Lab 8 : Algorithm Design

Steven Robles *

University of Texas, El Paso, TX

May 10, 2019

This paper aims to demonstrate the different approaches of algorithm design methods there are. More specifically, this lab demonstrates two different algorithm designs: randomization and backtracking. Each program each of these designs functions better under different circumstances. The parts of this lab explores and showcases these circumstances while demonstrating each algorithm design implementations.

Contents

I. Introduction

Algorithm design is an important aspect when developing effective program for specific jobs. The design of the algorithm is highly dependent of the intended task that the program is to complete. There are five primary algorithm design techniques: 1. Divide and Conquer 2. Greedy Algorithms 3. Dynamic Programming 4. Backtracking 5. Randomized Algorithms As mentioned before, each of the algorithm design techniques are intended for specific purpose. For example, we commonly design the divide and conquer technique in sorting algorithms such as merge sort and quick sort. Greedy algorithm are design to reach a solution to a task with the least amount of "work". Such an algorithm is Dijkstra's algorithm which find a the path with the least total cost form vertex A to vertex B in a weighted graph. Dynamic programming finds solutions to a given tasks through recursions and creating sub-problems to combine and compute.

In this lab, we will focus and the last two algorithm design techniques: Backtracking and Randomized algorithms. Randomized algorithms are typically utilized in situations to avoid pathological cases. Utilizing this algorithm design in a quick sort function is one type of application. Another application for randomized algorithms is to compute a solutions that are correct with a high probability. This computations mainly consist of math algorithms to verify a certain result. Backtracking is a clever implementation of a exhaustive search. If a certain solution is incorrect, then the algorithm simply backtracks to its previous steps and continues from there.

*ID Number: 80678755

II. Design

This lab is intended to showcase an example for randomized algorithm and backtracking algorithm design techniques. It only has two part and it is pretty straight forward. Of course, the results of each implementation in shown as well as the time cases of the backtracking algorithm. The following two sub sections describe each of the design algorithm.

A. Part 1: Randomized Algorithm

The first part of the lab is to demonstrate how a randomized algorithm design can "discover" trigonometric identities through evaluations. This is achieved by taking all of the different combinations of the given set of trigonometric functions and compare them to one another. The resulting equality's will indicate and verify if the trigonometric function holds (i.e. $tan(x) = sin(x)/cos(x)$). This is achieved by forming combinations through a list containing strings of the functions and iterating through them in $O(n^2)$ time. After a combination set is extracted, they would both be evaluated by generating a random number between to range. When the evaluation is complete, the difference between the two set of functions is computed the analyze the similarity of the results. If the difference is greater than a provided tolerance (in this case its *0.0001*), then the two functions are no longer considered to be the same by definition. The evaluation then repeats for *999* more times to ensure the results are correct with a high probability.

Since this parts deals with evaluating the equality's of any set of two trigonometric functions, the functions in the list are combined in a perumation approach. This eliminates the cases where the qualities is already computed and effectively reducing the amount of cost and time it takes to complete the task. Also, in order to increase the accuracy of the evaluations, only the tolerance needs to be decreased as well as the number of iterations increase. Executing the opposite will yield less accurate results. The whole program is calculated to take $\mathrm{O(n^3)}$ time.

B. Part 2: Backtracking Algorithm

The second part of the lab is to demonstrate how a backtracking algorithm design can solve a specific kind subset sum problem. In this lab, the backtracking algorithm technique is tasked to find a way to partition a set of integers, S, into two subsets; S1, and S2. The summation of the two subsets must be equal to each other and when combiner, must be equal to the total summation of the original set, S. Also, the two new subsets must not share any element of S in common. This task is suitable for implementing the backtracking technique since it deals with the combination of integers and must keep search if a certain set of integers does not reach the solution.

First thing I noticed, a solution can not be reached if the total summation of the original set of integers S is an odd number. If its an odd number, then it cannot be divided into two equal parts and only containing integer numbers. So before the backtracking function is called, set S must first passed through this first test case. If the set S passes through the initial test case, it the proceeds to be evaluated through the backtracking algorithm. The backtracking algorithm is a recursive functions which returns only true or false statements while having the sets as lists in the parameters. The true and false statements help the function track which possibility it already have computed and thus helping it backtrack to another possible solution if it finds a combination that does not work. Admitidly, the function only serves to find the solution of one subset, S1. Subset S2 is appended with the remanding numbers of S and has its total summation of the elements checked if a solution of S1 is found. If S1 returns as an empty lists, then there exits no solution within S which could be partitioned in two equal parts.

III. Results

After the implementations of both algorithm design techniques, the results of both parts are displayed. Part 1 only shows the results of the equivalence between two trigonometric identities. Part 2 shows the the two subset if they exits and if they do not, the program indicates so. The following are examples of the program running

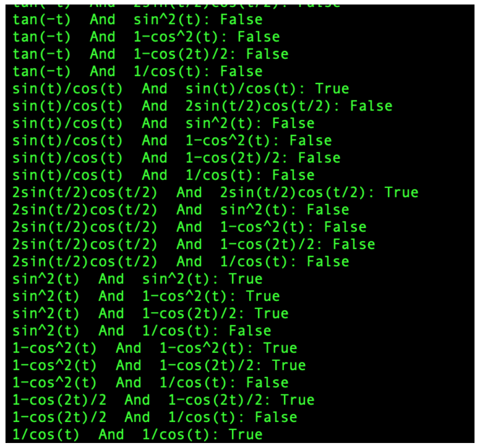
Figure 1: Trigonometric Identity Equivalence

```
sin(t)  And  sin(t): True
sin(t)  And  cos(t): False
sin(t)  And  tan(t): False
sin(t)  And  sec(t): False
sin(t)  And  -sin(t): False
sin(t)  And  -cos(t): False
sin(t)  And  -tan(t): False
sin(t)  And  sin(-t): False
sin(t)  And  cos(-t): False
sin(t)  And  tan(-t): False
sin(t)  And  sin(t)/cos(t): False
sin(t)  And  2sin(t/2)cos(t/2): True
sin(t)  And  sin^2(t): False
sin(t)  And  1-cos^2(t): False
sin(t)  And  1-cos(2t)/2: False
sin(t)  And  1/cos(t): False
cos(t)  And  cos(t): True
cos(t)  And  tan(t): False
cos(t)  And  sec(t): False
cos(t)  And  -sin(t): False
cos(t)  And  -cos(t): False
cos(t)  And  -tan(t): False
cos(t)  And  sin(-t): False
cos(t)  And  cos(-t): True
cos(t)  And  tan(-t): False
cos(t)  And  sin(t)/cos(t): False
cos(t)  And  2sin(t/2)cos(t/2): False
cos(t)  And  sin^2(t): False
cos(t)  And  1-cos^2(t): False
cos(t)  And  1-cos(2t)/2: False
cos(t)  And  1/cos(t): False
tan(t)  And  tan(t): True
tan(t)  And  sec(t): False
tan(t)  And  -sin(t): False
tan(t)  And  -cos(t): False
tan(t)  And  -tan(t): False
tan(t)  And  sin(-t): False
tan(t)  And  cos(-t): False
tan(t)  And  tan(-t): False
tan(t)  And  sin(t)/cos(t): True
tan(t)  And  2sin(t/2)cos(t/2): False
tan(t)  And  sin^2(t): False
tan(t)  And  1-cos^2(t): False
tan(t)  And  1-cos(2t)/2: False
tan(t)  And  1/cos(t): False
sec(t)  And  sec(t): True
sec(t)  And  -sin(t): False
sec(t)  And  -cos(t): False
sec(t)  And  -tan(t): False
sec(t)  And  sin(-t): False
sec(t)  And  cos(-t): False
sec(t)  And  tan(-t): False
sec(t)  And  sin(t)/cos(t): False
sec(t)  And  2sin(t/2)cos(t/2): False
sec(t)  And  sin^2(t): False
```

Figure 2: Trigonometric Identity Equivalence

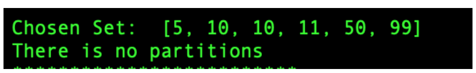
```
sec(t)  And  1-cos^2(t): False
sec(t)  And  1-cos(2t)/2: False
sec(t)  And  1/cos(t): True
-sin(t)  And  -sin(t): True
-sin(t)  And  -cos(t): False
-sin(t)  And  -tan(t): False
-sin(t)  And  sin(-t): True
-sin(t)  And  cos(-t): False
-sin(t)  And  tan(-t): False
-sin(t)  And  sin(t)/cos(t): False
-sin(t)  And  2sin(t/2)cos(t/2): False
-sin(t)  And  sin^2(t): False
-sin(t)  And  1-cos^2(t): False
-sin(t)  And  1-cos(2t)/2: False
-sin(t)  And  1/cos(t): False
-cos(t)  And  -cos(t): True
-cos(t)  And  -tan(t): False
-cos(t)  And  sin(-t): False
-cos(t)  And  cos(-t): False
-cos(t)  And  tan(-t): False
-cos(t)  And  sin(t)/cos(t): False
-cos(t)  And  2sin(t/2)cos(t/2): False
-cos(t)  And  sin^2(t): False
-cos(t)  And  1-cos^2(t): False
-cos(t)  And  1-cos(2t)/2: False
-cos(t)  And  1/cos(t): False
-tan(t)  And  -tan(t): True
-tan(t)  And  sin(-t): False
-tan(t)  And  cos(-t): False
-tan(t)  And  tan(-t): True
-tan(t)  And  sin(t)/cos(t): False
-tan(t)  And  2sin(t/2)cos(t/2): False
-tan(t)  And  sin^2(t): False
-tan(t)  And  1-cos^2(t): False
-tan(t)  And  1-cos(2t)/2: False
-tan(t)  And  1/cos(t): False
sin(-t)  And  sin(-t): True
sin(-t)  And  cos(-t): False
sin(-t)  And  tan(-t): False
sin(-t)  And  sin(t)/cos(t): False
sin(-t)  And  2sin(t/2)cos(t/2): False
sin(-t)  And  sin^2(t): False
sin(-t)  And  1-cos^2(t): False
sin(-t)  And  1-cos(2t)/2: False
sin(-t)  And  1/cos(t): False
cos(-t)  And  cos(-t): True
cos(-t)  And  tan(-t): False
cos(-t)  And  sin(t)/cos(t): False
cos(-t)  And  2sin(t/2)cos(t/2): False
cos(-t)  And  sin^2(t): False
cos(-t)  And  1-cos^2(t): False
cos(-t)  And  1-cos(2t)/2: False
cos(-t)  And  1/cos(t): False
tan(-t)  And  tan(-t): True
tan(-t)  And  sin(t)/cos(t): False
tan(-t)  And  2sin(t/2)cos(t/2): False
```

Figure 3: Trigonometric Identity Equivalence



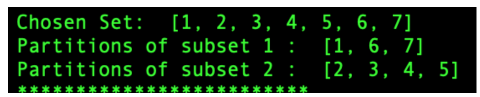
```
tan(-t)  And  sin^2(t): False
tan(-t)  And  1-cos^2(t): False
tan(-t)  And  1-cos(2t)/2: False
tan(-t)  And  1/cos(t): False
sin(t)/cos(t)  And  sin(t)/cos(t): True
sin(t)/cos(t)  And  2sin(t/2)cos(t/2): False
sin(t)/cos(t)  And  sin^2(t): False
sin(t)/cos(t)  And  1-cos^2(t): False
sin(t)/cos(t)  And  1-cos(2t)/2: False
sin(t)/cos(t)  And  1/cos(t): False
2sin(t/2)cos(t/2)  And  2sin(t/2)cos(t/2): True
2sin(t/2)cos(t/2)  And  sin^2(t): False
2sin(t/2)cos(t/2)  And  1-cos^2(t): False
2sin(t/2)cos(t/2)  And  1-cos(2t)/2: False
2sin(t/2)cos(t/2)  And  1/cos(t): False
sin^2(t)  And  sin^2(t): True
sin^2(t)  And  1-cos^2(t): True
sin^2(t)  And  1-cos(2t)/2: True
sin^2(t)  And  1/cos(t): False
1-cos^2(t)  And  1-cos^2(t): True
1-cos^2(t)  And  1-cos(2t)/2: True
1-cos^2(t)  And  1/cos(t): False
1-cos(2t)/2  And  1-cos(2t)/2: True
1-cos(2t)/2  And  1/cos(t): False
1/cos(t)  And  1/cos(t): True
```

Figure 4: Subset Partition



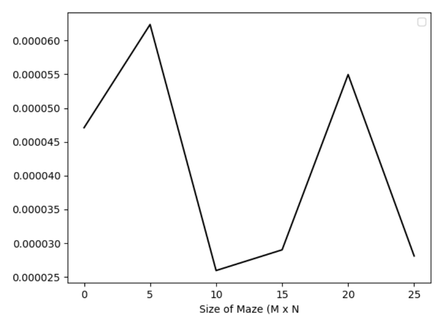
```
Chosen Set:  [5, 10, 10, 11, 50, 99]
There is no partitions
```

Figure 5: Subset Partition



```
Chosen Set:  [1, 2, 3, 4, 5, 6, 7]
Partitions of subset 1 :  [1, 6, 7]
Partitions of subset 2 :  [2, 3, 4, 5]
************************
```

Figure 6: Run Times



IV. Discussion

Overall, implementing the two algorithm design techniques to preform each specific given task resulting in a successful execution of the program. If you notice, in the time runs figure 6 there are low values. These low values occur when the original set of numbers has an odd summation and does not proceed to be passed along the backtracking method. But overall, the average of the time values for the set partition is consistent across various of numerical sets.

V. Source Code

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 05/09/2019
Discreption: Lab 8:
    This program is desinged to act as the main file of this lab. It produces the main menu and recieves
    user input to call funcitons. It also records the time it takes to partition different sets.
"""

from Randomized import TrigCombo
from Backtrack import findPartition
import matplotlib.pyplot as plt
import numpy as np
import timeit


loop = True #commences the loop

#the is the while loop which pompts the user.
while loop:
  print("1. Triginomic Functions\n2. Sum Partitions\n3. Time Trial")
  number = input("4. Exit\n")
  print("************************")
  #trys converting the input into an int. if it fails, the pormpt runs again
  try:
    choice = int(number)
  except:
    choice = -1

  #The fist if statement does the trigonmaic functions
  if choice == 1:
    TrigCombo()

  if choice == 2:   #The second if statement does the does the partion functions
    loop2 = True
    S = []
    while loop2:
      print("Enter a value to insert into the origiona subet :")
      choice = input("Enter 'done' to indicate when finished : \n")

      if choice != 'done':
        #the following converts the input into ints if it's possible
        try :
          choice = int(choice)
          S.append(choice)
        except:
          print("Try Again")
      else :
        loop2 = False
    print("************************")
    S.sort()
    findPartition(S)


  #Choice number two times the preformance of the functions with a determined
  #set of sisze mazes.
  elif choice == 3:
    numSet = [[2,5,8,9,12,21,33], [2, 4, 5, 9, 12], [1,2,3,4,6], [2, 4, 6, 70], [12,13,14,21,25,32], [2,4,6,80]]
    times = []
    for i in range(len(numSet)):
      start = timeit.default_timer() # starts timer
      findPartition(numSet[i])
```

```
            stop = timeit.default_timer() # ends timers
            times.append(stop-start)

        fig, ax = plt.subplots()
        #proceeds to plot the time results
        plt.xlabel('Size of Maze (M x N')
        plt.ylabel('Time (Seconds)')
        x = np.arange(0,30,5)
        plt.plot(x, times, 'k', label= 'BFS')
        plt.legend()
        plt.savefig('RunTimes')
        plt.show()
    #program exits
    elif choice == 4:
        print("Good Bye!")
        loop = False
    else:
        print("Try Again")
    print("************************")
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 05/09/2019
Discreption: Lab 8:
        This program is desinged to to check the validation of any two given grigonomic fucntions through randamization
        desing. The trigonomic funciton is listed and each combination is compared and displayed
"""
import random
import numpy as np
from mpmath import *
import math
import mpmath

def equal(f1, f2,tries=1000,tolerance=0.0001):
    #this if statment repeats for 1000 times for high probability of success
    for i in range(tries):
        t = random.uniform(-math.pi, math.pi)
        y1 = eval(f1)
        y2 = eval(f2)
        if np.abs(y1-y2)>tolerance:
            return False #there is too much differene thus they are not the same
    return True

def TrigCombo():
    trigFucntions = ['sin(t)', 'cos(t)','tan(t)', 'sec(t)', 'sin(t)*-1', 'cos(t)*-1', 'tan(t)*-1', 'sin(t*-1)', 'cos(t*-1)', 'tan(t*-1)', 'sin(t)/cos(t)', '2*sin(t/2)*cos(t/2)', 'sin(t)*sin(t)', '((cos(t)*cos(t)) *-1) +1', '((cos(2*t)*-1)+1)/2', '1/cos(t)']
    displayTrigCombo = ['sin(t)', 'cos(t)', 'tan(t)', 'sec(t)', '  sin  (t)', '  cos  (t)', '  tan  (t)', 'sin(  t  )', 'cos(  t  )', 'tan(  t  )', 'sin(t)/cos(t)', '2sin(t/2)cos(t/2)', 'sin^2(t)', '1  cos  ^2(t)', '1  cos  (2t)/2', '1/cos(t)']

    for i in range(len(trigFucntions)): #nested foor loops does a permutatoin transverse
        for j in range(i,len(trigFucntions)):
            print(displayTrigCombo[i], ' And ', displayTrigCombo[j], end=': ')
            print(equal(trigFucntions[i], trigFucntions[j]))
```

```
"""
Author: Steven J. Robles
Class: CS 2302 Data Structures III
Instructor: Olac Fuentes
TA: Anindita Nath And Maliheh Zargaran
Last Modified: 05/09/2019
Discreption: Lab 8:
    This program is desinged to take set of numbers and partition it into two sets where the sum of both sets
```

```
    equals to the sum of the origional set. Also, the numbers within both new sets are not the same. If there
    is not possible partiions for the requirements, it would tell so.
"""

#the main recursice 'backtracking' function
def partition(S,last, part, part2, currSum, goal, construct2=False):
  if construct2 and last >= 0: #this if statment exits to fill the rest of the items when set 1 is solved
    part2.append(S[last])
    return partition(S,last-1, part, part2, 0, goal, True)

  if currSum == goal: #goal of the sum is met
    partition(S,last, part, part2, 0, goal, True)
    return True

  if goal < currSum  or last < 0: #the current combination is not possible
    return False

  res = partition(S,last-1,part, part2, currSum+S[last], goal) #Takes S[last]

  if res : #set1 is solved and appended
    part.append(S[last])
    return True

  else:
    part2.append(S[last]) #since it was not part of the path, it is passed to set2
    return partition(S,last-1, part, part2, currSum, goal) # kips S[last]


#function called from the main program
def findPartition(numSet2):
  print('Chosen Set: ', numSet2)
  total = 0
  for i in range(len(numSet2)): #gets the total sum first
    total += numSet2[i]
  if total % 2 == 1: #if the sum is odd, then the subests dont exits
    print("There is no partitions")

  else:
    set1, set2 = [], []
    partition(numSet2, len(numSet2)-1, set1, set2, 0,total//2)
    if len(set1)>0: #prints each subset is solvable
      set1.sort()
      set2.sort()
      print('Partitions of subset 1 : ', set1)
      print('Partitions of subset 2 : ', set2)
    else:
      print("There is no partitions")
```

VI. Academic Dishonesty

Scholastic Dishonesty

Any student who commits an act of scholastic dishonesty is subject to discipline. Scholastic dishonesty includes, but not limited to cheating, plagiarism, collusion, the submission for credit of any work or materials that are attributable to another person.

- **Cheating**
  - Copying form the test paper of another student
  - Communicating with another student during a test
  - Giving or seeking aid from another student during a test
  - Possession and/or use of unauthorized materials during tests (i.e. Crib notes, class notes, books, etc)
  - Substituting for another person to take a test
  - Falsifying research data, reports, academic work offered for credit
- **Plagiarism**
  - Using someone's work in your assignments without the proper citations
  - Submitting the same paper or assignment from a different course, without direct permission of instructors
- **Collusion**
  - Unauthorized collaboration with another person in preparing academic assignments

Sign: ______________________________ **Date:** __05/10/2019____