31 Genetic Algorithms

Robert Klein and Oliver Faust

31.1 General Idea

Many optimization problems of the type arising in scheduling and routing (see Chaps. 10 and 12) are of combinatorial nature, i.e. solutions are obtained by combining and sequencing solution elements. When solving such problems to optimality, the number of solutions to be examined exponentially grows with the problem size. For example, for n solution elements $n!$ different sequences exist.

Since the 1990s, *genetic algorithms* (GA) have become increasingly popular as a means for solving such optimization problems heuristically, i.e. for determining near-optimal solutions within reasonable time. One of the main reasons for this popularity is the relative ease of programming at least a simple genetic algorithm. Furthermore, many researchers have observed empirically that already basic versions of GA will give very acceptable results without excessively fine-tuning them for the problem on hand. Finally, since GA work on a representation (coding) of a problem (see Sect. 31.2), it is possible to adapt existing procedures to modified problem versions quite easily or to write one general computer program for solving many different problems. GA were initially developed by Holland and his associates at the University of Michigan and the first systematic but rather technical treatment was published in Holland (1975). Reeves (1997) and Reeves (2010) provide a detailed overview on the topic. For comprehensive descriptions from a practical point of view, we refer to Goldberg (1989), Haupt and Haupt (2004), Michalewicz (1999) and Reeves and Rowe (2003).

According to the biological evolution, GA work with *populations* of *individuals* which represent feasible solutions for the problem considered. The populations are constructed iteratively through a number of *generations*. Following the idea

R. Klein • O. Faust (✉)
University of Augsburg, Universitätsstraße 16, 86159 Augsburg, Germany
e-mail: robert.klein@wiwi.uni-augsburg.de; oliver.faust@wiwi.uni-augsburg.de

H. Stadtler, C. Kilger and H. Meyr (eds.), *Supply Chain Management and Advanced Planning*, Springer Texts in Business and Economics,
DOI 10.1007/978-3-642-55309-7__31, © Springer-Verlag Berlin Heidelberg 2015

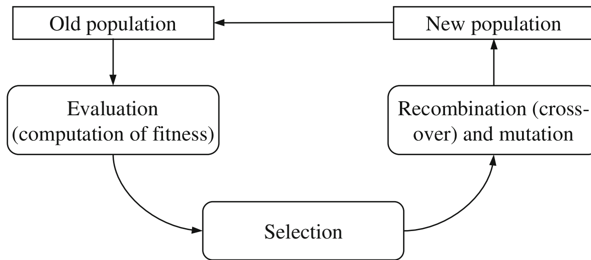

Fig. 31.1 Template for a single iteration of a genetic algorithm

Table 31.1 Data of an example problem

j	1	2	3	4	5	6	7	8
d_j	5	4	7	8	3	2	6	4
rd_j	0	2	4	16	18	28	25	28
dd_j	17	10	13	28	22	31	36	36
c_j	3	2	5	3	4	1	3	4

of Darwinism ("survival of the fittest"), each individual of the current generation "contributes" to the subsequent one according to its quality which is measured by a *fitness value*. This is achieved by *selecting* individuals randomly with the probability of choosing a certain individual depending on its fitness value (see Sect. 31.3). In order to obtain the next generation from the individuals selected, two basic operations exist (see Sect. 31.4). Using a *crossover*, the features of two (parent) individuals are *recombined* to one or more new (child) ones. By *mutation*, some features of an individual are modified randomly. A template for a single iteration of a genetic algorithm is depicted in Fig. 31.1. Usually, GA are executed until a prespecified stopping criterion is fulfilled, e.g. a certain number of generations has been evaluated or a time limit is reached.

Recently, hybrid approaches have been developed which incorporate *local search algorithms* (LSA) in GA in order to leverage problem-specific knowledge (see Sect. 31.5). The local search part of these so-called *memetic algorithms* (MA) is typically executed after recombination.

In the following, we discuss the different aspects of GA in more detail. To ease presentation, the following *production scheduling problem* is considered. A number n of jobs has to be processed on a single machine (with simultaneous execution being impossible). Each job $j = 1, \ldots, n$ has a fixed processing time (duration) of d_j periods and preemption is not allowed. Furthermore, job j cannot be started before its release date rd_j and should be terminated until a due date dd_j. In case it is finished later than dd_j, a penalty cost c_j for each time unit of tardiness arises. Hence, the problem consists of finding a schedule, i.e. a starting time s_j for each job, such that the total tardiness costs are minimized. The data of an example with $n = 8$ jobs are given in Table 31.1.

31.2 Populations and Individuals

As stated before, a population consists of a set of individuals. Each individual is *represented* by a vector (*string*) of fixed length in which the corresponding solution is coded by assigning specific values to the vector elements (*string positions*). In order to obtain the solution associated with an individual, the respective string has to be *decoded*. Both the dimension (length) of the string as well as the domains (sets of feasible values) of the string positions depend on which representation is chosen for coding the solution.

In our example, a solution can be represented by a sequence S of jobs. That is, the string consists of n positions and each position can take one of the values $1, \ldots, n$ (with all positions having different values). For decoding the string, we proceed as follows. The jobs are considered in accordance to the sequence S. The job j in turn is started at the smallest possible point in time $s_j \geq rd_j$ at which its execution does not overlap with a job already scheduled. After having scheduled all jobs, the total tardiness costs can be computed. Consider the string $S = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$ for our example. By decoding this sequence, the solution shown in the Gantt-chart of Fig. 31.2 is obtained. The numbers within the bars denote the job numbers and the tardiness of the jobs, respectively. The lengths of the bars correspond to the processing times.

Job 1 can be started at the earliest point in time $s_1 = 0$. Scheduling job 2 results in $s_2 = 5$ due to the processing of job 1 which does not allow for a smaller starting time. After terminating job 2, job 3 can begin at $s_3 = 9$, hence, finishing three periods after its due date $dd_3 = 13$. The jobs 4 and 5 are scheduled subsequently. Job 6 cannot be launched earlier than $s_6 = rd_6 = 28$. Finally, the jobs 7 and 8 are considered with the latter terminating after 40 periods. The total tardiness costs are $3 \cdot 5 + 5 \cdot 4 + 4 \cdot 4 = 51$.

Note that we have chosen the above representation, because it is well suited for a large number of scheduling and routing problems and, hence, is used by a large number of GA for such problems (Reeves 1997). Alternatively, GA are often applied using a representation where solutions are coded in a bitwise fashion, i.e. each string position can take either the value 0 or 1. Such a representation is appropriate, when it has to be decided whether certain elements are part of a solution or not.

In any application of GA, an important question consists of choosing an appropriate *population size* P, i.e. the number of individuals considered in each iteration. If the population size is too small, the search space of feasible solutions may only be evaluated partially, because just a few existing individuals are recombined in each iteration and these individuals increasingly resemble each other with each additional generation. Otherwise, in case of a too large population size, also rather poor individuals may be considered for recombination. This is in particular disadvantageous in case that for each new string large parts of the corresponding solution have to be reconstructed as in our example, which requires a considerable computational effort. Hence, with an increasing problem size, most of the computational time will be spent for constructing solutions rather than

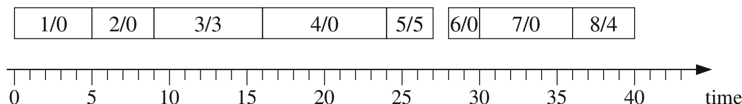

Fig. 31.2 Gantt-chart for $S = \langle 1, 2, 3, 4, 5, 6, 7, 8 \rangle$

examining the search space and the search will only proceed slowly towards high-quality solutions. In the literature, most successful applications of GA propose an even-numbered population size of $P \in [50, 100]$ (Reeves 1997).

Finally, an *initial population* has to be determined before starting a genetic algorithm. Most commonly, the corresponding individuals are obtained by randomly assigning values to the string positions. In our example, sequences of jobs may be constructed randomly. Alternatively, simple heuristics, such as randomized priority-rule based approaches, may be applied in order to start the search with promising solutions.

31.3 Evaluation and Selection of Individuals

As stated previously, individuals contribute to the next generation with a probability depending on their fitness value. For this purpose, a *gene pool* consisting of P copies of individuals is constructed. For those individuals with a high fitness value, several copies are included in the pool, i.e. the individuals are selected several times, whereas for those with low values no copy may be contained at all. This reflects the analogy to biological evolution. The best individuals should contribute to the next generation the most often, i.e. their positive features are reproduced in many of the new individuals. By way of contrast, the worst ones with a low selection probability should be discarded and, hence, "die off".

In the most simple form, determining the fitness values v_i for the individuals $i = 1, \ldots, P$ consists in computing the objective function values f_i of the corresponding solutions.

For maximization problems, the selection process often used within GA can be subdivided in the following two steps. In the first step, a roulette wheel with $i = 1, \ldots, P$ slots sized according to the fitness values $v_i = f_i$ is constructed. For this purpose, the total fitness of the population is computed by $T = \sum_{i=1}^{P} v_i$. Subsequently, each individual i is assigned a selection probability of $p_i = v_i / T$ as well as a cumulative probability $q_i = \sum_{h=1}^{i} p_h$. In the second step, the roulette wheel is spun P times. In each iteration, a single individual is selected, i.e. a copy is included in the gene pool, as follows. After generating a random float number $\beta \in (0, 1)$, the individual $i = 1$ is chosen in case of $\beta \leq q_1$. Otherwise, the i-th individual with $q_{i-1} < \beta \leq q_i$ is picked.

The above selection process bears the difficulty that if the objective is minimization instead of maximization as in our example, a transformation of the objective function values has to be performed. One simple transformation consists

of defining an upper bound F which exceeds all possible objective function values and subsequently using the fitness value $v_i = F - f_i$. Another difficulty is that the scale on which the values are measured may not be considered appropriately. For example, values of 1,020 and 1,040 are less distinctive than values of 20 and 40.

Therefore, two possible alternatives for designing the selection process have been proposed in the literature. When using a *ranking* approach, the individuals are ordered according to non-deteriorating fitness values with r_i denoting the rank of individual i. Subsequently, a selection probability is computed by, e.g. $p_i = 2r_i / (P \cdot (P + 1))$. In this case, the best individual with $r_i = P$ has the chance of $p_i = 2/(P+1)$ of being selected. This is roughly twice of that of the median whose chance is $p_i = 1/P$. With the values p_i on hand, the selection can be performed by spinning the roulette wheel as described above.

The other possibility is the *tournament selection*. In this approach, a list of individuals is obtained by randomly permuting their index numbers $i = 1, \ldots, P$. Afterwards, successive groups of L individuals are taken from the list. Among these individuals, the one with the best objective function value is chosen for reproduction and a copy is added to the gene pool. Then, the process is continued with the next L individuals until the list is exhausted or the gene pool contains P copies, whatever comes first. In the first case, the tournament process is continued to determine the missing members of the gene pool after determining a new list randomly.

Except for the tournament selection, the above approaches have in common that there is no guarantee that the best of all individuals is selected for reproduction. From the optimization perspective this may not be efficient. Therefore, the concept of *elitism* has been introduced which consists of putting a copy of the best individual into the gene pool by default and applying the roulette wheel and ranking approaches only $P - 1$ times. In generalized versions, a larger number of individuals is chosen by default.

31.4 Recombination and Mutation

For the *recombination* process, a pair of individuals is chosen from the gene pool either randomly or systematically. A *crossover* is carried out with a certain probability γ, i.e. the pair is recombined into two new individuals. In case that no recombination is performed, the original individuals become part of the new population with a probability of $1 - \gamma$. This process is repeated until P individuals have been considered and, hence, a new population with P individuals has been obtained. In the literature, different values for γ have been proposed with values of $\gamma < 0.6$ not being efficient (Reeves 1997).

In the following, we describe the simple *1-point crossover* which is the one most commonly used. In general, it is defined for strings with length n as follows. For each pair of parent individuals X and Y, a *crossover point* $h \in [1, n - 1]$ is determined randomly. Afterwards, a first individual is obtained by concatenating the first h string positions of X with the $n - h$ last positions of Y. The second individual

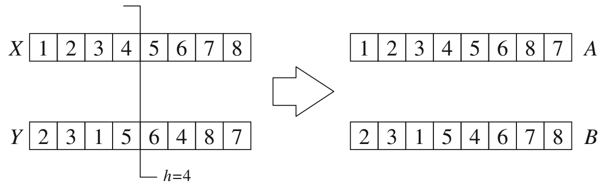

Fig. 31.3 1-Point crossover for sequence representations

is obtained just the other way round. Unfortunately, this definition does not work for every possible representation of solutions. For our example problem, such a crossover results in individuals with feasible solutions when the representation based on priority values is applied but fails for the representation relying on sequences. In the latter case, it yields individuals with some jobs occurring twice and others being discarded.

Therefore, a different approach is used for sequence based representations, the principle of which is depicted for two possible strings of our example problem (Fig. 31.3). After selecting a crossover point $h \in [1, n-1]$ randomly, the first h string positions of the parent individual X are copied into the child one A. Subsequently, the remaining $n - h$ positions are filled up with those elements which have not been considered yet in the order in which they are contained in individual Y. The second child B is constructed accordingly now starting with the first h positions of individual Y. Note that in our example both individuals obtained by the crossover yield better objective function values than their parent ones. The total tardiness costs of the schedules represented by X and Y are 51 and 93, whereas for A and B we obtain 47 and 29, respectively.

In addition to recombination, *mutation* is applied for some of the new individuals to diversify the search, i.e. to avoid that the same set of solutions is examined repeatedly through a number of consecutive generations. For this purpose, a probability δ with which each individual is mutated has to be specified. According to the selection process, the decision whether to mutate an individual or not can be made by randomly generating a number from the interval $(0, 1)$. The usual approaches for determining δ are either to choose a very small value, e.g. $\delta = 0.01$ or to use a value $\delta = 1/n$, because there is some theoretical and practical evidence that this is a reasonable value for many problems (Reeves 1997). In general, mutation consists of randomly altering the value at a random string position. In order to preserve the feasibility for sequence based representations, two more versatile mutation possibilities are distinguished. Within an *exchange* mutation, two string positions are randomly selected and the corresponding elements are interchanged. In our example, the positions three and six may be selected for individual A resulting in the mutated sequence $A' = \langle 1, 2, 6, 4, 5, 3, 8, 7 \rangle$. A *shift* mutation consists of randomly choosing a single string position and moving the corresponding element by a random number of positions to the left or right. After selecting position six of individual B and left shifting the element by three positions, we yield $B' = \langle 2, 3, 6, 1, 5, 4, 7, 8 \rangle$.

31.5 Memetic Algorithms

GA are able to quickly explore the search space and find regions with high-quality solutions. However, they typically exhibit less efficiency with regard to the exploitation of promising regions of the search space. In contrast, LSA are able to quickly exploit a given region, scanning the *neighborhood* of a current solution in search of better adjacent solutions. Therefore, LSA implement a *move operator* which is used for moving through the neighborhood. While this approach allows for quickly finding local optima using a problem-specifically defined neighborhood, LSA tend to insufficiently explore the search space.

MA constitute a hybridization of GA and LSA, and are therefore meant to embody the best of both worlds. The name of this class of evolutionary algorithms stems from the word *memes* defined by Dawkins (2006). A meme can be regarded as valuable knowledge which is transferred from one generation to another. In the context of optimization, LSA are used to incorporate problem-specific knowledge into chosen individuals by finding new individuals with better fitness values within their neighborhood. There are two ways to achieve this: *Lamarckian* MA replace every individual altered by LSA whereas *Baldwinian* MA keep the original individuals and replace their fitness values with those of the new individuals generated by the LSA.

As LSA can be regarded as systematic mutations guided by problem-specific knowledge, LSA in MA most commonly substitute the mutation operator and are thus executed after recombination. If mutation shall also be applied, the mutation operator should differ from the move operator of the applied local search algorithm in order to achieve diversification by searching in a different neighborhood. Individuals are either chosen randomly for alteration by LSA or according to their fitness value.

MA have successfully been used for solving a number of scheduling problems. For further reading we refer to Eiben and Smith (2007) and Moscato and Cotta (2010).

31.6 Conclusions

The previous expositions aim at reviewing the basic ideas of GA in the context of solving combinatorial optimization problems. They also show that a large variety of design possibilities exist when implementing GA for particular problems. This includes choosing a representation of solutions, a selection mechanism as well as efficient recombination and mutation strategies. Fortunately, as stated at the beginning, already basic versions of GA are robust in the sense that they are able to yield satisfying results for many problems.

Depending on the problem to be solved, it may be difficult to consider constraints appropriately, i.e., to avoid that infeasible solutions are obtained throughout the solution process. Within production scheduling, such constraints may be due to

generalized precedence relationships among jobs or to time windows for their execution. In order to overcome such difficulties, several concepts have been developed. The most common one is to modify the objective function by a penalty term such that infeasible solutions are assigned a low fitness value.

References

Dawkins, R. (2006). *The selfish gene* (30th anniversary ed.). New York: Oxford University Press.

Eiben, A., & Smith, J. (2007). *Introduction to evolutionary computing* (1st ed., corrected 2nd printing). Natural Computing Series. Berlin: Springer.

Goldberg, D. (1989). *Genetic algorithms in search, optimization, and machine learning*. Reading: Addison-Wesley.

Haupt, R.,& Haupt, S. (2004). *Practical genetic algorithms* (2nd ed.). Hoboken: Wiley.

Holland, J. (1975). *Adaptation in natural and artificial intelligence*. Ann Arbor: University of Michigan Press.

Michalewicz, Z. (1999). *Genetic algorithms + data structures = evolution programs* (3rd ed.). Berlin: Springer.

Moscato, P., & Cotta, C. (2010). A modern introduction to memetic algorithms. In M. Gendreau & J.-Y. Potvin (Eds.), *Handbook of Metaheuristics* (2nd ed.). *International Series in Operations Research and Management Science* (Vol. 146, Chap. 6, pp. 141–183). New York: Springer.

Reeves, C. (1997). Genetic algorithms for the operations researcher. *INFORMS Journal on Computing 9*, 231–250.

Reeves, C. (2010). Genetic algorithms. In M. Gendreau & J.-Y. Potvin (Eds.), *Handbook of Metaheuristics* (2nd ed.). *International Series in Operations Research and Management Science* (Vol. 146, Chap. 5, pp. 109–139). New York: Springer.

Reeves, C., & Rowe, J. (2003). *Genetic algorithms: Principles and perspectives: A guide to GA*. Boston: Kluwer Academic.