SOLVING CNDP USING GA

Algorithm analysis and designing

Members: Nyamdavaa
Michidmaa
Munkhdelger
Sarnai

Objective Function

```python
def calc_fitness(remove_nodes):
    graph = gr.graph
    reduced_graph = gr.remove_nodes(graph, remove_nodes)
    return gr.calculate_fitness(reduced_graph)


def calc_pop_fitness(pop):
```

```python
def calculate_fitness (graph) :
    components = list(nx.connected_components(graph))
    res = 0
    for component in components :
        res += len(component) * (len(component) - 1) / 2
    return res
```

Fitness value

```
def calc_pop_fitness(pop):
    with multiprocessing.Pool() as pool:
        fitness_values = pool.map(calc_fitness, pop)
    return min(fitness_values)
```

Selection

```
def select_mating_pool(pop) :
    pop_fitness = {}
    with multiprocessing.Pool() as pool:
        fitness_values = pool.map(calc_fitness, pop)
    pop_fitness = {i: fitness_values[i] for i in range(len(fitness_values))}
    sorted_pop_fitness = sorted(pop_fitness.items(), key=lambda x: x[1], reverse=False)
    parent1 = pop[sorted_pop_fitness[0][0]]
    parent2 = pop[sorted_pop_fitness[1][0]]
    return [parent1, parent2]
```

Crossover

```python
def crossover(parents) :
    offsprings = parents.copy()
    crossover_point = int(len(parents[0]) / 2)
    offsprings[0] = parents[0][:crossover_point] + parents[1][crossover_point:]
    offsprings[1] = parents[1][:crossover_point] + parents[0][crossover_point:]
    return offsprings
```

Mutation

```
def mutation (offspring) :
    mutation_point = numpy.random.randint(0, len(offspring))
    new_gene = numpy.random.randint(0, len(gr.graph.nodes))
    offspring[mutation_point] = new_gene
    return offspring
```

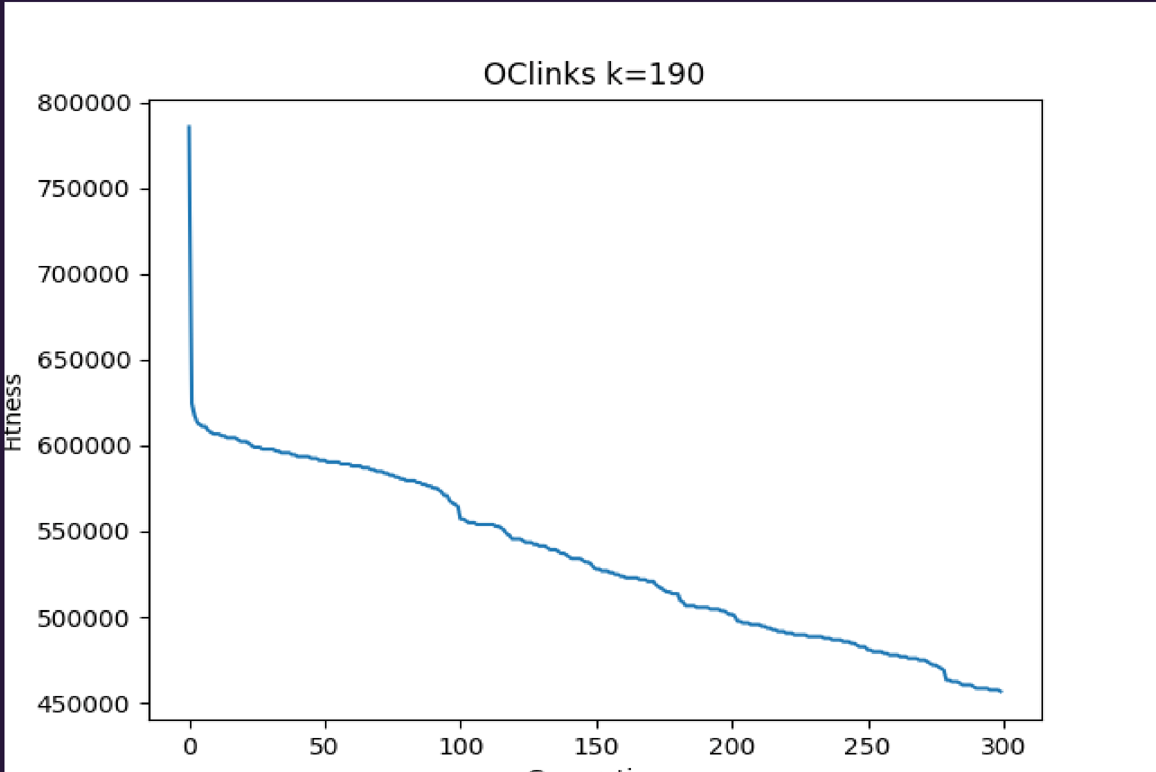
Population

```python
# Creating initial population randomly
population = []
for i in range(pop_size):
    individual = helpers.select_random_distinct_numbers(int(gene_cnt), 0, len(graph.nodes))
    population.append(individual)

population.append(gr.select_max_degree_nodes(graph, int(gene_cnt)))
```

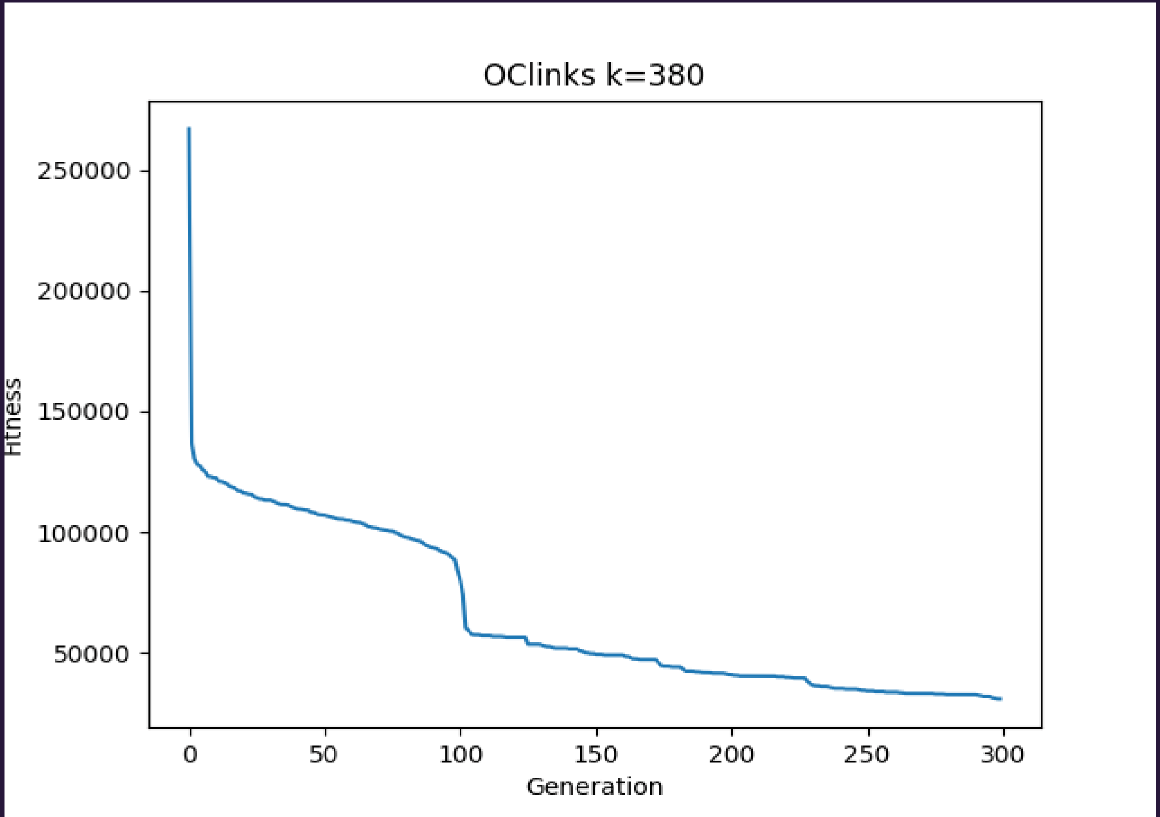
Main Function

```
for generation in range(generation_cnt):
    pop_fitness = ga.calc_pop_fitness(population)
    print("Generation : ", generation)
    print("Best result : ", pop_fitness)
    parents = ga.select_mating_pool(population)
    offsrings = ga.crossover(parents)
    for i in range(len(offsrings)) :
        offsrings[i] = ga.mutation(offsrings[i])
    # replace worst individuals with offsrings
    pop_fitness = {}
    with multiprocessing.Pool() as pool:
        fitness_values = pool.map(ga.calc_fitness, population)
    pop_fitness = {i: fitness_values[i] for i in range(len(fitness_values))}
    sorted_pop_fitness = sorted(pop_fitness.items(), key=lambda x: x[1], reverse=True)
    for i in range(len(offsrings)) :
        population[sorted_pop_fitness[i][0]] = offsrings[i]
```

Result 1


OClinks k=190
800000
750000
700000
650000
600000
550000
500000
450000
Fitness
0
50
100
150
200
250
300

Result 2



OClinks k=380
250000
200000
150000
100000
50000
Fitness
0
50
100
150
200
250
300
Generation

Algorithm 1

1. Representation: G = (V, E) графаас k тоогоор R оройн олонлогийг санамсаргүйгээр үүсгэнэ.
2. Initialization: Санамсаргүйгээр нэг сүрэгт 100 ширхэг individual-тай байхаар generation 0-ийг үүсгэнэ. Individual бүрт G графт харьяалагдах оройн дугааруудаас k ширхэг байна.

3. Evaluation: Сүргийн individual бүр дээр Objective Function-оор тус бүрчлэн Fitness value-г тооцоолно.
4. Selection: Crossover үйлдлийг хийхийн тулд тооцоолж гаргасан Fitness value array-гаас Tournament Selection аргыг ашиглан эцэг болон эх хромсомыг сонгоно.

5. Reproduction:
    a. Crossover:
    b. Mutation:

6. Replacement: Өмнөх generation-ны хромсомууд болон Reproduction шатаас үүссэн шинэ хромсомуудын дундаас дараагийн generation-ыг үүсгэнэ.
7. Termination: Хэрэв maximum generation-ны өгөгдсөн тоонд хүрээгүй бол 3-р алхам болох Evaluation оператороос ахин эхлэнэ. Үгүй бол дараагийн алхам руу шилжилнэ.

8. Solution Extraction: Эцсийн generation дахь хамгийн сайн Fitness value-тай хромсом нь CNDP-ийн хамгийн оновчтой шийд болно.

Objective Function

```python
def calc_fitness(gr, individual):
    reduced_graph = gr.copy()
    reduced_graph.remove_nodes_from(individual)

    components = list(nx.connected_components(reduced_graph))
    fitness = 0
    for component in components :
        fitness += len(component) * (len(component) - 1) / 2
    return fitness
```

Generate Population

```
numbers = list(range(0, 1898))
# Creating initial population randomly
population = []
for i in range(pop_size):
    random.shuffle(numbers)
    res = numbers[:gene_cnt]
    population.append(res)
```

Selection

```
def tournament_selection(population, pop_fitness, tournament_size):
    # Select random individuals for the tournament
    randnum = list(range(0, len(population)))
    random.shuffle(randnum)
    tournament_idx = randnum[:tournament_size]

    # Evaluate the fitness of the tournament participants
    tournament_fitness = [pop_fitness[idx] for idx in tournament_idx]
    # print(tournament_fitness)

    # Find the individual with the highest fitness in the tournament
    winner_index = tournament_fitness.index(min(tournament_fitness))
    winner = tournament_idx[winner_index]
    # print(pop_fitness[winner])
    return winner
```

Crossover

```python
offsprings = population.copy()
for i in range(int(pop_size / 2)):
    # Parents selection
    parent1_key = tournament_selection(population, pop_fitness, 3)
    parent2_key = tournament_selection(population, pop_fitness, 3)
    parents = [population[parent1_key], population[parent2_key]]

    # Crossover
    crossover_point = int(gene_cnt / 2)

    off = [[0] * gene_cnt for _ in range(2)]

    for k in range(int(gene_cnt/2)):
        off[0][k] = parents[0][k]
        off[1][k] = parents[1][k]

    j = 0
    for k in range(int(gene_cnt/2), gene_cnt):
        while j < gene_cnt:
            if parents[1][j] not in off[0]:
                off[0][k] = parents[1][j]
                j += 1
                break
            else:
                j += 1

    j = 0
    for k in range(int(gene_cnt/2), gene_cnt):
        while j < gene_cnt:
            if parents[0][j] not in off[1]:
                off[1][k] = parents[0][j]
                j += 1
                break
            else:
                j += 1

    offsprings[i] = off[0]
    offsprings[pop_size - 1 - i] = off[1]
```

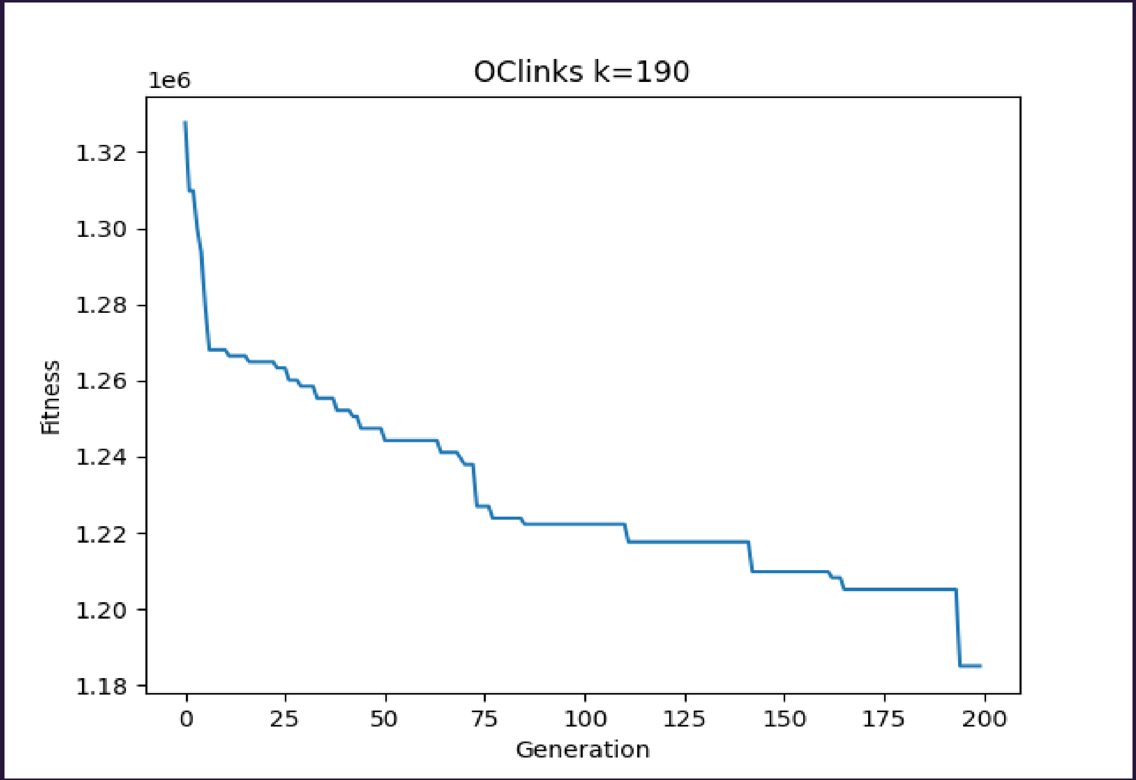
Mutation

```
mut_prob = 0.1
mutrandom = list(range(0, pop_size))
random.shuffle(mutrandom)
res = mutrandom[:int(pop_size * mut_prob)]
for i in range(int(pop_size * mut_prob)) :
    mutation_point = numpy.random.randint(0, gene_cnt)
    new_gene = numpy.random.randint(0, 1899)
    offsprings[res[i]][mutation_point] = new_gene
```

Fitness value

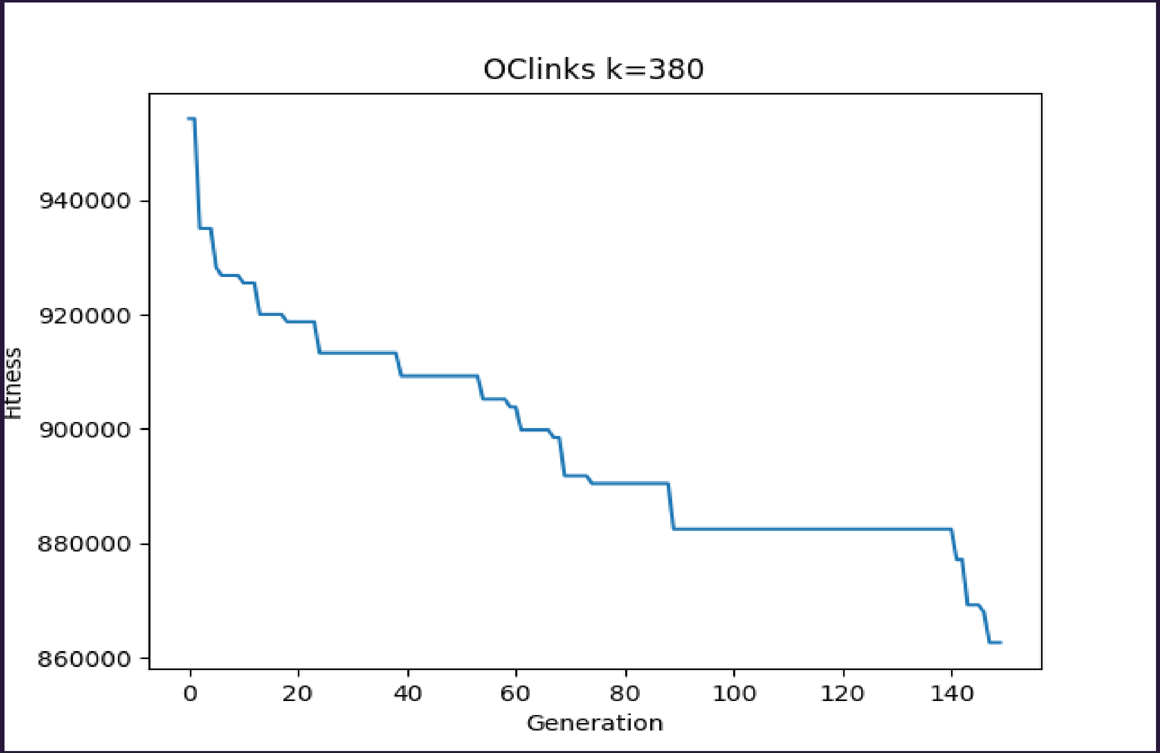
```
pop_fitness = {}
# fitness value бага байвал сайн шийд
fitness_min = float('inf')
for i in range(pop_size) :
    pop_fitness[i] = calc_fitness(graph, population[i])

    if pop_fitness[i] < fitness_min :
        fitness_min = pop_fitness[i]

print("Generation : ", generation)
print("Best result : ", fitness_min)
```

Result 1



OClinks k=190
1e6
1.32
1.30
1.28
1.26
1.24
1.22
1.20
1.18
Fitness
0
25
50
75
100
125
150
175
200
Generation

Result 2


OClinks k=380
Fitness
940000
920000
900000
880000
860000
0
20
40
60
80
100
120
140
Generation