Predictive Search Distributions

Edwin V. Bonilla EDWIN.BONILLA@ED.AC.UK
Christopher K. I. Williams CKIW@INF.ED.AC.UK
Felix V. Agakov FELIXA@INF.ED.AC.UK
John Cavazos JCAVAZOS@INF.ED.AC.UK
John Thomson JOHN.THOMSON@ED.AC.UK
Michael F. P. O'Boyle MOB@INF.ED.AC.UK
School of Informatics, University of Edinburgh, 5 Forrest Hill, Edinburgh EH1 2QL, UK

Abstract

Estimation of Distribution Algorithms (EDAs) are a popular approach to learn a probability distribution over the "good" solutions to a combinatorial optimization problem. Here we consider the case where there is a collection of such optimization problems with learned distributions, and where each problem can be characterized by some vector of features. Now we can define a machine learning problem to predict the distribution of good solutions $q(s|\mathbf{x})$ for a new problem with features $\mathbf{x}$, where s denotes a solution. This predictive distribution is then used to focus the search. We demonstrate the utility of our method on a compiler optimization task where the goal is to find a sequence of code transformations to make the code run fastest. Results on a set of 12 different benchmarks on two distinct architectures show that our approach consistently leads to significant improvements in performance.

1. Introduction

In this paper we consider optimization problems and their solution. As input we are given a description X of the optimization problem, for example the edge weights between all vertices of a graph for a minimum balanced cut (MBC) problem[1]. For any input X there is a set $S(X)$ of valid solutions; for MBC this is the set of bisections that are balanced. We also require an evaluation function f which takes as input a problem description X and a valid solution $s \in S(X)$, and outputs the quality of the solution. For the MBC problem f is the sum of the edge weights in the cut. Our goal is then to find the optimal solution $s^*(X)$ such that

$$s^*(X) = \text{argmin}_{s \in S(X)} f(X, s). \tag{1}$$

Another example of an optimization problem, indeed the one that motivated this work, is in compiler optimization. For a given benchmark X we can apply a sequence of transformations to the code, so as to produce the same input-output behaviour but different runtimes. (Examples of transformations are loop unrolling and common subexpression elimination.) Our goal is to find the sequence of transformations that makes the code run fastest. See section 4.1 for more details on this problem.

Imagine now that we have a collection of examples of the same kind of problem, but with different descriptions X drawn from a space $\mathcal{X}$. In this case if we have solved the optimization problems we will have a set of solutions $(X_1, s^*(X_1)), (X_2, s^*(X_2)), \ldots, (X_n, s^*(X_n))$. (Here $s^*(X_i)$ could be an approximate solution to the problem rather than the true global optimum.) One idea to apply machine learning to this optimization problem is to learn the mapping between X and $s^*(X)$ based on examples. However, this may be a very difficult mapping to learn; one problem is that small changes to the input X of a combinatorial optimization problem may lead to very different solutions.

We will take a rather different approach, mapping from a problem description X to a probability distribution

[1] The minimum cut of a graph can be found in polynomial time using network flow methods. However, this algorithm does not guarantee that the two halves are balanced, i.e. that there are equal numbers of nodes in each half. The MBC problem is NP-hard.

Appearing in *Proceedings of the 23rd International Conference on Machine Learning*, Pittsburgh, PA, 2006.

$q(s|X)$ over good solutions. Our motivation is that it may be very hard to predict the optimal solution $s^*(X)$ for an instance X, but that it may well be easier to define a search distribution which gives high probability to $s^*(X)$. An inspiration for this idea is recent work on *estimation of distribution algorithms* (EDAs), see e.g. Larrañaga and Lozano (2001), where for a given problem X a search over solutions is conducted by repeatedly re-estimating a probability distribution over good solutions (see section 2.1 for more details on EDAs). However, note that EDAs are concerned with finding an optimal solution $s^*(X)$ for a given problem X, while our goal is to make use of what has been learned from previous problems in predicting the search distribution for a new instance.

Our goal of mapping from X to a predictive distribution $q(s|X)$ is actually similar to the standard probabilistic machine learning set up where the output is a predictive distribution, e.g. a Bernoulli probability or a univariate Gaussian distribution. However, we note two special aspects of modelling search distributions:

- The distribution $q(s|X)$ is meant to *focus* our search. Thus we might well make multiple draws from it in the hope of finding better solutions.
- The predictive distribution $q(s|X)$ might be complicated (e.g. it might be a graphical model); this kind of complexity is not commonly used in standard machine learning situations, although it does arise, e.g. in conditional random fields.

The structure of the remainder of the paper is as follows: in section 2 we give some background on EDAs and describe the theory of predictive search distributions. In section 3 we relate our approach to previous work. In section 4 we discuss the set up for experiments on compiler optimization, and in section 5 we give our results on this task. We conclude with a discussion in section 6.

2. Theory

This section briefly describes estimation of distribution algorithms (EDAs) as a method for solving combinatorial optimization problems and explains how our approach makes use of EDAs by presenting the theoretical framework of predictive search distributions.

2.1. EDAs

Many combinatorial optimization problems are NP-hard and for such problems it is common to use heuristic optimization methods, for example those based on population search. Such methods include genetic algorithms, which explore the search space by evolving populations of candidate solutions. Estimation of distribution algorithms (EDAs) may be viewed as a way of evolving a probabilistic graphical model $g(s) \in \mathcal{G}$ describing a distribution of good candidates, rather than evolving a specific population (Pelikan et al., 1999). These methods are particularly popular for addressing combinatorial optimization problems, although they have also been applied in continuous domains. In the rest of the paper we will focus specifically on the case when $\{s\}$ is space of strings defined over finite alphabets.

The algorithm for a typical EDA is given below. Here N is the size of the population, assumed to be the same at each iteration, and $\mathcal{U}$ denotes the uniform distribution.

1. *initialization*: constrain the family of the EDA search distributions $\mathcal{G}$; set $t = 0$, N, etc.
2. *generate initial population*: $\mathcal{S}^{(0)} = \{s_i|_{i=1}^N \sim \mathcal{U}\}$;
3. *evaluation*: $\forall s_i \in \mathcal{S}^{(t)}$ compute $f(s_i)$;
4. *selection*: choose a subset $\tilde{\mathcal{S}}^{(t)} \subseteq \mathcal{S}^{(t)}$ biased towards better-performing solutions; define the empirical distribution $\tilde{p}^{(t)}$ on the subset;
5. *learning*: learn $g^{(t+1)}$ by optimizing a discrepancy measure $D(\tilde{p}^{(t)}; g \in \mathcal{G})$ (e.g. KL-divergence);
6. *sampling*: generate $\mathcal{S}^{(t+1)} \stackrel{def}{=} \{s_i|_{i=1}^N \sim g^{(t+1)}\}$;
7. iterate steps 3–6 until a termination criterion is met.

After termination, the optimal solution s^* is approximated as the $\arg\max_{s \in \mathcal{S}^{(T)}} f(s)$, where $\mathcal{S}^{(T)}$ is the final population. Usually the models $g \in \mathcal{G}$ are constrained to lie in tractable parametric families, and there are a large number of EDAs which are based on specific parameterizations. For example, the population-based incremental learning (PBIL) algorithm assumes that $\mathcal{G}$ is a family of factorized distributions; mutual information-maximizing input clustering (MIMIC) methods constrain $\mathcal{G}$ to be a family of Markov chains, etc. See e.g. Pelikan et al. (1999) for an overview of specific algorithms.

2.2. Predictive Search Distributions

The key issue that we now address is how to make predictions on a new problem instance X given a training set of problems and their corresponding search (or empirical) distributions. We assume that for any problem description $X \in \mathcal{X}$ we can extract a number of features $\mathbf{x}$ that (partially) characterize the problem. We seek to learn a predictive distribution $q(s|\mathbf{x}, \theta)$ that outputs a distribution over s given an input $\mathbf{x}$ and parameters θ. Thus θ can be set by maximizing the conditional likelihood

$$\mathcal{L}_{s|X} \stackrel{def}{=} \sum_{i=1}^{n} \sum_{s} \tilde{p}^{(T)}(s|X_i) \log q(s|\mathbf{x}_i, \theta), \qquad (2)$$

where $\tilde{p}^{(T)}(s|X_i)$ is the empirical distribution over solutions output at the end of the EDA run on problem X_i. An alternative objective function would be

$$\sum_{i=1}^{n} \sum_{s} g^{(T)}(s|X_i) \log q(s|\mathbf{x}_i, \theta). \qquad (3)$$

This is similar to (2) but uses the output EDA distribution $g^{(T)}(s|X_i)$ for each problem rather than the empirical distribution. However, in cases where the family of distributions $\mathcal{G}$ in the EDA has hidden variables (e.g. a HMM) then (2) should be easy to evaluate while (3) may not be.

In practice we can consider a number of parametric and/or structural constraints on the family of predictive distributions $q(s|\mathbf{x}, \theta)$. Since in our case the distribution is defined over strings, choices for $q(s|\mathbf{x}, \theta)$ include logistic regression and conditional random field models (Lafferty et al., 2001). Once again we note that we will be using $q(s|\mathbf{x}, \theta)$ to focus search for a new problem X. An alternative approach is to use a k nearest-neighbours (k-NN) method. For example using 1-NN, we can set the predictive search distribution to be the EDA distribution of the problem whose features are the closest to the new problem. For $k > 1$ we could use a mixture of distributions from the k closest neighbours. This method may be particularly useful if the number of training problems n is small.

It is also worth remarking that the predictive search distribution methodology may not always lead to speed-ups, and could in principle degrade performance relative to uniform search. This would be the case if the features extracted did not provide useful information about the search distribution, or if $g(s|X)$ varies very rapidly with changes in the input X so that a very large amount of training data would be needed to characterize the problem class.

3. Related Work

We are not aware of much previous work in this area. One common way to help speed up search problems is through memoization, i.e. remembering the answers to previous problems (or partial problems) so as to eliminate search if they are encountered again. The proposed method goes beyond this in that it affords inductive generalization to new search problems rather than simply storing earlier results.

There has also been a lot of work on learning search-control knowledge, see e.g. chapter 10 in Langley (1996) for an overview. This work focuses on planning in sequential decision problems (SDPs), i.e. the task of reaching a goal state from a start state; this general area is addressed by reinforcement learning. There has also been work on speeding up search using explanation-based learning (EBL) using solution traces and a domain theory. However, the search problems we are addressing are not SDPs and so this work does not apply.

One possible alternative approach is to learn a regression function $\hat{f}$ that approximates $f(X, s)$ from data samples over the $\mathcal{X} \times S$ input space. However, if space of solutions S is very large then even if this proxy function can be learned accurately it would be very time consuming to find the string s that optimizes $\hat{f}(X, s)$; thus we prefer an approach that directly outputs a distribution of "good" solutions.

Over the past few years, there has been a lot of interest in the topic of *inductive transfer*, see e.g. Thrun and O'Sullivan (1996) and Caruana (1997) as some of the earlier references. In these (and later) papers a common set up is that there are multiple, related supervised learning problems and that the goal is to avoid tabula rasa learning for a new problem by extracting information from the problems seen before. Thus the learning is at a higher level, e.g. by using the previously seen problems to define priors on parameters for the new problem. Our suggestion for learning search distributions is actually more like the standard probabilistic machine learning set up at the lower level, where the prediction is a probability distribution.

4. Experiments

This section introduces the problem of compiler optimization and describes the experiments that were carried out with the goal of improving iterative compiler optimization with predictive search distributions.

4.1. The Problem

Compiler optimization deals with the problem of making a compiler produce better code, i.e. code that runs fast. Numerous program transformations have been proposed in the literature and implemented in commercial and research compilers for this purpose. A program transformation can be thought of as a process that changes the code in order to exploit the resources of a target architecture more efficiently while maintaining the meaning and correctness of the original program. An example of a program transformation is loop unrolling, which replicates the body of a loop several times. This transformation can be beneficial as it exposes opportunities for instruction level parallelism (running multiple instructions at the same time) and also reduces the overhead due to loop control. However, loop unrolling can also be detrimental when the loop body is augmented excessively so that it cannot be kept in the cache. As with loop unrolling, there are many other transformations for which it is difficult to know when or how a compiler should apply them to a specific program. Additionally, these transformations interact in an almost mysterious way making the problem of producing optimal code even harder.

An interesting scenario in compiler optimization is *iterative compilation*, where one can afford several program executions in order to determine a set of program transformations that significantly increase performance. This task can be formulated as a combinatorial optimization problem where a set of transformations can be combined into sequences of arbitrary length. This approach of searching the space of transformation sequences has been shown to provide excellent performance at the cost of a large number of evaluations of a program (Franke et al., 2005).

Considering that similar programs may have similar behaviour under the application of several code transformations we propose Predictive Search Distributions for improving iterative optimization. Making explicit the notation used throughout this paper, $\mathbf{x}$ is a set of features extracted from a program and the predictive distribution $q(s|\mathbf{x})$ is a distribution over good transformation sequences. Thus, our goal is to learn $q(s|\mathbf{x})$ and use this distribution to guide search on a new program that has not been seen before.

4.2. Technical Details

Twelve different **benchmarks** from the UTDSP (Lee, 1997) suite have been used for the experiments. This set of C programs contains small kernels as well as larger applications. These are regarded as compute-intensive programs by the DSP community, and continuously used in stream-processing applications.

We have considered source-to-source **transformations** applicable to C programs by using the restructuring compiler framework SUIF 1 (Hall et al., 1996). Using these transformations, we have investigated two different spaces: a **small space** composed by 14 transformations (selected by compiler experts) combined into sequences of length 5 and a **large space** with 90 code transformations forming sequences of length 20. The former represents a space of 14^5 sequences which we have exhaustively enumerated. The latter represents a space of 90^{20} sequences, which we have sampled by using a PBIL-like algorithm (Franke et al., 2005), obtaining around 2000 samples per program. Collecting the data for this task is a time-consuming activity as every sample corresponds to a compilation and an execution of a program. It takes around 3 days to run one benchmark over all 14^5 sequences.

The experiments were executed on two distinct **platforms** to show that our approach does not depend on a specific architecture or compiler. The first platform is a Texas Instruments C6713 board, a high end floating point DSP running at 225MHz with 256kB of internal memory. The programs were compiled using the Texas Instruments' Code Composer Studio Tools Version 2.21 with the highest -O3 optimization level. We will refer to this platform henceforth as the **TI** board. The second architecture used is an AMD Alchemy Au1500 processor running at 500MHz with 16KB instruction cache and 16kB data cache. The programs were compiled using GCC 3.2.1 with -O3 flag, which according to the manufacturer provided the best performance. This platform will be called henceforth the **AMD** architecture.

4.3. Speed-ups Obtained

In order to evaluate the quality of a transformation sequence we use the speed-up (u) as a measure of performance: $u = f(X, \emptyset)/f(X, s)$, where $f(X, \emptyset)$ is the execution time of program X when no transformations are applied (the baseline) and $f(X, s)$ is the execution time of the same program when a transformation sequence s is applied. Note that this measure of performance ranges in the interval $(0, \infty)$, where a number between zero and one means that a transformation sequence slows down the execution of the program and a speed-up greater than one indicates an improvement in performance. In practice, however, we can consider speed-ups greater than 1.05 as significant improvements and speed-ups close to 2 as excellent improvements as this means that the execution time has been reduced to half of the original program's.

Table 1 shows the best speed-ups obtained for the small and large spaces in both architectures: the TI and the AMD. We see that significant speed-ups have been obtained on average for both platforms and that most benchmarks could be improved with the exper-

Table 1. Best speed-ups obtained in the experiments.

	Speed-up TI		**Speed-up AMD**	
Program	**Small**	**Large**	**Small**	**Large**
FFT	1.04	1.84	1.05	1.08
FIR	1.84	1.86	1.36	1.61
IIR	1.19	1.19	1.42	1.42
LATNRM	1.00	1.02	1.37	1.54
LMSFIR	1.00	1.00	1.43	1.43
MULT	1.00	1.06	1.81	2.00
ADPCM	1.32	1.33	1.01	1.01
COMPRESS	1.64	1.65	1.79	1.89
EDGE	1.30	1.52	1.45	1.39
HISTOGRAM	1.00	1.01	1.33	1.41
LPC	1.12	1.16	1.06	1.07
SPECTRAL	1.08	1.19	1.09	1.36
Average	1.21	1.32	1.35	1.43

iments. These results are important as they show that good improvements can be obtained with iterative compilation and that the data generated presents opportunities for learning. Additionally, the speed-ups obtained for the AMD are greater than those obtained for the TI, showing that the compiler in the former (GCC) is easier to improve than the commercial compiler in the latter. However, the speed-ups obtained on the TI board are more than encouraging in the compiler community, as the compiler used on this board is believed to produce high quality code for these types of applications. Finally, these results show that searching a large space such as the one considered in our experiments yields further improvements. Thus, it makes sense to use techniques such as Predictive Search Distributions in order to focus search over good subspaces of transformation sequences.

4.4. Experimental Setup

We have fitted two classes of distributions to the set of good transformation sequences for each program. For our results, we have (arbitrarily) defined a good transformation sequence as a sequence that has an improvement in performance at least 95% of the maximum improvement achieved.

The first distribution class we have used is an **iid** distribution, where the transformations within a sequence are considered independent, so that $g(s_1, s_2, \ldots, s_L) = \prod_{i=1}^{L} g(s_i)$ where L is the length of the sequence considered. This iid model neglects the effect of interactions among transformations, which can be very restrictive as some transformations enable the applicability of others, and there are transformations that yield good performance only when others have been previously applied. Bearing in mind that more complex models can involve a much greater number of parameters, we have used a stationary **Markov** chain as our second model to focus search. In this model, the probability of a transformation being applied in a specific position of a sequence depends on the one that has been previously applied, so that $g(s_1, s_2, \ldots, s_L) = g(s_1) \prod_{i=2}^{L} g(s_i|s_{i-1})$. Note that searching with this Markov distribution differs from the MIMIC algorithm (de Bonet et al., 1997), which uses a non-stationary Markov chain. We have fitted these distributions by maximum likelihood estimation and using pseudocounts of 0.001.

As in any other machine learning task a significant difficulty for this problem is to extract relevant **features** for learning. For this purpose, we have relied on the knowledge of compiler experts who have identified thirty-four program features believed to describe the characteristics of a program well and to be relevant for our specific task. The number of instructions in loops and the number of array references in loops are examples of such features. A complete list of the features and compiler transformations used is available at `http://www.anc.ed.ac.uk/machine-learning/colo/psd_list.html`.

Given the high-dimensional search space and the limited amount of training data (only twelve benchmarks), it seems rather difficult to learn search distributions for our set of programs. Therefore, we have reduced the dimensionality of the input to five features by using PCA and tested our approach with **1-nearest neighbour** predictor in a Leave-One-Out Cross-Validation (LOOCV) procedure. Thus, we have predicted the search distribution for a program by simply using the distribution of its nearest neighbour.

5. Results

This section analyzes the results of applying our approach to the problem of iterative compiler optimization.

5.1. Evaluation on Small Space

Before presenting the results for the predictive search distributions we want to evaluate the potential of our method by using the actual distribution that has been fitted to each program. Let us call these distributions the *IID-oracle* and the *Markov-oracle*. They are oracles in the sense that they provide an upper bound on our expectations of speeding-up search by our learned models. Therefore, our first experiment aims to show that these oracles do improve search. Clearly, if this

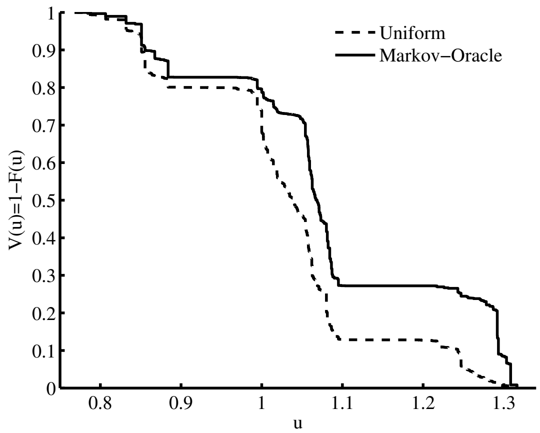

Figure 1. Survival function of uniform and Markov-oracle adpcm (TI) for the small space of 14^5 sequences.

is not the case, it is not worth expending effort trying to learn these distributions.

Figure 1 shows an example of a survival function $V(u) = 1 - F(u)$, where $F(u)$ is the cumulative distribution function for the speed-ups u achieved by uniform search, and search guided by the Markov-oracle distribution. This has been obtained for the benchmark *adpcm* on the TI board. By definition, $V(u)$ is the probability of a speed-up being greater than u, i.e. $P(U > u)$. Therefore, greater values of $V(u)$ for near-optimal speed-ups translate into higher probabilities of finding good transformation sequences. Indeed, the ideal case would occur if all the density mass was concentrated on the maximum speed-up $u^* = \max_{u \in U}(u)$ so $V(u)$ would be one for all $u < u^*$ but zero for $u = u^*$. We can see that the Markov-oracle $V(u)$ outperforms the uniform $V(u)$ for all speed-ups greater than one. This means that if we use the Markov-oracle distribution to guide search we will have a greater chance of obtaining a good transformation sequence, and that the number of samples needed to achieve good performance will be reduced (this behaviour was consistent for all the benchmarks using both oracle distributions in both architectures).

One can compute the expected number of samples $E[n]$ needed to reach a good data-point (first success) when assuming random sampling. If we define a good data-point in the performance space as the one for which the speed-up is greater than certain value u^+, the expected number of samples needed to achieve this performance is $1/V(u^+)$. In order to evaluate the benefits of using a distribution for guiding search let us define the Search Improvement Factor $SIF = E_u[n]/E_s[n]$, where $E_u[n]$ is the expected number of samples needed to achieve good performance when using the uniform distribution and $E_s[n]$ is the expected number of samples required when using a specific search distribution (such as the IID-oracle). Thus, we will prefer SIF values greater than one as they indicate that a distribution speeds up search.

Table 2. Expected number of samples for uniform distribution and search improvement factors for **oracle** distributions to obtain 95% of maximum performance.

	TI			**AMD**		
		SIF			***SIF***	
Pr.	$E_u[n]$	**I**	**M**	$E_u[n]$	**I**	**M**
FFT	11.3	2.7	5.4	1.4	1.1	1.2
FIR	20.6	12.2	14.2	8.8	3.6	5.1
IIR	1.8	1.1	1.2	17.7	7.2	10.6
LAT	-	-	-	21.8	7.1	11.7
LMS	-	-	-	15.2	6.9	9.5
MUL	-	-	-	82.6	17.5	39.8
ADP	66.3	3.6	13.8	-	-	-
COM	13.8	7.2	9.2	55.9	12.0	25.9
EDG	96.4	18.1	45.4	29760	457	26720
HIS	-	-	-	9.5	6.3	7.0
LPC	83.6	15.7	36.7	13.6	3.6	6.2
SPE	2.6	1.5	1.8	14.3	3.3	5.2
Av.	37.0	5.0	8.8	2727	8.1	17.4

Table 2 shows the expected number of samples $E_u[n]$ needed to achieve good performance for uniform search and the search improvement factor SIF for the oracle distributions, where a good solution has been defined to be at least 95% of the maximum performance achieved for each benchmark. The first column corresponds to the benchmarks used and the columns I and M correspond to iid and Markov distributions respectively. Note that those benchmarks for which no speed-up was obtained during the exhaustive experiments are marked with '-' indicating that their spaces are not worth searching. It can be seen that both oracle distributions consistently improved search in both architectures. Furthermore, although in some easy-to-search spaces such as *iir* and *spectral* for the TI or *fft* for the AMD only marginal benefits are obtained, in difficult spaces such as *adpcm*, *edge* and *lpc* for the TI or *mult*, *compress* and *edge* for the AMD, the oracles speeded up search by an order of magnitude or more[2]. It is also possible to conclude that modelling interactions by using a Markov chain does lead to greater improvements compared to an iid distribution. On average[3], the iid distribution and the Markov model

[2] The benchmark *edge* is a needle-in-a-haystack problem and the oracle distributions dramatically improved search.

[3] We have used the arithmetic mean for computing the

Table 3. Expected number of samples for uniform distribution and search improvement factors for **predictive** distributions to obtain 95% of maximum performance.

	TI			AMD		
		SIF			***SIF***	
Pr.	$\boldsymbol{E_u[n]}$	**I**	**M**	$\boldsymbol{E_u[n]}$	**I**	**M**
FFT	11.3	0.8	0.8	1.4	1.1	1.1
FIR	20.6	6.5	7.8	8.8	1.4	1.5
IIR	1.8	1.0	1.0	17.7	6.3	6.9
LAT	-	-	-	21.8	6.3	6.8
LMS	-	-	-	15.2	3.3	4.4
MUL	-	-	-	82.6	12.7	24.2
ADP	66.3	1.4	1.4	-	-	-
COM	13.8	6.8	7.9	55.9	13.4	27.4
EDG	96.4	2.2	2.3	29760	6.8	0.2
HIS	-	-	-	9.5	6.3	7.0
LPC	83.6	2.2	2.3	13.6	3.3	4.7
SPE	2.6	0.9	0.9	14.3	3.3	4.8
Av.	37.0	2.0	2.1	2727	4.5	4.1

improved search by factors of 5.0 and 8.8 for the TI and 8.1 and 17.4 for the AMD respectively.

Having shown that using oracle distributions leads to an improvement in the search, the next step is to evaluate the predictive distributions when using 1-nearest neighbour. These results are shown in Table 3, where we have abbreviated the names of the programs and distributions as before, and the benchmarks marked with '-' indicate that their spaces are uninteresting to search. Not surprisingly, the search improvement factors of the predictive distributions are upper-bounded by the search improvement factors of the oracle distributions (with the exception of *compress* on the AMD). However, for most benchmarks the predictive distributions did improve search by focusing on regions of the space where good performance was obtained. Indeed, only for two benchmarks on the TI board (*fft* and *spectral*) and one benchmark on the AMD (*edge* when using the Markov distribution) did the predictive distributions harm performance. This latter case suggests overfitting, as the iid model speeded up search and has fewer parameters than the Markov model. It also confirms that transfer of knowledge to a needle-in-a-haystack problem can be very difficult, especially when having a small number of training points. However, one can alleviate this effect by mixing the fitted distributions with the uniform distribution using weight factors of α and $(1-\alpha)$ respectively[4].

average of $E_u[n]$ and the geometric mean for averaging SIFs.

[4]Using $\alpha = 0.8$ the SIF for *edge* on the AMD with the Markov model increased to 0.7 while the average SIF for all the other benchmarks decreased from 5.7 to 4.9.

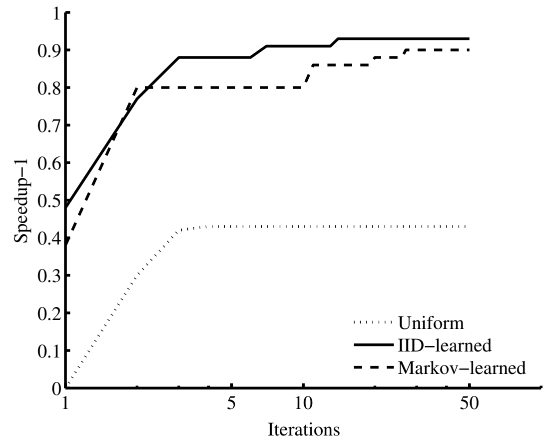

Figure 2. Performance curves for *mult* on AMD.

5.2. Evaluation on Large Space

In contrast with the small space, we do not have exhaustive data for the large space and thus we cannot compute $E_u[n]$ in order to evaluate the benefits of our approach. Therefore, we have considered a different measure of comparison: the area under the performance curve (AUC) computed on empirical results averaged over 10 repetitions. Figure 2 shows an example of performance curves for the benchmark *mult* in the AMD architecture. A performance curve describes the best speed-up achieved so far by a search algorithm as a function of the number of iterations. Note that we have used speed-up minus one on the y-axis (so that no speed-up corresponds to zero) and a log-scale for the x-axis. It is clear that the AUC will reward those methods that reach better performance and those that achieve good speed-ups in fewer iterations. The AUCs for the TI and AMD are shown in Figure 3, where the computations have been done using performance curves for each benchmark like the one illustrated in Figure 2 after fifty iterations. Note that the results are shown only for those benchmarks for which an improvement was obtained with these experiments. On the TI board, the iid distribution provides the best performance for most benchmarks, with a dramatic improvement achieved for *fft*. The Markov distribution improves performance over uniform in some cases but decreases it in others. On the AMD, both predictive distributions improve performance on most benchmarks; of the 10, the best AUC performance is given by iid on 5, and by Markov on the other 5.

With these empirical results we conclude that having a predictive distribution generally improved search

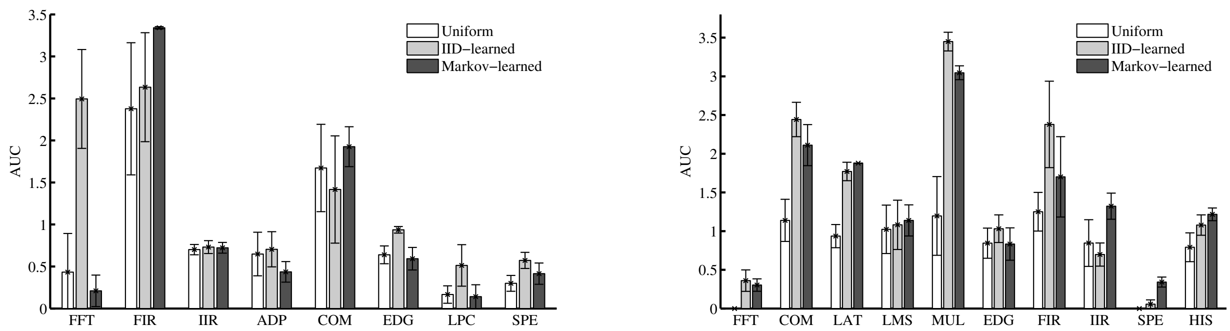

Figure 3. AUC for TI (left) and AMD (right) on the large space. The error bars denote one standard deviation of the mean. The × symbol for *fft* and *spectral* on AMD means that uniform search did not provide any improvement in performance for the number of iterations considered.

on the large space. A detailed analysis of the results shows that the improvements achieved by the predictive distribution after five iterations are at least as good as the ones obtained by uniform search after fifty iterations, which translates into an speed-up of iterative optimization of an order of magnitude. For the small space the Markov distribution provided the best performance for most benchmarks. On the large space the choice between iid and Markov was less clear cut, but both predictive distributions give improvements over uniform search.

6. Discussion

Above we have outlined the theory for learning predictive search distributions and have demonstrated that it can lead to significant improvements on the compiler optimization problems studied. We are currently collecting more data in order to be able to compare parameterized predictive distributions with nearest-neighbour methods.

Other examples of domains where there are families of optimization problems include finding the ground state of a spin glass, or the minimum balanced cut graph partitioning problem (Pelikan et al., 1999). Here families are induced by varying the edge weights in the input graph.

Acknowledgements

This work is supported under EPSRC grant GR/S71118/01 *Compilers that Learn to Optimize* and in part by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778. This publication only reflects the authors' views.

References

Caruana, R. (1997). Multitask Learning. *Machine Learning*, *28(1)*, 41–75.

de Bonet, J., Isbell, C., & Viola, P. (1997). MIMIC: Finding Optima by Estimating Probability Densities. *Advances in Neural Information Processing Systems* (p. 424). The MIT Press.

Franke, B., O'Boyle, M., Thomson, J., & Fursin, G. (2005). Probabilistic source-level optimisation of embedded programs. *Proc. LCTES'05* (pp. 78–86).

Hall, M. W., Anderson, J.-A. M., Amarasinghe, S. P., Murphy, B. R., Liao, S.-W., Bugnion, E., & Lam, M. S. (1996). Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, *29*, 84–89.

Lafferty, J., McCallum, A., & Pereira, F. (2001). Conditional random fields: Probabilistic models for segmenting and labeling sequence data. *Proc. 18th International Conference on Machine Learning* (pp. 282–289). Morgan Kaufmann, San Francisco, CA.

Langley, P. (1996). *Elements of Machine Learning*. San Francisco, USA: Morgan Kaufmann.

Larrañaga, P., & Lozano, J. A. (2001). *Estimation of distribution algorithms: A new tool for evolutionary computation*. Norwell, MA, USA: Kluwer Academic Publishers.

Lee, C. (1997). UTDSP benchmark suite. `http://www.eecg.toronto.edu/~corinna/`.

Pelikan, M., Goldberg, D. E., & Lobo, F. (1999). *A survey of optimization by building and using probabilistic models* (Technical Report IlliGAL-99018). Illinois Genetic Algorithms Laboratory.

Thrun, S., & O'Sullivan, J. (1996). Discovering structure in multiple learning tasks: The TC algorithm. *Proc. 13th International Conference on Machine Learning* (pp. 489–497). Morgan Kaufmann.