Union-Find

- *Algorithm:* Union-Find (algo. 1)
- *Input:* A graph of points G
- *Complexity:* Depending on the implementation, $\mathcal{O}(\log^* n)$ for the most common implementation
- *Data structure compatibility:* Array or tree
- *Common applications:* Graphs, statistics

Problem. Union-Find

Given a graph of points G, find whether two points x and y are connected, or find whether point x is in a connected subset S of G, or join two connected subsets S_1 and S_2 together.

Description

The Union-Find data structure is one of the most commonly used data structure in the field of graph. As discussed in the class, it can handle multiple tasks including finding whether two points in a graph are connected or joining two connected subset. As a matter of fact, the detailed implementation of the Union-Find structure can be very different. The efficiency of **Find** method and **Union** method can also differ a lot with respect to the underlying data structure.

An intuitive implementation is using an array. The index of the array corresponds to the order of the elements, and the data stored is the order number of subset the point is in, as shown in (Table 1)

Table 1: The array based Union-Find structure

1	2	3	4	5	6	7	8	9	10
1	1	2	1	1	2	4	2	3	2

In this case, it is efficient to find whether two elements are in the same connected subset – just compare the data they stored, which has a time complexity of $\mathcal{O}(1)$. However, if one wants to combine two connected subset. For example, the subset 1 and subset 2 in (Table 1), the whole array should be traversed in order to find and change the data to the same number, which takes $\mathcal{O}(n)$ time. Therefore, one may wants to somehow sacrifice some of the **Find** performance in exchange to the **Union** performance. Then a tree can be adopted for implementation.

If using a tree to implement Union-Find set, one can represent the Union-Find set as one forest: the connected points are in the same tree. The node in the tree should contain one field as its parent. Then, the `union` operation will become pretty simple: to union A, B, just set $A.parent$ as B (or opposite). The `find` is recursively calling $.parent$, and compare if the two parents are the same.

The main time bottleneck of the above tree-involved method is the *find* step. One should minimize the depth, which will be traversed when one *find* is called. There are two methods: **union by rank** and **path compression**.

The rank of one (parent) node is a measure of the size of the tree spanning from that node. When union by rank, if the rank of two nodes are the same, one increases the rank of the new node by 1; otherwise, set the *parent* of the node with smaller rank as the node with larger rank. In a more intuitive way, union by rank means union by height: (without the inference of path compression,) the node with larger rank will have larger height, because node with $rank = k+1$ is always obtained by merging two nodes with $rank = k$. Thus, by union by rank, the node with smaller height (denote m here) will be a child of the node with larger height (denote n here). The height of the tree after union will be n, rather than the opposite strategy $n+1$.

Path compression is a cleverer strategy. When `find`, one use recursive calls to backwardly traverse from the node to the root. With a common assumption saying that the recently visited data will be visited more frequently in the near future, Path compression re-connect the nodes in the `find` backward traversal path to the root. This method effectively shortens the length of visiting path in `find`.

It can be shown that the time complexity of the Union-Find algorithm is down to the level of inverse Ackermann function $\alpha(n)$, which is a extremely slow-growing function and can be seen as a constant smaller than 4. The theorem said, any sequence of m `union` and `find` operations upon n elements, with the optimization of union by rank and path compression, can be finished with the time complexity of $m \cdot \alpha(m, n) + n$. The detailed step-by-step proof can be seen in [13]. In 1999, it is also found that the complexity of Union-Find is $\Omega(\alpha(m, n))$ [1], which means the upper bound is asymptotically optimal [11]. Moreover, there are some works focusing on verifying the time complexity by state-of-the-art program reasoning methods like separate logic, with an OCaml library implemented offering "a full functional correctness guarantee as well as an amortized complexity bound". [5] [6]

The pseudo-code is shown in 1. [11]

Algorithm 1: Union-Find

```
1  Function construct(x):
2      x.parent ← x;
3      x.rank ← 0;
4  end
5  Function find(x):
6      if x.parent ≠ x then
7          x.parent ← find(x.parent);                    /* path compression */
8      end if
9  end
10 Function union(x, y):
11     xp ← find(x); yp ← find(y);
12     if xp.rank > yp.rank then
                                                          /* union by rank */
13         yp.parent ← xp;
14     else
15         xp.parant ← yp;
16     end if
17     if xp.rank = yp.rank then
18         yp.rank + +;
19     end if
20 end
21 return
```

References.

[1] Stephen Alstrup, Amir M Ben-Amram, and Theis Rauhe. "Worst-case and amortised optimality in union-find". In: *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*. 1999, pp. 499–506 (cit. on p. 2).

[2] Amal Blustein James; El-Maazawi. "optimal case for general Bloom filters". In: (2002) (cit. on p. 5).

[3] *Chapter 8. PNG Basics*. libpng. URL: http://www.libpng.org/pub/png/book/chapter08.html (cit. on p. 17).

[4] *Chapter 9. Compression and Filtering*. libpng. URL: http://www.libpng.org/pub/png/book/chapter09.html (cit. on p. 14).

[5] Arthur Charguéraud and François Pottier. “Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation”. In: *International Conference on Interactive Theorem Proving*. Springer. 2015, pp. 137–153 (cit. on p. 2).

[6] Arthur Charguéraud and François Pottier. “Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits”. In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365 (cit. on p. 2).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. on pp. 16, 17).

[8] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. zlib, 1997. URL: http://www.zlib.org/feldspar.html (cit. on pp. 15, 17).

[9] Xing Huang and Jun Luo. “A Fast Way to Traverse a Generalized Suffix Tree”. In: () (cit. on p. 7).

[10] Amir Kamil. *EECS37notes, Foundations of Computer Science, Release 0.2-beta, UM*. 2021 (cit. on p. 10).

[11] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on pp. 2, 14).

[12] Keith Schwarz. *Chap. 8, Mathematical Foundation of Computing, CS103, Stanford*. 2012 (cit. on p. 13).

[13] Raimund Seidel. *Path Compression and Making the Inverse Ackermann Function Appear Natural(ly), COS 423, Princeton University*. 2009. URL: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/path-compression.pdf (cit. on p. 2).

[14] Devadas Srini. *Intro to Algorithm, 6,006, MIT*. 2011 (cit. on p. 12).

Bloom Filters

- *Algorithm:* Bloom filtering (algo. 2)
- *Input:* A set S with n elements and a searching target x
- *Complexity:* $\mathcal{O}(nk)$ (k is the number of hash functions used in bloom filter)
- *Data structure compatibility:* Hash table
- *Common applications:* Web crawler, news push, history finding

Problem. Bloom Filters

Given a set S with n elements and a target x, return whether x is in the set S. Note that the set is so big such that it is nearly impossible to store all the elements in the memory.

Description

In most of the case, checking whether an element is in a set with n elements is easy. A simple traversal and comparison will gives the result. The time complexity is $\mathcal{O}(n)$. If a hash table has already been built for this set, the time complexity can be further reduced to $\mathcal{O}(1)$ with large possibility. However, all these methods have one critical precondition, which is that the set S should be able to be read from input and then stored in the memory. Storing a set with small size is not a difficulty for modern computers. But either a set with extremely large size, or a set with each of the components to be extremely large, will cost a considerably huge amount of storage. Also, sometimes we care more about whether an element is **not** in the set, like finding one of a thousand persons who is absent. In this condition, if it is not a hash table, the traversal will go over the whole set.

To deal with this problem, notice that the hash table will perform far more better according to the discussion above. It will store a relatively small table instead of the whole raw set, which saves a lot of space and enables the storage of huge input. It also gives a fast check method for a target element. Therefore, the advantage of hash table should be adopted.

But when we use the hash table as our data structure, one problem is inevitable, which is hash collision. In this problem, when the size of the input set is much greater than the buckets of the hash table, the probability of hash collision will be unacceptable. For instance, 100 items are equally hashed to 10 buckets. Then there will be 10 elements end up in being filled to one bucket on average. We mark the buckets that has elements in it to be 1 and others to be 0. Under this crowded condition, if the searching target is hashed to one of the bucket marked by 1, what one can tell by experience is that there is a large possibility that the target is actually not in the original set. Actually, an extreme thought for this scenario would be: Since there are far more elements than buckets, all the buckets are tended to be marked as 1, then no one can tell whether a target is in the original set simply by observing the bucket to be 1.

To make a brief conclusion. If we use a hash table and find that the corresponding bucket is 0, then we are sure that the target is not in the original set. But for those elements that are mapped to 1, the hash collision prevent us from determining whether the element is indeed in the set or not. So the next optimization will focus on reducing the probability of hash collision.

The intuitive thought is to increase the size of the hash table after the load fact is over some threshold. But the general case is that, if we want to reduce the possibility of hash collision to be $a\%$, then a hash table with m buckets can only contain less than $\frac{am}{100}$ elements. This cannot be a space-efficient data structure anymore. Therefore, the Bloom Filter is introduced, with each elements hashed by k hash functions. The experience yields that the hash collision can be effectively reduced. Only all the k hash functions map the target to buckets marked by 1, then the target element is regarded as "in the original set".

The analysis of time complexity of this filtering algorithm is not complicated. Assume that we have not built the hash table yet. Then for each elements, hash it to the hash table using k hash functions takes $\mathcal{O}(k)$ time. Since there are totally n elements in the set, the total time complexity is $\mathcal{O}(nk)$. To judge whether an element is in the original set, one just repeat the hashing process for that particular element, which uses $\mathcal{O}(k)$ time. As a result, the overall time complexity is $\mathcal{O}(nk)$. The detailed algorithm is shown in algo. 2

Algorithm 2: Bloom Filtering

Input : A set S with n elements and a searching target x
Output: Whether x is in S

```
HashTable[0: m-1] initialized with 0;
foreach element a in S do
    for i ← 0 to k-1 do
        calculate hash function h_i(a);
        if HashTable[h_i(a)] ≠ 1 then
            HashTable[h_i(a)] ← 1;
        end if
    end for
end foreach
flag ← true ;
for i ← 0 to k-1 do
    calculate hash function h_i(x);
    if HashTable[h_i(x)] = 0 then
        flag = false; break;
    end if
end for
return flag
```

Review

As mentioned in the algorithm part, as long as all the k hash functions map the target to 1 buckets, the bloom filter will regard the element as in the original set S. One can imagine that if the buckets are filled by different

elements separately, it is possible for the Bloom filter to think targets that are not in the set S to be in S. This is called the **probability of false positives**. This probability can be calculated as follows: [2]
we assume that the Bloom filter uses k hash functions, and a hash table with m buckets.
For one element, the probability that a bucket is not mapped to is

$$1 - \frac{1}{m}$$

while there are k hash functions. The probability that the bucket is 0 after k hashing is

$$(1 - \frac{1}{m})^k$$

Also, we have n elements in total. The probability that the bucket is 0 after n elements are all inserted is

$$(1 - \frac{1}{m})^{nk}$$

which means that the probability for a bucket to be 1 after n elements are inserted is

$$1 - (1 - \frac{1}{m})^{nk}$$

we know that the element is regarded as "in" if all the k hash functions give 1. Thus, for any given element, the Bloom filter will have

$$P = (1 - (1 - \frac{1}{m})^{nk})^k \approx (1 - e^{-\frac{kn}{m}})^k$$

possibility to think it is "in". That is, for an element that is not in S, the filter may have P possibility to regard it as "in".

Based on the discussion above, for given m and n, a Bloom filter should choose

$$k = \frac{m}{n} ln2$$

to minimize the probability of false positives, which result in

$$P \approx (\frac{1}{2})^{\frac{m}{n} ln2}$$

From that, one can also find out that, bigger the hash table is (bigger m), smaller the probability of false positives will be. This is consistent to the experience.

To conclude, the Bloom filter at least offers a method to judge whether an element is in a huge set that is hard to store in memory. On the one hand, it inherits the property of hash table, which make it extremely easy to rule out some elements that are definitely not in the set S. On the other hand, it overcomes part of confusion caused by hash collision by appropriately choosing the number of hash functions k. The overall performance of this algorithm is excellent.

References.

[1] Stephen Alstrup, Amir M Ben-Amram, and Theis Rauhe. "Worst-case and amortised optimality in union-find". In: *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*. 1999, pp. 499–506 (cit. on p. 2).

[2] Amal Blustein James; El-Maazawi. "optimal case for general Bloom filters". In: (2002) (cit. on p. 5).

[3] *Chapter 8. PNG Basics*. libpng. URL: http://www.libpng.org/pub/png/book/chapter08.html (cit. on p. 17).

[4] *Chapter 9. Compression and Filtering*. libpng. URL: http://www.libpng.org/pub/png/book/chapter09.html (cit. on p. 14).

[5] Arthur Charguéraud and François Pottier. “Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation”. In: *International Conference on Interactive Theorem Proving*. Springer. 2015, pp. 137–153 (cit. on p. 2).

[6] Arthur Charguéraud and François Pottier. “Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits”. In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365 (cit. on p. 2).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. on pp. 16, 17).

[8] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. zlib, 1997. URL: http://www.zlib.org/feldspar.html (cit. on pp. 15, 17).

[9] Xing Huang and Jun Luo. “A Fast Way to Traverse a Generalized Suffix Tree”. In: () (cit. on p. 7).

[10] Amir Kamil. *EECS37notes, Foundations of Computer Science, Release 0.2-beta, UM*. 2021 (cit. on p. 10).

[11] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on pp. 2, 14).

[12] Keith Schwarz. *Chap. 8, Mathematical Foundation of Computing, CS103, Stanford*. 2012 (cit. on p. 13).

[13] Raimund Seidel. *Path Compression and Making the Inverse Ackermann Function Appear Natural(ly), COS 423, Princeton University*. 2009. URL: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/path-compression.pdf (cit. on p. 2).

[14] Devadas Srini. *Intro to Algorithm, 6,006, MIT*. 2011 (cit. on p. 12).

Generalized Suffix Trees

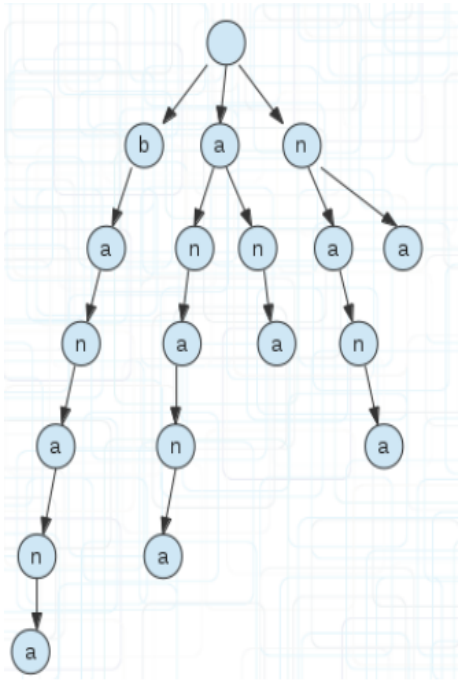
- *Algorithm: Generalize suffix tree building* (algo. 3)
- *Input:* k strings with the maximum length n (Probably shorter than n).
- *Complexity:* $\mathcal{O}(n^2k)$
- *Data structure compatibility:* Generalized suffix tree
- *Common applications:* Search engine building, information finding

Problem. Generalized Suffix Trees

The input are k strings with all the strings having their length smaller or equal to n. The task is to find the longest sub-string that contained by all the strings. Note that a sub-string means that the string must be continuous.

Description

To be general, the input string may contain any character that can be represented using ASCII code. For example, the inputs are ”banana”,”ana”, and ”mybanana”. The output should be ”ana”. Also, the input can be case-sensitive. To solve this problem, we use a data structure called generalized suffix tree. But before that, it is beneficial to first look at the suffix tree. The suffix tree store all the suffix of a string. Each character is stored by a node. Then a root to leave path will represent one of all the possible suffixes of a string.



pic.1 The suffix tree for string "banana"

One thing should be mentioned here. In some cases, one of the suffix will become a prefix of another suffix. Just like pic.1 shows, the suffix "ana" is the prefix of the suffix "anana". If we insert then without other operation, suffix "ana" will be extended due to suffix "anana", which may cause a loss of suffix. In order to prevent such conditions, every time we deal with a new suffix, we add the character "$" at the end of the suffix. This operation will give us an extra leaf node with character "$". In this scenario, the number of leaf nodes will be equal to the number of suffix, which ensures no loss.

For a suffix tree, a crucial fact is that, any sub-string of a given string will be **a prefix of a suffix**. This fact actually make the suffix tree a suitable tool for sub-string finding.

Based on a signal suffix tree, if we combine several suffix trees together, we can get the generalized suffix tree. The main difference is that, apart from the character it contains, the node also contains a stack of numbers that mark down which string the suffix is belonging to. For example, we have three strings: "aka","ana" and "ajax". Then the generalize suffix tree will have a node at first level such that it contains character "a" and a stack with numbers "{1,2,3}".

The method to build a generalized suffix tree is almost the same as building a suffix tree. The basic strategy is to insert each suffix character by character into the existing suffix tree. If the character has already exists, we go to that node and recursively insert the next character. Either inserting a character or go to the existing character takes $\mathcal{O}(1)$ time. For each string with length n. there will be $n+1$ suffixes, with the length from n to 1. To insert all the suffix into the suffix tree takes $\mathcal{O}(n^2) \times \mathcal{O}(1) = \mathcal{O}(n^2)$ time complexity. So, to insert all the suffixes into the generalized suffix tree will take $\mathcal{O}(n^2k)$ time. Then comes to the finding part. We need to find the longest path composed of nodes with the size of stack to be k. The worst case is that the input k strings are identical. In that case, we need to traverse all the nodes in the suffix tree since we cannot rule out any nodes by checking whether the size of the stack is equal to k. To conclude, the generalized suffix tree will give us a method with $\mathcal{O}(n^2k)$ time complexity to find the answer for this problem. In particular, if the number of strings are comparable with the length of the string, i.e., $k = \mathcal{O}(n)$, then the time complexity is $\mathcal{O}(n^3)$. Actually, some subtle tricks can be applied to traverse the generalized suffix tree and reduce the time complexity by avoiding recurrence ([9]).

Algorithm 3: Generalized Suffix Tree Building

Input : k strings with the maximum length to be n
Output: The longest sub-string of these k strings

```
foreach string str do
    Suffix ← [ ];
    for i = 0 to strlen(str) do
        add str[i:] to Suffix;
    end for
end foreach
foreach suffix do
    current_node ← root;                    /* begin insertion at the root of suffix tree */
    for i = 0 to strlen(suffix) do
        if current_node.ischild(suffix[i]) = true then
            current_node ← current_node.gotochild(suffix[i]);
            i++;
        else
            current_node.addchild(suffix[i]);
            i++;
        end if
    end for
end foreach
cur_node ← root;
Function Length(cur_node):
    if cur_node.find_full_stack_child = NULL then
        return 1;
    else
        return 1+max{Length(all the children returned by cur_node.find_full_stack_child)};
        cur_node.maxlengthchild ← the corresponding node; /* Store the pointer for further access
         */
    end if
end
result ← [ ];
cur_node ← root;
while cur_node.maxlengthchild ≠ NULL do
    add cur_node's character to result; cur_node ← cur_node.maxlengthchild;
end while
return result;
```

Review

Now, we have seen that we could find the longest sub-string of k strings with maximum length n in $\mathcal{O}(n^2k)$ time. How efficient is this time complexity?

If we consider the case that there are only two input strings, then the time complexity becomes $\mathcal{O}(n^2)$ int the worst case. The major time is consumed by the process of building a suffix tree. The traversing and finding process can be optimized to be linear time as discussed before. Yet another intuitive algorithm can also solve this particular "two string case". That is, a $n \times n$ matrix is built with the rows to be the characters of the first string and the columns to be the characters of the second string. An detailed example is shown in pic.2.

	b	a	n	a	n	a
a	0	1	0	1	0	1
n	0	0	1	0	1	0
a	0	1	0	1	0	1
/	/	/	/	/	/	/
/	/	/	/	/	/	/
/	/	/	/	/	/	/

pic.2 The matrix built for string "banana" and "ana"

By matching each character in the second string with those in the first string, a two-dimensional matrix with zeros and ones are built. Then, a traversal in the diagonal lines from top left to bottom right will select the longest "1" sequence, whose length corresponds to the length of the longest sub-string. In this algorithm, it takes $\mathcal{O}(n)$ to build the frame of the matrix and $\mathcal{O}(n^2)$ time to fill in the content of matrix slot by slot. It will take another $\mathcal{O}(n^2)$ time to traverse each diagonal line to find the sequence with maximum length. As a result, the time complextity for this algorithm is $\mathcal{O}(n^2)$, which is the same as the algorithm using suffix tree. However, when using this matrix-building algorithm to deal with three input strings, a tree-dimensional matrix is required. The time complexity becomes $\mathcal{O}(n^3)$. On the contrary, if the string length n is much more greater than 3, the time complexity of suffix tree building algorithm remains $\mathcal{O}(n^2)$, which is more feasible than the matrix building algorithm. When the input string grows larger, it is apparent that the suffix tree building algorithm is more efficient in time. More over, the space complexity for the matrix building algorithm is $\mathcal{O}(n^k)$ for the additional k-dimensional matrix, while the space complexity costed by the suffix tree is $\mathcal{O}(n^2k^2)$ due to the stack stored in each node. When k is greater or equal to 3, the suffix tree is also more space efficient than the matrix.

To conclude, the algorithm of suffix tree building is a relatively efficient way in finding the longest sub-string both in time and space.

References.

[1] Stephen Alstrup, Amir M Ben-Amram, and Theis Rauhe. "Worst-case and amortised optimality in union-find". In: *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*. 1999, pp. 499–506 (cit. on p. 2).

[2] Amal Blustein James; El-Maazawi. "optimal case for general Bloom filters". In: (2002) (cit. on p. 5).

[3] *Chapter 8. PNG Basics*. libpng. URL: http://www.libpng.org/pub/png/book/chapter08.html (cit. on p. 17).

[4] *Chapter 9. Compression and Filtering*. libpng. URL: http://www.libpng.org/pub/png/book/chapter09.html (cit. on p. 14).

[5] Arthur Charguéraud and François Pottier. "Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation". In: *International Conference on Interactive Theorem Proving*. Springer. 2015, pp. 137–153 (cit. on p. 2).

[6] Arthur Charguéraud and François Pottier. "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits". In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365 (cit. on p. 2).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. on pp. 16, 17).

[8] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. zlib, 1997. URL: http://www.zlib.org/feldspar.html (cit. on pp. 15, 17).

[9] Xing Huang and Jun Luo. "A Fast Way to Traverse a Generalized Suffix Tree". In: () (cit. on p. 7).

[10] Amir Kamil. *EECS37notes, Foundations of Computer Science, Release 0.2-beta, UM*. 2021 (cit. on p. 10).

[11] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on pp. 2, 14).

[12] Keith Schwarz. *Chap. 8, Mathematical Foundation of Computing, CS103, Stanford*. 2012 (cit. on p. 13).

[13] Raimund Seidel. *Path Compression and Making the Inverse Ackermann Function Appear Natural(ly), COS 423, Princeton University*. 2009. URL: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/path-compression.pdf (cit. on p. 2).

[14] Devadas Srini. *Intro to Algorithm, 6,006, MIT*. 2011 (cit. on p. 12).

Karatsuba's Multiplication

- *Algorithm:* Karatsuba Multiplication (algo. 4)

- *Input:* Two n-bits multipliers x and y
- *Complexity:* $\mathcal{O}(n^{1.58})$
- *Data structure compatibility:* N/A
- *Common applications:* Multiply two extremely long integers

Problem. Karatsuba's Multiplication

For two long integers x and y with n-bits, multiply them and get the result.

Description

When one calculate the time complexity of a certain algorithm, he or she may conventionally assume the time complexity of multiplication as $\mathcal{O}(1)$. Indeed, if two multipliers are both within 32-bits, the time needed for calculating the result is bounded by a constant for a 32-bit computer. However, when the number of bits become unlimited for the multipliers, it will no longer take a computer constant time to deal with a multiplication. In particular, we are now interested in how to compute a multiplication for long integers, say, with both of them having n bits.

As one may have done for tens of years, the traditional multiplication can be regarded as a bit-by-bit operation. Namely, one will take the least significant digit of the second multiplier and do the multiplication bit-by-bit with the first multiplier. Since there are totally n bits for each bit of the second multiplier to cover, the overall time complexity will be $\mathcal{O}(n^2)$, without doubt. The task left is just do some bit-shifting and add all of the results together. The most time consuming step is adding the last two results, which can be treated as an addition for $\mathcal{O}(2n) = \mathcal{O}(n)$ bits, which takes $\mathcal{O}(n)$ time. Thus, the total time complexity is $\mathcal{O}(n^2)$.

Now, considering the size of the integers are relatively big for the algorithm, one may trying to apply the divide-and-conquer method to break the integers down. (The following content refers to [10]). A simple dividing strategy is just cut both numbers into upper $\frac{n}{2}$ bits and lower $\frac{n}{2}$ bits. This results in four numbers, calling them a, b, c, d respectively. If we consider both numbers are decimal numbers (numbers with other base are the same). Then the multiplication can be re-written as

$$x \times y = (a \times 10^{\frac{n}{2}} + b) \times (c \times 10^{\frac{n}{2}} + d) = a \times c \times 10^n + (a \times d + b \times c) \times 10^{\frac{n}{2}} + b \times d \ (*)$$

Note that multiplication by 10^n or $10^{\frac{n}{2}}$ just means left-shifting the number by $\mathcal{O}(n)$ bits, which takes $\mathcal{O}(n)$ time. Then we can write the corresponding recurrence equation to be

$$T(n) = 4T(\frac{n}{2}) + \mathcal{O}(n)$$

applying the Master Theorem, the time complexity is

$$T(n) = \mathcal{O}(n^{\log_2 4}) = \mathcal{O}(n^2)$$

It seems that the divide-and-conquer algorithm does not save as much time. However, the mathematician Karatsuba came up a optimization that can effectively reduce the time complexity, which is then called **Karatsuba's Multiplication**.

For all the four multiplications of integers in the $(*)$ equation, they are actually the four terms got from multiplying $(a + b) \times (c + d)$. In other word, we can let

$$m_1 = (a + b) \times (c + d) = a \times c + a \times d + b \times c + b \times d$$

$$m_2 = a \times c$$

$$m_3 = b \times d$$

Then we can replace the two multiplications in the middle, i.e.

$$a \times d + b \times c = m_1 - m_2 - m_3$$

which can be regarded as an addition with time complexity $\mathcal{O}(n)$.

Consequently, the recurrence relation becomes

$$T(n) = 3T(\frac{n}{2}) + \mathcal{O}(n)$$

Again, using the Master Theorem, the time complexity of Karatsuba's multiplication is

$$T(n) = \mathcal{O}(n^{\log_2 3}) \approx \mathcal{O}(n^{1.58})$$

The detailed algorithm can refer to 4

Algorithm 4: Karatsuba's Multiplication

Input : Two long integers x, y with each of them being n bits
Output: The multiplication result $x \times y$

```
Function Multi(x, y):
    if x, y are 1-bit then
        return x × y;
    else
        a ← x[n : n/2];
        b ← x[n/2 − 1 : 0];
        c ← y[n : n/2];
        d ← y[n/2 − 1 : 0]
        aplusb ← a + b;
        cplusd ← c + d;
        m1 ← Multi(aplusb,cplusd);
        m2 ← Multi(a,c);
        m3 ← Multi(b,d);
        p1 ← m2;
        p2 ← m1 − m2 − m3;
        p3 ← m3; for i ← 0 to n do
            p1 ← p1 << 1;
        end for
        for i ← 0 to n/2 do
            p2 ← p2 << 1;
        end for
        return p1 + p2 + p3;
    end if
end
return
```

Review

The Karatsuba's Multiplication gives a classical way of solving a huge problem by divide-and-conquer algorithm and reduces the time complexity to $\mathcal{O}(n^{1.58})$. Actually, after this algorithm is founded, there are some further development on it. For example, people tried to divide the multipliers into three parts instead of two. This result in the Toom-Cook algorithm. It describe the condition that the numbers are divided into k parts. In particular, dividing them into three parts will reduce the time complexity to $\mathcal{O}(n^{1.465})$. The most recent (until 2011) algorithm for multiplication is called Schonhage-Stramen algorithm with the time complexity $\mathcal{O}(n \log n \log\log n)$.[14]

References.

[1] Stephen Alstrup, Amir M Ben-Amram, and Theis Rauhe. "Worst-case and amortised optimality in union-find". In: *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*. 1999, pp. 499–506 (cit. on p. 2).

[2] Amal Blustein James; El-Maazawi. "optimal case for general Bloom filters". In: (2002) (cit. on p. 5).

[3] *Chapter 8. PNG Basics*. libpng. URL: http://www.libpng.org/pub/png/book/chapter08.html (cit. on p. 17).

[4] *Chapter 9. Compression and Filtering*. libpng. URL: http://www.libpng.org/pub/png/book/chapter09.html (cit. on p. 14).

[5] Arthur Charguéraud and François Pottier. "Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation". In: *International Conference on Interactive Theorem Proving*. Springer. 2015, pp. 137–153 (cit. on p. 2).

[6] Arthur Charguéraud and François Pottier. "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits". In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365 (cit. on p. 2).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. on pp. 16, 17).

[8] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. zlib, 1997. URL: http://www.zlib.org/feldspar.html (cit. on pp. 15, 17).

[9] Xing Huang and Jun Luo. "A Fast Way to Traverse a Generalized Suffix Tree". In: () (cit. on p. 7).

[10] Amir Kamil. *EECS37notes, Foundations of Computer Science, Release 0.2-beta, UM*. 2021 (cit. on p. 10).

[11] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on pp. 2, 14).

[12] Keith Schwarz. *Chap. 8, Mathematical Foundation of Computing, CS103, Stanford*. 2012 (cit. on p. 13).

[13] Raimund Seidel. *Path Compression and Making the Inverse Ackermann Function Appear Natural(ly), COS 423, Princeton University*. 2009. URL: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/path-compression.pdf (cit. on p. 2).

[14] Devadas Srini. *Intro to Algorithm, 6,006, MIT*. 2011 (cit. on p. 12).

PNG (encoding and decoding)

- *Algorithm: PNG encoding* (algo. 5-10)
- *Input:* the original picture
- *Complexity:* $\mathcal{O}(n)$, for a picture with n bytes
- *Data structure compatibility:* binary original picture
- *Common applications:* compress the picture losslessly, useful for pictures on internet

Problem. PNG (encoding and decoding)

The space-consuming raw data pictures can be compressed by lossy compression method but the original picture cannot be recovered perfectly. Thus, some compression algorithm which can reduce the consumed space of original picture (when encoding) without any loss (when decoding) needed to be developed. PNG is one of this kind.

Description

Pictures, are saved as bitmaps which often cost lots of space, if keeping raw data. JPEG compression algorithm effectively compresses raw pictures and save space, however, it is one of *lossy compression*, which means the recovery of original picture is only an approximation.

In many cases, the damage of raw data is not expected when compressing. Thus, the compression which can compress the original picture without any loss, that is, a) to make its size smaller in the most cases (cannot ensure all cases, due to pigeonhole principle [12]), b) to ensure it can be perfectly reconstructed from the compressed PNG format. The requirement a) corresponds with encoding, and the requirement b) corresponds with decoding.

PNG encoding consists of two algorithms: a) filtering, which differentiates the bytes and replaces them with the differences; b) deflate compression algorithm, which is an improved version of LZ77 algorithm: it Huffman-encodes the LZ77 output.

PNG decoding is the "inverse" of PNG encoding, after a series of operations containing parsing the PNG binary, finding out the data chunks, filtering methods, and Huffman dictionary.

Encoding

1 Filtering

Before the real process of compressing (the deflate algorithm), it is feasible to use a simple method called **filtering** to decrease the complexity of raw data first. The filtering method is introduced in the manuel of libpng: [4].

The repeating pattern of bytes are easier to compress than those non-repeating. Thus, for the bytes patterns those contain linearly-changing series, a replacement of the original bytes by the linear combination, usually the difference of two bytes, is taken. Considering the wideness of "linear pattern" in pictures, filtering can make the compression much better.

There are four filtering methods, in Table 2, where we denote the current byte as *currB*, the byte at the left of *currB* is *leftB*, the byte above *currB* is *upB*, and the byte at the left of *upB* is *luB*. When the *currB* is the byte at the topmost, *upB* is 0; when the *currB* is the byte at the leftmost, *leftB* is 0. All the bytes are regarded as unsigned char, unsigned number between 0 and 255.

The standard method to choose which filtering method to use is introduced in Algorithm 5, defining the "good" bytes after the filtering, and determining the optimal method for each line of bytes.

Table 2: Filtering method

Name	Replacement
None	$currB \leftarrow currB$
Sub	$currB \leftarrow (currB - leftB) \mod 256$
Up	$currB \leftarrow (currB - upB) \mod 256$
Average	$currB \leftarrow (currB - \lfloor \frac{leftB+upB}{2} \rfloor) \mod 256$
Paeth	$currB \leftarrow (currB - PaethPredictor) \mod 256$

Algorithm 5: Minimum sum of absolute differences heuristic

Input : A row of bytes *row*

Output: The optimal filtering method (among five methods)

$nonSum \leftarrow 0$; $subSum \leftarrow 0$; $upSum \leftarrow 0$; $avrSum \leftarrow 0$; $paethSum \leftarrow 0$;
foreach $currB \in row$ **do**
  read *leftB*, *upB*, *luB*;
  $PaethPredictor \leftarrow$ `paethPredict(`*leftB*, *upB*, *luB*`)`;
  $nonSum \leftarrow nonSum + |$`(char)` $currB|$; `/* Change unsigned char to char, e.g.128 → −127, 255 → −1 */` $subSum \leftarrow subSum + |$`(char)` $(currB - leftB)|$;
  $upSum \leftarrow upSum + |$`(char)` $(currB - upB)|$;
  $avrSum \leftarrow avrSum + \left|\texttt{(char)}\ (currB - \lfloor \frac{leftB+upB}{2} \rfloor)\right|$;
  $paethSum \leftarrow paeth + |currB -$ `(char)` $PaethPredictor|$;
end foreach
return argmin(*nonSum*, *subSum*, *upSum*, *avrSum*, *paethSum*)

For the *PaethPredictor* in Paeth method, it takes one of the left, upper, and left-upper byte, by the Paeth's method in Algorithm 6.

Notice that in Algorithm 6, the addition and minus are normal arithmetic, while in the final calculation of the replacement in Table 2, the arithmetic with mod 256 is used.[4]

In practice, the pictures are often owning multiple color channels. In that case, filtering will be applied to each color channel, in order to filter the bytes which are closed to each other.[11]

Algorithm 6: Paeth's predicting method

Input : current byte *currB*, the left byte *leftB*, the upper byte *upB*, and the left-upper byte *luB*
Output: the Paeth predictor *PaethPredictor*

Function `paethPredict(`*leftB*, *upB*, *luB*`)`**:**
    base ← *leftB* + *upB* − *luB*;
    PaethPredictor ← $\text{argmin}_{b=leftB,upB,luB}|b - base|$;
    return *PaethPredictor*;
end
read *leftB*, *upB*, *luB*;
return `paethPredict(`*leftB*, *upB*, *luB*`)`;

2 Deflate

The deflate algorithm is a combination of two algorithm, **LZ77** and **Huffman coding**. The filtered bytes are first encoded by LZ77, then by Huffman coding. There is a detailed introduction of Deflate algorithm in the official documentation of zlib: [8].

LZ77 is a compression algorithm, which turns the original list into a series of integer pairs, by the method of replacing the repetitively-appearing sequences into the relative position of them with respect to the position where those sequences first appears. The algorithm is shown in Algorithm 7, which moves a pointer along the list, and keeps a so-called "sliding window" backward the pointer.

Algorithm 7: LZ77 compression

Input : A list of bytes to be compressed *bytes*, the width of sliding window *w*
Output: Compressed *comp*

pointer ← 0;
while *pointer* < *bytes.length*() **do**
    windowStart ← max(*pointer* − *w*, 0); `/* start point of sliding window */`
    maxLength ← 0;
    maxPos ← 0;
    for *i* ← *windowStart*; *i* < *pointer*; *i* + + **do**
        length ← 0; `/* finding the longest match */`
        currPtWin ← *i*;
        currPtBuf ← *pointer*;
        while *bytes*[*currPtWin*] = *bytes*[*currPtBuf*] **do**
            currPtBuf + +;
            currPtWin + +;
            length + +;
            if *length* > *maxLength* **then**
                maxLength ← *length*;
                maxPos ← *pointer* − *i*; `/* backward postion, from pointer to the head of longest matching */`
            end if
        end while
    end for
    if *maxLength* > 0 **then**
        comp ← *comp* + "(*maxPos*, *maxLength*)"; `/* common substring matching found */`
    else
        comp ← *comp* + " *bytes*[*pointer*]"; `/* no matching found, just add the character itself */`
    end if
end while

Sliding window is an area of the original array, having left and right edge, while the right edge is the *pointer*. As *pointer* going forth in one iteration, the right edge of the sliding window will go forth correpsondingly. When the length of sliding window reaches the previously set-uped length *w*, the left edge will also move – that is why

it is called sliding window.

LZ77 compresses the longest match of the substring from *pointer* to the end of the original string, between some substring in the sliding window. The longest match will be replaced by the pair $(maxPos, maxLen)$. The former records the position where the longest match appeared, while the latter records the length of longest match. This pair will be utilized by the decoder.

The time complexity of LZ77 is $\mathcal{O}(n)$, where n is the length of *bytes*, because the length of sliding window is a constant.

Huffman coding is the optimal coding of message, (for the proof see page 433 of the book [7]) which means if replace the bytes with code encoded by Huffman coding, the total length of the encoded bytes will be the minimum. Huffman coding places bytes to different leaves of one tree, ensuring the depth of the most frequent byte is the smallest; after that, denote every node of the tree (except root) by 0 and 1, and get the encoded code by reading 0 and 1 in the path towards the leaves, the most frequently appeared byte will have shortest code. The algorithm of Huffman coding is in Algorithm 8.

Algorithm 8: Huffman coding

Input : An array of original codes
Output: Encoded array and Huffman encoding dictionary

```
1  Function dict ← getCode(root, code):
2      if root.byte ≠ null then
3          code ← code + root.num;
4          dict[byte] ← code;
5          code.clear();
6          return dict;
7      else
8          code ← code + root.num;
9          dict ← getCode(root.left, code);
10         dict ← getCode(root.right, code);
11     end if
12 end
13 Function main():
14     scan the array, get the frequency of each byte;
15     store in a new list arrFreq storing nodes, with fields freq, num and byte;
16     initialize freq, byte, and remain num as null;
17     while arrFreq.num() > 1 do
18         sort arrFreq by the order n1.freq > n2.freq → n1 > n2;
19         create a new node n0;
20         n1, n2 ← two nodes with smallest frequency;
21         arrFreq.pop(n1); arrFreq.pop(n2);
22         n0.freq ← n1.freq + n2.freq;
23         n0.byte ← null;
24         n0.left ← n1; n0.right ← n2;
25         n1.num ← 0; n2.num ← 1;
26         arrFreq.insert(n0);
27     end while
28     root ← arrFreq[0];
29     code ← emptyString();
30     dict ← getCode(root, code);
31     scan the array, and replace char with dict[char], generate encodedArr;
32     return encodedArr and dict.
33 end
```

The time complexity of Huffman coding is $\mathcal{O}(n \log n)$: the scanning in line 14, and the replacing in line 31 cost $\mathcal{O}(n)$; the n-node-merging costs exact $n-1$ times, also $\mathcal{O}(n)$; while the generating of *dict* in function *getCode* requires $\mathcal{O}(n \log n)$. Thus, the total time complexity is $\mathcal{O}(n \log n)$.

A more detailed illustrated example of Huffman coding is shown in Figure 1, from the book [7]. In the illustration, the character with highest frequency *a* get the shortest code 0, while the character that appears least *f* get the longest code 1100.

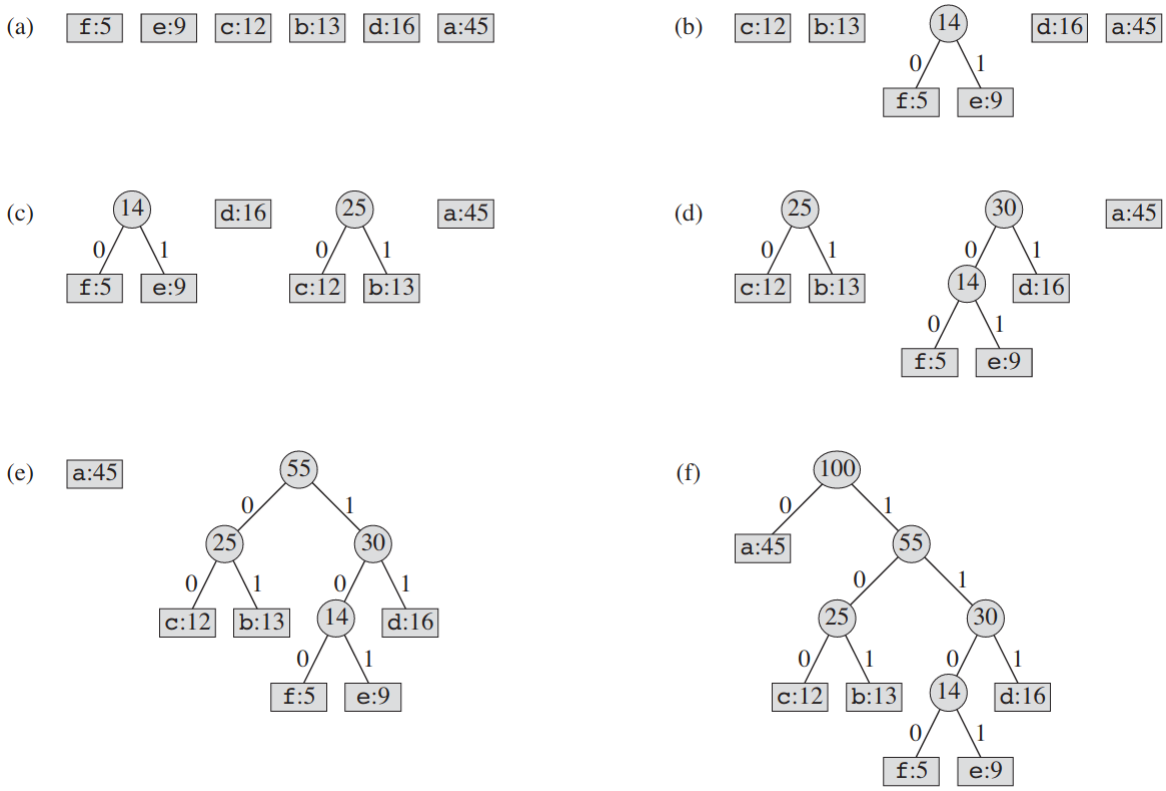


Figure 1: An example of Huffman encoding

Decoding

The decoding of PNG-compressed file is to reverse everything: first apply inverse of deflate – **inflate**, then inverse of filtering – **unfilter**.

1 Inflate

Decoding of deflate algorithm is the inverse of encoding, that is, first apply inverse of Huffman coding and then apply inverse of LZ77. The two algorithms are explained in pseudocode in Algorithm 9 and 10. The Huffman tree is set as one input of Algorithm 9; however, in practice, the Huffman tree is constructed runtime, using the string length of codes (which is the real stored codes in *.png*). [8] The time complexity of the two inverses are the same as their encoding variants.

The application of decoding requires extra efforts to extract messages from the compressed bytes. Here is a project implements the whole PNG-decoding:

2 Unfilter

The filtered code differs from one kind of filtering method to another. Thus, the filter method needs to be recovered from PNG IHDR chunk, one of the three basic PNG chunks: IHDR, IDAT, IEND. [3] Once the filter method is chosen, the unfilter process only remains calculating the inverse of that filter method. If the method

Algorithm 9: Inverse of Huffman coding

Input : *code*, Huffman tree *tree*
Output: Original string *orig*

```
1  counter ← 0;
2  while counter < code.length() do
3  |   curr ← tree;
4  |   while curr.byte = null do
5  |   |   if code[counter] = 0 then
6  |   |   |   curr ← curr.left;
7  |   |   else
8  |   |   |   curr ← curr.right;
9  |   |   end if
10 |   end while
11 |   orig ← orig + curr.byte;
12 |   counter + +;
13 end while
```

Algorithm 10: Inverse of Huffman coding

Input : Encoded *code*
Output: Original *orig*

```
1  pointer ← 0;
2  windowEdge ← 0;                          /* the right edge of the enlarging sliding window */
3  while pointer < code.length() do
4  |   windowEdge + +;
5  |   if code[pointer] ∈ A to Z then
6  |   |   pointer + +;
7  |   |   orig ← orig + code[pointer];
8  |   else
9  |   |   read in "(int, int)" from code, stored in (pos, len);      /* the character is a left brace */;
10 |   |   pointer ← pointer + length("(pos, len)");
11 |   |   orig ← orig + orig[windowEdge − pos, windowEdge − pos + len];
12 |   end if
13 end while
```

uses upper byte to filter, the first row of bytes is unchanged, then unfilter from the first row; if the method uses left byte, start from the first column.

References.

[1] Stephen Alstrup, Amir M Ben-Amram, and Theis Rauhe. "Worst-case and amortised optimality in union-find". In: *Proceedings of the thirty-first annual ACM Symposium on Theory of Computing*. 1999, pp. 499–506 (cit. on p. 2).

[2] Amal Blustein James; El-Maazawi. "optimal case for general Bloom filters". In: (2002) (cit. on p. 5).

[3] *Chapter 8. PNG Basics*. libpng. URL: http://www.libpng.org/pub/png/book/chapter08.html (cit. on p. 17).

[4] *Chapter 9. Compression and Filtering*. libpng. URL: http://www.libpng.org/pub/png/book/chapter09.html (cit. on p. 14).

[5] Arthur Charguéraud and François Pottier. "Machine-checked verification of the correctness and amortized complexity of an efficient union-find implementation". In: *International Conference on Interactive Theorem Proving*. Springer. 2015, pp. 137–153 (cit. on p. 2).

[6] Arthur Charguéraud and François Pottier. "Verifying the correctness and amortized complexity of a union-find implementation in separation logic with time credits". In: *Journal of Automated Reasoning* 62.3 (2019), pp. 331–365 (cit. on p. 2).

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2009 (cit. on pp. 16, 17).

[8] Antaeus Feldspar. *An Explanation of the Deflate Algorithm*. zlib, 1997. URL: http://www.zlib.org/feldspar.html (cit. on pp. 15, 17).

[9] Xing Huang and Jun Luo. "A Fast Way to Traverse a Generalized Suffix Tree". In: () (cit. on p. 7).

[10] Amir Kamil. *EECS37notes, Foundations of Computer Science, Release 0.2-beta, UM*. 2021 (cit. on p. 10).

[11] Manuel. *VE477 – Introduction to Algorithms (lecture slides)*. 2019 (cit. on pp. 2, 14).

[12] Keith Schwarz. *Chap. 8, Mathematical Foundation of Computing, CS103, Stanford*. 2012 (cit. on p. 13).

[13] Raimund Seidel. *Path Compression and Making the Inverse Ackermann Function Appear Natural(ly), COS 423, Princeton University*. 2009. URL: https://www.cs.princeton.edu/courses/archive/spr09/cos423/Lectures/path-compression.pdf (cit. on p. 2).

[14] Devadas Srini. *Intro to Algorithm, 6,006, MIT*. 2011 (cit. on p. 12).