Selection on $X_1 + X_2 + \cdots X_m$ via Cartesian product trees

Patrick Kreitzberg [1] , **Kyle Lucke** [2] , **Jake Pennington** [1] , **Oliver Serang** [Corresp. 2]

[1] Department of Mathematics, University of Montana, Missoula, Montana, United States

[2] Department of Computer Science, University of Montana, Missoula, Montana, United States

Corresponding Author: Oliver Serang
Email address: oliver.serang@umt.edu

Selection on the Cartesian product is a classic problem in computer science. Recently, an optimal algorithm for selection on X+Y, based on soft heaps, was introduced. By combining this approach with layer-ordered heaps (LOHs), an algorithm using a balanced binary tree of $X+Y$ selections was proposed to perform selection on $X_1 + X_2 + \cdots X_m$ $X_1+X_2+\cdots+X_m$) in $X_1 + X_2 + \cdots X_m$ $o(n\cdot m + k\cdot m)$, where $X_1 + X_2 + \cdots X_m$ X_i have length $X_1 + X_2 + \cdots X_m$ n. Here, that $X_1 + X_2 + \cdots X_m$ $o(n\cdot m + k\cdot m)$ algorithm is combined with a novel, optimal LOH-based algorithm for selection on $X_1 + X_2 + \cdots X_m$ $X+Y$ (without a soft heap). Performance of algorithms for selection on $X_1 + X_2 + \cdots X_m$ $X_1+X_2+\cdots+X_m$) are compared empirically, demonstrating the benefit of the algorithm proposed here.

Selection on $X_1 + X_2 + \cdots + X_m$ via Cartesian product trees

Patrick Kreitzberg[1], Kyle Lucke[2], Jake Pennington[1], and Oliver Serang[2]

[1]**University of Montana, Depart. of Mathematics, Missoula, MT, USA**
[2]**University of Montana, Depart. of Computer Science, Missoula, MT, USA**

Corresponding author:
Oliver Serang[2]

Email address: oliver.serang@umontana.edu

ABSTRACT

Selection on the Cartesian product is a classic problem in computer science. Recently, an optimal algorithm for selection on $A + B$, based on soft heaps, was introduced. By combining this approach with layer-ordered heaps (LOHs), an algorithm using a balanced binary tree of $A + B$ selections was proposed to perform selection on $X_1 + X_2 + \cdots + X_m$ in $o(n \cdot m + k \cdot m)$, where X_i have length n. Here, that $o(n \cdot m + k \cdot m)$ algorithm is combined with a novel, optimal LOH-based algorithm for selection on $A + B$ (without a soft heap). Performance of algorithms for selection on $X_1 + X_2 + \cdots + X_m$ are compared empirically, demonstrating the benefit of the algorithm proposed here.

INTRODUCTION

Sorting all values in $A + B$, where A and B are arrays of length n and $A + B$ is the Cartesian product of these arrays under the $+$ operator, is nontrivial. In fact, there is no known approach faster than naively computing and sorting them which takes $O(n^2 \log(n^2)) = O(n^2 \log(n))$ time (Bremner et al., 2006); however, Fredman (1976) showed that $O(n^2)$ comparisons are sufficient, though no $O(n^2)$ time algorithm is currently known. Frederickson and Johnson (1982) published an algorithm which selects the kth value in $A + B$ with runtime $O(n + p \log(k/p))$ where $p = min(k, n)$. In 2018, Kaplan *et al.* described an optimal method for selecting the top k values of $A + B$, in terms of soft heaps (Kaplan et al., 2019)(Chazelle, 2000). Serang (2020) created an optimal method for selecting the top k values of $A + B$ which does not rely on a pointer-based data structure like the soft heap and is fast in practice. Any reference to "Serang's method" or "Serang's pairwise method" will be in reference to this optimal method which performs pairwise selection on $A + B$.

Johnson and Mizoguchi (1978) extended the problem to selecting the kth value in $X_1 + X_2 + \cdots + X_m$ and did so with runtime $O(m \cdot n^{\lceil \frac{m}{2} \rceil} \log(n))$; however, there has not been significant work done on the problem since. Kreitzberg et al. (2020a) introduced the FastSoftTree method which performs selection on $X_1 + X_2 + \cdots + X_m$ by creating a balanced binary tree of pairwise selection problems. FastSoftTree uses Kaplan *et al.*'s soft heap-based pairwise selection method and performs selection in $O(n \cdot m + k \cdot m^{\log_2(\alpha^2)})$ time with space usage in $O(n \cdot m + k \log(m))$ for constant $\alpha > 1$. If only the kth value is desired, then Johnson and Mizoguchi's method is the fastest known when $k > \frac{m \cdot n^{\lceil \frac{m}{2} \rceil} \log(n)}{m^{\log_2(\alpha^2)}}$, for $k < \frac{m \cdot n^{\lceil \frac{m}{2} \rceil} \log(n)}{m^{\log_2(\alpha^2)}}$ (or if the top k values are desired) Kreitberg *et al.*'s FastSoftTree is fastest.

Selection on $X_1 + X_2 + \cdots + X_m$ is important for max-convolution (Bussieck et al., 1994) and max-product Bayesian inference (Serang, 2015; Pfeuffer and Serang, 2016). Computing the k best quotes on a supply chain for a business, when there is a prior on the outcome (such as components from different companies not working together), becomes solving the top values of a probabilistic linear Diophantine equation (Kreitzberg and Serang, 2020) and thus becomes a selection problem. Finding the most probable isotopologues of a compound such as hemoglobin, $C_{2952}H_{4664}O_{832}N_{812}S_8Fe_4$, may be done by solving $C + H + O + N + S + Fe$, where C would be the most probable isotope combinations of 2,952 carbon molecules (which can be computed via a multinomial at each leaf, ignored here for simplicity), H would be the most probable isotope combinations of 4,664 hydrogen molecules, and so on. The selection

46 method proposed in this paper has already been used to create the world's fastest isotopologue calculator
47 (Kreitzberg et al., 2020b).

Layer-ordered heaps

49 In a standard binary heap, the only known relationships between a parent and a child is $A_i \leq A_{children(i)}$. A
50 layer-ordered heap (LOH) has stricter ordering than the standard binary heap, but is able to be created in
51 $\Theta\left(n\log(\frac{n}{n \cdot (\alpha-1)+1}) + \frac{n \cdot \alpha \cdot \log(\alpha)}{\alpha-1}\right) = \Theta(n)$ time for constant $\alpha > 1$ (Pennington et al., 2020). α is the rank
52 of the LOH and determines how fast the layers grow. A LOH partitions the array into several layers, L_i,
53 which grow exponentially such that $|L_1| = 1$ and $\frac{|L_{i+1}|}{|L_i|} \approx \alpha$. Every value in a layer L_i is $\leq$ every value in
54 proceeding layers $L_{i+1}, L_{i+2}, \ldots$ which we denote as $L_i \leq L_{i+1}$, this is the "layer-ordering property." If
55 $\alpha = 1$, then all layers are size one and the LOH is sorted; therefore, to be constructed in $\Theta(n)$ time the
56 LOH must have $\alpha > 1$.

Pairwise selection

58 Serang's method of selection on $A + B$ utilizes LOHs to be both optimal in theory and fast in practice. In
59 this section we provide a summary of Serang's method, for a more detailed description with analysis see
60 Serang (2020). The method has four phases. Phase 0 is simply to LOHify (make into a layer-ordered
61 heap) the input arrays which can be done in $\Theta(n)$ time.

62   Phase 1 finds which layer products may be necessary for the selection. A layer product, $A^{(u)} + B^{(v)}$, is
63 the Cartesian product of layers $A^{(u)}$ and $B^{(v)}$: $A_1^{(u)} + B_1^{(v)}, A_2^{(u)} + B_1^{(v)}, \ldots, A_1^{(u)} + B_2^{(v)}, \ldots$. Finding which
64 layer products are necessary for the selection can be done using a standard binary heap. A layer product is
65 represented in the binary heap in two separate ways: a min tuple $\lfloor(u,v)\rfloor = (min(A^{(u)} + B^{(v)}), (u,v), false)$
66 and a max tuple $\lceil(u,v)\rceil = (max(A^{(u)} + B^{(v)}), (u,v), true)$. Creating the tuples does not require calculating
67 the Cartesian product of $A^{(u)} + B^{(v)}$ since $min(A^{(u)} + B^{(v)}) = min(A^{(u)}) + min(B^{(v)})$ which can be found
68 in a linear pass of A and B separately. The same argument applies for $\lceil(u,v)\rceil$. $false$ and $true$ note that
69 the tuple contains the minimum or maximum value in the layer product, respectively. Also, let $false = 0$
70 and $true = 1$ so that a min tuple is popped before a max tuple even if they contain the same value.

71   Phase 1 uses a binary heap to retrieve the tuples in sorted order. When a min tuple, $\lfloor(u,v)\rfloor$, is popped,
72 then its neighbors $(\lfloor(u+1,v)\rfloor, \lfloor(u,v+1)\rfloor)$ and the corresponding max tuple are pushed, assuming
73 $\lfloor(u+1,v)\rfloor, \lfloor(u,v+1)\rfloor$ are in bounds (a set is used to ensure a layer product is inserted only once). When
74 a max tuple is popped, a variable s is increased by $|A^{(u)} + B^{(v)}| = |A^{(u)}| \cdot |B^{(v)}|$ and (u,v) is appended to a
75 list q. This continues until $s \geq k$. Figure 1 shows an example of phase 1 when $k = 10$.

76   In phases 2 and 3 all max tuples still in the heap have their index appended to q, then the Cartesian
77 product of all layer products in q are generated. A linear time one-dimensional k-select is performed on
78 the values in the Cartesian products to produce only the top k values in $A + B$. The time complexity of the
79 algorithm is linear in the overall number of values produced which is $O(k)$.

Figure 1. **(A): Nine layer products of** $A + B$. **(B): The layer product tuples in the order they
would pop from the heap, the number of values in their Cartesian product, and** s, **the cumulative
size of the layer products whose max tuples have been popped.** The two axes are the input arrays
after being LOHified. The values of all 18 possible layer product tuples are shown (nine min tuples in
blue and nine max tuples in green). If $k = 10$, then the tuples will be popped in the order shown in (B).
After $(20, (3,1), true)$ is popped, s (the total number of items in the Cartesian product of all max tuples)
exceeds k. Note that the values in the layers of A and B are not necessarily in sorted order.

80   In this paper we efficiently perform selection on $X_1 + X_2 + \cdots + X_m$ by utilizing Serang's pairwise
81 selection method.

METHODS

83 In order to retrieve the top k values from $X_1 + X_2 + \cdots + X_m$, a balanced binary tree of pairwise selections is
84 constructed. The top k values are calculated by selection on $X_1 + X_2, X_3 + X_4, \ldots$ then on $(X_1 + X_2) + (X_3 +$
85 $X_4), (X_5 + X_6) + (X_7 + X_8), \ldots$. All data loaded and generated is stored in arrays which are contiguous in
86 memory, allowing for great cache performance compared to a soft heap-based method.

Tree construction

88 The tree has height $\lceil \log_2(m) \rceil$ with m leaves, each one is a wrapper around one of the input arrays which
89 are unsorted and have no restrictions on the type of data they contain. Upon construction, the input arrays
90 are LOHified in $\Theta(n \cdot m)$ time, which is amortized out by the cost of loading the data. Each node in the
91 tree above the leaves performs pairwise selection on two LOHs: one generated by its left child and one
92 generated by its right child. All nodes in the tree generate their own LOH, but this is done differently for
93 the leaves vs the pairwise selection nodes. When a leaf generates a new layer it simply allows its parent
94 to have access to the values in the next layer of the LOHified input array. For a pairwise selection node,
95 generating a new layer is more involved.

Pairwise selection nodes

97 Each node above the leaves is a pairwise selection node, each of which has two children that may be
98 leaves or other pairwise selection nodes. In contrast to the leaves, the pairwise selection nodes will have to
99 calculate all values in their LOHs by generating an entire layer at a time. Generating a new layer requires
100 performing selection on $A + B$, where A is the LOH of its left child and B is the LOH of its right child.
101 Due to the combinatorial nature of this problem, simply asking a child to generate their entire LOH can
102 be exponential in the worst case so they must be generated one layer a time and only as necessary.
103   The pairwise selection performed is Serang's method with a few modifications (though Figure 1 is
104 still representative of the method). The size of the selection is always the size of the next layer, $k = |L_i|$, to
105 be generated by the parent. The selection begins in the same way as Serang's: a heap is used to pop min
106 and max layer product tuples. When a min tuple, $\lfloor (u, v) \rfloor$, is popped the values in the Cartesian product
107 are generated and appended to a list of values to be considered in the k-selection. The neighboring layer
108 products inserted into the heap are determined using the scheme from Kaplan *et al.* which differs from
109 Serang's method: $\lceil (u, v) \rceil, \lfloor (u, 2v) \rfloor$, and $\lfloor (u, 2v + 1) \rfloor$ are always inserted and, if $v = 1$, $\lfloor (2u, v) \rfloor$ and
110 $\lfloor (2u + 1, v) \rfloor$ are inserted as well. This insertion scheme will not repeat any indices and therefore does
111 not require the use of a set to keep track of the indices in the heap. When any min tuple is proposed, the
112 parent asks both children to generate the layer if it is not already available. If one or both children are not
113 able to generate the layer (*i.e.* the index is larger than the full Cartesian product of the child's children)
114 then the parent does not insert the tuple into its heap. The newly generated layer is simply appended to
115 the parent's LOH and may now be accessed by the parent's parent. An example of a pairwise selection
116 node generating a new layer which requires a child to generate a new layer is shown in Figure 2.
117   The dynamically generated layers should be kept in individual arrays, then a list of pointers to the
118 arrays may be stored. This avoids resizing a single array every time a new layer is generated.
119   Theorem 1 in Serang (2020) proves that Serang's method performs selection in $O(k)$ time. Lemmas
120 6 and 7 show that the number of all values generated is $O(n + k)$; however, Lemma 7 may be amended
121 to show that any layer product of the form $(u, 1)$ or $(1, v)$ will generate $\leq \alpha \cdot |(u - 1, 1)| \in O(k)$ or
122 $\leq \alpha \cdot |(1, v - 1)| \in O(k)$ values, respectively, to show that the total values generated is $O(k)$. Thus, the
123 total number of values generated when a parent adds a new layer L_i is $O(|L_i|)$.

Selection from the root

125 In order to select the top k values from $X_1 + X_2 + \cdots + X_m$, the root is continuously asked to generate new
126 layers until the cumulative size of the layers in their LOH exceeds k. Then a k-selection is performed
127 on the layers to retrieve only the top k. Due to the layer-ordering property, a selection only needs to be
128 performed on the last layer, all values in previous layers will be in the top k.
129   The Cartesian product tree is constructed in the same way as the FastSoftTree (Kreitzberg et al.,
130 2020a) as both methods dynamically generate new layers in a similar manner with the same theoretical
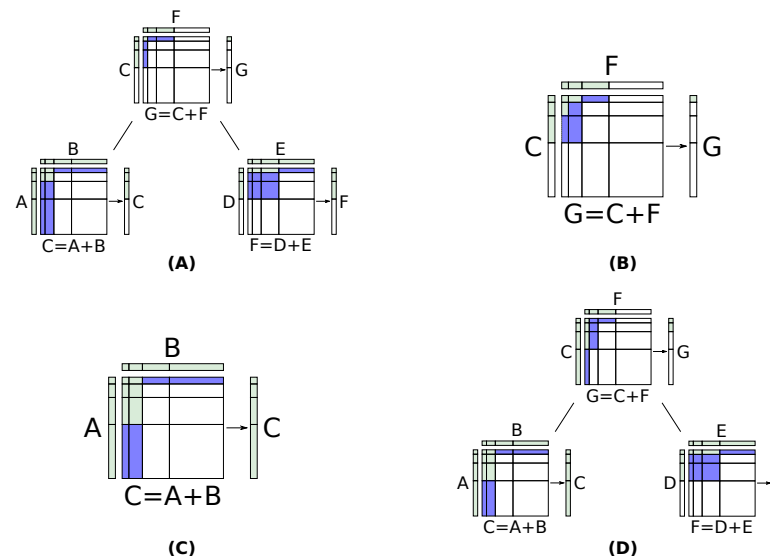
Figure 2. **The process of adding a layer to a pairwise selection node's LOH, *G*, when both children are pairwise selection nodes. Each node has a LOH it generates for its parent to access as well as a (not realized) matrix formed by the Cartesian product of its children's LOHs.** Blue layer products currently have their min tuple in the heap. Green layer products have had at least their min tuple popped from the heap (and thus have inserted other tuples into the heap). (A) The triplet before adding a new layer to *G*. (B) The parent generating the next layer in its LOH. The parent pops $\lfloor(1,2)\rfloor$ and must now insert $\lfloor(1,4)\rfloor, \lfloor(1,5)\rfloor$ (ignored here for simplicity), and, $\lceil(1,2)\rceil$; however, the left child has not yet generated the fourth layer in its LOH, *C*, so the parent can not insert $\lfloor(1,4)\rfloor$. (C) The left child generating the fourth layer in its LOH, *C*. The left child pops $\lfloor(1,3)\rfloor, \lfloor(2,3)\rfloor$ and performs the appropriate insertions (including $\lceil(1,3)\rceil$ and $\lceil(2,3)\rceil$). The left child then pops $\lceil(1,3)\rceil$ and $\lceil(2,3)\rceil$ at which point it has enough values to select the next layer in *C*. Now that *C* has its fourth layer, the parent is able to pop $\lfloor(1,2)\rfloor$ and $\lceil(1,2)\rceil$ then select its second layer. The parent did not need *F* to generate a new layer and so the right child remains the same as before. (D) The triplet after the parent and left child perform the necessary operations to generate the next layer in *G*.

131   runtime. In both methods, the pairwise selection creates at most $O(\alpha^2 \cdot k)$ values. Thus the theoretical

132   runtime of both methods is $O(n \cdot m + k \cdot m^{\log_2(\alpha^2)})$ with space usage $O(n \cdot m + k \log(m))$.

Wobbly version

134   In Serang's pairwise selection, after enough layer product tuples are popped from the heap to ensure they

135   contain the top k values, there is normally a selection performed to remove any excess values. Strictly

136   speaking, this selection is not necessary anywhere in the tree except for the root when the final k values

137   are returned. When the last max tuple $\lceil (u,v) \rceil$ is popped from the heap, $max(A^{(u)} + B^{(v)})$ is an upper

138   bound on the kth value in the k-selection. Instead of doing a k-selection and returning the new layer, which

139   requires a linear time selection followed by a linear time partition, we can simply do a value partition on

140   $max(A^{(u)} + B^{(v)})$.

141   A new layer generated from only a value partition and not a selection is not guaranteed to be size k, it

142   is at least size k but contains all values $\leq max(A^{(u)} + B^{(v)})$. In the worst case, this may cause layer sizes

143   to grow irregularly with a constant larger than α. For example, if $k = 2$ and $|L_1| = |L_2| = 1$ then in the

144   worst case every parent will ask their children to each generate two layers and the value partition will not

145   remove any values. Each leaf will generate two values, their parents will then have a new layer of size

146   $2^2 = 4$, their parent will have a new layer of size $(2^2)^2 = 16$, etc. Thus the root will have to perform a

147   2-selection on 2^m values which will be quite costly.

148   In an application like calculating the most probable isotopologues of a compound, this version can be

149   quite beneficial. For example, to generate a significant amount of the isotopologues of the titin protein

150   may require k to be hundreds of millions. Titin is made of only carbon, hydrogen, nitrogen, oxygen, and

151   sulfur so it will only have five leaves and a tree height of three. The super-exponential growth of the

152   layers for a tree with height three is now preferential because it will still not create so many more than

153   k values but it will do so in many fewer layers with only value partitions and not the more costly linear

154   selections. We call this the "wobbly" Cartesian product tree.

RESULTS

156   All experiments were run on a workstation equipped with 256GB of RAM and two AMD Epyc 7351

157   processors running Ubuntu 18.04.4 LTS. The data was randomly generated using the built-in `rand()`

158   function (seeded with the value 1) in `C++`. All values were integers between 0 and 10,000. Though arrays

159   of random values are used, the performance gain does not depend on the values in the arrays. If the input

160   arrays are homogeneous so that the value for all min and max tuples are the same, they will be popped in

161   ascending order of their index tuple and will still generate at most $O(\alpha^2 \cdot k)$ values.

Cartesian product trees vs FastSoftTree

163   In a Cartesian product tree, replacing the pairwise $A + B$ selection steps from Kaplan *et al.*'s soft heap-

164   based algorithm with Serang's optimal LOH-based method provides the same $o(n \cdot m + k \cdot m)$ theoretical

165   performance for the Cartesian product tree but is practically much faster (Table 1). This is particularly

166   true when $k \cdot m^{\log_2(\alpha^2)} \gg n \cdot m$, where popping values dominates the cost of loading the data. When

167   $k \geq 2^{10}$, $k \cdot m^{0.2750} < n \cdot m$ which is reflected in our results where for $k = 2^{20}$ we get a 630.4× speedup,

168   significantly larger than for $k = 2^{10}$ which only has a 10.27× speedup. For any reasonable ε and α,

169   the difference in number of values generated (and thus in memory usage) will not differ significantly.

170   Regardless of which algorithm generates less values, the performance gain due to generating less overall

171   values will be negligible compared to the difference in cache performance between soft heaps and LOHs.

Standard vs Wobbly

173   As we see in Table 2, for small m the Cartesian product tree can gain significant increases in performance

174   when there are no linear selections performed in the tree and the layers are allowed to grow super-

175   exponentially. As k grows, the speedup of the wobbly version continues to grow, resulting in a 1.786×

176   speedup for $k = 2^{30}$. When $m \gg 5$ the growth of the layers near the root start to significantly hurt the

177   performance. For example, if $n = 32, m = 256$ and $k = 256$ the wobbly version takes 0.5805 seconds and

178   produces 149,272 values at the root compared to the non-wobbly version which takes 1.8810×10^{-03}

179   seconds and produces just 272 values at the root.

k	Cartesian product tree (seconds)	FastSoftTree (seconds)
2^2	1.404×10^{-03}	3.146×10^{-03}
2^3	1.504×10^{-03}	2.855×10^{-03}
2^4	1.521×10^{-03}	3.163×10^{-03}
2^5	1.592×10^{-03}	2.618×10^{-03}
2^6	1.689×10^{-03}	4.172×10^{-03}
2^7	1.718×10^{-03}	4.830×10^{-03}
2^8	1.881×10^{-03}	8.864×10^{-03}
2^9	2.080×10^{-03}	0.01143
2^{10}	1.745×10^{-03}	0.01792
2^{11}	2.217×10^{-03}	0.02362
2^{12}	3.123×10^{-03}	0.04459
2^{13}	3.318×10^{-03}	0.07026
2^{14}	5.099×10^{-03}	0.111
2^{15}	6.240×10^{-03}	0.2296
2^{16}	8.724×10^{-03}	0.4952
2^{17}	0.01266	0.9609
2^{18}	0.01663	1.610
2^{19}	0.02684	12.77
2^{20}	0.0405	25.53

Table 1. Runtimes for Cartesian product tree vs FastSoftTree with $n = 32$, $m = 256$ and $\alpha = 1.1$. The runtime is averaged over 20 iterations. For small problems the soft heap-based tree is competitive with the Cartesian product tree; however, for large enough k the cache performance of the LOH significantly outperforms the soft heap resulting in a $630.4\times$ speedup for $k = 2^{20}$.

DISCUSSION

Replacing pairwise selection which uses a soft heap with Serang's method provides a significant increase in performance. Since both methods LOHify the input arrays (using the same LOHify method) the most significant increases are seen when $k \cdot m^{\log_2(\alpha^2)} \gg n \cdot m$. For small m, the performance can be boosted using the wobbly version; however, for large m the super-exponentially sized layers can quickly begin to dampen performance. It may be possible to limit the layer sizes in the wobbly version by performing selections only at certain layers of the tree: either by performing the selection on every i^{th} layer or only on the top several layers.

For all experiments $\alpha = 1.1$, though the optimal α is not known. As α approaches 1 the layer sizes also approach 1 and the LOHs become fully sorted. Having all layer sizes of 1 will mean not producing any unnecessary values; however, this will lead to slower practical performance due to popping k layer product tuples from the binary heap which will cause runtime to be in $\Omega(k\log(k))$. As α grows, both the number of pops from the binary heap and the time to LOHify the input arrays decreases but the amount of unnecessary values increases. The unnecessary values will slow practical performance both when they are calculated and in the subsequent linear selections and partitions performed to create the layer. In practice, $\alpha = 1.1$ seems to find a good balance between number of pops from binary heaps and the number of unnecessary values generated, both in Table 1 and Table 2 as well as when calculating the most probable isotopologues of a compound.

Due to corruption, when using a soft heap-based method, online computation of values requires retrieving the top k_1 values and then top k_2 remaining values and so on in order to be performed with optimal runtime. Selection on $X_1 + X_2 + \cdots + X_m$ requires optimal, online computation of values in each pairwise selection node in order to be performed in $o(n \cdot m + k \cdot m)$ time, but this applies to any soft heap-based method which requires online computation of the top values.

The number of corrupt values is bounded by $\varepsilon \cdot I$, where I insertions have been performed to date; therefore, there are at most $\varepsilon \cdot c \cdot k_1$ corrupt values. The top k_1 values can be selected by retrieving no more than $k_1 + \varepsilon \cdot c \cdot k_1$ values from the soft heap and then performing a k_1-selection (via median-of-medians) on the retrieved values. The $\varepsilon \cdot k_1$ corrupt values are reinserted into the soft heap, bringing the total insertions to $k_1 \cdot \varepsilon \cdot (1 + c)$. To retrieve the top k_2 remaining values, $k_2 + \varepsilon \cdot k_1 \cdot \varepsilon \cdot (1 + c) \in \Omega(k_2 + k_1)$

k	Standard version (seconds)	Wobbly version (seconds)
2^2	1.544×10^{-04}	1.777×10^{-04}
2^3	1.754×10^{-04}	1.468×10^{-04}
2^4	2.086×10^{-04}	1.846×10^{-04}
2^5	2.386×10^{-04}	2.046×10^{-04}
2^6	2.080×10^{-04}	1.935×10^{-04}
2^7	3.060×10^{-04}	2.672×10^{-04}
2^8	3.481×10^{-04}	3.225×10^{-04}
2^9	4.289×10^{-04}	2.978×10^{-04}
2^{10}	6.119×10^{-04}	4.087×10^{-04}
2^{11}	7.976×10^{-04}	4.585×10^{-04}
2^{12}	1.000×10^{-03}	7.263×10^{-04}
2^{13}	1.711×10^{-03}	1.189×10^{-03}
2^{14}	2.344×10^{-03}	1.465×10^{-03}
2^{16}	7.531×10^{-03}	4.890×10^{-03}
2^{15}	3.919×10^{-03}	2.578×10^{-03}
2^{17}	0.0113	9.090×10^{-03}
2^{18}	0.01741	0.01583
2^{19}	0.02777	0.02511
2^{20}	0.04904	0.04228
2^{21}	0.08572	0.07773
2^{22}	0.1623	0.1424
2^{23}	0.3274	0.234
2^{24}	0.636	0.4838
2^{25}	1.210	1.029
2^{26}	2.306	1.588
2^{27}	4.993	3.487
2^{28}	9.995	8.441
2^{29}	19.7	14.31
2^{30}	43.45	24.33

Table 2. Runtimes for standard Cartesian product tree vs wobbly Cartesian product tree with $n = 256$, $m = 5$ **and** $\alpha = 1.1$**.** The runtime is averaged over 20 iterations for the two methods. With $m = 5$ the tree only has three layers and so the super-exponential growth of the layers as they go from the leaves to the roots does not become intractable. As k becomes extremely large the ability of the wobbly tree to generate huge layers at the root without performing any selections significantly reduces the runtime resulting in a $1.786\times$ speedup.

208   values need to be popped. These top k_2 values can be retrieved in optimal $O(k_2)$ time if $k_2 \in \Theta(k_1)$.
209   Likewise, $k_3 \in \Theta(k_1 + k_2)$, and so on. Thus, for optimal, online computation the sequence of k_j values
210   must grow exponentially.

211   Rebuilding the soft heap (rather than reinserting the corrupted values into the soft heap) instead does
212   not alleviate this need for exponential growth in $k_1, k_2, \ldots$ required to achieve optimal $O(k_1 + k_2 + \cdots)$
213   total runtime. When rebuilding, each next k_j must be comparable to the size of the entire soft heap (so
214   that the cost of rebuilding can be amortized out by the optimal $\Theta(k_j)$ steps used to retrieve the next k_j
215   values). Because $c \geq 1$, the size of the soft heap is always $\geq k_1 + k_2 + \cdots + k_{j-1}$ for the selections already
216   performed, and thus the rebuilding cost is $k_1 + k_2 + \cdots + k_{j-1}$, which must be $\in \Theta(k_j)$. This likewise
217   requires exponential growth in the k_j.

218   The necessity to have exponential growth in k_j enforces the layer-ordering property on the resulting
219   values. It is the layer-ordering property which guarantees that a proposal scheme such as that in Kaplan
220   *et al.* does not penetrate to great depth in the combinatorial heap, which could lead to exponential time
221   complexity when $c > 1$. In this manner, the $k_1, k_2, \ldots$ values can be seen to form layers of a LOH, which
222   would not require retrieving further layers before the current extreme layer has been exhausted. Thus,
223   both FastSoftTree and Cartesian product trees have to have pairwise selection nodes which can generate
224   LOHs one layer at a time. FastSoftTree nodes require a soft heap to generate the LOHs where Cartesian
225   product tree nodes generate layers in a straight-forward manner.

226   This method has already proved to be beneficial in generating the top k isotopologues of chemical
227   compounds, but it is not limited to this use-case. It is applicable to fast algorithms for inference on random
228   variables $Y = X_1 + X_2 + \cdots + X_m$ in the context of graphical Bayesian models. It may not generate a value
229   at every index in a max-convolution, but it may generate enough values fast enough to give a significant
230   result.

Code availability

232   A `C++` implementation can be found in
233   `https://bitbucket.org/seranglab/cartesian-product-tree/` under the MIT license.
234   The code is free for academic and commercial use.

REFERENCES

236   Bremner, D., Chan., T. M., Demaine, E. D., Erickson, J., Hurtado, F., Iacono, J., Langerman, S., and
237   Taslakian, P. (2006). Necklaces, convolutions, and $X + Y$. In *Algorithms–ESA 2006*, pages 160–171.
238   Springer.

239   Bussieck, M., Hassler, H., Woeginger, G. J., and Zimmermann, U. T. (1994). Fast algorithms for the
240   maximum convolution problem. *Operations research letters*, 15(3):133–141.

241   Chazelle, B. (2000). The soft heap: an approximate priority queue with optimal error rate. *Journal of the*
242   *ACM (JACM)*, 47(6):1012–1027.

243   Frederickson, G. N. and Johnson, D. B. (1982). The complexity of selection and ranking in $X + Y$ and
244   matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208.

245   Fredman, M. L. (1976). How good is the information theory bound in sorting? *Theoretical Computer*
246   *Science*, 1(4):355–361.

247   Johnson, D. and Mizoguchi, T. (1978). Selecting the $k^t h$ element in $x + y$ and $x_1 + x_2 + \ldots x_m$. *SIAM*
248   *Journal on Computing*, 7(2):147–153.

249   Kaplan, H., Kozma, L., Zamir, O., and Zwick, U. (2019). Selection from heaps, row-sorted matrices and
250     $X + Y$ using soft heaps. *Symposium on Simplicity in Algorithms*, pages 5:1–5:21.

251   Kreitzberg, P., Lucke, K., and Serang, O. (2020a). Selection on $X_1 + X_2 + \cdots + X_m$ with layer-ordered
252   heaps. *arXiv preprint arXiv:1910.11993*. In review.

253   Kreitzberg, P., Pennington, J., Lucke, K., and Serang, O. (2020b). Fast exact computation of the k most
254   abundant isotope peaks with layer-ordered heaps. *Analytical Chemistry*, 92(15):10613–10619.

255   Kreitzberg, P. and Serang, O. (2020). Toward a unified approach for solving probabilistic linear diophantine
256   equations.

257   Pennington, J., Kreitzberg, P., Lucke, K., and Serang, O. (2020). Optimal construction of a layer-ordered
258   heap.

259   Pfeuffer, J. and Serang, O. (2016). A bounded p-norm approximation of max-convolution for sub-quadratic
260   Bayesian inference on additive factors. *Journal of Machine Learning Research*, 17(36):1–39.

261   Serang, O. (2015). A fast numerical method for max-convolution and the application to efficient max-

262   product inference in Bayesian networks. *Journal of Computational Biology*, 22:770–783.

263   Serang, O. (2020). Optimal selection on X + Y simplified with layer-ordered heaps. *arXiv preprint*

264   *arXiv:2001.11607*.