Walking on Thin Ice Evolutionary Computation

André F. Moreira, João P. Pinto

FCTUC-DEI, Coimbra, Portugal
andremoreira@student.dei.uc.pt
jopepinto@student.dei.uc.pt

Abstract. The Frozen Lake problem, available in the Gymnasium Framework, involves finding a policy that allows an agent to move from the starting position to a destination across a frozen lake, avoiding any openings on the icy surface. In this work, we developed an Evolutionary based approach to solve the problem. Our findings indicate that specific combinations of crossover and mutation operators significantly influence the algorithm's success rate. Additionally, we investigated the impact of population size, crossover probability, elitism, and others on the EA's performance for a 12x12 map. Statistical analysis revealed optimal parameter settings.

Keywords: Genetic, Algorithm, Evolutionary, Fitness, Genotype, Crossover, Mutation, Configuration, Representation

1 Introduction

The Frozen Lake problem is a popular entry point to reinforcement learning that presents a simple yet challenging scenario: navigate an agent from a starting position to the goal across a frozen lake, while avoiding any openings on the surface. While it is commonly tackled using reinforcement learning algorithms like Q-learning [1] [2], and many others [3], this work explores an alternative approach – Evolutionary Algorithm (EA).

This report will delve into the design of the EA, addressing key elements, such as: Representation; Initialization; Variation Operators; Parent Selection; Survivor Selection; Fitness Function. We will also explore the impact of different fitness functions on the EA's performance.

By comparing the performance of various EA configurations, using statistical analysis, we aim to identify the most effective approach and do an informed discussion about the results obtained.

2 Materials

To develop our solution, we used the Frozen Lake environment available from the Gymnasium Framework.

The algorithm was developed using Python programming language with some of the code provided during classes from the professors and adapted to tackle this specific problem.

The code follows a typical and simple EA, where it generates an initial population, and through multiple crossovers and mutations between generations and individuals would eventually reach the goal with a certain fitness value.

To find the best possible configuration for the EA, we used multiple statistical analysis methods, such as: Kolgomorov-Smirnov test; Wilcoxon test; Student's t-test. We also compared some fitness value graphs through generations in different configurations.

3 Methods

In this section, we specify how we approached the problem, and explain all components used along the way.

3.1 Representation

The genotype is defined as a list of integers. Each integer represents an action (0, 1, 2 or 3) the agent will take in a specific step within the Frozen Lake environment. An individual is generated with a random size between the minimum possible number of steps to reach the goal *(2 * map_size – 1)* and the maximum possible number of steps *(map_size^2)*.

In our approach, the genotype acts as the phenotype, as it directly encodes the agent's policy. The mapping function returns the steps taken by the agent, not the actions. It is used to easily calculate the fitness of the individual.

3.2 Initialization Procedure

The algorithm starts by generating an initial population of a determined and customizable size. For each individual, a genotype is generated according to the previous explanation. After this, every individual will be assessed, selected, crossed over and mutated, according to their fitness value and probability.

3.3 Variation Operators

Multiple crossover and mutation functions were tested and evaluated; these are the ones we used the most:

3.3.1 Crossover Functions

- The one-point crossover selects an index in each parent and generates an offspring with the beginning of parent1 until the first index and the end of parent2 after the second index. This was the crossover function that gave us best results (more on that later).

```
def one_point_crossover(parent1, parent2):
    size_1 = len(parent1['genotype'])
    size_2 = len(parent2['genotype'])
    pos_1 = random.randint(1, size_1 - 1)
    pos_2 = random.randint(1, size_2 - 1)
    genotype = parent1['genotype'][:pos_1] +
parent2['genotype'][pos_2:]
    return {'genotype': genotype, 'fitness': None}
```

- The two point crossover selects two indexes, creates an offspring that inherits the beginning of parent1, middle of parent2 and the end of parent1.

```
def two_point_crossover(parent1, parent2):
    cut_points1 =
sorted(random.sample(range(len(parent1['genotype'])), 2))
    cut_points2 =
sorted(random.sample(range(len(parent2['genotype'])), 2))
    genotype = parent1['genotype'][: cut_points1[0] ] +
parent2['genotype'][ cut_points2[0] : cut_points2[1]] +
parent1['genotype'][ cut_points1[1] : ]
    return {'genotype': genotype, 'fitness': None}
```

[Two-point crossover function]

3.3.2 Mutation Functions

We decided to develop three different simple and self-explanatory mutation functions: *delete_mutation; change_value_mutation; insert_mutation:*

```
def delete_mutation(p):
    p['fitness'] = None
    at = random.randint(0, len(p['genotype']) - 1)
    p['genotype'].pop(at)
    return p

def change_value_mutation(p):
    p['fitness'] = None
    at = random.randint(0, len(p['genotype']) - 1)
    p['genotype'][at] = random.randint(0, 3)

    return p

def insert_mutation(p):
    p['fitness'] = None
    action = random.randint(0, 3)
    at = random.randint(0, len(p['genotype']) - 1)
    p['genotype'].insert(at, action)
    return p
```

Also, we had an idea to use all three at once to possibly reach a better solution. In this way, there is a chance to use each one:

```
def main_mutation(p):
    op = random.randint(1, 3)
    if op == 1:
        p = delete_mutation(p)
    elif op == 2:
        p = change_value_mutation(p)
    else:
        p = insert_mutation(p)
    return p
```

3.4 Parent Selection Mechanisms

For parent selection, we used a tournament-based approach, where it selects a population of a pre-determined tournament size and returns the best individual of the pool:

```
def tournament(tournament_size : int, maximization=True)
-> dict:
    def tournament(population):
        pool = random.sample(population, tournament_size)
        pool.sort(key=lambda i: i['fitness'],
reverse=maximization)
        return copy.deepcopy(pool[0])
    return tournament
```

3.5 Survivor Selection Mechanisms

For survivor selection, we used elitism in order to maintain a certain percentage of the best of each generation to next:

```
def survivor_elitism(elite : float, maximization=True):
    def elitism(parents,offspring):
        pop_size = len(parents)
        elite_size = int(pop_size * elite)
        parents.sort(key=lambda x: x['fitness'],
reverse=maximization)
        new_population = parents[ : elite_size] +
offspring[ : pop_size - elite_size]
        return new_population
    return elitism
```

For further tests, we also tried without elitism. Using a survivor generational approach, by returning only the offspring.

3.6 Fitness Function

To make a good fitness function to tackle this problem, there are some key things to have in mind:

- It must complete the scenario.
- Preferably the smallest route to the goal.
- There shouldn't be any repeated positions.
- It must avoid all holes.

For the first version of our fitness function, we made it a bit confusing by giving each factor a certain weight, which made it very hard to balance it and get consistent results.

This early version of the fitness function would:

- Give a penalty to solutions that would end with less steps than the minimum.
- Penalty for how far the solution came to the goal using the Manhattan distance.
- Penalty for stepping in a hole or didn't finish.
- Check for how many steps
- Penalty for repeating positions

Since we didn't get great and consistent results, we decided to develop another variation that would help us get faster and more consistent the results desired.

For this function we removed weights and used essentially the same parameters. However, it was used as a way to converge individuals into having the desired behavior.

```
return dist_to_goal**2 + len(steps) + (not steps[-1][1]
or (steps[-1][1] and not steps[-1][2])) * map +
(len(positions) - len(set(positions)))**2
```

As we can see, the individuals would be evaluated for what they did and didn't do, making it easier to distinguish between good and bad fitness values. This was the fitness function with best overall results.

Later, we decided to add the penalty for early termination like we did in the first fitness function, so we added the following to the return line of the function:
`((len(steps) < min_steps)*(len(steps)-min_steps))**2`

However, it didn't give the results expected and it was very easily discarded as an option. Sometimes, for simple scenarios like 4x4 and 8x8, it is better to have a minimalistic function to increase the consistency of results.

4. Experiments

For our experiments we decided to do statistical tests to verify if a configuration is better than another. We decided to focus the test on 2 key values: The average fitness obtained and the average first generation to reach best fitness. To achieve this, we used the 3 following statistical tests.

4.1 Kolgomorov-Smirnov test

For our first statistical test we used Kolgomorov-Smirnov, this test allowed us to see if the data was normally distributed, allowing us to know what statistical test to apply next.

```
def test_normal_ks(data):
    """Kolgomorov-Smirnov"""
    norm_data = (data -
np.mean(data))/(np.std(data)/np.sqrt(len(data)))
    D, p_value = stats.kstest(norm_data, 'norm')

    if p_value > 0.05:
        print("Data is likely normally distributed (fail
to reject null hypothesis)")
        return True
    else:
        print("Data is not likely normally distributed
(reject null hypothesis)")
        return False
```

4.2 Wilcoxon test

For data that is not normally distributed, we used the Wilcoxon test, this test allows us to know if a configuration is statistically better than another.

```
def wilcoxon(data1,data2):
    U_statistic, p_value = stats.wilcoxon(data1,data2)

    if p_value < 0.05:
        print("There is a statistically significant
difference between the two samples.")
        return True
    else:
        print("There is no statistically significant
difference between the two samples.")
        return False
```

4.3 Student's t-test

If the data is normally distributed, we used the student's t-test, this would allow us to take the same conclusions as the Mann-Whitney test.

```
def ttest(previous_avg_fitness , current_avg_fitness):

    # Perform t-test for average fitness
    avg_t_stat, avg_p_value =
stats.ttest_rel(previous_avg_fitness,
current_avg_fitness)

    # Interpret results
    alpha = 0.05
    if avg_p_value < alpha:
        print("There is a statistically significant
difference in average fitness compared to previous
runs.")
        return True
    else:
        print("There is no statistically significant
difference in average fitness compared to previous
runs.")
        return False
```

5. Results

While working on the project, we could observe that certain configurations together with different fitness functions were much better and consistent than others.

From various testing, without using any statistical analysis we came to a configuration that satisfied us for our objective: small fitness average; able to complete the scenario in very few generations; not an overfitting solution.
These tests were all done using the 12x12 map.

The configuration that was found was the following:
- Population size: 150
- Crossover Probability: 80%
- Mutation Probability: 5%
- Mutation: Main Mutation
- Crossover: One Point Crossover
- Parent Selection: Tournament, 5
- Elitism: True, 0.02
- Fitness Function: Simple

This configuration gave us the following results after running for 30 runs with different random seeds:

Average fitness: 36.868 ± 3.273

Now, for the actual statistic analysis, we decided to work with certain variations of this main configuration. In these tests, we tested using the Kolgomorov-Smirnov test, to see if the data is normally distributed. If it is, we use two-way dependent t-Student test, because the tests use the same random seed with different configurations, so it is dependent. If it is not normally distributed, we will use the Wilcoxon test for non-parametric two sample dependent data. These tests were done with a 95% confidence interval.

Pop size	Cross Prob	Mut Prob	Cross Function	Mut Function	Elitism	Fitness Average	Passed Test
50	0.8	0.05	One_point	main	True	47.389 ± 11.49	Yes
150	0.8	0.05	One_point	delete	True	37.189 ± 3.59	No
150	0.8	0.1	One_point	main	True	37.564 ± 4.17	No
150	0.9	0.05	One_point	main	True	36.629 ± 2.78	No
150	0.7	0.05	One_point	main	True	37.808 ± 3.97	No
150	0.8	0.05	Two_point	main	True	40.410 ± 3.25	Yes
150	0.8	0.05	Two_point	main	False	45.817 ± 6.50	Yes
150	0.8	0.05	One_point	main	False	37.835 ± 2.91	No
100	0.8	0.05	One_point	main	False	41.225 ± 5.31	Yes
150	0.8	0.05	One_point	Insert	True	37.638 ± 3.54	No
150	0.8	0.05	One_point	Change_value	True	37.760 ± 3.07	No
150	0.8	0.2	One_point	main	True	38.329 ± 4.56	No

When looking at the results of the tests, we can see that the following configurations passed the test (i.e. p-value < 0.5):

Pop size	Cross Prob	Mut Prob	Cross Function	Mut Function	Elitism	Fitness Average	Passed Test
50	0.8	0.05	One_point	main	True	47.389 ± 11.49	Yes
150	0.8	0.05	Two_point	main	True	40.410 ± 3.25	Yes
150	0.8	0.05	Two_point	main	False	45.817 ± 6.50	Yes
100	0.8	0.05	One_point	main	False	41.225 ± 5.31	Yes

This means that the previous configurations are all less consistent and slower than the main configuration. However, the rest of the tests were proven to be unconclusive to whether they are better or worse than the main configuration.

The following graph demonstrates the fitness value of the main configuration through 50 generations in one run:

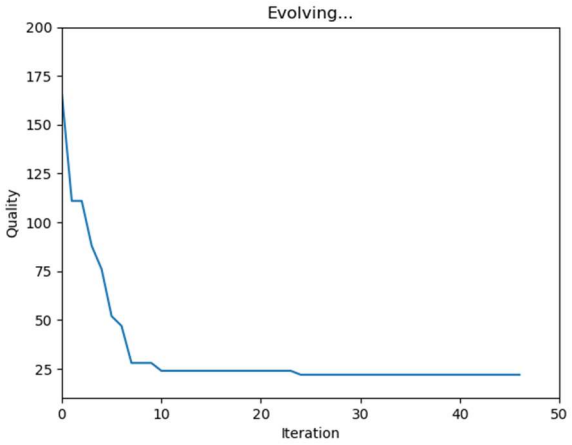

Fig. 1 – Fitness value graph of main configuration

As we can observe, the fitness value starts out very high with the first random population. And after very few generations, through the use of crossover and mutation, it reaches a solution. However, it is not the optimal solution, this comes later in the simulation.

Comparing the previous graph with the graph of the worse variation from our tests (pop size = 50), it is evident that it takes a long time for the simulation to find a good path to the goal, since there is little population to work with.

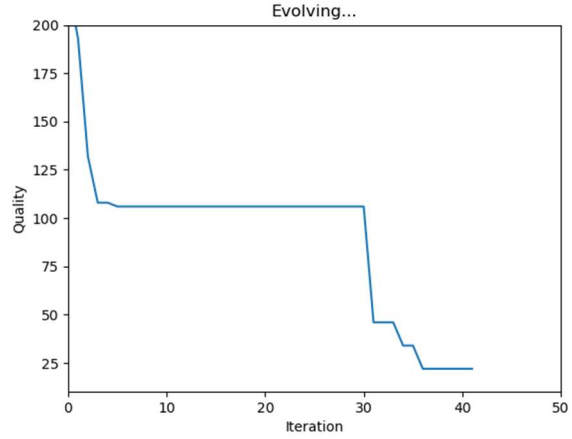

Fig. 2 – Fitness value graph of variation pop size=50

6. Discussion

Using our tests, we can now take some conclusions about their respective result.

The configurations using two-point crossover were very indicative that this crossover function statistically does not work well with our representation of the genotype.

Population sizes matters a lot, if there isn't a lot of variability in the first generation, it gets very hard to get to a solution in the 12x12 map. However, in maps 4x4 and 8x8, it the population size is not so significant to the outcome of the solution.

Since the main mutation function is a combination of the other mutation functions, there was no statistical difference between configurations. However, it is noted that the simulation took, on average, more generations to reach the best fitness using the separate mutation functions, in comparison to using the main mutation.

We couldn't conclude anything about the effect of the probability of mutation and crossover since the tests would reach unconclusive. We can see some differences, but statistically they are not significant enough to prove worse or better.

No elitism was tested using the main configuration and together with some variations. Only when reducing the population size to 100, it was noticeable the significance and the test were positive. So, we can probably conclude that elitism in this scenario is important to keep some of the best individuals for further generations. However, on other smaller maps like the 4x4, elitism seems to not do as good as in bigger maps. When using the two-point crossover function and no elitism, in contrast to the 12x12 map and two-point crossover with elitism, there is no statistically significant difference between the main configuration and this, as we can see in the following table:

Pop size	Cross Prob	Mut Prob	Cross Function	Mut Function	Elitism	Fitness Average	Passed Test
150	0.8	0.05	Two-point	main	Yes	7.21 ± 1.65	Yes
150	0.8	0.05	Two-point	main	No	6.87 ± 1.19	No

In the 4x4 map, the main configuration got an average fitness of:
6.586 +- 0.542

7. Conclusion

In our study on the Frozen Lake problem, we employed an Evolutionary Algorithm (EA) with various configurations to identify the most effective solution. Our experiments demonstrated that certain combinations of crossover and mutation functions, along with specific fitness functions, yielded more consistent and efficient results.

The optimal configuration for a 12x12 map was determined through statistical analysis, highlighting the importance of population size, crossover probability, and the use of elitism in larger maps.

Our findings suggest that while the EA approach is viable, further research could explore the impact of different parameters on the algorithm's performance in various map sizes. Also, comparing this approach with some available online about Q-learning, would be a valuable future investigation.

8. References

[1] Thomas Simonini, Q-Learning With Frozen Lake, 13/05/2024
https://github.com/simoninithomas/Deep_reinforcement_learning_Course/blob/master/Q%20learning/FrozenLake/Q%20Learning%20with%20FrozenLake.ipynb

[2] Arjit Sharma, Reinforcement Learning Using Q-Table – FrozenLake, 13/05/2024
https://www.kaggle.com/code/sarjit07/reinforcement-learning-using-q-table-frozenlake

[3] Shifei Ding, A new asynchronous reinforcement learning algorithm based on improved parallel PSO, 13/05/2024
https://www.researchgate.net/publication/333382233_A_new_asynchronous_reinforcement_learning_algorithm_based_on_improved_parallel_PSO