A Bayesian Method for Online Code Selection: Toward Efficient and Robust Methods of Automatic Tuning

Reiji Suda
Department of Computer Science, the University of Tokyo

1 Introduction

Software developers have to make several decisions, and sometimes experiments are needed. Since the experimental results depend on the conditions of the experiments, the decision might be inappropriate under unconsidered conditions. Software automatic tuning is a methodology for solving this problem: Software is equipped with functionality of doing experiments, analyzing the results, and making decisions, in order to adapt to its environments, hopefully even to ones unknown to the developers. The concept of automatic tuning attracts much attention after the celebrated successes of ATLAS[1], FFTW[2] and others.

This paper discusses design of experiments for automatic tuning. One of the simplest designs is evaluating every possible configuration and choosing the best one. Then a high performance configuration will be chosen, but the experiments are very costly and sometimes impractical. In order to reduce the search space effectively, some prior knowledge is usually introduced, e.g., removing configurations that are unlikely to be optimum from the set of the candidates, or introducing a model (mathematical expression) that approximates the effects of the parameters. Vuduc et al.[3] proposed a very interesting alternative based solely on statistical analysis, which is complementary to the above approach. This paper does assume prior knowledge.

Prior knowledge may be imprecise, and then the obtained solution may be suboptimal. Thus one should solve trade-off between the costs of experiments and the quality of the results. If the number of experiments is limited severely, then results of high quality cannot be expected. Still it is desirous to obtain results of a certain degree of quality. Let us call this property efficiency. On the contrary, when it is allowed to make a large number of experiments, then results of high quality must be attained. If the number of experiments approaches to the infinity, the solution should approach the optimum, even if the prior knowledge is imprecise. Let us call this property robustness (against the prior knowledge). The aim of the research discussed in this paper is to develop efficient and robust methodologies of experiments for software automatic tuning.

This paper proposes to use Bayesian methods for software automatic tuning, which naturally leads efficient and robust design of experiments. Section 2 proposes a method of sequential experimental design, and section 3 applies it to a problem of choosing a matrix multiplication routine from several unrolled codes. The proposed method is shown to be efficient and robust in comparison to some other methods.

2 A Bayesian method for automatic tuning

2.1 The problem

This paper considers a problem of online code selection. Suppose the following situation. There is an application program which calls a library function iteratively, and there are several implementations (let us call them candidates) of the library routine, all are equivalent in the functionality and different only in their speeds. It is unknown which of the candidates is the
fastest a priori, and we want to evaluate the candidates online in the course of iterations. Note that the experiments are also valid calls of the library function, and the total number of library calls is constant. The objective is to minimize the total execution time. This problem is known as “multi-armed bandit problem”[4].

2.2 Variance of execution times

First the variance of the execution times is analyzed. Sometimes the execution times of a computation were evaluated only once or a few times, but it is not enough. Figure 1 plots the flops performance of a simple subroutine of matrix-matrix multiply on a machine with Xeon 2.4 GHz. The horizontal axis shows the size $N$ of the matrices (they are all square), the vertical axis gives the flops performance, and eighty experimental results are plotted for each $N$. If the execution times were the same for all trials, then only one dot would be shown for each $N$. But in reality they look like line segments, and thus appreciable variance of the execution times is observed. The author did the same set of experiments on more than ten machines, and observed non-zero variances on every machine.

Figure 2 plots the data standardized as $\xi_{Nk} = (x_{Nk} - \bar{x}_N)/\sigma_N$ and sorted, where $x_{Nk}$ is the $k$th result of size $N$, $\bar{x}_N$ and $\sigma_N$ are sample mean and standard deviation for size $N$, respectively. By plotting data with the standardized times as $x$-coordinates and the rank (fractional rank against the total number of samples) as $y$-coordinates, we can obtain a graph similar to the cumulative distribution function. As is seen in Figure 2, it resembles the cumulative distribution function of the standard normal distribution, which is shown in the same figure. These data imply needs of statistical treatments in performance evaluation.
2.3 Bayesian data analysis

In this paper some prior knowledge about the execution time of the candidates is assumed. Such knowledge can be conveniently treated in the Bayesian framework. For example, if we expect the mean $\mu$ of the execution time $y$ is likely to be within $\mu_0 \pm \tau$, then the prior distribution for $\mu$ can be normal distribution with mean $\mu_0$ and variance $\tau^2$, $\mu \sim N(\mu_0, \tau^2)$. After observations of $y$, the posterior distribution $p(\mu|y)$ for $\mu$ is obtained by means of the Bayes formula

$$p(\mu|y) \propto p(y|\mu)p(\mu),$$

where $p(y|\mu)$ is the assumed distribution of $y$ given $\mu$. Actually in this paper the conjugate prior for normal distribution$^5$ is used:

$$p(y|\mu, \sigma^2) \sim N(\mu, \sigma^2),$$
$$p(\mu|\sigma^2) \sim N(\mu_0, \sigma^2/\kappa_0),$$
$$p(\sigma^2) \sim \text{Inv}-\chi^2(\nu_0, \sigma^2_0).$$

After observing $(y_1, y_2, \cdots, y_n)$, the posterior distribution of $\mu$ is Student’s $t$-distribution

$$p(\mu|y_1, y_2, \cdots, y_n) \sim t_{\nu_n}(\mu_n, \sigma^2_n/\kappa_n)$$

where $\kappa_n = \kappa_0 + n$, $\nu_n = \nu_0 + n$,

$$\mu_n = \frac{\kappa_0}{\kappa_0 + n} \mu_0 + \frac{n}{\kappa_0 + n} \bar{y},$$

$$\nu_n \sigma^2_n = \nu_0 \sigma^2_0 + \sum_{i=1}^{n} (y_i - \bar{y})^2 + \frac{\kappa_0 \nu_0}{\kappa_0 + n} (\bar{y} - \mu_0)^2,$$

and $\bar{y} = (1/n) \sum y_i$. Here note that the Bayesian estimation of the mean $\mu_n$ is a weighted average of the prior mean $\mu_0$ and the sample mean $\bar{y}$. Their weights are $\kappa_0$ and $n$, and thus it
can be regarded that the prior mean is worth experimental data of $\kappa_0$ trials. We have

$$E(\mu|y_1, y_2, \cdots, y_n) = \mu_n,$$

$$\text{Var}(\mu|y_1, y_2, \cdots, y_n) = \frac{\nu_n - \sigma_n^2}{\nu_n - 2\kappa_n},$$

where the latter implies gradual decrease of uncertainty of $\mu$. The posterior predictive distribution is also $t$-distribution

$$p(y|y_1, y_2, \cdots, y_n) \sim t_{\nu_n}(\mu_n, \sigma_n^2\kappa_n+1/\kappa_n).$$

More about Bayesian data analysis can be found in a standard textbook such as [5].

2.4 A sequential experimental design

The optimum solution of the bandit problem in the Bayesian framework can be obtained as follows[4]. First consider the last iteration. The expected execution time of the last iteration with choosing $i$th candidate is

$$w_i^{(1)} = \mu_i^{(1)},$$

where $\mu_i^{(1)}$ is the expected execution time of the $i$th candidate in the knowledge just before the last iteration. The optimum choice is the candidate which gives the minimum expected execution time.

Next consider the $k$th last iteration. The expected execution time of the last $k$ iterations with choosing $i$th candidate for the $k$th last iteration is

$$w_i^{(k)} = \mu_i^{(k)} + E(\min_j\{w_j^{(k-1)}(y_i)\}),$$

where $\mu_i^{(k)}$ is the expected execution time before the $k$th last iteration, and $w_j^{(k-1)}(y_i)$ is recursively defined with an assumed observation of the execution time $y_i$ at the $k$th last iteration. The optimum choice for the $k$th last iteration is to use the $i$th candidate where $i$ attains the minimum of $w_i^{(k)}$.

This solution is known for decades[4], but the complexity of computing the optimum solution is explosively high. So we need suboptimal but efficient and robust solution. The author proposes to use an approximation

$$w_i^{(k)} = \mu_i^{(k)} + (k-1)E(\min\{\mu_i^{(k-1)}, \mu_{\min'}^{(k-1)}\})$$

(1)

in place of $w_i^{(k)}$, where $\mu_{\min'}^{(k)}$ is the minimum over the candidate set except $i$. This cost estimate can be read as “try $i$ once, and use the faster one from $i$ and min’ for the remaining iterations.”

Remark the following properties of the resulting design of experiments, which are common with the optimum design.

(i) Note that $E(\min\{\mu_i^{(k-1)}, \mu_{\min'}^{(k-1)}\})$ becomes smaller if $y_i$ takes smaller values than $\mu_{\min'}^{(k)}$ with higher probability. Thus even if $\mu_i^{(k)}$ is larger than the minimum to a certain degree, $i$ can be chosen if its uncertainty is high.

(ii) A candidate with longer expected execution time and higher uncertainty is more likely to be chosen when the number of the remaining iterations is more. This is because of the coefficient $k - 1$ at the second term of (1).

(iii) Candidate $i$ is never chosen if there is a candidate $j$ which has shorter expected execution time and higher uncertainty than $i$.

Note that the property (ii) implies robustness: If the number of iterations is large enough, then the candidates of high uncertainty are tried more, and inference of higher precision will be drawn.
3 An application to selection from unrolled codes

3.1 The problem

This section reports an application of the proposed method. The task is multiplication of two square matrices, and the candidates are a set of unrolled codes. The unrolled codes are generated by ABCLibScript[6] developed by Katagiri et al. 576 distinct codes are generated in the range of

\[(u_i, u_j) \in [1, 128] \times [1, 1] \cup [1, 64] \times [1, 2] \cup [1, 32] \times [1, 4] \cup [1, 16] \times [1, 8] \cup [1, 8] \times [1, 16] \cup [1, 4] \times [1, 32] \cup [1, 2] \times [1, 64] \cup [1, 1] \times [1, 128],\]

where \(u_i\) and \(u_j\) denotes the number of times of unrolling for the outer two loops. Figure 3 shows the loop body that ABCLibScript outputs for (2, 2) unrolling.

3.2 Performance model

One can compute the number of executions of the instructions of each line of the code of Figure 3. For example, once for lines 1 and 2, \(N/u_i\) times for lines 3–5, and \((N/u_i)(N/u_j)\) times for lines 6–10, where \((N/u_i)\) means quotient of integer division. Aggregating the loads of elements of A of lines 7–10, the number of loads there is \((N/u_i)(N/u_j)u_iu_j\), which is not always equal to \(N^2\), because \(u_i\) and \(u_j\) may not divide \(N\).

By classifying the instructions in terms of the number of executions computed in the above mentioned way, 15 classes are obtained. Assuming that the set of instructions in each class takes a constant time, we get a linear model of the execution time

\[t(N, u_i, u_j) \approx \sum_{k=1}^{15} a_k(N, u_i, u_j)x_k, \quad (2)\]

where \(a_k(N, u_i, u_j)\) denotes the number of executions computed in the above mentioned way, and \(x_k\) are coefficients (to be estimated). Note that a similar model can be readily constructed for a code generated by this kind of loop transform.

For \(u_i = u_j = 1\), the above linear model is reduced to a third order polynomial \(t(N, 1, 1) \approx x_0 + x_1N + x_2N^2 + x_3N^3\), but it is clear that such a model cannot explain the irregular behavior shown in Figure 1. So in the following discussion, the coefficients \(x_k\) are estimated for each \(N\), independently. Only 7 out of 15 \(a_k\) are linearly independent for a fixed \(N\).

The above model does not consider register allocation, spill codes, and compiler optimizations. As is reported below, it works well on some machines and ill on others. Please remark that the author is not recommending to give up on constructing a performance model for \(N\), and not claiming the above simple performance model is better than some others. The aim is to illustrate the utility of an imprecise performance model for efficient and robust automatic tuning. Actually the author found no machine on which the above model gives the optimum code for all matrix sizes.

3.3 Priors for Bayesian model

In order to apply the data analysis explained in section 2.3, four values \(\kappa_0, \mu_0, \nu_0, \sigma^2_0\) are required as prior knowledge. They were determined as follows.

As a preprocessing, the variance \(\sigma^2\) of the execution times are sampled. Then \(\log \sigma^2\) is approximated by piecewise linear function of \(\log N\). The degree of freedom \(\nu_0\) are estimated with the method of moments, and then an average of \(\log(\nu_0 - 4)\) is taken. \(\sigma^2_0\) is computed as \(\sigma^2((\nu_0 - 2)/\nu_0)\). They are constant for each \(N\).
Figure 3: Loop body for $N \times N$ matrix multiply with (2,2) unrolling, generated by ABCLib-Script
The linear model (2) determines $\mu_0$. At least 7 experimental results are needed to estimate the coefficients $x_k$. Thus the first 7 iterations are used to obtain the first estimates of $x_k$, which are updated by the least squares method on further observations.

Last, $\kappa_0$ is a constant, and its value is chosen so to minimize the total execution time.

The weakest point of the work in this paper is on these priors. These methods are ad hoc and not proved to be robust. We need much data to obtain those priors, but those data are not utilized in online selection. These problems come from the lack of explicit scheme of offline experiments to obtain data needed to determine the priors. This is one of the most important topics of our future research.

3.4 Experimental evaluations

The execution times of the unrolled codes are collected by ABCLibScript. At least ten samples are collected for each $N$ and $(u_i, u_j)$ on each machine. The total number of data is more than a million per machine, but it is still far from enough for the following experiments. So synthetic data are generated from the experimental data according to the standard Bayesian method with noninformative prior, and they are added to and shuffled with the genuinely experimental data. The same sequence of data thus generated is applied to the following methods.

- Method $M_0^1$ has two phases. In the first phase it tries all the candidates once. The order of the candidates is random. In the second phase it chooses the candidate that was fastest in the first phase. Thus each candidate is evaluated only once and that information is not updated in the second phase.

- Method $M_1$ is an improvement of $M_0^1$. The first phase is the same as $M_0^1$, but in the second phase it chooses the candidate that was fastest in the preceding executions. The average execution times are updated with the observations in the execution phase.

- In method $M_2^0$, the first seven iterations are used to obtain the initial estimate of the parameters of the linear model (2). After that, it chooses the candidate with the minimum estimated execution time in the linear model. The coefficients of the linear model are updated with the newest timing data.

- Method $M_2^2$ is an improvement of $M_2^0$. The first seven iterations are the same as $M_2^0$. After that, it chooses the candidate with the minimum estimated execution time in the linear model, but for the candidates already run, the estimated execution times are replaced by the observed averages.

- Method $M_3$ is the proposed method. (In the following experiments, the $t$-distribution is approximated by a normal distribution with the same mean and variance, for ease of computations.)

Two values are computed for evaluation purposes. One is the relative regret (denoted by $rr$), which is

$$rr = (T_M - T_{opt})/T_{opt},$$

where $T_M$ is the total execution time with method $M$, and $T_{opt}$ is the average execution time of the fastest code times the number of iterations. The best possible value for $rr$ is 0, which is attained when we know the fastest candidate a priori. In practice we have to make experiments to find out which candidate is the fastest one, and thus $rr$ represents how efficient are the experiments to gain information for that purpose. Here, the computational times for the tuning mechanism are not included. Another is denoted by $cr$, which is defined as

$$cr = (t_M - t_{opt})/t_{opt},$$
where $t_M$ is the average execution time of the most frequently chosen candidate by method $M$, and $t_{\text{opt}}$ is that of the fastest one. Again the best possible value for $cr$ is 0, and for a robust method of experiments it should approach 0 if the number of iterations grows infinitely.

Table 1: Evaluations on a Xeon

<table>
<thead>
<tr>
<th></th>
<th>$M_0^M$</th>
<th>$M_1$</th>
<th>$M_2^M$</th>
<th>$M_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_0$</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>$rr$</td>
<td>15.8</td>
<td>15.8</td>
<td>2.08</td>
<td>2.04</td>
<td>2.34</td>
</tr>
<tr>
<td>$cr$</td>
<td>—</td>
<td>0.26</td>
<td>0.20</td>
<td>0.12</td>
<td>0.11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$M_1^M$</th>
<th>$M_1$</th>
<th>$M_2^M$</th>
<th>$M_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_0$</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1.0</td>
<td>3.0</td>
</tr>
<tr>
<td>$rr$</td>
<td>10.2</td>
<td>9.95</td>
<td>0.45</td>
<td>0.39</td>
<td>0.42</td>
</tr>
<tr>
<td>$cr$</td>
<td>0.6</td>
<td>0.008</td>
<td>0.27</td>
<td>0.20</td>
<td>0.03</td>
</tr>
</tbody>
</table>

Table 2: Evaluations on a Core2Duo

<table>
<thead>
<tr>
<th></th>
<th>$M_0^M$</th>
<th>$M_1$</th>
<th>$M_2^M$</th>
<th>$M_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_0$</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>3e-6</td>
</tr>
<tr>
<td>$rr$</td>
<td>1.754</td>
<td>1.754</td>
<td>1.520</td>
<td>1.405</td>
<td>1.38</td>
</tr>
<tr>
<td>$cr$</td>
<td>—</td>
<td>1.478</td>
<td>1.35</td>
<td>0.49</td>
<td>0.70</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>$M_1^M$</th>
<th>$M_1$</th>
<th>$M_2^M$</th>
<th>$M_2$</th>
<th>$M_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\kappa_0$</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>3e-6</td>
</tr>
<tr>
<td>$rr$</td>
<td>1.02</td>
<td>1.006</td>
<td>1.483</td>
<td>1.358</td>
<td>0.764</td>
</tr>
<tr>
<td>$cr$</td>
<td>0.04</td>
<td>0.0008</td>
<td>1.478</td>
<td>1.353</td>
<td>0.0003</td>
</tr>
</tbody>
</table>

Table 1 shows evaluations on a 3.8 GHz single-core Xeon machine, compiled by mpif77 -O3 with backend g77, measured by MPI_Wtime() of Lam MPI (because it is the default method of measuring execution times at ABCLibScript even in the sequential cases), and averages are taken for $8 \leq N \leq 256$. Figures are averages over five runs with different seeds of random numbers, and the numbers of digits roughly imply the standard deviations among the runs. The best $\kappa_0$ was 3.0 to 10.0, which can be understood that the linear model is worth several observations. No data of $cr$ is given for $M_1$ with 100 iterations, because it is still in the first phase. For 1000 iterations, $M_3$ attains much less regret than $M_1$ (efficiency is improved), but $M_1$ chose a better candidate. In comparison to $M_2$, the relative regret of $M_3$ is not improved much (efficiency is similar), but the most frequently chosen candidate is much better (robustness is improved). Also note that $M_0^M$ is much less robust than $M_1$, which warns us not to evaluate the speed of a code by a single observation. Similarly $M_2^M$ is somewhat worse than $M_2$, which implies the linear model does not always identify the fastest code.

Table 2 shows results on a Core2Duo 2.33 GHz. Compiler is g77 -O3, and execution times are measured by gettimeofday(). On this machine $M_2$ gives worse regret than $M_1$, which implies bad fit of the linear model, and it is consistent with very small values of best $\kappa_0$. In spite of the poor fit of the linear model, $M_3$ gives a better regret than $M_1$. Note that with a smaller
$\kappa_0$, $M_3$ can choose better candidates than $M_1$, which suggests that $M_1$ is not necessarily more robust than $M_3$. Again $M_1^0$ and $M_2^0$ were worse than $M_1$ and $M_2$, respectively.

Figure 4 shows the relative regret ($rr$) values on several machines for 1000 iterations. The $\kappa_0$ is optimized on each machine. The order is sorted by the $rr$ performance of $M_2$. First note that $M_3$ gives the best regrets for all machines, from the leftmost Pentium M, for which the linear model has error as small as 1%, to the rightmost Core2Duo, for which the linear model fits very badly, as reported above. The best $\kappa_0$, not shown here, was larger when $M_1$ performs worse, and smaller when $M_2$ performs worse. Thus $\kappa_0$ solves the trade-off between the observations and the model in $M_3$. The two plots for Core2Duo are of the same machine, and the second from the left is compiled by g77 version 3.3.5, and the right most is by g77 version 3.4.6, both with -O3.

In many cases, the computation time for each Bayesian inference amounts to a matrix-matrix multiplication of size $N \approx 30$ to 80, as of observed execution times, in the cases of 1000 iterations. The computational costs depend on the number of candidates already run and thus the number of iterations. The computational costs of the other methods ($M_1$, $M_2$, etc.) are much less, so taking the controlling costs in considerations, $M_3$ gives better performance only for large matrices.

Consider the Power5 machine in Figure 4 where our method performs relatively well. Our method $M_3$ gives relative regret 0.19 while $M_2$ gives 0.47, and thus the difference is 0.28. The control costs of $M_3$ and $M_2$ per iteration were 3.7e-4 seconds and 4.8e-5 seconds, respectively. So $M_3$ prevails over $M_2$ only if a matrix multiplication takes $(3.7 \times 10^{-4} - 4.8 \times 10^{-5})/0.28 \approx 1.1 \times 10^{-3}$ or more seconds. This is attained when $N \approx 130$ or larger. For the Xeon 2.4 GHz machine in Figure 4, the difference of the relative regrets is only 0.003, and $N \approx 420$ is required for $M_3$ to defeat $M_2$.

There are several methods to speedup computations of $M_3$ which are not implemented here. For example, the coefficients of the linear model can be updated using the downdating technique of the QR decomposition. Many integrations to get expectations can be avoided by exploiting the property (iii) in section 2.4. Search for the next candidate to execute can be boosted by using a heap data structure. More importantly, we should consider group experimental design, where the choices for the next several experiments are determined at once, rather than the strict
sequential designs discussed in this paper. There is a trade-off between the computational costs and the performance in the automatic performance tuning in general, and it is of course an important problem. However, the main purpose of the above experiments is to demonstrate utility of Bayesian statistics and methods in automatic performance tuning, and matrix multiplication is not our final target but just a sample of applications.

4 Final remarks

This paper has proposed a Bayesian method of online code selection, and demonstrated its efficiency and robustness through application to online selection of unrolled codes of matrix-matrix multiplication.

This study is still in a preliminary phase. Even for the case shown here, there remain many unsolved problems: How can one search for the best $\kappa_0$? It possibly depends on $N$ and the number of iterations. Is it practical to store data for every choice of $N$, $u_i$ and $u_j$? What if the times for the tuning mechanism are included? Was the normal/inverse chi square model appropriate for this purpose? Also there will be many other competitive methods. In spite of those problems, the author expects that this paper has succeeded to illustrate that the Bayesian framework can neatly treat uncertainty in automatic performance tuning.

Acknowledgements

The author presents sincere gratitude to Professor Akimichi Takemura at the University of Tokyo, who gives many helpful suggestions. The author is also very grateful to the members of the project “Research on Mathematical Core for Robust Auto-Tuning System in Information Explosion Era” and of the Automatic Tuning Research Group. The author also thanks Mr. Saito and Professor Matsumoto at the Hiroshima University for their help for using random number generator SFMT.

This research is supported by “Cyber Infrastructure for the Information-Explosion Era,” Grant-in-aid for scientific research on priority areas, MEXT Japan.

References


