Search-Based Test Generation - Part 1

December 10, 2024

1 Search-Based Test Generation - Part 1

[1]:
```
import random
import matplotlib.pyplot as plt
import numpy as np
from mpl_toolkits.mplot3d import Axes3D

import inspect
import ast
import astor

import sys

# For presenting as slides
#plt.rcParams['figure.figsize'] = [12, 8]
#plt.rcParams.update({'font.size': 22})
#plt.rcParams['lines.linewidth'] = 3
```

1.1 The Test Data Generation Problem

The problem we will consider in this chapter is the following: Given an entry point to a program (a function), we want to find values for the parameters of this function such that the execution of the function reaches a particular point in the program. In other words, we aim to find a test input to the program that covers a target statement. We will then generalise this problem to finding test inputs to cover *all* statements in the program.

Assume we are aiming to test the following function under test:

[2]:
```
def test_me(x, y):
    if x == 2 * (y + 1):
        return True
    else:
        return False
```

The `test_me` function has two input parameters, `x`and `y`, and it returns `True` or `False` depending on how the parameters relate:

[3]:
```
test_me(10, 10)
```

[3]:
```
False
```

[4]:
```
test_me(22, 10)
```

[4]:
```
True
```

In order to address the test generation problem as a search problem, we need to decide on an encoding, and derive appropriate search operators. It is possible to use bitvectors like we did on previous problems; however, a simpler interpretation of the parameters of a function is a list of the actual parameters. That is, we encode a test input as a list of parameters; we will start by assuming that all parameters are numeric.

The representation for inputs of this function is lists of length two, one element for `x` and one for `y`. As numeric values in Python are unbounded, we need to decide on some finite bounds for these values, e.g.:

[5]:
```
MAX = 1000
MIN = -MAX
```

For generating inputs we can now uniformly sample in the range (MIN, MAX). The length of the vector shall be the number of parameters of the function under test. Rather than hard coding such a parameter, we can also make our approach generalise better by using inspection to determine how many parameters the function under test has:

[6]:
```
from inspect import signature
sig = signature(test_me)
num_parameters = len(sig.parameters)
num_parameters
```

[6]:
```
2
```

As usual, we will define the representation implicitly using a function that produces random instances.

[7]:
```
def get_random_individual():
    return [random.randint(MIN, MAX) for _ in range(num_parameters)]
```

[8]:
```
get_random_individual()
```

[8]:
```
[278, -311]
```

We need to define search operators matching this representation. To apply local search, we need to define the neighbourhood. For example, we could define one upper and one lower neighbour for each parameter:

- `x-1, y`
- `x+1, y`
- `x, y+1`
- `x, y-1`

[9]:
```
def get_neighbours(individual):
    neighbours = []
    for p in range(len(individual)):
        if individual[p] > MIN:
            neighbour = individual[:]
            neighbour[p] = individual[p] - 1
            neighbours.append(neighbour)
        if individual[p] < MAX:
            neighbour = individual[:]
            neighbour[p] = individual[p] + 1
            neighbours.append(neighbour)

    return neighbours
```

[10]:
```
x = get_random_individual()
x
```

[10]:
```
[681, -409]
```

[11]:
```
get_neighbours(x)
```

[11]:
```
[[680, -409], [682, -409], [681, -410], [681, -408]]
```

Before we can apply search, we also need to define a fitness function. Suppose that we are interested in covering the `True` branch of the if-condition in the `test_me()` function, i.e. `x == 2 * (y + 1)`.

[12]:
```
def test_me(x, y):
    if x == 2 * (y + 1):
        return True
    else:
        return False
```

How close is a given input tuple for this function from reaching the target (true) branch of `x == 2 * (y + 1)`?

Let's consider an arbitrary point in the search space, e.g. `(274, 153)`. The if-condition compares the following values:

[13]:
```
x = 274
y = 153
x, 2 * (y + 1)
```

[13]:
```
(274, 308)
```

In order to make the branch true, both values need to be the same. Thus, the more they differ, the further we are away from making the comparison true, and the less they differ, the closer we are from making the comparison true. Thus, we can quantify "how false" the comparison is by calculating the difference between `x` and `2 * (y + 1)`. Thus, we can calculate this distance as `abs(x - 2 * (y + 1))`:

[14]:
```
def calculate_distance(x, y):
    return abs(x - 2 * (y + 1))
```

[15]:
```
calculate_distance(274, 153)
```

[15]: 34

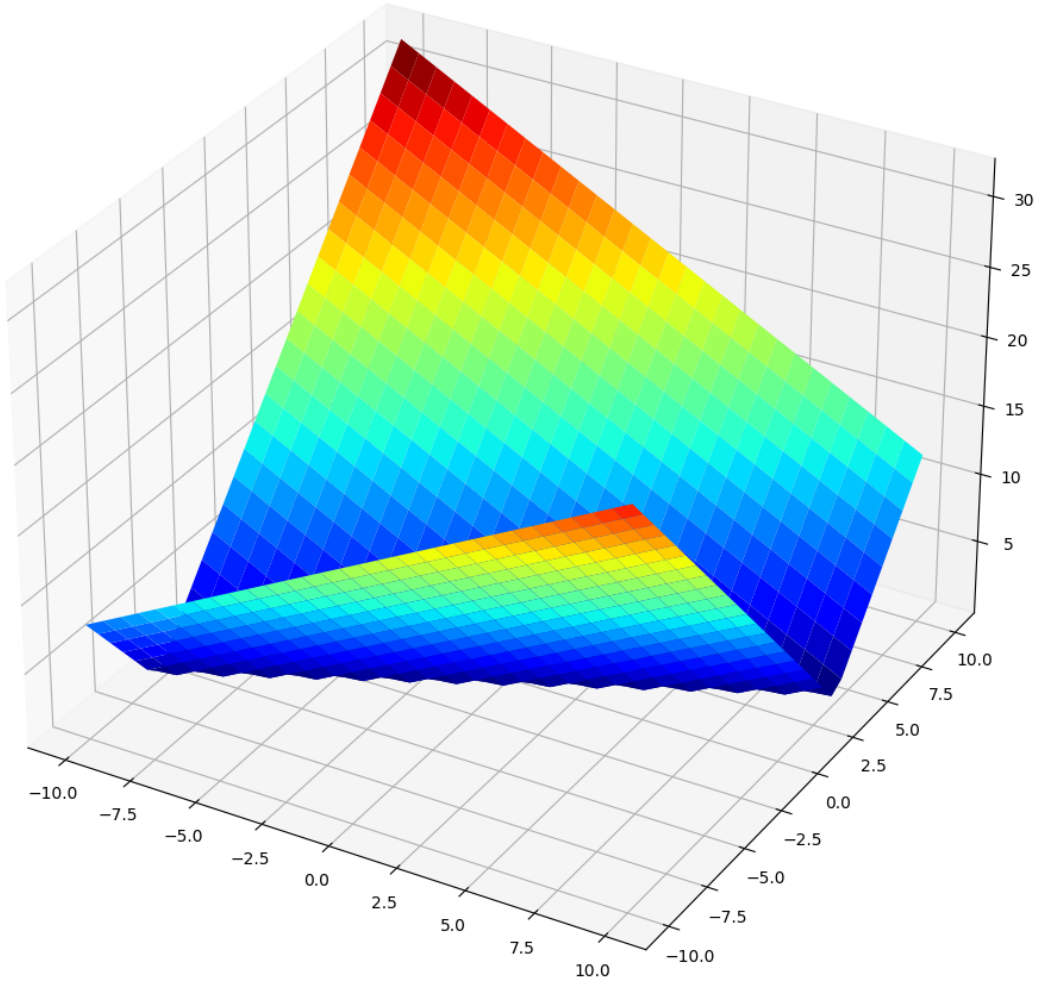
We can use this distance value as our fitness function, since we can nicely measure how close we are to an optimal solution. Note, however, that "better" doesn't mean "bigger" in this case; the smaller the distance the better. This is not a problem, since any algorithm that can maximize a value can also be made to minimize it instead.

For each value in the search space of integer tuples, this distance value defines the elevation in our search landscape. Since our example search space is two-dimensional, the search landscape is three-dimensional and we can plot it to see what it looks like:

[16]:
```
x = np.outer(np.linspace(-10, 10, 30), np.ones(30))
y = x.copy().T
z = calculate_distance(x, y)

fig = plt.figure(figsize=(12, 12))
ax  = plt.axes(projection='3d')

ax.plot_surface(x, y, z, cmap=plt.cm.jet, rstride=1, cstride=1, linewidth=0);
```



The optimal values, i.e. those that make the if-condition true, have fitness value 0 and can be clearly seen at the bottom of the plot. The further away from the optimal values, the higher elevated the points in the search space.

This distance can serve as our fitness function if we aim to cover the true branch of the program in our example:

[17]:
```
def get_fitness(individual):
    x = individual[0]
    y = individual[1]
    return abs(x - 2 * (y + 1))
```

We can now use any local search algorithm we have defined previously, with only one modification: In the prior examples where we applied local search we were always maximising fitness values; now

we are minimising, so a hillclimber, for example, should only move to neighbours with *smaller* fitness values:

[18]:
```
max_steps = 10000
fitness_values = []
```

Let's use a steepest ascent hillclimber:

[19]:
```
def hillclimbing():
    current = get_random_individual()
    fitness = get_fitness(current)
    best, best_fitness = current[:], fitness
    print(f"Starting at fitness {best_fitness}: {best}")

    step = 0
    while step < max_steps and best_fitness > 0:
        neighbours = [(x, get_fitness(x)) for x in get_neighbours(current)]
        best_neighbour, neighbour_fitness = min(neighbours, key=lambda i: i[1])
        step += len(neighbours)
        fitness_values.extend([best_fitness] * len(neighbours))
        if neighbour_fitness < fitness:
            current = best_neighbour
            fitness = neighbour_fitness
            if fitness < best_fitness:
                best = current[:]
                best_fitness = fitness
        else:
            # Random restart if no neighbour is better
            current = get_random_individual()
            fitness = get_fitness(current)
            step += 1
            if fitness < best_fitness:
                best = current[:]
                best_fitness = fitness
            fitness_values.append(best_fitness)


    print(f"Solution fitness after {step} fitness evaluations: {best_fitness}:␣
↪{best}")
    return best
```

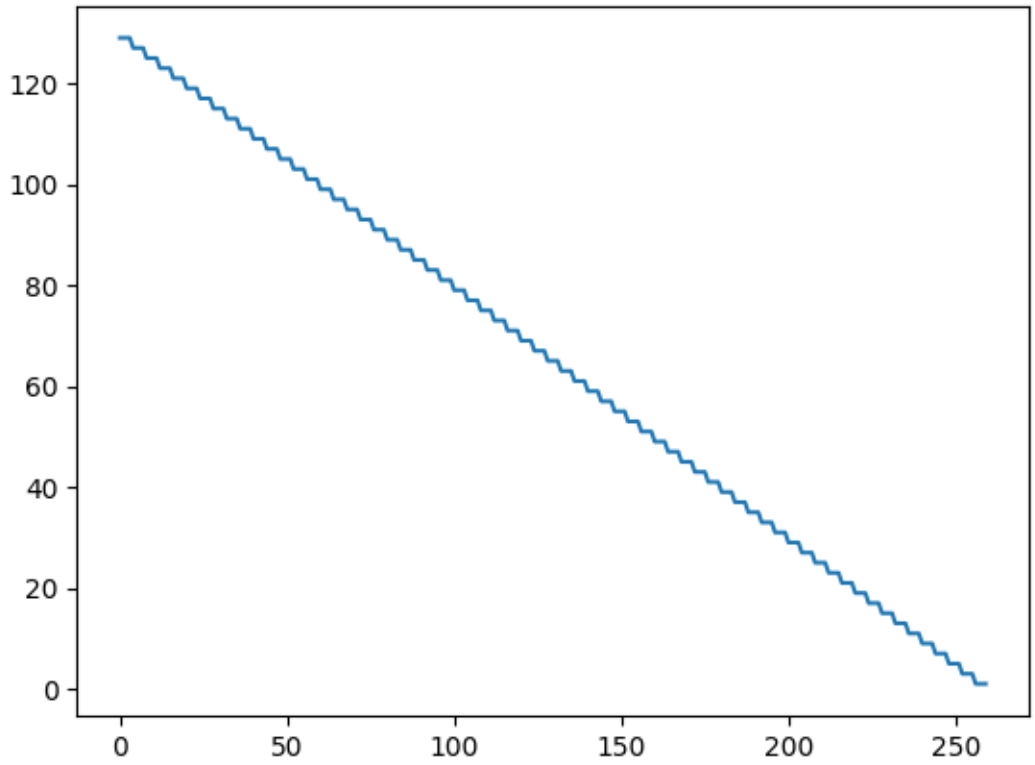
[20]:
```
max_steps = 10000
fitness_values = []
hillclimbing()
```

```
Starting at fitness 129: [-153, -142]
Solution fitness after 260 fitness evaluations: 0: [-154, -78]
```

[20]: [-154, -78]

[21]:
```
plt.plot(fitness_values)
```

[21]: [<matplotlib.lines.Line2D at 0x110dda8f0>]



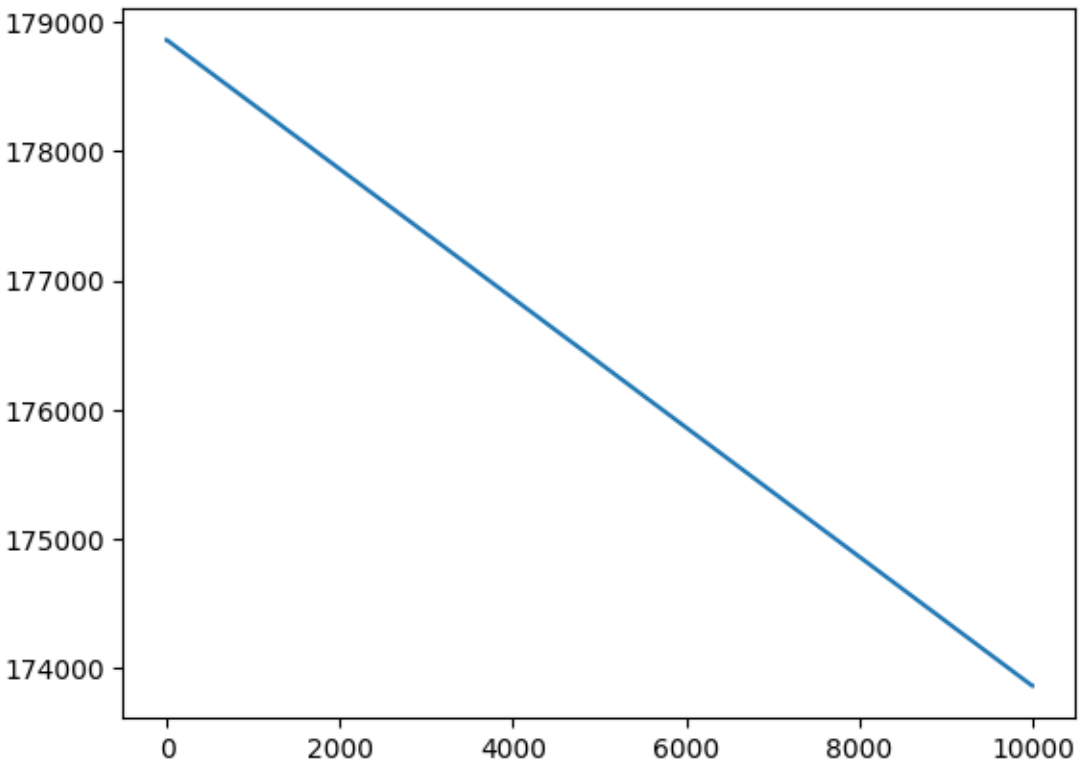
Since there are no local optima, the hillclimber will easily find the solution, even without restarts. However, this can take a while, in particular if we use a larger input space:

[22]:
```
MAX = 100000
MIN = -MAX
```

[23]:
```
fitness_values = []
hillclimbing()
plt.plot(fitness_values)
```

```
Starting at fitness 178861: [4057, 91458]
Solution fitness after 10000 fitness evaluations: 173861: [4057, 88958]
```

[23]: [<matplotlib.lines.Line2D at 0x11181e440>]



Unless the randomly chosen initial point is already close to an optimal solution, the hillclimber is going to be hopeless in moving through the search space within a reasonable number of iterations.

1.2 Alternating Variable Method

The search problem represented by the `test_me` function has an easy fitness landscape with no local optima. However, it still takes quite long to reach the optimum, depending on where the random starting point lies in the search space. This is because the neighbourhood for real program inputs can be quite large, depending on the number of parameters, and even just the search space for each parameter individually can already be very large. In our example we restricted `MAX` and `MIN` to a very narrow range, but imagine doing this for 32 bit integers. Both these problems are addressed with an adapted version of our hillclimber known as the *Alternating Variable Method*, which differs from the hillclimber in two ways: 1. Rather than considering the neighbourhood of all input parameters at once, we apply search to each parameter individually in turn 2. Rather than taking only small steps of size 1, we allow larger jumps in the search space.

Let's first consider the second aspect, larger jumps in the search space. The idea is to apply a *pattern* search where we first decide on a direction in the search space to move, and then apply increasingly larger steps in that direction as long as the fitness improves. We only consider a single parameter, thus the "direction" simply refers to whether one increases or decreases this value. The function thus takes (1) the individual on which to perform the search, (2) a particular parameter we are considering, (3) the direction of the search, and (4) the starting fitness values.

[24]:
```
def pattern_search(individual, parameter, direction, fitness):
    print(f"  {individual}, direction {direction}, fitness {fitness}")

    individual[parameter] = individual[parameter] + direction
    new_fitness = get_fitness(individual)
    if new_fitness < fitness:
        fitness_values.append(new_fitness)
        return pattern_search(individual, parameter, 2 * direction, new_fitness)
    else:
        # If fitness is not better we overshot. Undo last move, and return
        fitness_values.append(fitness)
        individual[parameter] = individual[parameter] - direction
        return fitness
```

For example, let's assume `y` is a large value (1000), and `x` is considerably smaller. For our example function, the optimal value for `x` would thus be at 2002. Applying the search to `x` we thus need to move in the positive direction (1), and the function will do this with increasing steps until it "overshoots".

[25]:
```
x = [0, 1000]
f = get_fitness(x)
pattern_search(x, 0, 1, get_fitness(x))
```

```
  [0, 1000], direction 1, fitness 2002
  [1, 1000], direction 2, fitness 2001
  [3, 1000], direction 4, fitness 1999
  [7, 1000], direction 8, fitness 1995
  [15, 1000], direction 16, fitness 1987
  [31, 1000], direction 32, fitness 1971
  [63, 1000], direction 64, fitness 1939
  [127, 1000], direction 128, fitness 1875
  [255, 1000], direction 256, fitness 1747
  [511, 1000], direction 512, fitness 1491
  [1023, 1000], direction 1024, fitness 979
  [2047, 1000], direction 2048, fitness 45
```

[25]:
```
45
```

If `x` is larger than `y` we would need to move in the other direction, and the search does this until it undershoots the target of 2002:

[26]:
```
x = [10000, 1000]
f = get_fitness(x)
pattern_search(x, 0, -1, get_fitness(x))
```

```
  [10000, 1000], direction -1, fitness 7998
  [9999, 1000], direction -2, fitness 7997
  [9997, 1000], direction -4, fitness 7995
  [9993, 1000], direction -8, fitness 7991
```

```
[9985, 1000], direction -16, fitness 7983
[9969, 1000], direction -32, fitness 7967
[9937, 1000], direction -64, fitness 7935
[9873, 1000], direction -128, fitness 7871
[9745, 1000], direction -256, fitness 7743
[9489, 1000], direction -512, fitness 7487
[8977, 1000], direction -1024, fitness 6975
[7953, 1000], direction -2048, fitness 5951
[5905, 1000], direction -4096, fitness 3903
[1809, 1000], direction -8192, fitness 193
```

[26]: 193

The AVM algorithm applies the pattern search as follows: 1. Start with the first parameter 2. Probe the neighbourhood of the parameter to find the direction of the search 3. Apply pattern search in that direction 4. Repeat probing + pattern search until no more improvement can be made 5. Move to the next parameter, and go to step 2

Like a regular hillclimber, the search may get stuck in local optima and needs to use random restarts. The algorithm is stuck if it probed all parameters in sequence and none of the parameters allowed a move that improved fitness.

[27]:
```python
def probe_and_search(individual, parameter, fitness):
    new_parameters = individual[:]
    value = new_parameters[parameter]
    new_fitness = fitness
    # Try +1
    new_parameters[parameter] = individual[parameter] + 1
    print(f"Trying +1 at fitness {fitness}: {new_parameters}")
    new_fitness = get_fitness(new_parameters)
    if new_fitness < fitness:
        fitness_values.append(new_fitness)
        new_fitness = pattern_search(new_parameters, parameter, 2, new_fitness)
    else:
        # Try -1
        fitness_values.append(fitness)
        new_parameters[parameter] = individual[parameter] - 1
        print(f"Trying -1 at fitness {fitness}: {new_parameters}")
        new_fitness = get_fitness(new_parameters)
        if new_fitness < fitness:
            fitness_values.append(new_fitness)
            new_fitness = pattern_search(new_parameters, parameter, -2,␣
↪new_fitness)
        else:
            fitness_values.append(fitness)
            new_parameters[parameter] = individual[parameter]
            new_fitness = fitness
```

```
    return new_parameters, new_fitness
```

[28]:
```
def avm():
    current = get_random_individual()
    fitness = get_fitness(current)
    best, best_fitness = current[:], fitness
    fitness_values.append(best_fitness)
    print(f"Starting at fitness {best_fitness}: {current}")

    changed = True
    while len(fitness_values) < max_steps and best_fitness > 0:
        # Random restart
        if not changed:
            current = get_random_individual()
            fitness = get_fitness(current)
            fitness_values.append(fitness)
        changed = False

        parameter = 0
        while parameter < len(current):
            print(f"Current parameter: {parameter}")
            new_parameters, new_fitness = probe_and_search(current, parameter,␣
 ↪fitness)
            if current != new_parameters:
                # Keep on searching
                changed = True
                current = new_parameters
                fitness = new_fitness
                if fitness < best_fitness:
                    best_fitness = fitness
                    best = current[:]
            else:
                parameter += 1

    print(f"Solution fitness {best_fitness}: {best}")
    return best
```

[29]:
```
fitness_values = []
avm()
```

```
Starting at fitness 13790: [44904, 29346]
Current parameter: 0
Trying +1 at fitness 13790: [44905, 29346]
  [44905, 29346], direction 2, fitness 13789
  [44907, 29346], direction 4, fitness 13787
  [44911, 29346], direction 8, fitness 13783
  [44919, 29346], direction 16, fitness 13775
  [44935, 29346], direction 32, fitness 13759
```

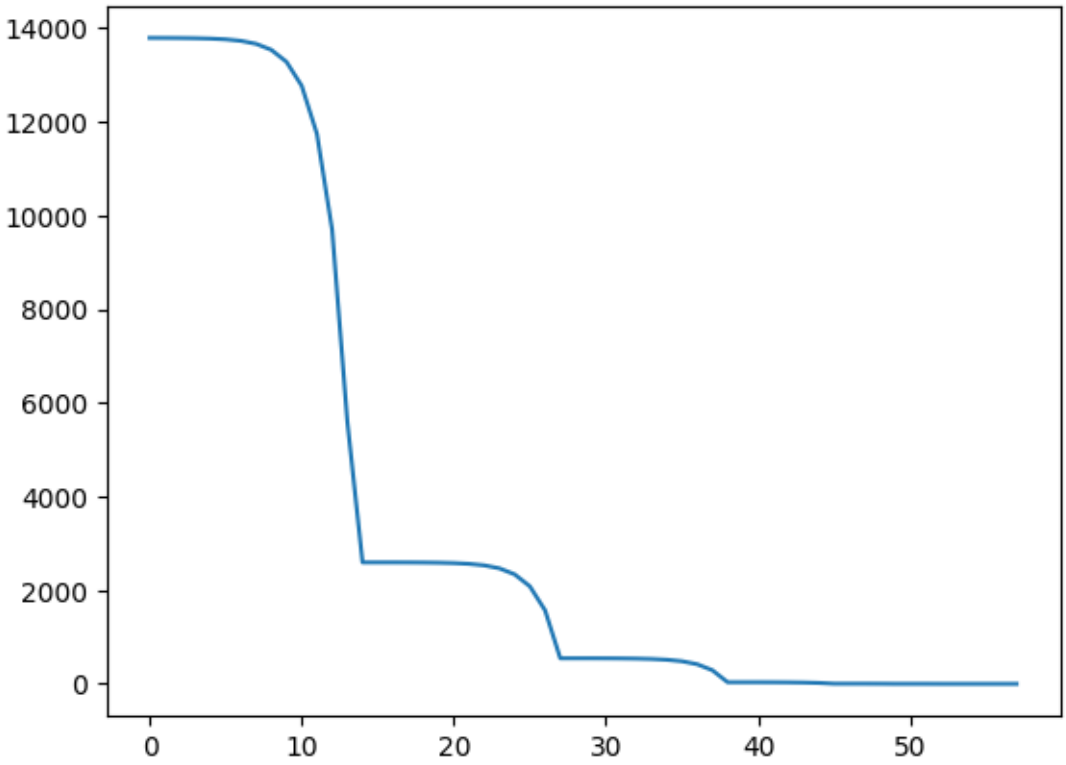
```
  [44967, 29346], direction 64, fitness 13727
  [45031, 29346], direction 128, fitness 13663
  [45159, 29346], direction 256, fitness 13535
  [45415, 29346], direction 512, fitness 13279
  [45927, 29346], direction 1024, fitness 12767
  [46951, 29346], direction 2048, fitness 11743
  [48999, 29346], direction 4096, fitness 9695
  [53095, 29346], direction 8192, fitness 5599
  [61287, 29346], direction 16384, fitness 2593
Current parameter: 0
Trying +1 at fitness 2593: [61288, 29346]
Trying -1 at fitness 2593: [61286, 29346]
  [61286, 29346], direction -2, fitness 2592
  [61284, 29346], direction -4, fitness 2590
  [61280, 29346], direction -8, fitness 2586
  [61272, 29346], direction -16, fitness 2578
  [61256, 29346], direction -32, fitness 2562
  [61224, 29346], direction -64, fitness 2530
  [61160, 29346], direction -128, fitness 2466
  [61032, 29346], direction -256, fitness 2338
  [60776, 29346], direction -512, fitness 2082
  [60264, 29346], direction -1024, fitness 1570
  [59240, 29346], direction -2048, fitness 546
Current parameter: 0
Trying +1 at fitness 546: [59241, 29346]
Trying -1 at fitness 546: [59239, 29346]
  [59239, 29346], direction -2, fitness 545
  [59237, 29346], direction -4, fitness 543
  [59233, 29346], direction -8, fitness 539
  [59225, 29346], direction -16, fitness 531
  [59209, 29346], direction -32, fitness 515
  [59177, 29346], direction -64, fitness 483
  [59113, 29346], direction -128, fitness 419
  [58985, 29346], direction -256, fitness 291
  [58729, 29346], direction -512, fitness 35
Current parameter: 0
Trying +1 at fitness 35: [58730, 29346]
Trying -1 at fitness 35: [58728, 29346]
  [58728, 29346], direction -2, fitness 34
  [58726, 29346], direction -4, fitness 32
  [58722, 29346], direction -8, fitness 28
  [58714, 29346], direction -16, fitness 20
  [58698, 29346], direction -32, fitness 4
Current parameter: 0
Trying +1 at fitness 4: [58699, 29346]
Trying -1 at fitness 4: [58697, 29346]
  [58697, 29346], direction -2, fitness 3
  [58695, 29346], direction -4, fitness 1
```

```
Current parameter: 0
Trying +1 at fitness 1: [58696, 29346]
Trying -1 at fitness 1: [58694, 29346]
  [58694, 29346], direction -2, fitness 0
Current parameter: 0
Trying +1 at fitness 0: [58695, 29346]
Trying -1 at fitness 0: [58693, 29346]
Current parameter: 1
Trying +1 at fitness 0: [58694, 29347]
Trying -1 at fitness 0: [58694, 29345]
Solution fitness 0: [58694, 29346]
```

[29]:
```
[58694, 29346]
```

[30]:
```
plt.plot(fitness_values)
```

[30]:
```
[<matplotlib.lines.Line2D at 0x111875ea0>]
```



The pattern search even works efficiently if we increase the size of the search space to 64-bit numbers:

[31]:
```
MAX = 2**32
MIN = -MAX
```

[32]:
```
fitness_values = []
avm()
```

```
Starting at fitness 11536505364: [-3762375150, 3887065106]
Current parameter: 0
Trying +1 at fitness 11536505364: [-3762375149, 3887065106]
  [-3762375149, 3887065106], direction 2, fitness 11536505363
  [-3762375147, 3887065106], direction 4, fitness 11536505361
  [-3762375143, 3887065106], direction 8, fitness 11536505357
  [-3762375135, 3887065106], direction 16, fitness 11536505349
  [-3762375119, 3887065106], direction 32, fitness 11536505333
  [-3762375087, 3887065106], direction 64, fitness 11536505301
  [-3762375023, 3887065106], direction 128, fitness 11536505237
  [-3762374895, 3887065106], direction 256, fitness 11536505109
  [-3762374639, 3887065106], direction 512, fitness 11536504853
  [-3762374127, 3887065106], direction 1024, fitness 11536504341
  [-3762373103, 3887065106], direction 2048, fitness 11536503317
  [-3762371055, 3887065106], direction 4096, fitness 11536501269
  [-3762366959, 3887065106], direction 8192, fitness 11536497173
  [-3762358767, 3887065106], direction 16384, fitness 11536488981
  [-3762342383, 3887065106], direction 32768, fitness 11536472597
  [-3762309615, 3887065106], direction 65536, fitness 11536439829
  [-3762244079, 3887065106], direction 131072, fitness 11536374293
  [-3762113007, 3887065106], direction 262144, fitness 11536243221
  [-3761850863, 3887065106], direction 524288, fitness 11535981077
  [-3761326575, 3887065106], direction 1048576, fitness 11535456789
  [-3760277999, 3887065106], direction 2097152, fitness 11534408213
  [-3758180847, 3887065106], direction 4194304, fitness 11532311061
  [-3753986543, 3887065106], direction 8388608, fitness 11528116757
  [-3745597935, 3887065106], direction 16777216, fitness 11519728149
  [-3728820719, 3887065106], direction 33554432, fitness 11502950933
  [-3695266287, 3887065106], direction 67108864, fitness 11469396501
  [-3628157423, 3887065106], direction 134217728, fitness 11402287637
  [-3493939695, 3887065106], direction 268435456, fitness 11268069909
  [-3225504239, 3887065106], direction 536870912, fitness 10999634453
  [-2688633327, 3887065106], direction 1073741824, fitness 10462763541
  [-1614891503, 3887065106], direction 2147483648, fitness 9389021717
  [532592145, 3887065106], direction 4294967296, fitness 7241538069
  [4827559441, 3887065106], direction 8589934592, fitness 2946570773
Current parameter: 0
Trying +1 at fitness 2946570773: [4827559442, 3887065106]
  [4827559442, 3887065106], direction 2, fitness 2946570772
  [4827559444, 3887065106], direction 4, fitness 2946570770
  [4827559448, 3887065106], direction 8, fitness 2946570766
  [4827559456, 3887065106], direction 16, fitness 2946570758
  [4827559472, 3887065106], direction 32, fitness 2946570742
  [4827559504, 3887065106], direction 64, fitness 2946570710
  [4827559568, 3887065106], direction 128, fitness 2946570646
```

```
  [4827559696, 3887065106], direction 256, fitness 2946570518
  [4827559952, 3887065106], direction 512, fitness 2946570262
  [4827560464, 3887065106], direction 1024, fitness 2946569750
  [4827561488, 3887065106], direction 2048, fitness 2946568726
  [4827563536, 3887065106], direction 4096, fitness 2946566678
  [4827567632, 3887065106], direction 8192, fitness 2946562582
  [4827575824, 3887065106], direction 16384, fitness 2946554390
  [4827592208, 3887065106], direction 32768, fitness 2946538006
  [4827624976, 3887065106], direction 65536, fitness 2946505238
  [4827690512, 3887065106], direction 131072, fitness 2946439702
  [4827821584, 3887065106], direction 262144, fitness 2946308630
  [4828083728, 3887065106], direction 524288, fitness 2946046486
  [4828608016, 3887065106], direction 1048576, fitness 2945522198
  [4829656592, 3887065106], direction 2097152, fitness 2944473622
  [4831753744, 3887065106], direction 4194304, fitness 2942376470
  [4835948048, 3887065106], direction 8388608, fitness 2938182166
  [4844336656, 3887065106], direction 16777216, fitness 2929793558
  [4861113872, 3887065106], direction 33554432, fitness 2913016342
  [4894668304, 3887065106], direction 67108864, fitness 2879461910
  [4961777168, 3887065106], direction 134217728, fitness 2812353046
  [5095994896, 3887065106], direction 268435456, fitness 2678135318
  [5364430352, 3887065106], direction 536870912, fitness 2409699862
  [5901301264, 3887065106], direction 1073741824, fitness 1872828950
  [6975043088, 3887065106], direction 2147483648, fitness 799087126
Current parameter: 0
Trying +1 at fitness 799087126: [6975043089, 3887065106]
  [6975043089, 3887065106], direction 2, fitness 799087125
  [6975043091, 3887065106], direction 4, fitness 799087123
  [6975043095, 3887065106], direction 8, fitness 799087119
  [6975043103, 3887065106], direction 16, fitness 799087111
  [6975043119, 3887065106], direction 32, fitness 799087095
  [6975043151, 3887065106], direction 64, fitness 799087063
  [6975043215, 3887065106], direction 128, fitness 799086999
  [6975043343, 3887065106], direction 256, fitness 799086871
  [6975043599, 3887065106], direction 512, fitness 799086615
  [6975044111, 3887065106], direction 1024, fitness 799086103
  [6975045135, 3887065106], direction 2048, fitness 799085079
  [6975047183, 3887065106], direction 4096, fitness 799083031
  [6975051279, 3887065106], direction 8192, fitness 799078935
  [6975059471, 3887065106], direction 16384, fitness 799070743
  [6975075855, 3887065106], direction 32768, fitness 799054359
  [6975108623, 3887065106], direction 65536, fitness 799021591
  [6975174159, 3887065106], direction 131072, fitness 798956055
  [6975305231, 3887065106], direction 262144, fitness 798824983
  [6975567375, 3887065106], direction 524288, fitness 798562839
  [6976091663, 3887065106], direction 1048576, fitness 798038551
  [6977140239, 3887065106], direction 2097152, fitness 796989975
  [6979237391, 3887065106], direction 4194304, fitness 794892823
```

```
  [6983431695, 3887065106], direction 8388608, fitness 790698519
  [6991820303, 3887065106], direction 16777216, fitness 782309911
  [7008597519, 3887065106], direction 33554432, fitness 765532695
  [7042151951, 3887065106], direction 67108864, fitness 731978263
  [7109260815, 3887065106], direction 134217728, fitness 664869399
  [7243478543, 3887065106], direction 268435456, fitness 530651671
  [7511913999, 3887065106], direction 536870912, fitness 262216215
Current parameter: 0
Trying +1 at fitness 262216215: [7511914000, 3887065106]
  [7511914000, 3887065106], direction 2, fitness 262216214
  [7511914002, 3887065106], direction 4, fitness 262216212
  [7511914006, 3887065106], direction 8, fitness 262216208
  [7511914014, 3887065106], direction 16, fitness 262216200
  [7511914030, 3887065106], direction 32, fitness 262216184
  [7511914062, 3887065106], direction 64, fitness 262216152
  [7511914126, 3887065106], direction 128, fitness 262216088
  [7511914254, 3887065106], direction 256, fitness 262215960
  [7511914510, 3887065106], direction 512, fitness 262215704
  [7511915022, 3887065106], direction 1024, fitness 262215192
  [7511916046, 3887065106], direction 2048, fitness 262214168
  [7511918094, 3887065106], direction 4096, fitness 262212120
  [7511922190, 3887065106], direction 8192, fitness 262208024
  [7511930382, 3887065106], direction 16384, fitness 262199832
  [7511946766, 3887065106], direction 32768, fitness 262183448
  [7511979534, 3887065106], direction 65536, fitness 262150680
  [7512045070, 3887065106], direction 131072, fitness 262085144
  [7512176142, 3887065106], direction 262144, fitness 261954072
  [7512438286, 3887065106], direction 524288, fitness 261691928
  [7512962574, 3887065106], direction 1048576, fitness 261167640
  [7514011150, 3887065106], direction 2097152, fitness 260119064
  [7516108302, 3887065106], direction 4194304, fitness 258021912
  [7520302606, 3887065106], direction 8388608, fitness 253827608
  [7528691214, 3887065106], direction 16777216, fitness 245439000
  [7545468430, 3887065106], direction 33554432, fitness 228661784
  [7579022862, 3887065106], direction 67108864, fitness 195107352
  [7646131726, 3887065106], direction 134217728, fitness 127998488
  [7780349454, 3887065106], direction 268435456, fitness 6219240
Current parameter: 0
Trying +1 at fitness 6219240: [7780349455, 3887065106]
Trying -1 at fitness 6219240: [7780349453, 3887065106]
  [7780349453, 3887065106], direction -2, fitness 6219239
  [7780349451, 3887065106], direction -4, fitness 6219237
  [7780349447, 3887065106], direction -8, fitness 6219233
  [7780349439, 3887065106], direction -16, fitness 6219225
  [7780349423, 3887065106], direction -32, fitness 6219209
  [7780349391, 3887065106], direction -64, fitness 6219177
  [7780349327, 3887065106], direction -128, fitness 6219113
  [7780349199, 3887065106], direction -256, fitness 6218985
```

```
  [7780348943, 3887065106], direction -512, fitness 6218729
  [7780348431, 3887065106], direction -1024, fitness 6218217
  [7780347407, 3887065106], direction -2048, fitness 6217193
  [7780345359, 3887065106], direction -4096, fitness 6215145
  [7780341263, 3887065106], direction -8192, fitness 6211049
  [7780333071, 3887065106], direction -16384, fitness 6202857
  [7780316687, 3887065106], direction -32768, fitness 6186473
  [7780283919, 3887065106], direction -65536, fitness 6153705
  [7780218383, 3887065106], direction -131072, fitness 6088169
  [7780087311, 3887065106], direction -262144, fitness 5957097
  [7779825167, 3887065106], direction -524288, fitness 5694953
  [7779300879, 3887065106], direction -1048576, fitness 5170665
  [7778252303, 3887065106], direction -2097152, fitness 4122089
  [7776155151, 3887065106], direction -4194304, fitness 2024937
Current parameter: 0
Trying +1 at fitness 2024937: [7776155152, 3887065106]
Trying -1 at fitness 2024937: [7776155150, 3887065106]
  [7776155150, 3887065106], direction -2, fitness 2024936
  [7776155148, 3887065106], direction -4, fitness 2024934
  [7776155144, 3887065106], direction -8, fitness 2024930
  [7776155136, 3887065106], direction -16, fitness 2024922
  [7776155120, 3887065106], direction -32, fitness 2024906
  [7776155088, 3887065106], direction -64, fitness 2024874
  [7776155024, 3887065106], direction -128, fitness 2024810
  [7776154896, 3887065106], direction -256, fitness 2024682
  [7776154640, 3887065106], direction -512, fitness 2024426
  [7776154128, 3887065106], direction -1024, fitness 2023914
  [7776153104, 3887065106], direction -2048, fitness 2022890
  [7776151056, 3887065106], direction -4096, fitness 2020842
  [7776146960, 3887065106], direction -8192, fitness 2016746
  [7776138768, 3887065106], direction -16384, fitness 2008554
  [7776122384, 3887065106], direction -32768, fitness 1992170
  [7776089616, 3887065106], direction -65536, fitness 1959402
  [7776024080, 3887065106], direction -131072, fitness 1893866
  [7775893008, 3887065106], direction -262144, fitness 1762794
  [7775630864, 3887065106], direction -524288, fitness 1500650
  [7775106576, 3887065106], direction -1048576, fitness 976362
  [7774058000, 3887065106], direction -2097152, fitness 72214
Current parameter: 0
Trying +1 at fitness 72214: [7774058001, 3887065106]
  [7774058001, 3887065106], direction 2, fitness 72213
  [7774058003, 3887065106], direction 4, fitness 72211
  [7774058007, 3887065106], direction 8, fitness 72207
  [7774058015, 3887065106], direction 16, fitness 72199
  [7774058031, 3887065106], direction 32, fitness 72183
  [7774058063, 3887065106], direction 64, fitness 72151
  [7774058127, 3887065106], direction 128, fitness 72087
  [7774058255, 3887065106], direction 256, fitness 71959
```

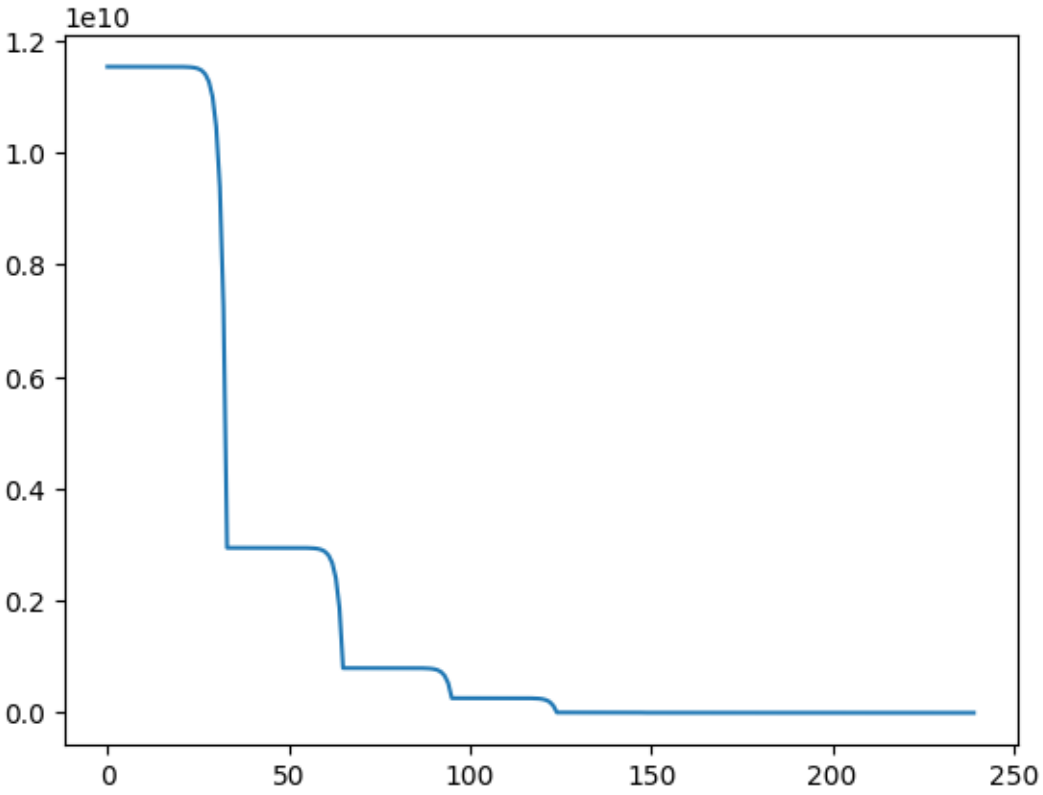
```
  [7774058511, 3887065106], direction 512, fitness 71703
  [7774059023, 3887065106], direction 1024, fitness 71191
  [7774060047, 3887065106], direction 2048, fitness 70167
  [7774062095, 3887065106], direction 4096, fitness 68119
  [7774066191, 3887065106], direction 8192, fitness 64023
  [7774074383, 3887065106], direction 16384, fitness 55831
  [7774090767, 3887065106], direction 32768, fitness 39447
  [7774123535, 3887065106], direction 65536, fitness 6679
Current parameter: 0
Trying +1 at fitness 6679: [7774123536, 3887065106]
  [7774123536, 3887065106], direction 2, fitness 6678
  [7774123538, 3887065106], direction 4, fitness 6676
  [7774123542, 3887065106], direction 8, fitness 6672
  [7774123550, 3887065106], direction 16, fitness 6664
  [7774123566, 3887065106], direction 32, fitness 6648
  [7774123598, 3887065106], direction 64, fitness 6616
  [7774123662, 3887065106], direction 128, fitness 6552
  [7774123790, 3887065106], direction 256, fitness 6424
  [7774124046, 3887065106], direction 512, fitness 6168
  [7774124558, 3887065106], direction 1024, fitness 5656
  [7774125582, 3887065106], direction 2048, fitness 4632
  [7774127630, 3887065106], direction 4096, fitness 2584
  [7774131726, 3887065106], direction 8192, fitness 1512
Current parameter: 0
Trying +1 at fitness 1512: [7774131727, 3887065106]
Trying -1 at fitness 1512: [7774131725, 3887065106]
  [7774131725, 3887065106], direction -2, fitness 1511
  [7774131723, 3887065106], direction -4, fitness 1509
  [7774131719, 3887065106], direction -8, fitness 1505
  [7774131711, 3887065106], direction -16, fitness 1497
  [7774131695, 3887065106], direction -32, fitness 1481
  [7774131663, 3887065106], direction -64, fitness 1449
  [7774131599, 3887065106], direction -128, fitness 1385
  [7774131471, 3887065106], direction -256, fitness 1257
  [7774131215, 3887065106], direction -512, fitness 1001
  [7774130703, 3887065106], direction -1024, fitness 489
Current parameter: 0
Trying +1 at fitness 489: [7774130704, 3887065106]
Trying -1 at fitness 489: [7774130702, 3887065106]
  [7774130702, 3887065106], direction -2, fitness 488
  [7774130700, 3887065106], direction -4, fitness 486
  [7774130696, 3887065106], direction -8, fitness 482
  [7774130688, 3887065106], direction -16, fitness 474
  [7774130672, 3887065106], direction -32, fitness 458
  [7774130640, 3887065106], direction -64, fitness 426
  [7774130576, 3887065106], direction -128, fitness 362
  [7774130448, 3887065106], direction -256, fitness 234
  [7774130192, 3887065106], direction -512, fitness 22
```

```
Current parameter: 0
Trying +1 at fitness 22: [7774130193, 3887065106]
  [7774130193, 3887065106], direction 2, fitness 21
  [7774130195, 3887065106], direction 4, fitness 19
  [7774130199, 3887065106], direction 8, fitness 15
  [7774130207, 3887065106], direction 16, fitness 7
Current parameter: 0
Trying +1 at fitness 7: [7774130208, 3887065106]
  [7774130208, 3887065106], direction 2, fitness 6
  [7774130210, 3887065106], direction 4, fitness 4
  [7774130214, 3887065106], direction 8, fitness 0
Current parameter: 0
Trying +1 at fitness 0: [7774130215, 3887065106]
Trying -1 at fitness 0: [7774130213, 3887065106]
Current parameter: 1
Trying +1 at fitness 0: [7774130214, 3887065107]
Trying -1 at fitness 0: [7774130214, 3887065105]
Solution fitness 0: [7774130214, 3887065106]
```

[32]:
```
[7774130214, 3887065106]
```

[33]:
```
plt.plot(fitness_values)
```

[33]:
```
[<matplotlib.lines.Line2D at 0x1118c9e40>]
```



1.3 Program Instrumentation

Deriving fitness functions is not quite so easy. Of course we could come up with an equation that captures the relation between the sides of the triangle, but then essentially we would need to reproduce the entire program logic again in a function, which certainly does not help generalising to other programs. For example, consider how the fitness function would look like if the comparison was not made on the input parameters, but on values derived through computation within the function under test. Ideally, what we would want is to be able to pick a point in the program and come up with a fitness function automatically that describes how close we are to reaching this point.

There are two central ideas in order to achieve this:

- First, rather than trying to guess how close a program inputs gets to a target statement, we simply *run* the program with the input and observe how close it actually gets.
- Second, during the execution we keep track of distance estimates like the one we calculated for the `test_me` function whenever we come across conditional statements.

In order to observe what an execution does, we need to *instrument* the program: We add new code immediately before or after the branching condition to keep track of the values observed and calculate the distance using these values.

Let's first consider what is done here conceptually. We first define a global variable in which we will store the distance, so that we can access it after the execution:

[34]:
```
distance = 0
```

Now the instrumented version just has to update the global variable immediately before executing the branching condition:

[35]:
```
def test_me_instrumented(x, y):
    global distance
    distance = abs(x - 2 * (y + 1))
    if x == 2 * (y + 1):
        return True
    else:
        return False
```

Let's try this out for a couple of example values:

[36]:
```
test_me_instrumented(0, 0)
distance
```

[36]:
```
2
```

[37]:
```
test_me_instrumented(22, 10)
distance
```

[37]:
```
0
```

Using this instrumented version of `test_me()`, we can define a fitness function which simply calculates the distance for the condition being true:

[38]:
```
def get_fitness(individual):
    global distance
    test_me_instrumented(*individual)
    fitness = distance
    return fitness
```

Let's try this on some example inputs:

[39]:
```
get_fitness([0, 0])
```

[39]:
```
2
```

When we have reached the target branch, the distance will be 0:

[40]:
```
get_fitness([22, 10])
```

[40]:
```
0
```

When implementing the instrumentation, we need to consider that the branching condition may have side-effects. For example, suppose that the branching condition were `x == 2 * foo(y)`, where `foo()` is a function that takes an integer as input. Naively instrumenting would lead to the following code:

```
distance = abs(x - 2 * foo(y))
if x == 2 * foo(y):
...
```

Thus, the instrumentation would lead to `foo()` being executed *twice*. Suppose `foo()` changes the state of the system (e.g., by printing something, accessing the file system, changing some state variables, etc.), then clearly invoking `foo()` a second time is a bad idea. One way to overcome this problem is to *transform* the conditions, rather than *adding* tracing calls. For example, one can create temporary variables that hold the values necessary for the distance calculation and then use these in the branching condition:

```
tmp1 = x
tmp2 = 2 * foo(y)
distance = compute_distance(tmp1, tmp2)
if tmp1 == tmp2:
...
```

[41]:
```
def evaluate_equals(op1, op2):
    global distance
    distance = abs(op1 - op2)
    if distance == 0:
        return True
    else:
```

```
        return False
```

Now the aim would be to transform the program automatically such that it looks like so:

[42]:
```
def test_me_instrumented(x, y):
    tmp1 = x
    tmp2 = 2 * (y + 1)
    if evaluate_equals(tmp1, tmp2):
        return True
    else:
        return False
```

Replacing comparisons automatically is actually quite easy in Python, using the abstract syntax tree (AST) of the program. In the AST, a comparison will typically be a tree node with an operator attribute and two children for the left-hand and right-hand operators. To replace such comparisons with a call to `calculate_distance()` one simply needs to replace the comparison node in the AST with a function call node, and this is what the `BranchTransformer` class does using a NodeTransformer from Python's `ast` module:

[43]:
```
import ast
```

[44]:
```
class BranchTransformer(ast.NodeTransformer):

    def visit_FunctionDef(self, node):
        node.name = node.name + "_instrumented"
        return self.generic_visit(node)

    def visit_Compare(self, node):
        if not isinstance(node.ops[0], ast.Eq):
            return node

        return ast.Call(func=ast.Name("evaluate_equals", ast.Load()),
                        args=[node.left,
                              node.comparators[0]],
                        keywords=[],
                        starargs=None,
                        kwargs=None)
```

The `BranchTransformer` parses a target Python program using the built-in parser `ast.parse()`, which returns the AST. Python provides an API to traverse and modify this AST. To replace the comparison with a function call we use an `ast.NodeTransformer`, which uses the visitor pattern where there is one `visit_*` function for each type of node in the AST. As we are interested in replacing comparisons, we override `visit_Compare`, where instead of the original comparison node we return a new node of type `ast.Func`, which is a function call node. The first parameter of this node is the name of the function `calculate_distance`, and the arguments are the two operands that our `calculate_distance` function expects.

You will notice that we also override `visit_FunctionDef`; this is just to change the name of the method by appending `_instrumented`, so that we can continue to use the original function together

with the instrumented one.

The following code parses the source code of the `test_me()` function to an AST, then transforms it, and prints it out again (using the `to_source()` function from the `astor` library):

[45]:
```
import inspect
import ast
import astor
```

[46]:
```
source = inspect.getsource(test_me)
node = ast.parse(source)
BranchTransformer().visit(node)

# Make sure the line numbers are ok before printing
node = ast.fix_missing_locations(node)
print(astor.to_source(node))
```

```
def test_me_instrumented(x, y):
    if evaluate_equals(x, 2 * (y + 1)):
        return True
    else:
        return False
```

To calculate a fitness value with the instrumented version, we need to compile the instrumented AST again, which is done using Python's `compile()` function. We then need to make the compiled function accessible, for which we first retrieve the current module from `sys.modules`, and then add the compiled code of the instrumented function to the list of functions of the current module using `exec`. After this, the `cgi_decode_instrumented()` function can be accessed.

[47]:
```
import sys
```

[48]:
```
def create_instrumented_function(f):
    source = inspect.getsource(f)
    node = ast.parse(source)
    node = BranchTransformer().visit(node)

    # Make sure the line numbers are ok so that it compiles
    node = ast.fix_missing_locations(node)

    # Compile and add the instrumented function to the current module
    current_module = sys.modules[__name__]
    code = compile(node, filename="<ast>", mode="exec")
    exec(code, current_module.__dict__)
```

[49]:
```
create_instrumented_function(test_me)
```

[50]:
```
test_me_instrumented(0, 0)
```

[50]:
```
False
```

[51]:
```
distance
```

[51]:
```
2
```

[52]:
```
test_me_instrumented(22, 10)
```

[52]:
```
True
```

[53]:
```
distance
```

[53]:
```
0
```

The estimate for any relational comparison of two values is defined in terms of the *branch distance*. Our `evaluate_equals` function indeed implements the branch distance function for an equality comparison. To generalise this we need similar estimates for other types of relational comparisons. Furthermore, we also have to consider the distance to such conditions evaluating to false, not just to true. Thus, each if-condition actually has two distance estimates, one to estimate how close it is to being true, and one how close it is to being false. If the condition is true, then the true distance is 0; if the condition is false, then the false distance is 0. That is, in a comparison `a == b`, if `a` is smaller than `b`, then the false distance is `0` by definition.

The following table shows how to calculate the distance for different types of comparisons:

Condition	Distance True	Distance False
a == b	abs(a - b)	1
a != b	1	abs(a - b)
a < b	b - a + 1	a - b
a <= b	b - a	a - b + 1
a > b	a - b + 1	b - a

Note that several of the calculations add a constant `1`. The reason for this is quite simple: Suppose we want to have `a < b` evaluate to true, and let `a = 27` and `b = 27`. The condition is not true, but simply taking the difference would give us a result of `0`. To avoid this, we have to add a constant value. It is not important whether this value is `1` – any positive constant works.

We generalise our `evaluate_equals` function to an `evaluate_condition` function that takes the operator as an additional parameter, and then implements the above table. In contrast to the previous `calculate_equals`, we will now calculate both, the true and the false distance:

[54]:
```
def evaluate_condition(op, lhs, rhs):
    distance_true = 0
    distance_false = 0
    if op == "Eq":
        if lhs == rhs:
            distance_false = 1
        else:
```

```
            distance_true = abs(lhs - rhs)

    # ... code for other types of conditions

    if distance_true == 0:
        return True
    else:
        return False
```

Let's consider a slightly larger function under test. We will use the well known triangle example, originating in Glenford Meyer's classical Art of Software Testing book

[55]:
```
def triangle(a, b, c):
    if a <= 0 or b <= 0 or c <= 0:
        return 4 # invalid

    if a + b <= c or a + c <= b or b + c <= a:
        return 4 # invalid

    if a == b and b == c:
        return 1 # equilateral

    if a == b or b == c or a == c:
        return 2 # isosceles

    return 3 # scalene
```

The function takes as input the length of the three sides of a triangle, and returns a number representing the type of triangle:

[56]:
```
triangle(4,4,4)
```

[56]: 1

Adapting our representation is easy, we just need to correctly set the number of parameters:

[57]:
```
sig = signature(triangle)
num_parameters = len(sig.parameters)
num_parameters
```

[57]: 3

For the `triangle` function, however, we have multiple if-conditions; we have to add instrumentation to each of these using `evaluate_condition`. We also need to generalise from our global `distance` variable, since we now have two distance values per branch, and potentially multiple branches. Furthermore, a condition might be executed multiple times within a single execution (e.g., if it is in a loop), so rather than storing all values, we will only keep the *minimum* value observed for each condition:

[58]:
```
distances_true = {}
distances_false = {}
```

[59]:
```
def update_maps(condition_num, d_true, d_false):
    global distances_true, distances_false

    if condition_num in distances_true.keys():
        distances_true[condition_num] = min(distances_true[condition_num],␣
 ↪d_true)
    else:
        distances_true[condition_num] = d_true

    if condition_num in distances_false.keys():
        distances_false[condition_num] = min(distances_false[condition_num],␣
 ↪d_false)
    else:
        distances_false[condition_num] = d_false
```

Now we need to finish implementing the `evaluate_condition` function. We add yet another parameter to denote the ID of the branch we are instrumenting:

[60]:
```
def evaluate_condition(num, op, lhs, rhs):
    distance_true = 0
    distance_false = 0

    # Make sure the distance can be calculated on number and character
    # comparisons (needed for cgi_decode later)
    if isinstance(lhs, str):
        lhs = ord(lhs)
    if isinstance(rhs, str):
        rhs = ord(rhs)

    if op == "Eq":
        if lhs == rhs:
            distance_false = 1
        else:
            distance_true = abs(lhs - rhs)

    elif op == "Gt":
        if lhs > rhs:
            distance_false = lhs - rhs
        else:
            distance_true = rhs - lhs + 1
    elif op == "Lt":
        if lhs < rhs:
            distance_false = rhs - lhs
        else:
```

```
            distance_true = lhs - rhs + 1
    elif op == "LtE":
        if lhs <= rhs:
            distance_false = rhs - lhs + 1
        else:
            distance_true = lhs - rhs
    # ...
    # handle other comparison operators
    # ...

    elif op == "In":
        minimum = sys.maxsize
        for elem in rhs.keys():
            distance = abs(lhs - ord(elem))
            if distance < minimum:
                minimum = distance

        distance_true = minimum
        if distance_true == 0:
            distance_false = 1
    else:
        assert False

    update_maps(num, normalise(distance_true), normalise(distance_false))

    if distance_true == 0:
        return True
    else:
        return False
```

We need to normalise branch distances since different comparisons will be on different scales, and this would bias the search. We will use the normalisation function defined in the previous chapter:

[61]:
```
def normalise(x):
    return x / (1.0 + x)
```

We also need to extend our instrumentation function to take care of all comparisons, and not just equality comparisons:

[62]:
```
import ast
class BranchTransformer(ast.NodeTransformer):

    branch_num = 0

    def visit_FunctionDef(self, node):
        node.name = node.name + "_instrumented"
        return self.generic_visit(node)
```

```
    def visit_Compare(self, node):
        if node.ops[0] in [ast.Is, ast.IsNot, ast.In, ast.NotIn]:
            return node

        self.branch_num += 1
        return ast.Call(func=ast.Name("evaluate_condition", ast.Load()),
                        args=[ast.Num(self.branch_num - 1),
                              ast.Str(node.ops[0].__class__.__name__),
                              node.left,
                              node.comparators[0]],
                        keywords=[],
                        starargs=None,
                        kwargs=None)
```

We can now take a look at the instrumented version of `triangle`:

[63]:
```
source = inspect.getsource(triangle)
node = ast.parse(source)
transformer = BranchTransformer()
transformer.visit(node)

# Make sure the line numbers are ok before printing
node = ast.fix_missing_locations(node)
num_branches = transformer.branch_num

print(astor.to_source(node))
```

```
def triangle_instrumented(a, b, c):
    if evaluate_condition(0, 'LtE', a, 0) or evaluate_condition(1, 'LtE', b, 0
        ) or evaluate_condition(2, 'LtE', c, 0):
        return 4
    if evaluate_condition(3, 'LtE', a + b, c) or evaluate_condition(4,
        'LtE', a + c, b) or evaluate_condition(5, 'LtE', b + c, a):
        return 4
    if evaluate_condition(6, 'Eq', a, b) and evaluate_condition(7, 'Eq', b, c):
        return 1
    if evaluate_condition(8, 'Eq', a, b) or evaluate_condition(9, 'Eq', b, c
        ) or evaluate_condition(10, 'Eq', a, c):
        return 2
    return 3
```

To define an executable version of the instrumented triangle function, we can use our `create_instrumented_function` function again:

[64]:
```
create_instrumented_function(triangle)
```

[65]:
```
triangle_instrumented(4, 4, 4)
```

[65]:
```
1
```

[66]:
```
distances_true
```

[66]:
```
{0: 0.8, 1: 0.8, 2: 0.8, 3: 0.8, 4: 0.8, 5: 0.8, 6: 0.0, 7: 0.0}
```

[67]:
```
distances_false
```

[67]:
```
{0: 0.0, 1: 0.0, 2: 0.0, 3: 0.0, 4: 0.0, 5: 0.0, 6: 0.5, 7: 0.5}
```

The branch distance functions above are defined only for atomic comparisons. However, in the `triangle` program all of the atomic comparisons are part of larger predicates, joined together by `and` and `or` connectors.

For conjunctions the branch distance is defined such that the distance to make `A and B` true equals the sum of the branch distances for `A` and `B`, as both of the two conditions would need to be true. Similarly, the branch distance to make `A or B` true would be the *minimum* of the two branch distances of `A` and `B`, as it suffices if one of the two conditions is true to make the entire expression true (and the false distance would be the sum of false distances of the conditions). For a negation `not A`, we can simply switch from the true distance to the false distance, or vice versa. Since predicates can consist of nested conditions, one would need to recursively calculate the branch distance.

Assume we want to find an input that covers the third if-condition, i.e., it produces a triangle where all sides have equal length. Considering the instrumented version of the triangle function we printed above, in order for this if-condition to evaluate to true we require conditions 0, 1, 2, 3, 4, and 5 to evaluate to false, and 6 and 7 to evaluate to true. Thus, the fitness function for this branch would be the sum of false distances for branches 0-5, and true distances for branches 6 and 7.

[68]:
```
def get_fitness(x):
    # Reset any distance values from previous executions
    global distances_true, distances_false
    distances_true  = {x: 1.0 for x in range(num_branches)}
    distances_false = {x: 1.0 for x in range(num_branches)}

    # Run the function under test
    triangle_instrumented(*x)

    # Sum up branch distances for our specific target branch
    fitness = 0.0
    for branch in [6, 7]:
        fitness += distances_true[branch]

    for branch in [0, 1, 2, 3, 4, 5]:
        fitness += distances_false[branch]

    return fitness
```

[69]:
```
get_fitness([5,5,5])
```

[69]: 0.0

[70]:
```
get_fitness(get_random_individual())
```

[70]: 6.9999999997044995

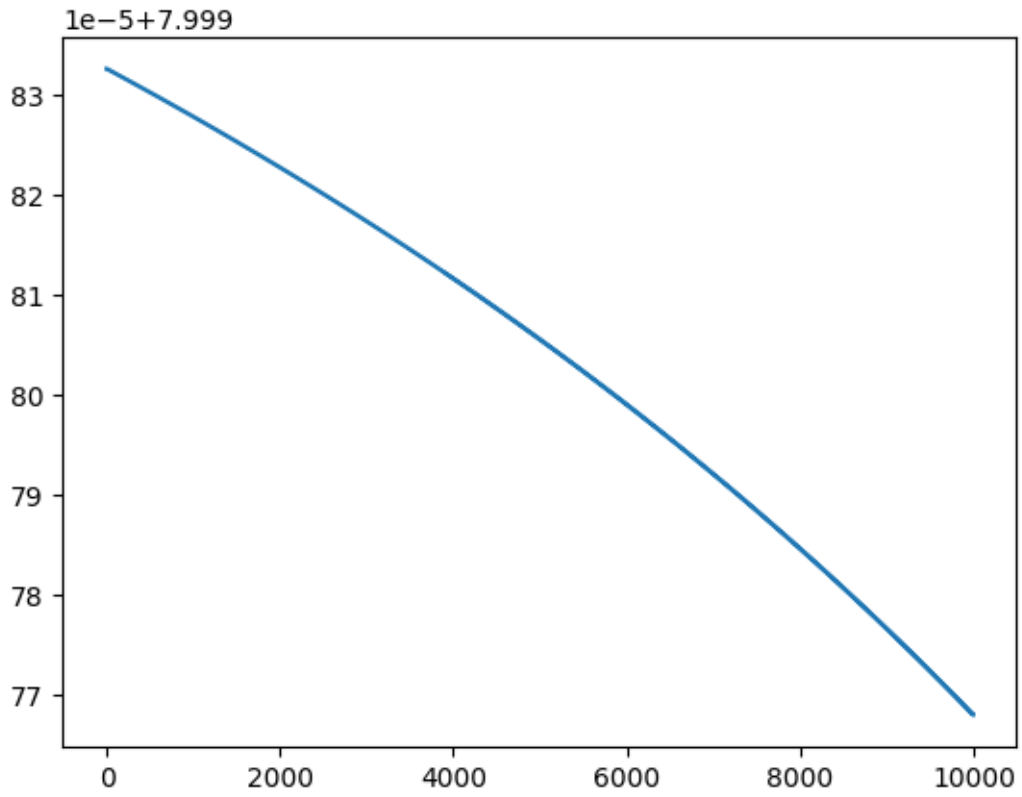
[71]:
```
MAX = 10000
MIN = -MAX
fitness_values = []
max_gen = 1000
hillclimbing()
```

```
Starting at fitness 7.999832663989291: [-5974, -6079, -3017]
Solution fitness after 10002 fitness evaluations: 7.999767927593409: [-4307,
-6079, -3017]
```

[71]: [-4307, -6079, -3017]

[72]:
```
plt.plot(fitness_values)
```

[72]: [<matplotlib.lines.Line2D at 0x11196c490>]



[73]:
```
fitness_values = []
avm()
```

```
Starting at fitness 5.999893910460429: [8490, 7076, -9424]
Current parameter: 0
Trying +1 at fitness 5.999893910460429: [8491, 7076, -9424]
Trying -1 at fitness 5.999893910460429: [8489, 7076, -9424]
Current parameter: 1
Trying +1 at fitness 5.999893910460429: [8490, 7077, -9424]
Trying -1 at fitness 5.999893910460429: [8490, 7075, -9424]
Current parameter: 2
Trying +1 at fitness 5.999893910460429: [8490, 7076, -9423]
  [8490, 7076, -9423], direction 2, fitness 5.999893899204244
  [8490, 7076, -9421], direction 4, fitness 5.999893876684707
  [8490, 7076, -9417], direction 8, fitness 5.999893831616944
  [8490, 7076, -9409], direction 16, fitness 5.999893741366487
  [8490, 7076, -9393], direction 32, fitness 5.9998935604044705
  [8490, 7076, -9361], direction 64, fitness 5.999893196625013
  [8490, 7076, -9297], direction 128, fitness 5.999892461555006
  [8490, 7076, -9169], direction 256, fitness 5.99989096063679
  [8490, 7076, -8913], direction 512, fitness 5.9998878295008415
  [8490, 7076, -8401], direction 1024, fitness 5.99988099488278
  [8490, 7076, -7377], direction 2048, fitness 5.999864480281881
  [8490, 7076, -5329], direction 4096, fitness 5.999812417932846
  [8490, 7076, -1233], direction 8192, fitness 5.99919028340081
  [8490, 7076, 6959], direction 16384, fitness 1.9992932862190813
Current parameter: 2
Trying +1 at fitness 1.9992932862190813: [8490, 7076, 6960]
Trying -1 at fitness 1.9992932862190813: [8490, 7076, 6958]
Current parameter: 0
Trying +1 at fitness 1.9992932862190813: [8491, 7076, 6959]
Trying -1 at fitness 1.9992932862190813: [8489, 7076, 6959]
  [8489, 7076, 6959], direction -2, fitness 1.9992927864214993
  [8487, 7076, 6959], direction -4, fitness 1.9992917847025495
  [8483, 7076, 6959], direction -8, fitness 1.9992897727272727
  [8475, 7076, 6959], direction -16, fitness 1.9992857142857143
  [8459, 7076, 6959], direction -32, fitness 1.9992774566473988
  [8427, 7076, 6959], direction -64, fitness 1.9992603550295858
  [8363, 7076, 6959], direction -128, fitness 1.999223602484472
  [8235, 7076, 6959], direction -256, fitness 1.9991379310344828
  [7979, 7076, 6959], direction -512, fitness 1.9988938053097345
  [7467, 7076, 6959], direction -1024, fitness 1.9974489795918369
Current parameter: 0
Trying +1 at fitness 1.9974489795918369: [7468, 7076, 6959]
Trying -1 at fitness 1.9974489795918369: [7466, 7076, 6959]
  [7466, 7076, 6959], direction -2, fitness 1.9974424552429668
```

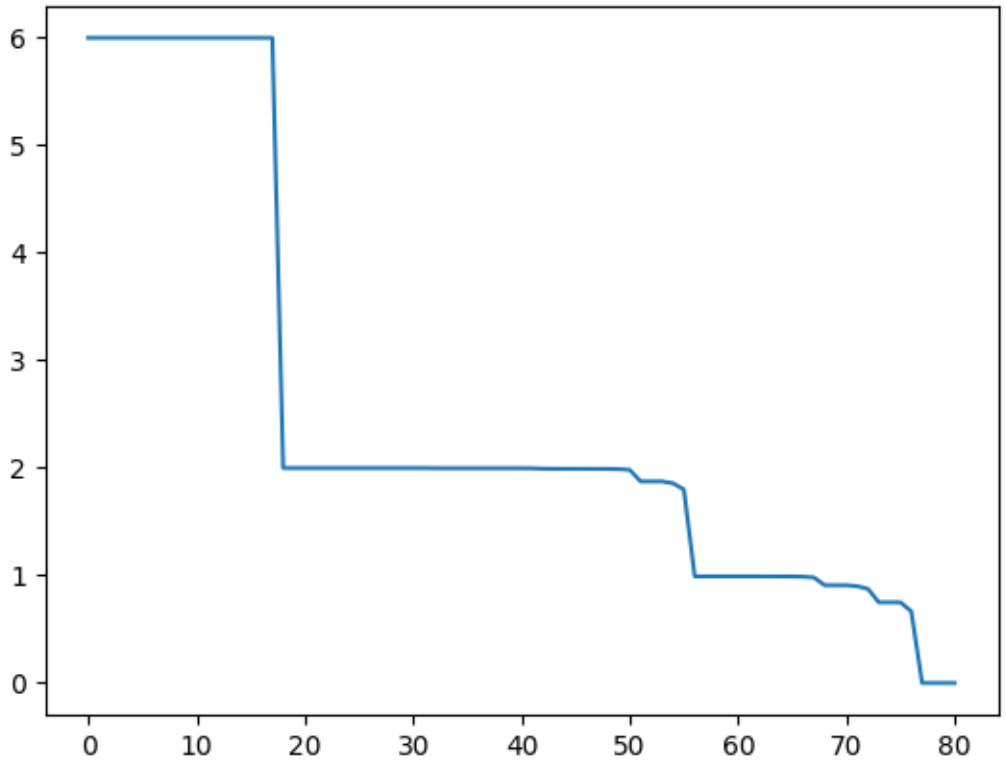
```
  [7464, 7076, 6959], direction -4, fitness 1.9974293059125965
  [7460, 7076, 6959], direction -8, fitness 1.9974025974025973
  [7452, 7076, 6959], direction -16, fitness 1.9973474801061006
  [7436, 7076, 6959], direction -32, fitness 1.997229916897507
  [7404, 7076, 6959], direction -64, fitness 1.9969604863221884
  [7340, 7076, 6959], direction -128, fitness 1.9962264150943396
  [7212, 7076, 6959], direction -256, fitness 1.9927007299270074
  [6956, 7076, 6959], direction -512, fitness 1.9917355371900827
Current parameter: 0
Trying +1 at fitness 1.9917355371900827: [6957, 7076, 6959]
  [6957, 7076, 6959], direction 2, fitness 1.9916666666666667
  [6959, 7076, 6959], direction 4, fitness 1.9915254237288136
  [6963, 7076, 6959], direction 8, fitness 1.9912280701754386
  [6971, 7076, 6959], direction 16, fitness 1.990566037735849
  [6987, 7076, 6959], direction 32, fitness 1.988888888888889
  [7019, 7076, 6959], direction 64, fitness 1.9827586206896552
  [7083, 7076, 6959], direction 128, fitness 1.875
Current parameter: 0
Trying +1 at fitness 1.875: [7084, 7076, 6959]
Trying -1 at fitness 1.875: [7082, 7076, 6959]
  [7082, 7076, 6959], direction -2, fitness 1.8571428571428572
  [7080, 7076, 6959], direction -4, fitness 1.8
  [7076, 7076, 6959], direction -8, fitness 0.9915254237288136
Current parameter: 0
Trying +1 at fitness 0.9915254237288136: [7077, 7076, 6959]
Trying -1 at fitness 0.9915254237288136: [7075, 7076, 6959]
Current parameter: 1
Trying +1 at fitness 0.9915254237288136: [7076, 7077, 6959]
Trying -1 at fitness 0.9915254237288136: [7076, 7075, 6959]
Current parameter: 2
Trying +1 at fitness 0.9915254237288136: [7076, 7076, 6960]
  [7076, 7076, 6960], direction 2, fitness 0.9914529914529915
  [7076, 7076, 6962], direction 4, fitness 0.991304347826087
  [7076, 7076, 6966], direction 8, fitness 0.990990990990991
  [7076, 7076, 6974], direction 16, fitness 0.9902912621359223
  [7076, 7076, 6990], direction 32, fitness 0.9885057471264368
  [7076, 7076, 7022], direction 64, fitness 0.9818181818181818
  [7076, 7076, 7086], direction 128, fitness 0.9090909090909091
Current parameter: 2
Trying +1 at fitness 0.9090909090909091: [7076, 7076, 7087]
Trying -1 at fitness 0.9090909090909091: [7076, 7076, 7085]
  [7076, 7076, 7085], direction -2, fitness 0.9
  [7076, 7076, 7083], direction -4, fitness 0.875
  [7076, 7076, 7079], direction -8, fitness 0.75
Current parameter: 2
Trying +1 at fitness 0.75: [7076, 7076, 7080]
Trying -1 at fitness 0.75: [7076, 7076, 7078]
  [7076, 7076, 7078], direction -2, fitness 0.6666666666666666
```

```
  [7076, 7076, 7076], direction -4, fitness 0.0
Current parameter: 2
Trying +1 at fitness 0.0: [7076, 7076, 7077]
Trying -1 at fitness 0.0: [7076, 7076, 7075]
Solution fitness 0.0: [7076, 7076, 7076]
```

[73]:
```
[7076, 7076, 7076]
```

[74]:
```
plt.plot(fitness_values)
```

[74]:
```
[<matplotlib.lines.Line2D at 0x11199fd30>]
```



Besides the local search algorithms, we can also use evolutionary search in order to find solutions to our test generation problem. We therefore need to define the usual search operators:

[75]:
```
tournament_size = 3
def tournament_selection(population):
    candidates = random.sample(population, tournament_size)
    winner = min(candidates, key = lambda x: get_fitness(x))
    return winner
```

[76]:
```
elite_size = 2
def elitism_standard(population):
```

```
    population.sort(key = lambda k: get_fitness(k))
    return population[:elite_size]
```

[77]:
```
def mutate(solution):
    P_mutate = 1/len(solution)
    mutated = solution[:]
    for position in range(len(solution)):
        if random.random() < P_mutate:
            mutated[position] = int(random.gauss(mutated[position], 20))
    return mutated
```

[78]:
```
def singlepoint_crossover(parent1, parent2):
    pos = random.randint(0, len(parent1))
    offspring1 = parent1[:pos] + parent2[pos:]
    offspring2 = parent2[:pos] + parent1[pos:]
    return (offspring1, offspring2)
```

[79]:
```
population_size = 20
P_xover = 0.7
max_gen = 100
selection = tournament_selection
crossover = singlepoint_crossover
elitism = elitism_standard
MAX = 1000
MIN = -MAX
```

[80]:
```
def ga():
    population = [get_random_individual() for _ in range(population_size)]
    best_fitness = sys.maxsize
    for p in population:
        fitness = get_fitness(p)
        if fitness < best_fitness:
            best_fitness = fitness
            best_solution = p
    print(f"Iteration 0, best fitness: {best_fitness}")

    for iteration in range(max_gen):
        fitness_values.append(best_fitness)
        new_population = elitism(population)
        while len(new_population) < len(population):
            parent1 = selection(population)
            parent2 = selection(population)

            if random.random() < P_xover:
                offspring1, offspring2 = crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1, parent2
```

```
            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            new_population.append(offspring1)
            new_population.append(offspring2)

        population = new_population
        for p in population:
            fitness = get_fitness(p)
            if fitness < best_fitness:
                best_fitness = fitness
                best_solution = p
        print(f"Iteration {iteration}, best fitness: {best_fitness}, size␣
↪{len(best_solution)}")

    return best_solution
```

[81]:
```
fitness_values = []
ga()
```

```
Iteration 0, best fitness: 1.9932885906040267
Iteration 0, best fitness: 1.9932885906040267, size 3
Iteration 1, best fitness: 1.9919354838709677, size 3
Iteration 2, best fitness: 1.990566037735849, size 3
Iteration 3, best fitness: 1.9859154929577465, size 3
Iteration 4, best fitness: 1.9859154929577465, size 3
Iteration 5, best fitness: 1.9565217391304348, size 3
Iteration 6, best fitness: 1.9565217391304348, size 3
Iteration 7, best fitness: 1.8, size 3
Iteration 8, best fitness: 1.8, size 3
Iteration 9, best fitness: 0.99830220713073, size 3
Iteration 10, best fitness: 0.9981378026070763, size 3
Iteration 11, best fitness: 0.9981378026070763, size 3
Iteration 12, best fitness: 0.9981378026070763, size 3
Iteration 13, best fitness: 0.9981378026070763, size 3
Iteration 14, best fitness: 0.9980276134122288, size 3
Iteration 15, best fitness: 0.9980276134122288, size 3
Iteration 16, best fitness: 0.9979035639412998, size 3
Iteration 17, best fitness: 0.9979035639412998, size 3
Iteration 18, best fitness: 0.9978813559322034, size 3
Iteration 19, best fitness: 0.9978813559322034, size 3
Iteration 20, best fitness: 0.9978354978354979, size 3
Iteration 21, best fitness: 0.9978354978354979, size 3
Iteration 22, best fitness: 0.9978354978354979, size 3
Iteration 23, best fitness: 0.9978165938864629, size 3
Iteration 24, best fitness: 0.9978165938864629, size 3
Iteration 25, best fitness: 0.9978165938864629, size 3
```

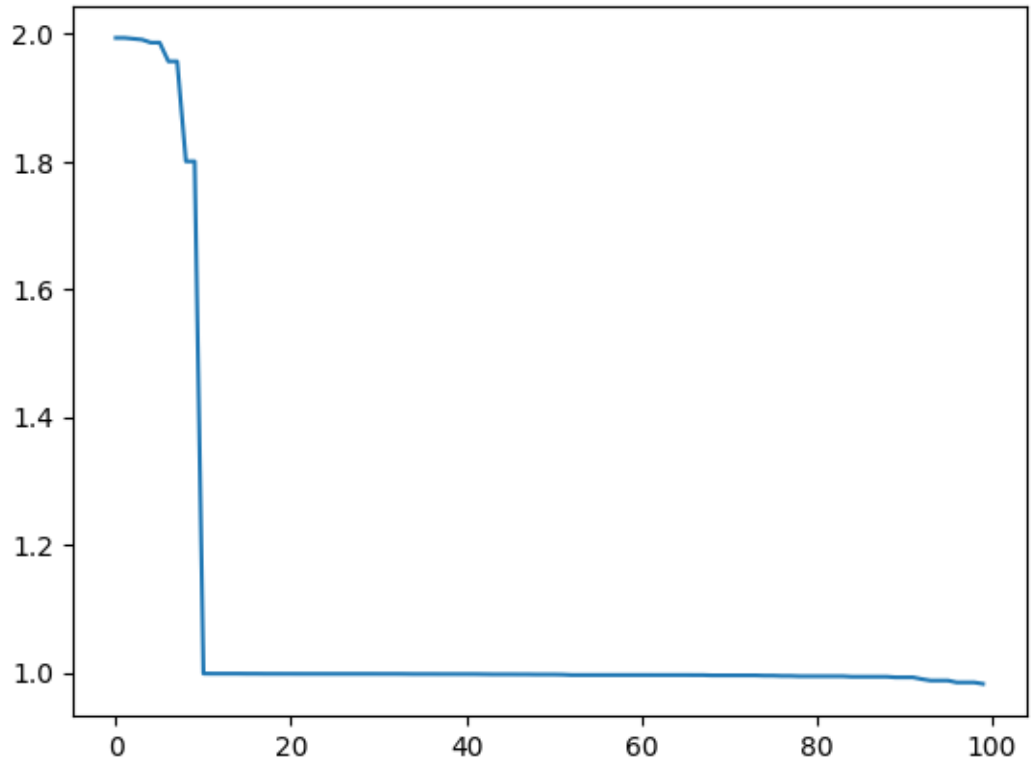
```
Iteration 26, best fitness: 0.9978165938864629, size 3
Iteration 27, best fitness: 0.9978165938864629, size 3
Iteration 28, best fitness: 0.9978165938864629, size 3
Iteration 29, best fitness: 0.9977777777777778, size 3
Iteration 30, best fitness: 0.9977777777777778, size 3
Iteration 31, best fitness: 0.9977777777777778, size 3
Iteration 32, best fitness: 0.9976958525345622, size 3
Iteration 33, best fitness: 0.9975550122249389, size 3
Iteration 34, best fitness: 0.9975550122249389, size 3
Iteration 35, best fitness: 0.9975550122249389, size 3
Iteration 36, best fitness: 0.9975062344139651, size 3
Iteration 37, best fitness: 0.9975062344139651, size 3
Iteration 38, best fitness: 0.9974811083123426, size 3
Iteration 39, best fitness: 0.9974811083123426, size 3
Iteration 40, best fitness: 0.9974424552429667, size 3
Iteration 41, best fitness: 0.997289972899729, size 3
Iteration 42, best fitness: 0.9971098265895953, size 3
Iteration 43, best fitness: 0.9971098265895953, size 3
Iteration 44, best fitness: 0.9971098265895953, size 3
Iteration 45, best fitness: 0.9970414201183432, size 3
Iteration 46, best fitness: 0.9970414201183432, size 3
Iteration 47, best fitness: 0.9968652037617555, size 3
Iteration 48, best fitness: 0.9968553459119497, size 3
Iteration 49, best fitness: 0.9968553459119497, size 3
Iteration 50, best fitness: 0.9966442953020134, size 3
Iteration 51, best fitness: 0.9960159362549801, size 3
Iteration 52, best fitness: 0.9960159362549801, size 3
Iteration 53, best fitness: 0.9960159362549801, size 3
Iteration 54, best fitness: 0.9960159362549801, size 3
Iteration 55, best fitness: 0.9960159362549801, size 3
Iteration 56, best fitness: 0.9960159362549801, size 3
Iteration 57, best fitness: 0.9960159362549801, size 3
Iteration 58, best fitness: 0.9960159362549801, size 3
Iteration 59, best fitness: 0.9960159362549801, size 3
Iteration 60, best fitness: 0.996, size 3
Iteration 61, best fitness: 0.996, size 3
Iteration 62, best fitness: 0.996, size 3
Iteration 63, best fitness: 0.996, size 3
Iteration 64, best fitness: 0.996, size 3
Iteration 65, best fitness: 0.9959183673469387, size 3
Iteration 66, best fitness: 0.9959183673469387, size 3
Iteration 67, best fitness: 0.995475113122172, size 3
Iteration 68, best fitness: 0.995475113122172, size 3
Iteration 69, best fitness: 0.995475113122172, size 3
Iteration 70, best fitness: 0.995475113122172, size 3
Iteration 71, best fitness: 0.995475113122172, size 3
Iteration 72, best fitness: 0.995475113122172, size 3
Iteration 73, best fitness: 0.995049504950495, size 3
```

```
Iteration 74, best fitness: 0.995049504950495, size 3
Iteration 75, best fitness: 0.9945945945945946, size 3
Iteration 76, best fitness: 0.9945945945945946, size 3
Iteration 77, best fitness: 0.9941176470588236, size 3
Iteration 78, best fitness: 0.9941176470588236, size 3
Iteration 79, best fitness: 0.9941176470588236, size 3
Iteration 80, best fitness: 0.9941176470588236, size 3
Iteration 81, best fitness: 0.9941176470588236, size 3
Iteration 82, best fitness: 0.9941176470588236, size 3
Iteration 83, best fitness: 0.9933333333333333, size 3
Iteration 84, best fitness: 0.9933333333333333, size 3
Iteration 85, best fitness: 0.9933333333333333, size 3
Iteration 86, best fitness: 0.9933333333333333, size 3
Iteration 87, best fitness: 0.9933333333333333, size 3
Iteration 88, best fitness: 0.9923664122137404, size 3
Iteration 89, best fitness: 0.9923664122137404, size 3
Iteration 90, best fitness: 0.9923664122137404, size 3
Iteration 91, best fitness: 0.9895833333333334, size 3
Iteration 92, best fitness: 0.9873417721518988, size 3
Iteration 93, best fitness: 0.9873417721518988, size 3
Iteration 94, best fitness: 0.9873417721518988, size 3
Iteration 95, best fitness: 0.984375, size 3
Iteration 96, best fitness: 0.984375, size 3
Iteration 97, best fitness: 0.984375, size 3
Iteration 98, best fitness: 0.9821428571428571, size 3
Iteration 99, best fitness: 0.9795918367346939, size 3
```

[81]:
```
[841, 841, 793]
```

[82]:
```
plt.plot(fitness_values)
```

[82]:
```
[<matplotlib.lines.Line2D at 0x111a0a500>]
```



We set `MAX` to a value as low as 1000, because the optimisation with our small mutational steps may take long to achieve that multiple values are equal, which some of the branches of the triangle program require (such as the one we are optimising for currently).

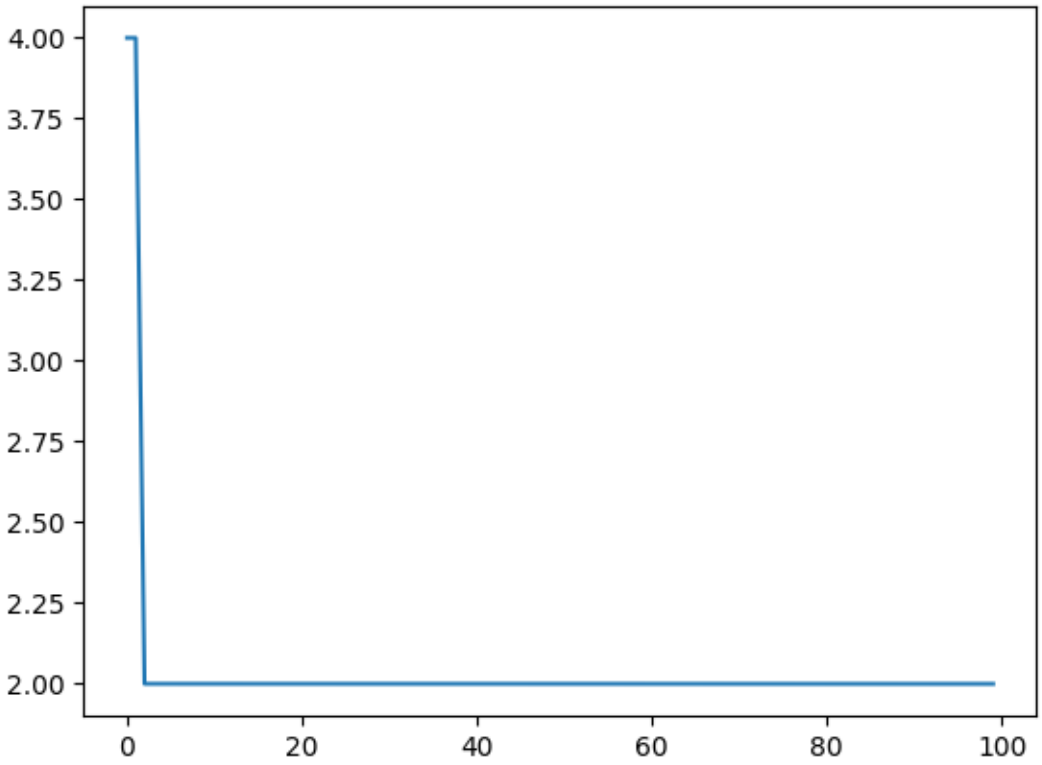
[83]:
```
MAX = 100000
MIN = -MAX
fitness_values = []
ga()
plt.plot(fitness_values)
```

```
Iteration 0, best fitness: 3.999737049697607
Iteration 0, best fitness: 3.999737049697607, size 3
Iteration 1, best fitness: 1.9997445068983137, size 3
Iteration 2, best fitness: 1.9997445068983137, size 3
Iteration 3, best fitness: 1.9997424008243172, size 3
Iteration 4, best fitness: 1.9997416688194265, size 3
Iteration 5, best fitness: 1.9997390396659709, size 3
Iteration 6, best fitness: 1.9997390396659709, size 3
Iteration 7, best fitness: 1.9997365648050578, size 3
Iteration 8, best fitness: 1.9997329773030708, size 3
Iteration 9, best fitness: 1.9997329773030708, size 3
Iteration 10, best fitness: 1.9997329773030708, size 3
Iteration 11, best fitness: 1.9997316876844646, size 3
```

```
Iteration 12, best fitness: 1.9997298027560118, size 3
Iteration 13, best fitness: 1.9997298027560118, size 3
Iteration 14, best fitness: 1.9997282608695652, size 3
Iteration 15, best fitness: 1.9997269997269997, size 3
Iteration 16, best fitness: 1.999725425590335, size 3
Iteration 17, best fitness: 1.99972191323693, size 3
Iteration 18, best fitness: 1.99972191323693, size 3
Iteration 19, best fitness: 1.999719730941704, size 3
Iteration 20, best fitness: 1.999719730941704, size 3
Iteration 21, best fitness: 1.999718864211414, size 3
Iteration 22, best fitness: 1.999716633607254, size 3
Iteration 23, best fitness: 1.9997150997150999, size 3
Iteration 24, best fitness: 1.9997136311569301, size 3
Iteration 25, best fitness: 1.999708284714119, size 3
Iteration 26, best fitness: 1.999708284714119, size 3
Iteration 27, best fitness: 1.9997052755673446, size 3
Iteration 28, best fitness: 1.9997038791827064, size 3
Iteration 29, best fitness: 1.9997014034040013, size 3
Iteration 30, best fitness: 1.9997010463378175, size 3
Iteration 31, best fitness: 1.999697519661222, size 3
Iteration 32, best fitness: 1.9996966019417477, size 3
Iteration 33, best fitness: 1.9996907854050712, size 3
Iteration 34, best fitness: 1.9996907854050712, size 3
Iteration 35, best fitness: 1.9996899224806202, size 3
Iteration 36, best fitness: 1.9996862252902416, size 3
Iteration 37, best fitness: 1.9996854356715947, size 3
Iteration 38, best fitness: 1.9996823379923763, size 3
Iteration 39, best fitness: 1.9996823379923763, size 3
Iteration 40, best fitness: 1.9996782496782497, size 3
Iteration 41, best fitness: 1.9996770025839794, size 3
Iteration 42, best fitness: 1.9996770025839794, size 3
Iteration 43, best fitness: 1.9996757457846952, size 3
Iteration 44, best fitness: 1.999672131147541, size 3
Iteration 45, best fitness: 1.9996684350132625, size 3
Iteration 46, best fitness: 1.9996684350132625, size 3
Iteration 47, best fitness: 1.9996636394214597, size 3
Iteration 48, best fitness: 1.9996612466124661, size 3
Iteration 49, best fitness: 1.9996612466124661, size 3
Iteration 50, best fitness: 1.9996612466124661, size 3
Iteration 51, best fitness: 1.9996600951733514, size 3
Iteration 52, best fitness: 1.9996567112941985, size 3
Iteration 53, best fitness: 1.9996542185338866, size 3
Iteration 54, best fitness: 1.999649245878639, size 3
Iteration 55, best fitness: 1.9996467679265277, size 3
Iteration 56, best fitness: 1.9996417054819062, size 3
Iteration 57, best fitness: 1.9996417054819062, size 3
Iteration 58, best fitness: 1.9996324880558618, size 3
Iteration 59, best fitness: 1.9996324880558618, size 3
```

```
Iteration 60, best fitness: 1.9996290801186944, size 3
Iteration 61, best fitness: 1.9996290801186944, size 3
Iteration 62, best fitness: 1.9996281145407213, size 3
Iteration 63, best fitness: 1.9996234939759037, size 3
Iteration 64, best fitness: 1.9996234939759037, size 3
Iteration 65, best fitness: 1.9996184662342618, size 3
Iteration 66, best fitness: 1.999616564417178, size 3
Iteration 67, best fitness: 1.9996125532739248, size 3
Iteration 68, best fitness: 1.999609832227858, size 3
Iteration 69, best fitness: 1.9996081504702194, size 3
Iteration 70, best fitness: 1.9996055226824456, size 3
Iteration 71, best fitness: 1.9996039603960396, size 3
Iteration 72, best fitness: 1.9996012759170654, size 3
Iteration 73, best fitness: 1.9995983935742971, size 3
Iteration 74, best fitness: 1.9995957962813258, size 3
Iteration 75, best fitness: 1.999591169255928, size 3
Iteration 76, best fitness: 1.9995855781185248, size 3
Iteration 77, best fitness: 1.9995808885163453, size 3
Iteration 78, best fitness: 1.999577524292353, size 3
Iteration 79, best fitness: 1.9995713673381912, size 3
Iteration 80, best fitness: 1.9995704467353952, size 3
Iteration 81, best fitness: 1.9995638901003052, size 3
Iteration 82, best fitness: 1.9995621716287215, size 3
Iteration 83, best fitness: 1.999558693733451, size 3
Iteration 84, best fitness: 1.999558693733451, size 3
Iteration 85, best fitness: 1.9995503597122302, size 3
Iteration 86, best fitness: 1.9995503597122302, size 3
Iteration 87, best fitness: 1.9995446265938068, size 3
Iteration 88, best fitness: 1.9995446265938068, size 3
Iteration 89, best fitness: 1.9995355318160706, size 3
Iteration 90, best fitness: 1.999532710280374, size 3
Iteration 91, best fitness: 1.9995289684408855, size 3
Iteration 92, best fitness: 1.9995183044315992, size 3
Iteration 93, best fitness: 1.9995183044315992, size 3
Iteration 94, best fitness: 1.9995166747220878, size 3
Iteration 95, best fitness: 1.9995131450827652, size 3
Iteration 96, best fitness: 1.9995095635115252, size 3
Iteration 97, best fitness: 1.9995064165844028, size 3
Iteration 98, best fitness: 1.9994954591321896, size 3
Iteration 99, best fitness: 1.9994786235662148, size 3
```

[83]: `[<matplotlib.lines.Line2D at 0x111a648e0>]`



Different mutation operators may yield different results: For example, rather than just adding random noise to the individual parameters, we can also probabilistically copy values from other parameters:

[84]:
```
def mutate(solution):
    P_mutate = 1/len(solution)
    mutated = solution[:]
    for position in range(len(solution)):
        if random.random() < P_mutate:
            if random.random() < 0.9:
                mutated[position] = int(random.gauss(mutated[position], 20))
            else:
                mutated[position] = random.choice(solution)
    return mutated
```

Let's see the performance of the resulting algorithm:

[85]:
```
fitness_values = []
MAX = 100000
MIN = -MAX
ga()
```

```
Iteration 0, best fitness: 1.9999603709281129
```

```
Iteration 0, best fitness: 1.999395039322444, size 3
Iteration 1, best fitness: 1.999395039322444, size 3
Iteration 2, best fitness: 0.9999570225201995, size 3
Iteration 3, best fitness: 0.9999570225201995, size 3
Iteration 4, best fitness: 0.99812734082397, size 3
Iteration 5, best fitness: 0.99812734082397, size 3
Iteration 6, best fitness: 0.997979797979798, size 3
Iteration 7, best fitness: 0.997979797979798, size 3
Iteration 8, best fitness: 0.0, size 3
Iteration 9, best fitness: 0.0, size 3
Iteration 10, best fitness: 0.0, size 3
Iteration 11, best fitness: 0.0, size 3
Iteration 12, best fitness: 0.0, size 3
Iteration 13, best fitness: 0.0, size 3
Iteration 14, best fitness: 0.0, size 3
Iteration 15, best fitness: 0.0, size 3
Iteration 16, best fitness: 0.0, size 3
Iteration 17, best fitness: 0.0, size 3
Iteration 18, best fitness: 0.0, size 3
Iteration 19, best fitness: 0.0, size 3
Iteration 20, best fitness: 0.0, size 3
Iteration 21, best fitness: 0.0, size 3
Iteration 22, best fitness: 0.0, size 3
Iteration 23, best fitness: 0.0, size 3
Iteration 24, best fitness: 0.0, size 3
Iteration 25, best fitness: 0.0, size 3
Iteration 26, best fitness: 0.0, size 3
Iteration 27, best fitness: 0.0, size 3
Iteration 28, best fitness: 0.0, size 3
Iteration 29, best fitness: 0.0, size 3
Iteration 30, best fitness: 0.0, size 3
Iteration 31, best fitness: 0.0, size 3
Iteration 32, best fitness: 0.0, size 3
Iteration 33, best fitness: 0.0, size 3
Iteration 34, best fitness: 0.0, size 3
Iteration 35, best fitness: 0.0, size 3
Iteration 36, best fitness: 0.0, size 3
Iteration 37, best fitness: 0.0, size 3
Iteration 38, best fitness: 0.0, size 3
Iteration 39, best fitness: 0.0, size 3
Iteration 40, best fitness: 0.0, size 3
Iteration 41, best fitness: 0.0, size 3
Iteration 42, best fitness: 0.0, size 3
Iteration 43, best fitness: 0.0, size 3
Iteration 44, best fitness: 0.0, size 3
Iteration 45, best fitness: 0.0, size 3
Iteration 46, best fitness: 0.0, size 3
Iteration 47, best fitness: 0.0, size 3
```

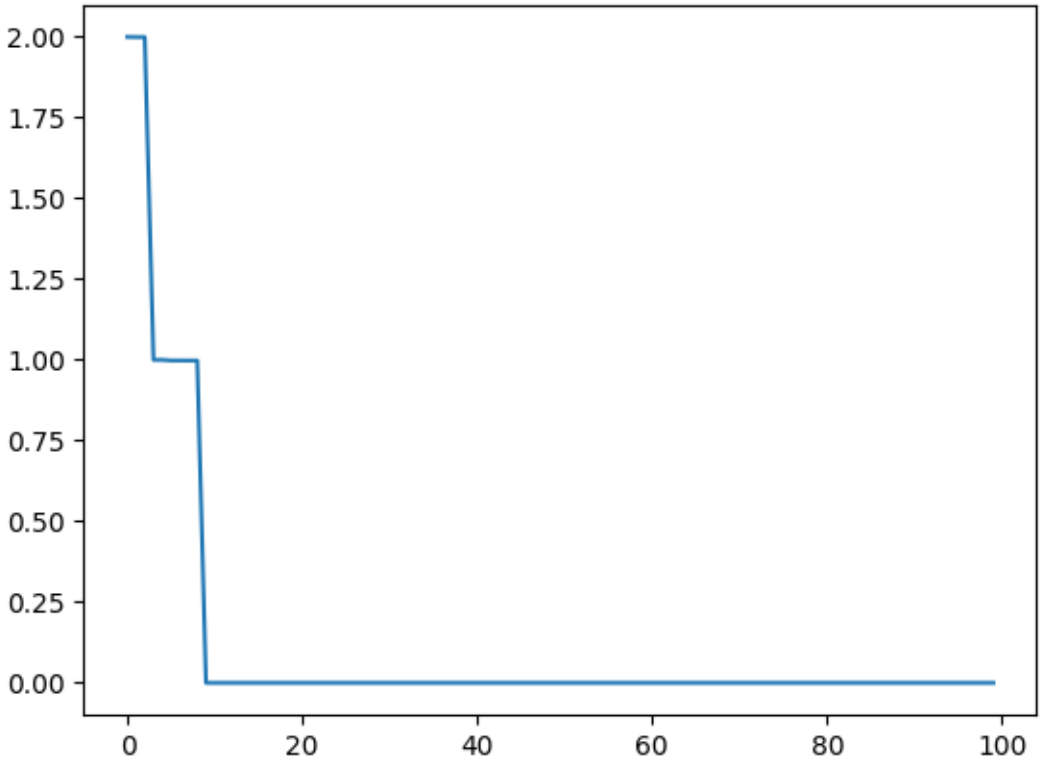
```
Iteration 48, best fitness: 0.0, size 3
Iteration 49, best fitness: 0.0, size 3
Iteration 50, best fitness: 0.0, size 3
Iteration 51, best fitness: 0.0, size 3
Iteration 52, best fitness: 0.0, size 3
Iteration 53, best fitness: 0.0, size 3
Iteration 54, best fitness: 0.0, size 3
Iteration 55, best fitness: 0.0, size 3
Iteration 56, best fitness: 0.0, size 3
Iteration 57, best fitness: 0.0, size 3
Iteration 58, best fitness: 0.0, size 3
Iteration 59, best fitness: 0.0, size 3
Iteration 60, best fitness: 0.0, size 3
Iteration 61, best fitness: 0.0, size 3
Iteration 62, best fitness: 0.0, size 3
Iteration 63, best fitness: 0.0, size 3
Iteration 64, best fitness: 0.0, size 3
Iteration 65, best fitness: 0.0, size 3
Iteration 66, best fitness: 0.0, size 3
Iteration 67, best fitness: 0.0, size 3
Iteration 68, best fitness: 0.0, size 3
Iteration 69, best fitness: 0.0, size 3
Iteration 70, best fitness: 0.0, size 3
Iteration 71, best fitness: 0.0, size 3
Iteration 72, best fitness: 0.0, size 3
Iteration 73, best fitness: 0.0, size 3
Iteration 74, best fitness: 0.0, size 3
Iteration 75, best fitness: 0.0, size 3
Iteration 76, best fitness: 0.0, size 3
Iteration 77, best fitness: 0.0, size 3
Iteration 78, best fitness: 0.0, size 3
Iteration 79, best fitness: 0.0, size 3
Iteration 80, best fitness: 0.0, size 3
Iteration 81, best fitness: 0.0, size 3
Iteration 82, best fitness: 0.0, size 3
Iteration 83, best fitness: 0.0, size 3
Iteration 84, best fitness: 0.0, size 3
Iteration 85, best fitness: 0.0, size 3
Iteration 86, best fitness: 0.0, size 3
Iteration 87, best fitness: 0.0, size 3
Iteration 88, best fitness: 0.0, size 3
Iteration 89, best fitness: 0.0, size 3
Iteration 90, best fitness: 0.0, size 3
Iteration 91, best fitness: 0.0, size 3
Iteration 92, best fitness: 0.0, size 3
Iteration 93, best fitness: 0.0, size 3
Iteration 94, best fitness: 0.0, size 3
Iteration 95, best fitness: 0.0, size 3
```

```
Iteration 96, best fitness: 0.0, size 3
Iteration 97, best fitness: 0.0, size 3
Iteration 98, best fitness: 0.0, size 3
Iteration 99, best fitness: 0.0, size 3
```

[85]:
```
[56569, 56569, 56569]
```

[86]:
```
plt.plot(fitness_values)
```

[86]:
```
[<matplotlib.lines.Line2D at 0x111ad41f0>]
```



In our fitness function, we manually determined which branches need to evaluate which way, and how to sum up the fitness functions. In practice, this can be automated by combining the branch distance metric with the *approach level*, which was introduced (originally named approximation level) in this paper:

Wegener, J., Baresel, A., & Sthamer, H. (2001). Evolutionary test environment for automatic structural testing. Information and software technology, 43(14), 841-854.

The approach level calculates the distances of an execution from a target node in terms of graph distances on the control dependence graph. However, we will not cover the approach level in this chapter.

1.4 Whole Test Suite Optimisation

Besides the question of how the best fitness function for a coverage goal looks like, there are some related questions: How much time should we spend on optimising for each coverage goal? It is possible that some coverage goals are infeasible (e.g., dead code, or infeasible branches), so any time spent on these is wasted, while it may be missing for other goals that are feasible but would need more time. Test cases typically cover multiple goals at the same time; even if a test is optimised for one specific line or branch, it may coincidentally cover others along the execution. Thus, the order in which we select coverage goals for optimisation may influence the overall result, and the number of tests we require. In principle, one way to address these issues would be by casting test generation as a multi-objective optimisation problem, and aiming to produce tests for *all* coverage goals at the same time. However, there is an issue with this: Multi-objective algorithms like the ones we considered in the previous chapter typically work well on 2-3 objectives, but code will generally have many more coverage objectives, rendering classical multi-objective algorithms infeasible (Pareto-dominance happens rarely with higher numbers of objectives). We will therefore now consider some alternatives.

The first alternative we consider is to switch our representation: Rather than optimising individual test cases for individual coverage objectives, we optimise entire test *suites* to cover *all* coverage objectives at the same time. Our encoding thus should describe multiple tests. But how many? This is very much problem specific. Thus, rather than hard coding the number of tests, we will only define an upper bound, and let the search decide what is the necessary number of tests.

[87]:
```
num_tests = 30
```

[88]:
```
def get_random_individual():
    num = random.randint(1, num_tests)
    return [[random.randint(MIN, MAX) for _ in range(num_parameters)] for _ in␣
 ↪range(num)]
```

When applying mutation, we need to be able to modify individual tests as before. To keep things challenging, we will not use our optimised mutation that copies parameters, but aim to achieve the entire optimisation just using small steps:

[89]:
```
def mutate_test(solution):
    P_mutate = 1/len(solution)
    mutated = solution[:]
    for position in range(len(solution)):
        if random.random() < P_mutate:
            mutated[position] = min(MAX, max(MIN, int(random.
 ↪gauss(mutated[position], MAX*0.01))))

    return mutated
```

However, modifying tests is only one of the things we can do when mutating our actual individuals, which consist of multiple tests. Besides modifying existing tests, we could also delete or add tests, for example like this.

[90]:
```
def mutate_set(solution):
    P_mutate = 1/len(solution)
    mutated = []
    for position in range(len(solution)):
        if random.random() >= P_mutate:
            mutated.append(solution[position][:])

    if not mutated:
        mutated = solution[:]
    for position in range(len(mutated)):
        if random.random() < P_mutate:
            mutated[position] = mutate_test(mutated[position])

    ALPHA = 1/3
    count = 1
    while random.random() < ALPHA ** count:
        count += 1
        mutated.append([random.randint(MIN, MAX) for _ in␣
 ↪range(num_parameters)])

    return mutated
```

With a certain probability, each of the tests can be removed from a test suite; similarly, each remaining test may be mutated like we mutated tests previously. Finally, with a probability `ALPHA` we insert a new test; if we do so, we insert another one with probability `ALPHA`^{2}, and so on.

When crossing over two individuals, they might have different length, which makes choosing a crossover point difficult. For example, we might pick a crossover point that is longer than one of the parent chromosomes, and then what do we do? A simple solution would be to pick two different crossover points.

[91]:
```
def variable_crossover(parent1, parent2):
    pos1 = random.randint(1, len(parent1))
    pos2 = random.randint(1, len(parent2))
    offspring1 = parent1[:pos1] + parent2[pos2:]
    offspring2 = parent2[:pos2] + parent1[pos1:]
    return (offspring1, offspring2)
```

To see this works, we need to define the fitness function. Since we want to cover *everything* we simply need to make sure that every single branch is covered at least once in a test suite. A branch is covered if its minimum branch distance is 0; thus, if everything is covered, then the sum of minimal branch distances should be 0.

There is one special case: If an if-statement is executed only once, then optimising the true/false distance may lead to a suboptimal, oscillising evolution. We therefore also count how often each if-condition was executed. If it was only executed once, then the fitness value for that branch needs to be higher than if it was executed twice. For this, we extend our `update_maps` function to also keep track of the execution count:

[92]:
```
condition_count = {}
def update_maps(condition_num, d_true, d_false):
    global distances_true, distances_false, condition_count

    if condition_num in condition_count.keys():
        condition_count[condition_num] = condition_count[condition_num] + 1
    else:
        condition_count[condition_num] = 1

    if condition_num in distances_true.keys():
        distances_true[condition_num] = min(
            distances_true[condition_num], d_true)
    else:
        distances_true[condition_num] = d_true

    if condition_num in distances_false.keys():
        distances_false[condition_num] = min(
            distances_false[condition_num], d_false)
    else:
        distances_false[condition_num] = d_false
```

The actual fitness function now is the sum of minimal distances after all tests have been executed. If an if-condition was not executed at all, then the true distance and the false distance will be 1, resulting in a sum of 2 for the if-condition. If the condition was covered only once, we set the fitness to exactly 1. If the condition was executed more than once, then at least either the true or false distance has to be 0, such that in sum, true and false distances will be less than 0.

[93]:
```
def get_fitness(x):
    # Reset any distance values from previous executions
    global distances_true, distances_false, condition_count
    distances_true =  {x: 1.0 for x in range(num_branches)}
    distances_false = {x: 1.0 for x in range(num_branches)}
    condition_count = {x:   0 for x in range(num_branches)}

    # Run the function under test
    for test in x:
        triangle_instrumented(*test)

    # Sum up branch distances
    fitness = 0.0
    for branch in range(num_branches):
        if condition_count[branch] == 1:
            fitness += 1
        else:
            fitness += distances_true[branch]
            fitness += distances_false[branch]

    return fitness
```

Before we run some experiments on this, let's make a small addition to our genetic algorithm: Since the size of individuals is variable it will be interesting to observe how this evolves. We'll capture the average population size in a separate list. Since the costs of evaluating fitness are no longer constant per individual but depend on the number of tests executed, we will also change our stopping criterion to the number of executed tests.

[94]:
```
from statistics import mean

length_values = []
max_executions = 10000

def ga():
    population = [get_random_individual() for _ in range(population_size)]
    best_fitness = sys.maxsize
    tests_executed = 0
    for p in population:
        fitness = get_fitness(p)
        tests_executed += len(p)
        if fitness < best_fitness:
            best_fitness = fitness
            best_solution = p
    while tests_executed < max_executions:
        fitness_values.append(best_fitness)
        length_values.append(mean([len(x) for x in population]))
        new_population = elitism(population)
        while len(new_population) < len(population):
            parent1 = selection(population)
            parent2 = selection(population)

            if random.random() < P_xover:
                offspring1, offspring2 = crossover(parent1, parent2)
            else:
                offspring1, offspring2 = parent1, parent2

            offspring1 = mutate(offspring1)
            offspring2 = mutate(offspring2)

            new_population.append(offspring1)
            new_population.append(offspring2)
            tests_executed += len(offspring1) + len(offspring2)

        population = new_population
        for p in population:
            fitness = get_fitness(p)
            if fitness < best_fitness:
                best_fitness = fitness
                best_solution = p
```

```
    print(f"Best fitness: {best_fitness}, size {len(best_solution)}")

    return best_solution
```

Since we now have all the operators we need in place, let's run a first experiment aiming to achieve 100% coverage on the triangle example.

[95]:
```
max_executions = 1000000
MAX = 1000
MIN = -MAX
crossover = variable_crossover
selection = tournament_selection
elitism   = elitism_standard
mutate    = mutate_set
tournament_size = 4
population_size = 50
fitness_values = []
length_values = []
ga()
```

```
Best fitness: 3.6666666666666665, size 808
```

[95]:
```
[[450, 780, 984],
 [-136, 14, 587],
 [587, -406, 396],
 [107, 659, -948],
 [132, 164, -914],
 [-249, 735, 943],
 [358, -826, -325],
 [-376, 119, 416],
 [946, 474, 298],
 [824, 647, 966],
 [-966, 353, 858],
 [151, -67, -648],
 [821, -989, -875],
 [-932, 347, -532],
 [-210, 452, -722],
 [708, 855, 738],
 [-261, -494, 903],
 [556, 144, 795],
 [-319, -489, -161],
 [966, 806, -947],
 [-989, 659, 570],
 [378, 331, -169],
 [-932, 347, -532],
 [-210, 452, -722],
 [708, 855, 738],
 [436, -114, 340],
```

```
[-242, -489, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[744, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[-553, 59, 130],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[805, -949, -73],
```

```
[-249, 735, 943],
[358, -826, -325],
[-376, 119, 416],
[946, 474, 298],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[378, 331, -169],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-242, -489, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[752, 750, 698],
```

```
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[-553, 59, 130],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 750, 699],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[749, 748, 693],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
```

```
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[744, -868, 550],
[-925, 37, -193],
[821, -989, -875],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -495, 904],
[556, 144, 795],
[-319, -488, -161],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-37, -624, -941],
[616, -938, 86],
[-513, 66, -765],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[832, 865, 832],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[154, 286, 197],
[518, -823, 869],
[909, 178, 457],
[886, 239, -269],
[-287, 470, 25],
[735, 949, 106],
[452, -785, -124],
[236, 372, 787],
[271, 484, 350],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
```

```
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-37, -624, -941],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[518, -823, 869],
[914, 178, 457],
[-550, -685, -43],
[886, 239, -269],
[-287, 470, 25],
[-997, -587, 191],
[814, -648, 223],
[320, 392, 349],
[801, -71, 967],
[820, -264, -523],
[834, 865, 827],
[13, 870, 207],
[729, -499, -393],
[-278, -701, 775],
[-744, 368, -593],
[-158, -382, 590],
[-829, 198, -994],
[161, 376, 455],
[946, 474, 298],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[-932, 347, -532],
```

```
[-210, 452, -722],
[708, 855, 738],
[436, -114, 322],
[-261, -494, 903],
[-319, -489, -161],
[-989, 659, 570],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[-966, 353, 858],
[821, -989, -875],
[-932, 347, -532],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 732, 680],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[744, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[805, -949, -73],
[-454, 834, 831],
[267, 329, 734],
[922, -409, 503],
```

```
[814, -648, 223],
[320, 392, 349],
[801, -71, 967],
[820, -264, -523],
[834, 865, 827],
[-744, 368, -593],
[-158, -382, 590],
[-829, 198, -994],
[161, 376, 455],
[-650, -527, 297],
[169, -917, 223],
[-661, -810, -171],
[-287, 470, 25],
[-997, -587, 191],
[814, -648, 223],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[-966, 353, 858],
[821, -989, -875],
[-932, 347, -532],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 750, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[744, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[142, -746, 162],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[432, -156, 224],
[-892, -98, 62],
```

```
[834, 865, 827],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[518, -823, 869],
[914, 178, 457],
[-550, -685, -43],
[886, 239, -269],
[-287, 470, 25],
[-997, -587, 191],
[814, -648, 223],
[320, 392, 349],
[801, -71, 967],
[820, -264, -523],
[834, 865, 827],
[13, 870, 207],
[729, -499, -393],
[-278, -701, 775],
[-744, 368, -593],
[-158, -382, 590],
[-829, 198, -994],
[946, 474, 298],
[824, 647, 966],
[-989, 659, 570],
[378, 331, -169],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[969, 816, -951],
[-989, 659, 570],
[378, 331, -169],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-242, -489, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[394, -81, 908],
[717, 78, 876],
```

```
[543, 980, 190],
[81, -339, -236],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[744, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-379, 324, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[-553, 59, 130],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[805, -949, -73],
[357, 934, 374],
[-674, 22, -516],
[-210, 109, -976],
[675, -549, 829],
[-273, -250, -928],
[-585, -751, -103],
[138, -746, 159],
```

```
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[426, 624, -580],
[824, 647, 966],
[-966, 353, 858],
[821, -989, -875],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 750, 699],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[744, 748, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[-385, 326, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[744, -868, 550],
[-925, 37, -193],
[821, -989, -875],
[-932, 347, -532],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
```

```
[556, 144, 795],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-37, -624, -941],
[-513, 66, -765],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[832, 865, 832],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[154, 286, 197],
[518, -823, 869],
[909, 178, 457],
[891, 239, -277],
[-287, 470, 25],
[735, 949, 106],
[452, -785, -124],
[236, 372, 787],
[271, 484, 350],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-438, -851, 514],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
```

```
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[-553, 59, 130],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[805, -949, -73],
[-249, 735, 943],
[358, -826, -325],
[-376, 119, 416],
[946, 474, 298],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[378, 331, -169],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-242, -489, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
```

[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[752, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[-553, 59, 130],
[744, -868, 550],
[-925, 37, -193],
[-95, -824, -534],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 750, 699],
[820, -264, -523],
[834, 865, 827],
[-932, 347, -532],
[708, 855, 738],
[436, -109, 330],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[749, 748, 693],
[113, 830, 78],
[148, 125, -201],

```
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-438, -851, 514],
[432, -156, 224],
[-892, -98, 62],
[-761, 182, -137],
[744, -868, 550],
[-925, 37, -193],
[821, -989, -875],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -495, 904],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-37, -624, -941],
[616, -938, 86],
[-513, 66, -765],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[832, 865, 832],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[154, 286, 197],
[518, -823, 869],
[909, 178, 457],
[886, 239, -269],
```

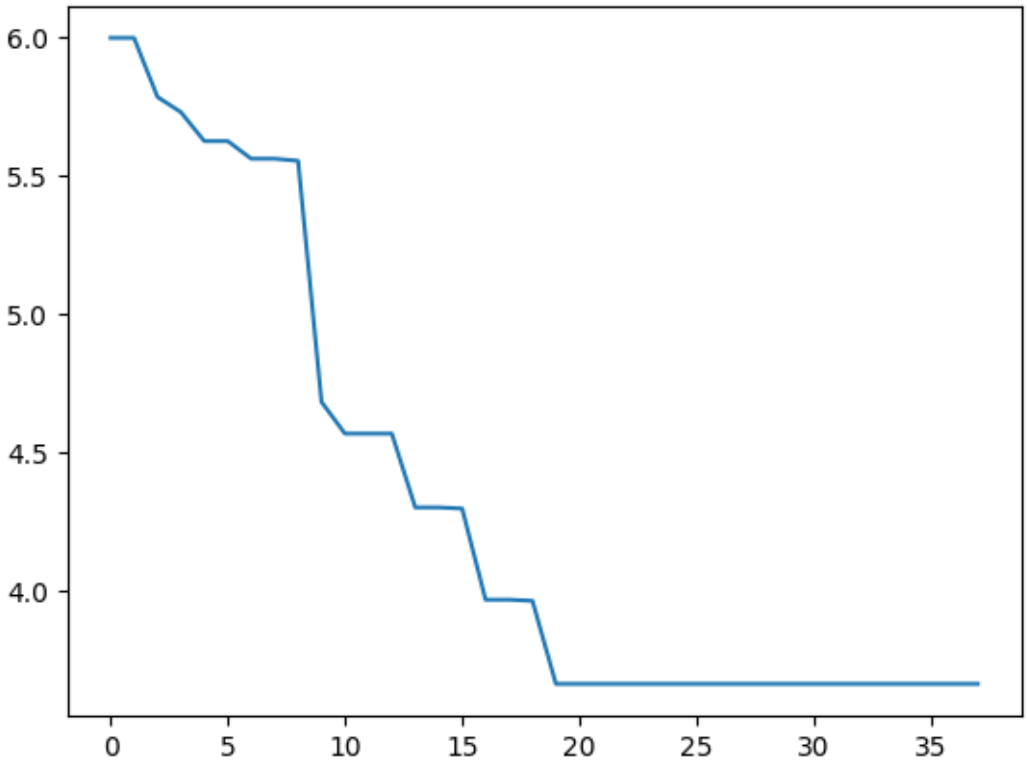
```
[-287, 470, 25],
[735, 949, 106],
[452, -785, -124],
[236, 372, 787],
[271, 484, 350],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[378, 331, -169],
[655, 725, 509],
[394, -81, 908],
[717, 78, 876],
[543, 980, 190],
[81, -339, -236],
[-37, -624, -941],
[777, -670, 729],
[616, -938, 86],
[-513, 66, -765],
[-336, -691, -918],
[485, 708, -481],
[-934, 9, -447],
[820, -264, -523],
[834, 865, 827],
[599, -788, -449],
[821, -989, -875],
[128, -589, -893],
[-860, 827, -379],
[518, -823, 869],
[914, 178, 457],
[-550, -685, -43],
[886, 239, -269],
[-287, 470, 25],
[-997, -587, 191],
[814, -648, 223],
[320, 392, 349],
[801, -71, 967],
[820, -264, -523],
[834, 865, 827],
[13, 870, 207],
[729, -499, -393],
[-278, -701, 775],
[-744, 368, -593],
```

```
[-158, -382, 590],
[-829, 198, -994],
[161, 376, 455],
[946, 474, 298],
[824, 647, 966],
[-966, 353, 858],
[151, -67, -648],
[821, -989, -875],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-319, -489, -161],
[-989, 659, 570],
[-932, 347, -532],
[-210, 452, -722],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[556, 144, 795],
[-319, -489, -161],
[966, 806, -947],
[-989, 659, 570],
[-966, 353, 858],
[821, -989, -875],
[708, 855, 738],
[436, -114, 340],
[-261, -494, 903],
[-264, -169, 110],
[-585, -751, -103],
[138, -746, 159],
[758, 732, 680],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[744, 750, 698],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[-585, -751, -103],
[138, -746, 159],
[758, 742, 699],
[113, 830, 78],
[148, 125, -201],
[-374, 329, 107],
[432, -156, 224],
```

```
 [-892, -98, 62],
 [-761, 182, -137],
 [744, -868, 550],
 [-925, 37, -193],
 [-95, -824, -534],
 [805, -949, -73],
 [-454, 834, 831],
 [267, 329, 734],
 [922, -409, 503],
 [814, -648, 223],
 [320, 392, 349],
 [801, -71, 967],
 [820, -264, -523],
 [834, 865, 827],
 [-744, 368, -593],
 [-158, -382, 590],
 [-829, 198, -994],
 [161, 376, 455],
 [-650, -527, 297],
 [169, -917, 223],
 [-661, -810, -171],
 [-381, 139, 568],
 [-851, 42, -697],
 [236, -136, 249],
 [-564, -299, 870],
 [-494, -533, -565],
 [938, -704, -401],
 [-358, 578, 903],
 [441, -603, 567],
 [858, 461, 463]]
```

[96]:
```
plt.plot(fitness_values)
```

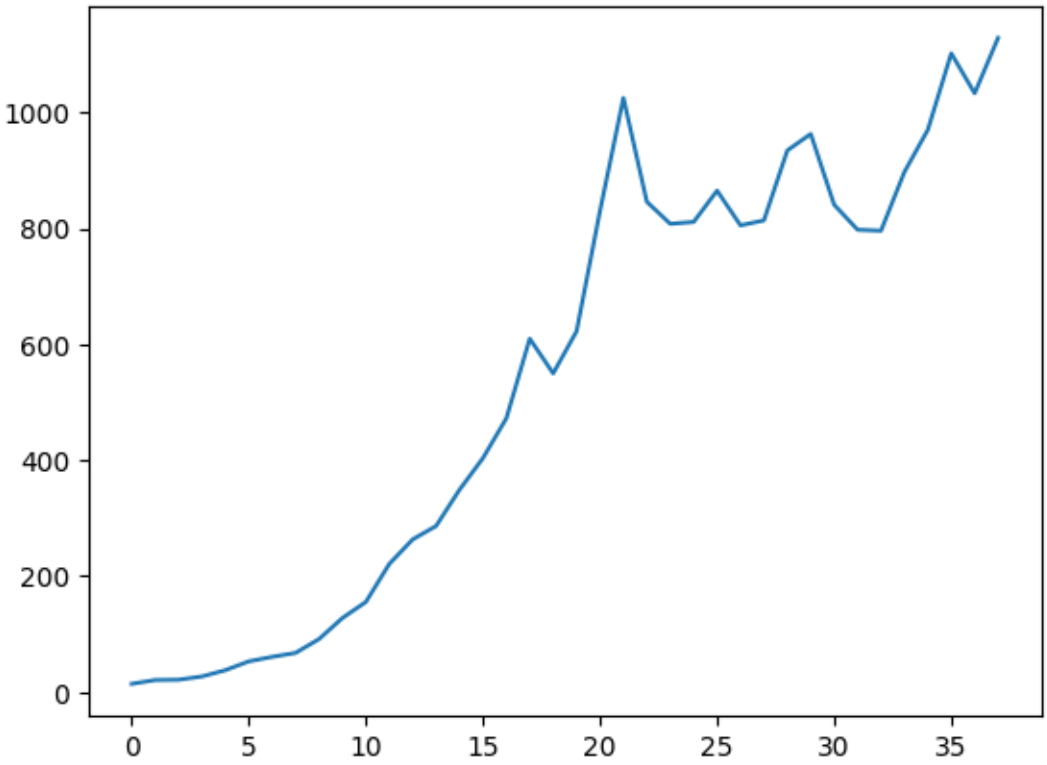
[96]:
```
[<matplotlib.lines.Line2D at 0x111bd8a90>]
```



The plot shows iterations of the genetic algorithm on the x-axis. Very likely, the result likely isn't great. But why? Let's look at the average population length.

[97]:
```
plt.plot(length_values)
```

[97]:
```
[<matplotlib.lines.Line2D at 0x111bcc400>]
```



What you can see here is a phenomenon called *bloat*: Individuals grow in size through mutation and crossover, and the search has no incentive to reduce their size (adding a test can never decrease coverage; removing a test can). As a result the individuals just keep growing, and quickly eat up all the available resources for the search. How to deal with this problem will be covered in the next chapter.