1 -) CODE

In line 165 we are creating initial population using uniform random generator.

Then, in a for loop;

- in line 172 we are restoring infeasible solutions.
- in line 175 we are analyzing the generation.
    - update best solution if necessary
    - calculate the average of the solutions in the generation
    - calculate worst solution in the generation
- in line 180 we are calculation fitness values of each solution in the population.
- in line 183 we are creating a mating pool according to calculated fitness values and using binary tournament method.
- in line 186 we are shuffling the mating pool.
- in line 189 we are applying one point crossover operation.
- in line 192 we are applying mutation operation.

```
163 # create initial population with uniform randoms
164 population, fitness = [], []
165 for i in range(population_size):
166     random_solution = ''.join(['1' if k>=0.5 else '0' for k in np.random.uniform(low=0.0, high=1.0, size=no_of_nodes)])
167     population.append(random_solution)
168
169 # Start genetic algorithm
170 for i in range(no_of_generations):
171     # restore infeasible solutions
172     population = restore_infeasible_solutions(population, population_size, graph_matrix, no_of_nodes)
173
174     # update best solution
175     best_solution, average_solution, gen_worst_solution = analyze_population(population, population_size, node_weights, best_
176     average_solutions.append(average_solution)
177     print("Generation: {}, Average Solution: {}, Best Solution: {}".format(i, average_solution, best_solution))
178
179     # update fitness values
180     fitness = update_fitness(population, population_size, node_weights, no_of_nodes, gen_worst_solution)
181
182     # create mating pool
183     population = create_mating_pool(population, population_size, fitness)
184
185     # shuffle the population
186     population = random.sample(population, population_size)
187
188     # apply crossover
189     population = apply_crossoveer(population, population_size, crossover_prob, no_of_nodes)
190
191     # apply mutation
192     population = apply_mutation(population, population_size, mutation_prob, no_of_nodes)
```

1.1 -) restore_infeasible_solutions

In this function, firstly, we are checking the feasibility of the solution. If it is not feasible, we are picking a ‘0’ uniform randomly, and we are flipping the selected bit. We are again checking the new solution, if it is not feasible we are applying the same process again until we get a feasible solution.

```python
def restore_infeasible_solutions(population, population_size, graph_matrix, no_of_nodes):
    def check_feasiblity(solution, graph_matrix, no_of_nodes):
        solution = list(solution)
        reachable = deepcopy(graph_matrix)
        false_row = [False]*no_of_nodes
        for i in range(no_of_nodes):
            if solution[i] == '0':
                continue
            for j in range(no_of_nodes):
                if graph_matrix[i][j]:
                    reachable[j] = false_row
                    reachable[:, j] = [False]

        return reachable.any() == False

    for k in range(population_size):
        solution = population[k]
        # if solution is feasible
        while (not check_feasiblity(solution, graph_matrix, no_of_nodes)):
            # if solution is not feasible
            solution = list(solution)
            _, counts = np.unique(solution, return_counts=True) # counts zeros in the string
            flip_bit = int(counts[0]*np.random.uniform(low=0.0, high=1.0, size=1))
            for i in range(no_of_nodes):
                if (solution[i] == '0'):
                    flip_bit = flip_bit - 1
                if (flip_bit < 0):
                    solution[i] = '1'
                    break
            population[k] = ''.join(solution)

    return population
```

1.2 -) update_fitness

In this function, we are calculating each solution’s fitness values for parent selection method. For parent selection, we are generating two uniform random variables and taking the solution which has greater fitness value.

```python
def update_fitness(population, population_size, node_weights, no_of_nodes, gen_worst_solution):
    # initialize fitness values
    fitness = []
    for i in range(population_size):
        solution = list(population[i])
        solution_cost = 0
        for j in range(no_of_nodes):
            if(solution[j] == '1'):
                solution_cost += node_weights[j]
        fitness.append((1+gen_worst_solution-solution_cost)**2)

    return fitness

def create_mating_pool(population, population_size, fitness):
    mating_pool = []
    while(len(mating_pool) < population_size):
        candidate0 = int(np.random.uniform(low=0.0, high=1.0, size=1)*population_size)
        candidate1 = int(np.random.uniform(low=0.0, high=1.0, size=1)*population_size)
        if(fitness[candidate0] > fitness[candidate1]):
            mating_pool.append(population[candidate0])
        else:
            mating_pool.append(population[candidate1])
    return mating_pool
```

1.3 -) apply_crossover & apply_mutation

For apply_crossover, we are generating a uniform random number. If the number is less than the crossover probability value, we are applying 1-point crossover operation.

```
def apply_crossoveer(population, population_size, crossover_prob, no_of_nodes):
    for i in range(0,population_size-1,2):
        crossover_point = int(no_of_nodes*np.random.uniform(low=0.0, high=1.0, size=1))
        if (crossover_prob < np.random.uniform(low=0.0, high=1.0, size=1)):
            continue
        # apply crossover
        population[i], population[i+1] = population[i+1][0:crossover_point] + population[i][crossover_point:], \
                                        population[i][0:crossover_point] + population[i+1][crossover_point:],
    return population
```

For apply_mutation, we are generating a uniform random number for each bit of the solution. If the number is less than the mutation probability value, we are flipping the bit.

```
def apply_mutation(population, population_size, mutation_prob, no_of_nodes):
    for i in range(population_size):
        solution = list(population[i])
        for j in range(no_of_nodes):
            if (mutation_prob < np.random.uniform(low=0.0, high=1.0, size=1)):
                continue
            if solution[j] == '0':
                solution[j] = '1'
            else:
                solution[j] = '0'
        population[i] = ''.join(solution)
    return population
```

2-) OUTPUTS

Best results for each configuration and each file is shown tables below.

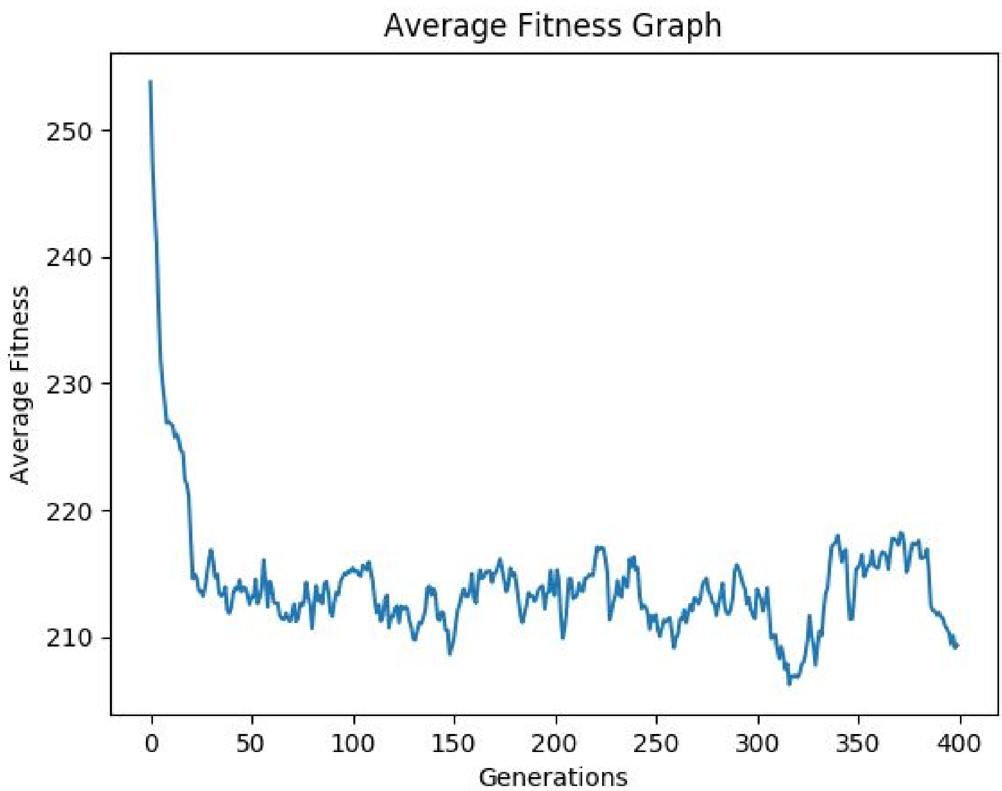
003.txt					
	Crossover Pr.	0.5		0.9	
# Generations	Pop. Size \ Mut. Pr.	1/n	0.05	1/n	0.05
100	100	126.80	172.23	113.77	170.29
	200	113.93	163.43	96.84	170.89
400	100	37.33	172.72	34.77	161.79
	200	36.71	170.43	29.91	168.20

015.txt					
	Crossover Pr.	0.5		0.9	
# Generations	Pop. Size \ Mut. Pr.	1/n	0.05	1/n	0.05
100	100	136.63	185.64	129.67	184.75
	200	123.49	182.46	102.72	184.07
400	100	33.15	182.1	26.12	181.17
	200	23.01	177.4	10.25	174.63

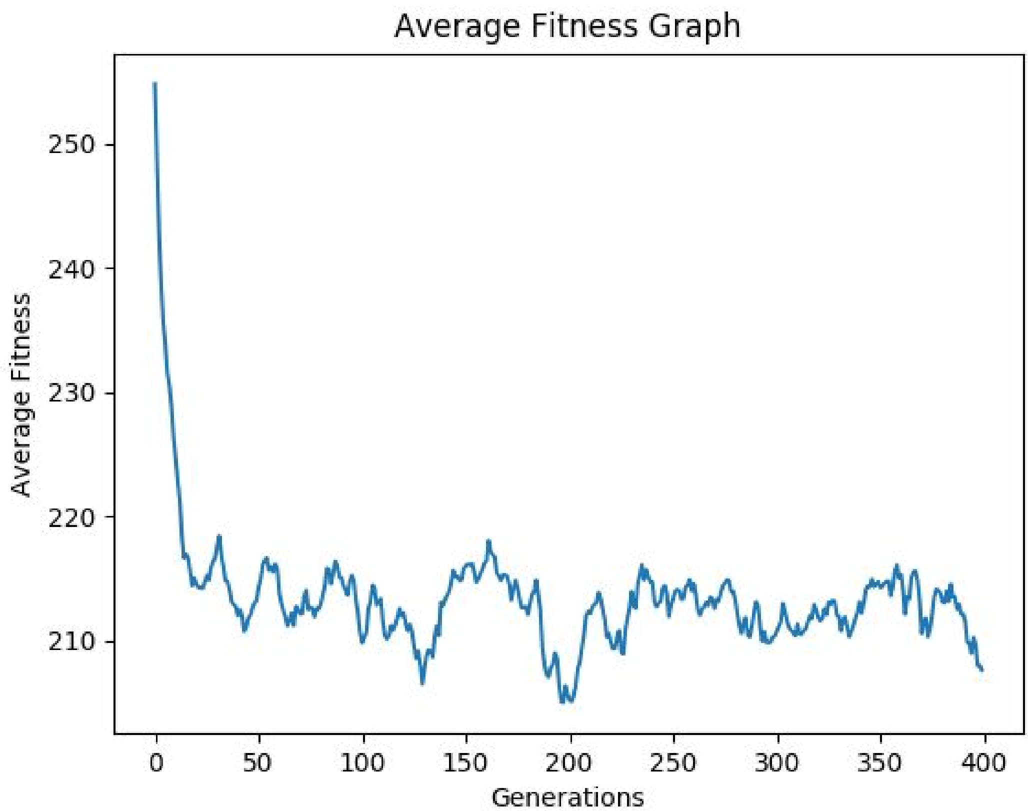
030.txt					
	Crossover Pr.	0.5		0.9	
# Generations	Pop. Size \ Mut. Pr.	1/n	0.05	1/n	0.05
100	100	128.21	184.09	121.80	179.15
	200	116.36	185.11	103.36	177.69
400	100	34.78	181.65	29.43	179.23
	200	18.83	180.89	13.05	176.54

Graphs of average solutions for 030.txt is listed below.

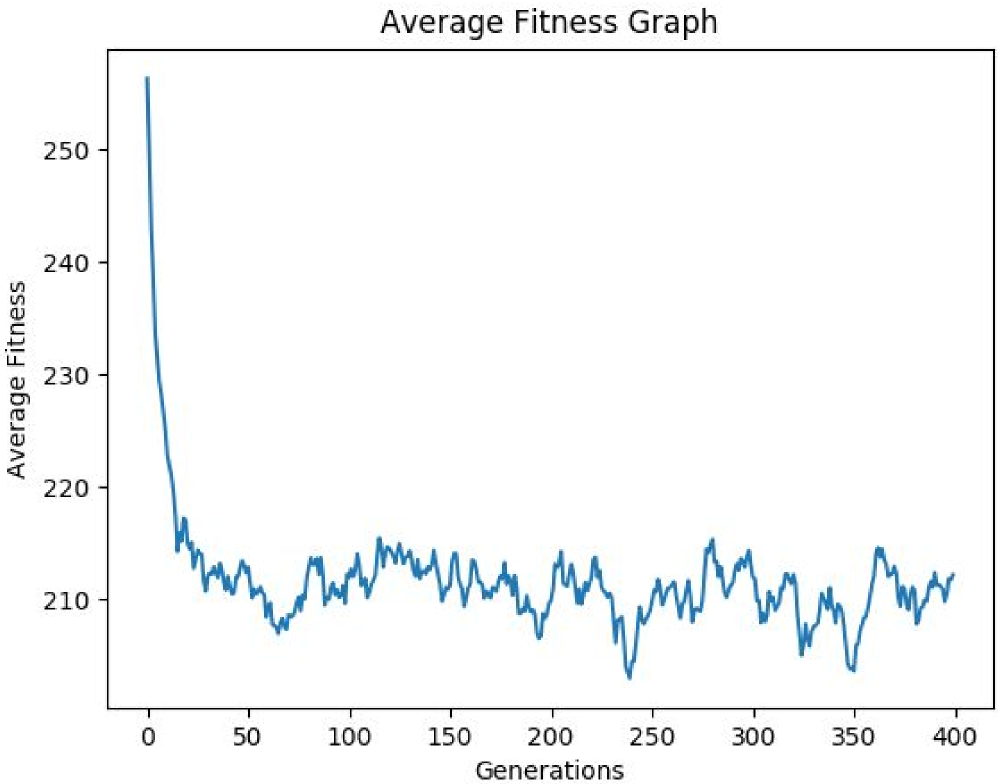
1. File: 030.txt, # Generations: 400, Pop. Size 100, Crossover Prob. 0.5, Mut. Prob. 0.05.



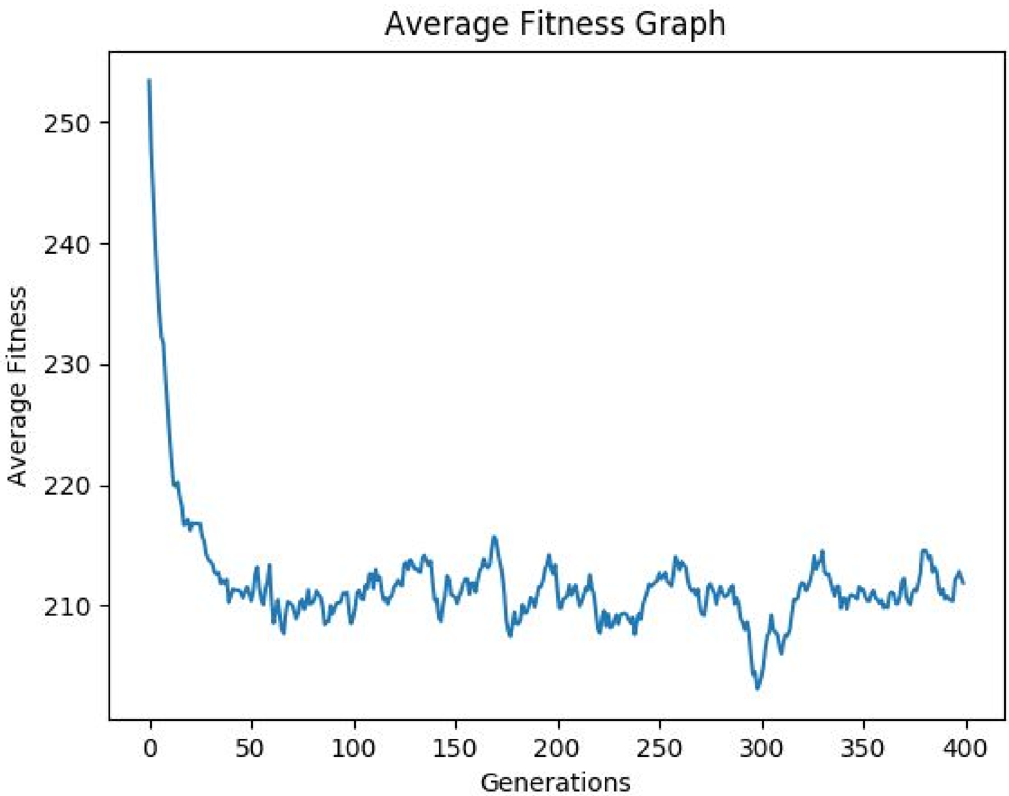
2. File: 030.txt, # Generations: 400, Pop. Size 200, Crossover Prob. 0.5, Mut. Prob. 0.05.



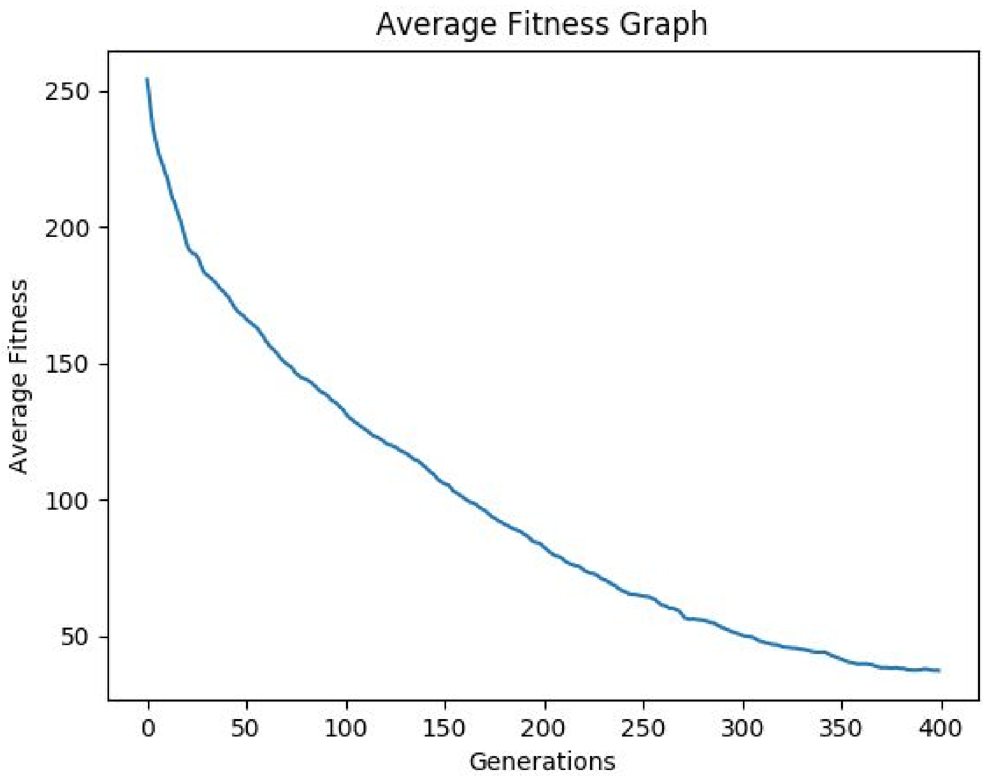
3. File: 030.txt, # Generations: 400, Pop. Size 100, Crossover Prob. 0.9, Mut. Prob. 0.05.



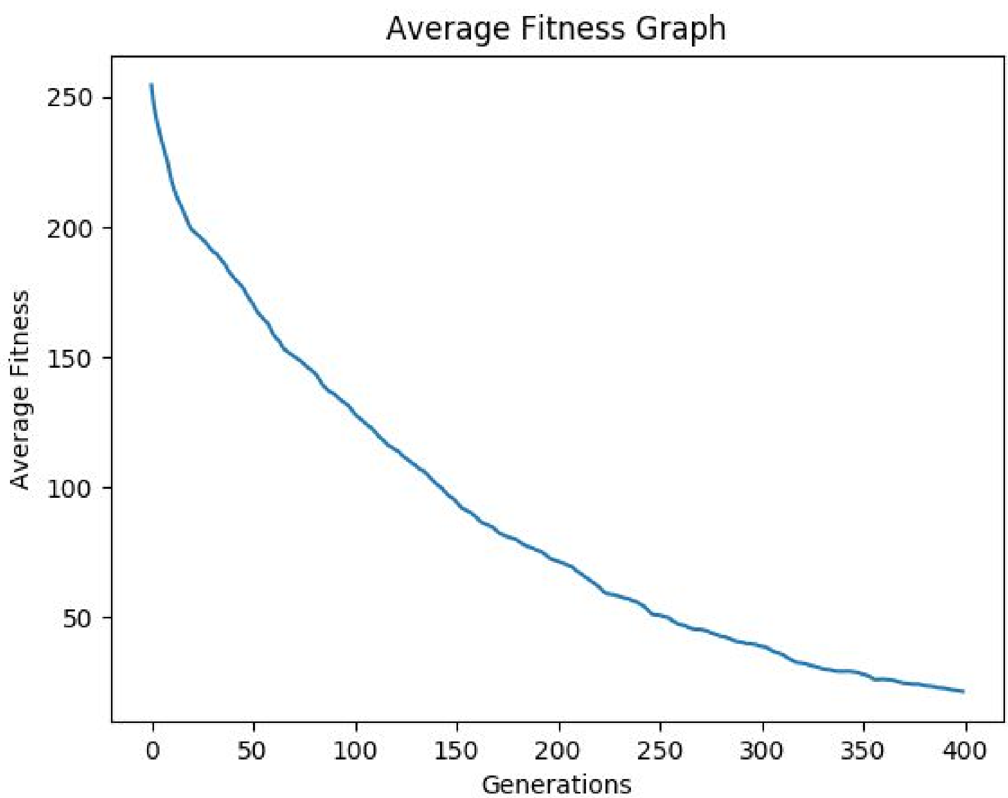
4. File: 030.txt, # Generations: 400, Pop. Size 200, Crossover Prob. 0.9, Mut. Prob. 0.05.



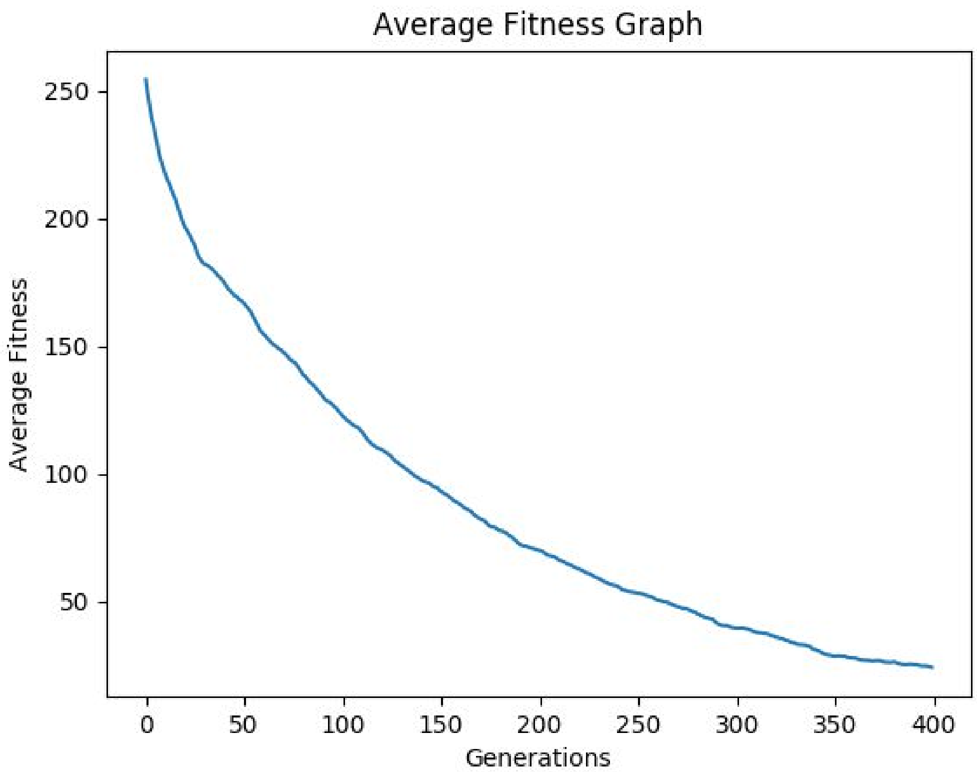
5. File: 030.txt, # Generations: 400, Pop. Size 100, Crossover Prob. 0.5, Mut. Prob. 1/n.



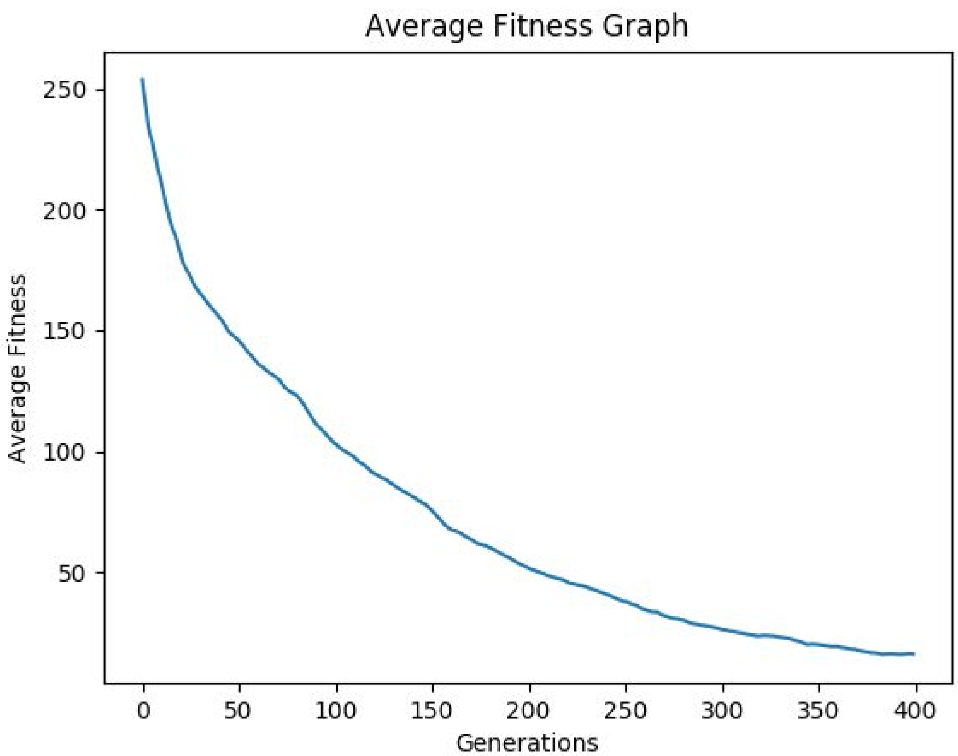
6. File: 030.txt, # Generations: 400, Pop. Size 200, Crossover Prob. 0.5, Mut. Prob. 1/n.



7. File: 030.txt, # Generations: 400, Pop. Size 100, Crossover Prob. 0.9, Mut. Prob. 1/n.



8. File: 030.txt, # Generations: 400, Pop. Size 200, Crossover Prob. 0.9, Mut. Prob. 1/n.



3-) CONCLUSION

As we can see from the graphs above, when all other parameters are equal, lower mutation probability generates better results. Because, in mating pool we are selecting better solutions mostly, effect of the mutation becomes negative. Also, when mutation probability equal to 0.05, average solution does not improve after 25th-30th generation. When mutation probability equal to 1/1000, average solution keeps improving while number of generations increasing.

When all other parameters are equal, higher crossover probability generates a little bit better results.

As a result, mutation probability affects most among the given parameters, and we get the best results of our algorithm for 030.txt when:

- #Generations: 400
- Population Size: 200
- Crossover Probability: 0.9
- Mutation Probability: 0.001