Advanced Techniques of Nature-Inspired Search and Optimisation
Lab 2

Brendan Hart - 1560038

Question 4)

Algorithm 1: Tournament Selection

Require: Tournament size $k \in \mathbb{N}$

Require: List of pairs representing the population (where the first element is the bit string and the second is the fitness value) $population = \{p_1, p_2, ..., p_n\}$

```
1. tournament_selection(k, population)
2.     selected = []
3.     for i = 0 to k
4.         chosen ~ Unif(population)
5.         selected += chosen
6.     end for
7.     fittest = []
8.     max_fitness = MIN_INT_VALUE
9.     for i = 0 to len(selected)
10.         if fittest is empty
11.             fittest += selected[i]
12.             max_fitness = snd(selected[i])
13.         else
14.             if snd(selected[i]) > max_fitness
15.                 fittest = [ selected[i] ]
16.                 max_fitness = snd(selected[i])
17.             else if snd(selected[i]) equals max_fitness
18.                 fittest += selected[i]
19.             end if
20.         end if
21.     end for
22.     selection ~ Unif(fittest)
23.     return selection
```

The above algorithm was used to perform tournament selection, which samples k members of the population uniformly and then chooses the one with the highest fitness. If multiple have the highest fitness, one is randomly chosen.

Algorithm 2: Mutation Operator

Require: Bistring $x \in \{0,1\}^n$

Require: Expected number of mutated bits $chi \in \mathbb{R}^+$

```
1. mutation_operator(x, chi)
2.     rate = chi / len(x)
```

```
3.     result = ‘’
4.     for bit in x
5.         if random_float() < rate
6.             result += not bit
7.         else
8.             result += bit
9.         end if
10.     end for
11.     return result
```

The above algorithm was used to mutate a bitstring, with a mutation rate of chi divided by the problem size.

Algorithm 3: Fitness Function

Require: Bitstring $assignment \in \{0,1\}^n$

Require: List of clauses $clauses = \{c_1, c_2, ..., c_m\}$

```
1. fitness_function(assignment, clauses)
2.     count = 0
3.     for clause in clauses
4.         if assignment satisfies clause
5.             count = count + 1
6.         end if
7.     end for
8.     return count
```

The fitness function procedure calculates the number of clauses that a particular assignment satisfies.

Algorithm 4: Uniform Crossover

Require: Bistring $x, y \in \{0,1\}^n$

```
1. uniform_crossover(x, y)
2.     result = ‘’
3.     for i = 0 to len(x)
4.         if x[i] equals y[i]
5.             result += x[i]
6.         else
7.             random_bit ~ Unif({0,1})
8.             result += random_bit
9.         end if
10.     end for
11.     return result
```

The above algorithm describes uniform crossover, which was used in the genetic algorithm to produce the offspring of two selected MAXSAT assignments.

Algorithm 5: Genetic Algorithm

Require: List of clauses $clauses = \{c_1, c_2, ..., c_m\}$

Require: Problem size $n \in \mathbb{N}$

Require: Expected number of mutated bits $chi \in \mathbb{R}^+$

Require: Tournament size $k \in \mathbb{N}$

Require: Population size $lambda \in \mathbb{N}$

Require: Number of fittest members to be promoted $elitism \in \mathbb{N}$

Require: Fitness function $fitness\_function : \{0,1\}^n \to \mathbb{R}$

Require: The target fitness $goal\_fitness \in \mathbb{R}$

Require: The timeout in seconds $timeout \in \mathbb{N}$

```
1. genetic_algorithm(clauses, n, chi, k, lambda, elitism, fitness_function,
   goal_fitness, timeout)
2.     population = []
3.     for i = 0 to lambda
4.         population += ''
5.         for j = 0 to n
6.             rand_bit ~ Unif({0,1})
7.             population[i] += rand_bit
8.         end for
9.         population[i] = (population[i], fitness_function(population[i],
10.                                                          clauses))
11.     end for
12.
13.     num_of_gens = 0
14.     start_time = current time in seconds
15.     current_time = start_time
16.     while ((current_time - start_time) < timeout) and (goal_fitness has
   not been achieved)
17.         new_population = []
18.         sortDescendingFitness(population)
19.         for i = 0 to lambda
20.             if i < elitism
21.                 new_population += population[i]
22.                 continue
23.             end if
24.             x = tournament_selection(k, population)
25.             y = tournament_selection(k, population)
26.             mutant_x = mutation_operator(fst(x))
27.             mutant_y = mutation_operator(fst(y))
28.             child = uniform_crossover(mutant_x, mutant_y)
29.             new_population += (child, fitness_function(child, clauses))
30.         end for
31.         population = new_population
32.         num_of_gens += 1
33.         current_time = current time in seconds
34.     end while
35.     return (population, num_of_gens)
```

The above algorithm details the main structure of the genetic algorithm, and makes use of the 4 algorithms defined before it. Elitism is also implemented. The population is sorted in descending order based on the fitness value, which is the second component of the pair representation of a population member. Then, the best $elitism$ number of elements from the current population and forwarding them to the next generation. Following this, two parents are chosen via tournament selection, they are both mutated and then crossed over. The child is then added to the new population. The algorithm terminates when the time budget has passed or the maximum possible fitness value (the number of clauses) is reached.

Question 5)

The following experiments were carried out on three MAXSAT instances. The files used were brock200_2.clq.wcnf, kbtree9_7_3_5_90_6.wcnf and sbox_4.wcnf. Each test was repeated 100 times for each parameter and value combination, with the best fitness value used to calculate the quality for a particular iteration of a combination. A timeout of 20 seconds was used for all the experiments. Three parameters were chosen to compare in this report, namely Chi, Lambda and Elitism. Chi is used to calculated the mutation rate by the formula $mutation\ rate\ =\ chi\ /\ problem\ size.$ Lambda is the population size and finally elitism is the forwarding of the best n members to the next population.

Of course, between MAXSAT instances, the problem size also can vary significantly. Therefore, to ensure the values used for the parameters are comparable between instances, this report looks at percentages of the problem size as values for each parameter. E.g. if we vary a parameter with values in the list [10%, 20%, 30%], what is meant is that the experiment used values which were 10% of the problem size, 20% of the problem size, and 30% of the problem size.

When comparing the success of the algorithm between instances, the raw fitness value may not tell us much. When there are 200 clauses in a MAXSAT instance, then a fitness of 200 would be perfect. However, a fitness of 200 when there are 1000000 clauses would be terrible. As a result, this report looks at the quality, defined as $quality(p) = fitness(p)\ /\ maximum\ fitness,$ where fitness(p) is the fitness of a particular solution, and maximum fitness is the number of clauses for that particular MAXSAT instance.

Quality vs Chi

Chi was varied with the following percentages of the problem size: [0.2%, 0.4%, 0.6%, 0.8%, 1.0%, 2.0%, 3.0%, 4.0%, 5.0%, 6.0%, 7.0%, 8.0%, 9.0%, 10.0%]. The results of the experiment on the three MAXSAT instances can be found in Fig 1, 2 and 3.

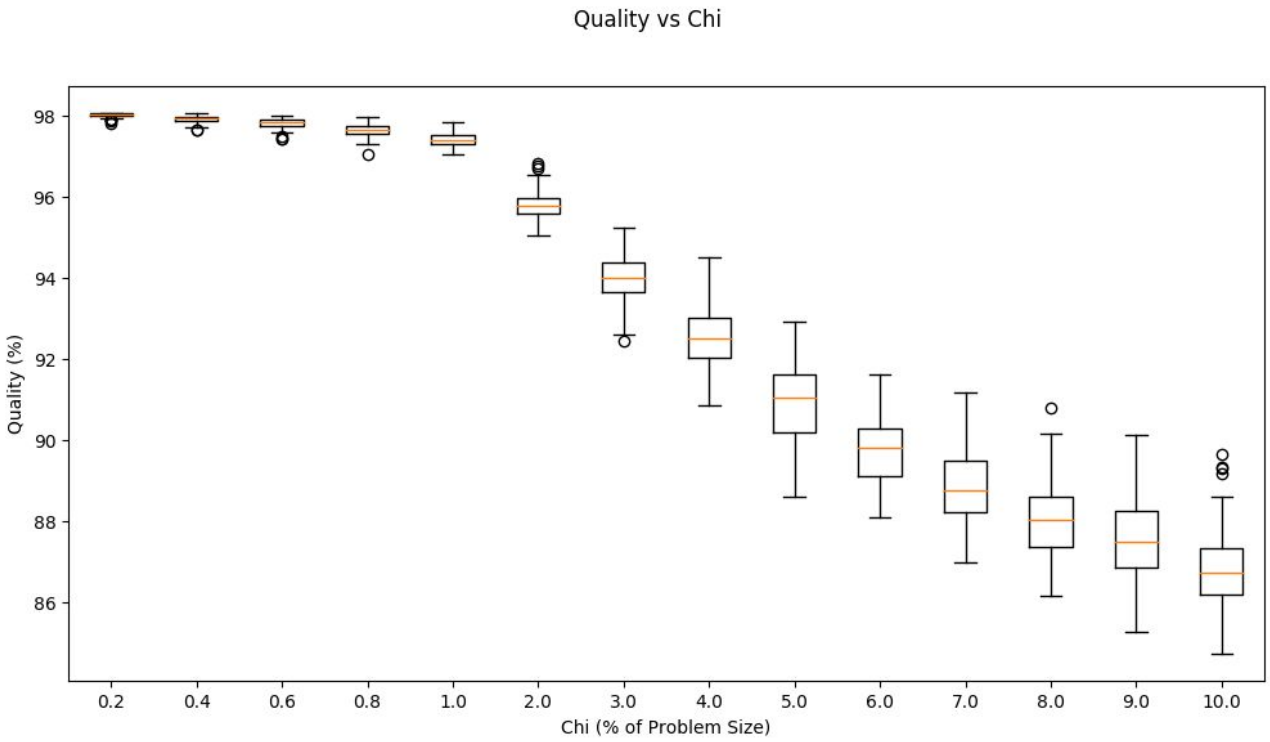

Fig 1. brock200_2.clq.wcnf

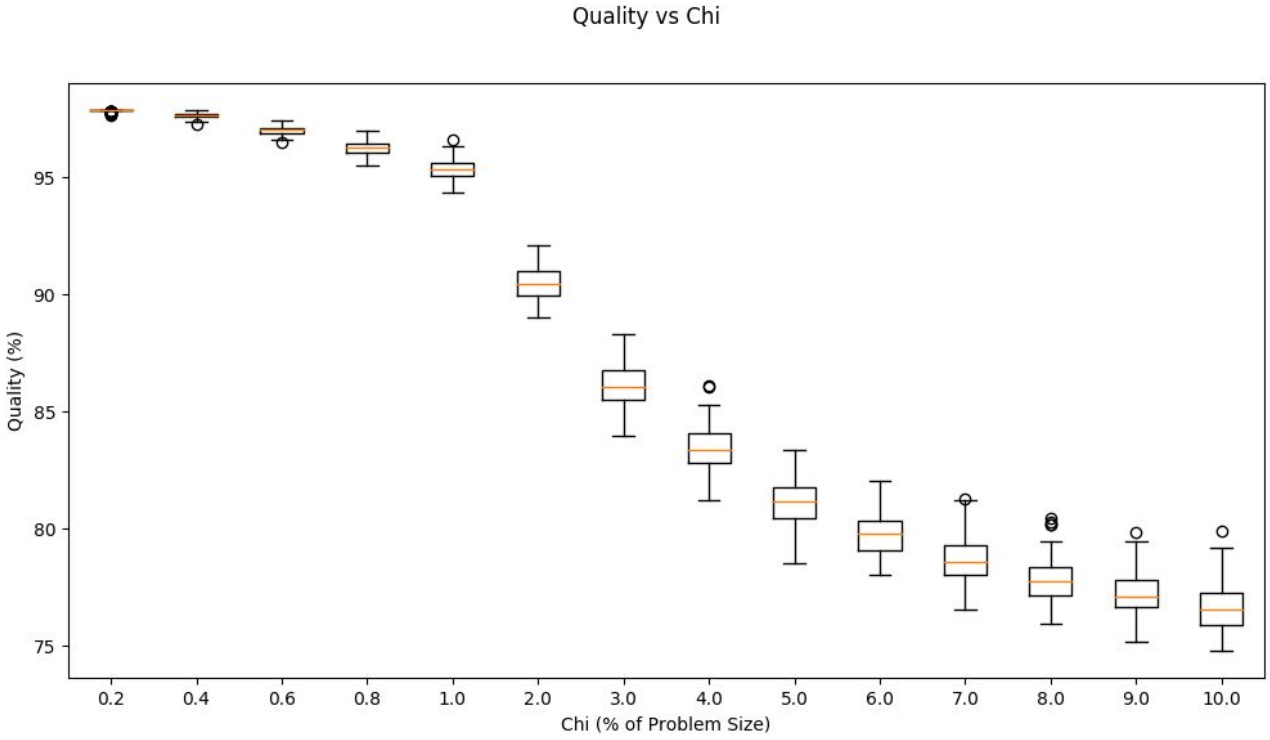

Fig 2. kbtree9_7_3_5_90_6.wcnf

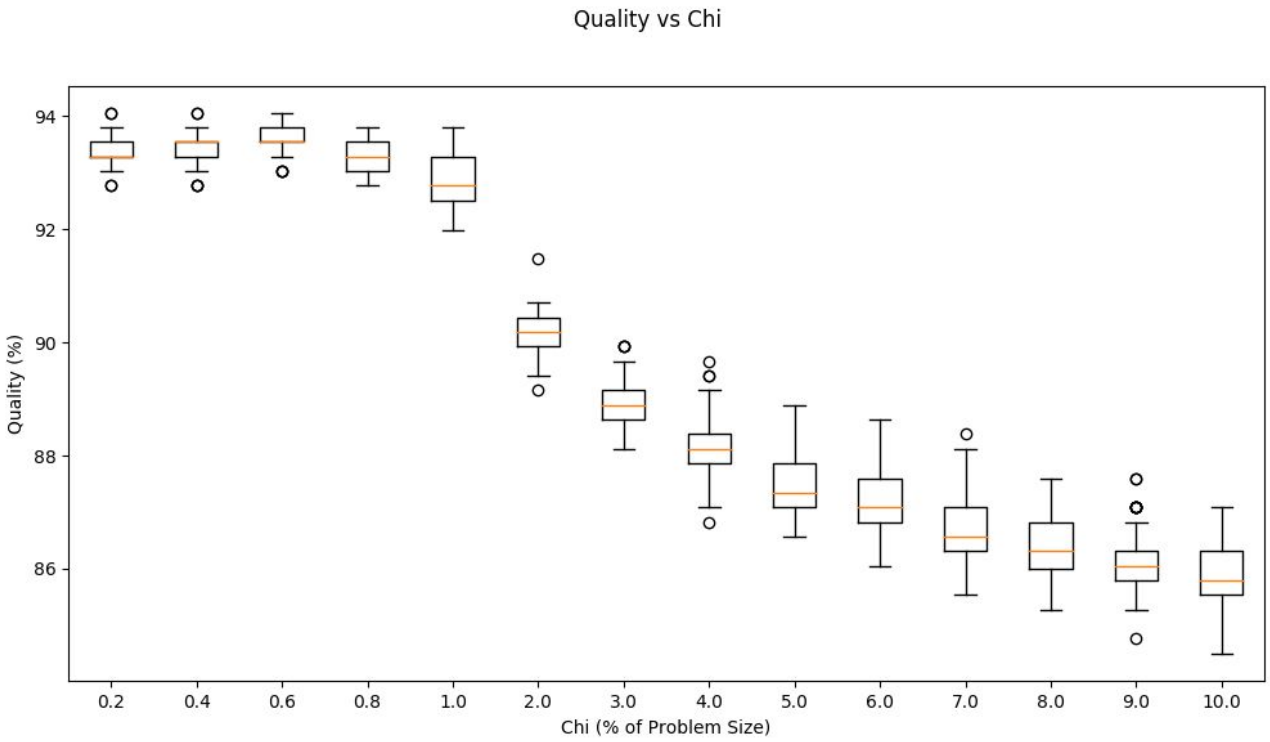

Fig 3. sbox_4.wcnf

All three of the results seem to suggest the trend that as Chi increases, the quality of the population reduces. There is a very slight difference to this trend for sbox_4.wcnf, which suggests a small increase in quality until 0.6% of the problem size, and a decrease in quality afterwards. However, this difference is miniscule and probably not of statistical significance. A possibility for a small Chi being preferred is that changing any number of variables may have a drastic effect on the quality, as a good solution may then be turned into a terrible solution, due to many clauses being able to depend on each variable. As a result, it may be better to make small changes.

Quality vs Lambda

The population size Lambda was varied with the following percentages of the problem size: [1.0%, 5.0%, 10.0%, 25.0%, 50.0%, 100.0%, 150.0%, 250.0%, 500.0%, 1000.0%]. The results for this parameter can be found in Fig 4, 5 and 6.

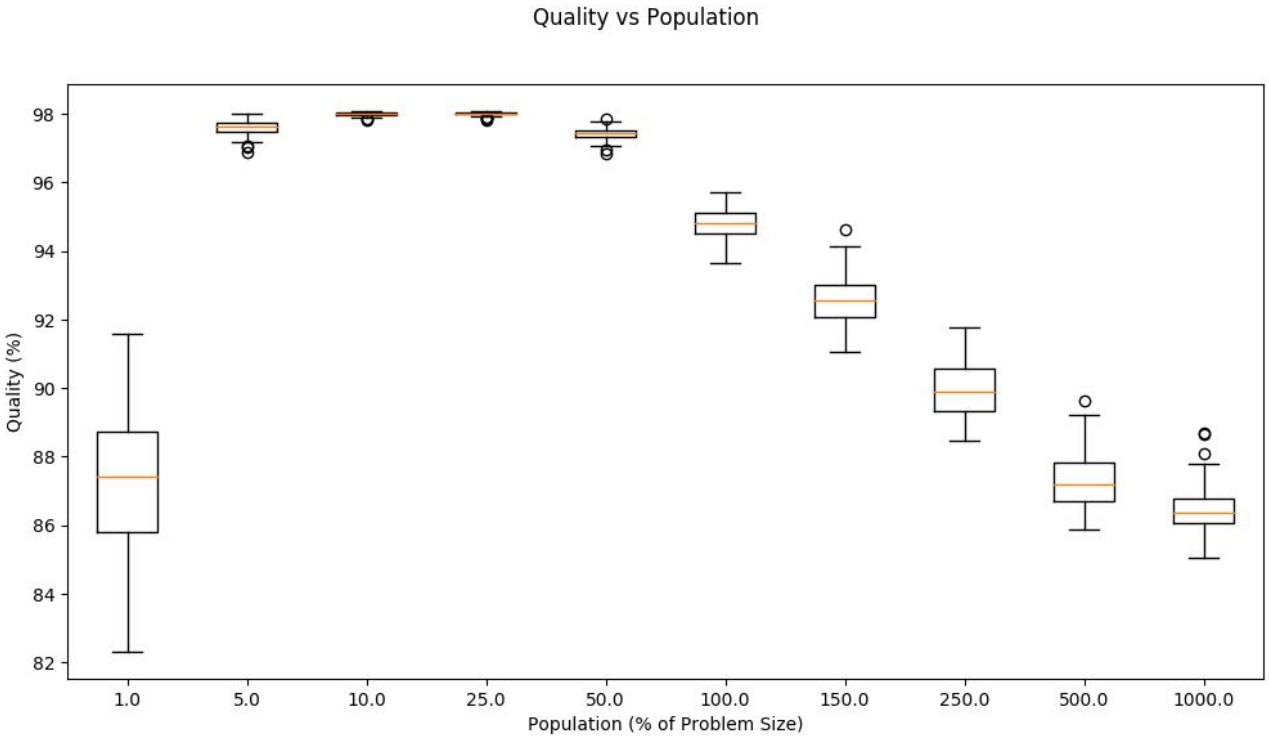

Fig 4. brock200_2.clq.wcnf

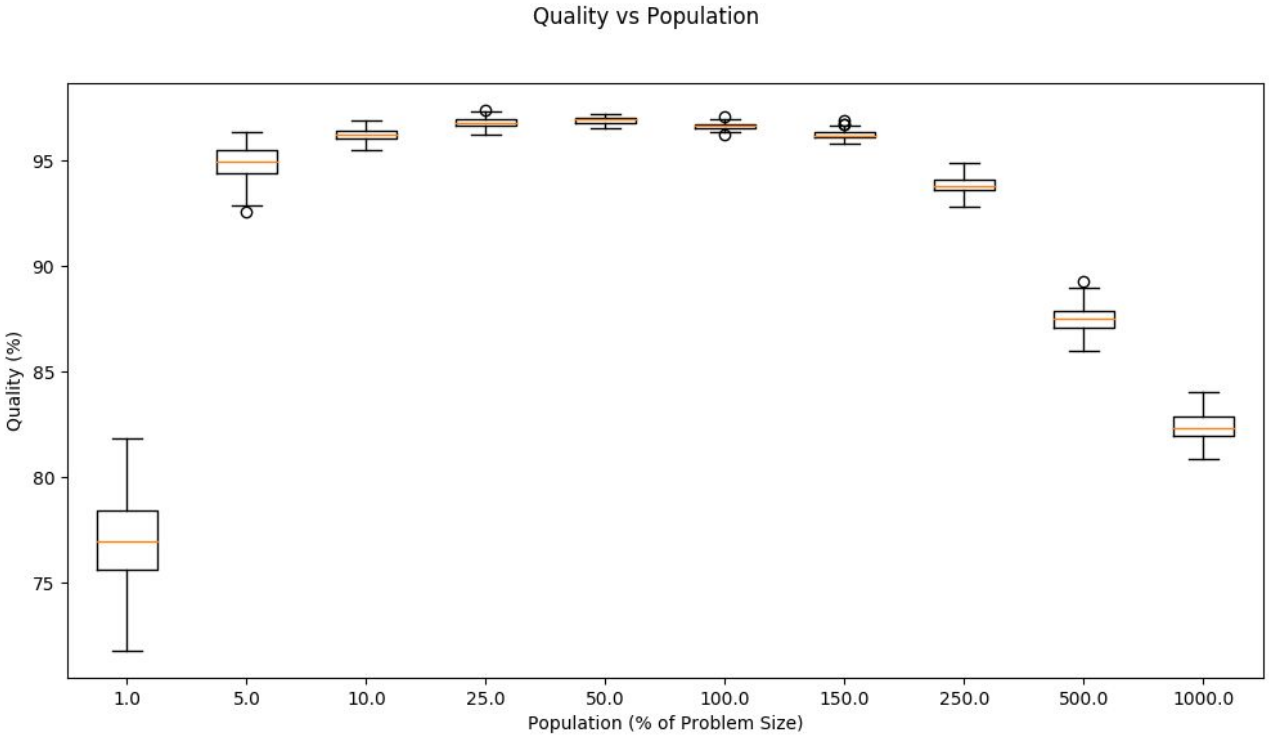

Fig 5. kbtree9_7_3_5_90_6.wcnf

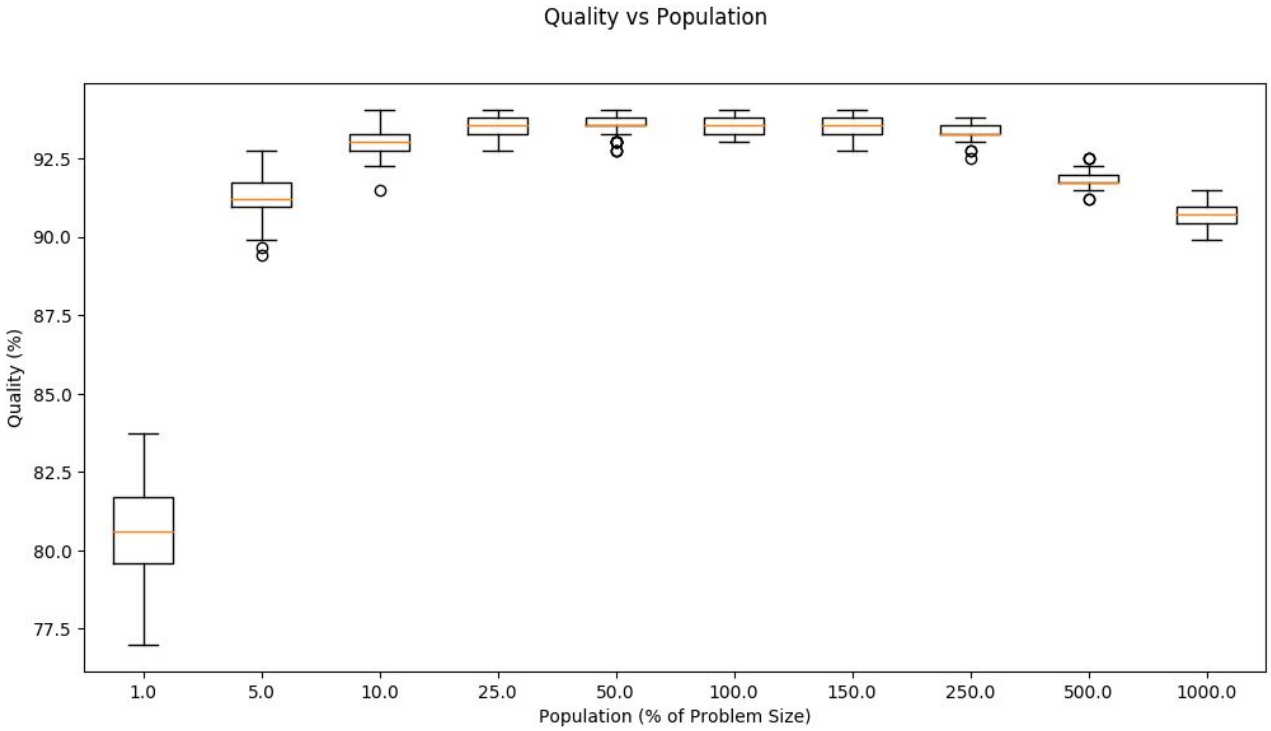

Fig 6. sbox_4.wcnf

For all three MAXSAT instances, a very low population size tends to perform poor, as shown by 1% of the problem size achieving the worst quality. However, if the problem size becomes too large, the quality also seems to decrease. The optimal population size would seem to be around 25-50% of the problem size. A small problem size may perform poorly because there is not enough diversity in the population, causing the algorithm to find a poor local optima. A large population may also be poor since in the given time budget of 20 seconds, less generations are able to be processed in the time and the population may not have enough time to evolve to reach the optimas.

Quality vs Elitism

Elitism was varied with values which were also percentages of the population size. The percentages were from the list [0.0%, 2.5%, 5.0%, 7.5%, 10.0%, 20.0%, 30.0%, 40.0%, 50.0%, 60.0%, 70.0%, 80.0%, 90.0%]. The results for varying elitism on the three MAXSAT instances are found in Fig 7, 8 and 9.

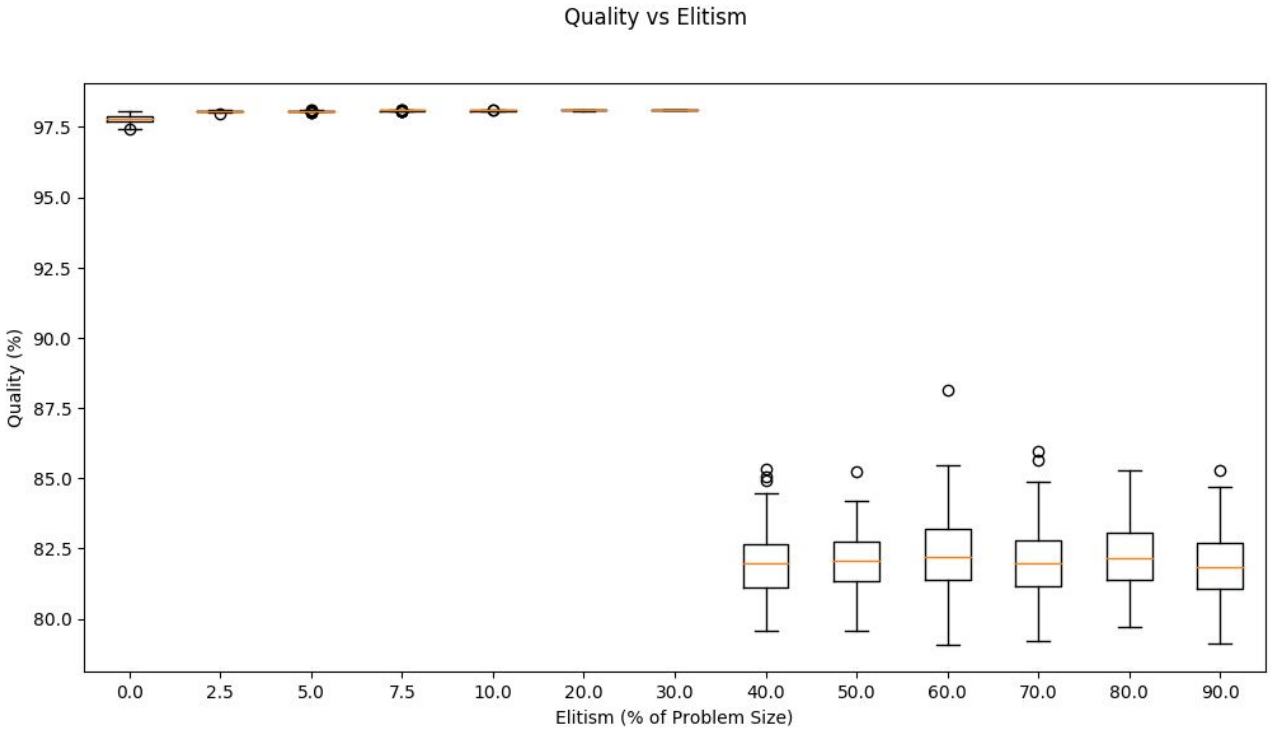

Fig 7. brock200_2.clq.wcnf

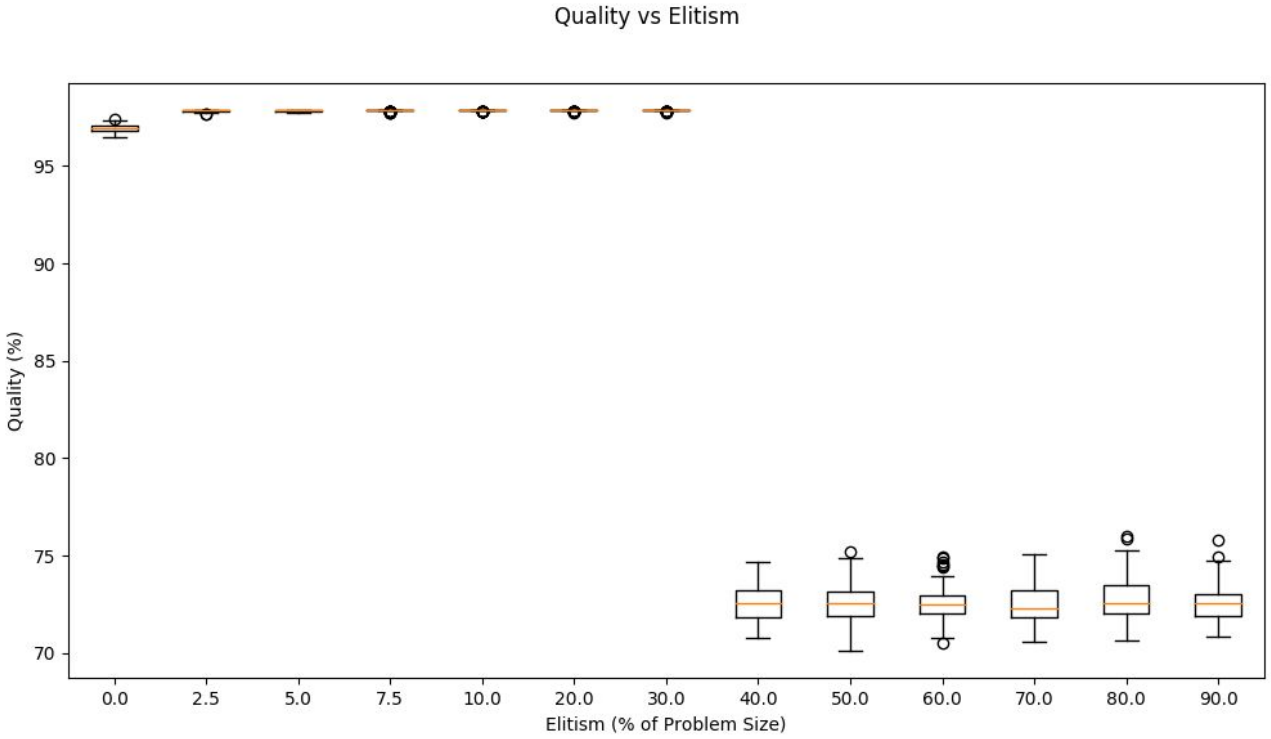

Fig 8. kbtree9_7_3_5_90_6.wcnf

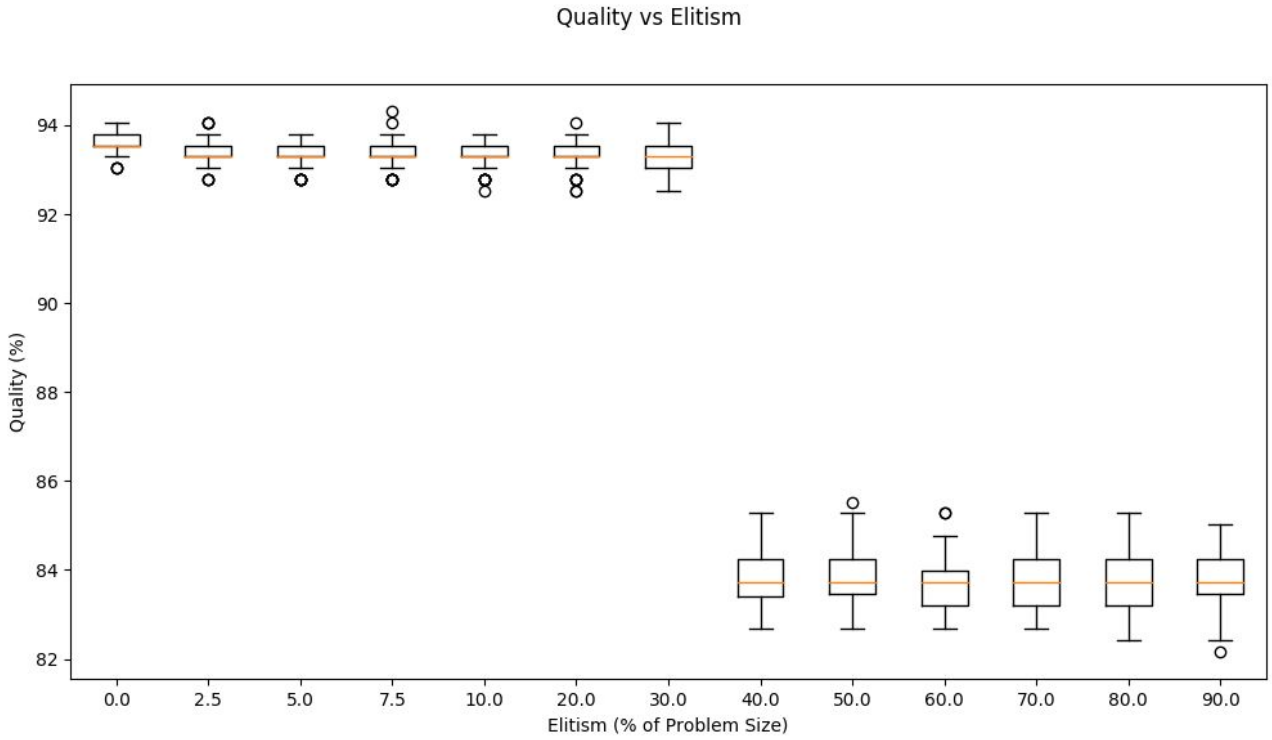

Fig 9. sbox_4.wcnf

As can be seen from the graphs, elitism seems to have very little effect on the quality of the results, up until 40% elitism (meaning a number equal to 40% of the problem size of the fittest candidates were automatically promoted to the next generation) where all three MAXSAT instances seen a dramatic reduction in quality. This quality then remained at a similar level until 90% elitism. The cause for this may be that there becomes a point where high elitism compromises the diversity of the algorithm and reduces the chance of finding a good solution.