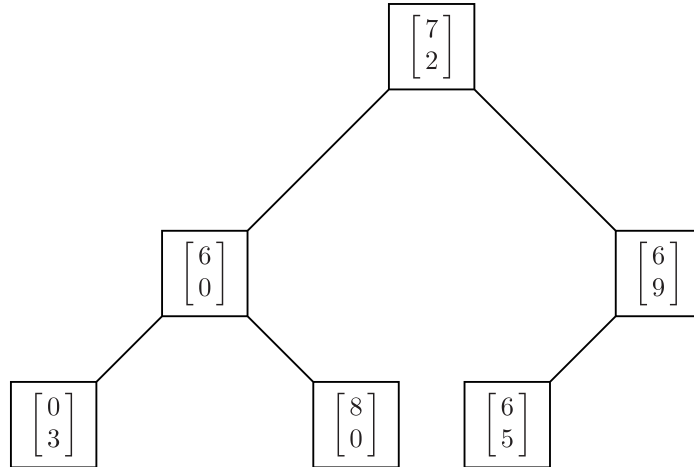


Figure 8: *Tree resulting from the AVL insertion sequence (5) rather than (3), which correctly supports all exact searches*

The primary concern is the uncertainty regarding the behavior of the tree. Depending on the current configuration of a tree, errors may or may not arise during the course of a search. Additionally, tree rotations may introduce the possibility for errors that are corrected by subsequent rotations. Furthermore, while a particular point may be unreachable by an exact search during a certain stage of a tree's execution, if there is no search for this point during this stage then no error will occur — even though the possibility for a false negative exists. The same goes for similarity searches. Some points may not match based on ϵ-equality at certain stages of a tree's execution. However, depending on when and if those searches are performed, those errors may never be detected. The subtleties of these errors no doubt explain why the possibility for false negative searches went undetected by the authors of [4].

3.4 Revisiting Alternatives

3.4.1 Selecting a New Design

Although the ϵ-metric does not function with binary search trees, it still has some advantages. It addresses the problem of floating-point error and is easy to compute, requiring only a simple iteration over each point. The worst-case time complexity is $O(n)$, though the ϵ-metric can be implemented so as to terminate as soon as it becomes clear that points cannot be ϵ-equal or are ϵ-less. Maintaining the same metric requires a change in data structure. An obvious simple alternative is a list using an exhaustive

search. A list-based approach will not perform as well as a binary search tree-based approach but prevents the errors that are associated with the ϵ-metric under binary search trees.

Using a list comes with a standard set of design decisions: should the data be stored in a strict lexicographic order in order to speed up searches, and should an array or linked list be used. In terms of the lexicographic order, consider searching for $\begin{bmatrix} 3 & 0 \end{bmatrix}^T$ in the following ordered list

$$\begin{bmatrix} 1 \\ 5 \end{bmatrix}, \begin{bmatrix} 3 \\ 2 \end{bmatrix}, \begin{bmatrix} 4 \\ 4 \end{bmatrix}, \begin{bmatrix} 6 \\ 5 \end{bmatrix}, \begin{bmatrix} 10 \\ 1 \end{bmatrix}$$

using the ϵ-metric with $\epsilon = 1$. An exhaustive search by comparison will conclude that there are no ϵ-equal matches. However, because the list is ordered, the search can terminate after comparing $\begin{bmatrix} 3 & 0 \end{bmatrix}^T$ with $\begin{bmatrix} 3 & 2 \end{bmatrix}^T$, which is ϵ-greater. The ordering provides the potential for terminating searches early but carries the extra cost of maintaining the lexicographic order.

In terms of whether to use an array or a linked list, the standard trade-offs apply. Arrays provide regular memory access and the limit on the number of objective function evaluations used in DiSCOTech provides a natural default for choosing the size of an array. Linked lists have an advantage when storing data in order because inserting a new node requires only changing node links. Arrays require a costly shift operation to create space for a new node. The insertion and retrieval efficiencies of array and linked list approaches are summarized in Table 1.

		Insertion Order	**Lexicographic Order**
Array	Insertion	$O(1)$	$O(n)$
	Retrieval	$O(n)$	$O(n)$
Linked List	Insertion	$O(1)$	$O(n)$
	Retrieval	$O(n)$	$O(n)$

Table 1: *Insertion and retrieval efficiencies of array and linked list cache approaches*

In order to determine which cache strategy provided the best performance, each strategy was implemented and time tests were conducted. The ordered array was not implemented and tested because the overhead associated with the shifting during insertion was assumed to be too inefficient.

3.4.2 Testing

To overcome some of the performance limitations in MATLAB code that utilizes large loop structures, the various list-based schemes were implemented in C and then converted to MATLAB MEX-files. MEX-files are dynamically linked subroutines produced from C or Fortran source code that, when compiled, can be run from within MATLAB in the same way as MATLAB M-files or built-in functions [10]. Additionally, the use of C code allowed addressing optimizations to be performed by hand for the array strategy.

All tests were performed on an dual-core 2.40 GHz Intel Xeon server with 1 GB of RAM running openSUSE 10.3 (kernel version 2.6.22.19) and using MATLAB 7.7.0 (R2008b).

In order to properly gauge the relative effectiveness of each strategy, a set of reasonably realistic test problems was necessary. A subset of the nonlinear programming problems present in the CUTEr [3] test suite were selected. A summary of the salient problem characteristics, as well as the maximum number of objective function evaluations allowed is given in Table 2.

Problem	**n**	**CUTEr Classification**	**Max Evals.**
AVION2	49	OLR2-RN-49-15	1500
BLEACHNG	17	SBR1-RN-17-0	540
DALLASS	46	ONR2-MN-46-31	1410
DALLASM	196	ONR2-MN-196-151	5910
EXPLIN	128	OBR2-AN-V-V	3870
HIMMELBI	100	OLR2-MN-100-12	3030
LOADBAL	31	OLR2-MN-31-31	960
SPANHYD	97	OLR2-RN-97-33	2940
WATER	31	ONR2-MN-31-10	960

Table 2: *Summary of CUTEr test problem characteristics*

3.4.3 Results

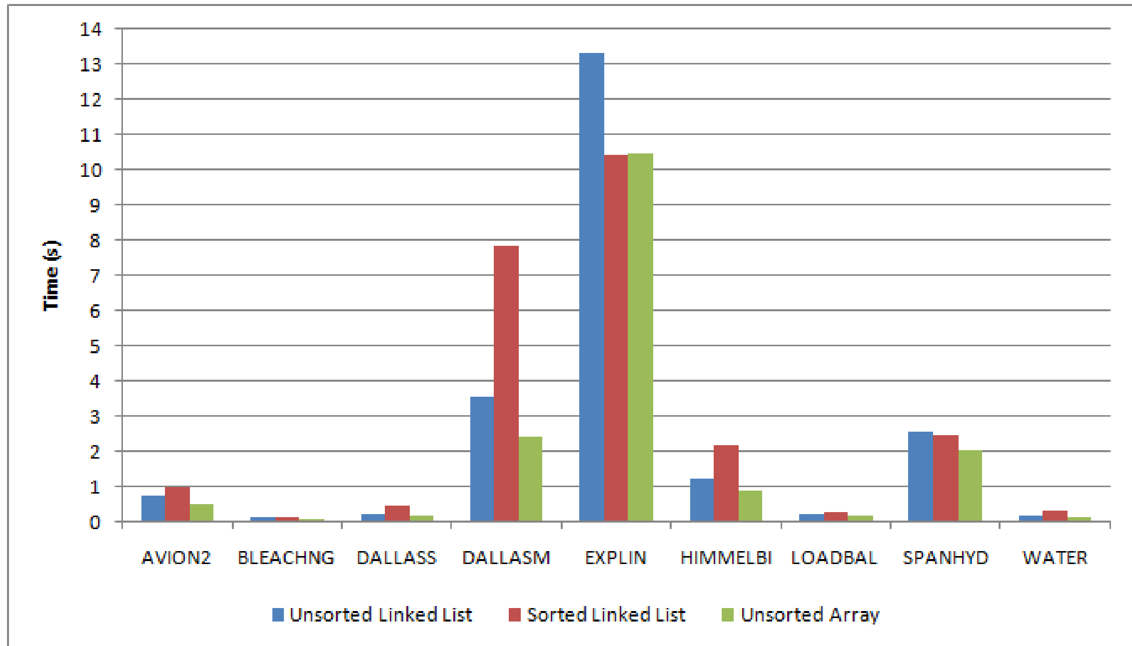
Each test was run three times to calculate the average amount of time (in seconds) that DiSCOTech spent in the cache while performing a computation. The results, summarized in Figure 9, show that average performance of the unsorted linked list strategy, sorted linked list strategy, and unsorted array strategy. A two-sample two-tailed t-test with $p = 0.05$ was used to determine if times were significantly different. Based on this data, the conclusion is that the unsorted array strategy out performs both linked list strategies. In every test except EXPLIN (where the array strategy was not significantly different from the sorted linked list strategy, but still faster than the unsorted linked list strategy) the array strategy took less time. This is mostly likely due to the array strategy's regular memory access patterns and optimized addressing. In terms of the linked list strategies, the unsorted linked list performed better in all but two cases, EXPLIN (13.3066 seconds vs. 10.4271 seconds) and SPANHYD (2.5632 seconds vs. 2.4498 seconds). It appears that, in general, the overhead associated with maintaining the lexicographic order proved too costly.

The original motivation of using a splay tree in [4] was to keep the most recently accessed nodes near the root for quick access. This takes advantage of the fact that direct search methods tend to concentrate objective evaluations in the neighborhood $\mathcal{S}$ the search believes contains a solution. Although a splay tree cannot be used with the ϵ-metric, this behavior can be partially mimicked by searching the array-based cache backwards, comparing the most recently inserted points first.

The unsorted array backward searching strategy was implemented (source code provided in Section 6) and tested against the unsorted array forward search strategy. The results are summarized in Figure 10. Again, a two-sample two-tailed t-test with $p = 0.05$ was used to determine if times were significantly different. Based on the data, the backward searching array outperformed the forward searching array. In six of the test problems (AVION2, DALLASM, EXPLIN, LOADBAL, SPANHYD, and WATER) the backward searching array was significantly faster, and in the remaining test problems (BLEACHNG, DALLASS, and HIMMELBI) there was no significant difference. The conclusion that can be drawn from these results is that the backward searching array will do no worse than the forward searching array, and in most cases will outperform the forward searching array.

Problem	Unsorted Linked List	Sorted Linked List	Unsorted Array
AVION2	0.7570	0.9962	0.5210
BLEACHNG	0.1087	0.1129	0.0561
DALLASS	0.2412	0.4392	0.1491
DALLASM	3.5709	7.8386	2.4239
EXPLIN	13.3066	10.4271	10.4590
HIMMELBI	1.2075	2.1690	0.8642
LOADBAL	0.1956	0.2445	0.1625
SPANHYD	2.5632	2.4498	2.0149
WATER	0.1717	0.3144	0.1254

(a) *Table of cache timing results*

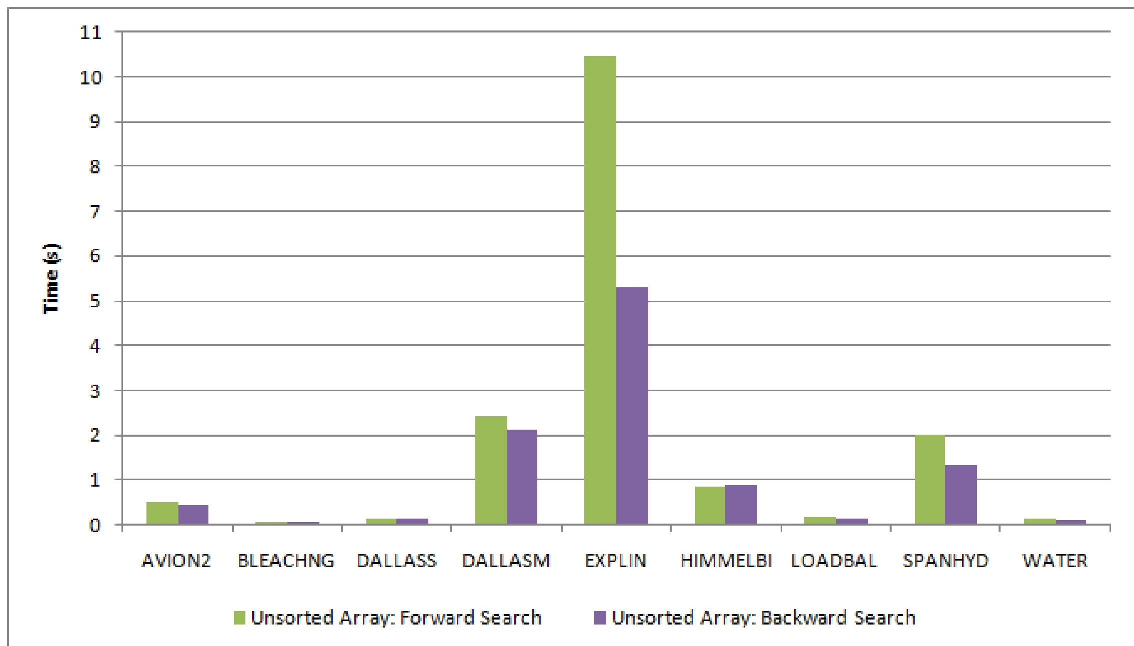


(b) *Graph of cache timing results*

Figure 9: *Total time (in seconds) spent in the cache by DiSCOTech on CUTEr test problems for the unsorted linked list, sorted linked list, and unsorted array caching strategies*

Problem	Unsorted Array: Forward Search	Unsorted Array: Backward Search
AVION2	0.5210	0.4378
BLEACHNG	0.0561	0.0564
DALLASS	0.1491	0.1471
DALLASM	2.4239	2.1303
EXPLIN	10.4590	5.3118
HIMMELBI	0.8642	0.8718
LOADBAL	0.1625	0.1319
SPANHYD	2.0149	1.3286
WATER	0.1254	0.1024

(a) *Table of cache timing results*



(b) *Graph of cache timing results*

Figure 10: *Total time (in seconds) spent in the cache by DiSCOTech on CUTEr test problems for the unsorted array: forward search and unsorted array: backward search caching strategies*

4 Comparing Heuristic Improvement Techniques

The heuristic improvement techniques implemented in DiSCOTech have no guarantees of providing improved performance. In the worst case, the extra objective evaluations associated with these techniques can waste time and prematurely exhaust the objective evaluation budget. Therefore, in order to understand the relative effectiveness of each technique, comparative tests were performed.

4.1 Comparison Setup

A set of reasonably realistic test problems was necessary. As in Section 3.4.2, a subset of the nonlinear programming problems present in the CUTEr test suite (summarized in Table 3) were selected.

The different types of heuristic improvement techniques provide three dimensions of comparison: the local search method, the local search algorithm, and the use of a trial step. The options in these three dimensions provided by the object-oriented DiSCOTech code, described in Section 2.2, are summarized in Table 3. Each combination of options was tested to determine if a particular combination of options yielded better results.

	Local Search Method	**Local Search Algorithm**	**Trial Step**
Basic	Local Search	Sequential Search	No Trial Step
Heuristic	Active Set Local Search	Polar Search	Pattern Step Active Set Step Pattern Active Set Step

Table 3: *Summary of heuristic improvement technique test options*

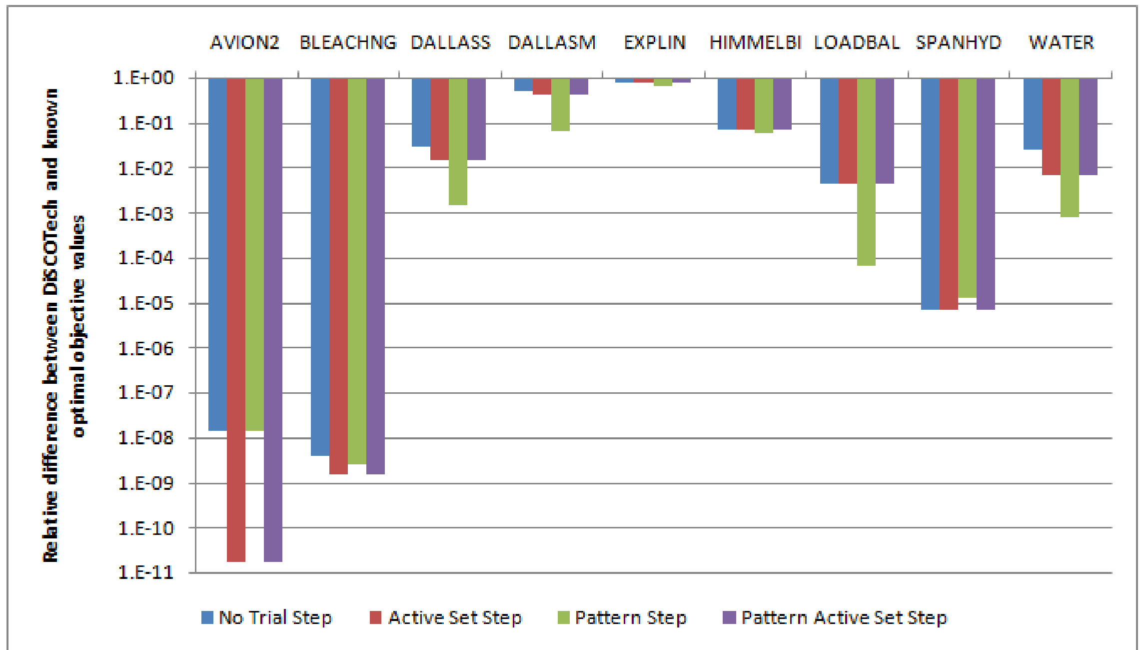
The measure of performance used was the relative difference between the optimal objective determined by DiSCOTech and the known optimal objective value provided by the CUTEr test suite. A combination of options provides a benefit if it allows DiSCOTech to determine a more accurate answer within the same objective evaluation budget, iteration limit, and Δ_{tol} constraints.

4.2 Results

Results are presented in Figures 11, 12, 13, and 14. Data are grouped by the combination of local search method and local search algorithm. For each such combination the relative differences between the optimal objective values determined by DiSCOTech and the known optimal objective values are presented as a table and a graph for each type of trial point.

Problem	No Trial Step	Active Set Step	Pattern Step	Pattern Active Set Step
AVION2	$1.3581460 \times 10^{-08}$	$1.6780309 \times 10^{-11}$	$1.3581456 \times 10^{-08}$	$1.6780309 \times 10^{-11}$
BLEACHNG	$3.7215319 \times 10^{-09}$	$1.5160023 \times 10^{-09}$	$2.4865639 \times 10^{-09}$	$1.5160023 \times 10^{-09}$
DALLASS	$2.9123591 \times 10^{-02}$	$1.4934443 \times 10^{-02}$	$1.4939530 \times 10^{-03}$	$1.4934443 \times 10^{-02}$
DALLASM	$4.8127483 \times 10^{-01}$	$4.2896386 \times 10^{-01}$	$6.2306471 \times 10^{-02}$	$4.2896386 \times 10^{-01}$
EXPLIN	$7.6502586 \times 10^{-01}$	$7.6502586 \times 10^{-01}$	$6.7076411 \times 10^{-01}$	$7.6502586 \times 10^{-01}$
HIMMELBI	$7.0050838 \times 10^{-02}$	$7.0050838 \times 10^{-02}$	$5.7874620 \times 10^{-02}$	$7.0050838 \times 10^{-02}$
LOADBAL	$4.4837644 \times 10^{-03}$	$4.4837644 \times 10^{-03}$	$6.5914384 \times 10^{-05}$	$4.4837644 \times 10^{-03}$
SPANHYD	$6.7932537 \times 10^{-06}$	$6.7932537 \times 10^{-06}$	$1.2199251 \times 10^{-05}$	$6.7932537 \times 10^{-06}$
WATER	$2.4025468 \times 10^{-02}$	$6.7051242 \times 10^{-03}$	$7.7960864 \times 10^{-04}$	$6.7051242 \times 10^{-03}$

(a) *Table of relative difference values*

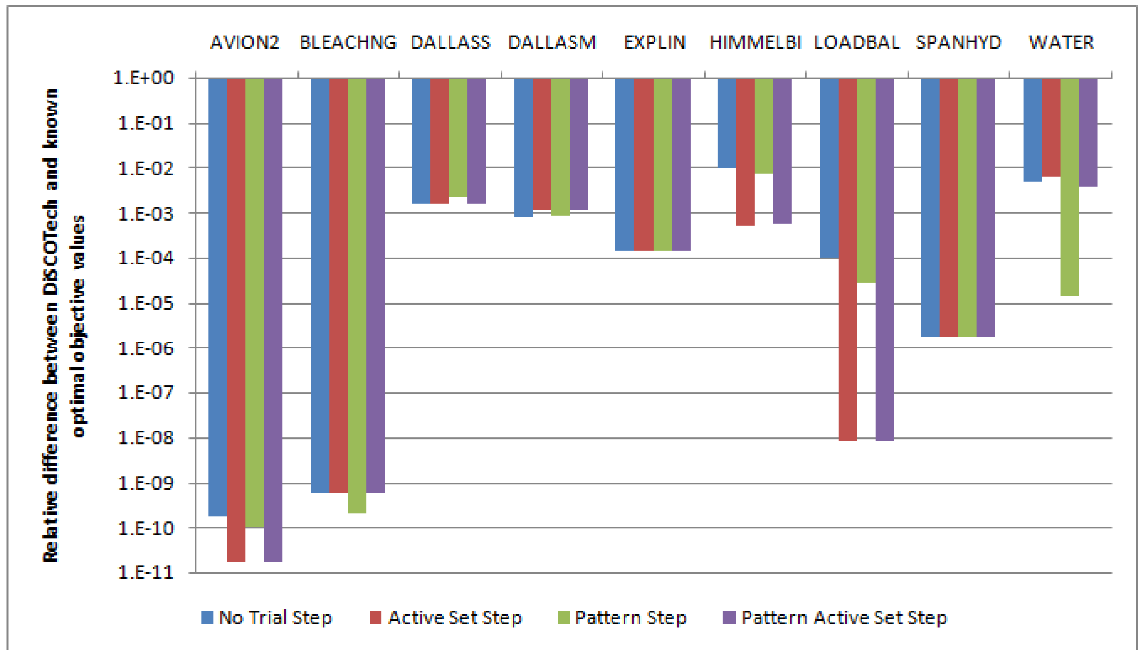


(b) *Graph of relative difference values*

Figure 11: *Relative difference between the DiSCOTech and known optimal objective values of CUTEr test problems using Local Search and Sequential Search*

Problem	No Trial Step	Active Set Step	Pattern Step	Pattern Active Set Step
AVION2	$1.6930737 \times 10^{-10}$	$1.6780309 \times 10^{-11}$	$9.8344381 \times 10^{-11}$	$1.6780309 \times 10^{-11}$
BLEACHNG	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$	$2.0498988 \times 10^{-10}$	$5.7823685 \times 10^{-10}$
DALLASS	$1.5086478 \times 10^{-03}$	$1.5869385 \times 10^{-03}$	$2.1672571 \times 10^{-03}$	$1.5869385 \times 10^{-03}$
DALLASM	$7.7200787 \times 10^{-04}$	$1.1086360 \times 10^{-03}$	$8.2185615 \times 10^{-04}$	$1.1036184 \times 10^{-03}$
EXPLIN	$1.4178793 \times 10^{-04}$	$1.4178797 \times 10^{-04}$	$1.4168499 \times 10^{-04}$	$1.4178798 \times 10^{-04}$
HIMMELBI	$9.7673569 \times 10^{-03}$	$4.9594990 \times 10^{-04}$	$7.1326597 \times 10^{-03}$	$5.7546930 \times 10^{-04}$
LOADBAL	$1.0111102 \times 10^{-04}$	$8.2940991 \times 10^{-09}$	$2.6382217 \times 10^{-05}$	$8.3420833 \times 10^{-09}$
SPANHYD	$1.7434006 \times 10^{-06}$	$1.7434006 \times 10^{-06}$	$1.7434006 \times 10^{-06}$	$1.7434006 \times 10^{-06}$
WATER	$4.7265703 \times 10^{-03}$	$6.2567691 \times 10^{-03}$	$1.3364120 \times 10^{-05}$	$3.8510993 \times 10^{-03}$

(a) *Table of relative difference values*

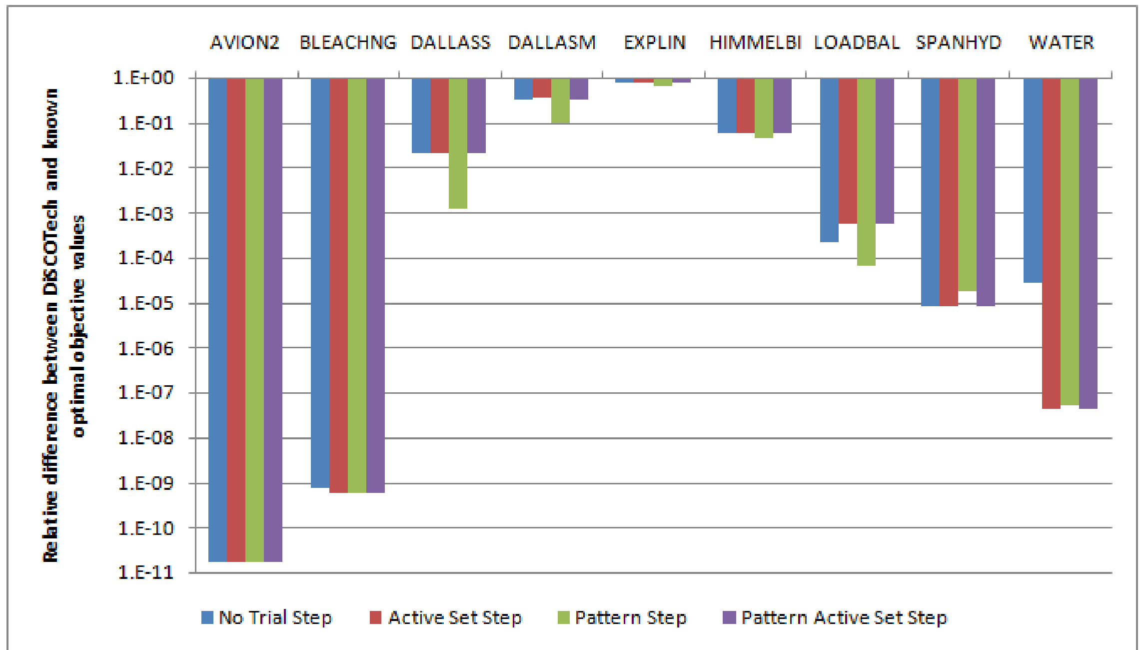


(b) *Graph of relative difference values*

Figure 12: *Relative difference between the DiSCOTech and known optimal objective values of CUTEr test problems using Local Search and Polar Search*

Problem	No Trial Step	Active Set Step	Pattern Step	Pattern Active Set Step
AVION2	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$
BLEACHNG	$7.5656535 \times 10^{-10}$	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$
DALLASS	$2.1404709 \times 10^{-02}$	$2.1404709 \times 10^{-02}$	$1.2269389 \times 10^{-03}$	$2.1404709 \times 10^{-02}$
DALLASM	$3.1810116 \times 10^{-01}$	$3.5569726 \times 10^{-01}$	$9.5648540 \times 10^{-02}$	$3.1692323 \times 10^{-01}$
EXPLIN	$7.6502586 \times 10^{-01}$	$7.6502586 \times 10^{-01}$	$6.7076411 \times 10^{-01}$	$7.6502586 \times 10^{-01}$
HIMMELBI	$5.5884833 \times 10^{-02}$	$5.6311560 \times 10^{-02}$	$4.4264523 \times 10^{-02}$	$5.6311560 \times 10^{-02}$
LOADBAL	$2.0800262 \times 10^{-04}$	$5.5893920 \times 10^{-04}$	$6.6146903 \times 10^{-05}$	$5.5893920 \times 10^{-04}$
SPANHYD	$8.4777104 \times 10^{-06}$	$8.4777104 \times 10^{-06}$	$1.7147471 \times 10^{-05}$	$8.4777104 \times 10^{-06}$
WATER	$2.6860822 \times 10^{-05}$	$4.4396100 \times 10^{-08}$	$4.9855427 \times 10^{-08}$	$4.4396100 \times 10^{-08}$

(a) *Table of relative difference values*

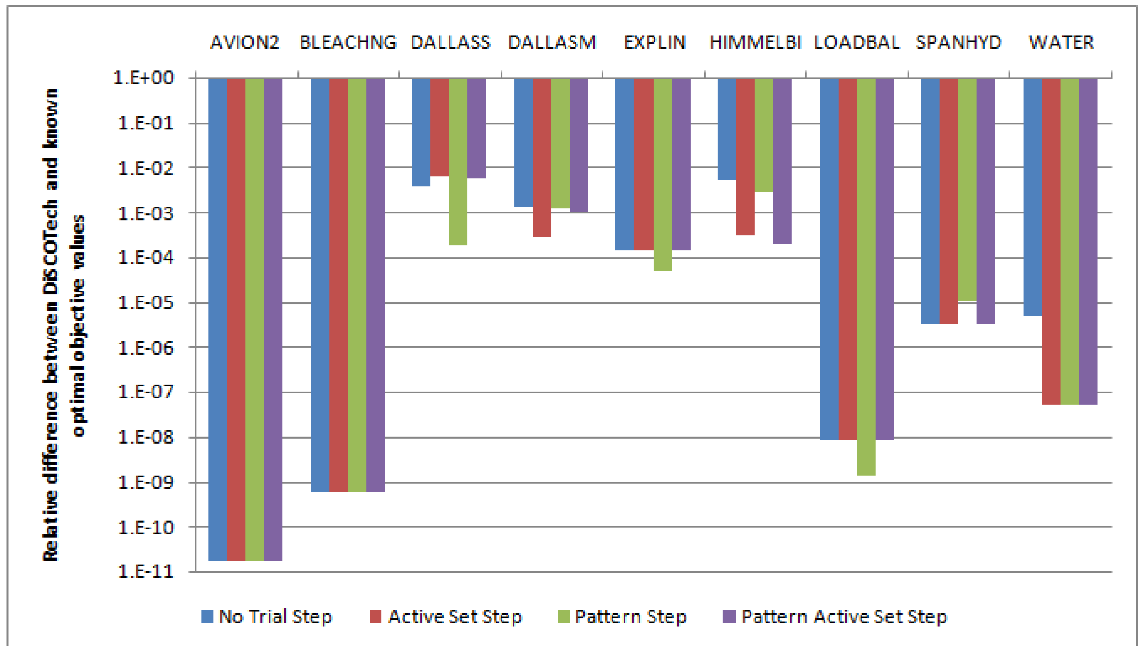


(b) *Graph of relative difference values*

Figure 13: *Relative difference between the DiSCOTech and known optimal objective values of CUTEr test problems using Active Set Local Search and Sequential Search*

Problem	No Trial Step	Active Set Step	Pattern Step	Pattern Active Set Step
AVION2	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$	$1.6785502 \times 10^{-11}$
BLEACHNG	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$	$5.7823685 \times 10^{-10}$
DALLASS	$3.7431700 \times 10^{-03}$	$6.1548594 \times 10^{-03}$	$1.8610492 \times 10^{-04}$	$5.7039261 \times 10^{-03}$
DALLASM	$1.2682750 \times 10^{-03}$	$2.7137939 \times 10^{-04}$	$1.1775976 \times 10^{-03}$	$1.0385598 \times 10^{-03}$
EXPLIN	$1.4178799 \times 10^{-04}$	$1.4178799 \times 10^{-04}$	$5.1733287 \times 10^{-05}$	$1.4178799 \times 10^{-04}$
HIMMELBI	$5.1468901 \times 10^{-03}$	$3.1742497 \times 10^{-04}$	$2.9227836 \times 10^{-03}$	$1.9773837 \times 10^{-04}$
LOADBAL	$8.2584601 \times 10^{-09}$	$8.4342461 \times 10^{-09}$	$1.3314533 \times 10^{-09}$	$8.4325174 \times 10^{-09}$
SPANHYD	$3.1940033 \times 10^{-06}$	$3.1940033 \times 10^{-06}$	$1.0647653 \times 10^{-05}$	$3.1940033 \times 10^{-06}$
WATER	$4.9410920 \times 10^{-06}$	$5.0449384 \times 10^{-08}$	$5.0272795 \times 10^{-08}$	$5.0311122 \times 10^{-08}$

(a) *Table of relative difference values*



(b) *Graph of relative difference values*

Figure 14: *Relative difference between the DiSCOTech and known optimal objective values of CUTEr test problems using Active Set Local Search and Polar Search*

4.2.1 Local Search Algorithm: Sequential Search vs. Polar Search

Using a polar search provided a clear advantage over sequential search (Figures 12 and 14). In almost every combination of local search method and trial step the use of a polar search yielded a smaller relative difference. For both local search methods, using a polar search improved the relative difference value of a search with no trial steps. In some cases, the improvement was small (WATER using local search decreased from 2.4×10^{-2} to 4.7×10^{-3}) and in other cases it was very significant (EXPLIN using local search decreased from 7.7×10^{-1} to 1.4×10^{-4}).

Tests with trial steps showed similar improvements. The pattern step on DALLASS using local search and the active set step on WATER using local search yielded slightly higher relative difference values, but all other tests showed lower or equal relative difference values using polar search. The most striking are the DALLASM, EXPLIN, and LOADBAL test problems which gained several orders of magnitude relative improvement across almost all the trial steps for both local search methods. On particular test problems, the improvement in the relative difference value of each trial step was similar, however in some cases (most notably the active set step and pattern active set step of LOADBAL using local search) some trial steps showed much better improvement.

4.2.2 Local Search Method: Local Search vs. Active Set Local Search

Active set local search yielded mixed results (Figures 13 and 14). For sequential search tests with no trial steps, the active set local search showed smaller or equal relative differences on all test problems except SPANHYD. Some improvements were small (DALLASS decreased from 2.9×10^{-2} to 2.1×10^{-2} and HIMMELBI decreased from 7.0×10^{-2} to 5.6×10^{-2}) or equal (EXPLIN), but AVION2, LOADBAL, and WATER showed very significant improvement. The increase in the relative difference value for SPANHYD was small (6.8×10^{-6} to 8.4×10^{-6}). Polar search tests with no trial steps showed less improvement compared to the sequential search tests. This is most likely because most of the improvement compared to the sequential search was provided by the polar search, rather than the active set local search. The DALLASS, DALLASM, EXPLIN, and SPANHYD test problems all showed small increases in their relative difference values. For the sequential search tests only SPANHYD exhibited this behavior. However, similar to the sequential search tests, LOADBAL and WATER yielded significant improvement. A conclusion that can be drawn here is that when no heuristic local search algorithms are used (i.e. when using sequential search) active set local search is useful. However, active set local search is less (but still modestly) useful when used in combination with a polar search because the polar search provides the majority of the observed improvement.

Tests with trial steps, in general, yielded equal or lower relative differences values using the active set local search. As previously mentioned, the LOADBAL and WATER test problems yielded strong improvement. However within a test problem using the same local search method and algorithm options, trial steps were affected differently. For example, the active set-based trial steps on LOADBAL using sequential search

improved by an order of magnitude, while the pattern step relative difference value increased slightly (6.59×10^{-5} to 6.61×10^{-5}). Conversely, on the same test problem using polar search the pattern step significantly improved (2.6×10^{-5} to 1.3×10^{-9}) while the active set-based trial steps improved only very slightly. In both these cases the trial step or steps that improved had originally performed worse compared to the other trial steps. The pattern step of the DALLASS test problem improved (slightly using sequential search and significantly using polar search) while the active set-based trial steps worsened. Trial steps tended to perform worse if the associated test for that particular problem with no trial step also performed worse using the active set local search. Overall, active set local search provided strong improvement on some problems and slightly worse results for particular trial steps on other problems.

4.2.3 Trial Steps

In general, trial steps yielded improvement. The active set-based trial steps either provided a slight change in the relative difference values (BLEACHNG, DALLASS, DALLASM, EXPLIN, and SPANHYD) or a significant decrease (AVION2 using local search, LOADBAL using local search and polar search, and WATER) compared to tests with no trial steps. The active set-based trial steps typically yielded very similar relative difference values. The pattern step also typically either yielded a relative difference value similar to the value obtained with no trial step, or provided a significant improvement. In many cases the pattern step and active set-based steps yielded similar values. However, the pattern step often provided better results in the absence of other heuristic improvement techniques (Figure 11).

5 Conclusion

The DiSCOTech direct search nonlinear optimization package is a powerful tool for solving optimization problems where derivative information is unavailable. By implementing various heuristic improvement techniques DiSCOTech seeks to improve the performance of direct search methods. However, before DiSCOTech could be released for general use, the codebase needed to be streamlined, tested, and provided with a robust caching mechanism. By using object-oriented design principles DiSCOTech was restructured to more clearly separate functionality and facilitate feature expansion. This restructuring provided the opportunity to remove outdated code.

References in nonlinear optimization literature indicated that the use of the ϵ-metric with a binary search tree would be an effective strategy to leverage the comparison of finite-precision floating-point vectors with the performance of a binary search tree to construct a cache. However, as the investigation revealed, the lack of transitivity of the ϵ-metric led to serious and subtle errors in the operation of the binary search tree. A binary search tree without restructuring may be unable to locate ϵ-equal points. When tree rotations are taken into account the situation proves to be even worse. In such cases exact matches may not be correctly identified if the path to the point is modified by a tree rotation. To get around these issues, a set of

simpler list-based cache strategies were proposed. Testing revealed that a backward searching array-based cache using the ϵ-metric provided the best performance with a guarantee to always match ϵ-equal points.

Lastly, extensive testing of the heuristic improvement techniques provided in DiSCOTech revealed that the techniques do increase performance compared to a search using only the basic local search and sequential search options. On only a few combinations of test problem and features did an improvement technique not yield a decreased relative difference value and in those cases the increase of the relative difference value was minor. In all the other tests the heuristic improvement techniques provided a slight or significant decrease. However, conclusions about the interaction between different techniques are much less clear. Polar search seems to provide strong improvement regardless of what other techniques are applied. Active set local search and the trial steps provide mixed, but generally positive, results. The effects are most likely problem-dependent. Some problems experience significant improvement in certain trial steps when active set search is used, and others experience modest decreases in performance. One clear conclusion is that the active set step and pattern active set step provide very similar results. Since the pattern active set step requires more work to perform a search around the point generated by the active set step, it is advisable to use an active set step rather than a pattern active set step. Overall, the tests indicate that using the active set local search, polar search, and either the pattern or active set trial step will most likely provide the best performance improvement.

6 Cache Source Code

```
/*
 * MATLAB implementation of an n-dimensional real number vector-value
 * pair cache
 *
 * Michael Liarakos
 * College of William and Mary
 * August 25, 2008
 *
 * The cache is represented as one contiguous array of doubles.
 * Vector-value pairs are stored as the value followed by the
 * elements of the vector.
 *
 * New elements are inserted at the next empty space in the cache.
 * Searching is performed backwards from the most recently inserted
 * vector-value pair to the beginning of the cache.
 *
 * Vectors are compared using the epsilon-metric.
 *
 * Commands from within MATLAB:
 *
 * > cache('new', cache_size, vector_size)
 *   Initialize a new cache with the provided initial cache size and
 *   vector size clearing any previous data
 *
 * > cache('epsilon', epsilon_value)
 *   Set the value of epsilon used for vector comparisons
 *
 * > cache('insert', vector, value)
 *   Insert the provided vector-value pair into the cache
 *
 * > [vector_found, value] = cache('find', vector)
 *   Attempt to find the provided vector in the cache. If a matching
 *   vector is found then true and the value of that vector are
 *   returned. Otherwise false and 0 are returned.
 *
 * > cache('count')
 *   Return the number of vector-value pairs currently in the cache
 */

#include "mex.h"
#include <stdlib.h>
#include <string.h>

/* Comparison tolerance */
double epsilon;
/* Length of vectors stored in the cache */
int vectorLength;
/* Size of vector in bytes */
int vectorByteSize;
/* Length of vector-value pair */
int nodeSize;
```

```c
/* Index of the beginning of the next vector-value pair */
int currentIndex;
/* Number of items in the cache */
int count;
/* Current maximum size of the cache */
int cacheSize;
/* Pointer to the beginning of the cache */
double *cache;

double cacheValueToReturn;

/*
 * Free the cache from memory
 *
 */
void deleteCache()
{
  if(cache != NULL)
  {
    mxFree((void *)cache);
  }
}

/*
 * Destructor method called by MATLAB when clearing the cache
 * environment from memory
 */
static void destroyOnExit(void)
{
  deleteCache();
}

/*
 * Clear any existing cache data and reset cache variables
 *
 */
void resetCache(int newCacheSize, int newVectorLength)
{
  deleteCache();
  epsilon = 0.001;
  cacheSize = newCacheSize;
  count = 0;
  currentIndex = 0;
  vectorLength = newVectorLength;
  vectorByteSize = vectorLength * sizeof(double);
  nodeSize = vectorLength + 1;
  cache = mxMalloc(nodeSize * sizeof(double) * cacheSize);

  /* Terminate with error if malloc failed */
  if(cache == NULL)
  {
    mexErrMsgTxt("Error: Unable to allocate memory for cache");
  }
```

```
  /* Mark new node as persistent so it is not auto-removed by
     MATLAB */
  mexMakeMemoryPersistent((void *)cache);
}

/*
 * Insert the provided vector-value pair into the cache
 *
 */
void insert(mxArray *matlabVector, double value)
{
  /* Expand the cache if it is full */
  if(count == cacheSize)
  {
    cacheSize *= 3;
    cache = mxRealloc(cache, nodeSize * sizeof(double) * cacheSize);

    /* Terminate with error if malloc failed */
    if(cache == NULL)
    {
      mexErrMsgTxt("Error: Unable to allocate memory for cache");
    }

    /* Mark new node as persistent so it is not auto-removed by
       MATLAB */
    mexMakeMemoryPersistent((void *)cache);
  }

  /* Insert the vector-value pair */
  cache[currentIndex] = value;
  memcpy(&(cache[currentIndex + 1]), mxGetPr(matlabVector),
      vectorByteSize);

  count++;
  currentIndex += nodeSize;
}

/*
 * Search the cache for the provided vector
 *
 * The search uses the epsilon-metric and proceeds backwards from
 * the most recently inserted element to the beginning of the cache
 *
 * Returns true if the vector is found, false otherwise. If the
 * vector is found then 'cacheValueToReturn' is set to the value
 * of that vector. This temporary variable is used so that the
 * true/false statement and the value can be returned to MATLAB
 * as separate variables by the main MEX function
 */
int checkContains(mxArray *matlabVector)
{
  int i, j, equal;
  double *matlabPointer = mxGetPr(matlabVector);
```

```
  for(i = currentIndex - vectorLength; i > 0; i -= nodeSize)
  {
    equal = 1;

    /* Check if the target vector and the current comparison
       vector are epsilon-equal */
    for(j = 0; j < vectorLength; j++)
    {
      if((cache[i + j] - matlabPointer[j] > epsilon) ||
         (matlabPointer[j] - cache[i + j] > epsilon))
      {
        equal = 0;
        break;
      }
    }

    if(equal)
    {
      cacheValueToReturn = cache[i - 1];
      return 1;
    }
  }

  return 0;
}

/*
 * Compare a MATLAB string to a C-style string for equality
 *
 * Strings are limited to a maximum length of 20
 */
int matlabStringsAreEqual(const mxArray *matlabStr, char *str)
{
  char *inputStr[20];
  int result;

  mxGetString(matlabStr, inputStr, 20);

  if(inputStr == NULL)
  {
    return 0;
  }

  result = strncmp(str, inputStr, 20);

  if(result == 0)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
```

```
/*
 * Main method called by MATLAB
 *
 */
void mexFunction(int nlhs, mxArray *plhs[ ], int nrhs,
                 const mxArray *prhs[ ])
{
  mxArray *command;
  int index;

  /* Register destructor method */
  mexAtExit(destroyOnExit);

  /* Check arguments */
  if(nrhs < 1)
  {
    mexErrMsgTxt("Provide a command argument");
  }

  command = prhs[0];

  /* Check secondary arguments and execute commands */
  if(matlabStringsAreEqual(command, "new"))
  {
    /* Expecting cache_size and vector_size arguments */
    if(nrhs < 3)
    {
      mexErrMsgTxt("Too few input arguments");
    }

    resetCache((int)*mxGetPr(prhs[1]), (int)*mxGetPr(prhs[2]));
  }
  else if(matlabStringsAreEqual(command, "epsilon"))
  {
    /* Expecting epsilon_value argument */
    if(nrhs < 2)
    {
      mexErrMsgTxt("Too few input arguments");
    }

    epsilon = *mxGetPr(prhs[1]);
  }
  else if(matlabStringsAreEqual(command, "insert"))
  {
    /* Expecting vector and value arguments */
    if(nrhs < 3)
    {
      mexErrMsgTxt("Too few input arguments");
    }

    insert(prhs[1], *mxGetPr(prhs[2]));
  }
  else if(matlabStringsAreEqual(command, "find"))
```

```
  {
    /* Expecting vector argument */
    if(nrhs < 2)
    {
      mexErrMsgTxt("Too few input arguments");
    }

    if(checkContains(prhs[1]))
    {
      plhs[0] = mxCreateDoubleScalar(1);
      plhs[1] = mxCreateDoubleScalar(cacheValueToReturn);
    }
    else
    {
      plhs[0] = mxCreateDoubleScalar(0);
      plhs[1] = mxCreateDoubleScalar(0);
    }
  }
  else if(matlabStringsAreEqual(command, "count"))
  {
    plhs[0] = mxCreateDoubleScalar(count);
  }
  else
  {
    mexPrintf("Unknown command\n");
  }
}
```

References

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Doklady Akadamii Nauk SSSR*, pages 1259–1263, 1962.

[2] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indicies. *Acta Informatica*, 1:173–189, 1972.

[3] N. I. M. Gould, D. Orban, and P. L. Toint. Cuter (and sifdec), a constrained and unconstrained testing environment, revisited. *ACM trans. Math. Software*, 29:373–394, 2003. See also `http://cuter.rl.ac.uk/cuter-www`.

[4] G. A. Gray and T. G. Kolda. Algorithm 856: Appspack 4.0: Asynchronous parallerl pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software*, 32:485–507, 2006.

[5] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. *Proceedings of the 19th Annual IEEE Symposium on Foundations on Computer Science*, pages 8–21, 1978.

[6] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: new perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.

[7] R. M. Lewis, A. Shepherd, and V. Torczon. Implementing generating set search methods for linearly constrained minimization. *SIAM Journal on Scientific Computing*, 29:2507–2530, 2007.

[8] R. M. Lewis and V. Torczon. Active set identification without derivatives. Technical Report 07, College of William and Mary, September 2008.

[9] R. M. Lewis, V. Torczon, and M. W. Trosset. Why pattern search works. *Optima*, pages 1–7, 1998.

[10] The MathWorks. Mex-file guide. Available at `http://www.mathworks.com/support/tech-notes/1600/1605.html`.

[11] D. D. Sleator and R. E. Tarjan. Self-adjusting binary search trees. *JACM*, 32:652–686, 1985.