Hashing - Hard Version

PR Version

Date: 2019-06-10

Chapter 1: Introduction

Given a hash table of size N, we can define a hash function H(x)=x%N. Suppose that the linear probing is used to solve collisions, we can easily obtain the status of the hash table with a given sequence of input numbers. However, now you are asked to solve the reversed problem: reconstruct the input sequence from the given status of the hash table.

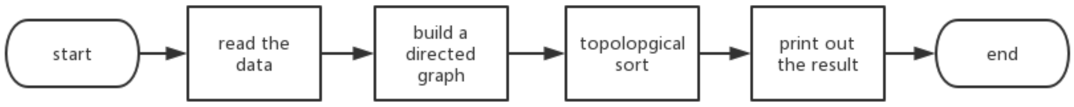
In other words, it's the inverse process of a hash table. Given a hash table sequence and a hash function, we are required to find the original sequence.

Moreover, when there are multiple results, the smallest number is always taken.

From the description of the problem, it's obviously that we can use topological sort to solve the problem. Thus, we store the points that don't have duplicates into a directed graph. Then, we put the points with degree 0 which are greater than 0 into the set. After that, we should pop the minimum point every time and push the points associated with it until the set is empty. When the set is empty, we get the answer of the problems.

Chapter 2: Algorithm Specification

2.1program frame



2.2 build a directed graph

```
BuildGraph(Hash[], n){
(Hash[] is the vector that store the given array, n is the size
of the hash table, A[][] is the adjacency List of Directed Graphs)
    for i:=0 to n
       if Hash[i] >= 0
       then s:=Hash[i]%n;
            if Hash[s]!=Hash[i]
       (If this is true ,it's said that Hash[i] has duplicates.
In other words, the degree of it is greater than 0)
            then for j:=s to i(When j>n, j:=j%n)
                  A[j][i]=1
        (build the adjacency List of Directed Graphs)
}
```

2.3 topological sort

```
TopSort(Hash[], Index[], n, count){
(Hash[] is the vector that store the given array, Index[] is the
order of data input in Hash[], n is the size of the hash table,
count is the number of significant figures)

    initialize degree[MAX]
   (degree[] is used to store the degree of each point in the
Hash[], MAX is the max size of the hash table)
   for i:=0 to n
      degree[i]:= degree of Hash[i]

   for i:=0 to n
      if degree[i]=0 and Hash[i]>0
      then insert Hash[i] into set p

   while p is not empty
      index:=Index[p.begin]
   (p.begin is the first element of p. Index[p.begin] is the
index of the current smallest element(positive integer) in the
Hash[])
         delete p.begin
         output Hash[index]
         for j:=0 to n
              if A[index][j] is not equal to 0
              then degree[j]:=degree[j]-1
                      if degree[j]=0
                      then insert Hash[j] into set p
    end while
}
```

Chapter 3:   Testing Results

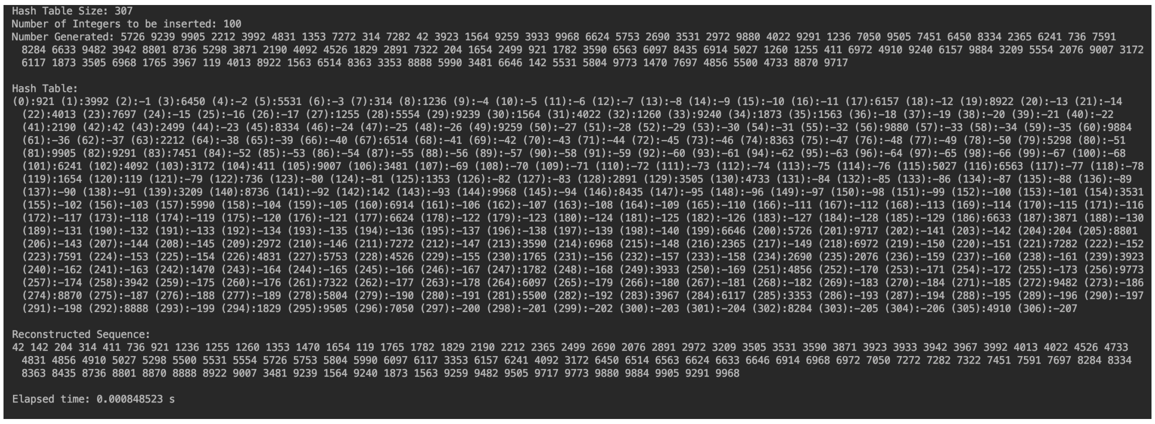
We wrote a randomized test program to automate the testing process, and our program is encapsulated in the target.h header file.

The automated test program:

```
#include <stdio.h>
#include <stdlib.h>
#include <chrono>
#include <time.h>
#include <math.h>
#include <iostream>
#include "target.h"
#define MAXN 10001

int Bucket[10001];
int HashTable[MAXN];

int is_prime(int n)
{
        if(n == 2 || n == 3) {
         return 1;
        }
        if(n == 1 || n == 0) {
         return 0;
        }

        int max = (int) sqrt(n);

        for(int i = 2; i <= max; i ++) {
         // if divides
```

```c
         if(! (n % i)) {
              return 0;
         }
        }

        return 1;
}

// n is an integer, the function returns a prime next to n
int get_next_prime(int n)
{
        for(int i = n; 1; i++) {
         if(is_prime(i)) {
              // printf("prime: %d\n", i);
              return i;
         }
        }
}

int generateRandomInteger(int min, int max)
{
        return min + rand() % (max - min);
}

int generateRandomUniqueInteger(int min, int max)
{
        int res;
        while(Bucket[res = generateRandomInteger(min, max)]);
        Bucket[res] = 1;

        return res;
}

// linear probing applied
int main() {
        // printf("%d", generateRandomInteger(0, 100));
        srand((unsigned)time(0));
        int N, a, count;
        N = 2000;//generateRandomInteger(1, 1000);
        N = get_next_prime(N);
        printf("Hash Table Size: %d\n", N);
        count = N / 3; //generateRandomInteger(1 , N / 10);
        printf("Number of Integers to be inserted: %d\n", count);
        printf("Number Generated: ");
        for(int i = 0; i < count; i++){
         int m = generateRandomUniqueInteger(1, 10000);

         printf("%d ", m);
         int index = m % N;

         for(int j = index; j < index + N; j++) {
              int t = j % N;
              if(!HashTable[t]) {
                   HashTable[t] = m;
                   break;
              }
         }
        }

        printf("\n\nHash Table:\n");

        int negative = -1;
        for(int i = 0; i < N; i++) {
         printf("(%d):", i);
         if(HashTable[i]) {
              printf("%d ", HashTable[i]);
         } else {
              HashTable[i] = negative--;
              // printf("%d ", negative--);
              printf("%d ", HashTable[i]);
         }
        }
        printf("\n\nReconstructed Sequence:\n");

        auto start = std::chrono::high_resolution_clock::now();

        target(N, HashTable);

        auto finish = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double> elapsed = finish - start;

        std::cout << "\n\nElapsed time: " << elapsed.count() << " s\n";
        return 0;
}
```

Following is an example of the output of the test program:



```
Hash Table Size: 307
Number of Integers to be inserted: 100
Number Generated: 5726 9239 9905 2212 3992 4831 1353 7272 314 7282 42 3923 1564 9259 3933 9968 6624 5753 2690 3531 2972 9880 4022 9291 1236 7050 9505 7451 6450 8334 2365 6241 736 7591
  8284 6633 9482 3942 8801 8736 5298 3871 2190 4092 4526 1829 2891 7322 204 1654 2499 921 1782 3590 6563 6097 8435 6914 5027 1260 1255 411 6972 4910 9240 6157 9884 3209 5554 2076 9007 3172
  6117 1873 3505 6968 1765 3967 119 4013 8922 1563 6514 8363 3353 8888 5990 3481 6646 142 5531 5804 9773 1470 7697 4856 5500 4733 8870 9717

Hash Table:
(0):921 (1):3992 (2):-1 (3):6450 (4):-2 (5):5531 (6):-3 (7):314 (8):1236 (9):-4 (10):-5 (11):-6 (12):-7 (13):-8 (14):-9 (15):-10 (16):-11 (17):6157 (18):-12 (19):8922 (20):-13 (21):-14
  (22):4013 (23):7697 (24):-15 (25):-16 (26):-17 (27):1255 (28):5554 (29):9239 (30):1564 (31):4022 (32):1260 (33):9240 (34):1873 (35):1563 (36):-18 (37):-19 (38):-20 (39):-21 (40):-22
  (41):2190 (42):42 (43):2499 (44):-23 (45):8334 (46):-24 (47):-25 (48):-26 (49):9259 (50):-27 (51):-28 (52):-29 (53):-30 (54):-31 (55):-32 (56):9880 (57):-33 (58):-34 (59):-35 (60):9884
  (61):-36 (62):-37 (63):2212 (64):-38 (65):-39 (66):-40 (67):6514 (68):-41 (69):-42 (70):-43 (71):-44 (72):-45 (73):-46 (74):8363 (75):-47 (76):-48 (77):-49 (78):-50 (79):5298 (80):-51
  (81):9905 (82):9291 (83):7451 (84):-52 (85):-53 (86):-54 (87):-55 (88):-56 (89):-57 (90):-58 (91):-59 (92):-60 (93):-61 (94):-62 (95):-63 (96):-64 (97):-65 (98):-66 (99):-67 (100):-68
  (101):6241 (102):4092 (103):3172 (104):411 (105):9007 (106):3481 (107):-69 (108):-70 (109):-71 (110):-72 (111):-73 (112):-74 (113):-75 (114):-76 (115):5027 (116):6563 (117):-77 (118):-78
  (119):1654 (120):119 (121):-79 (122):736 (123):-80 (124):-81 (125):1353 (126):-82 (127):-83 (128):2891 (129):3505 (130):4733 (131):-84 (132):-85 (133):-86 (134):-87 (135):-88 (136):-89
  (137):-90 (138):-91 (139):3209 (140):8736 (141):-92 (142):142 (143):-93 (144):9968 (145):-94 (146):8435 (147):-95 (148):-96 (149):-97 (150):-98 (151):-99 (152):-100 (153):-101 (154):3531
  (155):-102 (156):-103 (157):5990 (158):-104 (159):-105 (160):6914 (161):-106 (162):-107 (163):-108 (164):-109 (165):-110 (166):-111 (167):-112 (168):-113 (169):-114 (170):-115 (171):-116
  (172):-117 (173):-118 (174):-119 (175):-120 (176):-121 (177):6624 (178):-122 (179):-123 (180):-124 (181):-125 (182):-126 (183):-127 (184):-128 (185):-129 (186):6633 (187):3871 (188):-130
  (189):-131 (190):-132 (191):-133 (192):-134 (193):-135 (194):-136 (195):-137 (196):-138 (197):-139 (198):-140 (199):6646 (200):5726 (201):9717 (202):-141 (203):-142 (204):204 (205):8801
  (206):-143 (207):-144 (208):-145 (209):2972 (210):-146 (211):7272 (212):-147 (213):3590 (214):6968 (215):-148 (216):2365 (217):-149 (218):6972 (219):-150 (220):-151 (221):7282 (222):-152
  (223):7591 (224):-153 (225):-154 (226):4831 (227):5753 (228):4526 (229):-155 (230):1765 (231):-156 (232):-157 (233):-158 (234):2690 (235):2076 (236):-159 (237):-160 (238):-161 (239):3923
  (240):-162 (241):-163 (242):1470 (243):-164 (244):-165 (245):-166 (246):-167 (247):1782 (248):-168 (249):3933 (250):-169 (251):4856 (252):-170 (253):-171 (254):-172 (255):-173 (256):9773
  (257):-174 (258):3942 (259):-175 (260):-176 (261):7322 (262):-177 (263):-178 (264):6097 (265):-179 (266):-180 (267):-181 (268):-182 (269):-183 (270):-184 (271):-185 (272):9482 (273):-186
  (274):8870 (275):-187 (276):-188 (277):-189 (278):5804 (279):-190 (280):-191 (281):5500 (282):-192 (283):3967 (284):6117 (285):3353 (286):-193 (287):-194 (288):-195 (289):-196 (290):-197
  (291):-198 (292):8888 (293):-199 (294):1829 (295):9505 (296):7050 (297):-200 (298):-201 (299):-202 (300):-203 (301):-204 (302):8284 (303):-205 (304):-206 (305):4910 (306):-207

Reconstructed Sequence:
42 142 204 314 411 736 921 1236 1255 1260 1353 1470 1654 119 1765 1782 1829 2190 2212 2365 2499 2690 2076 2891 2972 3209 3505 3531 3590 3871 3923 3933 3942 3967 3992 4013 4022 4526 4733
  4831 4856 4910 5027 5298 5500 5531 5554 5726 5753 5804 5990 6097 6117 3353 6157 6241 4092 3172 6450 6514 6563 6624 6633 6646 6914 6968 6972 7050 7272 7282 7322 7451 7591 7697 8284 8334
  8363 8435 8736 8801 8870 8888 8922 9007 3481 9239 1564 9240 1873 1563 9259 9482 9505 9717 9773 9880 9884 9905 9291 9968

Elapsed time: 0.000848523 s
```

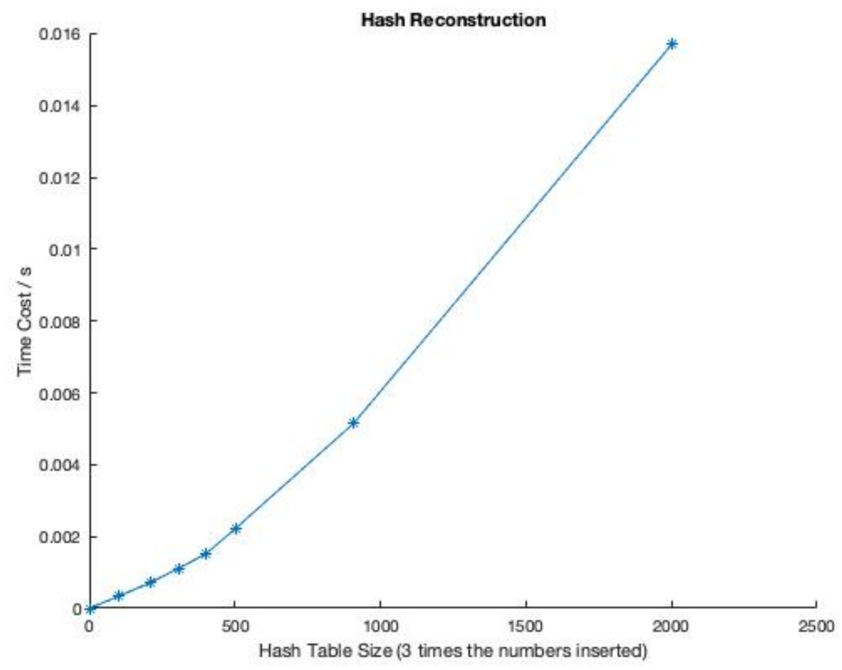
With this automated program, we are able to make a series of simulations when the hash table size is at the smallest prime number greater than 100, 200, 300, 400, 500, 900 and 2000. For each size, the integers generated are between 1 and 10000, and the lengths of the sequences inserted are 1/3 the size of each hash table to guarantee a smaller probability of primary collision. We made 3 parallel simulations for each size, calculated the average and plotted a graph in terms of size and time (in seconds) consumed for topological sort.

Table: Time Consumption for each table size

Table Size	Trial 1	Trial 2	Trial 3	Average Time
101	0.000335237	0.000369189	0.000314942	0.000339789
211	0.000626975	0.000735325	0.000802515	0.000721605
307	0.00106561	0.0010096	0.00126865	0.00111462
401	0.00153683	0.00152334	0.00151085	0.00152367
503	0.00221961	0.0023117	0.00216632	0.00223254
907	0.00511226	0.00481886	0.00550359	0.00514490

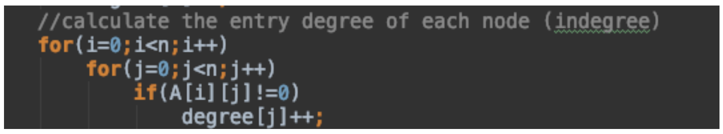
2003	0.0161041	0.0152485	0.0158316	0.0157281

Graph: Hash Table Size – Time Consuption



Chapter 4: Analysis and Comments

As we can infer from the Graph in Chapter 3 intuitively, the time complexity of our algorithm is probably $O(n^2)$. Now let’s confirm that with a more detailed analysis. As we can see in the source code:



```
//calculate the entry degree of each node (indegree)
for(i=0;i<n;i++)
    for(j=0;j<n;j++)
        if(A[i][j]!=0)
            degree[j]++;
```

We used an incident matrix representation of the graph and applied a 2-level for-loop to calculate the indegree of each node in the graph. This step is the decisive one in our algorithm, thus obtaining an overall $O(n^2)$ time complexity.

In conclusion, the performance of the program in the simulation process is in accordance with our expectation in algorithm analysis.

Appendix: Source Code (if required)

At least 30% of the lines must be commented. Otherwise the code will NOT be evaluated.

References

None

Author List

Declaration

We hereby declare that all the work done in this project titled " Hashing - Hard Version " is of our independent effort as a group.