Iterative approach to Museum Protection Problem using EA algorithms

1st Mansour Sami
School of Computing, Engineering & The Built Environment
Edinburgh Napier University
Edinburgh, United Kingdom
40617066@live.napier.ac.uk

I. Introduction

optimisation problems are ubiquitous in various domains, ranging from engineering to economics, and from healthcare to logistics [1]. One such problem is the Museum Protection Problem, where the goal is to optimally place a fixed number of cameras in a museum to maximize the total area covered. This problem is a classic example of a combinatorial optimisation problem, where the solution space is discrete and often vast, making exhaustive search infeasible [1]. To tackle such problems, stochastic search algorithms have emerged as a powerful tool [2]. These algorithms, inspired by natural phenomena such as evolution and ant colony behaviour, employ randomness in their search process to explore the solution space [2]. They have been successfully applied to a wide range of problems, including in combinatorial and continuous optimisation, single and multi-objective optimisation, and noisy and dynamic optimisation [1].

This work contributes to the growing body of research on stochastic search algorithms and their applications [3]. Tackling a real-world problem with practical implications demonstrates the power and versatility of these algorithms[Intro3]. A particular focus is the principles of different multi-objective optimisation techniques [4], the main concepts underlying the stochastic local search (SLS) framework [5], and providing a wider vision of genetic algorithms [6].

II. Approach and Methodology

This study adopts an iterative approach to problem-solving [7], which was divided into multiple rounds. Each round focused on one section of the algorithm, and applied solutions, to make the algorithm better.

Once the method/methods were implemented, the algorithm/algorithms were given the five given problem instances (MerchistonD, NMScotlandFloor5Left, NMScotlandFloor5Right, Wallstest1, Wallstest2), and the results of the fifteen runs were recorded, shown in the box plots. Note that, the values recorded for each round show the solution with the best performance on each round, the number of generations it took to calculate the best solution (NGS), the number of cameras, and the total time it took for each run.

This iterative process of problem description, algorithm implementation, experimentation, and analysis was repeated for each section of the algorithm. The insights gained from each round informed the approach taken in subsequent rounds, leading to a progressive refinement of the algorithms and solutions. This iterative approach offered a wide range of solutions, a robust understanding of the problem and the chosen methods, and continuous improvement based on the findings from each round.

III. Experiments and Design

The `CourseworkSkeleton` algorithm, the given algorithm, uses a bit string for representing the individual, which is the same size as the full map, and on each location, it randomly chooses the value of zero or one. One point crossover is used for crossover, and bit flips with the value of 0.05 for the probability of flipping the bit for mutation function. Lastly, for parent selection, tournament selection is used with the size of two parents. This algorithm uses 'EASimple' from the Deap Python library [10].

A. *Round 1: Penalty Function*

*1) **Problem Description**:* The handed version of the file had no penalty function, which led to a crash of the algorithm. The first part to focus on was to add the penalty function to the system and see the outputs and the results.

2) Algorithm Implementation: The algorithm for this instance is implemented in the file `CourseworkSkeleton Original.ipynb`

Let N be the number of cameras, S be the map size, C be the total number of cameras in the solution, and W be the number of cameras on the wall. The fitness is calculated as follows:

$$\text{fitness} = \begin{cases} S \times 2 \times (N - C), & \text{if } C > N \\ S \times 2 \times (C - N), & \text{if } C < N \\ S \times W, & \text{if } W > 0 \\ \text{Calculate the fitness}, & \text{Otherwise} \end{cases} \quad (1)$$

*3) **Experimentation and Results**:* As the results shows 1, all of the individuals are invalid, and have high fitness function, and this is the result of the wrong individual initialisation. The reason behind that is, the individual initialisation does not check how many 1's are needed, and at random chooses the value of a location on the map.

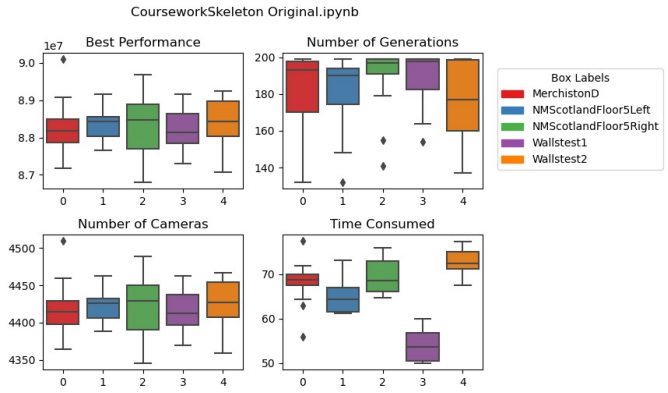

Fig. 1. Result of the algorithm for the fifteen runs

*4) **Analysis**:* This test to see how each section behaves helped to identify th sections to focus on during the individual creation stage. Using a high penalty function established valid solutions, allowing for solution removal.

TABLE I
MEDIAN VALUE OF THE RESULTS

CourseworkSkeleton Original				
	Best Fitness	NGS	Number of Cameras	Time Consumed
MerchistonD	88180000	193	4414	69
NMScotlandFloor5Left	88420000	190	4426	65
NMScotlandFloor5Right	88480000	197	4429	69
Wallstest1	88140000	198	4412	54
Wallstest2	88440000	177	4427	73

B. *Round 2: Individual Initialization*

*1) **Problem Description**:* As the results indicate in the previous round, 1 and I, the constraints for the number of cameras was violated by the high amount. This round focuses on how to make an individual initialisation which generates valid solutions without violating the constraints. Violating the constraints mean the algorithm will not progress.

*2) **Algorithm Implementation**:* Instead of creating a bit-string individual, which takes huge chunks of memory and uses high amount of computational power, the permutation is changed from 'bit string' to 'permutation with repetition'. In the 'permutation with repetition' instead of having a list the size of the whole map, and the locations we have camera with the value of one, we have a list the same size of the total number of the cameras, and we have the indexes of those cameras. At first, this leads to slower system, but this makes the program faster in later rounds, and changing the permutation gives more freedom for the mutation and the crossover operators, making it easier for to write customised function/operators. The permutation should be converted back to 'bit string' for the fitness calculation, plotting the results of the run, and visualising the solution.

The algorithm for this instance is implemented in the file `CW_v1_limitCam.ipynb`

*3) **Experimentation and Results**:* As depicted in Fig 2, the algorithm can generate valid solutions and achieve optimal fitness early on. However, this comes at the cost of a substantial increase in computational time.

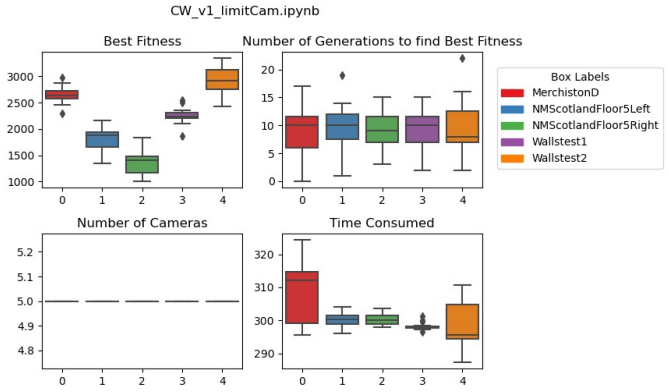

Fig. 2. Result of the algorithm for the fifteen run

*4) **Analysis**:* The results in Table II compared with the previous algorithm (Table I) reveal both positive and negative aspects of the work. Positives include a significant drop in fitness value and improved generation for finding the best solution. It now returns valid solutions instead of invalid ones. However, this has increased computational time by approximately 250 seconds per run.

TABLE II
ADD CAPTION

CW_v1_limitCam				
	Best Fitness	NGS	Number of Cameras	Time Consumed
MerchistonD	2641	10	5	313
NMScotlandFloor5Left	1876	10	5	301
NMScotlandFloor5Right	1401	9	5	301
Wallstest1	2237	8	5	296
Wallstest2	2916	8	5	296

C. *Round 3: Update Penalty Function*

1) Problem Description: The penalty function is designed to calculate the penalty function as soon as the first constraint is violated, rather than checking if other constraints are violated as well. In this round, the evaluation function is updated so that as more constraints are violated, more penalty is allocated to the solution.

2) Algorithm Implementation: Instead of using 1, a new formula for calculating the penalty function is proposed. Let N be the number of cameras, S be the map size, C be the total number of cameras in the solution, and W be the number of cameras on the wall. The fitness is calculated as follows:

$$\text{cameras penalty} = \begin{cases} N \times 10 \times (C - N), & \text{if } C > N \\ N \times 2 \times (N - C), & \text{if } C < N \\ 0, & \text{Otherwise} \end{cases} \quad (2)$$

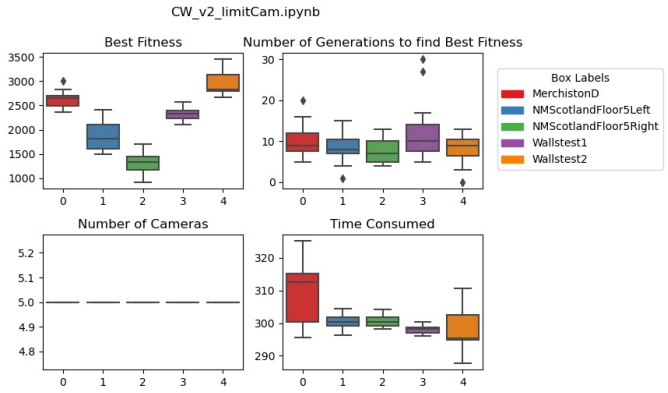

Fig. 3. Result of the algorithm for the fifteen run

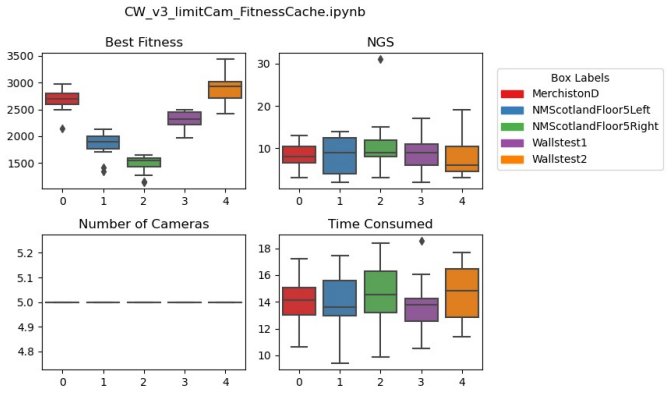

Fig. 4. Result of the algorithm for the fifteen run

$$\text{walls penalty} = \begin{cases} S \times W, & \text{if } W > 0 \\ 0, & \text{Otherwise} \end{cases} \tag{3}$$

$$\text{fitness} = \begin{cases} \text{cameras penalty} + & \text{walls penalty,} \\ & \text{if } \big(\text{cameras penalty} > 0 \vee \\ & \text{walls penalty} > 0\big) \\ \text{Calculate the fitness,} & \text{Otherwise} \end{cases} \tag{4}$$

3) Experimentation and Results: As the results shows in 3 there is not much difference between the previous penalty function 2. Due to the high penalty value, if an individual violates the constraints, the effect will be almost zero.

4) Analysis: The intention was to reduce the impact of the most violated solution to zero which was already a case.

TABLE III
ADD CAPTION

CW_v2_limitCam				
	Best Fitness	NGS	Number of Cameras	Time Consumed
MerchistonD	2648	9	5	313
NMScotlandFloor5Left	1809	8	5	301
NMScotlandFloor5Right	1328	7	5	301
Wallstest1	2327	10	5	299
Wallstest2	2836	9	5	296

D. *Round 4: Reducing the Computational time*

1) Problem Description: The previous rounds focused on how to optimise the algorithm, however this raised the issue of time. The algorithm takes around 5 minutes to run each instance of the problem, meaning running the algorithm multiple times will take a lot of time and computational power. At the same time it allows us to have higher number of population size, and the generations.

2) Algorithm Implementation: Two methods were used for this section. The first method is the fitness cache, meaning the fitness of each solution is saved for each instance. The other method was to use the 'multiprocessing' library, which allows us to use the cores parallel, which in theory should make the process faster.

3) Experimentation and Results: For the 'fitness cache' method, a dictionary is defined to save results of the solutions that we have seen and in case, we see the same solution, rather than calculating the whole fitness from scratch, we give the saved results.

'Multiprocessing' did not give any answers even after running for one hour to generate one instance. For this reason, this method was not pursued further.

4) Analysis: As the results indicate in the table 4, and figure IV there was a huge drop in the computational time, which in the future rounds leads us to run the algorithm with higher number in population and generations.

TABLE IV
ADD CAPTION

CW_v3_limitCam_FitnessCache				
	Best Fitness	NGS	Number of Cameras	Time Consumed
MerchistonD	2701	8	5	15
NMScotlandFloor5Left	1902	9	5	14
NMScotlandFloor5Right	1545	9	5	15
Wallstest1	2324	5	9	14
Wallstest2	2932	6	5	15

E. *Round 5: Individual Initialisation*

1) Problem Description: One of the constraints for the problem is we cannot have two cameras at the same location. The issue with the current initialisation method is it creates duplicate cameras in the same location, leading to violation of constraints. To prevent this problem and generate more valid solutions, the focus of this round is on the Individual Initialisation.

2) Algorithm Implementation: Instead of using 'permutation with repetition' we used 'permutations with a fixed number of certain integers' [8]. The difference is, in the previous

version, the camera could be any value from certain range, but in this one each value can be used once. The algorithm is shown below, by using the Python style pseudocode.

Algorithm 1 Permutation Function

```
function PERMUTATION(numbers, size)
    return random.sample(numbers, k=size)
end function
```

In here, size represents the number of cameras we want, num_cells represents the size of the map. The algorithm for this instance if implemented in the file `CW_v4_randomSample.ipynb`

3) Experimentation and Results: The expectation was, with less invalid solutions, the algorithm could lead to a better fitness, but to surprise, as it shown in the **??**, it did not, and the only effect it has was at the beginning it starts with valid solution.

4) Analysis: As shown in the table V in comparison with the previous version of the algorithm in table IV there wasn't much difference in the overall performance of the algorithm.

TABLE V
ADD CAPTION

CW_v4_randomSample				
	Best Fitness	NGS	Number of Cameras	Time Consumed
MerchistonD	2691	11	5	15
NMScotlandFloor5Left	1920	9	5	17
NMScotlandFloor5Right	1491	9	5	15
Wallstest1	2451	9	5	14
Wallstest2	2848	11	5	15

F. Round 6: Crossovers

1) Problem Description: In evolutionary algorithms (EAs), crossover operators play a crucial role in the reproduction process, influencing how genetic information is exchanged between individuals (or solutions) to create new offspring. Crossover is a genetic operator inspired by the process of recombination in biological evolution. It is used to combine the genetic material (chromosomes or solution representations) of two parents to produce one or more offspring with a mix of their characteristics. In this round, the comparison of three different crossovers has been done.

2) Algorithm Implementation: We compared three different crossovers, one point, two point, and uniform crossovers. One point crossover works by choosing a value between 1 to size of the parents, from beginning till that point, we use one of the parents genomes, and as for the rest from the other parent.

Two-point crossover is similar to one-point, with the difference being that instead of having one point, we have two random values from 1 to the size of the parents, and from the start till the first point from parent one, from first two second from parent two, and the rest is again from parent one.

For the uniform crossover, rather than using point/points to chose from parents, 'Independent probability for each attribute to be exchange' (indpb) is used. The value of 'indpb' is defined. At each index we generate a random value; if the random value is bigger or equal to the 'indpb' we choose from the second parent. If not, we choose from the first parent, and this process repeats until the end.

For determining the optimal value of 'indpb' in the context of a genetic algorithm using uniform crossover, the provided algorithm 2 employs Bayesian optimisation. The objective function, 'ObjectiveFunctionForoptimisation', generates a random individual and assesses its fitness using the 'EvalFunction' with the specified 'indpb' value. Utilizing the 'gp_minimize' function from scikit-optimize, Bayesian optimisation iteratively explores the 'indpb' parameter space, suggesting new values based on the evaluations of the objective function. The optimisation process efficiently converges towards the 'indpb' value that either maximizes or minimizes the fitness, depending on the specific nature of the problem. The final result is the determined optimal value for 'indpb'.

Algorithm 2 optimisation using Bayesian optimisation

```
procedure OBJECTIVEFUNCTIONFOROPTIMISATION(indpb)
    num_cells ← number of cells
    nb_cameras ← number of cameras
    individual ← []
    for i from 1 to nb_cameras do
        individual[i] ← random integer between 0 and num_cells
    end for
    fitness ← EvalFunction(individual, indpb)
    return fitness[0]                ▷ Return the fitness value
end procedure

space ← Define the search space for indpb:
        [Real(0.01, 0.5, name =' indpb')]

result ← gp_minimize(ObjectiveFunctionForoptimisation,
            space, n_calls = 200, random_state = 42)
```

There were experiments with the partially matched crossover, but due to the duplicate cameras it was not possible, which in the next round, we focus on how to prevent from it as well.

Note: The explanation provided, is original explanations, which we are generating one child, in here we are generating two child, and as for the second child it will be vise-versa of the first child. For example, if child one has the firs parent genomes from 1 till 7 and the rest from the other parent, for the second child, it has the second parents genomes from 1 till 7 and the rest from the first parent. The algorithms for this instance are implemented in the mentioned files: One-Point Crossover: `CW_v4_randomSample.ipynb`, Two-Point Crossover: `CW_v5_TwoPointCrossover_randomSample.ipynb`, and Uniform Crossover: `CW_v5_Best_Indpb_UniformCrossover_randomSample.ipynb`

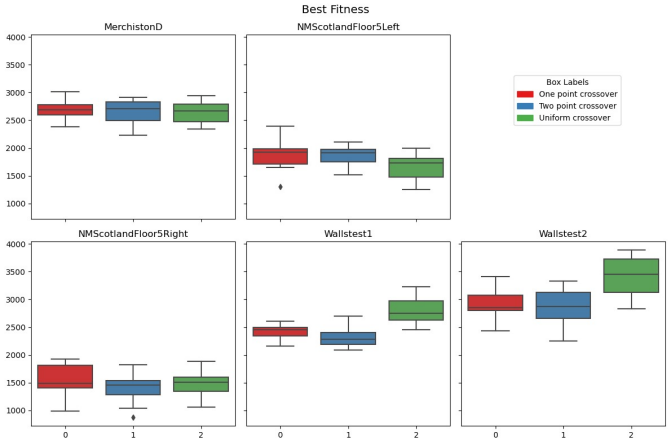

Fig. 5. Caption

3) Experimentation and Results: For this round, the main focus is on the best fitness value each algorithm can find, and as for the results in Fig 5. It is easy to notice that in most cases, one-point and two-point crossover is equal and giving the best results, except the 'NMScotlandFloor5Left' problem instance. In that instance, uniform crossover is by far found the best solutions, but for the most of the instances it's either average or the worse.

4) Analysis: As the the table VI shows, in most of the cases, one-point, and two-point are finding the best solutions, and in 'NMScotlandFloor5Right' two-point finds better solutions by 170 points difference than one-point. Uniform crossover, succeeds to find the best solutions compared to the previous two crossovers in the 'NMScotlandFloor5Left' problem instance, but other than that, in the rest is average or below, and in the 'Wallstest2' it is the worse by almost 600 points.

TABLE VI
ADD CAPTION

Crossover - Best Fitness			
	One-Point	Two-Point	Uniform
MerchistonD	2691	2704	2671
NMScotlandFloor5Left	1920	1917	1732
NMScotlandFloor5Right	1491	1456	1505
Wallstest1	2451	2279	2749
Wallstest2	2848	2870	3450

One-way Analysis of Variance (ANOVA) [9] is a statistical method that tests if the means of two or more populations are equal by comparing the variance within each group to the variance between the groups. The process involves formulating hypotheses, calculating group and overall means, calculating the sum of squares for within and between groups, calculating the degrees of freedom, calculating the mean squares, calculating the F-statistic, finding the P-value, and making a decision based on the P-value. If the P-value is less than the significance level (commonly 0.05), we reject the null hypothesis and conclude that at least one group mean is different. If the P-value is greater than the significance level, we fail to reject the null hypothesis and conclude that there is not enough evidence to say that at least one group mean is different. By in case of two by comparing both cases and in this case, comparing each case two by two, the best approach can be found. By using this methodology the best crossover for each instance is [1]:

```
MerchistonD: No significant difference
NMScotlandFloor5Left: No significant difference
NMScotlandFloor5Right: No significant difference
Wallstest1: Best Algorithm - Two Point
Wallstest2: Best Algorithm - Two Point
```

The results indicate that the two-point crossover was the best method in two of the instances, and in other instances, there weren't any significant difference. This leads to using two-point crossover as the crossover of the algorighm.

G. *Round 7: Mutation and Repair function*

1) Problem Description: The mutaion function which was used on the algorithm so far was the 'mutFlipBit' from the Deap Python library, which designed for the bit string permutation, which is not fit for the current permutaion is used on the algorithm ('permutations with a fixed number of certain integers'). In this round we focus on what permutaions are fit for the current state of the algorithm, and what could be done to make it better.

2) Algorithm Implementation: In our search for built-in mutation from the Deap library, we stumbled upon different mutation, which either were unfit for the current permutation, or they needed high computational power. It lead to having costume mutation 3. The way custom mutation function designed is, prevent individual to have the same values, it already have, and in case, one of the indexes changes to a new value, it prevents the next values to have the same value, which leads to introducing new individuals rather than the ones it already exists.

There is another mutaion function, which comes with a repair function within it as well, which is similar to mutaiotn function, and the only difference is in case of less cameras than total number of cameras or more, it added or removes cameras 4.

3) Experimentation and Results: As the results indicate in fig 6, it is visible that the costume mutation outperforms the the costume mutation function by far.

The interesting thing about the repair function is the amount of noise it had, fig 8.

4) Analysis: The one-way ANOVA was used for significant test. The results are:

```
MerchistonD: Best Algorithm - mutation
NMScotlandFloor5Left: No significant difference
NMScotlandFloor5Right: No significant difference
Wallstest1: No significant difference
Wallstest2: Best Algorithm - mutation
```

[1]The program to calculate the best crossover is in `Run Record - v5.ipynb`

Algorithm 3 Custom Mutation Function

```
function CUSTOM_MUTATE(individual, num_cells_rangeSet, indpb)
    possible_values ← list(num_cells_rangeSet − set(individual))
    for i ← 0 to len(individual) − 1 do
        if random.random() < indpb then
            if possible_values then
                unused_value ← random.choice(list(possible_values))
                possible_values.remove(unused_value)
                individual[i] ← unused_value
            end if
        end if
    end for
    return individual,
end function
```

Algorithm 4 Custom Mutation & Repair Function

```
function CUSTOM_MUTATE_REPAIR(individual, num_cells_rangeSet, indpb)
    possible_values ← list(num_cells_rangeSet − set(individual))
    while len(individual) < nb_cameras do
        if possible_values then
            unused_value ← random.choice(list(possible_values))
            possible_values.remove(unused_value)
            individual.append(unused_value)
        end if
    end while
    while len(individual) > nb_cameras do
        individual.pop()
    end while
    Continue with similar steps as in the Custom Mutaion Fucntion 3
    return individual,
end function
```

As the results of the one-way ANOVA shows, costume mutation is better than the mutation with the repair function.

Costume Mutate - Best Fitness		
	Mutate	Mutate & Repair
MerchistonD	2449	2730
NMScotlandFloor5Left	1470	1790
NMScotlandFloor5Right	1196	1506
Wallstest1	2059	2272
Wallstest2	2619	2924

Another interesting finding was, the computational time used, which in the costume mutation the median value was around 34 seconds, while the costume mutaion with the repair function took around 17 seconds.

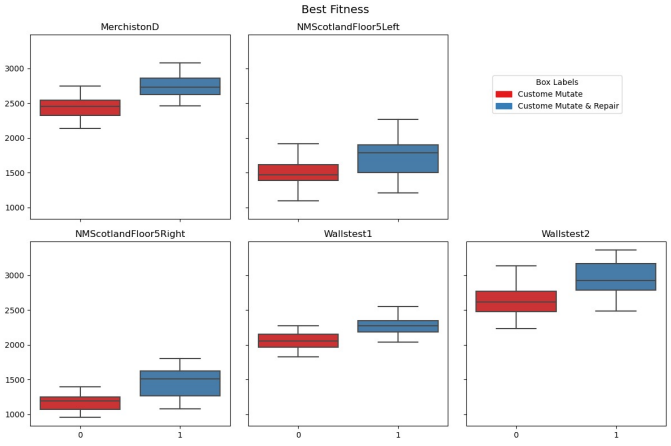

Fig. 6. Caption

Fig. 7. Plot of the custom mutation with repair function

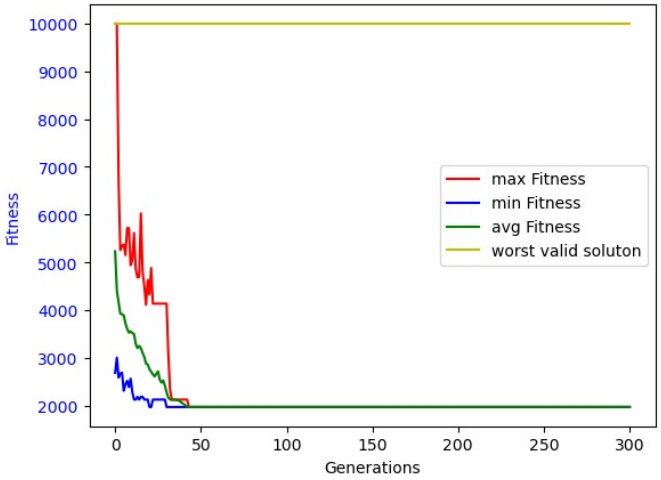

Fig. 8. Costume Mutate Repair Plot

IV. Solution Quality

The end algorithm is in the `CW_v7_customeMutateRepair.ipynb`, which is 'permutations with a fixed number of certain integers', using random.sample as the individual initialisaiton, two-point crossover, and the costume mutaion, with the 200 generations, and 50 population size.

TABLE VII
Solution Quality

Algorithm					
	MerchistonD	NMScotlandFloor5Left	NMScotlandFloor5Right	Wallstest1	Wallstest2
Best Fitness	2037	1142	840	1779	2157
Cordinates	6745, 1880, 6209, 7880, 1928	8479, 2921, 3088, 2460, 8218	8620, 5941, 2579, 1819, 7875	9051, 3779, 7484, 6118, 1634	8416, 1878, 2322, 8576, 5054
Number of Evaluations	1000	1000	1000	1000	1000
Random Search	2656	1629	1564	2593	2853

V. Conclusion

In this work, there were experiments on the methods to increase the fitness value of the solutions of the evolutionary algorithm and reducing computational time. two different permutaions, and

individual initialisation, three different crossover methods, and two different mutaion systems, which one of them had a repair function. In this research we did not take a look at the methodologies to increase the diversity, like island, population management, and saw-tooth. At the same time, we did not delve into the different section of the algorithm as well. It was limited by time.

VI. Future Work

There are lots of algorithms to implement and test to increase the diversity

References

[1] P. K. Lehre and P. S. Oliveto, "Theoretical Analysis of Stochastic Search Algorithms," arXiv:1709.00890 [cs.NE], Sep. 2017.

[2] S. Allassonniere, "Stochastic Algorithms and Their Applications," Algorithms, Special Issue, 2022.

[3] M. Affenzeller, S. Winkler, S. Wagner, and A. Beham, "Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications," CRC Press, 2009.

[4] "A Comprehensive Review on Multi-objective optimisation Techniques: Past, Present and Future" by Shubhkirti Sharma & Vijay Kumar2. This paper explains the principles of different multi-objective optimisation techniques and their applicability in various real-world application domains.

[5] "Stochastic Local Search Algorithms: An Overview" by Holger H. Hoos & Thomas Stützle5. This paper gives an overview of the main concepts underlying the stochastic local search (SLS) framework and outlines some of the most relevant SLS techniques.

[6] "A review on genetic algorithm: past, present, and future" by Sourabh Katoch, Sumit Singh Chauhan & Vijay Kumar6. This review will help the new and demanding researchers to provide the wider vision of genetic algorithms.

[7] M. Andrews, L. Pritchett and M. Woolcock, "Doing iterative and adaptive work," CID Working Paper Series, 2016.

[8] P. Diaconis, J. Fulman, and R. Guralnick, "On fixed points of permutations," Journal of Algebraic Combinatorics, vol. 28, pp. 189–218, 2008.

[9] A. Ross and V. L. Willson, "One-Way ANOVA," in Basic and Advanced Statistical Tests, SensePublishers, Rotterdam, 2017.

[10] Back, Fogel and Michalewicz, "Evolutionary Computation 1 : Basic Algorithms and Operators", 2000.