Parametric Optimization of Storage Systems

A RESEARCH PROFICIENCY EXAM PRESENTED

BY
ZHEN CAO
STONY BROOK UNIVERSITY

Technical Report FSL-16-01

January 2016

Abstract

Computers are consuming more than 10% of world's energy use, and this amount is still growing every year. This means that even a small percentage improvement in the performance and energy efficiency of computer systems could have a significant impact world wide. Many production systems are I/O bound, spending large amount of time waiting for file system and storage operations. All these facts motivate the need for optimizing storage systems—the slowest part of computing systems. Most modern storage systems come up with a large number of tunable parameters. Previous work has shown that tuning even a small subset of those parameters can improve the performance and energy efficiency by a as much as 9.4×. Manually tuning storage systems is impractical due to the huge number of valid configurations and the difficulty of evaluation all of them; moreover, many metrics are sensitive to the environment, which means that tuning results in one environment are not easily applicable to other environments. Even domain experts cannot explain and predict all the behaviors of storage systems. This significant complexity all but eliminates the possibility of applying any white-box optimization techniques or attempting to model the system accurately.

In this report we describe the tuning of storage systems parameters as the problem of finding a global optima in a high-dimensional search spaces. We propose to develop a black-box auto-tuning technique for storage systems. We argue that tuning techniques that work in other fields do not fit well in our case due to inherent properties of our problem (e.g., non-numeric parameters, huge search space, parameter dependence, environment sensitivity). By investigating various techniques, we found that Meta-Heuristics (MH) and Reinforcement Learning (RL) techniques are the most promising ones for our problem. We discuss these techniques in detail in this report. All of these techniques can be generally characterized by a trade-off among exploration, exploitation, and history information—the three key MH features for finding the global optimum in high-dimensional spaces. Although traditional supervised Machine Learning (ML) techniques requires a large and high-quality training dataset and thus are generally inapplicable, we argue that they still can serve as good supplements to our search algorithms. The long-term goal of our project is to develop hybrid auto-tuning algorithms, based on any helpful techniques, that are general enough to automatically and quickly optimize any storage systems for various goals—as well as quickly adapt re-tune a system in response to changes in the environment. Through our preliminary results, we show the promise of MH techniques in finding the near-optimal configuration accurately and efficiently, and also the possibility of using ML techniques as a supplement. We close this report by describing our future plans.

To My Family.

Contents

List of Figures iv

List of Tables v

List of Algorithms vi

1 Introduction 1

2 Background 4
- 2.1 Problem Statement . . . 4
- 2.2 Optimization Approaches . . . 6
- 2.3 Key Properties . . . 9
- 2.4 Meta-Heuristics . . . 10
  - 2.4.1 Meta-Heuristics . . . 10
  - 2.4.2 Simulated Annealing . . . 11
  - 2.4.3 Genetic Algorithms . . . 16
    - 2.4.3.1 Biological Background . . . 16
    - 2.4.3.2 Evolutionary Algorithms . . . 17
    - 2.4.3.3 Simple Genetic Algorithm . . . 18
    - 2.4.3.4 Theories on GAs . . . 26
    - 2.4.3.5 Variants of GAs . . . 26
    - 2.4.3.6 Applications of Genetic Algorithms . . . 27
  - 2.4.4 Differential Evolution . . . 27
  - 2.4.5 Particle Swarm Optimization . . . 28
  - 2.4.6 Tabu Search . . . 29
  - 2.4.7 Other Meta-Heuristic Techniques . . . 31
- 2.5 Machine Learning . . . 31
  - 2.5.1 Supervised Learning . . . 31
  - 2.5.2 Unsupervised Learning . . . 32
  - 2.5.3 Reinforcement Learning . . . 32
- 2.6 Summary . . . 33

3 Related Work 36
- 3.1 Auto-tuning in Storage Systems . . . 36
- 3.2 Auto-tuning in Other Areas . . . 37

4 Experiment Settings **40**
4.1 Hardware 40
4.2 Software 40
4.3 Dataset 41

5 Preliminary Results **44**
5.1 Data Collection 44
5.2 Genetic Algorithms 45
5.3 Machine Learning 48
5.4 Feature Selection 49

6 Future Work **53**

7 Conclusion **55**

List of Figures

2.1 Non-linearity property shown in the parameter *pagepool* of GPFS under the DATABASE workload . . . 6
2.2 Simplified Terrain Figure for the Parameter Space . . . 10
2.3 Flow chart of an Evolutionary Algorithm . . . 18
2.4 Configuration connectivity figure of Storage V2. . . . 22
2.5 1-Point Crossover . . . 25
2.6 Example of mutation operations in GAs . . . 26

4.1 Percentage of Different File Systems in the Whole Search Space . . . 42

5.1 GA results for *mailserver* workload on M1 machines . . . 47
5.2 GA results for *fileserver* workload on M2 machines . . . 47
5.3 Number of Configurations for Each FS Type . . . 48
5.4 Prediction Accuracy of Decision Tree Algorithms with Varying Size of Training Set and Number of Classes (mailserver) . . . 49
5.5 Prediction Accuracy of Decision Tree Algorithms with Varying Size of Training Set and Number of Classes (dbserver) . . . 50
5.6 Mutual Information for Different Parameter Types . . . 51
5.7 Mutual Information for Different Parameters Within Each File System (mailserver) . . . 51
5.8 Mutual Information for Different Parameter Within Each File System (dbserver) . . . 52

List of Tables

2.1 Example search spaces for storage systems 5
2.2 Examples of popular workloads 7
2.3 Comparison of four major EAs 19
2.4 Comparison and Summaries of MH Techniques 35

4.1 Details of Experiment Machines 40
4.2 Details of Parameter Space 43

5.1 Global Optima in Different Search Spaces 44
5.2 List of GA Parameters and Their Defaults 46
5.3 Results of GA experiments on M1 for Mailserver workload 46

List of Algorithms

1 Simulated Annealing 12
2 Simple Genetic Algorithm 20
3 Differential Evolution 28
4 Particle Swarm Optimization 29
5 Tabu Search 30

Chapter 1

Introduction

Modern storage systems are often characterized by multiple connected hardware units with different software running various user application workloads. They have a large number of tunable parameters at each layer that can affect performance, energy, and more. Often, these systems are deployed with default settings that directly come from vendors mostly due to two reasons: (1) it is nearly impossible for administrators to know the impact of every parameter across multiple layers; (2) vendors' default configurations are trusted to be "good enough." However, we have shown that the default configurations of storage systems are usually sub-optimal, and only by tuning a tiny subset of parameters, we were able to improve the power- and performance efficiency by a factor of 1.5–9.4$\times$ [318]. By 2013, computers were consuming more than 10% of the world's energy use, and this number is expected to continue growing [97]. Having such a huge base, a small percentage of improvement on energy or performance efficiency would mean a lot. We choose to focus on storage systems as they are the slowest computer components (at least a million times slower than CPUs). As Moore's law has slowed down, it becomes even more important to squeeze every bit of performance out of deployed storage systems. We model tuning storage systems as the problem searching for the best configuration in the given space. However, finding near-optimal configurations is challenging for several reasons:

- **The search space is huge**. While some parameters are merely boolean, many are integers that can take on thousands or even billions of values. For a storage system with NFS, and only considering a subset of 18 useful parameters (NFS version, r/wsize, a/sync, no_/wdelay, etc.), we counted that the total number of possible parameter permutations already exceeds 10^{18}.
- **Good configurations are sensitive to the environment**. Here *environment* refers to both the hardware and the workload, and we follow this definition throughout this report. Among all parameters, some are important to optimize, while others may have insignificant impact and are a waste of effort to try. Some parameters are highly correlated with others, and thus these dependencies need to be discovered (and perhaps "collapsed" together). The precise set of parameters that affect the targeted metric, and their values, depend heavily on the environment. Even a small change to one part of the environment can deviate the storage system from its optimal outcome. This problem results in a complex, multi-dimensional search space with many local optima, and searching for a near-optimal configuration in a timely manner is the essence of our proposed problem.
- **Evaluation is harder**. Keeping the evaluation for one configuration as quick as possible is the key to the success of optimizing systems. Within the same time limit, we may save more time for exploring more configurations, which results in higher probability to find better ones. However, as storage systems are the slowest parts of computer systems, evaluating one configuration for storage systems usually takes a much longer time. To make things worse, evaluation results for storage systems are

sometimes unstable under certain workloads, which means more evaluation on the same configuration might be needed.

The above facts make it nearly impossible to manually tune storage systems for every potential environment. As modern storage systems become more and more complex, even those field experts or system developers cannot fully understand the behavior of storage systems. As a result, in this report we propose to explore and develop auto-tuning techniques that can automatically optimize storage system parameters to find near-optimal configurations.

Many in the past proposed to solve similar problems in fields other than storage systems. White-box optimization techniques are generally not useful in our case, as storage systems are so complex to be fully understand. Exhaustive search or random sampling methods are not efficient enough, due to the huge search space. Control theory cannot help because it requires data to build a model and is designed largely for linear systems. Conversely, our space is non-linear and contains many parameters that do not even have numeric values (e.g., file system type, disk type).

The goal of our project is to develop techniques that would be efficient in auto-tuning storage systems. We propose to build it on the basis of a set of techniques called Meta-Heuristics (MHs) [34,215,243,369,396, 413, 416], as well as Machine Learning techniques (ML) [31, 244]. MH can be defined as a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality. It is a generic term and consists of techniques including Genetic Algorithms (GA), Simulated Annealing (SA), and more. MH algorithms have been widely applied in various areas. Most MH algorithms are nature-inspired as they have been developed based on the successful evolutionary behavior of natural systems. Reinforcement Learning (RL) is one type of ML technique inspired by behaviorist psychology; it is concerned with how agents ought to take actions in an environment so as to maximize some notion of a cumulative reward. One simple form of RL is Q-Learning, which maintains the history of previous moves and uses them to guide future moves. Supervised ML techniques require a long training phase and the quality of the results depends heavily on the amount and quality of the training data, which is either unavailable or difficult to generate for large spaces. However, we find that they still can serve as a good supplement tool for our search algorithm. We find that all search algorithms have three important properties: *exploration*, *exploitation* and *history*. *Exploration* means randomly searching the space to avoid getting stuck in local optima, while *exploitation* means finding better solutions in the neighborhood area. *History* is the amount of cached results from previous searches that is used in future searches. The essence of many search techniques is the trade-off among these three properties. Through our preliminary results, we have shown that MH have the ability to search large spaces of storage system parameters efficiently and find the near-optimal configuration in a given environment. We also prove that ML techniques have the potential in aiding our search process. Based on the premise of MH and ML algorithms, we propose to develop our own auto-tuning technique that will work well for storage systems. It will be able to automatically "rank" the parameters and eliminate the less important ones to reduce the search space exponentially. It should also contain some stopping criteria that can realize near-optimal configurations have been found and stop running the algorithm in a smart way so as to save on resources.

We describe the main contributions of our project as follows:

- **Investigate various techniques**. We do not limit ourselves in just one technique. We plan to investigate and explore many types of MH techniques, or even other promising non-meta-heuristic optimization techniques, such as Reinforcement Learning (RL). We are trying to understand the essence of each technique, i.e., why they are able to find the near-optimal quickly or why they are not working well in certain cases. Base on the knowledge we acquire, we plan to develop and propose our own auto-tuning algorithm specifically for storage systems.

- **Our technique is supposed to be general**. The ultimate goal of our project is to develop an auto-

tuning module (e.g., for Linux) that works for any storage system and any optimization goals, such as performance, energy consumption, etc., and in the end eliminate the need for anyone to hand-tune storage systems ever again.

- **We will produce valuable data-sets and make it public online**. We conducted our experiments in the following way. For every workload, we tried every configuration exhaustively and collected various metrics, such as performance and energy. By doing this we are able to simulate all types of techniques on these collected data-sets. We are collecting a set of 9 parameters and totally 24,888 unique configurations (per workload). We have finished 2 different workloads and it already took us 6 months. We will continue filling these valuable data-sets with new results (in a MySQL database), and this process will continue for several years. We will make our data-sets available online periodically to benefit the research community.

The rest of this report is organized as follows. Section 3 surveys related work. Section 2 provides background knowledge. We formally describe our experimental settings in Section 4. In Section 5 we show our current preliminary results. We discuss our future plans in Section 6 and conclude in Section 7.

Chapter 2

Background

In this chapter we describe the background knowledge required for our project. We start with a detailed description of our optimization problem in Section 2.1, emphasizing several main challenges. In Section 2.2 we give a brief introduction to conventional optimization techniques. We argue that many such techniques are not applicable in our case. Section 2.3 provides a high-level overview of the three key properties in any optimization algorithms. We mainly consider Meta-Heuristics (MH) and Machine Learning (ML) in this project, which seem to be promising for auto-tuning storage systems. We introduce the main concepts of MH and several popular MH techniques in Section 2.4 and the basic ideas in Machine Learning in Section 2.5. In Section 2.6 we provide a summary of the background Section by comparing all the discussed techniques in terms of several key properties.

2.1 Problem Statement

In this section we formally describe our optimization problem in detail. Computers are ubiquitous nowadays; as of 2013 they are responsible for 10% of the world's energy consumption [97]. Having such a huge base, even a small improvement on the performance or energy efficiency would save a lot. Among all the components of computers, the storage subsystem is usually the slowest one, and often "blamed" as the bottleneck of many different applications. Therefore, we feel it is quite important to optimize storage systems. Meanwhile, storage systems are usually deployed with default configurations provided by vendors, due to: (1) it is impossible for the system administrators to know the exact impacts of every system parameter under every workloads; and (2) the default configurations are believed to be "good enough." However, our previous work has shown that even by exploring and modifying a small subset of storage system parameters, we were able to improve its performance or energy efficients by a factor of 1.5–9.4× [318]. If we increase the size of the set of parameters that we are tuning, we may be able to achieve even better improvements. This makes our efforts of optimizing storage systems quite promising. However, it is not easy to accomplish this, due to several intrinsic properties of storage systems.

Our search space for storage systems has several unique and challenging properties:

- **Explosive size**. Modern storage systems are very complex and easily come with hundreds or even thousands of tunable parameters, and this makes it impossible to explore even a small fraction exhaustively and impractical to collecting data for a portion of such a huge space. Even human experts or file system developers cannot know the exact impact of every parameter and thus have no idea how to optimize them. In Table 2.1 we list several examples of the search space for storage systems. Local Storage V1 and V2 are the subset of storage system parameters that we are currently exploring in our experiments. They contain the 7 and 9 most important parameters, respectively, which we picked

based on our experience in storage systems. With only 9 parameters, the search space already contains 24,888 unique configurations. For only one workload, it takes more than one month of clock time to cover each configuration just once. When we start taking NFS into consideration, the number of unique configurations explodes to 1.2×10^{22}, which makes it impossible to finish the exhaustive search within a human lifetime. IBM's General Parallel File System (GPFS) is a parallel, distributed, shared-disk file system [308]. It contains more than 100 tunable parameters, and we plan to use it to prove the accuracy and effectiveness of our auto-tuning techniques later in our project.

System Type	#Total Params.	#Useful Params.	#Unique Useful Configs.	Example Useful Params.
Local Storage V1	7	7	2,074	file system type, inode size, journal options
Local Storage V2	9	9	$24,888$	Local Storage V1 + I/O scheduler, disk type
Local Storage + NFS	52	18	1.2×10^{22}	r/wsize, a/sync, no_/wdelay, NFS version
GPFS [308]	100^{+}	30^{+}	10^{40}	pagepool, maxMBpS, worker1Threads

Table 2.1: Example search spaces for storage systems

- **Discrete and non-numeric parameters**. Modern storage systems usually come with a large set of parameters. Among them, some can take arbitrary integer values, while many others are discontinuous and can only take limited number of values. Some parameter do not even have numerical values. This problem makes the gradient information for the objective function(s) unavailable, and prevents us from applying most gradient-based optimization techniques (e.g., Nelder-Mead Simplex Method [249]).

- **Non-linearity**. In physics and other sciences, a nonlinear system, in contrast to a linear system, is a system which does not satisfy the superposition principle; the output of a nonlinear system is not directly proportional to the input [399]. Figure 2.1 shows the average operation latency of GPFS by only changing the value of the parameter *pagepool* and setting all the others to default. Exp.1 starts from *pagepool* size of 32MB to 2GB and each time double the value. The *pagepool* size of Exp.2 ranges from 32MB to 128MB with a step size of 8MB. Clearly the average latency, which is a good metric of the system performance, is not directly proportional to the *pagepool* size. In fact, through our experiments, we've seen many more parameters with similar properties. This indicates that storage systems are highly nonlinear, which also prevents us from applying several conventional optimization approaches.

- **Multi-modality**. In statistics, a multi-modal distribution is a continuous probability distribution with two or more modes. For a storage system, the search space resulting from its parameters also contains multiple peaks, which makes our optimization work challenging, as the optimization algorithm has to avoid getting stuck in local optima [172]. To make things worse, our search space is very large, sparse, and irregular, which makes it even more challenging to reach the global optimum.

The environment where the storage system is running on can also have large impact on performance and energy efficiency. In this report, we use the term *environment* to refer to the running workload and the underlying hardware where the system is deployed.

Modern storage systems are facing diverse user applications, or workloads, from traditional Web servers to big data processing—and from high-performance computing to virtual desktop infrastructure. Moreover, the performance of storage system configurations depends heavily on the specific workload. Our previous work proved the critical importance of workloads: they drive storage systems to very different states, requiring different optimizations [50, 188, 201, 203, 204, 318, 319, 352, 408]. We list several currently pop-

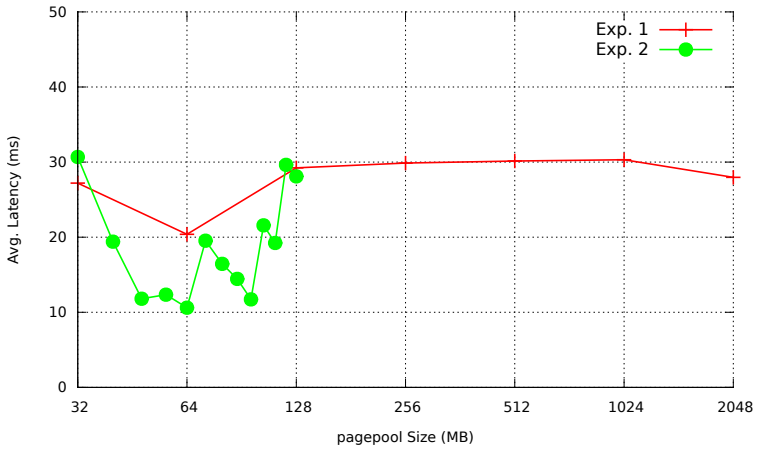


Figure 2.1: Non-linearity property shown in the parameter *pagepool* of GPFS under the DATABASE workload

ular workloads in Table 2.2, which are the main focus of our future experiments. This diversity of chosen workloads is essential to prove the generality of our work.

Storage systems can run on various types of devices: 1) traditional SATA/SAS HDDs; 2) SSDs; 3) shingled drives; and 4) modern Non-Volatile Memory (NVM) such as Phase-Change Memory (PCM). In this project, we use a wide range hardware components to find what hardware combinations are more optimal; configurations for the various workloads are listed in Table 2.2. Our experiments are conducted on several different classes of machines, which are explained in details in Section 4.

Our preliminary results have shown that optimal configurations for different optimization goals are dependent on the specific workloads and hardware. One configuration may work well under one workload, but achieve bad results for another workload. Even a small change in hardware may also deviate the storage system from its optimal outcome. Due to the sensitivity to the workloads and hardware, having this diversity of hardware and workloads in our experiments become fairly valuable. It allows us to measure and compare the effectiveness and efficiency of different optimization techniques and devise our own algorithm for self-tuning storage systems.

2.2 Optimization Approaches

Optimization is everywhere in our life, from simple mathematical function optimization to complex scheduling in daily activities, and from engineering problems to financial marketing. In general, all optimization problems can be summarized formally as follows [413, 416]:

$$x^* = argmin\, f_1(\mathbf{x}), ..., f_I(\mathbf{x}), \quad \mathbf{x} = (x_1, ..., x_d) \tag{2.1}$$

subject to

$$g_j(\mathbf{x}) = 0, \ (j = 1, 2, ..., J)$$
$$h_k(\mathbf{x}) \leq 0, \ (k = 1, 2, ..., K)$$

where $f_1(\mathbf{x}), ..., f_I(\mathbf{x})$ are the optimization objectives, $g_j(\mathbf{x})$ are equality constraints, and $h_k(\mathbf{x})$ are inequality constraints. When $I = 1$, the problem is referred to as a *single-objective* optimization; otherwise it is called *multi-objective*. In the case of storage system optimization, x^* is the global optimal configuration that we are searching for. In this report, we mainly focus on single-objective optimization of storage systems. Later in our project, we plan to deal with multi-objective optimization as well, which we discuss in more details in Section 6.

Workload	Description	Benchmarks
Web Server	Fast lookups and sequential reads of small-sized files; concurrent data and meta-data updates into a single, growing Web log file.	Filebench [104]
Database Server	Large file management; extensive concurrency; random reads/writes; explores the storage stack's efficiency for caching, paging, and I/O.	Filebench [104], SPEC SFS® 2014 [331]
File Server	Each thread explores different home directories; sequence of create, delete, append, read, write, and stat operations; exercising both the meta-data and data paths of the file system.	Filebench [104]
Mail Server	Read mails (open, read whole file, and close); compose (open/create, append, close, and fsync); delete mails; large directory support and fast lookups.	Filebench [104]
Software Build	Mimic running Unix "make" against several tens of thousands of files; file attributes are checked (meta-data operations) and if necessary, the file is read, compiled, then data is written back out to storage.	SPEC SFS® 2014 [331]
Video Data Acquisition	Maintain a minimum fixed bit rate per stream; maintain the fidelity of the stream; provide as many simultaneous streams as possible while meeting the bit rate and fidelity constraints.	SPEC SFS® 2014 [331]
Virtual Desktop Infrastructure	Extracted from traces between the hypervisor and storage when the VMs were running on ESXi, Hyper-V, KVM, and Xen environments.	SPEC SFS® 2014 [331], VMmark [363]
Big Data	(1) *volume*: large data set sizes; (2) *velocity*: high data arrival rates, such as click streams; (3) *variety*: high data type disparity, such as structured data from relational tables, semi- structured data from key-value Web clicks and un-structured data from social media content [121, 372].	HiBench [159], CloudSuite 2.0 [57], BigBench [121], Bigdatabench [372]
Scientific Computing	Mimics the critical computation and data movement involved in computational fluid dynamics and other "typical" scientific computation [370].	MADbench2 [216], SPEC HPG [330], NAS Parallel Benchmarks [248]

Table 2.2: Examples of popular workloads

In addition to classifying based on the number of objectives, there are several other ways to classify optimization techniques. We list several of them here:

- *Gradient-based vs. gradient-free*: gradient-based optimization takes advantage of the gradient information of the objective function and it is often efficient for certain problems; they often try to "walk up" a hill to find an even better solution in a neighborhood where some good solutions are predicted or found. Examples of this type are Hill Climbing [160, 298] and Gradient-descent methods [393]. However, sometimes the gradient information is not available, which means the derivative is hard to compute or is not well defined due to some discontinuity in the function itself. In this case, gradient-free methods such as Nelder-Mead Downhill Simplex method [249] can be more helpful.
- *Local vs. global*: the difference between local and global optimization is whether the algorithm may get stuck in a local optima. A simple gradient-based Hill-Climbing algorithm is a good example of local optimization. This type of algorithm works well if the search space is unimodal. However, as

discussed in Section 2.1, our search space is multi-modal, which limits us to only global optimization algorithms. For example, if we randomly pick one configuration and re-start the Hill-Climbing process after each iteration, it becomes a global optimization algorithm. This is often referred as Random Restart Hill Climbing [160].

- *White-box vs. black-box*: white-box optimization is the set of algorithms that tries to construct a mathematical model describing the internal properties and structures of a system. For example, Xi et al. applied fuzzy control theory to optimize the *MaxClient* parameter for Apache Web Server to achieve minimal response time [410]. They model the relationship between the parameter and the response time as a simple convex function. In contrast, black-box optimization assumes no a priori knowledge of the system. It queries the system for the values of optimization functions for certain configurations through pre-defined system interfaces. It knows nothing about the internals of the system, such as the gradient information of the optimization functions, and it makes no assumptions about the analytic forms for the functions. Sometimes people prefer to exploit both white-box and black-box optimization at the same time, and we call this kind of methods as "gray-box." White-box optimizations can be more accurate because they better understand a system's internal structure; on the other hand, however, they require more understanding of complex system internals, and that information may not be easily available if at all.

Although there are many optimization techniques, not all of them are applicable for auto-tuning storage systems. Based on the properties of our problem described in Section 2.1, we argue that some popular and conventional optimization methods cannot easily apply to storage systems. Note that while some of these techniques on their own cannot solve our problems, we can still leverage them in part to improve the overall optimization problem.

- *Exhaustive Search*: exhaustive search enumerates every possible combination of parameters and tries each configuration one by one. However, as we have a huge search space and each single evaluation of configuration takes a relatively long time for storage systems, it is generally not applicable in our case. For example, the average running time for the mailserver workload in Filebench [104] is about 3 minutes. In our experiments, Storage V2 contains 9 parameters and totally 24,888 unique configurations. This takes us around 52 clock days to exhaustively run each configuration on a single machine. When we added in NFS, and consider only a subset of 18 useful parameters, as shown in Table 2.1, it takes 10^16 years of time to try every permutation of parameters. This is impossible to finish within a human lifetime.

- *Monte Carlo Methods*: Monte Carlo methods (or Monte Carlo experiments) are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results, and are mainly applied in three categories of problems: optimization, numerical integration, and generating draws from a probability distribution [106, 110, 155, 174, 292]. However, due to the huge search space, pure random sampling would not be efficient enough, as the probability for hitting the global optimal configuration is very low. Note that the idea of random sampling is still helpful in our optimization problem, which we explain more in Section 6.

- *White-Box Optimization*: as explained above, white-box optimization techniques need to understand the system internals, and they usually assume some a priori knowledge about the systems. They are applicable for optimizing simple systems, for example only optimizing one parameter for the Web server [410]. However, due to the complexity of modern storage systems, and the sensitivity of good configurations to the environment, it is nearly impossible to fully understand their behaviors, even for experienced file system developers and administrators. This fact makes white-box optimization inapplicable for our problem. Our optimization method is actually a "gray-box" one. It is mainly

based on black-box optimization, which assumes no a priori knowledge of the underlying system. However, as we already partially understand the internals of storage systems, this kind of knowledge is also included in our optimization algorithm. That is why we call it "gray-box."

- *Control Theory*: Control theory (CT) was used historically to manage linear system parameters. CT builds a controller for a system, called the plant, so its output follows a desired control signal, called the reference [157, 197]. CT has been applied to database systems [77], storage systems [179, 198], Web servers [76, 213] and data centers [206, 373–375] to provide QoS guarantees; often a small number of parameters were optimized. Our previous work used CT in the past to model the energy consumption and performance of computing systems [202, 203]. Alas, we found CT unsuitable to solving the problems of this proposal for several reasons. (1) CT is unstable in controlling non-linear systems; (2) CT cannot handle non-numeric parameters; (3) CT requires a learning phase, called *identification* to build a good controller, which requires having lots of data to study; (4) CT is designed to control a small number of parameters at a *set point*, not to find the best fitness (e.g., throughput); and (5) CT has variants that attempt to handle digital and non-linear systems, by segmenting the search space into smaller subspaces, but this scales poorly.

2.3 Key Properties

After investigating several different techniques, we find that nearly all search algorithms work by maintaining a trade-off among three properties: *exploration*, *exploitation*, and *history*. This trade-off instead controls the effectiveness and efficiency of the search process. We define the three key properties as follows:

- **Exploration** is how much the technique searches the space randomly. This often includes a combination of pure and guided random search in a given neighborhood.
- **Exploitation** is how much the technique leverages its neighborhood. When in a certain configuration, we try to find nearby better ones and climb them.
- **History** is how much data from previous generations is kept. History information can be used to guide future exploration and exploitation.

Figure 2.2 illustrates how a simple search algorithm searches for the highest fitness peaks (e.g., best performance) as quickly as possible. The figure greatly simplifies the real search spaces in two ways. (1) It is 3D whereas real search spaces are many-dimensional. (2) The actual distance between sets of peaks (e.g., P1+P2 vs. P3+P4) can be vast.

Say you start at a random point near the front corner: the fitness (Z axis) is low. Next you make a move to find a better area to explore. If you take step 2a, you may find yourself going in the wrong direction for a long time. With enough exploration, however, or even a guided one, you will eventually hit on a better fitness value than you currently have, and then be able to exploit it. If you take step 2b, you will be at the foothill of P1: you can look in your nearby vicinity and be able to climb to the top of P1, in step 3. Next, we look around; everything will seem lower than P1: so one important criteria is *not* to get stuck on local maxima, P1. If you just exploit your local area, you may never get out of P1 to a nearby higher peak (P2), let alone get to the highest, yet distant peak (P4). So *both* exploration and exploitation are needed.

With a careful balancing of exploration and exploitation, you may be able to get out of P1, go via local minima L1, and find a path to the top of P2, a better peak than P1. To reach from P1 to P2, one needs to (randomly) explore nearby. However, to find even higher peaks that are far away requires taking bigger changes to reach distant areas of the search space. Exploring from P1, you can easily find yourself in dead space. But if you find yourself near peak P3, then you can use both exploration and exploitation to climb P3; and from there reach to P4, a higher peak than even P2.

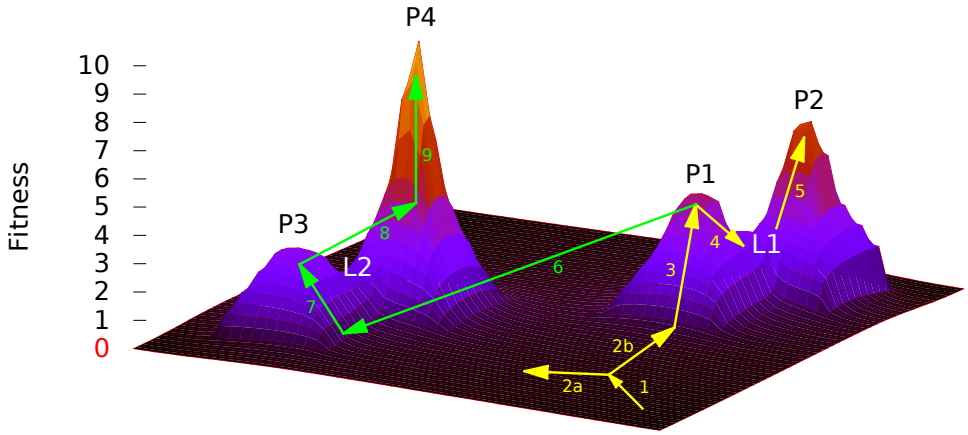


Figure 2.2: Simplified Terrain Figure for the Parameter Space

We can see from this simplified search process that neither exploration nor exploitation alone suffice. The balance between these two properties largely determines the search results.

History information has been included in several search algorithms. A good example is Tabu Search (TS), which we will introduce in details in Section 2.4.6. The ideas of the Tabu list, intensification, and diversification all rely on the history information maintained. The Smart Hill Climbing algorithm proposed by Xi et al. [410], also uses history to update the weights in the Weighted Latin Hypercube Sampling, so that promising configuration are more likely to be picked for evaluation. The amount of history kept also matters. If the algorithm stores too much history, it wastes memory and storage: thus it will need more time to process the history and get useful information. On the other hand, if we maintain to little history, it may be of little help in improving the search performance. As an extreme example, traditional Q-Learning algorithms maintain all the history information in one table, which greatly limit their scalability and practicality to large spaces.

2.4 Meta-Heuristics

This section covers a set of techniques called Meta-Heuristics(MH). We start by introducing the definition of MH in general. From Section 2.4.2 and onwards, we cover several popular MH techniques that proved effective in solving certain optimization problems. Our hope is that this survey of techniques can give us insights into the essence of those techniques, and help us develop our own algorithms for storage system optimization.

2.4.1 Meta-Heuristics

Meta-Heuristics(MH) techniques [34, 119, 128, 215, 243, 369, 394, 396, 413, 416] are mostly developed to solve difficult optimization problems, for which conventional optimization algorithms are not effective or

even not applicable at all. Most of techniques are nature-inspired as they have been developed based on the successful evolutionary behavior of natural systems—as nature has solved many difficult problems through billions of years of evolutionary history. People have been vague in the exact definition and scope of MH. For example, the term *heuristic* has been referred to as

> a technique designed for solving a problem more quickly when classic methods are too slow, or for finding an approximate solution when classic methods fail to find any exact solution, which is achieved by trading optimality, completeness, accuracy, or precision for speed [394].

In this report we follow Glover's definition, who referred *heuristic* as "to find or to discover by trial and error" and *meta-* as "beyond or higher level" [413]. Simply put, MH means techniques that would generally outperform simpler heuristics. Glover gave a formal definition of MH as "a MH technique is a master strategy that guides and modifies other heuristics to produce solutions beyond those that are normally generated in a quest for local optimality" [129]. Nearly all MH techniques make some trade-off in the extent of *local-* and *global search*. In this report, we refer these two characteristics as *exploitation* and *exploration*, respectively, which we already discussed in Section 2.3. In the following sub-sections, we introduce and discuss several popular MH techniques. We mainly focus on Simulated Annealing and Genetic Algorithms, as they are two of the most popular MH techniques. We briefly cover the basics for several other MH algorithms as well.

2.4.2 Simulated Annealing

Simulated Annealing (SA) [2, 27, 48, 183, 294, 358, 358, 403] is a meta-heuristic for approximating the global optimum in a large search space, which is inspired by the annealing technology in metallurgy. Annealing involves heating and controlled cooling of a material to get to the state with minimum thermodynamic free energy. The rate of cooling determines the magnitude of decrease in the thermodynamic free energy, with slower cooling producing a bigger decrease [403]. SA was proposed by Kirkpatrick et al. [183] in 1983 and Černỳ [48] in 1985, and it is an adaptation of the Metropolis-Hastings algorithm, which is a Monte Carlo method to generate sample states of a thermodynamic system invented by Metropolis et al. [232] in 1953. It has been proved that SA can be mathematically modeled using the theory of finite Markov chains [1, 2].

Algorithm 1: Simulated Annealing

Input : initial temperature T_0, initial state $initial()$, neighborhood function $neighbor(s)$, cooling schedule $temperature(t)$, number of iterations with same temperature $iteration(T)$, acceptance probability distribution $prob(F, F', T)$, stopping criteria $stopping(s, T, t)$, evaluation function $evaluate(s)$

Output : final state s_f

```
1  s ← initial();
2  T ← T_0;
3  t ← 0;
4  while not stopping(s, T, t) do
5  |   N ← iteration(T);
6  |   for i ← 1 to N do
7  |   |   s' ← neighbor(s);
8  |   |   if prob(evaluate(s), evaluate(s'), T) ≥ random(0, 1) then
9  |   |   |   s ← s';
10 |   |   end
11 |   end
12 |   T ← temperature(t);
13 |   t ← t + 1;
14 end
15 s_f ← s;
```

Algorithm 1 gives the pseudo-code for SA. As we shall see, the *evaluation function* is required by nearly every MH algorithm, as one basic property for MH, and more generally, black-box optimization, is *trial and error*. Note that different MH techniques usually have a different name for the output of the evaluation functions. For consistency, throughout the report we will call this the *fitness* value and thus we sometimes also refer to the evaluation function as a *fitness function*. In the algorithm, s is the current state and t acts as a virtual clock to guide the temperature reduction process. Besides, there are several important inputs for SA, which control and determine the effectiveness and efficiency of the algorithm. We discuss them next.

Initial state. In most cases SA randomly picks one state from the search space as the starting point of the algorithm. However, the effectiveness and efficiency of SA depend heavily on the choice of the starting point. Although SA is proved to be able to find the global optimum given enough time, regardless of where the starting point is, we cannot afford running the algorithm for an infinitely long time. We want the algorithm to converge fast but still find a good enough configuration. In this case, if we can start from a configuration which is far away from "bad areas," SA is more likely to result in a better final state. This is actually part of our future work, which we discuss more in Section 6. One simple example is to exploit some expert knowledge. Although human experts cannot fully understand all the behaviors of storage systems, they may have some basic ideas on what parameters and values are more likely to have large impact. These can act as good candidates from the initial states of SA. Another idea is to run some statistical-sampling methods before actually running SA itself. This actually allows us to get an overview of parts of the search space and be able to choose a good start point. Much work have been done in choosing a good start point for SA in order to accelerate its convergence time [150, 169, 360]. Moreover, some statistical methods can be applied before SA begins running, so as to start with a better state. For example, Latin Hypercube Sampling (LHS) [227, 395] has been applied in a Smart Hill-Climbing algorithm for find optimal configuration for Web servers [410]. It is applicable in SA as well. How to choose a good start point is definitely related to the search space itself, and we plan to investigate and try various methods to see if they work well for storage

system optimization.

Initial temperature. The choice of *initial temperature* is also important, and it should encourage more exploration. If the *initial temperature* is too low, then the state space cannot be explored enough, and we might miss finding many potentially better solutions. However, it is known that too high *initial temperature* may cause a long computation time or bad performance [261]. Various methods for estimating a proper initial temperature have been proposed:

- Kirkpatrick et al. [183] suggested taking

$$T_0 = (\Delta F)_{max}, \tag{2.2}$$

  where $(\Delta F)_{max}$ is the maximum difference of fitness values between any two neighboring states.

- Johnson et al. [169, 170] proposed to compute the *initial temperature* as shown in Equation 2.3.

$$T_0 = -\frac{\overline{\Delta F}}{\ln^{p_0}}. \tag{2.3}$$

  Here, $\overline{\Delta F}$ is the estimation of the fitness value increase after a strictly positive move, which means the neighbor state is better than the previous state. This value is estimated by randomly generating some moves [25]. p_0 is the initial acceptance rate that could be set.

- A more precise estimation was proposed by White [2, 377].

$$T_0 = K{\sigma_\infty}^2, K \in [5, 10] \tag{2.4}$$

  Here, K is a constant and ${\sigma_\infty}^2$ is the second moment of the fitness distribution when the temperature is set to ∞. The fitness distribution is estimated by randomly sampling a few states.

- When some methods for choosing a good *initial state* are already applied, people tend to start with a relatively lower *initial temperature* [150, 169, 360]. It has been proved that sometimes SA could be wasting time at the highest temperatures [169]. For example, J. Varanelli chose to set $T_0' = 0.33T_0$, where T_0 is the original estimation for the *initial temperature*, so as to accelerate the SA process.

- Park and Kim [261] proposed to use a simplex method to find proper parameter settings for SA, without much human intervention. This acts like a "heuristic beyond heuristic," as it uses one optimization algorithm to find the parameters for another optimization algorithm.

It is not easy to find the best *initial temperature* for optimizing storage systems, as the proper value also depends on the specific problem [183, 261]. Some problems may convergence faster, and thus only need a lower *initial temperature*; while others must have a high temperature in order to explore the space thoroughly. We plan to experiment with various values and methods to find the best way for our problem.

Neighborhood function. A *neighborhood function* is defined to list all possible s' states that can be reached from the current s: this allows SA to limit its exploration to a desired horizon. In the case of SA, one call to the neighborhood function returns only one neighbor state. The neighbors are produced by altering the current state in some well-defined way; one such alteration is usually called a "move." The moves of SA should result in minimal alterations of the last state in order to help the algorithm keep the better parts of the solution and change only the worse parts [403]. For example, in our project we cam simply define one move as just altering the value of one parameter. We can even define dynamic neighborhood functions that initially

change several parameters by a large amount; in later iterations, our function can be more conservative and change only one parameter slightly. For our parameter space, we need to investigate how to define the neighborhood of non-numeric parameters such as the I/O scheduler algorithm: one possibility is to consider their chronological release date to approximate a rough order (assuming that newer schedulers work better). To choose among a set of neighbors returned by the neighborhood function, we may consider a totally random selection or one coupled with a preference to configurations that have not been explored recently.

Cooling schedule. A *cooling schedule* is sometimes also called an *annealing schedule*. It decides how quickly the temperature of the system drops over time, until it is low enough to stop searching. The temperature determines the probability of choosing a "worse" neighboring state. When the temperature is lower, the probability of choosing a worse state is lower. If the temperature reduces too quickly, then we may not explore the search space enough; if the cooling rate is too low, we would spend too much time on exploring the space and not converge on a solution fast enough. Many different cooling schedules have been proposed and compared [59, 251]. We describe several of them here:

- *linear cooling schedule* [183]. In a linear schedule, we have:

$$T = T_0 - \beta t, \tag{2.5}$$

  where T_0 is the initial temperature and t is the pseudo time to record the number of times for temperature reduction. β is the cooling rate and should be picked carefully. Let's assume the final temperature is T_f and the final time is t_f. A good value of the cooling rate β should satisfy that when $t \to f_f$, we have $T \to T_f$. Thus, if we can set the values of t_f and T_f in advance, it gives:

$$\beta = \frac{T_0 - T_f}{t_f}. \tag{2.6}$$

- *geometric cooling schedule* [183]. A geometric cooling schedule decreases the temperature by a cooling factor α, where $0 < \alpha < 1$. In this case, we have:

$$T(t) = T_0 \alpha^t, \quad t = 1, 2, ..., t_f. \tag{2.7}$$

  One advantage of the geometric cooling schedule over the linear one is that there is no need to predefine the final time t_f, as when $t_f \to \infty$, we have $T \to 0$. In reality, the geometric cooling schedule is more often chosen, and a value $\alpha \in [0.7, 0.99]$ is commonly used [413].

- *logarithmic cooling schedule* [117]. In a logarithmic schedule we have:

$$T(t) = \frac{c}{\ln^{t+d}} \tag{2.8}$$

  where d is usually set to 1. Hajek proved that under this cooling schedule, the condition for SA convergence is that c be greater than or equal to the depth, suitably defined, of the deepest local minimum which is not a global minimum state [154]. However, due to its asymptotically slow temperature decrease, this cooling schedule is impractical. It simply counteracts the exponential Boltzmann acceptance function in SA and amounts to a purely random search in the huge space [251].

- *inversely linear cooling scheduling* [342]. Due to slow temperature decrease rate in logarithmic cooling schedule, Szu and Hartley proposed Fast Simulated Annealing (FSA), which has:

$$T(t) = \frac{T_0}{t+1} \tag{2.9}$$

  They proved FSA with an inversely linear cooling schedule converges faster than the classical simulated annealing proposed by Geman and Geman [117].

- *adaptive cooling schedule*. The aforementioned four cooling schedules are all static, in the sense that they are pre-determined and the temperature reduction process are not influenced by the actual search process. In contrast, many adaptive cooling schedules have been proposed [7, 8, 12, 309, 353]. For example, Azizi and Zolfaghari [12] proposed a cooling schedule as shown in Equation 2.10:

$$T(t) = T_0 - \lambda \ln^{1+r_{t-1}} \tag{2.10}$$

where λ is the coefficient that controls the temperature reduction rate and r_{t-1} is the number of consecutive upward moves during the last temperature value.

In our project, we are not limiting ourselves to just one simple cooling schedule. We are trying and comparing all the cooling schedules discussed in this section. This allows us to understand how different cooling schedule would impact the optimization for storage systems and find or devise the most suitable cooling schedule for them.

Iterations. For a given temperature, usually multiple evaluations for the fitness function are required. We call these "multiple evaluations" as *iterations*. We define this value as a function of the current temperature, as it does not need to be a fixed value during the whole algorithm. For example, we may want to increase the number of *iterations* for a lower in order to fully explore the nearby space. *iterations* can also have impacts on the effectiveness and efficiency of the final solution, and we need to carefully balance the choice of *iterations* and the solution quality. If we have too few *iterations*, we may not be able to explore the search space enough, i.e., not giving high-enough probability for worse configurations. However, if we try too many *iterations*, we may spend too long time for each temperature value and thus cause SA to converge very slowly.

Acceptance probability. The *acceptance probability distribution* defines the probability of accepting a neighboring state of the current one. It takes 3 arguments: the fitness value F of the current state s, the fitness value F' of one neighbor state s', and the current temperature T. Generally there are two types of acceptance probability distributions:

- *Metropolis Criterion*. In the initial version of SA [48, 183], the probability would equal to 1 when $F' > F$, which means the neighbor state is better than the current state. Most work on SA followed this rule, though it is not essential in order for SA to work [413]. When $F' < F$, we have:

$$prob(F, F', T) = e^{-\frac{\gamma(F'-F)}{k_B T}}, \tag{2.11}$$

where k_B is the Boltzmann's constant and γ is a real constant (for simplicity $\gamma = 1$). From Equation 2.11 it is obvious that when T is large, the probability for accepting worse states is high. This means that when the temperature is high, SA tends to explore the search space more and avoid getting stuck in local optima by allowing a relatively high probability for accepting bad configurations. When $T \to 0$, $prob \to 0$, which means SA tends to be in a more stable state and only do hill-climbing when the temperature is low.

- *Barker Criterion*. Another acceptance criterion, which arose naturally in the context of Boltzmann machines [1], is the Barker criterion [17] given by:

$$prob(F, F', T) = \frac{1}{1 + e^{\frac{\gamma(F'-F)}{k_B T}}}. \tag{2.12}$$

The meanings for the symbols in Equation 2.12 is the same as in Equation 2.11.

The acceptance probability distribution is one of the most important part of SA, as it controls how SA gets away from local optima and explores the search space efficiently.

Stopping criteria. It has been shown that by properly setting the parameters, SA is guaranteed to converge given enough time [154], and there is a theoretical upper bound for the convergence time [237, 279]. However, based on our preliminary results on SA experiments, as shown in Section 5, we feel that some stopping criteria have to be included in the algorithm. Sometimes after running for a while, SA spend a long time exploring the space but not improving the solution even a little bit. In this case it is difficult to tell whether the current solution is the global best or SA just gets stuck in local optima and needs more time for exploration. Many times we do not even allow such a long running time for SA. One simple example is using a time-window based method. If SA fails to improve the current solution within a certain time, we decide to stop SA. However, we definitely need more intelligent stopping criteria, in order to improve the effectiveness and efficiency of SA.

More recently, SA has been extended to allow parallel execution of the algorithm to solve complex optimization problems [1, 111, 219, 280]. SA has been applied in various areas and proved efficient in solving different types of problems, including Job Scheduling Problem (JSP) [185, 359], Travelling Salesman Problem (TSP) [1, 223, 358], graph partitioning and coloring [169, 170], University Course Timetabling Problem (UCTP) [54, 187, 200, 247], Very Large Scale Integration (VLSI) design [317, 406], vehicle routing [257], portfolio selection [63], power system optimization [221], radio channel assignment [86], network design [109, 111, 167], process mapping [256], software architecture [278]. It has even been applied to storage systems for before [83, 116]. However, our work is more general and is supposed to work for various optimization goals in storage systems.

2.4.3 Genetic Algorithms

In this section we provide a detailed introduction to Genetic Algorithms (GAs) [70, 73, 132, 163, 234, 324, 325, 332, 378]. GA is a nature-inspired algorithm and was first proposed in 1975 by Holland [163] and De Jong [73] in 1975 to build adaptive systems. De Jong was the first to apply GA to optimization problems, although later he argued that GA is not a really function optimizer, and that the success of GA in optimization is in some ways incidental to adaptation. Nevertheless, GA has become quite popular in solving complex optimization problems in various domains [132], and optimization is now the main focus and application area for GA [119, 128].

GA actually belongs to a category of algorithms called *Evolutionary Algorithms* (EAs). We start by covering some biological background that is necessary for understanding the internals of GA, as well as other EAs. We then provide a summary and comparison of different EAs. The major part of this section introduces GAs, including the key operators and characteristics that decide the effectiveness and efficiency of GAs, and the theory on why GAs work. We end with the list of applications of GA in various areas.

2.4.3.1 Biological Background

We start with a brief introduction to some of the key concepts taken from Genetic Biology. We are not intending for formal and accurate definitions for these terminologies here; this introduction is primarily intended to help understand how GA originated and how it works. The word "genetics" originates from the Greek word "genesis," which means "to grow" or "to become" [325]. *Genetics* usually refers to the study of genes, heredity, and genetic variation in living organisms [390]. The terminologies related to GA are as follows:

- *Chromosome*. *Chromosomes* are strings of DNA and serves as a model for the whole organism. All the genetic information is stored in the *chromosomes* [325].

- *Gene*. *Chromosomes* are divided into several parts called *genes*, which encodes a functional RNA or protein product and controls the properties of a species [388]. Each *gene* has its own position in the *chromosome*. This position is called the *locus*.

- *Allele*. An *allele* is one of all the alternative forms of the same *gene* [382].

- *Trait*. A *phenotypic trait*, or simply *trait*, is a distinct variant of a phenotypic characteristic of an organism that may be inherited, be environmentally determined or be a combination of the two [401]. One *trait* corresponds to one *allele*. Take an eye color for example. Blue is one of the *traits* for the eye color, while the underlying specific *gene* value (string of DNAs) that directly decides the blue eye is called an *allele*.

- *Genome*. A *genome* is the complete set of DNA of an individual, including all of its genes [391].

- *Genotype*. The *genotype* is part of the genetic makeup of an individual [392]. It can be a simple description of which alleles an individual has for a certain gene, or it might be a description of alleles for a number of genes.

- *Phenotype*. A *phenotype* is the composite of observable characteristics or *traits* of an individual, and it is decided by the corresponding *genotype* and also the environment [400]. A *genotype* and a *phenotype* are distinguished by the source of an observer's knowledge: one can know about genotype by observing DNA or *genes*, whereas one can know about phenotype by observing outward appearance of an organism.

- *Natural Selection*. Natural selection is the differential survival and reproduction of individuals due to differences in phenotype. It is a key mechanism of evolution, and it was popularized by Charles Darwin [67]. The key principle of natural selection is individuals with some traits that can give these individuals a reproductive advantage are more likely to survive [398].

- *Fitness*. *Fitness* describes the reproductive success of an individual [387]. That *alleles* with greater positive effects on individual fitness will become more common over time, is actually the central thesis for *natural selection*.

- *Crossover*. *Crossover*, or *chromosomal Crossover* is the exchange of genetic material between chromosomes with same genes in the same loci, which results in recombinant chromosomes during sexual reproduction [383].

- *Mutation*. In biology, a *mutation* is a permanent change of the *alleles* of one or more *genes* of an individual [397].

2.4.3.2 Evolutionary Algorithms

GA belongs to a set of algorithms called Evolutionary Algorithms (EAs), which includes all generic population-based Meta-Heuristic optimization algorithms inspired by adopting Darwinian principles [94, 385]. Nearly all EAs can be summarized as the flowchart in Figure 2.3.

An EA uses mechanisms inspired by biological evolution, such as reproduction, mutation, recombination (crossover), and selection. Candidate solutions to the optimization problem play the role of individuals in a population, and the fitness function determines the quality of the solutions. Evolution of the population then takes place after the repeated application of the above operators until some stopping criteria are satisfied [385]. There are four major types of EAs: GA, Genetic Programming (GP) [120, 389], Evolution Strategies [96,312,384], Evolutionary Programming [108,386], and the differences between them lie mainly

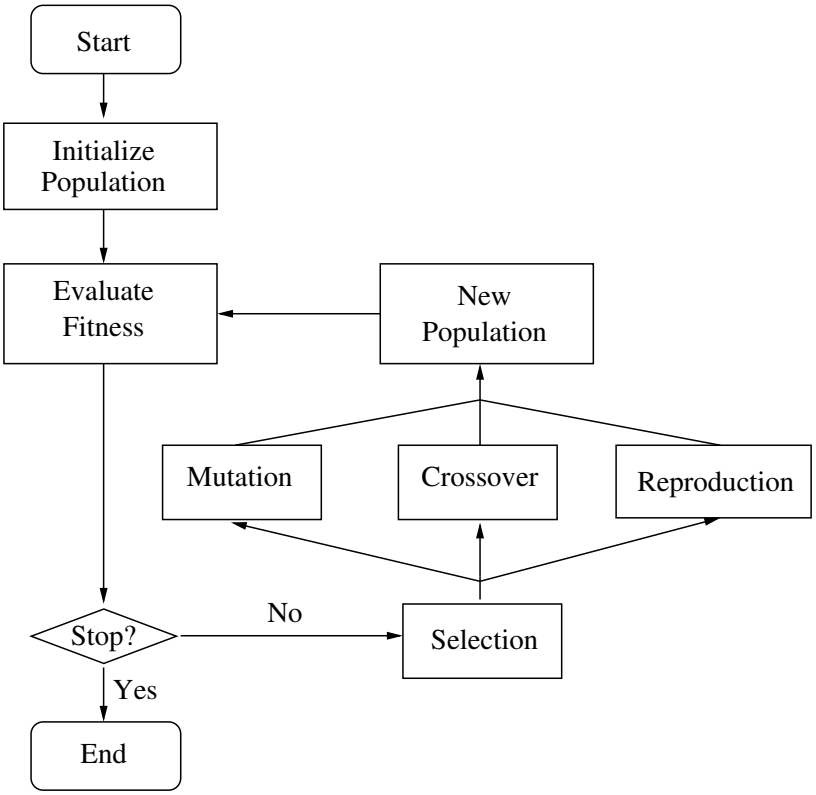


Figure 2.3: Flow chart of an Evolutionary Algorithm

in the problem representation, evolution operators and also the goal of the algorithm. This topic is not the main focus of this report; we just want the clarify the confusion between several similar terminologies. This four categories of techniques differ in the following aspects:

- *Goal*. This means that they are devised to solve different types of problems. For this reason, we believe GA is the best fit for our project among these techniques.
- *Selection*. The *selection* method decides how to choose solutions from the current generation as parents to produce children as new candidate solutions.
- *Representation*. The way all these Evolutionary Algorithms work is to encode information into a pre-defined format, which mimics the concept of *chromosomes* in Biology.
- *Recombination*. This is also known as *crossover*. It is the main method of the algorithm to generate new candidates for solutions by combining the good parts of previous ones.
- *Mutation*. This is to add some degree of randomness into the algorithm, and randomly change part of one current solution to get a new candidate.

Based on these five concepts, we briefly summarize the differences of the four aforementioned techniques in Table 2.3.

2.4.3.3 Simple Genetic Algorithm

Here we discuss several important aspects of GAs by using a Simple Genetic Algorithm (SGA) as an example, which is the basic form for all types of GAs. There are many other variants of GAs [325], and we will

Technique	Goal	Representation	Selection	Recombination	Mutation
Genetic Algorithms (GAs)	Originally for adaptive systems, now mainly for theoritical and real optimization problems; works for both discrete and continuous parameters	Fix-length bit string	Fitness-proportional	1-point crossover (and many others)	Bit-flip mutation
Genetic Programming (GP)	Find computer programs that perform best for a user-defined tasks	Variable-sized tree	Fitness-proportional	Subtree exchange	Randomly change one node in the tree
Evolution Strategies	Typically used for continuous parameter optimization [94]	Fix-length real-valued vector	Uniform random	*Intermediate recombination* (average of two parents)	*Gaussian mutation* (a random value from a Gaussian distribution is added to each element of the vector)
Evolutionary Programming	Originally developed to simulate evolution as a learning process with the aim of generating artificial intelligence [94]	Fixed-length real-valued vector	Deterministic (each parent will generate one offspring via mutation)	None	*Gaussian mutation*

Table 2.3: Comparison of four major EAs

introduce them later on. Algorithm 2 presents the pseudocode for SGA. Here for simplicity, the evaluation function takes the whole population as the argument. It evaluates all the individuals in the current population and returns a list of fitness values for each individual. t is the generation number, and in the end the algorithm returns the best individual in the last generation.

Algorithm 2: Simple Genetic Algorithm

Input : Initial population $initial()$, population size $population\_size(P, t)$, elitism method $elitism(P, F)$, selection method $selection(P, F)$, crossover method $crossover(s_1, s_2)$, mutation method $mutation(s)$, mutation rate $muatation\_rate()$, stopping criteria $stopping()$, evaluation function $evaluate(P)$

Output : Best individual s_f

```
P ← initial();
F ← evaluate(P);
t ← 1;
while not stopping() do
    p_size ← population_size(P, t);
    P' ← empty_list();
    P'.append(elitism(P, F));
    while size(P') < p_size do
        parent_1, parent_2 ← selection(P, F);
        child_1, child_2 ← crossover(parent_1, parent_2);
        if mutation_rate() ≥ random(0, 1) then
            child_1 ← mutation(child_1);
        end
        if mutation_rate() ≥ random(0, 1) then
            child_2 ← mutation(child_2);
        end
        P'.append(child_1);
        P'.append(child_2);
    end
    P ← P';
    F ← evaluate(P);
    t ← t + 1;
end
s_f ← P[max_index(F)];
```

As we see from Algorithm 2, GAs have several important features that are directly taken from Genetic Biology. These terminologies actually have similar meanings, and this is how GAs mimic the process of nature evolution. The idea is that evolution has been proved successful in producing us humans and other advanced creatures on Earth, then why not exploit these tried-and-true techniques for complex optimization problems?

Next we describe the important features in GAs in details. To help relate these concepts to our problem domain, we will use examples from storage systems.

Gene. A *gene* represents one single configuration parameter, such as the *file system type* and *disk type* in our Storage V2 (see Table 2.1). A specific value of a gene is an *allele*. Depending on the type of corresponding parameters, genes can take on different forms [310]:

- *Binary*. Binary genes can be encoded as 1 single bit in the chromosome.
- *Non-ordered discrete*. This type of genes can take on more than 2 alleles, but a limited number of alleles. Here *non-ordered* means that the alleles of the same gene are not comparable. One example of this category is the *file system type*: there's no logical ordering of, say, XFS vs. Btrfs vs. Ext4.

- *Ordered discrete*. Similar with the previous type of genes, *ordered discrete* type can take on a limited number of alleles. In this case, however, the alleles of the same gene *can* be compared. One example is the *block size*.

- *Real-valued*. We put all the other parameters, which can take on any integer and real numbers into this category. One example of this category is the *tcp_keepalive_time* in Linux.

Chromosome. In our problem, a *chromosome* corresponds to a single configuration. A configuration is defined as a precise series of parameters (*genes*). A chromosome can be as long and complex as needed to encode more parameters. Normally in GAs, chromosomes are represented as fixed-length bit strings, meaning that all configurations have the same number of parameters. One specific problem that arises when applying GAs to optimizing storage systems is that many file systems have options unique to just that file system. For example, the *notail* mount option is unique to Reiserfs and has a significant performance impact [318]. Such options make sense only for the file systems that support them, and this may cause some invalid configurations within our search space, which is regarded as a potential source of difficulty for applying GA to real optimization problems [286].

Fitness. In GAs, each configuration is evaluated for its fitness in a given environment. In our problem, fitness can take on various forms, such as I/O throughput, CPU speed, energy used, and complex cost functions thereof [204]. As long as the values returned by the fitness function are comparable by a GREATER-THAN relation, they can be defined as the *fitness*. If we were to model the multi-dimensional parameter space of a complex storage system and project it as a 3D figure, the fitness landscape would be a very sparse terrain. The landscape would have a few clustered peaks with several local maxima. These clusters would be separated by vast stretches of "dead" spaces. This actually makes our optimization more challenging, as we need to avoid spending too much time in those dead spaces and get to the global optima quickly.

Population. *Population* refers to the set of unique configurations (chromosomes) being evaluated. The larger the population, the more configurations can be evaluated. Individual members of a population can be evaluated in parallel. A population can grow or shrink over time.

Generation. The fitness of each population member is evaluated in the given environment. This is called one single *generation*. GAs produce the next generation by applying the pre-defined genetic operators on the individual members of the current generation.

Initialization. GAs usually take an initial population as the algorithm input. The quality of the initial population has a large impact on the convergence time and final results of GAs. For choosing an initial population, it is often assumed that initialization of GAs should be random when GAs were first proposed [119]. Later, various methods on how to pick the initial population were proposed, and they can be generally classified into three categories:

- *Sampling*. Rees and Koehler [283] used the theoretical model proposed by Vose [366] to demonstrate that sampling without replacement is preferable when the population size is relatively small. Reeves [119] suggested that more sophisticated statistical sampling methods, which can cover the search space more uniformly, are more advantageous than purely random sampling.

- *Including good solutions*. It has been shown that including good solutions obtained from other heuristic techniques can help GAs find better solutions more quickly [4, 175]. Bramlette [38] used the best of n randomly chosen individuals as the initial population for GAs. Whitley et al. [379] iteratively

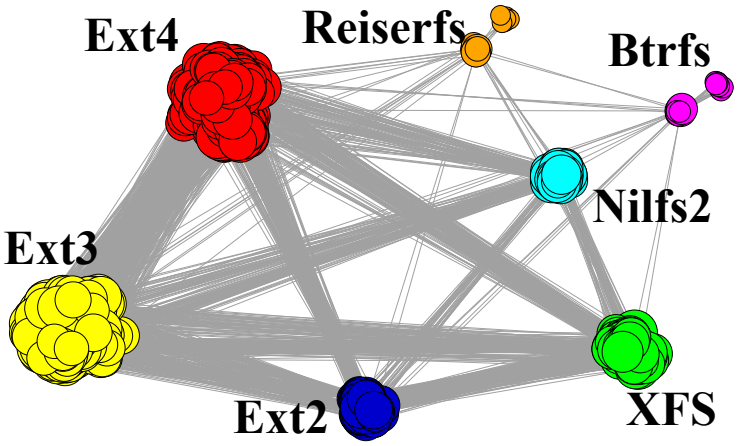


Figure 2.4: Configuration connectivity figure of Storage V2.

used best solutions from previous runs as a basis for altering the chromosome representation during restarts.

- *Domain knowledge*. Grefenstette [148] first proposed to incorporate problem-specific knowledge into the GAs, including seeding the initial population. Schultz and Grefenstette [311] and Gordon [142] applied GAs in learning decision rules, and showed the effectiveness of choosing an initial population with domain-specific knowledge. In case of a changing environment, Ramsey and Grefenstette [282] proposed *case-based initialization*. It includes strategies, which were learned under similar environments, into the new initial population for GAs.

However, these specialized initialization methods have the possibility of inducing *premature convergence*, which means most chromosomes in the population have similar allele values; applying crossover methods to similar chromosomes only results in another similar chromosome, and new areas of the search space are left unexplored [119, 175, 199].

Certainly, how to initialize the population is important for optimizing complex storage systems. Our initial experiments showed that this could lead to wasted time exploring poor choices. To make things worse, we found that sometimes, certain file systems were rarely explored.

In Figure 2.4 we present the connectivity figure of the whole Storage V2 search space. Each node in the figure represents one single configuration from Storage V2. Each edge represents whether two configurations can be reachable from each other with just changing the value of one parameter. We can see that file systems such as Ext3 and Ext4 not only have many possible configurations to explore, but are also very well connected to each other and other file systems. It means that a uniform random set of configurations is likely to bias towards these well-connected file systems. Conversely, file systems such as Reiserfs and Btrfs are poorly connected and hence much harder to reach via mutation; those file systems rarely got explored in our experiments when we used a random initialization, even if they produced superior performance. As one simple work-around for this problem, we experimented by initializing our population and allocating each file system a proportional share of the slots. This allowed weakly connected file systems to be explored to some extent. We plan to investigate other methods as well, such as *statistical sampling methods* and *domain knowledge*.

Population size. Another aspect of the population is its size. A smaller population would result in inadequate exploration of the space, while a larger population will induce high computation costs. Goldberg first

proposed a theory on how large the population size should be, and claimed that it grows exponentially with the length of the chromosome [133, 140]. However, experimental results from Grefenstette [147] and Schaffer et al. [304] suggested that the population size proposed by Goldberg's theory is not necessary, and they proved that population sizes as small as 30 are adequate in many cases. J. Alander suggested that a value between n and $2n$, where n is the length of the chromosome, is optimal for several application scenarios. Reeves [285] tried to answer the question regarding the minimum population size for a meaningful search to take place, and suggested that at the very least, the population size should be large enough so that every point within the search space should be reachable from the initial population by crossover only, which means that every possible value of each parameter should at least appear once in the initial population. This argument actually aligns with our discussion of the *sampling* initialization method, and Latin Hypercube Sampling (LHS) [395,410] is a simple qualified example of such sampling methods. Goldberg et al. also discussed the refinements for the population size in the existence of noises and fitness variances [135, 137].

All the methods discussed above use a fixed value for the population size throughout the experiment. However, people have found that the population sizing requirements for GAs is problem-dependent and is sometimes hard to estimate before the actual algorithm is run. Instead, many adaptive methods have been proposed and proved to achieve better results than those static methods [62, 89, 211]. Also, adaptive methods make life easier for users by eliminating the population size parameter. Example works on adaptive population sizing include *schema-variance based* [328, 329], *GAVaPS* [9], *competing subpopulations* [306, 307], *SAGA* [161], *parameter-less* [66, 156, 207, 210, 265, 266], *APGA* [14], *PRoFIGA* [95, 357]. Details of these methods are beyond the scope of this report.

Selection. Once the fitness of a population is evaluated, a selection process begins to choose individuals for birth. One of the key properties of GAs is their ability to *reinforce* better configurations: better configurations in this generation live on and multiply in the next generation; and those that did poorly, are discarded. This is also the principle for most selection methods: they decide based on the fitness of individuals. There are three commonly applied categories of selection mechanisms [13, 119, 136, 235, 303]:

- *Proportionate selection*. This category includes the *Roulette-wheel selection* proposed in the original version of GAs [163], *stochastic remainder selection* [36, 41], and *stochastic universal selection* [16, 149]. All these methods choose individuals for reproduction based on the probability distribution calculated from the fitness of all individuals, as summarized in Equation 2.13. They differ in the form of the probability distribution, specifically in the assignment of the coefficients m_j. For example, when $m_j = 1$ for any j, it becomes the *Roulette-wheel selection* mechanism.

$$p_i = \frac{f_i}{\sum_{j=1}^{n} m_j f_j} \tag{2.13}$$

- *Ranking selection*. *Ranking selection* was proposed by Baker [15] and improved by Grefenstette and Baker [149]. Instead of being calculated from the fitness values directly, this category of selection mechanisms decide the candidate individuals based on the rank of fitness values among the whole population. Although some information is lost, this technique has the advantage of eliminating the need for re-scaling, as shown in Equation 2.13; it thus reduces the computation costs for the selection phase [119].

- *Tournament selection*. The *tournament selection* mechanism was proposed by Brindle [41] and later applied and improved by Goldberg et al. [138], Mühlenbein [241] and Suh and Van Gucht [338]. It chooses a set of k individuals from the current population, compares their fitness, and pick the best one for parenthood. It has similar properties as *linear ranking selection* when $k = 2$ [119]. One potential advantage of the *tournament selection* over the other two categories is that it only needs a preference

between a group of k fitness values. This broadens the approach for defining fitness functions, and makes GAs suitable for more types of optimization goals.

Other selection methods include *non-linear ranking selection* [234], *Boltzmann selection* [225], *($\mu + \lambda$)-selection* [313, 314], *Boltzmann tournament selection* [134, 217] and *entropy-Boltzmann selection* [196].

Elitism. A concept related to the selection phase is *elitism*, which refers to the mechanism that the best or the top-ranked few chromosomes of the current population are directly "copied" into the new population, while the rest of the new population are produced by applying one of the selection methods. This is the same term as the *reproduction* shown in Figure 2.3 and appears in many MH techniques in various forms. Without elitism, such good individuals can be lost since crossover and mutation may destroy them. Intuitively, GAs work best when good configurations are reinforced or multiplied, allowing them to survive to the next generation. Conversely, poorly performing configurations are removed from the population. We may even support more complicated *elitism* mechanisms by assigning a *lifetime* variable to each elite individual. This is a similar idea as the adaptive population sizing method proposed in GAVaPS [9]. How to assign the *lifetime* value is an open question, which may further depend on the population size, age, fitness, etc. Another related question is how many offspring should pairs of configurations be allowed to have. We could allow better configurations to proportionally have more offspring, but if we carry this too far, our population would become somewhat incestuous and lack in diversity. Diversity is important to allow exploring unexplored parts of the search space.

Crossover. *Crossover* refers to the process of taking two parent chromosomes and producing from them one or more children [325]. It is sometimes called *recombination*. It is applied to the selected candidates after the selection phase, with the hope that it creates better offspring. There are many different types of crossover operations [71, 98, 119, 325]. We use *Single Point Crossover (1X)* as an example to explain how crossover operates on chromosomes. This is the crossover methods used in traditional Simple Genetic Algorithms (SGAs), where two selected parent chromosomes are cut at the same corresponding point and the subparts after the crossover point are exchanged between two parents. The crossover point is randomly picked along the chromosome. In Figure 2.5 we use chromosomes with 5 genes as an example, and the crossover point between gene *BG* and *Inode Size* is picked. Other crossover methods include *Two Point Crossover (2X)*, *Multi-Point (N-Point) Crossover*, *Uniform Crossover (UX)* [341], *Multi-Parent Crossover (MPX)* [90–93, 255, 315, 327, 354, 355, 364], *Crossover with Reduced Surrogate* [37], *Shuffle Crossover* [47], *Precedence Preservative Crossover (PPX)* [30, 33], *Ordered Crossover (OX)* [69], *Partially Matched Crossover (PMX)* [139], *Cycle Crossover* [253], etc. Crossover is one of the most important parts in GAs, as it controls how GAs search the neighborhood space, and thus partially determines the effectiveness and efficiency of GAs. We plan to try all the listed crossover methods here, and analyze how well they work for optimizing storage systems. This could give us a better understanding of GAs and help us devise crossover methods that better suit our needs.

Another important parameter in crossover methods is *crossover probability*, which controls the amount of crossover operations that will be performed. Usually it is set to a relatively high value to search more of the neighborhood region. However, if set too high, it may cause GAs to waste much time on exploring unpromising regions of the solution space. And it is also good to not set it to 100% to allow some individuals from the old generation to directly survive into the next generation [325]. The *crossover probability* can also be adaptively adjusted during the process of GAs [37, 332].

Mutation. After crossover, *mutation* is often performed on the new chromosomes. If *crossover* controls how GAs search the neighborhood regions, *mutation* avoids GAs from getting trapped in a local optimum and lets GAs explore other regions of the entire search space. Mutation acts as an insurance policy against

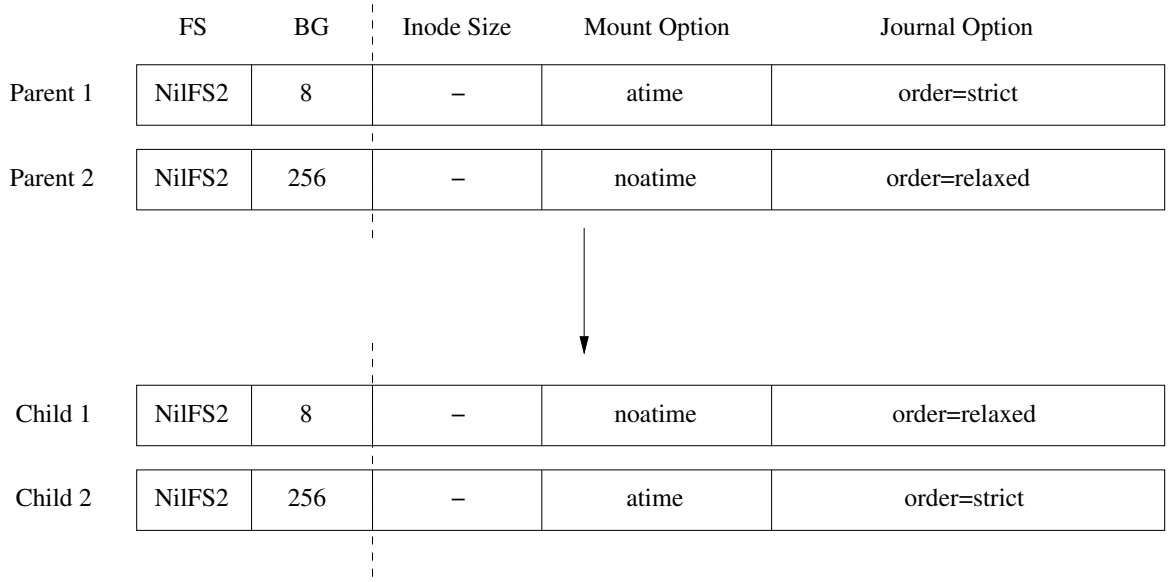


Figure 2.5: 1-Point Crossover

the irreversible loss of genetic material, and maintains genetic diversity in the population [325]. Most mutations are small and allow the population to randomly explore in the immediate neighborhood. In nature, mutations often follow the power law: most are small but a few can be large [21, 250]. Large mutations are important to allow a population to explore well outside its immediate neighborhood. If a mutation yields poorer fitness, that member will likely not survive to the next generation; but if by some chance it stumbled upon an even better neighborhood to explore, then such a large mutation could win and carry over a beneficial mutation to future generations. There are many different forms of mutation corresponding to different types of chromosome representation. For binary encoding, mutation can be *bit-flipping*, *interchanging* and *reversing*, etc. However, traditional mutation methods have to be modified in order to be applied in our project, as it is difficult to encode parameters of storage systems as bit strings. One simple method would be randomly changing the allele of a gene into other alleles of the same gene. We illustrate an example of this in Figure 2.6, where we randomly mutate the gene *File System* from *Ext4* to *Ext3*.

An important parameter in mutation is the *mutation probability*, which decides how often parts of chromosomes are mutated. If GAs mutate too little, they will not explore the search space enough and may miss higher peaks; if one mutates too much, GAs actually devolve into nearly pure random search and move away from known good fitness peaks. Mutation probability can be fixed as well as dynamically-adjusted [332].

Stopping criteria. Similar with Simulated Annealing, we need some stopping criteria for GAs as well. A lot of work was done on finding proper stopping criteria for GAs [10, 11, 145, 233, 297, 300]. Usually people would employ three types of stopping criteria for GAs [300]:

- an upper limit on the number of generations;
- an upper limit on the number of fitness evaluations; or
- the probability of achieving significant changes in the following generations is excessively low.

These simple criteria may not be adequate for optimizing storage systems, and as our project progresses, we plan to devise more complicated stopping criteria based our experiment results.

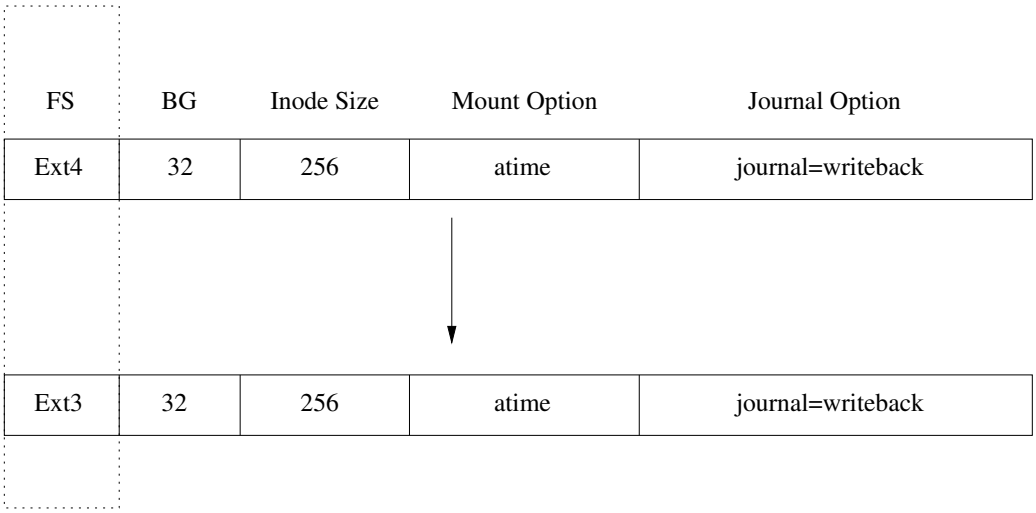


Figure 2.6: Example of mutation operations in GAs

2.4.3.4 Theories on GAs

It is still being hotly debated that why and how GAs work. Though many theories have been proposed to explain the essence of GAs, none of them provides more definite answers [119, 242, 287, 288, 367, 402].

Holland first proposed the Schema Theorem [163, 402] to explain the power of GAs. Here, *schema* is a template that identifies a subset of strings with similarities at certain string positions. For example, for binary bit strings of length 4, "1**1" represents a schema where the first and the last bits are 1. The number of fixed positions in the chromosomes are called *order*. The Schema Theorem states that short, low-order schemata with above-average fitness increase exponentially in successive generations. However, it holds under the assumption of a genetic algorithm that maintains an infinitely large population, which is rarely practical in real applications. Due to sampling error in the initial population, genetic algorithms may converge on schemata that have no selective advantage [402].

Other proposed theories on how GAs work include the Building Block Hypothesis [132], Markov processes [72] and the Dynamical System Model [287].

2.4.3.5 Variants of GAs

One possible limitation of traditional GAs is *premature convergence*. It means a non-optimal genotype takes over a population resulting in every individual being either identical or very similar, and the consequence is a population which does not contain sufficient genetic diversity to evolve further [119,199,325]. Many variants have been proposed to overcome this drawback of Simple Genetic Algorithms, such as Parallel Genetic Algorithms [143, 144, 240, 268], Distributed Genetic Algorithms [22], Hybrid Genetic Algorithms [324], Adaptive Genetic Algorithms [161], and Fast Messy Genetic Algorithms [138,325]. Here we briefly mention two types of Parallel Genetic Algorithms, the Island Model Genetic Algorithm [380] and Niching Genetic Algorithm [218]. In a parallel implementation of an *Island Model*, each machine executes a genetic algorithm and maintains its own subpopulation for search. The machines work in concert by periodically exchanging a portion of their populations in a process called *migration*. Niching Genetic Algorithms are particularly useful when the search space is multi-modal and one likes GAs to be able to identify several peaks simultaneously. A *niche* can be thought of as one of the peaks and a *species* is a collection of population members well suited for a particular niche. This actually extends traditional genetic algorithms to domains that require the location and maintenance of multiple solutions, especially machine learning and some multi-objective optimization

problems. We are interested in these two variants of GAs, because we already encountered the premature convergence problems in our experiments, and the idea of subpopulation in these two variants perfectly aligns with the nature of our problem, where different file systems may have totally different features and thus different behaviors. It might be a good idea to treat them in a slightly different way.

2.4.3.6 Applications of Genetic Algorithms

GAs have been widely applied to various areas, including the bin-packing problem [100, 101, 190, 284, 326], graph partitioning [44, 162, 171, 220, 271, 345, 365], quadratic assignment problem [5, 84, 107, 164, 344, 346], Job-shop Scheduling Problem (JSP) [29, 52, 53, 69, 75, 82, 102, 141, 269], Travelling Salesman Problem (TSP) [139,146,193,240,291,333], to real applications like water resource management [302,376], transport system [33,49,347,407], set-related problems [3,18,55], steiner tree problem [158,175,368], neural networks [23,184,238,305,381], feature selection [194,195,252,323], VLSI Design [28,60,205,226], High-Performance Computing (HPC) [19], power system optimization [152], software engineering [45, 151, 191], security [64, 65, 168], and computer system design [20, 51, 74, 212, 349]. GAs have also been applied to storage area before [79, 114, 180], although previous works limit themselves to a single objective and were optimizing a relatively smaller search space. Our work is more general, targeting at any optimization goals that can be measured, and we are optimizing a much larger and more complicated space.

2.4.4 Differential Evolution

Differential Evolution (DE) [40, 68, 274, 275, 334–336, 362] is a vector-based evolutionary algorithm, and can be considered as a further development of genetic algorithms. DE is mostly useful when applied to multidimensional real-valued functions. The fact the DE does not use the gradient information of the problem being optimized greatly extends the areas that DE can be applied to, especially the problems with fitness functions that are not even continuous or are noisy and change over time.

One big advantage of DE compared with GA is that it required fewer parameters, as shown in Algorithm 3. In most cases, there are two important features that will have huge impacts on the results of DE:

- *donor vector*: Generating the *donor vector* is actually the mutation operation in DE. In the original version of DE, we have
$$\overrightarrow{\mathbf{V}} = \overrightarrow{\mathbf{X_{r_1}}} + F(\overrightarrow{\mathbf{X_{r_2}}} - \overrightarrow{\mathbf{X_{r_3}}}) \tag{2.14}$$
where $\overrightarrow{\mathbf{X_{r_1}}}$, $\overrightarrow{\mathbf{X_{r_2}}}$ and $\overrightarrow{\mathbf{X_{r_3}}}$ are randomly picked from the current population. $F \in [0, 2]$ is often referred as the *differential weight*. There are many other ways to generate the donor vector, such as replacing the base vector with the best individual in the current population.

- *crossover probability*: The *crossover probability* determines whether to do crossover at the current index. Usually, the actual crossover can be carried out in two ways, *binomial* and *exponential*.

DE has been applied to various areas [272], including function optimization, DNA microarrays, neural network, data mining, etc.

One possible limitation of DE is that it requires all the parameter to be real-valued, while we have many binary, discrete, and non-numeric parameters in storage systems. We have to modify the original DE in order to apply to our problem.

Algorithm 3: Differential Evolution

Input : Initial population $initial()$, donor vector $donor()$, crossover probability $crossover\_prob()$, stopping criteria $stopping()$, evaluation function $evaluate(P)$

Output : Best individual $\overrightarrow{\mathbf{X_f}}$

```
1  P ← initial();
2  while not stopping() do
3  |   P' ← empty_list();
4  |   foreach X⃗ ∈ P do
5  |   |   V⃗ ← donor(P);
6  |   |   for i ← 1 to |X⃗| do
7  |   |   |   if crossover_prob() ≥ random(0, 1) then
8  |   |   |   |   X⃗'[i] ← Y⃗[i];
9  |   |   |   else
10 |   |   |   |   X⃗'[i] ← X⃗[i];
11 |   |   |   end
12 |   |   end
13 |   |   if evaluate(X⃗) < evaluate(X⃗') then
14 |   |   |   P'.append(X⃗');
15 |   |   else
16 |   |   |   P'.append(X⃗);
17 |   |   end
18 |   end
19 end
20 X⃗_f ← P[max_index(evaluate(P))];
```

2.4.5 Particle Swarm Optimization

Particle Swarm Optimization (PSO) [56, 87, 181, 182, 260, 322] is a MH technique with a population of candidate solutions. In PSO, one single candidate solution is called a *particle*, and the population of solutions is called a *swarm*. Each particle starts with an initial *velocity*, and particles can communicate with each other during the search process. The pseudo-code of PSO is shown in Algorithm 4.

Similar to other MH techniques, PSO also has several parameters that can impact the algorithm results. The swarm size can be computed following the formula [260]:

$$swarm\_size = \mathbf{int}(10 + 2\sqrt{n}) \tag{2.15}$$

where n is the parameter dimension. However, this formula is based on only an empirical result. $\omega, \varphi_p, \varphi_g$ from Algorithm 4 are the constant parameters set by users. They have large impact on the behavior and efficacy of PSO [260].

Note that in line 22 of Algorithm 4, the algorithm updates the particle's position according to the best position that this particle has ever reached as well as the known global best solution. This may cause the algorithm to get stuck in some local optima [262]. One simple improvement would be to replace the global known best position with the best known position of a sub-swarm around the current particle.

PSO has been applied to various areas since been proposed [273], including bio-medical, VLSI design, network design, graphics and visualization, etc.

Algorithm 4: Particle Swarm Optimization

Input : initial particle $initial\_particle()$, initial velocity $initial\_velocity()$, stopping criteria $stopping()$, evaluation function $evaluate(\overrightarrow{\mathbf{x}})$, the population size $swarm\_size()$, parameter dimension n, constants $\omega, \varphi_p, \varphi_g$

Output : final state $\overrightarrow{\mathbf{g}}$

$swarm\_list \leftarrow empty\_list()$;
$particle\_best\_list \leftarrow empty\_list$;
$velocity\_list \leftarrow empty\_list$;
$swarm\_size \leftarrow swarm\_size()$;
$\overrightarrow{\mathbf{g}} \leftarrow \mathbf{null}$;
for i **from** 1 **to** $swarm\_size$ **do**
  $\overrightarrow{\mathbf{x_i}} \leftarrow initial\_particle()$;
  $swarm\_list.append(\overrightarrow{\mathbf{x_i}})$;
  $particle\_best\_list.append(\overrightarrow{\mathbf{x_i}})$;
  $\overrightarrow{\mathbf{v_i}} \leftarrow initial\_velocity()$;
  $velocity\_list.append(\overrightarrow{\mathbf{v_i}})$;
  if $evaluate(\overrightarrow{\mathbf{x_i}}) > evaluate(\overrightarrow{\mathbf{g}})$ **then**
    $\overrightarrow{\mathbf{g}} \leftarrow \overrightarrow{\mathbf{x_i}}$;
  end
end
while not $stopping()$ **do**
  for i **from** 1 **to** $swarm\_size$ **do**
    for d **from** 1 **to** n **do**
      $r_p \leftarrow random(0,1)$;
      $r_g \leftarrow random(0,1)$;
      $\overrightarrow{\mathbf{p_i}} \leftarrow particle\_best\_list[i]$;
      $\overrightarrow{\mathbf{v_{i,d}}} \leftarrow \omega\overrightarrow{\mathbf{v_{i,d}}} + \varphi_p r_p(\overrightarrow{\mathbf{p_{i,d}}} - \overrightarrow{\mathbf{x_{i,d}}}) + \varphi_g r_g(\overrightarrow{\mathbf{g_d}} - \overrightarrow{\mathbf{x_{i,d}}})$;
    end
    $\overrightarrow{\mathbf{x_i}} \leftarrow \overrightarrow{\mathbf{x_i}} + \overrightarrow{\mathbf{v_i}}$;
    if $evaluate(\overrightarrow{\mathbf{x_i}}) > evaluate(\overrightarrow{\mathbf{p_i}})$ **then**
      $particle\_best\_list[i] \leftarrow \overrightarrow{\mathbf{x_i}}$;
      if $evaluate(\overrightarrow{\mathbf{x_i}}) > evaluate(\overrightarrow{\mathbf{g}})$ **then**
        $\overrightarrow{\mathbf{g}} \leftarrow \overrightarrow{\mathbf{x_i}}$;
      end
    end
  end
end

2.4.6 Tabu Search

Tabu Search (TS) [122, 124–126, 129, 343] is a Meta-Heuristic technique for solving complex optimization problems, designed to guide traditional local search processes to escape the trap of local optimality. TS has obtained optimal and near-optimal solution in a wide variety of applications, such as the vehicle routing problem [351], the Travelling Salesman Problem (TSP) [118], VLSI design [301], neural network [320], etc.

Traditional local search processes take a potential solution to the problem and check its immediate neighbors in the hope of finding an improved solution. They have a tendency to become stuck in suboptimal regions [343]. TS relaxes the rules of local search mainly in two aspects. First, some worsening moves can be

accepted if no improving move is available. Second, *tabu lists* are maintained to discourage the search from coming back to previously visited solutions. The *tabu list*, or more specifically the heavy use of maintained *history*, is one of the main differences of TS compared with many other Meta-Heuristic techniques. As we mentioned in Section 2.3, *history* is one of the three most important characteristics of any global search technique.

Algorithm 5: Tabu Search

Input : initial state $initial()$, neighborhood function $neighbor(s)$, stopping criteria $stopping()$, evaluation function $evaluate(s)$, the size of tabu list $tabu\_size()$

Output : final state s_f

```
1  s ← initial();
2  s_f ← s;
3  tabu_list ← empty_list();
4  tabu_list.append(s);
5  while not stopping() do
6  |   neighbor_list ← neighbor(s);
7  |   s ← argmax_{neighbor_list.contains(s') and not tabu_list.contains(s')}[evaluate(s')];
8  |   tabu_list.append(s) if tabu_list.size() > tabu_size() then
9  |   |   tabu_list.deleteLast();
10 |   end
11 |   if evaluate(s) > evaluate(s_f) then
12 |   |   s_f ← s;
13 |   end
14 end
```

In Algorithm 5 we show a typical template of TS. As we can see, a *tabu list* is used to record recently evaluated individuals and prevent the search process from visiting them again in the near future. In most cases, additional elements have to be included in TS in order to make it fully effective. Here we introduce two most important ones.

- *Intensification*. The main idea of *intensification* is exploring more thoroughly the portions of the search space that seem promising to ensure that the best solutions in these areas are found [119]. It is based on *intermediate-term memory*, compared to the *tabu list* as short-term memory. One typical approach for intensification is to restart the search from the currently known best solution.

- *Diversification*. The main purpose of *diversification* is to force the search process into previously unexplored regions of the space, and it is usually based the use of long-term memory. It records the total number of iterations that various parameter values were involved in explored solutions. There are two major diversification techniques [119]: *restart diversification* and *continuous diversification*. *Restart diversification* restarts the search process from a solution consisting of rarely used parameter values. *Continuous diversification* guides the search process directly into unexplored areas without restarting the search process. When deciding which neighbor state to access next, instead of only using the fitness values, continuous diversification adds a small term (directly or inversely) proportional to the frequencies of the parameter values, and uses the sum of these two to guide the search. In this way, unexplored states have a relatively higher probability to be selected.

These two concepts are actually similar to the ideas of *exploration* and *exploitation* which, along with *history*, are the three main characteristics of all MH techniques. We will discuss this topic in more detail in Section 2.3

TS has been applied to various areas, including the Traveling Salesman Problem (TSP), graph partitioning, scheduling problems, network topology design, etc. [122].

2.4.7 Other Meta-Heuristic Techniques

There are many other MH techniques which have been proved successful in various optimization domains. We briefly cover them here. Ant Colony Optimization is based on the foraging behavior of social ants [80, 81]. Bee Algorithms are a set of algorithm based on the foraging behavior of bees, and include Honey Bee Algorithm (HBA) [246], Virtual Bee Algorithm (VBA) [414], Bees Algorithm [270], Honey Bee Mating Optimization (HBMO) [153], and Artificial Bee Colony (ABC) [176–178]. Harmony Search(HS) is inspired by the improvisation process of musicians [115]. Firefly Algorithm (FA) is developed based on the flashing patterns and behavior of fireflies [415, 417, 418]. Cuckoo Search (CS) is based on the brood parasitism of some cuckoo species [112, 371, 419, 420]. Artificial Immune Systems are inspired by the characteristics of the immune system of mammals [26, 103, 350]. Bacterial Foraging Optimization is inspired by the social foraging behavior of bacteria such as Escherichia coli [263]. Cross-Entropy Method is a generalized Monte Carlo method, based on rare-event simulations [296]. Memetic Algorithms (MA) are synergy of evolutionary or any population-based approaches with separate individual learning or local improvement procedures for problem search [166, 189, 231, 239, 254]. Scatter Search (SS) derives its foundations from earlier strategies for combining decision rules and constraints, with the goal of enabling a solution procedure based on the combined elements to yield better solutions than one based on only the original elements [123, 127, 130, 131, 222]. Although there are many different MH techniques, we find they actually all work by maintaining a trade-off among exploration, exploitation, and history, as we discussed in Section 2.3. This trade-off is critical in guiding the search for the global optima. Nevertheless, we plan to study many of these MH alternative and leverage their best properties in this project.

2.5 Machine Learning

As we enter the era of big data, Machine Learning (ML) has becoming more popular in the last few decades. We can define ML as a set of methods that can automatically detect patterns in data, and then use the discovered patterns to predict future behavior, or to perform other kinds of decision making under uncertainty [244]. Generally, there are three types of ML techniques: *Supervised Learning*, *Unsupervised Learning*, and *Reinforcement Learning*. We discuss each next.

2.5.1 Supervised Learning

Supervised Learning is sometimes also called *predictive learning*, and its goal is to learn the mapping from the inputs $\overrightarrow{\mathbf{x}}$ to outputs y, based on a labeled set of input-output pairs $D = \{(\overrightarrow{\mathbf{x_i}}, y_i)\}_{i=1}^{N}$. D is often referred as a *training set* consisting of N training examples. In the training set, each input $\overrightarrow{\mathbf{x_i}}$ is usually a multi-dimensional vector, and the elements in the vector are called *features* or *attributes*. Depending on whether the output y is *categorical* or *real-valued*, supervised learning can be further classified into two categories, *classification* and *regression*.

- *Classification*. In classification we have $y \in \{1, ..., C\}$, where C is the number of possible classes. For example, in the application of spam email filtering, emails are classified into two classes: spam or not spam. Important classification algorithms include Naive Bayes Classifier, Support Vector Machines, Decision Trees, and Neural Networks.

- *Regression*. Regression is similar to classification except that y is continuous. Usually the estimation target is a function of the independent variables called the regression function. Commonly used

regression models include Linear Regression, Polynomial Regression, Ridge Regression, and Lasso Regression.

2.5.2 Unsupervised Learning

Unsupervised Learning (or *descriptive learning*) is another main type of Machine Learning, where the dataset is unlabeled: $D = \{(\overrightarrow{\mathbf{x_i}})\}_{i=1}^{N}$. The goal of Unsupervised Learning is often to find certain patterns existing on the dataset; that is why it is also called *knowledge discovery*. Unsupervised Learning is arguably more typical of human and animal learning behaviors. It is also more widely applicable than supervised learning, since it does not require a human expert to manually label the data [244]. Approaches of Unsupervised Learning include:

- *Clustering*. Clustering is the process of grouping a set of objects in such a way that objects in the same group are more similar to each other than to those in other groups, according to certain criteria. Here the groups are called *clusters*. Common clustering algorithms include Hierarchical Clustering, K-means Clustering, Distribution-base Clustering, and Density-based clustering [58].

- *Latent Variable Model*. A latent variable is a variable which is not directly observable and is assumed to affect other observable variables (i.e., manifest variables). A Latent Variable Model is a statistical model that relates a set of manifest variables to a set of latent variables. Some of the most widely-applied models are factor analysis model, item response theory model, and generalized linear mixed models.

2.5.3 Reinforcement Learning

Reinforcement Learning (RL) [289, 340] is an area of machine learning inspired by behaviorist psychology, concerned with how software agents take actions in an environment so as to maximize the defined cumulative rewards. Most RL algorithms can be formulated as a model consisting of:

- A set of environment states, S;

- A set of agent actions, A; and

- A set of scalar reinforcement signals.

Generally there are two categories of RL techniques: (1) Model-based RL, which builds an exact model of the environment; and (2) Model-free RL, which learns a policy without any model. Here *policy* refers to the rules that define how the agent will take actions, and the optimal policy leads to the sequence of actions resulting in the maximum cumulative rewards.

We will mainly discuss model-free RL here, as for our project, the system is usually so complex that it is nearly impossible to learn an accurate model. Note that sometimes GAs are also considered as one type of RL [173]; GAs also search in the space of actions in order to find one that performs well in the environment.

Reinforcement Learning is closely related to Markov Decision Processes (MDPs) [229]. In practice, for most conventional RL algorithms, the environments are formulated as MDPs. In the domain of MDPs, if the rewards for each state are given, then there are two types of algorithms searching for the optimal policy.

- *Value Iteration*. The idea behind it is that the immediate reward from each state is not the "true value" of that state. One state may have low initial rewards, but it is on the path to a high-reward state. The "true value" is sometimes also called the *utility*, and is calculated by the initial reward plus the expected discounted reward if the agent took optimal moves from that point on. The value iteration algorithm first assigns each state a random value; then updates the utility for each state based on the

utilities of its neighbor states until no values changes. It has the drawback of slow convergence. Also, it learns the utilities, which are not necessary. What really matters is finding the optimal policy.

- *Policy Iteration*. Unlike value iteration, policy iteration directly learns the optimal policy. It first creates a random policy and computes the utilities of each state under that policy; then it modifies the current policy by using the updated utilities. It continues this process until the policy no longer changes.

In practice, sometimes the reward for each state is not given ahead of time, which means the agent does not have expert domain knowledge about the environment. RL is proposed to deal with this situation. Actor-critic Learning and Q-Learning are two of the most popular model-free RL algorithms, and they corresponds to policy iteration and value iteration, respectively. For example, Q-Learning assigns each state-action combination an estimated value, called a *Q-value*. After the agent visits each state and received the actual rewards, it updates the Q-values of that state.

Another interesting fact in Q-Learning is that it also maintains a trade-off between exploitation, exploration, and history. In the early stages of execution, when the agent knows little about the environment, it will explore the space and try unknown actions. When it interacts enough with the environment, it will choose the actions that it knows will receive the most rewards. However, the algorithm has to give the agent enough chances to explore states with lower Q-values, even after running for a long time. Usually the Boltzmann distribution is used to achieve this. The probability of selecting the highest Q-value action will increase over time, but the chance of taking actions with lower Q-values will never become zero. Conventional Q-Learning algorithms maintain a lot of history information by storing all the rewards they received from the environment. Originally they use a table to store the learned data; but this becomes impractical when the possible states and actions are numerous. Function approximation is sometimes used to replace the table for Q-Learning in order to solve complex problems [276].

2.6 Summary

In this section we summarize the aforementioned techniques by explicitly discussing how they fit into our unified framework of exploration, exploitation, and history.

Actually others have also attempted to unify several search techniques. De Jong argues that all Evolutionary Algorithms (EAs) follow the similar pattern where:

- a population of constant size m is evolved over time,
- the current population is used as a source of parents to produce n offspring, and
- the expanded population is reduced from $m + n$ to m individuals.

In addition to the choices of m and n, an EA must also specify a method for selecting the parents to be used to produce offspring, and a method for selecting which individuals will survive into the next generation. Eiben and Schippers [88] provide an early discussion on evolutionary exploration and exploitation and they argue that these two properties are the two cornerstones of problem solving by search. Following this work, Črepinšek et al. [361] provide a thorough survey on existing approaches for maintaining the balance between exploration and exploitation. They classify the proposed approaches into three categories: diversity maintenance, diversity control, and diversity learning.

Our unified view is different from the aforementioned ones by taking history information into account. In Table 2.4 on Page 35 we summarize all the algorithms that we investigated in terms of these three key properties, as well as several other perspectives. As the No Free Lunch Theorem [405] states, there exists no single, best algorithm that will outperform all others on all types of optimization problems. We hope to

investigate various techniques to help us better understand how to search effectively and efficiently for the best configurations, and how to maintain a good balance among exploration, exploitation, and history. Our ultimate goal is to devise an algorithm that can optimize any storage systems for various goals.

Algorithm	Origin	Individual	Population	Exploration	Exploitation	History
Simulated Annealing (SA)	Annealing technology in metallurgy	State	N/A	Allowing moving to worse neighbor states based on certain acceptance probability distribution	Neighbor function	N/A
Genetic Algorithm (GA)	Natural evolution	Chromosome	Population	Mutation	Crossover	N/A
Tabu Search	Local search	Solution	N/A	Diversification	Local search; intensification	Tabu list; long term memory used by intensification and diversification.
Particle Swarm Optimization	Behavioral models of bird flocking	Particle	Swarm	Initially high velocity	Velocities slow towards zero	N/A
Hill-Climbing	Unknown	Solution	N/A	N/A	Neighbor	N/A
Random-restart Hill-Climbing	Unknown	Solution	N/A	Restart	Neighbor	N/A
Smart Hill-Climbing	Unknown	Solution	N/A	Restart guided by Weighted Latin Hypercube Sampling	Neighbor guided by Weighted Latin Hypercube Sampling	Updating weights in Weighted Latin Hypercube Sampling
Q-Learning	MDPs	State	N/A	Choosing states with lower Q-values	Choosing the state with the highest Q-value	Updating the Q-values for states

Table 2.4: Comparison and Summaries of MH Techniques

Chapter 3

Related Work

The idea of auto-tuning has been investigated and applied in various areas of computer system design. Here we start with the work on storage systems in Section 3.1. Section 3.2 will cover auto-tuning techniques applied in other areas.

3.1 Auto-tuning in Storage Systems

Auto-tuning techniques have already been applied to several aspects of storage systems before. Gaonkar et al. [114] apply GAs to design dependable data storage systems for multi-application environments, with the goal of minimizing the overall cost of the system while meeting business requirements. Kimberly et al. [180] formulate the data recovery scheduling problem as an optimization problem. They aim at finding the schedule that minimizes the financial penalties due to downtime, data loss, and vulnerability to subsequent failures. GAs are applied and compared with several other heuristics. GAs are also used to solve the NP-hard problem of designing Storage Area Network (SAN) [79], which is to find the cheapest SAN that supports the specified flows, while satisfying the port constraints, bandwidth constraints, and non-splitting of flows. From another perspective, Xue et al. [411, 412] propose an autonomic technique that learns the intensity patterns of user workload in tiered storage systems over long time-scales using a probabilistic model. They use the model to predict the coming workload patterns and proactively stop/start bulky internal system work. MINERVA [6], is a suite of tools for automating storage system design, which uses declarative specifications of application requirements and device capabilities; constraint-based formulations of the various sub-problems; and simple bin-packing heuristics to explore the search space of possible solutions. Various machine learning algorithms are applied in modelling the performance of the PIDX parallel I/O library and selecting appropriate tunable parameter values [192]. Chen et al. [51] propose an approach taking a high level description of the target workload and execution environment characteristics as inputs, and applies genetic algorithms to select high quality I/O plans. Behzad et al. [20] present an auto-tuning system for optimizing I/O performance of HDF5 applications. The system uses a genetic algorithm to search a large space of tunable parameters and to identify effective settings at all layers of the parallel I/O stack.

Ganger et al. [113] describe the design and implementation of "self-* storage systems," which is self-organizing, self-configuring, self-tuning, self-healing, self-managing systems of storage bricks. Later their group propose to use Classification And Regression Trees (CART) models [39] to predict the performance of storage systems under certain workloads, which they think is a key element of self-managed systems. Although it achieves acceptable accuracy, they admit that the quality of the generated models depends highly on the quality of the training workloads, which needs to be diverse and large enough. As we have few existing datasets for any new workload, it becomes difficult for us to directly apply any supervised ML techniques in solving our problem. We provide more discussion on this in Section 5.3. Conversely, GAs are applied

in their framework to provision storage systems as well [337]. They formulate a complex fitness function combining performance, availability, reliability, capacity, power consumption and system costs.

While these works only target one specific optimization perspective, our work is more general, and any measurements or functions with comparable return values can be taken as our optimization goals. Our approach is also supposed to recognize and react to the changes in the environment, which has not been addressed in previous work. Another difference is that we are not limiting ourselves to GAs only; instead, we are investigating a wide variety of different MH techniques, and we will come up with our own optimization algorithms as well.

3.2 Auto-tuning in Other Areas

In this section we cover the auto-tuning techniques proposed in various domains other than storage. We organize the text by classifying these works based their targeted systems.

General. Many theoretical projects used auto-tuning systems. Thonangi et al. [348] argue that when auto-tuning parameters in real software systems, evaluations of configurations can be very expensive. This motivates them to propose a new adaptive search algorithm that can find good configurations under the constraints of high dimensionality and few experiments. Osogami et al. [258] find that the bottleneck in most auto-tuning systems is the time taken for simulations or experiments. To address this, they propose a novel tuning algorithm that evaluates only the necessary neighborhood configurations and evaluates them only with the necessary accuracy. Filieri et al. [105] argue that the difficulty of modeling computer systems as dynamic systems arise from the fact that computer systems are usually non-linear, with varying workloads and heterogeneous components, which also make traditional control theory not applicable. To solve this, they propose a methodology that automatically builds suitable system models, which are continuously updated to compensate for changes in the execution environment, and then use those models to synthesize a controller. Coker et al. [61] claim that future-generation self-adaptive systems are designed to optimize for multiple interrelated, difficult-to-measure, and evolving quality properties. To address this complexity, they suggest the research community pursue the application of stochastic search techniques, such as hill climbing or genetic algorithms, that incorporate an element of randomness.

Web Servers. Menascé et al. [230] and Liu et al. [209] have some early work on online optimization of Web server performance, but their work is limited in that it either needs to modify the source code or requires an accurate analytical model of the targeted system. Later, a fuzzy controller was applied to enable automating parameter tuning of Web servers, which employs rules incorporating qualitative knowledge of the tuning parameters [78]. Raghavachari et al. [277] use purely random search to find the optimal configuration of Web servers. Liu et al. [208] applied Newton's Method, Fuzzy Control, and Saturation-Based Heuristic Optimization to find the optimal value of the parameter *MaxClients* for Apache Web server. Following that work, Xi et al. [410] propose a Smart Hill-Climbing Algorithm for black-box optimization of Web servers; this is actually a hybrid method combining traditional Hill-Climbing algorithm and Latin Hypercube Sampling. Osogami and Kato [259] propose the Quick Optimization via Guessing (QOG), which guesses the performance of candidate configurations based on previous evaluations of similar configurations, and picks the most promising one for the next measurement. Beltran et al. [24] compare the tuning complexity of Multi-threaded and Hybrid Web servers, and they argue that the hybrid architecture outperforms the multi-threaded one, not only in terms of performance, but also in terms of its tuning complexity and its adaptability over different workload types. A Reinforcement Learning approach is also applied in auto-configuration of Web servers [42].

Architecture. Reinforcement Learning (RL) is used by Ipek et al. [165] to enhance the scheduling of requests in the memory controller. They define a finite set of memory controller states and potential actions and use RL to train the memory controller Diegues and Romano study the problem of automatically tuning the policies used to regulate the activation of the fallback path of Intel TSX processors, and present a solution that combines a set of lightweight reinforcement learning techniques designed to operate in a workload-oblivious manner. Peled et al. [264] argue that for some workloads, memory access patterns follow semantic locality rather than conventional spatio-temporal locality, and they introduce a context-based memory prefetcher, which approximates semantic locality using RL.

Databases. Duan et al. [85] propose iTuned, a portable tool that automates the task of identifying good settings for database configuration parameters. It applies Adaptive Sampling that proactively brings in appropriate data through planned experiments to find high-impact parameters and high-performance parameter settings. It also supports online experiments in production database environments through a cycle-stealing paradigm that places near-zero overhead on the production workload.

Networking. Ye et al. [421, 422] argue that the problem of auto-tuning network configuration parameters has several important features, including *high efficiency*, *high dimensionality*, *noise*, *negligible configurations*, and *globally convex*. Similar features also exist in storage system optimization problem. They propose a Recursive Random Search (RRS) algorithm to find the optimal network configuration, which is based on random sampling and the algorithm restarts periodically with adjusted sampling space.

Virtualization. Bu et al. [43] propose CoTuner for coordinated configuration of VMs and resident applications with the goal of maximizing the system total throughput in cloud computing environment. It uses a model-free hybrid reinforcement learning (RL) approach, with policies that include a Simplex method, Reinforcement Learning, and system knowledge.

Distributed Systems Saboori et al. [299] apply an evolutionary algorithm called Covariance Matrix Adaptation (CMA) to automatically tune system parameters for performance improvement purposes. CMA uses a multivariate Gaussian distribution to summarize the statistics of good solutions and only the parameters of Gaussian distribution are updated at each step. Loffler et al. [212] use several Evolutionary Algorithms, including GAs, Cuckoo Search (CS), Particle Swarm Intelligence (PSO), and a hybrid algorithm of CS and PSO, to minimize both network and processing latency in Event-Based Systems by means of runtime migration of event detectors.

Security. Fulp et al. [64, 65, 168, 214] apply GAs to find suitable configuration changes that form a Moving Target Defense, where the fitness is given by the level of security inferred based on the number of security events detected during operation. Winterrose and Carter [404] use Genetic Algorithms to pick optimal attacker strategies against temporal platform diversity defenses, and the fitness functions take into account game play success, exploit creation success and strategic complexity costs.

Software. Ramirez et al. [281] propose an approach to leverage GAs in the decision-making process of automating software maintenance tasks, which enables the system to dynamically evolve reconfiguration plans at run time in response to the changing requirements and environments.

High Performance Computing. Tikir et al. [349] propose a GA-based approach that learns memory bandwidth as a function of cache hit rates per machine and predicts the performance of HPC applications.

Energy. Mishra et al. [236] propose LEO, a probabilistic graphical model-based learning system that provides accurate online estimates of the power and performance of an application as a function of system configuration, with the goal of minimizing energy consumption while still meeting the performance requirements.

Chapter 4

Experiment Settings

In this chapter we introduce the settings of our current experiments, consisting of three parts. Section 4.1 introduces the two set of machines that we are using. Section 4.2 covers the benchmark tools. We explain how we picked our parameter set in Section 4.3.

4.1 Hardware

We performed experiments on two machines with different hardware categorized as low-end (M1) and mid-range (M2). We list the details of these two sets of machines in Table 4.1. We also use Watts Up Pro ES power meters to measure the energy consumption during the experiments and store the data for future use.

Hardware	**M1**	**M2**
Model	Dell PowerEdge™SC1425	Dell PowerEdge™R710
CPU	Intel Xeon single-core 2.8GHz CPU × 2	Intel Xeon quad-core 2.4GHz CPU × 2
Memory	2GB × 1	4GB × 6
Disk	73GB Seagate ST373207LW SCSI drive × 2	250GB SATA × 1, 200GB SSD drive × 1, 147GB SAS drive × 1, 500GB SAS drive × 1

Table 4.1: Details of Experiment Machines

4.2 Software

We mainly used *Filebench* [104, 318] to generate various workloads for our experiment use. Filebench is versatile and comes with multiple pre-configured workloads. In our current experiments we used the following four workloads:

- *Mailserver*. The *mailserver* workload emulates a multi-threaded email server. It generates sequences of I/O operations to mimic the behaviors of reading emails (open, read the whole file, and close), composing emails (open/create, append, close, and fsync) and deleting emails. It uses a flat directory structure with all the files in one single directory, and thus exercises the ability of file systems to support large directories and fast lookups.

- *Dbserver*. The *dbserver* workload mimics the behaviors of Online Transaction Processing (OLTP) databases. It mainly consists of random asynchronous writes, random asynchronous reads and mod-

erate synchronous writes to the log file. It exercises the ability of large file management, extensive concurrency and random read/write operations.

- *Fileserver*. The *fileserver* workload emulates servers that host home directories for multiple users, and users only have access to files in their own home directories. In this workload, one thread acts as one user, performs create, delete, append, read, write, and stat operations on a unique set of files. It exercises both the meta-data and data paths of the targeted file system.

- *Webserver*. The *Webserver* mimics the behaviors of typical Web servers. It has a high read/write ratio, and reads entire files (Web pages) sequentially by multiple threads (users), and each thread append to a common log file (Web log). This workload exercises the ability for fast lookups, sequential reads of small files and concurrent data and meta-data management.

We plan to experiment with more benchmark tools and with real applications, to evaluate the effectiveness and efficiency of our algorithm.

Usually Filebench executes a fixed amount of workload or runs for a fixed time period, and reports the throughput in terms of I/O operations per second. Filebench includes multi-process, multi-thread support, asynchronous I/O, configurable options for directory depths, file sizes in workload definition, and extensive testing on variety of operating systems including FreeBSD, Linux, and Solaris.

For our experiments, we formatted and mounted storage devices with specific file systems and then ran Filebench on it. We used *Auto-Pilot* [409] to setup the storage device and drive Filebench automatically.

4.3 Dataset

We are optimizing storage systems, so we naturally considered the file system type as one of the key parameters [318, 408]. We tried the following seven file systems and their key parameters.

1. **Ext2** [46] is a plain FFS [228] based file system with fixed size blocks and block groups for localizing related files.

2. **Ext3** [356] adds journaling to Ext2 and controls the order of writes carefully. Most journaling file systems can control what to journal (data, meta-data, or both). Journaling can sometimes help and sometime hurt performance [318].

3. **Ext4** [99] adds extents support to Ext3, useful for large files.

4. **XFS** [321] uses B+Tree data structures to scale the number of inodes, files in a directory, and maximum file sizes.

5. **Reiserfs** [290] uses balanced S+Tree data structures to manage data and meta-data into objects called *items*, each with its own key. Reiserfs can also store small files at the end of files whose data blocks are not completely filled (*tail-packing*).

6. **Btrfs** [293] is a Copy-On-Write file system developed comparable to ZFS [35, 339, 423]. Btrfs uses B+tree data structure and also supports checksumming and compression.

7. **Nilfs2** [186] is log-structured file system [32, 224, 295]. It writes all data sequentially into one or more append-only zones; eventually data has to be *cleaned* and moved to its final location.

We defined two set of parameters, seen in Table 2.1 on Page 5. *Storage V1* contains the following common file system parameters: *file system type*, *block size*, *inode size*, *blocks per group*, *mount options*, *journal*

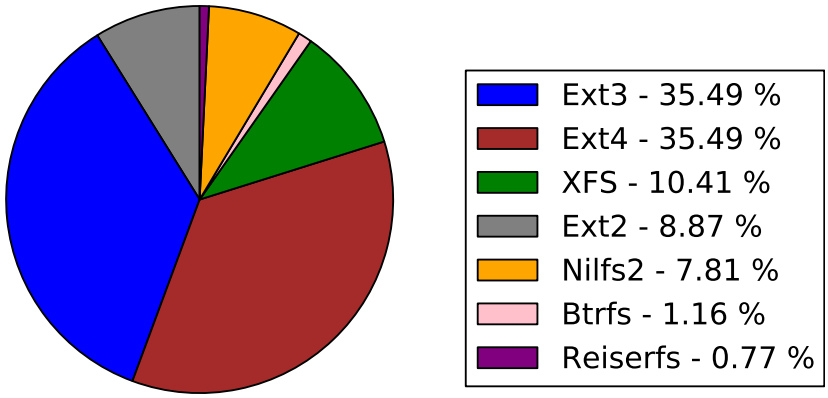


Figure 4.1: Percentage of Different File Systems in the Whole Search Space

options, and *special options*. *Storage V2* contains all the parameters in Storage V1 and also adds the *I/O scheduler* and *disk type*. Certain combinations of parameter values could produce an invalid configuration. For example, for Ext2, the journaling options make no sense because Ext2 does not have a journal. To handle this, we added a value *none* to the existing range of parameters. Any parameter with *none* value is considered invalid. Since we cannot benchmark invalid configuration combinations, we give them a zero fitness value, which ensures they are purged in an upcoming generation. Figure 4.1 shows the percentage of each file system among all valid configurations.

Some parameters take on powers-of-2 values within permitted ranges: block size, inode size, and blocks per group. The mount option is a Boolean to turn on/off `atime`. The journal gene supports three journaling modes. The special options capture parameters that are specific to certain file systems: `nodatasum`, `nodatacow`, and `compress` (Btrfs); and `notail` (Reiserfs). We provide a details list of our parameter space in Table 4.2.

Parameter	File System	Possible Values
File System	N/A	Ext2, Ext3, Ext4, XFS, Btrfs, Nilfs2, Reiserfs
Block Size	Btrfs, Reiserfs	4,096
	Ext2, Ext3, Ext4, XFS, Nilfs2	1,024, 2,048, 4,096
Inode Size	Reiserfs, Nilfs2	N/A
	Ext2, Ext3, Ext4, XFS, Btrfs	Default, 128, 256, 512, 1024, 2048, 4096, 8192
BG Count	Reiserfs, Btrfs	N/A
	Ext2, Ext3, Ext4	Default, 2, 4, 8, 16, 32
	Nilfs2, XFS	Default, 2, 4, 8, 16, 32, 64, 128, 256
Journal Option	Nilfs2	Default, order=strict, order=relaxed
	Ext3, Ext4, Reiserfs	Default, data=journal, data=ordered, data=writeback
	Ext2, XFS, Btrfs	N/A
Mount Option	N/A	atime, noatime
Special Option	Btrfs	Default, compress, nodatacow, nodatasum
	Reiserfs	Default, notail
	Ext2, Ext3, Ext4, XFS, Nilfs2	N/A
I/O Scheduler	N/A	noop, cfq, deadline
Disk Type	N/A	SAS_147GB, SAS_500GB, SATA, SSD

Table 4.2: Details of Parameter Space

Chapter 5

Preliminary Results

In this chapter we present the results from our preliminary experiments, and we also analyze the implications and insights. We start with an introduction to our data collection process in Section 5.1. Section 5.2 focuses on Genetic Algorithm experiments. In Section 5.3 we cover results from Machine Learning experiments.

5.1 Data Collection

We first exhaustively ran all configurations for each workload on our test machines, and stored the results in our database for future use. We collected the throughput in terms of I/O operations per second, as reported by filebench, as well as the running time (including setup time), and power and energy consumption. This data collection benefits our future experiments as we can simply simulate a variety of algorithms by just querying the database for the evaluation results for different configurations. The exhaustive run result let us know exactly what the global optimal configurations are, so that we can better understand how our search algorithms perform. We list the results of several workloads that we have completed in the past few months in Table 5.1.

Environment	**M1-mailserver**	**M1-fileserver**	**M2-mailserver**	**M2-dbserver**	**M2-fileserver**
Throughput (ops/s)	3,677	1,709	18,845	42,071	17,081
File System	Nilfs2	Ext4	Ext2	Ext4	Btrfs
Block Size	2,048	4,096	4,096	2,048	4,096
Blocks per Group	256	2	8	16	N/A
Inode Size	N/A	128	256	128	8,192
Mount Options	atime	noatime	atime	noatime	atime
Journal Options	order=relaxed	data=ordered	N/A	data=writeback	N/A
Special Options	N/A	N/A	N/A	N/A	compress
I/O Scheduler	(deadline)	(deadline)	noop	noop	noop
Disk Type	(SATA)	(SATA)	SSD	SAS_500GB	SAS_147GB

Table 5.1: Global Optima in Different Search Spaces

As we can see, the global best configuration for the *mailserver* workload on machine M1 uses Nilfs, which is actually 2.6× better than the default Ext4 configuration for most Linux systems. It is the same with all other workloads. For example, for the *fileserver* workload on M2, the global optimal configuration improves the throughput by 10.4% compared with the default. This proves our claim that default configurations are often not the best, and an intelligent and efficient tuning algorithm is necessary. When we run a different workload, *fileserver*, on M1, it converges on a different configuration, Ext4. This shows that the workload is a key part of the environment and significantly impacts the search space and optimal configurations. We then run the same experiments on different machine M2. Because M2 was faster than M1, we run

the larger *Storage v2* configuration with the I/O scheduler and disk-type parameters added, totaling 24,888 valid configurations, as shown in Table 2.1). In this case Nilfs2 was no longer the best configuration for the *mailserver* workload—Ext2 was. This indicates that the hardware is also part of the environment, and can affect the performance of storage systems significantly. The fact that the performance (and many other metrics) are sensitive to the environment (i.e., hardware and workloads) actually complicates our problem, as it requires our auto-tuning algorithm to possess the ability of recognize and react to the changes in the environment itself.

One surprise from the results presented in Table 5.1 is that the best configurations of the three completed workloads all use the `noop` I/O scheduler rather than the default `deadline` one. This proves that default configurations are suboptimal and even common wisdom (i.e., best accepted practices) can be wrong. Also, of the three workloads, only the *mailserver* picks the SSD in its best configuration; the other two selected the SAS drive. This may indicate that for certain workloads the more expensive SSD drive does not mean better performance than cheaper SAS drives. In future work, we plan to use more sophisticated fitness models. For example, a fitness model that incorporates the cost of storage along with throughput may result in an even higher probability of selecting the cheaper SAS drive over the SSD one.

This data collection process has been running for nearly one whole year, and it will continue for years to come, as we plan to try more types of workloads and even more complex parameter spaces. The dataset is valuable as it contains the performance and energy consumption of tens of thousands of different configurations under various workloads, which provides great insights into how file systems behave under different scenarios. As far as we know, there is no storage-related dataset of this size available online. One important, planned future contribution of our project is that we will make the dataset public online when it is ready and share with the research community.

5.2 Genetic Algorithms

With the exhaustive search results collected in Section 5.1, we are able to simulate various algorithms within a much shorter time compared with actually evaluating each configuration during the experiments. For example, within 5 minutes we can complete 50 runs of Simple Genetic Algorithms on Storage V2. This speedup allows as to try and compare different types of algorithms, and also explore many different parameter settings within the same technique (e.g., mutation rates in GA). We start with several GA simulations using the Pyevolve library [267], which is a python library for GAs. It contains more than ten configurable parameters, which can have significant impact on the results, as we discussed in Section 2.4.3. We list them and their default values in Table 5.2.

Table 5.3 and Figure 5.1 show the results of GA simulations on Storage V1 under the *mailserver* workload on M1 machines. We simulate GA five times, all with the default parameter settings from Pyevolve, except with a total number of generations set to 200.

From Table 5.3 we can see that for 3 out of 5 runs, Nilfs2 produced a better throughput than the default Ext4 one: 2.6× better. In fact, GA actually finds the global optimal or near optimal configuration three times (modulo small standard-deviation fluctuations). In Run 1, GA picks Ext4 but with different options that is still 15% better than Ext4's default options; and in Run 3, GA picks the old Ext2 file system and a throughput 16% better than that of Run 1. These results demonstrate GA's ability to find better configuration than default ones and reach close to near-optimal ones, but also show the fact that the optimum is not guaranteed for each run.

For the three runs that selected Nilfs2 as the best, the mount option `noatime` is selected only once, but it does not appear to impact the overall throughput. This may indicate that different parameters could have quite different impacts on certain metrics; that some may even are unimportant that they can be eliminated completely for future searches, which reduces the space exponentially. We will delve into this topic in

Parameter	Population Size	Generation	Selection Method	Mutation Rate	Crossover Rate	Elitism
Range	$p \in N^+$	$g \in N^+$	[Roulette, Rank, Tournament, Uniform]	$(0, 100\%]$	$(0, 100\%]$	$i \in N$
Default	80	100	Rank	2%	90%	1
Parameter	**Multiprocess**	**Termination**	**Sort**	**Crossover Method**	**Initialization**	
Range	[True, False]	N/A	[Raw, Scaled]	[SinglePoint, TwoPoint, OX, Edge, CutCrossFill, Uniform]	[Default, Random, Proportional Random]	
Default	False	N/A	Scaled	SinglePoint	Random	

Table 5.2: List of GA Parameters and Their Defaults

#Run	Throughput ops/s	File System	Block Size	Blocks per group	Inode Size	Mount Options	Journal Options	Special Options
Run 1	1,639	Ext4	2,048	4	128	noatime	data=writeback	N/A
Run 2	3,669	Nilfs2	2,048	256	N/A	atime	order=relaxed	N/A
Run 3	1,903	Ext2	4,096	8	512	noatime	N/A	N/A
Run 4	3,677	Nilfs2	2,048	256	N/A	atime	order=relaxed	N/A
Run 5	3,669	Nilfs2	2,048	256	N/A	noatime	order=relaxed	N/A
Default	1,420	Ext4	4,096	32	256	atime	data=ordered	N/A
Best	3,677	Nilfs2	2,048	256	N/A	atime	order=relaxed	N/A

Table 5.3: Results of GA experiments on M1 for Mailserver workload

Section 6.

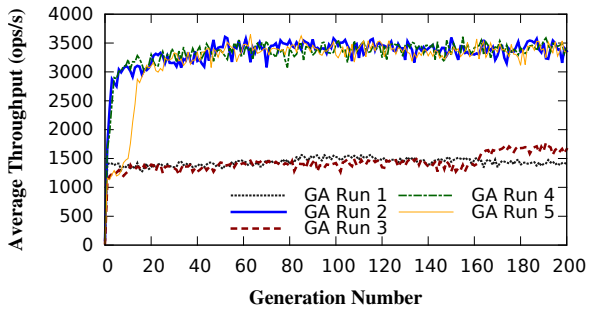
Figure 5.1a shows the average throughput of the population at the end of each generation, over 200 generations, for all five runs. The fluctuations seen are a natural outcome of the search process for GAs—sometimes finding better while sometimes worse configurations.

Figure 5.1b is similar but instead records only the best maximum throughput found up to the given generation. The stair-step pattern represents plateaus in our search terrain with higher fitness (e.g., a better peak in Figure 2.2). We can see that once GA finds a better configuration, it is carried over to future generations and reinforced with the *Elitism* mechanism. Recording the best seen configuration so far takes little effort and means that GAs generally get better over time.

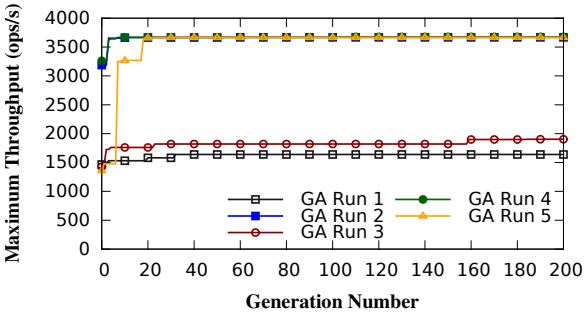
Figure 5.1c shows the maximum throughput by clock time instead of generation number. We can see that for Runs 1 and 3, GAs actually get "stuck" in a local optima (Ext2 and Ext4 configurations), without ever having a chance to "jump" to the better Nilfs2 configurations. This is partially because we use a random initialization method. As we can see from Figure 4.1, Nilfs has fewer valid configurations than Ext3 or Ext4. When the initial population is randomly generated, Nilfs2 would have a much lower probability of being chosen. This demonstrates the need to help the GA process along: tuning its parameters so it can explore better areas of the search space, and perhaps [re]seeding its population according to the shape of the search space.

Figure 5.1 also shows the importance of investigating appropriate stopping criteria. In most runs, 50 generations would have been enough—but for Run 3, we needed more than 150 generations to find the better configuration.

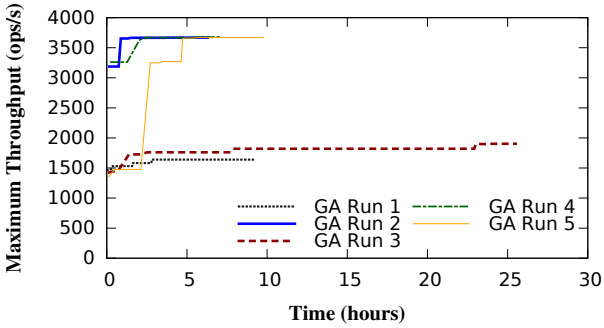
To further investigate the impact of different initialization methods on the results of GAs, we conducted



(a) Avg. Throughput vs. Generation

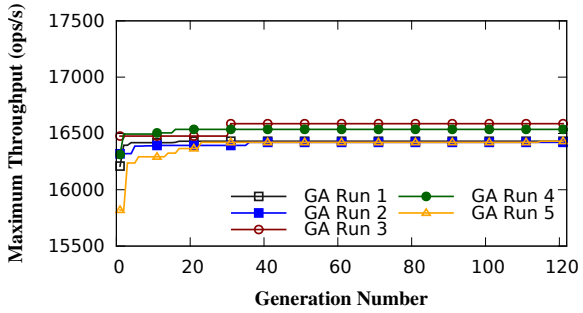


(b) Max Throughput vs. Generation

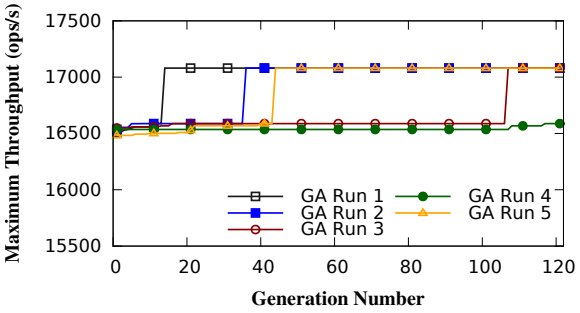


(c) Max Throughput vs. Time

Figure 5.1: GA results for *mailserver* workload on M1 machines



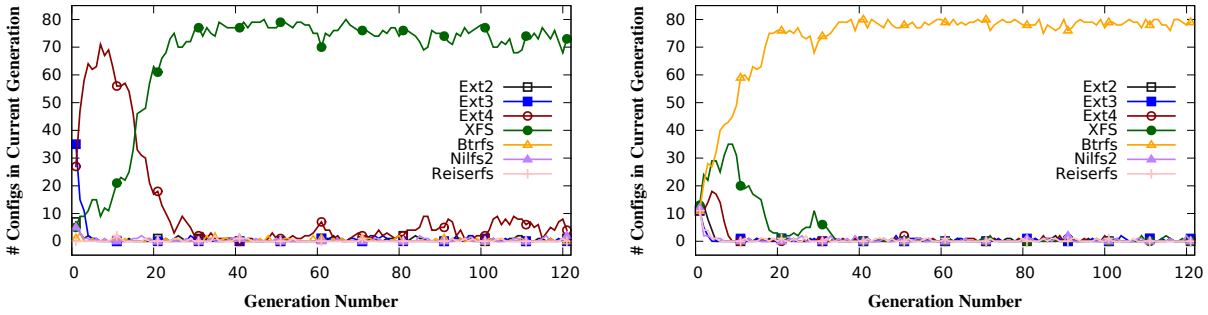
(a) Random Initialization



(b) Fair Initialization

Figure 5.2: GA results for *fileserver* workload on M2 machines

two additional sets of experiments. Figure 5.2a shows 5 GAs runs for the *fileserver* workload on M2 with random initialization; this means that each configuration in the search space has an equal chance to be picked into the initial population. All other GA parameters are set to default as shown in Table 5.2, except that the total number of generations is 120. Note that the global best for this environment is Btrfs (see Table 5.1.) None of the 5 GA runs find the global optimal configuration; they all got "stuck" in a different local optima. GA Runs 1, 2, and 5 converge to XFS configurations. GA Runs 3 and 4 do pick Btrfs configurations, they still fail to jump to the global optimum. We then conducted a similar set of experiments, only changing the



(a) GA Run 5 from Figure 5.2a

(b) GA Run 5 from Figure 5.2b

Figure 5.3: Number of Configurations for Each FS Type

initialization method. In this case all file system types have nearly equal proportion in the initial population. As we can see from Figure 5.2b, 4 out of 5 GA runs successfully found the global optimal configuration. This "fair initialization" outperforms the random initialization, as it seeds the initial population with diverse genes.

We further illustrates the problem by comparing the proportion of configurations of each file system during the search process. Figure 5.3a depicts GA Run 5 from Figure 5.2a. Clearly after Generation 20, XFS configurations dominate the population, beating Ext4 configurations, which were the majority before Generation 20. Btrfs does appear several times during this run, thanks to mutation. However, it soon becomes extinct. Figure 5.3b shows the details of file system proportions during GA Run 5 from Figure 5.2b. In this case, Btrfs dominates the population very quickly, beating all the other file systems. The above analysis indicates that guaranteeing the diversity of the population in the very first stages of the search process may be critical to the efficacy of the algorithms. We need more experiments to solidify this argument, which will be part of our future work: to explore different parameter settings of various search algorithms and to understand how to quickly and accurately converge to the global optimum.

Another finding from Figure 5.3 is that the Rank Selection method may not work well for certain search spaces. It selects configurations from the current generation completely by the rank of their fitness values, no matter how close they are. Consider the situation that Configuration A has a fitness value of 16,000 and ranks 1^{st}, while Configuration B has a fitness values of 15,950 and ranks 50^{th} in the population. If Rank Selection is applied, Configuration A will have a much higher probability of being picked than Configuration B does. This is why one type of file system becomes dominant in the population for all GA runs in Figure 5.2. It can easily cause the population to lose gene diversity, and get stuck in local optima. We can also conclude from Figure 5.3 that in this particular environment most XFS configurations will outperform configurations of other file system types except Btrfs, while most Btrfs configurations outperform the XFS ones. This may indicate that different parameters might have different impact on the fitness. Here, the file system type seems to be the dominating parameter. If we can prove the generality of this finding, it will largely benefit our search algorithms, as we can directly eliminate less important parameters, and thus reduce the search space exponentially. We provide more discussion on this topic in Section 5.4.

5.3 Machine Learning

In this section we discuss the practicality of applying Machine Learning (ML) techniques to solve our problem. We conducted experiments using the *scikit-learn* library [316]. We use a Decision Tree model [39] to train on a subset of our dataset and test the accuracy of the model by making predictions on the rest of the

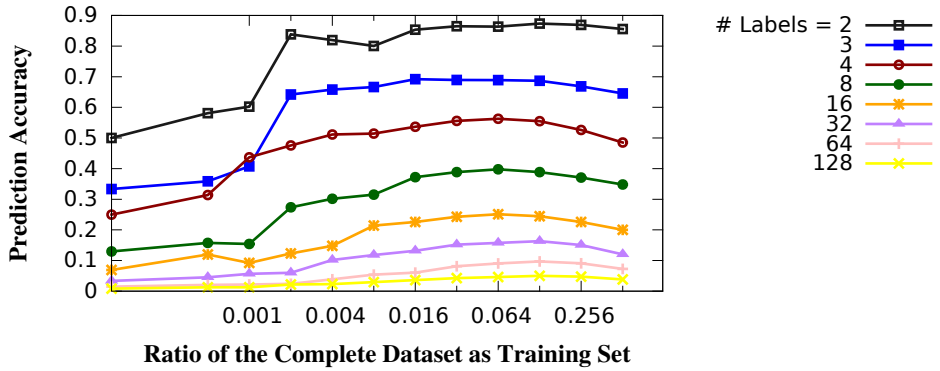


Figure 5.4: Prediction Accuracy of Decision Tree Algorithms with Varying Size of Training Set and Number of Classes (mailserver)

dataset. We label our dataset by dividing the real-valued throughputs into several subranges. For example, configurations with throughput values ranging from 1–10,000 belong to Category 1 and configurations with throughput values from 10,000–20,000 are Category 2. After this conversion, each category has an equal number of configurations.

Figure 5.4 shows the results from the mailserver dataset. As we can see, when the number of classes becomes larger than 8, the algorithm has a poor prediction accuracy, even if half of the dataset is used for training. This suggest that Machine Learning techniques may not be useful always to predict metrics like the throughput with acceptable accuracy. It also supports our argument that supervised Machine Learning techniques usually need a large and high-quality dataset for training use, while in our problem, there exist no such datasets. However, when the number of labels drops to 2, the prediction accuracy is pretty high, even only with 0.2% of the whole dataset. This inspires us to explore another direction of applying ML techniques in our optimization: using ML as a supplemental tool to our search algorithm. Assume that Label 1 means high throughput while Label 2 means low throughput. In this case, ML algorithms can tell our search process to try more configurations with Label 1 while avoiding going into areas with lots of Label 2 configurations.

We conducted the same experiments on the dbserver data set, shown in Figure 5.5. Compared with mailserver, dbserver generally needs more training data in order to achieve the same prediction accuracy, which may indicate that dbserver workloads contains fewer patterns and more "randomness" than mailserver does. This actually increases the complexity of our optimization.

5.4 Feature Selection

In this section we mainly talk about feature selection, or dimensionality reduction, which can serve as a powerful supplemental tool to our search algorithm. If we can find out unrelated parameters and remove them from the following search process, it will exponentially reduce our search space. We use Mutual Information (MI) [245] to measure the dependence of each parameter to the throughput in our datasets. MI is often used in probability theory and information theory as a measure of the mutual dependence of

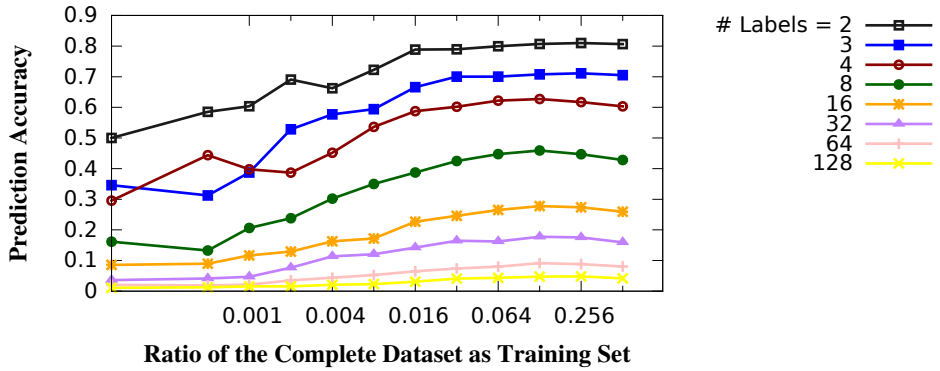


Figure 5.5: Prediction Accuracy of Decision Tree Algorithms with Varying Size of Training Set and Number of Classes (dbserver)

variables. For variable X and Y, MI is defined as:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log^{(\frac{p(x,y)}{p(x)p(y)})}. \tag{5.1}$$

When $I(X;Y) = 0$, X and Y are independent. Higher $I(X;Y)$ value indicates X and Y are more correlated.

Figure 5.6 depicts the mutual information of each parameter type with the throughput for the three workloads on M2 machines. Here we divide the real-values throughput into 6 classes. If the number of classes is set too large, over-fitting is likely to occur. We tried values from 3 to 8, and they all produce similar results.

From Figure 5.6 we can see that the File System type is the most important parameter for all three workloads. It has the largest impact on the throughput, which aligns with our findings shown in Figure 5.3. The parameter with the second highest MI depends on the specific workload. For *mailserver* and *fileserver*: it is the Disk Type. For *dbserver*, however, it is the Journal Options. On the opposite side, the MI values for Mount Option and I/O Scheduler nearly equal zero for all three workloads; this means that they are almost independent of the throughput, and possibly we can remove them completely to reduce the search space.

As a follow-up, we fixed the file system type, and calculated the MI values of every parameter with the throughput within each file system. The results are shown in Figure 5.7 (*mailserver*) and Figure 5.8 (*dbserver*). Surprisingly, under the *mailserver* workload, the I/O Scheduler turns out to be the highest correlated parameter in XFS and Reiserfs, which has an almost zero MI value in Figure 5.6. Inode Size is the most important one for Ext4 and Ext2; while Disk Type has the highest MI values for Ext3. This may indicate that the importance of some parameters depends highly on the specific file systems; this makes sense, as different file systems may be designed for fairly different goals. If the search space contains multiple file systems, we may even come up with different search strategies to handle them.

If we can combine this kind of feature selection procedure dynamically into our search algorithm, it should have a significant impact on the efficiency of the optimization process. However, as the MI values are also dependent on the environment, when the environment changes, the algorithm will need to somehow re-think its previous conclusions (e.g., add back eliminated parameters).

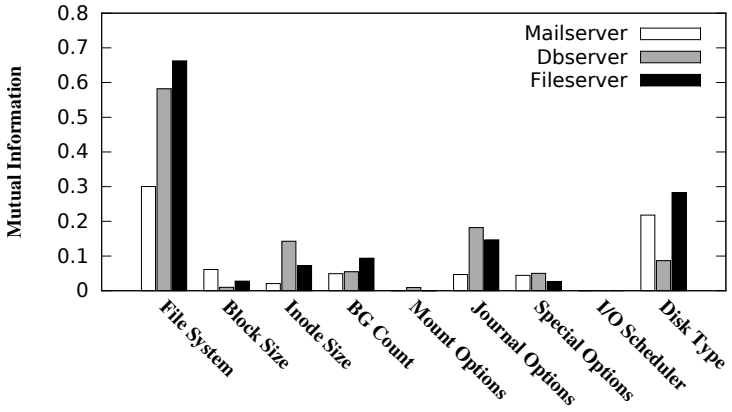


Figure 5.6: Mutual Information for Different Parameter Types

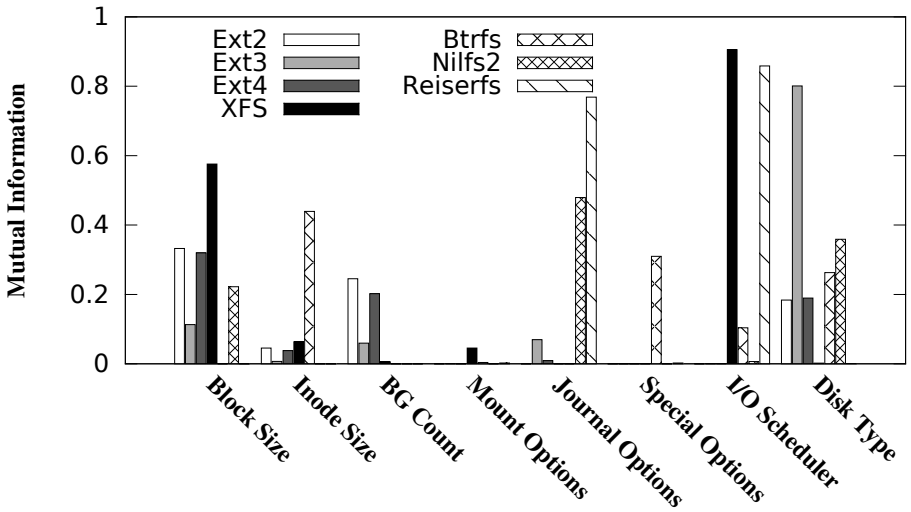


Figure 5.7: Mutual Information for Different Parameters Within Each File System (mailserver)

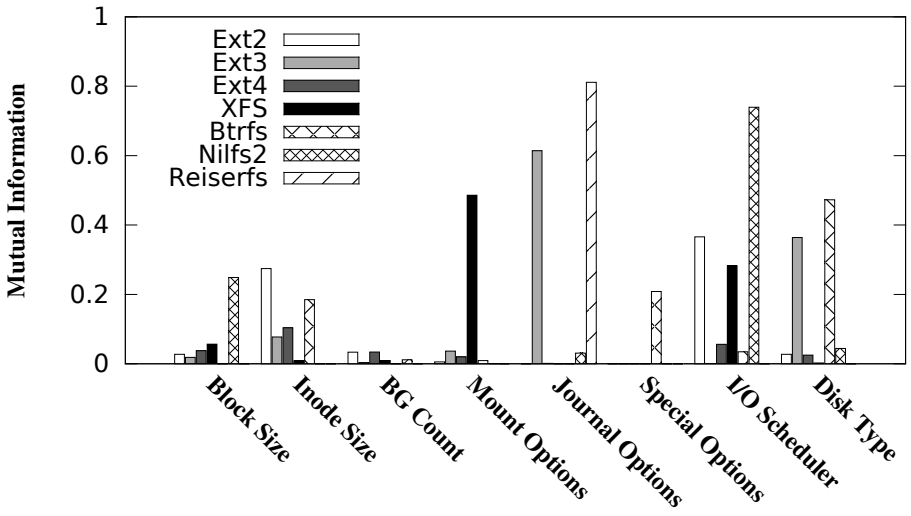


Figure 5.8: Mutual Information for Different Parameter Within Each File System (dbserver)

Chapter 6

Future Work

Through the discussion on previous sections, we can see that optimizing storage systems is a problem with high complexity and difficulty. Despite already having some progress, we are still far away from achieving our ultimate goal, which is to auto-tune any storage systems with various goals. We feel the following points are especially interesting to investigate:

- *Dimensionality Reduction*. As discussed in Section 5.4, it can serve as a powerful supplement tool to our search algorithm. If we can find out unrelated parameters and remove them from the following search process, it will exponentially reduce our search space. We showed in Figure 5.6 that some parameters are highly correlated with the throughput, which means they could have huge impacts on the performance of storage systems. Some others are less important, and can be possibly eliminated from our search space. It is still unclear how general is this approach, but we believe that we are on the right direction. When a human expert manually tunes a system, it is unlikely that he will try all combination of parameters. He will try to tune the most important parameters first, according to his domain knowledge. What we are doing here is letting the machine learn to find these "important" parameters through a process of trial-and-error. In the future we will explore more feature selection methods, and combine them with our main search algorithm.

- *Instability*. During our data collection process, we found that the throughputs of many configurations are not stable, especially for the *fileserver* and *dbserver* workload. It is still unknown whether this is caused by the environment or the internals of some file systems. Nevertheless, it will greatly affect our search algorithm, as nearly all conventional global search heuristics assume that the fitness values are stable. Simply trying each configuration for more times is not a feasible solution, which significantly adds the time taken by the search process and reduces the efficiency of our algorithm. One idea is to associate a confidence level with each fitness value of configurations.

- *Stopping Criteria*. Good stopping criteria are critical for the effectiveness of our algorithm. We already see from Section 5.2 that sometimes GAs will spend a long time searching the space without finding any better configurations. It is really an open question of when we should stop our algorithm, and simple criteria like sliding window based methods will not suffice. We plan to investigate various stopping criteria proposed on all kinds of techniques, and devise our own version that works best for storage systems.

- *Workload Recognition*. This is also important as we want our algorithm to sense the changes in the environment, and restart the search process if needed. It is also an difficult research problem, as it is unclear what features will be enough to accurately define a workload, and separate it with all other workloads. Benchmarks such as SPEC SFS® 2014 [331] list several of these features: file size

distribution, directory structure, I/O operation distribution, I/O size distribution, etc. We will explore this topic in the future.

- *Multi-Objective*. Another direction for auto-tuning systems is optimizing for multiple objectives. It has many applications in reality. For a distributed file system shared with many users, users may run various application on it with quite different goals. Optimizing for one single metric may cause the system to perform poorly in terms of other objectives. This has been a hot research topic for several decades, and is likely to remain so. In the end we will add multi-objective support to our search algorithm as well.

We only list several of the most important and interesting points above, and there are many more for us to explore. Today, people attempt to optimize complex storage systems manually. This is akin to the old days of writing and optimizing assembly programs. Modern compilers, however, largely eliminated the need to write assembly code manually. Our vision is that every future storage system will come with a self-tuning module that will obsolete the need to hand-tune storage systems ever again.

Chapter 7

Conclusion

In this report we make the very first steps towards an ambitious yet promising goal of auto-tuning storage systems. We model the optimization of storage systems as the problem of searching for the global optimum in a high-dimensional space. Our problem is characterized by its high dimensionality, the sensitivity of configurations to the environment, and the difficulty of evaluation. We argue that Meta-Heuristics (MH) techniques have the ability to solve this kind of problem by maintaining a trade-off among *exploration*, *exploitation*, and *history*. Our experimental results show the promise of MH techniques in finding the global optima accurately and efficiently. In addition, although Machine Learning (ML) techniques are not directly applicable to solving our optimization tasks, we still find them helpful to supplement to our main search algorithms. In the future, we will continue investigating various techniques, and we plan to devise a hybrid algorithm that combines the essential properties of many search approaches, as well as techniques like dimensionality reduction and supervised ML. Another part of our future work is to perform online tuning, where various practical issues will arise (i.e., when to stop and when to restart optimizing when the environment changes). Our ultimate goal is an auto-tuning module that can optimize for any storage system and for any metrics, so that there is no need for manually tuning ever again.

Bibliography

[1] Emile Aarts and Jan Korst. *Simulated annealing and Boltzmann machines*. New York, NY; John Wiley and Sons Inc., 1988.

[2] Emile Aarts, Jan Korst, and Wil Michiels. Simulated annealing. In *Search methodologies*, pages 187–210. Springer, 2005.

[3] Charu C Aggarwal, James B Orlin, and Ray P Tai. Optimized crossover for the independent set problem. *Operations Research*, 45(2):226–234, 1997.

[4] Ravindra K Ahuja and James B Orlin. Developing fitter genetic algorithms. *INFORMS Journal on Computing*, 9(3):251–253, 1997.

[5] Ravindra K Ahuja, James B Orlin, and Ashish Tiwari. A greedy genetic algorithm for the quadratic assignment problem. *Computers & Operations Research*, 27(10):917–934, 2000.

[6] Guillermo A. Alvarez, Elizabeth Borowsky, Susie Go, Theodore H. Romer, Ralph Becker-Szendy, Richard Golding, Arif Merchant, Mirjana Spasojevic, Alistair Veitch, and John Wilkes. Minerva: An automated resource provisioning tool for large-scale storage systems. *ACM Trans. Comput. Syst.*, 19(4):483–518, November 2001.

[7] Bjarne Andersen and Jeffrey M Gordon. Constant thermodynamic speed for minimizing entropy production in thermodynamic processes and simulated annealing. *Physical Review E*, 50(6):4346, 1994.

[8] Bjarne Anderson. Finite-time thermodynamics and simulated annealing. In *Entropy and entropy generation*, pages 111–127. Springer, 2002.

[9] Jaroslaw Arabas, Zbigniew Michalewicz, and Jan Mulawka. Gavaps-a genetic algorithm with varying population size. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 73–78. IEEE, 1994.

[10] Haldun Aytug and Gary J Koehler. Stopping criteria for finite length genetic algorithms. *INFORMS Journal on Computing*, 8(2):183–191, 1996.

[11] Haldun Aytug and Gary J Koehler. New stopping criterion for genetic algorithms. *European Journal of Operational Research*, 126(3):662–674, 2000.

[12] Nader Azizi and Saeed Zolfaghari. Adaptive temperature control for simulated annealing: a comparative study. *Computers & Operations Research*, 31(14):2439–2451, 2004.

[13] Thomas Bäck. Selective pressure in evolutionary algorithms: A characterization of selection mechanisms. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 57–62. IEEE, 1994.

[14] Thomas Bäck, A. E. Eiben, and N. A. L. van der Vaart. An empirical study on gas "without parameters". In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, PPSN VI, pages 315–324, London, UK, UK, 2000. Springer-Verlag.

[15] James E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 101–111, Hillsdale, NJ, USA, 1985. L. Erlbaum Associates Inc.

[16] James E Baker. Reducing bias and inefficiency in the selection algorithm. In *Proceedings of the second international conference on genetic algorithms*, pages 14–21, 1987.

[17] AA Barker. Monte carlo calculations of the radial distribution functions for a proton? electron plasma. *Australian Journal of Physics*, 18(2):119–134, 1965.

[18] John E Beasley and Paul C Chu. A genetic algorithm for the set covering problem. *European Journal of Operational Research*, 94(2):392–404, 1996.

[19] Babak Behzad, Joey Huchette, Huong Luu, Ruth Aydt, Quincey Koziol, Mr Prabhat, Suren Byna, Mohamad Chaarawi, and Yushu Yao. Auto-tuning of parallel io parameters for hdf5 applications. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1430–, Washington, DC, USA, 2012. IEEE Computer Society.

[20] Babak Behzad, Huong Vu Thanh Luu, Joseph Huchette, Surendra Byna, Prabhat, Ruth Aydt, Quincey Koziol, and Marc Snir. Taming parallel i/o complexity with auto-tuning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 68:1–68:12, New York, NY, USA, 2013. ACM.

[21] E.D. Beinhocker. *The Origin of Wealth: Evolution, Complexity, and the Radical Remaking of Economics*. Harvard Business School Press, 2006.

[22] Theodore C Belding. The distributed genetic algorithm revisited. *arXiv preprint adap-org/9504007*, 1995.

[23] Richard K. Belew, John Mcinerney, and Nicol N. Schraudolph. Evolving networks: Using the genetic algorithm with connectionist learning. In Christopher G. Langton, Charles Taylor, Doyne J. Farmer, and Steen Rasmussen, editors, *Artificial Life II*, pages 511–547. Addison-Wesley, Redwood City, CA, 1992.

[24] V. Beltran, J. Torres, and E. Ayguade. Understanding tuning complexity in multithreaded and hybrid web servers. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12, April 2008.

[25] Walid Ben-Ameur. Computing the initial temperature of simulated annealing. *Computational Optimization and Applications*, 29(3):369–385, 2004.

[26] Hugues Bersini and Francisco J Varela. Hints for adaptive problem solving gleaned from immune networks. In *Parallel problem solving from nature*, pages 343–354. Springer, 1991.

[27] Dimitris Bertsimas, John Tsitsiklis, et al. Simulated annealing. *Statistical science*, 8(1):10–15, 1993.

[28] MC Bhuvaneswari. *Application of Evolutionary Algorithms for Multi-objective Optimization in VLSI and Embedded Systems*. Springer, 2015.

[29] Christian Bierwirth. A generalized permutation approach to job shop scheduling with genetic algorithms. *Operations-Research-Spektrum*, 17(2-3):87–92, 1995.

[30] Christian Bierwirth, Dirk C Mattfeld, and Herbert Kopfer. On permutation representations for scheduling problems. In *Parallel Problem Solving from NaturePPSN IV*, pages 310–318. Springer, 1996.

[31] Christopher M Bishop et al. *Pattern recognition and machine learning*, volume 1. springer New York, 2006.

[32] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 23–23, Berkeley, CA, USA, 1995. USENIX Association.

[33] Joe L Blanton Jr and Roger L Wainwright. Multiple vehicle routing with time and capacity constraints using genetic algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 452–459. Morgan Kaufmann Publishers Inc., 1993.

[34] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

[35] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. http://hub.opensolaris.org/bin/view/Community+Group+zfs/docs, 2010.

[36] L. Booker. *Intelligent Behavior as a Adaptation to the Task Environment*. Phd thesis, University of Michigan, 1982.

[37] Lashon Booker. Improving search in genetic algorithms. *Genetic algorithms and simulated annealing*, pages 61–73, 1987.

[38] Mark F Bramlette. Initialization, mutation and selection methods in genetic algorithms for function optimization. In *ICGA*, pages 100–107, 1991.

[39] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[40] Janez Brest, Sašo Greiner, Borko Bošković, Marjan Mernik, and Viljem Zumer. Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. *Evolutionary Computation, IEEE Transactions on*, 10(6):646–657, 2006.

[41] A. Brindle. *Genetic Algorithms for Function Optimization*. Phd thesis, University of Alberta, 1981.

[42] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. A reinforcement learning approach to online web systems auto-configuration. In *Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, ICDCS '09, pages 2–11, Washington, DC, USA, 2009. IEEE Computer Society.

[43] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Trans. Parallel Distrib. Syst.*, 24(4):681–690, April 2013.

[44] Thang Nguyen Bui and Byung Ro Moon. Genetic algorithm and graph partitioning. *Computers, IEEE Transactions on*, 45(7):841–855, 1996.

[45] Gerardo Canfora, Massimiliano Di Penta, Raffaele Esposito, and Maria Luisa Villani. An approach for qos-aware service composition based on genetic algorithms. In *Proceedings of the 7th annual conference on Genetic and evolutionary computation*, pages 1069–1075. ACM, 2005.

[46] R. Card, T. Ts'o, and S. Tweedie. Design and implementation of the second extended filesystem. In *Proceedings to the First Dutch International Symposium on Linux*, Seattle, WA, December 1994.

[47] Richard A Caruana, Larry J Eshelman, and J David Schaffer. Representation and hidden bias ii: Eliminating defining length bias in genetic search via shuffle crossover. In *Proceedings of the 11th international joint conference on Artificial intelligence-Volume 1*, pages 750–755. Morgan Kaufmann Publishers Inc., 1989.

[48] Vladimír Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of optimization theory and applications*, 45(1):41–51, 1985.

[49] Partha Chakroborty, Kalyanmoy Deb, and PS Subrahmanyam. Optimal scheduling of urban transit systems using genetic algorithms. *Journal of transportation Engineering*, 1995.

[50] M. Chen, D. Hildebrand, G. Kuenning, S. Shankaranarayana, V. Tarasov, A. Vasudevan, E. Zadok, and K. Zakirova. Linux NFSv4.1 Performance Under a Microscope. In *Proceedings of USENIX LISA*. USENIX Association, November 2014. Extended Abstract.

[51] Y. Chen, M. Winslett, Y. Cho, and S. Kuo. Automatic parallel i/o performance optimization using genetic algorithms. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, HPDC '98, pages 155–, Washington, DC, USA, 1998. IEEE Computer Society.

[52] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithmsi. representation. *Computers & industrial engineering*, 30(4):983–997, 1996.

[53] Runwei Cheng, Mitsuo Gen, and Yasuhiro Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms, part ii: hybrid genetic search strategies. *Computers & Industrial Engineering*, 36(2):343–364, 1999.

[54] Marco Chiarandini, Mauro Birattari, Krzysztof Socha, and Olivia Rossi-Doria. An effective hybrid algorithm for university course timetabling. *Journal of Scheduling*, 9(5):403–432, 2006.

[55] Paul C Chu and John E Beasley. A genetic algorithm for the generalised assignment problem. *Computers & Operations Research*, 24(1):17–23, 1997.

[56] Maurice Clerc. *Particle swarm optimization*, volume 93. John Wiley & Sons, 2010.

[57] CloudSuite. http://parsa.epfl.ch/cloudsuite/cloudsuite.html.

[58] Cluster Analysis. https://en.wikipedia.org/wiki/Cluster_analysis.

[59] Harry Cohn and Mark Fielding. Simulated annealing: searching for an optimal temperature schedule. *SIAM Journal on Optimization*, 9(3):779–802, 1999.

[60] James Cohoon, John Kairo, and Jens Lienig. Evolutionary algorithms for the physical design of vlsi circuits. In *Advances in evolutionary computing*, pages 683–711. Springer, 2003.

[61] Zack Coker, David Garlan, and Claire Le Goues. Sass: Self-adaptation using stochastic search. In *Proceedings 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2015)*, 2015.

[62] Joao Carlos Costa, Rui Tavares, and Agostinho Rosa. An experimental study on dynamic random variation of population size. In *Systems, Man, and Cybernetics, 1999. IEEE SMC'99 Conference Proceedings. 1999 IEEE International Conference on*, volume 1, pages 607–612. IEEE, 1999.

[63] Yves Crama and Michaël Schyns. Simulated annealing for complex portfolio selection problems. *European Journal of operational research*, 150(3):546–571, 2003.

[64] M. Crouse and E.W. Fulp. A moving target environment for computer configurations using genetic algorithms. In *Configuration Analytics and Automation (SAFECONFIG), 2011 4th Symposium on*, pages 1–7, Oct 2011.

[65] Michael Crouse, Errin W Fulp, and Daniel Canas. Improving the diversity defense of genetic algorithm-based moving target approaches. In *Proceedings of the National Symposium on Moving Target Research*, 2012.

[66] Fernando Miguel Pais da Graça Lobo. *The parameter-less genetic algorithm: rational and automated parameter selection for simplified genetic algorithm operation*. PhD thesis, Universidade Nova de Lisboa, 2000.

[67] Charles Darwin. On the origins of species by means of natural selection. *London: Murray*, page 247, 1859.

[68] Swagatam Das and Ponnuthurai Nagaratnam Suganthan. Differential evolution: a survey of the state-of-the-art. *Evolutionary Computation, IEEE Transactions on*, 15(1):4–31, 2011.

[69] Lawrence Davis. Job shop scheduling with genetic algorithms. In *Proceedings of an international conference on genetic algorithms and their applications*, volume 140. Carnegie-Mellon University Pittsburgh, PA, 1985.

[70] Lawrence Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.

[71] Kenneth A De Jong and William M Spears. A formal analysis of the role of multi-point crossover in genetic algorithms. *Annals of mathematics and Artificial intelligence*, 5(1):1–26, 1992.

[72] Kenneth A De Jong, William M Spears, and Diana F Gordon. Using markov chains to analyze gafos. *Foundations of genetic algorithms*, 3:115–137, 1995.

[73] Kenneth Alan De Jong. *Analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan, Ann Arbor, Ann Arbor, MI, USA, 1975.

[74] Pablo de Oliveira Castro, Yuriy Kashnikov, Chadi Akel, Mihail Popov, and William Jalby. Fine-grained benchmark subsetting for system selection. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 132. ACM, 2014.

[75] Federico Della Croce, Roberto Tadei, and Giuseppe Volta. A genetic algorithm for the job shop problem. *Computers & Operations Research*, 22(1):15–24, 1995.

[76] Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, and D. M. Tilbury. Using MIMO feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Proceedings of the Network Operations and Management Symposium*, pages 219–234, 2002.

[77] Y. Diao, J. L. Hellerstein, A. J. Storm, M. Surendra, S. Lightstone, S. Parekh, and C. Garcia-Arellano. Using MIMO linear control for load balancing in computing systems. In *2004 American Control Conferences*, 2004.

[78] Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh. Optimizing quality of service using fuzzy control. In *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, DSOM '02, pages 42–53, London, UK, UK, 2002. Springer-Verlag.

[79] Elizabeth Dicke, Andrew Byde, Paul Layzell, and Dave Cliff. Using a genetic algorithm to design and improve storage area network architectures. In Kalyanmoy Deb, editor, *Genetic and Evolutionary Computation – GECCO 2004*, volume 3102 of *Lecture Notes in Computer Science*, pages 1066–1077. Springer Berlin Heidelberg, 2004.

[80] Marco Dorigo and Mauro Birattari. Ant colony optimization. In *Encyclopedia of machine learning*, pages 36–39. Springer, 2010.

[81] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *Computational Intelligence Magazine, IEEE*, 1(4):28–39, 2006.

[82] Ulrich Dorndorf and Erwin Pesch. Evolution based learning in a job shop scheduling environment. *Computers & Operations Research*, 22(1):25–40, 1995.

[83] Fred Douglis, Deepti Bhardwaj, Hangwei Qian, and Philip Shilane. Content-aware load balancing for distributed backup. In *LISA*, 2011.

[84] Zvi Drezner. A new genetic algorithm for the quadratic assignment problem. *INFORMS Journal on Computing*, 15(3):320–330, 2003.

[85] Songyun Duan, Vamsidhar Thummala, and Shivnath Babu. Tuning database configuration parameters with ituned. *Proc. VLDB Endow.*, 2(1):1246–1257, August 2009.

[86] Manuel Duque-Antón, Dietmar Kunz, and Bernhard Rüber. Channel assignment for cellular radio using simulated annealing. *Vehicular Technology, IEEE Transactions on*, 42(1):14–21, 1993.

[87] Russell C Eberhart and Yuhui Shi. Particle swarm optimization: developments, applications and resources. In *Evolutionary Computation, 2001. Proceedings of the 2001 Congress on*, volume 1, pages 81–86. IEEE, 2001.

[88] A.E. Eiben and C.A. Schippers. On evolutionary exploration and exploitation. *Fundam. Inf.*, 35(1-4):35–50, January 1998.

[89] AE Eiben, Martijn C Schut, and AR de Wilde. Is self-adaptation of selection pressure and population size possible?–a case study. In *Parallel Problem Solving from Nature-PPSN IX*, pages 900–909. Springer, 2006.

[90] Agoston E Eiben and Thomas Bäck. Empirical investigation of multiparent recombination operators in evolution strategies. *Evolutionary Computation*, 5(3):347–365, 1997.

[91] Agoston E. Eiben, Cees H.M. Kemenade, and Joost N. Kok. Orgy in the computer: Multi-parent reproduction in genetic algorithms. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1995.

[92] Agoston E Eiben, P-E Raue, and Zs Ruttkay. Genetic algorithms with multi-parent recombination. In *Parallel Problem Solving from NaturePPSN III*, pages 78–87. Springer, 1994.

[93] Ágoston E Eiben and CA Schippers. Multi-parent's niche: n-ary crossovers on nk-landscapes. In *Parallel Problem Solving from NaturePPSN IV*, pages 319–328. Springer, 1996.

[94] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing*. Springer Science & Business Media, 2003.

[95] Agoston Endre Eiben, Elena Marchiori, and VA Valko. Evolutionary algorithms with on-the-fly population size adjustment. In *Parallel Problem Solving from Nature-PPSN VIII*, pages 41–50. Springer, 2004.

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

[97] IT electricity consumption. http://www.theregister.co.uk/2013/08/16/it_electricity_use_worse_than_you_thought/.

[98] Larry J. Eshelman, Rich Caruana, and J. David Schaffer. Biases in the crossover landscape. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 10–19, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[99] Ext4. http://ext4.wiki.kernel.org/.

[100] Emanuel Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of heuristics*, 2(1):5–30, 1996.

[101] Emanuel Falkenauer and Alain Delchambre. A genetic algorithm for bin packing and line balancing. In *Robotics and Automation, 1992. Proceedings., 1992 IEEE International Conference on*, pages 1186–1192. IEEE, 1992.

[102] Hsiao-Lan Fang, Peter Ross, and Dave Corne. A promising genetic algorithm approach to job-shop schedulingre-schedulingand open-shop scheduling problems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 375–382, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[103] J Doyne Farmer, Norman H Packard, and Alan S Perelson. The immune system, adaptation, and machine learning. *Physica D: Nonlinear Phenomena*, 22(1):187–204, 1986.

[104] Filebench, 2016. http://filebench.sf.net.

[105] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM.

[106] George S Fishman. *Monte carlo: concepts, algorithms, and applications*. Springer Series in Operations Research. Springer, New York, 1996.

[107] Charles Fleurent and Jacques A Ferland. Genetic hybrids for the quadratic assignment problem. *Quadratic assignment and related problems*, 16:173–187, 1994.

[108] Lawrence J Fogel. *Intelligence through simulated evolution: forty years of evolutionary programming*. John Wiley & Sons, Inc., 1999.

[109] Terry L Friesz, Hsun-Jung Cho, Nihal J Mehta, Roger L Tobin, and G Anandalingam. A simulated annealing approach to the network design problem with variational inequality constraints. *Transportation Science*, 26(1):18–26, 1992.

[110] Kerry Gallagher, Malcolm Sambridge, and Guy Drijkoningen. Genetic algorithms: An evolution from monte carlo methods for strongly non-linear geophysical optimization problems. *Geophysical Research Letters*, 18(12):2177–2180, 1991.

[111] RA Gallego, AB Alves, A Monticelli, and R Romero. Parallel simulated annealing applied to long term transmission network expansion planning. *Power Systems, IEEE Transactions on*, 12(1):181–188, 1997.

[112] Amir Hossein Gandomi, Xin-She Yang, and Amir Hossein Alavi. Cuckoo search algorithm: a metaheuristic approach to solve structural optimization problems. *Engineering with computers*, 29(1):17–35, 2013.

[113] Gregory R Ganger, John D Strunk, and Andrew J Klosterman. Self-* storage: Brick-based storage with automated administration. Technical Report CMU-CS-03-178, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2003.

[114] Shravan Gaonkar, Kimberly Keeton, Arif Merchant, and William H. Sanders. Designing dependable storage solutions for shared application environments. *IEEE Trans. Dependable Secur. Comput.*, 7(4):366–380, October 2010.

[115] Zong Woo Geem, Joong Hoon Kim, and GV Loganathan. A new heuristic optimization algorithm: harmony search. *Simulation*, 76(2):60–68, 2001.

[116] Robert Geist, Darrell Suggs, Robert Reynolds, Shardul Divatia, Fred Harris, Evan Foster, and Priyadarshan Kolte. Disk performance enhancement through markov-based cylinder remapping. In *Proceedings of the 30th Annual Southeast Regional Conference*, ACM-SE 30, pages 23–28, New York, NY, USA, 1992. ACM.

[117] Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 6:721–741, 1984.

[118] Michel Gendreau, Gilbert Laporte, and Frédéric Semet. A tabu search heuristic for the undirected selective travelling salesman problem. *European Journal of Operational Research*, 106(2):539–545, 1998.

[119] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[120] Genetic Programming. http://www.genetic-programming.com/.

[121] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. Bigbench: towards an industry standard benchmark for big data analytics. In *Proceedings of the 2013 ACM SIGMOD international conference on Management of data*, pages 1197–1208. ACM, 2013.

[122] F. Glover. Tabu Search – Part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[123] Fred Glover. Heuristics for integer programming using surrogate constraints. *Decision Sciences*, 8(1):156–166, 1977.

[124] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Comput. Oper. Res.*, 13(5):533–549, May 1986.

[125] Fred Glover. Tabu Search Part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[126] Fred Glover. Tabu search: A tutorial. *Interfaces*, 20(4):74–94, 1990.

[127] Fred Glover. A template for scatter search and path relinking. In *Artificial evolution*, pages 1–51. Springer, 1998.

[128] Fred Glover and Gary A Kochenberger. *Handbook of metaheuristics*. Springer Science & Business Media, 2003.

[129] Fred Glover and Manuel Laguna. *Tabu Search*. Springer, 2013.

[130] Fred Glover, Manuel Laguna, and Rafael Martí. Fundamentals of scatter search and path relinking. *Control and cybernetics*, 29(3):653–684, 2000.

[131] Fred Glover, Manuel Laguna, and Rafael Martí. Scatter search. In *Advances in evolutionary computing*, pages 519–537. Springer, 2003.

[132] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[133] David E Goldberg. Sizing populations for serial and parallel genetic algorithms. In *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 70–79. Morgan Kaufmann Publishers Inc., 1989.

[134] David E Goldberg. A note on boltzmann tournament selection for genetic algorithms and population-oriented simulated annealing. *Complex Systems*, 4(4):445–460, 1990.

[135] David E Goldberg. Genetic algorithms and the variance of fitness. *Complex Systems*, 5:265–278, 1991.

[136] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. *Urbana*, 51:61801–2996, 1991.

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

[138] David E Goldberg, Bradley Korb, and Kalyanmoy Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex systems*, 3(5):493–530, 1989.

[139] David E Goldberg and Robert Lingle. Alleles, loci, and the traveling salesman problem. In *Proceedings of the first international conference on genetic algorithms and their applications*, pages 154–159. Lawrence Erlbaum Associates, Publishers, 1985.

[140] David Edward Goldberg. *Optimal initial population size for binary-coded genetic algorithms*. Clearinghouse for Genetic Algorithms, Department of Engineering Mechanics, University of Alabama, 1985.

[141] José Fernando Gonçalves, Jorge José de Magalhães Mendes, and Maurício G.C. Resende. A hybrid genetic algorithm for the job shop scheduling problem. *European journal of operational research*, 167(1):77–95, 2005.

[142] Diana F Gordon. A multistrategy learning scheme for assimilating advice in embedded agents. Technical report, DTIC Document, 1993.

[143] V Scott Gordon, Darrell Whitley, and Anton Pedro Willem Böhm. *Dataflow parallelism in genetic algorithms*. Colorado State University, Department of Computer Science, 1992.

[144] Martina Gorges-Schleuter. Explicit parallelism of genetic algorithms through population structures. In *Parallel Problem Solving from Nature*, pages 150–159. Springer, 1991.

[145] David Greenhalgh and Stephen Marshall. Convergence criteria for genetic algorithms. *SIAM Journal on Computing*, 30(1):269–282, 2000.

[146] John Grefenstette, Rajeev Gopal, Brian Rosmaita, and Dirk Van Gucht. Genetic algorithms for the traveling salesman problem. In *Proceedings of the first International Conference on Genetic Algorithms and their Applications*, pages 160–168. Lawrence Erlbaum, New Jersey (160-168), 1985.

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

[148] John J Grefenstette. Incorporating problem specific knowledge into genetic algorithms. *Genetic algorithms and simulated annealing*, 4:42–60, 1987.

[149] John J. Grefenstette and James E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 20–27, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[150] Lov K Grover. A new simulated annealing algorithm for standard cell placement. In *Proceedings of the International Conference on Computer-Aided Design*, pages 378–380, 1986.

[151] Jianmei Guo, Jules White, Guangxin Wang, Jian Li, and Yinglin Wang. A genetic algorithm for optimized feature selection with resource constraints in software product lines. *Journal of Systems and Software*, 84(12):2208–2221, 2011.

[152] Anju Gupta and Parsh Ram Sharma. Optimization of power system performance using real parameter genetic algorithm. *International Journal of Artificial Intelligence and Knowledge Discovery*, 3(3):16–22, 2013.

[153] Omid Bozorg Haddad, Abbas Afshar, and Miguel A Mariño. Honey-bees mating optimization (hbmo) algorithm: a new heuristic approach for water resources optimization. *water resources management*, 20(5):661–680, 2006.

[154] Bruce Hajek. Cooling schedules for optimal annealing. *Mathematics of operations research*, 13(2):311–329, 1988.

[155] John Michael Hammersley and David Christopher Handscomb. *Monte carlo methods*, volume 1. Springer, 1964.

[156] Georges R Harik and Fernando G Lobo. A parameter-less genetic algorithm. In *GECCO*, volume 99, pages 258–267, 1999.

[157] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tibury. *Feedback Control of Computing Systems*. Wiley-IEEE Press, 2004.

[158] J. Hesser, R. Männer, and O. Stucky. Optimization of steiner trees using genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 231–236, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[159] HiBench. https://github.com/intel-hadoop/HiBench.

[160] Hill climbing. http://en.wikipedia.org/wiki/Hill_climbing.

[161] Robert Hinterding, Zbigniew Michalewicz, and Thomas C Peachey. Self-adaptive genetic algorithm for numeric functions. In *Parallel Problem Solving from NaturePPSN IV*, pages 420–429. Springer, 1996.

[162] Christian Hohn and Colin R Reeves. Graph partitioning using genetic algorithms. In *in Proceedings of the 2nd International Conference on Massively Parallel Computing Systems*. Citeseer, 1996.

[163] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U. Michigan Press, 1975.

[164] Christopher R Houck, Jeffrey A Joines, and Michael G Kay. Comparison of genetic algorithms, random restart and two-opt switching for solving large location-allocation problems. *Computers & Operations Research*, 23(6):587–596, 1996.

[165] E. Ipek, O. Mutlu, J.F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 39–50, June 2008.

[166] Hisao Ishibuchi, Tadashi Yoshida, and Tadahiko Murata. Balance between genetic search and local search in memetic algorithms for multiobjective permutation flowshop scheduling. *Evolutionary Computation, IEEE Transactions on*, 7(2):204–223, 2003.

[167] Young-Jae Jeon, Jae-Chul Kim, Jin-O Kim, Joong-Rin Shin, and Kwang Y Lee. An efficient simulated annealing algorithm for network reconfiguration in large-scale distribution systems. *Power Delivery, IEEE Transactions on*, 17(4):1070–1078, 2002.

[168] David J. John, Robert W. Smith, William H. Turkett, Daniel A. Cañas, and Errin W. Fulp. Evolutionary based moving target cyber defense. In *Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation*, GECCO Comp '14, pages 1261–1268, New York, NY, USA, 2014. ACM.

[169] David S Johnson, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part i, graph partitioning. *Operations research*, 37(6):865–892, 1989.

[170] David S Johnson, Cecilia R Aragon, Lyle A McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation; part ii, graph coloring and number partitioning. *Operations research*, 39(3):378–406, 1991.

[171] Donald R Jones and Mark A Beltramo. Solving partitioning problems with genetic algorithms. In *ICGA*, pages 442–449, 1991.

[172] N. Joukov, A. Traeger, R. Iyer, C. P. Wright, and E. Zadok. Operating System Profiling via Latency Analysis. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 89–102, Seattle, WA, November 2006. ACM SIGOPS.

[173] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, pages 237–285, 1996.

[174] Malvin H Kalos and Paula A Whitlock. *Monte carlo methods*. John Wiley & Sons, 2008.

[175] A Kapsalis, Vic J Rayward-Smith, and George D Smith. Solving the graphical steiner tree problem using genetic algorithms. *Journal of the Operational Research Society*, pages 397–406, 1993.

[176] Dervis Karaboga. An idea based on honey bee swarm for numerical optimization. Technical report, Technical report-tr06, Erciyes university, engineering faculty, computer engineering department, 2005.

[177] Dervis Karaboga and Bahriye Basturk. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization*, 39(3):459–471, 2007.

[178] Dervis Karaboga, Beyza Gorkemli, Celal Ozturk, and Nurhan Karaboga. A comprehensive survey: artificial bee colony (abc) algorithm and applications. *Artificial Intelligence Review*, 42(1):21–57, 2014.

[179] M. Karlsson, C. Karamanolis, and X. Zhu. Triage: Performance differentiation for storage systems using adaptive control. *ACM Trans. Storage*, 1(4), 2005.

[180] Kimberly Keeton, Dirk Beyer, Ernesto Brau, Arif Merchant, Cipriano Santos, and Alex Zhang. On the road to recovery: Restoring data after disasters. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys'06, pages 235–248, New York, NY, USA, 2006. ACM.

[181] James Kennedy. Particle swarm optimization. In *Encyclopedia of Machine Learning*, pages 760–766. Springer, 2010.

[182] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks*, pages 1942–1948, 1995.

[183] S. Kirkpatrick, C D. Gelatt, M. P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[184] Hiroaki Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems Journal*, 4:461–476, 1990.

[185] Michael Kolonko. Some new results on simulated annealing applied to the job shop scheduling problem. *European Journal of Operational Research*, 113(1):123–136, 1999.

[186] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[187] Philipp Kostuch. The university course timetabling problem with a three-phase approach. In *Practice and theory of automated timetabling V*, pages 109–125. Springer, 2005.

[188] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[189] Natalio Krasnogor and Jim Smith. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *Evolutionary Computation, IEEE Transactions on*, 9(5):474–488, 2005.

[190] Berthold Kröger. Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84(3):645–661, 1995.

[191] Klaus Krogmann, Michael Kuperberg, and Ralf Reussner. Using genetic search for reverse engineering of parametric behavior models for performance prediction. *Software Engineering, IEEE Transactions on*, 36(6):865–877, 2010.

[192] Sidharth Kumar, Avishek Saha, Venkatram Vishwanath, Philip Carns, John A. Schmidt, Giorgio Scorzelli, Hemanth Kolla, Ray Grout, Robert Latham, Robert Ross, Michael E. Papkafa, Jacqueline Chen, and Valerio Pascucci. Characterization and modeling of pidx parallel i/o for performance optimization. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 67:1–67:12, New York, NY, USA, 2013. ACM.

[193] Pedro Larrañaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, 1999.

[194] Huong Thanh Le and Luan Van Tran. Automatic feature selection for named entity recognition using genetic algorithm. In *Proceedings of the Fourth Symposium on Information and Communication Technology*, pages 81–87. ACM, 2013.

[195] Riccardo Leardi, R Boggia, and M Terrile. Genetic algorithms as a strategy for feature selection. *Journal of chemometrics*, 6(5):267–281, 1992.

[196] Chang-Yong Lee. Entropy-boltzmann selection in the genetic algorithms. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 33(1):138–149, 2003.

[197] Ernest Bruce Lee and Lawrence Markus. Foundations of optimal control theory. Technical report, DTIC Document, 1967.

[198] H. D. Lee, Y. J. Nam, K. J. Jung, S. G. Jung, and C. Park. Regulating I/O performance of shared storage with a control theoretical approach. In *NASA/IEEE Conference on Mass Storage Systems and Technologies (MSST)*. IEEE Society Press, 2004.

[199] David Levine. Genetic algorithms: A practitioner's view. *INFORMS Journal on computing*, 9(3):256–259, 1997.

[200] Rhydian Lewis, Ben Paechter, and Olivia Rossi-Doria. *Metaheuristics for university course timetabling*. Springer, 2007.

[201] Z. Li, M. Chen, A. Mukker, and E. Zadok. On the Trade-Offs among Performance, Energy, and Endurance in a Versatile Hybrid Drive. *ACM Transactions on Storage (TOS)*, 11(3), July 2015.

[202] Z. Li, R. Grosu, K. Muppalla, S. A. Smolka, S. D. Stoller, and E. Zadok. Model discovery for energy-aware computing systems: An experimental evaluation. In *Proceedings of the 1st Workshop on Energy Consumption and Reliability of Storage Systems (ERSS'11)*, Orlando, FL, July 2011.

[203] Z. Li, R. Grosu, P. Sehgal, S. A. Smolka, S. D. Stoller, and E. Zadok. On the Energy Consumption and Performance of Systems Software. In *Proceedings of the 4th Israeli Experimental Systems Conference (ACM SYSTOR '11)*, Haifa, Israel, May/June 2011. ACM.

[204] Z. Li, A. Mukker, and E. Zadok. On the Importance of Evaluating Storage Systems' $Costs. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'14, 2014.

[205] Jens Lienig and James P Cohoon. Genetic algorithms applied to the physical design of vlsi circuits: A survey. In *Parallel Problem Solving from NaturePPSN IV*, pages 839–848. Springer, 1996.

[206] H. C. Lim, S. Babu, and J. S. Chase. Automated control for elastic storage. In *Proceeding of the 7th International Conference on Autonomic computing*, ICAC '10, pages 1–10. ACM, 2010.

[207] Cláudio F Lima and Fernando G Lobo. Parameter-less optimization with the extended compact genetic algorithm and iterated local search. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 1328–1339. Springer, 2004.

[208] Xue Liu, Lui Sha, Yixin Diao, Steven Froehlich, Joseph L. Hellerstein, and Sujay Parekh. Online response time optimization of apache web server. In *Proceedings of the 11th International Conference on Quality of Service*, IWQoS'03, pages 461–478, Berlin, Heidelberg, 2003. Springer-Verlag.

[209] Zhen Liu, Mark S. Squillante, and Joel L. Wolf. On maximizing service-level-agreement profits. In *Proceedings of the 3rd ACM Conference on Electronic Commerce*, EC '01, pages 213–223, New York, NY, USA, 2001. ACM.

[210] Fernando G Lobo and David E Goldberg. The parameter-less genetic algorithm in practice. *Information Sciences*, 167(1):217–232, 2004.

[211] Fernando G Lobo and CláudioF Lima. A review of adaptive population sizing schemes in genetic algorithms. In *Proceedings of the 7th annual workshop on Genetic and evolutionary computation*, pages 228–234. ACM, 2005.

[212] Christoffer Loffler, Christopher Mutschler, and Michael Philippsen. Evolutionary algorithms that use runtime migration of detector processes to reduce latency in event-based systems. In *Adaptive Hardware and Systems (AHS), 2013 NASA/ESA Conference on*, pages 31–38. IEEE, 2013.

[213] Y. Lu, A. Saxena, and T. F. Abdelzaher. Differentiated caching services; a control-theoretical approach. In *21st International Conference on Distributed Computing Systems*, pages 615–624, 2001.

[214] Brian Lucas, Errin W. Fulp, David J. John, and Daniel Cañas. An initial framework for evolving computer configurations as a moving target defense. In *Proceedings of the 9th Annual Cyber and Information Security Research Conference*, CISR '14, pages 69–72, New York, NY, USA, 2014. ACM.

[215] S. Luke. *Essentials of Metaheuristics*. Lulu, second edition, 2013.

[216] MADbench. http://crd.lbl.gov/departments/computational-science/c3/c3-research/madbench2/.

[217] Samir W Mahfoud. An analysis of boltzmann tournament selection. *IlliGAL Report*, 91007, 1994.

[218] Samir W Mahfoud. Niching methods for genetic algorithms. *Urbana*, 51(95001), 1995.

[219] Samir W Mahfoud and David E Goldberg. Parallel recombinative simulated annealing: a genetic algorithm. *Parallel computing*, 21(1):1–28, 1995.

[220] Harpal Maini, Kishan Mehrotra, Chilukuri Mohan, and Sanjay Ranka. Genetic algorithms for graph partitioning and incremental graph partitioning. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*, pages 449–457. IEEE Computer Society Press, 1994.

[221] AH Mantawy, Youssef L Abdel-Magid, and Shokri Z Selim. A simulated annealing algorithm for unit commitment. *Power Systems, IEEE Transactions on*, 13(1):197–204, 1998.

[222] Rafael Martí, Manuel Laguna, and Fred Glover. Principles of scatter search. *European Journal of Operational Research*, 169(2):359–372, 2006.

[223] Olivier Martin, Steve W Otto, and Edward W Felten. Large-step markov chains for the tsp incorporating local search heuristics. *Operations Research Letters*, 11(4):219–224, 1992.

[224] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.*, 31:238–251, October 1997.

[225] Michael de la Maza and Bruce Tidor. An analysis of selection procedures with particular attention paid to proportional and boltzmann selection. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 124–131, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[226] Pinaki Mazumder and Elizabeth M. Rudnick, editors. *Genetic Algorithms for VLSI Design, Layout &Amp; Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[227] Michael D McKay, Richard J Beckman, and William J Conover. Comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 21(2):239–245, 1979.

[228] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[229] Markov Decision Process. https://en.wikipedia.org/wiki/Markov_decision_process.

[230] Daniel A. Menascé, Virgilio A. F. Almeida, Rodrigo Fonseca, and Marco A. Mendes. Business-oriented resource management policies for e-commerce servers. *Perform. Eval.*, 42(2-3):223–239, October 2000.

[231] Peter Merz. Memetic algorithms for combinatorial optimization problems: Fitness landscapes and effective search strategies, 2001.

[232] Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

[233] Lee Meyer and Xin Feng. A fuzzy stop criterion for genetic algorithms using performance estimation. In *Fuzzy Systems, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the Third IEEE Conference on*, pages 1990–1995. IEEE, 1994.

[234] Zbigniew Michalewicz. *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media, 2013.

[235] Brad L Miller and David E Goldberg. Genetic algorithms, selection schemes, and the varying effects of noise. *Evolutionary Computation*, 4(2):113–131, 1996.

[236] Nikita Mishra, Huazhe Zhang, John D. Lafferty, and Henry Hoffmann. A probabilistic graphical model-based approach for minimizing energy under performance constraints. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 267–281, New York, NY, USA, 2015. ACM.

[237] Debasis Mitra, Fabio Romeo, and Alberto Sangiovanni-Vincentelli. Convergence and finite-time behavior of simulated annealing. In *Decision and Control, 1985 24th IEEE Conference on*, volume 24, pages 761–767. IEEE, 1985.

[238] David J Montana and Lawrence Davis. Training feedforward neural networks using genetic algorithms. In *IJCAI*, volume 89, pages 762–767, 1989.

[239] Pablo Moscato et al. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Caltech concurrent computation program, C3P Report*, 826:1989, 1989.

[240] Heinz Muhlenbein. Evolution in time and space-the parallel genetic algorithm. In *Foundations of genetic algorithms*. Citeseer, 1991.

[241] Heinz Mühlenbein. Parallel genetic algorithms, population genetics, and combinatorial optimization. In *Workshop on Evolutionary Models and Strategies, Workshop on Parallel Processing: Logic, Organization, and Technology: Parallelism, Learning, Evolution*, WOPPLOT '89, pages 398–406, London, UK, UK, 1991. Springer-Verlag.

[242] Heinz Mühlenbein. How genetic algorithms really work: Mutation and hillclimbing. In *PPSN*, volume 92, pages 15–25, 1992.

[243] Heinz Mühlenbein and Th Mahnig. Mathematical analysis of evolutionary algorithms. In *Essays and Surveys in Metaheuristics*, pages 525–556. Springer, 2002.

[244] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[245] Mutual Information. https://en.wikipedia.org/wiki/Mutual_information.

[246] Sunil Nakrani and Craig Tovey. On honey bees and dynamic server allocation in internet hosting centers. *Adaptive Behavior*, 12(3-4):223–240, 2004.

[247] M Nandhini and S Kanmani. A survey of simulated annealing methodology for university course timetabling. *International Journal of Recent Trends in Engineering*, 1(2):255–257, 2009.

[248] NAS Parallel Benchmarks. https://www.nas.nasa.gov/publications/npb.html.

[249] John A Nelder and Roger Mead. A simplex method for function minimization. *The computer journal*, 7(4):308–313, 1965.

[250] M. E. J. Newman. Power laws, pareto distributions and zipfs law. *Contemporary Physics*, 2005.

[251] Yaghout Nourani and Bjarne Andresen. A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41):8373, 1998.

[252] Il-Seok Oh, Jin-Seon Lee, and Byung-Ro Moon. Hybrid genetic algorithms for feature selection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(11):1424–1437, 2004.

[253] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the traveling salesman problem. In *Proceedings of the Second International Conference on Genetic Algorithms on Genetic Algorithms and Their Application*, pages 224–230, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[254] Yew-Soon Ong, Meng-Hiot Lim, Ning Zhu, and Kok-Wai Wong. Classification of adaptive memetic algorithms: a comparative study. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 36(1):141–152, 2006.

[255] Isao Ono and Shigenobu Kobayashi. A real coded genetic algorithm for function optimization using unimodal normal distributed crossover. In *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 246–253. Morgan Kaufmann, 1997.

[256] Heikki Orsila, Tero Kangas, Erno Salminen, and Timo D Hamalainen. Parameterizing simulated annealing for distributing task graphs on multiprocessor socs. In *System-on-Chip, 2006. International Symposium on*, pages 1–4. IEEE, 2006.

[257] Ibrahim Hassan Osman. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. *Annuals of operations research*, 41(4):421–451, 1993.

[258] Takayuki Osogami and Toshinari Itoko. Finding probably better system configurations quickly. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 264–275, New York, NY, USA, 2006. ACM.

[259] Takayuki Osogami and Sei Kato. Optimizing system configurations quickly by guessing at the performance. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 145–156, New York, NY, USA, 2007. ACM.

[260] Bijaya Ketan Panigrahi, Yuhui Shi, and Meng-Hiot Lim. *Handbook of swarm intelligence: concepts, principles and applications*, volume 8. Springer Science & Business Media, 2011.

[261] Moon-Won Park and Yeong-Dae Kim. A systematic procedure for setting parameters in simulated annealing algorithms. *Computers & Operations Research*, 25(3):207–217, 1998.

[262] Particle Swarm Optimization. https://en.wikipedia.org/wiki/Particle_swarm_optimization.

[263] Kevin M Passino. Biomimicry of bacterial foraging for distributed optimization and control. *Control Systems, IEEE*, 22(3):52–67, 2002.

[264] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 285–297, New York, NY, USA, 2015. ACM.

[265] Martin Pelikan and Tz-Kai Lin. Parameter-less hierarchical boa. In *Genetic and Evolutionary Computation–GECCO 2004*, pages 24–35. Springer, 2004.

[266] Martin Pelikan and Fernando G Lobo. Parameter-less genetic algorithm: A worst-case time and space complexity analysis. In *GECCO*, page 370, 2000.

[267] Christian S. Perone. Pyevolve: A python open-source framework for genetic algorithms. *SIGEVOlution*, 4(1):12–20, November 2009.

[268] Chrisila B Pettey, Michael R Leuze, and John J Grefenstette. Parallel genetic algorithm. In *Genetic algorithms and their applications: proceedings of the second International Conference on Genetic Algorithms: July 28-31, 1987 at the Massachusetts Institute of Technology, Cambridge, MA*, 1987.

[269] F Pezzella, G Morganti, and G Ciaschetti. A genetic algorithm for the flexible job-shop scheduling problem. *Computers & Operations Research*, 35(10):3202–3212, 2008.

[270] DT Pham, A Ghanbarzadeh, E Koc, S Otri, S Rahim, and M Zaidi. The bees algorithm–a novel tool for complex optimisation. In *Intelligent Production Machines and Systems-2nd I* PROMS Virtual International Conference 3-14 July 2006*, page 454. Elsevier, 2011.

[271] Hasan Pirkul and Erik Rolland. New heuristic solution procedures for the uniform graph partitioning problem: extensions and evaluation. *Computers & Operations Research*, 21(8):895–907, 1994.

[272] VP Plagianakos, DK Tasoulis, and MN Vrahatis. A review of major application areas of differential evolution. In *Advances in differential evolution*, pages 197–238. Springer, 2008.

[273] Riccardo Poli. Analysis of the publications on the applications of particle swarm optimisation. *Journal of Artificial Evolution and Applications*, 2008:3, 2008.

[274] Kenneth Price, Rainer M Storn, and Jouni A Lampinen. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[275] A Kai Qin, Vicky Ling Huang, and Ponnuthurai N Suganthan. Differential evolution algorithm with strategy adaptation for global numerical optimization. *Evolutionary Computation, IEEE Transactions on*, 13(2):398–417, 2009.

[276] Q-Learning. https://en.wikipedia.org/wiki/Q-learning.

[277] Mukund Raghavachari, Darrell Reimer, and Robert D. Johnson. The deployer's problem: Configuring application servers for performance and reliability. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 484–489, Washington, DC, USA, 2003. IEEE Computer Society.

[278] Outi Räihä, Erkki Mäkinen, and Timo Poranen. Using simulated annealing for producing software architectures. In *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*, pages 2131–2136. ACM, 2009.

[279] Sanguthevar Rajasekaran. On the convergence time of simulated annealing, 1990.

[280] D Janaki Ram, TH Sreenivas, and K Ganapathy Subramaniam. Parallel simulated annealing algorithms. *Journal of parallel and distributed computing*, 37(2):207–212, 1996.

[281] Andres J. Ramirez, David B. Knoester, Betty H.C. Cheng, and Philip K. McKinley. Applying genetic algorithms to decision making in autonomic computing systems. In *Proceedings of the 6th International Conference on Autonomic Computing*, ICAC '09, pages 97–106, New York, NY, USA, 2009. ACM.

[282] Connie Loggia Ramsey and John J Grefenstette. Case-based initialization of genetic algorithms. In *ICGA*, pages 84–91. Citeseer, 1993.

[283] Jackie Rees and Gary J Koehler. An investigation of ga performance results for different cardinality alphabets. In *Evolutionary Algorithms*, pages 191–206. Springer, 1999.

[284] Colin Reeves. Hybrid genetic algorithms for bin-packing and related problems. *Annals of Operations Research*, 63(3):371–396, 1996.

[285] Colin R Reeves. Using genetic algorithms with small populations. In *ICGA*, volume 5, pages 90–92. Citeseer, 1993.

[286] Colin R Reeves. Genetic algorithms for the operations researcher. *INFORMS journal on computing*, 9(3):231–250, 1997.

[287] Colin R. Reeves and Jonathan E. Rowe. *Genetic Algorithms: Principles and Perspectives: A Guide to GA Theory*. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[288] Colin R Reeves and Christine C Wright. Genetic algorithms and the design of experiments. In *Evolutionary Algorithms*, pages 207–226. Springer, 1999.

[289] Reinforcement Learning. https://en.wikipedia.org/wiki/Reinforcement_learning.

[290] H. Reiser. ReiserFS v.3 Whitepaper. http://web.archive.org/web/20031015041320/http://namesys.com/.

[291] Bernd Reisleben and Peter Merz. A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 616–621. IEEE, 1996.

[292] Christian P Robert and George Casella. *Monte Carlo statistical methods*. Springer Texts in Statistics. Springer, New York, 1999.

[293] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):9, 2013.

[294] Fabio Romeo and Alberto Sangiovanni-Vincentelli. A theoretical framework for simulated annealing. *Algorithmica*, 6(1-6):302–345, 1991.

[295] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[296] Reuven Y Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997.

[297] Günter Rudolph. Convergence analysis of canonical genetic algorithms. *Neural Networks, IEEE Transactions on*, 5(1):96–101, 1994.

[298] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[299] Anooshiravan Saboori, Guofei Jiang, and Haifeng Chen. Autotuning configurations in distributed systems for performance improvements using evolutionary strategies. In *Proceedings of the 2008 The 28th International Conference on Distributed Computing Systems*, ICDCS '08, pages 769–776, Washington, DC, USA, 2008. IEEE Computer Society.

[300] Martín Safe, Jessica Carballido, Ignacio Ponzoni, and Nélida Brignole. On stopping criteria for genetic algorithms. In *Advances in Artificial Intelligence–SBIA 2004*, pages 405–413. Springer, 2004.

[301] Sadiq M Sait, Mahmood R Minhas, Junhaid Khan, et al. Performance and low power driven vlsi standard cell placement using tabu search. In *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*, volume 1, pages 372–377. IEEE, 2002.

[302] Dragan A Savic and Godfrey A Walters. Genetic algorithms for least-cost design of water distribution networks. *Journal of water resources planning and management*, 123(2):67–77, 1997.

[303] J David Schaffer. Some effects of selection procedures on hyperplane sampling by genetic algorithms. *Genetic algorithms and simulated annealing*, 1(1):23–29, 1987.

[304] J David Schaffer, Richard A Caruana, Larry J Eshelman, and Rajarshi Das. A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the third international conference on Genetic algorithms*, pages 51–60. Morgan Kaufmann Publishers Inc., 1989.

[305] J David Schaffer, David Whitley, and Larry J Eshelman. Combinations of genetic algorithms and neural networks: A survey of the state of the art. In *Combinations of Genetic Algorithms and Neural Networks, 1992., COGANN-92. International Workshop on*, pages 1–37. IEEE, 1992.

[306] Dirk Schlierkamp-Voosen and Heinz Mühlenbein. Strategy adaptation by competing subpopulations. In *Parallel Problem Solving from NaturePPSN III*, pages 199–208. Springer, 1994.

[307] Dirk Schlierkamp-Voosen and Heinz Mühlenbein. Adaptation of population sizes by competing subpopulations. In *In Proceedings of the 1996 IEEE Conference on Evolutionary Computation, Piscataway*. Citeseer, 1996.

[308] F. Schmuck and R. Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the First USENIX Conference on File and Storage Technologies (FAST '02)*, pages 231–244, Monterey, CA, January 2002. USENIX Association.

[309] Johannes J Schneider and Markus Puchta. Investigation of acceptance simulated annealing–a simplified approach to adaptive cooling schedules. *Physica A: Statistical Mechanics and its Applications*, 389(24):5822–5831, 2010.

[310] Nicol N Schraudolph and Richard K Belew. Dynamic parameter encoding for genetic algorithms. *Machine learning*, 9(1):9–21, 1992.

[311] Alan C Schultz and John J Grefenstette. Improving tactical plans with genetic algorithms. In *Tools for Artificial Intelligence, 1990., Proceedings of the 2nd International IEEE Conference on*, pages 328–334. IEEE, 1990.

[312] Hans-Paul Schwefel. *Numerische optimierung von computer-modellen mittels der evolutionsstrategie*, volume 1. Birkhäuser, Basel Switzerland, 1977.

[313] Hans-Paul Schwefel. *Numerical optimization of computer models*. John Wiley & Sons, Inc., 1981.

[314] Hans-Paul Schwefel. Natural evolution and collective optimum-seeking. In Achim Sydow, editor, *Computational Systems Analysis*, pages 5–14. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 1992.

[315] Hans-Paul Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[316] scikit-learn: Machine Learning in Python. http://scikit-learn.org/stable/.

[317] Carl Sechen. *VLSI placement and global routing using simulated annealing*, volume 54. Springer Science & Business Media, 2012.

[318] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating Performance and Energy in File System Server Workloads. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[319] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3), September 2010.

[320] Randall S Sexton, Bahram Alidaee, Robert E Dorsey, and John D Johnson. Global optimization for artificial neural networks: A tabu search application. *European Journal of Operational Research*, 106(2):570–584, 1998.

[321] SGI. XFS Filesystem Structure. http://oss.sgi.com/projects/xfs/papers/xfs_filesystem_structure.pdf.

[322] Yuhui Shi. Particle swarm optimization. *IEEE Connections*, 2(1):8–13, 2004.

[323] Wojciech Siedlecki and Jack Sklansky. A note on genetic algorithms for large-scale feature selection. *Pattern recognition letters*, 10(5):335–347, 1989.

[324] Abhishek Sinha and David E Goldberg. A survey of hybrid genetic and evolutionary algorithms. *IlliGAL report 2003004*, 2003.

[325] SN Sivanandam and SN Deepa. *Introduction to genetic algorithms*. Springer Science & Business Media, 2007.

[326] Derek Smith. Bin packing with adaptive search. In *Proceedings of the 1st International Conference on Genetic Algorithms*, pages 202–207. L. Erlbaum Associates Inc., 1985.

[327] Jim Smith and Terence C Fogarty. Recombination strategy adaptation via evolution of gene linkage. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 826–831. IEEE, 1996.

[328] Robert E Smith. Adaptively resizing populations: An algorithm and analysis. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 653. Morgan Kaufmann Publishers Inc., 1993.

[329] Robert E Smith and Ellen Smuda. Adaptively resizing populations: Algorithm, analysis, and first results. *Complex Systems*, 9(1):47–72, 1995.

[330] SPEC HPG. https://www.spec.org/hpg/.

[331] SPEC SFS® 2014. https://www.spec.org/sfs2014/.

[332] Mandavilli Srinivas and Lalit M Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[333] T Starkweather, S Mcdaniel, D Whitley, K Mathias, D Whitley, et al. A comparison of genetic sequencing operators. In *Proceedings of the fourth International Conference on Genetic Algorithms*, 1991.

[334] Rainer Storn. On the usage of differential evolution for function optimization. In *Fuzzy Information Processing Society, 1996. NAFIPS., 1996 Biennial Conference of the North American*, pages 519–523. IEEE, 1996.

[335] Rainer Storn and Kenneth Price. Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces. *Journal of global optimization*, 11(4):341–359, 1997.

[336] Rainer Storn and Kenneth V Price. Minimizing the real functions of the icec'96 contest by differential evolution. In *International Conference on Evolutionary Computation*, pages 842–844, 1996.

[337] John D. Strunk, Eno Thereska, Christos Faloutsos, and Gregory R. Ganger. Using utility to provision storage systems. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 21:1–21:16, Berkeley, CA, USA, 2008. USENIX Association.

[338] Jung Y Suh and Dirk Van Gucht. *Distributed genetic algorithms*. Computer Science Department, Indiana Univ., 1987.

[339] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. www.sun.com/software/solaris/ds/zfs.jsp.

[340] Richard S Sutton and Andrew G Barto. *Introduction to Reinforcement Learning*, volume 135. MIT Press Cambridge, 1998.

[341] Gilbert Sywerda. Uniform crossover in genetic algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 2–9, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[342] Harold Szu and Ralph Hartley. Fast simulated annealing. *Physics letters A*, 122(3):157–162, 1987.

[343] Tabu Search. https://en.wikipedia.org/wiki/Tabu_search.

[344] Eric D Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location science*, 3(2):87–105, 1995.

[345] E-G Talbi and Pierre Bessiere. A parallel genetic algorithm for the graph partitioning problem. In *Proceedings of the 5th international conference on Supercomputing*, pages 312–320. ACM, 1991.

[346] David M Tate and Alice E Smith. A genetic approach to the quadratic assignment problem. *Computers & Operations Research*, 22(1):73–83, 1995.

[347] Sam R. Thangiah, Rajini Vinayagamoorty, and Ananda V. Gubbi. Vehicle routing and time deadlines using genetic and local algorithms. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 506–515, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[348] R. Thonangi, V. Thummala, and S. Babu. Finding good configurations in high-dimensional spaces: Doing more with less. In *Modeling, Analysis and Simulation of Computers and Telecommunication Systems, 2008. MASCOTS 2008. IEEE International Symposium on*, pages 1–10, Sept 2008.

[349] Mustafa M Tikir, Laura Carrington, Erich Strohmaier, and Allan Snavely. A genetic algorithms approach to modeling the performance of memory-bound computations. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, pages 47:1–47:12, New York, NY, USA, 2007. ACM.

[350] Jon Timmis, Mark Neal, and John Hunt. An artificial immune system for data analysis. *Biosystems*, 55(1):143–150, 2000.

[351] Paolo Toth and Daniele Vigo. The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on computing*, 15(4):333–346, 2003.

[352] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.

[353] Eric Triki, Yann Collette, and Patrick Siarry. A theoretical study on the behavior of simulated annealing leading to a new cooling schedule. *European Journal of Operational Research*, 166(1):77–92, 2005.

[354] Shigeyoshi Tsutsui. Multi-parent recombination in genetic algorithms with search space boundary extension by mirroring. In *Parallel Problem Solving from NaturePPSN V*, pages 428–437. Springer, 1998.

[355] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. Multi-parent recombination with simplex crossover in real coded genetic algorithms. In *Proceedings of the genetic and evolutionary computation conference*, volume 1, pages 657–664, 1999.

[356] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html.

[357] VA Valkó. Self-calibrating evolutionary algorithms: Adaptive population size. *Master's thesis, Free University Amsterdam*, 2003.

[358] Peter J Van Laarhoven and Emile H Aarts. *Simulated annealing: theory and applications*, volume 37. Springer Science & Business Media, 1987.

[359] Peter JM Van Laarhoven, Emile HL Aarts, and Jan Karel Lenstra. Job shop scheduling by simulated annealing. *Operations research*, 40(1):113–125, 1992.

[360] James M Varanelli. *On the acceleration of simulated annealing*. PhD thesis, Citeseer, 1996.

[361] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):35:1–35:33, July 2013.

[362] Jakob Vesterstrøm and Rene Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Evolutionary Computation, 2004. CEC2004. Congress on*, volume 2, pages 1980–1987. IEEE, 2004.

[363] VMMark. www.vmware.com/go/vmmark.

[364] Hans-Michael Voigt and Heinz Muhlenbein. Gene pool recombination and utilization of covariances for the breeder genetic algorithm. In *Evolutionary Computation, 1995., IEEE International Conference on*, volume 1, page 172. IEEE, 1995.

[365] Gregor Von Laszewski. Intelligent structural operators for the k-way graph partitioning problem. *Northeast Parallel Architecture Center*, 1991.

[366] Michael D Vose. Modeling simple genetic algorithms. *Foundations of genetic algorithms*, 2:63–73, 1993.

[367] Michael D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, Cambridge, MA, USA, 1998.

[368] Stefan Voß. Steiner's problem in graphs: heuristic methods. *Discrete Applied Mathematics*, 40(1):45–72, 1992.

[369] Stefan Voß. Meta-heuristics: The state of the art. In *Local Search for Planning and Scheduling*, pages 1–23. Springer, 2001.

[370] Edward Walker. Benchmarking amazon ec2 for hig-performance scientific computing. *; login:: the magazine of USENIX & SAGE*, 33(5):18–23, 2008.

[371] S Walton, O Hassan, K Morgan, and MR Brown. Modified cuckoo search: a new gradient free optimisation algorithm. *Chaos, Solitons & Fractals*, 44(9):710–718, 2011.

[372] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, et al. Bigdatabench: A big data benchmark suite from internet services. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 488–499. IEEE, 2014.

[373] X. Wang, M. Chen, and X. Fu. MIMO power control for high-density servers in an enclosure. *IEEE Trans. Parallel Distrib. Syst.*, 21:1412–1426, 2010.

[374] X. Wang and Y. Wang. Coordinating power control and performance management for virtualized server clusters. *IEEE Trans. Parallel Distrib. Syst.*, 22:245–259, February 2011.

[375] Y. Wang, X. Wang, M. Chen, and X. Zhu. Power-efficient response time guarantees for virtualized enterprise servers. In *Proceedings of the 2008 Real-Time Systems Symposium*, pages 303–312. IEEE Computer Society, 2008.

[376] Robin Wardlaw and Mohd Sharif. Evaluation of genetic algorithms for optimal reservoir system operation. *Journal of water resources planning and management*, 125(1):25–33, 1999.

[377] Steve R White. Concepts of scale in simulated annealing. *The Physics of VLSI*, 122(1):261–270, 1984.

[378] Darrell Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4(2):65–85, 1994.

[379] Darrell Whitley, Keith Mathias, and Patrick Fitzhorn. Delta coding: An iterative search strategy for genetic algorithms. In *ICGA*, volume 91, pages 77–84. Citeseer, 1991.

[380] Darrell Whitley, Soraya Rana, and Robert B Heckendorn. The island model genetic algorithm: On separability, population size and convergence. *Journal of Computing and Information Technology*, 7:33–48, 1999.

[381] Darrell Whitley, Timothy Starkweather, and Christopher Bogart. Genetic algorithms and neural networks: Optimizing connections and connectivity. *Parallel computing*, 14(3):347–361, 1990.

[382] Allele. https://en.wikipedia.org/wiki/Allele.

[383] Crossover. https://en.wikipedia.org/wiki/Chromosomal_crossover.

[384] Evolution Strategies. https://en.wikipedia.org/wiki/Evolution_strategy.

[385] Evolutionary Algorithms. https://en.wikipedia.org/wiki/Evolutionary_algorithm.

[386] Evolutionary Programming. https://en.wikipedia.org/wiki/Evolutionary_programming.

[387] Fitness. https://en.wikipedia.org/wiki/Fitness(biology).

[388] Gene. https://en.wikipedia.org/wiki/Gene.

[389] Genetic Programming. https://en.wikipedia.org/wiki/Genetic_programming.

[390] Genetics. https://en.wikipedia.org/wiki/Genetics.

[391] Genome. https://en.wikipedia.org/wiki/Genome.

[392] Genotype. https://en.wikipedia.org/wiki/Genotype.

[393] Gradient descent. https://en.wikipedia.org/wiki/Gradient_descent.

[394] Heuristic. http://en.wikipedia.org/wiki/Heuristic_(computer_science).

[395] Latin Hypercube Sampling. https://en.wikipedia.org/wiki/Latin_hypercube_sampling.

[396] Metaheuristic. http://en.wikipedia.org/wiki/Metaheuristic.

[397] Mutation. https://en.wikipedia.org/wiki/Mutation.

[398] Natural Selection. https://en.wikipedia.org/wiki/Natural_selection.

[399] Nonlinear system. https://en.wikipedia.org/wiki/Nonlinear_system.

[400] Phenotype. https://en.wikipedia.org/wiki/Phenotype.

[401] Phenotypic trait. https://en.wikipedia.org/wiki/Phenotypic_trait.

[402] wiki-schema. http://en.wikipedia.org/wiki/Holland's_schema_theorem.

[403] Simulated annealing. http://en.wikipedia.org/wiki/Simulated_annealing.

[404] Michael L. Winterrose and Kevin M. Carter. Strategic evolution of adversaries against temporal platform diversity active cyber defenses. *CoRR*, abs/1408.0023, 2014.

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

[406] DF Wong, Hon Wai Leong, and HW Liu. *Simulated annealing for VLSI design*, volume 42. Springer Science & Business Media, 2012.

[407] Anthony Wren and David O Wren. A genetic algorithm for public transport driver scheduling. *Computers & Operations Research*, 22(1):101–110, 1995.

[408] C. P. Wright, J. Dave, and E. Zadok. Cryptographic File Systems Performance: What You Don't Know Can Hurt You. In *Proceedings of the Second IEEE International Security In Storage Workshop (SISW 2003)*, pages 47–61, Washington, DC, October 2003. IEEE Computer Society.

[409] C. P. Wright, N. Joukov, D. Kulkarni, Y. Miretskiy, and E. Zadok. Auto-pilot: A Platform for System Software Benchmarking. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 175–187, Anaheim, CA, April 2005. USENIX Association.

[410] Bowei Xi, Zhen Liu, Mukund Raghavachari, Cathy H. Xia, and Li Zhang. A smart hill-climbing algorithm for application server configuration. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 287–296, New York, NY, USA, 2004. ACM.

[411] Ji Xue, Feng Yan, A. Riska, and E. Smirni. Proactive management of systems via hybrid analytic techniques. In *Cloud and Autonomic Computing (ICCAC), 2015 International Conference on*, pages 137–148, Sept 2015.

[412] Ji Xue, Feng Yan, Alma Riska, and Evgenia Smirni. Storage workload isolation via tier warming: How models can help. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 1–11, Philadelphia, PA, June 2014. USENIX Association.

[413] X. Yang. Metaheuristic Optimization. *Scholarpedia*, 6(8):11472, 2011. revision 91488.

[414] Xin-She Yang. Engineering optimizations via nature-inspired virtual bee algorithms. In *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach*, pages 317–323. Springer, 2005.

[415] Xin-She Yang. Firefly algorithms for multimodal optimization. In *Stochastic algorithms: foundations and applications*, pages 169–178. Springer, 2009.

[416] Xin-She Yang. *Engineering Optimization: An Introduction with Metaheuristic Applications*. John Wiley & Sons, 2010.

[417] Xin-She Yang. Firefly algorithm, levy flights and global optimization. In *Research and development in intelligent systems XXVI*, pages 209–218. Springer, 2010.

[418] Xin-She Yang. Firefly algorithm, stochastic test functions and design optimisation. *International Journal of Bio-Inspired Computation*, 2(2):78–84, 2010.

[419] Xin-She Yang and Suash Deb. Cuckoo search via lévy flights. In *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on*, pages 210–214. IEEE, 2009.

[420] Xin-She Yang and Suash Deb. Engineering optimisation by cuckoo search. *International Journal of Mathematical Modelling and Numerical Optimisation*, 1(4):330–343, 2010.

[421] Tao Ye and Shivkumar Kalyanaraman. A recursive random search algorithm for large-scale network parameter configuration. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, pages 196–205, New York, NY, USA, 2003. ACM.

[422] Tao Ye, Hema T. Kaur, Shivkumar Kalyanaraman, and Murat Yuksel. Large-scale network parameter configuration using an on-line simulation framework. *IEEE/ACM Trans. Netw.*, 16(4):777–790, August 2008.

[423] ZFS for Linux, January 2016. www.zfs-fuse.net.