Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2004-058
AIM-2004-021

September 27, 2004

The Interval Programming Model for Multi-objective Decision Making

Michael R. Benjamin

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

The Interval Programming Model for Multi-objective Decision Making

Michael R. Benjamin
Computer Science and Artificial Intelligence Laboratory, 32-330
Department of Ocean Engineering, 5-221B
MIT, Cambridge MA 02139

NAVSEA Division Newport, Code 22
Newport RI 02841

September 27, 2004

Abstract

The interval programming model (IvP) is a mathematical programming model for representing and solving multi-objective optimization problems. The central characteristic of the model is the use of piecewise linearly defined objective functions and a solution method that searches through the combination space of pieces rather than through the actual decision space. The piecewise functions typically represent an approximation of some underlying function, but this concession is balanced on the positive side by relative freedom from function form assumptions as well as the assurance of global optimality. In this paper the model and solution algorithms are described, and the applicability of IvP to certain applications are discussed.[1]

1 Introduction

Interval programming (IvP) is a model and set of algorithms for representing and solving multi-objective optimization problems. The basic idea, as with other mathematical programming models, is to provide a specific set of mathematical constructs for representing a class of problems, and a set of algorithms that generate useful results by exploiting the nature of those constructs. The aim of this paper is to describe the constructs used in interval programming, along with the algorithms, and describe why this particular arrangement is uniquely useful for a certain class of applications. A common thread in the applications motivating the development of IvP is the need to fully automate a decision (as in a robot or autonomous vehicle), or provide a decision to a human in a critical window of time with little or no room for human interaction. Furthermore, the environment is typically composed of distinct components, or "sub-situations" that, in isolation, are fairly familiar and easy to handle. Their combination however typically creates a completely unique and unfamiliar overall situation where the decision-making challenge is in finding the right balance between the concerns of each sub-situation.

The central characteristic of the model is the use of piecewise linearly defined objective functions that may represent only an approximation of an underlying objective function. The solution method searches through the combination space of pieces, one from each function, rather than through the actual decision

[1]This work was sponsored by the Office of Naval Research (ONR), Code 333 Adam Nucci, and Code 311 Don Wagner, under contract N00014-04-WX-2-0683.

space. By performing search using the piecewise linear functions rather than the actual underlying function, the search algorithm is freed from function form assumptions, and global optimality (modulo the error between the two function representations) can be guaranteed. The problem of producing a piecewise linear function that adequately represents the underlying function is indeed a significant part of the overall solution process, and the notion of adequacy may be subjective. However, the piecewise approximation may be produced either off-line and outside the critical decision window, or it may be done in real-time by a module that knows about the particular kind of function it is tasked with approximating and uses this insight to create sufficiently accurate piecewise approximations sufficiently fast. The distinct separation of these two aspects of the solution process, the construction of the piecewise functions, and searching through the combination of pieces, is a central design decision that was made not only in the interest of balancing speed and accuracy tradeoffs, but also in the interest of providing a flexible model without function form assumptions to accommodate an eclectic collection of objective function producing modules.

1.1 Decision Automation and the Problem of State Space Growth

A common method for decision making automation is to develop a policy that associates a prescribed decision with each possible distinct situation or state (the terms are used interchangeably here). Automation then is primarily a matter of determining which situation applies at the moment. However, the overhead of this method grows exponentially as each new possible sub-situation (e.g. aspect of the environment) is accounted for in the policy mapping situations to actions. This problem is compounded in applications where the objectives associated with sub-situations also vary in their relative priority since the possible combinations of priority values distinguish additional unique overall situations.

The problems associated with an exponential growth of state space can be countered by dealing with separate sub-situations independently. Rather than form a policy based on all possible configurations in which sub-situations A and B may simultaneously come to be, a policy is separately formed for each. This approach is more modular and grows linearly in complexity with each new sub-situation. The separate policies, which typically disagree in their prescribed actions, ultimately need to be dealt with in coordination during the final decision making process. Effective coordination is difficult, if not impossible, if the policies do not indicate alternatives in addition to the single best decision associated with a situation. If a policy rates *all* alternative decisions, it is, in effect, providing an objective function, and the coordination of policies to find the best overall decision can be characterized as solving a multi-objective optimization problem. A primary motivation for developing the IvP model was to use it in this way to counter the problem of exponential state space growth.

1.2 Decision Automation and the Problem of State Space Uncertainty

In addition to using objective functions to coordinate policies between two or more situations that arise simultaneously, they can also be used to coordinate policies when there is uncertainty as to which situation aptly describes the world facing the decision maker. Loosely speaking, in coordinating two or more *simultaneous* situations the aim is to "kill two birds with one stone", and when coordinating two or more *uncertain situations*, the aim is to "hedge one's bet".

Situation uncertainty is not exclusively a matter of the decision maker's ability to sense the present state of the world without error. It also includes uncertainty as to how the world is changing during and shortly after the decision will be made. Any configuration of sub-situations, each with a measure of uncertainty, is in itself a situation, but the number of such potential combinations prohibits developing a policy a priori for each unique overall situation. Choosing an action that will effectively hedge bets between two potential situations also depends on rating alternative actions in addition to the single best action for each situation, i.e., providing objective functions. In some cases the alternatives from each show that it is simply not

possible to hedge bets, allowing the decision maker to reap the benefits of being decisive. Applying multi-objective optimization in this manner was another primary motivation for the development of IvP.

1.3 The Scope of This Paper

The primary focus of this paper is to clearly define what constitutes an interval programming problem, and provide an algorithm for solving them. Actually, five algorithms are provided in progression, with relative empirical results reported on "randomly" generated problems. The methods to generate these random problems are presented in Section A.1. Among the five algorithm versions, some are completely dominated by others, and some may be the best for only certain types of problems. By "dominated" we mean that there is a faster algorithm for virtually all types of conceivable problem instances. Some of the dominated algorithms are simpler to state however, and provided for illustrative purposes. This paper does not include a discussion of all currently available algorithm versions, primarily since further versions are tied closely to assumptions of problem type, which are tied to the specifics of an application. This paper also does not address the use of multi-objective decision making in behavior-based control systems although it is a primary motivation for the development of IvP. See [4], [10], [11], and [12] for more on this topic. For more on behavior based control see [1] and [6], and for examples of behavior-based control applications see [2], [4], [5], [13] and [14].

2 The Interval Programming Model

The interval programming model consists of a set of constructs for representing multi-objective optimization problems. The primary aim in deciding on these constructs is to strike the right balance between constructs that allow for exploitation by a set of optimization algorithms, and at the same time concede as little as possible in the ability to express problems within the applications of interest. The constructs and assumptions are set forth here, and the algorithms are addressed in the next section.

2.1 The Decision Space

The decision space is the set of all possible decisions that could result from a decision making problem. Typically there are a number of decision variables, each with their own set of possible values, and the decision space is the set formed by the Cartesian product of each of the variables. In the IvP model, the assumption is made that for each decision variable, the set of possible values is finite, uniformly discrete, with a known upper and lower bound. Thus the overall decision space is also finite and uniformly discrete.

The discrete assumption was considered acceptable since decision variables typically correspond to decisions in the real world that have limited precision in their execution. For example, if the decision variable corresponds to an actuator controlling the heading of robot that has a precision of 0.5 degrees, it makes little sense to produce a decision of 90.003 degrees. Furthermore, with a discrete domain, a globally optimal point can always be found in a finite number of steps. Brute-force search may be prohibitively expensive when searching through the overall decision space, but is useful at times in conjunction with other methods. Proving global optimality for a given search method can be reduced to proving how each discrete point is implicitly ruled out from contention. The uniform spacing assumption is purely a convenience that the algorithms may exploit in readily knowing the value of the next higher or lower point in the domain.

2.2 Interval Programming Functions

Objective functions used in interval programming problems are piecewise defined functions, but with a few strategic restrictions on representation: an IvP function is piecewise defined such that each point in the

decision space is contained in at most one piece, and each piece is given by a set of *boundary intervals*, one for each decision variable, and a linear *interior function* evaluating each point in the piece. If a point is in the decision space is not contained in any piece, it is considered infeasible, as discussed below in Section 2.4. Note that the uniformly spaced discrete decision space discussed above does not imply that the piecewise function has uniformly shaped or distributed pieces.

Piecewise functions have the ability to serve as a common representation scheme that can level the differences between modules that produce objective functions. Each may in fact serve as an approximation to some other underlying function, perhaps an analytical expression or a block of software capable of evaluating a point in the decision space. The time and process used to create sufficiently accurate piecewise functions is outside the scope here, but discussed in [3].

2.3 Interval Programming Problems

An interval programming problem consists of a collection of IvP functions, each with an associated priority weighting. Each function typically corresponds to an aspiration of the decision maker, or autonomous agent, and maps each point in the decision space to a value that reflects the degree to which that decision supports the corresponding aspiration. The priority weightings reflect the degree to which the decision maker is willing to trade off achievement in one aspiration for another based on the overall context at the moment.

For a problem defined over a decision space with n decision variables $(x_1,\ldots,x_n)$, and having k objective functions $f_1(x_1,\ldots,x_n),\ldots,f_k(x_1,\ldots,x_n)$, with k priority weights $(w_1,\ldots,w_k)$, the general form is given by:

maximize	$w_1 f_1(x_1,\ldots,x_n)+\ldots+w_k f_k(x_1,\ldots,x_n)$
such that	f_i is an IvP piecewise defined function,
	w_i is a positive number.

The k objective functions are effectively combined into a single objective function, which begs the question as to whether or not this constitutes "multi-objective" optimization. This term is applied here to discern a subclass of single-objective optimization problems where the single objective function to be optimized is composed of components that are themselves meaningful objective functions.

In non-interactive applications, where the user is not engaged in a cycle of what-ifs that alter the preference structure between objectives, there indeed must be only one collective objective function. For this reason, Pareto optimality (see [8], [7], or [9] for further on Pareto optimality), and the Pareto optimal set of solutions, does not play a prominent role in the search algorithms, only in that the solution algorithms provide a globally optimal solution that is indeed Pareto optimal.

2.4 The Feasible Space

Generally speaking, the *feasible space* is a subset of the decision space carved out by a set of explicit constraint constructs. No such constructs are utilized in interval programming problems. However, constraints can be effectively embedded within an objective function by leaving points in the decision space unmapped, i.e., not contained in any piece, for at least one objective function. This effect is a by-product of the search process described in the next section.

3 Solving Interval Programming Problems

3.1 Overview

The *solution space* for a particular problem is none other than the decision space itself. However, since each point in the feasible decision space is contained in exactly one piece from each of the k objective functions, the solution space can also be thought of as the set of possible combinations of k pieces. The search algorithms in this section are based on this observation, using branch and bound to search over a tree formed from the set of all such possible combinations. Although this set can be larger than the actual decision space, effective pruning on the corresponding search tree can greatly reduce the number of nodes expanded, i.e., combinations considered. Furthermore, with correct pruning of the search tree, global optimality is preserved since all points in the feasible space are implicitly considered.

3.2 Piece Intersection and Piece Maximal Value

Two important operations need to be well defined for comparing two sets of k pieces in the search algorithms. One is the definition of the *intersection* of two pieces, and the other is the *maximal value* of a piece. The intersection of two pieces p_i and p_j, denoted $p_i \cap p_j$, is the piece p_k that has new boundary intervals given by the pairwise intersection of each of its intervals, and a new interior function equal to the sum of its two interior functions. The example in Figure 1 illustrates this for two pieces defined over the decision variables x and y.

4	$\le x \le$	10
5	$\le y \le$	40
$3x-8y+18$		

(a) one piece, p_i

8	$\le x \le$	20
1	$\le y \le$	25
$5x+6y-11$		

(b) another piece, p_j

8	$\le x \le$	10
5	$\le y \le$	25
$8x-2y+7$		

(c) the intersection, p_k

Figure 1: The intersection of two two-dimensional pieces with linear interior functions.

When one or more of the intervals from p_i and p_j do not overlap, p_k is said to be *empty*, and the intersection of an empty piece with any other piece is also an empty piece.

The maximal value of a piece, denoted $p_i \rightarrow$`maxval` is the greatest value of all points contained in the piece when evaluated by the interior function. For example, with a linear interior function, this is obtained readily, i.e., independent of the number of points in the piece, by examining the slope of each variable. For p_k above, x has a positive slope and y has a negative slope, so the function is evaluated at $x = 10$ and $y = 5$ to give a maximal value of 77. A set of k pieces can therefore be compared against another set by the maximal value of the piece representing the intersection of all pieces in the set. The maximal value of an empty piece is undefined.

3.3 The Search Tree

Each level of the search tree corresponds to a single objective function, and each node within the level corresponds to a single piece within the objective function. Figure 2 depicts the situation for a problem with $k = 3$ objective functions, each with m pieces. Each leaf node corresponds to a unique combination of k pieces. In what follows, the piece associated with a particular node is referred to as the `nodePiece`, the expression `p(i, j)` denotes the ith piece in the jth objective function, and `universe` refers to a special piece with boundary intervals spanning the entire decision space and an interior function of $f(x_1,\ldots,x_n) = 0$. Each node, in addition to its `nodePiece`, has a `nodeBox`, which is the intersection of its `nodePiece`

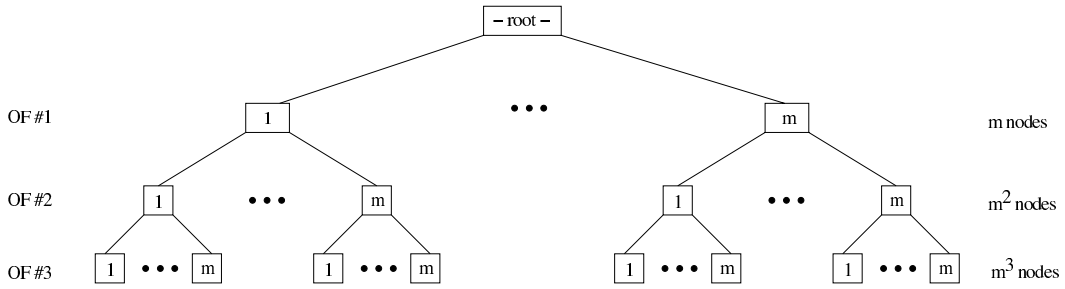


Figure 2: The search tree for $k = 3$ objective functions with m pieces each.

and its parent's `nodeBox`, with the root node having a `nodeBox` set to the `universe`. (For nodes on the first level, therefore, the `nodePiece` and `nodeBox` are equivalent in both boundary intervals and interior function.)

3.3.1 Full Tree Traversal

The algorithm in Figure 3 expands and traverses the entire search tree. The main function, `IPAL` (Interval Programming ALgorithm), sets up the root node and launches the tree traversal with the call to the recursive function `RIPAL`. The variable `bestBox`, initially null, is the best working solution, i.e., the intersection of the best combination of k pieces. The variable `bestVal` is the maximal value of `bestBox`. For simplicity in the pseudo-code, both are assumed to globally accessible rather than passing them as arguments to `RIPAL`. The value of `bestVal` cannot be meaningfully initialized, so when `bestBox` is null, the first non-empty leaf node will become the working solution, regardless of its maximal value.

```
IPAL()
0. bestBox = NULL
1. nodeBox = universe
2. RIPAL(nodeBox, 1)
3. return(bestBox)
```

```
RIPAL(nodeBox, level)
0. if(level == k)
1.    if(nodeBox is non-empty)
2.       if(!bestBox or nodeBox→maxval > bestVal)
3.           bestVal = nodeBox→maxval
4.           bestBox = nodeBox
5.    return
6. for(i = 1 to m)
7.    newNodeBox = nodeBox ∩ p(i, level)
8.    RIPAL(newNodeBox, level+1)
```

Figure 3: `IPAL` and `RIPAL` (version-1): Full search tree expansion.

The first part of `RIPAL` (lines 0-5) checks for the boundary, i.e., leaf node, condition (line 0) and potentially (lines 1-2) updates the best solution (lines 3-4). The second part (lines 6-8) iterates through the pieces associated with that level and builds a new node to branch on. This algorithm performs no pruning whatsoever and will expand all m^k leaf nodes, thus ensuring global optimality.

3.3.2 Simple Pruning of the Search Tree

The above algorithm will evolve over the coming sections, while preserving the guarantee of global optimality. In pruning the search tree, the time saved by pruning is balanced against the time needed to recognize a valid pruning opportunity. The first pruning strategy comes from slightly altering the first version of `RIPAL`

in Figure 3 so that the check for an empty `nodeBox` is done at internal nodes rather than exclusively at leaf nodes. The revised algorithm is given in Figure 4.

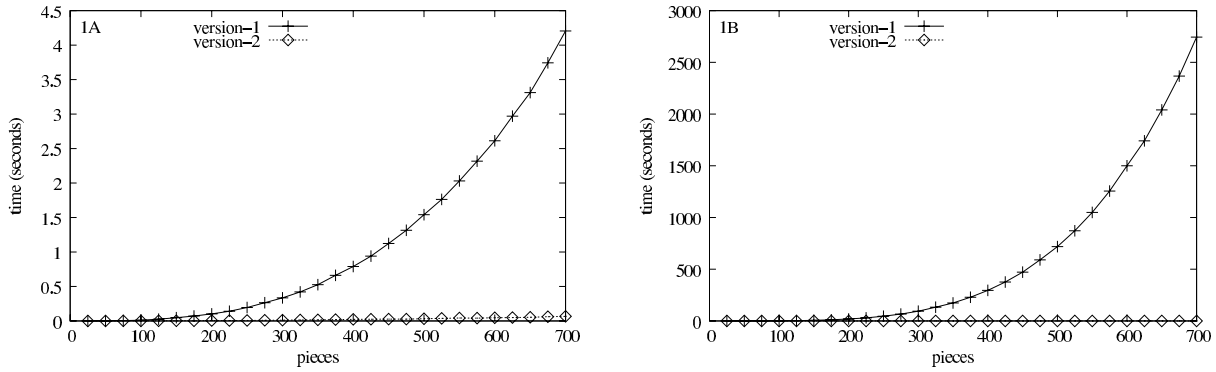
```
RIPAL(nodeBox, level)
0. if(level == k)
1.    if(!bestBox or nodeBox→maxval > bestVal
2.       bestVal = nodeBox→maxval
3.       bestBox = nodeBox
4.    return
5. for(i = 1 to m)
6.    newNodeBox = nodeBox ∩ p(i, level)
7.    if(newNodeBox is non-empty)
8.       RIPAL(newNodeBox, level+1)
```

Figure 4: `RIPAL` (version-2): Pruning nodes with an empty `nodeBox`.

The basic observation is that if a node has an empty `nodeBox`, the `nodeBox` for all its leaf children will also be empty. The check for non-emptiness on line 7 replaces the same check done on line 1 in the first version of `RIPAL` in Figure 3. In practice, this results in an enormous amount of pruning at very little cost. Not all pruning strategies are such clear winners.

3.3.3 Empirical Results

A comparison of `RIPAL`, between version-1 and version-2 is shown in Figure 5 below. Each data point represents the average solution time of 15 randomly generated problems in 2 dimensions. Note the difference in time scale between plots.



(a) IvP problems with 3 objective functions.

(b) IvP problems with 4 objective functions.

Figure 5: Comparison of `RIPAL`: version-1 vs. version-2.

Each problem tested was generated on a set of ramdomly parameterized *ring functions*, described in the appendix, where the motivation for this choice of experimentation is also discussed. The particulars of the two trials in Figure 5, are detailed further in Section A.2, along with the tabular data used to generate these plots, with high, low, and standard deviation results for each data set representing a point in a plot. This is also true for the empirical results reported in later sections.

Overall, two types of pruning are possible: a) pruning empty nodes as done in the above revised algorithm, and b) pruning non-empty nodes based on a calculated upper bound on solutions below a given

node. Pruning strategies discussed after this point utilize a grid structure introduced below in Section 3.4. Once this has been defined, further versions of `IPAL` and `RIPAL` are provided that are capable of further pruning.

3.4 IvP Grid Structures

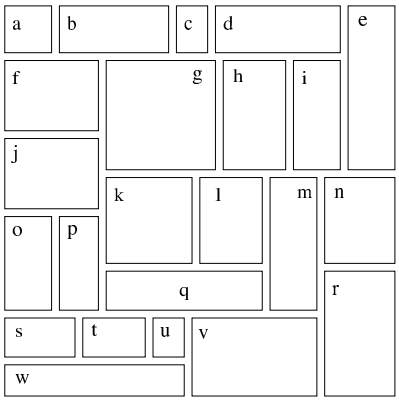
A grid structure instance is associated with each objective function in an IvP problem. It is utilized in `IPAL` to retrieve two types of information at each internal, i.e., non-leaf, node in the search tree. In considering whether to prune any of the m children of an internal node, a child node may be pruned either because its `nodePiece` does not intersect the `nodeBox` in question, or an upper bound on the its leaf nodes is inferior to the current working solution. The use of a grid structure, introduced in this section, is central to efficient implementation of both of the above types of pruning.

Although the grids are constructed and populated during the process of creating a piecewise defined approximation of an underlying function, which is outside the scope of this paper, grid configuration choices have a significant impact on solution time. For this reason two grid aspects of objective function creation are discussed below, grid initialization, and semi-uniform objective functions.

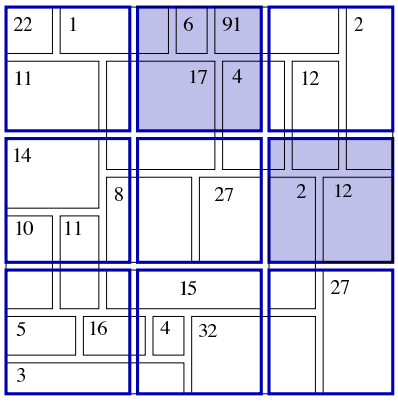
3.4.1 The Basic Grid Structure and Contained Information

Grids are composed of a multi-dimensional array of "grid elements", with two pieces of information associated with each element: `gridVal` and `gridList`. The latter is an initially empty linked list. When a piece from an objective function is added to a grid, the set of grid elements related to the piece is determined, and then the piece is added to the linked list of each such grid element. The value of `gridVal` indicates the maximum value across all pieces contained in the gridList. When a piece from an objective function is added to a grid element, the maximum value within the piece (always at a corner) is first determined, and this value is compared to the current `gridVal`, and updated if greater.

In the example in Figure 6(a), there are 23 pieces in an example objective function, and 9 grid elements in the grid shown in Figure 6(b) with a constant function interiors shown for each piece.



(a) Example 2D objective function with 23 pieces



(b) Grid containing the function pieces

Figure 6: Example grid with 9 grid elements and 23 rectilinear pieces.

The top shaded grid element in Figure 6(b), has the `gridList` containing pieces $\{b,c,d,g,h\}$, and the rightmost shaded grid element has the `gridList` $\{h,i,e,m,n\}$. Note that piece h belongs to both gridLists. The `gridVal` associated with the latter grid element would be 12, the maximum of $\{4,12,2,2,12\}$.

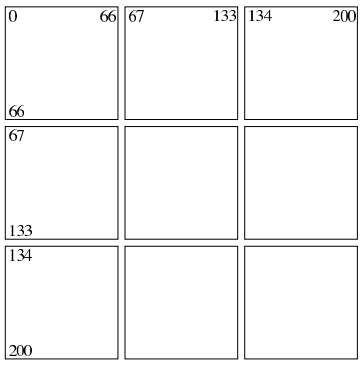
Each objective function may use a grid structure most suitable for its needs. The number of pieces and distribution of pieces can be quite different between objective functions, and therefore the number and shape of grid elements may be quite different as well to allow each grid structure to be efficient. The expression `grid[i]` refers to the grid associated with objective function *i*.

3.4.2 Initializing the Grid Element Layout

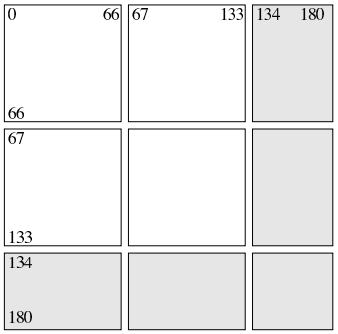
The initialization of the grid associated with objective function *j* takes two arguments as in:

$$\texttt{grid[j]} \rightarrow \texttt{initialize(universe, grid\_element\_box)}$$

The `universe` is given by the Cartesian product of each variable's domain. It is equivalent to the domain of the corresponding objective function, and is also identical between objective functions within an IvP problem. The second argument, `grid_element_box`, indicates the size of of each grid element. All grid elements within a grid will be the same size, provided that they divide evenly into the universe. For example, consider a 2D universe with both variable domains given by the interval $[0, 200]$, and a 2D `grid_element_box` with a length of 67 in both dimensions. The resulting grid is shown in Figure 7(a) with 9 identical grid elements.



(a) Uniform grid elements



(b) Non-uniform grid elements

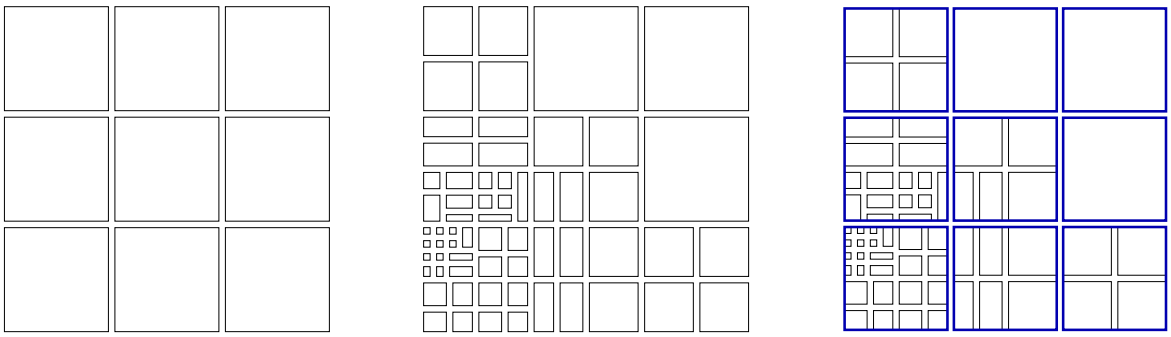
Figure 7: Two different universes result in different layouts with the same requested grid element size.

If the universe domains were instead $[0, 180]$, the higher indexed grid elements would be truncated, resulting in a non-uniform grid, as shown in Figure 7(b). The uniformity of all but possibly the highest indexed grid elements is a key property that allows for an efficient determination of which grid elements intersect a query box. In the example in Figure 7(b), the 9 grid elements could have been shaped in a more equitable way, which generally speaking, results in a more efficient grid. However, the initialization function respects the requested `grid_element_box` since it typically is coordinated with the shape of the objective function pieces it contains, as discussed next.

3.4.3 Coordinating a Grid with a Semi-uniform Objective Function

Semi-uniform IvP functions, are created by allocating an initial portion of the total number of pieces to represent the underlying function in a uniform piecewise manner. This idea is depicted in Figures 8(a) and 8(b). The piece distribution in Figure 8(a) is perfectly uniform, using only 9 pieces initially, with the remaining 53 pieces created by splitting the original 9 pieces and their descendents, as shown in Figure

8(b). The latter non-uniform pieces are allocated strategically to areas of the function domain that may need



(a) Uniform 2D IvP function (b) Semi-uniform IvP function (c) IvP function aligned with grid

Figure 8: Aligning the grid with the initial uniformity of an IvP function.

more pieces to adequately represent the underlying function. Grid efficiency is improved if the grid element layout is then aligned with the initial uniform pieces in the IvP function (Figure 8(c)).

The alignment ensures that no piece belongs to more than one grid element, and thus ensures the smallest possible `gridList` associated with each grid element. The alignment also improves the accuracy of the `gridVal` associated with each element, especially for pieces with linear interior functions. When a piece is added to the grid, the upper bound for the grid element is based on the maximal value of the linear interior function of the piece. If a portion of the piece lies outside the grid element, then the bound may not be accurate. This situation is eliminated when no piece lies in more than one grid element.

For all empirical results reported in this paper, objective functions are semi-uniform, and aligned with the grid (see Appendix A for more on the degree of semi-uniformity for specific trials). The use of a semi-uniform function allows the contributor of each objective function to retain significant latitude in deciding what configuration and amount of pieces is sufficient to represent its underlying function. In certain application domains, the source of each objective function can vary significantly, and this latitude may be important. If, however, it makes sense to further coordinate the objective functions within a problem to have identical grid element layouts and initial uniformity, the search time may be dramatically reduced since each piece is prone to intersect with less pieces from other objective functions, thus reducing the branching factor of the search tree.

3.5 Intersection Pruning Using IvP Grid Structures

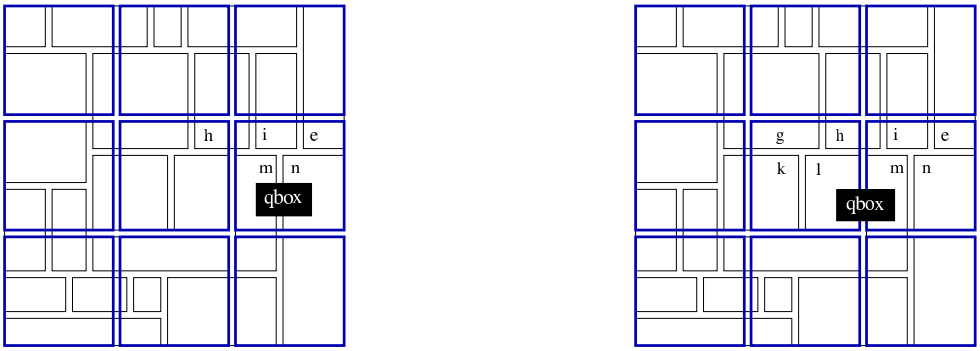
For a given internal node in the search tree, any of its *m* children may be pruned if the child's `nodePiece` does not intersect the `nodeBox`, as discussed in Section 3.3.2. Although it is possible to simply check each of the *m* children to see if they intersect the `nodeBox` (as in Figure 4), in practice, the great majority do not intersect. For a given internal node, a grid structure associated with the next level is used to return a list of pieces that are "close" to a given `nodeBox` and therefore "probably" intersect the `nodeBox`. A full intersection test is reserved only for pieces on this list, and any consideration whatsoever for the other "non-close" pieces is avoided.

3.5.1 Retrieving Intersection Information From a Grid Structure

A grid structure may be queried to determine which of the stored boxes intersect with a provided *query box*. The syntax for this query is:

$$\texttt{boxset} = \texttt{grid[i]} \rightarrow \texttt{getBoxes(qbox)}$$

where `grid[i]` is the grid associated with objective function i, the argument `qbox` is a query box, and `boxset` is the resulting set (linked list) of boxes intersecting the query box. In the example depicted in Figure 9(a), the query box intersects one grid element containing boxes $\{h,i,e,m,n\}$. Individual pairwise intersection tests are performed on these five, with the returned list being $\{m,n\}$.



(a) Query box intersects one grid element

(b) Query box intersects two grid elements

Figure 9: Retrieving intersection information from a grid given a query box.

In the second case, in Figure 9(b), the query box intersects two grid elements. The concatenation of lists from these two grid elements is $\{g,h,k,l,h,i,e,m,n\}$, which contains piece h twice. Thus duplicates must be removed from this list, and, in practice, this is done before the pairwise intersection tests are performed. In this simple case, the cost in removing duplicates is negligible, but in problems with more pieces, and higher dimensions (with usually coarser grids), the removal of duplicates may consume a significant amount of time. Thus grids that ensure that no piece lies in more than one grid element (discussed in Section 3.4.3) may have a noticeably more efficient implementation of this function by obviating the check for duplicates.

3.5.2 Search Tree Pruning Using Grid Intersection Information

The versions of `IPAL` and `RIPAL` given in Figure 10 use the grid associated with each of the k objective functions to avoid as many of the m intersection tests as possible at each node. The grid associated with each objective function is assumed to have been a priori constructed by the modules that produced the objective function. As before there are k objective functions, and m pieces in each function.

In this third version of `RIPAL`, lines 0-4 remain unchanged from Figure 4. Lines 7-10 now reflect that iteration through a linked list is conducted rather than the array of pieces of fixed length m as before. This linked list is the subset of the m pieces in the objective function at `level+1` determined, by a call to the grid in Line 5, to intersect the `nodeBox`. An empty returned `boxset`, or the end of the linked list is detected when `ibox=NULL`.

3.5.3 Empirical Results

A comparison of `RIPAL`, between version-2 and version-3 is shown in Figure 11 below. Each data point represents the average solution time of 15 randomly generated problems in 2 dimensions. This trial is similar to the previous trial shown in Figure 5 in that the x-axis shows a progression in the number of pieces in each

```
IPAL()
0. bestBox = NULL
1. nodeBox = universe
2. RIPAL(nodeBox, 1)
3. return(bestBox)
```

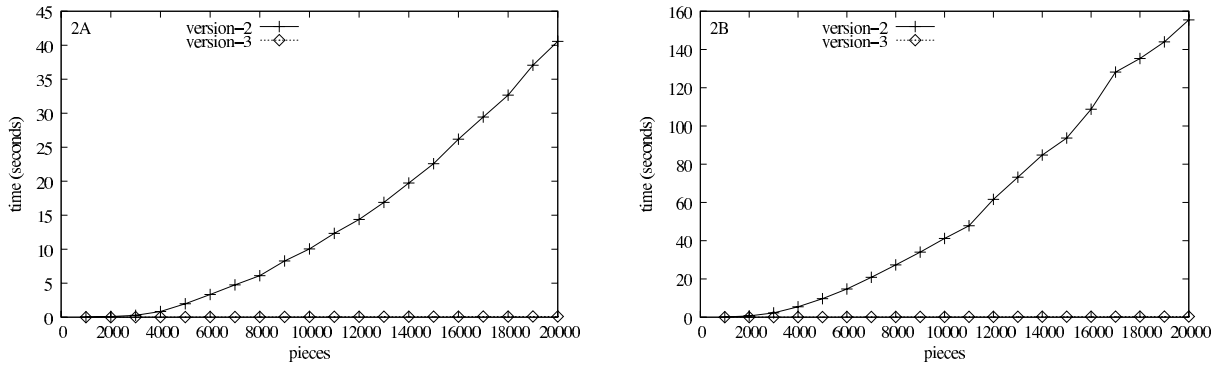
(a) IPAL (unchanged)

```
RIPAL(nodeBox, level)
0. if(level == k)
1.   if(!bestBox or nodeBox→maxval > bestVal
2.       bestVal = nodeBox→maxval
3.       bestBox = nodeBox
4.   return
5. boxset = grid[level+1]→getBoxes(nodeBox)
6. ibox = boxset→first
7. while(ibox ≠ NULL)
8.   newNodeBox = nodeBox ∩ ibox
9 .  RIPAL(newNodeBox, level+1)
10.  ibox = ibox→next
```

(b) RIPAL (version-3)

Figure 10: `IPAL` and `RIPAL` (version-3): Utilizing intersection information from grid structures.

objective function. Note that, as well as version-2 performed, relatively speaking, in the previous trial, it is completely dominated by version-3, which uses the grid structure to obviate many intersection tests. All solution times using version-3 and later algorithm versions using grids do not include the time taken to populate the grid data structures. Note the shape of the version-2 curve, and that, without the use of the grid to obviate intersection tests, m tests are needed at each node, and the algorithm has $\Omega(m^2)$ performance.



(a) IvP problems with 2 objective functions.

(b) IvP problems with 3 objective functions.

Figure 11: Comparison of `RIPAL`: version-2 vs. version-3.

The particulars of the two trials in Figure 11, are detailed further in Section A.3, along with the tabular data used to generate these plots, with high, low, and standard deviation results for each data set representing a point in a plot.

3.6 Upper Bound Pruning Using IvP Grid Structures

Grid structures are also used to quickly obtain an upper bound on solution for a particular node before any of its children are tested for intersection.

3.6.1 Retrieving Upper Bound Information From a Grid Structure

The second primary use of the grid structures is to find an upper bound associated with a particular search node. Recall the situation shown in Figure 2, where a particular node has m children. If an upper bound on all leaf children is shown to be poorer than a previously found solution, backtracking can be invoked immediately, precluding any further intersection tests and branching below the current node.

The tighter the bound, the more likely that pruning opportunities will be identified and taken advantage of. However, typically, the better the quality of the bound, the more expensive the cost of each node that is actually expanded. The bounding method presented here will be referred to with the following syntax:

$$\texttt{bound} = \texttt{grid[i]}\rightarrow\texttt{getBound(qbox, covered)},$$

where, as before, `qbox` is a query box, and `i` is an index indicating a particular objective function. The argument `covered` is a boolean return value set to be true if the query box intersects at least one grid element with a non-empty `gridList`, and false otherwise. The determination of the upper bound is given by the greatest `gridVal` of those grid elements intersecting the query box. In the example in Figure 12 below, the query box intersects the two shaded grid elements, which have a `gridVal` of 22 and 91 respectively, thus giving a upper bound of 91.

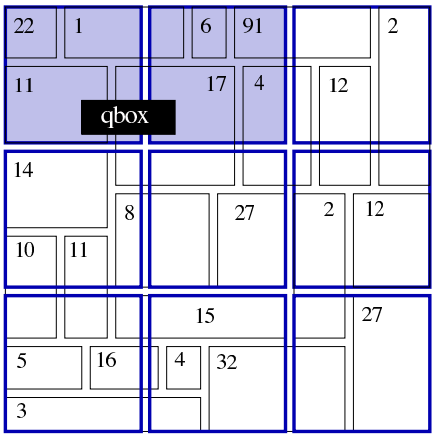


Figure 12: An example set of boxes from an objective function, with the `maxval` shown for each box.

This bound is extremely quick to obtain because the value associated with each grid element is set once, when the grid is built and populated. For a problem with k objective functions, an upper bound for a node at level i is the sum of the `maxVal` of the `nodeBox` and the upper bound on the portion of the tree below level i given by:

$$\sum_{j=i+1}^{k} \texttt{grid[j]}\rightarrow\texttt{getBound(qbox, covered)},$$

where the `qbox` is the `nodeBox` of the node in question. Note that the bound is progressively more likely to be tighter as the size of the query box shrinks. The `nodeBox` does indeed tend to shrink deeper in the tree, but this is tempered by the fact that pruning is also less rewarding deeper in the tree.

3.6.2 Search Tree Pruning Using Grid Upper Bound Information

To utilize the bounding information from the grid structures, version-4 of RIPAL is listed below in Figure 13. The primary difference is the addition of lines 9-15 between lines 8 and 9 of version-3 listed in Figure 10. In determining the upper bound, rather then simply iterating through all lower levels in lines 12-14, a

```
RIPAL(nodeBox, level)
0.  if(level == k)
1.    if(!bestBox or nodeBox→maxval > bestVal)
2.       bestVal = nodeBox→maxval
3.       bestBox = nodeBox
4.    return
5.  boxset = grid[level]→getBoxes(nodeBox)
6.  ibox = boxset→first
7.  while(ibox ≠ NULL)
8.    newNodeBox = nodeBox ∩ ibox
9.    bound = 0
10.   j = level + 1
11.   covered = true
12.   while((j ≤ k) and covered)
13.      bound = bound + grid[j]→getBound(newNodeBox, covered)
14.      j++
15.   if(covered and (newNodeBox→maxval + bound) > bestVal)
16.      RIPAL(newNodeBox, level+1)
17.   ibox = ibox→next
```

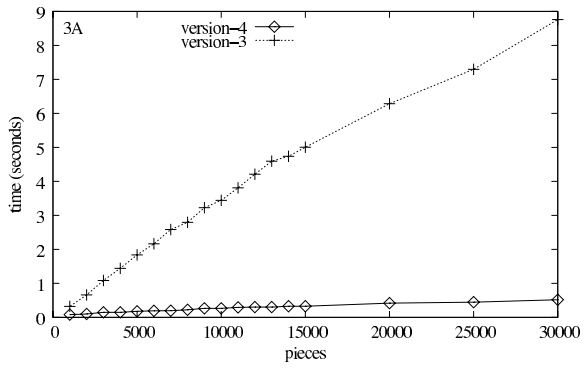
Figure 13: A version of RIPAL utilizing the bounding information from the grid of each objective function.

while loop is used to abort the bounding process if the newNodeBox is not covered by one of the objective functions.

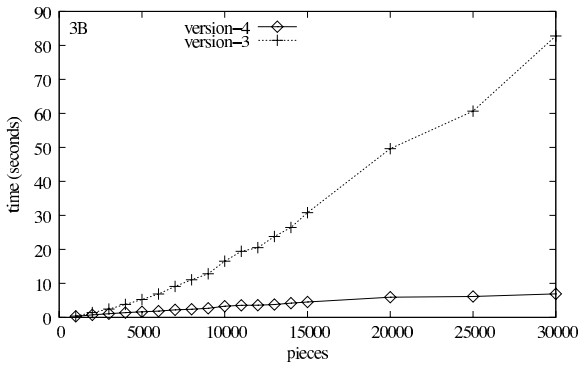
3.6.3 Empirical Results

A comparison of RIPAL, between version-3 and version-4 is shown in the three pairs of trials in Figures 14 thru 16 below. In the first trial pair, in Figure 14, each data point represents the average solution time of 60 randomly generated problems with 10 objective functions, first in (a) 2 dimensions, and then in (b) 4 dimensions. The x-axis shows a progression in the number of pieces in each of the objective functions, with a limit of 30,000 pieces versus the 20,000 limit in Figure 11. In this case, however, the search tree is significantly larger primarily due to the ten objective functions in each test problem. With 10,000 pieces, there are 10^{40} possible leaf nodes.

In this trial set, with 10 objective functions in each problem, the number of local optima is likely to be greater, and the solution time variance between similar problems was observed to be much higher. See the tables Appendix A.4 for variance data. For this reason, the test set size was chosen to be large enough to see a largely monotonic growth in time as the number of pieces increase. This trial pair shows the effectiveness of using the bounding technique described in the previous section. For problems with 30,000 pieces, version-4 is roughly 17 times faster in the 2D trials, and 12 times faster in the 4D trials. This ratio of relative effectiveness appears to grow as the number of pieces grow.



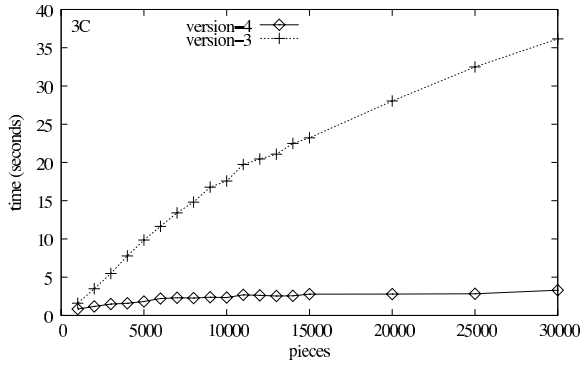
(a) IvP problems with 2 dimensions, 10 objective functions.



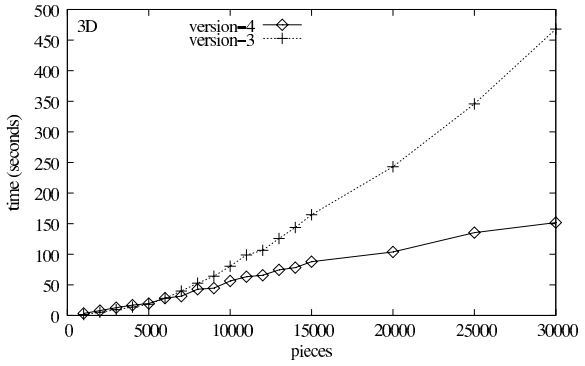
(b) IvP problems with 4 dimensions, 10 objective functions

Figure 14: Comparison of `RIPAL`: Version-3 versus Version-4.

In the second trial set, in Figure 15, the test files are nearly identical to the previous set, but have 20 objective functions each instead. As with the problems in the first trial set with 10 objective functions, the effectiveness of the bounding technique used in version-4 is apparent. However, the relative effectiveness is not as stark as with the problems with only 10 objective functions. For problems with 30,000 pieces, version-4 is roughly 13 times faster in the 2D trials, and 3 times faster in the 4D trials. In fact, for 4D problems with 6,000 pieces or less, version-3 is actually faster! This crossover is easier to discern in Table 7, in the appendix, corresponding to Figure 15(b).



(a) IvP problems with 2 dimensions, 20 objective functions.



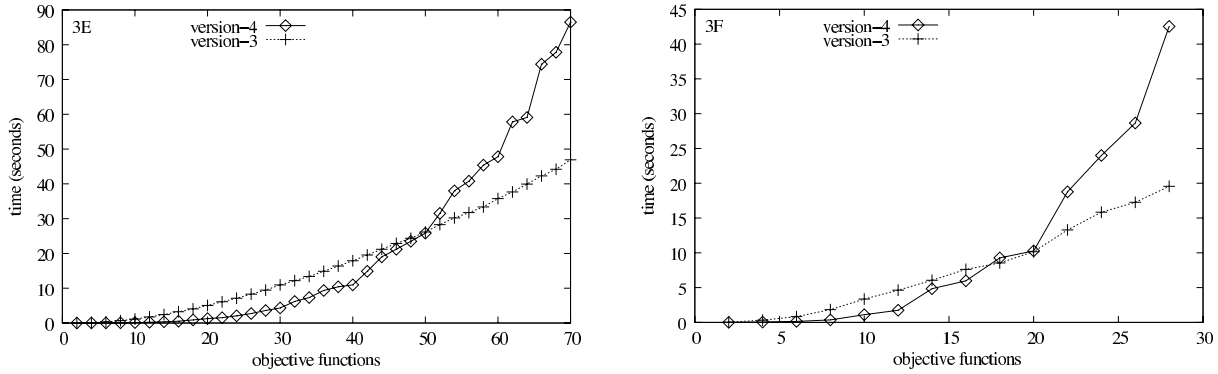
(b) IvP problems with 4 dimensions, 20 objective functions.

Figure 15: Comparison of `RIPAL`: Version-3 versus Version-4.

The diminished relative effectiveness between version-3 and version-4 between the two trial sets can be accounted for in the bounding algorithm. The accuracy of the bound calculated at each search node is not only diminished, but also more expensive to calculate as the number of objective functions grow. At some point, the overhead of attempting to bound the solution does not justify the benefits of the pruning resulting from the bounding.

To further demonstrate this relationship, the problems in the third pair of trials, in Figure 16, vary instead in the number of objective functions while holding the number of pieces constant at 4,000. In each trial pair there is a cross-over point where version-4, with bounding, no longer out-performs version-3. In the

2D trial, this roughly occurs in problems with 50 or more objective functions. In the 4D trial, this happens much sooner, at roughly 20 objective functions. (Note the data points in each plot at 20 objective functions. These points can also be seen in the trial pair in Figure 15, where the number of pieces equals 4,000.)



(a) IvP problems with 2 dimensions, 4000 pieces.

(b) IvP problems with 4 dimensions, 4000 pieces.

Figure 16: Comparison of `RIPAL`, version-3 vs. version-4 as the number of objective functions grow.

These results indicate the likely advantage of mixing the strategy of bounding, e.g., not bounding at all nodes as in version-4, and not forgoing all bounding as in version-3. Mixing strategies is outside the scope of this paper. The particulars of the three pairs of trials in Figures 14 thru 16, are detailed further in Section A.4, along with the tabular data used to generate these plots, with high, low, and standard deviation results for each data set representing a point in a plot.

3.7 Initial-Solution Utilization in Branch and Bound

An application of the branch and bound search method typically stands to benefit from the use of some heuristic that provides an initial solution that is likely to be pretty good for some reason or another. By measuring this solution prior to the traversal of the search tree, the hope is that the initial best-so-far solution is high enough that bounding and pruning will be effective earlier in the search tree. Otherwise, with no initial solution, it would suffer through an initial progression of weak and moderate solutions before the better solutions would allow the bounding methods to identify pruning opportunities higher up in the tree where pruning is most rewarding.

To accommodate an initial solution, the algorithm `IPAL` in Figure 3(a) and 10(a), is altered as follows:

```
IPAL(initialBestBox)
0. bestBox = initialBestBox
1. if(bestBox ≠ NULL)
2.     bestVal = bestBox→maxval
3. nodeBox = universe
4. RIPAL(nodeBox, 1)
5. return(bestBox)
```

Figure 17: A revised version of `IPAL` with an "initial solution" parameter.

The parameter `initialBestBox` could be provided by any one of possible heuristic algorithms, like the one provided below. If this parameter is `NULL`, the branch and bound search method simply proceeds as

normal, with the first best-so-far solution being the first leaf node representing a combination of intersecting pieces from the k objective functions.

3.7.1 An Initial-Solution Heuristic

For a problem with k objective functions, the initial-solution heuristic described here will evaluate k points in the decision space, where each point is the maximal point associated with one of the objective functions. To obtain the latter information, the grid population method described in Section 3.4.1 is augmented slightly by keeping a running indicator of the maximal point of any piece in the grid as it is populated. The maximal value of each piece is compared to this as each piece is added. The expression for obtaining this maximal point from the grid associated with objective function i is:

$$\texttt{box} = \texttt{grid[i]}\rightarrow\texttt{getMaxPoint}().$$

Given a maximal point for one of the k objective functions, the heuristic will need to evaluate this point from the perspective of the other objective functions. The expression for evaluating a point for an objective function i is:

$$\texttt{val} = \texttt{grid[i]}\rightarrow\texttt{evaluatePoint}(\texttt{box}, \texttt{covered}).$$

The argument, `covered`, is a boolean return value set to be true if the query box intersects at least one grid element with a non-empty `gridList`, and false otherwise. As mentioned in Section 2.4, if a point in the decision space is not contained within a piece of a particular objective function, this indicates that the point is infeasible. When a box is not covered, the numerical return value, `val`, is undefined.

The initial-solution heuristic, utilizing the above two expressions, is provided below in Figure 18. It is

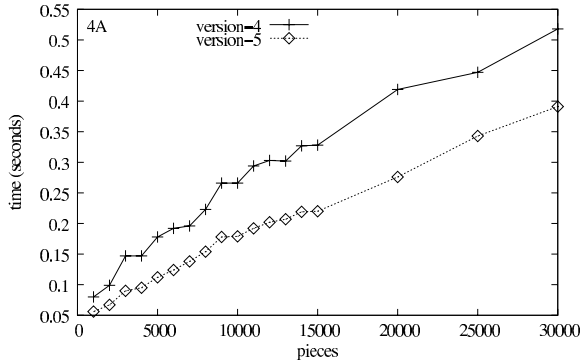
```
INITIAL_SOLUTION()
0.  bestPt = NULL
1.  for(i = 1 to k)
2.    boxPt = grid[i]→getMaxPoint()
3.    value = 0
4.    covered = true
5.    j = 1
6.    while((j ≤ k) and covered)
7.        value = value + grid[j]→evaluatePoint(boxPt, covered)
8.        j++
9.    if(covered and (!bestPt or (value > bestVal)))
10.       bestPt = boxPt
11.       bestPt→maxval = value
12. return(bestPt)
```

Figure 18: A heuristic for (usually) finding a good initial solution.

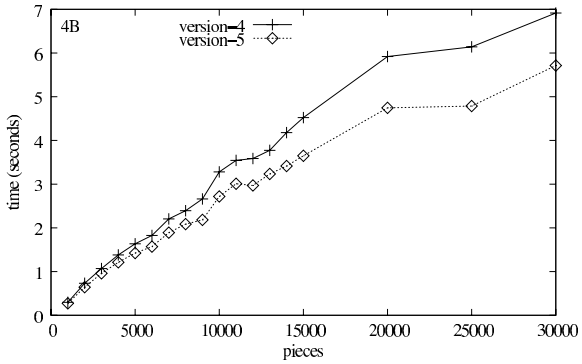
possible that this heuristic returns a null solution if all points individually optimal by one of the k objective functions are infeasible, i.e., uncovered, from the perspective of one of the other $k-1$ functions. For the purposes of testing the effectiveness of this heuristic in the following empirical tests, its use prior to the invocation of `IPAL(initialSolution)`, will be referred to as "version-5" of the search algorithm.

3.7.2 Empirical Results

A comparison of RIPAL, between version-4 and version-5 is shown in Figures 19 thru 21 below. In the first pair of trials shown in Figure 19, the test problems are the same used for comparing version-3 with version-4 reported previously in Figure 14. Each data point represents the average solution time of 60 randomly generated problems with 10 objective functions, first in (a) 2 dimensions, and then in (b) 4 dimensions, with the x-axis showing a progression in the number of pieces in each of the objective functions.



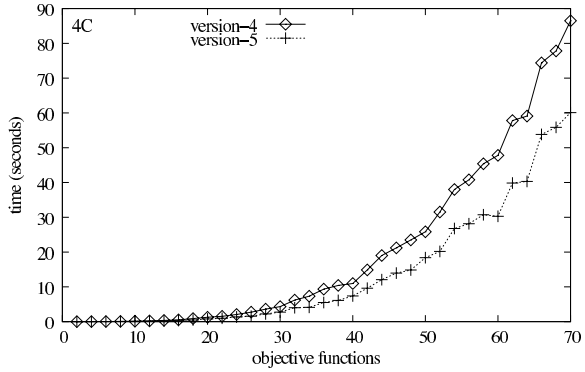
(a) IvP problems with 2 dimensions, 10 objective functions.



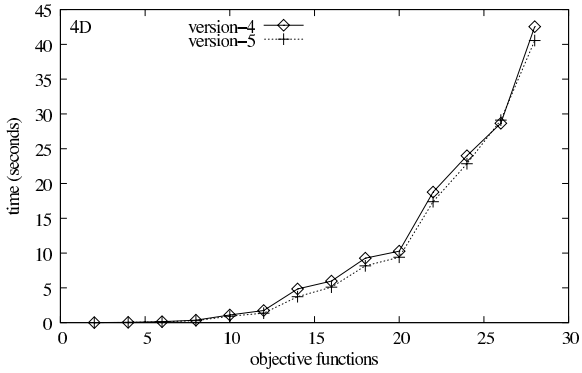
(b) IvP problems with 4 dimensions, 10 objective functions.

Figure 19: Comparison of RIPAL: Version 4 versus Version 5. Trials were run on the same problems reported in Figure 14.

In the second pair of trials shown in Figure 20, the test problems are the same used for comparing version-3 with version-4 reported previously in Figure 16. Each data point represents the average solution time of 80 randomly generated problems with 4000 pieces in 2 dimensions, and 18 randomly generated problems in 4 dimensions, with the x-axis showing a progression in the number of objective functions in each problem.



(a) IvP problems with 2 dimensions, 4000 pieces.

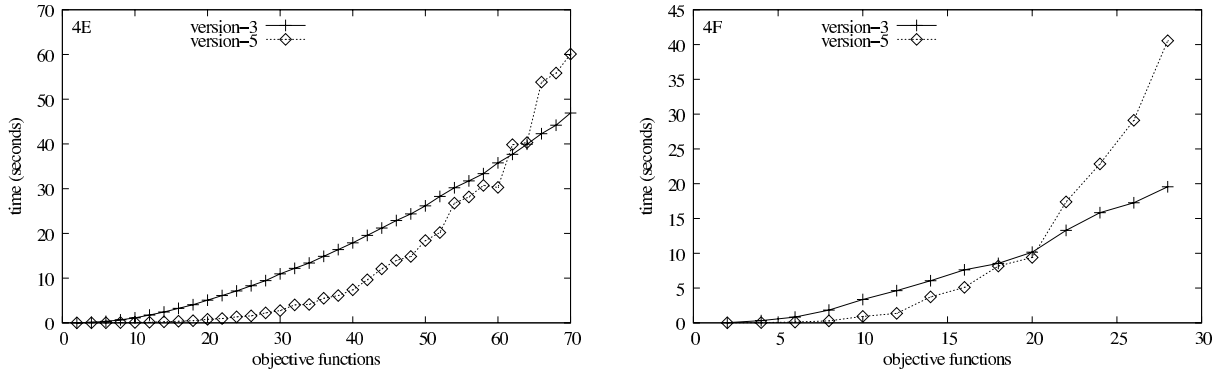


(b) IvP problems with 4 dimensions, 4000 pieces.

Figure 20: Comparison of RIPAL: version-4 versus version-5 as the number of objective functions grow. Trials were run on the same problems reported in Figure 16.

These results indicate that the use of this initial-solution heuristic is advantageous across a wide variety of problem types ranging in not only piece count, but also in the number of objective functions. In the last pair

of trials shown in Figure 21, version-5 is compared with version-3 instead, on the same problems reported in Figure 16. With the use of the initial-solution heuristic, the cross-over point where version-3 outperforms version-5 happens at roughly 60 objective functions, versus 50 previously. The effect in the 4D case is not as apparent.



(a) IvP problems with 2 dimensions, 4000 pieces.

(b) IvP problems with 4 dimensions, 4000 pieces.

Figure 21: Comparison of `RIPAL`: version-3 versus version-5 as the number of objective functions grow. Trials were run on the same problems reported in Figure 16.

The particulars of the three pairs of trials in Figures 19 thru 21, are detailed further in Section A.5, along with the tabular data used to generate these plots, with high, low, and standard deviation results for each data set representing a point in a plot.

3.8 The Effect of Dimension Size on the Solution Algorithms

The versions of `IPAL` and `RIPAL` presented, up through version-5, make no reference to *dimension*, i.e., the number of decision variables over which the objective functions are defined. (True not only in pseudo-code but the also in the C++ code used to generate empirical results.) The issue of problem dimension is embedded in the implementation of the two key operations, intersection and maximal value, described in Section 3.2. Grids have also been generalized to handle objective functions with arbitrary dimension.

Although the algorithms make no explicit mention of *dimension*, trials 3F and 4F in Figures 16 and 21, show that the 4D problems take longer to solve than their 2D counterparts. This is due to more than one issue. The simplest issue is that the core operations of intersection and finding the maximal value of a piece contain a number of steps that grows linearly in the number of variables. Broadly speaking, there are at least four influences on overall run-time of the search algorithm:

- *The number of nodes*: The number of nodes with a non-empty `nodeBox`. See Section 3.3. This is primarily determined by the number of objective functions, and the number of pieces in each function, but also the average number of pieces from one objective function that intersect a given piece from another. In functions defined over a larger domain, determined in part by dimension, the number of pieces needed to accurately represent the underlying function may increase.
- *The cost of the intersection and maxval operations*: The basic tree traversal operations. See Section 3.2. The number of steps in each grows linearly in the number of dimensions.

- *The cost of executing the grid functions*: The primary grid functions are `getBoxes` and `getBound` described in Sections 3.5.1 and 3.6.1 respectively. The number of steps in each grows linearly in the number of dimensions, for grids with identical numbers of grid elements.

- *The accuracy of the grid function results*: In functions defined over a larger domain, determined in part by dimension, each grid will will be defined over a greater expanse of the domain. For the function, `getBound`, this means that the upper bound will likely deviate more from, i.e., be less representative of, pieces not near the piece contributing to the upper bound. For the function `getBoxes`, this means that the query box may be tested for intersection with a greater number of pieces that don't actually intersect.

For these reasons, the empirical results, as dimensions vary, are not as conslusive. A full study of this issue is outside the scope of this paper.

4 Summary, Ongoing Work

The interval programming model is a mathematical programming model for representing and solving multi-objective optimization problems. The central characteristic of the model is the use of piecewise linearly defined objective functions that may represent only an approximation of an underlying objective function. The solution method searches through the combination space of pieces, one from each function, rather than through the actual decision space. By performing search using the piecewise linear functions rather than the actual underlying function, the search algorithm is freed from function form assumptions, and global optimality (modulo the error between the two function representations) can be guaranteed. In this paper, the model was defined, and a progression of solution algorithms was provided. Two of the algorithms outperform all the others on all test problems, and outperform each other on certain test problems.

The problem of producing a piecewise linear function that adequately represents the underlying function is indeed a significant part of the overall solution process, and the notion of adequacy may be subjective. However, the piecewise approximation may be produced either off-line and outside the critical decision window, or it may be done in real-time by a module that knows about the particular kind of function it is tasked with approximating and uses this insight to create sufficiently accurate piecewise approximations sufficiently fast. The issues of how such functions are created in particular applications was outside the scope of this paper.

The distinct separation of these two aspects of the solution process, the construction of the piecewise functions, and searching through the combination of pieces, is a central design decision that was made not only in the interest of balancing speed and accuracy tradeoffs, but also in the interest of providing a flexible model without function form assumptions to accommodate an eclectic collection of objective function producing modules. Work is ongoing to further the capabilities of the algorithms to solve larger and more widely varied classes of problems. Work is also ongoing that focusses on specific subclasses of problems with features that can be exploited. Some of this ongoing work is described below.

4.1 Ongoing Work

4.1.1 General Algorithm Development

- *Mixed bounding methods*: In this paper, the bounding method used at each node is the same, regardless of search history.

- *Epsilon backoff*: Rather than searching with a guarantee of global optimality, search can be conducted with a criteria of being within some epsilon percentage of global optimality.

- *Bounding with dynamic programming*: In some problem types, particularly with a large amount of objective functions, it may be advantages to pre-calculate the upper bound associated with each grid element with respect to the layers below it in the tree. This can be calculated once, prior to the overall search, as an alternative bounding method that is looser, but much quicker to retrieve.
- *Linear grid bounding*: The grid upper bound method described in Section 3.6.1, associates a scalar upper bound with each grid element. By using a linear function to represent this bound, the accuracy may sufficiently improve to warrant the extra overhead of processing the more complex bound.
- *Using priority queues within search levels*: The node expansion within a level of the search proceeds in an arbitrary order in the algorithms reported in this paper. It may be advantageous to not only calculate the upper bound for each node, but do so completely for all nodes in a layer, and using a priority queue to expand the nodes in order of the most promising first.
- *Initial-solution heuristics*

4.1.2 Extreme-Case Algorithm Development

Some types of problems contain characteristics that can be exploited by variations of the general algorithm particularly suited for special cases.

- *Unbalanced priority problems*: When the objective function priorities vary significantly, the initial-solution heuristics and ordering of the layers can affect the solution time.
- *Problems with fully-aligned grids*: By fully aligning the grids across all objective functions, a form of dynamic programming can be used effectively in problems with a large number of objective functions.
- *IvP Problems with weak decision variable overlap*: In all problems tested in this paper, all objective functions were defined over all variables. Ongoing work is exploring the algorithm behavior in problems where this is not the case to discover ways to exploit this variation.

4.1.3 Generalization of the IvP Model

The motivation for generalizing the model is to allow a wider class of applications to be represented with objective functions using fewer pieces to reduce generation and solution times.

- *Explicit linear constraints in the IvP model*: Constraints are currently implemented by leaving portions of the decision space uncovered within an IvP function. This can require a large number of pieces to accurately represent a constraint that may be expressed with a single linear inequality. A `nodeBox` in the branch-and-bound algorithm is augmented with one or more linear constraints. Linear programming is then used to determine an upper bound for the miniature integer programming problem represented at each node.
- *Piecewise quadratic functions*: Using quadratic interior functions within each piece, rather than strictly linear functions, would allow less pieces to be used to accurately represent underlying functions.
- *IvP problems with non-rectilinear piecewise functions*: Non-rectilinear pieces are formed by allowing intervals over functions of the decision variables. The motivation is to allow for a better representation of the underlying function using fewer pieces.

4.1.4 Generation of piecewise functions

- *Creating IvP piecewise defined functions*: The primary obligation of the IvP model is to provide fast, accurate solutions to problems that are as large and varied as possible. To some extent, it is the user's responsibility to come to the table with a problem cast in the required format. That being said, creating a piecewise defined representation of an underlying function is not trivial, and we have developed algorithms for doing so with no claim as to whether it could be done more accurately or quickly by some other approach. We bundle this functionality and code for the IvP user to use as they see fit, and we continue to incrementally advance this capability.

- *Creating bounded-epsilon piecewise defined functions*: In certain applications, it is preferable, for a given epsilon error bound, to have a guarantee that the error between an underlying function, and a piecewise representation of that function, is within that bound.

References

[1] Ronald C. Arkin. Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior. In *Proceedings of the IEEE Conference on Robotics and Automation*, pages 264–271, Raleigh, NC, 1987.

[2] Tucker Balch and Ronald C. Arkin. Behavior-Based Formation Control for Multiagent Robot Teams. *IEEE Transactions on Robotics and Automation*, December 1998.

[3] Michael R. Benjamin. *Interval Programming: A Multi-Objective Optimization Model for Autonomous Vehicle Control*. PhD thesis, Brown University, Providence, RI, May 2002.

[4] Michael R. Benjamin and Joe Curcio. COLREGS-Based Navigation in Unmanned Marine Vehicles. In *AUV-2004*, Sebasco Harbor, Maine, June 2004.

[5] Andrew A. Bennet and John J. Leonard. A Behavior-Based Approach to Adaptive Feature Detection and Following with Autonomous Underwater Vehicles. *IEEE Journal of Oceanic Engineering*, 25(2):213–226, April 2000.

[6] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation*, RA-2(1):14–23, April 1986.

[7] Ralph L. Keeney and Howard Raiffa. *Decisions with Multiple Objectives: Preferences and Value Tradeoffs*. Cambridge University Press, New York, NY, 1993.

[8] Kaisa M. Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer Academic Publishers, Boston, MA, 1999.

[9] Vilfredo Pareto. Cours d'Economie Politique. Libraire Droz, Genève (the first edition in 1896), 1964.

[10] Paolo Pirjanian. *Multiple Objective Action Selection and Behavior Fusion*. PhD thesis, Aalborg University, 1998.

[11] Jukka Riekki. *Reactive Task Execution of a Mobile Robot*. PhD thesis, Oulu University, 1999.

[12] Julio K. Rosenblatt. *DAMN: A Distributed Architecture for Mobile Navigation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, 1997.

[13] Alessandro Saffiotti, Enrique H. Ruspini, and Kurt Konolige. Using Fuzzy Logic for Mobile Robot Control. In H. J. Zimmerman, editor, *Practical Applications of Fuzzy Technologies*, chapter 5, pages 185–206. Kluwer Academic Publishers, 1999.

[14] Manuela Veloso, Elly Winner, Scott Lenser, James Bruce, and Tucker Balch. Vision-servoed Localization and Behavior-Based Planning for an Autonomous Quadruped Legged Robot. In *Proceedings of AIPS-2000*, Breckenridge, April 2000.

A Empirical Results

Ideally, it would be best to test IvP solution methods on a set of problems that are close in all aspects of difficulty to the problems expected to find in practice. This has two drawbacks. First is that while a method may work well for one application, it may be difficult to judge how it would scale to other domains with different problem characteristics. Second, it would require that the application be explained to some extent to justify why certain sets of objective functions are typical, which would significantly widen the scope of this paper.

Instead, the empirical results in this paper are based on a certain class of functions called *ring functions*, described next. This class is chosen because a) certain properties, such as non-linearity and non-convexity, will pose at least as great, or greater difficulties as those one might find in the "real" target applications, and b) they are easy to generate by using random number generators to set parameters, resulting in unique functions with significantly different properties in terms of the amount, degree, and location of local and global extrema, and c) one can crank up the parameters, such as dimension, number of objective functions, and pieces etc., to find trends in algorithm performance on types of problems not yet encountered in practice.

A.1 Ring Functions

A *ring function* is built by placing a $n-$dimensional ring somewhere in $n-$space, by choosing both its center and radius, and then associating values to all points in the domain based on the distance to the ring. Simple examples are depicted below in Figure 22(a) and (b). More than one ring can be used as shown on the right in Figure 22(c) and (d).

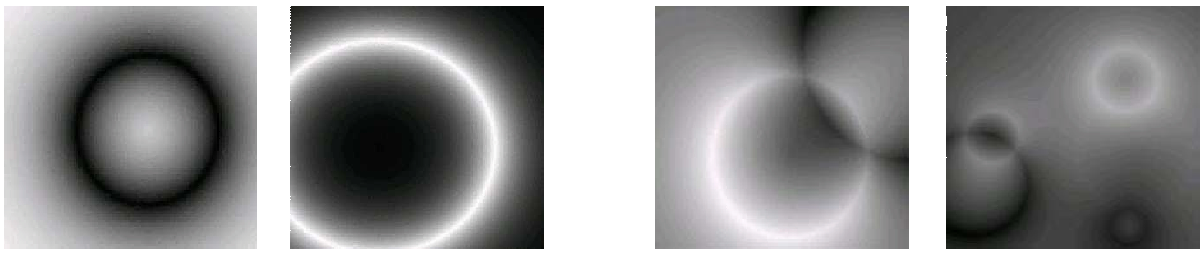


Figure 22: Example ring functions with one (a), (b), two (c) and five (d) rings.

With just one ring, this represents a non-linear, non-convex function. As more rings are added, the number of local optima also tends to rise. The following definition identifies the parameters of interest for generating sets of ring functions. A *ring function* is a function with the following form:

$$f(x,y) = f_1(x,y) + \ldots + f_p(x,y),$$

where each of the p functions $f_i(x,y)$ correspond to a particular ring given by:

$$f_i(x,y) = ((1 - \frac{|\sqrt{(x-h)^2+(y-k)^2} - \texttt{rad}|}{\texttt{max-dist}})^{\texttt{exp}} * \texttt{range}) + \texttt{base},$$

where each function $f_i(x,y)$ is equal to $g(r(x,y))$ given by:

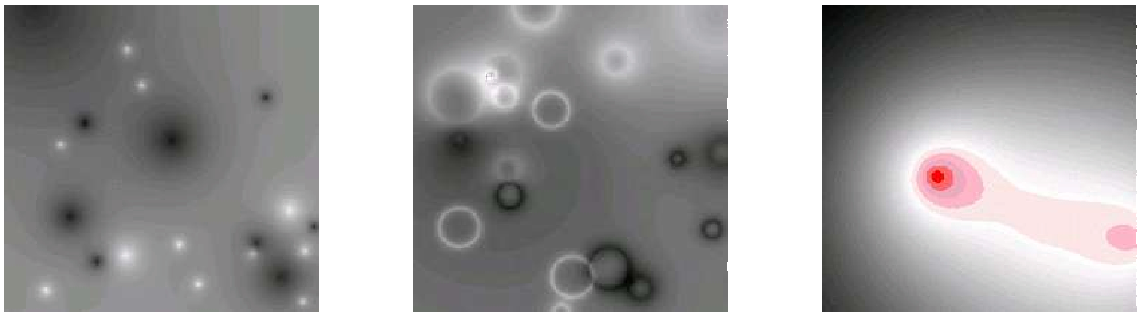
$$r(x,y) = |\sqrt{(x-h)^2+(y-k)^2} - \texttt{rad}|,$$

and

$$g(x) = ((1 - \frac{x}{\texttt{max-dist}})^{\texttt{exp}} * \texttt{range}) + \texttt{base}.$$

The function $r(x,y)$ indicates a circle (ring) with radius `rad`, and returns the shortest distance of a point, (x,y) to the ring. The function $g(x)$ takes this distance and produces the desired value based on the following intuition. The center of each ring is set to be somewhere in the universe given by the Cartesian product of each variable's domain. The value of `max-dist` is the maximum distance in this universe, i.e., the length from corner to opposite corner of the universe. The value of $\frac{r(x,y)}{\texttt{max-dist}}$ is therefore always in the range $[0,1]$. Subtracting $\frac{r(x,y)}{\texttt{max-dist}}$ from 1 and raising it to the exponent `exp` still leaves us with a value in the range $[0,1]$. Multiplying this by `range` and adding it to `base`, ensures that each function, $f_i(x,y)$, is guaranteed to range over [`base`, `base` + `range`], and thus $f(x,y)$ has the range $[p(\texttt{base}), p(\texttt{base} + \texttt{range})]$. The actual range of the function may be quite smaller and is unknown based solely on the parameters.

By controlling the parameters, p, `rad`, `exp`, `base`, and `range`, as well as the number of variables and their domains, we can get functions with rather predictable properties. For example, by using a large number of rings with a small radius and large `exp` factor, instances such as shown below in Figure 23(a) are typical. By then increasing the values to the radius parameter, instances such as that in Figure 23(b) are common.



(a) (b) (c)

Figure 23: Three types of ring functions based on varying parameter inputs.

And finally by lowering the ring parameter, p, and the `exp` parameter, instances like that in Figure 23(c) are typical. By first setting the above parameters, and then letting the ring positions vary randomly, significantly different looking functions can be generated within a class of functions with certain expected properties.

In all experiments reported in this paper, the domain of each variable ranged from 0 to 850, for a total of 851 possible values for each variable. Each ring function was randomly generated with the following values for the ring function parameters. The parameters `base` and `range` were set to -100 and 200 respectively. The value of `rad` was randomly chosen between 20 and 100, and the value of `exp` was randomly chosen between 1 and 20.

A.2 Trial 1: Comparing RIPAL version-1 vs. version-2

Trial 1 contains a pair of trials, 1A and 1B. The data in Tables 1 and 2 below correspond to the plots in Figures 5(a) and 5(b) respectively in Section 3.3.3. As with all trials reported in this paper, time is reported in seconds, and the superscript for each table entry contains 4 comma-separated values: the min-value, the max-value, the standard deviation, and the number of test problems contributing to that particular entry.

	Trial1A	
	ofs=3,dim=2	
	alg=version1	alg=version2
pieces	time	
25.0	0.000 (0.00,0.00,0.000,15)	0.000 (0.00,0.00,0.000,15)
50.0	0.001 (0.00,0.01,0.002,15)	0.000 (0.00,0.00,0.000,15)
75.0	0.005 (0.00,0.01,0.005,15)	0.002 (0.00,0.01,0.003,15)
100.0	0.011 (0.01,0.02,0.001,15)	0.001 (0.00,0.01,0.002,15)
125.0	0.026 (0.02,0.03,0.005,15)	0.003 (0.00,0.01,0.004,15)
150.0	0.047 (0.04,0.05,0.004,15)	0.003 (0.00,0.01,0.004,15)
175.0	0.071 (0.07,0.08,0.001,15)	0.003 (0.00,0.01,0.004,15)
200.0	0.102 (0.09,0.11,0.004,15)	0.006 (0.00,0.01,0.005,15)
225.0	0.141 (0.13,0.16,0.007,15)	0.006 (0.00,0.01,0.005,15)
250.0	0.195 (0.17,0.21,0.010,15)	0.007 (0.00,0.01,0.004,15)
275.0	0.264 (0.25,0.29,0.010,15)	0.010 (0.01,0.01,0.000,15)
300.0	0.335 (0.32,0.36,0.012,15)	0.012 (0.01,0.02,0.003,15)
325.0	0.421 (0.39,0.44,0.009,15)	0.014 (0.01,0.02,0.005,15)
350.0	0.526 (0.49,0.55,0.015,15)	0.016 (0.01,0.02,0.005,15)
375.0	0.661 (0.63,0.69,0.014,15)	0.020 (0.01,0.03,0.001,15)
400.0	0.790 (0.76,0.83,0.015,15)	0.023 (0.02,0.03,0.004,15)
425.0	0.940 (0.87,0.99,0.024,15)	0.025 (0.02,0.03,0.005,15)
450.0	1.123 (1.08,1.18,0.025,15)	0.027 (0.02,0.03,0.004,15)
475.0	1.315 (1.25,1.45,0.036,15)	0.027 (0.02,0.04,0.005,15)
500.0	1.541 (1.46,1.63,0.033,15)	0.032 (0.02,0.04,0.005,15)
525.0	1.763 (1.70,1.80,0.020,15)	0.037 (0.03,0.04,0.004,15)
550.0	2.031 (1.97,2.08,0.027,15)	0.043 (0.04,0.05,0.004,15)
575.0	2.317 (2.24,2.39,0.027,15)	0.041 (0.02,0.05,0.007,15)
600.0	2.613 (2.50,2.68,0.036,15)	0.046 (0.03,0.06,0.006,15)
625.0	2.970 (2.91,3.06,0.035,15)	0.049 (0.03,0.06,0.009,15)
650.0	3.312 (3.21,3.36,0.033,15)	0.053 (0.03,0.06,0.009,15)
675.0	3.743 (3.63,3.94,0.066,15)	0.061 (0.04,0.07,0.006,15)
700.0	4.205 (4.10,4.35,0.068,15)	0.067 (0.04,0.08,0.007,15)

Table 1: Two objective functions

	Trial1B	
	ofs=4,dim=2	
	alg=version1	alg=version2
pieces	time	
25.0	0.007 (0.00,0.01,0.004,15)	0.000 (0.00,0.00,0.000,15)
50.0	0.080 (0.07,0.09,0.003,15)	0.000 (0.00,0.00,0.000,15)
75.0	0.397 (0.38,0.42,0.011,15)	0.001 (0.00,0.01,0.002,15)
100.0	1.193 (1.16,1.23,0.014,15)	0.005 (0.00,0.01,0.005,15)
125.0	2.907 (2.87,2.95,0.022,15)	0.004 (0.00,0.01,0.005,15)
150.0	5.951 (5.87,6.03,0.040,15)	0.007 (0.00,0.01,0.004,15)
175.0	11.526 (10.9,13.0,0.738,15)	0.009 (0.00,0.02,0.004,15)
200.0	18.733 (18.6,18.9,0.049,15)	0.012 (0.01,0.02,0.003,15)
225.0	29.901 (29.7,30.0,0.086,15)	0.015 (0.01,0.02,0.005,15)
250.0	45.557 (45.3,46.2,0.120,15)	0.018 (0.01,0.03,0.004,15)
275.0	66.585 (66.3,66.9,0.147,15)	0.020 (0.01,0.03,0.003,15)
300.0	94.134 (93.8,94.4,0.153,15)	0.029 (0.02,0.04,0.005,15)
325.0	133.136 (129.0,142.5,4.932,15)	0.036 (0.02,0.04,0.005,15)
350.0	173.975 (173.2,174.9,0.392,15)	0.043 (0.03,0.05,0.006,15)
375.0	228.876 (227.8,230.4,0.535,15)	0.043 (0.02,0.06,0.008,15)
400.0	295.673 (294.8,296.8,0.433,15)	0.051 (0.04,0.06,0.008,15)
425.0	376.433 (374.9,378.3,0.810,15)	0.059 (0.04,0.08,0.008,15)
450.0	471.693 (470.2,472.9,0.643,15)	0.057 (0.03,0.08,0.011,15)
475.0	590.515 (583.5,627.5,8.745,13)	0.077 (0.06,0.09,0.010,15)
500.0	718.853 (717.8,719.9,0.842,4)	0.079 (0.05,0.10,0.012,15)
525.0	872.033 (868.9,877.9,2.924,4)	0.089 (0.05,0.11,0.014,15)
550.0	1048.995 (1048.2,1050.8,0.898,4)	0.097 (0.06,0.12,0.013,15)
575.0	1255.832 (1248.2,1269.5,6.844,4)	0.109 (0.07,0.14,0.015,15)
600.0	1500.445 (1477.0,1562.9,31.2,4)	0.111 (0.09,0.13,0.009,15)
625.0	1741.205 (1739.4,1743.1,1.620,4)	0.120 (0.07,0.15,0.021,15)
650.0	2040.062 (2036.7,2042.7,2.128,4)	0.134 (0.06,0.16,0.020,15)
675.0	2367.323 (2361.1,2371.0,3.106,4)	0.152 (0.09,0.17,0.015,15)
700.0	2743.937 (2735.9,2757.2,8.842,3)	0.155 (0.11,0.20,0.024,15)

Table 2: Three objective functions.

A.3 Trial 2: Comparing RIPAL version-2 vs. version-3

Trial 2 contains a pair of trials, 2A and 2B. The data in Tables 3 and 4 below correspond to the plots in Figures 11(a) and 11(b) respectively in Section 3.5.3. As with all trials reported in this paper, time is reported in seconds, and the superscript for each table entry contains 4 comma-separated values: the min-value, the max-value, the standard deviation, and the number of test problems contributing to that particular entry.

	Trial2A	
	ofs=3,dim=2	
	alg=version2	alg=version3
pieces	time	
1000.0	0.024 (0.02,0.03,0.005,15)	0.002 (0.00,0.01,0.003,15)
2000.0	0.101 (0.09,0.14,0.007,15)	0.004 (0.00,0.01,0.005,15)
3000.0	0.279 (0.22,0.37,0.055,15)	0.010 (0.00,0.02,0.001,15)
4000.0	0.838 (0.64,1.01,0.076,15)	0.016 (0.01,0.03,0.006,15)
5000.0	1.976 (1.64,2.13,0.095,15)	0.019 (0.01,0.02,0.001,15)
6000.0	3.334 (3.14,3.61,0.113,15)	0.026 (0.02,0.03,0.005,15)
7000.0	4.756 (4.59,5.01,0.101,15)	0.031 (0.02,0.04,0.004,15)
8000.0	6.106 (5.73,6.57,0.216,15)	0.032 (0.02,0.04,0.006,15)
9000.0	8.270 (8.10,8.39,0.064,15)	0.045 (0.02,0.06,0.009,15)
10000.0	10.045 (9.75,10.3,0.134,15)	0.051 (0.03,0.08,0.006,15)
11000.0	12.328 (12.1,12.5,0.093,15)	0.058 (0.03,0.10,0.014,15)
12000.0	14.379 (14.1,14.7,0.178,15)	0.061 (0.04,0.11,0.010,15)
13000.0	16.884 (16.4,17.1,0.147,15)	0.069 (0.04,0.10,0.012,15)
14000.0	19.740 (19.1,20.6,0.256,15)	0.074 (0.05,0.11,0.017,15)
15000.0	22.573 (21.8,23.0,0.215,15)	0.076 (0.06,0.09,0.009,15)
16000.0	26.184 (25.9,26.7,0.167,15)	0.084 (0.05,0.18,0.018,15)
17000.0	29.438 (28.5,30.3,0.349,15)	0.096 (0.06,0.14,0.015,15)
18000.0	32.658 (31.7,33.3,0.283,15)	0.090 (0.06,0.12,0.019,15)
19000.0	37.059 (35.8,38.0,0.463,15)	0.103 (0.05,0.15,0.016,15)
20000.0	40.569 (39.8,41.8,0.393,15)	0.107 (0.08,0.14,0.015,15)

Table 3: Two objective functions

	Trial2B	
	ofs=3,dim=2	
	alg=version2	alg=version3
pieces	time	
1000.0	0.117 (0.07,0.14,0.013,15)	0.011 (0.00,0.02,0.003,15)
2000.0	0.649 (0.51,0.82,0.062,15)	0.025 (0.02,0.04,0.006,15)
3000.0	2.251 (1.63,2.48,0.192,15)	0.035 (0.02,0.05,0.006,15)
4000.0	5.454 (4.58,6.07,0.435,15)	0.051 (0.03,0.07,0.008,15)
5000.0	9.687 (7.62,10.4,0.816,15)	0.069 (0.05,0.10,0.011,15)
6000.0	14.759 (8.65,16.2,1.252,15)	0.091 (0.06,0.11,0.010,15)
7000.0	20.846 (15.8,21.8,0.975,15)	0.115 (0.08,0.15,0.015,15)
8000.0	27.340 (20.6,29.6,2.515,15)	0.125 (0.09,0.17,0.016,15)
9000.0	34.031 (19.7,52.5,4.837,15)	0.144 (0.07,0.19,0.030,15)
10000.0	41.194 (25.5,45.3,4.150,15)	0.160 (0.11,0.22,0.023,15)
11000.0	47.823 (38.5,53.3,4.796,15)	0.189 (0.16,0.27,0.023,15)
12000.0	61.641 (46.8,63.5,2.183,15)	0.198 (0.17,0.25,0.018,15)
13000.0	73.213 (61.3,77.4,3.135,15)	0.214 (0.15,0.33,0.034,15)
14000.0	84.807 (78.1,89.4,2.506,15)	0.256 (0.21,0.33,0.034,15)
15000.0	93.675 (70.9,106.9,7.104,15)	0.244 (0.17,0.31,0.026,15)
16000.0	108.764 (68.9,126.1,14.5,15)	0.263 (0.13,0.42,0.050,15)
17000.0	128.201 (109.4,142.9,7.376,15)	0.295 (0.19,0.39,0.039,15)
18000.0	135.265 (76.5,149.6,11.7,15)	0.309 (0.22,0.41,0.039,15)
19000.0	143.978 (90.8,155.7,12.2,15)	0.327 (0.25,0.40,0.035,15)
20000.0	155.487 (98.2,175.4,18.6,15)	0.343 (0.24,0.51,0.072,15)

Table 4: Three objective functions.

A.4 Trial 3: Comparing RIPAL version-3 vs. version-4

Trial 3 contains three pairs of trials, 3A and 3B, 3C and 3D, 3E and 3F. The data in Tables 5 thru 10 below correspond to the plots in Figures 14(a) thru 16(b) respectively in Section 3.6.3. As with all trials reported in this paper, time is reported in seconds, and the superscript for each table entry contains 4 comma-separated values: the min-value, the max-value, the standard deviation, and the number of test problems contributing to that particular entry.

	Trial3A ofs=10,dim=2	
	alg=version4	alg=version3
pieces	time	
1000.0	0.080 (0.02,0.22,0.033,60)	0.323 (0.18,0.42,0.039,60)
2000.0	0.099 (0.02,0.28,0.040,60)	0.660 (0.46,0.84,0.066,60)
3000.0	0.147 (0.05,0.33,0.052,60)	1.085 (0.80,1.39,0.116,60)
4000.0	0.147 (0.04,0.35,0.057,60)	1.442 (0.92,1.99,0.169,60)
5000.0	0.178 (0.05,0.46,0.063,60)	1.838 (1.34,2.34,0.168,60)
6000.0	0.192 (0.06,0.40,0.068,60)	2.163 (1.51,2.79,0.208,60)
7000.0	0.196 (0.08,0.42,0.065,60)	2.583 (1.35,3.40,0.273,60)
8000.0	0.223 (0.09,0.54,0.074,60)	2.795 (1.99,3.45,0.286,60)
9000.0	0.266 (0.10,0.54,0.084,60)	3.226 (2.33,4.19,0.278,60)
10000.0	0.266 (0.12,0.56,0.093,60)	3.442 (2.56,4.33,0.368,60)
11000.0	0.294 (0.14,0.73,0.095,60)	3.810 (2.46,4.76,0.359,60)
12000.0	0.303 (0.13,0.76,0.092,60)	4.211 (2.87,5.11,0.406,60)
13000.0	0.302 (0.15,0.73,0.090,60)	4.593 (3.22,6.37,0.493,60)
14000.0	0.327 (0.15,0.69,0.098,60)	4.738 (3.66,6.30,0.484,60)
15000.0	0.328 (0.15,0.99,0.101,60)	5.005 (3.29,6.26,0.558,60)
20000.0	0.419 (0.20,1.00,0.131,60)	6.284 (4.52,8.00,0.676,60)
25000.0	0.447 (0.30,0.91,0.095,60)	7.297 (5.28,9.75,0.741,60)
30000.0	0.518 (0.31,1.18,0.131,60)	8.760 (6.44,11.7,1.023,60)

Table 5: Two dimensions, 10 objective functions.

	Trial3B ofs=10,dim=4	
	alg=version4	alg=version3
pieces	time	
1000.0	0.296 (0.06,0.81,0.131,100)	0.378 (0.14,0.63,0.089,100)
2000.0	0.733 (0.12,2.60,0.342,100)	1.401 (0.51,2.34,0.378,100)
3000.0	1.069 (0.09,4.17,0.585,100)	2.443 (0.69,8.02,0.699,100)
4000.0	1.380 (0.13,4.46,0.674,100)	3.802 (1.21,6.59,1.032,100)
5000.0	1.635 (0.24,5.38,0.758,100)	5.253 (2.00,12.9,1.422,100)
6000.0	1.827 (0.16,8.86,1.040,100)	6.879 (1.37,13.7,1.942,100)
7000.0	2.204 (0.25,9.11,1.232,100)	9.104 (1.31,17.8,2.763,100)
8000.0	2.393 (0.15,9.72,1.314,100)	11.049 (2.60,24.1,3.244,100)
9000.0	2.662 (0.38,9.87,1.515,100)	12.826 (1.86,24.5,4.203,100)
10000.0	3.280 (0.13,12.7,1.697,100)	16.515 (4.24,37.5,5.119,100)
11000.0	3.544 (0.21,15.6,2.063,100)	19.420 (6.02,38.4,6.127,100)
12000.0	3.586 (0.22,16.6,2.020,100)	20.515 (6.81,45.9,6.555,100)
13000.0	3.773 (0.37,16.6,2.525,100)	23.792 (4.93,49.3,8.515,100)
14000.0	4.179 (0.18,16.3,1.992,100)	26.441 (8.32,52.7,8.127,100)
15000.0	4.522 (0.47,18.0,2.536,100)	30.807 (8.78,64.1,10.6,100)
20000.0	5.920 (0.77,23.7,3.407,100)	49.613 (3.20,97.7,15.1,100)
25000.0	6.142 (0.37,27.8,3.507,100)	60.655 (9.23,127.4,19.0,100)
30000.0	6.916 (1.09,37.5,3.893,100)	82.764 (13.5,157.8,23.8,97)

Table 6: Four dimensions, 10 objective functions.

	Trial3C ofs=20,dim=2	
	alg=version4	alg=version3
pieces	time	
1000.0	0.831 (0.14,2.82,0.336,140)	1.600 (1.09,2.19,0.165,140)
2000.0	1.179 (0.17,4.51,0.544,140)	3.493 (2.01,4.75,0.378,140)
3000.0	1.488 (0.14,8.21,0.747,140)	5.484 (3.83,7.62,0.515,140)
4000.0	1.589 (0.30,5.15,0.699,140)	7.776 (4.66,9.94,0.766,140)
5000.0	1.817 (0.21,5.63,0.838,140)	9.859 (6.46,13.2,0.915,140)
6000.0	2.210 (0.36,6.84,1.025,140)	11.641 (8.43,15.0,1.070,140)
7000.0	2.313 (0.43,7.00,1.020,140)	13.413 (10.1,17.0,1.075,140)
8000.0	2.270 (0.42,8.94,1.035,140)	14.817 (10.3,18.2,1.148,140)
9000.0	2.379 (0.43,8.69,1.124,140)	16.764 (12.8,21.2,1.528,140)
10000.0	2.343 (0.46,7.04,0.966,140)	17.571 (13.2,27.0,1.676,140)
11000.0	2.692 (0.39,9.77,1.251,140)	19.737 (13.4,25.5,1.734,140)
12000.0	2.631 (0.41,8.39,1.177,140)	20.452 (16.0,31.6,1.822,140)
13000.0	2.537 (0.49,6.93,1.135,140)	21.080 (14.4,29.4,1.967,140)
14000.0	2.563 (0.48,9.47,1.214,140)	22.488 (15.7,39.3,1.931,140)
15000.0	2.786 (0.48,6.82,1.169,140)	23.218 (15.8,31.3,2.092,140)
20000.0	2.790 (0.59,10.4,1.226,140)	28.033 (19.7,36.8,2.290,140)
25000.0	2.832 (0.71,7.94,1.160,140)	32.493 (23.2,55.0,2.861,140)
30000.0	3.297 (0.92,10.8,1.453,140)	36.152 (25.1,54.9,3.319,140)

Table 7: Two dimensions, 20 objective functions.

	Trial3D ofs=20,dim=4	
	alg=version4	alg=version3
pieces	time	
1000.0	3.289 (0.55,8.15,0.996,110)	1.463 (0.56,2.10,0.219,110)
2000.0	7.906 (0.84,24.0,3.350,110)	5.246 (2.27,8.86,1.133,110)
3000.0	12.641 (1.99,34.1,5.473,110)	9.404 (3.68,15.8,2.044,110)
4000.0	16.786 (3.48,49.9,6.748,110)	13.618 (6.99,19.8,2.194,110)
5000.0	19.451 (2.27,64.9,7.868,110)	18.190 (8.23,29.5,3.119,110)
6000.0	28.270 (1.42,81.2,12.9,110)	28.002 (13.7,44.5,5.846,110)
7000.0	31.436 (2.33,99.7,14.7,110)	39.882 (20.1,65.6,7.895,110)
8000.0	42.826 (4.20,127.9,19.5,110)	52.673 (24.6,89.2,9.628,110)
9000.0	44.583 (5.18,151.6,25.0,110)	64.099 (24.1,104.0,14.3,110)
10000.0	56.288 (4.03,197.6,28.9,110)	80.422 (22.4,139.7,18.9,110)
11000.0	63.270 (1.92,177.1,28.9,110)	98.793 (34.2,169.6,21.9,110)
12000.0	65.703 (2.29,348.5,37.1,110)	106.408 (34.0,195.4,25.7,110)
13000.0	74.438 (1.67,212.5,38.7,110)	125.672 (51.6,210.7,31.4,110)
14000.0	78.170 (8.02,285.0,46.0,110)	143.729 (50.6,251.0,40.2,110)
15000.0	87.791 (5.66,328.0,50.0,110)	164.550 (53.8,297.7,37.9,110)
20000.0	103.770 (5.55,305.3,55.6,110)	243.027 (76.5,405.5,55.7,110)
25000.0	135.275 (7.32,623.1,80.6,110)	345.740 (147.8,742.1,78.2,110)
30000.0	151.734 (19.6,634.7,90.1,110)	467.933 (93.3,823.3,122.3,110)

Table 8: Four dimensions, 20 objective functions.

	Trial3E	
	dim=2,pieces=4000	
	alg=version4	alg=version3
ofs	time	
2.0	0.006 (0.00,0.01,0.005,80)	0.016 (0.01,0.02,0.005,80)
4.0	0.018 (0.01,0.03,0.004,80)	0.120 (0.06,0.17,0.012,80)
6.0	0.038 (0.02,0.07,0.010,80)	0.334 (0.15,0.47,0.038,80)
8.0	0.074 (0.04,0.17,0.020,80)	0.670 (0.47,0.89,0.069,80)
10.0	0.123 (0.05,0.30,0.042,80)	1.117 (0.81,1.50,0.132,80)
12.0	0.217 (0.05,0.79,0.089,80)	1.738 (1.21,2.13,0.169,80)
14.0	0.355 (0.08,1.19,0.140,80)	2.425 (1.73,3.00,0.190,80)
16.0	0.521 (0.09,1.26,0.211,80)	3.234 (2.19,3.91,0.313,80)
18.0	0.890 (0.19,2.74,0.417,80)	4.056 (2.74,5.12,0.275,80)
20.0	1.263 (0.17,3.85,0.571,80)	5.064 (3.65,6.16,0.421,80)
22.0	1.532 (0.34,3.34,0.575,80)	6.102 (4.42,7.78,0.481,80)
24.0	2.066 (0.60,7.56,0.846,80)	7.126 (5.68,8.59,0.543,80)
26.0	2.725 (0.15,6.53,1.200,80)	8.300 (6.14,9.54,0.573,80)
28.0	3.600 (0.48,9.57,1.586,80)	9.451 (7.23,11.0,0.712,80)
30.0	4.329 (0.72,11.0,2.155,80)	10.956 (8.64,13.4,0.666,80)
32.0	6.188 (1.32,18.9,2.489,80)	12.184 (9.68,14.0,0.848,80)
34.0	7.309 (0.27,24.6,3.078,80)	13.381 (10.3,15.6,0.956,80)
36.0	9.353 (1.44,24.7,4.202,80)	14.868 (11.5,17.3,0.956,80)
38.0	10.413 (2.68,23.5,4.184,80)	16.382 (13.5,19.7,1.020,80)
40.0	10.960 (1.71,36.8,5.311,80)	17.892 (15.0,20.1,1.017,80)
42.0	14.874 (2.52,41.6,6.994,80)	19.549 (16.5,23.5,0.941,80)
44.0	19.009 (2.57,66.6,8.273,80)	21.203 (16.4,24.4,1.342,80)
46.0	21.209 (3.93,55.3,8.874,80)	22.880 (20.1,25.6,1.183,80)
48.0	23.485 (2.83,65.0,10.6,80)	24.353 (19.4,27.6,1.398,80)
50.0	25.831 (1.65,74.3,11.8,80)	26.159 (21.2,30.6,1.416,80)
52.0	31.515 (4.21,82.1,11.8,80)	28.270 (23.4,31.9,1.465,80)
54.0	37.993 (5.22,124.6,17.3,80)	30.205 (24.2,34.7,1.698,80)
56.0	40.779 (5.31,85.1,13.9,80)	31.740 (27.2,36.3,1.388,80)
58.0	45.376 (18.3,118.8,16.8,80)	33.361 (26.4,41.3,1.837,80)
60.0	47.809 (10.4,132.6,21.2,80)	35.764 (29.3,41.7,1.817,80)
62.0	57.810 (11.5,144.7,23.8,80)	37.673 (31.6,44.9,1.833,80)
64.0	59.085 (7.87,131.1,20.8,80)	39.931 (34.0,46.8,2.189,80)
66.0	74.377 (10.3,172.6,28.6,80)	42.291 (36.9,48.2,2.148,80)
68.0	77.827 (6.19,259.9,35.1,80)	44.194 (37.9,50.4,2.169,80)
70.0	86.493 (22.3,243.9,37.2,80)	46.920 (39.0,55.8,2.383,80)

Table 9: Two dimensions, 4000 pieces.

	Trial3F	
	dim=4,pieces=4000	
	alg=version4	alg=version3
ofs	time	
2.0	0.005 (0.00,0.01,0.005,18)	0.029 (0.01,0.04,0.010,18)
4.0	0.042 (0.02,0.12,0.019,18)	0.322 (0.14,0.55,0.086,18)
6.0	0.139 (0.04,0.37,0.058,18)	0.831 (0.16,1.41,0.294,18)
8.0	0.353 (0.09,0.91,0.174,18)	1.835 (0.85,3.03,0.566,18)
10.0	1.108 (0.16,2.08,0.439,18)	3.357 (1.99,4.74,0.648,18)
12.0	1.744 (0.22,4.16,0.725,18)	4.626 (3.04,6.47,0.935,18)
14.0	4.844 (1.03,9.14,1.640,18)	6.056 (3.08,8.55,1.316,18)
16.0	5.970 (2.34,9.78,1.858,18)	7.623 (4.47,10.2,1.125,18)
18.0	9.263 (2.45,21.2,3.498,18)	8.543 (5.17,12.0,1.603,18)
20.0	10.276 (3.55,28.4,5.060,18)	10.158 (6.75,15.3,1.865,18)
22.0	18.762 (8.43,44.6,5.871,18)	13.292 (7.18,18.8,2.029,18)
24.0	23.988 (2.02,44.8,9.787,18)	15.840 (11.4,20.9,2.149,18)
26.0	28.658 (5.10,69.1,11.3,18)	17.257 (13.3,21.0,1.850,18)
28.0	42.539 (16.1,75.9,15.9,18)	19.558 (13.6,25.8,2.814,18)

Table 10: Four dimensions, 4000 pieces.

A.5 Trial 4A and 4B: RIPAL With and Without an Initial Solution

Trial 4 contains six pairs of trials, 4A and 4B, 4C and 4D, 4E and 4F. The data in Tables 11 thru 16 below correspond to the plots in Figures 19(a) thru 21(b) respectively in Section 3.7.2. As with all trials reported in this paper, time is reported in seconds, and the superscript for each table entry contains 4 comma-separated values: the min-value, the max-value, the standard deviation, and the number of test problems contributing to that particular entry.

	Trial4A ofs=10 dim=2	
	version-4	version-5
pieces	time	
1000.0	0.080 (0.02,0.22,0.033,60)	0.056 (0.01,0.14,0.023,60)
2000.0	0.099 (0.02,0.28,0.040,60)	0.067 (0.02,0.29,0.028,60)
3000.0	0.147 (0.05,0.33,0.052,60)	0.090 (0.03,0.22,0.032,60)
4000.0	0.147 (0.04,0.35,0.057,60)	0.095 (0.04,0.32,0.034,60)
5000.0	0.178 (0.05,0.46,0.063,60)	0.112 (0.05,0.25,0.034,60)
6000.0	0.192 (0.06,0.40,0.068,60)	0.124 (0.07,0.27,0.036,60)
7000.0	0.196 (0.08,0.42,0.065,60)	0.138 (0.08,0.30,0.043,60)
8000.0	0.223 (0.09,0.54,0.074,60)	0.154 (0.08,0.40,0.041,60)
9000.0	0.266 (0.10,0.54,0.084,60)	0.178 (0.09,0.50,0.054,60)
10000.0	0.266 (0.12,0.56,0.093,60)	0.179 (0.11,0.41,0.043,60)
11000.0	0.294 (0.14,0.73,0.095,60)	0.192 (0.12,0.37,0.047,60)
12000.0	0.303 (0.13,0.76,0.092,60)	0.202 (0.12,0.31,0.043,60)
13000.0	0.302 (0.15,0.73,0.090,60)	0.207 (0.14,0.48,0.045,60)
14000.0	0.327 (0.15,0.69,0.098,60)	0.219 (0.15,0.34,0.038,60)
15000.0	0.328 (0.15,0.99,0.101,60)	0.220 (0.15,0.40,0.047,60)
20000.0	0.419 (0.20,1.00,0.131,60)	0.276 (0.20,0.55,0.048,60)
25000.0	0.447 (0.30,0.91,0.095,60)	0.343 (0.26,0.81,0.051,60)
30000.0	0.518 (0.31,1.18,0.131,60)	0.391 (0.31,0.71,0.043,60)

Table 11: Two dimensions, 10 objective functions.

	Trial4B ofs=10 dim=4	
	version-4	version-5
pieces	time	
1000.0	0.296 (0.06,0.81,0.131,100)	0.273 (0.05,0.79,0.133,100)
2000.0	0.733 (0.12,2.60,0.342,100)	0.643 (0.11,2.39,0.312,100)
3000.0	1.069 (0.09,4.17,0.585,100)	0.960 (0.08,4.14,0.571,100)
4000.0	1.380 (0.13,4.46,0.674,100)	1.211 (0.11,4.21,0.621,100)
5000.0	1.635 (0.24,5.38,0.758,100)	1.424 (0.17,4.94,0.711,100)
6000.0	1.827 (0.16,8.86,1.040,100)	1.570 (0.15,6.95,0.989,100)
7000.0	2.204 (0.25,9.11,1.232,100)	1.890 (0.16,8.29,1.082,100)
8000.0	2.393 (0.15,9.72,1.314,100)	2.085 (0.10,9.61,1.202,100)
9000.0	2.662 (0.38,9.87,1.515,100)	2.183 (0.21,9.35,1.348,100)
10000.0	3.280 (0.13,12.7,1.697,100)	2.718 (0.13,11.0,1.548,100)
11000.0	3.544 (0.21,15.6,2.063,100)	3.009 (0.20,15.6,1.968,100)
12000.0	3.586 (0.22,16.6,2.020,100)	2.971 (0.21,16.1,1.842,100)
13000.0	3.773 (0.37,16.6,2.525,100)	3.234 (0.38,15.5,2.334,100)
14000.0	4.179 (0.18,16.3,1.992,100)	3.415 (0.18,11.0,1.785,100)
15000.0	4.522 (0.47,18.0,2.536,100)	3.649 (0.35,14.4,2.157,100)
20000.0	5.920 (0.77,23.7,3.407,100)	4.746 (0.38,23.3,3.001,100)
25000.0	6.142 (0.37,27.8,3.507,100)	4.787 (0.35,26.4,2.892,100)
30000.0	6.916 (1.09,37.5,3.893,100)	5.713 (0.86,33.4,3.558,100)

Table 12: Four dimensions, 10 objective functions.

	Trial4C dim=2,pieces=4000	
	version-4	version-5
ofs	time	
2.0	0.006 (0.00,0.01,0.005,80)	0.006 (0.00,0.01,0.005,80)
4.0	0.018 (0.01,0.03,0.004,80)	0.014 (0.01,0.02,0.005,80)
6.0	0.038 (0.02,0.07,0.010,80)	0.029 (0.02,0.05,0.006,80)
8.0	0.074 (0.04,0.17,0.020,80)	0.052 (0.03,0.11,0.011,80)
10.0	0.123 (0.05,0.30,0.042,80)	0.088 (0.04,0.22,0.026,80)
12.0	0.217 (0.05,0.79,0.089,80)	0.138 (0.05,0.44,0.055,80)
14.0	0.355 (0.08,1.19,0.140,80)	0.235 (0.08,1.06,0.113,80)
16.0	0.521 (0.09,1.26,0.211,80)	0.362 (0.09,0.98,0.153,80)
18.0	0.890 (0.19,2.74,0.417,80)	0.489 (0.16,1.65,0.233,80)
20.0	1.263 (0.17,3.85,0.571,80)	0.787 (0.16,2.61,0.356,80)
22.0	1.532 (0.34,3.34,0.575,80)	0.973 (0.21,2.38,0.420,80)
24.0	2.066 (0.60,7.56,0.846,80)	1.346 (0.24,3.79,0.545,80)
26.0	2.725 (0.15,6.53,1.200,80)	1.571 (0.16,4.36,0.706,80)
28.0	3.600 (0.48,9.57,1.586,80)	2.226 (0.40,7.18,1.171,80)
30.0	4.329 (0.72,11.0,2.155,80)	2.735 (0.72,7.72,1.330,80)
32.0	6.188 (1.32,18.9,2.489,80)	4.009 (0.28,8.93,1.924,80)
34.0	7.309 (0.27,24.6,3.078,80)	4.110 (0.27,17.2,1.993,80)
36.0	9.353 (1.44,24.7,4.202,80)	5.494 (0.78,15.1,2.576,80)
38.0	10.413 (2.68,23.5,4.184,80)	6.106 (1.42,23.2,2.851,80)
40.0	10.960 (1.71,36.8,5.311,80)	7.393 (0.98,26.0,3.846,80)
42.0	14.874 (2.52,41.6,6.994,80)	9.620 (0.82,24.2,4.575,80)
44.0	19.009 (2.57,66.6,8.273,80)	12.072 (1.78,38.4,5.298,80)
46.0	21.209 (3.93,55.3,8.874,80)	13.954 (1.63,41.0,7.368,80)
48.0	23.485 (2.83,65.0,10.6,80)	14.845 (2.42,50.8,7.165,80)
50.0	25.831 (1.65,74.3,11.8,80)	18.400 (1.45,60.2,9.551,80)
52.0	31.515 (4.21,82.1,11.8,80)	20.194 (2.54,62.0,8.632,80)
54.0	37.993 (5.22,124.6,17.3,80)	26.772 (2.43,101.8,14.9,80)
56.0	40.779 (5.31,85.1,13.9,80)	28.152 (3.98,68.1,12.4,80)
58.0	45.376 (18.3,118.8,16.8,80)	30.724 (5.58,113.4,14.2,80)
60.0	47.809 (10.4,132.6,21.2,80)	30.288 (7.44,86.0,13.2,80)
62.0	57.810 (11.5,144.7,23.8,80)	39.840 (7.50,143.6,18.1,80)
64.0	59.085 (7.87,131.1,20.8,80)	40.312 (5.32,92.1,14.9,80)
66.0	74.377 (10.3,172.6,28.6,80)	53.828 (9.37,154.3,22.1,80)
68.0	77.827 (6.19,259.9,35.1,80)	55.850 (4.32,244.7,28.7,80)
70.0	86.493 (22.3,243.9,37.2,80)	60.094 (19.2,237.0,26.4,80)

Table 13: Two dimensions, 4000 pieces.

	Trial4D dim=4,pieces=4000	
	version-4	version-5
ofs	time	
2.0	0.005 (0.00,0.01,0.005,18)	0.006 (0.00,0.01,0.005,18)
4.0	0.042 (0.02,0.12,0.019,18)	0.036 (0.01,0.11,0.017,18)
6.0	0.139 (0.04,0.37,0.058,18)	0.113 (0.03,0.33,0.058,18)
8.0	0.353 (0.09,0.91,0.174,18)	0.283 (0.05,0.89,0.154,18)
10.0	1.108 (0.16,2.08,0.439,18)	0.930 (0.13,1.93,0.381,18)
12.0	1.744 (0.22,4.16,0.725,18)	1.358 (0.20,3.99,0.605,18)
14.0	4.844 (1.03,9.14,1.640,18)	3.728 (0.30,8.59,1.580,18)
16.0	5.970 (2.34,9.78,1.858,18)	5.103 (1.86,8.98,1.585,18)
18.0	9.263 (2.45,21.2,3.498,18)	8.167 (2.29,20.5,3.494,18)
20.0	10.276 (3.55,28.4,5.060,18)	9.397 (2.41,27.9,5.151,18)
22.0	18.762 (8.43,44.6,5.871,18)	17.384 (7.69,40.7,5.660,18)
24.0	23.988 (2.02,44.8,9.787,18)	22.844 (2.10,45.6,10.0,18)
26.0	28.658 (5.10,69.1,11.3,18)	29.112 (5.08,71.4,12.8,18)
28.0	42.539 (16.1,75.9,15.9,18)	40.545 (16.3,77.6,15.1,18)

Table 14: Four dimensions, 4000 pieces.

	Trial4E dim=2,pieces=4000	
	version-3	version-5
ofs	time	
2.0	0.016 (0.01,0.02,0.005,80)	0.006 (0.00,0.01,0.005,80)
4.0	0.120 (0.06,0.17,0.012,80)	0.014 (0.01,0.02,0.005,80)
6.0	0.334 (0.15,0.47,0.038,80)	0.029 (0.02,0.05,0.006,80)
8.0	0.670 (0.47,0.89,0.069,80)	0.052 (0.03,0.11,0.011,80)
10.0	1.117 (0.81,1.50,0.132,80)	0.088 (0.04,0.22,0.026,80)
12.0	1.738 (1.21,2.13,0.169,80)	0.138 (0.05,0.44,0.055,80)
14.0	2.425 (1.73,3.00,0.190,80)	0.235 (0.08,1.06,0.113,80)
16.0	3.234 (2.19,3.91,0.313,80)	0.362 (0.09,0.98,0.153,80)
18.0	4.056 (2.74,5.12,0.275,80)	0.489 (0.16,1.65,0.233,80)
20.0	5.064 (3.65,6.16,0.421,80)	0.787 (0.16,2.61,0.356,80)
22.0	6.102 (4.42,7.78,0.481,80)	0.973 (0.21,2.38,0.420,80)
24.0	7.126 (5.68,8.59,0.543,80)	1.346 (0.24,3.79,0.545,80)
26.0	8.300 (6.14,9.54,0.573,80)	1.571 (0.16,4.36,0.706,80)
28.0	9.451 (7.23,11.0,0.712,80)	2.226 (0.40,7.18,1.171,80)
30.0	10.956 (8.64,13.4,0.666,80)	2.735 (0.72,7.72,1.330,80)
32.0	12.184 (9.68,14.0,0.848,80)	4.009 (0.28,8.93,1.924,80)
34.0	13.381 (10.3,15.6,0.956,80)	4.110 (0.27,17.2,1.993,80)
36.0	14.868 (11.5,17.3,0.956,80)	5.494 (0.78,15.1,2.576,80)
38.0	16.382 (13.5,19.7,1.020,80)	6.106 (1.42,23.2,2.851,80)
40.0	17.892 (15.0,20.1,1.017,80)	7.393 (0.98,26.0,3.846,80)
42.0	19.549 (16.5,23.5,0.941,80)	9.620 (0.82,24.2,4.575,80)
44.0	21.203 (16.4,24.4,1.342,80)	12.072 (1.78,38.4,5.298,80)
46.0	22.880 (20.1,25.6,1.183,80)	13.954 (1.63,41.0,7.368,80)
48.0	24.353 (19.4,27.6,1.398,80)	14.845 (2.42,50.8,7.165,80)
50.0	26.159 (21.2,30.6,1.416,80)	18.400 (1.45,60.2,9.551,80)
52.0	28.270 (23.4,31.9,1.465,80)	20.194 (2.54,62.0,8.632,80)
54.0	30.205 (24.2,34.7,1.698,80)	26.772 (2.43,101.8,14.9,80)
56.0	31.740 (27.2,36.3,1.388,80)	28.152 (3.98,68.1,12.4,80)
58.0	33.361 (26.4,41.3,1.837,80)	30.724 (5.58,113.4,14.2,80)
60.0	35.764 (29.3,41.7,1.817,80)	30.288 (7.44,86.0,13.2,80)
62.0	37.673 (31.6,44.9,1.833,80)	39.840 (7.50,143.6,18.1,80)
64.0	39.931 (34.0,46.8,2.189,80)	40.312 (5.32,92.1,14.9,80)
66.0	42.291 (36.9,48.2,2.148,80)	53.828 (9.37,154.3,22.1,80)
68.0	44.194 (37.9,50.4,2.169,80)	55.850 (4.32,244.7,28.7,80)
70.0	46.920 (39.0,55.8,2.383,80)	60.094 (19.2,237.0,26.4,80)

Table 15: Two dimensions, 4000 pieces.

	Trial4F dim=4,pieces=4000	
	version-3	version-5
ofs	time	
2.0	0.029 (0.01,0.04,0.010,18)	0.006 (0.00,0.01,0.005,18)
4.0	0.322 (0.14,0.55,0.086,18)	0.036 (0.01,0.11,0.017,18)
6.0	0.831 (0.16,1.41,0.294,18)	0.113 (0.03,0.33,0.058,18)
8.0	1.835 (0.85,3.03,0.566,18)	0.283 (0.05,0.89,0.154,18)
10.0	3.357 (1.99,4.74,0.648,18)	0.930 (0.13,1.93,0.381,18)
12.0	4.626 (3.04,6.47,0.935,18)	1.358 (0.20,3.99,0.605,18)
14.0	6.056 (3.08,8.55,1.316,18)	3.728 (0.30,8.59,1.580,18)
16.0	7.623 (4.47,10.2,1.125,18)	5.103 (1.86,8.98,1.585,18)
18.0	8.543 (5.17,12.0,1.603,18)	8.167 (2.29,20.5,3.494,18)
20.0	10.158 (6.75,15.3,1.865,18)	9.397 (2.41,27.9,5.151,18)
22.0	13.292 (7.18,18.8,2.029,18)	17.384 (7.69,40.7,5.660,18)
24.0	15.840 (11.4,20.9,2.149,18)	22.844 (2.10,45.6,10.0,18)
26.0	17.257 (13.3,21.0,1.850,18)	29.112 (5.08,71.4,12.8,18)
28.0	19.558 (13.6,25.8,2.814,18)	40.545 (16.3,77.6,15.1,18)

Table 16: Four dimensions, 4000 pieces.