```python
import random

TARGET = "ArtificialIntelligenceLab"
POPULATION_SIZE = 70

def generate_random_string(length):
    return ''.join(random.choice("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()_+-=[]{}|;:,.<>?") for _ in range(le

def calculate_fitness(individual):
    return sum(1 for a, b in zip(individual, TARGET) if a != b)

def evolve_population(population):
    # Selection
    selected = sorted(population, key=calculate_fitness)[:int(POPULATION_SIZE * 0.3)]

    # Crossover
    offspring = []
    for _ in range(POPULATION_SIZE - len(selected)):
        parent1, parent2 = random.sample(selected, 2)
        crossover_point = random.randint(1, len(TARGET) - 1)
        child = parent1[:crossover_point] + parent2[crossover_point:]
        offspring.append(child)

    # Mutation
    for i in range(int(POPULATION_SIZE * 0.1)):
        mutant = random.choice(offspring)
        mutation_point = random.randint(0, len(TARGET) - 1)
        mutated_gene = random.choice("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789!@#$%^&*()_+-=[]{}|;:,.<>?")
        mutant = mutant[:mutation_point] + mutated_gene + mutant[mutation_point + 1:]
        offspring.append(mutant)

    # Replacement
    new_population = selected + offspring

    return new_population

# Main loop
population = [generate_random_string(len(TARGET)) for _ in range(POPULATION_SIZE)]
generation = 1

while True:
    population = evolve_population(population)
    best_individual = min(population, key=calculate_fitness)

    print(f"Generation {generation}, Best: {best_individual}, Fitness: {calculate_fitness(best_individual)}")

    if calculate_fitness(best_individual) == 0:
        print("Target reached!")
        break

    generation += 1
```

```
['p}wJE4z_K[HI8CnntgQrD@ZTP', 'ge:g-h+M]JZ#II,.jH3s5y6Nr', '^ox1qJ+vake%6+rZP|gk3(zNm', '3n<Xcdz%py_xl},cf6AQf;}UQ', 'E*Nq:hzS,NQtL&[
Generation 1, Best: T2FbA+3@S@&ht!9w[DyhCeG#1, Fitness: 23
Generation 2, Best: ^ox1qJ+vake%6+rZtgQrD@-Hb, Fitness: 22
Generation 3, Best: orv2-%6=7P_Yssl2tgQrD@-Hb, Fitness: 21
Generation 4, Best: orv2-%+vake%6+l2tgQrD@-Hb, Fitness: 20
Generation 5, Best: orv2-%+vake%6+l2tgQrD@-Hb, Fitness: 20
Generation 6, Best: orv2-%+vake%6+l2tgQrD@-Hb, Fitness: 20
Generation 7, Best: orv2-%+vake%6+l2tgQhCe-Hb, Fitness: 19
Generation 8, Best: orv2-%+vake%6+l2tgQhCe-Hb, Fitness: 19
Generation 9, Best: orv2-%+vake%6+l2tgQhCe-Hb, Fitness: 19
Generation 10, Best: orv2-S+va?&htsl2tgQrDeGHb, Fitness: 18
Generation 11, Best: orv2-S+va?&htsl2tgQrDeGHb, Fitness: 18
Generation 12, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 13, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 14, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 15, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 16, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 17, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 18, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 19, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 20, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 21, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 22, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 23, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 24, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
```

```
Generation 25, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 26, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 27, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 28, Best: orv2-S+va?&htslwtgQnDeG#b, Fitness: 17
Generation 29, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 30, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 31, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 32, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 33, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 34, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 35, Best: Arv2-S+va?&htslwtgQnDeG#b, Fitness: 16
Generation 36, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 37, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 38, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 39, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 40, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 41, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 42, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 43, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 44, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 45, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 46, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 47, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 48, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 49, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 50, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 51, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 52, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 53, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 54, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 55, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
Generation 56, Best: Arv2-S+va?&ntsl2tgQnDeG#b, Fitness: 15
```

```
from collections import deque

class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'S': 5,
            'A': 3,
            'B': 4,
            'C': 2,
            'D': 6,
            'G': 0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            finalcost = 0
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:

                    reconst_path.append(n)

                    n = parents[n]

                reconst_path.append(start_node)
                finalcost = g[reconst_path[0]]
                reconst_path.reverse()


                print('Path found: {}'.format(reconst_path), "with cost:")
                print(finalcost)
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
```

```
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None


adjacency_list = {
    'A': [('B', 2), ('C', 1),('S',1)],
    'B': [('A',2),('D', 5)],
    'C': [('A',1),('D', 3),('G',4)],
    'D': [('G',7),('B',3),('C',3)],
    'S': [('G',10),('A',1)],
    'G': [('S',10),('D',7),('C',4)]
}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('S', 'D')
```

```
Path found: ['S', 'A', 'C', 'D'] with cost:
5
['S', 'A', 'C', 'D']
```

```
import random

TARGET = "ABCDA"
POPULATION_SIZE = 10

def generate_random_string(length):
    return ''.join(random.choice("ABCD") for _ in range(length))

def calculate_fitness(individual):
    return sum(1 for a, b in zip(individual, TARGET) if a != b)

def evolve_population(population):
    # Selection
    selected = sorted(population, key=calculate_fitness)[:int(POPULATION_SIZE * 0.3)]

    # Crossover
    offspring = []
    for _ in range(POPULATION_SIZE - len(selected)):
        parent1, parent2 = random.sample(selected, 2)
        crossover_point = random.randint(1, len(TARGET) - 1)
        child = parent1[:crossover_point] + parent2[crossover_point:]
        offspring.append(child)

    # Mutation
    for i in range(int(POPULATION_SIZE * 0.1)):
        mutant = random.choice(offspring)
        mutation_point = random.randint(0, len(TARGET) - 1)
        mutated_gene = random.choice("ABCD")
        mutant = mutant[:mutation_point] + mutated_gene + mutant[mutation_point + 1:]
        offspring.append(mutant)

    # Replacement
    new_population = selected + offspring

    return new_population

# Main loop
population = [generate_random_string(len(TARGET)) for _ in range(POPULATION_SIZE)]
generation = 1

while True:
    population = evolve_population(population)
    best_individual = min(population, key=calculate_fitness)

    print(f"Generation {generation}, Best: {best_individual}, Fitness: {calculate_fitness(best_individual)}")

    if calculate_fitness(best_individual) == 0:
```