Solving 0-1 knapsack problem using Genetic Algorithms

Genetic Algorithms (GAs)

Genetic Algorithms are computer algorithms that search for good solutions to a problem from among a large number of possible solutions. They were proposed and developed in the 1960s by John Holland, his students, and his colleagues at the University of Michigan. These computational paradigms were inspired by the mechanics of natural evolution, including survival of the fittest, reproduction, and mutation. These mechanics are well suited to resolve a variety of practical problems, including computational problems, in many fields. Some applications of GAs are optimization, automatic programming, machine learning, economics, immune systems, population genetic, and social system.

Basic idea behind GAs

GAs begin with a set of candidate solutions (chromosomes) called population. A new population is created from solutions of an old population in hope of getting a better population. Solutions which are chosen to form new solutions (offspring) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied.

Basic elements of GAs

Most GAs methods are based on the following elements, populations of chromosomes, selection according to fitness, crossover to produce new offspring, and random mutation of new offspring.

Chromosomes

The chromosomes in GAs represent the space of candidate solutions. Possible chromosomes encodings are binary, permutation, value, and tree encodings. For the Knapsack problem, we use binary encoding, where every chromosome is a string of bits, 0 or 1.

Fitness function

GAs require a fitness function which allocates a score to each chromosome in the current population. Thus, it can calculate how well the solutions are coded and how well they solve the problem.

Selection

The selection process is based on fitness. Chromosomes that are evaluated with higher values (fitter) will most likely be selected to reproduce, whereas, those with low values will be discarded. The fittest chromosomes may be
selected several times, however, the number of chromosomes selected to reproduce is equal to the population size, therefore, keeping the size constant for every generation. This phase has an element of randomness just like the survival of organisms in nature. The most used selection methods, are roulette-wheel, rank selection, steady-state selection, and some others. Moreover, to increase the performance of GAs, the selection methods are enhanced by elitism. Elitism is a method, which first copies a few of the top scored chromosomes to the new population and then continues generating the rest of the population. Thus, it prevents loosing the few best found solutions.

**Crossover**

Crossover is the process of combining the bits of one chromosome with those of another. This is to create an offspring for the next generation that inherits traits of both parents. Crossover randomly chooses a locus and exchanges the subsequences before and after that locus between two chromosomes to create two offspring. For example, consider the following parents and a crossover point at position 3:

Parent 1 1 0 0 | 0 1 1 1
Parent 2 1 1 1 | 1 0 0 0
Offspring 1 1 0 0 1 0 0 0
Offspring 2 1 1 1 0 1 1 1

In this example, Offspring 1 inherits bits in position 1, 2, and 3 from the left side of the crossover point from Parent 1 and the rest from the right side of the crossover point from Parent 2. Similarly, Offspring 2 inherits bits in position 1, 2, and 3 from the left side of Parent 2 and the rest from the right side of Parent 1.

**Mutation**

Mutation is performed after crossover to prevent falling all solutions in the population into a local optimum of solved problem. Mutation changes the new offspring by flipping bits from 1 to 0 or from 0 to 1. Mutation can occur at each bit position in the string with some probability, usually very small (e.g. 0.001). For example, consider the following chromosome with mutation point at position 2:

Not mutated chromosome: 1 0 0 0 1 1 1
Mutated: 1 1 0 0 1 1 1
The 0 at position 2 flips to 1 after mutation.

Outline of basic GA s
1. Start: Randomly generate a population of N chromosomes.
2. Fitness: Calculate the fitness of all chromosomes.
3. Create a new population:
   a. Selection: According to the selection method select 2 chromosomes from the population.
   b. Crossover: Perform crossover on the 2 chromosomes selected.
   c. Mutation: Perform mutation on the chromosomes obtained.
4. Replace: Replace the current population with the new population.
5. Test: Test whether the end condition is satisfied. If so, stop. If not, return the best solution in current population and go to Step 2.
   Each iteration of this process is called generation.

1.2 The Knapsack Problem (KP)
The goal of knapsack problem is to Maximize
\[ N \sum_{i=1}^{N} B_i X_i \]
Subject to the constraints
\[ N \sum_{i=1}^{N} V_i X_i \leq V \]
And
\[ 0 \leq X_i \leq Q_i . \]
If one or more of the \( Q_i \) is infinite, the KP is unbounded; otherwise, the KP is bounded. The bounded KP can be either 0-1 KP or Multiconstraint KP. If \( Q_i = 1 \) for \( i = 1, 2, \ldots, N \), the problem is a 0-1 knapsack problem. In the current paper, we have worked on the bounded 0-1 KP, where we cannot have more than one copy of an item in the knapsack.
How do we select an item to be put into the knapsack? There are several possibilities:

**Greedy by Profit**

At each step select from the remaining items the one with the highest profit (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing the most profitable items first.

**Greedy by Weight**

At each step select from the remaining items the one with the least weight (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by putting as many items into the knapsack as possible.

**Greedy by Profit Density**

At each step select from the remaining items the one with the largest profit density, \( \frac{p_i}{w_i} \) (provided the capacity of the knapsack is not exceeded). This approach tries to maximize the profit by choosing items with the largest profit per unit of weight.

While all three approaches generate feasible solutions, we cannot guarantee that any of them will always generate the optimal solution. In fact, it is even possible that none of them does.

**Example of a 0-1 KP**

Suppose we have a knapsack that has a capacity of 13 cubic inches and several items of different sizes and different benefits. We want to include in the knapsack only those items that will have the greatest total benefit within the constraint of the knapsack's capacity. There are three potential items (labeled 'A,' 'B,' 'C'). Their volumes and benefits are as follows:

<table>
<thead>
<tr>
<th>Item #</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Benefit</td>
<td>4</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Volume</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

We seek to maximize the total benefit:

\[
\sum_1^3 B_i X_i = 4X_1 + 3X_2 + 5X_3
\]

Subject to the constraints:
Solving 0-1 knapsack problem using Genetic Algorithms

\[ \sum_{i=1}^{3} V_i X_i = 6X_1 + 7X_2 + 8X_3 \leq 13 \]

And

\[ X_i \in \{0, 1\}, \ for \ i = 1, 2, \ldots, n. \]

For this problem there are 2^3 possible subsets of items:

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Volume of the set</th>
<th>Benefit of the set</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>14</td>
<td>-</td>
</tr>
<tr>
<td><strong>1</strong></td>
<td><strong>1</strong></td>
<td><strong>0</strong></td>
<td><strong>13</strong></td>
<td><strong>7</strong></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

In order to find the best solution we have to identify a subset that meets the constraint and has the maximum total benefit. In our case, only rows given in italics satisfy the constraint. Hence, the optimal benefit for the given constraint \( V = 13 \) can only be obtained with one quantity of A, one quantity of B, and zero quantity of C, and it is 7.

**NP problems and the 0-1 KP**

NP (non-deterministic polynomial) problems are ones for which there are no known algorithms that would guarantee to run in a polynomial time. However, it is possible to “guess” a solution and check it, both in polynomial time. Some of the most well-known NP problems are the traveling salesman, Hamilton circuit, bin packing, knapsack, and clique.

GAs have shown to be well suited for high-quality solutions to larger NP problems and currently they are the most efficient ways for finding an approximately optimal solution for optimization problems. They do not involve extensive search algorithms and do not try to find the best solution, but they simply generate a candidate for a solution, check in polynomial time whether it is a solution or not and how good a solution it is. GAs do not always give the optimal solution, but a solution that is close enough to the optimal one.
Solving 0-1 knapsack problem using Genetic Algorithms

3. SYSTEM DESIGN AND ANALYSIS

3.1 BASIC GENETIC ALGORITHM

Genetic algorithms begin with a set of candidate solutions (chromosomes) called population. A new population is created from an old population in hope of getting better population. Solutions which are chosen to form new solutions (offspring) are selected according to their fitness. The more suitable the solutions are the bigger chances they have to reproduce. This process is repeated until some condition is satisfied.

BEGIN
    Generate initial population;
    Compute fitness of each individual;
    REPEAT /* New generation */
        FOR population_size / 2 DO
            Select two parents from old generation;
            /* biased to the fitter ones */
            Recombine parents for two offspring;
            Compute fitness of offspring;
            Insert offspring in new generation
        END FOR
    UNTIL population has converged
END

3.2 FLOW CHART OF THE MAIN PROGRAM
Solving 0-1 knapsack problem using Genetic Algorithms

START

Initialize array items reading from a file - the data (volume and benefit) for each item.

Initialize the first population by randomly generating a population of size chromosomes

Calculate the fitness and volume of all chromosomes

Check what percentage of the chromosomes in the population has the same fitness value

Does 90% of them have the same fitness value?

yes

Randomly select 2 chromosomes from the population

Perform crossover on the 2 chromosomes selected

Perform mutation on the chromosomes obtained

Does 90% have the same fit value? & & Is the number of generations greater than the limit?

no

no

STOP
Solving 0-1 knapsack problem using Genetic Algorithms

**Representation of the items**

In this knapsack problem we used 1,000 items whose weights and values are stored in a data file. We accessed these data files by implementing a sequence container called `vector`. The capacity of the knapsack is limited to 50 with weights and values of items between 1 and 30.

**Fitness function design**

We calculate the fitness of each chromosome by summing up the benefits of the items that are included in the knapsack, while making sure that the capacity of the knapsack is not exceeded.

Here is a flowchart of the fitness function algorithm:

---

**Selection**

We implemented this selection method in order to increase the probability of choosing fitter chromosomes to reproduce more often than chromosomes with lower fitness values. Since we could not find any selection method like this one in the literature, we decided to call it group selection.

Here is a flowchart of the group selection algorithm:
Crossover

One point crossover
Randomly one position in the chromosomes is chosen.

Two point crossover
Randomly two positions in the chromosomes are chosen.

Arithmetic crossover
Some arithmetic operation is performed to make a new offspring.

Mutation
In the reparation process the child first gets checked if it contains some genetic information too much or is missing some.
4. SYSTEM REQUIREMENTS

4.1 HARDWARE REQUIREMENTS

Processor - Intel Pentium 4 or higher
Hard disk - 20 GB or higher
RAM - 512 MB or higher
Input and Output - Standard

4.2 SOFTWARE REQUIREMENTS

Operating System - Linux
Software - C++
Compiler - G++ (GNU C++ Compiler)
5. IMPLEMENTATION DETAILS

5.1 Representation of the items
In this knapsack problem we’re using 1,000 items whose weights and values are stored as a data file. We’re accessing these data files by implementing a sequence container called vector. The capacity of the knapsack is limited to 50 with weights and values of items given between 1 and 30.

5.2 Initial population generation
Initial population is generated using the items which satisfy the constraints.

5.3 Fitness function
We calculate the fitness of each chromosome by summing up the weights of the items that are included in the knapsack, while making sure that the capacity of the knapsack is not exceeded. If the weight of the chromosome is greater than the capacity of the knapsack then we have a penalty condition for invalid solutions.

To choose which children from the newly created population will be favored to breed, the fitness of every individual has to be computed. This is done by summing up all the weights of its chromosomes and using them in the following formula where the wsum is the total weight of one individual found in this generation.

\[
\text{If}(\text{wsum}>\text{limit}) \\
\text{fit}=-7*(\text{wsum}-\text{limit}) \\
\text{return fit}
\]

Therefore the fitness is not an absolute measure like the weight(which can be compared over different generations) but a local measurement for this generation. The Fitness can take values between 0 (which is assigned to the individual with the maximal weight) up to theoretically 100(which is practice is not reached)

5.4 Selection Functions
In the implementation of the program, we tried the selection method group selection, combined with elitism, where two of the fittest chromosomes are copied without changes to the new population, so the best solutions found will not be lost.

Group Selection
We implemented this selection method in order to increase the probability of choosing fitter chromosomes to reproduce more often than chromosomes with lower fitness values. Since we could not find any selection method like this one in the literature, we decided to call it group selection. For the implementation of the group selection method, we use sorting to select the most fittest chromosome for next population generation.

5.5 Crossover

We tried single point, double point and arithmetic crossover strategies. Since there was not a big difference in the results we got from these methods. In one point crossover the crossover point is determined randomly by generating a random number between 0 and dimw - 1. We perform crossover with a certain probability. If crossover probability is 100% then whole new generation is made by crossover. If it is 0% then whole new generation is made by exact copies of chromosomes from old population. We decided upon crossover rate of 35% by testing the program with different values. This means that 35% of the new generation will be formed with crossover.

5.6 Mutation

Mutation is made to prevent Genetic Algorithms from falling into a local extreme. We perform mutation with 0.1 % probability. In the implementation we use the mutation strategy flip bit with mutation probability which is usually a very small value between 0.5% to 1%. We use a coin toss like strategy to select either the crossover or the mutation operations. In the implementation, the function int coin() takes the crossover ratio as its argument and check whether it is greater than the random number generated. If the condition is satisfied, the function returns a true value and perform crossover else performs mutation.

After performing mutation, we calculated the fitness of each chromosome considering the constraints to the 0-1 knapsack problem and calculated the average and best fitness values for each of them.

5.7 Termination Conditions

In the implementation, the chromosome population size is taken as 250 and it converges when the best fitness value becomes equal to 675.

5.8 Complexity of the program
Solving 0-1 knapsack problem using Genetic Algorithms

Since the number of chromosomes in each generation (pop) and the number of generations (gens) are fixed, the complexity of the program depends only on the number of items that may potentially be placed in the knapsack. We will use the following abbreviations, for the number of items, POP for the size of the population, and GENS for the number of possible generations.

The function that initializes the initial population has a complexity of O(N). The fitness, crossover function, and mutation functions have also complexities of O(N).

The complexity of the selection function do not depend on N (but on the size of the population) and it has constant times of running O(1).

The selection, crossover, and mutation operations are performed in a for loop, which runs pop (population size) times. Since, pop is a constant, the complexity of the whole loop is O(N). Finally, all these genetic operations are performed in another for loop, which runs at most gens (no. Of generations) times. Since gens is a constant, it will not affect the overall asymptotic complexity of the program. Thus, the total complexity of the program is O(N).
6. TESTING

The main philosophy behind testing is to discover errors and improve existing system. We used following software engineering methods for testing our application.

6.1 CODE TESTING

Code testing means checking correctness of code fragment by examining its logic using test cases.

This algorithm used test cases,
Case 1: 1000 items
Case 2: 10000 items

But this particular configuration is well suited for the 1,000 item test case, but may not be the best for other ones.

As there are more crossover techniques available in this implementation, we can test the algorithm with the parameters to achieve better results at different, possibly harder problem sets.

6.2 UNIT TESTING

In unit testing we concentrated on individual modules. Modules for Selection, crossover, Mutation etc were checked individually and errors were corrected.

6.3 INTEGRATION TESTING

In integration testing we tested our application by combining different modules.

6.4 SYSTEM TESTING

We tested our application in different platforms, like Windows and Linux. Our application was found to work efficiently on Linux based systems having G++ (GNU C++ compiler) compiler.

7. CONCLUSION

In this project we have seen how Genetic Algorithms can be used to find good solutions for the 0-1 Knapsack Problem.

The results from the program show that the implementation of good selection method and elitism are very important for the performance of genetic algorithm.

The algorithm used 1,1MB of memory for the 1,000 items, and still less than 3,5MB for the 10,000 item problem sets.
Solving 0-1 knapsack problem using Genetic Algorithms

The average time needed to compute the optimum with 1,000 items and a limit of 50 is found to be 0.05s - that's 1/20th of a second.

8. FUTURE SCOPE

To get an idea about problems solved by GA, here is a short list of some applications:

- Nonlinear dynamical systems - predicting, data analysis
- Designing neural networks, both architecture and weights
- Robot trajectory
- Evolving LISP programs (genetic programming)
- Strategy planning
- Finding shape of protein molecules
- TSP and sequence scheduling
- Functions for creating images