Pesquisa Operacional (2015) 35(2): 423-437
© 2015 Brazilian Operations Research Society
Printed version ISSN 0101-7438 / Online version ISSN 1678-5142
www.scielo.br/pope
doi: 10.1590/0101-7438.2015.035.02.0423

SOME ILLUSTRATIVE EXAMPLES ON THE USE OF HASH TABLES

Lucila Maria de Souza Bento[1], Vinícius Gusmão Pereira de Sá[1*]
and Jayme Luiz Szwarcfiter[2]

Received February 28, 2013 / Accepted January 13, 2015

ABSTRACT. Hash tables are among the most important data structures known to mankind. Through hashing, the address of each stored object is calculated as a function of the object's contents. Because they do not require exorbitant space and, in practice, allow for constant-time dictionary operations (insertion, lookup, deletion), hash tables are often employed in the indexation of large amounts of data. Nevertheless, there are numerous problems of somewhat different nature that can be solved in elegant fashion using hashing, with significant economy of time and space. The purpose of this paper is to reinforce the applicability of such technique in the area of Operations Research and to stimulate further reading, for which adequate references are given. To our knowledge, the proposed solutions to the problems presented herein have never appeared in the literature, and some of the problems are novel themselves.

Keywords: hash tables, efficient algorithms, practical algorithms, average-time complexity.

1 INTRODUCTION

Choosing the most appropriate algorithm for a problem requires awareness with a number of data structures and general computational techniques that may be used to actually implement the algorithm in some programming language. In a paper entitled "Data Structures and Computer Science Techniques in Operations Research" [8], Fox discussed some applications of existing algorithms and techniques – such as heaps, divide-and-conquer and balanced trees – to well-known operational research problems including incremental allocation, event lists in simulations, and the kth shortest path. The present paper pursues a similar goal, albeit focusing on a specific technique that has over the years gained increasing attention in Computer Science in general: *hashing*.

*Corresponding author.
[1]Department of Computer Science, Institute of Mathematics, Federal University of Rio de Janeiro, 21941-590 Rio de Janeiro, RJ, Brazil. E-mails: lucilabento@ppgi.ufrj.br; vigusmao@dcc.ufrj.br
[2]Institute of Mathematics, NCE and COPPE, Federal University of Rio de Janeiro, 21941-916 Rio de Janeiro, RJ, Brazil. E-mail: jayme@nce.ufrj.br

The hashing technique provides a way to store and retrieve data that is efficient in both time and space. Basic theory can be found, for instance, in [4]. Numerous recent publications deal with the technical aspects of hashing implementations, exploring new ways to solve its intrinsic problems such as handling collisions, devising nearly ideal hash functions and improving their performance in a number of ways. Among them, we cite the work on perfect hash functions and dynamic perfect hashing in [7, 5], the cuckoo hashing technique [25], the work on uniform hashing in [24] and the ingenious use of random hypergraphs in perfect hashing described in [2]. A recent trend is the use of machine learning tools to devise good hash functions [26, 17, 28, 9, 23, 20]. Other recent publications include [12, 1, 13, 19].

The hashing technique is widely used nowadays in database indexing, compiler design, caching, password authentication, software integrity verification, error-checking, and many other applications. Still, a seemingly uncountable number of further problems do allow for efficient, practical algorithms through the use of hashing.

Some aspects of hashing have already been discussed in the Operations Research literature, and hashing-based improvements for known Operations Research problems are indeed not rare. See, for instance, [14, 29, 3]. We believe, however, that the full potential of hashing has yet to be explored in the field, allowing for plenty of further improvements.

After briefly revisiting some concepts related to hashing (Section 2), we focus on a number of applications (Sections 3.1–3.5), illustrating how it can be successfully employed in conceiving efficient solutions to problems that might not suggest its use at first glance. Though it is certainly possible to find several much more complicated examples where hash tables may come in handy in a less-than-obvious way, we selected but a few simple problems. We believe the examples provided are sufficiently illustrative and dispense with otherwise naive attempts to categorize exhaustively the cases in which the hashing technique could and should be considered.

The problems presented in Sections 3.2, 3.3 and 3.5 are original. Likewise, the solutions proposed in this text are all, to our knowledge, yet unpublished.

2 HASHING

The central idea of the hashing technique is to use a *hash function* to translate a unique identifier (or *key*) belonging to some relatively large domain into a number from a range that is relatively small [18]. Such number corresponds to the memory address (basically the position in a "table", the *hash table*) in which the *object* – usually a (*key, value*) entry, where the value can be of any data type – should be stored. Owing to the key-value mapping it provides, a hash table is usually called a *hash map*. Sometimes, though, the object to be stored is but the key itself. In these cases, the hash table is often referred to as a *hash set*.

The implementation of a hash table is usually based on an underlying array where the objects will actually reside. In other words, the hash functions' images are indexes of positions in an array. Ideally, a hash function should map each distinct key onto a distinct hash table index. This can be accomplished, in some cases, by employing advanced techniques such as *(minimal) perfect hashing* [5], where injective hash functions are used. When the whole set of keys to be stored is not known beforehand, it may not be possible to define such a perfectly injective function, hence the same position in a hash table may be assigned to different keys in a situation known as *collision*.

When collisions occur, it is necessary to store the objects with colliding keys in alternative positions [16]. Among the most widely adopted methods for resolving collisions is the method of *chaining*, in which the hash table positios are regarded as *buckets*, each one containing a pointer to a linked list (or other data structure) where the colliding objects will be located. It is a simple exercise to show that the average number of elements in each bucket is equal to the *load factor* of the hash table, defined as $\alpha = n/m$ for a hash table with m positions and n stored objects.

The actual complexities of searches, insertions and deletions depend on the choice of the hash function (as well as on the load factor). Because good hash functions are known for the vast majority of data types, the *simple uniform hashing assumption* (SUHA), whereby the distribution of keys into buckets approaches a uniform distribution, is often a reasonable premise. The average case analysis in algorithms that involve hashing can therefore rely on the SUHA and still provide a fairly accurate indicator of their practical performances.

By using a hash function which disperses the keys with near uniform distribution along the buckets, and provided the load factor never exceeds a certain constant (which is achieved simply by dimensioning the hash table proportionally to the number of objects to be stored), it can be shown that those operations are performed in $O(1)$ time, on average [4]. Even though a "bad" instance may demand some worst-case performance that is quite poorer than the average, the probability of its occurrence is usually extremely low, resulting from an unbelievably unfortunate choice of the hash function.

It is certainly not in the scope of this text the design of hash functions that are most appropriate to each case, a subject that requires statistical tools and specific study [15]. The algorithms presented in this paper, however, do not depend upon complicated, custom-tailored hash functions. Backed by the SUHA, we assume the existence of a near-uniform hash function that maps each key onto a number in the range $[0, m-1]$, where m is the size of the hash table. Indeed, the vast majority of modern programming languages provides efficient inbuilt implementations of hash tables, so the basic hashing operations, and collision handling, and even the choice of hash functions become totally transparent to the programmer. In other words, it is *not* necessary to even think about the low-level details of the technique — the programmer is free to concentrate on the algorithm itself. Such is the focus of this paper.

In the next sections, we look into some examples of hashing in action.

3 ILLUSTRATIVE EXAMPLES

3.1 The Pair Sum Problem

Given an integer k and a list A of n unordered integers, find out whether there is a pair $\{x, y\} \subset A$ such that $x + y = k$. This is a particular case of the well-known Subset Sum Problem, in which there are no restrictions on the size of the subsets summing k.

Unlike the Subset Sum Problem, which is NP-hard and admits a number of interesting approximation algorithms [21], the Pair Sum Problem is clearly polynomial. For a naive solution, just check all pairs (x, y) of elements of A. If $x + y$ equals k, return *yes* and stop. Although no extra space is required, the number of additions and comparisons to be performed is $O(n^2)$ in the worst case.

We want to do better. First note that, if we select an element x of A, we can look for its complement $y = k - x$ in A, and the problem is now a search problem. If, for each x, we need to traverse the whole list to check whether it contains its complement, then our algorithm still demands $O(n^2)$ time. A better alternative would be to first sort the list A, and then, for each $x \in A$, look for its complement using binary search, whereupon our time complexity would be $O(n \log n)$.

In the pursuit of a linear-time algorithm, we could try and use a boolean array to represent the elements of A: the ith position of the array contains a '1' (or 'true') if and only if $i \in A$. Each complement could then be looked for in constant time. However, such *direct addressing* approach, in which the address of the data related to some key i is exactly i, has a serious caveat: the space allocated for object storage must be at least as large as the maximum possible key. In other words, the length of our boolean array would depend on the *maximum element* w of A, not on the size n of A. Since w can be represented by $\log_2 w$ bits, that means exponential space.

We can use a hash set to store the elements of A. The amount of space we need is that to store n integers (scattered along their respective buckets) plus an underlying array with n/α positions. For a constant load factor α, that means $O(n)$ space, while the average time of all dictionary operations is as good as $O(1)$ under the simple uniform hashing assumption. The overall expected time of such algorithm –"for each element x in the list, if $k - y$ is in the (initially empty) hash set H, then return *yes*, otherwise insert x in H"– is therefore also $O(n)$, corresponding to the $O(n)$ insertions and lookups in the hash set, in the worst case.

A similar approach can be used to determine the intersection of two lists A, B. Create a hash set H and populate it using the elements of one of the lists, say A. Next, for each element $x \in B$, look for x in H, for an expected $\Theta(|A| + |B|)$ overall time.

3.2 The Sum of Functions Problem

Given a set A and a function f, whose domain contains A and which can be calculated in time $O(1)$, find all non-trivial quadruples (x, y, z, w) of elements $x, y, z, w \in A$ such that $f(x) + f(y) = f(z) + f(w)$.

For a trivial solution, we may simply test each quadruple of distinct elements of the set A. We can do this in $\Theta(n^4)$ time, where $n = |A|$.

A more efficient solution can be achieved by sorting. Pick all pairs $(x, y) \subset A$, put them on a list L, and sort them by $d_{x,y} = f(x) + f(y)$. Now, all pairs of elements whose images under f sum up to a same value d will constitute a contiguous sublist of L. Therefore, they can be easily

```
input  : array $A$, function $f$
output: all $(x, y, z, w) \subset A$ satisfying $f(x) + f(y) = f(z) + f(w)$
$H \leftarrow$ a hash map with size $|A|^2$
foreach $(x, y) \subset A$ do
    $d \leftarrow f(x) + f(y)$
    if $d$ is a key stored in $H$ then
        $S_d \leftarrow H.get(d)$  // the value stored with $d$ in $H$
    else
        $S_d \leftarrow \{(x, y)\}$
        insert $(d, S_d)$ in $H$
    end
    $S_d \leftarrow S_d \cup \{(x, y)\}$
end
foreach $(d, S_d) \in H$ do
    foreach $(x, y) \in S_d$ do
        for each $(z, w) \in S_d \setminus \{(x, y)\}$ do
            print $(x, y, z, w)$
        end
    end
end
```

Algorithm 1 – *Sum_Functions* (A, f).

combined to produce the intended quadruples in constant time per quadruple. Owing to sorting, the overall time complexity of this approach is $O(n^2 \log n + q)$, in the worst case, where q is the number of quadruples in the output.[3]

Now we look into a hashing-based approach. We start with an empty hash map H. We consider each unordered pair (x, y) of distinct elements of A, obtaining $d = f(x) + f(y)$, one pair at a time. If d is *not* a key stored in H, we insert the (key, value) pair (d, S_d) in H, where S_d is a collection (which can be implemented as a linked list, a hash set or whatever other structure) containing only the pair (x, y) initially. On the other hand, if d is already stored in H, then we simply add (x, y) to the non-empty collection S_d which d already maps to in H. After all pairs of distinct elements have been considered, our hash map will contain non-empty collections S_d whose elements are pairs (x, y) satisfying $f(x) + f(y) = d$. For each non-unitary collection S_d, we combine distinct pairs $(x, y), (z, w) \in S_d$, two at a time, to produce our desired quadruples (x, y, z, w).

The pseudocode of the proposed hash-based solution is given as Algorithm 1.

As for the time complexity of the algorithm, each key d can be located or inserted in H in $O(1)$ average time, and each of the $\Theta(n^2)$ pairs (x, y) of elements of A can be inserted in some list S_d in $O(1)$ time (assuming, for instance, a linked list implementation for S_d). Thus, the whole step of populating H can be performed in $\Theta(n^2)$ time on average. Now, each of the q quadruples that satisfy the equality can be obtained, by combining pairs in a same list, in $O(1)$ time. Thus, the whole algorithm runs in $\Theta(n^2 + q)$ expected time.

[3]The authors would like to thank the anonymous referee who suggested this solution.

If we tried to achieve the same performance by using an array and direct addressing, such array would have to be twice as large as the maximum element in the image of f. The use of hashing avoids that issue seamlessly.

3.3 The Permutation Bingo Problem

In the standard game of bingo, each of the contestants receives a card with some integers. The integers in each card constitute a subset of a rather small range $A = [1, a]$. Then, subsequent numbers are drawn at random from A, without replacement, until all numbers in the card of a contestant have been drawn. That contestant wins the prize.

In our variation of the game, the contestants receive an empty card. Subsequent numbers are drawn at random from some unknown, huge range $B = [1, b]$, *with* replacement, forming a sequence S. The contestant who wins the game is the one who first spots a set $W = \{x, y, z\}$ of three distinct numbers such that all $3! = 6$ permutations of its elements appear among the subsequences of three consecutive elements of S. As an example, the sequence

$$S = \big(2, 19, 4, 1, 100, 1, 4, 19, 1, 4, 1, 19, 100, 192, 100, 4, 19, 2, 1, 19, 4\big)$$

is a minimal winning sequence. There is a set, namely $W = \{1, 4, 19\}$, such that every permutation of W appears as three consecutive elements of S. Even though the set B is huge, the distribution of the drawing probability along B is not uniform, and the game finishes in reasonable time almost surely.

We want to write a program to play the permutation bingo. Moreover, we want our program to win the game when it plays against other programs. In other words, we need an efficient algorithm to check, after each number is added to S, whether S happens to be a winning sequence.

One possible approach would be as follows. Keep S in memory all the time. After the n-th number is added, for all $n \geq 8$ (a trivial lower bound for the size of a winning sequence), traverse $S = (s_1, s_2, \ldots, s_n)$ entirely for each set $R_i = \{s_i, s_{i+1}, s_{i+2}\}$ of three consecutive distinct elements (with $1 \leq i \leq n - 2$), checking whether all permutations of R_i appear as consecutive elements of S. Checking a sequence of size n this way demands $\Theta(n^2)$ time in the worst case.

We can do better by performing a single traversal of S, during which we add each subsequence of three consecutive distinct elements of S into a bin. To make sure that two permutations go to the same bin if and only if they have the same elements (in different orders), we may label each bin using the ascending sequence of its elements. Now, for each subsequence of three distinct elements in S, we compute the label of its bin, find the bin, and place the subsequence therein, making sure we do not add the same subsequence twice in a bin. Whenever a bin has size six, we have a winner. We can implement each bin labeled (x, y, z), with $x < y < z$, using an array of integers, or a linked list, or a hash set, or even a bit array if we care to compute the position of any given permutation in a total order of the permutations of those elements. As a matter of fact, the way we implement each bin is not important, since its size will never exceed six. Moreover, rather than building the bins from scratch by traversing the whole S each time a new element

s_n is added, we can keep the bins in memory and proceed to place only each freshly formed subsequence (s_{n-2}, s_{n-1}, s_n) into the appropriate bin – an approach which also allows us to keep but the two last elements of S in memory.

We still have a big problem, though. Given a label, how to find the bin among the (possibly huge number of) existing bins? If we maintain, say, an array of bins, then we will have to traverse that entire array until we find the desired bin. It is not even possible to attempt direct addressing here, for the range B of the numbers that can be drawn is not only unknown but also possibly huge – let alone the number of subsets of three such numbers! Even if we did know B, there would probably not be enough available memory for such an array. We know, however, that the game always ends in reasonable time, owing to an appropriate probability distribution along B enforced by the game dealer. Thus, the actual number of bins that will *ever* be used is small, and available memory to store only *that* number of bins shall not be an issue. We can therefore keep the bins in a linked list, or even in an array without direct addressing. When the sequence has size n, such list (or array) of bins will have size $O(n)$, and the time it takes to traverse it all and find the desired bin will also be $O(n)$.

The use of hashing can improve the performance even further. We use a hash map where the keys are the bin labels and the values are the bins themselves (each one a list, or bit array etc.), so every time a new element is added, we can find the appropriate bin and update its contents in expected constant time. Since we do not know the number of bins that will be stored in our hash map throughout the game, and we want to keep the load factor of the hash map below a certain constant, we implement the hash map over an initially small dynamic array and adopt a geometric expansion policy [10] to resize it whenever necessary. By doing this, the amortized cost of inserting each new bin is also constant, and the algorithm processes each drawn number in overall $O(1)$ expected time.

3.4 The Ferry Loading Problem

The Ferry Loading Problem (FLP) is not unknown to those used to international programming competitions. It appears in [27] in the chapter devoted to dynamic programming, the technique employed in the generally accepted, pseudo-polynomial solution. By using a clever backtracking approach aided by hashing, we obtain a simpler algorithm which is just as efficient, in the worst case. As a matter of fact, our algorithm may demand significantly less time and space, in practice, than the usual dynamic programming approach.

There is a ferry to transport cars across a river. The ferry comprises two lanes of cars throughout its length. Before boarding on the ferry, the cars wait on a single queue in their order of arrival. When boarding, each car goes either to the left or to the right lane at the discretion of the ferry operator. Whenever the next car in the queue cannot be boarded on either lane, the ferry departs, and all cars still in the queue should wait for the next ferry. Given the length $L \in \mathbb{Z}$ of the ferry and the lengths $\ell_i \in \mathbb{Z}$ of all cars $i > 0$ in the queue, the problem asks for the maximum number of cars that can be loaded onto the ferry.

A brute force approach could be as follows. Starting from $k = 1$, attempt all 2^k distributions of the first k cars in the queue. If it is possible to load k cars on the ferry, proceed with the next value of k. Stop when there is no feasible distribution of cars for some k. Clearly, if the number n of cars loaded in the optimal solution is not ridiculously small, such an exponential-time algorithm is doomed to run for ages.

A dynamic programming strategy can be devised to solve the FLP in $\Theta(nL)$ time and space. This is not surprising, given the similarity of this problem with the Knapsack Problem [30], the Partition Problem [11], and the Bin Packing Problem [6]. Such complexity, as usual, corresponds to the size of the dynamic programming table that must be filled with answers to subproblems, each one computed in constant time.

Apart from the dynamic programming approach, one could come up with a simple backtracking strategy as follows. Identify each feasible configuration of k cars by a tuple $(x_1, \dots, x_k)$, where $x_i \in \{\mathit{left}, \mathit{right}\}$ indicates the lane occupied by the ith car, for $i = 1, \dots, k$. Each configuration $(x_1, \dots, x_k)$ leads to configurations $(x_1, \dots, x_k, \mathit{left})$ and $(x_1, \dots, x_k, \mathit{right})$, respectively, if there is room for the $(k + 1)$th car on the corresponding lane. Starting from the configuration (left) – the first car boards on the left lane, without loss of generality –, inspect all feasible configurations in backtracking fashion and report the size of the largest configuration that is found. It works fine. Unfortunately, such approach demands exponential time in the worst case.

However, we can perfectly well identify each configuration by a pair (k, s), where k is the number of cars on the ferry and s is the total length of the *left lane* that is occupied by cars. Each configuration (k, s) leads to at most two other configurations in the implicit backtracking graph:

- $(k + 1, s + \ell_{k+1})$, by placing the $(k + 1)$th car in the left lane, which is only possible if $s + \ell_{k+1} \leq L$; and
- $(k + 1, \ell)$, by placing the $(k + 1)$th car in the right lane, which is only possible if $\left(\sum_{i=1}^{k+1} \ell_i\right) - s \leq L$.

If neither placement of the $(k + 1)$th car is possible, backtrack. Notice that a same configuration $(k + 1, s')$ may be reached by two distinct configurations (k, s_1) and (k, s_2). It happens precisely if $|s_1 - s_2| = \ell_{k+1}$. However, we do not want to visit a same configuration twice, and therefore we must keep track of the visited configurations.

The algorithm searches the whole backtracking graph, starting by the root node (0, 0) – there are zero cars on the ferry, initially, occupying zero space on the left lane. The reported solution will be the largest k that is found among the visited configurations (k, s).

As for the time complexity, the whole search runs in time proportional to the number of nodes (configurations) in the backtracking graph. Since $0 \leq k \leq n$, $0 \leq s \leq L$, and both k and s are integers, such number is $O(nL)$, bounding the worst-case time complexity of our algorithm *if* we make sure we can check whether a configuration has already been visited in $O(1)$ time. If we use a 2-dimensional matrix to store the configurations, the problem is solved. However, by doing so, we demand $\Theta(nL)$ space, even though our matrix will most likely be absolutely sparse.

We can resort to hashing to solve that. By using a hash set to store the visited configurations, we can still perform each configuration lookup in (expected) constant time, and the required space will be proportional to the number $O(nL)$ of configurations that are actually visited during the search. In practice, the $O(nL)$ time and space required by our algorithm can be remarkably smaller than the $\Theta(nL)$ time and space of the dynamic programming solution, specially if there are many cars with the same size.

3.5 The Trading Company Problem

Considerer the following hypothetical problem. A certain company X operates in the stock market, entering a huge number of financial transactions – called *trades* – every day. On the other side of each trade, there is some company Y, call it a *counterparty* of X. To each trade of X is assigned a *portfolio* number, which is *not* a unique identifier of the trade, and is not even unique per counterparty. In other words, trades with different counterparties may be assigned the same portfolio, and different trades with the same counterparty may be assigned distinct portfolios as well, according to some internal policy of the company. Whenever X enters into a trade, a new line such as "+ 101 25" is appended to a log file, where "+" indicates that a new trade was initiated by X, "101" is the counterparty id, and "26" is the portfolio of that trade. At any time during a typical day, however, a counterparty Y may withdraw all transactions with X. From X's standpoint, that means all trades it had entered into with Y that day are now considered void, and a line such as "− 101" is logged. Here, the "−" sign indicates that a cancelation took place, and "101" is the id of the counterparty the cancelation refers to. Table 1 illustrates the basic structure of the log file of X.

Table 1 – Sample log file.

+	101	26
+	2	25
+	101	25
+	3005	4550
−	101	
+	3005	26
+	4	184
+	101	4550
−	2	

By the end of each day, the company X needs to determine the set of distinct portfolios associated to trades that are still *active* by then, that is, the portfolios of trades that have not been canceled during the day. What is the most efficient way to achieve that?

Hash set. As usual, it is easy to think of different ways to process the file and come up with the desired information. For example, we can traverse the file in a top-down fashion (i.e. in chronological order), and, for each line "$+\ Y\ P$", we check whether there is any line below it (i.e. logged after it) indicating the cancelation of previous trades with Y. In case there is no such line, add P to a list of active portfolios, taking care not to add the same portfolio twice to the list. To avoid duplicates, one can already think of using a hash set, where portfolios are inserted only if they are not there already, something that can be verified in constant time, on average. This approach surely works, and it is easy to see that the time complexity of this strategy is $\Theta(n^2)$, where n denotes the number of lines in the log file.

Yet the file can be really huge, and determining the active portfolios is a time-critical operation for X. A quadratic-time algorithm is not good enough.

Two mirroring hash maps. Consider the use of two hash maps, this time: (i) The first one will hold a counterparties-by-portfolio map: the keys are portfolio numbers, and the value stored with each key P is a list containing the counterparties Y of X for which there are active negotiations under that portfolio P. (ii) The second is a portfolios-by-counterparty hash map: the keys are the counterparties of X, and the values associated with each counterparty Y is a list containing the portfolios assigned to active negotiations with that partner.

The file is traversed top-down. Each line with a "+" sign, be it "+ $Y\ P$", triggers two updates in our data structure: the first is the inclusion of Y in the list of counterparties associated with portfolio P in the counterparties-by-portfolio hash map (the key P will be inserted for the first time if it is not already there); the second change is the inclusion of P in the list of portfolios associated with counterparty Y in the portfolios-by-counterparty hash map (the key Y will be first inserted for the first time if it is not already there).

Whenever a line with a "–" sign is read, be it "– Y", we must remove all occurrences of Y from both tables. In the second, portfolios-by-counterparty hash map, Y is the search key, hence the bucket associated to Y can be determined directly by the hash function, and the list of portfolios associated to Y can be retrieved (and deleted, along with Y) in average time $O(1)$. In counterparties-by-portfolio, however, Y can belong to lists associated to several distinct portfolios. To avoid traversing the lists of all portfolios, we can use the information in the first, portfolios-by-counterparty map (sure enough, before deleting the key Y) so we know beforehand which portfolios will have Y on their counterparty lists. We may thus go directly to those lists and remove Y from them, without the need to go through the entire table. If the list of counterparties associated to a portfolio P becomes empty after the removal of Y, then we delete the portfolio P from that first hash map altogether.

After all lines of the file have been processed, just return the portfolios corresponding to keys that remain in the first, counterparties-by-portfolio hash map.

Figure 1 illustrates the data structures used in this solution. Note that, for clarity, different keys in the same bucket (a collision in the hash table) are not being even considered. As already mentioned, collisions (as well as the choice of the hash function and other technicalities) are being taken care of by the lower level hash implementation, allowing us to focus on the high level *usage* of it, as happens most of the time in practice.

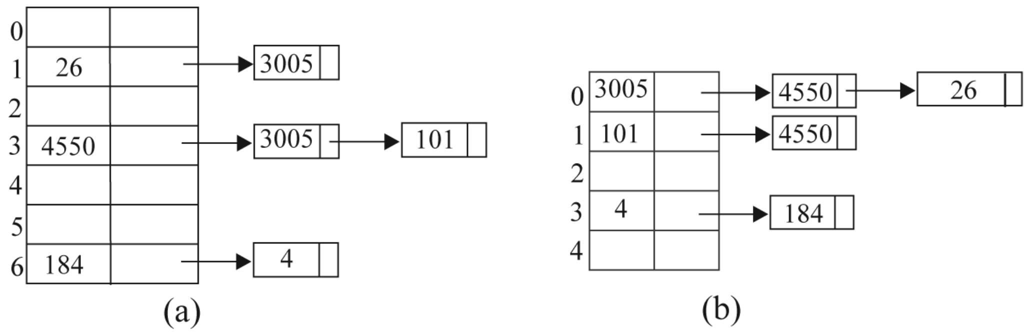

Figure 1 – Hash maps: (a) counterparties-by-portfolio; (b) portfolios-by-counterparty.

Determining the lists from which a certain counterparty Y must be removed (due to a cancelation read from the file) can be made in average $O(1)$ time per list, and each counterparty can be removed at most once for each line of the file that inserted it there. So far so good. However, in order to find Y in each list (something we also need to do by the time we *insert* Y, so we do not have the same counterparty appear more than once in the same list), we still need to traverse the whole list, which certainly increases the computational time. Since we wish we can cope with the whole file processing task in linear time, we must think of a remedy to this.

Multi-level hashing. So our problem is the time-consuming task of traversing whole lists to locate a single element. We therefore want to try and replace those lists with more performatic structures, such as binary search trees, or, better yet... hash sets! In other words, the value associated with each key P in the counterparties-by-portfolio hash map will be, instead of a list, a hash set whose keys are the ids of the counterparties associated with P (see Figure 2). The same goes for the second, portfolios-by-counter-party hash map, where portfolios will be stored in a hash set associated to each counterparty Y. Thus, counterparties and portfolios can be included and removed without the need to traverse possibly lengthy lists, but in average constant time instead. Since the cost of processing each line is now $O(1)$ (due to a constant number of hash table dictionary operations being performed), the whole algorithm, implemented this way, runs in expected linear time in the size of the file. (Sure enough, the same time complexity could have been achieved, in the previous approach, by making a cumbersome use of pointers across the hash tables.)

Bottom-up traversal (bare creativity). The most elegant solution for the Trading Company Problem, though, consists in reading the log file in a bottom-up fashion (i.e. in anti-chronological order), avoiding the need to process trades that would be, later in the day, canceled anyway.

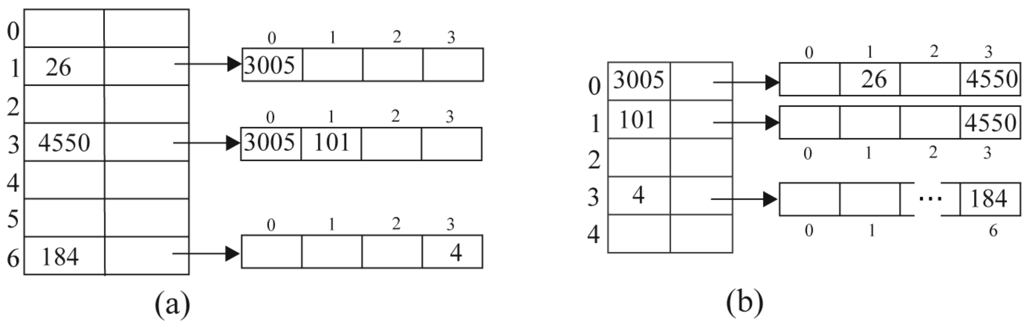

Figure 2 – Multi-level hashing: (a) Hash map of counterparties (represented by hash sets) per portfolio; (b) Hash map of portfolios (represented by hash sets) per counterparty.

Note that, as the file is being read upwards, the first line found with a "$-$" sign, be it "$-\ Y$", indicates the cancelation of all trades company X had entered into with counterparty Y earlier in the day. Thus, such a line L_Y in the log file partitions all other lines associated to Y into two groups: those which occurred chronologically before L_Y (and which have not been processed yet, since they appear above it in the file), and those which occurred chronologically after L_Y (and therefore have already been processed). From that moment on, the lines of the former group may be simply ignored, since the corresponding trades would not be active by the end of the day anyway. On the other hand, all lines regarding Y that have already been processed (those in the latter group) correspond to trades that are active indeed by the end of the day, since they appear, in the log file, chronologically after the last cancellation involving Y.

We can therefore use again two hash sets: one for the "active portfolios", whose keys will be returned by the end of the algorithm execution, and another for the "bypassed counterparties", that is, the counterparties for which a cancelation line has already been read in the bottom-up traversal of the file. With such a simple structure, each "$-\ Y$" line triggers the insertion of Y into the bypassed counterparties hash set (if it is not already there), and each "$+\ Y\ \ P$" line either triggers the insertion of P in the active portfolios hash set, in case it is not already there *and* Y is not in the bypassed counterparties table; otherwise the line is ignored.

This time again, each line gives rise to a constant number of average constant-time operations, but in a much simpler fashion. The whole process runs in $O(n)$ time, where n is the number of lines in the log file.

Quite surprisingly, however, this latter solution can be made even better. By keeping track of the number p of bypassed counterparties and comparing it, at each processed line, against the total number c of counterparties the company X has traded with during that day (such piece of information can easily be kept track of during the generation of the log file), the bottom-up traversal can perfectly halt if ever p becomes equal to c. If the rate of cancelations is high, then most probably the algorithm stops before all n lines have been processed, for a practical sub-linear-time execution.

4 CONCLUSIONS

The use of appropriate data structures is one of the main aspects to be considered in the design of efficient algorithms. Certain structures, however, are somewhat stigmatized, restricted to particular applications and some classic variants. Though the use of hash tables in Operations Research is not new (some nice improvements based on hashing have been reported in the literature for known OR problems), we believe its possibilities are quite often neglected – or overlooked – by part of the OR community.

It is certainly important to add some creativity to one's theoretical toolset when considering the choice of data structures. Hash tables, however, bear such an immense likelihood of producing efficient, practical algorithms, that one should always feel inclined to at least consider them, even in applications that, at a first glance, do not seem to suggest their use. In this paper, we illustrated the use of hashing to produce neat, efficient solutions to simple, didactic problems, in a selection we hope to have been as instructive as motivating.

Though we have not included a whole section dedicated to computational results, implementing the algorithms we discussed is straightforward. We encourage the reader to do so, specially if he or she is not too familiar with formal complexity theory. As an example of the kind of result the reader should be able to produce, Table 2 shows a comparison between the naive algorithm and the algorithm based on hashing discussed in Section 3.1. The source code, in Python, can be found in `https://www.dropbox.com/s/wm6goozuoq27lh6/pair_sum.py?dl=0`. It is noteworthy that the exact hash function that was used has not even be specified in the source code, which relied on inbuilt, general-purpose hash functions.[4]

Table 2 – Average running times (and standard deviations), in mili-seconds, for input lists of size n in the Pair Sum problem. For each value of n, we ran the algorithms on 100 lists of n integers chosen uniformly at random in the range $[1, 10^8]$.

n	without hashing	with hashing
250	13.754 (2.340)	0.095 (0.029)
500	53.312 (3.653)	0.185 (0.040)
1000	214.859 (9.441)	0.356 (0.092)
2000	871.815 (33.956)	0.722 (0.130)
4000	3453.420 (85.533)	1.345 (0.234)
8000	13804.324 (320.425)	2.706 (0.346)

[4]The actual implementation of Python hash tables, also known as *dictionaries* or *dicts*, can be found in `https://hg.python.org/cpython/file/52f68c95e025/Objects/dictobject.c`, where a detailed account on the employed hash functions and the collision resolution scheme is also given.

REFERENCES

[1] ANDREEVA E, MENNINK B & PRENEEL B. 2012. The parazoa family: generalizing the sponge hash functions. *International Journal of Information Security*, **11**(3): 149–165.

[2] BOTELHO FC, PAGH R & ZIVIANI N. 2007. Simple and space-efficient minimal perfect hash functions. In: *Proc. of the 10th Workshop on Algorithms and Data Structures (WADS'07). Lecture Notes in Comp. Sc.*, **4619**: 139–150.

[3] CHIARANDINI M & MASCIA F. 2010. A hash function breaking symmetry in partitioning problems and its application to tabu search for graph coloring. Tech. Report No. TR/IRIDIA/2010-025, Université Libre de Bruxelles.

[4] CORMEN TH, LEISERSON CE, RIVEST RL & STEIN C. 2001. *Introduction to Algorithms, Vol. 2.*, The MIT Press.

[5] DIETZFELBINGER M. 2007. Design strategies for minimal perfect hash functions. *Lecture Notes in Comp. Sc.*, **4665**: 2–17.

[6] DOWSLAND K & DOWSLAND WB. 1992. Packing problems. *European Journal of Operational Research*, **56**(1): 2–14.

[7] DIETZFELBINGER M, KARLIN AR, MEHLHORN K, MEYER AUF DER HEIDE F, ROHNERT H & TARJAN RE. 1994. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, **23**(4): 738–761.

[8] FOX BL. 1978. Data Structures and Computer Science Techniques in Operations Research. *Operations Research*, **26**(5): 686–717.

[9] GONG Y & LAZEBNIK S. 2011. Iterative quantization: a procrustean approach to learning binary codes. In: *Proc. of the 24th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'11)*, 817–824.

[10] GOODRICH MT & TAMASSIA R. 2002. *Algorithm Design: Foundations, Analysis and Internet Examples*, Wiley.

[11] HAYES B. 2002. The easiest hard problem. *American Scientist*, **90**(2): 113.

[12] HOFHEINZ D & KILTZ E. 2011. Programmable hash functions and their applications. *Journal of Cryptology*, **25**(3): 484–527.

[13] IACONO J & PĂTRAŞCU M. 2012. Using hashing to solve the dictionary problem. In: *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'12)*, 570–582.

[14] JAGANNATHAN R. 1991. Optimal partial-match hashing design. *Operations Research Society of America (ORSA) Journal on Computing*, **3**(2): 86–91.

[15] KNOTT GD. 1975. Hashing functions. *The Computer Journal*, **18**: 265–278.

[16] KNUTH DE. 1973. *The Art of Computer Programming 3: Sorting and Searching, Vol. 1*, Addison-Westley.

[17] KULIS B & DARRELL T. 2009. Learning to Hash with binary reconstructive embeddings. In: *Proc. of the 23rd Annual Conference on Neural Information Processing Systems (NIPS'09)*, 1042–1050.

[18] LEWIS TG & COOK CR. 1988. Hashing for dynamic and static internal tables. *IEEE Computer*, **21**: 45–56.

[19] LI X, LIN G, SHEN C, HENGEL A & VAN DEN DICK A. 2013. Learning hash functions using column generation. In: *Proc. of the 30th International Conference on Machine Learning (ICML'13). Journal of Machine Learning Research*, **28**(1): 142–150.

[20] LIU W, WANG J, JI R, JIANG Y-G & CHANG S-F. 2012. Supervised hashing with kernels. In: *Proc. of the 25th IEEE Conference on Computer Vision and Pattern Recognition (CVPR'12)*, 2074–2081.

[21] MARTELLO S & TOTH P. 1985. Approximation schemes for the subset-sum problem: Survey and experimental analysis. *European Journal of Operational Research*, **22**(1): 56–69.

[22] MAURER WD & LEWIS TG. 1975 Hash table methods. *ACM Computer Surveys*, **7**: 5–19.

[23] NOROUZI M & FLEET D. 2011. Minimal loss hashing for compact binary codes. In: *Proc. of the 28th International Conference on Machine Learning (ICML'11)*, 353–360.

[24] PAGH A & PAGH R. 2008. Uniform hashing in constant time and optimal space. *SIAM J. Comput.*, **38**(1): 85–96.

[25] PAGH R & RODLER FF. 2004. Cuckoo hashing. *J. Algorithms*, **51**(2): 122–144.

[26] SALAKHUTDINOV R & HINTON G. 2009. Semantic hashing. *International Journal of Approximate Reasoning*, **50**(7): 969–978.

[27] SKIENA SS & REVILLA MA. 2003. *Programming Challenges*. Springer.

[28] WANG J, KUMAR S & CHANG S-F. 2010. Sequential projection learning for hashing with compact codes. In: *Proc. of the 27th International Conference on Machine Learning (ICML'10)*, 1127–1134.

[29] WOODRUFF DL & ZEMEL E. 1993. Hashing vectors for tabu search. *Annals of Operations Research*, **41**(2): 123–137.

[30] YANASSE HH, SOMA NY & MACULAN N. 2000. An algorithm for determining the k-best solutions of the one-dimensional knapsack problem. *Pesquisa Operacional*, **20**(1): 117–134.