IMPLEMENTATION

To implement a secondary solution we want to use genetics algorithms. We will also utilize local search to improve our eploitation

CHROMOSOME

We should now define our chromosomes. Based on the problem the chormosomes are binary arrays of size J (number of columns/sets). each cell is either 0 or 1, if the i-th index is 1 it means that in that possible solution, the i-th set is included in the solution, and if 0 it is not.

To have a fair comparison with the algorithm in the paper, we define the constants the same as the paper:

```
GenCount = 100
PopCount = 20
crossover_rate = 0.8
mutation_rate = 0.33
rho1 = 0.15
rho2 = 1.1
targetWeight = 492
oldpopulationkeeprate = 0.1

que = open("scp48.txt", "r")

#initialize the constants
universalSetCount, subsetCount = map(int,
que.readline().strip().split(" "))
print(universalSetCount, subsetCount)

subsets = [[0] for i in range(subsetCount)]

cost = []
heuristic = []

i = 0
while i < subsetCount:
    temp = list(map(int, que.readline().lstrip(" ").rstrip("\
n").rstrip(" ").split(" ")))
    cost = cost + temp
    i += len(temp)

for i in range(universalSetCount):
    cnt = int(que.readline().lstrip(" ").rstrip("\n").rstrip(" "))
    j = 0
    while j < cnt:
```

```
        temp = list(map(int, que.readline().lstrip(" ").rstrip("\
n").rstrip(" ").split(" ")))
        #print(temp)
        for lis in temp:
            subsets[lis-1].append(i)
        j += len(temp)

heuristic = [len(subsets[i])/cost[i] for i in range(len(subsets))]

# Given sets and variables
alphai = [[] for _ in range(universalSetCount)]
for i in range(universalSetCount):
    for j in range(len(subsets)):
        if i in subsets[j]:
            alphai[i].append(j)

200 1000
```

HELPER FUNCTION

```
def is_covered(solution):
    """gets a solution array and determines wether the
    union of this solution covers all the elements in
    the universal set or not.

    Args:
        solution (list): a solution array for the problem

    Returns:
        bool: True if the solution is an answer, False otherwise
    """
    uncovered_lines = [1 for _ in range(universalSetCount)]
    for i in range(len(subsets)):
        if solution[i] == 1:
            for j in subsets[i]:
                uncovered_lines[j-1] = 0
    return (sum(uncovered_lines) == 0)
```

COST FUNCTION

The cost of a solution is defined as the sum of the costs of all the sets that are included in that solution. a.k.a.

$$f(solution) = \Sigma_{j=1...n} Cost_j * solution[j]$$

```python
def costt(solution):
    """gives cost for a solution

    Args:
        solution (list): a potential answer

    Returns:
        int: a number representing the solution cost
    """
    totalCost = 0
    for i in range(len(subsets)):
        if solution[i] == 1:
            totalCost += cost[i]
    if is_covered(solution) == False:
        totalCost += 2000 #huge penalty for not having full coverage
    return totalCost
```

CROSSOVER

In order to do crossover, assuming that we have chosen two highly fit parents, we do one-point swap cross over. meaning that we choose a random index and split and swap the first part of the parents to create two childeren.

```python
import random

def crossover (parent1, parent2):
    """cross over function using one-point swap technique.

    Args:
        parent1 (list): a highly fit solution
        parent2 (list): another highly fit solution

    Returns:
        list: a pair of two solutions as results of crossover
    """
    if random.random() < crossover_rate:
        # Use one-point crossover
        point = random.randint(1, len(subsets) - 2) # Random crossover
point
        child1 = parent1[:point] + parent2[point:] # First offspring
        child2 = parent2[:point] + parent1[point:] # Second offspring
    else:
        # No crossover, offspring are copies of parents
        child1 = parent1.copy()
        child2 = parent2.copy()
    return [child1, child2]
```

MUTATION

We are going to use random bit flip mutation. Meaning for mutation for each index i, there is a probabiliy it gets mutated, meaning if i is included in the solution it gets removed, if not it gets added.

```
def mutation(solution):
    """mutation function that uses random but flip technique.

    Args:
        solution (list): a solution to the problem
    """
    for i in range(len(solution)):
        if random.random() < mutation_rate:
            solution[i] = 1 - solution[i]
    return
```

ELIMINATION OF THE REDUNDANT COLUMNS

The paper introduces an iterative algorithm to delete redundant sets. We implement this algorithm.

```
def eliminate(solution):
    """A function that gets a solution array that is an answer
    to the problem, finds the redundant sets among the sets present
    in the array and deletes them starting from the one with the most
    cost.

    Args:
        solution (list): a solution list that is also an answer
    """
    # Step 1: Initialize wi
    wi = [0 for _ in range(universalSetCount)]
    for i in range(len(solution)):
        if solution[i] == 1:
            for j in range(len(subsets[i])):
                wi[subsets[i][j]-1] += 1

    # Step 2-6: Iterative elimination of redundant columns
    for i in reversed(range(len(subsets))):
        if solution[i] == 1:
            canDelete = True
            for j in range(len(subsets[i])):
                if wi[subsets[i][j]-1] <= 1:
                    canDelete = False
                    break
            if canDelete:
```

```
                solution[i] = 0
                for j in range(len(subsets[i])):
                    wi[subsets[i][j]-1] -= 1
    return
```

LOCAL SEARCH ALGORITHM

The paper also introduces a local search algorithm. The algorithm is as follows:

```
def localSearch (solution):
    """Local search algorithm that gets a solution that is an answer
    and searches for any neighboring solution that might have a lower
    costt than this solution. Implemented based on the directives
given
    by the paper

    Args:
        solution (list): a solution array to the problem

    Returns:
        list: A neighboring solution that might have a better costt
    """
    #print(len(solution))
    number_of_columns = 0
    for i in solution:
        if i == 1:
            number_of_columns += 1

    max_cost_element = -1e9
    for i in range(len(solution)):
        if solution[i] == 1:
            max_cost_element = max(max_cost_element, cost[i])

    wi = [[0] for _ in range(universalSetCount)]
    for i in range(len(solution)):
        if solution[i] == 1:
            for j in subsets[i]:
                wi[j].append(i)

    D = int(rho1 * number_of_columns)
    E = rho2 * max_cost_element

    # Choose D columns to eliminate from the solution solution
    for _ in range(D):
        j = random.choice(list(range(sum(solution))))
        i = 0
        for k in range(len(subsets)):
            if solution[k] == 1:
```

```
                if i == j:
                    solution[k] = 0
                    break
                i += 1


    # Perform the covering
    while is_covered(solution) == False:
        #print(len(solution), flush=True)
        record_columns = []
        for j in range(len(subsets)):
            if solution[j] == 0 and cost[j] <= E:
                record_columns.append(j)

        min_ratio = float('inf')
        selected_column = -1

        for column in record_columns:
            ratio = cost[column] / len(subsets[column])
            if ratio < min_ratio:
                min_ratio = ratio
                selected_column = column

        solution[selected_column] = True

    return solution
```

INFERENCE

We should now do inference.

```
#Generating initial population
population = []
for i in range(PopCount):
    individual = [random.randint(0, 1) for _ in range(len(subsets))] #
Random binary array
    while is_covered(individual) == False:
        j = random.randint(0, len(subsets) - sum(individual) - 1)
        for k in range(len(subsets)):
            if individual[k] == 0:
                if j == 0:
                    individual[k] = 1
                else:
                    j -= 1
    eliminate(individual)
    population.append(individual)

# Main loop
```

```python
for gen in range(GenCount):
    # Evaluate fitness of population
    fitness_values = []
    for individual in population:
        fitness_values.append(costt(individual))

    # Print best solution and fitness in current generation
    best_index = fitness_values.index(min(fitness_values))
    best_solution = population[best_index]
    best_fitness = fitness_values[best_index]

    # Check termination criterion
    if best_fitness <= targetWeight: # If optimal solution is found
        break

    # Select parents for reproduction
    parents = []
    for i in range(PopCount):
        # Use tournament selection with size 2
        p1 = random.choice(population)
        p2 = random.choice(population)
        if costt(p1) < costt(p2): # Choose the fitter parent
            parents.append(p1)
        else:
            parents.append(p2)

    # Generate offspring using crossover and mutation
    offspring = []
    for i in range(0, PopCount, 2):
        # Select two parents randomly
        p1 = random.choice(parents)
        p2 = random.choice(parents)

        # Apply crossover
        c1, c2 = crossover(p1, p2)

        # Apply mutation
        mutation(c1)
        mutation(c2)

        # Add offspring to new population
        offspring.append(c1)
        offspring.append(c2)

    # Replace old population with new offspring
    offspring = population + offspring
    offspring = sorted(offspring, key=costt)
    population = offspring[0:(int(oldpopulationkeeprate*PopCount))]
    while len(population) != PopCount:
```

```
population.append(offspring[random.randint(int(oldpopulationkeeprate*P
opCount), len(offspring)-1)])

print("Best fitness:", best_fitness)
print()
```

```
Best fitness: 512
```

COMPARISON:
I have also ran the genetic algorithm 3 times for each test case and written then results below

480	449	456	41
569	559	598	42
537	537	537	43
577	568	559	44
656	644	529	45
676	625	649	46
512	643	509	48
572	572	572	410
331	351	341	51
404	384	331	52
320	255	306	53
337	326	344	54
323	316	212	55
363	371	283	A2
266	317	362	A3
346	276	335	A4
343	377	362	A5
71	92	84	B1
95	101	94	B2
79	100	95	B4
102	100	99	B5

I have also compared my best results for problem sets with paper's best results and the actual answers to the problem sets:

Name	41	42	43	44	45	46	48	410	51	52	53	54	55	A2	A3	A4	A5	B1	B2	B4	B5
GEN	449	559	537	559	529	625	509	572	331	331	255	326	212	283	266	276	343	71	94	79	99
ACS	431	512	516	496	512	577	511	516	257	302	233	242	211	252	238	234	236	71	76	79	72
Percent	**96.0%**	91.6%	**96.0%**	88.7%	**96.8%**	92.3%	**101%**	90.2%	77.6%	91.2%	91.4%	74.2%	**99.5%**	89.0%	89.5%	84.8%	68.8%	**100%**	81.0%	**100%**	72.7%
Actual	429	512	516	494	512	560	492	514	253	302	226	242	211	252	232	234	236	69	76	79	72