Applications of Evolutionary Algorithms in Scheduling, Planning and Timetabling

Norbert Logiewa
nl253

Evolutionary algorithms (EA) are a class of procedures that are used when finding a solution to a problem using regular means is not possible or very difficult. EAs are often employed when dealing with multi-dimensional problems where sampling of every candidate is not possible [2, p. 29]. This essay will discuss how GA can be applied to scheduling. *Scheduling* and *timetabling* in this essay will refer to ordering of tasks that aims to maximise the productivity. The ordering may schedule some activities to take place in parallel if business rules and the amount of resources allow for that. Efficient scheduling of tasks if of interests to anyone who wishes to increase their productivity *without* investing additional resources. EAs allow to find good-enough solution to problems that don't necessarily need an *optimal* solution[4, p. 44]. This is often because the search space is too large to be directly traversed so brute-force methods are not possible.

Case Study – University

A hypothetical university may wish to improve the productivity of students and lecturers by using an EA for timetabling. Suppose the university offers 200 modules to 16000 students. Each year every student must take 8 unique modules. Each module needs to have 2 one-hour lectures during the working week. Additionally, each of those lectures or can take place in one of 50 lecture halls. Moreover, each of those lectures may be given by a lecturer from the relevant department. Finally, each lecture or may take place a time between 9am and 7pm. The task would be to ensure that: every student and lecturer is in at most one lecture at a time, only one lecture takes place in a lecture hall at the same time, etc. A timetable i.e. a candidate solution that obeys all of these business rules would be favoured in the process of natural selection and would receive a higher fitness score as evaluated by the fitness function.

Solving the Problem

GA is well suited for the task because there is likely many good-enough solutions to the problem i.e. many timetables could satisfy the constraints listed above. In other words, the fitness landscape has many peaks, hence, using a global-search algorithm is beneficial. Furthermore, the complexity of the problem (large search space and multiple dimensions) means that a brute force search in this multi-dimensional space would take too much time.

Phenotype and Genotype

Each candidate solution (timetable) would be encoded as a bit-string [1] representing all the lectures along with time, place and the lecturer taking place during the week. This choice of bitstring encoding for candidate representation is motivated by the fact that this encoding has been extensively discussed in the literature on GA [3, p. 103]. Bitstrings are the most common encoding [1, p. 127] possibly because this low-level representation uses less resources and operators such as cross-over mutation invented by researchers tend to behave well when we encode solutions in this manner. The crossover operator, for instance, is extremely easy to apply to a bit-string but less easy to other data structures such as sets. In this scenario we allocate a chunk of bits for every lecture that must take place. In every sub-string representing a lecture the first bit determines whether the lecture takes place in the first or the second semester. The following m bits encode an integer index in the modules array. The next l bits encode an integer index in a lecturer array. The following t bits encode an integer referring to the time that lecture will take place. The last h bits encode an integer index in a lecture halls array. Hence each lecture is be a bit-string of length $1+m+l+t+h$ and each candidate is be a concatenation of 400 of these giving us in total a bit-string of length $400(1+m+l+t+h)$. If we delegate, say, 16 bits to representing the time and 8 for each indexing integer, we end up with each candidate having a size of $400 * (1 + 8 + 8 + 8 + 16)\ bits = 16400\ bits = 2050\ Bytes = approx.\ 2\ KB$. It's useful to bear in mind the size of each candidate when deciding on the population size. Some researchers suggest population sizes of 1000 [4, p. 25]. This would mean our application would require $2\ KB * 1000 = 2000\ KB = 2\ MB$ memory. The *phenotype* might be a list of lectures, for instance, which we would decode from the *genotype* (bitstring).

Selection Operator

The GA literature discusses a number of operators to be used for selection: proportionate, rank-based and tournament among others [2, 3, 4]. A good choice of selection operator would be rank-based selection (RBS). The reason for this is that in this scenario the exact fitness value of an individual is irrelevant. It's sufficient to know that a candidate is better than others for it to make it into the next gene-pool. Moreover, It's not clear that other more sophisticated selection methods such as tournament selection would yield better results. Furthermore, RBS avoids issues that proportionate selection faces: when all candidates have a similar fitness or one candidate has a large high fitness score, the selection turns into a random search. [2, p. 23] Lastly, the use of RBS simplifies implementation and is in line with the principle of parsimony.

[1]this is the genotype and is analogous to DNA

Genetic Operators & Adaptive GA

Application of a selection operator reduces the size of the population. To replenish it, pairs of winners are used as input to the crossover operator which creates new individuals. Crossover is used because it is a standard and well-understood operator. Additionally, mutation is introduced with a small probability to further increase variation in the population. These two tend to be combined because cross-over *recombines* existing genes whereas mutation introduces *novelty*. [4, p. 27]. A good starting values for the parameters would be as follows: $p(crossover) = 0.8$ [3, p. 117] and $p(mutation) = 0.2$ [4, p. 25] although some researchers suggest lower rates e.g. 0.005 [3, p. 117].

However, the crossover operator is problematic. In the initial phase of looking for a solution across the fitness landscape it is useful to vary candidates a lot so using crossover is highly beneficial. However, once a satisfactory candidate has been found, we might only wish to tweak it a bit. Altering the candidate too much would be equivalent to abandoning the progress that has been made to reach this fitness score. Hence, a less "invasive" and more fine-grained operator such as mutation [2, p. 28] may become more advantageous as the number of iterations grows. In a bit-string a mutation would flip a bit which may, for example, increment an integer which represents a lecturer thus making a change in a candidate. The benefit of using crossover, arguably, diminishes with time. To implement that we might increase the probability of mutation and increase the probability of cross-over at the beginning of every iteration until p reaches 0 for cross-over and until p reaches 1 for mutation.

An alternative way of preserving the fittest candidates in the next population would be to use elitism. This would protect against degrading of solutions when cross-over is applied to fittest candidates. [2, p. 26]

However, this enhancement may not be necessary. Norvig [1, p. 128] points out that when the algorithm begins to converge on a solution after being run for a while, the candidates in our population are not very diverse. This means, in theory, that the cross-over operator would take "smaller steps" because of how similar candidates are.

The fitness function

The fitness function deciphers each of the 400 sub-strings within each candidate bit-string and assigns a score to each candidate which reflects the extent to which it obeys the business rules. [2, p. 22] In pseudo (python-like) code:

```
def evalFitness(candidate: BitStr, modulesTaken: MultiSet[FrozenSet[int]]) -> int:
    """candidate is a concatenation of all lectures."""
    sBits  = 1  # 0 == semester 1 else semester 2
    mdBits = 8  # bits encoding the module 2^8 = 256
    mBits  = 4  # bits encoding hours since 8am 2^4 = 16
    hBits  = 6  # bits encoding minutes 2^6 = 64
    dBits  = 3  # bits encoding day of the week 2^3 = 8
    lBits  = 8  # bits encoding int index in lecturers array 2^8 = 256
    lhBits = 8  # bits encoding int index in halls array 2^8 = 256
    subStrSize = sBits + mBits + hBits + dBits + lBits + lhBits + mdBits
    totalFitness = 0
    for lecture in candidate.split(subStrSize):
        fitness = 1000000  # initial fitness
        time = decodeTime(lecture)
        semester = lecture[0]
        lecturer = decodeLecturer(lecture)
        hall = decodeHall(lecture)
        module = decodeModule(lecture)
        if crossDepartament(lecturer, module):
            fitness -= 100
        for lecture2 in candidate.split(subStrSize):
            time2 = decodeTime(lecture2)
            semester2 = lecture2[0]
            lecturer2 = decodeLecturer(lecture2)
            hall2 = decodeHall(lecture2)
            if semester == semester2:
                if timesOverlap(time, time2):
                    if lecturer == lecturer2:
                        fitness -= 100
                    if hall == hall2:
```

```
                    fitness -= 100
                for modules in modulesTaken.keys():
                    if module in modules:
                        noEnrollments = modulesTaken[module]
                        fitness -= noEnrollments * 100
        totalFitness += fitness
    return totalFitness
```

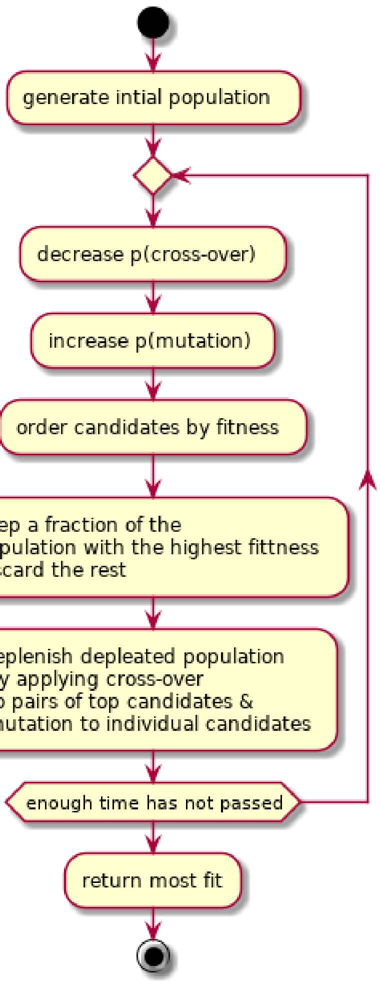
The pseudocode illustrates that the fitness function has a bad $O(l^2 * c)$ complexity where: l = number of lectures [2] for all modules and c = number of combinations of modules. Initially it may seem that the fitness function is not computationally feasible. With 200 modules and 8 modules to be takes for every student the maximum number of combinations is $8C200 = 55098996177225$. However, there are bound to be restrictions on the combinations of modules. For example, a computer science student is unlikely to be allowed to take "Introduction to Criminology". In fact, we would expect each student to only be able to choose from around 15 modules from their own department. $8C15 = 6435$ is a much more manageable number. [3]

The invariant

GA is run repeated until either enough time has passed or other condition has been met. [3, p. 108]. In this scenario a good choice would be to wait for a high enough fitness value in a candidate but to also introduce a timeout of, say, 10 minutes in case such a candidate cannot be found.

Summary

The basic idea behind this particular version of GA is described by the following UML activity diagram:


generate intial population
decrease p(cross-over)
increase p(mutation)
order candidates by fitness
keep a fraction of the population with the highest fittness discard the rest
replenish depleated population by applying cross-over to pairs of top candidates & mutation to individual candidates
enough time has not passed
return most fit

This essay demonstrated how EA can be used to aid timetabling and scheduling. It illustrated how to apply EA using a case study of a university, it discussed how to encode the candidate timetables and stated how carry out the algorithm in this scenario.

[2]in this scenario $l = 200 * 2$

[3]the pseudo-code assumes that module configurations for all students are presented in a multiset where every key is itself a set of modules (similar to frozen sets in Python, see https://docs.python.org/3/library/stdtypes.htmlfrozenset) and value is the number of students who chose this particular combination of modules

References

[1] Stuart Russel and Peter Norvig, *Artificial Intelligence*, *A MODERN APPROACH*, p. 126-129, 3rd Ed., Pearson Education 2010.

[2] Dario Floreano and Claudio Mattiussi, *Bio-Inspired Artificial Intelligence*, *THEORIES, METHODS, AND TECHNOLOGIES*, p. 1-38, MIT Press, 2008.

[3] Russel C. Eberhart, *Computational Intelligence*, *Concepts to Implementations*, p. 103-118, p. 51-68, Denise E.M. Penrose, 2007.

[4] Jeff Heaton, *Artificial Intelligence for Humans*, *Volume 2: Nature Inspired Algorithms*, p. 1-100, Heaton Research, Inc, 2014.