Generating General Solutions to the Knapsack Problem via Genetic Algorithms

CS 461 Program 2 Writeup

Ryan Phillips (rcpvhm@umsystem.edu)

https://github.com/ryanphilly/cs461/tree/main/knapsack_genetic

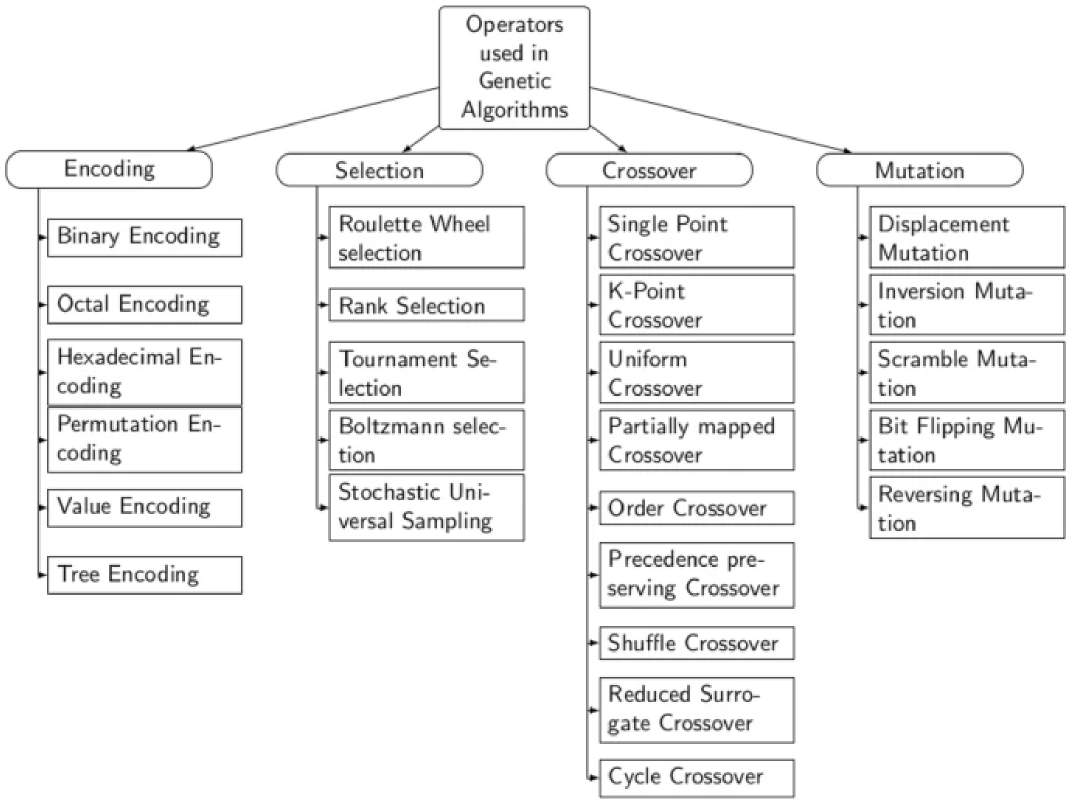
Knapsack Problem Formulation:

$$\max \sum_{i=1}^{I} v_i x_i$$

$$\text{such that} \sum_{i=1}^{I} w_i x_i \leq W$$

$$x_i \in \{0, 1\} \ \forall i \in [I]$$

Genetic Algorithm Implementation Details:



The figure above shows all operations widely used for constructing a genetic algorithm.

Encoding (how to represent a selection):

Permutation Encoding (sets of item indices) was used in this implementation to represent the selected items in the input data. In the case of selections with substantial amounts of items present, representing these selections may potentially require more memory than a Boolean array or raw bytes. However, it seems most viable selections are a small subset of the total items, which results in sparse Boolean arrays, bytes, or string representations. Therefore, the selections require needless computation for querying and evaluating compared to permutation encoding.

e.g.

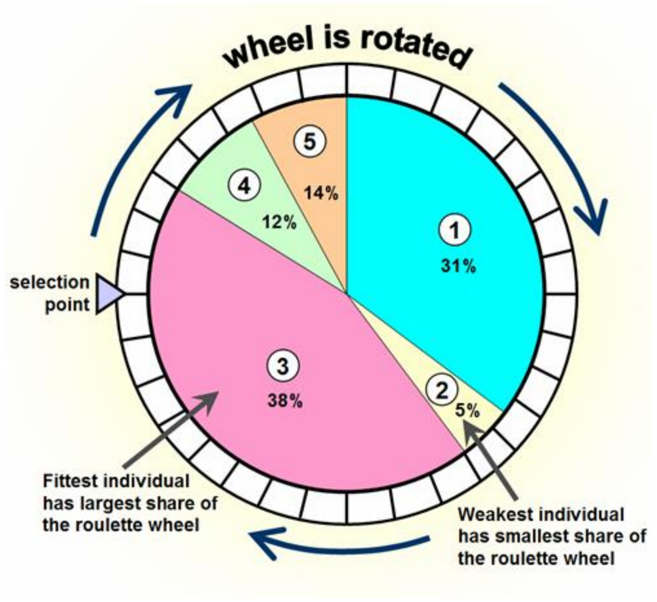
All Items: {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}

Chromosome A	{1 3 6 9 15 }
Chromosome B	{8 5 7 2 3 4 9}

Selection (choosing what selections should reproduce):

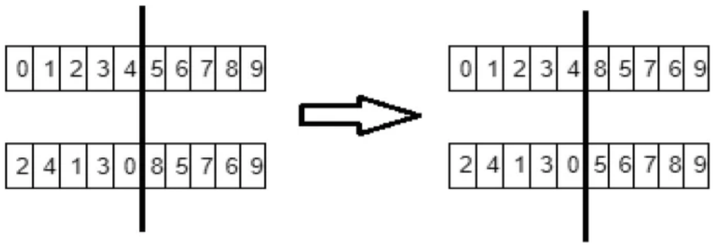
Fitness Proportionate Selection (roulette wheel selection) was used for the selection process. The selection is weighted by the L2 normalization of the population's fitness. Population size is kept steady meaning no culling or elitism is performed, which gives a higher chance of a more diverse breeding pool.

e.g.


wheel is rotated
5
14%
4
12%
1
31%
selection point
3
38%
2
5%
Fittest individual has largest share of the roulette wheel
Weakest individual has smallest share of the roulette wheel

Crossover (how offspring are generated):

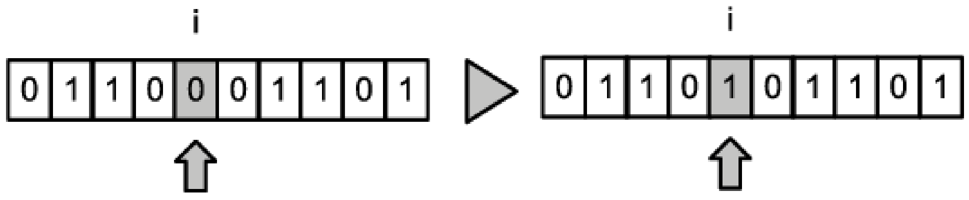
Single Point Crossover is used to generate two offspring from two parents in the breeding pool keeping the population size steady.



Mutation (how offspring are randomly altered):

Bitflip Mutation is applied on all genes of both offspring with a specified probability. The probability or mutation rate is constant throughout training/maximizing.

e.g.



Optimization and Conclusions:

The implementation described above is completely serial, and there is no vectorization or parallelization done. While the implementation does generate complete solutions to the knapsack problem, it does so very slowly. Parallelization was attempted across multiple CPU cores using pythons' multiprocessing library. However, distributing and congregating the data across the cores resulted in an even slower runtime. Future work could use the massive number of threads on modern day GPUs to efficiently parallelize the implementation but was out of the scope of this project. When testing the implementation on multiple different starting population sizes, a couple of insights appear: 1) Bigger populations avoid premature convergence at the cost of computation time. This is due to having a much bigger and diverse population to sample and breed from; and 2) Bigger populations require fewer generations to converge. However, no measurements were taken to verify if this convergence was faster or slower than smaller populations.

Assignment Specific Observations:

The assignment given data contained only utility and weight. When computing the utility over-weight heuristic and then picking the top scores until the weight limit is hit, it resulted in a fitness of 740 in only a few milliseconds. When running the genetic algorithm with a starting population of 5,000, it converged at a fitness of ~741.1, but it took several minutes. The results would obviously be different if more constraints were added to the data such as volume, material flexibility, etc.