Interactive Universal Hashing Algorithm

Bo Han, Meet Mukadam, Nidhi Arun Harwani
Department of Computer Science, Rutgers University
New Brunswick, NJ, USA
Email: bo.han@rutgers.edu, meet.mukadam@rutgers.edu, nidhi.arun.harwani@rutgers.edu

***Abstract*— Algorithms have existed and have been used since ancient times to solve problems. Some of the basic algorithms can be explained simply by using the chalk and board method. But, as the complexity or difficulty of the algorithm increases, it becomes problematic for a professor to explain the working of the algorithm using this old method. Nowadays, there are a growing number of algorithm-practice websites which have introduced animations and snippets to help their users understand the advanced algorithms. However, the universities and professors are still disturbed by the in-class display of the algorithms. To help the professors to teach their students and for the better understanding of the algorithm by the students, we are trying to make an animation model of the algorithm. This can also be used by software testers and debuggers as well as researchers. This we believe can contribute to the education of people in computer science.**

I. Project Description

The project is a deterministic animation model of a known advance algorithm. It is based on the visualization of the algorithm. Here, we will try to use visual effects to help the users understand the actual working of the said algorithm. The end result would be a interactive system which would contain illustrations of the advanced algorithm with a user-friendly interface. This website would be used by the professors and teacher's assistants to help the students study the algorithm by showing them a practical example. To make this project feasible, we are restraining ourselves to the animation of one algorithm, the universal hashing algorithm. The major difficulty that we could possibly face is in the animation part of the project.

The project has four stages: Gathering, Design, Infrastructure Implementation, and User Interface.

A. Stage1 - The Requirement Gathering Stage.

- The general system description:
  This project will eventually provide a website with a user-friendly interface to display and illustrate advanced algorithms based on small animations. It would consist a back-end algorithm runner implementation, and a front end animation playing page. The algorithm page will include user customization. Taking the universal hash algorithm as an example, the page will allow users to input the data and choose a different hash function. The back-end and front-end will exchange data by API calls.
- Types of users:
  - User Category 1: This type of users includes the people that are familiar with the computer science stream and have a more advanced knowledge about algorithms. The professor teaching a class of students can be an example of this category of users. Also, software testers/developers fall into this category since they are aware of the algorithm and can use this system to visually track the flow of information in case of debugging. This will assist them in finding bugs or errors more efficiently.
  - User Category 2: This type of users include people who are beginners in algorithm learning and would like to study the algorithm without any input from themselves. For example, any non-technical person or beginners can make use of the website to understand the specific algorithm (universal hashing algorithm).
- The user's interaction modes:
  There will be two modes of interaction:
  - CS community user mode: This mode allows the users to customize their inputs and function.
  - Novice user mode: This mode will use predefined data/inputs, the data that can illustrate the essence of the algorithm, its cons and pros most.
- Real world scenario 1:
  - Scenario1 description: The first real world scenario can be of a class, when teachers use this application to illustrate the mechanism of the algorithm step by step. Users should be allowed to customize the input of algorithm and run it step by step.
  - System Data Input: Inputs for demonstrating Universal Hashing Algorithm. The number of elements and the value of elements.
  - Input Data Types: For simplicity, we are considering only integer values as inputs to the system, so input data types include integers.
  - System Data Output: Returns the array after the hashing algorithm has been applied to the given input.
  - Output Data Types: Again, since the inputs are integers, for simplicity we keep the hash values of integers as integers. So the output data types include integers.
- Real world scenario 2:
  - Scenario1 description: Software developers / testers can have system failures because of inefficient or incorrect hashing implementations. This application

can aid them to visualize the information flow and detect the point of failure quickly and intuitively.
  - System Data Input: Inputs for demonstrating Universal Hashing Algorithm. The number of elements and the value of elements.
  - Input Data Types: For simplicity, we are considering only integer values as inputs to the system, so input data types include integers.
  - System Data Output: Returns the array after the hashing algorithm has been applied to the given input.
  - Output Data Types: Again, since the inputs are integers, for simplicity we keep the hash values of integers as integers. So the output data types include integers.
- Real world scenario 3:
  - Scenario1 description: A user without the background knowledge of the algorithm can study the algorithm by visualizing the flow as the algorithm runs. In this case this inputs are predefined and the system runs on this predefined data.
  - System Data Input: Since this mode is input restricted, the user can only visualize the working of the algorithm based on predefined data and values.
  - Input Data Types: No input, only visualization on predefined input values.
  - System Data Output: Returns the array after the hashing algorithm has been applied to the given input.
  - Output Data Types: Again, since the inputs are integers, for simplicity we keep the hash values of integers as integers. So the output data types include integers.
- Real world scenario 4:
  - Scenario1 description: Generating random hash sets to use for research purposes, since this provides the hash values of the predefined values, these will behave as random hashed values for some other application that calls for this data.
  - System Data Input: Since this mode is input restricted, the user can only visualize the working of the algorithm based on predefined data and values.
  - Input Data Types: No input, only visualization on predefined input values.
  - System Data Output: Returns the array after the hashing algorithm has been applied to the given input.
  - Output Data Types: Again, since the inputs are integers, for simplicity we keep the hash values of integers as integers. So the output data types include integers.
- Tentative Time line of the project:
  Week 1 - Studying the algorithm thoroughly and deciding the form of representation.
  Week 2 - Implementation of the algorithm for the back-end.
  Week 3 - Deciding user inputs based on different modes and implementation of the user interface.
  Week 4 - Integrating the system, testing and documentation.
- Division of work:
  Back-end algorithm implementation - Meet Mukadam, Nidhi Arun Harwani
  User interface implementation - Bo Han
  Integration and Documentation - Nidhi Arun Harwani, Bo Han
  Testing and debugging - Bo Han, Meet Mukadam

B. Stage2 - The Design Stage.

This project consists of a web page exhibiting the algorithm running process and a server to calculate the results off each process. Since universal hashing is a compute-bound algorithm without much data storage requirements, this server will either be stateless or use the memory database only. The first algorithm is naive and simple hashing, which take any ASCII characters, numbers, texts or strings as input. This will show why there is always collision within simple hashing algorithm. The algorithm can also generate input data randomly with fixed hashing slots, so that can further show the collision probability. The second algorithm will be universal hashing based on either prime number or matrix calculation. Both input and output will be the same as the naive hashing algorithm.

- Project Description: The algorithm display module will consists of 3 parts: hashing operator, input array and the hashing slots. When hashing operator is determined and the user inputs the input data, the display module will show the hash key, operator calculation and the hash collision, if there is. The page will interact with our server to get these data. The overall time and space complexity will mainly depend on the hashing algorithm, which will be explained later.
  Space complexity - O(N) ; where N is the number of input elements.
  Time complexity - O(1); attempts to achieve O(1) but cannot always guarantee.
- Flow Diagram.

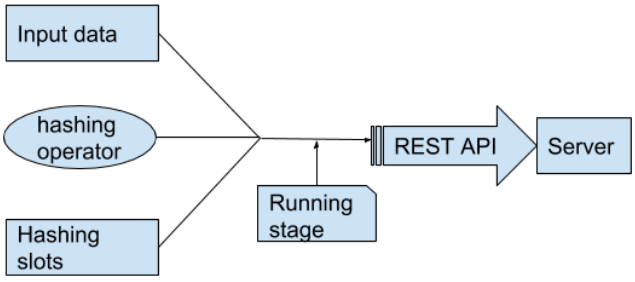


Fig. 1. Flow Diagram

- High Level Pseudo Code System Description.

```
Psuedo Code:
a) Front End
HttpRequest()
        while(true)
                Accept user input: data, operator, length of slot
                for i = 1 to total running stage
                        Http.send( data, operator, length, i )
                        Http.receive()
                        DrawGraphic()
b) Server
HttpServer()
        while(true)
                data = Http.receive()
                result = hash(data)
                Http.send(result)
hash()
        input[n] <-- user
        H = createH();
        for every x = input[i]
                select h from H at random
                hash_list.get(j).add(x)

        print(hash_list)

createH()

        H = list of hash functions h(i)
        m --> number of buckets
        p --> prime number > number of elements in the input array
        for number_of_hash_functions
                h(i) = ((ax+b)%p)%m; where a --> 1:p-1 & b --> 0:p-1
                H.add(h(i))

        return H
```

Fig. 2. Project Pseudo Code

- Algorithms and Data Structures. Both naive and universal hashing will need input data to calculate hashing value, this could be number, character or string. Generally speaking, all these data can be expressed as an array of ASCII code. And output will be a number indicating the hashing slot. As for the hash table itself, we will use an array table with linked list. The size will be M. Here will explain the universal hashing algorithm by prime number method in detail. In universal hash function, we create a family of hash functions. Let the input be x and the number of buckets be m. First, we select a prime number p between m and 2m. Next, we select two numbers a and b from a uniform distribution (randomly) between 1 and p. Now, we generate a hash function (((a*x + b) mod p) mod m) where x is the input integer. The algorithm is shown below.

1. Let **m** be the number of buckets in the hash table.
2. Generate a random prime number **p** between **m** and **2m.**
3. Generate two random numbers **a** and **b** between **1** and **p.**
4. Generate a hash function **(((a*x + b) % p) % m)** where **x** is the input integer.

- Flow Diagram Major Constraints: For a hash algorithm, there is actually no data constraints, if only your input data is 'calculable' data. No matter it is number or string, eventually it will be taken as a series of number. However, here we still put forward a data constraint. This is not because of the limitation of hash algorithm itself, but for the sake of algorithm process exhibition. For the input, it must be an integer with b bits. Value of b can be determined by users. For the hash table size, which is actually the length of slot array, it will be no more than 50.

C. Stage3 - The Implementation Stage.

The project is implemented using Java programming language with HTML and JavaScript used for web designing. The deliverable for this stage include the following items:

- Sample small data snippet:
  The program asks the users for the input, which we have considered as integers to maintain the simplicity of the code, and the number of buckets to fill these numbers. An example of the input is as follows:
  Input string : 2,5,2,7,3,6,6,2,2
  No. of buckets : 5
- Sample small output
  The program outputs the integers filled in their respective buckets. Also, a histogram showing the collision rate is generated. For the above given input, the output we receive is as follows:
  [[2, 7], [6], [3, 6], [5, 2, 2], [2]]
- Working code

```java
import java.util.*;
class univHash{
    static int m; //Number of buckets
    static int n; //Number of input integers
    static List<Integer> prime_list = new ArrayList<Integer>();
    static List<List<Integer>> hash_table = new ArrayList<List<Integer>>();
    static List<String> reconstruct = new ArrayList<String>();
    static Random rand = new Random();

    static void initHashTable() {
        for(int i=0;i<m;i++) {
            List<Integer> list_item = new ArrayList<Integer>();
            hash_table.add(list_item);}
    }
    static void calcPrimes(){
        for (int i = m;i<2*m;i++) {
            boolean isPrimeNumber = true;
            for (int j = 2;j<i;j++) {
                if (i % j == 0) {
                    isPrimeNumber = false;
                    break; // exit the inner for loop
                }
            }
            if (isPrimeNumber) {
                prime_list.add(i);}
            }
    }
```

```
    static void computeHash(int x){
        //Select a prime number p between m,2m;
        int r = rand.nextInt(prime_list.size());
        int p = prime_list.get(r);
        //Select a,b between 1,p;
        int a = rand.nextInt(p)+1;
        int b = rand.nextInt(p)+1;
        //hashValue(x) = (( a*x + b )%p)%m
        int hashValue = (a*x + b)%p%m;
        String str = x+","+hashValue+","+a+","+b+","+p+","+m;
        reconstruct.add(str);
        List<Integer> hash_list = hash_table.get(hashValue);
        hash_list.add(x);
        hash_table.set(hashValue, hash_list);
    }
    public static void main (String[] args){
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter number of buckets: ");
        m = sc.nextInt();
        String input = "2,5,2,7,3,6,6,2,2";
        System.out.println(input);
        String inputs[] = input.split(",");
        n = inputs.length;
        int input_numbers[] = new int[n];
        for(int i = 0; i<n;i++){
          //  System.out.println(inputs[i]);
            input_numbers[i] = Integer.parseInt(inputs[i]);
        }
        calcPrimes();
        initHashTable();
        for(int i = 0; i<n;i++){
            computeHash(input_numbers[i]);
        }
        System.out.println(hash_table);
//      System.out.println(reconstruct);
    }
}
```

- Demo and sample findings
  In this project, we have developed a family of hash functions, called the universal hash function. In universal hash function method, the hash function is selected at random so as to reduce the collision between the inputs. When we use the prime number method, we select the prime number p based on the input, and then we select a and b randomly from a set of numbers. Thus, for even the same input, we may get different values of a, b and p, which would change the value of the output of hash function. This is not the case with any normal hash function. In normal hash function, if we give the same input, it returns the same value of the hash slot, thus increasing collision. For example, if we give an input of 10 two's with 5 buckets, the normal hash function would allocate all two's to the same bucket. But, universal hash function would not typically send it to one bucket.

5 | 2,2,2,2,2,2,2,2,2,2

Get Universal Hash

Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5
2 2 2	2	2 2 2	2	2 2

The example shown above takes the number of buckets (5 here) as the first input and the integers to be hashed (10 two's) as the second input and returns a hash table with values allocated to each bucket shown.

10 | 0,1,2,3,4,5,0,1,2,3,4,5,0,1,

Get Universal Hash

Bucket 1	Bucket 2	Bucket 3	Bucket 4	Bucket 5	Bucket 6	Bucket 7	Bucket 8	Bucket 9	Bucket 10
1 3 4 0	1 4 2 0 0	0 4 5 5 2 5	1 3 0 2 4 4	4 1 5	2 5 2 0 3 1 2	2 0 3 1 5	1 3	4	5 3 3

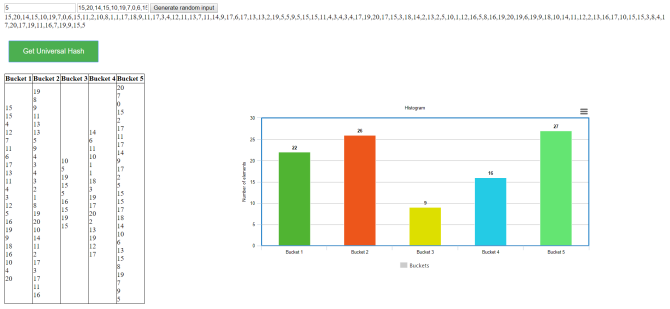
Here is another example where the number of buckets is taken as 10 and the input values are (0,1,2,3,4,5) taken 7 times. As we can see, the numbers in the buckets are not the same.
For retrieval of the input from the hash table, we store the a, b and p values with the input every time. This is done so that we know the hash function used to calculate the hash value for this input. After we get these values form the array, we calculate the hash value again using these values and thus we get the bucket where this input is stored.

D. Stage4 - User Interface.

- The system has a basic user interface with two input boxes and two buttons. The functions of each tab and button is explained below:
  - The first tab takes the number of buckets as the input (integer value).
  - The second tab takes the list of integers to be hashed.
  - The first button "Generate random input" can be used to create a random set of 100 integers between 1 and 20.
  - The second button reading "Get Universal Hash" is used to compute the output of the hash function.
- The information messages or results that will pop-up in response to user interface events are as follows:
  - When we click the "Generate random input", it generates a set of 100 integers which is chosen randomly from the set of natural numbers 1 to 20.
  - Upon clicking the "Get Universal Hash" button on the website, it returns a table showing the integers hashed in each bucket.
  - Also, a histogram is generated showing the number of integers hashed to each bucket.
- The output displayed shows the buckets with the integers that are mapped to that bucket. The process doesn't end here. A back-end process which is used for retrieval of the hash value from the bucket is also present. While hashing as random numbers are selected, these values are stored

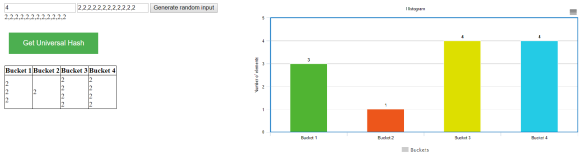
in a dictionary with the input integers as the key i.e. the values a, b, and p are stored for every value of x. This makes it easy for retrieving the given input.

- As the input can be generated by clicking on the generate random input button, we do not have any error messages for the data range.
- A sample snapshot of the final User Interface is as follows:



The above image is an example of the final view of the project where you have generated the random input and then calculated the hash values based on the universal hash family considered. Also, a histogram showing the number of integers mapped to each bucket can also be seen in this view.

- A sample snapshot of the final User Interface with the second method of input is as follows:



The other scenario where the user sets the input himself is the one shown here. The only difference is in the generation of input. The output method remains the same. The buckets are displayed as well as the histogram is generated in this scenario as well.