Seminar Assignment 1

Intelligent Systems - FRI

Gasper Spagnolo

November 13, 2022

1 Introduction

In the first seminar assignment, your goal is to use genetic algorithms to find a path out of a maze, represented as a vector of strings, where # characters represent walls, . represent empty spaces, and S and E represent the starting and ending points, as in a given example below:

```
maze = c("####E######",
         "##...#.####",
         "#..#.#.####",
         "#.##...####",
         "#.##.#..S##",
         "###########")
```

You can move through the maze in four directions, left, right, up, and down. In the example above, the shortest path from the starting position S to the exit E is composed of the following moves: left, left, up, left, left, up, up, up. In your solution, this should be represented as a string "LLULLUUU". Your task is to create a function that will be able to find path as short as possible out of any maze represented in such a way

2 Solution

2.1 Task 1

I decided to write this assignment in python using the pygad library becouse I am more familiar with this programming language.

2.1.1 Task description

Create a function that reads the 2D representation of a maze and returns the shortest path found by a genetic algorithm. To do this, you will need to:

- Read the map into a suitable format (for example, a matrix).
- choose a suitable representation of your solutions (the path). Hint: you don't need to use strings when working with the genetic algorithm. You can use numeric or binary representations for the GA function and then convert the result to a string as the final result.
- Define the fitness function. Make sure to penalise paths through walls - those are invalid solutions
- Run the genetic algorithm with suitable settings.

2.1.2 Read the map into a suitable format

I decided to read all the maps provided in the assignment into a list of lists. Each list represents a maze and each element of the list represents a row of the maze.

2.1.3 Choose a suitable representation of your solutions

I decided to use a binary representation of the solution same size as an original maze. Each bit represents a move. 0 means that the agent did not visit the cell and 1 means that the agent visited the cell. So if maze is of size N x M, the solution will be of size N x M. But there is no such thing as N-dimensional array that GA accepts. So I reshaped the matrix into a vector of size N * M and worked with that kind of solution.

2.1.4 Define the fitness function

This part was the most difficult for me. I maybe overcomplicated that part but at least it yields good results. Before runing the algorithm I have decided to construct a punish matrix, which is a matrix of the same size as the maze. Each cell in the punish matrix is evaluated before the algorithm starts. The evaluation is based on the position of walls and valid paths. So if there is a wall in the cell, the fitness value in that cell is set to some low scalar. If there is a valid move then the fitness value in that cell is high. So everytime the fitness function is called, the matrix product will be executed and some initiall fitness value will be computed asfollows:

```
fitness = np.sum(path * maze.punish_matrix.reshape(-1))
```

But though experimentation I found that this approach was not good enough so I modified the function by adding punsihment if the agent did not start at the starting position and if the agent did not end at the ending position. Still the results were not good enough so I decided to check if there is a valid path from the starting position to the ending position. If there is no valid path then I would punish the agent otherwise I would give him some reward. This approach yielded better results. But still I was not satisfied with the results so I decided to add some more punishes and rewards:

- Add a reward if agent finds a shorter path than the best path found so far.
- Update weights in punish matrix so that the agent will prefer to move on best path found so far.
- If the agent does not find any valid path until 80% of the GA iterations then activate critical search phase. That means that the agent will be rewarded if he finds **any** path from start to end, even if it maybe isn't the correct one. This way the weights are updated so that it converges to the correct path.

The critical section evaluation in code is done as follows:

```
def walk_through_maze(self, solution_matrix, critical_situation):
    queue = [[self.start_pos]]
```

```
        def add_to_queue(full_path, x, y):
            if (x,y) not in full_path:
full_path = full_path.copy()
                full_path.append((x, y))
                queue.append(full_path)

        while queue != []:
            full_path = queue.pop()
            x, y = full_path[-1]
            if(self.maze[x][y] == 'E'):
                return full_path
            if x + 1 < len(self.maze) :
                if solution_matrix[x+1, y] == 1 and
(critical_situation or (self.maze[x+1][y] == "." or self.maze[x+1][y] == "E")):
                    add_to_queue(full_path, x+1, y)
            if x - 1 >= 0:
                if solution_matrix[x-1, y] == 1 and
(critical_situation or (self.maze[x-1][y] == "." or self.maze[x-1][y] == "E")):
                    add_to_queue(full_path, x-1, y)
            if y + 1 < len(self.maze) :
                if solution_matrix[x, y+1] == 1 and
(critical_situation or(self.maze[x][y+1] == "." or self.maze[x][y+1] == "E")):
                    add_to_queue(full_path, x, y+1)
            if y - 1 >= 0:
                if solution_matrix[x, y-1] == 1 and
(critical_situation or (self.maze[x][y-1] == "." or self.maze[x][y-1] == "E")):
                    add_to_queue(full_path, x, y-1)
        return []
```

2.1.5 Run the genetic algorithm with suitable settings

I used the following settings wen running the algorithm:

- `number_of_genes = N * M`
  (if the maze is of size N x M) So the solution is a vector of size N * M.
- `num_of_generations = 1000`
  How many generations will the algorithm run.
- `sol_per_pop = 20`
  Number of possible solutions in the population.
- `num_parents_mating = 15`
  Number of solutions to be selected as parents in the mating pool.
- `keep_parents = -1`
  If -1, this means all parents in the current population will be used in the next population
- `allow_duplicate_genes = True`
  If True, then a solution/chromosome may have duplicate gene values.
- `mutation_type = "random"`
  Mutation type is random.

- `crossover_type = "two_point"`
  Applies the 2 points crossover. It selects the 2 points randomly at which crossover takes place between the pairs of parents
- `parent_selection = "tournament"`
  Selects the parents using the tournament selection technique. Later, these parents will mate to produce the offspring.
- `gene_type = int`
  We will be predicting integer values.
- `gene_space = [0,1]`
  Define binary subset to be gene space.
- `fitness_func = fitness_func`
  Specify fitness function.
- `parallel_processing = 4`
  Spawn 4 additional threads to speed up computing.

2.1.6 Results

1. On first maze I got a perfect score: *The shortest path is [(3, 1), (2, 1), (2, 2), (1, 2), (0, 2)]*

Figure 1: Solution to the first maze

2. Same for the second one: *The shortest path is [(4, 5), (4, 4), (4, 3), (4, 2), (3, 2), (2, 2), (2, 3), (2, 4), (2, 5), (1, 5), (0, 5)]*

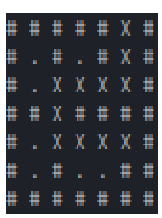

Figure 2: Solution to the second maze

3. The third one had many problems and it did not want to converge to propper soluition.

4. The fourth one also found the solution pretty quickly. *The shortest path is [(5, 5), (4, 5), (3, 5), (3, 6), (3, 7), (3, 8), (2, 8), (1, 8), (1, 7), (1, 6), (1, 5), (0, 5)]*

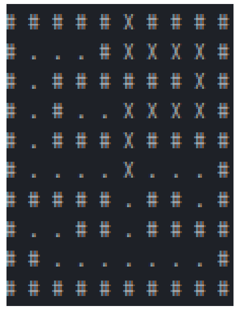

Figure 3: Solution to the fourth maze

Other mazes found also found some solutions, but they were not optimal. Or they were trying to go through a wall becouse the critical section was activated. I think that the problem is that the mutation and crossover operators are not good enough.

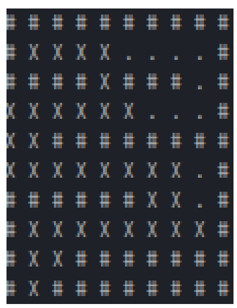

Figure 4: Example of solution using the critical section

So I will try to improve them in the following sections.

2.2 Task 2

2.2.1 Task description

The default mutation and crossover functions in R are not well-suited for this task because they do not necessarily return valid paths (for example, the mutation might introduce a move that goes through a wall). To fix this, modify the mutation and selection functions so that they take the walls into account. Additionally, try to create a starting population in a way that takes walls into account. You can base your crossover and mutation functions on existing GA library functions. Modify at least one crossover or mutation function in a way that makes them more suitable for this task.

2.2.2 Mutation function

I initially used the random mutation type provided by library pygad. It was not good enough for this task, because it was not taking into account the walls. So I redefined mutation function in such way, that we add random bits where there is no wall. If there is a wall, we set random number of bits to 0.

The function is defined as follows:

```
def on_mutation(generations, ga_instance):
    maze = mazes[maze_ix]

    # Firtly find the instances where there are no walls
    no_wall_instances = np.where(maze.mutation_matrix.reshape(-1) == 1)[0]
    wall_instances = np.where(maze.mutation_matrix.reshape(-1) == 0)[0]

    # Loop through the population
    for i in range(len(generations)):
        #  select random number of the instances where there are walls
        random_false_instances = np.random.choice(wall_instances,
size=int(len(wall_instances)* random.uniform(0.01, 1.0)), replace=False)
        # Then randomly select random number of the instances where there are no walls
        random_true_instances = np.random.choice(no_wall_instances,
size=int(len(no_wall_instances)* random.uniform(0.01, 1.0)), replace=False)
        # Then apply those values to generation
        generations[i][random_true_instances] = 1
        generations[i][random_false_instances] = 0

    return generations
```

I also generated the initial population using the same function, but firstly I generated some random bitarrays and then applied the same function to them.

```
        initial_population = np.random.choice([0, 1],
size=(self.punish_matrix.size, self.initial_population_size))
```

The results I got using this approach were suprising! I got a perfect score on all mazes. I think that the reason for this is that the mutation function is not only taking into account the walls, but also the previous solution. The algorithm converges really fast now. In 5 generations of 400 specimens we get a shortest path! I even generated a 1000 x 1000 maze and it solved it in around 1000 secnds!

2.3 Task 3

2.3.1 Task description

In Task 3, mazes also contain treasure (marked with T). For example:

```
maze2 = c("####E######",
"##...#.####",
"#..#.#.####",
"#.##...####",
"#T##T#..S##",
"###########")
```

Your task is to modify your approach so that the solution returns as short a path as possible that also collects all the treasure.

2.3.2 Approach

For treasures to be found I had to modify my fitness function, initial population generation and mutation function. I introduced clustering of cells which are close to the treasures. That way the genetic algoritm will also mutate in that way. Clusterization is done using simple alogirthm, which just checks for K valid cells arround the treasure and adds them to the cluster. The cluster is then used to influence generation of the initial population and mutation of the population.

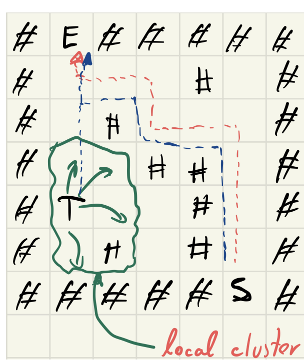

Figure 5: Clustering example

The fitness function had also be redefined to also take into account the treasures. The fitness function is defined as follows:

```
def fitness_func(path, solution_idx):
.........................
    if path[maze.start_pos] == 1 and path[maze.end_pos] == 1:
        fitness += 300
        # Check if there is a valid path
        # First check if there is a path from start to end
        paths = []
        complete_path = maze.walk_through_maze(path, maze.end_pos)
        paths.extend(complete_path)

        # Then for each treasure find a path from start to treasure
        treasures_found = 0
        if complete_path != []:
            for treasure in maze.treasures:
                treasure_path = maze.walk_through_maze(path, treasure)
                if treasure_path != []:
                    treasures_found += 1
                    paths.extend(treasure_path)
            # Remove duplicates
            path = list(set(paths))

        path_len = len(path)
        # Set the first path found as the shotest one
        if maze.shortest_path == [] and path_len > 0 and treasures_found >= len(maze.treasures) // 2:
            fitness += treasures_found * 1000
            print('First path found')
            maze.shortest_path = path
            maze.treasures_found = treasures_found
            maze.adjust_weights(complete_path)

        #Check if the current path is shorter than the shortest one
        elif treasures_found > maze.treasures_found and path_len > 0:
            fitness += 1000 * treasures_found
            print('Path with more treasures found!')
            maze.shortest_path = path
    maze.treasures_found = treasures_found
            maze.adjust_weights(complete_path)

        elif path_len < len(maze.shortest_path) and treasures_found > maze.treasures_found and path_le
            fitness += 1000 * treasures_found
            print('Path with less steps found!')
            maze.shortest_path = path
            maze.treasures_found = treasures_found
            maze.adjust_weights(complete_path)
    ..........................
```

So now each time there is a treasue in the path or if there is a shorter path, the fitness function will increase the fitness of the specimen. The fitness function also takes into account the number of treasures found. If there is a path with more treasures found, the fitness function will increase the fitness of the specimen.

And the mutation function is same as in task 2, but now it also takes into account the treasures. The mutation function is defined as follows:

```
def on_mutation(generations, ga_instance):
.........................
    cluster_instances = np.reshape(np.array(maze.clusters), -1)

    # Loop through the population
    for i in range(len(generations)):
        #  randomly select random number of the instances where there are clusters
        random_cluster_instances = np.random.choice(cluster_instances, size=int(len(cluster_instances)
.........................
        generations[i][random_cluster_instances] = 1
.........................
    return generations
```

2.3.3 Results

Results are very good! The algorithm finds the shortest path and collects all the treasures on all provided test mazes.

Example of the shortest path found on maze 4: *The shortest path is [(12, 4), (4, 9), (5, 1), (3, 13), (5, 10), (10, 6), (7, 1), (1, 15), (18, 1), (16, 13), (18, 10), (7, 10), (9, 1), (3, 15), (17, 14), (13, 1), (1, 8), (6, 4), (18, 3), (7, 12), (14, 8), (17, 16), (1, 10), (15, 9), (18, 5), (7, 5), (7, 14), (3, 1), (14, 1), (3, 10), (13, 5), (1, 3), (16, 1), (1, 12), (16, 10), (18, 7), (14, 3), (3, 12), (1, 5), (6, 1), (1, 14), (16, 12), (18, 9), (14, 5), (4, 4), (3, 14), (10, 1), (1, 7), (18, 2), (1, 16), (16, 14), (7, 11), (12, 1), (14, 7), (5, 4), (4, 6), (3, 16), (8, 6), (10, 3), (1, 9), (18, 4), (12, 3), (4, 8), (10, 5), (1, 2), (2, 1), (1, 11), (6, 10), (12, 5), (4, 1), (3, 11), (4, 10), (8, 1), (19, 10), (1, 4), (1, 13), (14, 4), (17, 15), (1, 6), (15, 8), (6, 5), (7, 13), (14, 6), (4, 5), (8, 5), (11, 1), (17, 17), (2, 16), (15, 1), (15, 10), (18, 6), (4, 7), (17, 1), (10, 4), (9, 6), (11, 3), (1, 1), (16, 11), (18, 8)]*

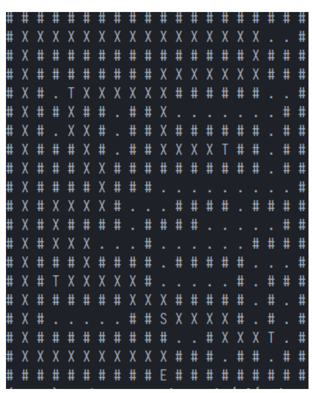

Figure 6: shortest path in maze 4, collecting all treasures

2.4 Task 4

2.4.1 Task description

Present a report that describes your approach, shows highlights of your code, and presents the results. The results have to include performance comparisons between different settings of the genetic algorithm (different mutation, crossover and selection functions, different starting populations and so on). Make sure to evaluate your approach on different mazes, the one in the instructions is just an example. The mazes.r file on ucilnica contains several additional examples of various sizes and complexities. Find the largest size of a maze that can still be solved with your approach - feel free to create your own mazes if the example mazes are too small. Produce a graph to show how the maze size affects the running time of the genetic algorithm.

2.4.2 Speed comparison between algorithm used in fist task and algorithm used in second task

This graph shows time comparrison between algorithm used in first task where there are random mutations and random initial population versus algorithm used in second task where custom mutations are applied and initial population is created based on maze structure data. Mazes used in the test are the ones which were provided with the task description.

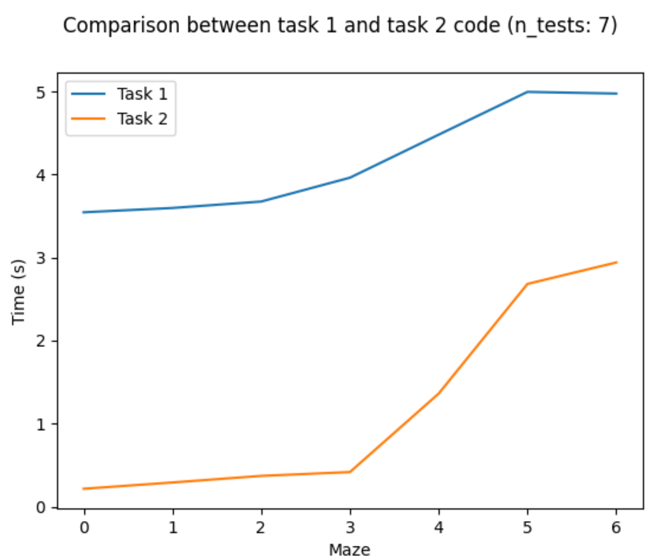

Figure 7: Default mutations and random population vs custom mutation and custom population

2.4.3 How algorithm efficency decreases with maze size

In this test, I firstly randomly generated mazes of size 50 - 950 with step size 50. So I generated 18 mazes. Then I ran the algorithm from task 2 on each of the mazes and measured the time it took to find the shortest path. The results are shown in the graph below.

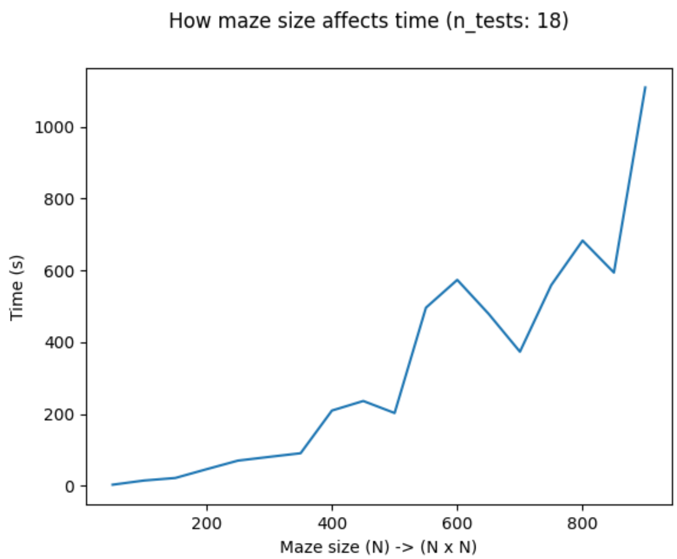

Figure 8: Computation time for different maze sizes

2.4.4 How algorithm performs when there are treasures included

In this test, I was interested with mazes with treasures. I used provided test mazes timed computation time using the algorithm from task 3.

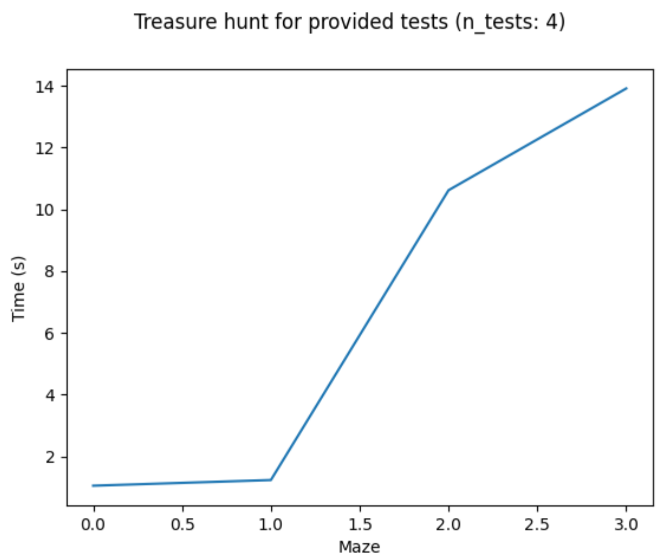

Figure 9: Computation time for different maze sizes with treasures