Genetic Algorithm Portfolios

Alex S. Fukunaga
Computer Science Department
University of California, Los Angeles
fukunaga@cs.ucla.edu

Abstract-

Comparative studies of sets of control parameter values are commonly performed when tuning an evolutionary algorithm for a class of problem instances. The standard approach is to identify the most useful set of control parameter settings for a domain. In this paper, we propose an alternative *anytime algorithm portfolio* technique in which computational resources are allocated among multiple sets of control parameter value settings. We show a method of optimizing such portfolios by applying a bootstrap sampling approach to a database of individual algorithm performance on instances from a problem distribution. Experiments with a genetic algorithms applied to the traveling salesperson domain show that the portfolio approach can yield better performance on a distribution of problem instances than the standard approach of trying to identify the single best configuration for the problem class.

1 Introduction

It is well known that control parameters such as population size and mutation rate have a significant impact on the performance of evolutionary algorithms. While recent theoretical work ([13]) has confirmed that it is not possible to find a set of control parameter values that is optimal in general for all problems to which an evolutionary algorithm can be applied, it is possible (and often necessary) to tune control parameters so that acceptable performance is obtained for some particular class of problems. Thus, an implicit goal in much of the previous empirical work on evolutionary algorithms is to identify useful control parameter values for some given class of problems.

A standard empirical methodology used in the field of evolutionary algorithms (as well as fields studying other heuristic search/optimization algorithms) is the following: for some problem class C, and evolutionary algorithm A, measure and report the performance of several parameterized instances of A on several instances of C. Common performance metrics used include the quality of the best solution found by the algorithm within a certain time/resource limit, or the time/resource required by the algorithm to find a solution with a given threshold quality. Usually, the researcher computes and reports the mean and variance of the measurements.

For example, a researcher proposing a new problem representation and/or genetic operator for the traveling salesperson problem (TSP) might proceed as follows: First, he implements an evolutionary algorithm which incorporates his new representation and genetic operators. He then selects a set of benchmark TSP instances and runs his algorithms with these instances as input. Because it is not known *a priori* which control parameter values are most useful, he would try a number of different sets of control parameter values in order to better understand the behavior of his algorithm and tune its performance. He might then compare his algorithm (using one or more sets of control parameter values) against previous algorithms by selecting a new set of benchmark problems and measuring the performance his algorithm on these test instances and comparing the results to those of other TSP-solver algorithms. If one or more parameterized configurations of his new algorithm significantly outperform the standard benchmarks, he would conclude that he may have discovered a promising, new approach.

An implicit assumption in the above methodology is that the goal of such an empirical study should be to evaluate the ***expected utility of individual candidate algorithms*** in comparison to the expected utility of other candidate algorithms for a given class of problems. In many cases, this is a useful experimental paradigm, advancing the state of algorithm research by yielding new "champion" algorithms with good average case performance for the problem class.

However, there is an alternative to this standard approach that has not been fully explored to date. Namely, we need not restrict ourselves to evaluating a single algorithm against other algorithms in order to determine which algorithm is the "best" algorithm for a given class of problem instances – instead, we can focus instead on the following problem: *Given a problem instance from a class of problems and a set of candidate algorithms which can be applied to the problem class, how best can we allocate computational resources among the algorithms in* *order to maximize the expected utility of our problem-solving effort?*

This paper proposes the use of ***anytime algorithm portfolios***, an approach for optimally allocating computational resources among candidate algorithms in order to maximize the expected utility of a problem solving episode. Specifically, we focus on the application of anytime algorithm portfolios to the control parameter value selection problem in evolutionary algorithms.

We begin by defining the framework of resource-bounded optimization, which is an appropriate model for many evolutionary algorithm applications which considers independent restarts of the evolutionary algorithm. We then define anytime algorithm portfolios in the resource-bounded optimization framework and describe a simple algorithm for automatically synthesizing anytime algorithm portfolios. We evaluate the utility of this approach on the application of a standard genetic algorithm to the traveling salesperson problem. We show that portfolios which allocate computational resources among several independent GA runs which use different sets of control parameter settings can outperform a set of independent restarts of the best "tuned" control parameter set found using the traditional aproach.

2 Anytime Portfolios for Resource-Bounded Optimization

2.1 Resource Bounded Optimization with Restarts

The framework of ***resource-bounded optimization*** is defined as follows: Let A be an optimization algorithm, and d be a problem instance (an objective function). Let T be a resource usage limit for A. In this paper, we assume that T is measured in a number of discrete "steps", or objective function evaluations – we assume that all objective function evaluations take approximately the same amount of time. Let $U(A, d, T)$, the utility of the algorithm A on d given time T, be the utility of the best solution found by the algorithm within the time bound. The task of resource-bounded optimization is to maximize U (i.e., obtain the best possible solution quality within a given time).

We assume that it does ***not*** matter when the maximal value of U is obtained within the time window $[0, T]$. This is a reasonable model of many real-world optimization scenarios, in which an optimization expert is given a deadline at which to present the best solution found. In this problem framework, the only thing that matters is the utility of the best solution found within the deadline. Metrics such as rate of improvement of the best-so-far solution, or convergence of the population are irrelevant with respect to how an algorithm's performance is evaluated.

If T is large enough, then it is possible to start a run of A, terminate it after t_k steps, start another independent run of A, and run for t_{k+1} steps. This process can be repeated n times, where $\sum_{i=1}^{n} t_i = T$. We call each of these independent run a ***restart***. A ***restart strategy*** determines $t_1, ...t_n$, and can be either ***static*** or ***dynamic***. Static restart strategies determine $t_1, ...t_n$ prior to running the algorithm. For example, a strategy which allocates resources equally among n restarts is a static strategy. Dynamic strategies, on the other hand, decide during runtime when a restart should take place. For example, we could repeat the following until the total number of steps taken is T: run A until a convergence criterion is met, then restart.

Earlier work [3] defined a ***static restart schedule*** as a set $S = \{t_1, t_2, ..t_n\}$, where $\sum_{i=1}^{n} t_i = T$. The execution model for the restart schedule for resource-bounded optimization is the following: Given an algorithm A and problem instance d, A is executed with d as the input for t_i steps, for each $i, 1 \leq i \leq n$. The best solution found among all of the restarts is stored and returned as the result of the restart schedule execution. It was shown in [3] that based on a performance database (see below), static restart schedules could be automatically synthesized which were competitive with dynamic restart schedules. The restart schedule $S = \{t, t, ..t\}$ (i.e., restart the algorithm every t steps), where t is chosen through trial and error, is frequently used in practice.

2.2 Anytime Algorithm Portfolios

The standard restart strategies described above assumes that the same algorithm is executed in each restart, with a different random seed or initial condition. By generalizing the framework to allow different algorithms to be executed in each restart, we have a framework for resource allocation among multiple candidate algorithms.

The intuition for applying multiple algorithms to a single problem instance is that if a class of problems has some structural diversity among its instances, then applying a diverse set of algorithms may be superior to relying on a single algorithm, particularly when the resource bound is large enough. As an example, consider the following scenario: Suppose we have two candiate algorithms A_1 and A_2, and suppose that we have a distribution of problem instances D, where half of the instances belong to subclass D_1 and the other half belong to subclass D_2. Let $U(A, t, d)$ denote the utility (expected performance) of algorithm A executed for time t on problem instance d. Suppose that $U(A_1, T, d) = 1.0$ for $d \in D_1$, and $U(A_1, T, d) = 0.0$ for $d \in D_2$. Also, let $U(A_2, T, d) = 1.0$ for $d \in D_2$ and $U(A_2, T, d) = 0.0$ for $d \in D_1$ (i.e., A_1 performs well on instances from subclass D_1 and poorly on instances from D_2, while the opposite is true for A_2.) Furthermore, assume that for both A_1 and A_2, for all $d \in D$, $U(A, T, d) = U(A, 2T, d)$ (i.e., running the algorithms twice as long does not improve performace since they reach the point of drastically decreasing marginal return after time T). Then, if the total resource bound available is $2T$, and test problem instances are drawn uniformly from D, then the best expected performance for any restart strategy using either A_1 or A_2 alone is 0.5. However, a strategy that dividies its time evenly between A_1 and A_2 has an expected performance of 1.0. This illustrates the potential gain due to diversification in a meta-level resource allocation strategy.

Let T be the resource bound for an instance of resource-bounded optimization. An anytime algorithm portfolio for this problem is a set $P = \{(A_1,t_1),(A_2,t_2),...(A_n,t_n)\}$, where $\sum_{i=1}^{n} t_i = T$, and $A_1, A_2, ...A_n$ are anytime algorithms that can be applied to the problem instance. For example, $A_1,..A_n$ can be sets of control parameter values for an evolutionary algorithm, where, e.g., A_1 could be the parameter set ($population = 100, MutationRate = 0.01$), and A_2 is the parameter set ($population = 200, MutationRate = 0.05$).

Given a portfolio $P = \{(A_1,t_1),...(A_n,t_n)\}$ and problem instance d, P is executed as follows: An independent run of A_i is executed with d as the input for t_i steps, for each i, $1 \leq i \leq n$. The best solution found among all of the restarts is stored and returned as the result of the portfolio execution. The portfolio is defined as a set, rather than a sequence, since the order in which the elements are executed does not matter (we assume that the restarts are independent, and that information from previous restarts is not used in subsequent restarts).

Let $U(A,d,t)$ denote the random variable which determines the utility when algorithm A is executed for time
t on problem instance d. Then $U(P,d,T)$, the utility of a portfolio, is also a random value related to those of the individual elements of the portfolio by $U(P,d,T) = max(U(A_1,d,t_1),U(A_2,d,t_2),...U(A_n,d,t_n))$. Thus, the goal of anytime algorithm portfolio design is to find the portfolio with highest expected utility $U(P,d,T)$.

The above definition of portfolio utility is still ambiguous, in that we have not yet defined how utility is measured. A natural utility metric is the best-so-far objective function value i.e., $U(P,d,T) = BestScore(P,d,T)$, the score of the best solution found by the portfolio running for time T on problem instance d. In addition, another natural metric to optimize is the variance of portfolio performance, $Var(P,d,T)$, which represents the *risk* of the portfolio. All other things being equal, a portfolio with smaller $Var(P,d,T)$ (less risk) is preferable to a portfolio with large variance (greater risk), especially in mission-critical real-time applications. We adopt a standard method from economics of combining expected return and risk on an investment and define utility as $U(P,d,T) = E[BestScore(P,d,T)] - RVar(P,d,T)$, where R is a constant of risk aversion of the user. If we choose to ignore portfolio variance, then $R = 0$, and portfolio utility is equal to the expected best-so-far score. Throughout the experiments in this paper, we used $R = 0.1$ in the utility computation.

In the discussion above, we have assumed that utility is positive, and that the goal is utility maximization. For optimization problems where the objective is to minimize an objective function (e.g., tour length in the traveling salesperson problem), we simply treat the objective function as a negative utility.

2.3 A Bootstrap Method for Meta-Level Optimization of Anytime Algorithm Portfolios

In order to maximize the expected value of U, $E[U(P,d,T)]$, we propose an approach which uses algorithm performance data collected in previous applications of the component algorithms of P to problems similar to d (i.e., problems drawn from the same class of problems) to determine the portfolio P. We assume that "similarity" has been defined elsewhere, and that classes of problems have been previously identified prior to application of the portfolio synthesis algorithm.

When A is executed on an instance d, we output the quality of the best-so-far solution at every q iterations in a *performance database* entry, $DB(A,d,runID) = \{(q,u_1),(2q,u_2),(3q,u_3),...(mq,u_m)\}$, where $runID$ is a tag which uniquely identifies the run (e.g., the random seed). By collecting a set of such entries, we collect a performance database which can serve as an empirical approximation of the distributions corresponding to the set of random variables $\mathcal{U}_{Ad} = \{U(A,d,q),U(A,d,2q),...U(A,d,mq)\}$. Data from runs on different problem instances (i.e., a set of sample problem instances which serves to approximate the underlying distribution of problem instances in the problem class) can be combined in order to generate approximations for $U(A,t)$.

Figure 1 shows a sample performance database based on a set of 3 independent runs of algorithm A_1 and A_2 on problem instances i_1 and i_2. From the database, we can compute, for example, that an approximation for the expected value of $U(A_1,i_1,30)$ is $(3+2+2)/3 = 2.33$, and $U(A_2,20) = (2+1+1+2+3+3)/6 = 2.0$.

The performance database provides the infrastructure necessary to automatically synthesize a static restart strategy that maximizes the expected utility $U(A,T)$, using a statistical bootstrap approach.

Synthesize-Portfolio (Figure 2) is a simple generate-and-test approach for finding good algorithm portfolios. Given a portfolio allocation unit size parameter k, where $T \bmod k = 0$, *synthesize-portfolio* searches the set of portfolios where each restart has a length which is an integral multiple of k. The size of this meta-level search

$$\begin{aligned}
D(A_1,i_1,0) &= \{(10,1),(20,2),(30,3),(40,3)\}\\
D(A_1,i_1,1) &= \{(10,1),(20,1),(30,2),(40,3)\}\\
D(A_1,i_1,2) &= \{(10,1),(20,1),(30,2),(40,2)\}\\
D(A_1,i_2,0) &= \{(10,2),(20,2),(30,2),(40,2)\}\\
D(A_1,i_2,1) &= \{(10,2),(20,3),(30,3),(40,3)\}\\
D(A_1,i_2,2) &= \{(10,1),(20,3),(30,3),(40,3)\}\\
D(A_2,i_1,0) &= \{(10,2),(20,2),(30,3),(40,3)\}\\
D(A_2,i_1,1) &= \{(10,1),(20,1),(30,1),(40,3)\}\\
D(A_2,i_1,2) &= \{(10,1),(20,1),(30,2),(40,3)\}\\
D(A_2,i_2,0) &= \{(10,2),(20,2),(30,3),(40,3)\}\\
D(A_2,i_2,1) &= \{(10,1),(20,3),(30,3),(40,4)\}\\
D(A_2,i_2,2) &= \{(10,2),(20,3),(30,3),(40,3)\}
\end{aligned}$$

Figure 1: Sample performance database, based on a set of 5 independent runs of algorithms A_1 and A_2 on problem instance $i_1$and 5 runs of A_1 and A_2 on instance i_2.

space explored by *synthesize-portfolio* grows exponentially as a function of T/k

GenerateNextPortfolio is a heuristic for generating candidate portfolios, and we currently use one of two simple algorithms: If it is feasible, we exhaustively enumerate the set of all portfolios with restart lengths which are multiples of k. For example, if A_0 and A_1 are the component algorithms and if $T = 5000, k = 2500$ (i.e., total resource allocation is 5000 objective function evaluations, and the algorithms can be restarted at multiples of 2500 iterations), the portfolios which are generated and evaluated are $P_0 = \{(A_0, 2500), (A_0, 2500)\}$, $P_1 = \{(A_0, 5000)\}$, $P_2 = \{(A_0, 2500), (A_1, 2500)\}$, and $P_3 = \{(A_1, 5000)\}$. This exhaustive search algorithm is most useful when optimizing portfolios restricted to using a single algorithm, i.e., standard restart strategies. However, for portfolios with multiple algorithms, exhaustive search quickly becomes intractable. Thus, if it is not feasible to enumerate the entire set of portfolios, *GenerateNextPortfolio* generates random portfolios where the algorithm and the length of the restart is selected using uniform sampling.

Each candidate portfolio is evaluated by estimating its expected utility via sampling (with replacement) from the performance database, i.e., we generate a bootstrap sampling distribution [2] and compute its mean and variance. The utility metric used is described in the section above (Section 2.2). It is important to note that evaluating a candidate portfolio with this bootstrap method is typically orders of magnitude less expensive than actually executing the portfolio (especially when the performance database is in memory). Thus, the meta-level search is able to evaluate thousands of candidate portfolio per second on a workstation, and its cost is negligible compared with the time required to generate the performance database.

```
Synthesize-portfolio(PerfDB,NumSamples,T,k)
bestPortfolio = {}
bestUtility = −∞
Repeat
  P=GenerateNextPortfolio(T,k)
  /* estimate expected utility of P using bootstrap sampling */
  SumUtility = −∞
  for i = 1 to NumSamples
    TrialUtility = −∞
    for each element (A_j,t_j) in P
      DBInst = ChooseRandomDBProbIndex
      DBSeed = ChooseRandomDBSeedIndex
      U_j = DBLookUp(PerfDB,A_j,DBInst,DBSeed,t_j)
      if U_j > TrialUtility
        TrialUtility = U_j
    end
    sumUtility = sumUtility + TrialUtility
  end

  U_S = sumUtility/NumSamples
  if U_S > bestUtility
    bestPortfolio = S
    bestUtility = U_S
Until some termination condition
Return bestPortfolio
```

Figure 2: *Synthesize-portfolio:* Returns a portfolio with highest expected utility.

3 Experiments and Results

We evaluated the anytime algorithm approach using a class of symmetric Traveling Salesperson Problem (TSP) instances.

3.1 Traveling salesperson domain and genetic algorithm

The TSP instances were generated by placing $N = 40$ cities on randomly selected (x, y) coordinates (where x and y are floating point values between 0 and 1) on a 1.0 by 1.0 rectangle. The cost of traveling between two cities c_i and c_j is the Euclidean distance between them, $d(c_i, c_j)$.

The objective is to find a tour π (a permutation of the cities) with minimal cost, $Cost_\pi = \sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) + d(c_{\pi(n)}, c_{\pi(1)})$

The problem representation used was a Gray-coded binary genome which was interpreted as follows: The ith allele (substring) was an integer between 1 and N, representing the ordering in the TSP tour for city i. Ties were broken in left to right order. For example, the genome $(3, 2, 1, 5, 3)$ for a 4-city TSP problem means that City1 is visited third, City2 second, City3 first, City4 fifth, City5 is fourth, and the tour completes by returning to City3. Although this is *not* a particularly good encoding of the TSP for a genetic algorithm (GA) (see, e.g. [10, 5] for encodings developed specifically to perform well on the TSP). However, our goal was to evaluate restart strategies (as opposed to finding good solutions for the TSP), and we used the TSP as a testbed because it is a well-known, convenient class of problems for which problem instances can be easily generated, encoded, and rapidly evaluated.

A steady-state genetic algorithm with uniform crossover [11] was used. The mutation operator was the standard bit flip operator. A rank-based selection method was used [12], where the individual selected had the rank: $SelRank = Population \frac{(Bias - \sqrt{Bias^2 - 4.0(Bias-1) \times RAND})}{2.0/(Bias-1)}$

i.e., the higher the bias, the more the likely it is that high ranked individuals are selected.[1] Likewise, when deleting individuals from the population, the index of the individual selected for deletion is ($Population - SelRank$) (the higher the bias, the less likely it is that high ranked individuals are deleted).

3.2 Performance Database Generation

We generated a performance database as follows:

Ten random 40-city TSP instances were generated as described above.

54 different configurations of the GA were generated, by selecting values for four control parameters (population size, crossover probability, mutation probability, selection bias), where:

- $Population \in \{10, 100, 250\}$,
- $Bias \in \{1.0, 1.5, 2.0\}$, the selection bias,
- $Pr(Crossover) \in \{0.0, 0.25, 0.5\}$, the probability of uniform crossover, and
- $Pr(Mutate) \in \{0.01, 0.05\}$, the probability of each bit being flipped.

For each of the TSP instances, we executed each of the GA configurations using 30 different random seeds. Each run was for 20000 iterations (objective function evaluations), i.e., the number of generations was chosen so that the $population \times NumGenerations = 20000$. Every 1000 iterations, the length of the shortest tour found so far by the current GA configuration for the current TSP instance for the current random seed was stored in the performance database.

For each TSP instance, we found the l_{max} and l_{min}, the longest and shortest tour lengths found (by all GA configurations and random seeds), and normalized all of the performance database entries by rescaling each value to a range between [0,1], according to the formula $v_{rescaled} = (v_{original} - l_{min})/(l_{max} - l_{min})$.

[1] $RAND$ is a function that returns a random floating point number between 0 and 1.

3.3 Anytime Algorithm Portfolios vs. Individual Algorithms

We compared the performance of the anytime algorithm portfolio approach versus the performance of the standard single-algorithm approach.

First, using the performance database generated above, we used Using ***synthesize-portfolio*** where T=30000, k=1000, $NumSamples$ = 100, we optimized a portfolio for our distribution of 40-city TSP instances. We call this the $BestPortfolio$. In this experiment, we found that $BestPortfolio = \{(A_1, 10000), (A_2, 3000), (A_3, 10000), (A_4, 7000)\}$, where:

- A_1 is the configuration with $Population$=10, $Pr(Crossover)$=0.25,$Pr(Mutate)$=0.01, $Bias$=2.0,
- A_2 is the configuration with $Population$=100, $Pr(Crossover)$=0.0,$Pr(Mutate)$=0.05, $Bias$=1.5,
- A_3 is the configuration with $Population$=10, $Pr(Crossover)$=0.25,$Pr(Mutate)$=0.01, $Bias$=2.0, and
- A_4 is the configuration with $Population$=100, $Pr(Crossover)$=0.5,$Pr(Mutate)$=0.01, $Bias$=1.5.

Then, for each of the 54 GA configurations from which the performance database was generated, we optimized a single-algorithm restart strategy $BestSingleAlgRestart_i$ ($1 \leq i \leq 54$). We executed ***synthesize-portfolio*** with T=30000, k=1000, $NumSamples$ = 100, but for each run, we restricted the domain of algorithms to A_i, where A_i was the ith GA configuration. That is, $BestSingleAlgRestart_i$ is the best static restart strategy for GA configuration i for our distribution of 40-city TSP instances. We then selected the single algorithm restart strategy with the best expected performance, $BestSingleAlgRestart$.

Note that $BestSingleAlgRestart$ is the configuration which would be selected as "the most useful single configuration" of the GA in a traditional parameter tuning approach, i.e., it is the set of control parameter settings which, on average, performs best on our problem distribution. In this experiment, we found that $BestSingleAlgRestart = \{(A, 10000), (A, 8000), (A, 3000), (A, 9000)\}$, where A is the GA configuration with parameter values: $Population = 10$, $Pr(Crossover) = 0.5$, $Pr(Mutate) = 0.01$, $Bias = 2.0$

Ten new random 40-city TSP instances were generated using the same problem generator used to generate the problems used for creating the performance database. These test problems were labeled *40-cities-1*, .. *40-cities-10*.

For each test problem, we applied both the $BestPortfolio$ and $BestSingleAlgRestart$ single algorithm restart strategies for T = 30000. The mean score

Problem	*BestPortfolio*	*BestSingleAlgRestart*
40cities-1	0.102 (0.041)	0.124 (0.045)
40cities-2	0.074 (0.042)	0.071 (0.050)
40cities-3	0.111 (0.019)	0.131 (0.029)
40cities-4	0.072 (0.021)	0.094 (0.018)
40cities-5	0.089 (0.027)	0.127 (0.024)
40cities-6	0.095 (0.038)	0.093 (0.030)
40cities-7	0.109 (0.024)	0.111 (0.034)
40cities-8	0.082 (0.015)	0.099 (0.032)
40cities-9	0.097 (0.032)	0.113 (0.036)
40cities-10	0.124 (0.036)	0.110 (0.022)

Table 1: Summary of results: mean and standard deviation of the best scores (normalized tour length) found by the *BestPortfolio* and *BestSingleAlgRestart* strategies on ten 40-city TSP instances.

(length of shortest tour found in run) and variance of 25 independent runs were recorded.

The results are summarized in Table 1. For each problem, we show the mean best objective function score and standard deviation (of 30 runs) of the *BestPortfolio* score, as well as the mean and variance (or 30 runs) of the *BestSingleAlgRestart* strategy.

For six out of the ten test TSP instances, *BestPortfolio* significantly outperformed *BestSingleAlgRestart*. For three instances (*40cities-2, 40cities-6, 40cities-7*), the results are comparable. In one instance (*40cities-10*), *BestSingleAlgRestart* clearly outpermed *BestPortfolio*. Thus, on average, *BestPortfolio* performed significantly better than *BestSingleAlgRestart*.

4 Related Work

Algorithm portfolios are based on the theory of investment portfolios developed in the field of economics. Portfolio theory was developed to answer the question: "How should one allocate his/her financial assets (stocks, bonds, etc) in order to maximize expected returns while minimizing returns?" The anytime algorithm portfolio framework is quite similar to the single-period portfolio model developed by Markowitz [8]. A significant difference between anytime algorithm portolios and investment portfolios is that the expected utility of an anytime algorithm portfolios is the expected value of the best solution computed by the portfolio (i.e., a MAX operation), while the expected rate of return of an investment portfolio is a weighted average of the expected returns of its component investments. As a consequence, computing the optimal anytime algorithm portfolio is not as straightforward as computing an optimal investment portfolio.

Huberman, Lukose and Hogg [7] recently proposed the use of "computational portfolios" composed of multiple algorithms to solve combinatorial search problems and other hard computational problems. They showed how two or more Las Vegas algorithms[2] executed concurrently can result in an ***algorithm portfolio*** whose expected time and variance to find a solution to an NP-complete problem (graph coloring) is significantly better than its component algorithms. Gomes and Selman [4] applied a portfolio Las Vegas algorithms to the quasigroup completion problem.

Here, we have extended the algorithm portfolio approach to resource-bounded optimization, combining instances of evolutionary algorithms into algorithm portfolios for optimization. Unlike the Las Vegas algorithms studied in previous work on algorithm portfolios, optimization algorithms such as evolutionary algorithms are ***anytime algorithms*** [14] that can be interrupted at any time to return a solution with some utility.[3] As a result of this, a significant difference between our work and this previous work in algorithm portfolios is that in the previous work, the goal is to minimize expected time to find a solution of acceptable quality (there is no deadline), while our framework has a resource bound.

The decision-theoretic framework we adopted in our study of algorithm portfolios is related to work on decision-theoretic reasoning and meta-level reasoning in artificial intelligence. Much of the previous work in the area has focused on meta-level control of the tradeoff between computation and action (c.f., [6, 1]), as well as lower-level resource allocation problems (e.g., which nodes to explore in a search tree) (c.f. [9]). Zilberstein and Russell [15] studied the composition of systems of anytime algorithms, where each component is an anytime algorithm whose output is the input to the next anytime algorithm in the system (e.g., a robot navigation system system composed of an anytime sensing algorithm and an anytime navigation algorithm).

5 Discussion/Future Work

This paper proposed anytime algorithm portfolios as a method for allocating limited computational resources among sets of candidate control parameter values for evolutionary algorithms. Using the TSP problem domain, we showed that the anytime algorithm portfolio approach yields a resource allocation among multiple several control parameter value sets which can be superior to that of the traditional single configuration resource allocation model.

It is important to note that applying the algorithm portfolio technique requires no more data than what is already collected in the course of a standard parameter tuning experiment. What

[2]Algorithms which always produce the correct solution to a problem, but with a distribution of solution times

[3]It is possible to view Las Vegas algorithms as a special case of an anytime algorithm where the utility of all partial solutions have the identical, worst utility score.

we propose is simply an alternative way to exploit the data that is collected in a standard parameter tuning experiment. In addition, the application of the technique to a new domain is a completely domain-independent process.

An interesting consequence of being able to synthesize algorithm portfolios is that in some domains, it may be worthwhile to focus some research on algorithms that have poor expected performance on most instances of a problem class but excel on some rare instances where the "better" algorithms exhibit pathological/poor performance. Although such algorithms would likely be discarded/ignored because of poor average performance in the standard methodology of comparative empirical algorithm research, the portofolio framework provides a rational approach to allocating some computational resources to these "outlier-specific" algorithms.

Although in this paper, we applied portfolios in a context where each of the component "algorithms" was a different parameterization of the same basic GA, portfolios can also be straightforwardly applied when the set of algorithms encompasses a much lager domain of algorithms. In future work, we will investigate combinations of evolutionary algorithms with significantly different algorithms (e.g., systematic heuristic search and simulated annealing). Furthermore, it should be noted that algorithm portfolios could be applied in conjunction with "self-adaptive" algorithms which automatically adjust their control parameters within a single run, since even these adaptive algorithms have meta-level parameters or initial values for parameters, and using portfolios to allocate resources between multiple sets of these meta-level parameters may be worthwhile.

The resource-bounded optimization model discussed in this paper is only one model to which anytime algorithm portfolios can be applied. Other interesting models which we will investigate include:

- Fixed resource cost - a model where the resource bound is not set a priori but each unit of resource usage has a cost, and the utility function for the portfolio must consider minimization of resource usage. This is the model adopted in [7, 4] in their work with portfolios of decision algorithms.

- Stochastic deadline model - a model where the resource bound is not known a priori; the system must maximize utility some distribution of deadlines. This model is particularly interesting because of the opportunity to exploit the risk-reduction potential of algorithm portfolios.

One major limitation of the current portfolio framework is the assumption that we restrict ourselves to restart strategies in which each restart is independent. While this is an accurate model for many circumstances, a significant extension to the model would address the issue of non-independent restarts, e.g., evolutionary algorithm variants that can be "seeded" by the best individuals from previous restarts. Conditional performance profiles, proposed by Zilberstein and Russell [15] for modeling sequences of anytime algorithms whose output quality is dependent on the input, may be useful for this purpose.

Acknowledgments

Portions of this research were supported by Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

Bibliography

[1] M. Boddy and T.L. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67(2):245–85, 1994.

[2] B. Efron and R. Tibshirani. Statistical data in the computerage. *Science*, 253:390–395, 1991.

[3] A. Fukunaga. Restart scheduling for genetic algorithms. In *Proc. Parallel Processing from Nature (PPSN)*, pages 357–358, 1998.

[4] C.P. Gomes and B. Selman. Algorithm portfolio design: theory vs. practice. In *Proc. Uncertain in Artificial Intelligence (UAI)*, 1997.

[5] A. Homaifar, S. Guan, and G.E. Liepins. A new approach on the traveling salesman problem by genetic algorithms. In *Proc. International Conf. on Genetic Algorithms (ICGA)*, pages 460–466, 1993.

[6] E.J. Horvitz. *Computation and action under bounded resources.* PhD thesis, Stanford University, Program in Medical Information Science, 1990.

[7] B.A. Huberman, R.M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275(5269):51–4, January 1997.

[8] H.M. Markowitz. Portfolio selection. *Journal of Finance*, 7(1):77–91, 1952.

[9] S.J. Russell and E. Wefaldf. *Do the Right Thing.* MIT Press, 1991.

[10] T. Starkweather, S. McDaniel, K. Mathias, D. Whitley, and C. Whitley. A comparison of genetic sequencing operators. In *Proc. International Conf. on Genetic Algorithms (ICGA)*, pages 69–76, 1991.

[11] G. Syswerda. Uniform crossover in genetic algorithms. In *Proc. International Conf. on Genetic Algorithms (ICGA)*, 1989.

[12] D. Whitley. The genitor algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In *Proc. International Conf. on Genetic Algorithms (ICGA)*, pages 116–121, 1989.

[13] D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical Report SFI-TR-05-010, Santa Fe Institute, 1995.

[14] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996.

[15] S. Zilberstein and S. Russell. Optimal composition of real-time systems. *Artificial Intelligence*, 82(1-2):181–213, 1996.