**The Effect of Different Genetic Algorithm Types on Processing Speed**

**Abstract**

Genetic algorithms are algorithms that are inspired by the biological process of natural selection and belong to a larger class of algorithms called evolutionary algorithms (Carr, 2014). The genetic algorithms used in this project rely on changing the crossover types, which means that the algorithm changes the way that the matrix randomly mutates during the process of finding the ideal solution (Umbarkar and Sheth, 2015). The purpose of this experiment was to determine which genetic algorithm would sort a dataset in numerical order the most quickly. The independent variable (IV) in this experiment was the different types of the genetic algorithm used to sort the data in numerical order. The dependent variable was the time spent on the algorithms to sort through the dataset. The four levels of the IV were single-point crossover, two-point crossover, uniform crossover, and flat crossover. It was hypothesized that the single point crossover algorithm would sort the dataset the most quickly. First, the materials (algorithms, dataset, and code to run the algorithms) were obtained from Github user “dawidkopczyk.” Then, the algorithms were run with the modified code, and the data was collected and analyzed. The experiment showed that single-point crossover was the fastest, then two-point, uniform, and finally flat crossover. The hypothesis was proven correct due to the fact that single-point crossover requires the least amount of changes in-between steps, which is less work for the computer. Many papers support the results found in this experiment such as one written by Alden H. Wright. For further research, the sizes of the dataset could be changed. An improvement that could be made would be to use a more diverse dataset. This could lead to a more balanced outcome.

**The Effect of Different Genetic Algorithm Types on Processing Speed**

**Introduction**

Genetic algorithms are algorithms that are inspired by the biological process of natural selection and belong to a larger class of algorithms called evolutionary algorithms (Carr, 2014). They are often used to generate high-quality solutions to complex optimization**,** which is the process of finding the ideal solution, and search algorithms. Genetic algorithms rely on complex biology-inspired operators such as mutation, crossover, and selection (Rowland, 2019). This project relies on changing the crossover types, which means that the algorithm changes the way that the matrix randomly mutates during the process of finding the ideal solution. This means that the way that the corner at which the mutation occurs leads to different algorithms as well (Umbarkar and Sheth, 2015).

Often times, the processing time of an algorithm is extremely important. It is important to have the most efficient processing speed because people deal with large amounts of data on a daily basis. The processing speed of data is oftentimes correlated with the efficiency of an algorithm. The more efficient an algorithm and the fewer steps, in most cases, the quicker the computer can compute the results, which would leave the computer with more time to do other tasks (Roberts, 2016). Also, if certain results are time-sensitive, the processing speed for the data is important as well because that means that the user could get the data more quickly. This can be used in situations such as stock trading or trying to figure out different combinations for different medical issues by attempting to see how each issue would progress and what it would ultimately lead to.

The processing speed of genetic algorithms is especially important because the point of genetic algorithms is to find the most efficient solution, and the quicker the algorithm can do it, the better it is for the computer and the results (Kopczyk, 2018). This would mean that the problem can be solved in a shorter time, which becomes more important when dealing with larger datasets. Because most algorithms depend on the size of the dataset and have to go through every single piece of data, the amount of repetitions of the dataset that the algorithm has to go through is important largely because it determines the speed of the algorithm. If it has to go through the dataset multiple times, then the algorithm is slowed down.

This can be used to help determine which algorithm is the most suitable to use in the real world. If this project can determine which algorithm is best used to sort the data, then the most efficient algorithm can be used to sort through the data quickly and save time for the results. This will give a better understanding of how different types of crossovers in genetic algorithms can influence the number of steps and time taken to run.

In this experiment, the independent variable was the different types of genetic algorithm used to sort the data in numerical order. This is a common metric used in computer science to determine the speed of an algorithm. The dependent variable was the time that it took for the algorithm to sort through the data. The constants were the type of computer used, the processing system used, the package that was used to import the type of data, the data that was being organized, and the code used to run the algorithms. There was no control in this experiment because each algorithm was being compared against the other, and they were all sorting the dataset. Thus, there was no “baseline.” This project tried to determine which genetic algorithm sorted a dataset in increasing numerical order the most quickly. It was hypothesized that if the single point crossover algorithm was used to sort the data, then it would be quicker than the other genetic algorithms. This is because there were fewer steps that had to be undertaken for the algorithm to fully sort the data. A single point crossover, as the name suggests, had only one point in which the genes were exchanged and passed to the child. However, all of the other algorithms had multiple steps in the crossover process.

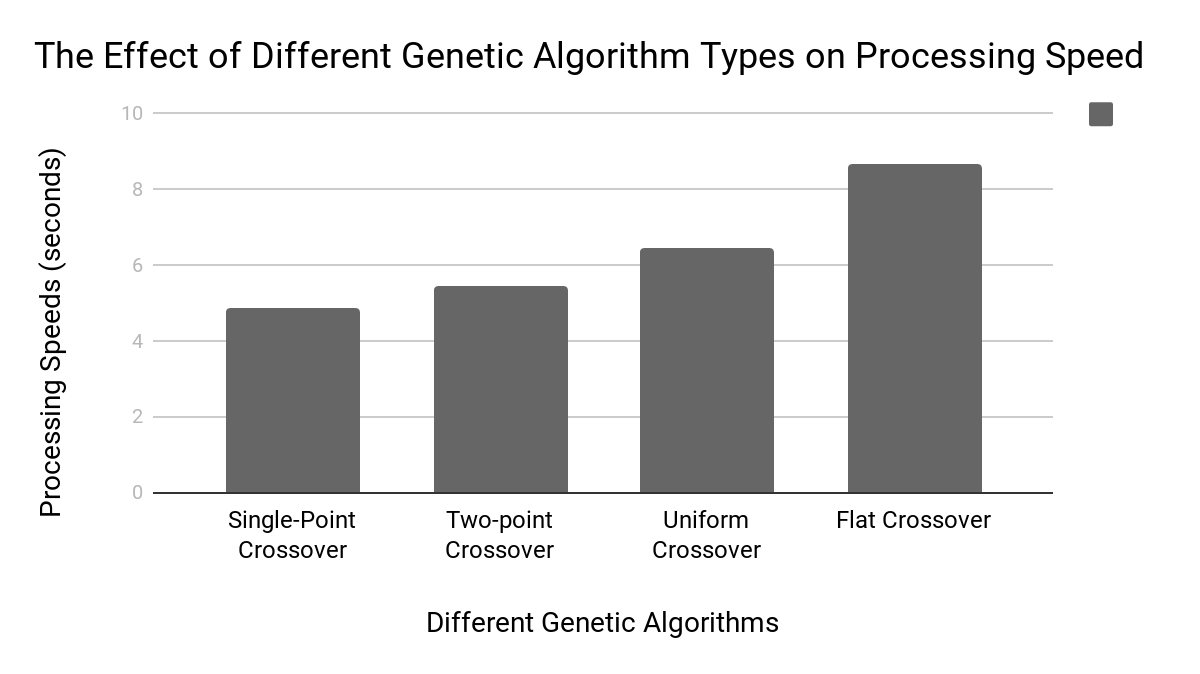
**Methods and Materials**

The dataset was acquired from the GitHub user “dawidkopczyk,” and the package that was used to import the sorting algorithms was as well. First, the dataset (a constant) was downloaded from GitHub. Then, the code that was written with changes made to accommodate each different crossover algorithm was run. This code was downloaded from the GitHub user “dawidkopczyk” and modified by the experimenter to fit the experiment. The code was run on an ASUS computer (a constant) with Windows processing system (a constant). The time of how long it took for each algorithm to run was recorded using a timer (a constant) and along with the number of steps (a constant). Afterward, the data collected from this experiment were analyzed, and the most efficient algorithm was determined.

The independent variable had 4 different levels with no control.There was no control in this experiment because none of the independent variables could be manipulated to become other independent variables. Thus, none of the levels of the independent variables could be used as a basis for comparison. The first level was a single-point crossover, which means that there was a single place where the crossover occurred between each repetition. The next level was a two-point crossover, which means that there were two places that the crossover, or the mutation, occurred between each repetition.The third level was a flat crossover, where the algorithm created a new result from two already existing results, but the number of points switched between repetitions was randomized. The final level was a uniform crossover, where each gene had a 50% chance of being switched. The dependent variable was the time that the algorithm took to sort the dataset. Each of the algorithms were repeatedly run on the same dataset 10 times to get 10 trials. There was not any need for safety measures to be taken on this experiment because there was nothing that posed a threat to the safety of the experimenter.

**Results**

The effect of different genetic algorithm types on processing speed is displayed in the graph and table below. The results of this experiment were measured in seconds. Mean was chosen as the measure of central tendency because there were no outliers in the data. The first level of the independent variable, single-point crossover, took an average of about 4.835 seconds to sort the data set and had a range of about 0.24 seconds. Its standard deviation was approximately 0.071 seconds. This was the quickest method. Two-point crossover took the second shortest time with an average of about 5.428 seconds and a range of ‭approximately 0.408‬ seconds. It had a standard deviation of about 0.118 seconds. Uniform crossover had an average of approximately 6.463 seconds with a range of ‭about 1.161 seconds and was slower than two-point crossover, making it the second slowest method. The standard deviation of this method was about 0.356 seconds. Finally, flat crossover was the slowest with an average of about 8.649 seconds and an approximate range of ‭1.675 seconds. It had a standard deviation of approximately 0.462 seconds. The results from this experiment support the hypothesis that the single-point crossover method would most quickly sort the dataset.



**Figure 1: Averages for the Effect of Different Genetic Algorithm Types on Processing Speed**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Descriptive**  **Statistics** | **Genetic Algorithms (seconds)** | | | |
|  | **Single-Point** | **Two-Point** | **Uniform** | **Flat** |
| **Mean** | **4.835** | **5.428** | **6.463** | **8.649** |
| **Range** | **0.24** | **0.408** | **1.161** | **1.675** |
| **Minimum** | **4.76** | **5.311** | **6.201** | **7.85** |
| **Maximum** | **5.001** | **5.719** | **7.362** | **9.525** |
| **Variance** | **0.005** | **0.0139** | **0.127** | **0.213** |
| **Standard Deviation** | **0.071** | **0.118** | **0.356** | **0.462** |
| **Number of Trials** | **10** | **10** | **10** | **10** |

**Table 1: The Effect of Different Genetic Algorithm Types on Processing Speed (seconds)**

**Conclusion**

The purpose of this experiment was to determine the effect of different genetic algorithms on the processing speed of a computer and which genetic algorithm would numerically sort a dataset the most quickly. It was hypothesized that if the single point crossover algorithm was used, it would sort the dataset in numerical order the most quickly.

This experiment was able to conclude that there is a difference between different genetic algorithms and that the fastest algorithm was a single-point crossover, which took approximately 4.835 seconds to run every time. The second fastest algorithm was a two-point crossover which took about 5.428 seconds per run. The third fastest algorithm was a uniform crossover with an average run time of 6.463 seconds. The flat crossover algorithm was the slowest, with a time of 8.649 seconds.

The goal of the genetic algorithms was to sort a dataset in increasing numerical order in the shortest amount of time. Due to this, the algorithms that required the least amount of crossovers took the shortest time to sort the dataset. Single-point crossover was the fastest out of the four genetic algorithms because it involved the least amount of data switched to the other gene. This is less work for the computer and therefore increases the processing speed. Two-point crossover has two locations on the gene where data is exchanged, therefore, it was slower than single-point. Uniform crossover took longer than two-point crossover because every single point on the gene has a 50% chance of being switched. Flat crossover took the longest amount of time to sort the dataset because the number of crossovers is randomized, which means that there is a possibility of a very high number of crossovers, which lengthens the process considerably.

This experiment did support the hypothesis of this paper because a single crossover was predicted to be the fastest and the experiments proved it.

Many papers supported the results found in this experiment, like Alden H. Wright’s paper. He stated that “the two crossover methods outperformed the binary-coded genetic algorithm,” which could be applied to this finding because the two-point crossovers (single and two) were considered to be less binary-coded than the uniform and flat crossovers (Wright, 2014). Robertson’s paper stated that “the execution time of his implementation was independent of the number of classifiers” when talking about different classifiers such as replacement and mating which means that basically, the only thing that affected the timing was the type of crossover performed (Robertson, 1987). Additionally, a paper stated that “the single-crossover has the best results,” which agreed with the hypothesis here (Magahaes-Mendes, 2013).

For further study and improvements to the experiment, the sizes of the dataset could be changed. The project could be improved by running a larger dataset or a smaller dataset with the genetic algorithm. This would help determine which genetic algorithms would be most suitable for datasets of different sizes. Additionally, the idea of working with a dataset where the size is unknown to the algorithm may also be explored in further research. This would help determine how each genetic algorithm deals with unknown data set sizes. Furthermore, another way the project could be improved and changed would be running the genetic algorithms on a different computer. The computer that ran the genetic algorithms was not the quickest nor powerful by any means, so changing the computer that the algorithms were run on could change the results. An additional improvement to this experiment could be to use a more diverse dataset. This could result in a more balanced outcome.

Literature Cited

A.J., Umbarkar, and Sheth P.D. “CROSSOVER OPERATORS IN GENETIC ALGORITHMS: A

REVIEW.” ICTACT Journal on Soft Computing, vol. 06, no. 01, 1 Oct. 2015, pp. 1083–1092, pdfs.semanticscholar.org/6888/7a8e96440f9e69c015c923a83455478f0290.pdf, 10.21917/ijsc.2015.0150. Accessed 24 Oct. 2019.

Carr, Jenna. "An Introduction to Genetic Algorithms." Whitman College, 16 May

2014, www.whitman.edu/Documents/Academics/Mathematics/2014/carrjk.pdf.

Accessed 22 Oct. 2019.

Kopczyk, Dawid. “Genetic Algorithm in Machine Learning.” *Dawid Kopczyk*, 12 Apr. 2019,

dkopczyk.quantee.co.uk/genetic-algorithm/.

Magalhães-Mendes , Jorge. “A Comparative Study of Crossover Operators for Genetic ……….Algorithms to Solve the Job Shop Scheduling Problem .” ……….*Https://Pdfs.semanticscholar.org/*, Apr. 2013, A Comparative Study of Crossover ……….Operators for Genetic Algorithms to Solve the Job Shop Scheduling Problem.

Roberts, Eric. “Algorithmic Efficiency.” *CS 54N*, 17 Oct. 2016,

web.stanford.edu/class/cs54n/handouts/11-AlgorithmicEfficiency.pdf.

Robertson, and G.g. “Parallel Implementation of Genetic Algorithms in a Classifier System.” ………*AGRIS*, Hillsdale, N.J. : L. Erlhaum Associates, 1987., 1 Jan. 1987, ………agris.fao.org/agris-search/search.do?recordID=US201301782168.

Rowland, Todd. “Genetic Algorithm.” *From Wolfram MathWorld*,

mathworld.wolfram.com/GeneticAlgorithm.html.

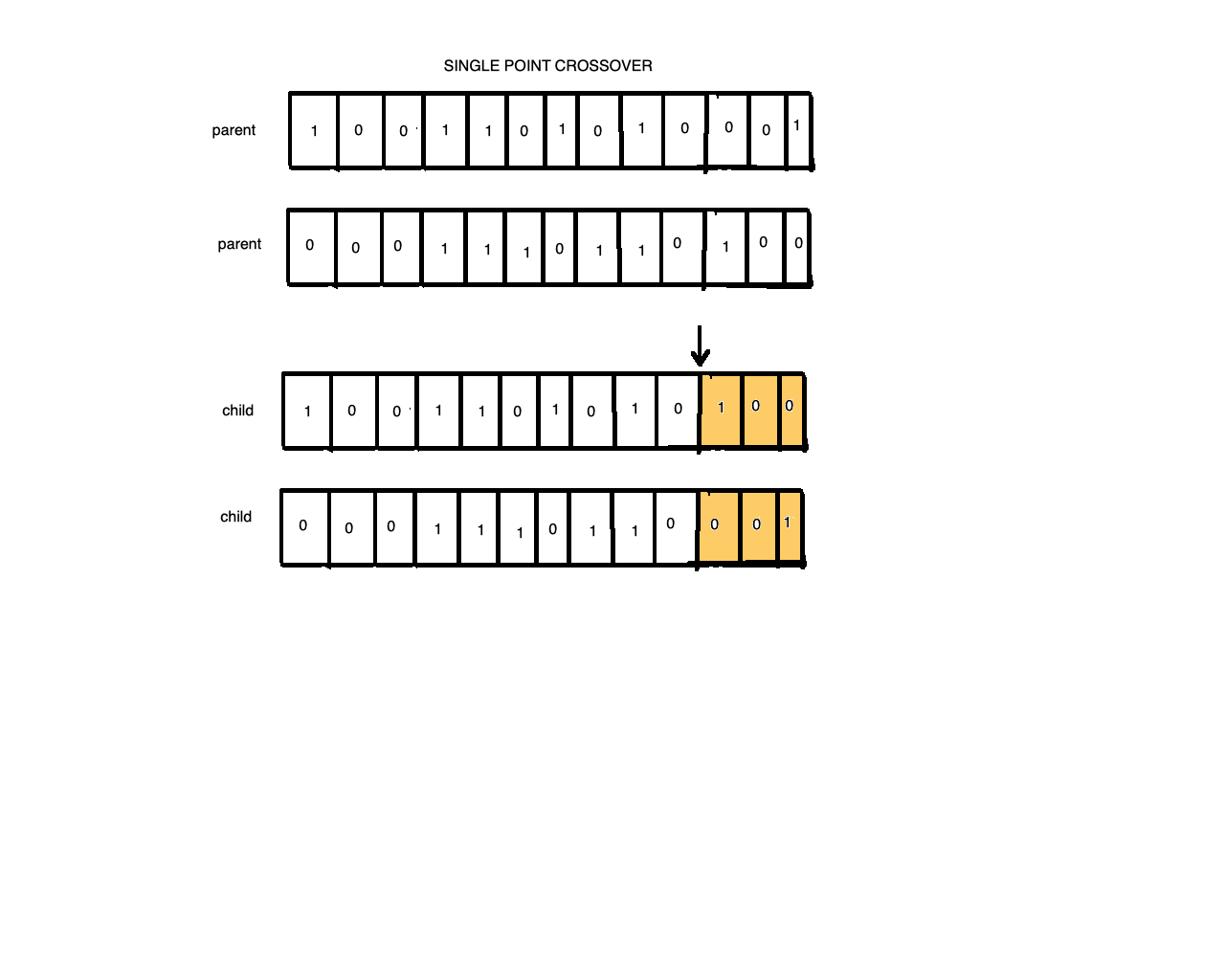
Wright, Alden H. “Genetic Algorithms for Real Parameter Optimization.” *Foundations of* ………*Genetic Algorithms*, Elsevier, 27 June 2014, ………[www.sciencedirect.com/science/article/pii/B9780080506845500161](http://www.sciencedirect.com/science/article/pii/B9780080506845500161).

**Appendix**

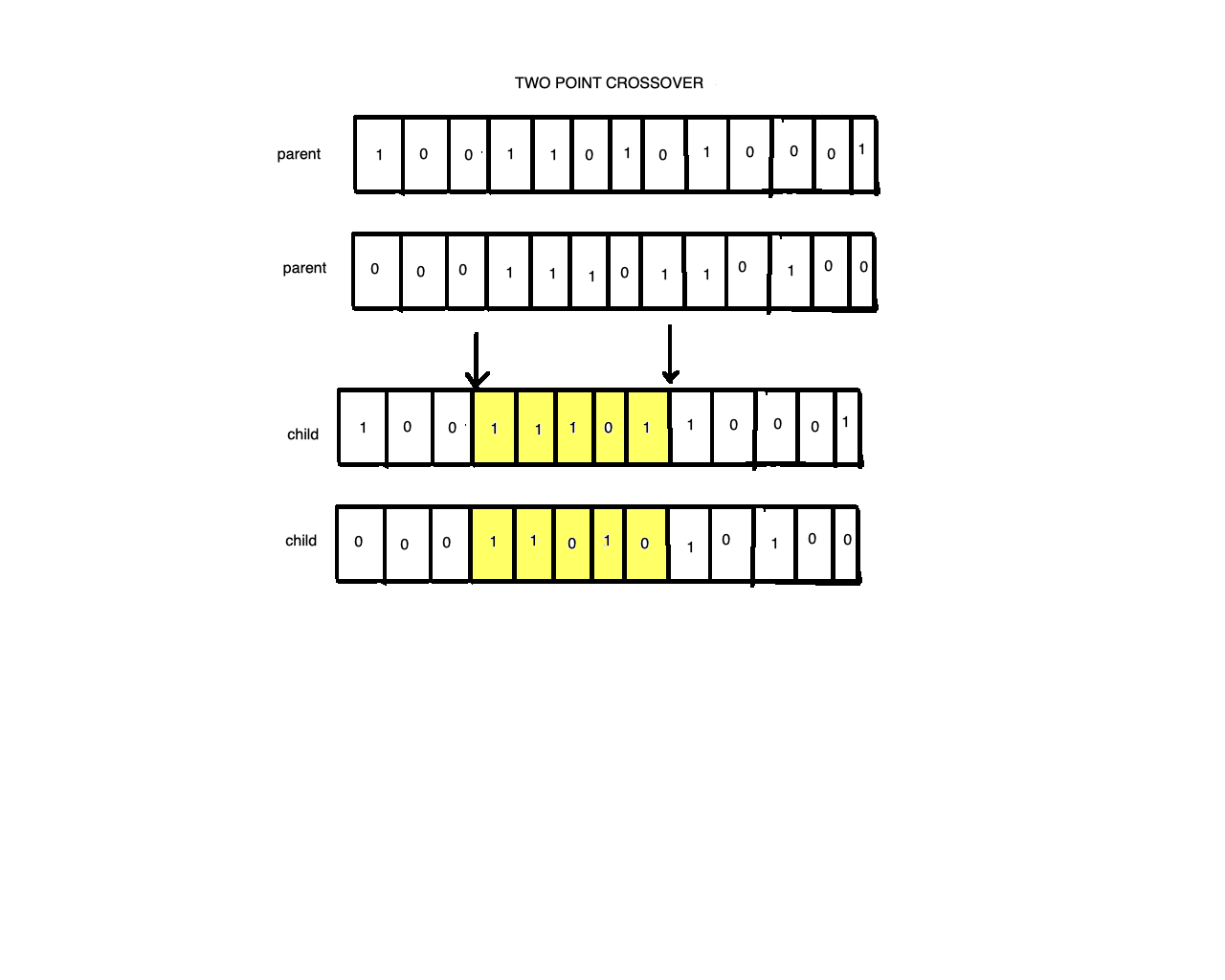
|  |  |
| --- | --- |
| **Different Genetic Algorithm Types** | **Trials: Processing Speed (seconds)** |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Trial 1** | **Trial 2** | **Trial 3** | **Trial 4** | **Trial 5** | **Trial 6** | **Trial 7** | **Trial 8** | **Trial 9** | **Trial 10** |
| **Single-Point Crossover** | 5.001 | 4.827 | 4.789 | 4.760 | 4.906 | 4.799 | 4.889 | 4.787 | 4.782 | 4.808 |
| **Two-Point Crossover** | 5.522 | 5.719 | 5.361 | 5.345 | 5.370 | 5.515 | 5.352 | 5.371 | 5.311 | 5.416 |
| **Uniform Crossover** | 6.880 | 6.349 | 7.362 | 6.318 | 6.204 | 6.495 | 6.300 | 6.316 | 6.207 | 6.201 |
| **Flat Crossover** | 8.429 | 8.800 | 8.437 | 8.387 | 8.396 | 7.850 | 9.128 | 9.102 | 8.431 | 9.525 |

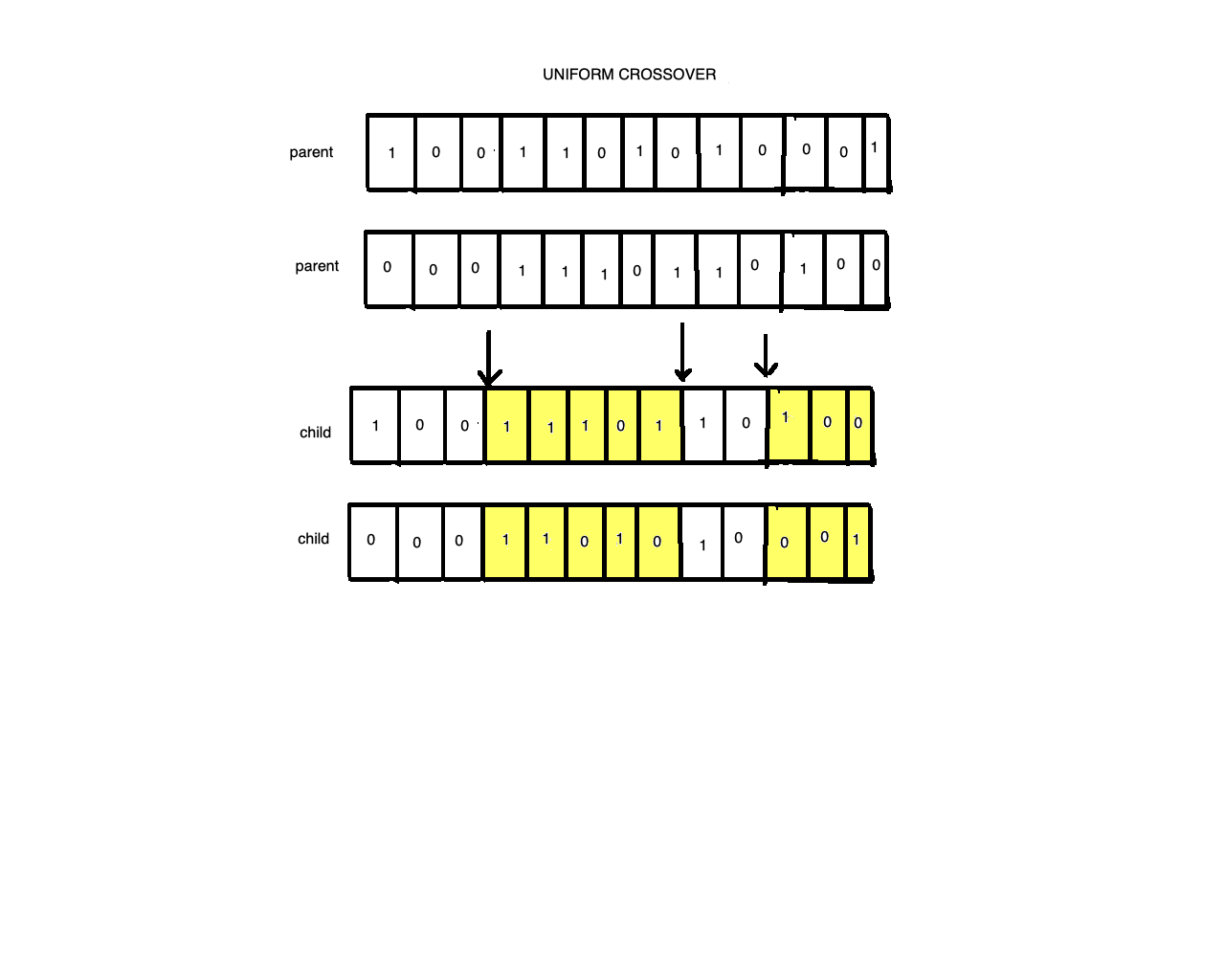
**Table 2: Individual Trials for the Effect of Different Genetic Algorithm Types on Processing Speed**



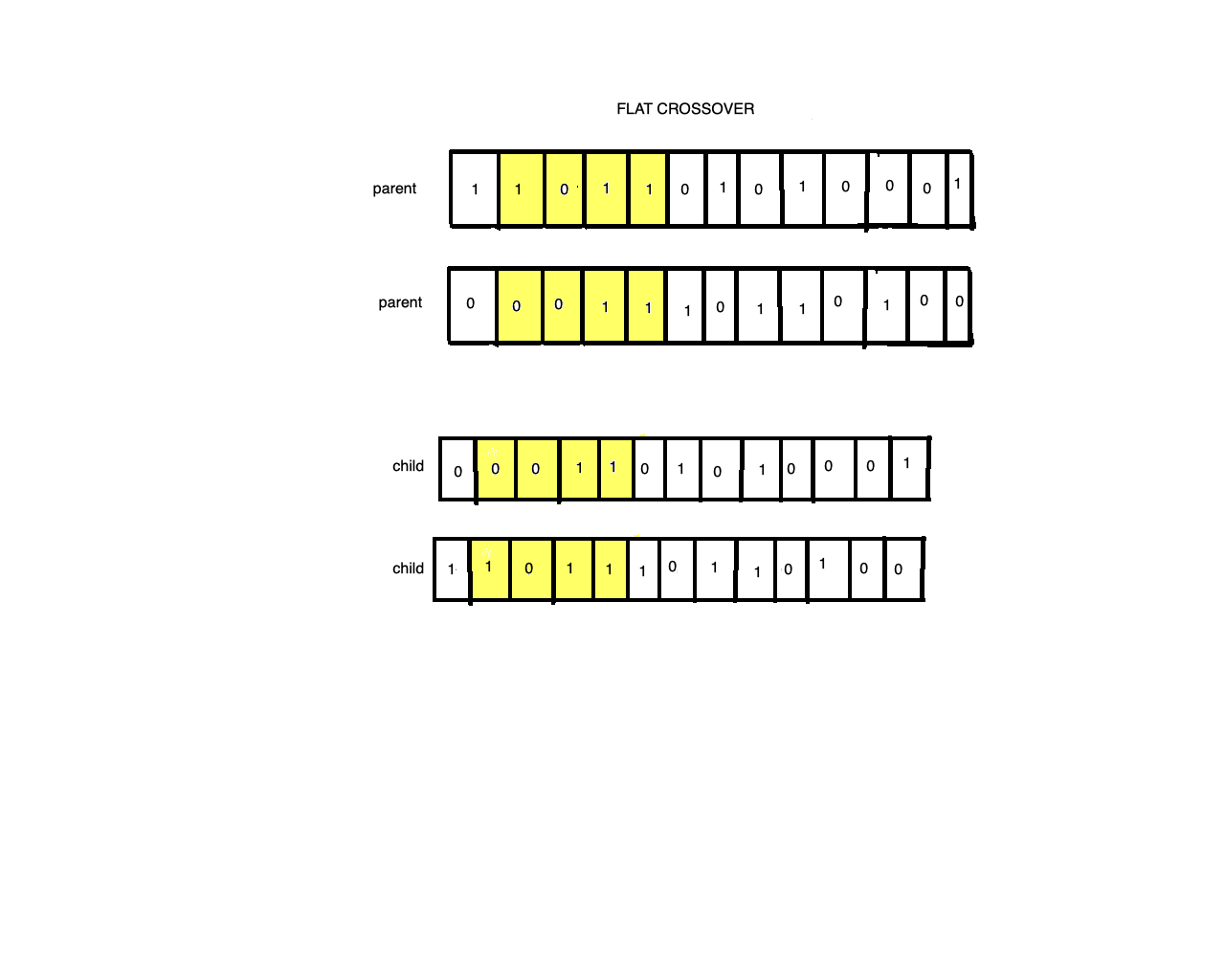
**Figure 2: Single Point Crossover**



**Figure 3: Two Point Crossover**



**Figure 4: Uniform Crossover**



**Figure 5: Flat Crossover**

**Code Used in Experiment**

**Highlighted sections indicate changes made by experimenter**

SEED = 2018

random.seed(SEED)

np.random.seed(SEED)

#==============================================================================

# Data

#==============================================================================

dataset = load\_boston()

X, y = dataset.data, dataset.target

features = dataset.feature\_names

#==============================================================================

# CV MSE before feature selection

#==============================================================================

est = LinearRegression()

score = -1.0 \* cross\_val\_score(est, X, y, cv=5, scoring="neg\_mean\_squared\_error")

print("CV MSE before feature selection: {:.2f}".format(np.mean(score)))

#==============================================================================

# Class performing feature selection with genetic algorithm

#==============================================================================

class GeneticSelector():

def \_\_init\_\_(self, estimator, n\_gen, size, n\_best, n\_rand,

n\_children, mutation\_rate):

#these are all changed to create a different algorithm every time

# Estimator

self.estimator = estimator

# Number of generations

self.n\_gen = n\_gen

# Number of chromosomes in population

self.size = size

# Number of best chromosomes to select

self.n\_best = n\_best

# Number of random chromosomes to select

self.n\_rand = n\_rand

# Number of children created during crossover

self.n\_children = n\_children

# Probablity of chromosome mutation

self.mutation\_rate = mutation\_rate

if int((self.n\_best + self.n\_rand) / 2) \* self.n\_children != self.size:

raise ValueError("The population size is not stable.")

def initilize(self):

population = []

for i in range(self.size):

chromosome = np.ones(self.n\_features, dtype=np.bool)

mask = np.random.rand(len(chromosome)) < 0.3

chromosome[mask] = False

population.append(chromosome)

return population

def fitness(self, population):

X, y = self.dataset

scores = []

for chromosome in population:

score = -1.0 \* np.mean(cross\_val\_score(self.estimator, X[:,chromosome], y,

cv=5,

scoring="neg\_mean\_squared\_error"))

scores.append(score)

scores, population = np.array(scores), np.array(population)

inds = np.argsort(scores)

return list(scores[inds]), list(population[inds,:])

def select(self, population\_sorted):

population\_next = []

for i in range(self.n\_best):

population\_next.append(population\_sorted[i])

for i in range(self.n\_rand):

population\_next.append(random.choice(population\_sorted))

random.shuffle(population\_next)

return population\_next

#the crossover number is changed in order for the different crossovers to happen

def crossover(self, population):

population\_next = []

for i in range(int(len(population)/2)):

for j in range(self.n\_children):

chromosome1, chromosome2 = population[i], population[len(population)-1-i]

child = chromosome1

mask = np.random.rand(len(child)) > 0.5

child[mask] = chromosome2[mask]

population\_next.append(child)

return population\_next

def mutate(self, population):

population\_next = []

for i in range(len(population)):

chromosome = population[i]

if random.random() < self.mutation\_rate:

mask = np.random.rand(len(chromosome)) < 0.05

chromosome[mask] = False

population\_next.append(chromosome)

return population\_next

def generate(self, population):

# Selection, crossover and mutation

scores\_sorted, population\_sorted = self.fitness(population)

population = self.select(population\_sorted)

population = self.crossover(population)

population = self.mutate(population)

# History

self.chromosomes\_best.append(population\_sorted[0])

self.scores\_best.append(scores\_sorted[0])

self.scores\_avg.append(np.mean(scores\_sorted))

return population

def fit(self, X, y):

self.chromosomes\_best = []

self.scores\_best, self.scores\_avg = [], []

self.dataset = X, y

self.n\_features = X.shape[1]

population = self.initilize()

for i in range(self.n\_gen):

population = self.generate(population)

return self

@property

def support\_(self):

return self.chromosomes\_best[-1]

def plot\_scores(self):

plt.plot(self.scores\_best, label='Best')

plt.plot(self.scores\_avg, label='Average')

plt.legend()

plt.ylabel('Scores')

plt.xlabel('Generation')

plt.show()

sel = GeneticSelector(estimator=LinearRegression(),

n\_gen=7, size=200, n\_best=40, n\_rand=40,

n\_children=5, mutation\_rate=0.05)

sel.fit(X, y)

sel.plot\_scores()

score = -1.0 \* cross\_val\_score(est, X[:,sel.support\_], y, cv=5, scoring="neg\_mean\_squared\_error")

print("CV MSE after feature selection: {:.2f}".format(np.mean(score)))