On the Connection between Hamming Codes, Heapsort and other Methods

Shmuel T. Klein

Department of Computer Science

Bar Ilan University

Ramat-Gan 52900, Israel

tomi@cs.biu.ac.il

Abstract: A connection between Hamming codes and Heapsort is shown, namely, that they can both be derived, the first from straightforward error correcting codes and the second from a simple sorting method, using almost identical derivation processes. It is then demonstrated that the same process can work for other well-known methods. This might trigger similar derivations in the future.

1. Introduction

A major tool in the design of algorithms is looking for connections between problems that might at first sight seem completely unrelated and to exploit the newly discovered links to otherwise unexpected improvements. In this note we show a connection between Hamming codes [2] and Heapsort [3], both well established CS milestones dating back to the 1960s, which, to the best of our knowledge, has not been noticed before. The same approach is then shown to yield Binary search and time/space tradeoffs for the solution of the Subset Sum and similar problems. It seems, *a priori*, that there is nothing to be gained from pointing to such connections. Nevertheless, presenting a method as belonging to a chain of similar ones, and connecting different topics might provide didactic help for students trying to remember one based on the other. Showing, moreover, that all are the ultimate step of an almost identical derivation process, starting on the one hand with a very simple error-correcting code, and on the other hand with a just as simple sorting or searching methods, might possibly yield interesting similar derivations in the future.

The general paradigm starts with a trivial solution to a given problem to be solved for a set on size n. An improvement is achieved by reorganizing the data into sets of size $\sqrt{n}$, and more generally, into k layers of sets of size $n^{1/k}$. The ultimate step of the derivation, yielding best performance, corresponds to $k = \log n$. The next sections present the details of these derivations for four different problems.

2. A sequence of Error Correcting Codes

Given are n data bits, that should be transmitted over a noisy channel which may corrupt them. We consider here how to deal with a *single* erroneous bit, corresponding to a scenario of a so low probability p for an error, that the possibility of two or more errors might be neglected.

If all that is needed is only the knowledge about the occurrence of an error, without its exact location, then this can be achieved by adjoining a single bit, often called *parity bit*, consisting of the XOR of the n data bits. If in addition to detection, one needs also error correction, a simple way to protect the data is to transmit every bit twice. If the additional bits are also vulnerable, one would even need a third copy of every bit to assure that the original data is recovered, using the majority rule for every bit-triple. But one can do better than adding $2n$ check-bits.

Organize the n data bits into a square with side length $\sqrt{n}$ bits, and add a parity bit for each column and each row. A single error in any of the n data bits can then be located by intersecting the row and the column corresponding to the only affected parity bits. We thus got error correction at the price of additional $2\sqrt{n}$ bits.

A further step would then be to rearrange the data into a cube of side length $\sqrt[3]{n}$. Three vectors of parity bits are then needed, each of length $\sqrt[3]{n}$ and corresponding to a partition of the cube into planes according to another dimension. A single error in one of the data bits will affect exactly three of the parity bits, and the location of the erroneous bit is at the intersection of the three corresponding planes. With such a layout, only $3\sqrt[3]{n}$ additional bits are needed.

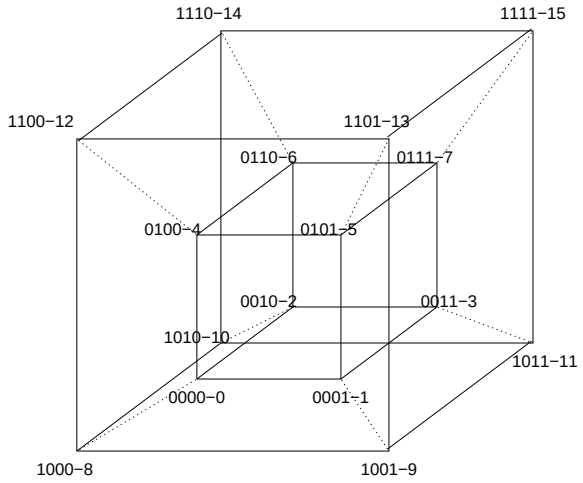
For the general case, the data bits are arranged into a k-dimensional hypercube with

side length $n^{1/k}$. The number of vectors of parity bits will be k, and each bit will be the XORing of all the data bits of a $(k-1)$-dimensional hyperplane, all the bits of the same vector corresponding to the $n^{1/k}$ disjoint $(k-1)$-dimensional hyperplanes forming the original hypercube, and each of the k vectors corresponding to such a partition in a different dimension. The number of parity bits is $kn^{1/k}$. Figure 1 shows a 4-dimensional hypercube with side length 2, and its partition into two 3-dimensional cubes in each of the four dimensions: left–right, front–back, upper–lower, inner–outer.

What value should be chosen for k? For a fixed n, the function $f(k) = kn^{1/k}$ gets its minimum value $e \ln n$ at $k = \ln n$, so the optimal side length of the hypercube should be e. Since this has to be rounded to an integer, it would seem that the preferred choice would be $k = \log_3 n$, yielding a value of $3 \log_3 n$, the relative loss being only $\frac{3}{e \ln 3}$, just 0.47% above the optimum. The resulting hypercube would be partitioned into three $(k-1)$-dimensional hyperplanes in each dimension. We shall however prefer to round the optimal side length e down to 2, which gives $k = \log_2 n$ dimensions and $2 \log_2 n$ parity bits, incurring a loss of $\frac{2}{e \ln 2}$ or 6.1% over the optimum. A side length of 2 is also the minimum possible value.

Figure 1 is the corresponding hypercube for $n = 16$, each vertex corresponding to one of the n data bits and being labeled by the 4-bit binary representation of its index i, $0 \le i < 2^4$; vertices are connected by edges if and only if they correspond to numbers differing exactly by a single bit in their binary representations. The parity bits appear next to the cube in the figure, the left–right dimension corresponding to the first (rightmost) bit in the binary representation, that is, left is the XORing of all the data bits having a **0** in the first bit of the binary representation of their index, and right is the XORing of the complementing set of those with **1** in the first bit. Similarly, the front–back dimension corresponds to the second bit, so front is the XORing of the bits indexed 0, 1, 4, 5, 8, 9, 12 and 13 (**0** in the second bit), and back is the XORing of the complementing set. Finally, the upper–lower dimension corresponds to the third bit, and inner–outer to the fourth (leftmost) bit.

The reason for rounding e down to 2 rather than up to 3 is the fact that an error in a single data bit will have an effect on exactly one of the parity bits in each dimension, but since we are left with a binary choice in each dimension, one of the bits will be redundant,



left	0,2,4,6,8,10,12,14	right	1,3,5,7,9,11,13,15
front	0,1,4,5,8,9,12,13	back	2,3,6,7,10,11,14,15
lower	0,1,2,3,8,9,10,11	upper	4,5,6,7,12,13,14,15
inner	0,1,2,3,4,5,6,7	outer	8,9,10,11,12,13,14,15

FIGURE 1: *Layout of 16 data bits in a 4-dimensional hypercube with 8 parity bits*

so that the number of necessary check bits will be only $\log_2 n$. For example, suppose the data bit indexed 13 is flipped. This will have an effect on all the parity bits that include 13 in their lists: right, front, upper and outer. Rearranging the bits from left to right and reconverting to the corresponding **0** and **1** values, one gets: outer,upper,front,right = **1101**, which is the binary representation of 13, the index of the wrong bit.

In fact, it would suffice to keep only the parity bits corresponding to **1** values, that is right, back, upper and outer. This corresponds to indicating only the 1-bits of the index of the wrong bit, which is enough to recover it, unless its index is 0 and thus has no 1-bits. In addition, one also needs to deal with the case in which no error has occurred. To solve both problems, one might reduce the set of data bits and index them only by the non-zero values. This method of adding $\log_2 n$ parity bits defined according to the binary representation of the indices of the n data bits is the well-known *Hamming code* [2]. Hamming also suggested to store the parity bits interleaved with the data bits at the positions whose indices are powers of 2.

We now turn our attention to sorting algorithms and show that essentially the same derivation used above leads from a trivial sorting method to heapsort.

3. A sequence of Sorting Algorithms

Given are n data elements, which should be rearranged into a sorted sequence. A simple method would be to repeatedly find and remove the largest element of the remaining set and store the removed elements in order of their processing. Selection sort, Insertion sort and Bubble sort are all variants of this basic idea, which runs in time $\theta(n^2)$. But one can do better.

Organize the set A_1 of the n data elements into $\sqrt{n}$ subsets of $\sqrt{n}$ each, find and remove a maximal element of each of these subsets and define the set A_2 of these $\sqrt{n}$ maxima. The maximal element of the entire set can thus be found by finding a maximum of A_2, so $\theta(n)$ comparisons are needed. But the second element can be found by retrieving the largest element only from the subset of A_1 from which the first element originated, and then again from the set A_2. This can be done with $2\sqrt{n}$ comparisons, and the same is then true also for the subsequent elements, giving a total of $2n\sqrt{n}$.

A further step is then to rearrange the data into $n^{2/3}$ subsets of size $\sqrt[3]{n}$, extract the $n^{2/3}$ maximal elements into a set A_2, and then partition A_2 itself into $\sqrt[3]{n}$ subsets of size $\sqrt[3]{n}$ each. Retrieving the maxima of these latter sets will define a set A_3 on the third level, the maximal element of which is the global maximum. The number of comparisons to get this maximum is thus $\theta(n)$ as before, but only $3\sqrt[3]{n}$ comparisons are needed for the second and subsequent elements, for a total of $3n\sqrt[3]{n}$.

For the general case, a tree with k layers representing the sets $A_1, A_2, \ldots, A_k$ is constructed, and the set A_i is partitioned into $n^{(k-i)/k}$ subsets of size $n^{1/k}$. Taking the maximal element of each one of the subsets of A_i forms the set A_{i+1} of the next layer. The overall maximal element is found in $\theta(n)$ steps, and each of the subsequent ones in $kn^{1/k}$. Figure 2 shows an example data set of size $n = 8$, partitioned into 3 layers, with each subset of size $\sqrt[3]{n} = 2$. Elements that have been chosen as maxima in their subsets and have been removed appear in light grey so that their path in the tree may be retraced.

The function $kn^{1/k}$ is the same we encountered previously for the error-correcting codes. To continue the analogy, we shall choose as above $k = \log_2 n$, corresponding to a partition of all the layers into subsets of size 2. These subsets appear in the boxes in Figure 2. If

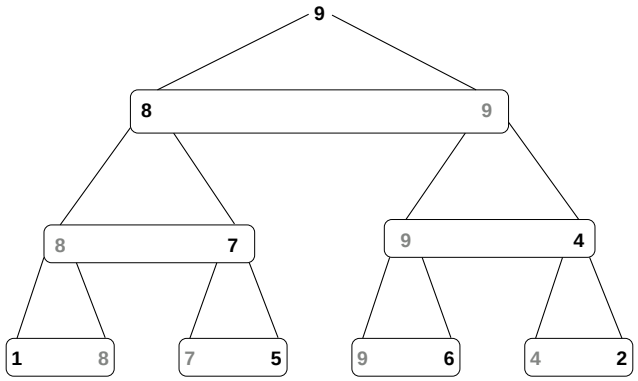


FIGURE 2: *Partition of 8 data elements into subsets of size* $\sqrt[3]{8}$

one considers the edges that have been added in Figure 2, one gets a complete binary tree of depth $\log_2 n$, in which the element in each node is at least as large as the element in any of its descendants — this is the definition of a *heap*. Finding the maximal element, which is equivalent to building the heap, can be done in time n, but then repeatedly removing the maximal elements of the remainder of the set requires only one comparison for each layer, that is $\log n$ comparisons per element, for a total of $O(n \log n)$. Building a heap and maintaining the heap property while removing the elements in order is the basic idea of *heapsort*.

Note that the original heapsort algorithm exploits the fact that the elements are organized into a full binary tree to simulate the tree behavior by means of an array. This is done by replacing the removed maximal element at the root with the last element of the list and then updating the heap structure top-down. The variant one gets from the derivation process above uses bottom-up updates of the tree structure, so there might be "holes" in the tree. The array implementation is unlikely to be possible, though the algorithm stays essentially the same. To refer to the example in Figure 2, after removing 9, the subset to be processed is the one containing 6, which is promoted two levels up; there, 6 is compared with 8, which is the new maximum. The next step starts at the subset from which 8 originated, that is, the one with element 1. We see that certain sets are emptied before we are done. In the original top-down approach, after removing 9, we would replace it by 2 and let it percolate down.

4. A sequence of Search procedures

Given is a sorted array of n elements and a number X to be searched for in the array. The simple solution would call for a sequential search, call it a 1-layer search, requiring $n-1$ comparisons, but one can do better. In a 2-layer search, partition the array into $\sqrt{n}$ subsets of size $\sqrt{n}$ each, compare X to the $\sqrt{n}-1$ elements separating the subsets until the first part is found to which X ought to belong to, and then search sequentially only within this part. This can be done in at most $2(\sqrt{n}-1)$ comparisons.

A further step is then to devise a 3-layer search, by rearranging the data into $\sqrt[3]{n}$ subsets of size $n^{2/3}$, locating the proper subset containing X in at most $\sqrt[3]{n}-1$ comparisons and continuing within this subset as explained above in a 2-layer search. This yields up to $3(\sqrt[3]{n}-1)$ comparisons. For the general k-layer search case, the array is partitioned into $n^{1/k}$ subsets of size $n^{(k-1)/k}$. The subset containing X is located, and a $(k-1)$-layer search is performed within it, with a total running time of $k(n^{1/k}-1)$.

The number of comparisons within a given layer decreases with increasing k, and one is left with a single comparison if $n^{1/k}=2$, that is $k=\log_2 n$, as above. The resulting method compares X in the highest level to the middle element of the array, and continues to the middle element of the located half in the lower levels, with up to $\log_2 n$ comparisons; this is well known as *binary search*.

5. A sequence of time/space tradeoffs for the Subset Sum problem

Consider a set $A=\{a_1,\ldots,a_m\}$ of m integers and a constant B. The *Subset Sum* Problem is whether there exists a subset $A'\subseteq A$, whose elements sum up to B, or in other terms, does there exist a binary vector $X=x_1x_2\cdots x_m$, with $x_i\in\{0,1\}$, such that $\sum_{i=1}^{m} x_i a_i = B$. This problem is well-known to be NP-complete [1].

For a scenario in which the set A is constant and many queries with different values of B have to be answered, the time complexity can be reduced at the price of additional space. In the extreme case, a table $T_{1,1}$ with $n=2^m$ entries is prepared, each containing the sum of a different subset of A. This table is sorted off-line, and can then be (binary)

searched in time $O(m)$ for each new value of B. The space, however, is now $\theta(n) = \theta(2^m)$, which is exponential in the number of elements in A and that might be prohibitive.

A sequence of time/space tradeoffs solving this problem may be derived with decreasing space complexities, at the price increasingly worse processing times. Following the analogy to the above problems, consider now the space complexity as the main target. The straightforward solution with a single table of size $n = 2^m$ might be the fastest, however, it is the worst from the space point of view. But one can do better, see, e.g., [4].

Build two tables with $\sqrt{n}$ entries each, the first, $T_{2,1}$, corresponding to all the partial sums of the first half of the elements $\{a_1, \ldots, a_{\frac{m}{2}}\}$, the other, $T_{2,2}$, to the partial sums of the complementing set $\{a_{\frac{m}{2}+1}, \ldots, a_m\}$. Sort the tables off-line into non-decreasing order and initialize a pointer p_1 to the smallest (first) element in $T_{2,1}$, and another pointer p_2 to the largest (last) element in $T_{2,2}$. The following simple loop then finds a target value B in time linear in the size of the tables, that is, in time $O(2^{\frac{m}{2}}) = O(\sqrt{n})$, but reducing the space needed for the tables to $2\sqrt{n}$:

```
found ⟵ false
while not found and p1 < 2^{m/2} and p2 > 0 do
    if T2,1[p1] + T2,2[p2] = B then found ⟵ true
    else    if T2,1[p1] + T2,2[p2] < B then p2 ⟵ p2 − 1
            else                            p1 ⟵ p1 + 1
```

A further step is then to partition the set A into three parts of equal size and to prepare three tables of the partial sums $T_{3,1}$, $T_{3,2}$ and $T_{3,3}$, each of size $2^{m/3} = \sqrt[3]{n}$. To search for a target value B, scan the first table, and for each value $b \in T_{3,1}$, perform a 2-table search for $B - b$ in the sorted tables $T_{3,2}$ and $T_{3,3}$ as above, in time $\sqrt[3]{n} \cdot O(\sqrt[3]{n}) = O(n^{2/3})$, and overall space $3n^{1/3}$.

For the general case, k tables $T_{k,1}, \ldots, T_{k,k}$ are prepared, table $T_{k,i}$ holding the $2^{m/k} = n^{1/k}$ partial sums of the elements of $\{a_{\frac{i-1}{k}+1}, \ldots, a_{\frac{i}{k}}\}$, for $1 \leq i \leq k$. Only the last two tables need be sorted. To search for a target value B, scan the first table, and for each

value $b \in T_{k,1}$, perform a $(k-1)$-table search for $B - b$ in the tables $T_{k,2}$ to $T_{k,k}$ as above, in time $n^{1/k} \cdot O(n^{(k-2)/k}) = O(n^{(k-1)/k})$, and overall space $kn^{1/k}$. It should be noted that solutions with better tradeoffs have been suggested for this problem [5].

The lowest possible space complexity is obtained for $k = m = \log_2 n$, as above. There are then m tables of size 2, each corresponding to a single variable a_i which either participates in the chosen subset, or not. In this case, the algorithm obtained at the end of this derivation chain is just *exhaustive search*.

6. Concluding remarks

Are there other problems that could be tackled by the same approach as those above? Maybe a similar derivation could lead to a hitherto unknown improved solution. Another line of research could investigate relations between other sets of algorithms that follow a different derivation altogether.

References

[1] Garey M.R., Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, San Francisco (1979).

[2] Hamming R.W., *Coding and Information Theory*, second edition, Prentice-Hall, Englewood Cliffs, NJ (1986).

[3] Williams J.W. J., Algorithm 232 – Heapsort, *Communications of the ACM* **7**(6) (1964) 347-348.

[4] Horowitz E, Sahni S., Computing partitions with applications to the Knapsack problem, *Journal of the ACM*, **21** (1974) 277–292.

[5] Schroeppel R., Shamir A., A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-complete problems, *SIAM Journal of Computing* **10**(3) (1981) 456–464.