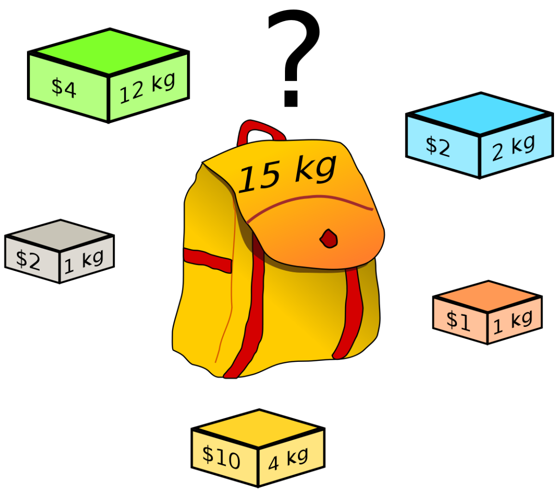
Askar Kolushev, 14.05.2020, Prague, Czechia

BIE-ZUM Report

Knapsack problem solution using genetic algorithm



Problem description

We are given a set *I* of *n* items and a knapsack of capacity *W*. Each item *i* is assigned a specific weight w_i and a value v_i. Our goal is to find the most profitable way to select *n'* items from those *n* to form a subset *I'* in such a way that we could put it into the knapsack. In other words, their total weight would not exceed *W* while the total value would be maximized:

$$I' \in I : (\sum_{i'=0}^{n-1} w_i \leq W) \wedge (\sum_{i'=0}^{n-1} v_i \ is\ max) .$$

Method of solution

The decision problem form of this problem is NP-complete, thus there's no known algorithm for solving it that is correct and fast at the same time. There exist some algorithms that output exactly the best solution. All of them, however, are highly inefficient. For example, one could use dynamic programming to optimize it at least to some extent. Some algorithms do output the maximum obtainable value, but do not tell us which exact items must be selected to get it.

In this work, I tried solving the problem using the *genetic algorithm*.

By definition, the genetic algorithm does not guarantee the best result, but rather approximates it still being quite fast.

In this approach, each way of selecting the items is represented using a binary array of *n* bits. Each bit s_i signifies whether or not the item *i* is being put into the knapsack. Each array represents one individual in the population. The array is then called a genome, and its length is equal to the number *n* of items.

A population is a set of individuals some of which are being picked for transmitting their genomes to the next generation. At the first step, the whole population is formed of individuals whose genomes were generated completely randomly.

Selection of two individuals from a particular population is done based on their calculated fitness. The fitness function is designed to simply return the total value of items selected by an individual if its total weight is less or equal to *W*, or zero otherwise. This way the fittest individual is the closest to the ultimate goal - maximal value with limited weight.

The evolutionary procedure then selects two individuals that will produce an offspring becoming a part of the population in the next generation. The selection is random, however, it gives the individuals with higher fitness value a higher chance to be selected.

The offspring production consists of two stages - crossover and mutation.

During the crossover, the genome of the offspring is formed from parts of the genomes of the “parents”. The sequence of bits is split randomly into several parts, and each part is filled with the bits from the genome of the Parent 1 and the Parent 2 in turn.

On the mutation stage, each bit in the offspring’s genome changes its value with some predefined probability. That assures the possibility that new types of genome appear in the population during evolution.

The whole process of selecting individuals and producing a population for the next generation is repeated a predefined number of times. The algorithm then outputs the fittest individual in the last generation’s population, and its genome is interpreted as the calculated selection of items to be put into the knapsack.

Implementation

The project is implemented in C++. The source code is available at https://github.com/kolusask/BIE-ZUM_SemesterProject. Commands build and run of the Makefile can be used to test the program.

- Genomes are represented by `std::vector<bool>` wrapped by the `Individual` class to provide reproduction logic.
- Logic for selection is implemented in the class `Population`.
- The driver code can be found in *`main.cpp`*, and
- `Random` is just a wrapper class for generating random values by means of the standard *`<random>`* header.
- Some parameters for tuning the process of evolution are defined in *`Constants.hpp`* header.

The program expects the input to contain the following space/newline separated values:

- *W* - a double value of the maximum weight capacity of the knapsack;
- *n* - an integer number of the items in *I* to select from;
- *n* pairs of doubles where each pair represents one item:
    - v_i - value of the item
    - w_i - weight of the item

Examples and results

In order to test my code, I needed a set of problems solved by an algorithm that guarantees the best result. I did not want and did not need to implement such an algorithm myself (nor solve the problems by hand). Instead, I just went online and found some examples solved using a "certain" algorithm and then compared their results with mine.

Conditions of the testing could be changed by setting the parameters in the *`Constants.hpp`* header.

I wanted to see how population size and number of generations in the process affect the correctness of the solution.

So I tried the following configurations:

`CROSSOVER_POINTS`	2		
`CROSSOVER_PROB`	0.63		
`MUTATION_PROB`	0.09		
`NUMBER_OF_GENERATIONS`	500	500	10
`POPULATION_SIZE`	100	10	100

In each of them, I ran the program 10 times, compared the results with expected (obtained by the author in the source) and counted how many times my results were correct.

The results are:

Source link	Input data	Correct output	Correct out of 10		
			NUMBER_OF_GENERATIONS		
			500	500	10
			POPULATION_SIZE		
			100	10	100
https://www.youtube.com/watch?v=oTTzNMHM05I	15 7 10 2 5 3 15 5 7 7 6 1 18 4 3 1	Elements to put into a knapsack: Number Value Weight 1: 10 2 2: 5 3 3: 15 5 5: 6 1 6: 18 4 Common value: 54 Common weight: 15	*9*	*3*	*8*
https://www.hackerearth.com/practice/notes/the-knapsack-problem/	10 4 10 5 40 4 30 6 50 3	Elements to put into a knapsack: Number Value Weight 2: 40 4 4: 50 3 Common value: 90 Common weight: 7	*10*	*9*	*10*
https://www.youtube.com/watch?time_continue=2058&v=gA0I_tFBCTE&feature=emb_title	5 4 3 1 9 3 12 4 8 2	Elements to put into a knapsack: Number Value Weight 2: 9 3 4: 8 2 Common value: 17 Common weight: 5	*10*	*10*	*10*
https://www.guru99.com/knapsack-problem-dynamic-programming.html	11 5 3 3 4 4 4 5 10 9 4 4	Elements to put into a knapsack: Number Value Weight 1: 3 3 2: 4 4 5: 4 4 Common value: 11 Common weight: 11	*10*	*8*	*10*
	15 5 4 12 2 2 1 1 2 1 10 4	Elements to put into a knapsack: Number Value Weight 2: 2 2 3: 1 1 4: 2 1 5: 10 4 Common value: 15 Common weight: 8	*10*	*8*	*7*
https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/	50 3 60 10 100 20 120 30	Elements to put into a knapsack: Number Value Weight 2: 100 20 3: 120 30 Common value: 220 Common weight: 50	*10*	*10*	*10*
	6 3 10 1 15 2 40 3	Elements to put into a knapsack: Number Value Weight 1: 10 1 2: 15 2 3: 40 3 Common value: 65 Common weight: 6	*10*	*10*	*10*
https://www.youtube.com/watch?v=8LusJS5-AGo	7 4 1 1 4 3 5 4 7 5	Elements to put into a knapsack: Number Value Weight 2: 4 3 3: 5 4 Common value: 9 Common weight: 7	*10*	*10*	*10*

As we can see, the success rate lowers noticeably when the population size is decreased. An impact of the reduced number of generations is not as significant but still can be seen. We can also observe a slight positive dependence of this drop on the size of the input data.

General explanation for this behavior is that in a smaller population, the individuals have a higher chance of being selected for reproduction even when their fitness is not as high as we would like. A smaller number of generations, on the other hand, does not give the population enough time to develop an individual with such fitness.

It is also worth noting that even outputting an incorrect solution, the algorithm gave a well approximated result, which is quite expected.

Conclusion

In this project, I implemented a solution to the Knapsack problem using evolutionary approach and the genetic algorithm.

The algorithm did not show itself as an ideal method for finding the best way of solving it, as the solution provided by it was not the most optimal quite often.

On the other hand, it was not the case in the majority of the tests, and even when the output was not the best, it was still approximated quite well.

The general performance has shown itself to be highly dependent on the evolution parameters, namely the number of generations and especially the size of the population.