Implementation of Combinatorial Algorithms using Optimization Techniques

Youssef Bassil
Researcher, LACSC – Lebanese Association for Computational Sciences Registered, Beirut, Lebanon

ABSTRACT
In theoretical computer science, combinatorial optimization problems are about finding an optimal item from a finite set of objects. Combinatorial optimization is the process of searching for maxima or minima of an unbiased function whose domain is a discrete and large configuration space. It often involves determining the way to efficiently allocate resources used to find solutions to mathematical problems. Applications for combinatorial optimization include determining the optimal way to deliver packages in logistics applications, determining taxis best route to reach a destination address, and determining the best allocation of jobs to people. Some common problems involving combinatorial optimizations are the Knapsack problem, the Job Assignment problem, and the Travelling Salesman problem. This paper proposes three new optimized algorithms for solving three combinatorial optimization problems namely the Knapsack problem, the Job Assignment problem, and the Traveling Salesman respectively. The Knapsack problem is about finding the most valuable subset of items that fit into the knapsack. The Job Assignment problem is about assigning a person to a job with the lowest total cost possible. The Traveling Salesman problem is about finding the shortest tour to a destination city through travelling a given set of cities. Each problem is to be tackled separately. First, the design is proposed, then the pseudo code is created along with analyzing its time complexity. Finally, the algorithm is implemented using a high-level programming language. As future work, the proposed algorithms are to be parallelized so that they can execute on multiprocessing environments making their execution time faster and more scalable.

KEYWORDS: Combinatorial Algorithms, Optimization Techniques, Knapsack, Job Assignment, Traveling Salesman

I. KNAPSACK PROBLEM
The knapsack problem is a problem in combinatorial optimization [1]. Given \(n\) items of weights \(w_1, w_2, \ldots, w_n\) and values \(v_1, v_2, \ldots, v_n\) and a knapsack (container) of capacity \(W\). The problem is to find the most valuable subset of items that fit into the knapsack [2].

A. Proposed Solution
The algorithm is based on exhaustive search approach which suggests generating every combinational object of the problem and performing the appropriate calculations. The algorithm use three one-dimensional arrays, one to store the item weights, another one to store the item values, and a last one to store the generated subsets.

B. Design
Figure 1 shows the process flow diagram of the Knapsack problem design

C. Algorithm
//ALGORITHM Knapsack (itemsValue[n], itemsWeight[n])
// Knapsack Problem
// INPUT: itemsValue[n], itemsWeight[n]
// OUTPUT: optimalSubset: array of integers
ITEMS_COUNT: integer constant that holds the # of items
itemsValue[n]: array of integers that holds item Values
itemsWeight[n]: array of integers that holds item Weights

Figure 1: Process Flow for the Knapsack problem

@IJTSRD | Unique Paper ID – IJTSRD22925 | Volume – 3 | Issue – 3 | Mar-Apr 2019 | Page: 660
BEGIN
optimalValue ← 0

// Step1: Generates integer numbers
FOR i ← 0 TO Pow(2, ITEMS_COUNT) DO
{
   // Step2: Convert integer Numbers to binary numbers
   // Step3: Generating Subsets
   j ← 0
   WHILE i<>0
   {
      bitString[j] ← i MOD 2
      i ← i/2
   }
   // Step4: Calculate the Item values corresponding to each subset
   sumValues<-0
   sumWeights<-0
   FOR k ← 0 TO ITEMS_COUNT DO
   {
      // Replaces TRUE flag with its corresponding Item value
      IF bitString[k] = TRUE THEN
      {
         sumValues <- sumValues + itemsValue[k]
         sumWeights <- sumWeights + itemsWeight[k]
      }
   }
   k ← k+1

   // Step5: Store the highest value with its corresponding subset
   IF (sumWeights <= knapsackCapacity AND sumValues > optimalValue)
   THEN
   {
      optimalValue ← sumValues
      FOR p←0 TO ITEMS_COUNT DO
      {
         optimalSubset[p] ← bitString[p]
         p ← p+1
      }
   }
   i ← i+1
} // end of step1 FOR LOOP
// Step6: Return the Subset that has highest Items value
RETURN optimalSubset
END

D. Analysis
The proposed algorithm can find the optimal subset of items with their corresponding optimal value while falling under the below efficiency class:
Knapsack (a[n],b[n]) € \( O(n^2) \quad (n^2 > n) \)
Knapsack (a[n],b[n]) € \( \Omega(1) \quad (1 < n) \)
Knapsack (a[n],b[n]) € \( \Omega(n) \quad (n = n) \)

Performance wise, it requires about 9 milliseconds to handle the problem with 50 items.

E. Implementation
Figure 2 depicts the screenshot of the program that implements the Knapsack problem using C#.NET [3].

II. JOB ASSIGNMENT PROBLEM
The assignment problem is a fundamental combinatorial optimization problem [4]. Given \( n \) people who need to be assigned to \( n \) jobs, one person per job. The cost of \( ith \) person is assigned to \( jth \) job is stored in table[i][j]. The problem is to find an assignment with the lowest total cost [5].

A. Proposed Solution
Developing an algorithms based on the brute force technique which tests and evaluates all possible objects combinations involved in the problem and performs appropriate calculations. The algorithm uses a one-dimentional array to store permutations and a two-dimentional array to store Person/Job cost

B. Design
Figure 3 shows the process flow diagram of the Job Assignment problem design

C. Algorithm
// ALGORITHM Assignment (table[n][n], COUNTER)
// Person/Job Assignment Problem
// INPUT: table[n][n], COUNTER
/// OUTPUT: optimalList : array of integers
// table[n][n]: 2D integer array that Stores all costs entered by the user
COUNTER: integer that holds the # of persons(or the # of jobs)
list[COUNTER]: array of integers that holds permutation
pointers[COUNTER]: array of integers that holds present direction of each permutation
increasingPtr[COUNTER]: array of integers that holds left to right arrows -> -> -> ...
decreasingPtr[COUNTER]: array of integers that holds right to left arrows <-> <-> <-> ...
optimalSum: integer that holds the lower cost per person/job assignment
optimalList [COUNTER]: array of integers that holds the permutation with the lower cost
mobile: integer that holds the mobile element
mobileIndex: integer that holds the index of the mobile element
flag: boolean variable that indicates if a mobile exists or not
temp: integer used FOR swapping purposes
sum: integer that holds the cost of a particular permutation instance
BEGIN
optimalSum ← INFINITY

//Fill array lists with 1 2 3 4 5 6 ...(depending on variable COUNTER)
FOR i ← 0 TO COUNTER DO
{
    list[i] ← i+1
    i ← i+1
}

//Initialize pointers <-> <-> <-> ...
FOR i ← COUNTER-1 TO 0 DO
{
    pointers[i] ← i-1
    i ← i+1
}

//Initialize increasingPtr -> -> -> ...
FOR i ← 0 TO COUNTER DO
{
    increasingPtr[i] ← i+1
    i ← i+1
}

//Initialize decreasingPtr <-> <-> <-> ...
FOR i ← COUNTER-1 TO 0 DO
{
    decreasingPtr[i] ← i-1
    i ← i+1
}

// Johnson-Trotter ALGORITHM
// Generates Permutations
FOR i ← 0 TO fac(COUNTER)-1 DO
{
    //Calculate the cost for each permutation instance
    sum ← 0
    FOR j ← 0 TO COUNTER DO
    {
        sum ← sum+table[j,list[j]-1]
        j ← j+1
    }
    // Holds the lowest sum
    IF sum < optimalSum THEN
    {
        optimalSum ← sum
        FOR k ← 0 TO COUNTER DO
        {
            optimalList[k] ← list[k]
            k ← k+1
        }
    }
    mobile ← 0
    mobileIndex ← 0
    flag ← false

    // Step1 : Find the largest Mobile
    FOR i ← 0 TO COUNTER DO
    {
        IF(pointers[i]<>1 && pointers[i]<>COUNTER AND list[i]>mobile AND list[pointers[i]]<list[i])
        THEN
        {
            mobile ← list[i]
            mobileIndex ← i
            flag ← TRUE
        }
    }
    IF flag=TRUE THEN
    {
        // Swap the mobile with the element that it points to
        list[mobileIndex] ← list[pointers[mobileIndex]]
        list[pointers[mobileIndex]] ← mobile
        IF(pointers[pointers[mobileIndex]]=mobileIndex) THEN
        {
            // Indicates the mobile is at the left side
            IF(pointers[mobileIndex] > mobileIndex) THEN
            {
                // Swap the pointers of mobile and the element that it points to
                Temp ← pointers[pointers[mobileIndex]]
                pointers[pointers[mobileIndex]] ← pointers[mobileIndex]+1
                pointers[mobileIndex] ← temp-1
            }
        ELSE // Indicates the mobile is at the right side
        {
            // Swap the pointers of mobile and the element that it points to
        }
Temp ← pointers[pointers[mobileIndex]]
pointers[pointers[mobileIndex]] ← pointers[mobileIndex]-1
pointers[mobileIndex] ← temp+1
}
}

// Reverse Directions
FOR i ← 0 TO COUNTER DO
{
  IF list[i] > mobile THEN
    IF pointers[i] ← increasingPtr[i] THEN
      pointers[i] ← decreasingPtr[i]
    ELSE IF pointers[i] ← decreasingPtr[i] THEN
      pointers[i] ← increasingPtr[i]
    i ← i+1
}

//Calculate the cost FOR the last permutation instance
sum ← 0
FOR j ← 0 TO COUNTER DO
{
  sum ← sum + table[j, list[j]-1]
  j ← j+1
}

// Holds the lowest sum
IF sum < optimalSum THEN
{
  optimalSum ← sum
  FOR k ← 0 TO COUNTER DO
  {
    optimalList[k] ← list[k]
    k ← k+1
  }
}
// optimal list should hold the less costly person/job assignment
RETURN optimalList
END

D. Analysis
The proposed algorithm can find the optimal person/job assignment with its corresponding lowest cost. It is very practical even on large number of persons, however it exhausts processing time due to Johnson-trotter algorithm [6] whose order of growth is always exponential. The algorithm falls under the below efficiency class:

Assignment (table[n][n], c) ∈ Θ n^3 (n^3 > n^2)
Assignment (table[n][n], c) ∈ Ω n (n < n^2)
Assignment (table[n][n], c) ∈ Φ n^2 (n^2 = n^2)

Performance wise, it requires 12 seconds to handle a problem with 100 jobs 100! = 9.33262154439441 52681699238856267e+157 permutations

E. Implementation
Figure 4 depicts the screenshot of the program that implements the Job Assignment problem using C#.NET.

III. TRAVELING SALESMAN PROBLEM
The Traveling Salesman Problem is a classic algorithmic problem in the field of computer science that focuses on optimization [7]. The problem ask to find the shortest tour through a given set of n cities or nodes that visits each city exactly once before returning to the city where it started [8].

A. Proposed Solution
Exaustive search technique is so far the most appropriate approach to solve this problem. It consists of generating all possible paths with their corresponding lengths so eventually the shortest path can be identified. The algorithm uses a one-dimensional array to store permutations, a one-dimensional array to store distinct cities, and a two-dimensional array to store from city, to city, and length variables.

B. Design
Figure 5 shows the process flow diagram for the Traveling Salesman problem design

C. Algorithm
// ALGORITHM Salesman(table[n][3], startCity)
// Person/Job Assignment Problem
// INPUT: table[n][n], startCity
// OUTPUT: optimalList : array of characters
cities[citiesCounter]: array of characters holds Distinct cities
newList[citiesCounter+1]: array of characters that holds: startcity+permutation+startcity
citiesCounter: integer holds # of distinct cities
startCity: Character holds the name of the start city table[n][3]: 2D integer array that Stores all routes with their corresponding length
list[citiesCounter-1]: array of characters that holds permutation
pointers[citiesCounter-1]: array of integers that holds present direction of each permutation
increasingPtr[citiesCounter-1]: array of integers that holds left to right arrows -> -> ->
decreasingPtr[citiesCounter-1]: array of integers that holds right to left arrows <- <- <-
optimalSum: integer that holds the shortest path summation
optimalList[citiesCounter+1]: array of characters that holds the permutation with the shortest path
mobile: integer that holds the mobile element
mobileIndex: integer that holds the index of the mobile element
flag: boolean variable that indicates if a mobile exists or not
temp: integer used for swapping purposes
sum: integer that holds the cost of a particular permutation instance
BEGIN
//Step1: Recognize and store in array cities only the distinct cities
i←0
WHILE(i<citiesCounter) DO
{    IF table[i][1]<>cities[i] THEN
        i←i+1
    ELSE
        {    i ← citiesCounter+1
             s ← i
        }
    }
// Adding the found city to the array
IF i=citiesCounter THEN
{    cities[citiesCounter] ← table[s][1]
    citiesCounter ← citiesCounter+1
 }
//Step2: create an array named list that contains all distinct cities
k←0
FOR i←0 TO citiesCounter DO
{    IF cities[i] ← startCity THEN
        {    list[k] ← cities[i]
             k ← k+1
        }
     i ← i+1
}
//Initialize pointers <- <- <- ...
FOR i ← citiesCounter-1 TO 0 DO
{    pointers[i] ← i-1
     i ← i+1
}
//Initialize increasingPtr -> -> -> ...:
FOR i←0 TO citiesCounter DO
{    increasingPtr[i] ← i+1
     i ← i+1
}
BEGIN
//Step1: Recognize and store in array cities only the distinct cities
i←0
WHILE(i<citiesCounter) DO
{    IF table[i][1]<>cities[i] THEN
        i←i+1
    ELSE
        {    i ← citiesCounter+1
             s ← i
        }
    }
// Adding the found city to the array
IF i=citiesCounter THEN
{    cities[citiesCounter] ← table[s][1]
    citiesCounter ← citiesCounter+1
 }
//Step2: create an array named list that contains all distinct cities
k←0
FOR i←0 TO citiesCounter DO
{    IF cities[i] ← startCity THEN
        {    list[k] ← cities[i]
             k ← k+1
        }
     i ← i+1
}
//Initialize pointers <- <- <- ...
FOR i ← citiesCounter-1 TO 0 DO
{    pointers[i] ← i-1
     i ← i+1
}
//Initialize increasingPtr -> -> -> ...
FOR i←0 TO citiesCounter DO
{    increasingPtr[i] ← i+1
     i ← i+1
}
flag ← true
}
i ← i + 1
// Step2: test whether a mobile was found
// Step3: Swap the mobile with the element that it points to
// Step4: Swap the pointers of mobile and the element that it points to
// Step5: Reverse Directions of all elements that are greater than mobile
IF flag=TRUE THEN
{
    // Swap the mobile with the element that it points to
    list[mobileIndex] ← list[pointers[mobileIndex]]
    list[pointers[mobileIndex]] ← mobile

    IF pointers[pointers[mobileIndex]]=mobileIndex THEN
    {
        // Indicates the mobile is at the left side
        IF(pointers[mobileIndex] > mobileIndex) THEN
        {
            // Swap the pointers of mobile and the element that it points to
            Temp ← pointers[pointers[mobileIndex]]
            pointers[pointers[mobileIndex]] ← pointers[mobileIndex]+1
            pointers[mobileIndex] ← temp-1
        }
    }
    ELSE // Indicates the mobile is at the right side
    {
        // Swap the pointers of mobile and the element that it points to
        Temp ← pointers[pointers[mobileIndex]]
        pointers[pointers[mobileIndex]] ← pointers[mobileIndex]-1
        pointers[mobileIndex] ← temp+1
    }
}
// Reverse Directions
FOR i ← 0 TO citiesCounter DO
{
    IF list[i]=mobile THEN
        IF pointers[i]=increasingPtr[i] THEN
            pointers[i] ← decreasingPtr[i]
        ELSE IF pointers[i]=decreasingPtr[i] THEN
            pointers[i] ← increasingPtr[i]

        i ← i + 1
    }
}
RETURN optimalList
END

D. Analysis
The proposed algorithm can find the shortest path among many alternatives starting from a given city, passing through all the available cities only once to end at the same starting point. Even though it is based on Johnson-Trotter algorithm to generate permutations, the proposed algorithm is considered quite efficient due to the complexity of the original problem. Therefore to solve a complex problem such the traveling salesman problem, somehow you are going to lose some processing time. The algorithm falls under the below efficiency class:

Salesman (table[n][3] , sCity) € O n^3 (n^3 > n^2)
Salesman (table[n][3] , sCity) € Ω n (n < n^2)
Salesman (table[n][3] , sCity) € Θ n^2 (n^2 = n^2)

Performance wise, it requires 17 seconds for a problem with 100 cities (100! = 9.3326215443944152681699238856267e+157 permutations)

E. Implementation
Figure 6 depicts the screenshot of the program that implements the Traveling Salesman problem using C#.NET.

IV. Conclusions & Future Work
This paper proposed three new optimized algorithms for solving three combinatorial optimization problems namely the Knapsack problem, the Job Assignment problem, and the Traveling Salesman problem respectively. Each problem was tackled from a design, analysis, and implementation point of views. The proposed designs showed the optimized versions of the algorithms while listing their complete pseudo code. Furthermore, a thorough time complexity analysis was performed to finally end up implementing the algorithms and testing them using C#.NET.

As future work, the proposed algorithms are to be parallelized using multithreading and multiprogramming techniques so as to speeding up their execution time and making them more adaptable to large computing architectures.

Acknowledgment
This research was funded by the Lebanese Association for Computational Sciences (LACSC),
Beirut, Lebanon, under the "Parallel Programming Algorithms Research Project – PPARP2019".

References
[1] Caccetta, L., Kulanoot, A, "Computational Aspects of Hard Knapsack Problems". Nonlinear Analysis, 47 (8): 5547–5558, 2001
[2] Poirriez, Vincent; Yanev, Nicola; Andonov, Rumen, "A hybrid algorithm for the unbounded knapsack problem", Discrete Optimization, 6 (1): 110–124, 2009
[3] Petzold, Charles, "Programming Microsoft Windows with C#", Microsoft Press. ISBN 0-7356-1370-2, 2002
[4] Munkres, James, "Algorithms for the Assignment and Transportation Problems", Journal of the Society for Industrial and Applied Mathematics, 5 (1): 32–38, 1957
[5] Brualdi, Richard A., "Combinatorial matrix classes. Encyclopedia of Mathematics and Its Applications", Cambridge: Cambridge University Press, ISBN 978-0-521-86565-4, 2006
[6] Dershowitz, Nachum, "A simplified loop-free algorithm for generating permutations", Nordisk Tidskr Informations, 15 (2): 158–164, 1957
[7] Cook, William, "In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation", Princeton University Press, ISBN 9780691152707, 2012
[8] Steinerberger, Stefan, "New Bounds for the Traveling Salesman Constant", Advances in Applied Probability, (47): 27–36, 2015.