Unique Permutation Hashing, Experiment Results

by

Patrick Gichuiri

Technical Report #10-08

August 24, 2010

UNIQUE PERMUTATION

Instructor: Prof. Shlomi Dolev

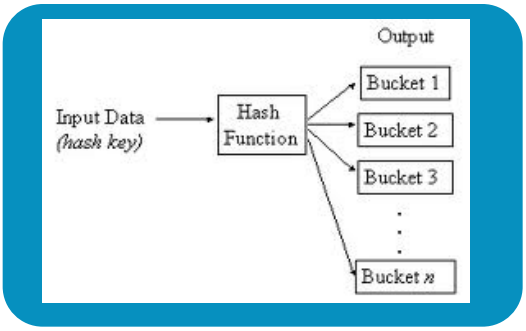
Patrick Gichuiri

Summer Internship Report • Ben-Grunion University of the Negev • August 24, 2010

REPORT

Patrick Nginyah Gichuiri • email: gichuiri@mit.edu • Ben-Grunion University of the Negev

Table of Contents



Patrick Nginyah Gichuiri • email: gichuiri@mit.edu • Ben-Grunion University of the Negev

Introduction:

The Unique Permutation hash function,an ingenious proposition by Prof. Shlomi Dolev, offers superior and preferable data structure hashing implementations of applications that support dictionary operations of *inserting , searching & deleting* .It does so in comparison to other hashing functions such as linear probing,quadratic probing and doubling hashing(currently one of the preeminent methods available for open addressing).

Performance analysis of hashing computations:

Below is code (written in the python language) that implements the newly proposed Unique Permutation Hashing and provides a performance scheme of its hashing properties in comparison to linear probing, quadratic probing and double hashing.

```
from random import *

def factorial(x):
  If x == 1:
    return 1
  else:
    return x * factorial(x-1)

Tests = input('How many tests do you want to run?')
tableSize = input('Please state the size of the table you would wish to have...')
            # preferably should be prime
r = factorial(tableSize)   # This reduces the probability for key duplication
Table = [False] * tableSize
empty = False
```

```
probes = 0
loadFactor = input('Choose your preferred loadFactor...')
        # (Numbers of Keys / Table slots)
print '\n'
knum = int(loadFactor*tableSize)  # Number of keys based on load factor

# Generates a set of random keys to be hashed into the table
def setOfKeys():
  lists = []
  i = 0
  while i < knum:
    lists.append(randint(0,r-1))
    i += 1
  return lists

def numberOfTests():
  keys_list = []
  for i in range(Tests):
    keys_list += [setOfKeys()]
  return keys_list

key_sets = numberOfTests()

def linearProbe(key):
  global probes
  h = ( key % tableSize)
  for i in range(len(Table)):
    h1 = ( (h + i) % len(Table) )
    if Table[h1] == empty:
      Table[h1] = key
      return
    else:
      probes += 1   # Number of probes made
```

```
def quadHashing(key):
  global probes
  h = ( key % tableSize)
  for i in range(tableSize):
    h1 = (( h + i + 3*(i**2)) % tableSize)
    if Table[h1] == empty:
      Table[h1] = key
      return
    else:
      probes += 1   # Number of probes made

def doubleHashing(key):
  global probes
  h = ( key % tableSize)
  h1 = 1 + (key % (tableSize-1))
  for i in range(tableSize):
    h2 = ( h + (i * h1) ) % tableSize
    if Table[h2] == empty:
      Table[h2] = key
      return
    else:
      probes += 1  # Number of probes made

def variance(listOfValues,mean):
   variance = 0
   for i in listOfValues:
     variance = variance + (i - mean)**2
   variance = float(variance) / (len(listOfValues))
   return variance

def perm(N):  # Generates permutations and helps with visual puporses
   table_size = len(N)      # to ensure that permutations generated in Unique
```

```
    if table_size <= 1:      # are indeed correct
        return [N]
    return [p[:i]+[N[0]]+p[i:] for i in xrange(table_size) for p in perm(N[1:])]

def permutations(n):
    permutations = perm(range(1,n+1))
    return permutations



def convert(probe,seq): # finds next probe number not taken in the probe_seq
        seq = sorted(seq)
        index = 0
        if seq == []:
            return probe
        while index < len(seq) and seq[index] <= probe:
            probe += 1
            index += 1
        return probe

class Initialize:
    def __init__(self,i,key,probe_seq): # Gets input of the
        self.tableSize = tableSize    # necessary data
        self.i = i # tracks the number of probes already made
        self.Kprime = key
        self.probe_seq = probe_seq
        self.M = factorial(tableSize)
        self.key = key

    def __str__(self): # Prints the Permutation Object created
        return ' Number of slots in the table = '+ str(self.tableSize) +'\n' \
        ' Probe counter(tracks # of probes made) at initial state = '+str(self.i)\
        +'\n' \
        ' Probe sequence = '+ str(self.probe_seq) + '\n' \
```

```
    ' Key to be hashed ='+str(self.key) + '\n' \
    ' M (N-1-i) = '+ str(self.M)

  def get(self):
   return self.i,self.Kprime,self.probe_seq

  def computation(self): # Finds the kth permutation
    if (self.tableSize - self.i) == 0:
      return
    M = self.M / (self.tableSize - self. I)
    probe = (self.Kprime / M) + 1 #Probes
    k = self.Kprime % M # indicates probed numbers so far
    next_probe = convert(probe,self.probe_seq)
    self.probe_seq.append(next_probe)
    self.i += 1 # Keeps track of the probes already made
    self.M = M
    self.Kprime = k
    if len(self.probe_seq) == self.tableSize:
      return next_probe
    return next_probe

  def Hash(self,key): # Uniquely Inserts a key to its appropriate slot
    global probes                  # in the table
    full = (empty not in Table)
    if full:
      print 'Table Overload: The table is FULL'
      return
    index = self.computation()
    if index is None: # Error in Inserting Key
      return
    if Table[index - 1] == empty:
       Table[index - 1] = key
    else:
```

```
        self.Hash(key)
        probes += 1

def create():  # Creates a list of objects
  MasterList = []
  Objects = [] # list of objects created.Important for visual reasons only
  j = 0
  while j < len(key_sets):
    ListWithinList = []
    for key in key_sets[j]:
      ListWithinList.append(Initialize(0,key,[]))
    MasterList.append(ListWithinList)
    j += 1
  return MasterList

  #Prints the Physical Representation of the objects created.
  #Code is however skipped as its only vital for visual purposes
  for lists in MasterList:
   list1 = []
   for objects in lists:
     list1.append(objects.get())
   Objects.append(list1)
  print Objects

def genericTest(name,func = None):
  global Table
  global probes
  j = 0
  probeRecord = []
  keyObjects = create()
  while j < len(key_sets):
    Table = [False] * tableSize # Empties table before beginning new test
    probes = 0
```

```
    if name != 'UNIQUE HASHING TEST':
      for key in key_sets[j]: # Iterates through keys in each list of keys
        func(key)
        #print probes
        #print Table ### uncommment to print the various individual tables
      probeRecord.append(probes)
    else:
      for objects in keyObjects[j]:
        objects.Hash(objects.key)
       # print probes
      probeRecord.append(probes)
    j += 1

  mean = float(sum(probeRecord)) / len(key_sets)
  vary = variance(probeRecord,mean)
  stddev = var**(0.5)

  print name
  print '------------------------'
  print 'Table slots = ' ,tableSize
  print 'Load factor(# keys/#table entries) = ',loadFactor
  print 'Number of keys = ',knum
  print 'Number of tests conducted = ',len(key_sets)
  print 'Total Number of probes = ', sum(probeRecord)
  print 'Average number of probes = ', mean
  print 'Variance(Probes) = ',var
  print 'Standard Deviation of probes = ',stddev

def test1():
  genericTest('LINEAR PROBING TEST',linearProbe)

def test2():
  genericTest('QUADRATIC PROBING TEST',quadHashing)
```

```
def test3():
  genericTest('DOUBLE HASHING TEST',doubleHashing)

def test4():
  genericTest('UNIQUE HASHING TEST')

#########################
# TEST PANEL#       #
def masterTests():    #
  test1()         #
  print '\n'       #
  test2()         #
  print '\n'       #
  test3()         #
  print '\n'       #
  test4()         #
#########################

masterTests()
```

Upon Running the above code, one gets statistics on the number of probes made for each hashing technique. The tests are run using the same set of keys.The large number of tests is to ensure that the

```
LINEAR PROBING TEST
-----------------------
Table slots =  97
Load factor(# keys/#table entries) =  1
Number of keys =  97
Number of tests conducted =  1000
Total Number of probes =  533844
Average number of probes =  533.844
```

Variance(Probes) = 22788.641664
Standard Deviation of probes = 150.959072811

QUADRATIC PROBING TEST

Table slots = 97
Load factor(# keys/#table entries) = 1
Number of keys = 97
Number of tests conducted = 1000
Total Number of probes = 395047
Average number of probes = 395.047
Variance(Probes) = 5674.056791
Standard Deviation of probes = 75.3263353084

DOUBLE HASHING TEST

Table slots = 97
Load factor(# keys/#table entries) = 1
Number of keys = 97
Number of tests conducted = 1000
Total Number of probes = 311074
Average number of probes = 311.074
Variance(Probes) = 3116.102524
Standard Deviation of probes = 55.8220612661

UNIQUE HASHING TEST

Table slots = 97
Load factor(# keys/#table entries) = 1

Number of keys = 97

Number of tests conducted = 1000

Total Number of probes = 308883

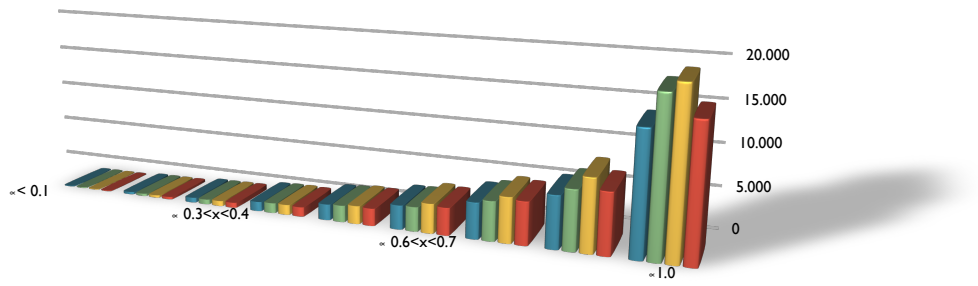
Average number of probes = 308.883

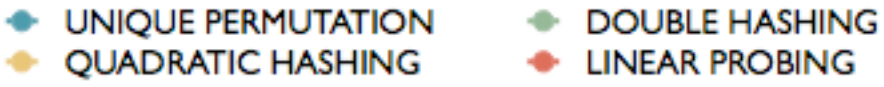
Variance(Probes) = 2740.773311

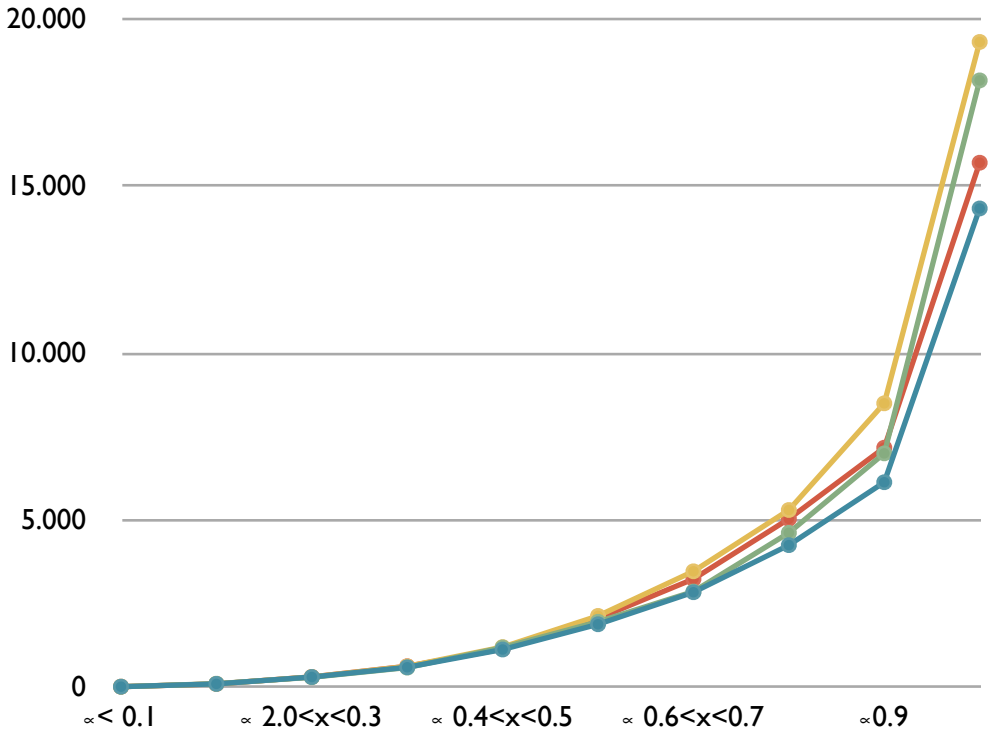
Standard Deviation of probes = 52.3523954657

The snippet above clearly shows the efficiency of the Unique Permutation Hash function.

Below are statistics and graphical representations of tests run with table size(n) = 11







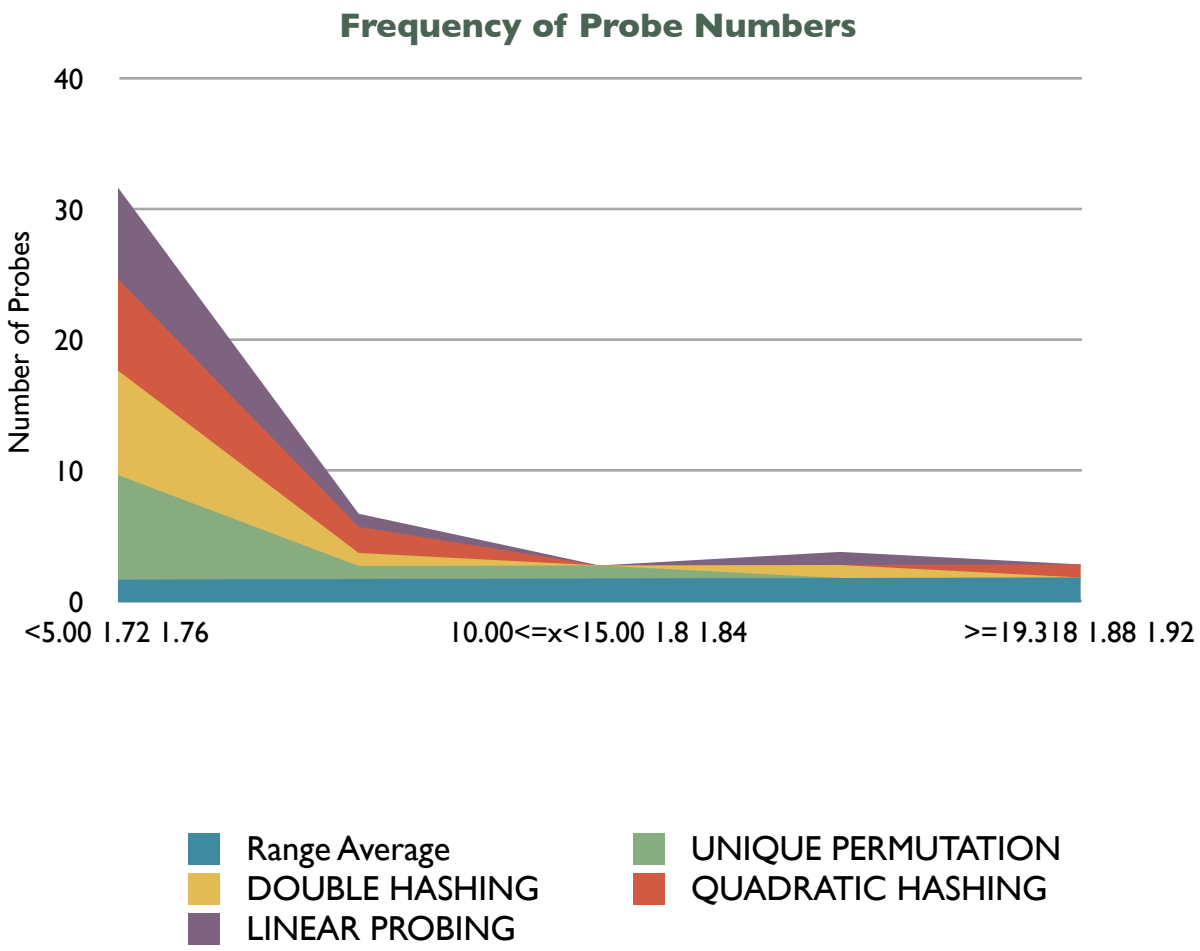
Patrick Nginyah Gichuiri • email: gichuiri@mit.edu • Ben-Grunion University of the Negev

Calculated Statistics

	UNIQUE PERMUTATION	DOUBLE HASHING	QUADRATIC HASHING	LINEAR PROBING
Count (ALPHA)	10	10	10	10
Mean	3.147	3.668	4.085	3.528
Median	1.492	1.560	1.654	1.586
Standard deviation	4.410	5.571	6.007	4.887
Variance	19.44512	31.03161	36.08057	23.87900
Alpha	0.05	0.05	0.05	0.05
T-value	2.26	2.26	2.26	2.26
Confidence interval	2.73309	3.45263	3.72293	3.02870
Upper limit	5.87959	7.12073	7.80813	6.55620
Lower limit	0.41341	0.21547	0.36227	0.49880
T-interval	3.15426	3.98469	4.29284	3.49234
Upper limit	6.30076	7.65279	8.37804	7.01984
Lower limit	-0.00776	-0.31659	-0.20764	0.03516

Frequency Table

Range	UNIQUE PERMUTATION	DOUBLE HASHING	QUADRATIC HASHING	LINEAR PROBING
<5.00	8	8	7	7
5.00<=x<10.00	1	1	2	1
10.00<=x<15.00	1	0	0	0
15.00<=x<19.318	0	1	0	1
>=19.318	0	0	1	0



Patrick Nginyah Gichuiri • email: gichuiri@mit.edu • Ben-Grunion University of the Negev

SUMMARY

Overall, unique permutation represents improvement over other hashing functions , particularly double hashing . This is because the former uses permutations of the table size as unique identifiers to insert keys while the later has a max number of probe sequences that range to the table size exponentiated to 2 .

Patrick Nginyah Gichuiri • email: gichuiri@mit.edu • Ben-Grunion University of the Negev

BIBLIOGRAPHY

Cormen , Thomas ; Leadsperson , Charles ; Rivets ,Ronald ." INTRODUCTION TO ALGORITHMS" The MIT Press : Cambridge , Massachusetts London , England

Shlomi Dole ; Limo Lahiani ; Yinnon Aviv ."UNIQUE PERMUTATION HASHING "

Technical Report #2009-03,Revised version of # 2007-09