Time Series Prediction with Genetic Programming

Edward Celella
emc918@student.bham.ac.uk

Algorithm Design

This section outlines the algorithm developed and utilised in exercise 3. Any given variables (e.g. parameters) are proceeded by the **Data** tag. The algorithm makes use of a class object called $Node$, which is used to construct trees. The classes attributes are described below. Any solution/tree generated is therefore an object of this class. Node objects attributes are accessed using a decimal point followed by the attribute name.

$Node$:

$operation \rightarrow$ Operation to perform on children.
$branches \rightarrow$ A list of children node objects.
$size \rightarrow$ The amount nodes beneath the node in the tree, including itself.
$depth \rightarrow$ The depth of the tree from the Node down.
$fitness \rightarrow$ Float value which stores calculated fitness (only used in root nodes).

The pseudocode below shows the main driving function. Each sub section of this chapter expands upon utilised functions.

Data: $populationSize \rightarrow$ Size of the population.
Data: $timeBudget \rightarrow$ Maximum time to run algorithm.

$population := [\,]$
For $i := 0$ **to** $populationSize$ **do:**
    $newTree := GenerateRandomTree()$
    $newTree.fitness := Fitness(newTree)$
    $population :: newTree$
End

$elapsedTime := 0$
While $elapsedTime < timeBudget$ **do:**
    $population := GenerateNewPopulation(population)$
    $elapsedTime := UpdateTime()$
End

$population := Sort()$
Return $population[0]$

Where $Fitness()$ calculates the mean squared error of a solution, $UpdateTime()$ records the time elapsed since the start of the algorithm, and $Sort()$ sorts a list of solutions by fitness in ascending order.

Generate Random Tree

Data: $numberFeatures \rightarrow$ The number of input values.
Data: $maxDepth \rightarrow$ Maximum depth of trees allowed.

Function GenerateRandomTree($numberFeatures, maxDepth$) :

If $maxDepth <= 1$ **then**
If $R(0,1) < 0.3$ **then**
$operation :=$" $const$ " \\ Insert constant
Else
$operation :=$" $data$ " \\ Insert variable
End
$size := 1$
$constant := RI(0, numberFeatures)$
$depth := 1$
Return $Node(operation,\ constant,\ size,\ depth)$
End

$operation,\ numberBranches := RandomOperation()$

$branches := [\,]$
$size := 0$
$depth := 0$
For $i = 0$ **to** $numberBranches$ **do:**
$child := GenerateRandomTree(numberFeatures, maxDepth - 1)$
$branches :: child$
$size := size + child.size$
If $child.depth > depth$ **then** $depth := child.depth + 1$ **End**
End

Return $Node(operation,\ branches,\ size,\ depth)$

End

Where $RI()$ generates a random integer from 0 to $numberFeatures$ as a list of length one, and $RandomOperation()$ returns a random operation from a predefined list, as well as the amount of branches (children) required for that operation (e.g. addition requires two children: $x + y$, exponential required one child: e^x etc.). This functions behaviour is implemented as a hash table in the code base.

Generate New Population

Data: $population$ $\rightarrow$ List of Node objects.
Data: $populationSize$ $\rightarrow$ Size of the population list.
Data: $mutationProbility$ $\rightarrow$ Probability of mutation.
Data: $eliteRate$ $\rightarrow$ Amount of original population to keep.

Function $GenerateNewPopulation(population,\ populationSize,\ mutationProbability,\ eliteRate)$:

$children := [\,]$
$totalChildren := 0$

While $totalChildren < populationSize$ **do:**

$parent_1, parent_2 := Select(population)$
$child_1, child_2 := Crossover(parent_1,\ parent_2)$

If $R(0,1) < mutationProbability$ **then** $child_1 := mutate(child_1)$ **End**
If $R(0,1) < mutationProbability$ **then** $child_2 := mutate(child_2)$ **End**

$child_1.\,fitness := Fitness(child_1)$
$child_2.\,fitness := Fitness(child_2)$

$totalChildren := totalChildren + 2$
$children :: child_1 :: child_2$
End

$population := Sort(population)$
$children := Sort(children)$

$newPopulation := population[0$ **to** $eliteRate]$
$newPopulation :: children[0$ **to** $totalChildren - eliteRate]$

Return $newPopulation$

End

Where $R(0,1)$ generates a uniformly random number between 0 and 1, $Fitness()$ calculates the mean squared error of a solution, and $Sort()$ sorts a list of Node objects (the sorting mechanism is explained in the corresponding section).

Select

The following algorithm is the selection mechanism for the genetic algorithm. It is an implementation of tournament selection.

Data: $tournamentSize \rightarrow$ Size of the tournament selection.
Data: $population \rightarrow$ List of Node objects.
Data: $populationSize \rightarrow$ Size of the population.

Function $Select(population,\ populationSize,\ tournamentSize)$:

$selectedSolutions := [\,]$
$amountSelected := 0$

While $amountSelected < tournamentSize$ **do:**

$randonNum := RI(0, populationSize)$
$tree := population[randomNum]$
$alreadySelected := False$

For $i = 0$ **to** $amountSelected$ **do:**
If $tree \equiv selectedSolutions[i]$ **then**
$alreadySelected := True$
break
End
End

If $alreadySelected \equiv False$ **then**
$selectedSolutions :: tree$
$amountSelected := amountSelected + 1$
End

End

$selectedSolutions := Sort(selectedSolutions)$
Return $selectedSolutions[0],\ selectedSolutions[1]$

End

Where $RI()$ is a function which outputs a random integer between two given values, and $Sort()$ sorts a list of Node objects (the sorting mechanism is explained in the corresponding section).

Crossover

Data: $parent_1 \rightarrow$ Node object (Selected solution from tournament).
Data: $parent_2 \rightarrow$ Node object (Selected solution from tournament).

Function $Crossover(parent_1,\ parent_2)$:

$parentBranch_1,\ parentPath_1 := ChooseBranch(parent_1)$
$parentBranch_2,\ parentPath_2 := ChooseBranch(parent_2)$

$child_1 := ReplaceBranch(parent_1,\ parentPath_1,\ parentBranch_2)$
$child_2 := ReplaceBranch(parent_2,\ parentPath_2,\ parentBranch_1)$

Return $child_1,\ child_2$

End

Mutation

Data: $tree \rightarrow$ Node object (Produced from crossover of parents).
Data: $numberFeatures \rightarrow$ The number of input values.
Data: $maxDepth \rightarrow$ Maximum depth of trees allowed.

Function $Mutation(tree,\ numberFeatures,\ maxDepth)$:

$branch,\ path := ChooseBranch(parent_1)$
$remainingDepth := maxDepth - Length(path)$
$newBranch :=$ GenerateRandomTree$(numberFeatures,\ remainingDepth)$

$newTree := ReplaceBranch(tree,\ path,\ newBranch)$

Return $newTree$

End

Where $Length()$ returns the length of a list.

Choose Branch

Data: $tree \rightarrow$ Node object (Produced from crossover of parents).
Data: $path \rightarrow$ List of integers which are list indexes. Represents a path down a tree of Node objects.

Function $ChooseBranch(tree,\ path = [\,]) :$

$choice \quad := R(0, 1)$
$increment := 1/tree.size$

If $choice <= increment$ **then Return** $tree,\ path$ **End**

$count \quad := increment$
$numberBranches := Length(tree.branches)$

For $i = 0$ **to** $numberBranches$ **do:**

$count := tree.branches[i].size * increment$

If $choice <= count$ **then**
$path :: i$
Return $ChooseBranch(tree.branches[i], path)$
End

End

End

Where $Length()$ returns the length of a list.

Replace Branch

Data: $tree$ → Node object (Produced from crossover of parents).
Data: $path$ → List of integers which are list indexes. Represents a path down a tree of Node objects.
Data: $replacementBranch$ → List of integers which are list indexes. Represents a path down a tree of Node objects.

Function $ReplaceBranch(tree,\ path,\ replacementBranch)$:

If $path \equiv [\,]$ **then Return** $replacementBranch$ **End**

$direction := path[0]$
$path.pop(0)$

$tree.branches[direction] := ReplaceBranch(tree.branches[direction],\ path,\ replacementBranch)$

If $tree.branches[direction].depth >= tree.depth$ **then** $tree.depth := tree.branches[direction].depth + 1$ **End**

$tree.size := 0$
$numberBranches = Length(tree.branches)$

For $i = 0$ **to** $numberBranches$ **do:**
    $tree.size = tree.size + tree.branches[i].size$
End

Return $tree$

End

Where $Length()$ returns the length of a list.

Sorting in Generate New Population and Select

The $Sort()$ function utilised in the $GenerateNewPopulation()$ and $Select()$ functions, sorts Node objects (solutions) in ascending order using the following calculated value for each Node (solution):

$$\lambda\ x \rightarrow \begin{cases} x.fitness & \text{If x.depth <= maxDepth} \\ x.fitness \times 1.3^{x.depth - maxDepth} & \text{Otherwise} \end{cases}$$

The fitness is multiplies by this expression in order to reduce the growth of trees over the max depth, which is caused over time by the $Crossover()$ function, whilst still weighting the fitness obtained at a higher precedent.

Results and Analysis

The following section details how the optimum parameters for the algorithm were developed, and how the use of these parameters effect the overall fitness of produced models.

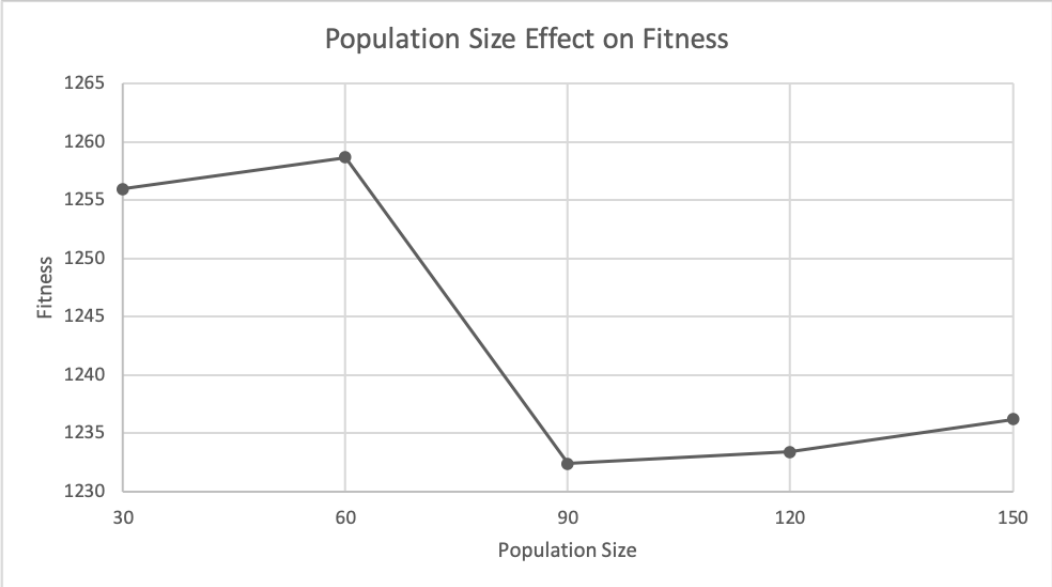
Finding Optimum Parameters

In order to decipher the optimal value for each hyperparameter, a set of the tests were run for each one. The hyperparameters of the algorithm are:

- *Population Size* (30) - The size of the population in each generation.
- *Maximum Depth* (5) - The maximum depth of each tree for a solution.
- *Mutation Probability* (0.5) - The probability an offspring will undergo mutation.
- *Elitism Proportion* (0.1) - How much of the original population to keep in each generation.
- *Tournament Size* (2) - The size of the tournament during selection.
- *Time Budget* (10) - How long the algorithm has to run.

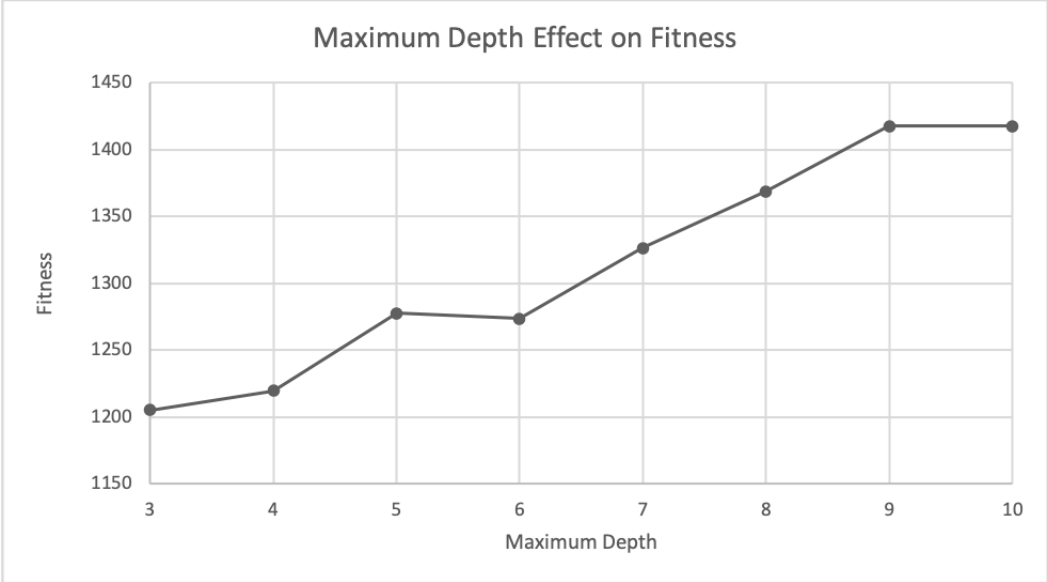
Each of these hyperparameters were tested independently using the same initial population, and using fixed constants for the other parameters (the value of these constants are shown in the brackets). Each parameter set was run 100 times, and the average fitness achieved over the 100 iterations was used as the final score to be plotted.

Population Size



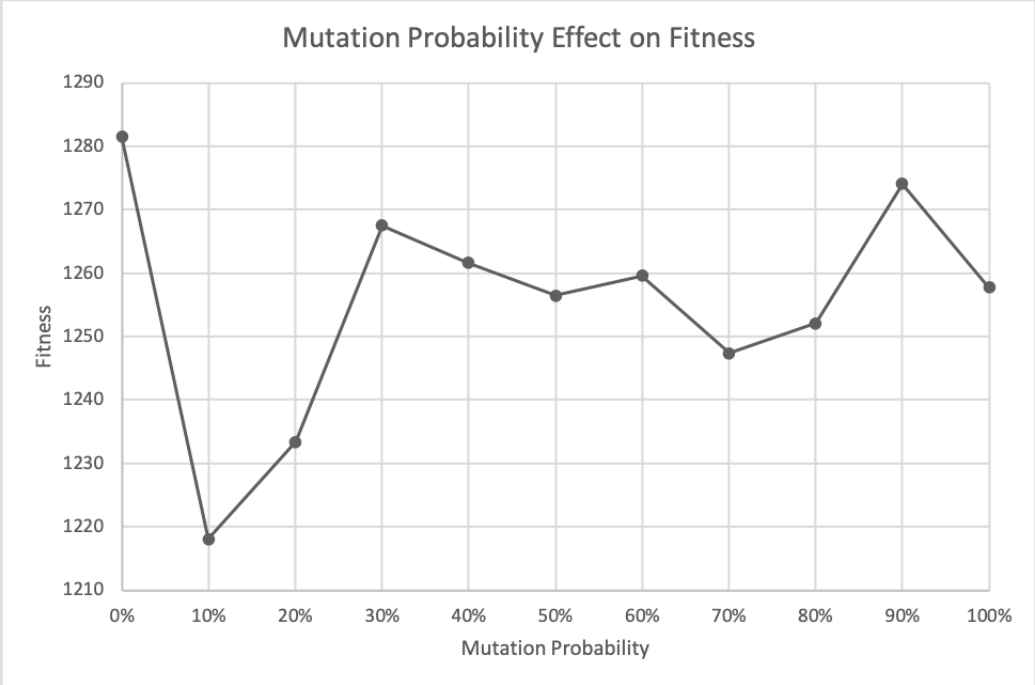
The results of this experiment show that the algorithm favours a higher population size, with the fitness undergoing a sharp drop between a population size of 60 to 90. This is probably due to the increase in diversity, allowing the algorithm to search more of the result space. However, the fitness obtained does begin to increase as the population size increased from 90. This trend is likely to continue, as a higher population size means that each generation will take longer to process, meaning that less generation can be run within the given time budget. The reduction in the amount of generations runs means that less variations operations are run, and thus the algorithms ability to exploit good solutions is reduced.

Maximum Depth



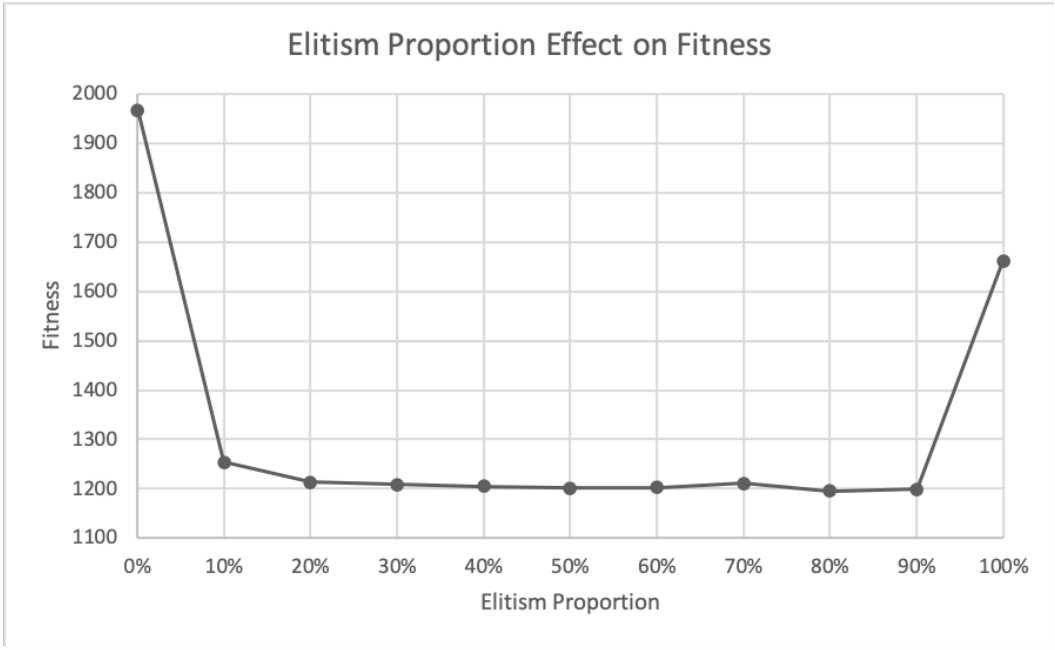
As shown in the graph, as the maximum depth of the trees increases, the fitness also increases. This trend is due to the fact that trees with a higher depth have a higher complexity. This complexity means that solutions can not accuratly fit the data.

Mutation Probability



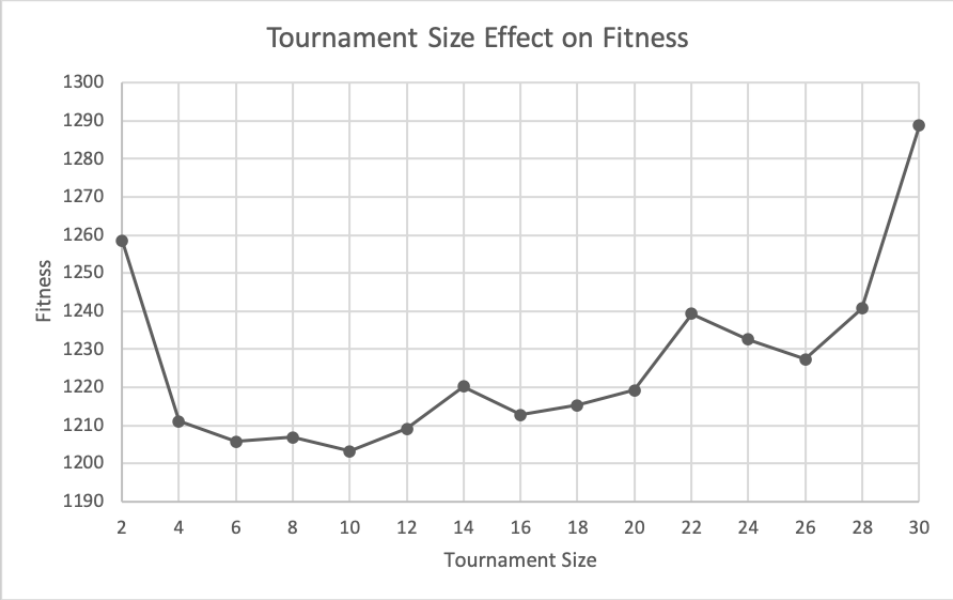
The trend in the graph shows that as the mutation probability increases, so to does the fitness. This indicates that the algorithm favours the ability to exploit already good solutions, over the ability to search the result space. However, the initial downwards spike between 0% and 10%, shows that a low mutation rate is still required, in order to escape local optimum.

Elitism Proportion



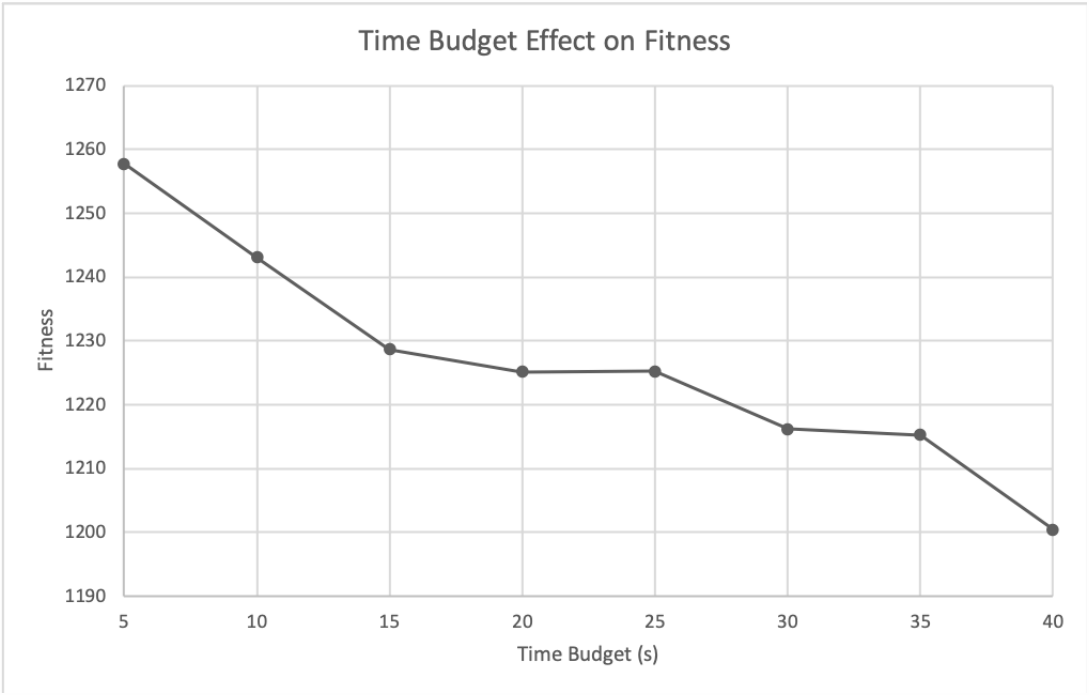
The results obtained from the elitism experiment, correspond with the indication given by the mutation results, that the algorithm favours exploitation over exploration. However, just like in mutation, some exploration is necessary, hence the spike between 90% and 100%.

Tournament Size

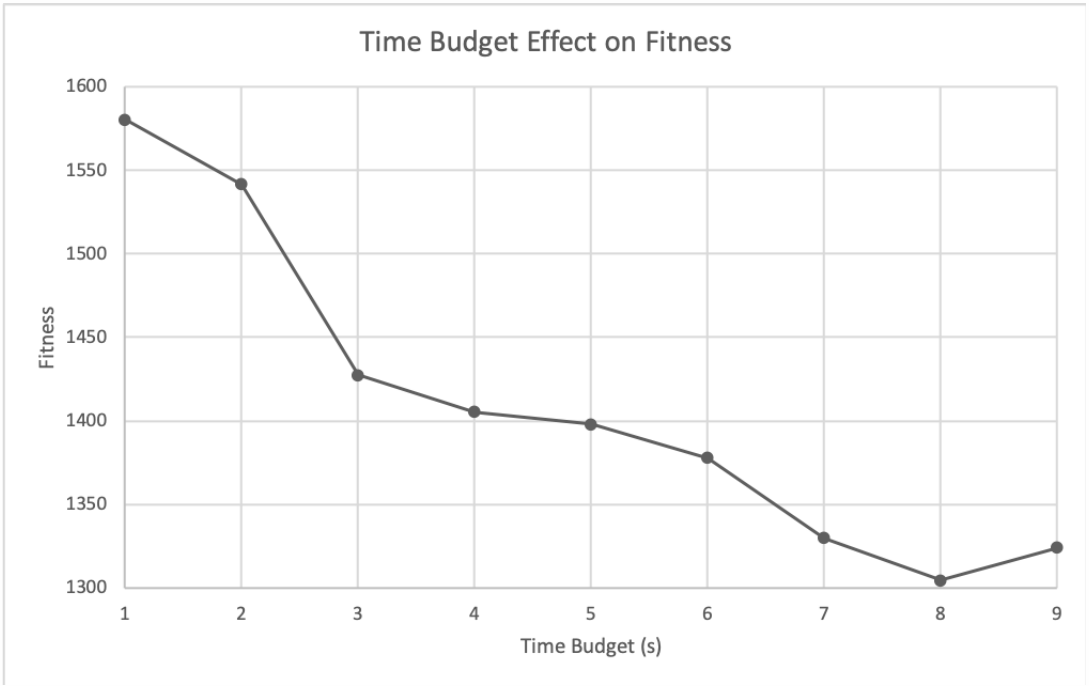


The trend in this graph shows that a lower tournament size is favoured by the algorithm. On inspection this is due to the fact that a higher tournament size causes one good solution to dominate. This results in a convergence on a local optimum, reducing the ability for exploration.

Time Budget



As shown by the trend in the graph, as the time budget increases, the fitness decreases. The reason for this relationship is fairly obvious, which is that the longer the algorithm has to run, the more time it has to develop a better solution. On closer inspection however, most of the improvement happens within the first 10 seconds of the algorithms run (shown below using a new initial population). Indicating that the algorithm converges on an optimum within that time frame.



Testing Parameter Settings

The following section shows the results of different parameter settings. Each experiment was run 100 times, with a time budget of 10 seconds. The time budget for 10 seconds was selected due to the results gathered in the previous section, showing that most of the improvement happened within this time period.

Maximising Exploitation

The first experiment ran used hyperparameter values to maximise exploitation. This meant using a low population and mutation rate, whilst having a relativly high tournament size and elitism. This is so any good solutions found would have a higher chance of being selected for repoduction. The hyperparameter values used are shown below, and the results obtained by these parameters are labelled in the boxplot as "*Exploit*".

- *Population Size*: 30
- *Maximum Depth*: 4
- *Mutation Probability*: 0.1 (10%)
- *Elitism Proportion*: 0.8 (80%)
- *Tournament Size*: 10

Maximising Exploration

The second experiment utlised hyperparameter values which maximised exploration. This therefore meant setting the elite proportion and tournament size to relativly low values, whilst doing the opposite for the mutation probablity and population size. The high population size means an increase in diversity, and a high mutation rate means many of the offspring will undergo random changes. Thus by using high values for both these hyperparameters (whilst keeping elitism and tournament size low) more of the result space can be searched. The hyperparameter values used are shown below, and the results obtained by these parameters are labelled in the boxplot as "*Explore*".

- *Population Size*: 150
- *Maximum Depth*: 4
- *Mutation Probability*: 0.8 (80%)
- *Elitism Proportion*: 0.1 (10%)
- *Tournament Size*: 2

Optimal Parameters

The final experiment used hyperparameter values which were found to be optimal in the previous section of this report. The only parameter which was changed from the optimal is the mutation probability. This is because the reduction in fitness caused by the increase in mutation probability is mitigated once combined with a high elitism, thus allowing the algorithm to explore more of the result space. The hyperparameter values used are shown below, and the results obtained by these parameters are labelled in the boxplot as "*Optimised*".

- *Population Size*: 90
- *Maximum Depth*: 4
- *Mutation Probability*: 0.3 (30%)
- *Elitism Proportion*: 0.3 (30%)
- *Tournament Size*: 8

Box Plot of Results

As the graph shows, the parameter values which obtained the best result were the exploitation parameters. However, this came at the cost of a high variance, which is caused by the parameter values preventing the algorithm from escaping local optima.

Conversly, the exploration parameters did not obtain results that were as good as the exploitation parameters. Both the median value and the lowest value of the exploration parameters are higher than the exploitations. But this loss in results caim at the gain of less variation. This is shown clearly as both the third quartile and highest value obtained by the exploration parameters are drastically lower than those obtained by the exploitation parameters. This is due to the parameters having the opposite effect on algorithms behaviour, allowing it to escape local optima, at the cost of not being able to exploit good solutions as effectivly.

The optimised parameter values solves the problems with both the previous parameter settings. Although it does not obtain a lowest value which beats the one obtained by the exploitation parameter settings (albeit the difference between the lowest values is only 8), both the lower quartile and median value beats both of the previously discussed parameter settings. Thus indicating that this set of hyperparameter values achieves better results a majority of the time. The results also show the parameter values reduce variance even further than the reduction obtained by the exploration parameters. The reason for the overall improvement in results is that the parameter values allow the algorithm to escape local optima, whilst also striking a balance with its ability to exploit when a good solution is found.