[AM-07-078] Genetic Algorithms and Evolutionary Computing

Abstract

Genetic algorithms (GAs) are a family of search methods that have been shown to be effective in finding optimal or near-optimal solutions to a wide range of optimization problems. A GA maintains a population of solutions to the problem being solved and uses crossover, mutation, and selection operations to iteratively modify them. As the population evolves across generations, better solutions are created and inferior ones are selectively discarded. GAs usually run for a fixed number of iterations (generations) or until further improvements do not obtain. This contribution discusses the fundamental principles of genetic algorithms and uses Python code to illustrate how GAs can be developed for both numerical and spatial optimization problems. Computational experiments are used to demonstrate the effectiveness of GAs and to illustrate some nuances in GA design.

Keywords: evolutionary computation, GA, genetic algorithms, heuristic search, location allocation modelling, optimization

Author & citation

Xiao, N. and Armstrong, M. P. (2020). Genetic Algorithms and Evolutionary Computing. The Geographic Information Science & Technology Body of Knowledge (1st Quarter 2020 Edition), John P. Wilson (ed.). DOI:10.22224/gistbok/2020.1.1.

This Topic is also available in the following editions:

1. DiBiase, D., DeMers, M., Johnson, A., Kemp, K., Luck, A. T., Plewe, B., and Wentz, E. (2006). GA and global solutions. The Geographic Information Science & Technology Body of Knowledge. Washington, DC: Association of American Geographers. (2nd Quarter 2016, first digital).
2. DiBiase, D., DeMers, M., Johnson, A., Kemp, K., Luck, A. T., Plewe, B., and Wentz, E. (2006). Genetic algorithms and artificial genomes. The Geographic Information Science & Technology Body of Knowledge. Washington, DC: Association of American Geographers. (2nd Quarter 2016, first digital).

Explanation

1. Introduction

Problems come in all shapes and sizes. Some problems are easy to solve, while others are

far more difficult. Some problems can be analyzed qualitatively. Other problems require that computational methods be applied to derive a solution. This paper is concerned with the latter case, and in particular, those problems with discrete components that require search through many alternatives in order to find either the best solution or a class of near-best solutions. This class of problems can be solved by brute force enumeration for small problem sizes. Larger problems are a different matter. One combinatorial optimization problem, p-median, which selects p best locations from n candidate locations and is described in more detail in Section 3 below, requires that $\frac{n!}{(n-p)!p!}$ alternatives be evaluated. For a problem where n=10 and p=3, only 120 alternatives need to be evaluated. But when n and p are increased to 100 and 15, respectively, the number of alternatives increases to 253,338,471,349,988,640.

Because of this combinatorial complexity, the solution process for realistic problems requires excessive amounts of computer time and memory, thus restricting the size of the problem that can be addressed. In response to this shortcoming, a wide range of methods, called heuristics, have been developed to reduce search requirements. Genetic algorithms (Goldberg, 1989) are a family of heuristic search methods that can be used to solve large combinatorial optimization problems. The purpose of this paper is to provide a brief perspective on heuristic algorithms and then to introduce the basic concepts that are employed to implement genetic algorithms using two example programs written in Python.

To begin, consider a simple example heuristic. It is assumed that the universe of solutions takes the form of a continuous parabolic hill, as shown in Figure 1, and that the maximum height of the hill (maximum y) needs to be found. A classical heuristic method called hill climbing can be used: a start is made from a random initial position along the x-axis, the y-value at that location is evaluated, and is declared to be the current optimum. Then the algorithm moves away from this start point along the x-axis and a determination is made about improvement (increase) in y (yes or no). If yes, the current y-value replaces the old value and becomes a candidate for the maximum y. This process is repeated until no improvement can be found. While this gradient method works for simple examples, many problems have more complex forms and present a challenge to most heuristics: how to escape local optimal solutions (Figure 2) during the search process. This can be addressed by randomly re-starting the search process at a number of different x-locations in the hope that search will be initiated on the highest peak. However, unlike humans that can see the form of the curve, these algorithms operate without “vision,” and they can never be sure to find the global maximum. Genetic algorithms are designed to escape local optima and achieve the objective of finding the global optimal solution.

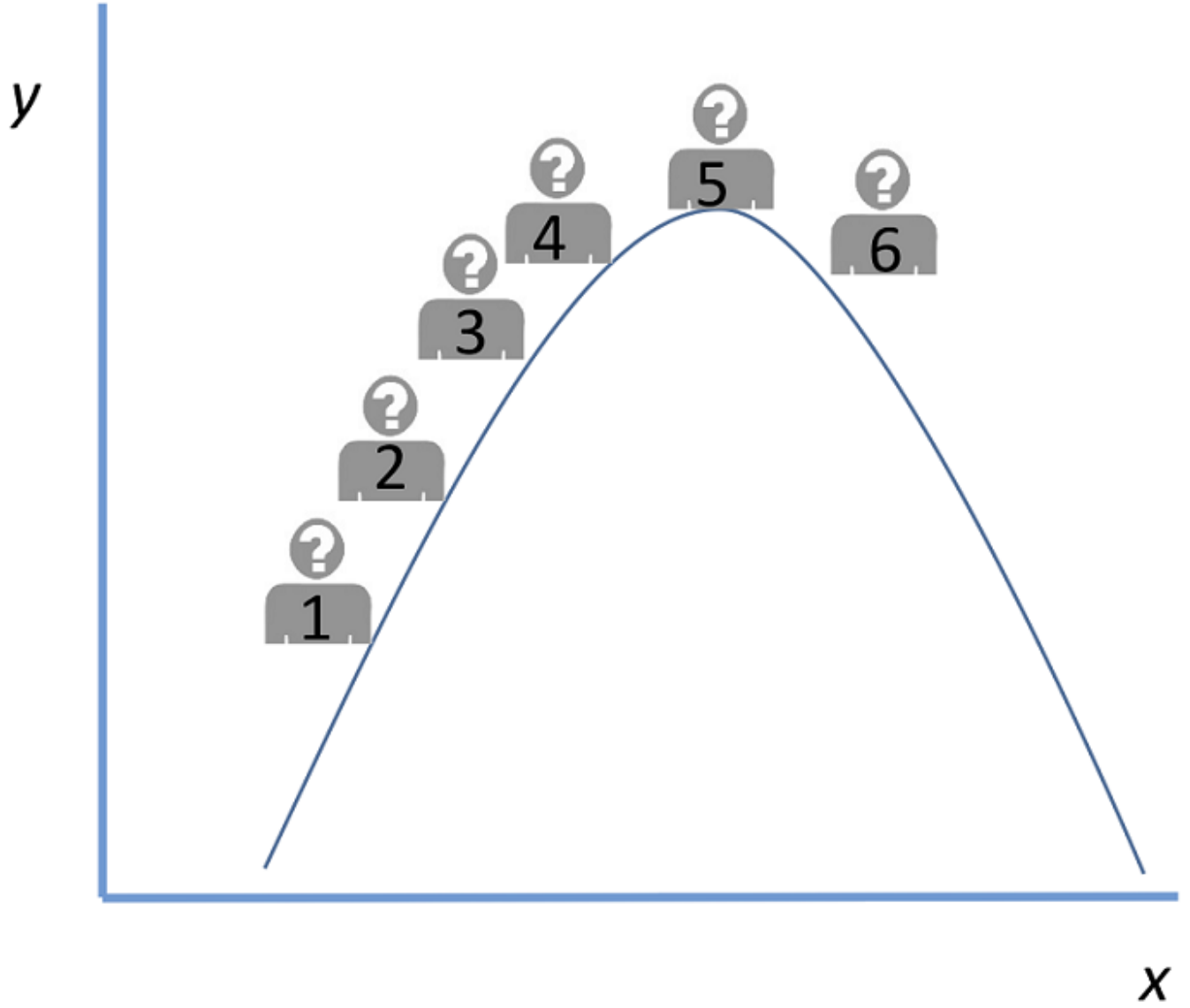

Figure 1. In this simplistic hill climbing example, a random position is generated (1). A change is made (2) and if the result is higher, it becomes the current optimum. The process (change and evaluate) is repeated until step 6 which is lower than step 5, at which time step 5, the former highest position in the process, is declared to be the maximum (peak) y-value. Source: authors.

A genetic algorithm (GA) is formed by generating a group of individual solutions that together are called a population. Individuals in the population are randomly initialized. Each individual solution is evaluated by computing its fitness value, which is often based on an objective function, such as the y-value in the previous example. Individuals in the population repeatedly go through a reproduction process in which individuals with high fitness values are likely to be used to create new offspring.

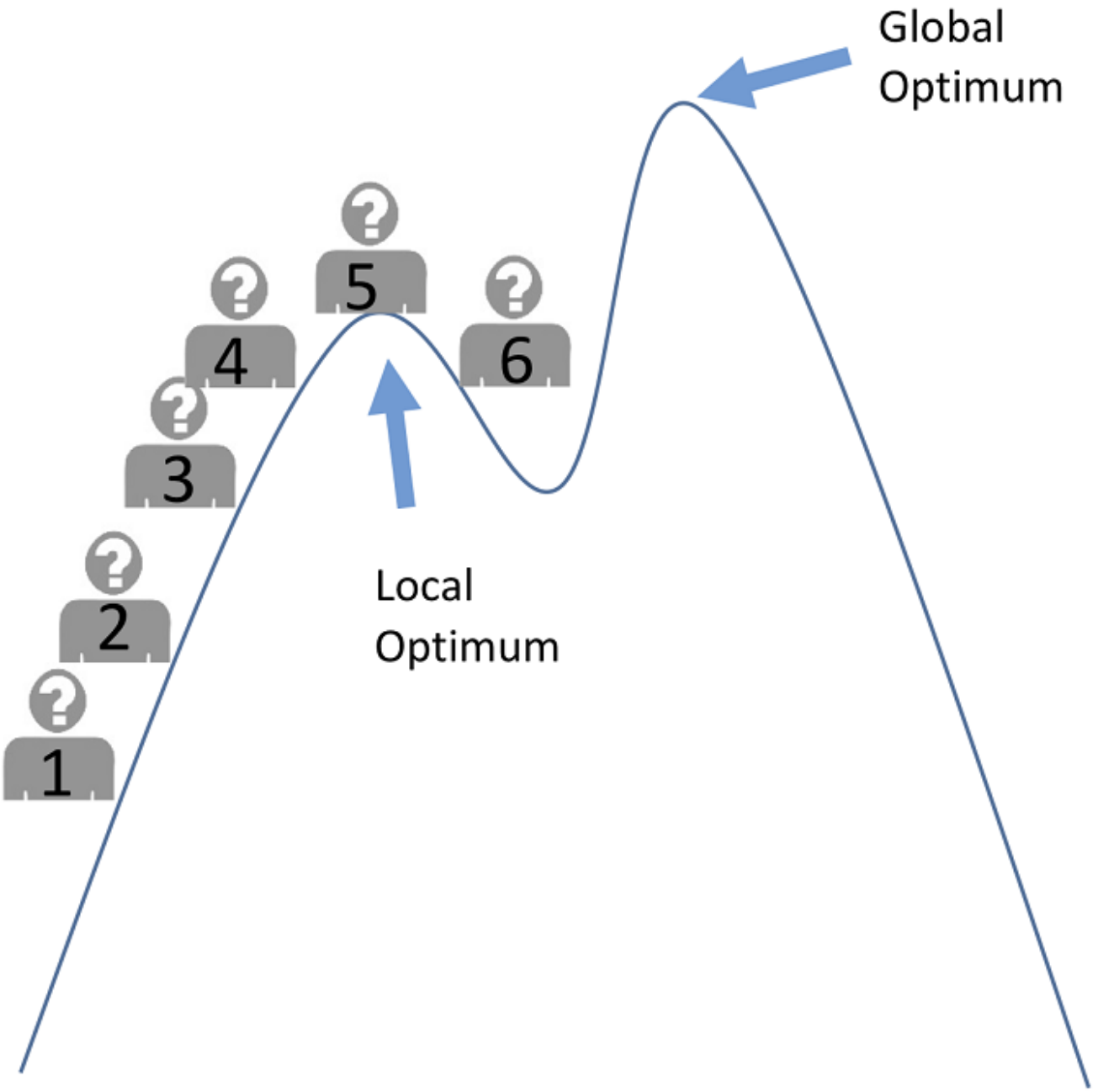

Figure 2. Hill climbing approaches can be fooled, if the hill increases non-monotonically. The solution process would find the local optimum, but fail to find the global optimum. Source: authors.

In implementation, the reproduction process iterates and during each iteration, a selection operation is used to choose individuals from the current population, where individuals with high fitness values have a high likelihood to be selected. A crossover operation is used to recombine the values in the selected parent solutions in order to create their offspring. A small number of the offspring will also go through a mutation operation that will randomly change the values in the solution. New individuals generated in this way are then used to replace the current population and the process continues until a user-specified maximum number of generations is reached. Figure 3 illustrates the iterative process of a GA.

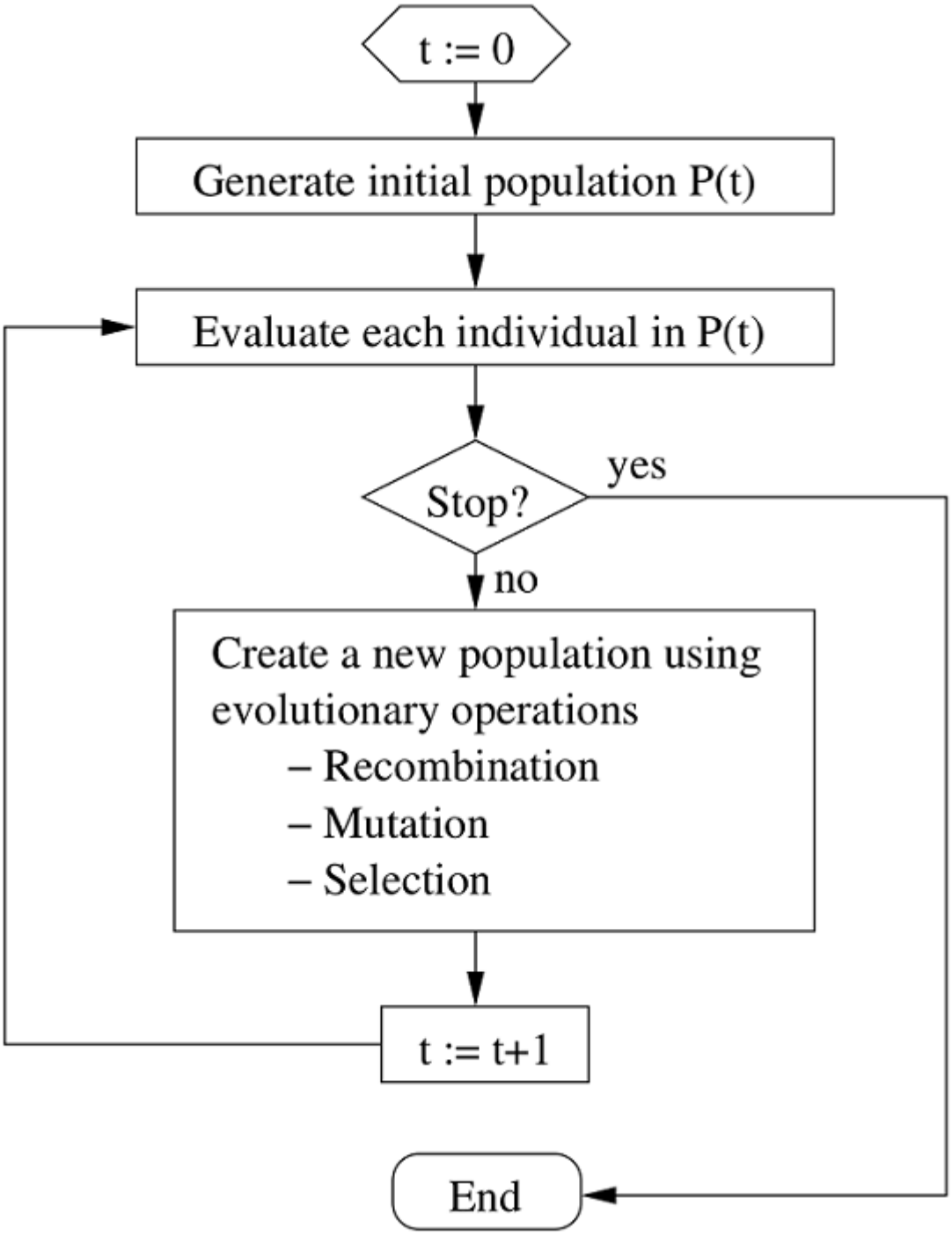

Figure 3. General procedure of a genetic algorithm. Source: authors.

In the remainder of this contribution, we discuss elements of GAs using simple example optimization problems. We use Python programs to demonstrate how GAs are designed and used for these problems. All the code used here are available in the companion Github site at https://github.com/ncxiao/genetic-algorithms.

2. Elements of Genetic Algorithms

Let us start by considering a numerical optimization problem with the following objective function of three decision variables:

max $x_1^2 - x_2 - x_3 + 1$

where x1, x2, and x3 are integers in the range of 0 and 7. This is a simple problem where the maximum value is 50 and the optimal solution is x1 = 7, x2 = 0, x3 = 0.

2.1 Encoding and Decoding

To represent a solution to the problem, we encode each base10 integer using a binary string of 3 digits. For example, a binary string of 010 represents a value of 2. Together, a binary string of 9 digits (three groups of three) can be used to represent a solution. Thus, a string of 110010001 represents the following values: 6 (110), 2 (010), and 1 (001) for variables x1, x2, and x3, respectively.

With this encoding strategy in place, there is now a need to decode the binary string after we calculate the objective function value of a binary encoded solution. The following Python function decodes a 9-digit binary string, and returns a tuple of three integers.

```
def decode(s):
    def _decode(s):
        num = 0
        n = len(s)
        for i in range(n):
            c = s[n-i-1]
            num += int(c) * 2 ** i
        return num
    return  decode(s[:3]),  decode(s[3:6]),  decode(s[6:])
```

Listing 1. Decoding a binary string.

The following is a test of the decoding function on an example binary string.

```
>>> decode('110010001')
(6, 2, 1)
```

2.2 Generating Random Solutions

Using this encoding strategy, we can now design a method to generate random solutions to the problem. For the specific problem used here, an initial solution can be created by randomly assigning 0 or 1 to each of the 9 digits in the binary string. To do so, we first introduce a useful function called flip (Listing 2) that returns a True value as a Bernoulli random variable with a given probability.

```
from random import random, randint, sample, uniform
def flip(prob):
    if random() < prob:
        return True
    return False
```

Listing 2. The flip function.

For example, the following code tests how many True values will be returned when the probability is set to 0.9. The count should be close to 9.

```
>>> vals = [flip(0.9) for _ in range(10)]
>>> print(vals)
>>> print(sum(vals))
[True, True, True, True, True, True, True, True, True, False]
9
```

The following code shows how to generate random solutions to the above optimization and the use of the decode function. Placing the code in an appropriate looping structure generates a population of solutions in base 2 that are then decoded into base 10.

```
>>> ''.join(['1' if flip(0.5) else '0' for i in range(9)])
'100111110'
>>> s = ''.join(['1' if flip(0.5) else '0' for i in range(9)])
>>> print(s)
011100110
>>> decode(s)
(3, 4, 6)
```

2.3 Evaluation

There are two key issues in evaluating a given solution. First, the objective function of the problem is implemented and then the specific objective function value for each solution must be calculated. The obj function in Listing 3 computes the objective function values given a tuple of three values:

```
def obj(x):
    return x[0] ** 2 - x[1] - x[2] + 1
```

Listing 3. Calculating objective function value.

And we can test it as follows:

```
>>> obj(decode('110010001'))
34
```

Objective function values, however, are normally not used directly in a GA. Instead, they are transformed into fitness values that indicate how close an individual solution is to the optimal solution. A higher fitness value is always more desirable than a lower one, and fitness values should also be non-negative.

The function obj2fitness (Listing 4) does this conversion. More specifically, the obj2fitness function takes a list of objective function values (objs) as its input and returns a transformed list of fitness values. All objective function values are linearly shifted so that the lowest value will become non-negative. Also, we do not want the lowest value to be

zero, because individuals with zero fitness will not have any chance to be selected later in the GA process. In this example, 10 percent of the range between the highest and lowest objective function value serves as the minimum fitness value. This is an arbitrary quantity, but it is effective in making the implementation work correctly.

```
1 def obj2fitness(objs):
2     low = min(objs)
3     minimal = (max(objs) - low) * 0.1
4     fitnesses = [val - low + minimal for val in objs]
5     return fitnesses
```

Listing 4. Converting objective function values to fitness values.

2.4 Initialization

The next step is to generate the initial population used by the GA. The initialization function (Listing 5) generates an initial population of solutions and evaluates them. It returns three lists: randomly generated solutions to the problem, the objective function values of the random solutions, and the transformed fitness values, respectively. Here, each (initial) solution is a string of 0's and 1's, where each digit is randomly decided using the flip function with a probability of 0.5 (Lines 4 and 5).

```
1 def initialization(popsize):
2     population = []
3     for i in range(popsize):
4         sol = ''.join(['1' if flip(0.5) else '0'
5                        for i in range(9)])
6         population.append(sol)
7     objs = [obj(decode(s)) for s in population]
8     fitnesses = obj2fitness(objs)
9     return population, objs, fitnesses
```

Listing 5. Initialization.

The following is an example of generating an initial population of 2 random solutions with their binary representations along with their objective function and transformed fitness values:

```
1 >>> initialization(2)
2 (['000011010', '011100101'], [-4, 1], [0.5, 5.5])
```

2.5 Selection

Given a population of individual solutions and their fitness values, a selection operation can be defined so that those individuals with high fitness values will have a high chance to be selected and used to create the next generation, while those with low fitness have a small (but normally not zero) chance to be selected. The function in Listing 6 returns the index of

the solution selected from a list of fitness values using a roulette mechanism where the probability of an individual being selected is proportional to its fitness value, following a uniform distribution.

```
1 def select(fitnesses):
2     total = sum(fitnesses)
3     r = uniform(0, total)
4     acc = 0
5     for i in range(len(fitnesses)):
6         acc += fitnesses[i]
7         if acc >= r:
8             return i
```

Listing 6. Selection.

The select function is run twice below on an initial population of 10 individuals, to select two solutions. It should be noted that the solution with highest fitness may not always be selected, though its chance to be selected can be very high when selection is repeated multiple times.

```
1 >>> population, objs, fitnesses = initialization(10)
2 >>> i, j = select(fitnesses), select(fitnesses)
3 >>> print(objs)
4 >>> print(fitnesses)
5 >>> print(objs[i], objs[j])
6 [40, -3, 0, 19, 19, -5, -5, 4, 12, 19]
7 [49.5, 6.5, 9.5, 28.5, 28.5, 4.5, 4.5, 13.5, 21.5, 28.5]
8 40 4
```

2.6 Crossover

A commonly used method to recombine two solutions is called single point crossover. In this method, we identify a random position in the binary string and the digits after this point in the two parent solutions are swapped. For example, if we have the following two parent solutions and use the position after the fourth digit (shown as the blank space) as the point to swap:

```
1 1100 00110
2 1110 11111
```

we will create the following two child solutions:

```
1 1100 11111
2 1110 00110
```

The function crossover (Listing 7) takes two parent solutions (p1 and p2) and a crossover probability. The probability is used to determine whether the crossover operation will be carried out, following the Bernoulli distribution using the flip function. When no crossover is conducted, the function returns the parent solutions (Lines 2 and 3). Otherwise, a randomly

determined crossover point (Line 4) is used to create two child solutions.

```
1 def crossover(p1, p2, prob):
2     if not flip(prob):
3         return [p1, p2], 0
4     x = randint(0, len(p1)-1)
5     c1 = p1[:x] + p2[x:]
6     c2 = p2[:x] + p1[x:]
7     return [c1, c2], x
```

We test the effect of crossover using the two solutions selected from the initial 10 solutions generated from the above code. The result clearly shows that after the use of crossover a better solution (43) has entered the population.

```
1 >>> parent1, parent2 = population[i], population[j]
2 >>> obj1, obj2 = obj(decode(parent1)), obj(decode(parent2))
3 >>> [child1, child2], x = crossover(parent1, parent2, 1)
4 >>> obj1c, obj2c = obj(decode(child1)), obj(decode(child2))
5 >>> print('Parents    Obj     Children   Obj')
6 >>> print(parent1[:x], parent1[x:], obj1, ' -> ', \
7 ...       child1[:x], child1[x:], obj1c)
8 >>> print(parent2[:x], parent2[x:], obj2, ' -> ', \
9 ...       child2[:x], child2[x:], obj2c)
10 Parents    Obj     Children   Obj
11 10111 1000 19  ->  10111 0100 16
12 11111 0100 40  ->  11111 1000 43
```

2.7 Mutation

The mutation operation serves a purpose that is different than crossover in the GA search process. Crossover is exploitive because it only uses information that is already in the population and tries different combinations. In contrast, mutation works as an explorative force that brings new information into the population. This can be accomplished by simply flipping the digits in a solution from 0 to 1, or vice versa. However, we typically do not want to flip every digit in a solution, since this may drastically disturb the solutions in the population and may induce devolution. We use a probability parameter to control whether a digit is to be mutated (Listing 8). Later, we will see how the magnitude of this probability affects the overall performance of the GA.

```
1 def mutation(s, prob):
2     _mutate = lambda i: '1' if i=='0' else '0'
3     mutated = ''
4     for c in s:
5         if flip(prob):
6             mutated += _mutate(c)
7         else:
8             mutated += c
9     return mutated
```

Listing 8. Mutation.

2.8 Elitism

We now have all the essential parts of the GA and are ready to assemble them into a functional program. The function generation (Listing 9) goes through a reproduction process to create a new population of solutions using selection, crossover, and mutation operations. New individuals are continuously added to the population until the maximum size of the population is reached (Line 6).

```
1  def generation(population, objs, fitnesses,
2      pcrossover, pmutation, elitism=True):
3      oldbest = max(objs)
4      ibest = objs.index(oldbest)
5      newpop = []
6      while len(newpop) < len(population):
7          p1 = population[select(fitnesses)]
8          p2 = population[select(fitnesses)]
9          offspring = crossover(p1, p2, pcrossover)[0]
10         for s in offspring:
11             c = mutation(s, pmutation)
12             if len(newpop) < len(population):
13                 newpop.append(c)
14             else:
15                 break
16     newobjs = [obj(decode(s)) for s in newpop]
17     newfitnesses = obj2fitness(newobjs)
18     return newpop, newobjs, newfitnesses
```

Listing 9. Reproduction of the population.

The following is our first GA, which uses a population size of 10. It runs for 10 generations, using a crossover probability of 0.9 and mutation 0.1.

```
1 >>> population, objs, fitnesses = initialization(10)
2 >>> for i in range(10):
3 >>>     population, objs, fitnesses = generation(\
4 ...         population, objs, fitnesses, 0.9, 0.1)
5 >>> print(max(objs))
6 46
```

It should be evident that the GA as implemented here may lack the ability to find the optimal solution. The following test illustrates this problem.

```
>>> all_result = []
>>> for _ in range(10):
>>>     population, objs, fitnesses = initialization(10)
>>>     for i in range(10):
>>>         population, objs, fitnesses = generation(\
...             population, objs, fitnesses, 0.9, 0.1)
>>>     all_result.append(max(objs))
>>> print(all_result)
[48, 47, 42, 48, 49, 43, 44, 47, 45, 50]
```

As shown in this example, the GA was run ten times and it found the global optimal solution (50) only once. This level of performance is common for GAs: good solutions found in early GA generations tend to be lost when crossover and mutation operations are applied (Goldberg, 1989). A simple approach to addressing this issue is elitism: make sure the best solution found so far by the GA is kept in the population. The following is a revised version of function generation where an elitism mechanism is implemented.

```
def generation(population, objs, fitnesses,
               pcrossover, pmutation, elitism=True):
    oldbest = max(objs)
    ibest = objs.index(oldbest)
    newpop = []
    while len(newpop) < len(population):
        p1 = population[select(fitnesses)]
        p2 = population[select(fitnesses)]
        offspring = crossover(p1, p2, pcrossover)[0]
        for s in offspring:
            c = mutation(s, pmutation)
            if len(newpop) < len(population):
                newpop.append(c)
            else:
                break
    newobjs = [obj(decode(s)) for s in newpop]
    if elitism:
        newbest = max(newobjs)
        inewworst = newobjs.index(min(newobjs))
        if oldbest > newbest: # oldbest is better
            newpop[inewworst] = population[ibest]
            newobjs[inewworst] = oldbest
    population = newpop
    objs = newobjs
    newfitnesses = obj2fitness(newobjs)
    return newpop, newobjs, newfitnesses
```

Listing 10. Reproduction with elitism added.

The test below with the same settings as the previous example reveals that the use of elitism not only enables the GA to find a greater number of optimal solutions in the ten runs, it also improves the overall quality of solutions even when the optimum is not found.

```
>>> all_result = []
>>> for _ in range(10):
>>>     population, objs, fitnesses = initialization(10)
>>>     for i in range(10):
>>>         population, objs, fitnesses = \generation(
...             population, objs, fitnesses, 0.9, 0.1)
>>>     all_result.append(max(objs))
>>> print(all_result)
[50, 50, 50, 50, 50, 50, 50, 50, 49, 50]
```

2.9 GA and Parameters

Now we wrap up the code using a class to specify all the parameters needed to configure the GA and then define a specific function called GA. The GA function returns the best and average objective function value, along with the last population evolved by the algorithm.

```
class Parameters:
    def __init__(self, popsize, numgen,
                 pcrossover, pmutation, elitism):
        self.popsize = popsize
        self.numgen = numgen
        self.pcrossover = pcrossover
        self.pmutation = pmutation
        self.elitism = elitism

def GA(param):
    population,objs,fitnesses = initialization(param.popsize)
    for i in range(param.numgen):
        population, objs, fitnesses = generation(
            population, objs, fitnesses,
            param.pcrossover, param.pmutation, param.elitism)
    return max(objs), sum(objs)/param.popsize, population
```

Listing 11. GA and parameters.

2.10 GA Convergence

The previous example shows that running a GA through multiple generations will lead to an optimal (or near-optimal) solution. Now we provide an empirical analysis of how the GA population evolves. The GA was run ten times, each time with a total number of generations of 20, a population size of 20, a crossover probability of 0.9, and a mutation probability of 0.1. The best, worst, and average objective function values in the population in each generation are plotted in Figure 4, which clearly shows the best solution found in each generation all converge to the optimal value (50); average objective values also show a trend of improving throughout the generations. The worst solutions in the population fluctuate as expected, given the design of the mutation operation.

3. GA Performance

The GA designed here requires the specification of a number of parameters and it is important to understand the effects of those parameters on performance. To illustrate, we consider the values of the parameters listed in the first column of Table 1. We test the effects of these parameters using the combinations of values listed in the second column of the table. The goal is to examine how each parameter affects the ability of the GA to find the optimal solution.

For each combination of parameter values, the GA is run 10 times and we report the best solution found in each run. Figure 4 shows the results of the experiments. Among the 10 runs, the median solutions are plotted on the left side of the figure and worst solutions on the right.

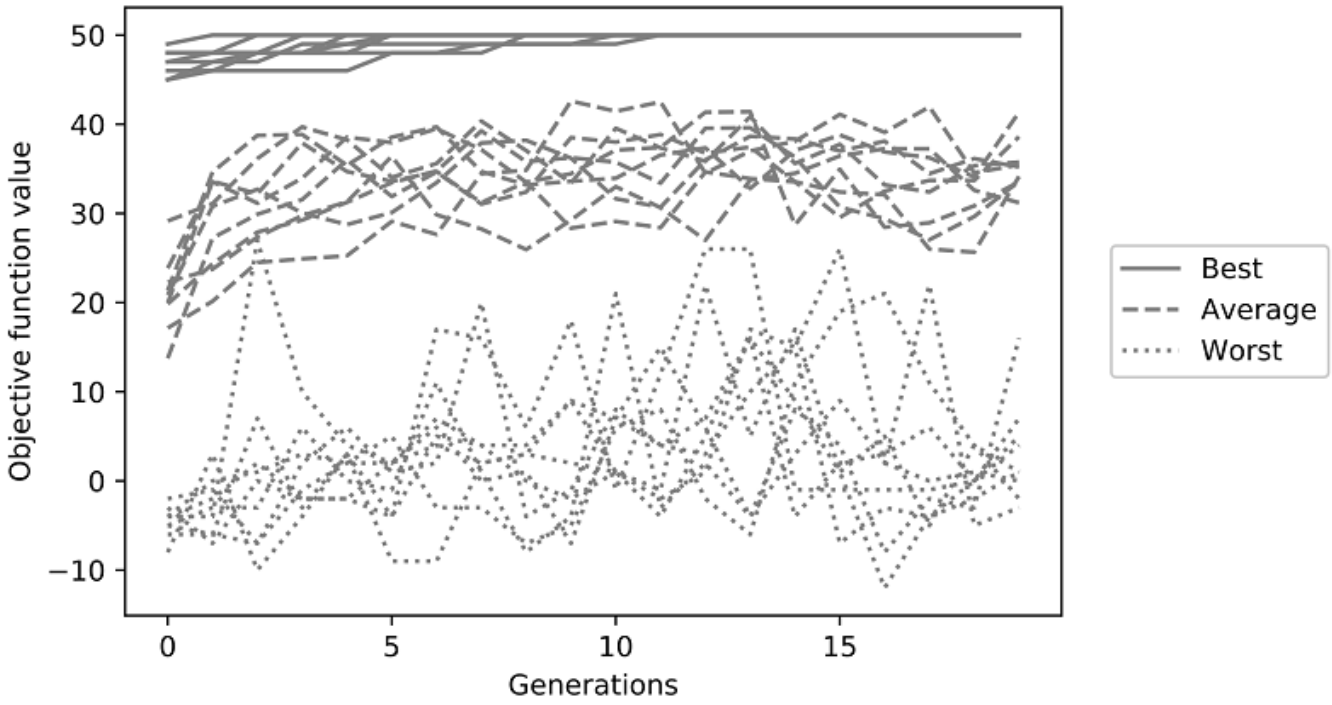

Figure 4. Results from experiments on GA convergence. Each solid line, for example, shows the best objection function values of the population through the 20 generations in a run of the GA. Source: authors.

Table 1. GA Parameters Tested.

Parameter	Values
Population size	2, 4, 6, 8, 10, 12, 12, 14, 16, 18, 20
Number of generations	5, 10, 15, 20, 25, 30
Crossover probability	0, 0.1, 0.3, 0.5, 0.7, 0.9
Mutation probability	0, 0.1, 0.3, 0.5, 0.7, 0.9
Elitism	True, False

A few observations about the ability of the GA finding the optimal solution to the problem can be made based on these test results:

- GA performance increases with population size and the number of generations.
- GA performance increases with crossover probability.
- The benefit of population size and number of generations decreases when these parameters reach a certain level. The same trend can also be observed for the crossover probability parameter.
- GA performance increases when mutation is used. However, increasing mutation probability may not necessarily increase overall performance, as indicated by the gap between the red lines (low mutation probability) and the green and grey lines (high probabilities).

4. Genetic Algorithms for Spatial Optimization

So far we have discussed GAs in their original form: the use of binary encoding, along with the crossover and mutation operations designed to be used with such encoding. While this approach can be used effectively, other types of optimization problems require different encoding strategies and genetic operations. For many location-allocation and spatial optimization problems, for example, the goal is find a discrete set of spatial units and in such cases it is common to use strings of integers to represent the solutions (Xiao and Murray, 2019).

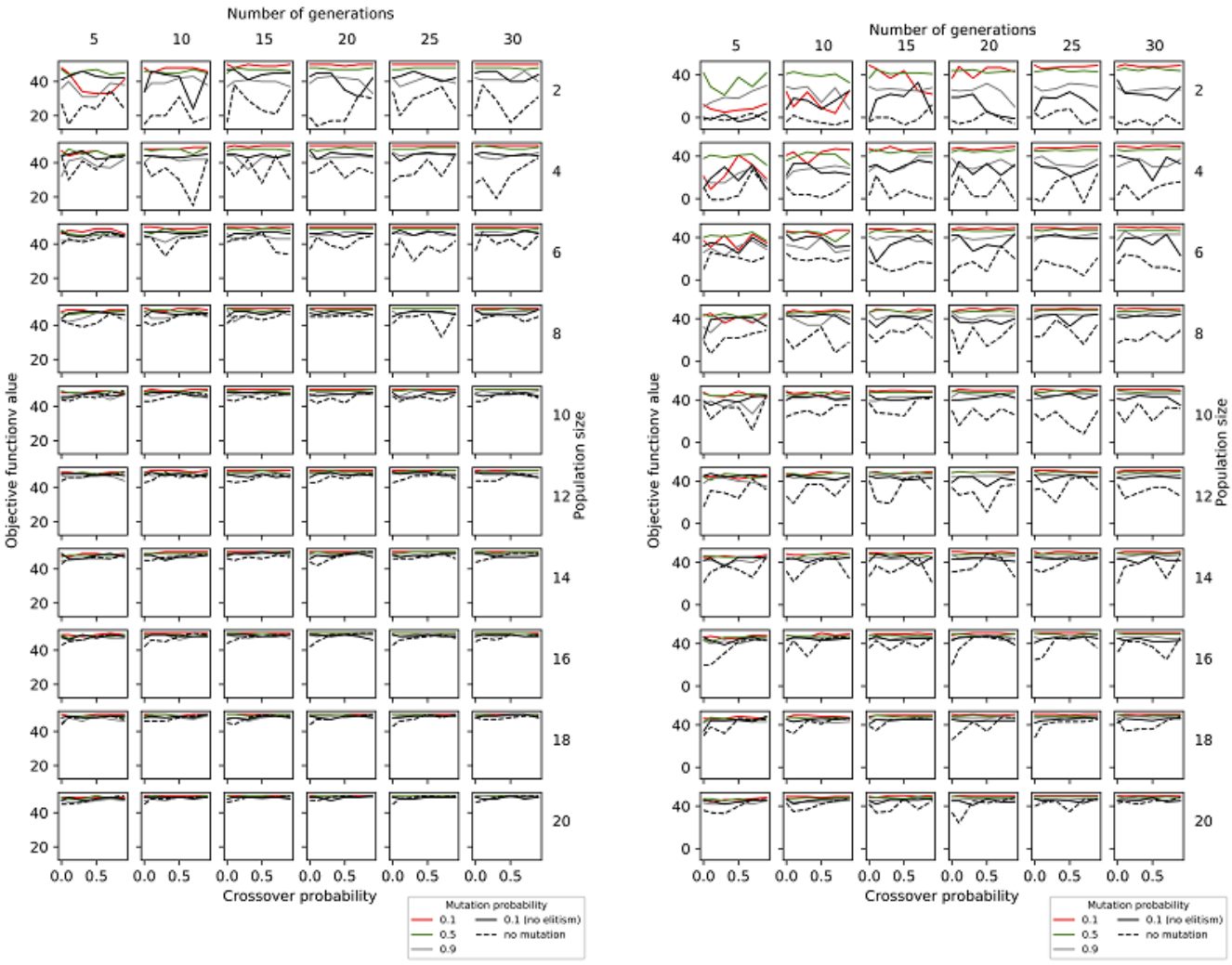

Figure 5. Results from experiments on GA performance. Median values are on the left side and worst solutions are on the right side. Source: authors.

We use the p-median problem, a classical optimization model in the location-allocation literature (Daskin, 1995), to demonstrate some special considerations of using GA for spatial optimization. The goal of solving a p-median problem is to locate p facilities on a network of n nodes so that the total distance from each node to its nearest facility is minimized. While there are infinite possibilities for the facilities to be located along network edges, it has been shown that at least one optimal solution exists when the facilities are located at the network nodes (Hakimi, 1964). This property makes it possible to encode a p-median problem using a string of p integers.

To formulate a p-median problem, we need a few inputs: the number of nodes in the network, the matrix that provides the distance between each pair of nodes, and the number of facilities to be selected. We use a hypothetical network (Figure 6) where these three inputs are specified using the following three variables, respectively.

```
1 n = 8
2 distmatrix = [
3     [0, 3, 13, 5, 12, 16, 17, 20],
```

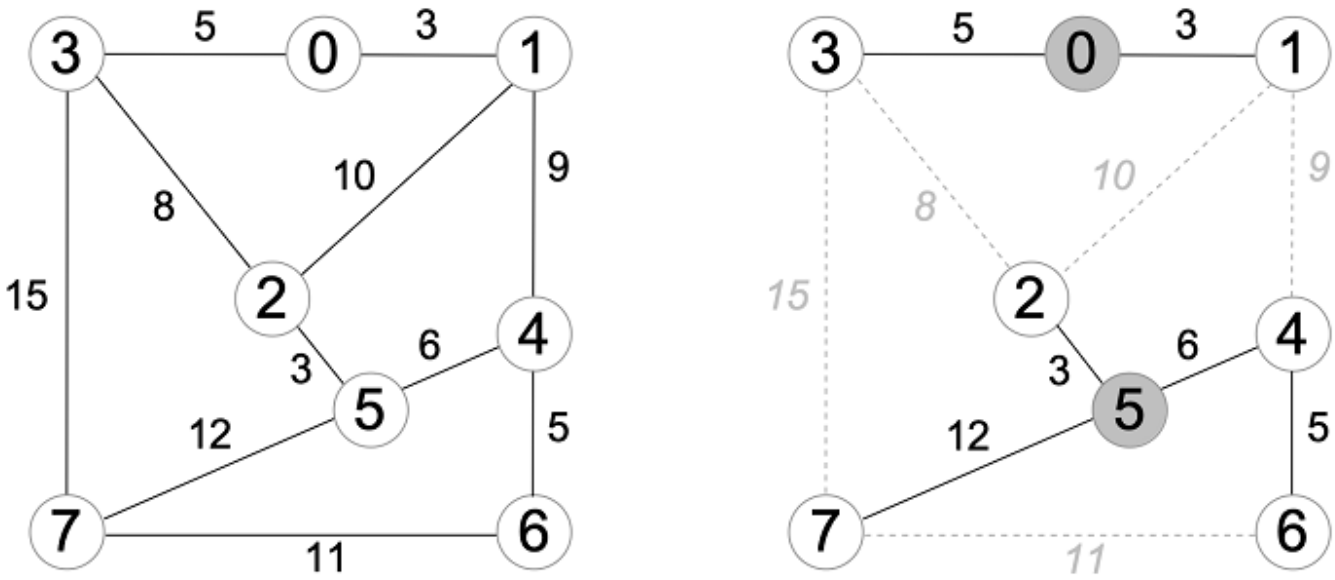

Figure 6. An example network with 8 nodes (redrawn from Xiao, 2016). The figure is not drawn to scale. The number on each edge is the distance. The figure on the right shows the optimal solution for p = 2 where the total distance is 40 and the selected nodes are marked as grey circles. Source: authors.

```
4     [3, 0, 10, 8, 9, 13, 14, 23],
5     [13, 10, 0, 8, 9, 3, 14, 15],
6     [5, 8, 8, 0, 17, 11, 22, 15],
7     [12, 9, 9, 17, 0, 6, 5, 16],
8     [16, 13, 3, 11, 6, 0, 11, 12],
9     [17, 14, 14, 22, 5, 11, 0, 11],
10    [20, 23, 15, 15, 16, 12, 11, 0]]
11 p = 2
```

The GA for the p-median problem will follow the structure of the code presented in the previous section. However, specific implementations of some functions are changed because the problem setting is different. In particular, a solution to the problem is encoded as a list of p integers, each representing the index of the node being selected to locate a facility. The objective function value of a solution can therefore be calculated using the function in Listing 12, which sums distances from each node to its nearest node in the solution (Lines 5 through 9).

```
1  def obj(s):
2      n = len(distmatrix)
3      d_total = 0
4      for i in range(n):
5          d_min = float('inf')
6          for j in s:
7              if distmatrix[i][j] < d_min:
8                  d_min = distmatrix[i][j]
9          d_total += d_min
10     return d_total
```

Listing 12. Calculating the objective function for the p-median problem.

With the obj function, we can then write the function to compute the fitness values for a population of solutions (Listing 13). Here we simply use the reciprocal of the objective function value as the fitness value.

```
1  def obj2fitness(objs):
2      fitnesses = [1/val for val in objs]
3      return fitnesses
```

Listing 13. Converting objective to fitness values for the p-median problem.

To initialize a solution to the problem, we randomly pick two nodes from the network (Listing 14).

```
1  def initialization(popsize, n, p):
2      population = []
3      for i in range(popsize):
4          sol = sample(range(n), p)
5          population.append(sol)
6      objs = [obj(s) for s in population]
7      fitnesses = obj2fitness(objs)
8      return population, objs, fitnesses
```

Listing 14. Initializing population for the p-median problem.

The crossover operation (Listing 15) is quite different from a typical GA using a binary encoding. We follow a specific crossover method (Alp et al., 2003) in which all the unique integers from the two parent solutions are first combined. This may lead to an infeasible solution with more than p integers. The method continuously removes integers from the combined list until there are exactly p unique integers left in the list. This is implemented in the loop (starting at Line 6) that keeps reducing the size of list child until it has exactly p elements. Every time a node is removed from the list, the total distance will increase, and the method removes the node that, by removing it, causes the least increase in total distance (Lines 9 to 14).

```
1  def crossover(s1, s2, prob):
2      if not flip(prob):
3          return s1
4      p = len(s1)
5      child = list(set(s1 + s2)) # get unique integers
6      while len(child) > p:
7          d = float('inf')
8          to_remove = -1
9          for i in child:
10             c1 = [j for j in child if j != i]
11             d1 = obj(c1)
12             if d1 < d:
13                 d = d1
14                 to_remove = i
15         child = [j for j in child if j != to_remove]
16     return child
```

Listing 15. Crossover for the p-median problem.

The mutation operation (Listing 16) serves the same purpose as in the example that employed a binary string representation: we bring new information into the population in order to explore different combinations of the nodes. However, we need to ensure that each solution is valid after mutation. The mutation function assigns a new value to a node only if the new value does not exist in the solution.

```
1  def mutation(s, prob):
2      for i in range(len(s)):
3          if not flip(prob):
4              continue
5          while True:
6              j = randint(0, n-1)
7              if not j in s:
8                  s[i] = j
9                  break
10     return s
```

Listing 16. Mutation for the p-median problem.

The selection operation will be exactly the same as the one we have already described: a roulette selection process based on the fitness values of the solutions in the population (Listing 6). The generation function (Listing 17), though largely similar to the previous example, minimizes the objective function (recall that we are using the reciprocal value) and elitism is implemented such that the solution with the minimum objective function value (Line 16) is used to replace the one with the highest objective value.

```
def generation(pop, objs, fitnesses, pcrossover,
               pmutation, elitism=True):
    oldbest = min(objs)
    ibest = objs.index(oldbest)
    newpop = []
    for i in range(len(pop)):
        p1 = pop[select(fitnesses)]
        p2 = pop[select(fitnesses)]
        child1 = crossover(p1, p2, pcrossover)
        child1 = mutation(child1, pmutation)
        newpop.append(child1)
    newobjs = [obj(c) for c in newpop]
    if elitism:
        newbest = min(newobjs)
        iworst = newobjs.index(max(newobjs))
        if oldbest < newbest: # old best is better
            newpop[iworst] = pop[ibest]
            newobjs[iworst] = oldbest
    newfitnesses = obj2fitness(newobjs)
    return newpop, newobjs, newfitnesses
```

Listing 17. Reproduction for the p-median problem.

Finally, the GA function (Listing 18) uses all the code implemented so far, and we also use the same class of Parameters to organize the parameters used in the GA.

```
class Parameters:
    def __init__(self, popsize, numgen,
                 pcrossover, pmutation, elitism):
        self.popsize = popsize
        self.numgen = numgen
        self.pcrossover = pcrossover
        self.pmutation = pmutation
        self.elitism = elitism

def GA(param):
    population, objs, fitnesses = initialization(
        param.popsize, n, p)
    for i in range(param.numgen):
        population, objs, fitnesses = generation(
            population, objs, fitnesses,
            param.pcrossover, param.pmutation, param.elitism)
    return min(objs), sum(objs)/param.popsize, population
```

Listing 18. GA for the p-median problem.

The following code tests how many times the optimal solution is found after running the GA 100 times.

```
>>> param = Parameters(
>>>     popsize = 4,
>>>     numgen = 10,
>>>     pcrossover = 0.9,
>>>     pmutation = 0.8,
>>>     elitism = True)
>>>
>>> res = [GA(param)[0] for _ in range(100)]
>>> print(sum([1 if r==40 else 0 for r in res]))
78
```

As shown in the numerical optimization example, parameters such as number of generations and population size have similar impacts on p-median problem performance. Elitism is also effective. However, the above test uses a relatively high mutation probability of 0.8, and the result is somewhat different than what we observed in the numerical problem—with such a high mutation rate we would expect lower performance but in this case, the GA finds the optimal solution 78 out of 100 times. It is therefore useful to further examine the impact of mutation probability on the performance of the p-median GA. More specifically, we test different combinations of mutation and crossover probabilities as listed in Table 2. For each combination of these two parameters, we run the GA 1000 times and see how often the optimal solution (total distance of 40) is found. We fix the size of the population (4) and the number of generations (10) for these experiments.

Table 2. GA Parameters Tested

Parameter	Values
Crossover probability	0, 0.1, 0.3, 0.5, 0.7, 0.9
Mutation probability	0, 0.1, 0.3, 0.5, 0.7, 0.9

The test results are shown in Figure 7, where the numbers on each line indicate the mutation probability. It is clear from this experiment that mutation plays a crucial and different role: higher mutation probabilities tend to outperform lower ones. This is reasonable given the relatively small population size—it is unlikely that 4 solutions can explore all the nodes. (We leave it as a question for further experiments to reveal the impact of higher population sizes on the GA performance.) Without crossover, the GA can only find the optimal solution by chance, which increases with mutation probability.

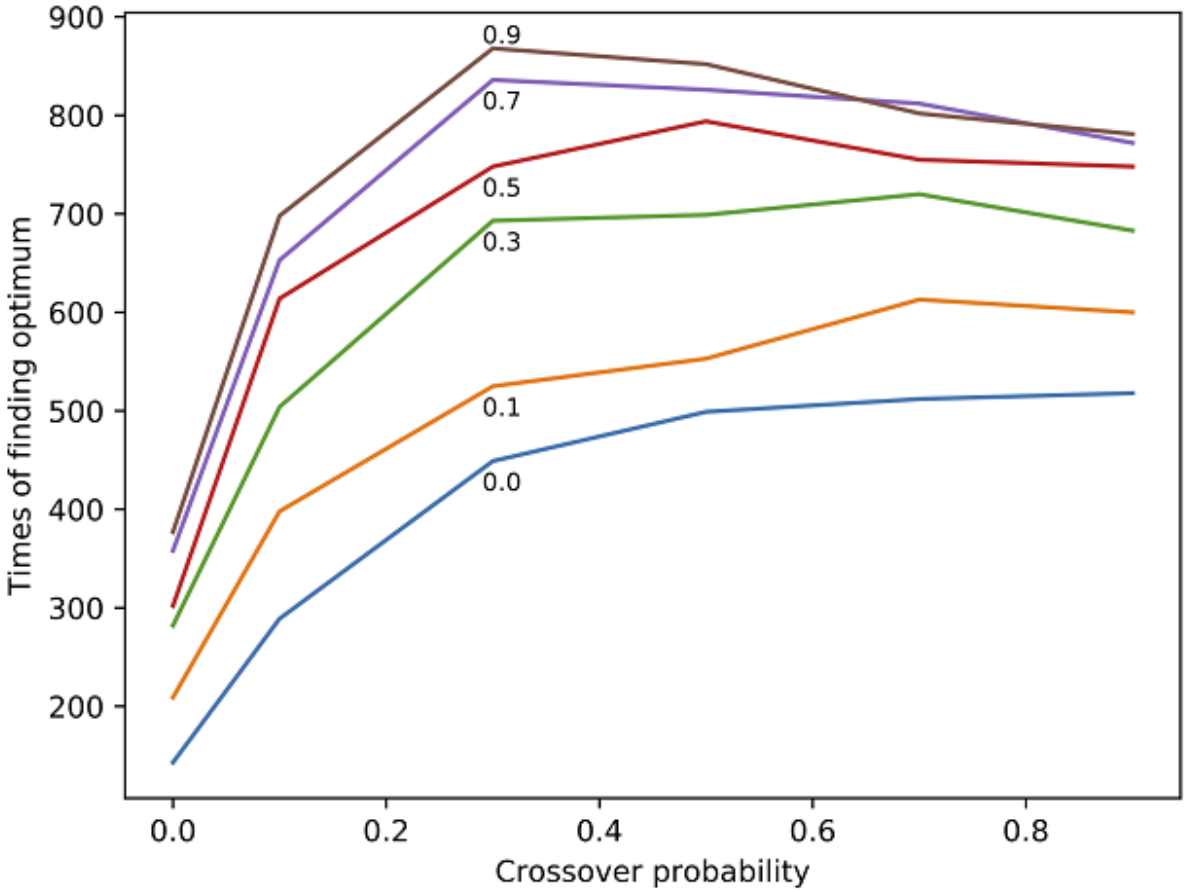

Figure 7. Results from experiments on GA performance for the p-median problem. Source: authors.

5. Conclusions and Discussion

The GA implementations developed here were able to efficiently and effectively find optimal solutions to the simple problems posed. The effect of selected parameters on the performance of the algorithms was explored by systematically varying them and evaluating changes in the results. It is clear that mutation and elitism are key components to the efficient achievement of optimality. The approach described here can be extended to include multiple objectives (Deb, 2001) and the results can be examined in light of the trade-offs that exist among the different objectives. As a consequence, with suitable visualization tools genetic and evolutionary approaches can serve as a core analytical element for spatial decision support systems (Xiao et al., 2007).

Genetic algorithms belong to a general field of evolutionary computation (Ba¨ck et al., 1997) inspired by evolutionary processes that can be observed in natural and biological systems (Holland, 1975; Forrest and Mitchell, 2016). The selection operation in a GA plays a critical role in the search process, reflecting the Darwinian principle of survival of the fittest. In addition to GAs, there are other types of search algorithms that apply similar logic but are designed using different encoding and reproduction strategies. For example, an

evolutionary strategy (ES) uses real values instead of a binary strings (Rechenberg, 1965). An ES contains only 1 parent solution and various mutation methods are used to create a number of mutants that will compete with and replace the parent. Genetic programming (Koza, 1992) aims to evolve computer programs (instead of numerical optimization problems) that are represented as trees.

One of the problems with evolutionary computing is the maintenance of a population of solutions, which may lead to a high demand for computing resources. Researchers have long recognized the parallel features in these algorithms as the individuals can be treated concurrently so that a distributed computing environment can be used to expedite evolutionary processes (Gong et al., 2015). This is becoming increasingly important for analyses applied to large data sets in this big data era (Tsai et al., 2015).

The advantages of GAs in solving optimization problems have long been recognized in the spatial optimization literature. Early efforts were mainly focused on solving location problems and, as shown in the previous section using the example of the p-median problem, GAs can be used to effectively solve such problems. In addition to solving problems derived from location-allocation models, GAs are also used to help find solutions to land use planning problems that often exhibit multiple and conflicting objectives (Xiao and Murray, 2019).

With the success of GAs, we can point out a few challenges and future developments for spatial problems. First, other flavors of evolutionary computing have yet to gain traction in the spatial problem solving literature. Openshaw (1998), for example, tested the use of genetic programming methods in a search for spatial interaction models, and it will be interesting to see how this line of research can be extended, especially in the big data era. Second, the literature of spatial optimization has a long history before the emergence of genetic algorithms and evolutionary computing. However, it is not yet clear how the knowledge developed prior to GAs can be fully incorporated into them in order to design more effective algorithms. Third, there are many other nature-inspired optimization methods (e.g., ant colony optimization and particle swarm optimization algorithms), and it will be interesting to see how hybrid algorithms can be developed to utilize these methods.

References

Alp, O., Drezner, Z., and Erkut, E. (2003). An efficient genetic algorithm for the p-median problem. Annals of Operations Research 122, 21–42.

Baeck, T., D. B. Fogel, and Z. Michalewicz (Eds.) (1997). Handbook of Evolutionary Computation. New York: Oxford University Press/IOP.

Daskin, M. S. (1995). Network and Discrete Location: Models, Algorithms, and Applications, 1st Edition. New York: John Wiley & Sons.

Deb, K. (2001). Multi-Objective Optimization Using Evolutionary Algorithms. Chichester: John Wiley & Sons.

Forrest, S. and M. Mitchell (2016). Adaptive computation: The multidisciplinary legacy of John H. Holland. Communications of the Association for Computing Machinery 59, 58–63.

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization and Machine Learning. Reading, MA: Addison-Wesley.

Gong, Y.-J., W.-N. Chen, Z.-H. Zhan, J. Zhang, Y. Li, and Q. Zhang (2015). Distributed evolutionary algorithms and their models: A survey of the state-of-the-art. Applied Soft Computing 34, 286–300.

Hakimi, S. L. (1964). Optimum location of switching centers and the absolute centers and medians of a graph. Operations Research 12, 450–459.

Holland, J. H. (1975). Adaptations in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. Ann Arbor, MI: University of Michigan Press.

Koza, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MA: The MIT Press.

Openshaw, S. (1998). Neural Network, Genetic, and Fuzzy Logic Models of Spatial Interaction. Environment and Planning A 30, 1857–1872.

Rechenberg, I. (1965). Cybernetic Solution Path of an Experimental Problem. Farnborough, Hants, U.K.: Royal Aircraft Establishment, Library Translation No. 1122.

Tsai, C. W., Lai, C. F., Chao, H. C., and Vasilakos, A. V. (2015). Big data analytics: a survey. Journal of Big Data 2, 21.

Xiao, N. (2016). GIS Algorithms. London: SAGE Publications.

Xiao, N. and Murray, A. T. (2019). Spatial optimization for land acquisition problems: A review of models, solution methods, and GIS support. Transactions in GIS, 23(4): 645-671.

Xiao, N., Bennett, D. A., and Armstrong, M. P. (2007). Interactive evolutionary approaches to multiobjective spatial decision making: A synthetic review. Computers, Environment and Urban Systems 30, 232–252.