Efficient Memoization For Approximate Function Evaluation Over Sequence Arguments

Tamal Biswas and Kenneth W. Regan

Department of CSE
University at Buffalo
Amherst, NY 14260 USA
{tamaltan,regan}@buffalo.edu

Abstract. This paper proposes a universal strategy for maintaining a database of computational results of functions f on sequence arguments $\boldsymbol{x}$, where $\boldsymbol{x}$ is sorted in non-decreasing order and $f(\boldsymbol{x})$ has greatest dependence on the first few terms of $\boldsymbol{x}$. This scenario applies also to symmetric functions f, where the partial derivatives approach zero as the corresponding component value increases. The goal is to pre-compute exact values $f(\boldsymbol{u})$ on a tight enough net of sequence arguments, so that given any other sequence $\boldsymbol{x}$, a neighboring sequence $\boldsymbol{u}$ in the net giving a close approximation can be efficiently found. Our scheme avoids pre-computing the more-numerous partial-derivative values. It employs a new data structure that combines ideas of a trie and an array implementation of a heap, representing grid values compactly in the array, yet still allowing access by a single index lookup rather than pointer jumping. We demonstrate good size/approximation performance in a natural application.

Key words. Memoization, sequences, metrics, topology, machine learning, cloud computing.

1 Introduction

In many computational tasks, we need to evaluate a function on many different arguments, in applications such as aggregation where we can tolerate approximation. Each evaluation $f(x)$ may be expensive enough to demand *memoization* of pre-computed values, creating a fine enough grid of argument-value pairs to enable approximating $f(x)$ via one or more neighboring pairs $(u, f(u))$. In this paper we limit ourselves to external memoization, here meaning building the complete grid in advance. Such applications of grids in high-dimensional real spaces $\mathbb{R}^\ell$ are well known (see history in [1]). What distinguishes this paper is a different kind of space in which the arguments are homogeneous *sequences* not just arbitrary vectors, where f and the space obey certain large-scale structural properties.

To explain our setting and ideas, consider first the natural grid strategy in $\mathbb{R}^\ell$ of employing the Taylor expansion to approximate $f(x)$ via a nearby gridpoint u:

$$f(x) = f(u) + \sum_{i=1}^{\ell}(x_i - u_i)\frac{\partial f}{\partial x_i}(u) + \frac{1}{2}\sum_{i,j}(x_i - u_i)(x_j - u_j)\frac{\partial^2 f}{\partial u_i \partial u_j} + \cdots$$

If we make the grid fine enough, and assume that f is reasonably smooth, we can ignore the terms with second and higher partials, since $(x_i - u_i)(x_j - u_j)$ is quadratically small in the grid size. Doing this still requires knowledge of the gradient of f on the gridpoints, however. Memoizing—that is, precomputing and storing—all the partials on the gridpoints might be ℓ times as expensive as memoizing the values $f(u)$. Hence, depending on the application, one may employ an approximation to the gradient or a recursive estimation of the partials.

In our setting, we are given a different kind of structure with a little more knowledge. Here we need to compute functions f on sequence arguments $\boldsymbol{x} = (x_1, x_2, x_3, \dots)$ under the following circumstances.

(a) The sequence entries and function values belong to $[0, 1]$.
(b) For all $i < j$ and $\boldsymbol{x}$, $\partial f / \partial x_i > \partial f / \partial x_j$ at $\boldsymbol{x}$.
(c) The sequences are non-decreasing, and for all i, $\partial f / \partial x_i$ becomes small as x_i approaches 1.
(d) While exact computation of $f(\boldsymbol{x})$ is expensive, moderate precision suffices, especially when there is no bias in the approximations.

Part (b) says that the initial terms have the highest influence on the result, while (c) together with (b) implies that as the sequence approaches its ceiling, terms lose their influence. Part (c) also allows us to assume all sequences have the same length ℓ, using 1.0 values as trailing padding if needed. Applications obeying (d) include calculation of means and percentiles and other aggregate statistics, as are typical for streamnig algorithms [2], and various tasks in curve fitting, machine learning, complex function evaluation, and Monte Carlo simulations (see [3]).

For further intuition, note that this setting applies to any function f that is symmetric, that is whose value is independent of the order of the arguments. Such a function really depends on the values of the arguments in a ranking structure. We may without loss of generality restrict the arguments to be sequenced by rank. Then (b) says that the first-ranked arguments matter most, while (c) says that elements with not only poor rank but also poor underlying scores have negligible marginal influence.

The goal is to build a data structure U of arguments and values $f(\boldsymbol{u})$ with the properties:

1. Coding: For all argument sequences $\boldsymbol{x}$ there exists $\boldsymbol{u} \in U$ such that $|f(\boldsymbol{u}) - f(\boldsymbol{x})| < \epsilon$.
2. Size: $|U|$ is not too large, as a function of ϵ.
3. Efficiency: A good neighbor or small set of neighbors $\boldsymbol{u}$ can be found in time proportional to the length of the sequence, with only $O(1)$ further computation needed to retrieve the value $f(\boldsymbol{u})$.
4. Only a “black box” memo table of values $f(\boldsymbol{u})$ is needed, with the remainder of the approximation algorithm staying (essentially) independent of f.

Our main contribution is the construction of a family $\mathcal{G}$ of grid structures U and a simple memoizing algorithm A that employs the following “universal gradient” (indexing the components i from 1):

$$w_i(\boldsymbol{x}) = \frac{1}{i}(1 - x_i). \tag{1}$$

The intuition is that the first i elements have size no larger than u_i, which is to say equal or higher rank and influence on $f(\boldsymbol{u})$ to u_i. If they were all equal, then each would have a share $1/i$ of their total influence. The multiplier $(1-u_i)$ aims to moderate the influence as the argument component u_i itself increases. It is also the simplest multiplier that goes to zero as u_i goes to 1. Our algorithm A uses the w_i as weights in a balancing strategy that reflects the other sequence elements x_j, not only for $j < i$ but also $j > i$.

We report success on a fairly general range of functions f that arise from a natural application in which probabilities are estimated. Some of the f represent salient basic mathematical problems in their own right. Key features of our data structure, algorithm, and overall strategy are:

– The grid is not regular but "warped": it starts fine but becomes coarse as i increases, eventually padding with nonce 1 values.
– The grid has an efficient compact mapping to an array, like a "warped" array implementation of a heap, so that values $f(\boldsymbol{u})$ can be looked up by one index rather than pointer jumping.
– The actual gradient of f is replaced by the universal substitute (1), which reflects properties (b) and (c) above.
– The algorithm to find a good grid neighbor $\boldsymbol{u}$ to the given argument $\boldsymbol{x}$ uses the weights w_i to balance rounding, exploiting the decreasing size of w_i to obtain reasonable approximation to a Knapsack-type problem.
– The algorithm preserves monotonicity of $\boldsymbol{u}$ by working from both ends, in what we call a "meet in the middle" strategy. This and the coarseness help keep the overall grid size manageable even when ℓ is large.

This seems like a "cookbook" approach, but our point is that we have more structure to work with, before the steps of the recipe that have f as a particular ingredient need to be acted on. In our application there is no connection between the weights w_i and the functions f except for the axiomatic properties (b) and (c) and lack of unusual pathology in f. We demonstrate performance that is almost ten times faster than without memoization, and with four-place accuracy from a grid that starts with only two-place fineness. We also compare with a "non-warped" grid that has a sharp index-i cutoff instead, beyond which it pads with 1s. First we describe the application.

2 Estimating Probabilities and Means

Although our application comes from the chess decision-making model of [4], the present computational task is simple and general and can be described without reference to chess. The main model-design parameter is a real function h, and varying h implicitly defines the functions f treated in this paper. The argument is a non-*increasing* sequence of ℓ-many real numbers a_i beginning with $a_1 = 1$. The goal is to fit ℓ-many probability values p_i according to the equations

$$\frac{h(p_i)}{h(p_1)} = a_i; \qquad 0 \le p_i \le 1, \quad \sum_{i=1}^{\ell} p_i = 1.$$

In modeling our grid, we transform $x_i = 1 - a_i$ so that the most influential values x_i as those closest to 0. The function value $f(\boldsymbol{x})$ is just the first probability, p_1. This is hence a fairly broad setting of curve-fitting. Our application further involves data with myriad sequences $\boldsymbol{x}$, for which we need to compute sums $M = \sum_{\boldsymbol{x} \in S} f(\boldsymbol{x})$ over sampled subsets S of the data, which when divided by $|S|$ become means. When S is moderately large, it suffices to have good approximation for f provided it is unbiased. Again this is highly typical in computational applications.

When h is the identity function id, we can solve for f in closed form:

$$f_{id}(\boldsymbol{a}) = \frac{1}{\sum_i a_i}, \quad \text{so} \quad f_{id}(\boldsymbol{x}) = \frac{1}{\ell - \sum_i x_i}.$$

For other functions h, however, we know only iterative techniques to compute $f(\boldsymbol{x}) = p_1$, and from p_1 the other probabilities, to desired precision. These iterations are expensive, so we seek to save them. When h is the logarithm function—more precisely the function $h(p) = 1/\log(1/p)$—we obtain the equations:

$$\frac{\log(1/p_1)}{\log(1/p_i)} = a_i, \quad \text{so} \quad p_i = p_1^{1/a_i}.$$

Viewing $b_i = 1/a_i$ as a general non-decreasing sequence, not necessarily beginning at 1, defines the following mathematical problem: Given y and real numbers $b_1, \ldots, b_\ell$, find a real number p such that

$$y = p^{b_1} + p^{b_2} + \cdots + p^{b_\ell}.$$

When $\ell = 1$ so there is just one number b, then $p = y^{1/b}$, so this is just the problem of root-finding. Hence we think of this problem as computing "vectorized roots" $\sqrt[\boldsymbol{b}]{y}$. Above we have the case $y = 1$ and also $b_1 = 1$.

The model of [4] uses the resulting vector-root function f_{vr} because f_{id} performs extremely poorly. Other functions h such as $h_e(p) = p/\log(1/p)$ or the (inverted) hyperbolic arc-secant function $h_{as}(p) = 1/\log(1/x + \sqrt{1/x - 1})$ may possibly work better, but they seem to lack even the nice format of f_{vr} and to make iteratively computing the resulting f_e or f_{as} even more expensive. These are the main functions we address. In general we have $p_i = h^{-1}(a_i h(p_1))$, so the task of inverting h is also on the table.

To restore some of the intuition from chess, the x_i values are the perceived differences of various chess moves m_i in a chess position from the optimal move m_1. These are obtained from the differences $\Delta(m_1, m_i)$ given by analysis from a strong computer chess program, after factoring in model parameters representing the skill of a particular chess player P. The p_i are estimates for the probabilities that P will choose the respective moves, and in particular p_1 is the probability of finding the move the computer thinks is best. The value M becomes the expected number of agreements with the computer on the set S of moves. We imagine that for testing a small set of a few hundred moves one might pay the time for exact computation, but for *training* the model on sets of many tens of thousands, using sequences of values from 10–20 different levels of analysis on each move, explains the need for memoization. The better moves have x_i

close to zero. If there are k such moves, then p_1 will be order-of $1/k$, and so a $(k+1)$st good move will have influence only at most about $1/(k+1)$. Moves with x_i close to 1 are "blunders," and the exact value of a blunder matters little in the phenomenon of player choice that we are modeling. Hence the axiomatic properties in the Introduction are fulfilled in an intuitive sense based on chess, but our point here is that they flow originally from a quite general mathematical system of equations.

3 The Grid Data Structure

When working with sequences, it is natural first to think of a *trie* data structure, that is a tree whose root branches to the possible first sequence elements (often restricted to those actually used), each of those nodes to possible second elements, and so on. When presented with a new sequence $\boldsymbol{x}$, upon replacing entries x_i by ones in the domain if needed, one follows tree links until reaching a node for which $\boldsymbol{x}$ is known to be the unique sequence through it, whereupon $f(\boldsymbol{x})$ can be stored at that node, or the node is deep enough to store a good approximation. The main advantage of tries is flexibility to add new sequences and values dynamically and compactly, but this comes with high use of main memory and cache misses with pointer jumps.

We sacrifice the dynamism to store pre-computed values. This brings, however, the need to cover all possible next elements, not just those present in the dynamically built data. Since chess positions can have 50 or more legal moves we regard $\ell = 50$, so to store the result for all possible $\boldsymbol{x}$ with 2-decimal-place accuracy would involve powers of 101 that bust the information capacity of the universe. Hence our first task is to finalize the ordered vector $\boldsymbol{s}$ such that $u_i \in s$ for all $\boldsymbol{u} \in U$ where i belong to $[1, \ell]$. Keeping in mind the constraint that, $u_i \leq u_j \iff i \leq j$ if we construct a grid, the resulting grid would be a self-similar tapered tree.

Definition 1. *A self-similar tapered tree is a tree where every node has a branching label b, which is also its number of children. Its children have branches labeled $b, b-1, \ldots, 1$ going from left to right.*

One more definition is useful in this context:

Definition 2. *The branching factor of the grid is the maximum number of children any node can have at any depth. The branching factor of any depth is maximum number of children any node can have at that particular depth.*

The braching factor of the root is the same as that of the grid which is $|s|$ in our case. The choice of elements in s and its size are implementation dependent, though we start with compact branches and later space out the gaps as the argument entries approach the ceiling 1.0.

Because the grid grows exponentially in size with increasing depth, we need to use at least one of the two compromises:

- Truncate: cut off the *depth* of the grid at some depth $d_0 \ll \ell$.
- Warp: reduce the *branching factor* geometrically as the rank index i increases.

To demonstrate the ideas, we develop two schemes. The first idea is utilized by both of our schemes. For best exposition we implement the first before introducing the second. After choosing a fixed spacing vector s, we parameterize our first scheme by depth D and a fixed branching factor B at every depth. In other words, at every depth, the left most node has branching level B. Our improved scheme replaces B by a specifier G of a rounded geometric progression to define the family $\mathcal{G}$ of grids $U = U(d, G)$, in which the omission of "ℓ" as a parameter is deliberate.

For our first scheme, once we finalize the branching factor b, the depth D and the spacing vector s, we can generate all the sequences, where the root always contains the value 0.0. Of all the sequences, $(0.0, 0.0, 0.0, \dots)$ is the infimum (in which all moves have equal value) and $(0.0, 1.0) \equiv (0.0, 1.0, 1.0, 1.0, \dots)$ is the supremum. All other monotone sequences are totally ordered between them by prefix lexicographical order. We evaluate $f(\boldsymbol{u})$ for each of the sequences in the same order and store the output in a file,array or any sequential data structure that supports random access.

For our second implementation, we add the idea of "warping". The grid is built in such a way that, at every subsequent depth, the branching factor gets reduced by half. If the branching factor for the root is $2^n + 1$, then the maximum depth possible for such a grid is $n + 2$. The leaf nodes will contain values u_0 and $u_{|s|-1}$.

We compute evaluations of the function f only once and then store it in file. We can store the generated file in the secondary memory, or the Web where space is abundant. Later, whenever we need to evaluate the function for any vector $\boldsymbol{c} = (c_1, \dots, c_m)$, we can random access the file and fetch the stored approximate evaluation of $f(\boldsymbol{c})$. Here m does not necessarily agree with d. If $m > d$, we simply discard c_{d+1} to c_m, and if $m < d$, we append u_l, $d - m$ times with $\boldsymbol{c}$. Next, we replace each value of $\boldsymbol{c}$ with the nearest neighbor from set s. As both $\boldsymbol{c}$ and set s is ordered, mapping $\boldsymbol{c}$ to any vector $\boldsymbol{u} \in U$ takes $O(1)$ time. At this point, we know, the evaluation for new $\boldsymbol{c}$ is somewhere in the file. We now need to find its exact index in the file and fetch the value.

It is easy to generate all the sequences in increasing order, evaluate the function and just store the evaluations in file. The real complication is calculating the location/index where the evaluation for the given vector is stored in the file.

Lemma 1. *For the grid with fixed branching factor B at every depth, given any vector $\boldsymbol{u} = (u_1, u_2, \dots, u_D)$ where depth $D > 0$, we can find the index of $\boldsymbol{u}$ in $O(D)$ time.*

Proof. For performing the indexing, we first need to create a table. The table holds $V_{d,b}$,the number of nodes at any depth d for any core branch b. core branches are branches generated from the root. d and b represents corresponding row and column of the table respectively. The constructed table has D rows and B column. The entries for the table for the first implementation can be generated using the equation 2.

$$V_{d,b} = \begin{cases} 0 & \text{if } d = 1 \\ 1 & \text{if } d = 2 \\ \sum_{j=b}^{B} V_{d-1,j} & \text{otherwise} \end{cases} \tag{2}$$

or

$$V_{d,b} = \begin{cases} 0 & \text{if } d = 1 \\ 1 & \text{if } d = 2 \\ V_{d-1,b} + V_{d,b+1} & \text{if } b < B \\ V_{d-1,b} & \text{otherwise,} \end{cases}$$

where d ranges from $(1, D)$. All the elements in $V_{1,b}$ is 0 and $V_{2,b}$ is 1. The creation of the table, which is a one-time event, takes $O(BD)$ time. After creation of the table, for calculating the index for the vector $\boldsymbol{u} = (u_1, \dots, u_D)$, we first calculate the position of u_i where $i \in (0, D)$ in the table. We can define a mapping function $h : u_i \to m_i$ where m_i is the corresponding column in the grid for u_i. Then the index would be:

$$index = \sum_{i=1}^{D-1} \sum_{j=h(u_i)}^{h(u_{i+1})-1} V_{D+1-i,j}. \tag{3}$$

Calculating the index requires summing $O(BD)$ terms. By generating a second table which stores sub-sum $\sum_{j=0}^{b} V_{d,j}$ for every b and d, we can perform the inner sum operation in constant time, which is nothing but the subtraction of two values. This makes the whole indexing to take $O(D)$ time.

Example 1. The example we illustrates uses a grid with branching factor 17, and the selection of branches are shown in Table 1. we generate the table 2 using equation 2.

Table 1: Core Branches

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	0.01	0.02	0.03	0.04	0.05	.06	0.1	0.2	0.3	.4	.5	.6	.7	.8	.9	1

Table 2: Fixed Branching Factor

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Depth 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Depth 3	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Depth 4	153	136	120	105	91	78	66	55	45	36	28	21	15	10	6	3	1
Depth 5	969	816	680	560	455	364	286	220	165	120	84	56	35	20	10	4	1

For example, after interpolation, for an arbitrary vector if we get $\boldsymbol{u} = (0, 0, 0.02.0.04, 0.8)$ then, mapping function h maps $\boldsymbol{u}$ to index vector $\boldsymbol{m}$ which is corresponding column for u in table 1. The generated mapping is shown in table 3

Table 3: Mapping from Value to Index

Value(u_i)	Index ($f(u_i) = m_i$)
0	1
0	1
.02	3
.04	5
.8	15

Table 4: Calculation of Index

Depth	Table Row	From	To	Sum
1	5	1	1	0
2	4	1	3	153+136
3	3	3	5	15+14
4	2	5	15	1+1+1+1+1+1+1+1+1+1
SumTotal				328 (zero based index)

Now using Table 2 for lookup, we can calculate the index using equation 3

From Table 4, our algorithm evaluates the index to 328 (0 based indexing). Seeking the 328_{th} element from the file gives the evaluation of the vector.

The fixed branching factor algorithm is well suited for most of the application. But in cases where precision of evaluation of function f is crucial for the initial values of the vector $\boldsymbol{u}$, we can introduce the concept of 'warping'. A warped grid of depth n has branching factor of $2^n + 1$. We assume at every depth, branching factor gets reduced by half. The model can be extended to other reducing factors without loss of generality.

Lemma 2. *For a grid with exponentially reducing branching factor B, given any vector $\boldsymbol{u} = (u_1, u_2, \ldots, u_D)$ and depth $D > 0$, we can find the index of the sequence in $O(D)$ time.*

Proof. Just like grid for fixed branching, for preforming indexing we need to generate a table. The constructed table has D rows and B column. The entries for the table for the second implementation can be generated using the equation 3.

$$V_{d,b} = \begin{cases} 0 & \text{if } d = 1 \\ 1 & \text{if } d = 2 \\ V_{d-1,b} + V_{d,b+2^{d-2}} & \text{if } b + 2^{d-2} \leq B \\ V_{d-1,b} & \text{otherwise,} \end{cases}$$

where d ranges from $(1, D)$. $V_{d,b}$ is the number of nodes at depth d and for core branch index b. All the elements in $V_{1,b}$ is 0 and $V_{2,b}$ is 1. The creation of the table again takes $O(BD)$ time.

The indexing for this scheme is different than that of our first scheme. If the mapping function is $g : (u_i, d) \rightarrow m_{i,d}$ we can evaluate $m_{i,d}$ using equation 3.

$$m_{i,d} = \begin{cases} 0 & \text{if } (h(u_i) - 1) \bmod 2^{d-1} \neq 0 \\ D - (h(u_i) - 1)/2^{d-1}) & \text{otherwise,} \end{cases}$$

Function h is the mapping function used for fixed branched grid and $m_{i,d} = 0$ indicates the corresponding u_i is not present for the particular depth. This information can be stored for future lookup. A illustration of the mapping is provided in the example. Finally, the revamped indexing function for this reduced brached grid is:

$$index = \sum_{i=1}^{D-1} \sum_{j=g(u_i,i)}^{g(u_{i+1},i)-1} V_{D+1-i,j} \tag{4}$$

Again, by the same procedure as described for fixed branching, we can calculate the index in $O(D)$ time.

Example 2. We again construct the table for the grid, but this time with exponentially reducing branching factor. The construed table for branching factor B equal to 17 and depth D equal to 5 is shown in table 5.

Table 5: Exponential Reduction in Branch

Column	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth 1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Depth 2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Depth 3	9	8	8	7	7	6	6	5	5	4	4	3	3	2	2	1	1
Depth 4	25	20	20	16	16	12	12	9	9	6	6	4	4	2	2	1	1
Depth 5	35	26	26	20	20	14	14	10	10	6	6	4	4	2	2	1	1

Table 6: Mapping from (value, depth) to Index

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Depth 1	0	0.01	0.02	0.03	0.04	0.05	.06	0.1	0.2	0.3	.4	.5	.6	.7	.8	.9	1
Depth 2									0	0.02	0.04	.06	0.2	.4	.6	.8	1
Depth 3													0	0.04	0.2	.6	1
Depth 4															0	0.2	1
Depth 5																0	1

Table 6 gives an example of a mapping function illustrated in this example. The first row represents the core branches of the grid, and subsequent rows represents branches possible at different depth. The column number for any value at any depth represents the mapped index, for example$g(0.04, 3) = 14$.

We are illustrating an example of this approach for a vector $\boldsymbol{c} = (0.0, 0.03, 0.4, 0.6, 1)$ after interpolation from any arbitrary vector. Table 6 represents the mapping function g where Table 7 shows specific mapping for our example. Finally by means of eqation 4 we can calculate the index. The breakdown of the calculation is shown in Table 8.

Table 7: Example of Mapping

Depth	From Value(c_i)	From Index ($g(u_i, i) = m_i$)	To Value(c_{u+1})	To Index ($g(u_{i+1}, i) = m_{i+1}$)
1	0.0	1	0.03	4
2	0.03	11	0.4	14
3	.4	16	0.6	16
4	.6	17	1	17

The final calculation of the index returns 101. This indicates that the 101^{th} entry in the corresponding file contains the value for this specific vector.

Table 8: Calculation of Index

Depth	Table Row	From	To	Sum
1	5	1	4	35 +26 +26
2	4	11	14	6+4+4
3	3	16	16	0
4	2	17	17	0
SumTotal				101 (zero based index)

4 Interpolation Algorithm

Although the members of U are totally ordered lexicographically, the values $f(\boldsymbol{u})$ are generally *not* monotone in this ordering, and this makes the neighborhood topology play havoc when given a non-grid sequence $\boldsymbol{x}$. For instance, if the spacing starts $0.00, 0.02, 0.04, \ldots$ in the first few index places $i \geq 2$, then the sequence

$$\boldsymbol{x} = 0.00, 0.01, 0.20, 0.40, 0.60, 0.80, 1.0 \ldots$$

has as its *lower* neighbor the sequence $0.00, 0.00, 1.0, 1.0 \ldots$, which generally gives much *higher* p_1, and its *upper* neighbor the sequence $0.00, 0.02, 0.02, 0.02, 0.02 \ldots$ in which all moves are close to equal and p_1 has nearly its mimimum possible value, which is $1/\ell$. Thus attempting to use the neighbors in the *trie* structure of the sequences is bad. Instead, given (any) $\boldsymbol{x}$, we define its *component-wise bounds* x^+ and x^-. In this case, assuming the spacing continues $\dots, 0.20, 0.35, 0.50, 0.75, 1.0$, we get:

$$\boldsymbol{x}^+ = 0.00, 0.02, 0.20, 0.50, 0.75, 1.0, 1.0 \dots$$
$$\boldsymbol{x}^- = 0.00, 0.00, 0.20, 0.35, 0.50, 0.75, 1.0 \dots$$

Since these are members of U, it is plausible to output some weighted average of $f(\boldsymbol{x}^+)$ and $f(\boldsymbol{x}^-)$. We mix this idea with interpolating to find a better argument $\boldsymbol{u}$ between $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$. Here is where the oblivious weights

$$w_i = w_i(\boldsymbol{x}) = \frac{1}{i}(1 - x_i)$$

are employed in place of the actual partial derivatives of f at x_i. To define u_i for each i, we have the chose to "round up" to x_i^+, or "round down" to x_i^-. The choice depends not merely on which is closer, but also on balancing an accumulated "credit" value. For this purpose we utilized the spacing vector $\boldsymbol{s}$, which contains all possible points in ascending order.

Algorithm 1 Interpolation Algorithm

```
    procedure INTERPOLATE($x, s, d, u$)
        $c \leftarrow 0$
        $j \leftarrow 2$
        for $i \leftarrow 1$ to $d$ do
            $w \leftarrow \frac{1}{i}(1 - x[i])$
            if $(x[i] * w + c) \geq s[j]$ then
                $j \leftarrow j + 1$
            else
                $u[i] \leftarrow s[j-1]$
                $c \leftarrow c + x[i] * w - u[i]$
            end if
        end for
    end procedure
end
```

The procedure in algorithm 1 for interpolation takes a vector $\boldsymbol{x}$ for which we want to evaluate the function, spacing vector $\boldsymbol{s}$, output vector $\boldsymbol{u}$ and its size d which usually represents the the depth of the grid as input. In the procedure w and c represent the weight of current element and the accumulated credit respectively. The algorithm takes $O(d)$ time. This procedure converts a vector $\boldsymbol{x}$ to a vector $\boldsymbol{u} \in U$.

5 Experimental Results

We implemented both of the schemes in c++ and the code was compiled with highest optimization enable. We ran the code on a shared Red Hat Enterprise Linux Server release 5.7 (Tikanga) (64-bit) system with 32 gB ram which is configured for non-interactive, CPU-intensive and long-running processes. For both the scheme, we used vector-root f_{vr} (see section 2) as the function f. The core branches for the fixed brached grid is shown in table 5. For this scheme, we set the branching factor to 16 and the depth to 15. we generated the evaluation for every possible vector using the core branches values, and stored the evaluation in a file. The generated file had a total of $77, 558, 760$ entries, starting from the evaluation for a vector of all 0's to the entry for vector $(0, 1)$.

Index	1	2	3	4	5	6	7	8	9	10	11	14	13	14	15	16
Value	0.0	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.60	0.70	0.80	0.90	1.0

For testing the performance of our scheme, we used 8000 random vectors. The length of the vectors were fixed to 50 where the values in the vectors are ordered and ranged between 0 and 1. To make sure the vector size is sufficiently large for any practical application before it reaches 1 , we generate the vectors with bias which also replicates evaluation of chess moves for a particular position more realistically.

For each of the entries, we first calculated the exact vector-root using newton's method and then calculated the closest neighbor of that vector using our interpolation algorithm(refer section 4) and found the corresponding index for that vector. Finally we accessed the file to fetch the evaluation at that location.

For interpolation, we tested with various mapping function, for example, nearest neighbor, nearest neighbor based on progressive credit, lower bound for the vector, upper bound of the vector, and the mean of the lower and upper bound of the vector. The "progressive credit" strategy—that is knapsack strategy— gives the best accuracy. Figure 1 shows a scatter plot of the deviations from the true value of the vector root function.

Beside giving 2 decimal point accuracy in approximation, we have logged a substantial improvement over the time it requires to provide the result. We ran 20 iterations of $1, 000, 000$ random vectors completion timing analysis for both our grid implementation and naive vector-root implementation. The performance is consolidated in Figure 2. From our experiments we found, using averages of 20 runs, that the knapsack-based grid calculation was 8.668 times faster than conventional vector-root calculation using Newtons method.

We tested similar experiment with the reduced-branching-factor implementation. The branching factor of the grid was set to 257, where each branches were equally spaced between 0 and 1, and the the depth of the grid was 10.For interpolation we mapped the vector to the nearest neighbor in the grid. We tested the implementation with 8,000 iterations. Figure 3 shows the closeness of fit. The average deviation for the scheme from the vector-root was -0.0008 where standard deviation was 0.0052. On an average, The execution time is 9.6056 times faster over real-time vector-root evaluation.

Fig. 1: Performance Analysis for fixed branched Grid

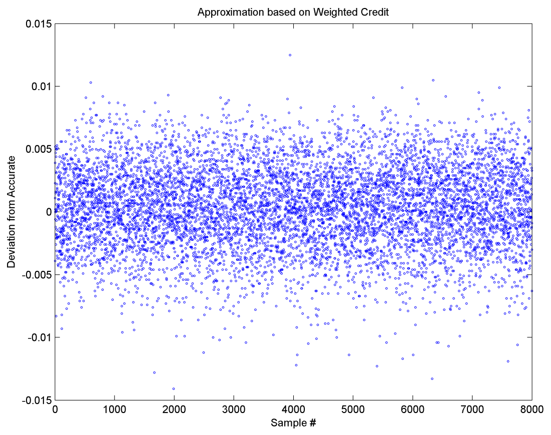
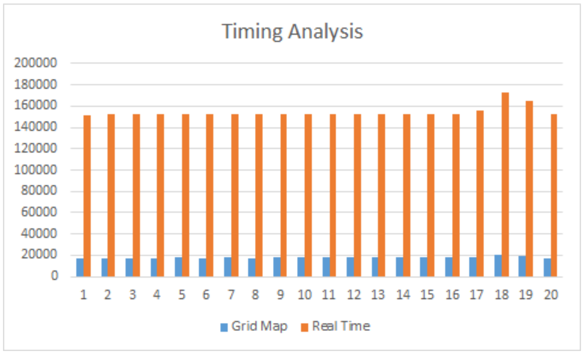


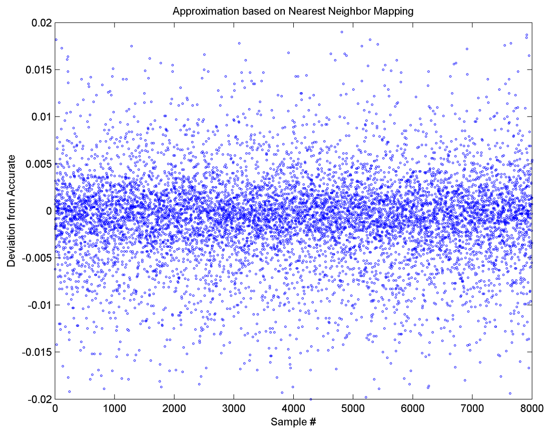
Fig. 2: Timing Analysis



6 Future Work

The model can be extended for distributed computing . Index calculation at each depth is independent of other depths, thus it is possible to compute those in parallel. We are working on a more complex model where this concept can be further extended to allow selective reduced branching at various depth ,i.e. reduction of the depth can occur at particular depth instead of each depth. This allows going to higher depth than $n+2$, but still keeping the branching factor 2^n+1. We are currently working on to fix a branching factor and depth which guarantees the closeness of the evaluation to a threshold for a particular funciton.

Fig. 3: Approximation Performance Analysis for Reduced Branched Grid



7 Conclusion

In this paper we have developed a special purpose data structure for storing sample points of a function f so that the values of f at other points can be interpolated efficiently.This data structure can be used to store the result of computation intensive function values. This data structure can be used for faster remote computing. As the function is not evaluated at real time, this suits the requirement of embeded system, where both the processing power and convervation of energy is vital [5]. Along with these benefits, this data structure provides alomst tenfold performance enhancement beside good approximation.

References

1. Campbell-Kelly, M., Croarken, M., Flood, R., Robson, E.: The History of Mathematical Tables. Oxford University Press, USA (2003)
2. Muthukrishnan, S.: Data Streams: Algorithms and Applications. Now Publishers (2005)
3. Liang, F.: Stochastic approximation monte carlo for mlp learning. In: Encyclopedia of Artificial Intelligence. Wiley (2009) 1482–1489
4. Regan, K., Haworth, G.: Intrinsic chess ratings. In: Proceedings of AAAI 2011, San Francisco. (2011)
5. Alidina, M., Monteiro, J.C., Devadas, S., Ghosh, A., Papaefthymiou, M.C.: Precomputation-based sequential logic optimization for low power. IEEE Trans. VLSI Syst. **2** (1994) 426–436