Advanced Techniques in Nature Inspired Search and Optimization
Lab 3

Brendan Hart (1560038)

March 16, 2018

1 Question 4

The following algorithms were used for the genetic algorithm to solve the El Farol bar problem. They make use of two classes. The first being a *Strategy* class to represent a particular strategy, which contains the vector *probs* and two matrices a and b. The second is the *Member* class which represents a member of the population, and contains a *Strategy* instance, a value to represent the $payoff$, the current *state* and whether the member is planning on going to the bar *going_bar*.

Below is the tournament selection algorithm which was used to select members of the population for crossover.

Algorithm 1 Tournament selection

```
Require: The tournament size k
Require: The population population
function TOURNAMENT_SELECTION(k, population)
    selected ← []
    for $i = 0$ to $(k - 1)$ do
        chosen ~ Unif(population)
        selected ← selected + chosen
    end for
    fittest ← []
    max_fitness ← MIN_INT_VALUE
    for member in selected do
        if fittest is empty or member.payout is greater than max_fitness then
            fittest ← [member]
            max_fitness ← member.payout
        else if member.payout equals max_fitness then
            fittest ← fittest + member
        end if
    end for
    result ~ Unif(fittest)
    return result
end function
```

The following algorithm was used to take a probability vector and normalise it, since after applying the mutation operator there was no guarantee a given probability vector would sum to 1.

Algorithm 2 Normalise a probability vector

```
Require: A vector of probability values vector
function NORMALISE(vector)
    total ← sum(vector)
    if total equals 0 then
        return [(1.0 / len(vector)) for v in vector]
    else
        return [(v / total) for v in vector]
    end if
end function
```

The mutation operator made use of samples from a Gaussian distribution *random_gauss* which takes two parameters, the mean and standard deviation. A sample was added to each value in the strategy. To ensure the values were in the correct bounds, *min* and *max* were used.

Algorithm 3 Mutate a member of the population

```
Require: A member of the population $x$
Require: The chance of mutation $prob$
Require: The standard deviation of the Gaussian noise $sd$
function MUTATE(x, prob, sd)
    if $random\_float(0, 1) \geq prob$ then
        return
    end if
    for $i = 0$ to $(len(x.strategy.a) - 1)$ do
        for $j = 0$ to $(len(x.strategy.a[i]) - 1)$ do
            $x.strategy.a[i][j] \leftarrow max(0, x.strategy.a[i][j] + random\_gauss(0, sd))$
            $x.strategy.a[i][j] \leftarrow max(0, x.strategy.b[i][j] + random\_gauss(0, sd))$
        end for
    end for
    for $i = 0$ to $(len(x.strategy.probs) - 1)$ do
        $x.strategy.probs[i] \leftarrow max(0.0, min(1.0, x.strategy.probs[i] + random\_gauss(0, sd)))$
    end for
end function
```

The below function takes a probability vector, which is not necessarily normalised, and normalises it before choosing an index according to the distribution.

Algorithm 4 Sample from a probability vector

```
Require: The probability vector $probs$
function SAMPLE_INDEX_FROM_PROBS(probs)
    normalised_probs ← $NORMALISE(probs)$
    $r = random\_float(0, 1)$
    $cumulative \leftarrow 0$
    for $i = 0$ to (len(normalised_probs)-1) do
        $cumulative \leftarrow cumulative +$ normalised_probs[i]
        if r < cumulative then
            return i
        end if
    end for
end function
```

The below function takes a strategy, the state and whether the bar is crowded in week t to produce a decision and new state for week $t + 1$.

Algorithm 5 Apply a single step of a given strategy

```
Require: The current state $state$
Require: Whether the bar is crowded $crowded$
Require: The strategy to apply $strategy$
1: function APPLY_STRATEGY(state, crowded, strategy)
2:     trans_matrix ← strategy.a if crowded else strategy.b
3:     new_state ← SAMPLE_INDEX_FROM_PROBS(trans_matrix[state])
4:     going_to_bar ← 1 if random_float(0, 1) < strategy.probs[new_state] else 0
5:     return (going_to_bar, new_state)
6: end function
```

Since it is difficult to come up with a fitness function for a given strategy, a simulation is used for a given number of weeks. The payoff for each member is calculated accordingly and this is used to decide which strategies are better than others in tournament selection.

Algorithm 6 Carry out a simulation for the given amount of weeks and calculate the payoff for each member

```
Require: The population $population$
Require: The number of weeks to simulate $weeks$
1: function EVALUATE_POP(population, weeks)
2:     pop_limit ← 0.6 * population
3:     for member in population do
4:         member.payout ← 0
5:         member.state ← 0
6:         member.going_bar ← random_float(0, 1) < member.strategy.probs[member.state]
7:     end for
8:     crowded ← len([m for m in population if m.going_bar]) ≥ pop_limit
9:     for member in population do
10:        if (member.going_bar and not crowded) or ((not member.going_bar) and crowded) then
11:            $member.payout \leftarrow member.payout + 1$
12:        end if
13:    end for
14:    for $w = 1$ to $(weeks - 1)$ do
15:        for member in population do
16:            (going, new_state) ← apply_strategy(member.state, crowded, member.strategy)
17:            member.state ← new_state
18:            member.going_bar ← going
19:        end for
20:        crowded ← len([m for m in population if m.going_bar] ≥ pop_limit
21:        for member in population do
22:            if (member.going_bar and not crowded) or ((not member.going_bar) and crowded) then
23:                $member.payout \leftarrow member.payout + 1$
24:            end if
25:        end for
26:    end for
27: end function
```

Once two members of the population are selected, a way to create a child from them was needed. The below method of crossover does not mutate the values inside either strategy. Instead discrete recombination was used, taking a row from a given strategy matrix uniformly to create the new matrices for the child, and taking uniformly from either probability vector from member

x and y to create a new probability vector.

Algorithm 7 Crossover two members of the population to create an offspring

```
Require: A member of the population $x$
Require: A member of the population $y$
 1: function CROSSOVER(x, y)
 2:     $new\_strategy \leftarrow$ new $Strategy([], [], [])$
 3:     for $i = 0$ to $(len(x.strategy.probs) - 1)$ do
 4:         $prob \leftarrow$ x.strategy.probs[i] if random_float(0, 1) $<$ 0.5 else y.strategy.probs[i]
 5:         $d$
 6:     end for
 7:     for $i = 0$ to $(len(x.strategy.a) - 1)$ do
 8:         if $random_float(0, 1) < 0.5$ then
 9:             $a\_line \leftarrow [a$ for $a$ in $x.strategy.a[i]]$
10:         else
11:             $a\_line \leftarrow [a$ for $a$ in $y.strategy.a[i]]$
12:         end if
13:         if $random\_float(0, 1) < 0.5$ then
14:             $b\_line \leftarrow [b$ for $b$ in $x.strategy.b[i]]$
15:         else
16:             $b\_line \leftarrow [b$ for $b$ in $y.strategy.b[i]]$
17:         end if
18:         $new\_strategy.a \leftarrow new\_strategy.a + a\_line$
19:         $new\_strategy.b \leftarrow new\_strategy.b + b\_line$
20:     end for
21:     return new$Member(new\_strategy, 0)$
22: end function
```

The initial population consisted of members whose strategies consist of uniform probability values. The below function creates such a member.

Algorithm 8 Creates a member of the population which has uniform probabilities.

```
Require: The number of states
 1: function UNIFORM_MEMBER(h)
 2:     $a \leftarrow []$
 3:     $b \leftarrow []$
 4:     $probs \leftarrow []$
 5:     for $i = 0$ to $(h - 1)$ do
 6:         $a \leftarrow a + []$
 7:         $b \leftarrow b + []$
 8:         $probs \leftarrow probs + (1.0/2.0)$
 9:         for $j = 0$ to $(h - 1)$ do
10:             $a[i] \leftarrow a[i] + (1.0/float(h))$
11:             $b[i] \leftarrow b[i] + (1.0/float(h))$
12:         end for
13:     end for
14:     return new $Member($new $Strategy(probs, a, b), 0)$
15: end function
```

Below is the main genetic algorithm, which makes use of the simulation to evaluate members of the population, tournament selection to select individuals, discrete recombination to crossover two members of the population, and the mutation operator described above. The algorithm runs for *timeout* number of generations, after which the most recent *population* is returned.

Algorithm 9 The main genetic algorithm

```
Require: The tournament size $k$
Require: The population size $lambda$
Require: The number of states $h$
Require: The number of weeks to simulate $weeks$
Require: The desired chance of mutation $mutation\_rate$
Require: The standard deviation used in mutation $sd$
Require: The number of generations until timeout $timeout$
function GENETIC_ALGORITHM(k, lambda, h, weeks, mutation_rate, sd, timeout)
    $population \leftarrow []$
    for $i = 0$ to $(lambda - 1)$ do
        $population \leftarrow population + UNIFORM\_MEMBER(h)$
    end for
    $num\_of\_gens \leftarrow 0$
    while (timeout equals None) or $(num\_of\_gens < timeout)$ do
        $new\_pop \leftarrow []$
        $EVALUATE\_POP(population, weeks)$
        for $i = 0$ to $lambda$ do
            $x \leftarrow TOURNAMENT\_SELECTION(k, population)$
            $y \leftarrow TOURNAMENT\_SELECTION(k, population)$
            $MUTATE(x, mutation\_rate, sd)$
            $MUTATE(y, mutation\_rate, sd)$
            $child \leftarrow CROSSOVER(x, y)$
            $new\_pop \leftarrow new\_pop + child$
        end for
        $num\_of\_gens \leftarrow num\_of\_gens + 1$
        $population \leftarrow new\_pop$
    end while
    return (population, num_of_gens)
end function
```

2 Question 5

The following experiments were carried out using the above algorithm. Each value and parameter pair was carried out for 100 repetitions for a time of 20 seconds, at which point the average attendance of the bar over the weeks for the most recent generation was recorded. Box plots for each parameter were then plotted. The initial population was created using the uniform member function defined above. The following parameters were varied:

- Standard deviation used for the Gaussian distributions.
- The mutation rate used to decide if a given member should mutate or not.
- The tournament size.
- The number of states used in the strategies.
- The population size.

2.1 Standard Deviation vs Average Attendance

The standard deviation was varied with values in the list [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]. The population size was set to 1000, tournament size to 2, mutation rate to 0.1 and number of states to 4. The simulation was ran for 10 weeks. After 100 repetitions for each value, the following box plot was the result.

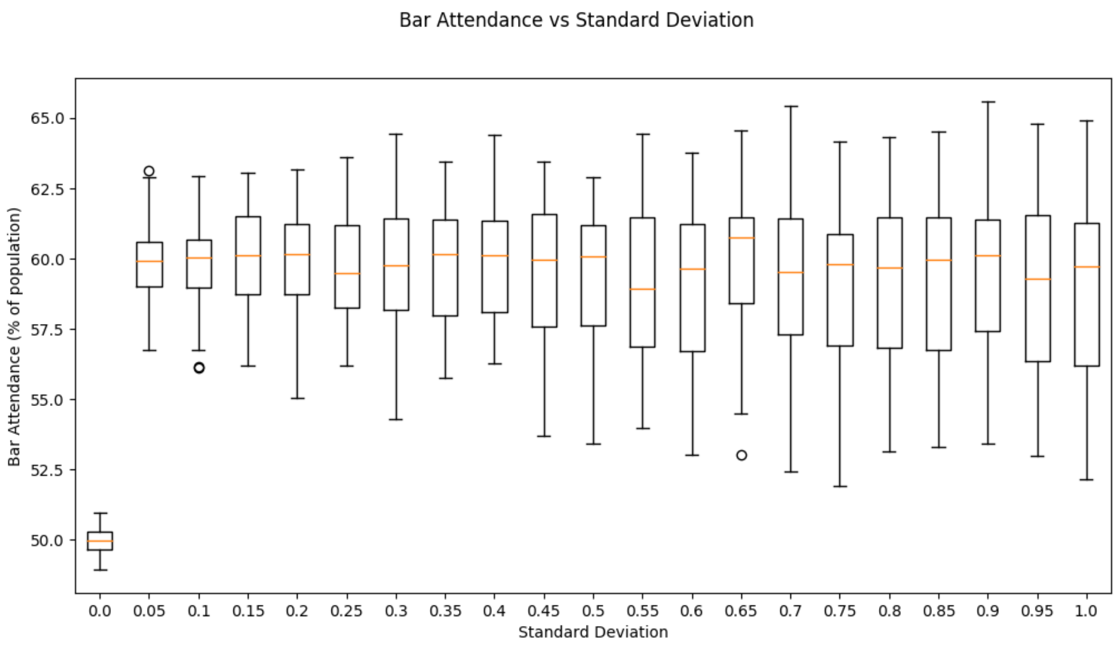

Figure 1: Bar attendance vs standard deviation.

The standard deviation seemed to have very little impact on the median of the bar attendance. The exception of that is 0.0 standard deviation. This is because the population will not mutate and instead will have to rely on crossover. Since the initial population consists of uniform strategies, lack of mutation caused by 0.0 standard deviation will cause poor performance. When considering the rest of the results, a standard deviation of around 0.05 to 0.1 seemed to perform best, achieving results very close to 60% bar attendance. The other values tend to have very high variance with means over 60% and some below. The cause for this better performance at lower standard deviation values is that once a good solution is found, it may require only small changes to turn it into an even better one. A high standard deviation may cause too big of a change to the probabilities of a good strategy, and therefore turn it into a poor strategy.

2.2 Mutation Rate vs Average Attendance

The mutation rate was varied with values in the list [0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0]. The population size was set to 1000, tournament size to 2, standard deviation to 0.1 and number of states to 4. The simulation was ran for 10 weeks. After 100 repetitions for each value, the following box plot was the result.

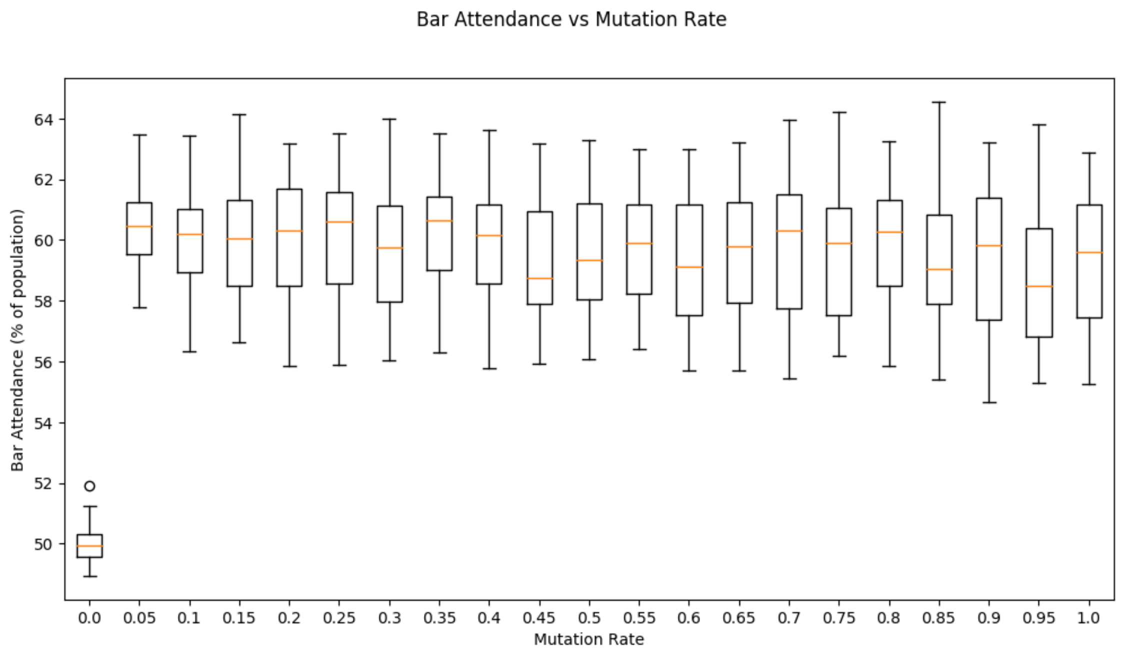

Figure 2: Bar attendance vs mutation rate.

Similarly to standard deviation, a mutation rate of 0.0 performed poorly in comparison to the rest of the values. The reason is the same, which is that there is no ability to explore the solution space in new directions, and instead only crossover is used. In the graph, there seems to be no clear indication of which particular values are best or any trend amongst the values. the ranges of the box plots overlap for all of the values for mutation rate except for 0.0. The medians of the box plots also seem to show no obvious trend, and as a result I believe the only valid conclusion is that a mutation rate of 0.0 performs poorly.

2.3 Tournament Size vs Average Attendance

The tournament size was varied with values in the list [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]. The population size was set to 1000, standard deviation to 0.1, mutation rate to 0.1 and number of states to 4. The simulation was ran for 10 weeks. After 100 repetitions for each value, the following box plot was the result.

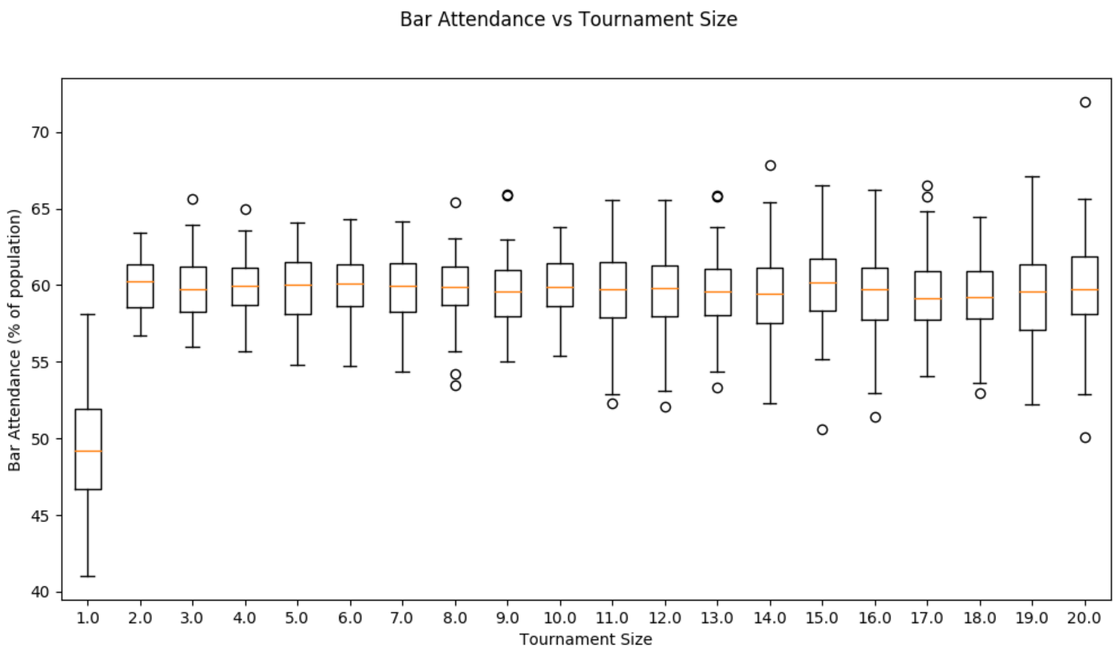

Figure 3: Bar attendance vs tournament size.

Again, there only seems to be one value of interest in the graph of tournament size. When the tournament size is 1, even the median of the bar attendance is considerably lower than the optimal, whereas the rest of the values achieve around the optimal attendance on average. The cause of this may a tournament size of 1 means any member of the population is selected regardless of its fitness value in relation to the others. This means the worst members of the population may be chosen too easily, resulting in the next generation also being as poor or worse. When the tournament size is k, we can be sure that the selected member is at least as good as k-1 members of the population. Therefore, the higher the tournament size, the more sure we are that we're picking a good member of the population in relation to the rest. However, a high tournament size reduces diversity.

2.4 Number of States vs Average Attendance

The number of states was varied with values in the list [2, 4, 6, 8, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70]. The population size was set to 1000, tournament size to 2, mutation rate to 0.1 and standard deviation to 0.1. The simulation was ran for 10 weeks. After 100 repetitions for each value, the following box plot was the result.

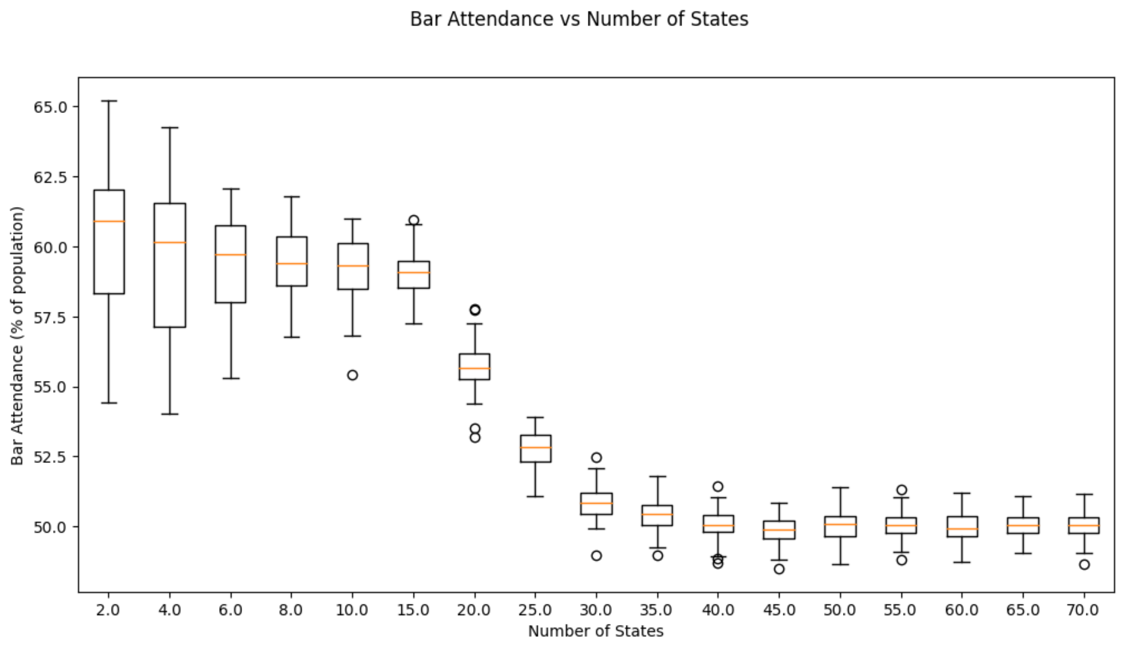

Figure 4: Bar attendance vs number of states.

The initial trend from values between 2 and 15 seem to suggest as the number of states increase, the reliability of the bar attendance falling close to 60% attendance increases. However, past value 15, the trend seems to halt. Instead, as the number of states increases, the performance of the algorithm decreases. I postulate the cause of this is that it takes the strategies longer to become close to optimal when the number of states is larger as there are more probabilities to optimize which may require more than the 20 second time budget given.

2.5 Population Size vs Average Attendance

The population size was varied with values in the list [10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 250, 350, 500, 750, 1000, 1250, 1500]. The standard deviation was set to 0.1, tournament size to 2, mutation rate to 0.1 and number of states to 4. The simulation was ran for 10 weeks. After 100 repetitions for each value, the following box plot was the result.

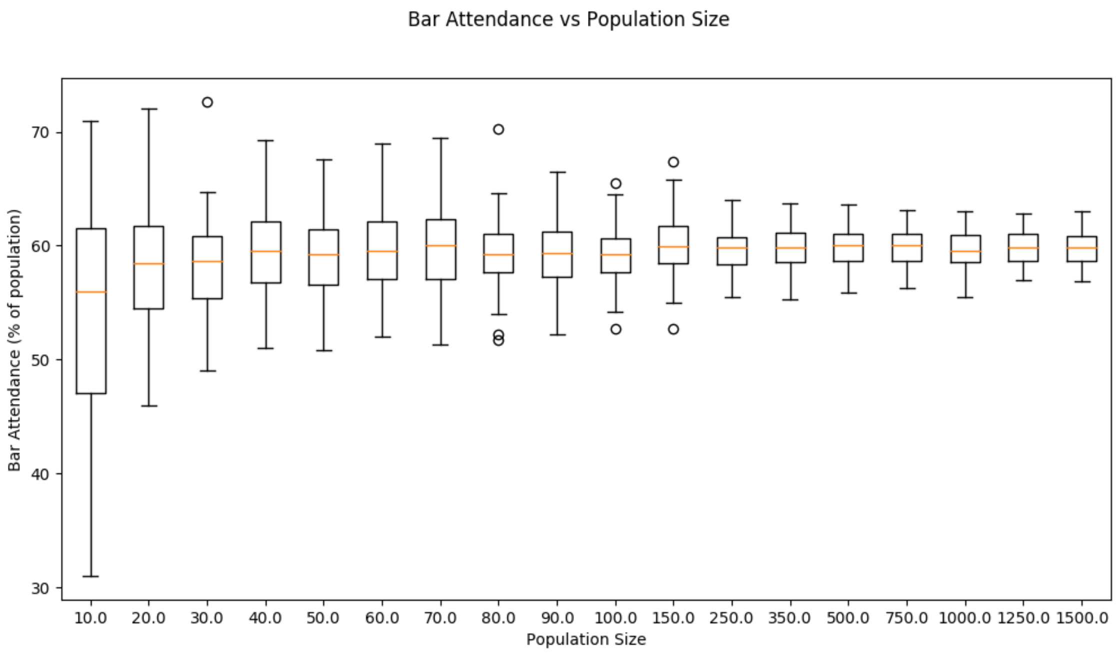

Figure 5: Bar attendance vs population size.

The median of all values of population size seemed to perform reasonably well, although the trend seems to be that higher population size results in better performance. The most obvious trend is that as population size increases, the range of the results seems to decrease and become more concentrated around the optimal bar attendance. Conclusively, it seems a higher population size results in better performance. I suggest the reason for this is that the optimal population is one that contains members whose strategies complements the rest, and result in them attending the bar at different times. As a result, diversity in the population can be beneficial and a larger population may mean a more diverse population due to the wider range of possible crossovers and mutations.