Chapter 5

Entropy, Working Sets and Doubly-Exponential Series

Until now, we have studied data structures that had good worst-case behaviour. That is, for any sequences of queries the data structures offer good query times, usually $O(\log n)$, where n is the number of elements stored in the data structure. In the usually accepted comparison-tree model of computation, $O(\log n)$ is optimal if the distribution of queries is uniform, that is each query is equally likely. However, when the distribution of queries is not uniform data structures may take advantage of this to perform operations in $o(\log n)$ time. This chapter is about such data structures.

5.1 Entropy

Let $S = \{k_1, \ldots, k_n\}$ be a set of objects let $D = p_1, \ldots, p_n$ be a probability distribution, so that $p_i > 0$ is the probability associated with k_i. The *entropy* of D is defined as

$$H(D) = -\sum_{i=1}^{n} p_i \log p_i = \sum_{i=1}^{n} p_i \log(1/p_i) \ . \tag{5.1}$$

Entropy is used in the context of coding theory. Imagine we have a sender and a receiver and the sender wants to sent a sequence $s_1, \ldots, s_m$ where each element s_j is chosen randomly and independently according to D, so that $s_j = k_i$ with probability p_i. The main result of coding theory is Shannon's Theorem, which states that for any protocol the sender and receiver might use, the expected number of bits required to transmit $s_1, \ldots, s_m$ using that protocol is at least $mH(D)$.

As an example, consider the *uniform distribution* $p_i = 1/n$ for all $1 \leq i \leq n$. Then (5.1) becomes $H(D) = \sum_{i=1}^{n} (1/n) \log n = \log n$. In this case, Shannon's theorem says that, on average, we require $\log n$ bits to encode each symbol, which can be achieved using a standard binary encoding. At the other extreme, we could consider the *geometric distribution* $p_i = 1/2^i$ for all $1 \leq i < n$ and $p_n = 1/2^{n-1}$. In this case, (5.1) becomes $H(D) = \sum_{i=1}^{n} (i/2^i) \leq 2$. Shannon's theorem only gives a lower bound of 2 bits per symbol. In this case a lot of bits can be saved by an encoding scheme that achieves $H(D)$ bits per symbol.

Shannon's Theorem has had a profound impact in many areas, including data structures. Imagine $k_1, \ldots, k_n$ are taken from some total order, so that we can compare any two values k_i and k_j. Then the sender and receiver use the following protocol: Initially, the sender and receiver both store the elements $k_1, \ldots, k_n$ in some comparison-based data structure that is agreed on before hand.

When the sender wants to transmit s_1, he performs a search for s_1 in the data structure. This results in a sequence of comparisons of the form $a \mathbf{op} b$, where $\mathbf{op} \in \{<, >, \leq, \geq, =\}$ which are either true or false. This gives a sequence of 1s (for true) and 0s (for false) that the sender sends to the receiver. On the receiving end, the receiver runs the search algorithm without knowing the value of s_1. The receiver can do this, because she is doing exactly the same comparisons that the sender did, and knows the results of those comparison because the sender has sent them. After doing this, the receiver deduces that the element sent is s_1 since it has just completed a search for s_1. The sender and receiver then continue in this manner to transmit $s_2, \ldots, s_m$.

Note that the sender and receiver can use any data structure they like, and can modify the data structure as they are transmitting. However, Shannon's Theorem says that no matter which data structure they use, and no matter which rules they use to reorganize this data structure, the expected number of bits sent is still at least $mH(D)$. However, the number of bits sent when sending s_i is exactly the same as the number of comparisons performed while searching for s_i. Therefore, Shannon's Theorem implies that the expected number of comparisons while searching for s_i is $\Omega(H(D))$ and the expected number of comparisons required to handle the request sequence $s_1, \ldots, s_m$ is $\Omega(mH(D))$, for any comparison-based data structure.

In the remainder of this chapter, we study data structures that can access the sequence $s_1, \ldots, s_m$ in close to $O(mH(D))$ expected time, and are thus optimal by Shannon's Theorem. In fact, some of these data structures work without even knowing the probability distribution D. In studying these data structures, it sometimes helps to fix a specific sequence $s_1, \ldots, s_m$. Let m_i be the number of times the symbol i occurs in $s_1, \ldots, s_m$. Then, the *empirical entropy* of $s_1, \ldots, s_m$ is given by

$$H(s_1, \ldots, s_m) = -\sum_{i=1}^{n} (m_i/m) \log(m_i/m) = \frac{1}{m} \sum_{i=1}^{n} m_i / \log(m/m_i) \ .$$

Some of the data structures described in this section will be able to access the sequence $s_1, \ldots, s_m$ in close to $O(mH(s_1, \ldots, s_m))$ time. If each of the s_i are chosen independently according to distribution D, then

$$H(s_1, \ldots, s_m) \stackrel{\mathcal{L}}{=} H(D)$$

as m goes to infinity. Thus, for sufficiently large m (compared to n), the expected time of the data structure to access $s_1, \ldots, s_m$ is $O(mH(D))$.

5.2 Nearly-Optimal Search Trees

Suppose we are given they keys $k_1, \ldots, k_n$, with $k_i < k_{i+1}$ for each $i \in \{1, \ldots, n-1\}$ and with each key k_i we are also given its access probability p_i. A common tool in the design of entropy-sensitive data structures is *probability splitting*. One way to implement this idea is to find a key k_i such that

$$\sum_{j=1}^{i-1} p_j \leq (1/2) \sum_{j=1}^{n} p_j \tag{5.2}$$

and

$$\sum_{j=i+1}^{n} p_j \leq (1/2) \sum_{j=1}^{n} p_j \ . \tag{5.3}$$

Note that at least one and at most 2 values of i satisfy this property.

The key k_i becomes the root of a binary search tree, and the left and right child are constructed recursively from $(k_1, \ldots, k_{i-1}, p_1, \ldots, p_{i-1})$ and $(k_{i+1}, \ldots, k_n, p_{i+1}, \ldots, p_n)$. Let T denote the resulting tree.

Observe that, in T, if a node k_i has depth $\text{depth}_T(k_i)$, then

$$\sum_{k_j \in T(k_i)} p_j \leq 1/2^{\text{depth}_T(k_i)} \ ,$$

where $T(k_i)$ is the set of all nodes in the subtree rooted at k_i. In particular $p_i \leq 1/2^{\text{depth}_T(k_i)}$, so $\text{depth}_T(k_i) \leq \log(1/p_i)$. The expected depth (distance from the root) of a key chosen according to the distribution D is

$$\sum_{i=1}^{n} p_i \cdot \text{depth}_T(k_i) \leq \sum_{i=1}^{n} p_i \log(1/p_i) = H(D) \ .$$

where $d(k_i)$ is the depth of k_i. Therefore, the search tree T is a data structure that can answer queries using $O(1 + H(D))$ comparisons in $O(1 + H(D))$ time, so is optimal by Shannon's Theorem.

How long does it take to construct T? Finding the key k_i that satisfies (5.2) and (5.3) can easily be done in $\Theta(n)$ time. Unfortunately, for some distributions, this can lead to an overall construction time of $\Theta(n^2)$. (Exercise: describe such a distribution). By searching simultaneously starting and k_1 and working forward and starting at k_n and working backwards, the time to find k_i can be reduced to $O(\min\{i, n-i+1\})$. This leads to the recurrence

$$T(n) = O(\min\{i, n-i+1\}) + T(i-1) + T(n-i-1) \ ,$$

for some $i \in \{1, \ldots, n\}$, which resolves to $O(n \log n)$.

An even better solution starts searching simultaneously from 1 and n using an *exponential search*. That is, we check k_1, k_2, k_4, k_8 and so on until finding the first k_{2^j} with $2^j \geq i$ and then perform binary search on $k_{2^{j-1}}, \ldots, k_{2^j}$. This allows us to find k_i in $O(\log i)$ time. Simultaneously doing the search working backwards from n allows us to find k_i in $O(\log \min\{i, n-i+1\})$ time. This gives an overall running time that is defined by the recurrence

$$T(n) = O(\log \min\{i, n-i+1\}) + T(i-1) + T(n-i-1) \ ,$$

whose solution is $O(n)$, as can be verified using induction.

Theorem 10. *If $k_1, \ldots, k_n$ are given in sorted order then the above data structure can be constructed in $O(n)$ time, uses $O(n)$ space and the expected cost of searching for a random key chosen according to probability distribution D is $O(1 + H(D))$.*

5.3 Optimal In-Place Static Searching

Suppose again that we are given n keys $k_1, \ldots, k_n$ with corresponding distribution $D = p_1, \ldots, p_n$ where p_i represents the probability of accessing k_i. Without loss of generality, assume $p_i \leq p_{i+1}$ for

all $1 \leq i < n$, since we can always relabel the keys to achieve this. Our goal is to find a static data structure that allows us to perform searches, so that the cost of searching for k_i is $O(\log(1/p_i))$.

To achieve this goal we store $k_1, \ldots, k_n$ in an array of size n that is partitioned into $\Theta(\log\log n)$ *groups*. Group 0 contains the keys k_1 and k_2. Group 1 contains the keys $k_3, \ldots, k_6$. Group 2 contains the keys $k_7, \ldots, k_{22}$. In general, group i contains 2^{2^i} keys. The data structure maintains each group in sorted order, so that searching within group i takes $O(\log 2^{2^i}) = O(2^i)$ time using binary search.

To search for a key k_j in this data structure, we search first in group 0, then in group 1, and so on until we find k_j, or we have searched in all groups. If we find k_j in group i, then the cost of the search is

$$\sum_{g=0}^{i} O(2^g) = O(2^i) \ .$$

We claim that, if the we find k_j in group i, then $p_j < 1/2^{2^{i-1}}$. To see this, observe that $j > \sum_{g=0}^{i-1} 2^{2^g} > 2^{2^{i-1}}$. In other words, there are more than $2^{2^{i-1}}$ keys with access probability larger than p_j. Therefore $p_j < 1/2^{2^{i-1}}$, since all the access probabilities sum to 1.

We finish by observing that

$$\log(1/p_j) \geq \log\left(2^{2^{i-1}}\right) = 2^{i-1} = \Theta(2^i) \ ,$$

so that the time to access key k_j is $O(\log(1/p_j))$, as required. Therefore, if we search for a random key chosen according to distribution D then the expected cost of the search is

$$\sum_{i=1}^{n} p_i O(1 + \log(1/p_i)) = O(1 + H(D)) \ .$$

Theorem 11. *The above data structure can be constructed in* $O(n \log n)$ *time (by sorting), uses* $O(n)$ *space and the expected cost of searching for a random key chosen according to probability distribution* D *is* $O(1 + H(D))$.

5.4 Optimal Dynamic Searching

In the previous section, we assumed that the distribution D was known to the data structure. In this section we show that the data structure does not need to know D to achieve good performance. The setup is the same, we have a data structure that stores keys $k_1, \ldots, k_n$, but we know nothing about the access probabilies of these keys.

The data structure is based on the same doubly-exponential (2^{2^i}) sequence as the previous data structure, but it is *self-organizing*. That is, after every access it modifies itself, in the hopes of speeding up subsequent accesses. The data structure consists of $O(\log\log n)$ balanced search trees that support insertion, deletion and searching in logarithmic time. The tree T_i contains exactly 2^{2^i} elements. Additionally, the data structure maintains, for each tree T_i, a linked-list L_i of the elements contained in T_i. Initially, the elements $k_1, \ldots, k_n$ are assigned to trees arbitrarily.

When searching for the key k_j, we start by searching T_0, T_1, and so on until we find k_j in some tree, say T_i. We then remove the key k_j from T_i and L_i and insert k_j into T_0 and put it at the front of

the list L_0. Now T_0 contains one element too many, so we take the last element in L_0, remove it from T_0 and L_0 and insert it into T_1 and put it at the front of L_1. Now T_1 contains one element too many, so we take the last element from L_1, remove it from L_1 and T_1 and insert it into T_2 and put it at the front of L_2. We proceed in this manner until reaching T_i, at which point the process stops because T_i contained one element too few, due to the removal of k_j.

In each tree T_g, $0 \leq g \leq i$ we perform one insertion, one deletion and one search operation, at a cost of $O(\log(2^{2^g})) = O(2^g)$. Therefore, the cost of accessing k_j is

$$\sum_{g=0}^{i} O(\log(2^{2^g}) = \sum_{g=0}^{i} O(2^g) = O(2^i) \ .$$

Unfortunately, it's not clear that the value of i has anything to do with the probability of accessing k_j. However, one thing we do know is the following *working set property*. Suppose we have accessed k_j at some point in the past, say t accesses previously. Immediately after we accessed k_j it was put into T_0. During each subsequent access, it moved down at most one unit in the list it was contained in or, if it was the last element in some list L_g, it moved into L_{g+1}. Therefore, k_j is contained in T_i for some i such that $\sum_{g=0}^{i-1} 2^{2^g} < t$. It follows that the time to access k_j is $O(\log t)$.

Thus, if we accessed k_j recently, it won't cost much to access it again. Next, we formalize the following intuition: If we access k_j frequently, then the average amount of time between successive accesses is small and the overall cost of accessing k_j is small. Before we begin, we require an inequality due to Jensen:

Lemma 2 (Jensen's Inequality)**.** *Let $f : \mathbb{R} \to \mathbb{R}$ be a strictly increasing concave function. Then, the sum $\sum_{i=1}^{n} f(t_i)$ subject to the constraint $\sum_{i=1}^{n} t_i < m$ is maximized when $t_1 = t_2 = \cdots = t_n = m/n$.*

A nice way to visualize Jensen's inequality is to imagine placing $n-1$ points on an interval of length m, so that the interval is split into n intervals of length $t_1, \ldots, t_n$. Jensen's inequality says that if we want to maximize $\sum_{i=1}^{n} f(t_i)$ then the best thing we can do is to make all the intervals of equal length.

Returning to our data structuring problem, suppose $s_1, \ldots, s_m$ is a sequence of keys representing accesses, and suppose that the key k_j appears m_j times in this sequence. Then the total cost of all accesses to k_j using the above data structure is

$$O(n \log n) + \sum_{i=1}^{m_j} O(\log(1 + t_i)) \ ,$$

where t_i is the number of accesses to keys other than k_j between the ith and the $(i+1)$th access to k_j. (The extra $O(n \log n)$ term comes from the fact that the first access to each element takes $O(\log n)$ time.) Of course, the t_i obey the inequality $\sum_{i=1}^{m_j} t_i < m$ and log is a concave function so, by Jensen's inequality,

$$\text{cost of accesses to } k_j = O(\log n) + \sum_{i=1}^{m_j} O(\log(t_i)) = O(\log n + m_j \log(m/m_j)$$

This is true for all $1 \leq j \leq n$, so the total cost of handling the sequence $s_1, \ldots, s_m$ is

$$\text{cost of } s_1, \ldots, s_m \quad = \quad \sum_{j=1}^{n} (\text{cost of accesses to } k_j)$$

$$\begin{aligned} &= \sum_{j=1}^{n} O(\log n + m_j(1 + \log(m/m_j))) \\ &= O(n \log n + m(1 + H(s_1, \ldots, s_n))) \ . \end{aligned}$$

Theorem 12. *The above data structure can be constructed in* $O(n \log n)$ *time and can access the sequence* $s_1, \ldots, s_m$ *in* $O(n \log n + m(1 + H(s_1, \ldots, s_m)))$ *time.*

5.5 MTF Compression

The optimal dynamic search data structure from Section 5.4 can be used as a compression scheme, as described in the introduction. This will give a compression scheme that uses $O(H(D))$ bits per symbol on average. In the remainder of this section we describe a compression algorithm that is basically a stripped down version of this idea. The algorithm is called *move-to-front* for reasons that will become apparent soon enough.

Consider the following encoding scheme for positive integers. The integer i is encoded as $\lceil \log i \rceil - 1$ zeros followed by the binary representation of i (which starts with a 1). To decode an integer, a receiver counts the number of leading zeros, reads the same number of bits from the remainder and decodes this as a binary number. This scheme uses $2\lceil \log i \rceil - 1 \leq 1 + 2\lfloor \log_i \rfloor$ bits to encode the integer i, and the first $\lceil \log i \rceil$ bits are always a sequence of zeroes followed by exactly one one. To get a scheme that uses $\lfloor \log i \rfloor + O(\log \log i)$ bits we just observe that the leading $\lceil \log i \rceil$ bits represent the positive integer $\lceil \log i \rceil \leq 1 + \log i$ and can therefore be encoded using $1 + 2\lfloor \log(1 + \log i) \rfloor$ bits, using the same scheme recursively.

In MTF (move-to-front) compression, the sender and receiver each maintain identical lists that contain integers $1, \ldots, n$. To send the symbol j, the sender looks for j in his list and finds it at position i (say). The sender then encodes i using $\log i + O(\log \log i)$ bits and sends it to the receiver. The receiver decodes i, looks at the ith element in his list and finds j. The sender and receiver then both move the element j to the front of their lists and continue.

To analyze the cost of sending the sequence $s_1, \ldots, s_m$, we first observe that MTF also has the working set property. If the number of distinct symbols between two consecutive occurences of the integer j is $t - 1$ then the cost of encoding the second j is $\log t + O(\log \log t)$. Therefore, by the same argument used in previous section, the total cost of encoding all occurences of j is at most

$$\begin{aligned} C_j &= \log n + O(\log \log n) + \sum_{i=1}^{m_j} (\log t_i + O(\log \log t_i)) \\ &\leq \log n + O(\log \log n) + m_j \left(\log(m/m_j) + O(\log \log m/m_j)\right) \ , \end{aligned}$$

where $t_i - 1$ is the number of symbols between the $i - 1$th and the ith occurence of j. Summing this over all j, we see that the number of bits required to compress the sequence $s_1, \ldots, s_m$ is

$$\begin{aligned} B(s_1, \ldots, s_m) &\leq C_j \\ &\leq n \log n + O(n \log \log n) + mH(s_1, \ldots, s_m) + \sum_{j=1}^{n} O(m_j \log \log m/m_j) \\ &\leq n \log n + mH(s_1, \ldots, s_m) + O(n \log \log n + m \log H(s_1, \ldots, s_m)) \end{aligned}$$

Although this equation looks quite complicated, it really consists of a a $n \log n$ startup cost plus the empirical entropy of $s_1, \ldots, s_m$ plus some lower order terms. If m is much larger than n, the $n \log n$ terms become negligible and the overall cost is very close to the empirical entropy of $s_1, \ldots, s_m$.

Theorem 13. *The MTF compression algorithm compresses the sequence* $s_1, \ldots, s_m$ *into*

$$n \log n + mH(s_1, \ldots, s_m) + O(n \log\log n + m \log H(s_1, \ldots, s_m))$$

bits.

5.6 Discussion and References

The notion of entropy was introduced by Shannon in his groundbreaking work on information theory [6]. The construction of nearly optimal binary search trees in linear time is due to Mehlhorn [5]. The optimal dynamic search structure is due to Iacono [4]. The variable length integer encoding scheme is due to Elias [3]. The MTF compression algorithm is due to Bentley *et al* [1]. MTF is a nice simple idea that is easy to implement and gives good compression. Unfortunately, it's patented [2], but at least the patent expires soon.

Bibliography

[1] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4), 1986.

[2] J. L. Bentley, D. D. K. Sleator, and R. E. Tarjan. Data compaction. *US Patent 4,796,003*, January 1989.

[3] P. Elias. Universal codeword sets and the representation of the integers. *IEEE Transactions on Information Theory*, 21:194–203, 1975.

[4] J. Iacono. Alternatives to splay trees with $O(\log n)$ worst-case access times. In *Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-01)*, pages 516–522, 2001.

[5] K. Mehlhorn. Nearly optimal binary search trees. *Acta Informatica*, 5:287–295, 1975.

[6] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423, 623–656, 1948.