Homework #2

Micah Runner

February 12, 2021

Due: Feb 12, 2021 6pm MST.

Problem 1

1.1 Statement

Knapsack problems are popular examples of discrete optimization problems which arise in a number of application domains. The goal is to maximize the value of the collection of items placed into a container, the knapsack, which is subject to a weight constraint. The most common example is the 0-1 knapsack problem:

$$\text{maximize} \sum_{i=1}^{n} v_i x_i$$
$$\text{subject to} \sum_{i=1}^{n} w_i x_i \leq W \quad \text{and} \quad x_i \in \{0, 1\}$$

where v_i represents the value of item i, w_i is the weight of the item, W is the maximum weight capacity and x_i represents whether the item is present or not. Traditionally it is addressed via dynamic programming or branch and bound techniques.

For this homework, you will write an evolutionary algorithm to find candidate solutions (maximal and valid item lists).

Instructors Hints

You may initialize v and w as positive integer arrays and set W to be a positive number. Note that W should be larger than the smallest w_i but not larger than $\sum w_i$ (to avoid no solution or a trivial solution). For the solution of the problem, we are searching for the vector x which represents which items we load into the knapsack.

The occurrence string x is ready to go as a (genetic algorithm) chromosome. The challenge is how to implement the genetic operators: mutation and recombination. Options for mutation are random "bit flips" or element swaps in x. Recombination can be done in the traditional manner.

Both can be destructive. Application of the genetic operators on valid x vectors (occurrence strings) can produce new x vectors which are not valid (violate the weight constraint). You need to either remove or penalize invalid occurrence strings. You need to find a good fitness function which is easy to evaluate, rewards valid high value strings and penalizes invalid strings.

Can you think of a way to test your code? Can you come up with a collection of weights and values for which you know the solution?

1.2 Method

Evolutionary Algorithm and 0-1 Knapsack Problem

The 0-1 knapsack problem was solved using an evolutionary algorithm that used the combination of elitism and roulette selection methods, along with random bit mutations to achieve the exact or closet solution to the knapsack problem. This approach simulates nature in that reproduction tends towards producing the most adept creature for its particular problems related to its survival. This algorithm will only work for any problem that requires the use of comparing weights and values given that selected values and weights are chosen given a 0 or 1. This algorithm will not work at all or as well for other more complicated items. We wish to make artificial life of self-replicating robots that consume the universe but that is still very out of reach, so don't worry about this algorithm consuming the universe...

Yet. Or our world will be consumed by some other artificial life that was built by a more advance and ancient civilization.

0-1 Knapsack Problem

The 0-1 knapsack problem has a few rules to follow that are reality simple to implement. First, there is a list of items, which have a value and weight assigned to each item. Second, there is a knapsack, backpack, or some other object that can only hold a maximum weight limit. Third, the items will be selected using 1 within a selection list, or given a selection value of 0 to represent that the item is not selected. Fourth, the list of selected items must maximize the item values within the given weight limit.

As with most problems, P turns out to be NP, we can solve this problem using **Dynamic Programming**, which eats up all your memory to save computations previously done in a different iteration. We borrowed a mid-range algorithm from GeeksforGeeks by Bhavya Jain, to find the exact solution to compare to our evolutionary algorithm. Bhavya's algorithm crunches through all the possible combinations in a tree like structure, but using an array instead of pointers. This speeds up time compared to brute-force method, which would try every combination and select the best. This is the reason why Bhavya's algorithm was selected as it is a pretty fast algorithm, which means the EA (evolutionary algorithm) must be faster, use less memory, and get within a percent error of the actual algorithm.

Evolutionary Algorithm

As stated before the evolutionary algorithm is pretty simple as it involves creating a fitness function that allows the problem to be elevated and therefor quantifiable. EA also needs the population members that can be reproduce with each other and allow for those individuals to get cancer... mutate. Each generation will have a set population size, which will all be evaluated using the fitness function for reproduction. We will first talk about the choice of the fitness function for the 0-1 knapsack problem.

Fitness

The fitness function is defined as assigning a 0 or a 1 to an array that holds which items are selected. This of course is boring if there is not a weight limit, so an added touch is assigning a fitness value of 0 if the total weight of selected items is over the weight limit. This fitness allows the reproduction of the algorithm to push overweight knapsacks for eating too many items, while rewarding all of the high value knapsacks for staying under the weight limit.

Selection

The selection choice for the algorithm uses to different methods for each parent. One of the parents uses elitism, which is selecting the cream of the crop or the best of the best. Elitism selection takes a set amount from the population that has the greatest fitness value, for our algorithm we only took the best out of each generation and assigned it to a parent. The second parent was selected using roulette selection, which sums up all of the fitness values of the total population and then creates a percentage for each individual fitness value. The next step is used to add the percentage on top of each other until it reaches a random value between 0 and 1, or 0% to 100%. This means that there is a random chance that the elite fitness values will not reproduce, which adds in extra genetics to the next generation. This combination of the selection choices results in the algorithm getting over local maximums for the weight-to-value items, that occur from only using elitism.

Where Artificial Babies Come From

The male robot is composed of 0's and 1's, which allows the bits to be combined, spliced, and recombined with the female robot's bits, to produce a new generation of children. Our recombination selected a random bit-string position that selects which bits of the father and mother to use to recombine into the next generation of children. The parents remain in the next generation, until one of the children replace them.

Mutation

We used a simple one-bit mutation to separate the children further from the parents. The one-bit mutation is selected randomly within each child and it's 1 or 0 is then switched to a 0 or 1, respectively. We have considered that using a reversing of bits or more than one-bit swapping maybe more useful for large data sets at the beginning set of generations. The starting children will vary largely and then get more refined as time goes on until single bit flipping.

How to Use

To use the evolutionary algorithm, the user must assign an object of the *mutation* class: *object = Mutation(popsize, itemsize, generation)*. The input values are as follows: *popsize* - integer greater than 2, *itemsize* - integer value greater than 2, and *generation* - integer value greater than 0. The *popsize* variable is used to determine how many knapsacks are created per generation. The *itemsize* variable is used to hold how many items will be randomly generated. The *generation* variable is used to determine how many generations will be ran throughout the programs life.

Next the user will need to use *obj.int_population(min_weight, max_weight, min_value, max_value, animation)* to create the weights and items and generate a matrix of knapsacks given the size of the user defined population. The variable *min_weight* must be a positive number less than the *max_weight* and this variable determines the minimum weight value. The variable *max_weight* which must be a value greater than *min_weight.* The variable *min_value* must be less than *max_value*, which determines the minimum value of the items. The variable *max_value* must be greater than *min_value* and determines the maximum value that an item can be assigned. Lastly, the variable *animation* is a Boolean that determines if the bar graph should be animated or not. The user should use **False** if they do not want an animated bar graph as the population changes over each generation, or use **True** if they wish to be sucked into black hole that is watching natural computing algorithms change over time.

1.3 Results

Roulette Only

We first tested with just a population size of 10, a generation number of a 10, and an item size of 10, while using only roulette selection. Item weights and values were set from a range of 1 to 15 for all of these tests. This resulted in not finding the correct item list due to the fact that both the parents selected can be worse than the previous generation. This is shown in Figure 3 as the graph goes up and down, instead of just trending towards the maximum. The EA was faster than the dynamic algorithm, but had a 60% error from the actual value as shown in Figure 2.

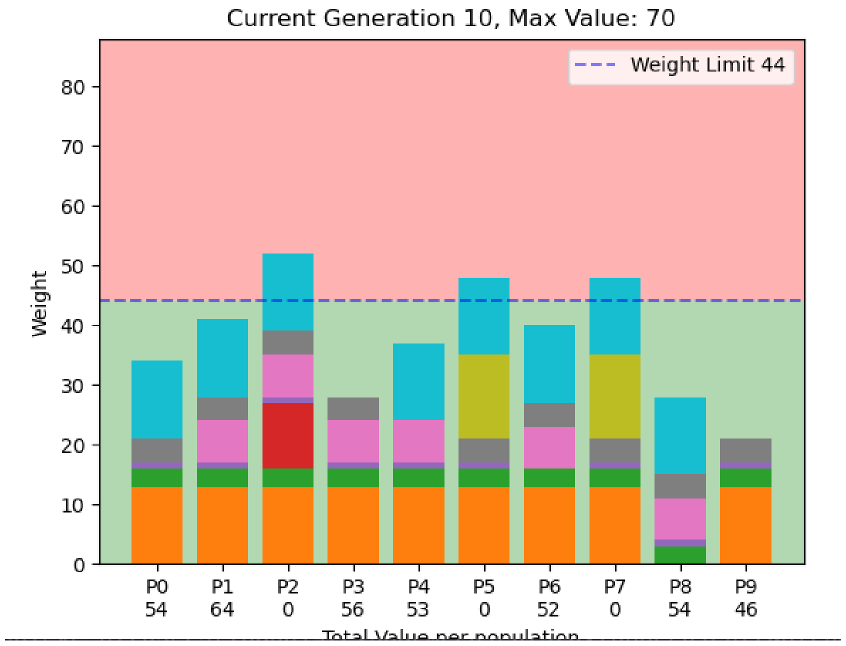
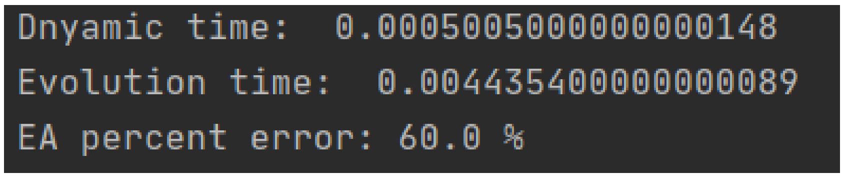


Figure 1: *Image of the final population represented by the bar graph over 10 generations.*



```
Dnyamic time:  0.0005005000000000148
Evolution time:  0.004435400000000089
EA percent error: 60.0 %
```

Figure 2: *Image of the times for both the EA and the dynamic algorithm, while also showing how far off the from the actual value the EA's champion.*

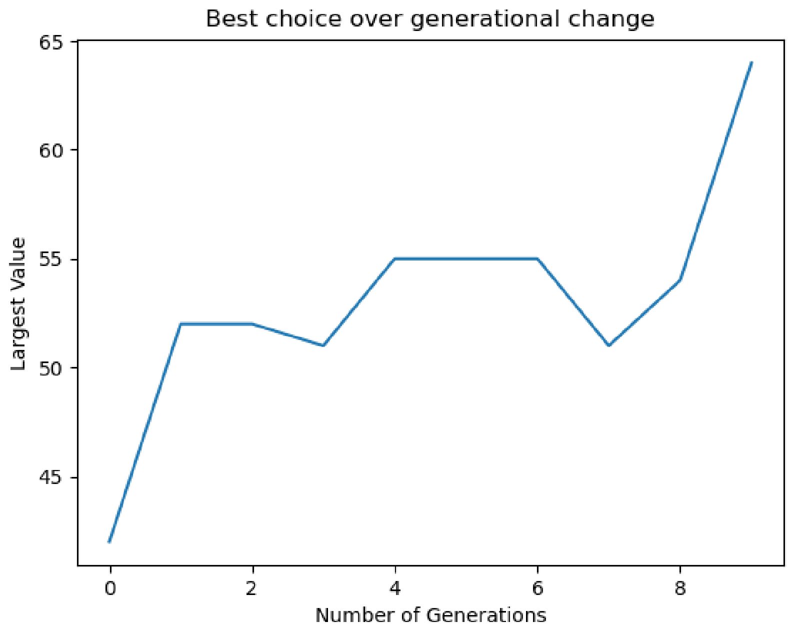


Figure 3: *Image of the best selected population throughout the 10 generations.*

We decided to increase the amount of generations that occurred to allow for more mutations and a longer time to reach the maximum value. This did not happen as shown in the figures below. Figure 6 shows that the EA was trending towards the maximum value and then decided to through all of their ancestor's handwork. The percent error was found to be 54.9% error in Figure 5.

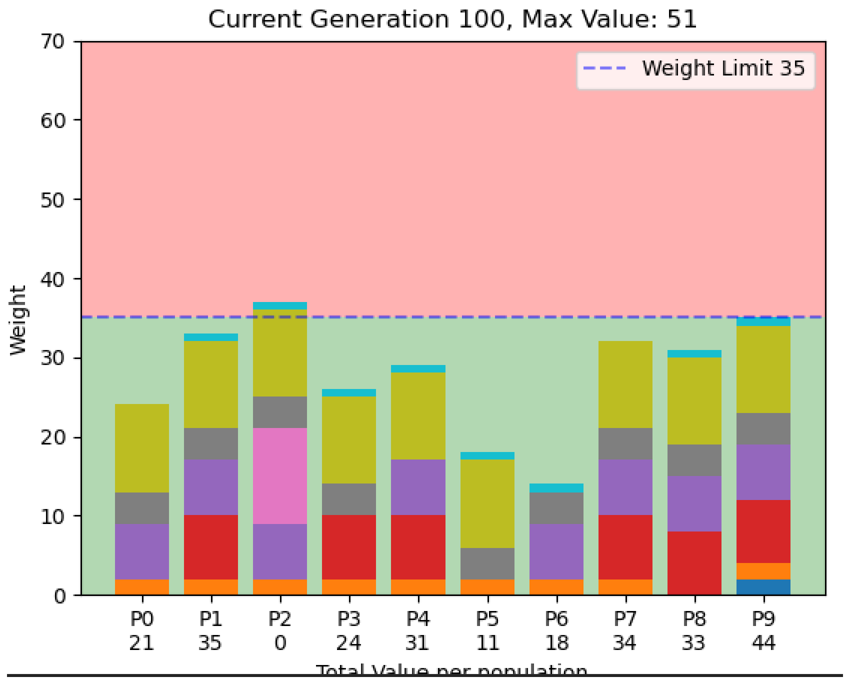
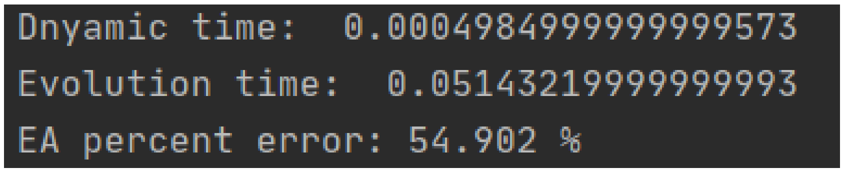


Figure 4: *Image of the final population represented by the bar graph over 100 generations.*



```
Dnyamic time:  0.0004984999999999573
Evolution time:  0.05143219999999993
EA percent error: 54.902 %
```

Figure 5: *Image of the times for both the EA and the dynamic algorithm, while also showing how far off the from the actual value the EA's champion.*

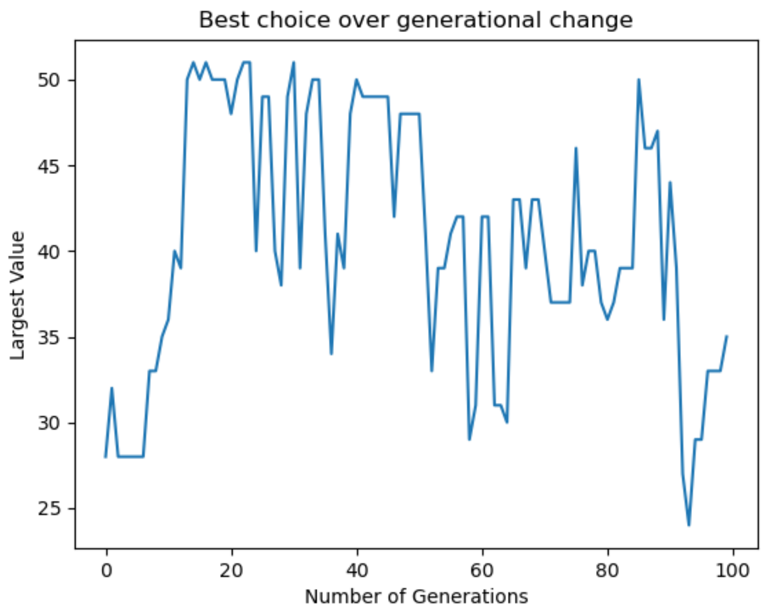


Figure 6: *Image of the best selected population throughout the 100 generations.*

Roulette and Elitism

This combination of roulette and elitism selection worked out to be the best as the about 95% of the runs got the exact answer and the answers that were incorrect came out within 92% of the actual answer.

We tested the 10 generational setup as with the roulette only selection method, but this time with the mixed elitism and roulette choices. Figure 7 shows the final population of generation 10 with the correct items selected. Figure 8 shows the steps of upward trending gain after each generation unlike that for the roulette only selection, which changed widely. Figure 9 shows that the EA was just as fast as the dynamic algorithm.

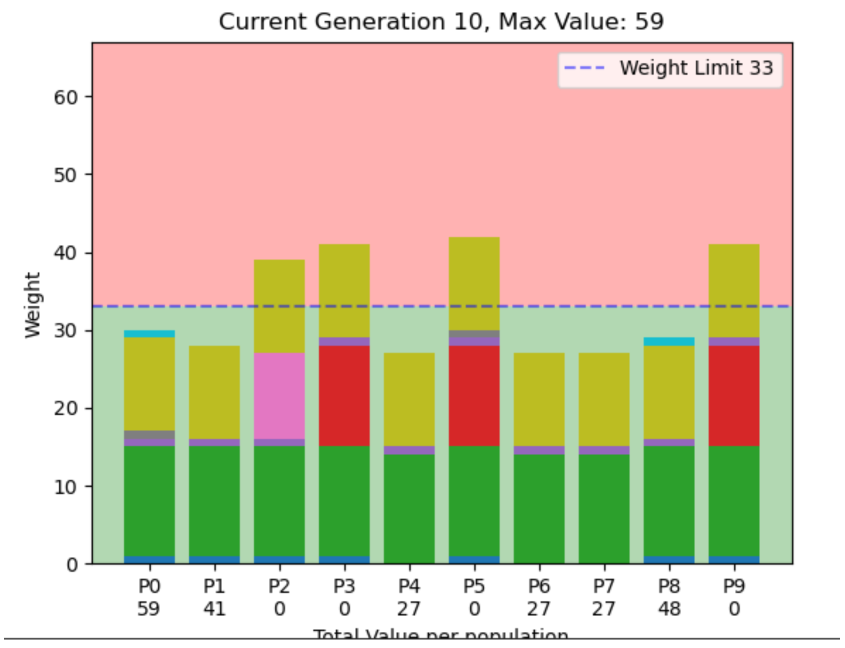


Figure 7: *Image of the final population represented by the bar graph over 10 generations, using mixed selection methods.*

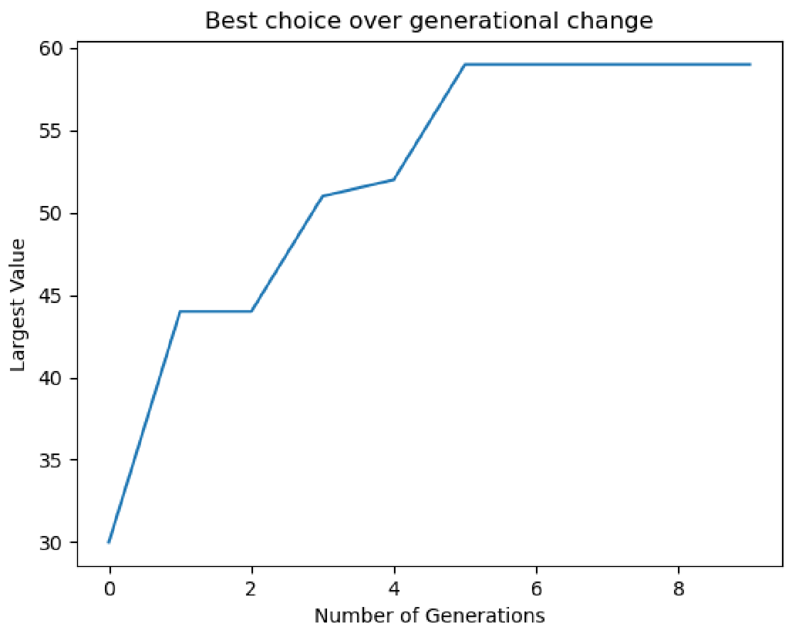
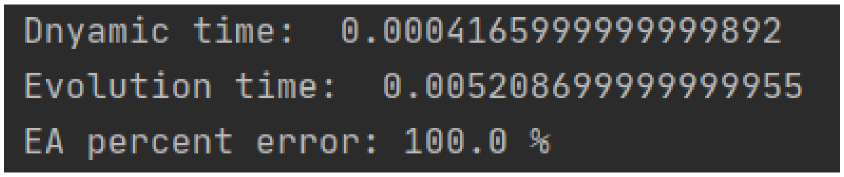


Figure 8: *Image of the best selected population throughout the 10 generations using mixed selection.*



```
Dnyamic time:  0.0004165999999999892
Evolution time:  0.005208699999999955
EA percent error: 100.0 %
```

Figure 9: *Image of the times for both the EA and the dynamic algorithm, while also showing how far off the from the actual value the EA's champion.*

We then tested using 100 generations to compare against the roulette selection at its best, which means more time and generations. Again the mixed roulette and elitism worked to find the exact answer as shown in Figure 10. Figure 11 shows that we wasted 80 generations as the actual answer had been found. Figure 12 shows that the EA was much slower than the dynamic algorithm.

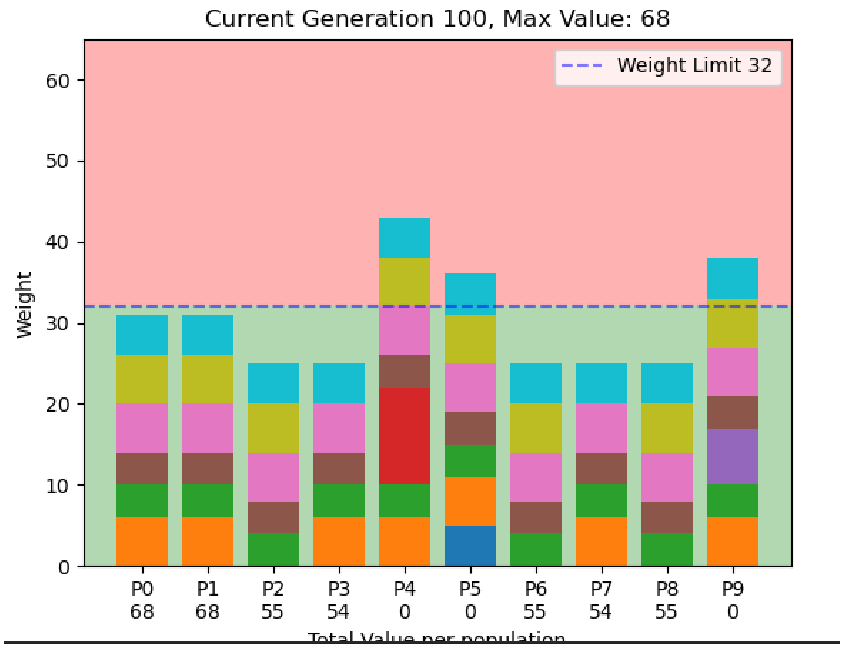


Figure 10: *Image of the final population represented by the bar graph over 100 generations, using mixed selection methods.*

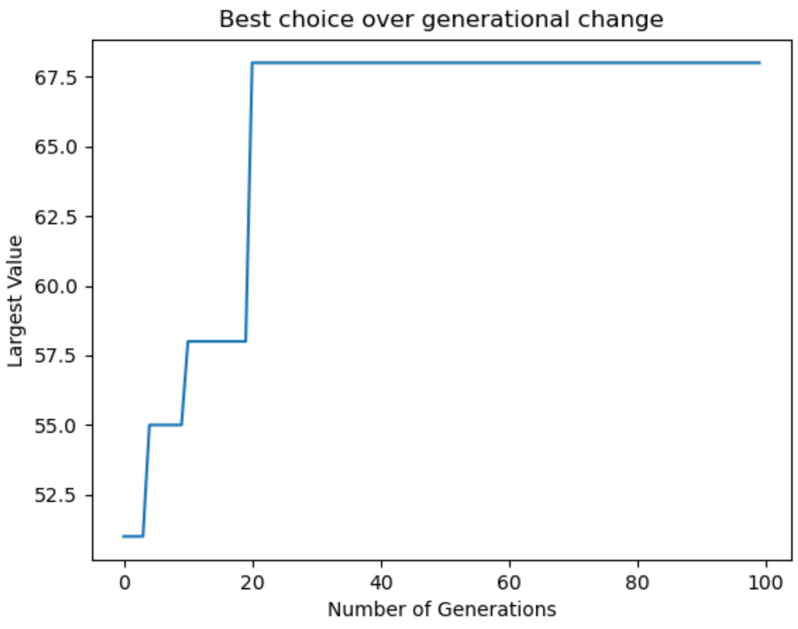
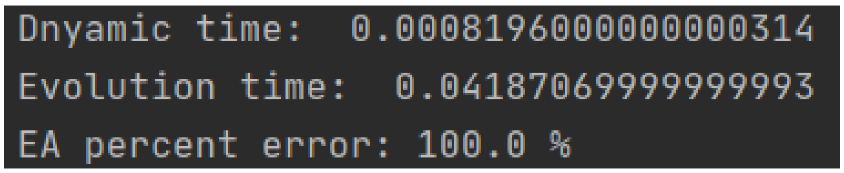


Figure 11: *Image of the best selected population throughout the 100 generations using mixed selection.*



```
Dnyamic time:  0.000819600000000314
Evolution time:  0.04187069999999993
EA percent error: 100.0 %
```

Figure 12: *Image of the times for both the EA and the dynamic algorithm, while also showing how far off the from the actual value the EA's champion.*

Time and Precision

Lastly, we tested a large amount of items, which causes the dynamic program to take more time to calculate as well as almost crash our laptop from our crippling *Chrome* tab addiction. While the EA did not find the exact it was however much faster at finding a set of items that was within 85% error of the exact answer. This test was conducted with a population size of 20, an item size of 150, and 100 generations. The items max weights and values were changed to 50.

Figure 13 shows how the best of the each population was being selected over the 100 generations. Looking at the upward trending line shows that the algorithm would have found a maximum value if the generation size was changed to a much larger size. Figure 14 shows the amazing features of having a huge value amount shown in the standard window size.

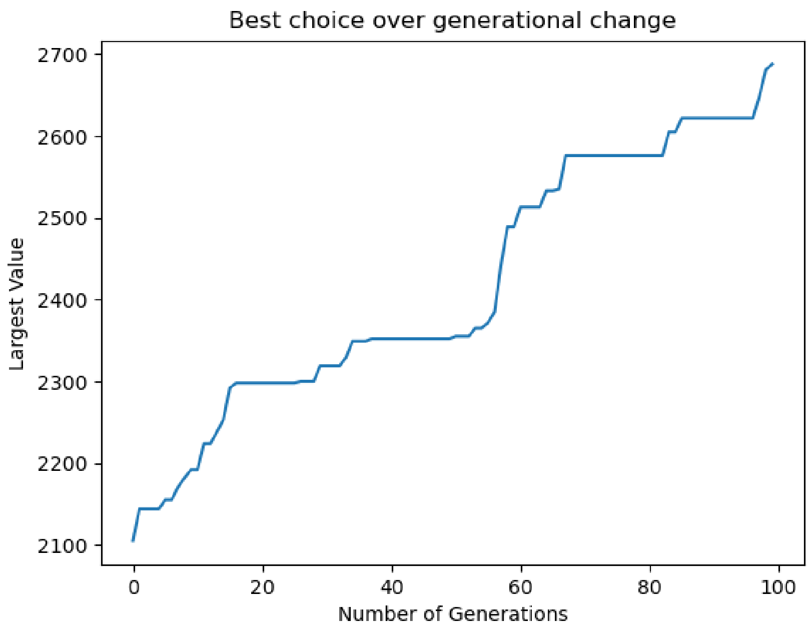


Figure 13: *Image of the final population represented by the bar graph over 100 generations, using mixed selection methods.*

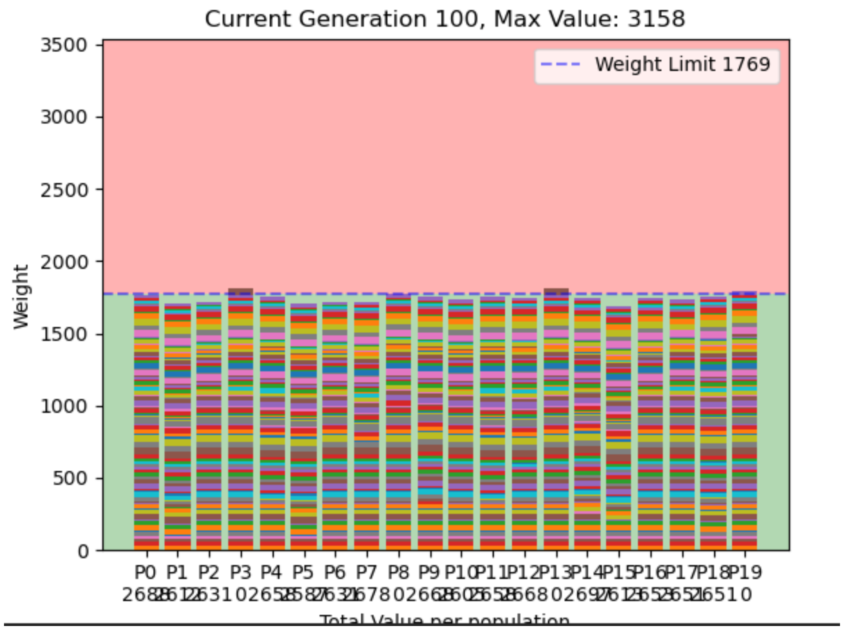
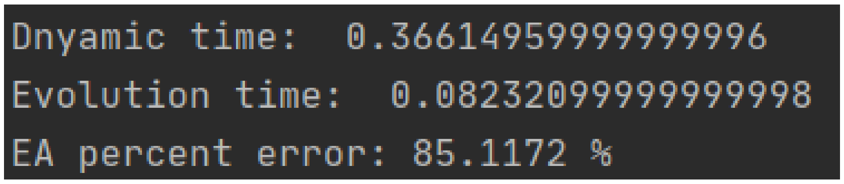


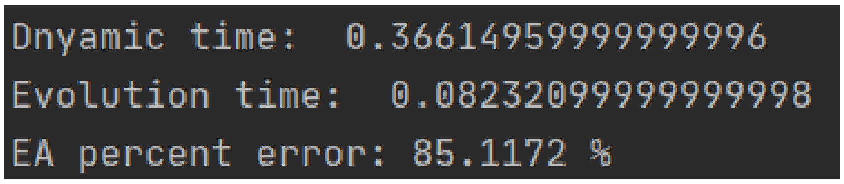
Figure 14: *Image of the best selected population throughout the 100 generations using mixed selection.*



```
Dnyamic time:  0.36614959999999996
Evolution time:  0.08232099999999998
EA percent error: 85.1172 %
```

Figure 15: *Image of the times for both the EA and the dynamic algorithm, while also showing how far off the from the actual value the EA's champion.*

The figure above shows the speed increase of using an evolution algorithm over a dynamic algorithm if one is willing to sacrifice getting the exact solution. Since only 100 generations were allowed, the user could increase the amount of generations to get a more exact answer. The figure below shows some off screen testing results of using a generational size of 300 instead of 100 and provided around, while requiring less time than the dynamic algorithm. We consider this be the greatest advantage of EA as at some point the time and memory complexity of the dynamic algorithm will be to great find the exact solution. EA has been showing about an 80% or higher accuracy, which is not the exact best, but provides a starting reference point for a dynamic program. Unfortunately, we could not try larger item sizes as the school decided to provide laptops with weak processing power as well as having very little RAM, so the dynamic algorithm will crash the laptop.



```
Dnyamic time:  0.36614959999999996
Evolution time:  0.08232099999999998
EA percent error: 85.1172 %
```

Figure 16: *Image of the time difference between EA and the dynamic algorithm while using a large item size and .*

1.4 Code

Main File for Testing

```python
import numpy as np, matplotlib.pyplot as plt
from Homework1 import HillClimbing, SimulatedAnnealing
from Homework2 import Mutation

def main():
    mut = Mutation(20, 150, 300)
    mut.int_population(1, 50, 1, 50, False)
    mut.generate()

if __name__ == "__main__":
    main()
```

Code for Evolutionary Algorithm

```
import numpy as np
import matplotlib.pyplot as plt
import time


class Mutation (object):

    # Initialize class
    def __init__(self, popsize=10, itemsize=10, generation=100):

        # holds the population per generation
        self.popsize = popsize

        # holds the size of how many items are within the knapsack
        self.itemsize = itemsize

        # holds the number of generations
        self.generation = generation

        # holds an empty ndarray for the fitted population
        self.fitted_list = np.array([])

    def int_population(self, min_weight, max_weight, min_value, max_value, animation=
        ↪ False):
        # create the population size using either 0 or 1 based off of population size
            ↪  and the number of items
        self.population = np.random.randint(2, size=(self.popsize, self.itemsize))

        # creates the item wights
        self.w = np.random.randint(min_weight, max_weight, size=self.itemsize)

        # creates the item values
        self.v = np.random.randint(min_value, max_value, size=self.itemsize)

        # make sure that the W is less than sum of w
        self.W = int(np.sum(self.w) / 2)

        # used to animate the bar graph
        self.animation = animation

        # used only to compare actual value verses the GA
        self.max_value = 0

    def weightfit(self, index):
        # had a feeling this works and it does... I think I did something like this
           ↪ in ML or I have transcend spacetime
        return np.sum(np.where(self.population[index] == 1, self.w, 0))

    def valuefit(self, index):
        # returns the value of the total value of the items
        return np.sum(np.where(self.population[index] == 1, self.v, 0))

    def popfit(self, current_generation):
        # create empty lists for weights and values
        weight_list = np.array([], dtype=int)
        value_list = np.array([], dtype=int)

        for i in range(0, self.popsize):
            # get the populations total value
            weight = self.weightfit(i)
            value = self.valuefit(i)

            # gather the wights if the total if less than W else 0
            if weight < self.W:
                weight_list = np.append(weight_list, weight)
                value_list = np.append(value_list, value)
            else:
                weight_list = np.append(weight_list, 0)
                value_list = np.append(value_list, 0)

        # create the fitted population array
        index = np.arange(weight_list.size)
        self.fitted_list = np.vstack((weight_list, value_list, index)).T

        # to animate or not to animate
        if self.animation:
            self.barchart(current_generation, self.animation)

        # figured out how to change the argsort from max to min given this website
        # https://www.kite.com/python/answers/how-to-use-numpy-argsort-in-descending-
           ↪ order-in-python
        # column 1 has the value of the objects and it what we want to sort by
        self.fitted_list = self.fitted_list[np.argsort(-1*self.fitted_list[:, 1])]

    def generate(self):
        max = np.array([])
        tic1 = time.perf_counter()
        self.max_value = self.dynamic(self.W, self.w, self.v, self.itemsize)
        toc1 = time.perf_counter()

        if self.animation:
            plt.show()
            plt.ion()
            plt.figure()
            plt.ylim(top=np.sum(self.w))

        tic2 = time.perf_counter()

        for i in range (self.generation):
            # create an array that keeps the fitness of each iteration
            # calculate the fitness of the current population
            self.popfit(i)
            if i == 0:
                max = self.fitted_list[0]
            else:
                max = np.vstack((max, self.fitted_list[0]))
            # mutate the children given the parents
            self.mutate()
        toc2 = time.perf_counter()
        if self.animation:
            plt.ioff()
            plt.close()

        self.barchart(self.generation, False)

        # graph
        self.graph(max)
        print("Dnyamic time: ", toc1 - tic1)
        print("Evolution time: ", toc2 - tic2)
        final_value = self.fitted_list[0][1]
        final_value = int(final_value)
        print(f"EA percent error: {round((final_value/self.max_value*100),4)} %")

    def roulette(self, fitted):
        # start at 0
        n = 0

        # get the total value
        total = np.sum(fitted)

        # get a random value between 0 and 1
        prob = np.random.rand() #* self.mutation_rate

        # start finding
        sum = fitted[0][1] / total
        index = 0
        # while sum < prob, increase n and sum
        while sum < prob:
            n += 1
            index = n % self.popsize
            wait = fitted[index][1] / total
            sum += wait

        # return the
        return index

    def mutate(self):
        # find parents
        dad = 0 # this seems to yields the best results
        mom = self.roulette(self.fitted_list)

        # combine mom and dad for parents
        self.recomb(dad, mom)

        # loop through each of the population, while keeping parents
        for i in range(2, self.popsize):
            # use random integer and take the mod using the item size, or number of
               ↪ bits
            bit_location = np.random.randint(0, self.itemsize) % self.itemsize

            # switch random child's bit
            self.population[i][bit_location] = not(self.population[i][bit_location])

    def recomb(self, dad, mom):
        # move dad and mom to the top of the population list [0, 1]
        dad_sel = self.fitted_list[dad][2]
        dad_pop = self.population[dad_sel]
        mom_sel = self.fitted_list[mom][2]
        mom_pop = self.population[mom_sel]
        self.population[0] = dad_pop
        self.population[1] = mom_pop

        # find random location for children
        for i in range (2, self.popsize):
            # random integer % number of bits
            location = np.random.randint(0, self.itemsize) % self.itemsize

            # a child is born into pain
            child = np.hstack((self.population[0][:location], self.population[1][
               ↪ location:]))

            # set child to the combination of the parents
            self.population[i] = child

    def graph(self, max):
        weight, value, postion = np.hsplit(max, 3)
        x = np.concatenate(value, axis=0)
        y = np.arange(self.generation)
        plt.plot(y, x)
        plt.ylabel('Largest Value')
        plt.xlabel('Number of Generations')
        plt.title('Best choice over generational change')
        plt.show()

    def barchart(self, current_generation, animation):
        # make labels the only way I know how, but it took like 20 minutes to figure
           ↪ this simple thing out
        labels = [str(i) for i in range(self.popsize)]
        labels = ['P' + i for i in labels]

        # empty lists for weights and values
        weight = []
        value = []

        # graph stuff
        if animation:
            plt.clf()

        # limit the height of the graph and make a max weight line
        limit = np.sum(self.w)
        plt.ylim(0, limit)
        plt.axhline(y=self.W, color='b', linestyle='--', alpha=0.5, label=f'Weight
           ↪ Limit {self.W}')

        # graph regions, red = bad, green = good
        plt.axhspan(0, self.W, facecolor='g', alpha=0.3)
        plt.axhspan(self.W, limit, facecolor='r', alpha=0.3)

        # graph each population
        for i in range(self.popsize):
            # find the weights for each population
            weight = np.where(self.population[i] == 1, self.w, 0)

            # find the values for each population
            value = np.where(self.population[i] == 1, self.v, 0)

            # find the total height of the weight and value
            total_value = np.sum(value)

            # check to see if the weight invalidates the item value
            if np.sum(weight) > self.W:
                total_value = 0
            # add in the total value for each
            labels[i] = labels[i] + '\n' + str(total_value)

            # output each of the item as a stacked bar
            for idx in range(weight.size):
                plt.bar(labels[i], weight[idx], bottom=np.sum(weight[:idx]))

        # label stuff
        plt.ylabel('Weight')
        plt.xlabel('Total Value per population')
        title = 'Current Generation ' + str(current_generation) + ', Max Value: ' +
            ↪ str(self.max_value)
        plt.title(title)
        plt.legend()

        if animation:
            plt.pause(0.01)
        else:
            plt.show()

    # code from https://www.geeksforgeeks.org/python-program-for-dynamic-programming-
        ↪ set-10-0-1-knapsack-problem/ by
    # Bhavya Jain, this code is only used to provide the exact answer and for
        ↪ comparison against the genetic algorithm
    def dynamic(self, w, wt, val, n):
        k = [[0 for x in range(w +1)] for x in range (n + 1)]

        for i in range(n+1):
            for W in range(w+1):
                if i == 0 or W == 0:
                    k[i][W] = 0
                elif wt[i-1] <= W:
                    k[i][W] = max(val[i-1] + k[i-1][W-wt[i-1]], k[i-1][W])
                else:
                     k[i][W] = k[i-1][W]
        return k[n][w]
```