An Adaptive Data Structure for Evolutionary Multi-Objective Algorithms with Unbounded Archives

Joseph Yuen, Sophia Gao, Markus Wagner and Frank Neumann
School of Computer Science, The University of Adelaide, Australia

***Abstract*—Archives have been widely used in evolutionary multi-objective optimization in order to store the optimal points found so far during the optimization process. Usually the size of an archive is bounded which means that the number of points it can store is limited. This implies that knowledge about the set of non-dominated solutions that has been obtained during the optimization process gets lost. Working with unbounded archives allows to keep this knowledge which can be useful for the progress of an evolutionary multi-objective algorithm. In this paper, we propose an adaptive data structure for dealing with unbounded archives. This data structure allows to traverse the archive efficiently and can also be used for sampling solutions from the archive which can be used for reproduction.**

***Index Terms*—Archive, Data Structures, Evolutionary Algorithm, Multi-Objective Optimization**

I. Introduction

Evolutionary algorithms (EAs) have been widely used in the field of evolutionary multi-objective optimization (EMO) [3], and is one of the techniques in evolutionary computation that has seen a wide range of practical applications. In fact, the use of evolutionary algorithms is particularly well suited for MOO problems when the goal is not to simply compute a single solution, but to obtain a diverse set of solutions that that represents the trade-offs with respect to the given objective functions. ‘ In general, the population of an EA is rather small compared to the number of trade-offs that the objective functions have. In the case of continuous problems, the number of these trade-offs is usually infinite. For classical combinatorial optimization problems such as the computation of multi-objective shortest paths or multi-objective minimum spanning trees, the number of trade-offs may grow exponentially with respect to the dimensionality of the given input (see [6] which also shows the hardness of these problems).

Archives are frequently used in evolutionary multi-objective optimization to store a set of non-dominated points that have been found during the optimization process. One well-known EMO algorithm is PAES (Pareto-Archived Evolution Strategy) [10] which uses a grid structure for partitioning the objective space, similar to a Quad-Tree [7] which recursively partitions by a factor of 4 as certain regions in the archived solution space becomes densely populated. Furthermore, theoretical studies have investigated the use of archives with respect to convergence [11, 12] as well as their impact on the runtime behavior of simple EMO algorithms [8, 9, 14]. All these studies consider bounded archives and set the number of partitions of the objective space in advance.

Bounded archives can lose information when its capacity is reached and the archive is forced to discard some points. While some research has focused on approaches employed to preserve a good sample of all visited points (like PAES), results from benchmark testing AGE cite show that NSGA-II develops cyclic behaviour when it uses a bounded archive

In this paper, we introduce a new data structure based on trees that allows to work efficiently with large unbounded archive sizes in an adaptive way. Recently, an algorithm called AGE (Approximation Guided EA) [2] working with unbounded archives has been introduced. This algorithm stores every non-dominated point found so far and uses the archive to evaluate the quality of a population by computing its approximation with respect to the archive. It has been shown that this algorithm achieves very good results in terms of ϵ-approximation for various benchmarks problems and high dimensions. Our goal is to study how unbounded archives can be used in a more efficient way to improve the performance of archive-based algorithms such as AGE.

We propose an Adaptive Objective Space Partitioning Tree (APT) structure to regionalise the objective space and archive points in appropriate *partitions*. This tree structure stores partitions in its leaf nodes, and the partition size, shape and location in the objective space is determined by the traversal path from the root node to the leaf node, which can be further sub-divided/branched as necessary. Traversing down the tree structure from the root is equivalent of ’zooming in’ on a region of the objective space. This is comparable to the approach of preserving non-dominated solutions using an Adaptive Grid (AG) Archive in PAES. AG Archive breaks down the objective space into bisections and adjusts the ranges and granularity of each grid location based on the points added to the archive.

However, there are several significant differences between PAES’s AG Archive and the APT Archive we are introducing. The ’adaptive’ behavior of the grid archive subdivides according to the range limits (cut grid into two equally sized pieces), regardless of how the solutions distribute into the two new subdivided grid locations. In our tree structure we will subdivide according to the range values (cut partition into two pieces each containing equal number of points). PAES requires each solution in the archive to be represented by

a binary string, which is used to identify the grid location for comparison and selection, thus it can only perform fixed number of N-sections. The length of the binary string is $2^{\ell \cdot d}$, where ℓ is the number of bisections for each objective and d is the number of objectives. Considering problems with high dimensions such a representation is unsuitable even if ℓ is small, i.e. 2 or 3. Note that the default value for PAES is $\ell = 5$ which implies a vector of length $2^{5 \cdot 5} = 2^{25}$ for problems with 5 dimensions.

In our approach the 'grid location' of the solutions will be implied by the tree branching, and thus divisions at each stage can actually be any number of cuts, which is a lot more flexible. PAES performs density control by discarding some points in crowded regions. In our approach we are not discarding any points thus all non-dominated points are kept, but the tree structure may be used for the selection process of an evolutionary algorithm. For example denser (more crowded) regions can be assigned a lower probability of being selected from such that it is unlikely that over-crowding would occur in our APT structure. PAES updates (add and subdivide grids) has a worst case time of $O((a+n)d)$ where a is the Archive size, n is the population size and d is the number of dimensions. The equivalent updates (add and subdivide partition) we will employ in the APT archive has a worst case time of $O(Dp+plogp)$, where p is the number of points in the partition and D is the depth of the tree. Selecting a point from AG takes $O(D \cdot b)$ and APT archive $O(D \cdot b \log b)$, where b is the number of bi-sections or branches for each objective. This is important as our Archive is unbounded compared to PAES's bounded archive ($a = n$) but can achieve in the worst case similar or faster access and update time even as the archive grows.

We proceed as follows. In Section II, we give an introduce into archive-based multi-objective optimization. Our adaptive data structure for dealing with unbounded archives is presented in Section III. Section IV shows how to use this data structure to sample points from the archive. We finish with some conclusions and topics for future work.

II. Archive-based Evolutionary Multi-Objective Algorithms

In the case of multi-objective optimization, the fitness function maps from the search space X to a vector of real values, i e. $f\colon X \to \mathbb{R}^k$. We consider the case where each of the d objectives should be maximized. For two search points $x \in X$ and $x' \in X$, $f(x) \geq f(x')$ holds iff $f_i(x) \geq f_i(x')$, $1 \leq i \leq d$. In this case, we say that x weakly dominates x' and write $x \succeq x'$. A search point x dominates a search point ($x \succ x'$) x' iff $f(x) \geq f(x')$ and $f(x) \neq f(x')$. In this case x is considered strictly better than x'. The notion of dominance and weak dominance transfers to the corresponding objective vectors. A Pareto optimal search point x is a search point that is not dominated by any other search point in X. The set of non-dominated search points is called the Pareto optimal set and the set of the corresponding objective vectors is called the Pareto front. The archive preserves all points that belong to the Pareto front.

Ranking is the process of performing dominance tests between new candidate solutions with those in the archive to determine if Pareto front has moved closer to the optimal Pareto front. This process may involve adding non-dominated candidate solutions to the archive as well as removing points in the archive if they are now dominated. To perform efficient ranking (dominance testing) with a growing archive, we use internal storage of ranks and a modified ranking update algorithm (see Algorithm 1) to achieve the same ranking mechanism as traditional NSGA-II. The archive-based ranking assumes all points in archive already have a rank. The ranking mechanism itself is broken down into four steps to assimilate the offspring solutions into the archive.

1) The current rank of archive points are placed in temporarily storage. This will be used later for deciding where to move points from if they have new ranks. This is an $O(A)$ operation, as each archive point is accessed once at this step.
2) Compare offsprings with archive points one at a time. Once an offspring's rank is determined with respect to archive, it is added into the archive before the next offspring is ranked. If an offspring dominates an archive point, the archive point loses a rank, and vise versa. The runtime complexity of this step is $O(\lambda \cdot A)$, where λ is the number of offsprings generated at each iteration, and A is the archive.
3) Move each archive point that changed rank during Step 2 to its new front. The runtime complexity of this step is $O(M)$, where M is the number of archive points that needs to be repositioned. In the worst case, all archive points need to be moved and so the worst case runtime complexity of this step is $O(A)$.
4) Determine which archive points to discard. As mentioned, there is no archive size limit, however we retain only non-dominated solutions **and** additional points (if necessary) to form a complete parent population. So we keep the non-dominated fronts plus enough fronts to reach population size, then discard all other dominated fronts. The worst case complexity of this step is $O(f)$, where f is the number of fronts that should be discarded. This step is mostly used to keep the archive as small as practically possible without throwing away any points potentially belonging to the Pareto-optimal front.

Overall, the expected runtime complexity of Algorithm 1 is $O((\lambda + 1) \cdot A + f)$. The runtime is dominated by the size of the archive and its impact will be more apparent for higher dimension problems as majority of the archive will be made up of non-dominated solutions (which won't be discarded).

However, by having internal storage of existing ranks, no redundant calculations are performed as would had been necessary if the entire archive had to recalculate rank each time, which otherwise would have a complexity of $O(A^2)$.

We expect that using an Archive with ranking will cause the runtime to increase, but since all non-dominated points are kept, overall performance in terms of approximation should improve when using the same selection mechanism for any equal number of iterations. Making use of such an archive which includes for each non-dominated objective vector seen so far is the key idea for AGE [2]. The unbounded archive captures in some sense the whole knowledge about the assumed Pareto front and does not loose this information as in the case of bounded archives.

Algorithm 1: Ranking update

Input: A - Archive, O - Offsprings, F - Fronts, μ - population size
Output: Y - points to Discard
Initialize empty set M (holds points that needs repositioning later)
Initialize Map old_ranks (ranks of points before they were updated)
$Y \leftarrow \emptyset$
for $P \in M$ **do**
  $old\_ranks \leftarrow old\_ranks \cup P.rank$;
end for
for $P \in O$ **do**
  $P.rank \leftarrow 0$;
  $old\_ranks[0] \leftarrow P$;
  for $Q \in A$ **do**
    if $P \prec Q$ **then**
      P.rank++; Q.rank–;
    else if $P \succ Q$ **then**
      Q.rank++; P.rank–;
    end if
    if $Q.rank \neq old\_rank(P)$ && $Q \notin M$ **then**
      $M \leftarrow M \cup Q$
    end if
  end for
  if P.rank != old_rank(P) **then**
    $M \leftarrow M \cup P$
  end if
end for
for $P \in M$ **do**
  $i \leftarrow old\_ranks(P)$
  remove P from front i
  add P to front P.rank
end for
$archive\_size = Front(0).size()$
for $f \in F$ **do**
  if $archive\_size > \mu$ **then**
    $Y \leftarrow Y \cup f$
  else
    $archive\_size = archive\_size + f.size()$
  end if
end for
return Y

III. Adaptive Tree Structure

The idea of breaking down the archive into grids for regional density analysis has been considered before, but mostly in the context of a fixed-size archive stored as a List of sorts. Below are the two most popular Archiving data structures:

- Linear Lists. Due to its simplicity in implementation and use, a simple List is a popular choice for MOEAs with elitism. This is most commonly used by SPEA [16].
- Adaptive Grid. breaks up the objective space into n-section hypergrids, where n is the number of equally sliced sub-regions each time an insert triggers a subdivision occurs. PAES [10] and some extensions of NSGA-II [4] uses Adaptive Grid archives. A quad-tree is a special type of Adaptive Grid.

Table below shows a quick comparison of runtime complexity using different non-dominated Archive structures for the basic two operations: insert into archive with dominance check/update, and select (when choosing next generation of parents). Let A be the Archive size, μ be the offspring population size and d be the dimensionality of the objective space.

Archive type	Insert with Update	Select
Linear List	$O(d \cdot A \cdot \mu)$[1]	$O(\|A\|)$[1]
Adaptive Grid	$O((A+\mu) \cdot d \cdot \mu)$[10]	$O(d \cdot \mu)$[10]

The majority of archived-based MOEAs use bounded archives [13], and the general focus of different types of archives is effective elite-preservation given the limited storage. They usually set the number of partitions in advance, and adaptive algorithms such as PAES [10] only adapt the ranges for the partitions in the objective space. As a result, existing gridding (even adaptive) and selection approaches in EAs are designed for overall balance in spatial distribution of archive points rather than to aid effective exploring and pushing of the approximated Pareto front to the Pareto-optimal front, especially in higher dimensions.

Before we start describing the structure we introduce some definitions.

- For a tree representing an archive in a d-dimensional objective space, the *base tree* contains the nodes residing in first d levels. This excludes any leaf nodes that has been sub-divided (which leads to further branching beneath that leaf node).
- A *partition* is a subset of the archive where solutions belonging in this subset resides in the same spatial region of the objective space based on the bounds of the partition. Partitions are stored at leaf nodes of the tree.
- The *branching factor* determines how many branches are to be created when subdividing and as a result the objective value ranges (bounds) that are set for each branch node.
- the *partition threshold* determines the maximum number of archive points that can be stored in any given partition. If this threshold is breached the corresponding partition will be sub-divided into smaller partitions.

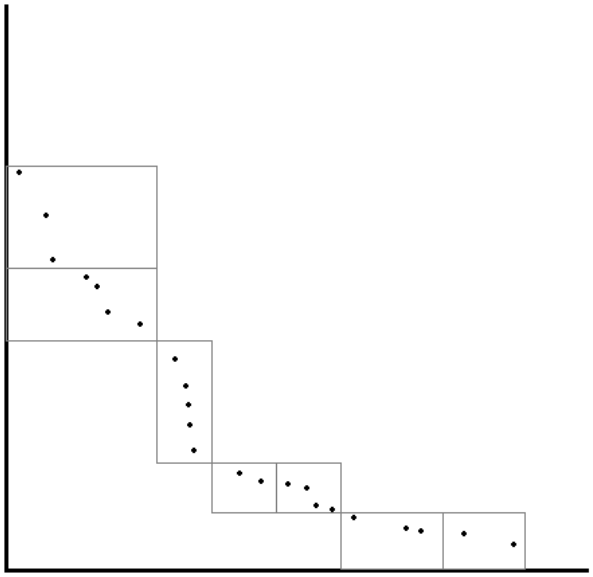


Figure 1: Example of partitioning archive in 2-d objective space (objectives x and y)

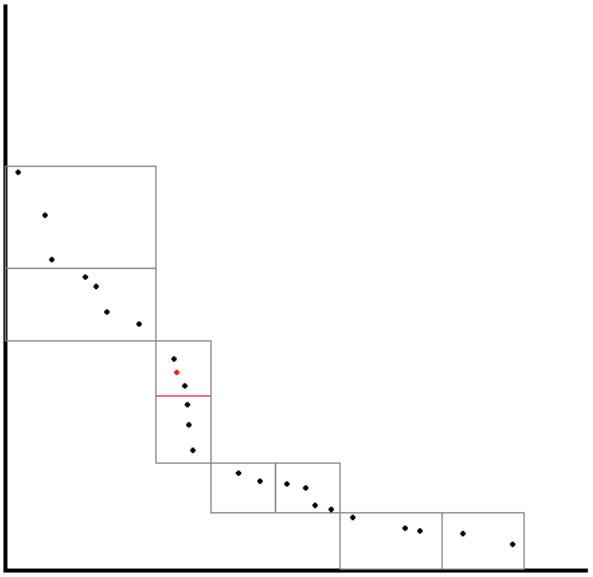


Figure 3: Adding a point to the archive from Figure 1 causing subdivision on the dimension with widest distribution (in this case, y). The number of sub-partitions created by this process is the same as the branching factor.

- A tree node is considered *active* if there is at least one archive point residing in a partition at or below itself. All other nodes are considered inactive (e.g. inactive branch nodes may have been created if branching occured to add a solution into a new partition).
- The *density* of an active node (branch or leaf) is calculated here as the number of archive points at or beneath this node (may include multiple partitions) divided by its objective value range. This density value is used in the population selection process.

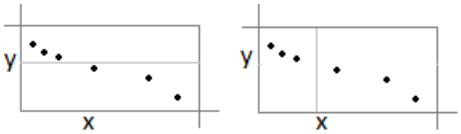


Figure 5: Given the distribution of points within the example partition, a subdivision over objective x is preferred as there are larger gaps over the x values than y

A. General Tree Structure

Initially, each level of the base tree represents one dimension of the objective space, and each branch node represents a value range corresponding to the objective at that level. Thus, a MOO problem with d objectives, the main tree will have d levels, where the branch nodes in tree level i corresponds to objective i. The branching factor and partition threshold will affect the eventual sizes and number of partitions there will be in the tree-structured archive. However when subdivision occurs at the deepest level of the base tree, it could change the objective to divide upon if deemed necessary, although the leaf nodes must be grown beneath the base tree.

Figure 1 shows an example of a partitioned archive and Figure 2 a corresponding APT representation. The first level of the tree is branched based on the objective x, and the second level of the tree is branched based on objective y. See table of Figure 2 for more information regarding each node.

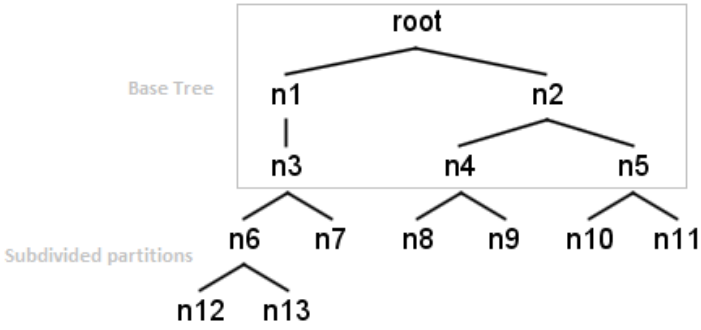
Each node on the tree is filled with information such as its type (branch, leaf or inactive), its objective value range, the objective used to determine branching from this node, its density value and the number of points that are beneath it. Branch nodes can access its parent and child nodes, and leaf nodes store the partition (set of solutions in this objective space region). Self-contained nodes allows for faster access time for adding, removing, subdividing and selecting points since required information is always immediately available during a tree traversal.

At the beginning the tree (archive) is empty, and so the APT structure only consists of the root node (initially inactive, in charge of objective 0). Branches are only grown on a needs basis, usually as a result of adding a new point to the archive.

As the range of possible values for each objective of solutions is unknown at the beginning of the algorithm, a default global range of -10.0 to 10.0 is used on each objective when setting up the base tree, and if any offspring solutions presents an objective value outside this range, the branch range nearest to its value will be widened to accommodate it. Since branches can be expanded on-demand, the initial branch ranges is inconsequential.

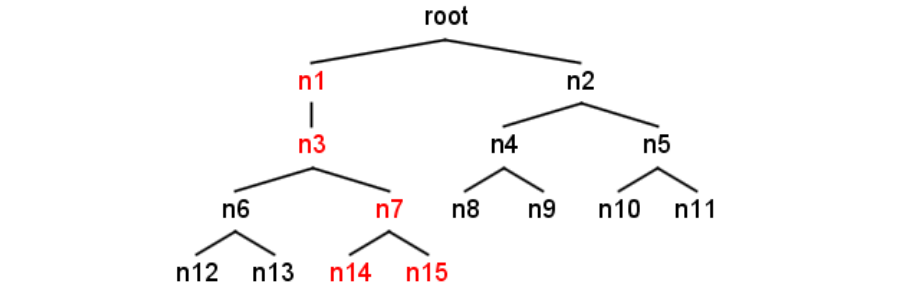
B. Subdividing partitions

When a partition contains more points than the allowed threshold, it is forced to subdivide its partition into smaller



Node	root	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12	n13
Node Type	B	B	B	B	B	B	B	L	L	L	L	L	L	L
Objective	-	x	x	x	x	x	y	-	-	-	-	-	-	-
Points under	23	12	11	12	6	5	7	5	2	4	3	2	3	4

Figure 2: The adaptive tree representation for the partitioned archive from Figure 1. Branch factor is 2, and Partition Threshold is 5.



Node	root	n1	n2	n3	n4	n5	n6	n7	n8	n9	n10	n11	n12	n13	n14	n15
Node Type	B	B	B	B	B	B	B	B	L	L	L	L	L	L	L	L
Objective	-	x	x	x	x	x	y	y	-	-	-	-	-	-	-	-
Points under	24	13	11	13	6	5	7	6	2	4	3	2	3	4	3	3

Figure 4: Updated tree structure on Figure 3. Changes are highlighted in red.

partitions using Algorithm 2. In higher dimensions it is very likely to end up with too many partitions, so during each partitioning, subdivision is performed using only the objective with the widest distribution of values.

To determine which objective value has the widest distribution for the given set of points in the partition, the standard deviation is calculated for each objective value set, and the objective with highest deviation is chosen as the one to subdivide along (see Figure 5 for example). Then the partitions are sorted and evenly grouped by that objective value to form their prospective new partitions. The new value ranges are set using the midpoints between the upper/lower values of adjacent sub-groups.

The most time-consuming part of the subdivision algorithm is determining the standard deviation of each objective value in the partition and sorting partitiong by those objective values, which takes $O(p \times (1 + logp))$ for each dimension of the objective space. As such, runtime complexity of adding a solution to an existing partitiong and breaching threshold using this strategy is $O(d \cdot (p + plogp))$, where d is the depth of the tree and p is the number of points in partition.

C. Adding and removing solutions

Algorithm 3 shows how to add a solution to the archive.

Adding and removing solutions from the archive is as follows. Each *active* branch node stores an ordered list of links to branches or leaves directly beneath it, and stores information such as the objective (dimension) its range values are based on, as well as its own value range for which it will accept points from the branch above itself. Starting from the root node of the tree, the new point is passed down the branches of the tree until it reaches the leaf node it belongs in, and each leaf node stores the partition of points represents a particular region of the objective space.

While traversing, the current branch node will check the relevant objective value of the solution. If value is outside the current branch ranges, the accepting range of the appropriate branch is increased and the point is passed to that branch. Otherwise the point is passed to the branch with an objective value in its accepting range. If point is being added to an

Algorithm 2: Subdividing overfull partition

Input: n - node to be subdivided
$best \leftarrow 0$
for $O \leftarrow 1$ to $no\_of\_objectives$ **do**
  $tmp \leftarrow$ standard deviation of objective O values in partition
  if $tmp > best$ **then**
    $best = tmp$
  end if
end for
set n to branch node, and branching objective to $best$
go up parents to get current value range of objective $best$
create leaf nodes beneath n
sort solutions by objective value $best$
group solutions into $partition.size/branch\_factor$ groups
set branch ranges to midpoints between group ends
for $b \in leaves$ **do**
  add corresponding group of solutions to b
end for
update node info for n
return

Algorithm 3: Adding solution to Adaptive Objective Space Partitioning Tree

Input: n - node of tree, s - new solution
Output: boolean
if n is a leaf node **then**
  add s to $n.partition$
  if $partition.size > threshold$ **then**
    subdivide(n);
  end if
  return $true$
else if n is a branch node **then**
  $i \leftarrow n.objective\_id$
  if $s.obj(i).value < objMin(i)$ **then**
    $objMin(i) = s.obj(i).value$
    update left-most branch's range
  end if
  if $s.obj(i).value > objMax(i)$ **then**
    $objMax(i) = s.obj(i).value$
    update right-most branch's range
  end if
  for $b \in n.branches$ **do**
    if $s.obj(i).value <= b.upperLimit$ & $s.obj(i).value > b.lowerlimit$ **then**
      addToAPTArchive(b, s)
      update node info if add was successful
    end if
  end for
else
  create branches and add to corresponding branch
end if

inactive branch, that branch will be flagged active and its branches and leaves will be grown according to the objective it will be in charge of. If adding a leaf causes the partition to breach the allowed threshold, the leaf will subdivide itself (see Algorithm 2).

Note that even with the expanding of branch ranges and subdivision occuring, *no two partitions will ever overlap in the objective space*. This is because the need to increase range only occurs on the global minimum and maximum values (i.e. left-most and right-most value ranges for each objective). Also, subdividing occurs within the existing partition space, and any sub-divided partitions will never expand its range unless it is the left-most or right-most range for the objective it is dividing over. Thus, there is never any ambiguity on which partition a point belongs in.

Runtime complexity of adding a solution to a new partition is $O(D{\cdot}b)$, where D is the depth of the tree and b is the number of branches checked at each node traversed. In the worst case, we could have a skewed tree where all points are added to the right-most branch (branches are checked left to right), and tree grows very deep in that direction. In this worst case, $d = archiveSize/partition\_threshold$ and $b = branch\_factor$. Thus adding N new points to the tree will take no worse than $O(N \cdot (A/t) \cdot f \cdot O(D \cdot p + p \log p))$ time, where A is size of the archive, t is the partition threshold, and f is branch factor, and we assume there is need for subdivision of the partition the point is added to. However subdivision only occurs once when the threshold is breached, thus if we assume that $N < t$, the worst case becomes $O(N \cdot (A/t) \cdot f + O(D \cdot p + p \log p))$

Removing points is much simpler, as the traversal path down to the corresponding partition will again be determined by its objective values. If the to-be-removed point does not exist in the tree, it will not affect the tree structure. As removing points from the archive will never cause partitions to subdivide, the worst case complexity of removing a point on the worst case tree is $O((A/t) \cdot f)$.

IV. Selection based on Tree Structure

Different variants for selecting a point based on the tree structure are possible as different type of information can be stored at the nodes of the tree. These points could for example be used to produce offspring in the next iteration.

In the following, we present two approaches based on a measure of density. The notion of density is very common in evolutionary multi-objective algorithms (see for example NSGA-II [5] and SPEA2 [15]) and it seems to be useful make use of density information when traversing the tree structure. When traversing down the APT structure to select a point, we prefer to take branches (representing objective space regions) with lower density. To select a parent population from the APT archive we will need to traverse the tree once for each point, and because we favor less dense regions, more points will probably be selected from them.

Algorithm 4: selectPoint

```
if node is a leaf && partition.size() > 0 then
    randomly select one point from partition
    return point
end if
branch_arr ← ∅
for b ∈ branches do
    branch_arr ← branch_arr ∪ b
end for
sort ascending branch_arr by density
i = 0
while branch not chosen && i < branches.size − 1 do
    select branch_arr(i) with probability p
    if branch chosen, break loop
    i++
end while
if branch not chosen then
    chosen_branch = densest branch
end if
return chosen_branch.selectPoint()
```

The simplest approach based on density is density-strict selection. This requires that at each level the branch with the lowest density is chosen. Where multiple branches have the same density value, the path less travelled is chosen. This option allows for even greater focus on the sparse regions, but remains fair due to the incremental nature of the archive. Once sufficient points are explored in the least dense regions it will no longer be least dense and the next least dense partition will become preferred.

A. Density-probability selection

In the following, we present a density-probability approach to select point from the archive. We store at each node of the tree the number of points that belong to this interval of the tree. Based on the number of points that are in an interval its density is determined. An example algorithm for selecting a point from the archive is outlined in Algorithm 4. We sort each branch based on its density. Then we select the least dense branch with probability $1/p$, or second least dense branch with probability $(1-1/p)\cdot(1/p)$, the third least dense branch with probability $(1-1/p)^2\cdot(1/p)$, etc $(0<p<1)$. And once we reach a leaf node we randomly select a point from that partition. Using this method all partitions have a chance of being selected, but points in less explored regions of the objective space will definitely be preferred. Selecting p is very important as it will determine the degree of focus in our selection. If p is large, the focus is greater, and vice versa. The runtime complexity of density-probability selection is $O(D\cdot f\log f)$, where D is the depth of the tree and f is the branching factor.

Comparing the runtime complexity of the Adaptive Tree structure with other existing structures, Adaptive Tree has better worst case runtime performance than Archive structures currently preferred by state-of-the-art EAs and is expected to scale particularly well for higher dimensions.

Archive type	Insert with Update	Select
Linear List	$O(d\cdot A\cdot\mu)$[1]	$O(\|A\|)$[1]
Adaptive Grid	$O((A+\mu)\cdot d\cdot\mu)$[10]	$O(d\cdot\mu)$[10]
Adaptive Tree	$O(\mu\cdot(\frac{A}{t})\cdot f+d\cdot p\log p)$	$O(d\cdot f\log f)$

CONCLUSIONS

Archives have been frequently used in evolutionary multi-objective optimization. They are usually used to store a set of non-dominated objective vectors. There are different methods for partitioning the objective space but they are all non-adaptive with respect to the number of partitions. Furthermore, the size of an archive is often bounded.

The benefit of unbounded archives has been recently shown by the success of the algorithm AGE proposed in [2] and it is desirable to have efficient data structures for unbounded archives. With this paper, we have introduced a new adaptive data structure that can be used in such algorithms. It is based on a tree structure that partitions the objective space in an adaptive way that depends on the set of points stored so far. We have shown that various operations such as insertion, deletion and selection based on density measures are efficiently supported. In the future, we are planning to integrate this new data structure into archive-based multi-objective evolutionary algorithms such as AGE.

BIBLIOGRAPHY

[1] Quad-trees: A data structure for storing pareto sets in multiobjective evolutionary algorithms with elitism. *Evolutionary Multiobjective Optimization*, pages 81–104, 2005.

[2] K. Bringmann, T. Friedrich, F. Neumann, and M. Wagner. Approximation-guided evolutionary multi-objective optimization. In *Proceedings of 21nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pages 1198–1203, Barcelona, Spain, 16-22 July 2011. IJCAI/AAAI.

[3] K. Deb. *Multi-objective optimization using evolutionary algorithms*. Wiley, 2001.

[4] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA–II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, Apr. 2002.

[5] K. Deb, A. Pratap, S. Agrawal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions Evolutionary Computation*, 6(2):182–197, 2002.

[6] M. Ehrgott. *Multicriteria optimization*. Berlin, Springer, 2nd edition, 2005.

[7] R. Finkel and J. Bentley. *Quad Trees: A Data Structure for Retrieval on Composite Keys*. Springer, 1974.

[8] C. Horoba. Analysis of a simple evolutionary algorithm for the multiobjective shortest path problem. In I. I.

Garibay, T. Jansen, R. P. Wiegand, and A. S. Wu, editors, *Proceedings of Foundations of Genetic Algorithms (FOGA X)*, pages 113–120. ACM, 2009.

[9] C. Horoba and F. Neumann. Benefits and drawbacks for the use of epsilon-dominance in evolutionary multi-objective optimization. In C. Ryan and M. Keijzer, editors, *Proceedings of Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 641–648. ACM, 2008.

[10] J. D. Knowles and D. Corne. Approximating the nondominated front using the pareto archived evolution strategy. *Evolutionary Computation*, 8(2):149–172, 2000.

[11] J. D. Knowles and D. Corne. Properties of an adaptive archiving algorithm for storing nondominated vectors. *IEEE Transactions Evolutionary Computation*, 7(2):100–116, 2003.

[12] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler. Combining convergence and diversity in evolutionary multiobjective optimization. *Evolutionary Computation*, 10(3):263–282, 2002.

[13] M. López-Ibáñez, J. Knowles, and M. Laumanns. On sequential online archiving of objective vectors. In R. H. C. Takahashi, K. Deb, E. F. Wanner, and S. Greco, editors, *Proceedings of Evolutionary Multi-criterion Optimization (EMO 2011)*, volume 6576 of *Lecture Notes in Computer Science*, pages 46–60. Springer, 2011.

[14] F. Neumann and J. Reichel. Approximating minimum multicuts by evolutionary multi-objective algorithms. In *Proceedings of Parallel Problem Solving from Nature X (PPSN '08)*, volume 5199 of *Lecture Notes in Computer Science*, pages 72–81. Springer, 2008.

[15] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the strength Pareto evolutionary algorithm for multiobjective optimization. In *Proceedings of Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*, pages 95–100, 2002.

[16] E. Zitzler and L. Thiele, editors. *Multiobjective evolutionary algorithms: A comparative case study and the strength pareto approach*. IEEE.