Genetic Algorithms: application to cancer treatment

Optimization

Nieves Montes Gómez (1393150)

January 17, 2020

1 Introduction

The objective of this assignment is to implement a Genetic Algorithm (GA) to find a curative treatment for cancer, given an initial number of tumor cells. If that is not possible, then the same algorithm is employed to find a palliative treatment. This problem is very suitable for GAs, which do not ensure global optimization. However, a suboptimal curative treatment, despite taking longer than desirable, is still a solution to the patient's health problem. The framework for the problem is provided in example 5.3 of the paper by John McCall [1]. The tumor is modeled by the Gompertz growth model with linear cell-loss effect. The concrete parameter values to be used are provided in the proposal for the assignment.

The code is organized into the files `RKF78.h`, `RKF78.c`, `main.c`, `GA_functions.c` and `auxiliary.c`. The first two have been provided and are necessary for the resolution of the Gompertz model ordinary differential equation. `main.c` has little interest since it only contains the call to the `GeneticAlgorithm()` function, which actually implements the solution. The most interesting file containing the GA-related functions is `GA_functions.c`, which will be the focus of the discussion in the next section. Finally, the provided values for the many parameters, the functions to compute the fitness score and other auxiliary ones are included in the file `auxiliary.c`. They are discussed at the end of section 2.

2 Implementation of the Genetic Algorithm

2.1 Representation of candidates and random data generation

Candidate solutions are represented by the following C structure, which contains the information of the treatment dosages and the corresponding fitness score:

```
typedef struct {
  unsigned char Cij[(n_par-1)*d_par];
  double fitness;
} individual;
```

Despite only needing 4 bits for each drug concentration, the smallest data type (`unsigned char`) contains 8 bits. Only the first four bits in each `Cij` element are relevant. This requires some adjustments in the computation of the fitness as well as in the genetic functions.

The current population and next generation being considered by the algorithm is then a vector of `individual` structures. The population size of choice (`popsize`) is 40. In order to randomly initialize the population, random bits are read from the system file `/dev/urandom` into the `Cij` arrays. This device is also employed in the `uniform()` function, which returns a true random number between 0 and 1. This function is necessary for the genetic-related functions, which are introduced in the following sections:

```
double uniform() {
  int randomData = open("/dev/urandom", O_RDONLY);
```

```
  if (randomData < 0) ExitError("Could not open /dev/urandom", 1);
  unsigned char num;
  ssize_t result = read(randomData, &num, 1);
  if (!result) ExitError("could not read from urandom", 1);
  double randNum = (double)num / 255.0;
  close(randomData);
  return randNum;
}
```

2.2 Genetic functions

There are three main steps in GAs, each of which can be performed through a wide range of different algorithms. These are the choices for this implementation:

- For the Selection With Replacement step, I have chosen one vs one tournament selection, due to its simplicity and insensitivity to the features of the fitness function. The function responsible for this step is `TournamentSelection()`, which takes in a pointer to a vector of individuals (the population) randomly selects two different individuals and returns a pointer to the fittest:

```
individual* TournamentSelection(individual* population) {
  int cont1 = uniform() * (popsize - 1), cont2 = uniform() * (popsize -
 1);
  while (cont2 == cont1) cont2 = uniform() * (popsize - 1);
  if (population[cont1].fitness < population[cont2].fitness) return (
population + cont1);
  return (population + cont2);
}
```

- For the Mutate step, instead of using classical bit-flip mutation, a more efficient version has been implemented. Standard bit-flip mutation with probability $1/L$, where L is number of bits encoding a candidate, is expected to introduce one mutation per individual. In the cancer treatment problem, one mutation is always introduced to every offspring. The mutation position is selected randomly. This approach avoids scanning all bit positions and the evaluation of many conditional statements. It is implemented in the `Mutation()` function, which takes in a pointer to an individual and modifies it:

```
void Mutation(individual* candidate) {
  int overallPos = (4*(n_par-1)*d_par - 1) * uniform();
  int i = overallPos/4;
  int j = overallPos%4;
  candidate->Cij[i] = (candidate->Cij[i])^(1U << (7-j));
}
```

- For the Breeding step, one-point crossover is the method of choice. The bit crossover position is selected randomly. This is deemed to be more adequate than two-point crossover because it is simpler, and due to the sequential nature of the treatment encoded in `individual.Cij`, it is difficult that the first and last of its elements are linked to produce a very good treatment. This step is performed by the `OnePointCrossover()` function, which takes in pointers to the parents and children individuals:

```
void OnePointCrossover(individual* parent1, individual* parent2,
individual* child1, individual* child2) {
  unsigned short bitsPerInd = 4 * (n_par-1) * d_par;
  unsigned short overallPos = uniform() * (bitsPerInd - 1);
  unsigned short c = overallPos/4;
  unsigned short mask = 0xFFFFFFFFU << (8 - overallPos%4);
```

```
    for (int i = 0; i < c; i++) { child1->Cij[i] = parent1->Cij[i];
  child2->Cij[i] = parent2->Cij[i]; }
    for (int i = c+1; i < (n_par-1)*d_par; i++) { child1->Cij[i] =
  parent2->Cij[i]; child2->Cij[i] = parent1->Cij[i]; }
    unsigned char p1 = parent1->Cij[c], p2 = parent2->Cij[c];
    child1->Cij[c] = (p1 & mask) | (p2 & ~mask);
    child2->Cij[c] = (p2 & mask) | (p1 & ~mask);
  }
```

Since only the first four bits of each component in `individual.Cij` are relevant, the computation of the indices to introduce a mutation and to cross the individuals is not as straighforward as usual, when all bits are utilized to encode information.

2.3 `GeneticAlgorithm()` function

The most important function, and the one that actually solves the problem, is `GeneticAlgorithm()`. It does not take in any variable nor returns anything. It starts by allocating memory for the population (a vector of `individual` structures), randomly initializing them, assigns their curative fitness scores and finds the fittest individual (that with the *lowest* score). To do so, the `find_fittest()` function is called. It returns a pointer to the provisional solution, however in order not to lose its information through the algorithm iterations, the content of this pointer must be copied and stored in a separate variables (`individual fittestOverall`). Following is the definition of the `find_fittest()` function:

```
individual* find_fittest(individual* population) {
  individual* fittest = population;
  for (int i = 1; i < popsize; i++) if (population[i].fitness < fittest->
 fitness) fittest = population + i;
  return fittest;
}
```

The `GeneticAlgorithms()` function then performs the main loop of GA using the curative score as the fitness of individuals:

```
/* MAIN LOOP TO FIND A CURATIVE TREATMENT. */
printf("... Applying GA to find a curative treatment ...\n");

int max_iter = 4, iter = 0;
if (fittestOverall.fitness == DBL_MAX) max_iter = 10;

while (iter < max_iter) {
  individual * nextGen = NULL;
  if ((nextGen = (individual*) malloc(popsize*sizeof(individual))) == NULL)
 ExitError("could not allocate memory for the next generation", 1);
  /* make a new generation */
  for (int j = 0; j < popsize/2; j++) {
    individual * parent1 = TournamentSelection(population);
    individual * parent2 = TournamentSelection(population);
    OnePointCrossover(parent1, parent2, nextGen + 2*j, nextGen + 2*j + 1);
    Mutation(nextGen + 2*j); Mutation(nextGen + 2*j + 1);
    nextGen[2*j].fitness = Curative_Fitness(nextGen[2*j].Cij);
    nextGen[2*j+1].fitness = Curative_Fitness(nextGen[2*j+1].Cij);
  }
  /* replace previous population with new generation */
  individual * aux = population;
  population = nextGen;
  free(aux);
  /* find fittest inidividual in new population */
```

```
    fittestInCurPop = find_fittest(population);
    if (fittestInCurPop->fitness < fittestOverall.fitness) {
      for(int i = 0; i < (n_par-1)*d_par; i++) fittestOverall.Cij[i] =
   fittestInCurPop->Cij[i];
      fittestOverall.fitness = fittestInCurPop->fitness;
      max_iter = iter + 5;
    }
    iter++ ;
  }
```

The loop stops when the candidate solution has not been updated for four consecutive iterations. However, if the starting population consists of all unfeasible inidividuals (who are not cured by the treatment and have `fitness=DBL_MAX`), 10 iterations without an update are allowed.

If the algorithm is succesful in finding a curative treatment, it prints it, calls a function to write the evolution of $N(t)$ in a file, and exits the program. If not, it moves on to try and find a palliative treatment. It recomputes the fitnesses of individuals as the inverse of the lifespan and finds the new fittest candidate. Then, it enters into a loop that is identical to the one for finding a curative treatment. The only difference is the function call for computing the fitness scores.

Finally, if a succesful palliative treatment has been found, it is printed and $N(t)$ written on a file. If not, the program informs the user that it has not been able to find a feasible solution.

2.4 Fitness computation and other auxiliary functions

This subsection deals with the functions included in the `auxiliary.c` file. It includes the necessary functions to compute the fitness scores, as well as a few other auxiliary functions.

First, the `Curative_Fitness()` and `Paliative_Fitness()` functions compute the inverse of $\int N(t)dt$ and the inverse of the lifespan respectively. Before solving the Gompertz model differential equation, they check constraints 2 and 3. There are two versions of this checking since curative and palliative treatments use different parameters of η_{kj}. All the functions that make use of the values of `Cij` have to be shifted by four bit positions when doing the calculations, since we have set that only the first four (out of the total eight) are relevant. This ensure that the actual values of concetrations being used are between 0 and 15, as required by the problem. Regarding the computation of the palliative fitness score, the lifespan of the patient is considered beyond the end of the treatment, if by the end of it the tumour cell levels are still below the critical values.

The two other auxiliary functions are `setExcessDosagesToZero()` and `writeResult()`. The last one is responsible for writing $N(t)$ data into a file when a suitable treatment has been found. The first one is also called when the GA has produced a feasible solution. After the patient has been cured or has died, it sets the dosages of later sessions to zero, as they are not relevant anymore. Note that this step is performed once a solution has been found, and not during the main GA loop, to preserve the genetic variability of the population.

3 Results

In this section, example outputs for each possible situation are displayed. The computed evolution of the number of tumour cells $N(t)$ is shown in Figures 1 and 2 for the found curative and paliative treatments respectively.

An example of the program output when a curative treatment has been found is the following:

```
... Applying GA to find a curative treatment ...

Found a solution for a curative treatment.

Recommended dosages:
Session 1: 15 14 11 14 14 10 14 12 15 11
Session 2:  4  6 11 15 13  2  3 13 13 15
Session 3:  0 13 11 15  4 13 13  4 15  2
Session 4:  0  0  0  0  0  0  0  0  0  0
Session 5:  0  0  0  0  0  0  0  0  0  0
Session 6:  0  0  0  0  0  0  0  0  0  0
Session 7:  0  0  0  0  0  0  0  0  0  0
Session 8:  0  0  0  0  0  0  0  0  0  0
Session 9:  0  0  0  0  0  0  0  0  0  0
Session 10:  0  0  0  0  0  0  0  0  0  0

Integral N(t)dt = 4180
```

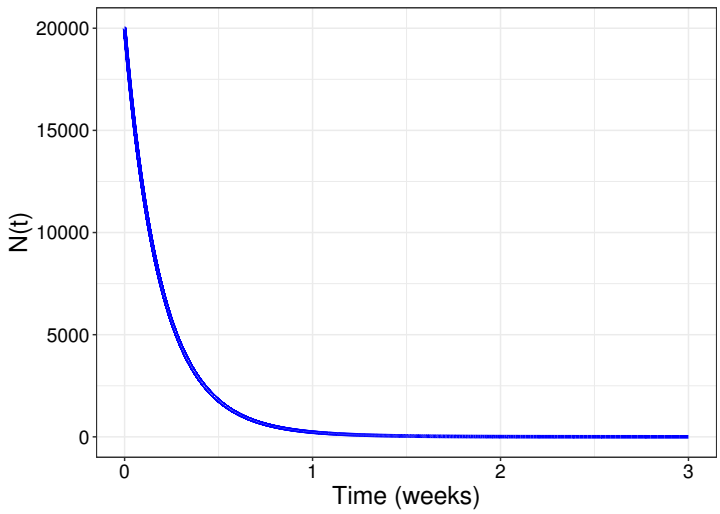

Figure 1: Evolution of the number of tumor cells with time for the displayed curative treatment. The time scale only reaches the end of the first treatment session that reduces the number of tumor cells below the recovery threshold (1000).

An example of the program output when a palliative treatment has been found is the following:

```
... Applying GA to find a curative treatment ...

A feasible curative treatment could not be found.

... Applying GA to find a paliative treatment ...

Found a solution for a paliative treatment.

Recommended dosages:
Session 1:  1  4 15 12  2 11  9  0  8  5
Session 2:  3 13 10 15 12  9 10  3  2  1
Session 3:  1  2 15  8 15  5  6  1  1  9
Session 4: 12  1  0  5  3 15  3 10 10  1
Session 5:  1 11  9  2  4  5  5 11 11  9
Session 6:  7 11  4  0  3 11  2  5  9 13
Session 7: 12  4  0  9  5 13  7  7 10  0
Session 8:  5  0  2  1 10 11  8  2 13  7
Session 9:  6  7  7  0  1  4  5  8  0  5
Session 10:  6 12 11  6  5  1 13 10  6  4

Expected lifespan: 50.4 weeks.
```

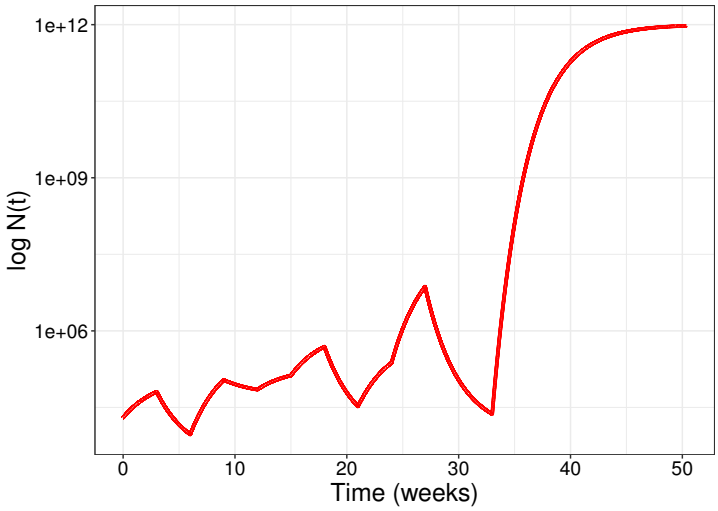

Figure 2: Evolution of the number of tumor cells with time for the displayed palliative treatment. A logarithmic transformation has been used for visualization purposes.

An example of a run when neither a curative nor a palliative treatment has been found:

```
... Applying GA to find a curative treatment ...

A feasible curative treatment could not be found.

... Applying GA to find a paliative treatment ...

A feasible paliative treatment could not be found.
```

4 Conclusions

In this assignment, a version of Genetic Algorithms has been implemented to find a curative or palliative drug treatment for a malign tumour whenever possible. This situation is very suitable for GAs, since a suboptimal solution is already a very important finding. The genetic functions of choice are the standard ones (one-point crossover, tournament selection), except for the mutation operation, that consistently introduces one mutation into each new individual, in order to favour the explorability of the search. Related to this point, excess dossages after the patient is cured or has died are set to zero only after the solution is found, to preserve the genetic variablity of the pool.

In conclusion, a Genetic Algorithm has been used to solve a difficult problem where a numerical method had to be employed to compute the fitness function and a suboptimal result is still a potential solution to this severe health problem.

References

[1] John McCall. "Genetic algorithms for modelling and optimisation". In: *Journal of Computational and Applied Mathematics* 184.1 (2005), pp. 205–222. ISSN: 03770427. DOI: 10.1016/j.cam.2004.07.034.