A comparison of Guided Local Search and NSGA-III for solving multi-objective optimization problems

Jeffrey Quesnelle
University of Michigan-Dearborn

December 9, 2015

Abstract

Naïve search algorithms that look for an optimal or near-optimal solution are unacceptable when the solution space is exponentially (or otherwise prohibitively) sized. For multi-objective optimization problems there often does not exist a single optimal solution for all objectives. Rather, our aim is to produce candidate solutions that offer the best "trade-offs" between the objectives. We compare Guided Local Search, a general stochastic method for searching a solution space, to NSGA-III, an evolutionary algorithm for solving multi-objective optimization problems.

1 Introduction

An optimization problem asks us to determine the "best" solution from all feasible solutions to a problem. Generally, we are provided with an *objective* function that "scores" each solution; our goal is to find either the minimum or maximum of this function. Augmenting the objective function are a set of *constraints* on the variables which must be satisfied for any solution. A multi-objective optimization problem (MOOP) is an extension of optimization problems that allows for more than one objective function. An example MOOP is given in Figure 1.

$$\begin{array}{rl} \text{minimize} & f_1(x_1, x_2, x_3) \\ \text{or} & f_2(x_1, x_2, x_3) \\ \text{maximize} & \\ \\ \text{subject to} & c_1(x_1, x_2, x_3) \leq 0 \\ & c_2(x_1, x_2, x_3) = 0 \\ & c_3(x_1, x_2, x_3) > 0 \end{array}$$

Figure 1: Example MOOP

Linear and integer programming has long been used to solve a wide variety of problems in engineering and the sciences, but these methods fail when the objective problems are inherently non-linear. Algorithms that can effectively deal with such non-linear objectives and constraints offer solutions to a strict superset of problems that can be modeled with linear programming, although algorithms for solving the latter are often much faster. We shall investigate two algorithms for solving MOOPs: a general search technique known as Guided Local Search and a specialized evolutionary algorithm known as NSGA-III.

1.1 Multi-objective optimization

Single-objective optimization is characterized by a single "cost" function that is either minimized or maximized. Since there is only one such function, we are asked to find the single assignment of variables that provides the designated optimal objective value.

Definition 1 (Objective function)**.** An *objective function* is a function $f : \mathbb{R}^n \to \mathbb{R}$.

In a multi-objective scenario with n variables and m objectives we can compose a new function F as given in Figure 2. We see that $F : \mathbb{R}^n \to \mathbb{R}^m$. Since there is no total order on $\mathbb{R}^m$ for $m > 1$

$$F(x_1, x_2, \ldots, x_n) = \begin{pmatrix} f_1(x_1, x_2, \ldots, x_n) \\ f_2(x_1, x_2, \ldots, x_n) \\ \cdots \\ f_m(x_1, x_2, \ldots, x_n) \end{pmatrix}$$

Figure 2: Construction of the composite objective

there is no single assignment of variables that we can call "the best". Thus for MOOPs, we do not ask for a single point that results in an optimal objective, but rather a collection of points that we determine to be "more optimal" than the rest.

Definition 2 (Pareto dominance)**.** Let P be an m-objective minimization MOOP on n variables with composite objective function F. Also, let x, y be vectors in $\mathbb{R}^n$ that satisfy the constraints of P. Then, $F(x)$ *dominates* $F(y)$ if and only if for each $k \in \{1, 2, \ldots, m\}$ $f_k(x) \leq f_k(y)$ and there exists some k such that $f_k(x) < f_k(y)$.

Definition 3 (Pareto Front)**.** Let P be an m-objective minimization MOOP on n variables with composite objective function F. The *Pareto Front* is the subset $\{F(x), F(y), \ldots\}$ of all valid solutions to P that are non-dominated by any other solution.

If a point $F(x)$ dominates $F(y)$ then we can say that $F(x)$ is "at least as good" as $F(y)$ for all objectives and is "better than" $F(y)$ in at least one objective. For a MOOP P, the concept of Pareto dominance assigns a partial order to the set of valid solutions. The set of solutions that are not dominated by any other point are known collectively as the Pareto Front; in essence it is the collection of best "trade-offs" between the objectives. The purpose of a MOOP algorithm is to find or approximate the Pareto Front.

2 Guided Local Search

Because most optimization problems don't have a simple algebraic method to solve them algorithms tend to use some sort of search to find the optimal points. If the objective functions do not behave in a smooth manner then an exhaustive search may be necessary, which is impossible. However, by assuming that the objectives behave in a somewhat continuous manner the search algorithms can make inferences about the behavior of the functions around points.

Guided Local Search (GLS) is a general stochastic metaheuristic that can be applied to an existing search algorithm. Many search algorithms will continue to refine a solution if the objective is improved on every iteration. GLS attempts to "break out" of local optimums so as to find the

global optimum. It achieves this by penalizing properties of locally optimal solutions so that the underlying search algorithm is biased away from refining those solutions; the hope is that this will allow it to discover the globally optimal solution. The structure of GLS is given in Algorithm 1.

Algorithm 1 Guided Local Search [1]

```
function GuidedLocalSearch(Iter_max, λ)
    f_penalties = ∅
    S_best = RandomSolution()
    for Iter_i ∈ Iter_max do
        S_curr = LocalSearch(S_curr, λ, f_penalties)
        f_utilities = CalculateFeatureUtilities(S_curr, f_penalties)
        f_penalties = UpdateFeaturePenalties(S_curr, f_penalties, f_utilities)
        if Cost(S_curr) ≤ Cost(S_best) then
            S_best = S_curr
        end if
    end for
end function
```

GLS penalizes "features", which can be any property of a solution to the given problem. $f_{utilities}$ is a vector of scalars calculated from a candidate solution S_{curr} which describes the magnitude of each feature in the solution. From $f_{utilities}$ the $f_{penalties}$ vector is calculated, which will be used to augment the Cost calculation inside of LocalSearch. Presumably the augmented Cost calculation will cause LocalSearch to search in a direction it otherwise would not have with the original Cost, thus penalizing any solutions that heavily exhibit features selected by UpdateFeaturePenalties. The λ parameter is a constant vector used to scale the penalties.

2.1 Greedy Pareto Local Search

The GLS algorithm requires a LocalSearch function that is able find new solutions to the problem based on the augmented cost functions given by the penalties. Such a search function must be picked specifically for the target problem. Greedy Pareto Local Search (GreedyPLS) is a search algorithm that attempts to build up a Pareto Front by exploring neighboring solutions and building up an "archive" of non-dominated points. The main procedure for GreedyPLS is given in Algorithm 2.

The generic Guided Local Search algorithm is designed for any problem, but the LocalSearch function must be tailored to the problem. GreedyPLS takes a set of augmented objective functions $[g_1, \ldots, g_k]$ and a candidate PF *archive* (which may be empty). The algorithm creates a new solution in the neighborhood of an existing solution in *archive* and determines if the new solution dominates the previous one. Once a solution is found that is non-dominated by the points in the PF, we call UpdateArchive which adds the solution to the archive and prunes solutions once a predetermined limit is reached (these are the HL and SL parameters). The UpdateArchive procedure is given in Algorithm 3.

The goal of GreedyPLS is to find an approximate PF and it is the responsibility of UpdateArchive to ensure that this PF is of a manageable size. A candidate point is only added to *archive* if it is non-dominated by all other points in the archive. If the point dominates other points in *archive* those points are removed, maintaining the invariant that all points in the PF are non-dominated. When the PF contains SL (soft limit) points $SL - HL$ points are removed from the archive. To choose which points to remove the crowding distance is calculated for each point and

Algorithm 2 Greedy Pareto Local Search [2]

```
function GreedyParetoLocalSearch([g_1, ..., g_k], archive, HL, SL)
    if archive = ∅ then
        s_0 = InitialSolution()
        archive = s_0
    end if
    while ∃s ∈ archive such that Visited(s) = false do
        for s' ∈ Neighborhood(s) = false do
            Evaluate(s', [g_1, ..., g_k])
            if s' dominates s then
                s = s'
                fails = 0
            else if s does not dominate s' then
                UpdateArchive(s', archive, HL, SL)
            else
                fails = fails + 1
                if fails > maxFails then
                    break
                end if
            end if
        end for
        Visited(s) = true
    end while
    return archive
end function
```

Algorithm 3 Update Archive [2]

function UPDATEARCHIVE($s, archive, HL, SL$)
  if $archive = \emptyset$ **then**
    $archive = \{s\}$
  else
    $added = true$
    for $s' \in archive$ **do**
      if s' dominates s **then**
        $added = false$
        break
      else if s dominates s' **then**
        $archive = archive \setminus \{s'\}$
      end if
    end for
    if $added = true$ **then**
      $archive = archive \cup \{s\}$
    end if
  end if
  if $|archive| > SL$ **then**
    $Dist =$ CROWDINGDISTANCEASSIGNMENT($archive$)
    $archive =$ SORT($archive, Dist$)
    $DelSet = \{archive[1], \ldots, archive[|archive| - HL]\}$
    $archive = archive \setminus DelSet$
  end if
end function

those points with the smallest distance are deleted. In essence, CrowdingDistanceAssignment assigns to each point a scalar that represents how clustered together each point is. The points that are more crowded are assumed to be less favorable than those farther away. CrowdingDistanceAssignment is given in Algorithm 4.

Algorithm 4 Crowding distance assignment [3]

function CrowdingDistanceAssignment(A)
  $K = |A|$
  for $1 \leq i \leq K$ **do**
    $Dist_{A[i]} = 0$
  end for
  for each objective m **do**
    $A = \text{Sort}(A, m)$
    $Dist_A[1] = \infty$
    $Dist_A[K] = \infty$
    for $2 \leq i \leq K - 1$ **do**
      $Dist_{A[i]} = Dist_{A[i]} + (A[i+1].m - A[i-1].m) \,/\, (A[K].m - A[1].m)$
    end for
  end for
end function

GreedyPLS was designed for discrete optimization problems such as Traveling Salesman. The ability to enumerate and visit all members with Neighborhood implies a discrete structure which we lack for generic multi-objective optimization. Research has been conducted in ways to deterministically search the neighborhood space by viewing the real numbers as binary strings [4] but this can be extremely computationally expensive for more than two objectives. We chose a slightly different approach to Neighborhood: using a genetic polynomial mutation approach that randomly mutates on average one variable per iteration to produce a new candidate solution. For brevity we exclude listing the algorithm here – see [5].

2.1.1 Running time of GreedyPLS

The running time of GreedyParetoLocalSearch depends on UpdateArchive which in turn may invoke CrowdingDistanceAssignment. CrowdingDistanceAssignment loops over each objective, and for each objective over every point. Since $|archive| \leq SL$, if M is the number of objectives then CrowdingDistanceAssignment is $O(M \times SL)$. UpdateArchive also loops over every point to determine if the candidate solution is non-dominated, which is $O(SL)$. Since $SL \leq M \times SL$, UpdateArchive is $O(M \times SL)$.

GreedyParetoLocalSearch searches the Neighborhood of each candidate solution once. Our implementation of Neighborhood generates a constant number G mutations of s per run. Each point may be non-dominated requiring a call to UpdateArchive. The Visited property of each solution is set to *true* regardless of whether or not the solution was actually replaced by a new mutated solution inside the "for" loop. Thus, the "while" loop is bounded by the maximum size of $|archive|$, which is SL. Therefore the complexity of GreedyParetoLocalSearch is $O(SL \times G \times M \times SL) = O(SL^2 \times G \times M)$.

2.2 Guided Pareto Local Search

The generic Guided Local Search (Algorithm 1) procedure must be adopted to our specific problem. We must identify properties of our solutions that can be penalized so as to escape local optimum. We define the utilities as whether or not the variable values for a solution are present in fixed size "bins". For example, in an one-objective, one-variable problem with the variable bounded by $0 \leq x \leq 1$, if we choose to create ten utilities then each utility would represent if the variable is between $[0, 0.1), [0.1, 0.2), \ldots, [0.9, 1.0]$. If a specific utility is penalized, then the next iteration of GreedyPLS will be less likely to search solutions with values in the penalized range. This method was first pioneered by [4]. Our implementation of GLS for multi-objective optimization is given by Algorithm 5.

Algorithm 5 Guided Pareto Local Search [2]

function GUIDEDPARETOLOCALSEARCH($[g_1, \ldots, g_M], \lambda, [I_1, \ldots, I_K], [c_1, \ldots, c_K], HL, SL, R$)
  $archive = \emptyset$
  for $1 \leq i \leq K$ **do**
    $p_i = 0$
  end for
  for $1 \leq j \leq M$ **do**
    $h_j = g_j + \lambda_j \cdot \sum_{i=1}^{K} p_i \cdot I_i$
  end for
  for $1 \leq r \leq R$ **do**
    GREEDYPARETOLOCALSEARCH($[h_1, \ldots, h_M], archive, HL, SL$)
    for $1 \leq i \leq K$ **do**

$$util_i = I_i(archive) \cdot \frac{\left(\frac{c_i \cdot \gamma_i}{|archive|}\right)}{1 + p_i}$$

    end for
    for each i such that $util_i$ is maximum **do**
      $p_i = p_i + 1$
      for $s \in archive$ such that $I_i(s) = 1$ **do**
        $Visited(s) = false$
      end for
    end for
  end for
end function

The augmented objective functions are created by adding the term $\lambda_j \cdot \sum_{i=1}^{K} p_i \cdot I_i$ to the original objective functions g. p_i is the specific accrued penalty for each feature and I_i is a 0-1 indicator function that determines if a given solution exhibits the ith feature (i.e. if the variable is inside our current "bucket").

The calculation of the utilities is performed by first taking the ratio of solutions that exhibit each feature (here γ_i is the number of solutions in the archive with i, that is $\gamma_i = \sum_{i=1}^{K} I_i$) and multiplying it by the cost for the ith feature. The cost vector is a tuning parameter to the algorithm that allows us to weight certain features more heavily than others. For general purpose multi-objective algorithms we often cannot surmise anything before about the behavior of the variables and may set all $c = 1$. This quantity is then divided by the current penalty value for this feature. The features that have the highest utility value after this calculation are chosen to be penalized – any solutions

that exhibit this feature will have their objective scores set higher than they otherwise would have in the next iteration of the local search. Assuming a minimization problem this translates to a bias against such solutions. Since the penalty value occurs in the denominator of the utility calculation subsequent evaluations of $util_i$ for this feature will be smaller, meaning that we are less likely to penalize this same feature again.

2.2.1 Running time of GPLS

GPLS is dominated by the loop that performs a local search and then calculates utilities and penalizes the highest scoring features. Since GPLS is an approximation algorithm we can run it as many times as is needed – the number of loops is controlled simply by parameter R. Since GreedyPLS is $O(SL^2 \times G \times M)$ we simply have that GPLS is $O(SL^2 \times G \times M \times R)$.

3 NSGA-III

Evolutionary (or genetic) algorithms are a class of approximation algorithms that work by mutating an existing solution and then evaluating its fitness. If the mutated version is better than the parent solution it is "selected" and the competing, less-fit solutions are allowed to "die off". The general structure of a genetic algorithm is given in Figure 3.

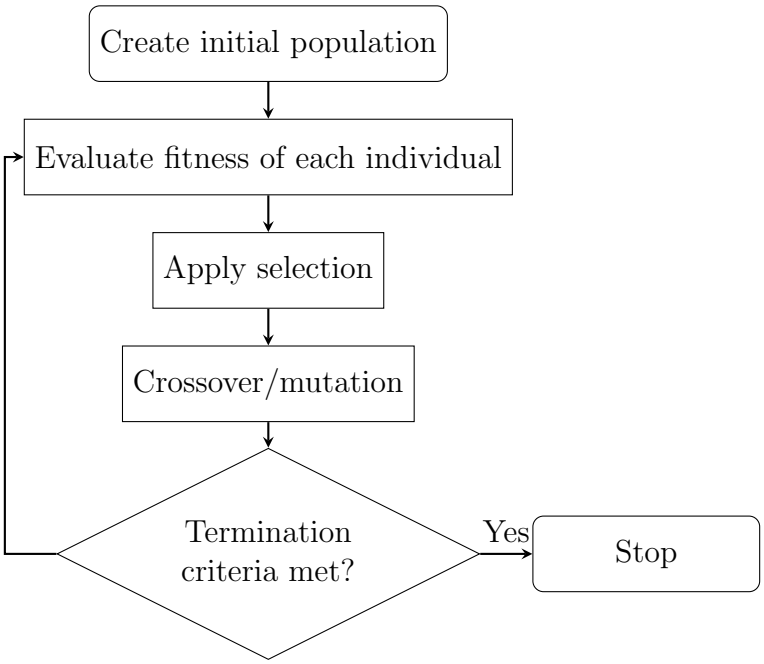


Figure 3: Genetic algorithm flowchart

NSGA-III is a state-of-the-art evolutionary multi-objective optimization algorithm (EMOA). It is heavily based (unsurprisingly) on NSGA-II, which is also an EMOA. NSGA-II performs well for two or three objective problems, but scales poorly as the number of objectives increase. The reason for this is that the number of non-dominated points in a proposed Pareto Front usually grows exponentially with the number of objectives. NSGA-II relies on sorting the PF via the partial order non-domination imposes on the set; if the number of non-dominated points is very large this can require significant computation time. By significantly modifying the selection process used by

NSGA-II, NSGA-III is able to improve both the accuracy of and time needed to find solutions to MOOPs. Since NSGA-III is a genetic algorithm, it can continue on for an indefinite number of iterations (known as generations). One generation of NSGA-III is given by Algorithm 6.

Algorithm 6 Generation t of NSGA-III [6]

Input: H constructed reference points Z^s or supplied aspiration points Z^a, parent population P_t
Output: P_{t+1}
$S_t = \emptyset, i = 1$
$Q_t =$ Recombination+Mutation(P_t)
$R_t = P_t \cup Q_t$
$(F_1, F_2, \dots) =$ Non-dominated-sort(R_t)
repeat
  $S_t = S_t \cup F_i$ and $i = i + 1$
until $|S_t| \geq N$
Last front to be included: $F_l = F_i$
if $|S_t| = N$ **then**
  $P_{t+1} = S_t$, **break**
else
  $P_{t+1} = \cup_{j=1}^{l-1} F_j$
  Points to be chosen from F_l: $K = N - |P_{t+1}|$
  Normalize objectives and create reference set Z_r: Normalize$(\mathbf{f}^n, S_t, Z^r, Z^s, Z^a)$
  Associate each member $\mathbf{s}$ of S_t with a reference point: $[\pi(\mathbf{s}), d(\mathbf{s})] =$ Associate (S_t, Z^r)
  Compute niche count of reference point $j \in Z^r$: $p_j = \sum_{\mathbf{s} \in S_t / F_t} ((\pi(\mathbf{s}) - j) \ ? \ 1 \ : \ 0)$
  Choose K members one at a time from F_l to construct P_{t+1}: Niching$(K, \rho_j, \pi, d, Z^r, F_l, P_{t+1})$
end if

Algorithm 6 begins at the crossover/mutation stage. The authors of NSGA-III recommend simulated binary crossover (SBX) and polynomial mutation for Recombination+Mutation. SBX merges two real values encoded as finite-length binary strings (typically floating point values in a computer). For a complete description of SBX, see [5]. Polynomial mutation was previously used in the description of GPLS. The objective values for each variable assignment generated by this procedure are automatically calculated.

After crossover and mutation, the original parent population along with the new mutated population are combined and enter the fitness phase. The first procedure the algorithm uses to determine fitness is to apply a partial order onto the population via a non-dominated sort. The partial order is achieved by assigning each point a rank; points that dominate another point in the population and are themselves non-dominated are assigned rank "1". These points are then removed from the population and the process is repeated, assigning this second set rank "2". Once again the process repeats until no points are left in the population and each point has a rank assigned to it. The steps for this sort are given in Algorithm 7.

Once a rank is assigned to each member we build up a unified population set by adding in all the ranks starting with F_1 (the most non-dominated points) until F_l, which is the point at which the size of the population is greater than some predetermined N. If the size of the new population is exactly N then the generation is complete, otherwise we continue on to the selection phase.

Since the population is greater than our target number of individuals, some must be pruned. It is at this point that NSGA-III significantly diverges from NSGA-II. NSGA-II uses CrowdingDis-

Algorithm 7 Non-dominated sort of the population [3]

```
function NON-DOMINATED-SORT(P)
    for p ∈ P do
        S_p = ∅
        n_p = 0
        for q inP do
            if p dominates q then
                S_p = S_p ∪ {q}
            else if q dominates p then
                n_p = n_p + 1
            end if
        end for
        if n_p = 0 then
            p_rank = 1
            F_1 = F_1 ∪ {p}
        end if
    end for
    i = 1
    while F_i ≠ ∅ do
        Q = ∅
        for p ∈ F_i do
            for q ∈ S_p do
                n_q = n_q − 1
                if n_q = 0 then
                    q_rank = i + 1
                    Q = Q ∪ {q}
                end if
            end for
        end for
        i = i + 1
        F_i = Q
    end while
    return {F_1, F_2, ...}
end function
```

tanceAssignment (see Algorithm 4) to break ties among points in the same rank. NSGA-III uses a more elaborate selection mechanism. Search algorithms tend to "cluster" their solutions together as incremental improvements are found. NSGA-II attempts to avoid this behavior by picking points that are assigned a large distance value by CrowdingDistanceAssignment. Given the exponential growth of non-dominated points as the number of objectives increase it appears unlikely that a fast algorithm will find well or uniformly distributed candidate points for the PF. As [6] says: *"It is getting increasingly clear that it is too much to expect from a single population-based optimization algorithm to have convergence of its population near the Pareto-optimal front and simultaneously it is distributed uniformly around the entire front in a large-dimensional problem."* NSGA-III cedes some style points by beginning with a presumably well-distributed set of reference points and selecting those points fit certain closeness metrics to these points.

During the fitness phase points were added to the population one rank at a time. The final rank that caused the population to exceed its maximum size, F_l, is the rank from which we need to remove members; all previous ranks will be included in P_{t+1}. After including all points with rank less than l in P_{t+1} we need to choose the remaining $K = N - |P_{t+1}|$ points. To achieve this normalized objectvies $\mathbf{f}^n$ and reference points Z^r are created from the supplied objectives and reference points (these reference points can be automatically generated: see [7]). This procedure can be found as Algorithm 8.

Algorithm 8 Normalize objectives and create reference set [3]

function Normalize($\mathbf{f}^n, S_t, Z^r, Z^s, Z^a$)
  for $1 \leq j \leq M$ **do**
    Compute ideal point: $z^{min} = \min_{\mathbf{s} \in S_t} f_j(\mathbf{s})$
    Translate objectives: $f'_j(\mathbf{s}) = f_j(\mathbf{s}) - z_j^{min} \quad \forall \mathbf{s} \in S_t$
    Compute extreme points ($z^{j,max}$), $j = 1, \ldots, M$) of S_t
  end for
  Compute intercepts a_j for $j = 1, \ldots, M$
  Normalize objectives: $f_i^n(\mathbf{x}) = \dfrac{f'_i(\mathbf{x})}{a_i}, \quad$ for $i = 1, 2, \ldots, M$.
  if Z^a is given **then**
    Map each (aspiration) point on a normalized hyper-plane using the normalized objectives and save the points in the set Z^r
  else
    $Z^r = Z^s$
  end if
end function

The normalized set is created by treating the optimal objective across all points in the population as a vector, i.e. creating the ideal point $\bar{z} = (z_1^{min}, z_2^{min}, \ldots)$ where z_i^{min} is the optimal value for objective i among all points in S_t. Augmented objective functions f'_j are created by subtracting off the ideal point $\bar{z}$, which effectively becomes a zero vector in the augmented objective space. The maximum point for each augmented objective $z^{j,max}$ is found in S_t. For each augmented objective the intercepts a_j are discovered and the normalized objectives are created by dividing the augmented objectives (calculated by subtracting off z_j^{min}) by the intercepts. The normalized objectives and their intercepts form a hyper-plane in M-space which has the effect of equalizing those objectives that are scaled differently. If the reference set was auto-calculated then these points already exist on the hyper-plane, otherwise the user-supplied points are translated to the normalized space and

returned as Z^r. After the objectives are normalized and a reference set is chosen each member $\mathbf{s}$ of the population S_t is associated with a reference point by a special distance metric. This association is given in Algorithm 9.

Algorithm 9 Associate each member $\mathbf{s}$ of S_t with a reference point [3]

function ASSOCIATE(S_t, Z^r)
  for $\mathbf{z} \in Z^r$ **do**
    Compute reference line $\mathbf{w} = \mathbf{z}$
  end for
  for $\mathbf{s} \in S_t$ **do**
    for $\mathbf{w} \in Z^r$ **do**
      Compute $d^{\perp}(\mathbf{s}, \mathbf{w}) = ||(\mathbf{s} - \mathbf{w}^T\mathbf{s}\mathbf{w}/||\mathbf{w}||^2)||$
    end for
    Assign $\pi(\mathbf{s}) = \mathbf{w} : \arg\min_{\mathbf{w} \in Z^r} d^{\perp}(\mathbf{s}, \mathbf{w})$
    Assign $d(\mathbf{s}) = d^{\perp}(\mathbf{s}, \pi(\mathbf{s}))$
  end for
end function

The distance metric between population points and reference points is calculated by first drawing hyper-lines from the origin of the normalized space through each reference point. The perpendicular distances of each point in the population from these reference lines are calculated and for each point the closest reference line is chosen to be associated with it. Now that each population point is associated with a reference point (it may be that each reference point has more than one population point), we use a "niching" procedure to selectively choose which points will be in the final population. The niching procedure is given in Algorithm 10.

Algorithm 10 Choose K members one at a time from F_l to construct P_{t+1} [3]

function NICHING($K, \rho_j, \pi, d, Z^r, F_l, P_{t+1}$)
  $k = 1$
  while $k \leq K$ **do**
    $J_{\min} = \{j : \arg\min_{j \in Z^r} \rho_j\}$
    $\bar{j} = \text{random}\,(J_{\min})$
    $I_{\bar{j}} = \{\mathbf{s} : \pi(\mathbf{s}) = \bar{j}, \mathbf{s} \in F_l\}$
    if $I_{\bar{j}} \neq 0$ **then**
      if $\rho_{\bar{j}} = 0$ **then**
        $P_{t+1} = P_{t+1} \cup \left(\mathbf{s} : \arg\min_{\mathbf{s} \in I_{\bar{j}}} d(\mathbf{s})\right)$
      else
        $P_{t+1} = P_{t+1} \cup \text{random}\left(I_{\bar{j}}\right)$
      end if
      $\rho_{\bar{j}} = \rho_{\bar{j}} + 1, F_l = F_l \setminus \mathbf{s}$
      $k = k + 1$
    else
      $Z^r = Z^r \,/\, \{\bar{j}\}$
    end if
  end while
end function

The NICHING procedure must select K additional members from F_l for P_{t+1}. Let ρ_j be the number of population points in all ranks except F_l associated to the jth reference point. We find $J_{\min}$, the set of reference points having the minimum number of population points associated with it (i.e. the smallest ρ). Let $\bar{j}$ be any one of these reference points. If $\rho_{\bar{j}} = 0$ then there is no member of P_{t+1} that is associated with reference point $\bar{j}$. In such a scenario the closest of the points from F_l that is associated with $\bar{j}$ is chosen for inclusion in P_{t+1} (assuming there exists such a point). If $\rho_{\bar{j}} > 0$, indicating that there already exists a population point associated with $\bar{j}$, a random point from those in F_l associated with $\bar{j}$ is included. In essence, NICHING selects points for P_{t+1} by favoring those points close to reference points.

After NICHING is finished there is a new population P_{t+1} with N members. The new population has individuals selected in order of their non-denomination ranks, with closeness to reference points used as the tie-breaker. The termination criteria can now be evaluated (usually simply a fixed number of iterations) and the algorithm restarted on this new population if it is not met.

3.1 Running time of NSGA-III

For a complete analysis of the complexity of a generation of NSGA-III see [6]. If N is the size of the target population and M is the number of objectives in the MOOP then the complexity of NSGA-III is $O(N^2 \log^{M-2} N)$ or $O(N^2 M)$, whichever is larger.

4 Experimental results

4.1 Quality indicators

In multi-objective optimization problems our goal is to find the Pareto Front, which may contain many members. To evaluate an algorithm, we must have some way of knowing how "good" a candidate PF is. The quality of a PF can be measured both by how close it is to the true PF as well its "spread" (how uniform its coverage over the true PF is) [9]. Several metrics such as the "C-Metric" for coverage and "D-metric" for distance are used throughout the literature. We shall use the inverse generational distance (IGD) metric, which combines both coverage and distance into a single value [10] [11].

4.2 Test problems

To compare the algorithms we used a set of four test problems described in [5]. The test problems are scalable to an arbitrary number of objectives, although the 3-objective versions are presented here. The parameters for GPLS are given in Figure 4 and the parameters for NSGA-III are given in 5.

	DTLZ1	DTLZ1	DTLZ3	DTLZ4
λ	0.5	0.1	0.2	0.2
Penalties	200	200	200	200
GPLS generations	400	250	1000	600
Polynomial mutation prob.	$1/n$	$1/n$	$1/n$	$1/n$
Polynomial mutation η	0.05	0.1	0.3	0.2
HL	100	100	100	100
GreedyPLS generations	10	10	20	10

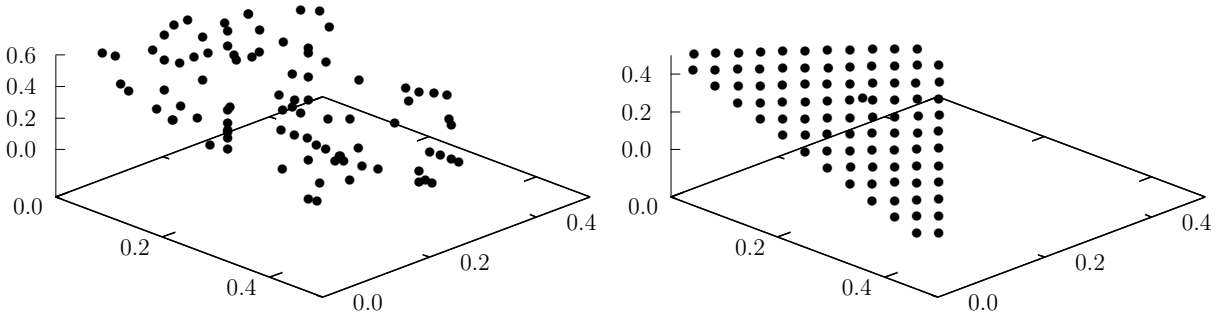
Figure 4: Parameters for GPLS

	DTLZ1	DTLZ1	DTLZ3	DTLZ4
SBX probability	1	1	1	1
SBX η	30	30	30	30
Polynomial mutation prob.	$1/n$	$1/n$	$1/n$	$1/n$
Polynomial mutation η	20	20	20	20
Generations	400	250	1000	600

Figure 5: Parameters for NSGA-III

4.2.1 DTLZ1

$$
\begin{aligned}
\text{Minimize} \quad & f_1(\mathbf{x}) = \tfrac{1}{2} x_1 x_2 \cdots x_{M-1}(1 + g(\mathbf{x}_M)), \\
\text{Minimize} \quad & f_2(\mathbf{x}) = \tfrac{1}{2} x_1 x_2 \cdots (1 - x_{M-1})(1 + g(\mathbf{x}_M)), \\
& \vdots \quad \vdots \\
\text{Minimize} \quad & f_{M-1}(\mathbf{x}) = \tfrac{1}{2} x_1 (1 - x_2)(1 + g(\mathbf{x}_M)), \\
\text{Minimize} \quad & f_M(\mathbf{x}) = \tfrac{1}{2} (1 - x_1)(1 + g(\mathbf{x}_M)), \\
\text{subject to} \quad & 0 \leq x_i \leq 1, \quad \text{for } i = 1, 2, \ldots, n, \\
\text{with} \quad & g(\mathbf{x}_M) = 100 \left[|\mathbf{x}_M| + \sum_{x_i \in \mathbf{x}_M} (x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5)) \right].
\end{aligned}
$$

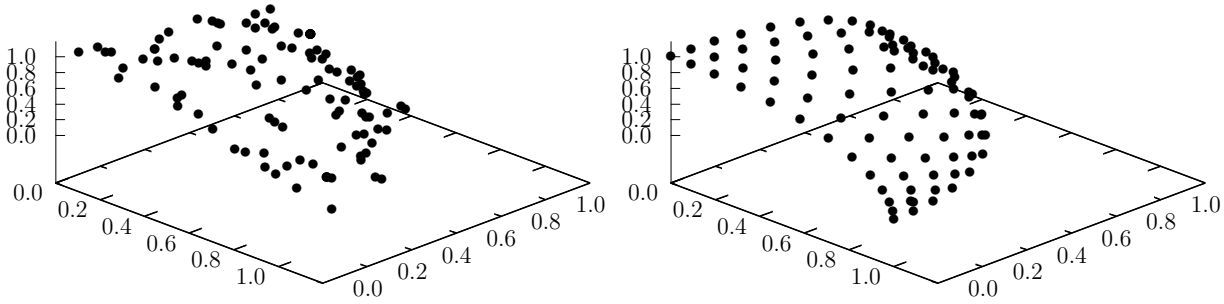


(a) GPLS, $IGD = 0.125268$ (b) NSGA-III, $IGD = 0.002023$

Figure 6: Obtained solutions for DTZL1

4.2.2 DTLZ2

$$
\begin{aligned}
\text{Minimize} \quad & f_1(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \cos(x_{M-2}\pi/2) \cos(x_{M-1}\pi/2), \\
\text{Minimize} \quad & f_2(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \cos(x_{M-2}\pi/2) \sin(x_{M-1}\pi/2), \\
\text{Minimize} \quad & f_3(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \sin(x_{M-2}\pi/2), \\
& \vdots \quad \vdots \\
\text{Minimize} \quad & f_{M-1}(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \sin(x_2\pi/2), \\
\text{Minimize} \quad & f_M(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \sin(x_2\pi/2), \\
\text{subject to} \quad & 0 \leq x_i \leq 1, \quad \text{for } i = 1, 2, \ldots, n, \\
\text{where} \quad & g(\mathbf{x}_M) = \sum_{x_i \in \mathbf{x}_M} (x_i - 0.5)^2.
\end{aligned}
$$

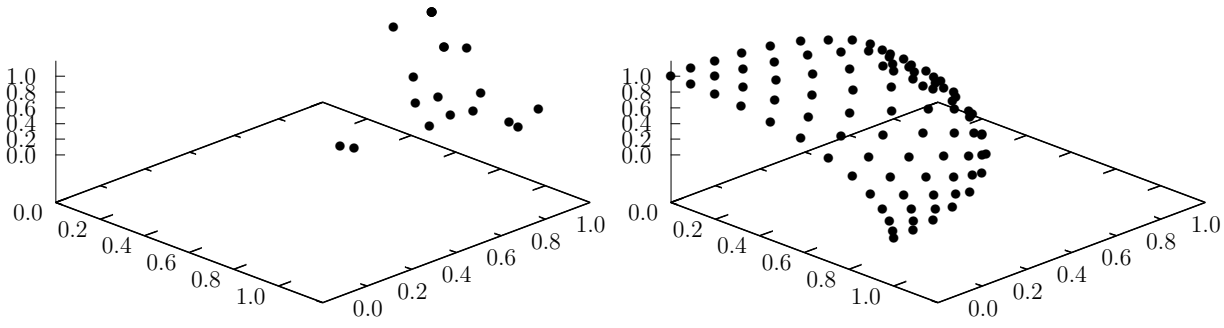


(a) GPLS, $IGD = 0.091910$ (b) NSGA-III, $IGD = 0.002349$

Figure 7: Obtained solutions for DTZL2

4.2.3 DTLZ3

$$
\begin{aligned}
\text{Minimize} \quad & f_1(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \cos(x_{M-2}\pi/2) \cos(x_{M-1}\pi/2), \\
\text{Minimize} \quad & f_2(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \cos(x_{M-2}\pi/2) \sin(x_{M-1}\pi/2), \\
\text{Minimize} \quad & f_3(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \cos(x_2\pi/2) \cdots \sin(x_{M-2}\pi/2), \\
& \vdots \quad \vdots \\
\text{Minimize} \quad & f_{M-1}(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1\pi/2) \sin(x_2\pi/2), \\
\text{Minimize} \quad & f_M(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \sin(x_2\pi/2), \\
\text{subject to} \quad & 0 \leq x_i \leq 1, \quad \text{for } i = 1, 2, \ldots, n, \\
\text{where} \quad & g(\mathbf{x}_M) = 100 \left[|\mathbf{x}_M| \sum_{x_i \in \mathbf{x}_M} (x_i - 0.5)^2 - \cos(20\pi(x_i - 0.5)) \right].
\end{aligned}
$$

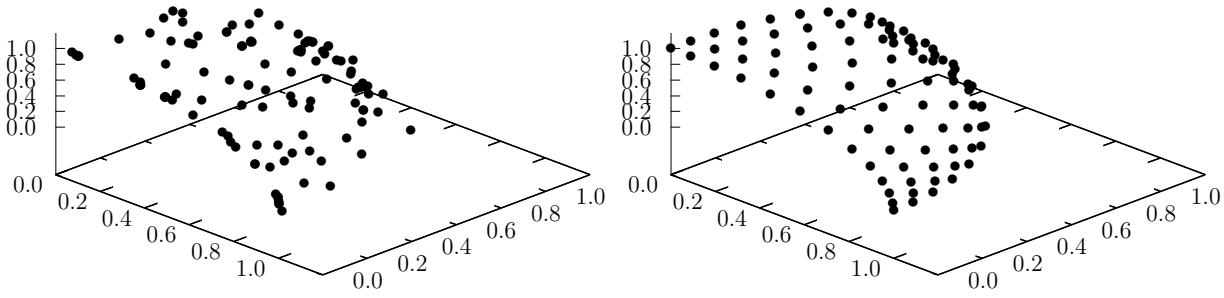


(a) GPLS, $IGD = 0.698297$ (b) NSGA-III, $IGD = 0.001373$

Figure 8: Obtained solutions for DTZL3

4.2.4 DTLZ4

$$
\begin{aligned}
\text{Minimize} \quad & f_1(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1^\alpha \pi/2) \cos(x_2^\alpha \pi/2) \cdots \cos(x_{M-2}^\alpha \pi/2) \cos(x_{M-1}^\alpha \pi/2), \\
\text{Minimize} \quad & f_2(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1^\alpha \pi/2) \cos(x_2^\alpha \pi/2) \cdots \cos(x_{M-2}^\alpha \pi/2) \sin(x_{M-1}^\alpha \pi/2), \\
\text{Minimize} \quad & f_3(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1^\alpha \pi/2) \cos(x_2^\alpha \pi/2) \cdots \sin(x_{M-2}^\alpha \pi/2), \\
& \vdots \quad \vdots \\
\text{Minimize} \quad & f_{M-1}(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \cos(x_1^\alpha \pi/2) \sin(x_2^\alpha \pi/2), \\
\text{Minimize} \quad & f_M(\mathbf{x}) = (1 + g(\mathbf{x}_M)) \sin(x_2^\alpha \pi/2), \\
\text{subject to} \quad & 0 \leq x_i \leq 1, \quad \text{for } i = 1, 2, \ldots, n, \\
\text{where} \quad & g(\mathbf{x}_M) = \sum_{x_i \in \mathbf{x}_M} (x_i - 0.5)^2, \quad \alpha = 100.
\end{aligned}
$$



(a) GPLS, $IGD = 0.075753$ (b) NSGA-III, $IGD = 0.000455$

Figure 9: Obtained solutions for DTZL4

4.3 Results

For each test problem and algorithm twenty instances were run. The results of these runs can be found in Figure 10.

		GPLS	NSGA-III
DTLZ1	Best	0.125268	0.000760
	Worst	0.715844	0.020530
	Average	0.327588	0.005068
DTLZ2	Best	0.084152	0.001585
	Worst	0.214039	0.006910
	Average	0.116311	0.002938
DTLZ3	Best	0.343200	0.001374
	Worst	2.178580	0.011295
	Average	1.138119	0.004961
DTLZ4	Best	0.075753	0.000245
	Worst	1.041450	0.532995
	Average	0.471198	0.133457

Figure 10: Results (IGD) of test problems

5 Conclusion

GPLS and NSGA-III are search algorithms for solving multi-objective optimization problems. Both algorithms prefer points that are non-dominated by other points in the candidate Pareto-optimal front. The algorithms differ in two main ways: in how they generate new points, and how points are removed from the solution set when it grows too large.

GPLS generates new points simply by randomly mutating one variable of an existing point and evaluating its fitness. NSGA-III first merges two existing points together via simulated binary crossover before the mutation. The mechanism chosen by NSGA-III helps ensure that candidate points are "pushed" towards the existing solution set, reducing the number of non-optimal points considered.

When the solution set grows too large GreedyPLS uses the crowding distance of the points to determine which will be removed from the solution set. Points that are heavily clustered together are less likely to be chosen than outlier points which (hopefully) will help the algorithm explore new areas of the Pareto Front. NSGA-III uses a reference point based approach: objectives are normalized and then points are associated with reference lines that shoot out from the origin to each reference point. Points are chosen preferentially if they are near the reference points.

Experimentally NSGA-III produces superior solutions to GPLS. GPLS, despite its penalization scheme, performed particularly poorly on DLTZ3 which was designed specifically to test an algorithm's ability to break out of a local optimum and converge to the global optimum. The core neighborhood search mechanism of GLS is ill-suited to problems of a continuous nature and the algorithm is unable to effectively navigate the search space. It is recommended to use NSGA-III for solving multi-objective optimization problems.

References

[1] Brownlee, J. 2011. Clever algorithms: Nature-Inspired Programming Recipes. `http://www.cleveralgorithms.com/nature-inspired/stochastic/guided_local_search.html` Accessed: November, 2015.

[2] Alsheddy, A. 2011. Empowerment Scheduling: A Multi-objective Optimization Approach Using Guided Local Search. Ph.D. Thesis. University of Essex: England.

[3] Deb, K., Pratap, A., Agarwal, S., and T. Meyarivan. 2002. A fast and elitist multiobjective generic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation, 6, 182-197. xiii, 48, 52, 173

[4] Voudouris, C. 1998. Guided Local Search: An Illustrative Example in Function Optimisation. BT Technology Journal, 16, 3, 46-50

[5] Deb, K. and R. B. Agrawal. 1995. Simulated binary crossover for continuous search space. Complex Systems, 9, 115-148.

[6] Deb, K. and J. Himanshu. 2014. An evolutionary many-objective optimization algorithm using reference-point-based nondominated sorting approach, part i: solving problems with box constraints. IEEE Transactions on Evolutionary Computation, 18, 4, 577-601

[7] Das, I. and J. Dennis. 1998. Normal-boundary intersection: A new method for generating the Pareto surface in nonlinear multicriteria optimization problems. SIAM Journal of Optimization, 8, 3, 631-657.

[8] Deb, K., Thiele, L., Laumanns, M. and E. Zitzler. 2005. Scalable test problems for evolutionary multi-objective problems. Evolutionary Multi-objective Optimization. London, U.K.: Springer-Verlag, 105-145

[9] Zitzler, E., Deb, K., and L. Thiele. 2001. Comparison of multiobjective evolutionary algorithms: Empirical results. Evoluationary Computation, 8, 173-195, 44

[10] Zhang, Q., Zhou, A., Zhao, S. Z., Suganthan, P. N., Liu, W., and S. Tiwari. 2008. Multiobjective optimization test instances for the CEC-2009 special session and competition. Nanyang Technological University, Singapore. `http://www.ntu.edu.sg/home/epnsugan/`

[11] Veldhuizen, D. V., and G. B. Lamont. 1998. Multiobjective evolutionary algorithm research: A history and analysis. Department of Electrical and Computer Engineering, Air Force Institute of Technology, Dayton, OH, USA. Technical Report TR-98-03.