OUT OF THE BOX OPTIMIZATION USING THE PARAMETER-LESS POPULATION PYRAMID

By

Brian W. Goldman

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor Of Philosophy

2015

ABSTRACT

OUT OF THE BOX OPTIMIZATION USING THE PARAMETER-LESS POPULATION PYRAMID

By

Brian W. Goldman

The Parameter-less Population Pyramid (P3) is a recently introduced method for performing evolutionary optimization without requiring any user-specified parameters. P3's primary innovation is to replace the generational model with a pyramid of multiple populations that are iteratively created and expanded. In combination with local search and advanced crossover, P3 scales to problem difficulty, exploiting previously learned information before adding more diversity.

Across seven problems, each tested using on average 18 problem sizes, P3 outperformed all five advanced comparison algorithms. This improvement includes requiring fewer evaluations to find the global optimum and better fitness when using the same number of evaluations. Using both algorithm analysis and comparison we show P3's effectiveness is due to its ability to properly maintain, add, and exploit diversity.

Unlike the best comparison algorithms, P3 was able to achieve this quality *without any problem-specific tuning*. Thus, unlike previous parameter-less methods, P3 does not sacrifice quality for applicability. Therefore we conclude that P3 is an efficient, general, parameter-less approach to black-box optimization that is more effective than existing state-of-the-art techniques.

Furthermore, P3 can be specialized for gray-box problems, which have known, limited, non-linear relationships between variables. Gray-Box P3 leverages the Hamming-Ball Hill Climber, an exceptionally efficient form of local search, as well as a novel method for performing crossover using the known variable interactions. In doing so Gray-Box P3 is able to find the global optimum of large problems in seconds, improving over Black-Box P3 by up to two orders of magnitude.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

A primary purpose of evolutionary optimization is to efficiently find good solutions to challenging real-world problems with minimal prior knowledge about the problem itself. This driving goal has created search algorithms that can escape user bias to create truly novel results, sometimes publishable or patentable in their own right [18]. While it is not possible for any algorithm to do better than random search across all possible problems [38], effectiveness can be achieved by assuming the search landscape has structure and then biasing the algorithm toward exploiting that structure.

In evolutionary optimization, and genetic algorithms (GAs) in particular, search is often biased through parameters. This can be beneficial as it allows practitioners to inject their knowledge about the shape of the search landscape into the algorithm. However, the quality of solutions found, and the speed at which they are found, is strongly tied to setting these parameters correctly [8]. As such, either expert knowledge or expensive parameter tuning [13] are required to leverage this feature to its fullest potential. Furthermore, parameters such as population size, mutation rate, crossover rate, tournament size, etc. usually have no clear relationship to the problem being solved, meaning even domain experts may not understand how the parameters will interact with the problem or with each other. To further complicate matters, there is mounting evidence that parameter values should change during search [11, 19].

There have been periodic efforts to reduce or remove the need for parameter tuning.

[27] introduced self-adaptive parameters, in which parameter values were included in each solution's genome and themselves underwent evolution. This allowed the search process itself to optimize some of its own parameters, resulting in a reduced need for expert tuning. [15] was able to design an entirely parameter-less GA by leveraging schema theory and parallel populations. Unfortunately these methods were provably less efficient than directly setting the parameters to optimal values [24].

One area that has been effective at reducing the number of algorithm parameters is model based search. [22]'s Hierarchical Bayesian Optimization Algorithm (hBOA) and [33]'s Linkage Tree Genetic Algorithm (LTGA) both require only a single parameter: population size. [25] leveraged model building to create a fully parameter-less algorithm, but it is restricted to only order-k, fully decomposable, noiseless problems.

Most recently we introduced the Parameter-less Population Pyramid (P3) [9]. This method uses a pyramid structure of populations to combine model based search with local search to achieve parameter-less optimization. Initial results suggest that, unlike previous parameter-less methods, P3 is more efficient than current state-of-the-art parameterized search algorithms. In this work we shall: extend these results to cover more comparison algorithms; compare both efficiency in reaching the global optimum and intermediate fitnesses; analyze algorithm complexity; and provide more in depth analysis of P3 itself.

Chapter 2

Comparison Optimizers

In order to fully understand the effectiveness of P3, we compare it with five advanced algorithms that have related features to P3. The Random Restart Hill Climber defined by [9] was chosen as an efficient form of repeated local search. As P3 combines this hill climber with crossover, comparing with local search alone shows the advantages of P3's overall approach. The $(1 + (\lambda, \lambda))$ algorithm [5] is the current best theory supported simple genetic algorithm and its method of crossover is in some sense a macro-mutation just as in P3. hBOA and Parameter-less hBOA are advanced model building search techniques that are effective at learning complex problem structure, designed to achieve similar goals as P3's linkage learning but using very different methods. Finally LTGA represents the current state-of-the-art in black-box search and is the origin of P3's linkage learning and crossover methods.

Only hBOA and LTGA require any parameters, with each of these only requiring a population size. This makes knowing the optimal behavior of these algorithms much more tractable. All of the algorithms are also gene order independent, fitness scale invariant, and unbiased. This means, for any problem, the order in which problem variables appear in the genome can be changed without changing the behavior of the search. The fitness can also be manipulated in any fashion as long as the rank ordering of solutions is unchanged. These algorithms are also unaffected by the meaning assigned to each bit, such that inverting a predetermined random subset of genes before evaluation will not impact search efficiency.

Our implementations of all of these algorithms as well as all of the population size

```
1: procedure HILL-CLIMBER
2:     options ← [0 ... N − 1]
3:     tried ← ∅
4:     while |tried| < |options| do
5:         for all index ∈ shuffled(options) do
6:             if index ∉ tried then
7:                 Flip bit index in solution
8:                 if solution's fitness increased then
9:                     tried ← ∅
10:                else
11:                    Revert change
12:                end if
13:                tried ← tried ∪ {index}
14:            end if
15:        end for
16:    end while
17: end procedure
```

Figure 2.1: Hill climbing algorithm used to improve randomly generated solutions until no single bit change results in a fitness improvement.

information, raw results, and processing scripts are available from our website.[1]

2.1 Random Restart Hill Climber

Perhaps the simplest black-box search heuristic is stochastic local search, also know as hill climbing. This optimization technique focuses on improving a single solution until it reaches a local optimum. Here we use the first-improvement hill climber defined by [9] and given in Figure 2.1. This algorithm works by flipping each bit in a random order, keeping modifications when fitness is improved, until single bit flips cannot result in further fitness improvements.

The hill climber requires an amortized cost of $\mathcal{O}(1)$ operations per evaluation. In order to terminate, at least one evaluation must be performed for each of the N bits in the solution. As such any operation that happens only once per search can be amortized over at least N evaluations, covering the initialization of *options* on Line 2. Line 6, which prevents wasted

[1] `https://github.com/brianwgoldman/FastEfficientP3`

evaluations, can be called at most twice per evaluation: once to add *index* into *tried* and once to prevent *index* from being unnecessarily evaluated again. The only way three or more calls could happen is if no fitness improvement was made for the entire previous iteration, which contradicts the loop invariant.

Due to its nature, this hill climber cannot escape basins of attraction. Once a solution is reached such that none of the single bit neighbors are fitness improvements, search stops. Thus this algorithm requires a restart mechanism to solve multimodal problems. We have chosen here to naïvely restart search from a random solution whenever a local optima is found. This ensures that on all landscapes there is always a non-zero probability of search finding the global optimum.

2.2 $(1 + (\lambda, \lambda))$

[5] presented the first genetic algorithm to provably show the advantages of performing crossover on the problem known as One Max. This comparatively simple algorithm maintains only a single individual and a self-controlled parameter λ.

Each iteration, the number of bits to flip is chosen from the binomial distribution $b \sim B(N, \frac{\lambda}{N})$, where N is the number of bits in the genome. Next, $\lfloor \lambda \rfloor$ offspring are produced by flipping b bits. The best mutant then produces $\lfloor \lambda \rfloor$ offspring via uniform crossover with the original parent, such that each gene comes from the mutant with probability $\frac{1}{\lambda}$. In the original algorithm the best offspring produced by crossover then replaces the original parent if its fitness is no worse. The λ parameter, which is initialized to 1, is decreased if the offspring replaced its parent and increased otherwise.

The original formulation was designed specifically for unimodal landscapes and as such were not directly suitable for multimodal problems. [9] extended $(1 + (\lambda, \lambda))$ to include random restarts. As search stagnates, the λ parameter increases in value. Eventually this results in $\lambda \geq N$ causing mutation to always flip all bits of the individual. As this prevents any future improvement, whenever $\lambda \geq N$ search is restarted from a random solution with

λ reset to 1.

A few other efficiency modifications were also made. If there is a tie in crossover offspring fitness, whichever has a larger hamming distance from the parent is retained. This encourages drifting across plateaus. The "mod" control strategy proposed by [5] was not used as it conflicted with the random restart strategy. If a crossover individual is identical to either of its parents, it is not evaluated. If mutation produces an offspring that is better than the best crossover offspring, it is used to compare against the original parent.

2.3 Hierarchical Bayesian Optimization Algorithm

[22] used statistical principles in combination with a decision tree structure to create the Hierarchical Bayesian Optimization Algorithm (hBOA). This method creates a model of epistatic relationships between genes which is then used to stochastically generate new solutions. Each generation a binary tournament with replacement is used to select μ solutions from the population. These solutions are then used to build the model, which in turn is used to generate μ new solutions. The new solutions are then integrated into the population using restricted tournament replacement.

Conceptually, the model built by hBOA is trying to infer rules of the form "Given that this subset of genes are set to these values, how frequently is gene x_i set to value v?" This can be represented using a directed acyclic decision forest, with each tree in the forest representing one gene in the solution. In the decision tree T_i, which is used to set the value of gene x_i, each internal node represents previous decisions on how to set some other gene x_j, with the children of that node representing how the decision was made. The leaves of each tree give the probability that x_i should be set to one of the possible gene values.

The forest is constructed iteratively, with each tree initially containing a single leaf and with each leaf storing a pointer for each selected solution. Each iteration the algorithm considers all possible ways of splitting an existing leaf using another gene x_j, such that solutions in the leaf are moved to the newly created leaves based on their value for x_j. The

general goal is to separate the solutions such that all solutions with $x_i = 0$ move to one leaf while solutions with $x_j = 1$ move to the other.

This goal is formalized using model scoring from Bayesian statistics. In its raw form this almost always creates near infinitesimal results, calculating fractions that include factorials of μ and products over N. However, through algebraic manipulation discussed in Appendix 11, we derived a simplified form shown in Equation 2.1. Here l is a leaf in tree i, with l' and l'' the results of splitting l. $m_i(l)$ is the number of solutions that reach l and $m_i(x_i, l)$ is the number of solutions that reach l with the given value for x_i. If no proposed split satisfies the inequality, iteration stops. If multiple splits do, whichever maximizes the right side is chosen.

$$2^{0.5log_2\mu} < \frac{(m_i(l)+1)!}{m_i(0,l)!m_i(1,l)!} \cdot \frac{m_i(0,l')!m_i(1,l')!m_i(0,l'')!m_i(1,l'')!}{(m_i(l')+1)!(m_i(l'')+1)!} \tag{2.1}$$

Initially there are $\Theta(N^2)$ possible ways to split existing leaves, as each of the N single node trees can be split by any of the other $N-1$ genes. Each iteration a new edge is added to the decision forest, meaning some of the previously tested splits cannot be used. For instance, if T_i, which is used to decide the value of x_i, is split using the value of x_j, T_j can no longer be split using x_i. As a split creates two new leaves, $\mathcal{O}(N)$ new potential splits must also be tested. Equation 2.1 parses all solutions that reach a leaf to count gene frequencies, requiring μ time. The number of total leaves created depends heavily on the problem and μ. However, assuming no splits are accepted or that the cost of testing all future splits is less than the initial $\Theta(N^2)$, constructing the model requires $\Omega(\mu N^2)$ time. Each model is used to generate μ solutions, leading to a cost per evaluation of $\Omega(N^2)$.

To generate a solution, the value of each gene x_i is set using its corresponding decision tree T_i. Because the forest is directed acyclic, there must be an ordering of T_i such that, before T_i is executed, all x_j it uses to make decisions have already been set. As such, previous decisions made by other trees are used to follow each T_i until a leaf is reached. The value of x_i is then set based on the probability that other solutions reached that leaf with each value

of x_i.

To perform replacement, hBOA uses restricted tournament replacement. After each new solution is generated and evaluated, a set of w solutions are chosen at random from the population, where $w = \min\{N, \frac{\mu}{20}\}$. From this set the solution which is most genetically similar to the offspring is chosen. If the offspring is at least as fit as the chosen solution, it replaces the chosen solution in the population. Otherwise the offspring is discarded. This method is designed to preserve genetic diversity as only genetically similar solutions compete on fitness.

hBOA is designed to work with large population sizes, resulting in a large number of evaluations per generation. As hBOA utilizes explicit diversity maintenance, standard methods for determining convergence are not considered very accurate. Therefore the authors suggest that an hBOA run should be terminated after performing generations equal to N.

Like other model based techniques, hBOA has few parameters. There is no mutation or crossover, and modeling does not rely on any explicit parameters. Solution selection, generation, and replacement are all derived from the population size, which must be set by the user.

2.4 Parameter-less hBOA

Using the methods first introduced by [15] for the Parameter-less GA, [23] created Parameter-less hBOA which automatically scales its population size to fit the problem. This is done by maintaining a list of concurrent populations using exponentially scaled population sizes.

A run of Parameter-less hBOA starts with a single population of size μ_0, conventionally set to $\mu_0 = 10$. After two generations are performed, a new population of size $\mu_1 = 2\mu_0$ is created and performs a generation. Evolution then continues with the μ_0 population performing two generations for each one performed by μ_1. Each time population μ_i performs its second generation a new population $\mu_{i+1} = 2\mu_i$ is created, which performs generations

half as often as μ_i. In this way an infinite number of parallel population can be simulated, with each population receiving the same number of total evaluations.

In all other aspects each population is identical to an hBOA population using a fixed μ. No search information is shared among populations, and each search is independently terminated. As such Parameter-less hBOA cannot perform better than hBOA using the optimal population size for a given instance, as it must also spend evaluations on populations of different sizes. This inefficiency is bounded by a log multiple of the total number of evaluations [24].

2.5 Linkage Tree Genetic Algorithm

[33] introduced the Linkage Tree Genetic Algorithm (LTGA) which automatically detects and exploits problem epistasis by examining pairwise gene entropy. Due to its enhanced ability to preserve high fitness gene subsets, LTGA was able to outperform state-of-the-art GAs across many benchmarks. Since its introduction, many variants of LTGA have been proposed [34, 12] so for clarity we have chosen the version presented by [35] as our model.

LTGA's effectiveness comes from its method of performing crossover. Instead of blindly mixing genes between parents, LTGA attempts to preserve important interrelationships between genes. Before performing any crossovers in a generation, LTGA first builds a set of hierarchical gene clusters that are then used to dictate how genes are mixed during crossover.

Figure 2.2 provides the agglomerative method LTGA uses to create gene clusters. This algorithm creates a tree of sets using pairwise gene entropy, such that the leaves of the tree contain a single gene and internal nodes are the union of their children's sets. These sets are then used by crossover to specify epistatic relationships that should be preserved. The process begins by creating the set of sets *unmerged* that tracks all top-level clusters. Initially *unmerged* contains single member sets for each gene. After each iteration the two sets with the minimum average pairwise distance (given in Equation 2.2) are merged to create a single cluster. This process is repeated until only a single set remains in *unmerged* which contains

1: **procedure** CLUSTER-CREATION
2: $unmerged \leftarrow \{\{0\}, \{1\}, \{2\}, \ldots, \{N-1\}\}$
3: $useful \leftarrow unmerged$
4: **while** $|unmerged| > 1$ **do**
5: $C_i, C_j \leftarrow \min_{C_i, C_j \in unmerged} D(C_i, C_j)$
6: $unmerged \leftarrow unmerged - \{C_i, C_j\} + \{C_i \cup C_j\}$
7: $useful \leftarrow useful + \{C_i \cup C_j\}$
8: **if** $D(C_i, C_j) = 0$ **then**
9: $useful \leftarrow useful - \{C_i, C_j\}$
10: **end if**
11: **end while**
12: Order $useful$ based on last merged first
13: Remove largest cluster from $useful$
return $useful$
14: **end procedure**

Figure 2.2: Algorithm describing how LTGA creates clusters using Equation 2.2 for a population. $unmerged$ and $useful$ are ordered sets of sets of gene loci.

all of the genes in the genome.

$$D(C_i, C_j) = \frac{1}{|C_i| \cdot |C_j|} \sum_{c_i \in C_i} \sum_{c_j \in C_j} 2 - \frac{H(c_i) + H(c_j)}{H(c_i \cup c_j)} \tag{2.2}$$

$$H(c) = -\sum_{s \in S} p_c(s) \log(p_c(s)) \tag{2.3}$$

Throughout this process $useful$ tracks the set of all gene clusters that should be preserved for use by crossover. This set begins with all genes in separate clusters, and each time a new cluster is created it is added to $useful$. However, not all clusters are necessarily worth keeping. For instance, in all versions of LTGA the cluster containing all genes is removed from $useful$ as preserving all genes during crossover can only create clones. [35] extended this removal to include any unsupported subsets. If the pairwise distance between two clusters is 0, this means there are no individuals in the population that disrupt the relationships between the two clusters. Therefore, when performing crossover, there is no reason to believe a fitness improvement can be achieved by breaking the stored pattern. As such a cluster is only kept if its direct superset has a non-zero distance. As a final step, Line 12 reorders $useful$ such that clusters appear in reversed order from which they were added to $useful$.

```
1: procedure CLUSTER-USAGE
2:     for all C_i ∈ useful do
3:         d ← rand_choice(P)
4:         Copy d's gene values for C_i into solution
5:         if solution was changed then
6:             if solution's fitness decreased then
7:                 Revert changes
8:             end if
9:         end if
10:    end for
11: end procedure
```

Figure 2.3: Algorithm describing how clusters are used to perform crossover.

Thus the most linked clusters and those containing single genes appear at the end of the returned list.

[35]'s version of LTGA does not use the entire population when determining pairwise entropy. Instead, binary tournament is used to select half of the population. This is done to ensure the model is built using only high-quality solutions, even during the first generation.

In order to efficiently perform clustering, a pairwise gene frequency table is constructed from the selected solutions. To calculate Equation 2.2, Equation 2.3 is called for each gene ($H(c_i)$) and pair of genes ($H(c_i \cup c_j)$). Extracting this information requires $\mathcal{O}(\mu N^2)$ time, where μ is the population size and N is the genome size. The process of converting this pairwise frequency information into clusters can be achieved in $\mathcal{O}(N^2)$ using the bookkeeping methods presented by [14]. This cost is performed only once per generation, and is then used to perform approximately $\mathcal{O}(\mu N)$ crossover evaluations. As a result, the amortized cost of LTGA's model building is $\mathcal{O}(N)$.

Figure 2.3 describes how the identified clusters are used by crossover to preserve gene linkage while still exploring the search space. Unlike more traditional crossover methods, LTGA crosses each individual with the entire population. Also, to produce a single offspring, multiple evaluations of the fitness function are performed.

During each generation, every individual in the population undergoes crossover. In a single crossover event, each cluster of genes C_i in $useful$ is applied as a crossover mask. A

random donor d is chosen from the entire population (not just the model selected population), and d's gene's for C_i are copied into the working solution. If a modification is made, an evaluation is then performed. If the crossover resulted in no worse fitness then the changes are kept, which allows for neutral drift across plateaus. The resulting solution, which must be at least as fit as its parent, is then copied into the next generation.

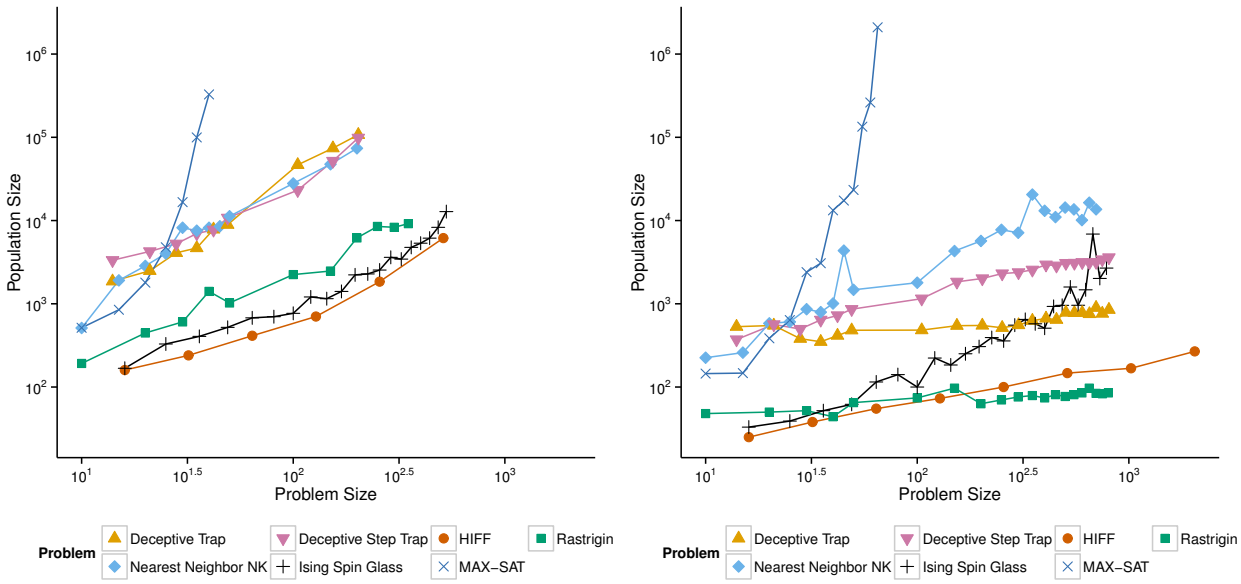
In total each individual can cause up to $|useful|$ evaluations. If all clusters were kept, even those deemed unhelpful, and all donations were evaluated, even those that did not change any genes, then Cluster-Usage would perform exactly $2N-2$ evaluations for each of the μ solutions in the population. This provides the amortizing evaluations required to make clustering only $\mathcal{O}(N)$ operations per evaluation. However, by skipping some evaluations, it is possible that clustering may be super-linear.

LTGA has no explicit form of diversity control and has no method for introducing new genetic information once the population has converged. Therefore an LTGA run is considered converged when two consecutive populations contain the same unique solutions.

By design, LTGA only has a single parameter: population size. LTGA uses no mutation, and crossover is defined in terms of the clustering algorithm. Selection between generations is fully elitist and embedded in the crossover, with selection of model building solutions fixed to a binary tournament. Neither Cluster-Creation nor Cluster-Usage rely on parameter values. LTGA does not provide any method for controlling or setting the population size, relying instead on a fixed user-specified size.

2.6 Comparison Algorithm Parameter Tuning

While four of the six algorithms in our experiments do not require any user-specified parameters, hBOA and LTGA both use a population size parameter. To ensure these techniques are not unfairly handicapped, we extensively tuned each using the bisection method [29] to determine the optimal population size for each problem size. Extended by [12], this method iteratively doubles the population size until some success criteria is met and then



(a) Tuned Population Sizes for hBOA

(b) Tuned Population Sizes for LTGA

Figure 2.4: Optimal population sizes found using bisection on each size for each problem.

performs bisection between the smallest successful and the largest unsuccessful sizes. In this way the minimum population size that meets the success criteria is found. [9] proposed a success criteria of performing r successful runs in a row, such that the expected failure rate can be bounded above by $\frac{3}{r+1}$ [17]. As P3 and the other three algorithms do not prematurely converge, we chose $r = 100$ to similarly ensure hBOA and LTGA almost never do. As bisection can make infinitely large population sizes, any run that had not found the global optimum after 100 million evaluations or 128 computing hours was considered unsuccessful.

Figure 2.4 shows the results from performing bisection on all problems to be used as comparison benchmarks in Chapter 5. In general hBOA required population sizes that were at least an order of magnitude larger than LTGA. Due to runtime and memory overhead, finding the optimal value for hBOA was much less tractable than for LTGA on moderate to large problem sizes. LTGA's population size also grew significantly slower than hBOA's as problem size increased, especially on the two Trap problems and Rastrigin. Both algorithms were ineffective on MAX-SAT, with neither able to tune to problems sizes over 60 bits. This

is likely due to the fact that some randomly generated MAX-SAT landscapes are quite flat and highly deceptive [26].

While not currently treated as a parameter, we also performed preliminary tests of integrating hill climbing into LTGA and hBOA as this is the procedure used in P3. To match P3, we applied first-improvement hill climbing to each algorithm's initial population. We then performed bisection on the modified algorithms for the largest problem sizes where hBOA without hill climbing was effective. We found that in general both methods performed worse when combined with hill climbing, in some cases up to an order of magnitude worse. There were three exceptions: both improved on MAX-SAT and hBOA improved on Rastrigin. In all cases the inclusion of hill climbing did not result in either algorithm outperforming P3 in terms of evaluations required to reach the global optimum. As such, all further experiments use the unmodified, published versions.

Chapter 3

Parameter-less Population Pyramid

[9] introduced the Parameter-less Population Pyramid (P3) as a method for performing optimization that does not require the user to provide any parameters. This is achieved by combining efficient local search with the model building methods of LTGA using an iteratively constructed hierarchy of populations.

The high level algorithm of P3 is presented in Figure 3.1. Unlike more traditional GAs, P3 does not follow a generational model. Instead, it maintains an iteratively expanding pyramid of expanding populations. Each iteration, a new random solution is generated. This solution is brought to a local optimum using the hill climbing algorithm in Figure 2.1. If that local optimum has not yet been added to any level of the pyramid, the solution is added to the lowest population P_0.

Next, the solution is iteratively improved by applying LTGA's crossover algorithm (Figure 2.3) with each population P_i in the pyramid. If this process results in a strict fitness improvement and has created a solution not yet stored in the pyramid, that new solution is added to the next highest pyramid level P_{i+1}. If P_{i+1} does not yet exist, it is created. In this way populations in the pyramid expand over time, and the number of populations stored increases over time. Initially the pyramid contains no solutions or populations, meaning the user does not need to specify a population size.

To accommodate P3's unique population structure, some of LTGA's clustering procedures were modified. In LTGA, clusters are identified at the start of each generation and are

```
1: procedure ITERATE-P3
2:     Create random solution
3:     Apply hill climber
4:     if solution ∉ hashset then
5:         Add solution to P_0 and rebuild P_0's model
6:         Add solution to hashset
7:     end if
8:     for all P_i ∈ pyramid do
9:         Mix solution with P_i
10:        if solution's fitness has improved then
11:            if solution ∉ hashset then
12:                Add solution to P_{i+1} and rebuild P_{i+1}'s model
13:                Add solution to hashset
14:            end if
15:        end if
16:    end for
17: end procedure
```

Figure 3.1: One iteration of P3 optimization. *pyramid* is an ordered set of populations and *hashset* is a set of all solutions in *pyramid*.

used to create all offspring in that generation. As P3 does not perform serial generations, P3 instead rebuilds the model each time a solution is added to a population. Furthermore, unlike our chosen variant of LTGA, all solutions in the population are used to generate the model, not just the winners of a binary tournament. We can do this because even the worst solutions in the pyramid are already high quality due to the previous application of local search. Using local search in LTGA was examined by [3] and found to provide no significant improvement. A likely cause was that that study applied local search to every solution, not just the initial population, resulting in significant overhead.

Beyond the changes in population structuring, P3 modifies LTGA's version of CLUSTER-CREATION and CLUSTER-USAGE. P3 changes Line 12 in Figure 2.2 from *last merged first* ordering to *smallest first* ordering. This method applies gene clusters during crossover based on how many genes are in each cluster,[1] and not on how tightly linked those genes are. [12] found that this alternative was better at preserving diversity, and therefore required smaller populations.

[1] Ties are broken randomly.

P3 also modified Line 3 in Figure 2.3. Instead of choosing a single genetic donor for each cluster, P3 iterates over the population in a random order until a solution in the population is found that has a least one gene different for that cluster of genes from the improving solution. This process increases the likelihood of an evaluation being performed for every cluster, and helps test rare gene patterns in the population.

In LTGA the cost of rebuilding the model is $\mathcal{O}(\mu N^2)$ as it must collect pairwise gene frequency information for all μ solutions in the population. P3 does not store a single population, and it does not have a fixed μ size for any population. However, each time a solution is added to the population, it requires $\mathcal{O}(N^2)$ time to update the table of pairwise frequencies and another $\mathcal{O}(N^2)$ time to rebuild the linkage model. The model is then used immediately to perform up to one evaluation for each of the up to $2N - 2$ clusters. Just as in LTGA, if no evaluation shortcuts were made, P3 has an amortized cost of $\mathcal{O}(N)$ modeling cost per evaluation. While P3 does rebuild the model more frequently per solution in the population, it also performs a number of local search evaluations that are quite efficient, meaning theoretical comparisons of their speed are difficult to perform. As a final note, P3's repeated attempts to find a useful donation make it less likely than LTGA to skip evaluations, but has an added cost to find these donations. Repeated donations could require as much as $\mathcal{O}(\mu)$ attempts per evaluation, but experimental evidence suggest that this operation actually saves more overhead than it costs by increasing the number of evaluations per model rebuild.

Each of the pieces of the P3 algorithm were selected not just for their standalone efficacy, but for the ways in which they interact. By using the hill climber to optimize randomly generated solutions, the underlying pairwise relationships in the problem are exposed. As a result, detecting clusters for use by crossover is much more effective. The crossover operator is extremely elitist, as each gene donation must result in no fitness loss, and a solution must strictly improve to be added to the next level of the pyramid. This is balanced by continual integration of new randomly generated, then locally optimized, solutions. Furthermore, each random restart decreases the probability of spurious linkages caused by shared ancestry. This

diversity is further preserved by applying gene clusters in smallest first order during crossover as this reduces the probability of genetic hitchhikers.

Other algorithms have proposed using multiple concurrent populations. [16] had a hierarchy of populations with solutions periodically advancing upward. This allows for continuous integration of diversity as the lowest population is reseeded with random solutions. However, this method resulted in increased parameterization as not only was a population size required, but also new parameters for how frequently generations advanced between levels and how many total levels to have. [15] used multiple independent populations of different sizes as a method for removing the population size parameter, but doing was provably less efficient than using an optimal population size as no information is shared between the populations.

Chapter 4

Problem Descriptions

4.1 Single Instance Problems

Understanding how a stochastic search algorithm will behave on arbitrary and complex search landscapes can be exceedingly difficult. Therefore a common practice for algorithm understanding is to perform search on well defined, well understood landscapes. To be of interest these landscapes need to represent interesting and important aspects of real-world problems.

One such landscape is the Deceptive Trap problem [8]. In this landscape the genome is broken up into k bit non-overlapping subproblems referred to as *traps*. Each subproblem is scored using Equation 4.1, where t is the number of bits in the trap set to 1. The global optimum in each trap is a string of all 1s, while all other solutions lead to a local optima of all 0s. This problem tests an algorithm's ability to overcome k sized deception and is commonly used to determine how effective crossover is at preserving building blocks. Any crossover event that mixes bits from different parents in the same trap will likely result in that trap being optimized to the local optima. For our experiments we chose $k = 7$ to ensure highly deceptive traps.

$$trap(t) = \begin{cases} k - 1 - t, & t < k \\ k, & t = k \end{cases} \tag{4.1}$$

[12] found that mixing local search with linkage learning rendered the Deceptive Trap problem trivial. This is because local search is able to optimize each trap to one of the two optima (all 1s or all 0s), providing linkage learning with perfect knowledge of gene interactions. In order to make the problem more difficult, the authors proposed the Deceptive Step Trap problem, given in Equation 4.2. This function modifies the results of Equation 4.1 to include plateaus of size s, introducing an exponential number of local optima in each trap. With $k = 7$ and $s = 2$, as used in our experiments, all traps with 0, 1, 3, 5, and 7 bits set are local optima. This means that half of all ways to set the trap are 1 bit local optima. More generally, the number of local optima grows at $\Theta(2^{k-1})$. As a result, the Deceptive Step Trap is much more challenging for linkage learning techniques, while still being highly deceptive.

$$step\_trap(t) = \left\lfloor \frac{(k-s) \pmod{s} + trap(t)}{s} \right\rfloor \tag{4.2}$$

Another challenging aspect of landscapes can be higher-order relationships. The Hierarchical If and only If (HIFF) problem [36] is designed to capture the difficulties of this class of problem. In HIFF the genome is broken up into a complete binary tree, such that each gene appears in exactly one leaf and each internal node is the subset of genes contained in its children. If all genes represented in a node of the tree are set to the same value, they score equal to the size of the set. In this way small subsets lead toward solutions to larger subsets. However, a node can score if all genes are either all 1s or all 0s, meaning that to solve higher-order subproblems it is necessary to perform crossovers that preserve lower order solutions. This problem is a natural fit for LTGA as the linkage tree can perfectly duplicate the problem's true relationships [35].

As a final class of well known problems, we have chosen to borrow the Rastrigin problem from real valued optimization. This problem's landscape, determined by Equation 4.3, is highly multimodal caused by the oscillating cosine function. [9] proposed the Discretized Rastrigin problem, such that each floating point x_i in Equation 4.3 is encoded using a 10 bit

gray code.

$$10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right] \forall x \in [-5.12, 5.12] \tag{4.3}$$

4.2 Randomly Generated Problem Classes

While well defined landscapes can provide specific insights into how an algorithm works, their static nature can be misleading. Specifically, algorithm quality might be so fragile that it is only effective at searching well behaved landscapes. A more realistic test of an algorithm's black-box effectiveness is to work with randomly generated instances drawn from a problem class. When tested over a sufficiently large sample it is then possible to draw more general conclusions about an algorithm's effectiveness. The challenge with these landscapes is determining the global optimum to gauge if an algorithm was successful.

Perhaps the most common model for generating random rugged landscapes is the NK model. An NK Landscape determines the fitness of each gene based on epistatic relationships with K other genes in the genome. This fitness is specified using a randomly generated table of fitness values, were each possible combination of the $K + 1$ genes is mapped to some floating point value $[0 - 1]$. In unrestricted NK landscapes the relationships between genes are also randomly chosen and as a result finding the global optimum is NP-Hard for $K > 1$. However, if epistasis is set such that each gene depends on the K directly following it in the genome, the solution can be found in polynomial time [39]. These Nearest Neighbor NK landscapes are therefore ideal for search algorithm testing. For all of our experiments using Nearest Neighbor NK we fixed $K = 5$ to ensure highly rugged landscapes.

[30] presents a combinatorial benchmark problem derived from physics: Ising Spin Glasses. A spin glass is defined by a weighted graph of interaction terms between vertices. Each gene assigns a value to each vertex, with the fitness calculated by Equation 4.4. In this equation, E is the set of all edges, e_{ij} is the edge weight connecting vertex i to vertex j, and x_i and x_j are the gene values for vertex i and j. Optimal fitness is when this sum is

minimized.

$$\sum_{e_{ij} \in E} x_i e_{ij} x_j \tag{4.4}$$

Similar to NK Landscapes, the general class is NP-Hard to optimize, but the $2D \pm J$ subset of Ising Spin Glasses can be polynomially solved.[1] In this subset the graph is restricted to be a two-dimensional torus, edge weights are randomly set to either -1 or 1, and vertex values must be -1 or 1.

As our final class of randomly generated problems we chose the Maximum Satisfiability (MAX-SAT) problem. Related to the more common 3-SAT problem, a MAX-SAT instance is defined by a set of three–term clauses. Each term is a randomly chosen variable, which may also be negated. A clause scores if an and only if at least one term in the clause evaluates to true. In order to make MAX-SAT instances with a known global optimum, [9] proposed constructing clauses around a fixed solution. In this way the signs of the terms are set to ensure the target solution satisfies the clause. To ensure each problem is challenging we chose a clause-to-variable ratio of 4.27 [31].

[1] `http://www.informatik.uni-koeln.de/spinglass/`

Chapter 5

Benchmarking P3

5.1 Finding the Global Optimum

Figure 5.1 shows the median number of evaluations required by each of the six algorithms to find the global optimum for multiple sizes of each problem. Each data point in Figure 5.1 represents the median of 100 runs, where unsuccessful runs are treated as requiring more evaluations than any successful run. If the median run was not successful no point is shown. Medians are used as the data is not normally distributed, and because it allows for more meaningful comparison between techniques with different success rates. For LTGA, the maximum problem size used for each problem was set to be the largest, optimal problem size we could feasibly determine. For HBOA, results on many large problems are not shown due to the extreme computational cost required to optimally determine the population size.

5.1.1 Quantitative Comparison

Of the 130 tested configurations, P3 found the global optimum using the least median evaluations on 114. The largest problem size for any problem where P3 was not the most efficient has 49 bits, with P3 achieving the best results on all 92 larger configurations. hBOA, LTGA, and Parameter-less hBOA only outperform P3 on the smallest 5, 4, and 1 Deceptive Step Trap instances, respectively. Random Restart Hill Climbing outperforms P3 on the smallest 3 Nearest Neighbor NK instances and the smallest Ising Spin Glass. $(1 + (\lambda, \lambda))$ has

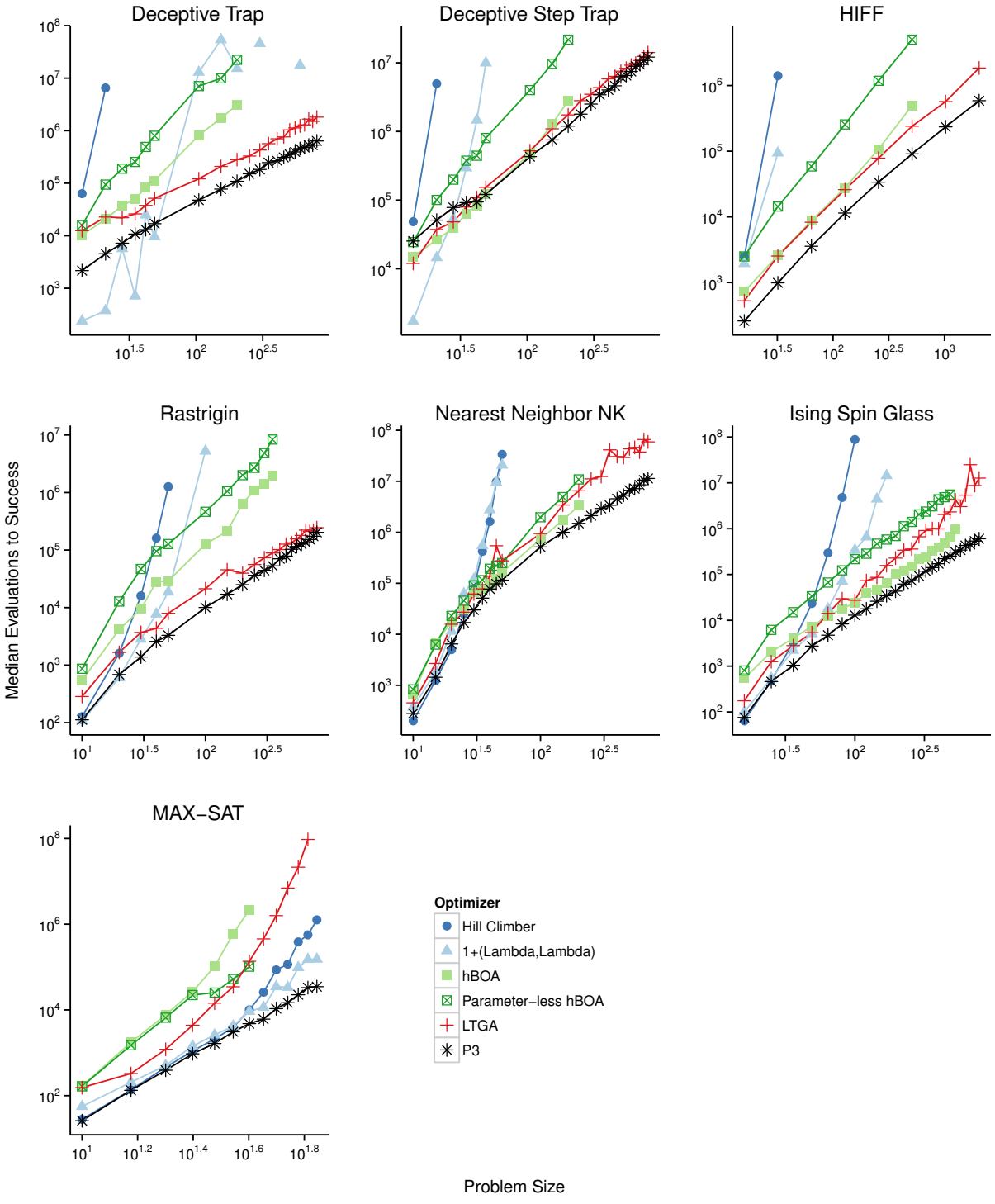


Figure 5.1: Comparison of the median number of evaluations to reach the global optimum for the six different optimization methods with respect to problem size. If the median run did not reach the global optimum no data element is shown. Results given on a log-log scale.

the most success outperforming P3, doing so on the smallest 5 Deceptive Traps, 3 smallest Deceptive Step Traps, and 2 smallest Rastrigin. The likelihood that P3 would achieve these pairwise results assuming its median result is actually worse is $p < 10^{-15}$ according to the binomial test. Pairwise comparison of LTGA and P3 on the largest problem size using the Mann-Whitney U test results in $p < 10^{-5}$ for all problems.

5.1.2 Local Search

The Random Restart Hill Climber and $(1 + (\lambda, \lambda))$ are both relatively effective on small problem sizes. This is especially true for the three randomly generated problem classes. These problems may contain relatively few local optima or just be exceptionally difficult for the model-based algorithms. On Deceptive Trap and Deceptive Step Trap using 4 or fewer traps, $(1 + (\lambda, \lambda))$ performs significantly better than any other algorithm. We believe this is because $(1 + (\lambda, \lambda))$ is able to overcome deception by probabilistically flipping entire traps. This ability also leads $(1 + (\lambda, \lambda))$ to outperform the Random Restart Hill Climber on all problems except Nearest Neighbor NK.

On larger problem sizes, the ability for local search to reach the global optimum quickly diminishes. Only on MAX-SAT are these optimizers competitive at larger tested problem sizes. However, we believe this is because the largest tested MAX-SAT was an order of magnitude smaller than the largest size tested for most other problems. As the problem size increases the number of local optima increases exponentially, which explains why Random Restart Hill Climbing was unable to scale. For larger problems it also becomes increasingly unlikely for $(1+(\lambda, \lambda))$ to make the right combination of changes required to reach the global optimum. This behavior causes high variance in success rate, as evident by the occasional successes on large Deceptive Trap problems.

5.1.3 Model Building

Only techniques that explicitly built models of gene epistasis were able to solve the largest problem instances. On single-instance problems LTGA was more effective than hBOA, with hBOA outperforming LTGA on Nearest Neighbor NK and Ising Spin Glasses. This may be caused by the differences in modeling method: unlike the single-instance problems, gene epistasis in the randomly generated problem classes cannot be be perfectly represented with a linkage tree.

Considering how different hBOA and LTGA are in performing optimization, it is somewhat surprising how similar their results are on HIFF. However, both techniques rely on populations large enough to support the diversity required to reach the global optimum and to model epistasis. Both techniques also only rebuild models once per generation. As the subproblems of HIFF are nested, it is unlikely that either technique can accurately model higher-order epistasis before solving lower order subproblems. Therefore both methods require one generation per subproblem order.

5.1.4 P3

Unlike the other model-based methods, P3 generally outperforms both Random Restart Hill Climber and $(1 + (\lambda, \lambda))$ even on small problem sizes. Unlike the other local search methods, P3 outperforms LTGA and hBOA even on large problem sizes. This implies that P3 is gaining the benefits of each, leveraging local search to solve easy problems and model building to solve harder ones.

Furthermore, the interaction between these two optimization tools explains some of the reason P3 outperforms each method alone. On Deceptive Trap P3's use of hill climbing ensures all traps are immediately optimized, allowing for perfect linkage detection and high-quality donation. On HIFF local search solves all pairwise subproblems, saving P3 a generation over LTGA and hBOA. In comparison P3 is only a slight improvement on Deceptive Step Trap, which is less amenable to local search.

5.2 Fitness Over Time

For some applications, finding the global optimum is less important than finding good solutions quickly. Therefore we examine this behavior in Figure 5.2. At regular intervals during optimization Figure 5.2 shows the median of the best fitnesses found at that time point of search across 100 runs. Figure 5.2 shows the largest problem size for which we were able to successfully gather results for all six algorithms, but the trends shown are representative of all larger problem sizes. The maximum reporting interval is set to include the slowest P3 run to reach the global optimum.

5.2.1 Quantitative Comparison

Of 181 sample points, P3 had the highest median fitness in 121. In pairwise competition, $(1 + (\lambda, \lambda))$ was the most likely to outperform P3, doing so on 50 sample points. LTGA, hBOA, and Parameter-less hBOA were the next best, outperforming P3 on 27, 20, and 18 sample points, respectively. Random Restart Hill Climbing almost never outperformed P3, doing so only 9 times. The likelihood that P3 would achieve these pairwise results assuming its median result is actually worse is $p < 10^{-9}$ according to the binomial test.

5.2.2 Local Search

Perhaps the most striking result is the quality of $(1 + (\lambda, \lambda))$. Until quite far into search this method performs better than both LTGA and hBOA. Given sufficient evaluations $(1 + (\lambda, \lambda))$ also outperforms Random Restart Hill Climbing on all 7 problems. For brief periods in the middle of search it performs the best of all techniques on: Deceptive Trap; Deceptive Step Trap; HIFF; Ising Spin Glass; and MAX-SAT problems. $(1 + (\lambda, \lambda))$'s ability to efficiently incorporate gene modifications of larger than one bit allows it to overcome the deception and plateaus in Deceptive Trap and Deceptive Step Trap, solve medium-sized subproblems in HIFF, flip the signs on multiple adjacent bits in Ising Spin Glass, and cross plateaus in MAX-SAT. However, this method is slow in reaching the global optima in many of

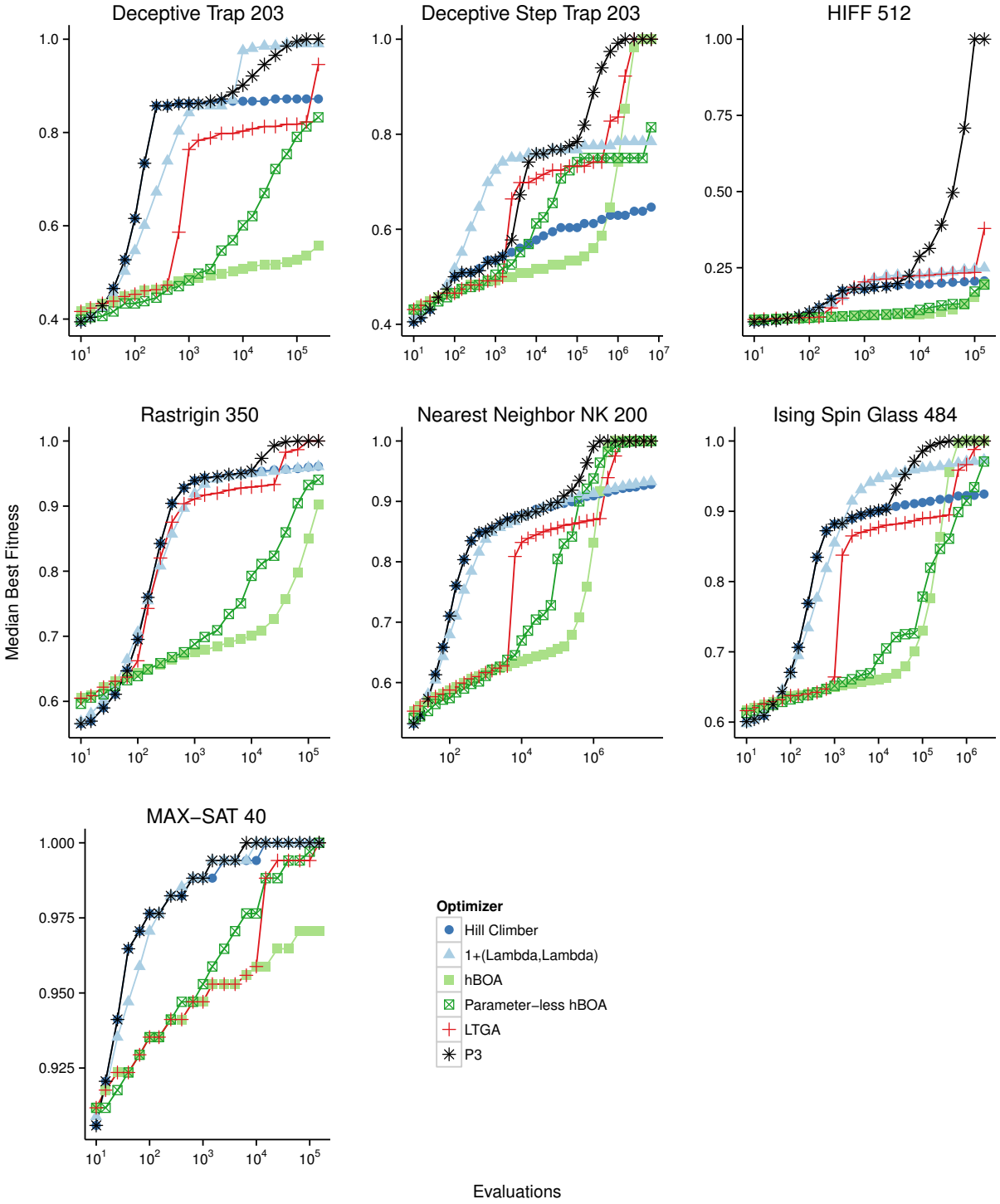


Figure 5.2: Compares the median best fitness reached during search for each of the six optimization methods.

these problems which causes it to eventually be overtaken by the model building techniques.

5.2.3 Model Building

Both hBOA and LTGA are marked by periods of little improvement followed by rapid improvement. In hBOA this is taken to the extreme, with all fitness improvement made at the very end of search. In both cases this is caused by model building. Before the model is accurate little improvement is made. Once it is accurate, fitness improves dramatically.

At 58% of the recording intervals hBOA has the worst fitness of any solver. Most of the exceptions occur when hBOA is still evaluating its initial population, allowing this random search to temporarily surpass the local search methods. After N evaluations, however, hBOA and LTGA both fall behind until their models begin to improve. Parameter-less hBOA reaches intermediate fitnesses faster than hBOA, doing so on 62% of intervals, as its models begin to optimize earlier than hBOA. However, this trend is reversed after a sufficient number of evaluations, most clearly on Deceptive Step Trap and Ising Spin Glasses, as hBOA's tuned population overtakes Parameter-less hBOA's parallel populations.

On every problem LTGA has five distinct periods: fitness plateau, near instantaneous improvement, fitness plateau, and improvement to global optimum. The early period corresponds with initialization of the population, with the first fitness gain achieved immediately upon completing the first generation. When using an inaccurate model, LTGA's mixing strategy performs a sort of less effective local search. Subsequent generations then make only minor fitness improvements. Once the model becomes accurate and the probability of a crossover using high-quality genetic material increases sufficiently, LTGA enters a second period of rapid improvement.

5.2.4 P3

The integration of hill climbing into P3 makes it strictly better than using hill climbing alone. Early in optimization P3 and the Random Restart Hill Climber have effectively

identical quality. This is because P3 performs the same evaluations as the Hill Climber for the first two restarts. Once P3 begins performing crossover it immediately improves over the Hill Climber. In 95% of intervals P3 had a fitness at least as high as Hill Climbing. As such P3 is better than a simple hill climber regardless of how long each technique is run and irrespective of how high quality the solution found has to be.

Unlike the model-based methods, which struggle until model accuracy improves, P3's iterative solution integration allows it to improve much more quickly. This behavior exists in most problems, but is easiest to understand on Deceptive Trap. On this problem, P3 immediately brings all traps to local optima, equaled only by the Random Restart Hill Climber in quality. In comparison LTGA must evaluate the entire population and perform multiple generations to reach similar quality. P3 is able to immediately integrate optimal versions of each trap into a single individual as they are found by local search, resulting in smoother fitness improvement than LTGA.

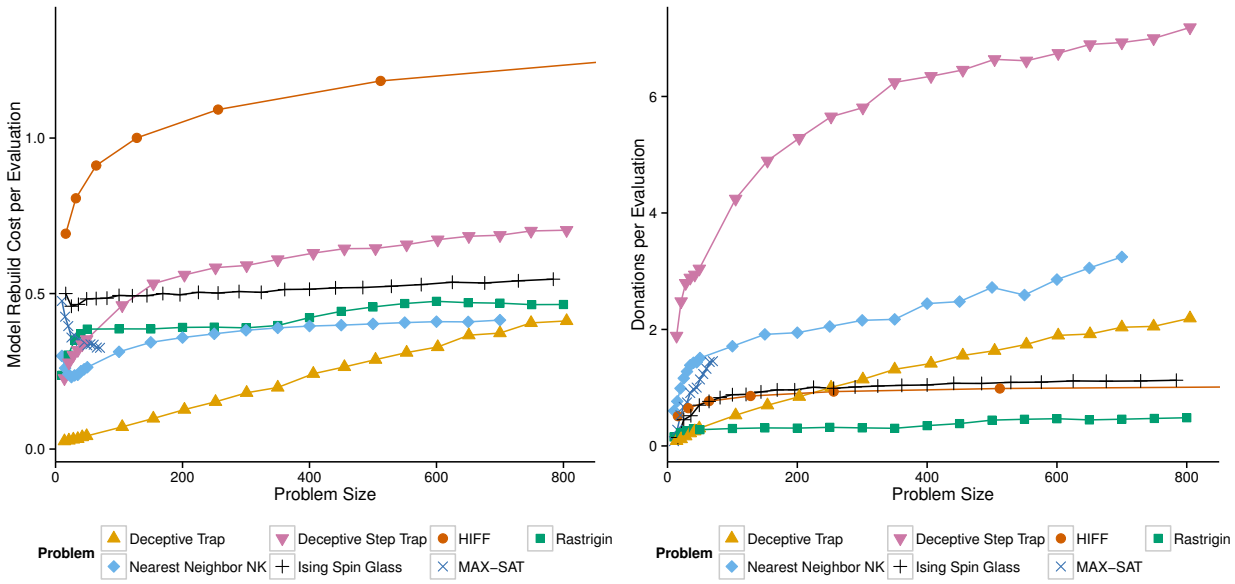
5.3 Computational Expenses

While it is common in evolutionary computation to assume the evaluation function will dominate algorithm complexity, in some domains this will not be true. Model-based methods are especially likely to violate this norm. Therefore, in order to assess P3's quality in solving problems with efficient fitness functions, we provide data on both its algorithmic complexity and wall clock time.

5.3.1 Operation Counting

When discussing the asymptotic complexity of P3 in Chapter 3, two aspects eluded precise analysis: how expensive is model rebuilding and how many gene donations are made. Figure 5.3 provides some insight into how often these two aspects of the algorithm are utilized.

Figure 5.3a reports in an algorithmic sense how expensive model rebuilding is for search.



(a) Rebuilds

(b) Donations

Figure 5.3: Estimated computation costs incurred by model rebuilding (Figure 5.3a) and repeated donations (Figure 5.3b) per evaluation as problem size increases.

In order to calculate this value we recorded how many times search rebuilt the model during each run. Figure 5.3a shows the estimated ratio of model rebuilding cost (N^2 per rebuild) over evaluation cost (N per evaluation). If the cost of model building scaled linearly with evaluations, the relationship plotted for each problem should be asymptotically constant. For Nearest Neighbor NK, Ising Spin Glasses, and Rastrigin this is the case. For both Trap problems and HIFF there is slow growth in the ratio. The problem sizes used for MAX-SAT were not sufficient to accurately gauge the asymptotic behavior. Together this suggests that while the cost of building the model is almost linear per evaluation, it can grow slowly. However, even in the worst case (HIFF) this growth was no more than twice the algorithmic cost of an evaluation even using 2048 bits.

When applying a crossover subset, P3 tries random donors from the population until one is found with at least one bit different from the improving solution. In theory this can result in up to $\mathcal{O}(\mu)$ operations. Figure 5.3b examines the observed average number of donations

per evaluation performed. Ising Spin Glass, HIFF, and Rastrigin all achieve effectively constant behavior here, implying repeated donation does not impact the asymptotic runtime of P3. Both Trap functions and Nearest Neighbor NK all increase in number of donations as problem size increases, potentially increasing algorithmic costs. An important note is that each donation may range in size from a single bit up to $N - 1$. However, repeated donation attempts are far more likely to happen with smaller clusters. As such this may cause some super-linear growth in P3, but it is unlikely to be very high.

5.3.2 Wall Clock Performance

To assess wall clock performance we provide Figure 5.4. Similar to Figure 5.1, each point represents the median of 100 runs, with unsuccessful runs treated as slower than successful runs. These results were collected using 2.5GHz Intel Xeon E5-2670v2 processors.

5.3.2.1 Model Building

hBOA and Parameter-less hBOA perform much worse when using wall clock time as the unit of comparison than when using evaluations. This makes sense as hBOA's model building requires $\Omega(N^2)$ time per evaluation while, under reasonable assumptions, P3 and LTGA require $\mathcal{O}(N)$ time per evaluation. This penalty is most clear on Ising Spin Glass where hBOA goes from being slightly more efficient than LTGA in terms of evaluations to three orders of magnitude worse in terms of seconds. As P3 and LTGA require a similar asymptotic complexity per evaluation as the Hill Climber and $(1 + (\lambda, \lambda))$, no similar change in ordering occurs.

5.3.2.2 P3

When LTGA is optimally tuned to a single-instance problem with an efficient evaluation function it can find the global optimum faster than P3 in terms of wall clock time. However, on randomly generated problem classes P3's efficient use of evaluations is enough to overtake LTGA.

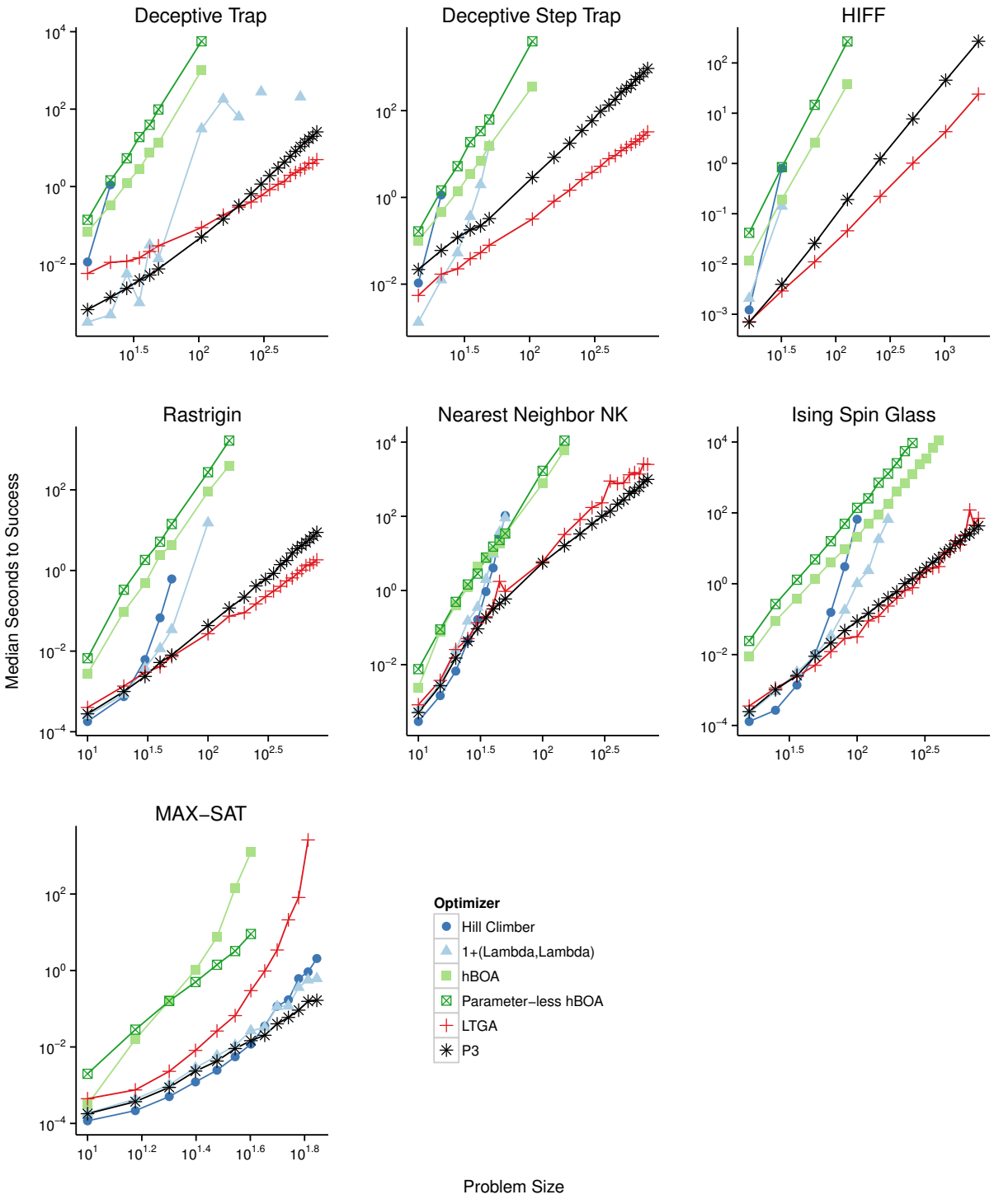


Figure 5.4: Comparison of the median number of seconds to reach the global optimum for the six different optimization methods with respect to problem size. If the median run did not reach the global optimum no data element is shown. Results given on a log-log scale.

On the four single-instance problems LTGA not only finds the global optimum using less wall clock time, the factor speedup increases as problem length does. Naïvely this suggests LTGA is achieving a lower order of complexity. However, for these experiments LTGA is growing at sub-linear time per evaluation, which is not asymptotically stable due to (at minimum) the time required to perform an evaluation. We suspect that the true cause is that N is small enough to be overshadowed by lower order polynomial terms. For example, LTGA requires $\mathcal{O}(N/\mu)$ time per evaluation to rebuild the linkage model from the frequency table. As a result, for small μ model building, and not extracting pairwise frequency, can dominate runtime.

When applied to randomly generated problem classes, the differences in P3 and LTGA's evaluation complexity dominates runtime complexity. Similar to with Figure 5.1, the amount of speedup P3 achieves over LTGA increases with problem size on Nearest Neighbor NK, Ising Spin Glasses, and MAX-SAT.

Across both types of problems we find that P3's time per evaluation grows approximately linearly. As such, we conclude that P3 requires asymptotically similar amounts of time per evaluation as the other efficient techniques.

Chapter 6

Internal Workings

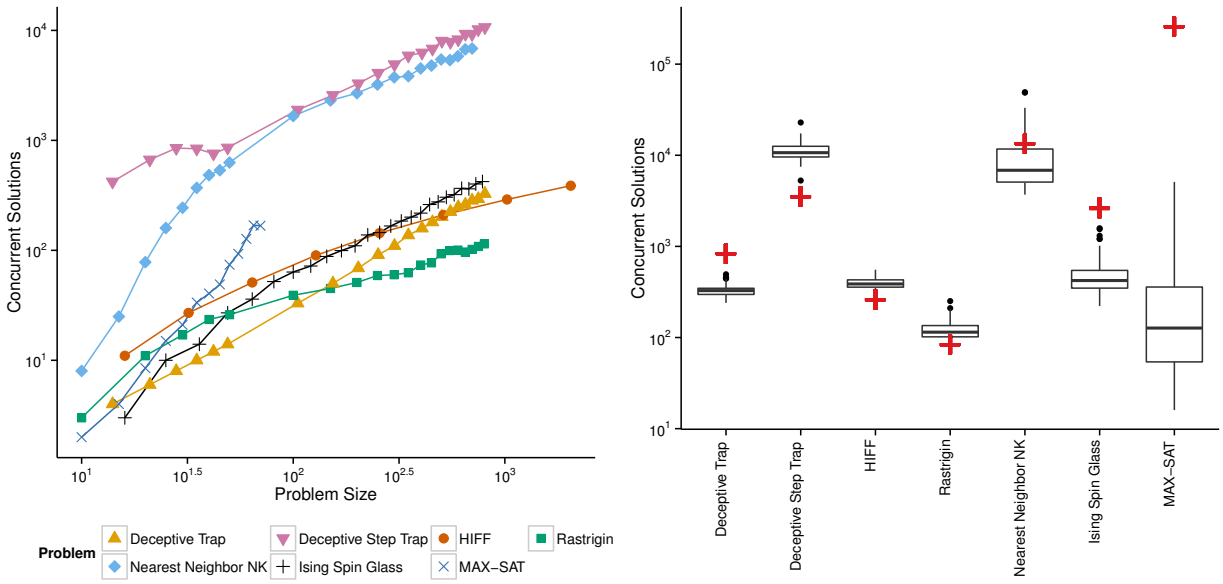
6.1 Population Sizing

A major advantage to P3 is that it does not require the user to set a population size parameter. Beyond making P3 easier to apply, this also conveys two additional advantages: diversity scaled to initialization and no need to sacrifice intermediate fitness for eventual optimality.

Figure 6.1a shows how the number of total solutions stored in the pyramid changes as problem size increases, similar to Figure 2.4 for hBOA and LTGA's tuned population sizes. As expected, the number of concurrently stored solutions increases as problem difficult increases, with the exact behavior dependent on the problem landscape. Figure 6.1b examines how the number of solutions stored is distributed on the largest problem sizes. Here we see that the behavior depends on the type of problem. On single-instance problems P3's stored variance is relatively low, and generally higher than optimally tuned LTGA's population size. On randomly generated problem classes P3 has a much higher variance in stored solutions, but in general requires smaller sizes than LTGA.

6.1.1 Problem Instance versus Problem Class

Our procedure for tuning LTGA and hBOA outlined in Section 2.6 involved finding the optimal population size for each class of problem. For real-world black-box optimization this



(a) As problem size increases

(b) Largest problem size

Figure 6.1: The total number of solutions stored by P3 when the global optimum is found. In Figure 6.1b the red "+" indicates LTGA's tuned population size.

is realistically the best either algorithm could hope for as tuning to a problem instance or population initialization involves repeatedly solving the problem being tuned. This limitation does not exist in parameter-less methods, which scale their diversity based on the problem instance without needing to solve that instance repeatedly.

To achieve high success rates on randomly generated problem classes, LTGA and hBOA must use a population which is large enough to solve the hardest instances in that class. Therefore these methods will have population sizes larger than necessary to solve easier instances in the class. Even on single-instance problems, both methods will require population sizes large enough to ensure the worst random initialization is diverse enough to solve the problem, which may be much larger than the best random initialization.

Figure 6.2 highlights how this can effect the required number of evaluations to reach the global optimum, showing the distribution of results when solving the largest size of each problem. On each problem except Ising Spin Glass and MAX-SAT, LTGA has a much

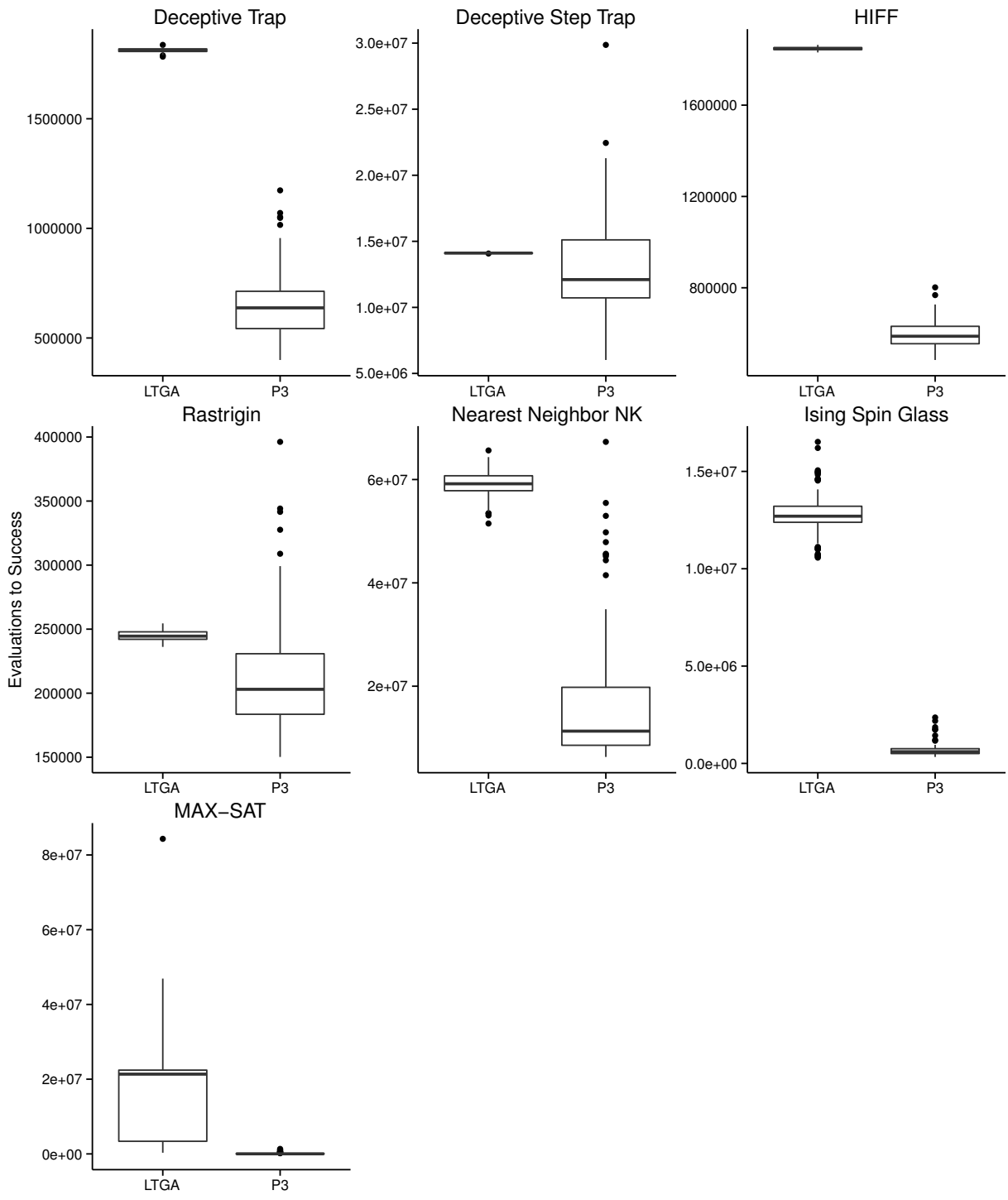


Figure 6.2: Distribution of evaluations required to reach the global optimum for P3 and LTGA on the largest size of each problem.

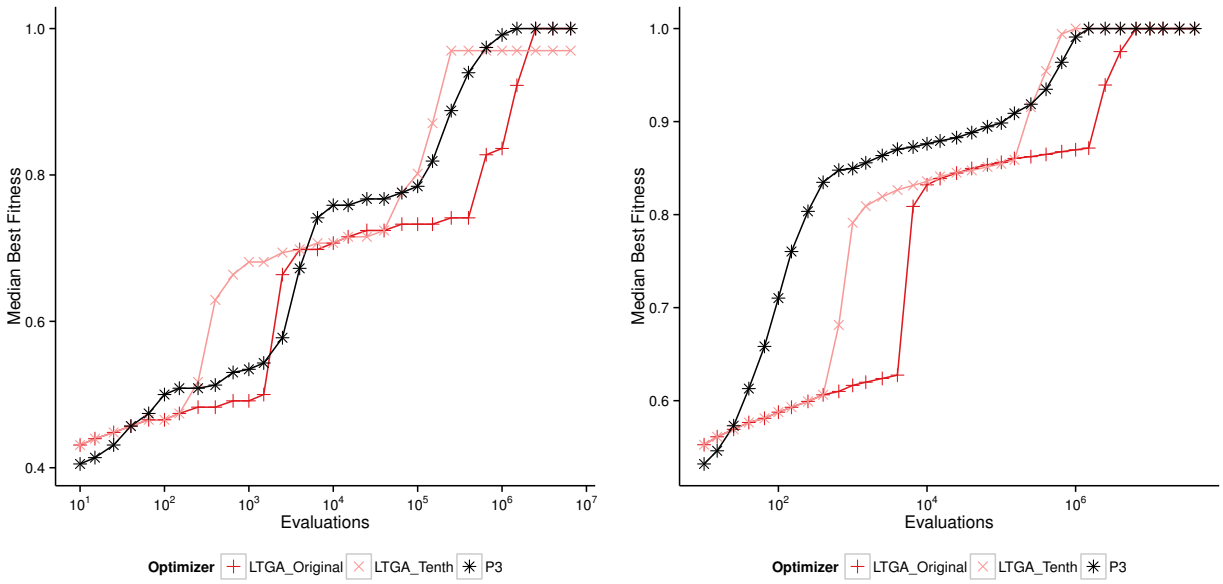
smaller difference between its best and worst runs. This makes sense as LTGA uses the same population size regardless of instance and because its search progresses generationally. In contrast P3 has a much higher split, with many runs finishing very quickly. On all problems except Deceptive Step Trap, P3's upper quartile is lower than LTGA's lower quartile. Furthermore, on Deceptive Trap, HIFF, and Ising Spin Glasses, P3's worst run is better than LTGA's best run. For Nearest Neighbor NK, most of P3's runs finish much faster than the fastest LTGA runs. However, some of P3's outliers take approximately as long as LTGA's tuned performance. This supports the hypothesis that P3 is able to scale its diversity not just to the problem class, but to the problem instance or even problem initialization, something wholly infeasible for tuned population sizing to do.

This tuning distinction is also apparent when comparing Parameter-less hBOA with hBOA in Figure 5.1. While generally performing worse than hBOA, the difference between the two algorithms is smallest on randomly generated problem classes. On MAX-SAT, Parameter-less hBOA actually outperformed both hBOA and LTGA, likely due to its ability to scale diversity to the problem instance instead of the entire problem class.

6.1.2 Fast versus Optimal

In Section 5.2 we examined intermediate fitness qualities of LTGA and hBOA when using population sizes tuned to reach the global optimum. As a result, both were exceptionally ineffective at quickly reaching high-quality solutions. This is because unlike P3, these methods have an explicit trade off between optimal performance and intermediate performance caused by their population size parameter.

Figure 6.3 examines the effect of population size on LTGA's intermediate fitness by reducing LTGA's population size to one tenth of the tuned value. The two problems shown are representative of the behavior of using a smaller population size on the other five problems. Reducing the population size caused LTGA to improve earlier but plateau at lower fitnesses. This caused LTGA's success rate to drop from 100 to 0 on Deceptive Step Trap and from 98



(a) Deceptive Step Trap 203

(b) Nearest Neighbor NK 200

Figure 6.3: Comparison of how reducing LTGA's population size affects the median best fitness reached during search.

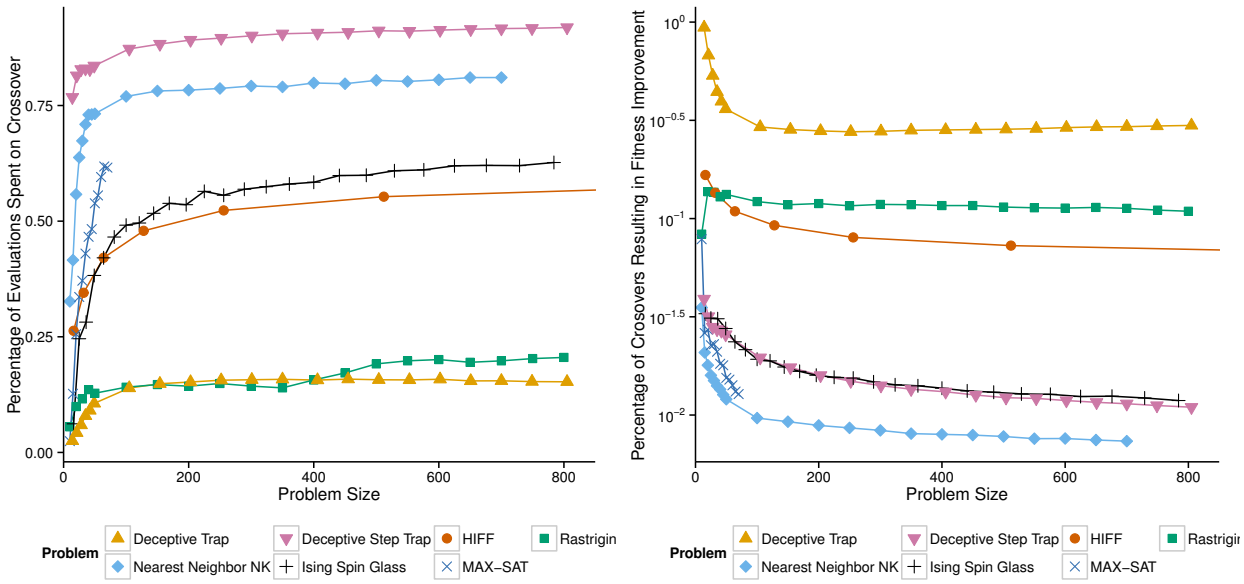
to 68 on Nearest Neighbor NK. Even when using a reduced population size for LTGA, P3 still achieved a fitness at least as high as LTGA at 80% of intervals. The likelihood that P3 would achieve these pairwise results assuming its median result is actually worse is $p < 10^{-15}$ according to the binomial test.

6.2 Inner Workings Specific to P3

While analysis of optimization speed is useful from a practitioner standpoint, doing so provides little insight into algorithm behavior. To better understand how P3 works in detail we present here a look at some internal features specific to P3.

6.2.1 Crossover

Figure 6.4a shows the proportion of evaluations P3 spends on crossover, as opposed to hill climbing, and Figure 6.4b shows what percentage of crossover evaluations resulted in a



(a) Crossover Proportion

(b) Crossover Success

Figure 6.4: For each problem Figure 6.4a shows the proportion of P3 evaluations spend on crossovers and Figure 6.4b shows the percentage of fitness-improving crossover evaluations.

fitness improvement. Together these figures provide some insight into the role of crossover within P3. The behavior for each is clearly problem dependent and generally asymptotically stable as problem size increases.

When solving problems where epistasis can be effectively detected and represented by a linkage tree, P3 tends to spend fewer evaluations performing crossover and each crossover is more likely to be successful. Deceptive Trap and Rastrigin are the easiest problems to capture epistasis, with local search quickly reducing pairwise entropy in each. These are also the problems where P3 uses the least evaluations on crossover and has the highest success rates for crossover. At the other extreme are Nearest Neighbor NK and Ising Spin Glasses, which both have overlapping linkage not representable by a linkage tree. These problems have the highest crossover usage and lowest crossover success of any problem except Deceptive Step Trap. While Deceptive Step Trap's epistasis can be accurately modeled by a linkage tree, the exponential number of plateaus makes detecting gene linkage very challenging.

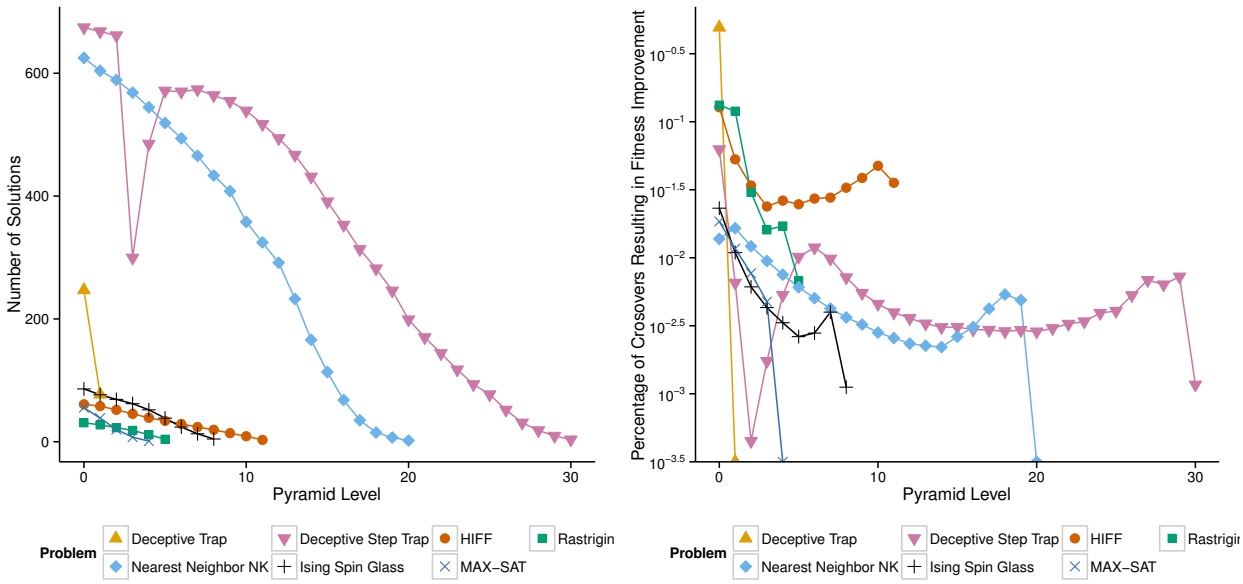
P3's crossover success rates are lower than LTGA's, but both produce the same ordering of the problems when using crossover success such that Nearest Neighbor NK is the least successful and Deceptive Trap is the most successful problem. Counterintuitively, the use of hill climbing on the initial population reduces P3's crossover success, not because it reduces model quality or the donation pool, but because it is much more challenging to improve locally optimal solutions than randomly generated ones. LTGA's crossover benefits from application to unoptimized solutions, which makes its aggregate crossover success incomparable to P3's.

Even when crossover success rates are quite low, such as Nearest Neighbor NK's 0.007% success, the results from Section 5.1 and Section 5.2 show how critical this small percentage is to optimization. Without crossover, P3's performance would be identical to the Random Restart Hill Climber, which was unable to solve even moderately-sized problems and quickly fell behind P3 in intermediate fitness quality. Therefore even infrequently successful crossover donations are critical to success. This does, however, suggest a potential avenue for future improvement by using more successful modeling and donation algorithms.

6.2.2 Pyramid

Another feature unique to P3 is the shape and size of the population pyramid constructed for each problem. Figure 6.5a shows the number of solutions stored at each level of the pyramid for the largest tested problem sizes. Each point is the median size across 100 runs. If a run did not store any solutions at a level it is treated as 0. No point is drawn if the median run had 0 solutions stored at that level. While pyramid size is affected by problem size, the overall shape is not. As such the behavior shown in Figure 6.5a is representative of that for all other tested problem sizes.

With the exception of the dip in Deceptive Step Trap, all of the pyramids show a monotonic reduction in size as the level increases. This is because a solution must be a strict fitness improvement over its previous version to be added to a higher level, which becomes



(a) Population Size

(b) Crossover Success

Figure 6.5: For each problem Figure 6.5a shows the number of solutions stored in each level of the pyramid and Figure 6.5b shows the percentage of fitness-improving crossover evaluations at each level.

less likely each time the solution improves. [20] found theoretical evidence and [11] found empirical evidence that the optimal population size decreases each generation of traditional evolutionary search. By decreasing in size, P3 implicitly stores more diversity in low levels and focuses search around high-quality solutions at higher levels. In comparison, LTGA and hBOA suboptimally use a fixed population size at each generation.

Figure 6.5b examines how crossover success changes at different levels of the pyramid. At low levels, success gets progressively lower as solution quality increases faster than the model's ability to improve solutions. At higher levels modeling becomes more accurate and donations contain higher frequencies of high-quality building blocks, resulting in increased crossover success. The highest level of most problems has a low crossover success rate, as solutions crossing with that level have already been improved by previous operations to the point where the only improvement would be to create the global optimum, which can only happen once.

6.2.2.1 Deceptive Step Trap

The number of solutions stored at the fourth level of Deceptive Step Trap is significantly lower than that of the third or fifth levels, breaking the decreasing trend of the other six problems. Figure 6.5b has a similar aberration, with crossover success dropping to 0.0004% before rebounding and following the more common trajectory. This behavior exists in all other problem sizes tested, with the dips occurring at exactly the same level.

This behavior is rooted in the peculiar nature of this landscape. After local search, all traps in all solutions have a total number of 1 bits equal to either 0, 1, 3, 5, or 7 as these correspond to the local optima when using $k = 7$ and $s = 2$. Crossover easily overcomes the two-bit plateaus, and as a result solutions in the second level generally do not contain the lowest fitness local optima (5 bits set) and the third level has few traps set to the next worst local optima (3 bits set). As a result, solutions that reach the third level can only be improved by replacing the deceptive local optima (0 and 1 bits set) with the global optimum (7 bits set). The global optimum is rare in the population, and with 8 ways to represent local optima linkage learning is inaccurate. Therefore it is unlikely for crossover to be successful, meaning few solutions will be added to the fourth level. Solutions that do improve by definition must have a higher frequency of optimal trap settings, meaning level four's model will be more accurate and donations are more likely to contain optimal trap values. Thus the level size and crossover success rates increase after contracting around level four.

Chapter 7

New Problem Domain: Gray-Box Optimization

All of the work presented so far has focused on the black-box optimization domain. These problems are characterized by a complete lack of available problem-specific information. A black-box algorithm can only propose a solution and measure the quality of that solution, using only that information to inform its search. Optimizers that are successful in this domain generalize well as there are minimal requirements to apply them to a new problem.

However, for many real-world applications there is potentially more information available. The other end of the spectrum is white-box optimization, in which the algorithm knows the exact problem class it is trying to solve. This domain is dominated by problem-specific search heuristics [6, 32, 39] that leverage all aspects of the problem to achieve efficiency. This makes such algorithms very specific, such that outside of their target domain they either cannot be applied or their application has no guarantee of search quality. As such, each must also be designed by hand for each new problem class, requiring deep understanding of the problem and the time to develop the algorithm.

In between those two domains there is another domain: gray-box optimization. In this domain some features general to multiple problem classes are exploited, beyond just an evaluation function. The goal is to create optimization methods that benefit from these

features without those methods becoming specialized to a small set of problem classes.

In addition to the black-box evaluation function, we add in two more exploitable features accessible to search algorithms: known variable epistasis and partial function evaluation.

7.1 Problems in this domain

To understand the types of problems in this domain, let us first examine a simple artificial problem we'll call 3-Equal. The quality of a solution is determined by how often three consecutive bits are set to the same value. As a result, the maximum fitness is N (all zero or all one) and the minimum fitness is 0.

The 3-Equal problem has both known epistasis and partial function evaluation. From the definition we know that each bit is epistatically linked to the two variables that precede it and the two that follow it. There are no other non-linear relationships with that bit. This also means it is possible to evaluate the fitness contribution of that bit only knowing the value of at most four others, regardless of the size of N. As we'll discuss in Section 7.3 this has enormous implications on search efficiency.

Every problem discussed in Chapter 4 except HIFF fits into this domain. Deceptive Trap and Deceptive Step Trap both can have known epistasis (which bits are in which traps) and partial reevaluation (score a single trap), the same as Rastrigin (separable). All NK problems where $K \ll N$ have a knowable epistasis table and the fitness contribution of each subfunction is calculable without evaluating the entire string. Similarly, MAX-SAT's clause list specifies epistasis, with each clause independently evaluable. Ising Spin Glasses are even simpler, with each edge in the graph evaluable using only two problem variables.

Beyond MAX-SAT and Ising Spin Glasses, which are interesting real-world problems unto themselves, many NP-Hard real-world problems can be expressed in these two requirements. This is especially true of graph problems.

- **Dominating set**: Find a minimum vertex set such that all vertices are either in the set or adjacent to something in the set. The fitness of a vertex is determined by if it is

in the set (add to set size) and if it is dominated (add to undominated size). Epistasis is the adjacency set for each vertex, plus itself. Used in wireless sensor networks [40].

- **Vertex Cover**: Find a minimum vertex set such that all edges are incident on at least one vertex in the set. Epistasis and fitness are calculated using each vertex (add to set size) and each edge (add to uncovered size), independently. Used in network security [28].

- **Max-Cut**: Find the set of vertices that maximize the number of edges incident on exactly one vertex in the set. Epistasis and fitness are calculated using each edge independently. Used in VLSI design [7].

- **Set Cover**: Given a universe of elements and a set of subsets of that universe, find a minimum set of subsets whose union recreates the entire universe. Each element in the universe is epistatically linked with subsets that contain that element (add to uncovered size) and each subset also contributes directly to fitness (add to set size). Used in computational biology [21].

- **Zero-One Linear Programming**: Given a set of constraints, maximize an objective function, all of which are linear combinations of variables. Each constraint creates an epistatic relationship between the variables in that constraint, and fitness is a linear combination of variables allowing for partial evaluation. Used in power systems [1].

7.2 Formal Requirements

In order to draw conclusions about search efficiency, it is necessary to make the features of the target problem domain more explicit.

The overall quality of a solution must be equal to the sum of applying all subfunctions to that solution, where a subfunction can be any mapping from a subset of problem variables to a measure of quality. As a consequence each subfunction must be independently evaluable.

The mapping of variables to subfunctions (epistasis) must also be known, and the maximum number of variables participating in each subfunction is represented by the variable k. To achieve maximum efficiency k should be constant in regards to problem size. The cost of evaluating a subfunction should also be bounded by some function $b(k)$. For example, on MAX-SAT k is the clause size, with $b(k)$ linear in clause size, and the fitness equal to the sum of all clause terms. On NK, $k = K + 1$ as each subfunction in NK depends on a variable and K others.

As the mapping of variables to subfunctions is known, it is also possible to calculate c, which is the maximum number of subfunctions any variable participates in. Algorithms are most efficient when c is constant with respect to problem size. For example for all Nearest Neighbor NK landscapes all variables participate in exactly $c = k = K + 1$ subfunctions. Randomly generated MAX-SAT instances have no guarantee that c is constant, but the expected number of subfunctions in which a variable appears is equal to the clause-to-variable ratio times the clause size, or 12.81. Assuming that all subfunctions use k variables and each variable appears in c subfunctions provides a bound on the total number of subfunctions $\frac{cN}{k}$.

7.3 Efficient Local Search

Section 2.1 presented an efficient method for performing local search for the black-box domain. Each potential bit flip, referred to here as a *move*, requires the entire solution to be reevaluated taking $\Omega(N)$ time. Furthermore, each time a fitness-improving move is made all previously tested moves must be tested again. As a result, improving a random solution to be a local optimum using this algorithm requires between $\Omega(N^2)$ and $\mathcal{O}(IN^2)$, where I is the number of improving moves. The lower bound is achieved when the while loop in Figure 2.1 runs a constant number of times. For instance, on One Max, Deceptive Trap, Deceptive Step Trap, and HIFF all possible single bit improvements are found during the first pass through the loop. The worst case is when each loop is expected to only make a single improving move, which causes all N moves to be reevaluated once per improvement,

```
1: procedure InitializeDelta(solution)
2:     fitness ← 0
3:     ∀_m delta[m] ← 0
4:     for all s ∈ subfunctions do
5:         pre_move ← f_s(solution)
6:         fitness ← fitness + pre_move
7:         for all m ∈ affected_moves(s) do
8:             post_move ← f_s(solution ⊕ m)
9:             delta[m] ← delta[m] + (post_move − pre_move)
10:        end for
11:    end for
12: end procedure
```

Figure 7.1: Algorithm used to efficiently determine the change in fitness associated with each potential move from a given solution.

such as on the Leading Ones problem.

Local search in the gray-box domain can be significantly more efficient [37]. This is due to two consequences of the domain: the fitness effect of making a move can be calculated in $\mathcal{O}(1)$ time, not $\mathcal{O}(N)$, and the number of moves that must be reevaluated per improvement is $\mathcal{O}(1)$, not $\mathcal{O}(N)$. This results in local search requiring $\mathcal{O}(N + I)$ time. If I is within a constant factor of N, this means generating random local optima is at most a constant factor slower than generating random solutions in the search space.

To achieve this performance, this local search technique begins by determining the change in fitness caused by making each potential move m, denoted as $delta[m]$. Figure 7.1 calculates the *delta* for each m starting at a given solution, as well as the fitness of the solution. Here, f_s evaluates the subfunction s on the given solution, and $solution \oplus m$ is the result of making move m on *solution*. For each of the $\frac{cN}{k}$ subfunctions InitializeDelta must determine the fitness of that subfunction before and after making each of k moves that overlap that subfunction. Combined, this results in less than $cN2b(k)$ operations, which is $\mathcal{O}(N)$ assuming c and $b(k)$ do not grow with N. Also, this procedure calls f_s only $k + 1$ times more than is required to find the fitness of the solution itself.

When performing hill climbing, only moves m such that $delta[m] > 0$ are fitness-

```
1: procedure MakeMove(m)
2:     fitness ← fitness + delta[m]
3:     for all s ∈ affected_subfunctions(m) do
4:         pre_both ← f_s(solution)
5:         just_m ← f_s(solution ⊕ m)
6:         for all m′ ∈ affected_moves(s) do
7:             just_m′ ← f_s(solution ⊕ m′)
8:             post_both ← f_s(solution ⊕ m ⊕ m′)
9:             delta[m′] ← delta[m′] − (just_m′ − pre_both) + (post_both − just_m)
10:            open(m′)
11:        end for
12:    end for
13:    close(m)
14:    solution ← solution ⊕ m
15: end procedure
```

Figure 7.2: Algorithm for updating stored information related to a solution when making a move.

improving moves. Initially all moves are considered *open*, meaning that they could potentially be fitness improvements. During each iteration a random move m is chosen from *open*, and $delta[m]$ is checked. If m is a fitness-improving move, MakeMove(m), shown in Figure 7.2, is called.

MakeMove updates the fitness and *delta* values to reflect the change in the solution. The fitness of the solution after making the move does not require any calls to f_s as $delta[m]$ already stores the change in fitness. This process requires updating all of the *delta* values for the k moves that interact with each of the c subfunctions affected by m. This update replaces outdated information that used the original solution ($just\_m' - pre\_both$) with how much the move improves over the new solution ($post\_both - just\_m$). As $delta[m']$ has updated, m' could potentially become a fitness-improving move and is therefore added into *open*. As m was just flipped, it cannot be a fitness improvement and is therefore removed from *open*. In total this requires less than $ck4b(k)$ time, which is $\mathcal{O}(1)$ assuming c, k, and $b(k)$ do not scale with N.

Each time an improving move is found at most ck additional moves are added to *open*. If *open* ever becomes empty a local optimum has been reached. Therefore the number of

times $delta[m]$ is checked is no more than $N + Ick$. As a result the total cost of improving a random solution until it reaches a global optimum is only $\mathcal{O}(N + I)$.

As an interesting addition, this method can actually be used to efficiently perform approximate steepest ascent hill climbing [37]. Instead of choosing moves from *open* at random, the moves are binned using their *delta* values. Moves are then chosen randomly from the highest quality non-empty bin, and changes in *delta* can cause moves to change bins. Assuming the total number of bins does not increase with N, both steps can be done in $\mathcal{O}(1)$ time. However, at least for MAX-SAT, there is evidence that when used in combination with subsequent search heuristics the less greedy, first-improvement algorithm was more effective.

7.4 Efficient Hamming Ball Search

Another consequence of the gray-box domain is that increasing the hamming-ball radius of local search becomes more tractable [4]. A hamming ball is the collection of all solutions within a given hamming distance, or radius, from a given solution. Instead of improving solutions until no single bit flip is a fitness improvement, the Hamming-Ball Hill Climber (HBHC) finds solutions which cannot be improved by flipping r or fewer bits.

In a black-box setting, verifying that no r-bit flip can improve a solution requires testing all $\binom{n}{r}$ neighbors. This quickly becomes intractable as r increases. However, in the gray-box domain not all combinations need to be tested. Consider that if two variables do not participate in the same subfunction, the relationship between their effects is, by definition, additive. As such there is no way for flipping both together to be a fitness improvement without flipping one of them being a fitness improvement. Therefore it is not necessary to try all possible r-sized subsets of the solution.

Consider the 3-Equal problem. If there are at least two loci between x_i and x_j then they do not share a common subfunction. Therefore, the change in fitness resulting from flipping both is equal to $delta[x_i] + delta[x_j]$. If $delta[x_i] \leq 0$ and $delta[x_j] \leq 0$ then $delta[x_i, x_j] \leq 0$. Now consider a solution such that the first half is set to 0 and the second half is set to 1.

```
1: procedure ConnectedInducedSubgraphs
2:     closed ← ∅
3:     found ← [ ]
4:     for all v ∈ V do
5:         closed ← closed ∪ {v}
6:         found ← found+CISG(v, ∅, closed, ∅)
7:     end for
8:     return found
9: end procedure
10: procedure CISG(v, subset, closed, open)
11:     subset′ ← subset ∪ {v}
12:     found ← [subset′]
13:     if |subset′| ≥ r then return found
14:     end if
15:     closed_here ← ∅
16:     open′ ← open ∪ adjacent(v)
17:     for all v′ ∈ open′ such that v′ ∉ closed do
18:         closed_here ← closed_here ∪ {v′}
19:         closed ← closed ∪ {v′}
20:         recurse ←CISG(v′, subset′, closed, open′)
21:         found ← found + recurse
22:     end for
23:     closed ← closed − closed_here
24:     return found
25: end procedure
```

Figure 7.3: Algorithm to recursively find all connected induced subgraphs of size r or fewer.

Even though $c = 3$ there is no way to improve this solution without simultaneously flipping all 0's to 1's or vice versa. Any smaller flip will not be a fitness improvement.

To determine which of the $\binom{n}{r}$ flips that must be checked, consider a graph where each vertex is a variable in the solution. An edge exists between two vertices if and only if the corresponding variables participate in at least one subfunction together. Restated, there is only an edge if the two variables have a direct, non-linear relationship. The maximum degree of a vertex in this graph is $c(k-1)$, making it sparse for sufficiently large N. If a subset of vertices is connected then it is possible that flipping all of those variables together will result in a fitness improvement even when flipping any subset of the subset will not. However, if the subset is not connected then each component of the subset can be tested independently.

In order to determine which moves in the hamming ball must be evaluated, we developed ConnectedInducedSubgraphs given in Figure 7.3. CISG is a recursive helper function that finds all subgraphs that contain a given *subset* and a given vertex v, while excluding any other vertices added to *closed*. To find all subgraphs, CISG is called once for each vertex in the graph, such that *closed* contains all previously searched vertices, and $subset = open = \emptyset$. In the initial call all desired subgraphs that contain v are found, which is why v remains in *closed* to prevent duplicate subgraphs from being returned.

At each recursive level CISG expands *open* to include any vertices adjacent to v in the graph. By construction this means that $open'$ contains all possible ways of adding a single vertex to the current $subset'$. As each v' is tested it is temporarily added to *closed* to prevent recursive calls from adding it again to $subset'$.

When applied to the sparse graphs inherent in the gray-box domain, this algorithm requires $\mathcal{O}(r!(ck)^r N)$ time, which reduces to $\mathcal{O}(N)$. The time spent in each call is dominated by the loop over $open'$. In the worst case, $open'$ increases in size by the full adjacency of v, which is bounded by ck. This creates a worse-case complexity for a single top-level call of $\prod_i^r ick = r!(ck)^r$. This must be called once for each of the N variables resulting in $\mathcal{O}(r!(ck)^r N)$.

As this algorithm finds all connected subsets in $\mathcal{O}(N)$ time for a fixed r, the number of moves that must be tested to determine if a solution is an r-bit local optimum is $\mathcal{O}(N)$. This means that while on Nearest Neighbor NK with $N = 8000$, $K = 5$, and $r = 3$ the black-box method would require 85 billion checks, the gray-box method requires only 248,000. Even when allowing connections to be completely random, which results in c increasing with N, the gray-box method still only requires approximately 375 million checks, two-and-a-half orders of magnitude less than assuming a black-box.

From these conclusions it is possible to modify the hill climber presented in Section 7.3 to efficiently find r-bit local optima [4]. The only change is that instead of having a move and *delta* for each bit, there must be a move and *delta* for each connected induced subgraph

```
1: procedure ITERATE-TUX
2:     Create random solution
3:     Hamming-Ball Hill Climb solution
4:     for all T_i ∈ T do
5:         if T_i is empty then
6:             T_i ← solution
7:             return
8:         end if
9:         Cross solution with T_i to create 2^{i+1} offspring
10:        Hamming-Ball Hill Climb each offspring
11:        solution ← best of offspring, solution, and T_i
12:        T_i ← empty
13:    end for
14:    Add solution to end of T
15: end procedure
```

Figure 7.4: One iteration of TUX optimization. T is an ordered list of solutions, each position of which could be empty, awaiting a crossover partner.

of r or fewer bits. The efficiency analysis is unchanged with the exception that the constant increases exponentially with r as the number of *delta* values that must be updated each move increases. Still, with c, r, and k constant with respect to N, it is possible to find r-bit hamming ball local optima in $\mathcal{O}(N + I)$ time.

7.5 Tournament Uniform Crossover: TUX

Hamming-Ball Hill Climbing is not sufficient to efficiently find the global optimum on problems with even moderate epistasis [4]. This is because, like all random restart hill climbers, it relies on random initialization to fall inside the global optimum's basin of attraction.

To remedy this limitation, we set out to develop a minimally complex memetic algorithm to help increase this probability. Figure 7.4 presents the Tournament Uniform Crossover (TUX) algorithm, which combines simplistic selection with equal probability uniform crossover, the most basic unbiased crossover, to generate starting solutions likely to be in the global optima's basin of attraction.

Conceptually TUX iteratively builds a structure similar to a single-elimination bracket for solutions. Each "match" in the tournament takes in two candidate solutions, produces offspring via uniform crossover, applies hill climbing to each offspring, with the "winner" being the best of all those solutions. The tournament "bracket" is constructed iteratively, storing an initially empty list of solutions T, such that $|T|$ is equal to the height of the tournament. Iterative construction is possible because, when a sub-bracket is complete, only a single solution emerges and solutions only need to be stored until their partner is found. TUX is fully elitist but does not prematurely converge. This is because search continuously integrates new randomly generated solutions through other parts of the bracket. Whenever the top of the current bracket is reached, TUX doubles the size of the virtual tournament.

When crossing solutions at T_i, TUX produces 2^{i+1} offspring. This relationship ensures that in total all levels of the tournament, including random initialization, perform the same number of hill climbing steps. It also shifts the focus of search toward areas expected to be of higher fitness. The expectation is also that it becomes progressively harder to improve solutions the higher up the tournament you advance, so more attempts are necessary to create new useful solutions.

The primary advantages of TUX are that it does not introduce any new parameters (though it still requires an r for the hill climber) and is relatively simple to implement. Even so it allows for learning from previous local optima and, as Section 9 will show, it is quite effective at optimization.

Chapter 8

Gray-Box P3

P3 represents a natural method for integrating the HBHC into a global optimization algorithm as P3 already utilizes local search. While HBHC's inclusion adds a parameter, Section 9.1 and Section 9.2 provide evidence that r can be fixed to 1, preserving the parameter-less nature of P3. Beyond using HBHC, there are also a number of ways in which P3 can be made more efficient by leveraging the additional information available in the gray-box domain.

First and foremost, linkage learning is no longer necessary. By definition the direct non-linear relationships between variables are known. As a result, Gray-Box P3 does not need to store the pairwise frequency information, reducing its required memory from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$. Instead, we have developed a method for creating a linkage tree that learns clusters from the same dependency graph defined in Section 7.4. The goal is for each cluster to be a connected induced subgraph, with the size of clusters mirroring those produced by the agglomerative linkage learning process normally used with P3. To form a single cluster, a random graph search is performed from a random starting vertex until a desired number of unique vertices have been explored. Cluster sizes are set recursively. For each cluster of size $l > 1$ a cluster of size a and a cluster of size $l - a$ are also created, with a chosen uniformly from the range $[1..l-1]$. This recursive process begins by initializing $l = N$ and splitting to form the first two clusters.

This linking algorithm has a number of useful properties. First, it creates exactly $2N-2$

clusters, distributed in size similarly to the black-box clustering algorithm. Performing a random graph search to find l unique vertices requires $\mathcal{O}(lck)$ time, meaning cluster creation is optimally efficient. The cluster splitting process has identical properties to random pivot quicksort, meaning the sum of cluster sizes is $\mathcal{O}(N \log N)$ in the average case. This efficiency allows new clusters to be created before every mixing event, unlike Black-Box P3 where they are created only when new solutions are added to the population. For simplicity the clusters are shuffled after they are created, not sorted on size like in Black-Box P3.

Beyond efficiency, there are good reasons to believe these clusters will be useful for search. The closer two variables are in the dependency graph, the more likely they are to appear in the same cluster. All variables on average are expected to appear in at least one cluster, but variables that are central in the graph will appear in more clusters than those on the periphery. The more paths of a given length between two variables, the higher the probability of them being in the same cluster. Unlike Black-Box P3, this linkage tree does not require clusters to be nested, allowing more diversity in the types of clusters appearing in a single tree.

In effect the clusters are sampling moves that the HBHC would make if $r \geq l$. For any solution in the search space that is not globally optimal, there must exist some move that will improve its quality. However, this move may be arbitrarily large and it is intractable to test all possible moves of even moderate size. By sampling from all possible large moves, we can maintain tractability while gaining a potentially non-zero probability of improvement. As clusters are used to donate values between solutions, these moves are always in the direction of previously found high-quality solutions. This assumes that the density of high-quality solutions is higher than average between good solutions.

Another efficiency gain is that each time a donation is made, only the affected part of the solution's fitness needs to be recalculated. As a result, the number of subfunction evaluations required to determine the change in fitness is only $\mathcal{O}(l)$. This also allows for Gray-Box P3 to efficiently reapply hill climbing after each donation as only affected moves need to be

rechecked. As a result, a donation and its resulting modifications from hill climbing are kept only if the new local optimum is at least as fit as the solution before the donation occurred. All combined, a single donation plus returning to a local optimum requires $\mathcal{O}(l + I)$ time, while just the donation in Black-Box P3 requires $\mathcal{O}(N)$.

Chapter 9

Benchmarking Gray-Box P3

To compare these gray-box optimization techniques we have chosen NKq-Landscapes [4] and Ising Spin Glasses [30]. NKq-Landscapes create a collection of randomly generated problem instances given a higher-level problem class description. Each instance is described by a series of N subfunctions, each corresponding to a variable in the solution. This subfunction uses its variable and K other variables in the solution to calculate a fitness value. Fitness values are represented as a randomly generated lookup table, such that table entries are integers in the range $[0..q-1]$. As each subfunction reads $K+1$ variables, the table's size and q are set to 2^{K+1}. The quality of a solution is equal to the sum of the values returned by these subfunctions.

In this work we consider two methods for choosing the K variables each subfunction depends on: Nearest Neighbor NKq and Unrestricted NKq. In Nearest Neighbor NKq each variable depends on the K variables that sequentially follow it in the solution, with dependencies wrapping around the end of the solution. Landscapes of this form can be solved in polynomial time [39], allowing comparisons of how quickly each optimization algorithm can find a global optima. Nearest Neighbor NKq also ensures that $c = k = K+1$ and that both c and k do not increase as N increases, meaning the efficiency conclusions made in Section 7.4 are applicable.

Unrestricted NKq landscapes draw the K dependencies at random without replacement. For $K > 1$ it is NP-Hard to find the global optimum of these landscapes. This also means

that while k remains fixed, the maximum number of subfunctions a variable appears in (c) can increase as N increases. As a result, some of the efficiency claims in Section 7.4 may not be applicable.

Ising Spin Glasses are a type of MAX-CUT problem relevant in statistical physics. Each spin glass encodes spins (vertices) and their relationships (edges) with the goal of assigning each spin a direction that minimizes relationship energy. Just as with NKq-Landscapes, Ising Spin Glasses as a whole are NP-Hard, but the $2D \pm J$ subset is polynomially solvable [30][1]. In this subset the graph is a 2D toroidal grid, with edge weights of ± 1. In gray-box terms, problems of this subset have $k = 2$ and $c = 4$ regardless of N.

For each problem class tested, we generated 50 instances. Extreme problem sizes were chosen to stress each algorithm. Each method was run once on each instance, and limited to 3 hours of computation and 4 GB of memory. Runs were performed on 2.5GHz Intel Xeon E5-2670v2 processors using the C++11 code available from our website.[2] Each time the run achieved a new best fitness we recorded the current amount of processing time used. Timing includes the discovery of subgraphs to allow for comparison between different radius values. When reporting the "best" fitness for an instance we mean the best fitness found by any method before the time limit is reached. On all Nearest Neighbor NKq instances the "best" fitness is also the global optimum, verified using dynamic programming. For Ising Spin Glass the "best" fitness was the global optimum 44 out of 50 times.

All figures report the median, upper and lower quartiles for either percentage error or seconds to reach the best. A run's percentage error is equal to the difference between its fitness and the best, divided by the best. When reporting seconds to reach the best fitness, any run that did not find the best fitness is treated as slower than any run that did. If the median run was unsuccessful, no data point is drawn.

[1] `http://www.informatik.uni-koeln.de/spinglass/`
[2] `https://github.com/brianwgoldman/GrayBoxOptimization/releases`

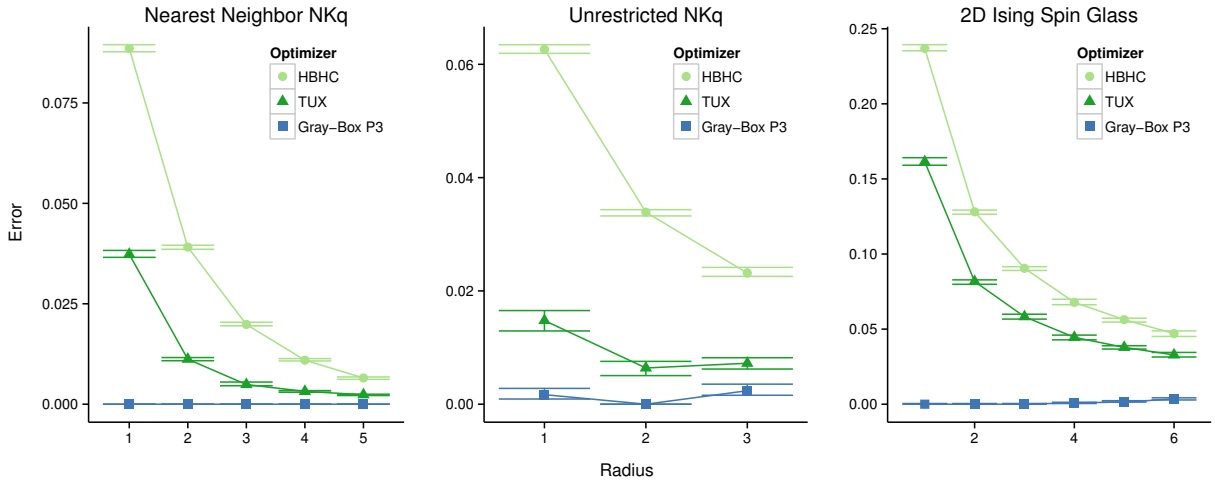


Figure 9.1: Comparison of how radius affects solution quality at termination. For NKq-Landscapes $N = 6,000$ and $K = 4$ and for Ising Spin Glasses $N = 6,084$. Range of radius values limited by memory constraints.

9.1 The Effect of Radius

The only algorithm parameter in the HBHC, TUX, and Gray-Box P3 is the radius of the hamming ball. Therefore our first experiments are designed to determine the effect of this parameter on solution quality.

Figure 9.1 shows the effect on final solution fitness as r increases. As expected from [4], the HBHC obtains higher quality as r increases, with the magnitude of the improvement decreasing. TUX has a similar relationship and outperforms HBHC on all three problems for all r values. Regardless of r, Gray-Box P3 outperforms both, with almost all r values reaching the same best fitness. For Nearest Neighbor NKq, Gray-Box P3 finds the global optimum in every run for $r < 4$, with only a single unsuccessful run at $r = 4$.

To further examine the effect of r on Gray-Box-P3, Figure 9.2 shows the number of seconds required to reach the global optimum. Setting $r = 1$ was the most efficient configuration for all $K > 1$, supporting the trend that Gray-Box P3 works best with small r values. With $K = 1$, the landscape is smooth enough that with a sufficiently high r the HBHC is able to find the global optimum.

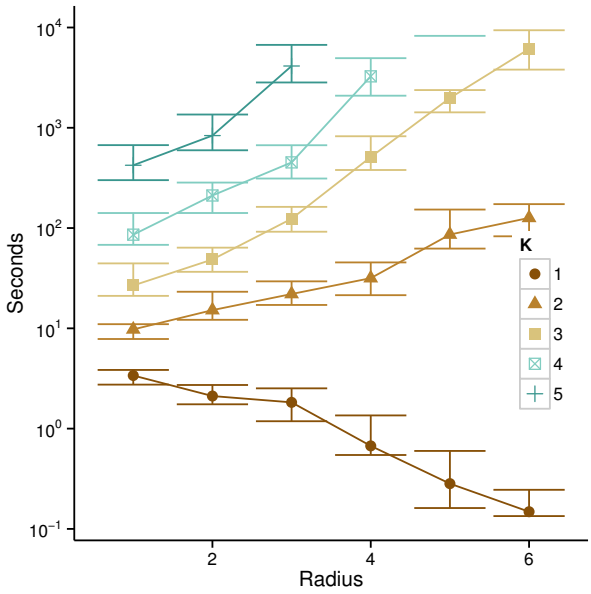


Figure 9.2: Time required for Gray-Box P3 to reach the global optimum of Nearest Neighbor NKq instances with $N = 6,000$.

9.2 Fitness Over Time

Reaching high-quality solutions quickly can sometimes be more important than reaching the global optimum eventually. Figure 9.3 shows how solution quality progresses for each algorithm. HBHC and TUX have significant early delays caused by their high r values. Larger r's require a large amount of initial partial evaluation before performing hill climbing. After one full iteration HBHC effectively stalls, with TUX continuing to improve. Both are eclipsed by Gray-Box P3, which quickly descends to the global optimum, outperforming HBHC and TUX at every time point.

Figure 9.4 further illustrates the effect of r on Gray-Box P3. On both Nearest Neighbor NKq and Ising Spin Glasses, increasing r does not change the shape of the curve. Instead, the quality reached is simply time shifted, such that given more time higher r values will reach the same quality. As a result, for these problems we conclude that higher r values simply add more expense for no overall gain. On Unrestricted NKq this relationship is less certain, with $r = 1$ potentially having a different, and worse, shape than $r > 1$. However,

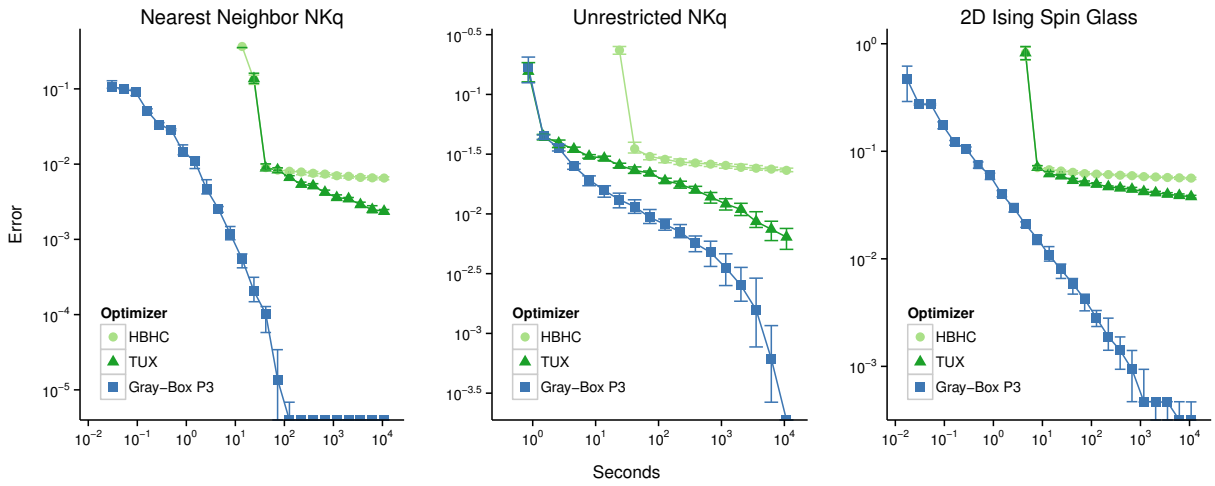


Figure 9.3: Comparison of solution quality during optimization on a log-log scale for different algorithms. For NKq-Landscapes $N = 6,000$ and $K = 4$ and for Ising Spin Glasses $N = 6,084$. Each algorithm uses its best-found r value.

due to memory and time restrictions it is difficult to know if this trend continues.

9.3 Scalability

Perhaps the most critical test of an optimization algorithm's quality is how it scales as problem difficulty increases. To test this behavior, we ran all three algorithms using the optimal r values determined experimentally in Section 9.1, varying N from 200 to 10,000 for NKq and 196 to 6,084 for Ising Spin Glass. In these plots we also include the black-box version of P3 to show the efficiency gains available for using gray-box information.

Figure 9.5 and Figure 9.6 show how long each algorithm required to reach the best overall quality found on Nearest Neighbor NKq and Ising Spin Glasses, respectively. For Nearest Neighbor NKq the best found is the global optimum for all runs of all lengths, while for Ising Spin Glasses the best quality found by any method was worse than the global optimum in 7 runs of $N = 4,096$ and 21 runs of $N = 6,084$. The median run of the HBHC was unable to reach the best fitness for any problems tested using more than 200 bits. TUX performed somewhat better, reaching the best fitness more than half of the time on problem sizes up to $N = 800$ and $N = 625$ for Nearest Neighbor NKq and Ising Spin Glass, respectively. Black-

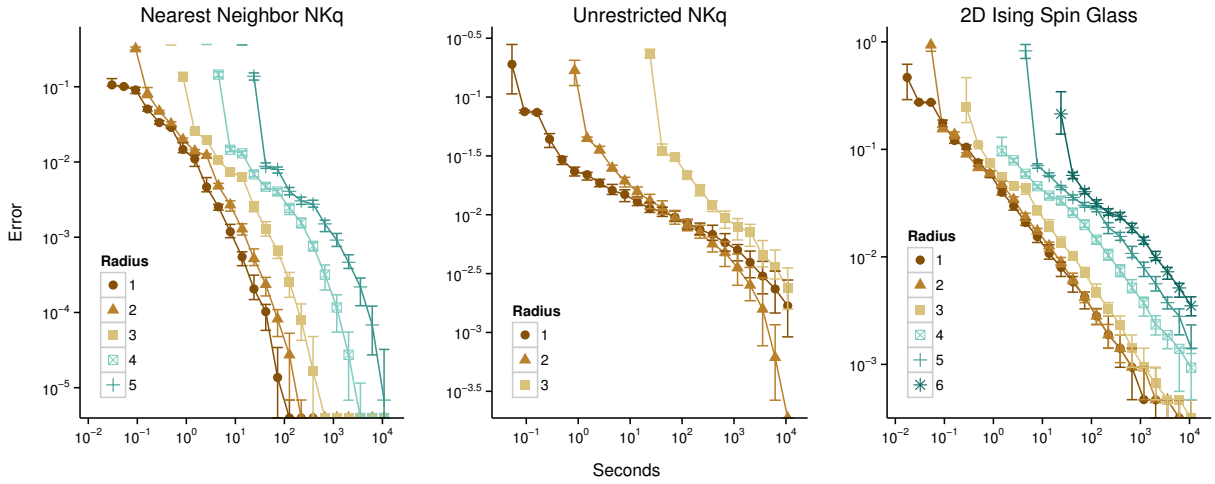


Figure 9.4: Comparison of Gray-Box P3's solution quality during optimization on a log-log scale for different r values. For NKq-Landscapes $N = 6,000$ and $K = 4$ and for Ising Spin Glasses $N = 6,084$.

Box P3, which does not utilize partial reevaluation or the HBHC, was able to consistently reach the best fitness until it it hit the memory limit on $N = 2,000$ for NKq and $N = 2,916$ for Ising Spin Glass. This limitation is due to Black-Box P3's $\mathcal{O}(N^2)$ memory requirements.

For all sizes of both problems, Gray-Box P3 was the fastest to reach the best fitness. On Nearest Neighbor NKq the improvement is substantial, with no alternative finishing within two orders of magnitude. On its largest successful problem size, the mean time to completion for Black-Box P3 was 375 times slower than Gray-Box P3. Using the Mann-Whitney test to compare their run times results in $p < 10^{-16}$. This is especially impressive considering previous work has shown Black-Box P3 is faster to reach the global optimum than other leading black-box methods [10]. Applying regression, we estimate that Black-Box P3's time to global optimum on Nearest Neighbor NKq is $\mathcal{O}(N^{2.75})$ while Gray-Box P3's is $\mathcal{O}(N^{1.98})$.

The results on Ising Spin Glass are similar, with a less extreme difference between Black-Box P3 and Gray-Box P3. In general, Gray-Box is the fastest technique to find the global optimum by an order of magnitude, with Black-Box P3's mean run finishing 4.6 times slower than Gray-Box on $N = 2,025$. Using the Mann-Whitney test to compare their run times results in $p < 10^{-14}$. The regression line suggests that while Gray-Box P3 scales at $\mathcal{O}(N^{3.35})$,

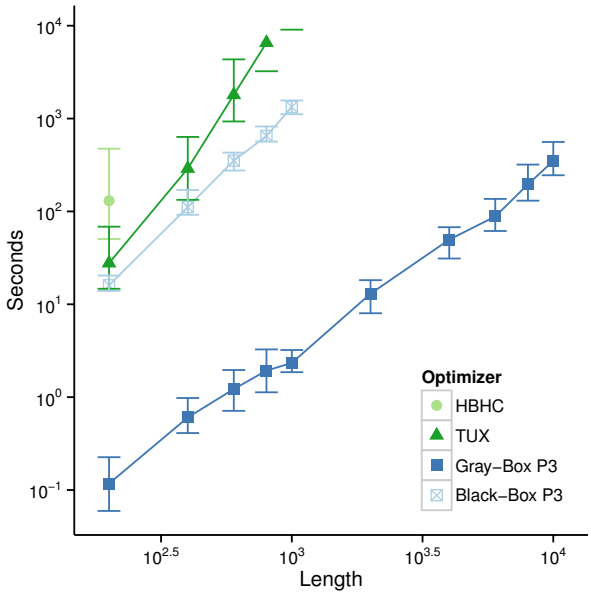


Figure 9.5: Comparison of how each algorithm's time required to reach the best fitness found scales with problem size on Nearest Neighbor NKq with $K = 4$. With $N = 1000$ Gray-Box P3 is 375x faster than Black-Box P3. HBHC was only successful on the smallest problem size.

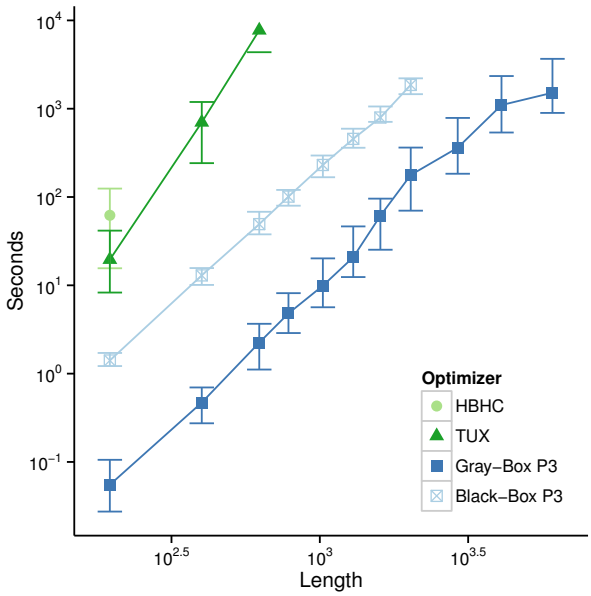


Figure 9.6: Comparison of how each algorithm's time required to find the best fitness found scales with problem size on Ising Spin Glass. With $N = 2025$ Gray-Box P3 is 4.6x faster than Black-Box P3. HBHC was only successful on the smallest problem size.

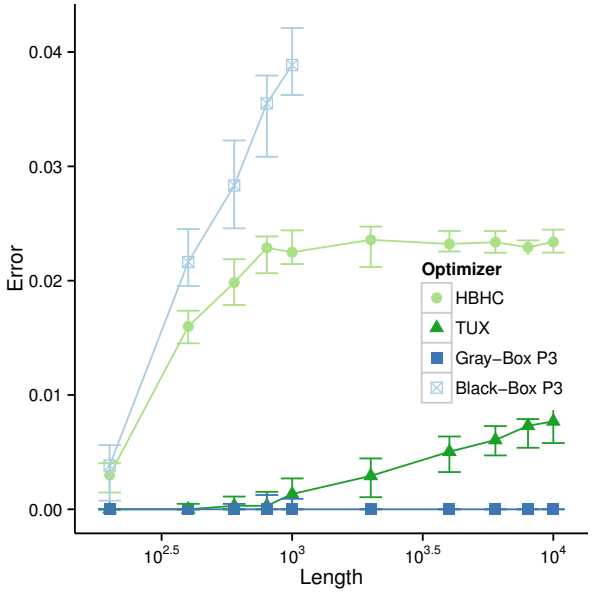


Figure 9.7: Relative qualities of each method as problem size increases on Unrestricted NKq with $K = 4$.

Black-Box P3 scales at $\mathcal{O}(N^{3.05})$.

As Unrestricted NKq does not have a known global optimum, and the different algorithms rarely found the same best fitness, Figure 9.7 compares the error for each technique at termination. Gray-Box P3 in general finds the best fitness, with TUX occasionally performing better. When $N > 2,000$, Gray-Box P3 finds better quality solutions than all other methods for every instance. Using the Mann-Whitney test to compare the fitness of Gray-Box P3 and TUX when $N = 10,000$ results in $p < 10^{-16}$. HBHC and Black-Box P3 only reach similar qualities as TUX and Gray-Box P3 when $N = 200$, doing so in 4 and 7 runs, respectively. As the problem size increase TUX begins to fall behind Gray-Box P3, with the HBHC stabilizing at about 2.5% worse than the best found. Here Black-Box P3 performs worse than the other techniques, falling further behind as the problem size increases.

9.4 Discussion

In line with previous work, we have found that HBHC cannot effectively find global optima on problems with even moderate epistasis. In general, it also obtains almost no

improvement in fitness after only a few restarts. We designed TUX as a simplistic way of choosing restart locations based on previously found local optima. Unlike HBHC alone, TUX was able to continue improving given more time, finding global optima on problems three times as large.

TUX's effectiveness is likely due to the HBHC acting as a super repair operator for uniform crossover. Given a sufficiently large r the HBHC can return sections of the crossover offspring to either parent's original version of a given subfunction. The HBHC is elitist meaning there is a bias toward returning to the better of the two parents' versions. Furthermore, by being so disruptive, uniform crossover potentially allows for the HBHC to also find unrelated improvements.

While TUX improves over plain HBHC, Gray-Box P3 is required to perform truly successful global optimization. Gray-Box P3 replaces naïve local search with the HBHC and utilizes known non-linear relationships instead of statistical linkage learning. On NKq-Landscapes this drastically improves search effectiveness. A major source of this improvement is likely how difficult it is for Black-Box P3 to learn linkage relationships on these landscapes. Furthermore, Gray-Box P3 can perform partial reevaluation and efficient hill climbing during the mixing phase.

Gray-Box P3's success is less dramatic on Ising Spin Glasses. While it still outperformed all competitors, Black-Box P3 may actually scale better to larger problems. One explanation for this deviation is that Ising Spin Glasses require more exploration of equal fitness plateaus. For instance, in Figure 9.3 and Figure 9.4 there is a significant pause in improvement when Gray-Box P3 reaches the second-best fitness in the landscape. Nothing in its design suggests that Gray-Box P3 should be more effective at neutral drift. Another potential issue is that on these landscapes the importance of each non-linear relationship may be detectably unequal. As a result, Black-Box linkage learning may better cluster variables that have meaningful impact on fitness while Gray-Box assumes all are equally important. A useful direction for future work would be to explore methods of performing efficient learning on top of the known

variable interactions.

Somewhat surprising is the difference in behavior between the polynomially solvable problems and Unrestricted NKq. While Black-Box P3 performed well in the former, it was the least successful in the latter. The optimal radius for Gray-Box P3 also shifted from 1 to 2. A potential explanation is that in both Ising Spin Glass and Nearest Neighbor NKq the number of unique variables within a given radius from a given variable is significantly lower than in the worst case. This explains why on Unrestricted NKq even moderately high r values hit our memory limit. For Black-Box P3 this may also be causing increased difficulty in linkage learning as variables become indirectly dependent on much larger sets. Furthermore, Black-Box P3 may be benefiting from an increased rate of duplicate dependencies on Nearest Neighbor NKq not present in Unrestricted NKq.

While the inclusion of HBHC into Gray-Box P3 introduces a parameter, it requires trivial configuration. In the worst case there may be a handful of r values to test. Furthermore, our evidence suggests setting $r = 1$ is quite powerful, with higher values likely to be only a time shift in quality. This is in contrast to r's role in HBHC, where low r values are never expected to reach the same quality as higher r values. Therefore we conclude that Gray-Box P3 maintains the out-of-the-box quality of Black-Box P3, while drastically improving efficiency for this new domain of problems.

Chapter 10

Understanding When P3 Excels

The ruggedness and high dimensionality of most interesting landscapes makes them challenging to visualize or otherwise analyze. However, doing so can be helpful in quantifying the difficulty of a problem, and how to best design algorithms to deal with those difficulties. Similarly, knowing which characteristics favor a particular algorithm can help researchers choose the algorithm most likely to perform well on their problem. To further this end, we explore the landscapes used in previous sections to understand what makes a landscape suited for P3 optimization.

10.1 Big Valley

One method for visualizing the global structure of a landscape is to examine the relationship between local optima [2]. In its original form, this process involves generating thousands of random solutions and then applying local search to each. This information is then displayed in two-dimensional plots: distance from the nearest global optimum and fitness difference from the global optima. For a number of interesting problems, a "Big Valley" of local optima exists, such that the higher a solution's fitness is, the closer it is in representation space to the global optimum. This result suggests that focusing search around known high-quality solutions increases the likelihood of finding even higher-quality solutions. This relationship is an underlying assumption of all evolutionary-based search methods, including

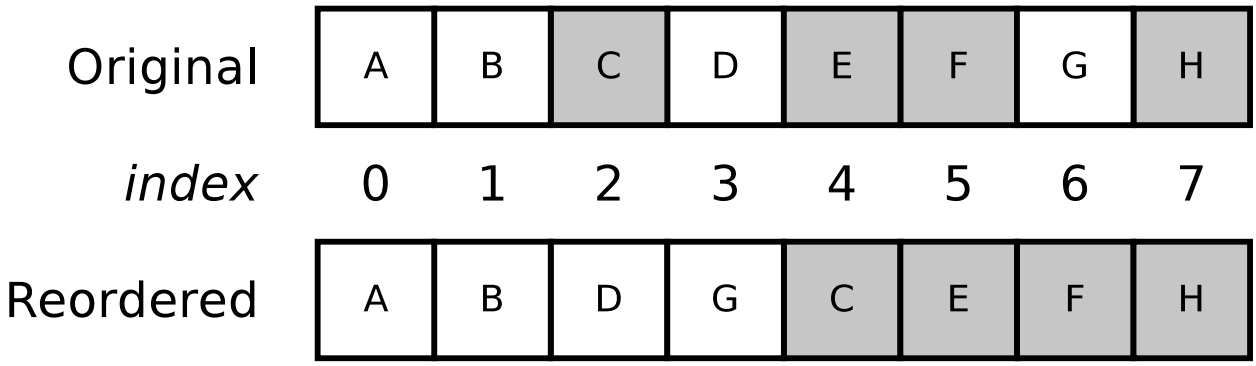


Figure 10.1: Example change of enumeration ordering. The gray loci represent all dependencies for some move m_i. By reordering, m_i's lowest *index* dependency improves from 2 to 4.

P3.

We have set out to extend this method of landscape visualization by considering not just a sample of local optima, but all local and global optima in the landscape. When considered as a black-box, finding all local optima requires each solution to be enumerated and all of its neighbors to be checked for fitness improvements. This enumeration is prohibitively expensive for even trivial landscapes, requiring $\Omega(N2^N)$ operations. However, by leveraging the gray-box domain, some efficiency improvements can be made.

10.1.1 Quickly Finding All Local Optima

For gray-box problems, we can determine the set of all fitness-improving moves from a given solution in $\mathcal{O}(N)$ time and this set can be updated in $\mathcal{O}(1)$ time when flipping a single bit. Therefore, checking if each new solution is locally optimal requires $\mathcal{O}(1)$ time and the overall enumeration process requires $\mathcal{O}(2^N)$.

Due to the limited non-linearity of the gray-box domain, it is possible to exclude large parts of the search space without missing any local optima. Consider the representation presented in the top of Figure 10.1. In a black-box domain, enumeration would progress as a binary counter, treating *index* zero (symbol A in the genome) as the least significant bit. This ensures that before changing *index* i, all possible settings of *index* 0 through $i-1$ have been tested. The gray-box domain makes it possible to skip combinations which

cannot be local optima. In Figure 10.1 there exists a move m_i which is a fitness improvement when enumeration starts (all variables set to 0). Due to the known relationships between variables, we know that the quality of m_i only depends on variables C, E, F, and H. Therefore, until one of those four variables are modified, the solution cannot be a local optimum. As a result, solutions 00000000, 10000000, 01000000, and 11000000 cannot be local optima and can therefore be skipped during enumeration. More generally, if at any point during enumeration there exists a fitness-improving move, no local optima can exist until at least one dependency of that move is modified.

This knowledge can be effectively exploited to skip parts of the enumeration, as shown in Figure 10.2. Before starting, each move is put into a table *move_bin* based on that move's lowest *index* dependency. This is the first *index* which can be modified by enumeration to change the fitness effect of making that move. In order to determine how much of the enumeration can be skipped, we must find the highest *index* in *move_bin* which contains a fitness-improving move, as done by FindNextIndex. If no move is fitness-improving, then a local optimum has been found.

The AllLocalOptima algorithm in Figure 10.2 works by repeatedly calling FindNextIndex and adding 1 to the resulting *index* position's value. Initially AllLocalOptima uses FindNextIndex to check all moves (initializes *index* to $N-1$). If at any point FindNextIndex returns -1 then no move is fitness-improving and the current solution is added to the list of local optima $found$. AllLocalOptima then adds a 1 to the *index* returned by FindNextIndex using the loop on Line 11 to perform carry operations and Line 16 to create the new 1 value. Iteration stops when the carry exceeds the solution length.

When performing subsequent checks, not all moves need to be retested for improvement. Instead, the highest *index* bin that must be tested is the highest *index* flipped by the previous iteration. This simplification is possible because the previous iteration has verified that all moves in higher *index* bins are not fitness-improving, and no action performed during that iteration can make them fitness-improving. Furthermore, no 1s can exist in lower *index*

```
1: procedure AllLocalOptima
2:     solution ← {0}^N
3:     found ← [ ]
4:     index ← N − 1
5:     while index < N do
6:         index ←FindNextIndex(index)
7:         if index = −1 then
8:             found ← found + [solution]
9:             index ← 0
10:        end if
11:        while index < N and solution[index] = 1 do
12:            MakeMove(index)
13:            index ← index + 1
14:        end while
15:        if index < N then
16:            MakeMove(index)
17:        end if
18:    end while
19:    return found
20: end procedure
21: procedure FindNextIndex(index)
22:    while index ≥ 0 do
23:        for all m ∈ move_bin[index] do
24:            if delta[m] ≥ 0 then return index
25:            end if
26:        end for
27:        index ← index − 1
28:    end while
29:    return −1
30: end procedure
```

Figure 10.2: Algorithm to find all local optima of a given gray-box problem. MakeMove, described in Figure 7.2, flips bit *index* and updates the fitness effect *delta* of making all moves dependent on *index*. *move_bin* stores moves based on their lowest *index* dependency.

positions, meaning iteration can continue immediately from the found *index*.

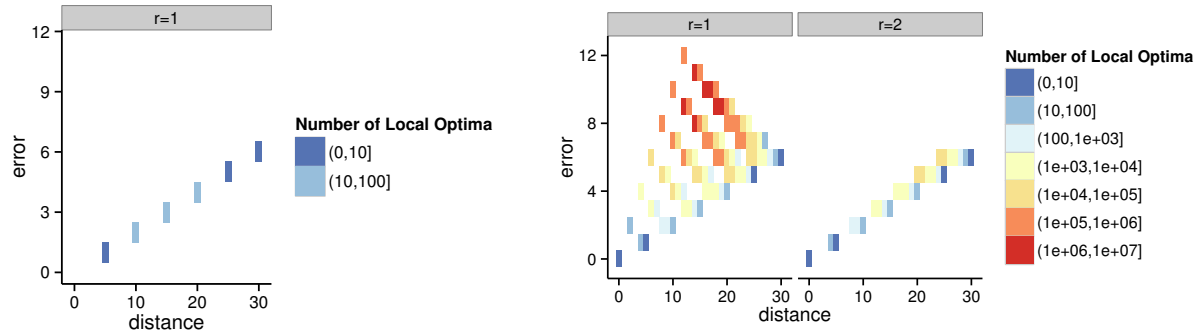
As a final efficiency, the order in which variables in the solution are *index*ed can be remapped. When a move is fitness-improving, the amount of search space that is skipped depends on how high its lowest *index* dependency is. Therefore, by rearranging the order to make its lowest *index* dependency higher, more search space can be skipped. We perform this remapping in a greedy fashion, such that the move with the least-unmapped dependencies has all of its remaining dependencies mapped to the most significant remaining positions. Figure 10.1 shows how changing the *index* order of variables improves m_i's lowest *index* dependency from 2 to 4. Now whenever m_i is a fitness improvement, FindNextIndex skips 4 times as much search space.

All together, these optimizations result in substantial efficiency improvements for some landscapes. For example, when applied to the OneMax problem (each bit scores one if set to 1, zero otherwise) this method finds all local optima in $\mathcal{O}(N)$ time. Deceptive Trap, regardless of how the bits are arranged, requires $\mathcal{O}(2^k 2^{N/k})$ time to find all $2^{N/k}$ local optima. This means a 60-bit Deceptive Trap problem with trap size of 4 requires 524,288 operations when using gray-box, but 1,152,921,504,606,846,976 with black-box.

As a final note, these methods extend to finding only r-bit local optima for all gray-box problems. However, the cost trade-off of doing so is unclear. By searching for only r-bit local optima, it is more likely that a fitness-improving move will exist, allowing more of the landscape to be skipped. Yet the increase in total number of moves means each time a bit is flipped more computation must be performed. Therefore, the runtime effect of increasing r depends on the specific problem.

10.1.2 Looking at Problems

In comparison to the randomly generated problem instances of Nearest Neighbor NKq and Ising Spin Glass, Deceptive Trap and Deceptive Step Trap have comprehensible landscapes. Therefore, these problems represent a good place to begin understanding what "big



(a) Deceptive Trap

(b) Deceptive Step Trap

Figure 10.3: Location and quality of local optima in comparison to the global optimum with $N = 30$ and $k = 5$.

valley" plots are showing. Figure 10.3 shows how the local optima in each of these problems are distributed.

The Deceptive Trap problem has few local optima, each of which actually provides little deception in terms of "big valley" properties. Given any two local optima, the one closer to the global optimum in fitness is also always closer to the global optimum in representation space. This makes sense as the worst local optimum contains all 0s, and each fitness improvement beyond that involves converting an entire trap to 1s, with the global optimum being all 1s. As a result, we should expect P3's method of elitist mixing to produce solutions that are progressively more and more similar to the global optimum until it is finally found.

Deceptive Step Trap's inclusion of fitness plateaus creates an enormous number of local optima. In total there are 24 million local optima on this problem, representing 2.2% of the entire search space. In comparison, Deceptive Trap using the same size has only 64 local optima. These additional local optima create a somewhat deceptive landscape, as selecting on fitness between two local optima can result in an increased genetic distance to the global optimum. Figure 10.3b with $r = 2$ shows how examining only 2-bit local optima causes Deceptive Step Trap using a step size of 2 to effectively revert to Deceptive Trap. This is because the hill climber can overcome all of the plateaus, leaving each trap either at the

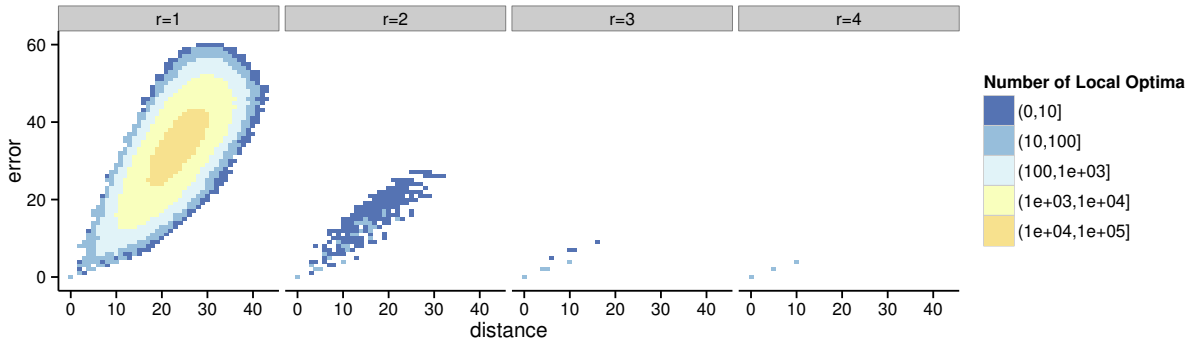


Figure 10.4: Location and quality of local optima in comparison to the global optima for a representative Nearest Neighbor NKq problem with $N = 60$ and $k = 2$.

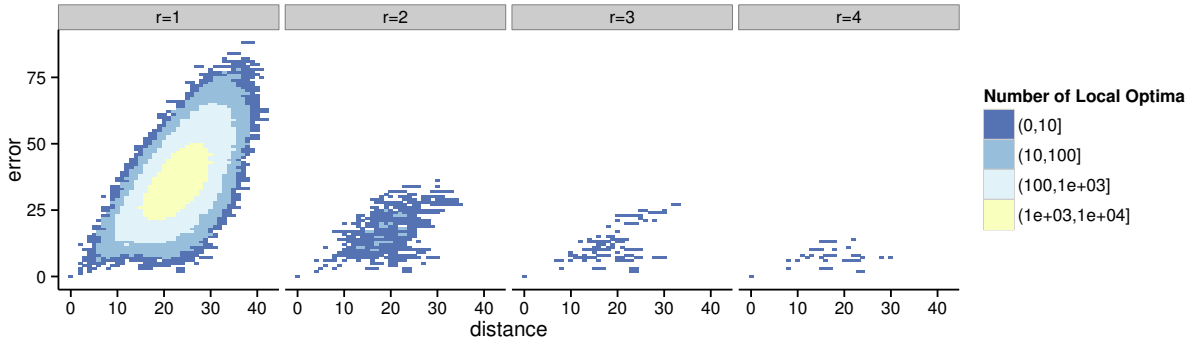


Figure 10.5: Location and quality of local optima in comparison to the global optima for a representative Unrestricted NKq problem with $N = 60$ and $k = 2$.

global optimum of all 1s or the local optimal value of all 0s or exactly one 1. Selection then becomes a nearly perfect predictor of distance to the global optimum.

Figure 10.4 shows how the local optima are distributed for Nearest Neighbor NKq. Unlike the trap problems, Nearest Neighbor NKq exhibits a traditional big valley shape. Increasing the radius of the local optima significantly reduces the total number of local optima, with those optima generally more similar to the global optima in both representation space and quality. Together this suggests that Nearest Neighbor NKq is a good candidate for selection-based methods like P3. Also, due to the general regularity and frequency of local optima, it may not be necessary to increase the radius in order to quickly find the global optimum.

While visually somewhat similar, Unrestricted NKq shown in Figure 10.4 suggests selec-

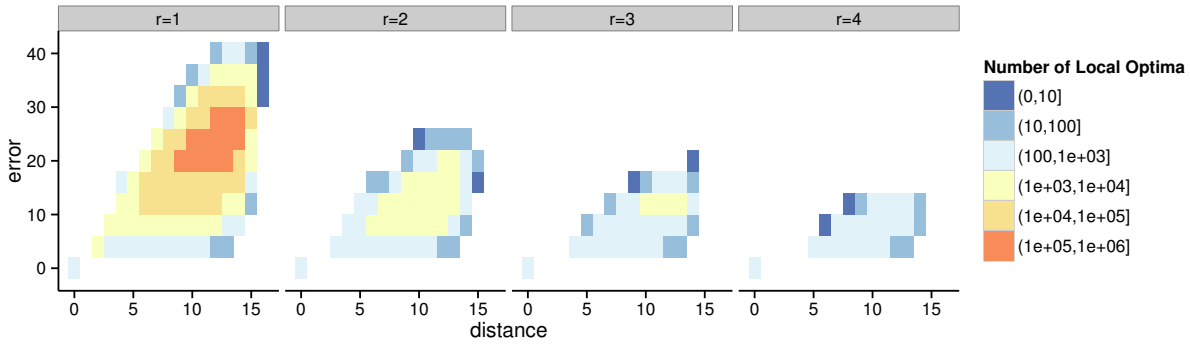


Figure 10.6: Location and quality of local optima in comparison to the global optima for a representative Ising Spin Glass problem with $N = 36$.

tion may be misleading. Consider the shape of the bottom of $r = 1$. At a distance of about 15 bits there is a slight hump, such that high-quality solutions are more frequent both closer and further from the global optimum than at the same distance. If search finds points in the area over 15 bits away from the global optimum, small modifications and elitist selection are more likely to lead away from the global optimum than toward it. At higher radius values this issue becomes more apparent, with optima flattening out away from the global optima in representation space. For many of these points, the only way to improve would require flipping over 15 bits correctly.

The Ising Spin Glass instance shown in Figure 10.4 provide some insight into the stalled behavior of Gray-Box P3 when solving that problem. In this landscape there are many local optima that have the second best fitness, but are very different from the nearest global optima. From a search perspective this means selection can only get you within a certain distance of a global optima, and then it becomes unhelpful. This issue is not improved by increasing the local optima radius, as even with $r = 4$ there are hundreds of local optima with the second best fitness that are over 10 bits different (of 36) from the nearest global optima.

Similar to Unrestricted NKq, the MAXSAT instance shown in Figure 10.7 suggests selection can be misleading. There is almost a negative correlation between fitness difference from the global optimum and representation distance from the global optimum. As before,

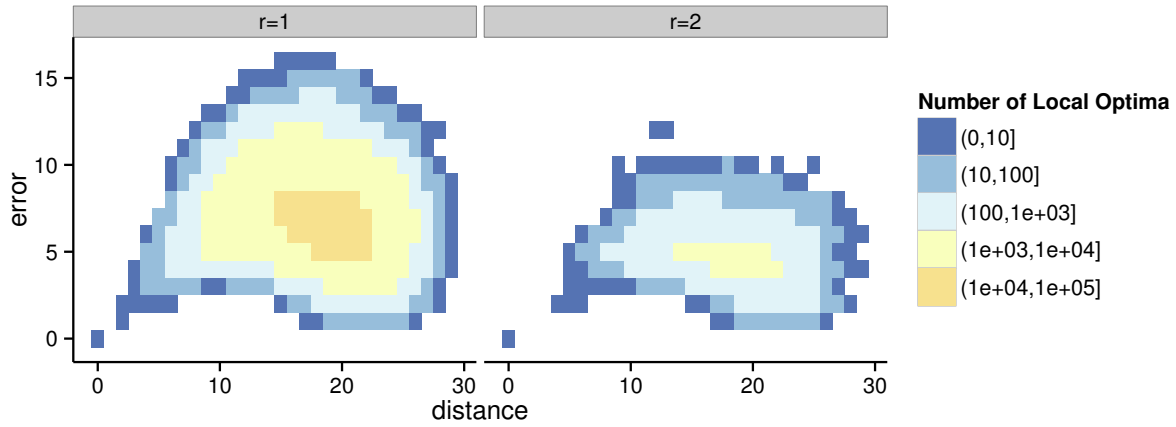


Figure 10.7: Location and quality of local optima in comparison to the global optima for a representative MAX-SAT problem with $N = 36$.

this problem is not solved by increasing the radius of the local optima. This may explain why methods like LTGA and hBOA, which extensively rely on selection, scale so poorly when tested on this problem. P3's comparative success may therefore be the result of the random restart hill climber eventually finding a solution in the correct area of the search space, with crossover finding the global optimum mostly by chance.

10.2 Pyramid Levels

One method for understanding how P3 performs search is to examine the types of solutions being stored at each level of the pyramid. By comparing each solution with the nearest best-found solution, we can create plots similar to the "big valley" plots in the previous section, even for problems too large and complex to find all possible global optima. This also provides a look at how the different levels of the pyramid focus search on different areas of the landscape.

Gray-Box P3's progression on Deceptive Step Trap, shown in Figure 10.8, follows directly from our expectations. The local optima found using only hill climbing and stored in level 0 are quite poor and far from the global optimum. The first application of crossover dramatically improves their quality. However, this improvement is likely just overcoming

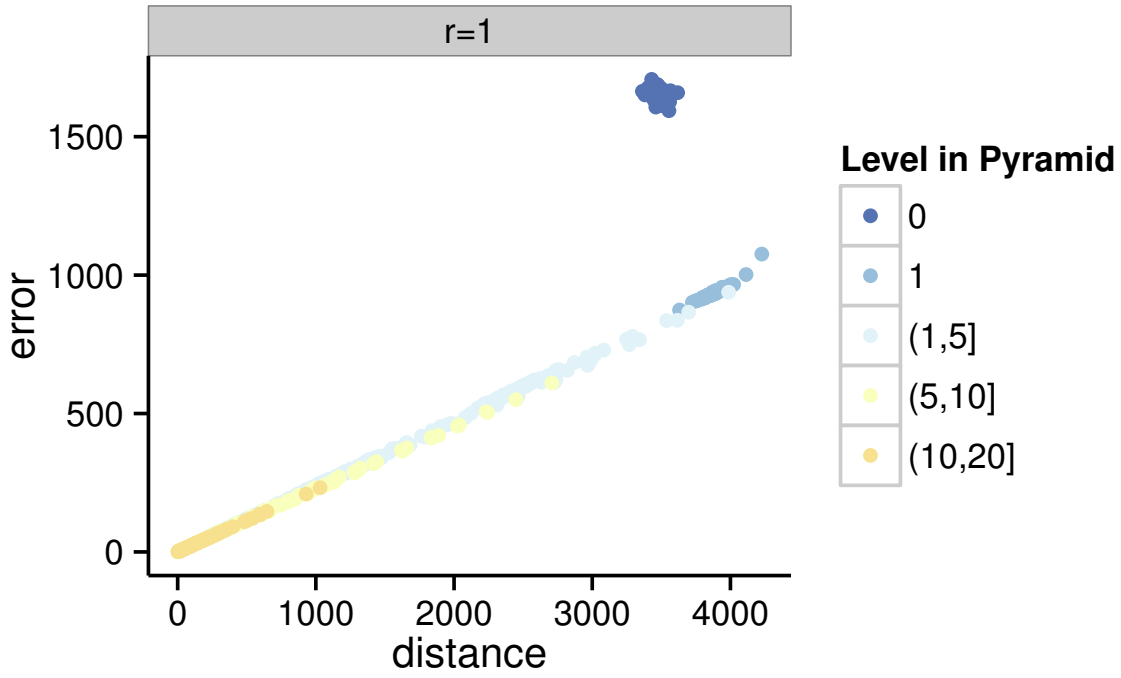


Figure 10.8: Distribution of local optima stored at each level of Gray-Box P3 in relation to the global optimum on the Deceptive Step Trap problem $N = 6000$ and traps of size 5.

the fitness plateaus, as solutions are actually being moved away from the global optimum. Subsequent levels store solutions that are generally closer to the global optimum in both fitness and representation space. This change means that at higher levels the frequency of traps being set to their values in the global optimum increases, resulting in better model building and better donation quality.

Nearest Neighbor NKq, shown in Figure 10.9, provides a landscape almost as free of higher-order deception as the Deceptive Step Trap problem. Shown with a logarithmic y-axis, stored solutions exhibit a near-perfect relationship between fitness and representational distance. Again, using only hill climbing creates local optima that are significantly worse and further away than even after a single application of crossover. While increasing the radius of the hill climber improves the quality of those initial solutions, even $r = 3$ is only able to equal a single crossover application. As before the higher level a solution is stored, generally the closer it is to a global optima in both fitness and representational distance.

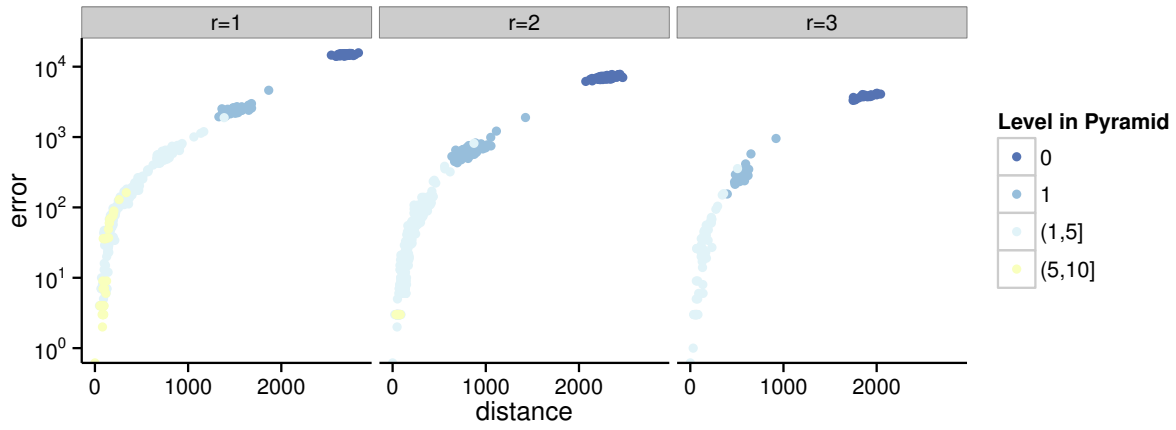


Figure 10.9: Distribution of local optima stored at each level of Gray-Box P3 in relation to the best found by the run on a Nearest Neighbor NKq problem $N = 6000$ and $K = 4$.

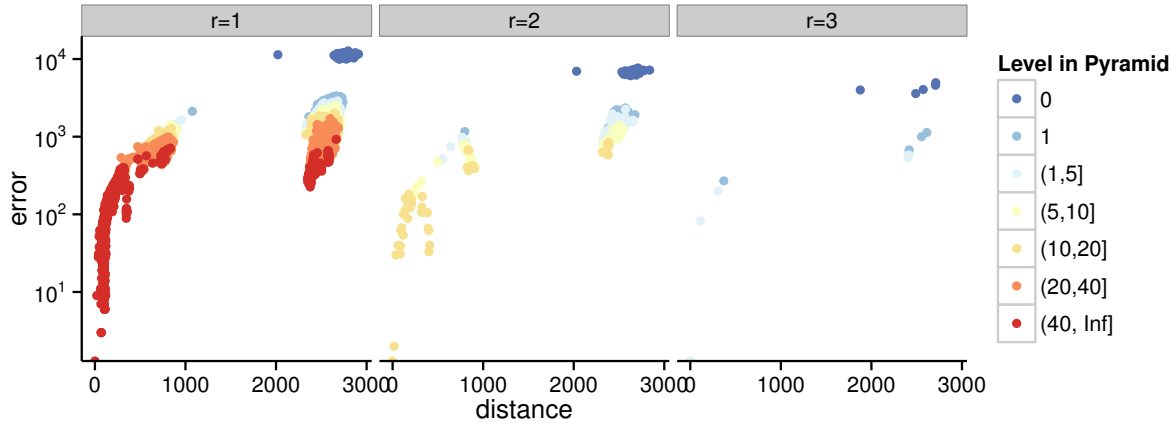


Figure 10.10: Distribution of local optima stored at each level of Gray-Box P3 in relation to the best found by the run on an Unrestricted NKq problem $N = 6000$ and $K = 4$.

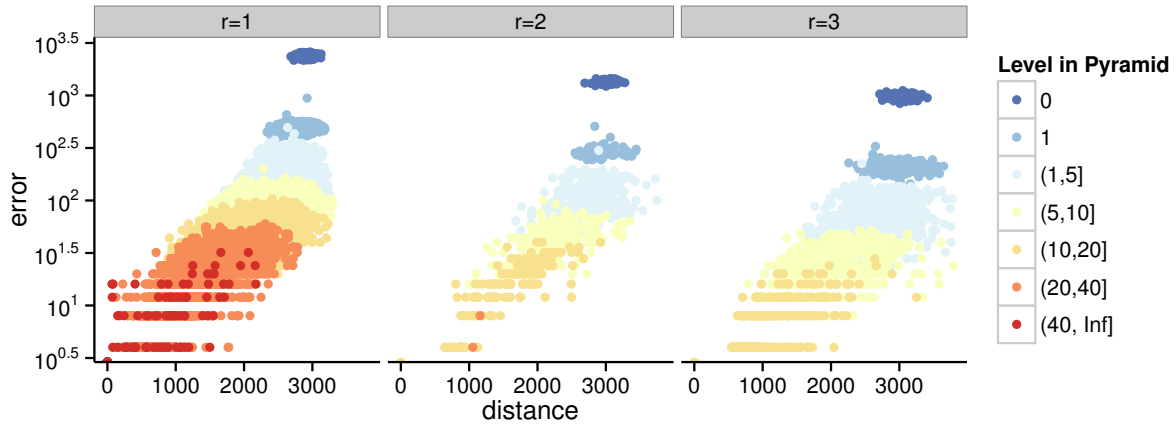


Figure 10.11: Distribution of local optima stored at each level of Gray-Box P3 in relation to the best found by the run on an Ising Spin Glass $N = 6084$.

Figure 10.10 gives insight into how different Unrestricted NKq is from Nearest Neighbor NKq. In this landscape Gray-Box P3, regardless of radius, appears to explore deceptive local optima that offer fitness improvements without moving closer to a global optima in representation space. In this problem there is a gap of over 1000 bits between the deceptive local optima and those likely to lead to the best-found solution. An interesting behavior on this problem is that the branch between the two groups always seems to begin with a single solution in level 1. This suggests that the stored solutions near the best found may all be similar due to sharing a common ancestor.

Gray-Box P3's progression on Ising Spin Glass, shown in Figure 10.11, is much more well behaved than on Unrestricted NKq. As has been typical, each application of crossover results in improved fitness, with the largest gains between the first few levels. When approaching higher qualities these improvements also begin to translate into increased representational similarity to the global optimum. However, as discussed previously for this problem, we again see evidence for a large number of diverse solutions with the second best fitness. These solutions can be over 1000 bits different from the eventual best-found solution. Increasing the radius of the hill climber does not seem to significantly overcome this issue.

Figure 10.12 shows that Gray-Box P3 acts more similarly on MAX-SAT to Unrestricted

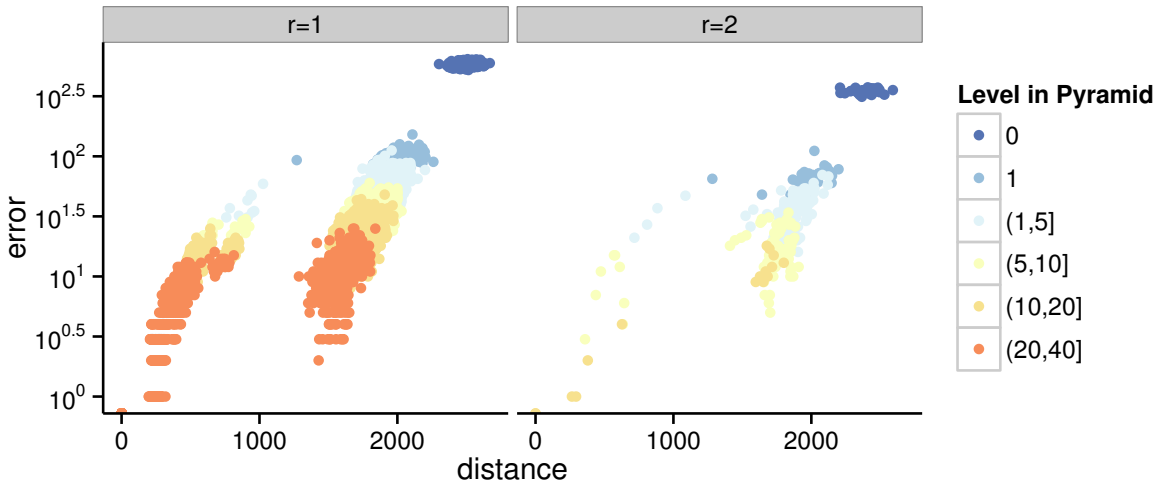


Figure 10.12: Distribution of local optima stored at each level of Gray-Box P3 in relation to the best found by the run on a MAX-SAT problem $N = 6000$.

NKq than on any of the other problems. Again there appear to be deceptive local optima between 500 and 1000 bits away from the nearest improving solution. Again a single solution appears to be the ancestor of all of the solutions near the best found. This suggests that similar to its performance on Unrestricted NKq, Gray-Box P3 will have a hard time finding the global optimum on MAX-SAT.

In general it appears that the problems that have polynomial-time solutions (Deceptive Step Trap, Nearest Neighbor NKq, 2D Ising Spin Glass) share a similar behavior of selection leading to the global optimum. However, the NP-Hard problems (Unrestricted NKq, MAX-SAT) seem to contain large amounts of higher-order deception, making them challenging for selection-based methods. This relationship warrants further investigation into other NP-Hard problems to determine if the repeated selection found in P3 and other evolutionary computation methods are able to overcome this deception. One potential avenue for improvement to P3 may also be a method for dealing with this issue. For instance, it may be beneficial to explicitly partition solutions when this deceptive behavior is observed.

Chapter 11

Conclusions and Future Work

The Parameter-less Population Pyramid (P3) is a recently introduced method for performing black-box optimization. P3's primary innovation is the replacement of the generational model with a pyramid of populations. This pyramid is constructed iteratively, with both the number of levels and the number of solutions stored at each level growing as search progresses. P3 uses a model based crossover method that learns a linkage tree from gene epistasis. Combined with a simple hill climber, P3's design contains many synergistic features.

Across a large number of problems and problem sizes P3 required fewer evaluations to reach the global optimum than optimally tuned state-of-the-art competitors. On single-instance problems P3's improvement was by a constant factor, while for the three randomly generated problem classes P3's improvement increased with problem size. This quality extends to intermediate points during evolution, with P3 generally reaching at least as high a fitness as the competitive techniques when using the same number of evaluations. While P3 does require modeling overhead, the expense of this overhead is approximately linear with respect to genome size. There is some evidence that even when compared on wall clock time, P3 performs on par with the best comparison techniques. All of these achievements are made without any problem-specific parameter tuning, making P3 easier to apply to new domains than its two closest competitors in quality.

P3's quality is due to a number of desirable traits. First, mixing local search with

model based crossover lets search focus on properly mixing high-quality solutions. Second, by adding diversity only as necessary P3 tends to use the minimal amount of random initialization, unlike other techniques that must overcompensate with larger population sizes on single-instance problems and consider the worst instance when solving problem classes. Third, by heavily exploiting existing diversity before adding more P3 is able to reach high-quality intermediate fitnesses quickly without prematurely converging. Fourth, the nature of the pyramid's shape allows search to preserve a desirable proportion of diversity at each fitness level, similar to a generational model using a decreasing population size.

When leveraging features of the new domain, Gray-Box P3 can obtain substantial improvement over Black-Box P3. This comes from the integration of the Hamming-Ball Hill Climber (HBHC) and a novel method for deriving linkage clusters from the known problem epistasis, both of which result in a factor N speedup over their black-box counterparts. Furthermore, Gray-Box P3 can perform partial reevaluation and efficient hill climbing during the mixing phase without significant overhead. In specializing, Gray-Box P3 is able to find global optima orders of magnitude faster than Black-Box P3.

There are a number of meaningful avenues for future P3 experimentation. Perhaps the most pressing for practitioner acceptance is to apply P3 to real-world problems and compare its results with other black-box or even problem-specific heuristics. While parameter-less, P3 is currently limited to discrete, fixed-length genomes evaluated using single-objective fitness. These limitations can be relaxed with future work to make P3 more widely applicable. While asymptotically linear in problem size, Black-Box P3's modeling techniques and local search methods are likely going to be prohibitively expensive for genome sizes in the hundreds of thousands or millions of genes, and the inability of the model to capture overlapping linkage may be hindering search efficiency. Overcoming these limitations by using a new modeling technique may allow the pyramid model even greater flexibility. Similarly, while P3 is able to overcome low-order deception via linkage learning, the iterative improvement method by which crossovers are made may mislead search on landscapes with higher-order deception.

This appears to be especially important on NP-Hard problems, where high-quality solutions may be very distant from the global optimum in representation space.

While empirically effective, we believe it is possible to derive theoretical foundations for Gray-Box P3. Specifically, this may be possible by considering the nature of crossover in terms similar to those used for finding r-bit local optima with HBHC. Consider that for any two parents the non-linear effects of crossover are constrained by where they contain different bit values. As a result, subsets of their differences can become linearly separable, allowing for each to be considered independently of all others. Furthermore, these conclusions about effective crossover in gray-box can likely be translated into a black-box setting to help explain the effectiveness of Black-Box P3.

However, even without these improvements our results show P3 is highly efficient at finding global optima on both black-box and gray-box problems without any problem-specific tuning.

APPENDIX

hBOA Simplification

To measure the quality of a decision forest, hBOA applies Equation 1. To favor compact models, Equation 1 is scaled by Equation 2, which provides increased cost for more total leaves. This quality is used for two purposes: comparison of potential changes from the existing model and comparison of that change with the existing model. The basis for our simplication is to rearrange $p(B)BDe(B) < p(B')BDe(B')$ to be $\frac{p(B)}{p(B')} < \frac{BDe(B')}{BDe(B)}$.

$$BDe(B) = \prod_{i=0}^{N} \prod_{l \in L_i} \frac{\Gamma(m'_i(l))}{\Gamma(m_i(l) + m'_i(l))} \prod_{x_i} \frac{\Gamma(m_i(x_i, l) + m'_i(x_i, l))}{\Gamma(m'_i(x_i, l))} \tag{1}$$

$$p(B) = c2^{-0.5(\sum_i |L_i|)log_2\mu} \tag{2}$$

The outermost product of Equation 1 iterates over all trees in the forest. However, each split can modify only one of the trees and therefore the contribution of all others can be canceled. The middle product is across all leaves in the tree. Again since only one leaf can be changed, all other terms can be canceled. By convention hBOA uses uninformed Bayesian priors of $m'_i(l) = 2$ and $m'_i(x_i, l) = 1$ for binary alphabets. As $\Gamma(a) = (a-1)!$ this means the top term in the middle product and the bottom term in the third product reduce to 1. The only remaining terms are then $m_i(l)$ and $m_i(x_i, l)$ which represent the number of solutions that reached leaf l and the number of solutions that reached leaf l with a specific value for x_i, respectively.

Equation 2 can also be simplified when doing comparisons. If model B' has exactly one more leaf than model B then the ratio $\frac{p(B)}{p(B')}$ simplifies to $2^{0.5log_2\mu}$ regardless of total model

size.

The resulting simplifications create Equation 2.1, where B' is different from B by exactly 1 split, such that l was split to create l' and l''. The best split is whichever maximizes its improvement over B, which is equal to the right side of the inequality. Note that these factorials can still be exceedingly large and therefore it is imperative that implementations avoid rounding errors and overflows.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] José M Arroyo and Antonio J Conejo. Optimal response of a thermal unit to an electricity spot market. *Power Systems, IEEE Transactions on*, 15(3):1098–1104, 2000.

[2] Kenneth D Boese, Andrew B Kahng, and Sudhakar Muddu. A new adaptive multi-start technique for combinatorial global optimizations. *Operations Research Letters*, 16(2):101–113, 1994.

[3] Peter A. N. Bosman and Dirk Thierens. The roles of local search, model building and optimal mixing in evolutionary algorithms from a bbo perspective. In *Optimization by building and using probabilistic models (OBUPM-2011)*, pages 663–670, Dublin, Ireland, 12-16 July 2011. ACM.

[4] Francisco Chicano, Darrell Whitley, and Andrew M. Sutton. Efficient identification of improving moves in a ball for pseudo-boolean problems. In *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 437–444, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[5] Benjamin Doerr, Carola Doerr, and Franziska Ebel. Lessons from the black-box: fast crossover-based genetic algorithms. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 781–788, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[6] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *Theory and applications of satisfiability testing*, pages 502–518. Springer, 2004.

[7] Paola Festa, Panos M Pardalos, Mauricio GC Resende, and Celso C Ribeiro. Randomized heuristics for the max-cut problem. *Optimization methods and software*, 17(6):1033–1058, 2002.

[8] David E. Goldberg, Kalyanmoy Deb, and James H. Clark. Genetic algorithms, noise, and the sizing of populations. *COMPLEX SYSTEMS*, 6:333–362, 1991.

[9] Brian W. Goldman and William F. Punch. Parameter-less population pyramid. In *GECCO '14: Proceedings of the 2014 conference on Genetic and evolutionary computation*, pages 785–792, Vancouver, BC, Canada, 12-16 July 2014. ACM.

[10] Brian W. Goldman and William F. Punch. Fast and efficient black box optimization using the parameter-less population pyramid. *Evolutionary computation*, 2015.

[11] Brian W. Goldman and Daniel R. Tauritz. Meta-evolved empirical evidence of the effectiveness of dynamic parameters. In *GECCO '11: Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pages 155–156, Dublin, Ireland, 12-16 July 2011. ACM.

[12] Brian W. Goldman and Daniel R. Tauritz. Linkage tree genetic algorithms: variants and analysis. In *GECCO '12: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference*, pages 625–632, Philadelphia, Pennsylvania, USA, 7-11 July 2012. ACM.

[13] John Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. on Systems, Man, and Cybernetics*, SMC-16(1):122–128, 1986.

[14] Ilan Gronau and Shlomo Moran. Optimal implementations of UPGMA and other common clustering algorithms. *Information Processing Letters*, 104(6):205–210, 2007.

[15] Georges R. Harik and Fernando G. Lobo. A parameter-less genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 258–265, Orlando, Florida, USA, 13-17 July 1999. Morgan Kaufmann.

[16] Gregory S. Hornby. ALPS: the age-layered population structure for reducing the problem of premature convergence. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, volume 1, pages 815–822, Seattle, Washington, USA, 8-12 July 2006. ACM Press.

[17] B. D. Jovanovic and P. S. Levy. A look at the rule of three. *The American Statistician*, 51(2):137–139, May 1997.

[18] Karthik Kannappan, Lee Spector, Moshe Sipper, Thomas Helmuth, William Lacava, Jake Wisdom, and Omri Bernstein. Analyzing a decade of human-competitive (HUMIE) winners:what can we learn? In *Genetic Programming Theory and Practice XII*, Genetic and Evolutionary Computation. Springer, Ann Arbor, USA, May 2014. In preparation.

[19] G Jake LaPorte, Juergen Branke, and Chun-Hung Chen. Adaptive parent population sizing in evolution strategies. *Evolutionary Computation*, 2014. Early Access.

[20] Fernando G. Lobo. Idealized dynamic population sizing for uniformly scaled problems. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 917–924, Dublin, Ireland, 12-16 July 2011. ACM.

[21] Amichai Painsky and Saharon Rosset. Optimal set cover formulation for exclusive row biclustering of gene expression. *Journal of Computer Science and Technology*, 29(3):423–435, 2014.

[22] Martin Pelikan and DavidE. Goldberg. Hierarchical bayesian optimization algorithm. In *Scalable Optimization via Probabilistic Modeling*, volume 33 of *Studies in Computational Intelligence*, pages 63–90. Springer Berlin Heidelberg, 2006.

[23] Martin Pelikan and Tz-Kai Lin. Parameter-less hierarchical BOA. In *Genetic and Evolutionary Computation – GECCO-2004, Part II*, volume 3103 of *Lecture Notes in Computer Science*, pages 24–35, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[24] Martin Pelikan and Fernando G. Lobo. Parameter-less genetic algorithm: A worst-case time and space complexity analysis. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, page 370, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[25] Petr Pošík and Stanislav Vaníček. Parameter-less local optimizer with linkage identification for deterministic order-k decomposable problems. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 577–584, Dublin, Ireland, 12-16 July 2011. ACM.

[26] Soraya Rana and Darrell L. Whitley. Genetic algorithm behavior in the MAXSAT domain. In *Parallel Problem Solving from Nature – PPSN V*, pages 785–794, Berlin, 1998. Springer. Lecture Notes in Computer Science 1498.

[27] Ingo Rechenberg. *Evolutionsstrategie Optimierung technischer Systeme nach Prinzipien der biologishen Evolution*. mit einem Nachwort von Manfred Eigen, Friedrich Frommann Verlag, Struttgart-Bad Cannstatt, 1973.

[28] Silvia Richter, Malte Helmert, and Charles Gretton. A stochastic local search approach to vertex cover. In *KI 2007: Advances in Artificial Intelligence*, pages 412–426. Springer, 2007.

[29] Kumara Sastry. *Evaluation-relaxation schemes for genetic and evolutionary algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 2001.

[30] Lawrence Saul and Mehran Kardar. The 2d±j ising spin glass: exact partition functions in polynomial time. *Nuclear Physics B*, 432(3):641–667, 1994.

[31] Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1–2):17–29, 1996.

[32] David Sherrington and Scott Kirkpatrick. Solvable model of a spin-glass. *Physical review letters*, 35(26):1792, 1975.

[33] Dirk Thierens. The linkage tree genetic algorithm. In *Parallel Problem Solving from Nature, PPSN XI*, pages 264–273. Springer, 2010.

[34] Dirk Thierens and Peter A. N. Bosman. Optimal mixing evolutionary algorithms. In *GECCO '11: Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 617–624, Dublin, Ireland, 12-16 July 2011. ACM.

[35] Dirk Thierens and Peter A. N. Bosman. Hierarchical problem solving with the linkage tree genetic algorithm. In *GECCO '13: Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference*, pages 877–884, Amsterdam, The Netherlands, 6-10 July 2013. ACM.

[36] Richard A. Watson, Gregory S. Hornby, and Jordan B. Pollack. Modeling building-block interdependency. In *Late Breaking Papers at the Genetic Programming 1998 Conference*, pages 234–240, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Stanford University Bookstore.

[37] Darrell Whitley, Adele E Howe, and Doug Hains. Greedy or not? best improving versus first improving stochastic local search for maxsat. In *AAAI*, 2013.

[38] David H. Wolpert and William G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[39] A. H. Wright, R. K. Thompson, and J. Zhang. The computational complexity of N-K fitness functions. *IEEE Transactions on Evolutionary Computation*, 4(4):373–379, November 2000.

[40] Liang Zhao, Hiroshi Kadowaki, and Dorothea Wagner. A practical approach for finding small {Independent, Distance} dominating sets in large-scale graphs. In *Algorithms and Architectures for Parallel Processing*, pages 157–164. Springer, 2013.