Instituto Politécnico Nacional

ESC Superior de Cómputo

"EVOLUTIONARY COMPUTING" 3-KNAPSACK, COIN AND THREE JUGS PROBLEM

Jorge Luis Rosas Trigueros

Saldaña Aguilar Gabriela

20/09/16 - 30/09/16

THEORETICAL FRAMEWORK

Evolutionary algorithms have been shown to be successful for a wide range of optimization problems. While these randomized search heuristics work well for many optimization problems in practice, a satisfying and rigorous mathematical understanding of their performance is an important challenge in the area of genetic and evolutionary computing. It has been proven for various combinatorial optimization problems that they can be solved by evolutionary algorithms in reasonable time using a suitable representation together with mutation operators adjusted to the given problem. The representations used in these papers are different from the general encodings working with binary strings as considered earlier in theoretical works on the runtime behavior of evolutionary algorithms. the chosen representations reflect some properties of partial solutions of the problem at hand that allow to obtain solutions that can be extended to optimal ones for the considered problem.

Dynamic programming is a well-known algorithmic technique that helps to tackle a wide range of problems. A general framework for dynamic programming has been considered by e. g. Woeginger and Kogan . The technique allows the design of efficient algorithms, that solve the problem at hand to optimality, by extending partial solutions to optimal ones.

Framework for Evolutionary Algorithms An evolutionary algorithm consists of different generic modules, which have to be made precise by the user to best fit to the problem. Experimental practice, but also some theoretical work, demonstrate that the right choice of representation, variation operators, and selection method is crucial for the success of such algorithms. We assume that the problem to be solved is given by a multi-objective function g that has to be optimized. We consider simple evolutionary algorithms that consist of the following components. The algorithm (see Algorithm 2) starts with an initial population P0.

During the optimization the evolutionary algorithm uses a selection operator sel(·) and a mutation operator mut(·) to create new individuals. The d-dimensional fitness function together with a partial order par on R d induce a partial order dom on the phenotype space, which guides the search. After the termination of the EA, an output function out(·) is utilized to map the individuals in the last population to search points from the original search space.

The first thing we need to ask ourselves is ¿What I´m going to evaluate with my genetic algorithm?

R= We need to evaluate how good is a solution for the given problem, it means that we will have a bunch of solutions and we want the better one. In short Chromosome=Possible solution.

KNAPSACK PROBLEM USING GENETIC ALGORITHMS

We are given a total knapsack capacity; we want to push the items with the best benefit we have.

*Chromosome Encoding: A binary array encoding, where:

1-> means that the element at the given position of the array was pushed.

0->means that the element at the given position of the array was not pushed.

*Population: We generate a bunch of chromosomes

*Fitness Function: We evaluate how good is this chromosome with the below parameters

-if it fits into the knapsack the it is a good solution; we return the benefit of pushing this element as fitness value.

-if it doesn´t fits into the knapsack it is a bad solution, but we need to measure how bad it is, so we subtract the total chromosome´s benefit to the fitness value finally we subtract to the fitness value remaining the weight of the given element.

*Selection: We use the roulette wheel algorithm and elitism to select the best candidates for reproduction.

*Crossover: it is do it using half parts of the parents chromosomes.

*Mutation: We interchange bits (0 to 1, 1 to 0) of the resultant chromosome with a given probability.

COIN CHANGE USING GENETIC ALGORITHMS

We are given a total amount of money; we want to get the minimum number of coins needed in order to change that amount of money. We have given a set of never ending count of denominations.

*Chromosome Encoding: A binary array encoding, where:

1-> means that the denomination at the actual position of the array was taken.

0-> means that the denomination at the actual position of the array was not taken.

*Population: We generate a bunch of chromosomes

*Fitness Function: First of all, we generate a random number of coins for each denomination taken in the chromosome then we evaluate how good is this solution with the below parameters

-if the amount of money the chromosome generates is closer to the real value we want to have it is a good solution.

-if the amount of money is not closer enough or it is bigger than the real one then it is not a good solution, we checked that by subtracting the amount we want to have change to the Total amount of accumulated money of the chromosome and keeping the absolute value of this as fitness value lastly we add to the fitness value the total amount of coins needed to accomplish that solution.

*Selection: We use the roulette wheel algorithm and elitism to select the best candidates for reproduction.

*Crossover: it is done by using half parts of the parents chromosomes.

*Mutation: We interchange bits (0 to 1, 1 to 0) of the resultant chromosome with a given probability.

EQUIPMENT AND MATERIAL.

SOFTWARE: The program is made in Python

BODY

We were told to write down the problems mentioned above.

Measures using bi={10,8,5,4,3} wi={9,5,4,3,2} W=20 (see e. g. [1, 2]).

Mutation probability	Generations to find the optimal: 50	100	200
0.1	f([0, 2, 3, 4])= 22 Current value: 22 Current weight: 18	f([0, 1, 2])= 23 Current value: 23 Current weight: 18	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20
0.25	f([0, 1, 2])= 23 Current value: 23 Current weight: 18	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20
0.5	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20
0.75	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20	f([0, 1, 2, 4])= 26 Current value: 26 Current weight: 20

F1:KP

Measures using Amount of money to be change= 100 denominations=[33,20,80]

Mutation probability \ Generations to find the optimal	50	100	200
0.1	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 4 Change ['3 of $33', '0 of $20', '0 of $80'] Total amount: $ 99 Total Coins: 3
0.25	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2
0.5	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2
0.75	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2	Fitness value : 2 Change ['0 of $33', '1 of $20', '1 of $80'] Total amount: $ 100 Total Coins: 2

F2:COIN

CODE

KP

```
#** GETTING WHAT ELEMENTS WILL BE ADDED TO DE KNAPSACK
def decode_chromosome(chromosome):
    global L_chromosome
    value = []
    for p in range(L_chromosome):
        if (chromosome[p] == 1):
            value.append(p)
    return value
#**FITNESS FUNCTION
#if the element fits in theknapsack we return the benefit
def f(lis):
    global KP_capacity, Elements
    fitness_val = CalcBenefit(lis)
    Wval = CalcWeight(lis)
    penalty = 0
    #print lis
    #print "debag: ", Wval, " / ", fitness_val
#if the total weight is bigger than the capacity
    if (Wval > KP_capacity):
        for i in range(Number_Elements):
            penalty += Elements[i][0]#total benefit
        fitness_val -= penalty
        penalty = 0
        penalty += Wval - KP_capacity
        fitness_val -= penalty
    return fitness_val
```

COIN

```
def decode_chromosome(chromosome):

    #print "decodificando chromosoma"
    #print chromosome
    global L_chromosome

    combination=[]

    Acc=0
    TC=0
    coin=0
    tam=0
    k=0
    for i in range(0,len(Elements)):
        tam=(int)(math.ceil((math.log(x,2))))+1
        #print chromosome
        m=0
        coin=0
        for j in range(tam-1+k,-1+k,-1):
            #print "j:",j,chromosome[j]
            coin+= chromosome[j]*(2**m)
            m+=1
            #print "m:",m

        combination.append(str(coin) + " of $" + str(Elements[i]))
        k+=tam
        Acc+=coin*Elements[i]
        TC+=coin

    #print "The amount of money is: ",Acc
    #print "with: ",TC, "coins"

    return Acc,TC,combination

#**PUNISHMENT, FITNESS FUNCTION
def f(Acc,totalCoins):

    global Max_Weight
    f_ch =abs(Acc-Max_Weight)*alfa
    f_ch +=totalCoins*beta

    return f_ch
```

CONCLUSION

I liked it the way we took the class in the lab like if it was free style, but with the help of the professor, also it is good to explain what will be the next topic to cover in the lab so the students can find it out and investigate a little bit before entering the class because it could be confusing if it is our first time coding evolutionary algorithms.

BIBLIOGRAPHY

Evolutionary Computing class

http://www.cut-the-knot.org/ctk/Water.shtml