Login Name	**:**	*jxk19u*
Family Name	**:**	*Kremser*
Other Names	**:**	*Jiri*

G52AIM (Artificial Intelligence Methods)

Lecturers: Graham Kendall & Rong Qu

Academic Year 2009-2010

(Coursework II – Solving the 0-1 Multidimensional Knapsack Problem)

A. Initial implementation and description.

I have implemented a genetic algorithm (GA) to solve the 7th instance of 0-1 MKP in the dataset mknap1 from the OR Library. This part will describe high level concepts of my GA. Many attributes of the GA can be modified and change the nature of the GA. I used the *Steady State* method, it means that in each iteration only one offspring is created and it replaces some member with worse fitness.

1. Representation

A member of population of the GA in a time represents one particular solution of 0-1 MKP. It is done by representing the solution of 0-1 MKP as a sequence of bits $\{0,1\}^n$, where n denotes the number of items and zero or one the presence of the corresponding item in the knapsack. In terms of evolutionary computing this vector is called chromosome and since it is arbitrary sequence of bits, it can represent the feasible solutions as well as the infeasible. Infeasible solution is such solution, whose at least one constraint is violated.

2. Initial population

At the beginning of the computation the population has to be filled by the members. The size of the population is invariant during whole computation process since in each iteration an offspring replaces just one member. I used two approaches for creating new solutions to fill the population.

2.1. Random constructor

Starts with the empty solution (zeros) and then randomly chooses bits (gens) from the chromosome and change them to one if the solution remains feasible. Once the change may violate the feasibility, this procedure stops and returns the feasible solution.

2.2. Greedy constructor

Can be parametrized by an integer k which determines how many randomly chosen bits will be set to one before the greedy improvement method is applied. Improvement method is the same method which is used by repair operator (**6.2**).

3. Parents selection

GA uses the Tournament method with parameter l denoting the *pool size*. This method randomly selects $2l$ members from population and places them into two pools. Two individuals with the highest fitness, each taken from the different tournament pool, are then chosen as future parents. Large value of parameter l leads to selecting the fittest members of the population, in particular when l = population size / 2, the fittest individual is always selected as a parent. If the value of l is small enough, the less fit individuals will be given the chance to reproduce which significantly contributes to keeping the population divergent.

4. Crossover

Once the parents are selected, the Uniform Crossover is used for creating a new individual. Each bit of the new offspring's chromosome is chosen randomly from the chromosome of one of its parents on the same position as shown at *Table 1*. *P1* and *P2* are the parents and *O* is the offspring.

P1	1	0	0	0	1
P2	1	1	0	1	0

from	P1	P2	P2	P1	P1
O	1	1	0	0	1

Table 1: Example of the Uniform Crossover

5. Mutation

Three types of mutation can be set in configuration file[1]. The first type is negation of one bit of chromosome, the second is negation of two bits and the third is negation of all bits with a certain degree of probability *p*. This probability *p* have to be set to a small value and may depend on the length of the chromosome (number of items). If the $p = 1$, offspring are created randomly and the benefits of GA are degraded to simple *generate and test*[2] algorithm.

6. Repair operator

Crossover and mutation may leave a solution infeasible, therefore GA have to deal with infeasible solutions. One way is penalizing infeasible solutions' fitness and allow them in the population. Another way, which I chose, is to repair infeasible solution after crossover & mutation phase. The repair operator does nothing if a solution is feasible. When the solution is not feasible, then two phases of repair operator are executed. Firstly the phase **drop** and consequently the phase **improve**. Both of them use the notion of *pseudo-utility* and greedy approach based on this *pseudo-utility* values.

Notation p_j, x_j, w_{ij}, c_i means price, presence in the knapsack, consumption of the resource *i* of item *j* and capacity of the resource *i*.

6.1. Drop phase

This phase finds out which constraint, denoted as *i*, is violated, if more are, takes the first. The item to drop is then $j = argmin_j \{u_j | u_j = p_j / w_{ij}\}$ with probability $p_{dropWorst}$ which is a parameter of GA. If the item with the lowest *pseudo-utility* value u_j is not chosen since probabilistic test with $p_{dropWorst}$ failed, then the second worst item is removed from knapsack. Repeat whole procedure until the solution is feasible. In the contrast with the improve phase (**6.2**) the vector $\vec{u} = (u_j)$ depends only on weights from one dimension (constraint).

1 Configuration file of GA is described in attached readme file.

2 More precisely generate, improve and test i. e. running greedy search many times on different initial solutions and accept the best solution.

6.2. Improve phase

This phase is dual to the previous one. It tries to add the best items into the knapsack since there can be some "free space". Index of the item to be added is the *j* from (1).

$$j = argmax_j \{u_j | u_j = p_j / v_j\} \tag{1}$$

$$v_j = max_{i=1,\dots,m} \{ \overbrace{(DA_i + w_{ij})}^{A} \underbrace{(\sum_{k \in SC} w_{ik} - w_{ij})^{\frac{1}{2}}}_{B} / \underbrace{(c_i - DA_i - w_{ij})^{\frac{1}{2}}}_{C} \} \tag{2}$$

In (2) v_j denotes the *aggregated pseudo-utility[LM78]* value for the *j-th* item, DA_i denotes an amount of resource that has already been consumed ($DA_i = \sum_j^n w_{ij} x_j$), set SC is a set of items not in the knapsack ($SC = \{j | x_j = 0\}$). Formula (2) can be divided into three parts *[LM78]*.

- **A** stands for total consumption of resource *i* if item *j* is added to current solution.
- **B** stands for future potential demand for resource *i* if item *j* is added to current solution.
- **C** stands for an amount of resource *i* remaining if item *j* is added to current solution.

Once the best candidate item is known, it will be added with probability $p_{addBest}$. If the probabilistic test fails, the second best item is added. If **C** is lower than 0 for some item and some constraint, the corresponding item will not take part in the next iteration (adding another item). Repeat these steps while the solution is still feasible, in other words while there is still some item to be added (exist item from SC and its "**C**" > 0 for all constraints).

7. General

GA will end after fixed amount of iterations (can be parametrized). If a new offspring has the same chromosome as some member from the population, the reproduction process is repeated. If the population is too uniform it may happen this algorithm does not converge, so that there is an parameter denoting the maximum number of such "unsuccessful" reproductions. If this number is exceeded the GA will end artificialy.

B. Analysis of the experiment results

The Genetic Algorithms are stochastic search techniques, it means that the result of a particular GA is non-deterministic. One of the parameters, which can influence the behaviour of the computation itself, is random seed. If the same random seed is given as a parameter to my GA, all the numbers generated by Java's pseudo-random generator will be the same no matter how many times the algorithm is run. In this sense it will behave deterministic and it is easy to debug such algorithm. If this parameter is set to -1, the pseudo-random generator will not be predictable.

1. Parameters of the GA

- population size
- initialize randomly (with parameter *fixed genes* which is the *k* from **A.2.2**)
- improve when repairing (no improve phase if this parameter is set to true)
- drop second probability (complement to $p_{dropWorst}$ from **A.6.1**)
- take second probability (complement to $p_{addBest}$ from **A.6.2**)
- pool size (pool size from **A.3**)
- mutate (paragraph **A.5**)
- replace worst (new offspring replaces either the worst member of population of some worse member)
- iterations, max unsuccessful reproduction attempts (**A.7**)
- random seed

2. Tests

It is not obvious how to set up the parameters. The default values for some parameters were inspired by [CB98]. However, I provided my algorithm with a set of bash scripts, which run the GA many times and compute the average best fitness since one measurement is useless. I created 39 configuration files by combining the values of the parameters and then run each parametrized GA circa five hundred times. The results are saved to files in a directory tests/results. The first line of a result file is the *average best fitness* and the second line is the degree of *confidence*, in other words the number how many times the test was run. If the test suite is run again, the result files will be updated and the precision of the *average best fitness* for particular configuration will be increased. The best results for the 7th instance of 0-1 MKP in the dataset mknap1 were achieved by “GA 26”. For 500 runs the average best fitness was 16521 i.e. 0.99903 % of optimum.

Then I run another test for the “GA 26”, which counts how many times the optimum solution was found. Since the experiment should be repeatable I set the random seed from 1 up to 100. The problem was solved to optimum 16 times. Nearly all other results were local optimum 16519.

The parameters used at configuration “GA 26”

- populationSize=*300*
- initializeRandomly=*true*
- improveWhenRepairing=*true*
- takeSecondProb=*0.01*
- dropSecondProb=*0.5*
- poolSize=*2*
- mutate=*2* (all gens)
- mutateProb=*0.005*
- replaceWorst=*false*
- iterations=*100000*
- maxReproductionAttempts=*3000*

3. Random vs greedy-like constructor

When the initial population of GA is created randomly, the GA gives, surprisingly, better solutions. This is given by the fact, that the population created randomly is more divergent than the population created by greedy heuristic. Other factor supporting this hypothesis is using the same greedy heuristics for generating initial population and for improving solutions. The reason why the greedy heuristic can accept the second best candidate is to make it non-deterministic, so that the GA will not suffer from uniformity of the population.

The average best solution returned by GA with greedy-like constructor is ~16422 (“GA 38”). The progress of evolution using the random and/or greedy-like constructor can be seen at the graphs in the **Appendix**.

4. Complexity

Let *n* is the number of items, *m* the number of dimensions, *p* the number of candidate parents in the pool (*poolSize* parameter) and *u* the *maxReproductionAttempts* parameter. Time complexity of evolution phase[1] is then

$$i \times O(\overbrace{O(u) \times [O(p)+O(n)+O(nm)+O(nm^2)+O(mn^2)]}^{reproduction}) = i \times O(un^2m^2)$$

Member $O(p)$ can by omitted since *p* can not be greater than *n*. In the formula's part “reproduction” there are the complexity of *parent selection* $O(p)$, the complexity of *crossover&mutation* $O(n)$, the complexity of *feasibility test* $O(nm)$, the complexity of *drop phase* $O(nm^2)$ and the complexity of *improve phase* $O(mn^2)$. The drop function and the improve function are in fact effective and the worst case nm^2 (mn^2) can never appear. On the average only a few items must be removed if the solution is broken and only a few items can be added subsequently.

5. Performance

On my laptop with Core2 Duo 2.1GHz, 2GB RAM the “GA 26” took approximately 7.5 seconds, “GA 38” took 7 seconds. The second best GA was algorithm with configuration 37, its average best fitness was 16517.8 and it took only 2 seconds.

1 The initialization phase is omitted since it is not critical.

Resources

[CB98] P. C. Chu, J. E. Beasley, *A Genetic Algorithm for the Multidimensional Knapsack Problem* , Imperial College, 1998.

[LM78] R. Loulou, E. Michaelides, *New Greedy-like Heuristics for the Multidimensional 0-1 Knapsack Problem*, McGill University, Montreal, 1978.

[TOY75] Y. Toyoda, *A Simplified Algorithm for Obtaining Approximative Solutions to Zero-one Programming Problems*, Aoyama Gakuin University, Tokyo, 1975

[FMD09] K. Florios, G. Mavrotas, D. Diakoulaki, *Solving multiobjective, multiconstraint knapsack problems using mathematical programming and evolutionary algorithms*, European Journal of Operational Research, 2009

Apendix

Fitness denotes the best fitness in population

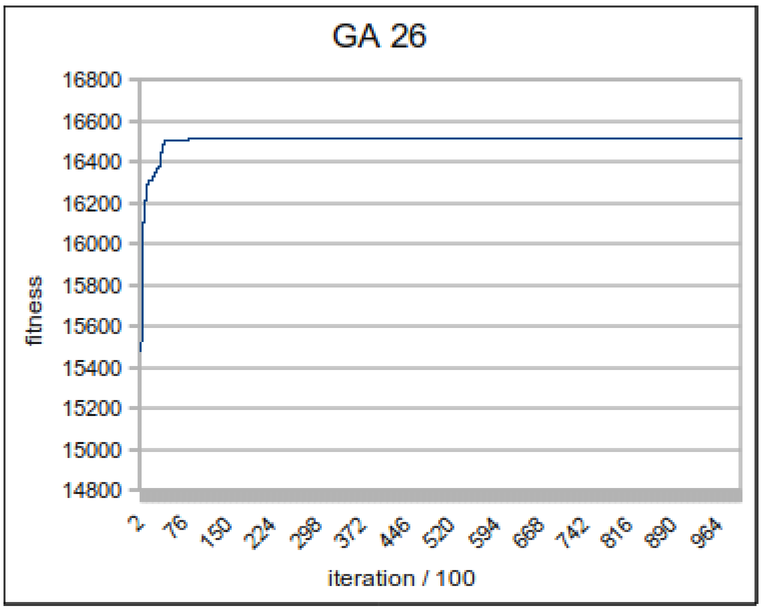

Figure 1: random constructor, average best fitness was 16521 with avg. time 7.5 sec

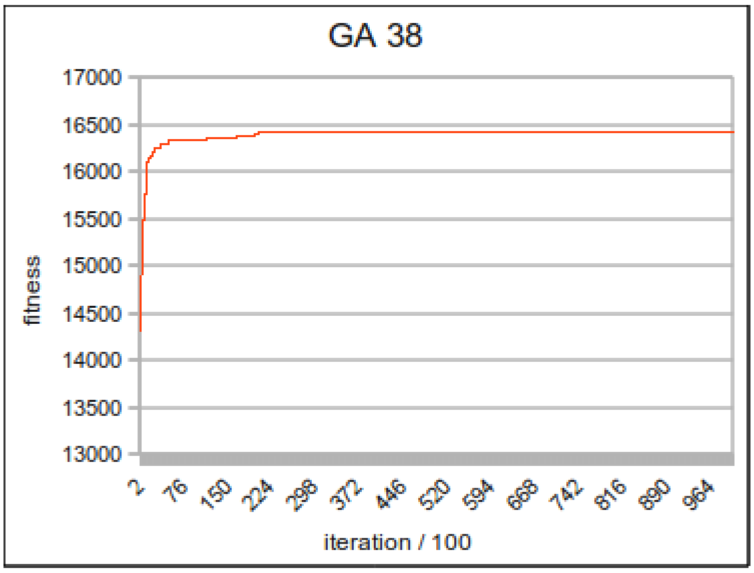

Figure 2: greedy constructor, average best fitness was 16430 with avg. time 7 sec

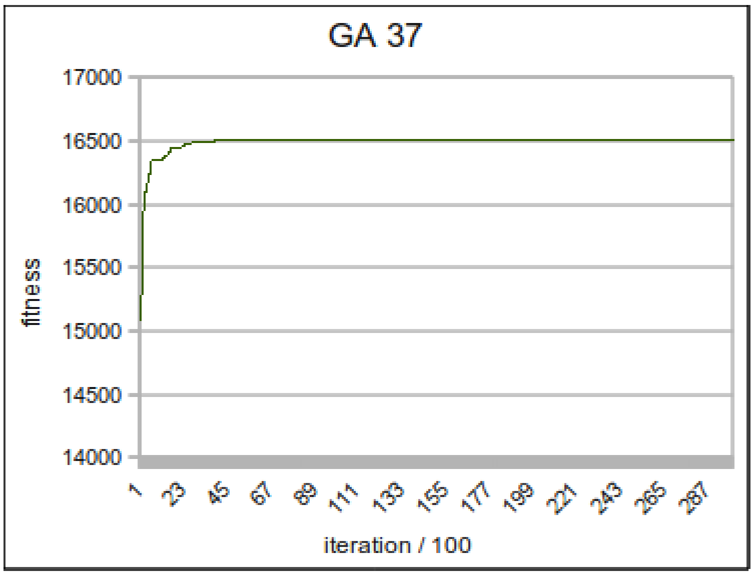

Figure 3: the second best alg., average best fitness was 16517.8 with avg. time 2 sec