Searching & Shortest Path Project

Group - 7

Course: CSE211

Sec: 4

Submitted by:

- Mahiyan Rahman Takbir 1820332
- Imam Tajnoor Hossain Amrit 1820735
- Fahim Shahriar 2022523
- Dipa Mistre 1822250

Instructor: *Md. Asif Bin Khaled*

Introduction of the paradigm of the algorithm (Brute-force)

Brute-force search, also known as construct and test in computer science, is a problem-solving method and algorithmic paradigm that includes systematically enumerating all possible solution candidates and evaluating if each alternative fits the problem's statement. It's history and whereabouts are still unknown.

Brute Force Algorithms are exactly what they sound like: simple ways of addressing a problem that rely on brute processing power and exhaustive testing rather than complex strategies to increase efficiency.

Every developer's favorite algorithmic paradigm is undoubtedly the brute force technique. The benefit of a brute force technique is that it is very easy, and we are almost certain to find a solution to a problem if one exists. This is also the most thorough approach, since we may consider all options before arriving at a conclusion.

Brute-force search is commonly employed when the issue size is limited or there are problem-specific heuristics that may be used to narrow the collection of potential answers to a manageable size. This is true, for example, in critical applications where algorithm errors might have devastating consequences, or when using a computer to prove a mathematical theorem. Brute-force search can be used as a baseline when comparing different algorithms or heuristics. The most fundamental met heuristic is brute-force search. Backtracking, in which huge sets of solutions are eliminated without being explicitly enumerated, should not be mistaken with brute force search.
This paradigm has several big downsides even after becoming popular and flexible.

These are some important advantages and disadvantages:

Advantages:

By listing all possible candidate solutions for the issue, the brute force technique provides a sure way to locate the proper solution.
It is a general approach that aren't used to a wide range of issues.

For minor and easy issues, the brute force technique is suitable.
It is well-known for its simplicity and may be used as a baseline for comparison.

Disadvantages:

The brute-force method is ineffective. Algorithm analysis for real-time issues frequently exceeds the O(N!) order of growth.

This technique depends less on a strong algorithm design and more on sacrificing the power of a computer system to solve a problem.

The use of brute force methods is time consuming.

When compared to algorithms built utilizing other design paradigms, brute force algorithms are not constructive or innovative.

Introduction of Rabin Karp-Algorithm

The Rabin-Karp algorithm, sometimes known as the Karp-Rabin algorithm, is a string-searching algorithm developed by Richard M. Karp and Michael O that uses hashing to discover matches between a particular search pattern and a text. It uses a rolling hash to quickly filter out text locations that don't match the pattern, and then looks for matches in the positions that remain. Numerous matches of a single pattern or multiple matches for several patterns can be identified using generalizations of the same idea. We can observe that such an algorithm is utilized to find strings when we deal with electronic books such as pdf, documents, etc.

Pseudo Code

Here,

t: The string in which we are looking into

p: The pattern to look for

cs: The total number of characters in a text we may deal with

size: Total length of t – Total length of p

pri: The prime module value

i: 1st index

Therefore:

Pseudo code for finding the initial hash values

HashValues (s, p, cs, pri):

hashOfT := 0

hashOfP := 0

ps := Length(p)

for x := 1 to ps do

*hashOfT := hashOfT + (int(t[x]) * cs^(ps-x))*

*hashOfP := hashOfP + (int(t[x]) * cs^(ps-x))*

hashOfT := hashOfT mod pri

hashOfP := hashOfP mod pri

Pseudo code of Rabin-Karp

RollingHash (t, cs, pri, ps, i, hashOfT):

total := (cs^ps-1) mod pri

*hashOfT := ((hashOfT – int(t[i]) * total) * cs + int(t[i+ps])) mod pri*

if hashOfT less than 0

hashOfT := hashOfT + pri

return hashOfT

SearchingThePattern (t, p, cs, pri, ps, i, hashOfT, hashOfP, size):

if hashOfT = hashOfP then

for y := 1 to ps do

if t[i+y-1] not = p[y] then break

counter := y

if counter = y then

print i

if i = size + 1 then

return

hashOfT := RollingHash (t, cs, pri, ps, i, hashOfT):

SearchingThePattern (t, cs, pri, ps, i = i + 1, hashOfT, hashOfP, size):

Implementation

```
#include <iostream>
#include <cmath>
#include <string>
using namespace std;

const int charSize = 256;
const int myPrime = 1279;

void searching(string mT, string mP, int steps, int hT, int hP, int c, int cP, int pS,
int t)
{
    if (hT == hP)
    {
        for (int z = 0; z < pS; ++z)
        {
            if (mT[t + z] == mP[z])
            {
                ++c;
            }
            else
            {
```

```
            break;
        }
    }
    if (c == pS)
    {
        cout << "Similar Pattern at Index[" << t << "]" << endl;
    }
}

if (t == steps)
{
    return;
}

c = 0;
hT = (hT - int(mT[t]) * cP) * charSize + int(mT[t + pS]);
hT = hT % myPrime;
if (hT < 0)
{
    hT = myPrime + hT;
}

searching(mT, mP, steps, hT, hP, c, cP, pS, ++t);
```

```
}

void rabinKarp(string myText, string myPattern)
{
    if (myText.size() < myPattern.size())
    {
        return;
    }

    int hashT = 0;
    int hashP = 0;
    int charPower = 0;
    int tSize = myText.size();
    int pSize = myPattern.size();

    charPower = pow(charSize, pSize - 1);
    charPower = charPower % myPrime;
    for (int x = 0; x < pSize; ++x)
    {
        hashT = hashT + (int(myText[x]) * pow(charSize, pSize - 1 - x));
        hashP = hashP + (int(myPattern[x]) * pow(charSize, pSize - 1 - x));
    }
    hashT = hashT % myPrime;
```

```
    hashP = hashP % myPrime;


    int counter = 0;
    int travel = 0;
    int steps = tSize - pSize;


    searching(myText, myPattern, steps, hashT, hashP, counter, charPower, pSize,
travel);
}


int main()
{
    string myText;
    string toFind;


    getline(cin, myText);
    getline(cin, toFind);


    rabinKarp(myText, toFind);


    return 0;
}
```

Complexity Analysis

Time Complexity:

i. **Best Case:** O(n+m)
ii. **Average Case:** O(n+m)
iii. **Worst Case:** O(nm)

Suppose we have a string variable name 'text' that stores "AAAAAAAAAAAA" inside it. Now, the pattern we will be searching from 'text' is "AA", hence we have another variable named 'pattern' that stores "AA".

Here, the size of 'text' is 12, and the size of 'pattern' is 2. Let p = 12 & m = 2. Also, m must be always less than or equal to the value of p. Therefore, n = p - m. We calculate and store the hash value for 'pattern' and then for the first m characters of 'text'. We recursively check through 0th index till nth index of 'text' while performing rolling hash.

A rolling hash is a hash function that hashes the input in a window that moves through it. As it traverse, the older value is removed, and the newer value is added. In Rabin-Karp, the window is traversing one character at a time, where (m - 1) + 1 characters are hashed at every recursion. As we do recursion n times, the recursive function costs O(n) time.

At each call, we compare the hash values of the 'pattern' with the first m characters. If they match, we again do a comparison. This time, we compare character by character till all the m characters are matched, one by one from both sides ('pattern' and first m characters from ' text'). If all the characters match, we print the index position and shift the window to the right and finds its hash value for comparison again. It is not necessary to shift the window only if there is a match, we shift the window even when there is no match. This whole process costs O(m) time if the 'pattern' is found within the 'text'. We continue this process for the first n character of 'text'.

In a scenario where there is the repetitiveness of similar characters, for example, the test case we are working with right now. Here, we can see that the hash value for every m character within the 'text' will match with the "pattern's" hash value, so when it does, we compare through m character again, resulting in a worst-case running time of O(nm) time.

To avoid this scenario to happen, we must ensure there is no repetitiveness and spurious hits. When we have a scenario where the hash value of the 'pattern' and the m character of 'text' will match, even when, both the sets have different characters among them, this is called a spurious hit, and this can be avoided at a greater scale if we take a very large value for module inside of a hash function. We can reduce the repetitiveness by choosing unique characters for the 'text'.

Hence, for the best-case and average-case; in maximum scenarios, the occurrence of hash values being similar will only occur when both the sets have the same characters. Only then, we again check and compare through m character. So, in every recursion, there will be less possibility of comparing m character of both sets. Therefore, for n recursion, the running time will reduce to O(n+m) time. Using of smaller sizes of p and m also reduces the running time.

Space Complexity: O(1)

For this algorithm to work, we do not require any extra space to find the result. Here, we just calculate the hash value of each interval of the window and store them into variables. Therefore, by using few variables, it does not cost us any large memory size.

Use Cases

- This algorithm uses hash and hash rolling methods greatly. One of the aims of hash use is to collect and represent a larger amount of data in a smaller form. This algorithm assigns values to map an input to an output by means of Hash functions. This algorithm's rolling hash approaches allows you to determine the Hash value without just releasing the whole line.
- Since this technique is a string search algorithm, it is typically utilized in numerous pattern search strings so long as they are more efficiently searched for the same length. At initially, the hash value is determined by examining the pattern one at a time, without checking for all characters. Then, if the hash value is matched, just each character is checked.
- This algorithm employs modular arithmetic to resolve mathematical issues. In order to avoid extensive manipulation of H values and integer overflows, all mathematics done by Rabin Karp algorithm requires Modul Q. It frequently takes place during hash collisions called fake hits.

Practical uses, where and for what purposes this algorithm is being used?

- Plagiarism is one of the most important practical uses of the Rabin Karp Algorithm. This technique may be used to breakdown texts which have to be compared into string tokens via which commonalities are discovered. In this way a work may readily identify plagiarism.
- This algorithm can be used in biology. Bioinformatics includes the use of IT and IT for genetic sequencing issues to identify patterns of DNA. Both the string search characteristics and the DNA analysis of these methods are collectively employed to discover the pattern set.

- This algorithm is used to find certain text strings that are relevant for the investigation in the digital forensic text.
- Spam filters are used to reject the spam in the string searching properties of this algorithm. For example, suspected spam keywords are searched for using matching algorithms in the text of the e-mail to categorize an e-mail as spam or not.
- This algorithm is used to categorize and arrange data effectively. Search keywords are used for categorization. This allows you to discover the information you are researching more easily by using string matching techniques.
- This algorithm applies to the data packets that include intrusion-related keywords. All of the malicious code is saved in the database, compared to the stored data. The alert is issued if a match is detected.

Recommended Uses

- Because this algorithm has a large influence on bioinformatics, it may be used to investigate crimes such as fingerprint findings, tests of victims' DNA patterns, tests of blood categories, and forensic reports.
- This algorithm is used in a library to organize books according to divisions such as science fiction, history, and literature because of its string search features.
- The algorithm is suitable for categorizing of student information, results, attendance records and plagiarist control in examination copies in schools, colleges and universities.
- This algorithm's pattern matches may be used in office to classify updated files into alphabetic commands. It is also possible to examine the documents if this algorithm is used or not.
- This algorithm can be utilized in business enterprises for the creation of a database. The user may also save vital data and strategies to develop or improve his/her company's safety.
- It may be used extensively in cyber security because harmful programs, the data in the database to determine any intrusion, can be stored by the string searching features of this method.

Comparison with Similar Algorithms

In addition to the Rabin Karp Algorithm, there are several almost identical string searching techniques. Some of them are lower and particular in some sectors are far better than Rabin Karp. The following is a comprehensive analysis of these algorithms and the Rabin Karp Algorithm:

Comparison between the Naive algorithm and the Rabin-Karp algorithm

Naive Algorithm

- The time complexity is O(m(n-m+1) in the worst case. In best case, it is O(n)
- No enhancement scope is available.
- This algorithm compares the pattern with the whole string for each bit.
- No additional calculations are required.
- The algorithm has a Space complexity of O (1)

Rabin-Karp algorithm

- In the worst case it has a time complexity of O(mn) time. In best case, it is O(n-m+1).
- This algorithm can be improved by better hash functions
- The method uses the notion of hash value to match strings rather than naïve (direct match).

- It repeats the calculation process of each sub-hash model's value.
- This algorithm's space complexity is O(m).

Comparison between the Knuth–Morris–Pratt (KMP) algorithm and the Rabin-Karp algorithm

Knuth-Morris-Pratt (KMP) Algorithm

- The preprocessing time complexity of this algorithm is O(m), whereas run time is O(m+n).
- From the beginning, this algorithm may identify faults in character to prevent future character matching.
- This method is significantly more confident and superior to Rabin-Karp Algorithm when it comes to string matching.
- This algorithm's space complexity is O(m)

Rabin-Karp algorithm

- Time Complexity of this algorithm is O(m). In case of run time, best case is O(m), average case is O(n+m) and worst case is O((n-m+1)m).
- This algorithm corresponds to the present substring text, the hash value of the pattern. Here, only then does it start to match specific characters if the hash value matches.
- When string matches, this algorithm is relative slower than KMP and, owing to hash table collisions, is less trustworthy. In addition to the KMP method, it is easy to build Rabin Karp Algorithm.

- The space complexity of this algorithm is also same as KMP algorithm, which is O(m).

Comparison between the Boyer-Moore string algorithm and Rabin-Karp algorithm

Boyer-Moore string algorithm

- The preprocessing time of this algorithm is O(m+ σ). In case of run time, best case is O(mn) and worst case is Ω(n/m).
- This algorithm starts by moving the characters. The bad character rule and good suffix rule is computed to make a shift.
- Compared to the other way around, the Rabin-Karp Algorithm is superior than this algorithm while looking for a number of string matches in a large text.
- Boyer-Moore Algorithm works effectively when its pattern is relatively large and decently proportioned with a wide vocabulary. It doesn't work properly also with too short binary strings or patterns
- The space complexity of this algorithm is also O(m+ σ).

Rabin-Karp algorithm

- Time Complexity of this algorithm is O(m). In case of run time, best case is O(m), average case is O(n+m) and worst case is O((n-m+1)m).
- This method is usually hash-dependent.
- In case of detection of numerous string matches, this algorithm is more reliable than Boyer-Moore. This method, for example, is highly recommended when identifying plagiarism.

- This algorithm is not trustworthy for big matching case patterns. Here's a lot better Boyer-Moore.
- This algorithm's space complexity is O(m).

Comparison between the Smith Waterman algorithm and the Rabin-Karp algorithm

Smith Waterman algorithm

- The time complexity of this algorithm is O(mn)
- It is commonly used for local alignment of the sequence to identify two comparable nucleotide or protein sequence areas. It is also often used in programming dynamics.
- In the event of the DNA identification, it is considerably superior than the Rabin-Karp algorithm for its efficiency and protein segment.
- The algorithm's space complexity is unknown.

Rabin-Karp algorithm

- Time Complexity of this algorithm is O(m). In case of run time, best case is O(m), average case is O(n+m) and worst case is O((n-m+1)m).
- As this algorithm depends heavily on hash functions, it is mostly used to search patterns, for examples of plagiarism.
- The DNA sequencing and fingerprints of this algorithm can be identified. However the confidence of the method is smaller than the Smith Waterman.
- This algorithm's space complexity is O (m)

Conclusion

Plagiarism detection is one of the algorithm's practical applications. The algorithm can quickly search through a document for instances of phrases from the source material, ignoring details such as case and punctuation, given source material. Single-string searching algorithms are impractical due to the large number of strings searched.

Rabin-Karp performs poorly when the majority of the hashes are identical, but it is an excellent pattern search method otherwise. Some of the worst instances can be checked in advance to avoid worst time complexity.

References

- https://medium.com/swlh/rabin-karp-algorithm-using-polynomial-hashing-and-modular-arithmetic-437627b37db6
- https://brilliant.org/wiki/rabin-karp-algorithm/?fbclid=IwAR2wGCTMZYuWblyzl50iH9YswKuHlZKXIws1iVtVza53zufhg_OtKXarZq0
- https://www.baeldung.com/cs/rabin-karp-algorithm?fbclid=IwAR2JjXtvzbgQI08CZ-iabWlMKSRwHw5nrAGkwDXZ-rc3el-67ZIOjxdYrwg
- https://en.wikipedia.org/wiki/Brute-force_search?fbclid=IwAR2V-7a9d64Xf11tK_9k_dgJqxWaBq99nSDLxWGhS3yxg5ScA4-7GpX5vEk
- https://cp-algorithms.com/string/rabin-karp.html?fbclid=IwAR13gr-8dw-oEpwFp7nDVMj-8hb2PoJ1oQqM4xb68GL17vBHDQj_jlahv-U
- https://studyalgorithms.com/theory/algorithmic-paradigms-brute-force/?fbclid=IwAR26YZdiW2Yaib_slgfrSVjQnbm8bieJz3xrR_8o2o9B98xQPcPHMj4DICc
- https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm
- https://pastebin.com/batBt5DF?fbclid=IwAR1z_QsUz8fFrYwrNpCoJZbHKjnPEt_1F1kN2Qeu_euBFtqKl_AyNs3CHiI
- https://stackoverflow.com/questions/36965063/how-spurious-hits-in-rabin-karp-algorithm-equal-to-o-n-q