Lab 2 – Genetic Processes:

Submitters: Inbar Lev Tov 316327246, Gal Gibly 315093435

Note that the assignment was done with python version 3.11.0

Introduction:

Coming into this assignment, we had a lot to think about. Not only did we want to implement our conclusions from the first assignment in order to improve our algorithm – we also wanted to add all the requirements from this assignment and create a big difference in performances.

That is why, even before working on this assignment, we made some changes:

The Bin Packing Problem:

When we tried using our old fitness calculations for each gene, we found some numeric mistakes that didn't add up to the item's order of insertion. That's why the fitness calculation for the bin packing problem has been changed:

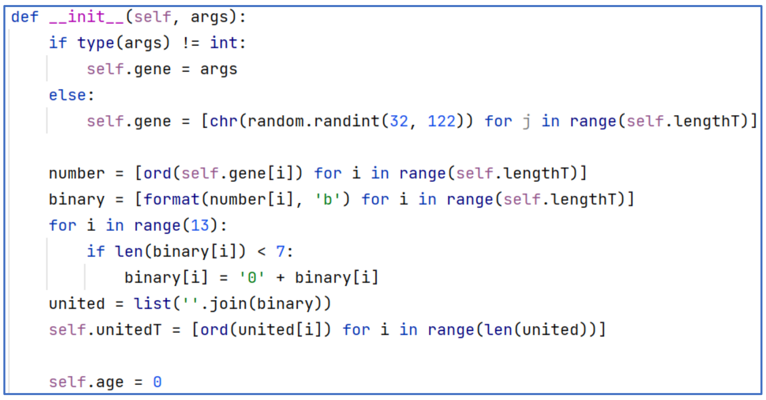
- For each of the 5 problems, we found the **permutation** (order of insertion of items) that would give us the **optimal** number of bins (as defined by the website in the first assignment).
- That permutation would serve as our goal – and so the fitness calculation for each gene would be – how many of the 120 items are in the correct order of insertion.
- First fit heuristic would be the same as before – and now after those changes it would serve as a better comparison to the regular fitness heuristic.

After testing our algorithm with the new fitness, we can now say for certain – that change has been necessary. Not only can we find a suitable solution to the problem in every single run – our algorithm converges to the solution in a much faster fashion. Therefore, we can now solve that problem with a much higher rate of success and accumulate better results to compare for.

Strings:

We have made some small changes – we added parent selection, aging, selection pressure and genetic diversity not just to queens and bins, but also to strings.
However, our biggest change has been changing the representation each gene to binary:

- Our target string is of length 13 – which means that each gene would draw 13 random characters and create its own string from them.
- Each character has an ASCII value between 32 (space) and 122 (lower case 'z').
- In binary representation, that means each character is represented with maximum of 7 bits ($2^7 = 128$).
- Note that, while converting, low value characters such as space ($32 = 2^{5\ bits}$) has been elongated in order to preserve uniform representation per character.
- The entire process was done in the constructor, which is presented here:



```python
def __init__(self, args):
    if type(args) != int:
        self.gene = args
    else:
        self.gene = [chr(random.randint(32, 122)) for j in range(self.lengthT)]

    number = [ord(self.gene[i]) for i in range(self.lengthT)]
    binary = [format(number[i], 'b') for i in range(self.lengthT)]
    for i in range(13):
        if len(binary[i]) < 7:
            binary[i] = '0' + binary[i]
    united = list(''.join(binary))
    self.unitedT = [ord(united[i]) for i in range(len(united))]

    self.age = 0
```

Heuristics For Strings:

Now that we have a binary representation for each gene, we can finally talk about heuristics:

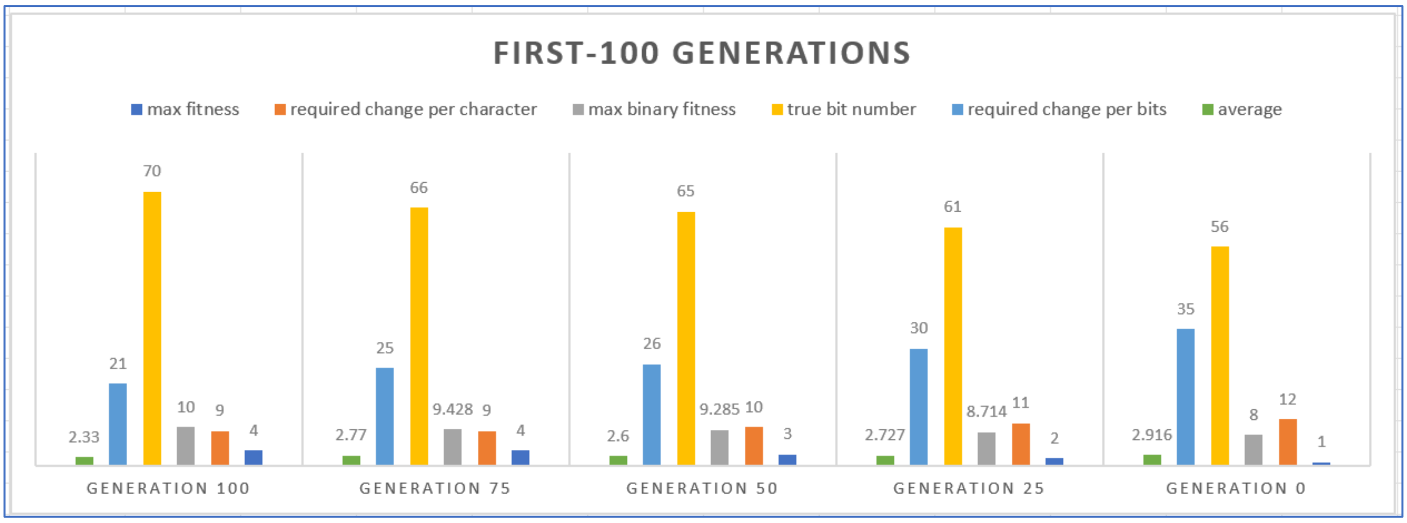
- **Fitness** heuristic only gives points to **characters** that are in the right place.
  Since our target string is of length 13, that's the maximum number of points that we can receive there.
- We would compare it with **Fitness Binary** – the same heuristic, performing the same calculation, only on **bits**.
- Since our target string is of length 13, and we have 7 bits per characters – that's 91 bits for the whole string. In order to perform a true comparison, and show the sensitivity of the algorithm, Fitness Binary would return a result divided by 7 – so it would stand on the same level of values between 0 to 13.

```python
# measure fitness
def fitness(self):
    score = 0
    for i in range(self.lengthT):
        if self.gene[i] == self.target[i]:
            score += 1
    return score
```

```python
# fitness binary
def fitnessBinary(self):
    score = 0
    for i in range(self.lengthB):
        if self.unitedT[i] == self.translateB[i]:
            score += 1
    return score / 7
```

First Comparison:

- For our first run, we did 100 generations, 100 genes per generation, non uniform mutation, SUS and 2 points crossover.
- It's important to mention that this run **doesn't** converge to the target string. This is a **bad run** – but we still chose it, if only to show other, more important aspects of the calculation.
- The results are right here:

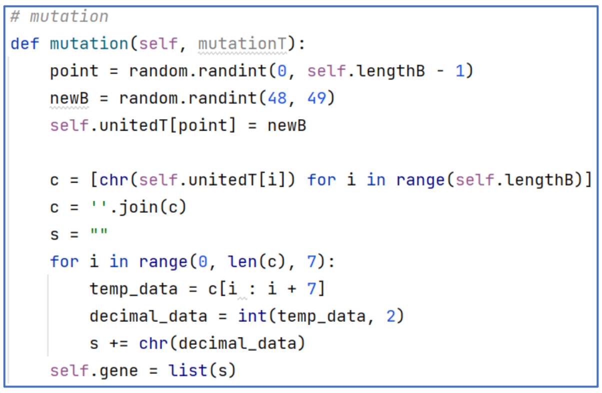


- As you can see, max fitness ranges between 1 and 4 – and the required number of characters to be changed ranges from 12 to 9.
- Supposedly, this is very bad – we need to make a lot of changes in order to reach our target string.
- But, if you look more closely through the "bits lenses", you would see a completely different picture.
- In generation 0, for example, we have 56 matches in bits.
- While only 7 of them assemble one full character in the target string (fitness 1), the other 49 are spread across the string, assembling "half finished" characters.
- Since binary representation is so much more important than the characters (this is how the computer stores information) – it means we are much more closer than we think in reaching our target string.

- So instead of thinking about the challenge of changing 12 characters, we only had to look at it as changing 35 bits through the entire string.
- On average, we only have to change 2.916 bits per character – because we want all 12 of them to match

$$\frac{35}{12} = 2.916 \sim 3$$

- That kind of mutation requires much less work than the original one – which made us also change our entire mutation method.
- Instead of changing an entire character – which would coincidently change 7 bits and cause irregularities we didn't aim for – we change one bit and than convert back to characters.



```python
# mutation
def mutation(self, mutationT):
    point = random.randint(0, self.lengthB - 1)
    newB = random.randint(48, 49)
    self.unitedT[point] = newB

    c = [chr(self.unitedT[i]) for i in range(self.lengthB)]
    c = ''.join(c)
    s = ""
    for i in range(0, len(c), 7):
        temp_data = c[i : i + 7]
        decimal_data = int(temp_data, 2)
        s += chr(decimal_data)
    self.gene = list(s)
```

- We know for a fact that method works better – thanks to the difference between generations 50 & 75.
- Generation 50 had 65 correct bits but only fitness 3.
- Generation 75 had 66 correct bits and suddenly, the fitness jumped to 4.
- Guess we only ever needed to change one bit in the correct spot.

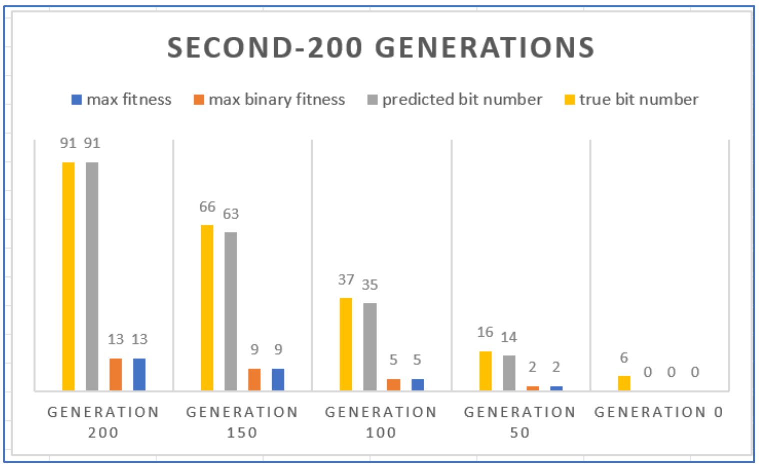
That shows us a couple of things:

- The true distance between genes could be measured with only a single bit, instead of 7 (a character).
- More importantly, the distance between letters, while seemed far away numerically, becomes very small once you convert them all to bits – because you only had to change 1 or 2 bits in order to pass from one letter to another.
- Changing them as characters, however, would have required us to draw them randomly from 91 possible characters – which would have drastically decreased our chances of having a meaningful mutation process.
- Lastly, it shows that when you work on bit level (higher resolution), the algorithm becomes much more sensitive to changes – and our results are much more accurate.

<u>Second Comparison:</u>

- For our second run, we did 200 generations, 200 genes per generation, adaptive mutation, ranking + tournament and uniform crossover.
- This run is considered a good one – we reach our target string at the end, and we don't get stuck at local optima.
- Therefore, our main objective here, was to check the effectiveness of the mutation itself.
- We have a few options here:
  - Replacing a bad bit with a good one somewhere in the string.
  - Replacing a bad bit with a good one and thus assembling a full character (sequence of 7).
  - Replacing a good bit with a bad one.
- The "mutation methods" section deals with option 3.
- As for option 1, we saw it in the first run – and though we appreciate the fact that our correct bit number is growing, it still creates false representation when it comes to fitness.

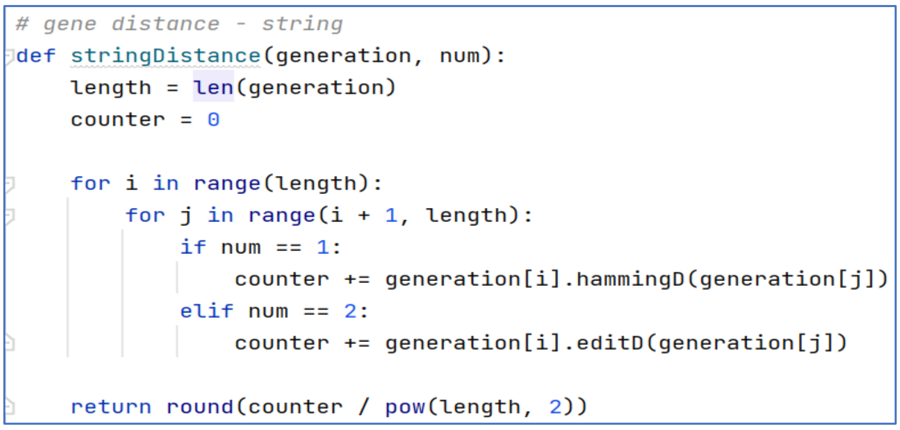
- This run, however, shows the gradual growth of both character fitness and binary fitness – and how they respond to one another:



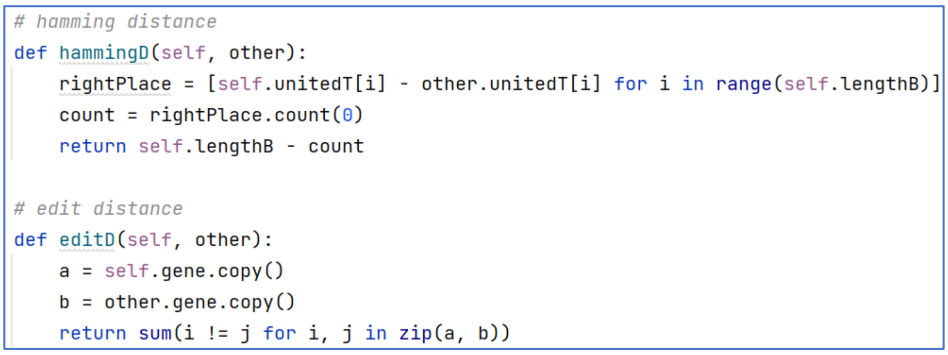
- As you can see in the graph, the actual bit number deviates a little from the predicted number – but that's OK, because most of the correct bits are bound together in sequences of 7 – which allow us to have true coordination between character fitness and binary fitness.
- It means the mutation itself is effective. Not only do we improve our correct bit number – we also improve it at the right place.
- That way, we can look at either character fitness or binary fitness – and still receive a fairly correct assessment about our situation.

Similarity Metrics For Strings:

- Hamming distance: we subtract the binary string between every 2 genes, and count the number of **different** bits (result of subtraction **is not** 0).
  Since we count it for every 2 genes in a generation, we basically have $(n) \cdot (n-1) = n^2 - n \sim O(n^2)$ calculations. The sum of results is extremely high. Therefore, we normalize it by dividing it by n^2 and than rounding the result – we don't have "half" of a different bit.
- Edit distance: we count the number of actions required to get from one string to another.
  Once again, since we count it for every pair of genes in our generation, we do the same normalization process we did in the hamming distance.



```python
# gene distance - string
def stringDistance(generation, num):
    length = len(generation)
    counter = 0

    for i in range(length):
        for j in range(i + 1, length):
            if num == 1:
                counter += generation[i].hammingD(generation[j])
            elif num == 2:
                counter += generation[i].editD(generation[j])

    return round(counter / pow(length, 2))
```



```python
# hamming distance
def hammingD(self, other):
    rightPlace = [self.unitedT[i] - other.unitedT[i] for i in range(self.lengthB)]
    count = rightPlace.count(0)
    return self.lengthB - count

# edit distance
def editD(self, other):
    a = self.gene.copy()
    b = other.gene.copy()
    return sum(i != j for i, j in zip(a, b))
```

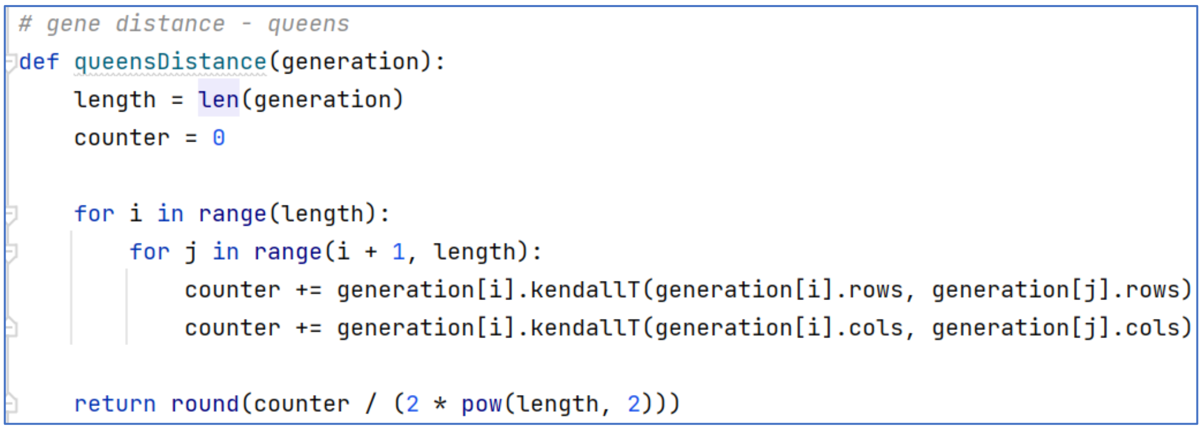
- After testing those 2 methods, we can clearly say that the hamming distance is much more accurate.
- Process wise, it's easier. In one run, hamming distance gives us all the indices where the bits are different. So technically, we can change all of them at once and than return the new improved string.
- Edit distance, on the other hand, requires us to constantly asses both strings in every run, decide on an action (insert, delete, update), and than preform it. It's a much more taxing process.
- Another aspect we might consider, is run time.
- Changing bits is a much more "atomically inclined" operation, because it works on the most basic pieces of data the computer has.
- Operations such as insertion, deletion and updating a list (string is a list of characters), are not atomic.
- Therefore, when we execute the conclusions from those distance metrics, it's much easier for us to follow hamming rather than it is to follow edit.

<u>Similarity Metrics For Queens:</u>

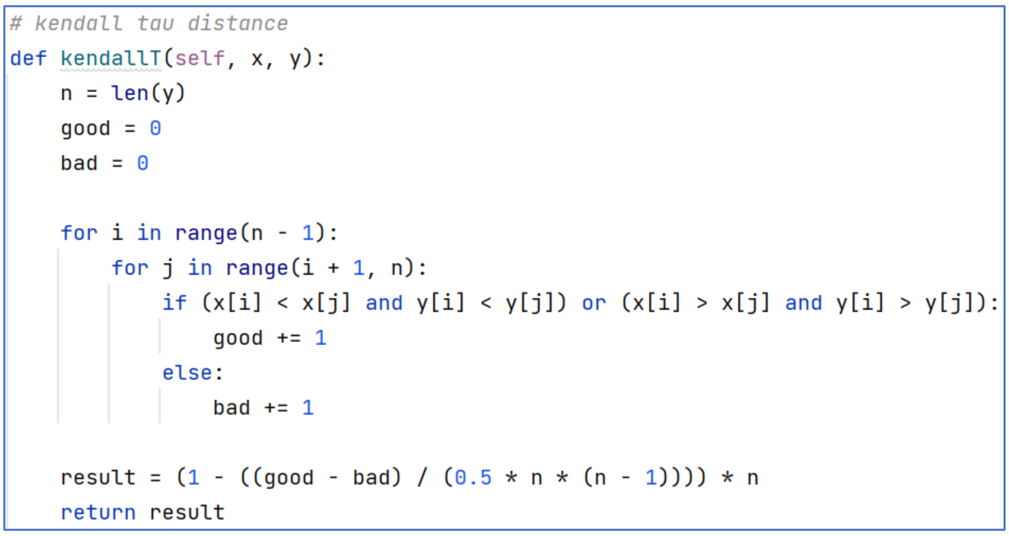
- Kendall tau: for each pair of genes in our generation, we count the number of alleles pairs that are in a different order in the second permutation than they are in the first
- For example, if those are our permutations

$$\begin{array}{ll} A & 0\ 3\ 1\ 6\ 2\ 5\ 4 \\ : & \\ B & 1\ 0\ 3\ 6\ 4\ 2\ 5 \end{array}$$

- We have 4 pairs of alleles that are problematic:
  - in A 0 is before 1 and in B 1 is before 0
  - in A 3 is before 1 and in B 1 is before 3
  - in A 2 is before 4 and in B 4 is before 2
  - in A 4 is before 5 and in B 5 is before 4
- As always, since we accumulated this number on $O(n^2)$ pairs of genes, we normalized it with the same process as the strings.



```python
# gene distance - queens
def queensDistance(generation):
    length = len(generation)
    counter = 0

    for i in range(length):
        for j in range(i + 1, length):
            counter += generation[i].kendallT(generation[i].rows, generation[j].rows)
            counter += generation[i].kendallT(generation[i].cols, generation[j].cols)

    return round(counter / (2 * pow(length, 2)))
```

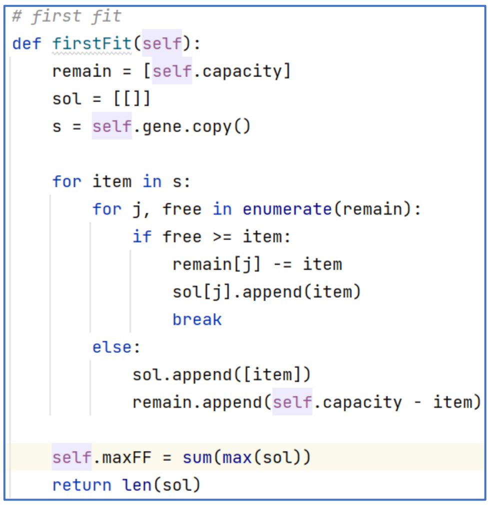


```python
# kendall tau distance
def kendallT(self, x, y):
    n = len(y)
    good = 0
    bad = 0

    for i in range(n - 1):
        for j in range(i + 1, n):
            if (x[i] < x[j] and y[i] < y[j]) or (x[i] > x[j] and y[i] > y[j]):
                good += 1
            else:
                bad += 1

    result = (1 - ((good - bad) / (0.5 * n * (n - 1)))) * n
    return result
```

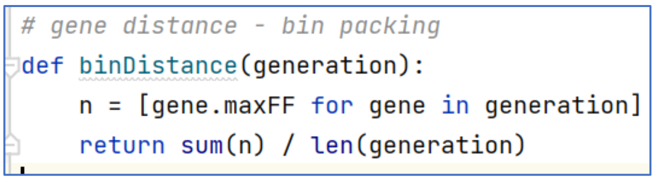
- We found it to be the most suitable metric for this kind of problem.
- We don't need to add new alleles to each permutation or change the alleles to new numbers – we only need to replace the order of alleles that already exist in our permutation.
- Those very simple actions save us time and complications – and bring us closer to the correct result.

Similarity Metrics For Bins:

- When we calculate the first fit heuristic, we keep the bin that utilizes its capacity the most.
  - The capacity is 150.
  - We have bins that stores items with accumulated worth ranging from 138 to 148.
  - Our most outstanding bin is the one that manages to store items worth 148.
- Now: for every 2 genes, we measure the distance between their outstanding bins, and of course normalize it for each generation.
- Notice the highlighted line:



```python
# first fit
def firstFit(self):
    remain = [self.capacity]
    sol = [[]]
    s = self.gene.copy()

    for item in s:
        for j, free in enumerate(remain):
            if free >= item:
                remain[j] -= item
                sol[j].append(item)
                break
        else:
            sol.append([item])
            remain.append(self.capacity - item)

    self.maxFF = sum(max(sol))
    return len(sol)
```



```python
# gene distance - bin packing
def binDistance(generation):
    n = [gene.maxFF for gene in generation]
    return sum(n) / len(generation)
```

- We found that distance metric to be the most effective – because it serves as an indicator as to the effectiveness of order of insertion for gene and thus the packing that stems from it.
- Across evolution, our way of packing becomes more effective (because the order of insertion converges to the right solution) – and the distance becomes smaller.
- More bins manage to utilize their full capacity – and the number of bins required decreases.

Mutations:

The main goal of performing mutation is to fight convergence to local optima by increasing the genetic diversity of the genes. The questions remain:

- Should every gene be mutated?
- How do we prevent hurting a good gene by accidently forcing mutation that would (for example) replace a good bit with a bad one in strings?

Let's review those 5 methods, and than compare and see a major difference in performing mutation.

Basic:

- We let the user choose the mutation probability – a number between 0 and 1.
- Than for each child, we draw a random probability. If that number is lower than the user's prediction – which means we are within range – we perform the mutation. Otherwise, the child stays the same.

Non uniform:

- Before calling the mutation function, the general loop within the genetic algorithm updates the global variable in accordance with the index of the loop (generation's number)

$$1 - \frac{current\ generation\ number}{number\ of\ generations}$$

- As you can see, early generations receive very high probability, while the last generations receive barely any probability at all.
- That way we favor early generations over the last ones.
- You can also see that the probability decreases in a linear line across evolution – we keep an equal ratio when passing from one generation to the next.

Adaptive:

- If an individual has relatively high fitness in comparison to the rest of his generation, we don't want to change him.
- That is why, when deciding mutation probability, we compare the child's fitness to the fitness mean of the entire generation.
- If the child is bigger – probability is 0.25.
- If the child is smaller (we do want to change him) – probability is 0.75.

THM:

- We want the children to be an improvement in comparison to the parents.
- Unlike adaptive mutation, we now take the **parent** with the highest fitness score and compare it to the child's.
- If a child's fitness is lower, it receives higher mutation probability (0.75).
- If a child's fitness is higher, it receives lower mutation probability (0.25).
- Another thing we might consider, is stagnation.
- The main loop in the genetic algorithm monitors the children's progress – and if even after mutation we see no improvement (global variable "not improved" is above 5) – we use it as a threshold to increase mutation probability even more.

Self adaptive:

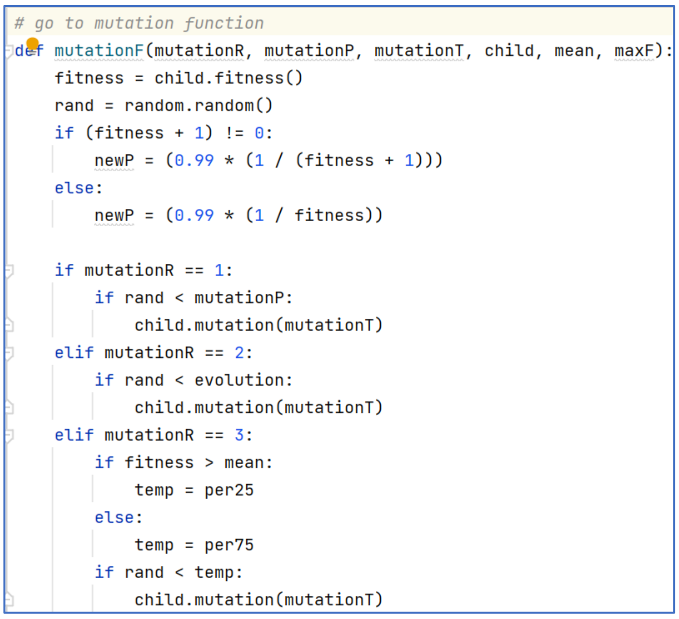
- We set a maximum value to mutation probability (0.99).
- We than adjust the mutation probability per individual according to formula.
- If fitness is 0

$$(0.99) \cdot \frac{1}{fitness + 1}$$

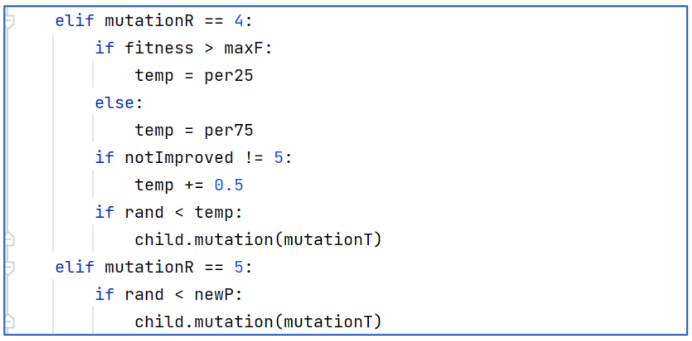
- If fitness is not 0

$$(0.99) \cdot \frac{1}{fitness}$$

- If it has high fitness, that formula would produce very small probability – and that's good, because we don't want to change the good ones.
- If it has low fitness we would receive a relatively high probability – helps us to change weak children.
- The full implementation is on the next page.



```python
# go to mutation function
def mutationF(mutationR, mutationP, mutationT, child, mean, maxF):
    fitness = child.fitness()
    rand = random.random()
    if (fitness + 1) != 0:
        newP = (0.99 * (1 / (fitness + 1)))
    else:
        newP = (0.99 * (1 / fitness))

    if mutationR == 1:
        if rand < mutationP:
            child.mutation(mutationT)
    elif mutationR == 2:
        if rand < evolution:
            child.mutation(mutationT)
    elif mutationR == 3:
        if fitness > mean:
            temp = per25
        else:
            temp = per75
        if rand < temp:
            child.mutation(mutationT)
```



```python
    elif mutationR == 4:
        if fitness > maxF:
            temp = per25
        else:
            temp = per75
        if notImproved != 5:
            temp += 0.5
        if rand < temp:
            child.mutation(mutationT)
    elif mutationR == 5:
        if rand < newP:
            child.mutation(mutationT)
```

<u>First Comparison:</u>

When we compared the mutation methods, we tried to understand two things:

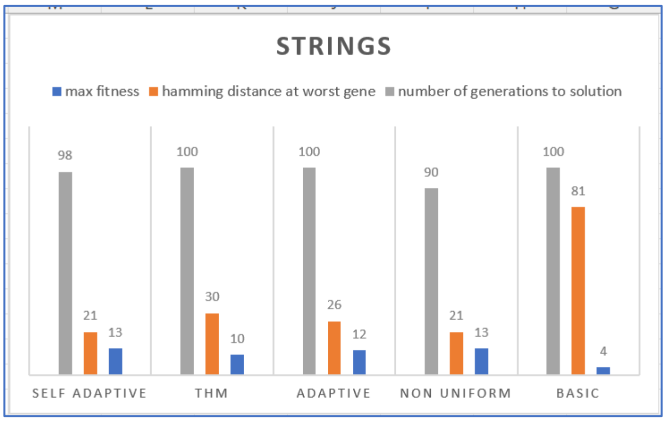
- Which mutation method is the best for each of the 3 problems?
- Which mutation method is the best in general?

In order to answer those questions, we ran all 5 methods on each of the problems, and compared the results in terms of fitness, genetic diversity and the number of generations it took in order to reach our solution.

It's important to point out, that all of them received the same parameters: 100 generations, 100 genes per generation, 0.5 for basic mutation, tournament + ranking for parent selection. the only true difference is the mutation probability.

Here are the results:

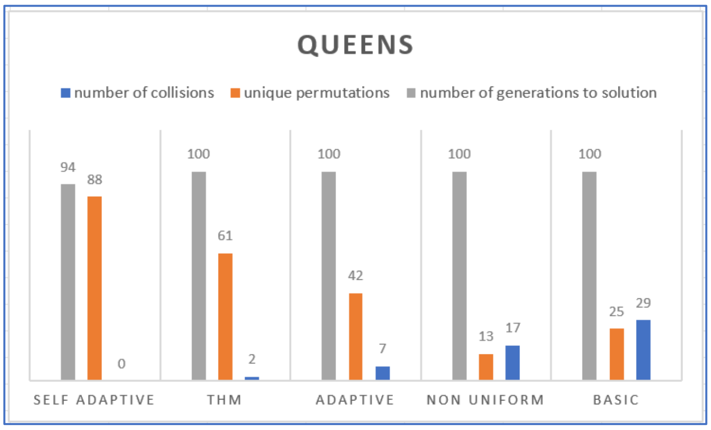
Strings:



- First of all, let's look at fitness. In terms of fitness, both non uniform and self adaptive have reached our solution, while the other three haven't. That puts them in a level of their own.
- Basic is, of course, the most unpredictable and therefore the worst. Worst fitness, worst number of incorrect bits – and because of that sheer number, those bits are scattered across the entire string – which makes it harder to fix them.
- Adaptive and THM, however, make an interesting comparison. At their worst gene, they roughly have the same number of incorrect bits. The difference here, we concur, is the way in which those incorrect bits are scattered across the string. Adaptive have more of an "ordered" scattering – which allows him to fix the string faster and reach the finish line with **supposedly** more fitness.
- If we had to choose a third place – it would have been adaptive.
- Now let's look at our two finalists. Both of them reach our solution in the end, but non uniform finishes 8 generations before self adaptive.
- If they had finished at generation 50 out of 100, we would have said it's too early. But since both of them finish very close to the limit of the algorithm, we consider them both to be good runs.

Non uniform is the winner

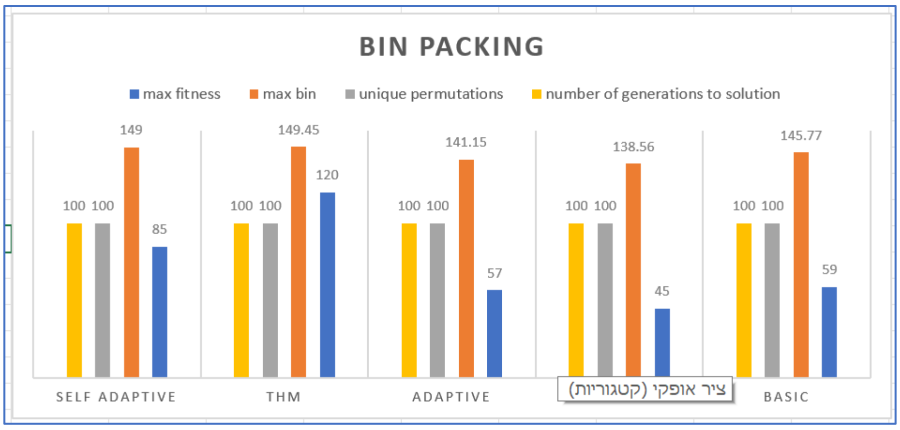
Queens:



- In terms of collisions, self adaptive wins (he is the only one that reached our goal – no collisions between queens).
- Once again, basic is the worst, because it has the biggest number of collisions. However, the interesting phenomenon is looking at the different number of permutations between basic and non uniform.
- As you can see, basic is more genetically diverse than non uniform – but non uniform has better number of collisions because each of his **unique** permutations is better than the basic.
- It shows that, sometimes, diversity doesn't necessarily help you. In fact, it might even slows you down. Each of those permutations now contains a big number of collisions – and we now have to fix all of them, thinking it would help us in the future.
- Another interesting thing to see, is the "almost winner".
- THM was very, very close to reach zero collisions, and, supposedly, had we had more time (let's say 10 more generations), we might have looked at 2 winners.
- If that was the case, we would have had to add another parameter to judge for – considering we don't want to judge based on diversity level and number of generations, it would have had to be something else.
- Our suggestion: Kendall tau distance. Because the way to resolve collisions is to change the permutations, and the number of changes to be performed is measured with the distance between every 2 permutations.

But anyway, final conclusion: self adaptive wins!

Bins:



- First thing to notice: basic has better fitness than adaptive. It's not the worst anymore.
- However, it also shows how unpredictable that is – we can have one good run with him and than 8 bad runs afterward – even if we put the same parameters.
- In terms of fitness, non uniform is now the least successful, and you can also see it in it's "utilizing his capacity" capabilities.
- On average, non uniform manages to store in its bins items worth 138.56 per bin. Since our capacity is 150, and all the other mutations manage to create an order of insertion that would help the first fit heuristic to store items worth 140+ ➔ that mutation simply isn't good enough.
- Another thing to notice about utilizing bin capacity, is the difference between adaptive and basic. Basic has two more items than adaptive that are in the correct order – and suddenly the utilizing capacity jumps by 4 points.
- It shows us how sensitive the algorithm is – because those 2 items don't just affect the current bin in which they are stored, but all the other bins that follow. Utilizing the most from a single bin can effect the entire chain – and maybe even reduce the number of bins that we need.
- In terms of diversity (number of permutations) – all of them are equal. That is because we truly do "sample" from an endless pool of possible permutations (120! is a huge number).
- That difference is important when you compare bins to queens on the same 5 permutations. Even at it's highest (10 queens), queens don't really have that many permutations to chose from, and that difference is visible through the 2 graphs.
- Lastly, let's look at our winner – THM. Reaches solution in the last generation, have the best numbers when utilizing its capacity – simply the best of all the 5.

THM is the winner.

<u>Second Comparison:</u>

After finishing all the simulations, the second question remains: which mutation method is the best?

The answer is… complicated:

- As we have already mentioned, basic is unpredictable and therefore problematic to compare with from the beginning.
- Adaptive was never the winner but did show some interesting aspects along the way.
- The other 3 also vary: one time they win, one time they almost win but only with more time (which we cannot always give), and one time they pummel to the ground so fast you wouldn't even think about considering them for the race.

Guess our answer is: depends on the problem, and… a little bit of random luck when it comes to basic.

Niching, Crowding, Clustering:

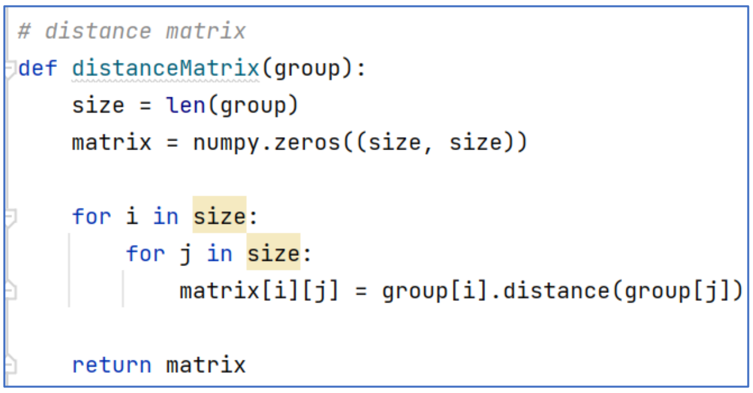
We have now moved on to the second part of the program – running algorithms that converge to multiple solutions. Niching is the first algorithm out of the 3 – and we would see how each of them (niching, non deterministic crowding and clustering) have distinct parameters that helps them produce unique results.

Niching:

The niching algorithm goes pretty much the same as the old genetic algorithm – except now we not only work on a generational level – we also work on groups (niches) level.

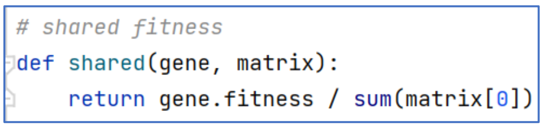
We made the necessary changes as follows:

- Initialization of generation 0: we had to get some form of partitioning, so we simply and arbitrarily divided the population to 20 groups (5% of population members in each group).
- Calculating fitness for each individual within the group.
- Calculate distance matrix for each pair of genes within the group



```python
# distance matrix
def distanceMatrix(group):
    size = len(group)
    matrix = numpy.zeros((size, size))

    for i in size:
        for j in size:
            matrix[i][j] = group[i].distance(group[j])

    return matrix
```

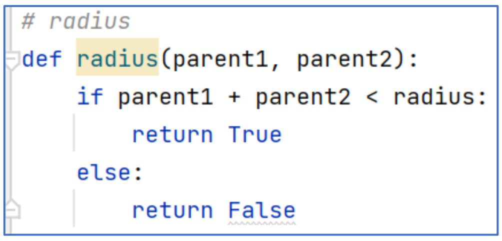
- Calculating share fitness for each individual within the group



```python
# shared fitness
def shared(gene, matrix):
    return gene.fitness / sum(matrix[0])
```

After that everything is pretty much the same – until we reach the part of choosing parents.

- Every generation would have the same 20 groups – with the difference being the genes those groups contain.
- We balance it out between elitism (exploitation) and new creation (exploration).
- That is why, for each group, we still have 10% that are the elite team – and 90% of new creation.
- Unlike crowding, which would be discussed later, here we wanted to play with the parameters a little bit and therefore increase the genetic diversity within each group – instead of maintaining it like in crowding.
- Therefore, when the time comes for choosing parents, we do the following:
- we use the parent selection method the same as before, but we also check with regards to radius:
  - each of the parents can stand alone within the radius limits of the niche
  - If their shared fitness is close (**sum** of them stays within the radius limits of niche) – that makes them too close together and less diverse for our taste.
  - We want them diverse (far apart in fitness and capabilities) – so we would search for a false.



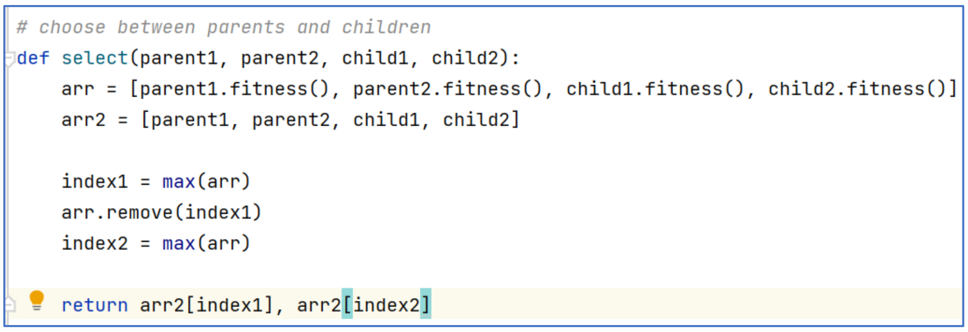
```python
# radius
def radius(parent1, parent2):
    if parent1 + parent2 < radius:
        return True
    else:
        return False
```

- That way we would get children that don't necessarily fit the same radius criteria as their parents, and therefore go from one niche to another.
- On a generational level, you could look at one generation and received a certain level of diversity between groups, and than look at another generation and receive:
  - Completely different groups in terms of members and their fitness.
  - Completely different radius limit for each group.
  - Completely different size for each group.
  - Completely different diversity level.
- We are now able to compare between different groups from each generation and that makes it more interesting.

<u>Non Deterministic Crowding:</u>

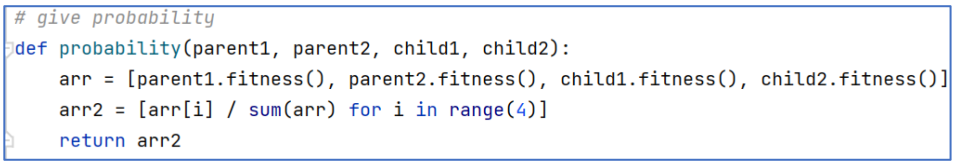
Non deterministic crowding derive most of its strategy from niching – but it does have some notable exceptions:

- Unlike with niching, here we **do** want to maintain the same level of genetic diversity between the groups.
- Therefore, whenever we check if parents can mate, we would search for a true – parents that are close together in their shared fitness would create a child that has the same level of fitness.
- Another thing we now consider is – is the child better than his parents, or is the new creation only interrupting the natural process of evolution?
- In order to determine that, instead of creating one child – we create two.
- Than we compare the children to the parents and always take one of 3 options:
  - Two parents.
  - One parent one child.
  - Two children.
- That way, we always make sure there is an improvement to the genes, and that the process of creating children actually helps us instead of hurting us.



```python
# choose between parents and children
def select(parent1, parent2, child1, child2):
    arr = [parent1.fitness(), parent2.fitness(), child1.fitness(), child2.fitness()]
    arr2 = [parent1, parent2, child1, child2]

    index1 = max(arr)
    arr.remove(index1)
    index2 = max(arr)

    return arr2[index1], arr2[index2]
```

- The non deterministic factor here comes from the probability that each of them would be chosen.
- After calculating the probability for each of them, we compare those probabilities with the actual selection results.
- If we were right, the entire niche receives an extra point to their radius. Otherwise there is no change.
- This is a way for us to show favoritism towards a successful group – since we want pairs of parents that can both stand together under the radius limitation, every point the radius gets increases our chances of that.
- This group would be able to constantly make children – and since we have a tendency to increase fitness with every new generation conceived, we might get to our solution faster than expected.

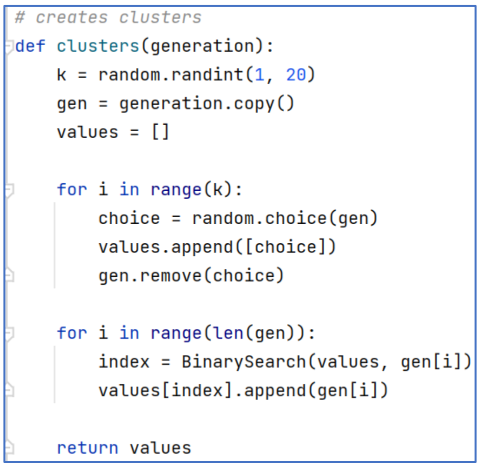


```python
# give probability
def probability(parent1, parent2, child1, child2):
    arr = [parent1.fitness(), parent2.fitness(), child1.fitness(), child2.fitness()]
    arr2 = [arr[i] / sum(arr) for i in range(4)]
    return arr2
```

Clustering:

The clustering algorithm is a bit different than niching and crowding:

- We don't automatically split the population into 20 groups – each generation, we randomly choose a number between 1 and 20.
- We than randomly choose k people that would serve as our medoids (center value for each of the groups).
- The remaining population gets split into niches – based on their closeness to the medoids. Each gene would join a group that "best represent" its value.
- We find the value that is closest to us using binary search.



```python
# creates clusters
def clusters(generation):
    k = random.randint(1, 20)
    gen = generation.copy()
    values = []

    for i in range(k):
        choice = random.choice(gen)
        values.append([choice])
        gen.remove(choice)

    for i in range(len(gen)):
        index = BinarySearch(values, gen[i])
        values[index].append(gen[i])

    return values
```

- After that we use the same methods we used in niching – only now the number of groups changes constantly for each generation.

Comparisons Bin Packing:

We compared those 3 methods using the bin packing problem. We used the same parameters for each of them: 500 generations, 500 genes per generation, THM mutation (we saw it was the winner for bins), RWS for parent selection, PMX and scramble for mutation.

Conclusions:

Partitions:

- Niching and crowding always preserved 20 groups, while clustering showed different variants for every generation.
- When number of partitions was 1 or 2 the population behaved the same as with the original genetic algorithm. Once you reach 3 or more groups, you see a lot more range in fitness (because the focus of smaller groups allow us to use minimal changes in fitness between the parents to our advantage).

Genetic diversity:

- Niching had the most genetic diversity, as mention above as one of the algorithm's goals.
- Clustering had some more diversity than the crowding (thanks to the differences in fitness discussed above), but that's OK because the difference was very small – not like niching, which had a huge range.
- Crowding kept a very steady number across evolution, with slight variation towards the end – where you have to converge to a single solution.

Number of solutions:

- Because he maintained a steady genetic diversity, crowding only had one gene tat truly reached our solution.

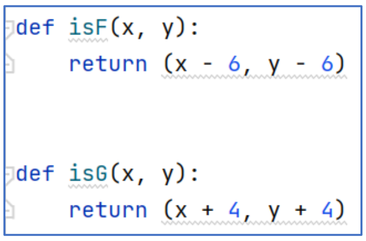
- Niching and clustering, however, had multiple genes.
- When number of groups for clustering was 1 or 2, niching had multiple solution while clustering only had 2. Once umber of groups reached 3 or more – clustering had about 10 solutions more than niching.

Number of generations to reach solutions: all of them reached the solution on the last generation.

Island Model:

The last part of the program focuses on the island model.

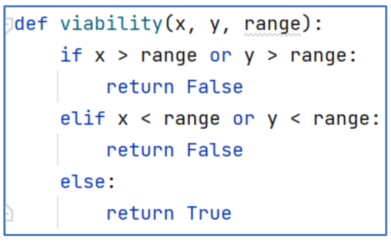
- We calculated the ranges of X and Y for each of the islands.
- F has an island with center at (0, 0) and radius 3. That means that $(-3 \leq x, y \leq 3)$
- G has an island with center at (5, 5) and radius 2. That means that $(3 \leq x, y \leq 7)$
- Supposedly, genes from each island don't ever meet because the circles are osculating. But, we could use a mathematical formula in order to "shift" the values between the 2 islands:
  - In order to move values from F to G we "shift" right (add a positive value).
  - In order to move values from G to F we "shift" left (add a negative value).



```
def isF(x, y):
    return (x - 6, y - 6)


def isG(x, y):
    return (x + 4, y + 4)
```

After the initial populations are established, we must decide a migration policy – hyper parameters:

- Migration rate: we wanted to keep the island as flexible as possible, and since migrating is supposed to help the island (bring new quality to a decaying population) – we decided to allow migration in every generation.
- Migration selection: the 10% that are the elite team are the ones who would migrate – they have the best qualities and could really contribute.
- Replacement selection: the elite team we just mention would be replaced with the other elite team from the other island. That way its equal change for the both of them
- Condition of viability: after we shifted the values, we would check viability of gene.



```
def viability(x, y, range):
    if x > range or y > range:
        return False
    elif x < range or y < range:
        return False
    else:
        return True
```

Conclusions:

- Since both elite teams were replaced with each other – we saw a major shift in diversity each time (we had a lot of new blood).
- Our optimal solutions were those who had been originally close to the point where the circles osculate.