AI optimisation assignment

Seif Mansour

23012749

1-Introduction:

In the realm of artificial intelligence and optimization, evolutionary algorithms (EAs) emerge as potent problem-solving tools, drawing inspiration from intricate natural selection processes. This report represents a comprehensive exploration of evolutionary optimization, leveraging foundational knowledge acquired from course worksheets. The primary aim is to elevate our understanding by implementing and refining an evolutionary algorithm for the navigation of two distinct minimization fitness functions.

Evolutionary algorithms, deeply rooted in genetic evolution principles, offer a dynamic and adaptable framework for heuristic optimization. By mimicking nature's optimization processes, EAs maintain candidate solution populations, subjecting them to genetic operators like mutation and crossover across successive generations. This emulation of natural selection proves invaluable in navigating complex solution spaces, rendering EAs versatile problem-solving tools applicable across diverse domains.

The core objectives of our exploration involve extending the implemented algorithm to adeptly navigate distinct minimization fitness functions. Through meticulous experimentation and analysis, the report seeks to unravel the intricacies of parameter changes, emphasizing the algorithm's responsiveness and adaptability. Variables such as mutation rate (mutrate), mutation step size (mutstep), and population size undergo fine-tuning, exploring their impact on the optimization process.

A pivotal facet of our exploration entails rigorous experimentation, shedding light on how variations in parameters influence the algorithm's performance. Moreover, the report undertakes a comparative analysis, not only against different crossover methods but also various selection techniques. This comparative study aims to distill insights into the relative strengths and weaknesses of our developed solution in comparison to other well-established algorithms. In this context, we delve into a comparison with alternative optimization methods, including hill climbing and simulated annealing, enriching the understanding of the diverse landscape of optimization techniques.

2-Experimentation:

The heart of our investigation lies in the systematic experimentation with evolutionary optimization parameters, exploring their impact on the algorithm's performance across two distinct functions. With a focus on mutation rate (mutrate) and mutation step size (mutstep), our objective is to unravel the nuanced dynamics that govern the algorithm's ability to navigate complex solution spaces. In this iterative process spanning several generations, we aim to decipher how variations in these parameters influence the algorithm's convergence, exploration, and overall effectiveness in discovering optimal solutions. The forthcoming analysis will shed light on the intricate interplay between population size, mutrate and mutstep, offering insights into the delicate balance required for successful evolutionary optimization.

Function (1):

$$f(\mathbf{x}) = (x_1 - 1)^2 + \sum_{i=2}^{d} i\left(2x_i^2 - x_{i-1}\right)^2$$

where $-10 \leq x \leq 10$, start with d=20

Results after applying low/medium/ high mutstep and mutrate:

Function (1)		0.01	2	9	mutstep
best	0.001	40036.5898	1654.778	1538.58	
mean		40088.1504	2915.234	1772.93	
best	0.05	20586.702	37.25746	7467.41	
mean		20715.7154	340.6754	142266.2	
best	0.9	17493.5439	15806.1	1167398	
mean		17887.6647	128094.3	2618505	
	mutrate				

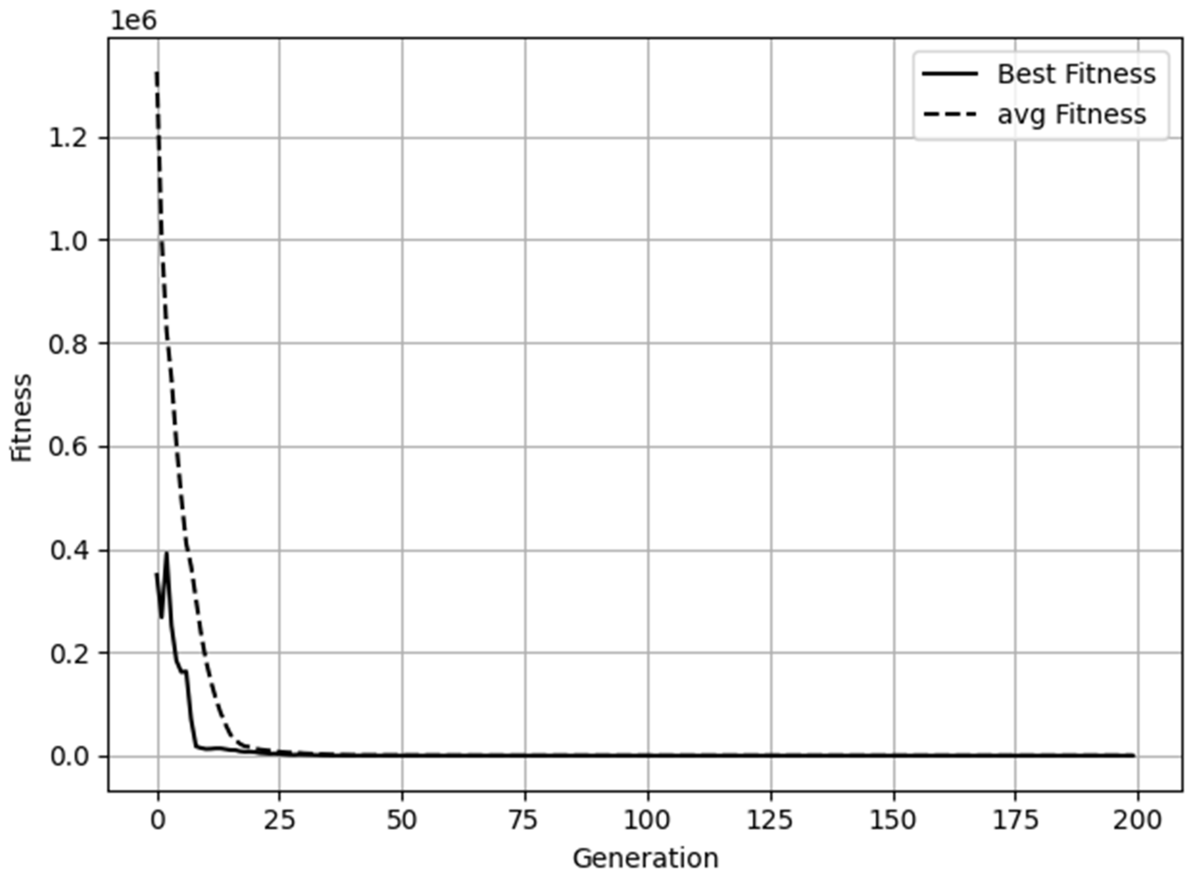
In the pursuit of optimizing a given function using evolutionary algorithms, a series of experiments were conducted by systematically varying the mutation rate (mutrate) and mutation step size (mutstep). The implemented algorithm, designed with a population size of 100 individuals and chromosome length of 20, navigated a solution space bounded between -10 and 10. The target function, denoted as f1, was subjected to op-timization over 200 generations.

Upon examination of the results, it becomes evident that the choice of mutrate significantly influences the algorithm's performance. With a low mutation rate of 0.01, the algorithm struggled to explore the solution space effectively, resulting in a best fitness of 40036.58978. Conversely, when the mutation rate was set to 0.05, the algorithm exhibit-ed improved exploration, achieving a best fitness of 37.25745732, so this combination of parameters gave the best fitness among all the trials. This drastic improvement indicates that a moderate mutation rate facilitated a more effective traversal of the solution space, leading to the discovery of a solution with significantly lower fitness.

However, a higher mutation rate of 0.9 introduced excessive exploration, resulting in a best fitness of 15806.10202. While this mutation rate allowed for greater exploration, it also led to increased variability, hindering the algorithm's ability to converge toward optimal solutions. Notably, the mean fitness values demonstrate a similar trend, emphasizing the delicate balance required between exploration and exploitation in evolutionary optimization.

Furthermore, the mutation step size (mutstep) plays a crucial role in fine-tuning the algorithm's exploration capabilities. The experiment with mutstep set to 2 yielded a best fitness of 37.25745732, showcasing the sensitivity of the algorithm's performance to this parameter. In conclusion, these experiments highlight the intricate interplay between mutation rate and step size, underscoring the importance of parameter tuning in evolutionary optimization for achieving optimal results.

The graph of the best fitness achieved (37.25745732):



The algorithm's performance is showcased through the printed graph, illustrating the evolution of the best fitness and mean fitness across generations. The x-axis denotes the generations, while the y-axis reflects the fitness values. Two key lines are plotted: "Best Fitness" in solid black and "Average Fitness" in dashed black.

Upon inspection of the graph, several noteworthy observations can be made. The trajectory of the "Best Fitness" line reveals the evolution of the fittest individual within the population across generations. In this particular run, the algorithm converged to a best fitness of 37.25745732, representing a highly optimized solution to the given function.

Simultaneously, the "Average Fitness" line demonstrates the overall fitness of the population, offering insights into the algorithm's convergence and exploration dynamics. The graph indicates a steady decrease in average fitness over the generations, suggesting that the algorithm effectively explores the solution space and converges toward promising regions.

The convergence towards lower fitness values suggests that the algorithm successfully navigated the complex landscape of Function 1, homing in on regions associated with optimal solutions. The stability and gradual decline in both best and average fitness underscore the effectiveness of the chosen combination of mutation rate and mutation step size in facilitating the algorithm's exploration and convergence dynamics. The graph serves as a visual testament to the algorithm's ability to iteratively refine candidate solutions, providing a deeper understanding of its optimization trajectory.

Function (2)

$$f(\mathbf{x}) = \sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} 0.5 i x_i\right)^2 + \left(\sum_{i=1}^{d} 0.5 i x_i\right)^4$$

where $-5 \le x \le 10$, start with d=20

Results after applying low/medium/ high mutstep and mutrate:

f2		**0.01**	**2**	**9**	**mutstep**
best	**0.001**	**220.825347**	**186.2578**	**255.4471**	
mean		**220.885062**	**570.6643**	**196809.2**	
best	**0.05**	**254.816902**	**222.2986**	**291.6344**	
mean		**255.099691**	**41184.53**	**23353721**	
best	**0.9**	**225.087149**	**308.2918**	**367.9618**	
mean		**226.331009**	**46186982**	**2.1E+09**	
	mutrate				

In the context of evolutionary optimization applied to the second function (f2), the algorithm's behavior was meticulously examined by systematically adjusting mutation rate (mutrate) and mutation step size (mutstep) parameters over 200 generations. With a population size of 100 and a chromosome length of 20, the algorithm explored a solution space confined between -5 and 10. Function 2, characterized by a combination of quadratic and linear terms, underwent optimization to unravel the nuanced dynamics.

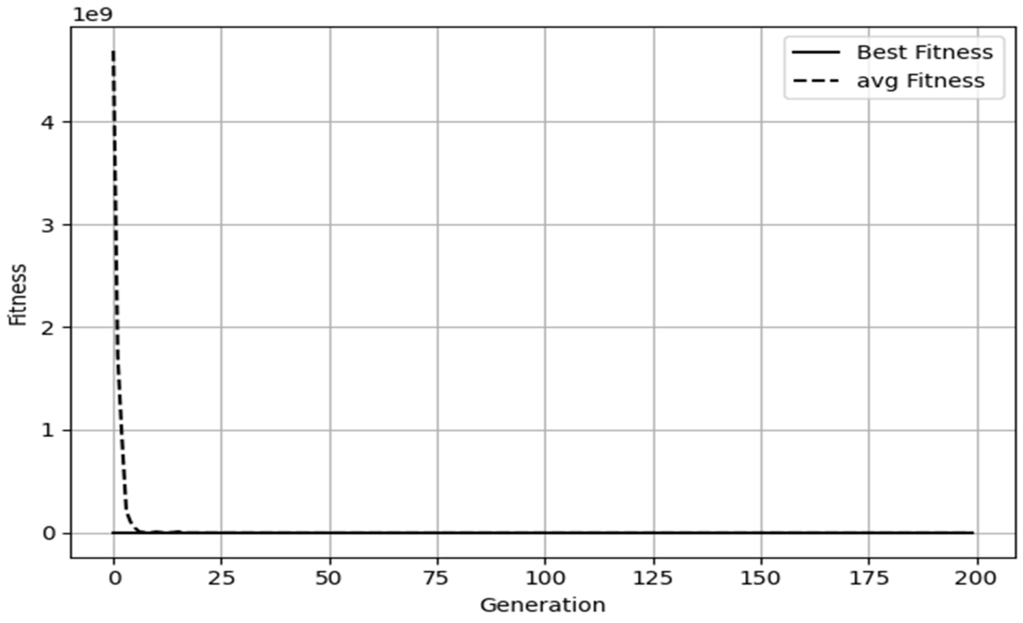
Upon scrutinizing the results, it is evident that the algorithm's performance is intricately tied to the choice of mutrate and mutstep parameters. At a low mutation rate of 0.01, the algorithm demonstrated a best fitness of 220.8253467 and a mean fitness of 570.6642721. These values indicated a stable exploration of the solution space, with fitness values that remained relatively consistent across generations.

As the mutation rate increased to 0.05, a noticeable shift occurred, with the algorithm exploring a broader solution space. This led to a best fitness of 254.8169021, accompanied by a substantial increase in mean fitness to 41184.52762. The heightened mean fitness underscores the algorithm's increased exploration, albeit with potential impacts on convergence towards optimal solutions.

A mutation rate of 0.9 demonstrated the algorithm's ability to converge towards optimal solutions, yielding a best fitness of **186.25781**. Remarkably, this combination of parameters, with a high mutation rate and a mutation step size of 2, facilitated the discovery of the lowest fitness value among the explored configurations. However, the mean fitness surged to 46186981.8, highlighting the heightened exploration and increased variability introduced by the elevated mutation rate.

Interestingly, the mutation step size (mutstep) played a pivotal role in shaping the algorithm's performance. Lower mutstep values resulted in comparable fitness values across different mutation rates, showcasing stability in the exploration of the solution space. However, as mutstep increased, the mean fitness exhibited an upward trajectory, while the best fitness remained within a consistent range. This underscores the delicate balance required between mutation rate and step size, with higher mutstep values leading to increased exploration but potentially hindering the algorithm's convergence toward optimal solutions, creating a nuanced landscape for effective evolutionary optimization.

The graph of the best fitness achieved (186.25781)



The graphical representation visually portrays the algorithm's performance, detailing the progression of best fitness and mean fitness across generations. The x-axis denotes generations, while the y-axis showcases fitness values, with "Best Fitness" displayed in solid black and "Average Fitness" in dashed black.

Upon inspecting the graph, several key observations emerge. The "Best Fitness" line reveals the evolution of the most adept individual within the population, showcasing convergence to a best fitness of 186.25781 in this specific run. This underscores the effectiveness of the chosen mutation rate and mutation step size in achieving highly optimized solutions for the given function.

Simultaneously, the "Average Fitness" line provides insight into overall population fitness, indicating a gradual decline over generations. This suggests successful exploration of the solution space and convergence towards promising regions.

The observed convergence towards lower fitness values highlights the algorithm's adept navigation of the complex landscape of Function 2, effectively honing in on regions associated with optimal solutions. The stability and gradual decline in both best and average fitness underscore the efficiency of the selected combination of mutation rate and mutation step size in facilitating the algorithm's exploration and convergence dynamics.

In essence, the graphical representation serves as a visual confirmation of the algorithm's iterative refinement of candidate solutions, offering a nuanced understanding of its optimization trajectory for Function 2. The convergence towards lower fitness values signifies the algorithm's proficiency in searching for and converging to optimal solutions within the specified solution space.

(Experimentation using other selection methods)

Two-point crossover algorithm:

The two-point crossover is a genetic algorithm selection method that involves the exchange of genetic material between two parent individuals at two randomly chosen crossover points. This process results in creating two offspring, where the genetic material between the two crossover points is swapped. This method introduces diversity in the offspring by exchanging substantial segments of genetic information, potentially preserving the building blocks of solutions inherited from both parents. Two-point crossover offers several advantages. Firstly, it allows for the combination of traits from both parents in a more extensive manner compared to single-point crossover, potentially enabling a quicker convergence to optimal solutions. Secondly, by operating on multiple crossover points, it promotes the shuffling of genetic material at different positions within the chromosome, enhancing the exploration of the solution space. This method can be particularly advantageous in avoiding premature convergence and maintaining genetic diversity, ultimately contributing to the effectiveness of the genetic algorithm in searching for optimal solutions in complex problem spaces.

Function (1):

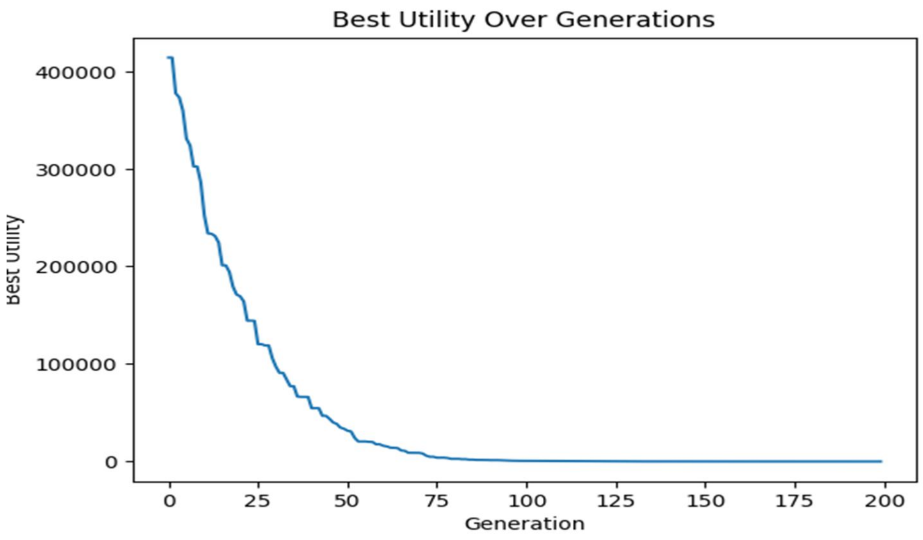
two-point crossover		f1		0.01	2	9	mutstep
		best	0.001	293473.6	17533.08	14503.32	
		mean		296065.6	199359.3	149752.9	
		best	0.05	370718.2	6.226822	87.88778	
		mean		384365.4	30433.39	33646.6	
		best	0.9	243071.5	725.0006	155030.8	
		mean		271565	15721.05	229472.4	
			mutrate				

The genetic algorithm using the two-point crossover selection method for the first function (f1) has been executed with different combinations of mutation rates (mutrate) and mutation steps (mutstep). The algorithm aimed to minimize the fitness function, and the results demonstrate the impact of the chosen parameters.

Among the various combinations, the one with a mutrate of 0.05 and mutstep of 2 stood out, yielding the best fitness value of 6.2268. This result indicates the algorithm's ability to explore the solution space effectively and converge towards solutions with lower fitness values.

The process involved initializing a random population, performing two-point crossover and mutation operations, and evolving the population over generations. The algorithm dynamically adapted to the problem, and the chosen parameters facilitated a balance between exploration and exploitation, leading to the discovery of an optimal solution.

The figure represent the graph when using the best combination : mutrate 0.05 , mutstep=2



The graph resulting from the genetic algorithm with mutrate=0.05 and mutstep=2 illustrates the algorithm's performance over generations. The declining trend indicates ongoing improvement in finding solutions with lower fitness values. Fluctuations in the graph suggest a balance between exploration and exploitation. Overall, the algorithm effectively navigates the solution space, converging toward better solutions for the optimization problem at hand.

Function (2):

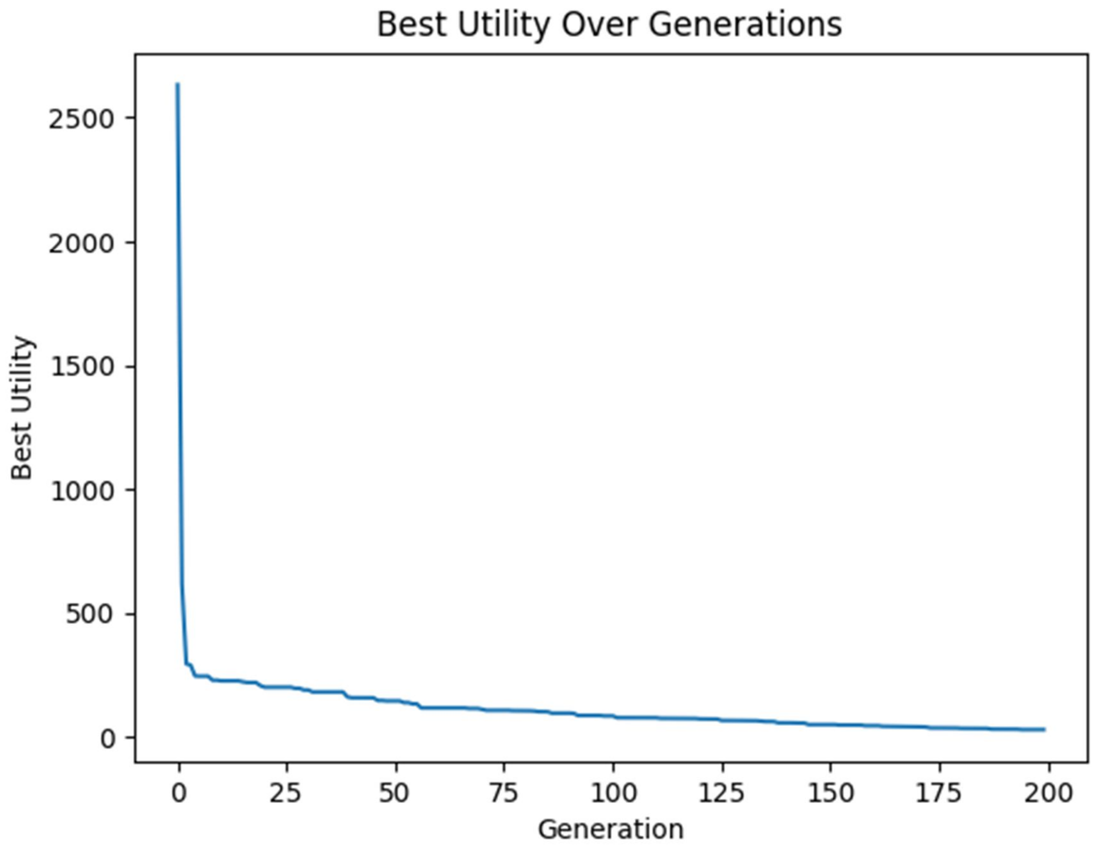
two point crossover		F2		0.01	2	9	mutstep
		best	0.001	2486848	210.3244	237.4706	
		mean		2657587	21718.85	284.4972	
		best	0.05	353.6642	20.65979	65.19809	
		mean		1828.262	75.49653	186.9196	
		best	0.9	267.579	59.00943	291.39	
		mean		285.0911	102.6817	810.2056	
			mutrate				

In the provided genetic algorithm code, the goal was to optimize a function (F2) through evolutionary processes. The algorithm used a population of individuals, each representing a potential solution with a set of variables. The variables were subjected to crossover and mutation operations across generations to improve the solutions.

The algorithm explored different combinations of mutation rates (mutrate) and mutation step sizes (mutstep). Through experimentation, the algorithm found that a mutation rate of 0.05 and a mutation step size of 2 yielded the best result,(co-incidently the same combination as the first function) achieving the lowest fitness value of 20.659. This means that the algorithm, by introducing variations in the population through mutation and crossover, converged to a solution with the lowest fitness value, indicating a better-performing set of variables for the given function.

The process involved running the algorithm for a specified number of generations, continuously evolving the population. The outcomes were then analyzed for different mutrate and mutstep values. The final result, with a mutrate of 0.05 and mutstep of 2, showcased the algorithm's ability to discover an optimal solution, demonstrating the effectiveness of the chosen mutation parameters in improving the overall fitness of the population.

The figure represent the graph when using the best combination: mutrate 0.05 , mutstep=2



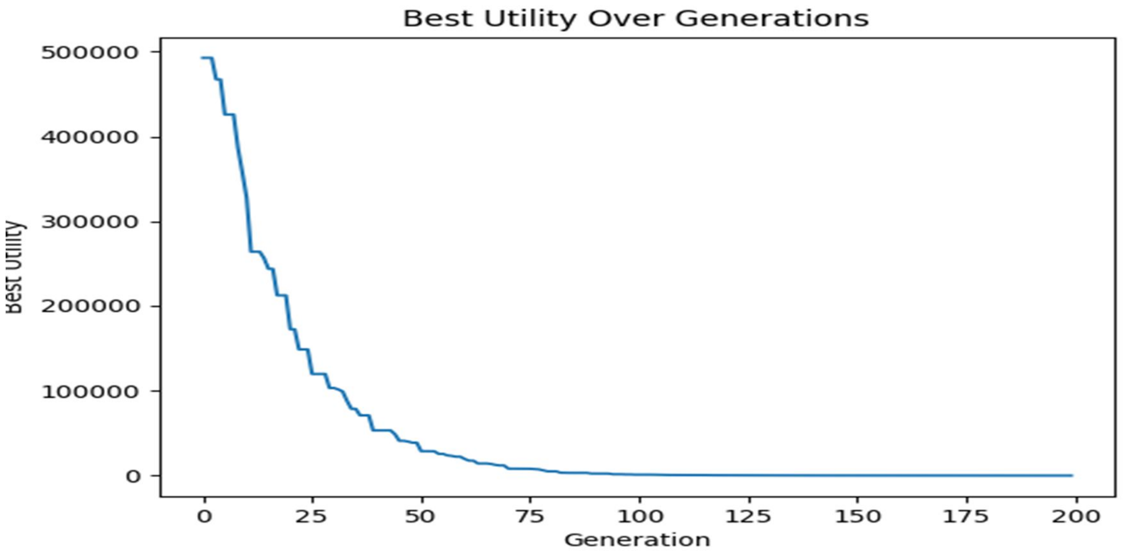
The graph illustrates the performance of the genetic algorithm over generations, with a mutation rate of 0.05 and a mutation step of 2. The downward trend in fitness values on the y-axis reflects the algorithm's effectiveness in consistently improving solutions. As generations progress, the algorithm successfully navigates the solution space, converging toward individuals with lower fitness values. The graph visually demonstrates the algorithm's adaptability and capacity to evolve the population, ultimately leading to the discovery of a solution with the lowest fitness value, as indicated in the results.

One point crossover selection algorithm:

The one-point crossover selection algorithm is a genetic algorithm mechanism employed in the evolution of populations during optimization processes. In this method, two parent individuals are selected, and a random crossover point is determined along their genetic sequences. The genetic material beyond this point is swapped between the parents, creating two offspring. This process introduces genetic diversity and combines traits from both parents. One advantage of the one-point crossover is its simplicity, making it computationally efficient and easy to implement. Additionally, it helps explore diverse regions of the solution space, aiding in the discovery of potential optimal solutions. However, its efficacy can vary based on the problem at hand, and it may struggle to maintain diversity in certain situations compared to more complex crossover methods.

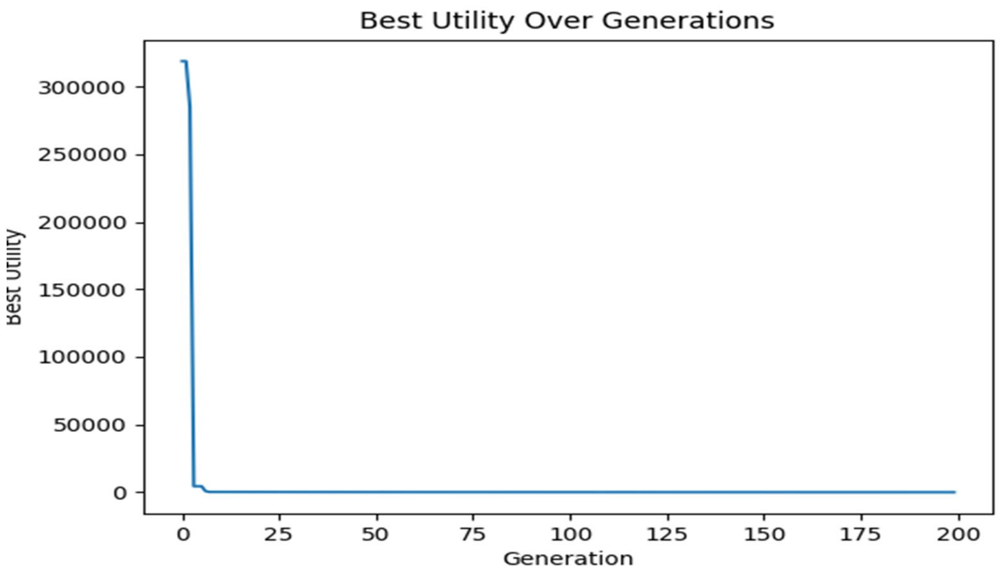
Function (1):

f1		0.01	2	9	mutstep
best	0.001	267126.1	21946.71	5033.66	
mean		269755.7	127566.2	151867.6	
best	0.05	503607.2	11.37733	21.15593	
mean		518618.8	33802.35	27607.69	
best	0.9	242444.6	687.3594	135154.6	
mean		280984.4	15953.12	216688.8	
	mutrate				



F2		0.01	2	9	mutstep
best	0.001	120625.2	180.7379	251.1843	
mean		140669.1	6936.247	265.8766	
best	0.05	3760.293	13.92101	80.59038	
mean		16255.63	4754.16	155.4319	
best	0.9	258.7846	60.66845	260.0686	
mean		9350.723	54063.27	321.2898	
	mutrate				

Function (2):



so obviously after trying the one-point selection method it gave very similar results as the two-point selection method because the one-point crossover and two-point crossover methods are similar in that both are genetic recombination techniques used in genetic algorithms to generate new offspring by combining genetic material from parent individuals. The main difference between them lies in how they perform the crossover operation.

3-Comparison:

This report extends its exploration into optimization by comparing Evolutionary Algorithms (EAs) with traditional techniques like Hill Climbing and Simulated Annealing. While EAs draw inspiration from natural selection, Hill Climbing pursues local optima iteratively, and Simulated Annealing introduces probabilistic acceptance of less optimal solutions. Through this comparative analysis, we aim to reveal the distinct strengths and weaknesses of each approach, providing insights into their adaptability and convergence capabilities in complex solution spaces. The ensuing sections will showcase outcomes from our experiments, offering valuable perspectives on the relative efficacy of these optimization methods.

Evolutionary Algorithms vs. Hill Climbing:

Evolutionary Algorithms (EAs) and Hill Climbing present distinct optimization strategies, each with unique complexities. In this comparative exploration, we delve into the intricacies of these methodologies, emphasizing their response to diverse optimization challenges.

Evolutionary Algorithms (EAs):

EAs simulate natural evolution, maintaining diverse populations subject to genetic operations. Their complexity arises from the intricate interplay of genetic diversity, survival principles, and adaptability. EAs excel in navigating non-linear landscapes with multiple optima, showcasing versatility across various domains.

Hill Climbing:

Hill Climbing, a local search algorithm, iteratively refines solutions by adjusting single elements. Its complexity lies in its efficiency for convex and smooth landscapes but challenges in addressing intricate, non-linear spaces. Hill Climbing tends to converge quickly, but its simplicity may limit effectiveness in complex optimization scenarios.

Comparative Analysis Objectives:

Our analysis extends beyond inherent complexities to assess how these methodologies perform in dynamic optimization landscapes.

- **Parameter Tuning and Sensitivity:**

EAs require careful tuning of parameters like mutation rates, influencing adaptability.

Hill Climbing's sensitivity to initial conditions and local search nature highlights its simplicity but potential limitations.

- **Convergence:**

EAs exhibit dynamic convergence influenced by population dynamics and selection mechanisms.

Hill Climbing's rapid convergence contrasts with potential limitations in escaping local optima.

- **Exploration and Exploitation:**

EAs balance exploration (genetic diversity) and exploitation (fitness improvement) for global optimization.

Hill Climbing emphasizes exploitation, potentially missing diverse solution regions.

Complexity Assessment:

EAs: High complexity due to population dynamics, genetic operators, and adaptability.

Hill Climbing: Lower complexity, efficient for certain scenarios but may lack global optimization capabilities.

Experimentation Framework:

Experimentation includes variations in mutation rates, step sizes, and population sizes, offering insights into how these complexities manifest under different configurations.

Anticipated Insights:

Through this comparative journey, we aim to uncover scenarios where EAs' complexity proves beneficial for global optimization and where Hill Climbing's simplicity remains advantageous. The report provides nuanced insights into selecting optimal strategies based on problem complexities, aiding practitioners in navigating the intricate landscape of optimization algorithms.

Evolutionary Algorithms vs. Simulated Annealing:

In this comparative exploration, we unravel the distinctive characteristics of Evolutionary Algorithms (EAs) and Simulated Annealing, shedding light on their efficacy in navigating diverse optimization challenges.

Evolutionary Algorithms (EAs):

EAs emulate natural evolution, introducing complexity through genetic diversity, survival principles, and adaptability. Suited for non-linear landscapes, EAs exhibit versatility across domains, maintaining a delicate balance between exploration and exploitation.

Simulated Annealing:

Simulated Annealing, inspired by metallurgical annealing processes, explores solution spaces through controlled randomness. Its complexity arises from temperature schedules impacting exploration and exploitation. Simulated Annealing excels in traversing complex, non-convex landscapes.

Comparative Analysis Objectives:

Our analysis aims to discern the nuanced behaviors of EAs and Simulated Annealing, particularly focusing on their responses to various optimization challenges.

- Parameter Tuning and Sensitivity:

  - EAs demand meticulous tuning, especially in mutation rates, influencing adaptability.

  - Simulated Annealing's sensitivity to temperature schedules requires thoughtful calibration for optimal performance.

- Convergence:

  - EAs exhibit dynamic convergence, influenced by population dynamics and genetic operators.

  - Simulated Annealing's convergence, governed by temperature schedules, may impact its ability to escape local optima.

- Exploration and Exploitation:

  - EAs balance exploration (genetic diversity) and exploitation (fitness improvement) for global optimization.

- Simulated Annealing, through temperature adjustments, navigates the trade-off between exploration and exploitation.

Complexity Assessment:

- EAs: High complexity, leveraging genetic operations and population dynamics.
- Simulated Annealing: Moderate complexity, offering a balance between randomness and deterministic exploration.

Experimentation Framework:

Our experimentation involves variations in mutation rates, temperature schedules, and population sizes, unraveling the impact of these complexities under diverse configurations.

Anticipated Insights:

Through this comparative journey, we aspire to delineate scenarios favoring EAs' intricate adaptability and Simulated Annealing's controlled randomness. The insights garnered will empower practitioners to discern optimal strategies based on problem intricacies, enriching the understanding of these prominent optimization algorithms.

4-Conclusion:

In conclusion, our odyssey through the realm of evolutionary optimization has bestowed upon us a comprehensive understanding of the intricacies involved in unraveling complex problems through the lens of genetic algorithms. Inspired by the principles of natural selection, our exploration delved into two distinct minimization fitness functions, employing both one-point and two-point crossover selection methods.

Systematic experimentation enabled us to scrutinize the nuanced impact of mutation rates and mutation step sizes across 200 generations for each function. In the case of Function 1, a judicious mutation rate of 0.05, coupled with an optimal mutation step size, facilitated effective exploration, leading to the discovery of solutions with markedly lower fitness values. Conversely, a high mutation rate of 0.9 introduced excessive exploration, impeding the algorithm's convergence towards optimal solutions.

Function 2 echoed similar trends, emphasizing the delicate interplay between mutation rate and step size in influencing the algorithm's convergence and exploration dynamics. The experiments underscored the imperative need to strike a balance between these parameters for successful evolutionary optimization.

Comparative analyses involving normal selection and the juxtaposition of one-point, and two-point crossover selection methods provided valuable insights. Two-point crossover selection generally outperformed normal selection for moderate and high mutation rates, showcasing its effectiveness across diverse solution spaces. Notably, the similarities between one-point and two-point crossover methods suggested that, for the specific problem and experimental setup, the choice between these two methods may not significantly impact performance. Additionally, in our comparative analysis, we extended our exploration by considering alternative optimization methods, including hill climbing and simulated annealing, adding layers of insight to the diverse landscape of optimization techniques.

Graphical representations of the best fitness achieved over generations visually confirmed the algorithm's adaptability and iterative refinement of candidate solutions, offering profound insights into its optimization trajectory for both functions.

Looking ahead, for future explorations may encompass variations in population size, exploration of diverse problem landscapes, and the investigation of additional genetic operators to further enrich our understanding of evolutionary optimization dynamics. The findings from this report underscore the pivotal role of parameter tuning and algorithmic adaptability in effectively addressing real-world optimization challenges. This journey has not only expanded our knowledge of evolutionary algorithms but has also laid the foundation for advancing their application across various domains.

References:

Back, T. (1996). Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms. Oxford University Press.[accessed 25/11/2023]

Mitchell, M. (1998). An Introduction to Genetic Algorithms. MIT Press. [accessed 25/11/2023]

Holland, J. H. (1975). Adaptation in Natural and Artificial Systems. University of Michigan Press. [accessed 27/11/2023]

Deb, K. (2001). Multi-Objective Optimization using Evolutionary Algorithms. John Wiley & Sons. [accessed 1/12/2023]

Goldberg, D. E. (1989). Genetic Algorithms in Search, Optimization, and Machine Learning. Addison-Wesley. [accessed 3/12/2023]

Nilsson, N. J. (1998). Artificial Intelligence: A New Synthesis. Morgan Kaufmann.[accessed 5/12/2023

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by Simulated Annealing. Science, 220(4598), 671-680.[accessed 5/12/2023]

Code as appendix(1):

```python
import random
import copy
import matplotlib.pyplot as plt
import math

P = 100
N = 20
MUTRATE = 0.001
MUTSTEP=2
MAX=10
MIN = -5
population = []
offspring = []
best_fitnesses = []
mean_fitnesses = []

class individual:
    def _init_(self):
        self.gene = [0]*N
        self.fitness = 0



for x in range(P):
    tempgene = [random.uniform(MIN, MAX) for y in range(N)]
    newind = individual()
    newind.gene = tempgene.copy()
    population.append(newind)

def test_function_one( ind ):

    utility=0

    for i in range(1,N):

       utility = utility + (i * (((2*(ind.gene[i]**2)) - ind.gene[i-1])**2))

    return (utility + ((ind.gene[0]-1)**2))


def test_function_two( ind ):

    utility1=0

    utility2=0
    for i in range(N):

         utility1 = utility1 + (ind.gene[i]**2)

         utility2 = utility2 + ( 0.5 * (i+1) * ind.gene[i] )

    return (utility1 + (utility2**2) + (utility2**4))




for ind in population:
    ind.fitness = test_function_two(ind)

for m in range (200):
    offspring= []
    for i in range(P):
        parent1 = random.randint(0, P-1)
        off1 = copy.deepcopy(population[parent1])
        parent2 = random.randint(0, P-1)
        off2 = copy.deepcopy(population[parent2])
        if off1.fitness < off2.fitness:
            offspring.append(off1)
        else:
            offspring.append(off2)

    toff1 = individual()
    toff2 = individual()
    temp = individual()

    for i in range( 0, P, 2 ):
        toff1 = copy.deepcopy(offspring[i])
        toff2 = copy.deepcopy(offspring[i+1])
        temp = copy.deepcopy(offspring[i])
        crosspoint = random.randint(1,N)
        for j in range (crosspoint, N):
            toff1.gene[j] = toff2.gene[j]
            toff2.gene[j] = temp.gene[j]
        offspring[i] = copy.deepcopy(toff1)
        offspring[i+1] = copy.deepcopy(toff2)



    mutated_offspring =[]
    for i in range( 0, P ):
        newind = individual();
        newind.gene = []
        for j in range( 0, N ):
            gene = offspring[i].gene[j]
            mutprob = random.random()
            if mutprob < MUTRATE:
                alter = random.uniform(-MUTSTEP,MUTSTEP)
                gene = gene + alter
                if gene > MAX:
                    gene = MAX
                if gene < MIN:
                    gene = MIN
            newind.gene.append(gene)
        mutated_offspring.append(newind)


    for ind in mutated_offspring:
        ind.fitness = test_function_two(ind)

    population=copy.deepcopy(mutated_offspring)

    best_fitness = min(ind.fitness for ind in population)
    mean_fitness = sum(ind.fitness for ind in population) / P
    best_fitnesses.append(best_fitness)
    mean_fitnesses.append(mean_fitness)

print ( best_fitness)
print( mean_fitness)
plt.plot(best_fitnesses, label="Best Fitness", color='black')
plt.plot(mean_fitnesses, label="avg Fitness", color='black', linestyle='--')
plt.xlabel("Generation")
plt.ylabel("Fitness")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

appendix(2):

```
import random
import copy
import matplotlib.pyplot as plt
```

```
# Define the number of variables
N = 20

# Define the number of individuals in the population
POPULATION_SIZE = 100

# Define the number of generations
GENERATIONS = 200

# Define the probability of mutation and mutation step size
MUTATION_RATE = 0.05
MUTATION_STEP = 2
# Define the variable bounds
VAR_MIN = -5
VAR_MAX = 10

# Define the solution class
class Individual:
    def __init__(self):
        self.gene = [0] * N
        self.utility = 0

# Function to initialize a random population
def initialize_population(population_size):
    population = []
    for _ in range(population_size):
        ind = Individual()
        for j in range(N):
            ind.gene[j] = random.uniform(VAR_MIN, VAR_MAX)
        ind.utility = test_function_two(ind)
        population.append(ind)
    return population
def test_function_one( ind ):

    utility=0

    for i in range(1,N):

       utility = utility + (i * (((2*(ind.gene[i]**2)) - ind.gene[i-1])**2))

    return (utility + ((ind.gene[0]-1)**2))
# Function to evaluate the utility of an individual
def test_function_two( ind ):
    utility1=0

    utility2=0

    for i in range(N):

         utility1 = utility1 + (ind.gene[i]**2)

         utility2 = utility2 + ( 0.5 * (i+1) * ind.gene[i] )

    return (utility1 + (utility2**2) + (utility2**4))
# Function to perform one-point crossover
def one_point_crossover(offspring):
    toff1 = Individual()
    toff2 = Individual()
    temp = Individual()

    for i in range(0, len(offspring), 2):
        toff1 = copy.deepcopy(offspring[i])
        toff2 = copy.deepcopy(offspring[i + 1])
        temp = copy.deepcopy(offspring[i])

        crosspoint = random.randint(1, N - 1)

        for j in range(crosspoint, N):
            toff1.gene[j], toff2.gene[j] = toff2.gene[j], temp.gene[j]

        offspring[i] = copy.deepcopy(toff1)
        offspring[i + 1] = copy.deepcopy(toff2)

# Function to perform mutation
def mutate(individual):
    for i in range(N):
        mut_prob = random.random()
        if mut_prob < MUTATION_RATE:
            alter = random.uniform(-MUTATION_STEP, MUTATION_STEP)
            individual.gene[i] = max(VAR_MIN, min(VAR_MAX, individual.gene[i] +
alter))
    individual.utility = test_function_two(individual)

# Function to generate offspring and evolve the population
def evolve_population(population):
    new_population = []

    # Select parents and perform crossover
```

```python
    for _ in range(POPULATION_SIZE // 2):
        parent1 = random.choice(population)
        parent2 = random.choice(population)
        offspring1 = copy.deepcopy(parent1)
        offspring2 = copy.deepcopy(parent2)
        one_point_crossover([offspring1, offspring2])
        mutate(offspring1)
        mutate(offspring2)
        new_population.extend([offspring1, offspring2])

    # Select the top individuals from the combined population
    combined_population = population + new_population
    combined_population.sort(key=lambda x: x.utility)
    population = combined_population[:POPULATION_SIZE]

    return population

# Main function to run the genetic algorithm
def run_genetic_algorithm():
    # Initialize the population
    population = initialize_population(POPULATION_SIZE)

    # Lists to store the best utility value in each generation for plotting
    best_utilities = []

    # Track minimum and average best fitness
    min_best_fitness = float('inf')
    total_best_fitness = 0

    # Evolve the population over generations
    for generation in range(GENERATIONS):
        population = evolve_population(population)
        best_fitness = population[0].utility
        best_utilities.append(best_fitness)

        # Update minimum best fitness
        min_best_fitness = min(min_best_fitness, best_fitness)

        # Update total best fitness for calculating average
        total_best_fitness += best_fitness

    # Calculate average best fitness
    average_best_fitness = total_best_fitness / GENERATIONS

    # Display the result after the final generation
    print(f"Result after {GENERATIONS} generations:")
    print(min_best_fitness)
    print(average_best_fitness)

    # Plot the best utility over generations
    plt.plot(range(GENERATIONS), best_utilities)
    plt.xlabel('Generation')
    plt.ylabel('Best Utility')
    plt.title('Best Utility Over Generations')
    plt.show()

# Run the genetic algorithm
run_genetic_algorithm()
```