Lab 2: Maximum Satisfiability (MAXSAT)

James Atkin

---

1 Exercise 4

To solve Exercises 1, 2 & 3, I used Java to implement a genetic algorithm. My solution is outlined is pseudocode below (Algorithm 1).

A time budget of T and maximum number of iterations K are specified. The algorithm runs until the T or K is reached, producing generations of n individuals and breeding until a best solution is found. I have not specified details of how I accomplished the timeout feature as I felt this was trivial and unrelated to the genetic algorithm itself. It is also highly dependent on the Java code used to achieve it and implementations in other languages may vary considerably.

The first step in the algorithm involves ranking solutions in descending order by the number of clauses satisfied.

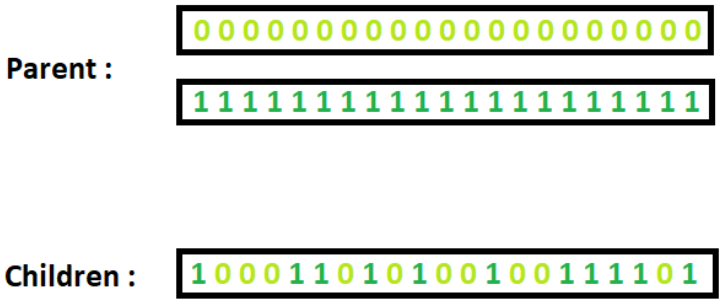
Then solutions are selected as parents for breeding. This process is two-fold, with the top E proportion automatically carried through as per elitism (Algorithm 2) and worse solutions being selected (Algorithm 3) with an exponential ranking function (Equation 1) involving a normalising factor C. Choosing this ranking function required less parameters than a power or geometric ranking function, allowing me to more finely tune the normalising factor.

$$S = \frac{1 - exp^{-i}}{C} \tag{1}$$

This selection approach ensures a balance between selecting very promising solutions for exploitation and having a chance to select worse solutions for exploration.

After selection, solutions are bred (Algorithm 4) with each other to create a new generation. This is achieved through uniform crossover (Figure 1), where each gene (digit) of the child solution has an equal chance of being from either parent. The aim of this is to provide a mix between each parent, with an added element of randomness to ensure the solution keeps evolving.

Figure 1: Uniform crossover



I decided against adding mutation to my algorithm. Crossover provides a form of variation already, motivated by the most promising solutions and producing higher quality child solutions. Mutation can work against this by muddling and disrupting fitter child solutions, taking the algorithm more time to converge and with poorer results.

Algorithm 1 Solving MAXSAT using GA

```
1:  function GENETIC_ALGORITHM(n, E, C, T, K)
2:      P ← random_population(n)
3:      t ← 0
4:      k ← 0
5:      while t < T and k < K do                    ▷ While not timed out
6:          P_new ← rank_solutions(P)
7:          P_elite ← select_elite(P_new, n, E)
8:          P_non_elite ← select_non_elite(P_new, n, E, C)
9:          P_breeding ← P_elite + P_non_elite      ▷ Use all selected as parents
10:         P_children ← breed_population(P_breeding, n)
11:         P_new ← P_elite + P_children            ▷ Carry elites forward
12:         P ← P_new
13:         t ← t + 1
14:         k ← k + 1
15:     P ← rank_solutions(P)
16:     return P[0]
```

Algorithm 2 Selecting elite individuals

```
1:  function SELECT_ELITE(P, n, E)
2:      S ← ∅
3:      g ← round_down(n * E)                       ▷ Get number of elite individuals
4:      S ← S ∪ P[0 : g]                            ▷ Only carry forward elites
5:      return S
```

Algorithm 3 Selecting non-elite individuals

```
1: function SELECT_NON_ELITE(P, n, E)
2:     S ← ∅
3:     g ← round_down(n * E)                ▷ Get number of elite individuals
4:     i ← g
5:     while i < |P| do
6:         r ← rand_ℝ(0, 1)
7:         p ← (1 − exp^−1) / C             ▷ Exponential ranking function
8:         if r > p then                    ▷ Accept with probability
9:             S ← S ∪ P[i]
10:        i ← i + 1
11:    return S
```

Algorithm 4 Breeding new individuals

```
1: function BREED_POPULATION(P, n, E)
2:     S ← ∅
3:     m ← n * (1 − E)                      ▷ Max children to breed
4:     i ← 0
5:     while i < m do
6:         j ← random(S)
7:         p_i ← P[i]                       ▷ Parent i
8:         p_j ← P[j]                       ▷ Parent j
9:         C ← ∅                            ▷ Child
10:        z ← 0
11:        while z < size(P[i]) do
12:            r ← random(0, 1)             ▷ Randomly pick parent to sample gene from
13:            if r = 0 then C ← C + p_i[z]
14:            else
15:                C ← C + p_j[z]
16:        S ← S ∪ C
17:        if i = size(P) − 1 then          ▷ If i going out of bounds
18:            i ← 0                        ▷ Rewind i
19:        else
20:            i ← i + 1
21:    return S
```

2 Exercise 5

My algorithm involved three key tuneable parameters:

- **Population size,** n: Number of solutions in a population.
- **Elite proportion,** E: Top proportion of population to carry straight through to the next generation, as per elitism.
- **Normalising factor,** C: Used in exponential ranking function in selection.

I conducted a series of experiments to determine the optimum values for each of these parameters. These were run across different files from the MAXSAT Evaluation 2017 competition:

- `kbtree/kbtree9_7_3_5_60_1.wcsp.wcnf` (280 variables, 1298 clauses)
- `frb/20-11-1.wcnf` (220 variables, 5854 clauses)

I will refer to these files as `kbtree` and `frb`, respectively.

To control the experiments, I ran each parameter's test individually with set values for the other two parameters. For each parameter value, the algorithm was run with 100 repetitions and a time budget of 10 seconds. After this, the results were averaged.

Each time, the algorithm was run for a maximum of 10 generations, or until the time budget ran out.

2.1 Population size

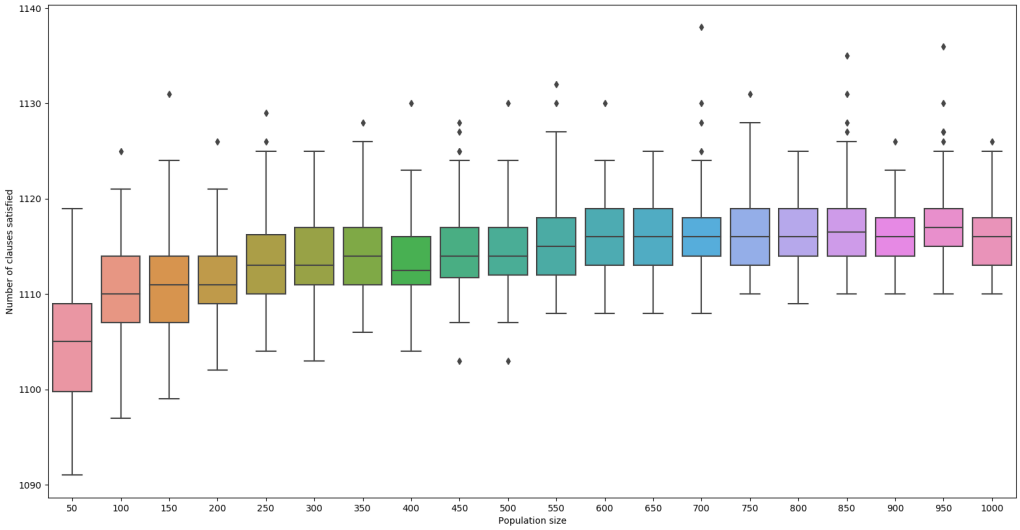
I ran my algorithm using values for n in the range [50,1000] with an interval of 50. E and C were kept constant at middling values, both of 0.5.

The results for `kbtree` are shown in Table 1 and Figure 2.

Table 1: Results of population size experiment for kbtree

Population size	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Min	1091	1097	1099	1102	1104	1103	1106	1104	1103	1103
Q1	1099	1107	1107	1109	1110	1111	1111	1111	1111	1112
Mean	1104	1110	1110	1111	1113	1113	1113	1113	1114	1114
Q2	1109	1114	1114	1114	1116	1117	1117	1116	1117	1117
Max	1119	1125	1131	1126	1129	1125	1128	1130	1128	1130
Population size	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Min	550	600	650	700	750	800	850	900	950	1000
Q1	1108	1108	1108	1108	1110	1109	1110	1110	1110	1110
Mean	1112	1113	1113	1114	1113	1114	1114	1114	1115	1113
Q2	1115	1116	1115	1116	1116	1116	1117	1115	1117	1116
Max	1118	1119	1119	1118	1119	1119	1119	1118	1119	1118

Figure 2: Box plot of population size experiment for kbtree

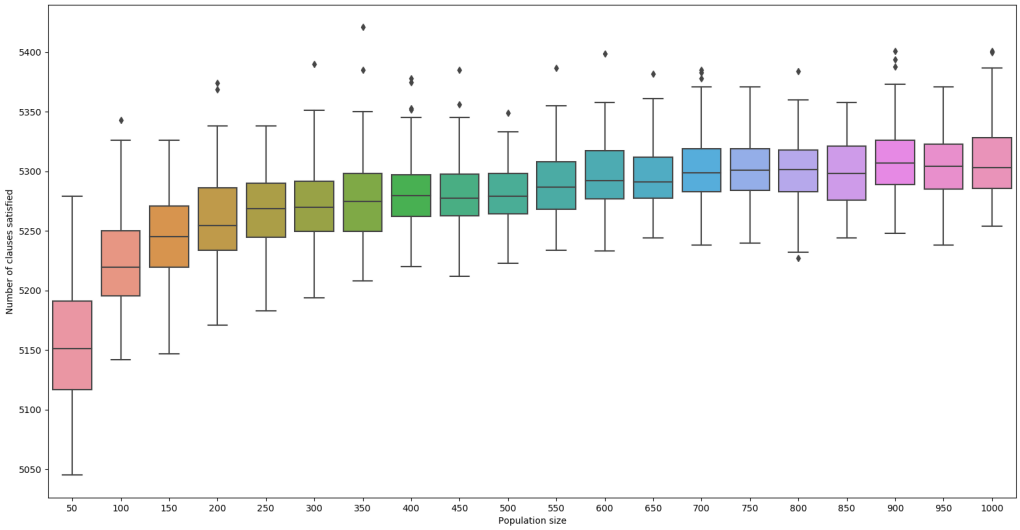


The results for `frb` are shown in Table 2 and Figure 3.

Table 2: Results of population size experiment for frb

Population size	**50**	**100**	**150**	**200**	**250**	**300**	**350**	**400**	**450**	**500**
Min	5045	5142	5147	5171	5183	5194	5208	5220	5212	5223
Q1	5117	5195	5219	5233	5244	5249	5249	5262	5262	5264
Mean	5154	5224	5242	5258	5267	5271	5277	5283	5281	5281
Q2	5191	5250	5271	5286	5290	5291	5298	5297	5297	5298
Max	5279	5343	5326	5374	5338	5390	5421	5378	5385	5349
Population size	**550**	**600**	**650**	**700**	**750**	**800**	**850**	**900**	**950**	**1000**
Min	5234	5233	5244	5238	5240	5227	5244	5248	5238	5254
Q1	5268	5277	5277	5283	5284	5283	5276	5288	5285	5285
Mean	5290	5298	5296	5303	5302	5301	5299	5310	5305	5307
Q2	5308	5317	5312	5319	5319	5318	5321	5326	5323	5328
Max	5387	5399	5382	5385	5371	5384	5358	5401	5371	5401

Figure 3: Box plot of population size experiment for frb



From these results, we see a clear trend that the number of clauses satisfied increases with the population size. This makes sense as a greater population means a greater pool of candidate solutions are available in each generation. Thus there are more available for elitism and breeding, maximising both exploitation and exploration.

However, we can also see in both graphs that this is not a linear increase. The trend resembles a curve which flattens out further along. This shows a law of diminishing returns where increasing the population size above a certain point results in little to no increase of clauses satisfied.

At this point, the population size is sufficient to explore most of the search space and create a good solution. By comparing results from the two files, the curve for `kbtree` is much flatter than that of `frb`. As the first file has fewer clauses, a smaller population size is needed to achieve the same coverage of search space, leading to the curve being flatter.

2.2 Elite proportion

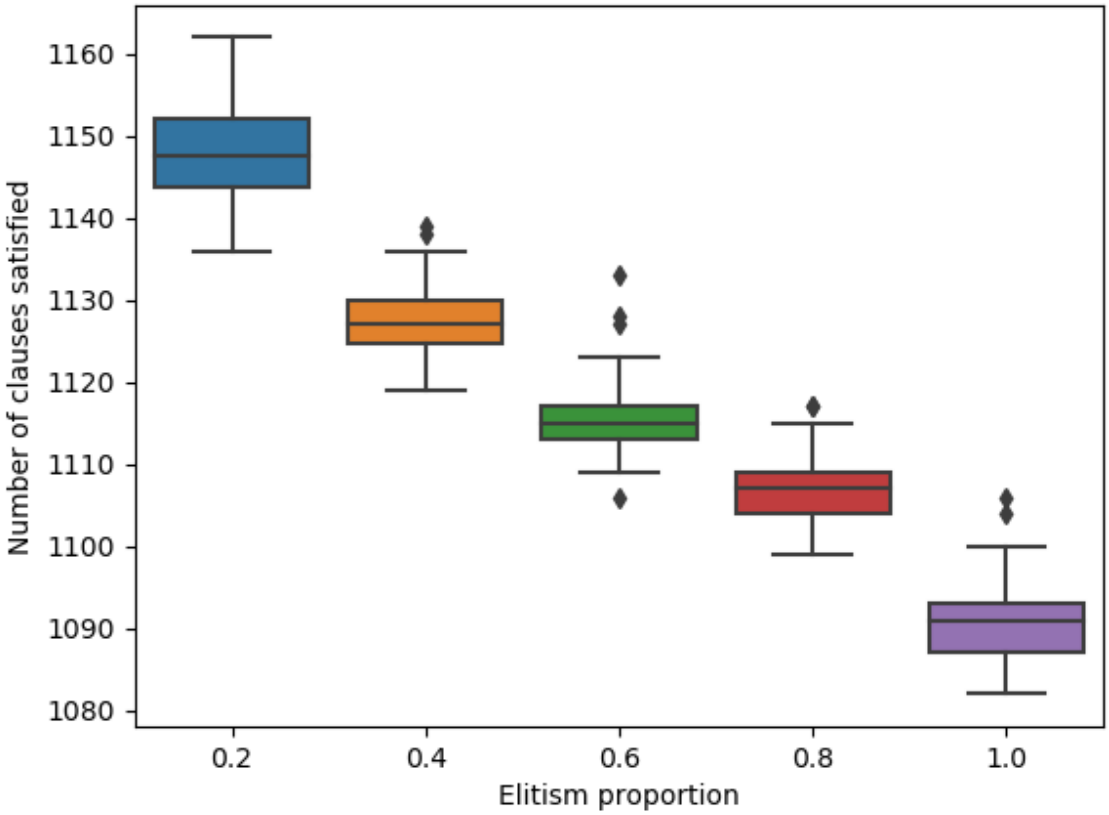
I ran my algorithm using values for E in the range [0.2,1.0] with an interval of 0.2. n and C were kept constant at middling values, of 500 and 0.5, respectively.

The results for `kbtree` are shown in Table 3 and Figure 4.

Table 3: Results of elitism proportion experiment for kbtree

Elitism proportion	**0.2**	**0.4**	**0.6**	**0.8**	**1.0**
Min	1136	1119	1106	1099	1082
Q1	1143	1124	1113	1104	1087
Mean	1147	1127	1115	1107	1090
Q2	1152	1130	1117	1109	1093
Max	1162	1139	1133	1117	1106

Figure 4: Box plot of elitism proportion experiment for kbtree

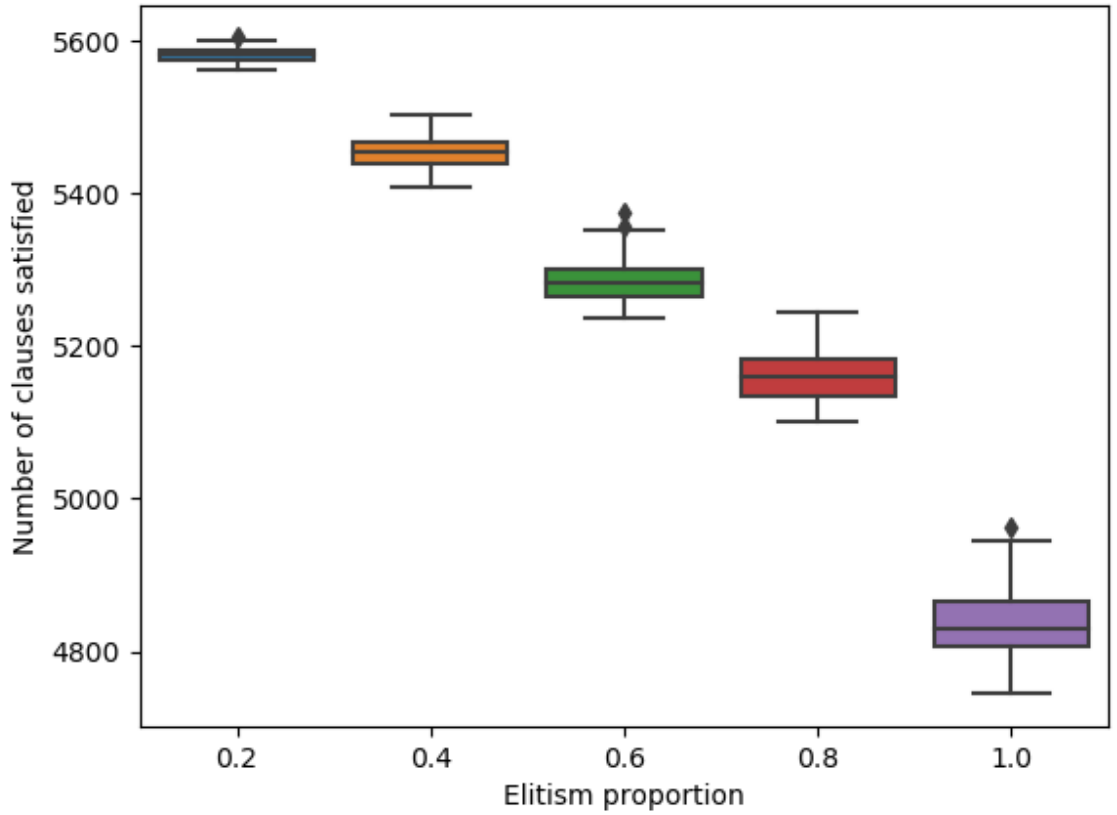


The results for `frb` are shown in Table 4 and Figure 5.

Table 4: Results of elitism proportion experiment for frb

Elitism proportion	**0.2**	**0.4**	**0.6**	**0.8**	**1.0**
Min	5563	5408	5236	5100	4744
Q1	5576	5440	5265	5134	4806
Mean	5581	5455	5286	5159	4838
Q2	5587	5468	5301	5183	4865
Max	5605	5504	5376	5244	4964

Figure 5: Box plot of elitism proportion experiment for frb



From the results for both files, we can see a clear trend that increasing the elitism proportion results in a lower number of clauses satisfied. This is consistent and there is no overlap from the means or quartiles in the box plots.

As the elitism proportion is increased, a higher percentage of the solutions are carried over immediately to the next generation, ensuring exploitation of these most promising solutions. This is also balanced with a need for exploration and so the rest of the next generation is made up of bred solutions. If the proportion of elite solutions is too great however, the exploration of the algorithm decreases, leading to it being less able to find new solutions. I believe this explanation best explains these results as the quality of the elite solutions is largely influenced by the random initial solutions. Indeed, this can be seen when 100% of the population is elite and thus does not vary from the initial generation to the last.

2.3 Normalising factor

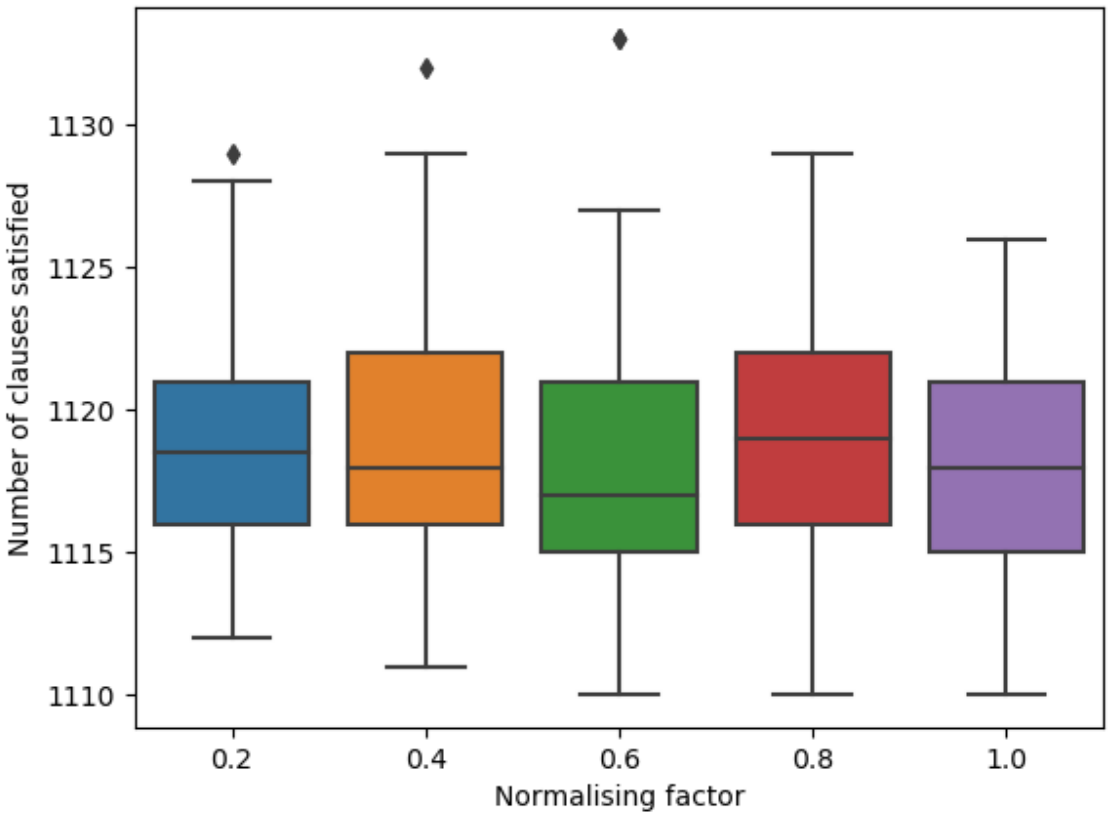
I ran my algorithm using values for C in the range [0.2,1.0] with an interval of 0.2. n and E were kept constant at middling values, of 500 and 0.5, respectively.

The results for `kbtree` are shown in Table 5 and Figure 6.

Table 5: Results of normalising factor experiment for kbtree

Normalising factor	**0.2**	**0.4**	**0.6**	**0.8**	**1.0**
Min	1112	1111	1110	1110	1110
Q1	1116	1116	1115	1116	1115
Mean	1118	1118	1118	1118	1118
Q2	1121	1122	1121	1122	1121
Max	1129	1132	1133	1129	1126

Figure 6: Box plot of normalising factor experiment for kbtree

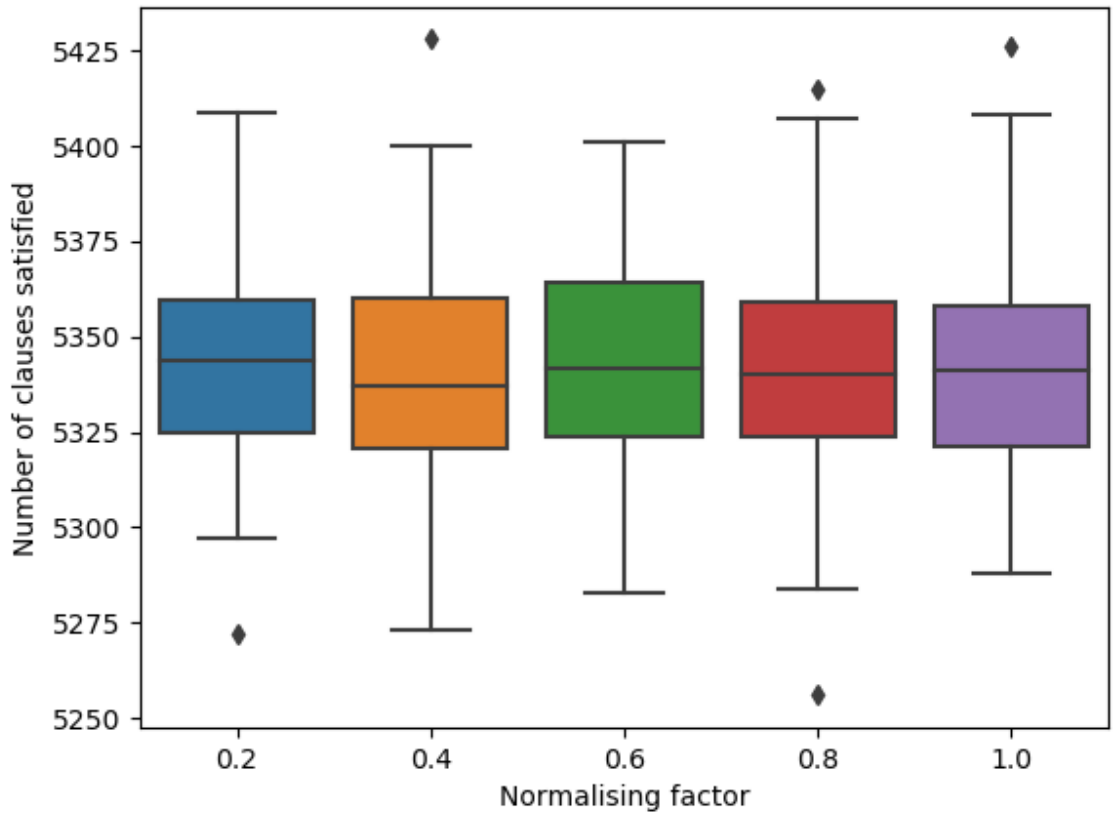


The results for `frb` are shown in Table 6 and Figure 7.

Table 6: Results of normalising factor experiment for frb

Normalising factor	**0.2**	**0.4**	**0.6**	**0.8**	**1.0**
Min	5272	5273	5283	5256	5288
Q1	5325	5320	5323	5324	5321
Mean	5343	5340	5342	5340	5340
Q2	5359	5360	5364	5359	5358
Max	5409	5428	5401	5415	5426

Figure 7: Box plot of normalising factor experiment for frb



There is no clear trend to describe the effect of varying the normalising factor in either set of results. The performance is broadly similar across all values of C. There are a couple of possible explorations for this.

The first is that C wasn't sample over a great enough range, and that there is little difference in the values sampled. $0 < C \leq \beta$ and it is possible that a trend may emerge if the experiment were re-run with $\beta > 1.0$.

The second explanation is that the effect of C on the performance of the algorithm is minor compared to that of n or E. Since C controls the quality of non-elite solutions which are used for

breeding, it is possible that convergence to a good solution can still be achieved with lower-quality children. These may explore untouched areas of the search space and produce solutions which are tempered over several generations or carried forwards via elitism.