

Bandit-Based Optimization on Graphs with Application to Library Performance Tuning

Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, Markus Püschel

▶ **To cite this version:**

Frédéric de Mesmay, Arpad Rimmel, Yevgen Voronenko, Markus Püschel. Bandit-Based Optimization on Graphs with Application to Library Performance Tuning. ICML, Jun 2009, Montréal, Canada. inria-00379523

HAL Id: inria-00379523
https://inria.hal.science/inria-00379523

Submitted on 28 Apr 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Bandit-Based Optimization on Graphs with Application to Library Performance Tuning

Frédéric de Mesmay[⋆] FDEMESMA@ECE.CMU.EDU
Arpad Rimmel[†] RIMMEL@LRI.FR
Yevgen Voronenko[⋆] YVORONEN@ECE.CMU.EDU
Markus Püschel[⋆] PUESCHEL@ECE.CMU.EDU

[⋆] Department of Electrical and Computer Engineering, Carnegie Mellon University, Pittsburgh, PA 15213 USA
[†] TAO (Inria), LRI, UMR 8623 (CNRS - Université Paris-Sud), 91405 Orsay, France

Abstract

The problem of choosing fast implementations for a class of recursive algorithms such as the fast Fourier transforms can be formulated as an optimization problem over the language generated by a suitably defined grammar. We propose a novel algorithm that solves this problem by reducing it to maximizing an objective function over the sinks of a directed acyclic graph. This algorithm valuates nodes using Monte-Carlo and grows a subgraph in the most promising directions by considering local maximum k-armed bandits. When used inside an adaptive linear transform library, it cuts down the search time by an order of magnitude compared to the existing algorithm. In some cases, the performance of the implementations found is also increased by up to 10% which is of considerable practical importance since it consequently improves the performance of all applications using the library.

1. Introduction and Related Work

The computing platforms available today can differ substantially in their memory hierarchies, numbers of processors, and many other microarchitectural details. Further, these details tend to change with every new generation of processors. As a consequence, code that runs fast on one platform may perform suboptimally if not poorly on another: performance cannot easily be ported. This problem particularly affects the developers of high performance libraries, who often create different implementations for each platform and whenever new platforms are released (Intel, 2008).

One way to reduce the recurring development costs is *automatic performance tuning*. The basic idea is to use feedback-driven search or learning techniques to automatically find the fastest implementation of a given functionality on a given platform among a set of possible choices. Variants arise from different possibilities of recursion for divide-and-conquer algorithms or from implementation decisions such as unrolling and parallelizing. Covered functionalities include dense (Whaley & Petitet, 2005) and sparse (Vuduc et al., 2005) linear algebra but we particularly focus on fast Fourier transforms (FFT) with adaptive libraries such as FFTW (Frigo & Johnson, 2005), UHFFT (Mirković et al., 2000), or the libraries generated by Spiral (Voronenko, 2008).

In each of the above cases, the space of alternatives is extremely large, requiring efficient search methods. Various have been tried for FFTs: hill climbing, genetic algorithms, regression trees (Singer & Veloso, 2002), etc. In practice, however, the most efficient search method, denoted DP, seems to be *restricting* the search space and using dynamic programming.

However, when the search space becomes even larger (large problem sizes or more complicated libraries), DP may take very long to terminate and the very best solution might not even be in the restricted space–a waste since the library may have the ability to run faster, and also of considerable practical relevance if the fastest existing code can be improved further.

Contributions. In this paper we first abstractly formulate the performance tuning problem of adaptive divide-and-conquer type libraries as an optimization problem associated with a large acyclic formal gram-

Appearing in *Proceedings of the 26th International Conference on Machine Learning*, Montreal, Canada, 2009. Copyright 2009 by the author(s)/owner(s).

mar. Each word in the language generated by the grammar is a recursion strategy the library can perform. We then solve the problem using a novel online-search algorithm, Threshold Ascend on Graph (TAG), that exploits the inherent structure of the problem. TAG is similar to UCT (Kocsis & Szepesvi, 2006), in that it gradually builds up the graph generating the language by considering local bandit problems and by valuating the nodes with Monte-Carlo simulations. TAG differs from UCT in that it optimizes for the best single reward over a graph instead of maximizing the accumulative reward over a tree.

We implemented TAG to be composable with adaptive transforms libraries generated by Spiral (Voronenko, 2008). These libraries are vectorized and parallelized, possess a very large search space and are in many cases faster than any other existing code. We show that, compared to the dynamic programming search (typically used for these kinds of problems), TAG can dramatically reduce the performance tuning time (i.e. a good solution is found much faster) and in some cases finds a better solution (and hence improves the library performance). Therefore, it is a suitable optimization technique in this domain. As an additional benefit, TAG is an anytime algorithm.

2. Formal Problem Statement

Below, we formally state the problem considered in this paper. Later, we will show that automatic tuning in the considered transform libraries is an instantiation of this problem.

Problem 1 Given is an acyclic formal grammar $F = (T, N, P, S)$ with T the set of terminals, N the set of nonterminals, P the set of production rules or simply rules, and S the starting symbol. $\mathcal{L}(F)$ is the associated language and f is an objective function from $\mathcal{L}(F)$ into the positive reals $\mathbb{R}^+$. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in \mathcal{L}(F)} f(w).$$

F has an associated *derivation graph* $G = G(F)$ which is directed, acyclic and weakly connected as shown in Figure 1: S is the root, the directed edges (arrows) correspond to applications of rules in P, the nodes are partially derived words in the language, and the sinks (outdegree = 0) are precisely the elements of $\mathcal{L}(F)$. Hence we can reduce Problem 1 to:

Problem 2 Given a weakly connected, acyclic, directed graph $G = (V, E)$ and an objective function f (as above) on the sinks $S(G)$ of G. We want to compute

$$w_{\text{best}} = \operatorname*{argmax}_{w \in S(G)} f(w).$$

$$\begin{aligned} T &= \{\text{a, b, ac}\} \\ N &= \{\text{S, A, B}\} \\ P &= \{\text{S} \to \text{AB}, \\ &\quad \text{A} \to \text{a}, \\ &\quad \text{B} \to \text{b}, \\ &\quad \text{aB} \to \text{ac}\} \end{aligned}$$

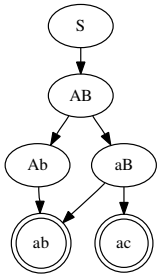


Figure 1. Formal grammar $F = (T, N, P, S)$ (*left*) and associated derivation graph $G(F)$ (*right*). S, A, B are nonterminals and a, b, c are terminals. The graph has two sinks (double circled), i.e., the language $\mathcal{L}(F)$ has two elements.

We assume the graph $G(F)$ to be large such that it is impossible to generate and evaluate all sinks in a reasonable time. Our goal is an algorithm that finds a "very good" sink with a small number of evaluations.

3. The TAG Algorithm

TAG is an *anytime* algorithm that determines an approximate solution to Problem 2. Due to the size of the graph it is not meant to run until completion, in which case it would be equivalent to an exhaustive search.

TAG finds solutions by incrementally growing and exploiting the subgraph $\hat{G} = (\hat{V}, \hat{E})$ of $G = (V, E)$: $\hat{V} \subset V$, $\hat{E} \subset E$, starting with $\hat{G} = (\{S\}, \{\})$. Evaluations are used to direct the growth of $\hat{G}$ towards the expected bests sinks.

Assume the current subgraph is $\hat{G}$. Then TAG proceeds in three high level steps visualized in Figure 2:

1. *Descend:* G is traversed starting at its root. Each choice along the way is solved by a *bandit algorithm*. The descent stops when it uses an arrow e that is not in $\hat{E}$.
2. *Evaluate:* If e is incident with a vertex not in $\hat{V}$, this vertex is evaluated using a Monte-Carlo expansion.
3. *Backpropagate:* The evaluation is stored in all ancestors of the vertex.

We proceed with describing the three steps in detail, describe the pseudocode and conclude the section with a presentation of related algorithms.

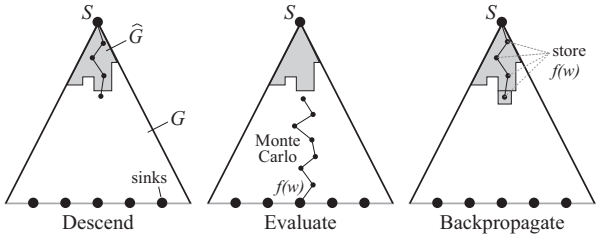


Figure 2. Visualization of the three main steps in TAG. Note that $\hat{G}$ (shaded area) and G are not trees (e.g., see Figure 1).

3.1. Descend

The goal of the *descent* step is to select the next edge to add to the subgraph $\hat{G}$. It is chosen so that $\hat{G}$ grows towards the sinks that present the best expected rewards. Starting from the root S, the most promising path is layed out by successively chosing the most promising outgoing edges. Each choice is solved using a bandit algorithm that we describe first.

Background: Max k-Armed Bandit Problem. The *maximum* k-armed bandit problem considers a slot machine with k arms, each one having a different pay-out distribution (Figure 3). The goal is to maximize the single best reward obtainable over $\bar{n}$ trials (Cicirello & Smith, 2005). Formally, if each arm has distribution D_i and $R_j(D_i)$ denotes the j-th reward obtained on arm i, the goal is to solve

$$\max_{\sum_{i=1}^{k} \bar{n}_i = \bar{n}} \; \max_{1 \le i \le k} \; \max_{1 \le j \le \bar{n}_i} R_j(D_i).$$

In this paper, we use a variation: an anytime version of the problem where the total number of pull $\bar{n}$ is not known in advance. Only the n previous pulls and their associated rewards are known.

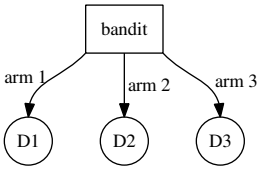


Figure 3. A 3-armed bandit. The choice of the arm i leads to a realization of the distribution D_i.

Streeter & Smith (2006) solve the problem using *Threshold Ascend*, an algorithm that makes no assumptions on the form of the distributions. Using their notations, we present here a straightforward adaptation to the anytime variation.

The main idea of the algorithm is to track only the s best rewards and the arms they are coming from. Let s_i be the number of such rewards among the n_i rewards received by the arm i. Also, let δ be a positive real parameter. The algorithm advises to pull the arm i_{best} given by

$$i_{\text{best}} = \operatorname*{argmax}_{1 \le i \le k} h(s_i, n_i),$$

$$\text{with } h(s_i, n_i) = \begin{cases} \frac{s_i + \alpha + \sqrt{2 s_i \alpha + \alpha^2}}{n_i}, & \text{if } n_i > 0 \\ \infty, & \text{else} \end{cases}$$

and $\alpha = \ln(2nk/\delta)$.

Descend. The graph descent is responsible for incrementally building the subgraph $\hat{G} \subset G$, initially restricted to the root. The purpose of the descent is to select an arrow in $E \setminus \hat{E}$ that leads towards an expected good sink. It does so by tracing a path starting from the root and considering each successor choice as a max k-armed bandit problem (Figure 4). For now, assume that a table of positive real rewards $R(v)$ has been maintained for each vertex $v \in \hat{V}$.

Let v denote the current vertex in the descent. Starting from v, there are multiple ways to continue the path since it can follow any of the arrows originating from v (we denote these with $E(v)$). The arrows in $E(v)$ that are also in $\hat{E}(v)$ lead to vertices of $\hat{V}$ corresponding to "arms" that have already been played (they have previous rewards attached to them). The other arrows lead to arms that have never been played. The bandit algorithm discussed above decides which arrow to follow, which has to be one that was not followed before if such an arrow exists (due to the infinite weight in $h(s_i, n_i)$). If the arrow belongs to $\hat{E}(v)$ and the successor is not a sink, the successor becomes the new descent vertex and the descent continues. If not, the descent ends.

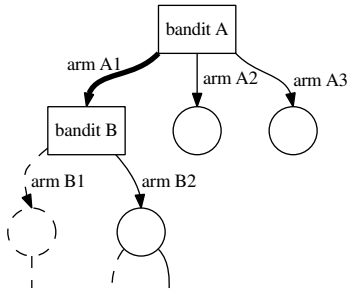


Figure 4. The descent in the graph is done as a cascade of multi-armed bandits. Solid arrows, circles and boxes are in $\hat{G}$, dashed arrows and circles are in $G \setminus \hat{G}$. For bandit A all arms had been played before, and A1 is chosen based on the stored rewards. Bandit B will now choose B1, since it is the only arrow not played before.

3.2. Evaluate

Assume the descent ended on an arrow pointing to a vertex v that is not part of $\hat{V}$. The arrow and vertex are then immediately added to $\hat{G}$ and v is *evaluated*.

If v is a sink of G, then $f(v)$ can be directly computed. Otherwise a path to a sink of G is chosen by "Monte-Carlo," which means in each step a (uniformly drawn) random choice is made until a sink w is obtained. The evaluation $f(w)$ gives a value for v.

Also, if the evaluation is better than $f(w_{\text{best}})$, the current best sink is replaced.

3.3. Backpropagate

After v has been evaluated, the reward is added to its reward list $R(v)$ and to the reward lists of all its ancestors.

Note that if the descent ended on an arrow pointing to a vertex v that is already a part of $\hat{V}$, we just discovered a new way to connect to an already evaluated vertex. In this case, we add the new arc to $\hat{E}$ and propagate the rewards of v only to the vertices that would not be ancestors of v without the new arrow (since the other ancestors already have these rewards).

3.4. Pseudocode and Remark

Pseudocode. Algorithm 1, the pseudocode of TAG, summarizes the previous discussion. After initialization, the graph $\hat{G} = (\hat{V}, \hat{E})$ is grown one arc at a time until the user signifies an interruption. The vertex pointed by an arrow e is denoted *head(e)*. `BANDIT` refers to the *Treshold Ascend* algorithm summarized in subsection 3.1. `RANDOM` refers to an uniform draw.

Remark. Practically, if the objective function is deterministic, it is useless to evaluate a sink twice. It is therefore possible to modify the algorithm to guarantee that it never returns in a branch where choices have been exhausted. While we implemented this version we do not present it due to lack of space.

3.5. Related Algorithms

The "classic" multi-armed bandit problem involves maximizing the expected sum of rewards of multiple slot machines with different pay-out distributions. Many proposed algorithms are based on optimism in front of uncertainty: the score of a slot machine is its current estimated value plus a term that grows with the uncertainty. For instance, *Upper Confidence Bounds* (UCB) proposes a term in $\sqrt{\log(n)/n_i}$ (Lai & Robbins, 1985; Auer et al., 2002). Using the Chernoff inequality includes the variance inside the term and significantly improves the performance (Audibert et al., 2007).

Algorithm 1 TAG

```
Ĝ ← S
w_best ← ∅
R(V̂) ← ∅
while not interrupted do
    e ← BANDIT(E(S))                                   Descend
    while e ∈ Ê & E(head(e)) ≠ ∅ do
        e ← BANDIT(E(head(e)))
    end while
    v ← head(e)
    if v ∉ Ĝ or e ∈ Ĝ then
        add v and e to Ĝ
        e ← RANDOM(E(v))                               Evaluate
        while E(head(e)) ≠ ∅ do
            e ← RANDOM(E(head(e)))
        end while
        w ← head(e)
        if f(w) > f(w_best) then
            w_best ← w
        end if
        r ← f(w)
        add r to R(v)                                  Backpropagate
        for a ancestor of v in Ĝ do
            add r to R(a)
        end for
    else
        for a ancestor of v in Ĝ do
            mark a
        end for
        add e to Ĝ
        for a ancestor of v in Ĝ do
            if a is marked then
                unmark a
            else
                add all R(v) to R(a)
            end if
        end for
    end if
end while
return w_best
```

Other Monte-Carlo based algorithms could be used to perform an optimization on a leaves-evaluated graph but, besides the fact that they usually are designed for trees, they differ from TAG by biasing the subgraph towards zones that are good on average which can be distinct from zones that are likely to contain maximums. Guillaume Chaslot et al. proposed an algorithm derived from the central limit theorem that gives good result on the production management problem (2006). UCT uses UCB as a local branch selector (Kocsis & Szepesvi, 2006) and is particularly efficient with huge search-spaces: it is at the origin of the current best computer Go players (Coulom, 2006; Wang & Gelly, 2007; Gelly & Silver, 2007).

4. Application: Performance Tuning in Adaptive Libraries

Our target application for TAG is the automatic performance tuning in adaptive libraries based on divide-and-conquer algorithms with inherent degrees of freedom. Specifically, we implemented TAG to operate as a search strategy in the adaptive general-size linear transform libraries generated by Spiral (Voronenko et al., 2009).

We first give brief background on transforms, transform algorithms, their implementations, and the notion of an adaptive library. Then we discuss the need for search and finally match the performance tuning problem to Problem 1, which shows that TAG is applicable.

4.1. Background: Linear Transforms

Transforms. A linear transform is a matrix-vector product $y = Mx$, where x is the input vector, y the output vector, and M the fixed transform matrix. We focus on the discrete Fourier transform (DFT) defined as

$$\mathbf{DFT}_n = [e^{-2\pi ik\ell/n}]_{0\le k,\ell<n}, \quad i = \sqrt{-1}.$$

Naïve computation of the matrix-vector product incurs $O(n^2)$ operations, however, fast, $O(n\log(n))$, algorithms, which exploit the particular structure of matrix M, exist for many transforms including the DFT.

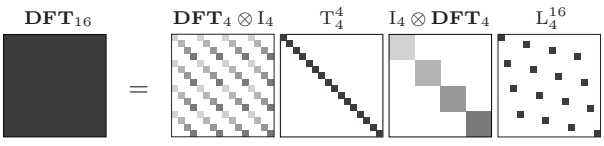
Fast algorithms. One way of writing transform algorithms is as sparse factorizations of the transform matrix. For example, the famous Cooley-Tukey fast Fourier transform (FFT) algorithm can be written as

$$\mathbf{DFT}_n = (\mathbf{DFT}_k \otimes \mathrm{I}_m)\,\mathrm{T}^n_m(\mathrm{I}_k \otimes \mathbf{DFT}_m)\,\mathrm{L}^n_k,\ n = km. \tag{1}$$

Here, I_n is the identity matrix of size n; T^n_m is a diagonal matrix and L^n_k a permutation matrix, whose precise definition is not relevant here. Finally, the tensor (or Kronecker) product $\otimes$ of two matrices is defined as

$$\mathrm{A} \otimes \mathrm{B} = [a_{k,l}\,\mathrm{B}],\ \text{where}\ \mathrm{A} = [a_{k,l}].$$

We show below a visualization of the non-zero values in the matrices for $k = m = 4$.



In both tensor products, all parts of equal gray shade constitute a single $\mathbf{DFT}_4$. We observe that all four matrices are sparse, that the computation uses a divide-and-conquer approach, and that there is a degree of freedom (choice of $k|n$). Assuming that n is a power of two[1], recursive applications of the algorithm yield $O(n\log(n))$ computations.

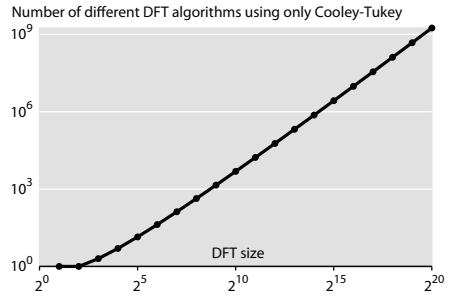


Figure 5. Number of DFT algorithms based on standard Cooley-Tukey FFT, implemented naïvely. All algorithms for a given DFT input size have roughly the same operations count.

Implementation and search space. The above FFT suggests a library implementation using a recursive function `dft`. Given the input x, the function would first permute ($t = \mathrm{L}^n_k\,x$), then call `dft` on multiple segments of x, then scale the result with the entries of T^n_m, and then call again `dft` on segments, extracted in a stride, of the result. The resulting library would have a simple call graph, as shown in Figure 6(a). Even such a simple implementation has a degree of freedom in the recursion due to the choice of k. Recursively compounded this yields an algorithm space of $\Theta(5^t/t^{3/2})$ that this library covers (see Figure 5) (Johnson & Püschel, 2000). All of these have roughly the same operations count, yet, the performance can differ widely due to cache misses and other effects.

The above implementation makes four passes through a vector of length n and has hence poor memory hierarchy performance. The performance can considerably improved as done in FFTW 2.x by replacing the explicit (and expensive) permutation L^n_k with a readdressing in the subsequent smaller DFTs. Similarly, scaling by T^n_m can be fused with the subsequent DFTs. However, this creates the need for additional functions—variants of the DFT with modified interfaces. The call graph of such a library is shown in Figure 6(b).

The situation gets considerably more complicated with state-of-the-art libraries on current off-the-shelf computers. The reason is that to get maximal performance

[1]Recursive application of equation (1) require to provide base cases for all prime factors of n. For simplicity, this paper will therefore only consider power-of-two sizes $n = 2^t$. Note that these sizes also happen to be the most important usage cases of the DFT.

the libraries need to apply several restructuring transformations to (1). In particular, the algorithm must be 1) vectorized (Franchetti et al., 2006b), to take advantage of vector instructions (e.g., SSE on x86 architectures); 2) parallelized (Franchetti et al., 2006a), to exploit multiple processor cores using threading; 3) transformed by loop optimizations for buffering (Frigo & Johnson, 2005; Voronenko, 2008); 4) allowed to load from a precomputed table the constant elements of T^n_m from (1), also called "twiddle factors".

Applying the restructuring transformations described above increases the number of different mutually recursive functions that comprise the library, and also enlarge the algorithm search space. For example the Spiral-generated DFT library with all optimizations 1–3 contains 31 different functions which form the call graph in Figure 6(c).

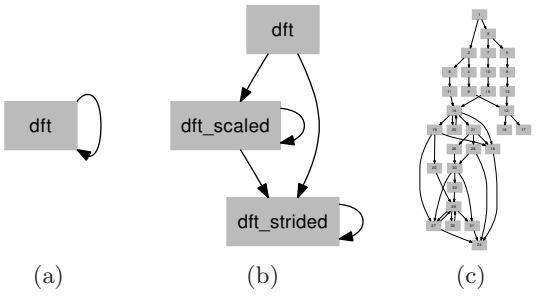


Figure 6. Call graphs of three different recursive libraries: (a) naïve, (b) optimized scalar and (c) optimized vectorized parallelized.

The above discussion holds for many other linear transforms including the discrete cosine transforms, the real discrete Fourier transform, finite impulse response filters, and the discrete wavelet transform. Further, not all algorithms decompose a transform into transforms of the same type. In this case the search space is further increased.

4.2. Adaptive Libraries and Search

Consider a recursive library as discussed above. In each recursion step, the library has a degree of freedom. As a consequence, it can compute the transform in many different ways. What makes the library adaptive is the use of online search to find a fast recursion strategy. This search is part of an initialization routine (called planner in FFTW) that takes the input size n and returns a function pointer implementing the fastest known recursion strategy. After this initial overhead, the user can now compute as many transforms of size n as desired, compensating for the overhead.

The main search strategy in FFTW, UHFFT, and Spiral generated libraries is dynamic programming (DP). It is based on the assumption that the best solution to a problem is built out of optimal solutions to subproblems. Here, this means that an algorithm's performance is independent of its context which, unfortunately, does not always hold[2]. However, in practice, DP has shown to work quite well except for very large transforms as we will see later in our benchmarks. Over these large search spaces, DP has another weakness which is that it is not an anytime algorithm: one has to wait for DP to solve all subproblems before it gives any solution. This waiting time is significant: for FFTW it can amount to days in the case of large transforms on multicore systems.

A simple anytime strategy is Monte-Carlo (MC) which, each time there is a decision to take, chooses according to a uniform distribution. At the end of the descent, it evaluates the candidate and restarts. At any point in time the user can interrupt the search to retrieve the best known candidate. Since at each step, there is an equal chance for all branches to be picked but branches are not laid out uniformly, the overall space is *not* sampled uniformly.

4.3. Applicability of TAG

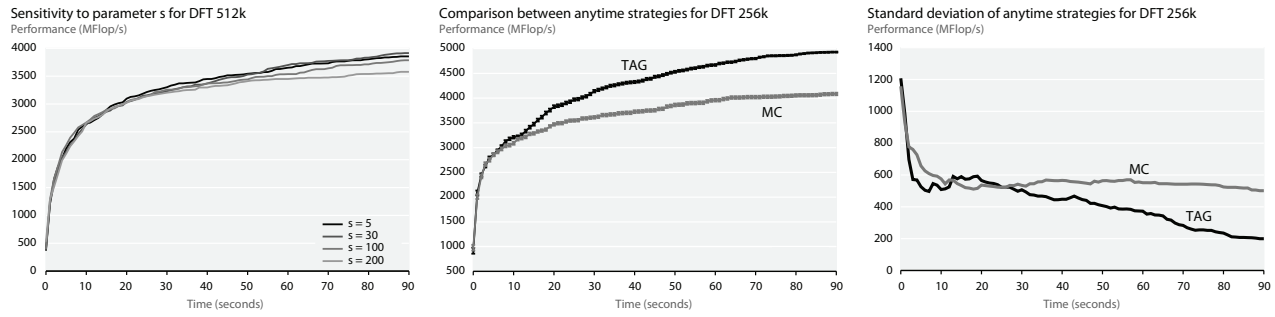
Applying TAG in the context of adaptive libraries requires to identify the grammar $G = (T, N, P, S)$ and the objective function f such that the performance optimization can be mapped to Problem 1.

The start symbol S is the transform specification as entered by the user. The terminals T are the *base cases*, the set of problems that can be directly solved by the library. The non-terminals N are the set of all non base case subproblems that could be needed to solve the problem. The production rules P breakdown a problem from N into one or more subproblems by fixing a degree of freedom. The function to maximize, f, is the performance of the implementation. The acyclicity of the grammar is guaranteed by the fact that the underlying algorithms provably finish. Note that the grammar itself changes with the problem size.

For instance, if a naïve DFT library based on Cooley-Tukey is used to compute $\mathbf{DFT}_8$, we would define

$$
\begin{aligned}
S &= \mathbf{DFT}_8 & P &= \{\mathbf{DFT}_8 \to (\mathbf{DFT}_2, \mathbf{DFT}_4),\\
T &= \mathbf{DFT}_2 & &\quad\ \mathbf{DFT}_8 \to (\mathbf{DFT}_4, \mathbf{DFT}_2),\\
N &= \{\mathbf{DFT}_8, \mathbf{DFT}_4\} & &\quad\ \mathbf{DFT}_4 \to (\mathbf{DFT}_2, \mathbf{DFT}_2)\}
\end{aligned}
$$

[2]It is fairly easy to build counter-examples: for instance, an algorithm running on one core will be slower if another core is active due to conflicts in the shared cache.



(a) (b) (c)

Figure 7. (a) Parameters for TAG are optimized on $\mathbf{DFT}_{2^{19}}$. (b) Mean performance (and standard error of the mean) for DP and Monte-Carlo on $\mathbf{DFT}_{2^{18}}$. Data is averaged over 100 runs. (c) Standard deviation on the same experiment.

5. Experiments

Experimental Setup. We evaluate our search algorithm on a complex DFT C++ library generated by Spiral from (1). The library is vectorized using intrinsics, threaded using OpenMP, and optimized as explained in Section 4.1.

We add TAG and Monte-Carlo (MC) methods to the already existing DP search infrastructure . We compile using the Intel Compiler 10.1 and benchmark on a 64-bit Linux platform using two dual core 3 GHz Intel Xeon 5160 processors.

We display performance using pseudo mega floating-point operations per second (MFlops) with the complex DFT operation count assumed to be $5n \log_2 n$ (standard practice).

Parameter tuning. We tune the parameters for TAG on a specific problem, $\mathbf{DFT}_{2^{19}}$. The sensitivity of the algorithm with variations in the s parameter of the bandit is shown on Figure 7(a). Since s is the size of the best rewards vector, a low s tweaks the bandit towards exploitation of previous good branches, while a bigger s leads to the exploration of new promising branches. We find that $\delta = 0.1$ and $s = 30$ work best and we use them for *all* following experiments.

Comparison with Monte-Carlo. We compare the performance of TAG and MC on $\mathbf{DFT}_{2^{18}}$. Figures 7(b) and 7(c) show that TAG performs better (higher mean) and more reliably (lower standard deviation) than Monte-Carlo. Note that the plots are done with respect to a fixed "wall clock" time and not to a fixed number of simulations. This is realistic in that the simpler MC algorithm performs more simulations than our more complex algorithm in the same time frame. Also it is worth remembering that, asymptotically, TAG and MC match since they both explore the full finite search space.

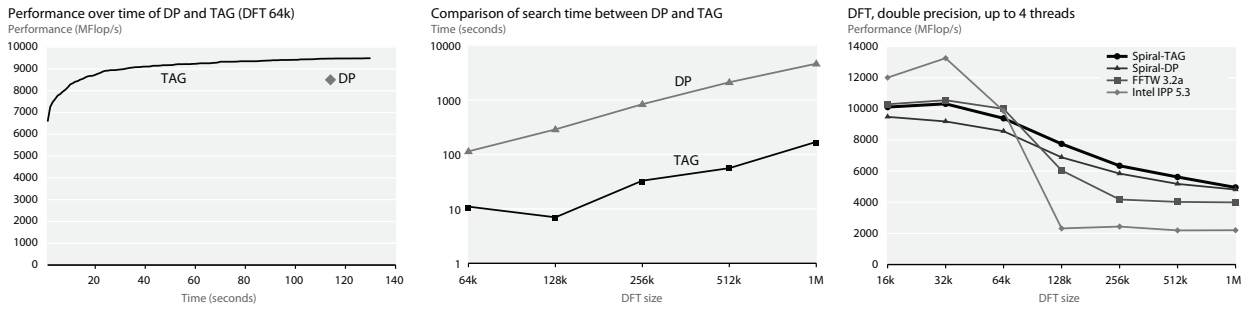
Comparison with dynamic programming. We compare TAG with DP on a single problem on 8(a). We observe that TAG quickly reaches the same performance as DP and then caps 10% above it. On Figure 8(b) we plot the time it takes for TAG to get results of the same quality as DP. We observe that TAG finds solutions of equal performance significantly faster on various DFT sizes.

Comparison with other FFTs. Figure 8(c) shows the best performance attained by the generated library (with TAG and DP) and its competitors. We compare against FFTW 3.2 alpha 2 and Intel IPP 5.3. FFTW does platform adaptation using dynamic programming. As far as we know, IPP does not use search and branches out to a specialized implementation for each platform.

6. Conclusion

In this paper we tackled the problem of optimizing an objective function over the sinks of a directed acyclic graph. We solved it using a new anytime algorithm, TAG, that grows a subgraph towards the expected best sinks. Similarly to UCT, TAG traces the most promising path by considering local bandits and valuates nodes using Monte-Carlo simulations. In our context however, the optimization problem requires to consider the *maximum* variant of the k-armed bandit problem.

Implementation inside a high-performance adaptive library for linear transforms considerably decreased the search time while providing a 10% increase in the quality of the solutions. One interesting feature of our problem setup is that evaluating "bad" nodes is much more costly than evaluating "good" ones since the objective function is the timer from the processor. In future work, we will try to modify the algorithm to take advantage of that fact.



(a) (b) (c)

Figure 8. (a)Average performance of TAG compared with DP on a single problem size. (b)Search time of TAG and DP to achieve the same performance on different libraries. (c)Comparison with different FFT libraries.

Acknowledgments

The authors are grateful to O. Teytaud for his many valuable suggestions. The authors would also like to thank M. Streeter, S. Smith and A. de Mesmay for their help with the max k-armed bandit algorithms.

This work was supported by NSF through awards 0325687, 0702386, by DARPA (DOI grant NBCH-1050009), the ARO grant W911NF0710416, and by Intel.

References

Audibert, J.-Y., Munos, R., & Szepesvári, C. (2007). Tuning bandit algorithms in stochastic environments. *ALT* (pp. 150–165). Springer.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*, 235–256.

Chaslot, G., De Jong, S., Saito, J.-T., & Uiterwijk, J. W. H. M. (2006). Monte-Carlo tree search in production management problems. *Proc. of the 18th BeNeLux Conference on AI, Namur, Belgium* (pp. 91–98).

Cicirello, V., & Smith, S. (2005). The max k-armed bandit: A new model for exploration applied to search heuristic selection. *Artificial Intelligence (AAAI-05)*.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. *Proc. of the 5th International Conference on Computers and Games*.

Franchetti, F., Voronenko, Y., & Püschel, M. (2006a). FFT program generation for shared memory: SMP and multicore. *Supercomputing (SC)*.

Franchetti, F., Voronenko, Y., & Püschel, M. (2006b). A rewriting system for the vectorization of signal transforms. *VECPAR LNCS 4395* (pp. 363–377). Springer.

Frigo, M., & Johnson, S. G. (2005). The design and implementation of FFTW3. *Proc. of the IEEE*, *93*, 216–231.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. *Proc. ICML'07* (pp. 273–280). ACM.

Intel (2008). Integrated performance primitives, User Guide.

Johnson, J., & Püschel, M. (2000). In search of the optimal Walsh-Hadamard transform. *Proc. IEEE ICASSP* (pp. 3347–3350).

Kocsis, L., & Szepesvi, C. (2006). Bandit based Monte-Carlo planning. *ECML'06 LNCS 4212* (pp. 282–293). Springer.

Lai, T., & Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, *6*, 4–22.

Mirković, D., Mahasoom, R., & Johnsson, L. (2000). An adaptive software library for fast Fourier transforms. *Proc. of the 14th Supercomputing* (pp. 215–224).

Singer, B., & Veloso, M. (2002). Learning to construct fast signal processing implementations. *Journal of Machine Learning Research, ICML 2001*, *3*, 887–919.

Streeter, M. J., & Smith, S. F. (2006). A simple distribution-free approach to the max k-armed bandit problem. *Principles and Practice of Constraint Programming (CP 2006)* (pp. 560–574).

Voronenko, Y. (2008). *Library generation for linear transforms*. Doctoral dissertation, Electrical and Computer Engineering, Carnegie Mellon University.

Voronenko, Y., de Mesmay, F., & Püschel, M. (2009). Computer generation of general size linear transform libraries. *Code Generation and Optimization (CGO)*.

Vuduc, R., Demmel, J. W., & Yelick, K. A. (2005). Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, *16*.

Wang, Y., & Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. *IEEE Symp. on Computational Intelligence and Games* (pp. 175–182).

Whaley, R. C., & Petitet, A. (2005). Minimizing development and maintenance costs in supporting persistently optimized BLAS. *Software: Practice and Experience*, *35*, 101–121.