CS420 - Project Report 01
Searching for the Knapsack

Members and Assignment plan

- **Nguyen Vu Dang Khoa - 20125007**
    - Prepare algorithm 3.
    - Visualize 3
    - Prepare report algorithm 3
    - Code DP find optimal solution
    - Compare performance
    - Make video demo
- **Tran Bao Loi - 20125010**
    - Prepare algorithm 4.
    - Visualize 4
    - Prepare report algorithm 4
    - Generate test
    - Compare performance
    - Make video demo
- **Phan Minh Duy - 20125027**
    - Prepare algorithm 1, 2.
    - Visualize 1, 2
    - Prepare report algorithm 1, 2
    - Review report
    - Make video demo

Self-Assessment for completion level for each requirements

No	**Criteria**	Self-Estimate	Maximum Score
1	Algorithm 1	10%	10%
2	Algorithm 2	15%	15%
3	Algorithm 3	15%	15%
4	Algorithm 4	10%	10%
5	Generate at least 5 small datasets of sizes 10-40. Compare with performance in the experiment section. Create videos to show your implementation.	10%	10%
6	Generate at least 5 large datasets of sizes 50-1000 with a number of classes of 5-10. Compare with performance in the experiment section. Create videos to show your implementation.	15%	15%
7	Report your algorithms, and experiments with some reflection or comments.	25%	25%
Total		100%	100%

I. Introduction

The original Knapsack problem is considered n items and each of it is associated by two values of weight and profit. Our main mission is that we will put some of these items into a W-kilogram bag, which satisfies that the total weight of these items is not larger than W and the total profit of these items is maximum. There are many different variations of this type of problem as people can develop them by adding more constraints in order to make this harder. In this project, we work with the original Knapsack problem with one adding constraint as “There are T classes and each item is classified in some particular class from these T classes. We must put the set of items from n items to satisfy that each class has at least one item in this set, and the capacity and profit conditions are the same as the original problem”. As we research on the Internet, the name of the problem in this project is “Multiple-choice Knapsack Problem”.

In this problem, each **weight is a float number**, so a **dynamic programming approach would not be feasible**. So to solve this problem, we will introduce more effective approaches. In this document, we mainly focus on how to solve this problem through four algorithms: Brute force searching, Branch and Bound, Local Beam Search, and Genetics algorithm. Our problem can be determined in a mathematical representation as

Maximize $\sum_{i=1}^{n} v_i x_i$ *subject to satisfy constraints:*

1. $\sum_{i=1}^{n} w_i x_i \leq W,\ x_i \in \{0,\ 1\}$
2. *At least one item from each class* $c_j,\ j \in \{1,\ 2,\ \ldots,\ m\}$

The video demo: https://youtu.be/9VILHL2L3Ng

Note: The sorting in calculating the future value function of the Branch and Bound method in this video is redundant.

The github link: baoloi2002/CS420-AI-Project-1 (github.com)

Note: will open after deadline

II. Dataset

We have generated 20 test cases. The first 10 test cases (0 to 9) are small, their size is about 10-40 and their class is about 3-5. The last 10 test cases (10 to 19) are larger, their size is about 50-1000 and their class is about 5-10. Below is the summary of our datasets.

```
[test\input\INPUT_0.txt] >> size: 4, cap: 71, class: 3, sum val: 1045,
sum weight: 118
[test\input\INPUT_1.txt] >> size: 8, cap: 124, class: 3, sum val: 3036,
sum weight: 184
[test\input\INPUT_2.txt] >> size: 12, cap: 179, class: 4, sum val: 3149,
sum weight: 404
[test\input\INPUT_3.txt] >> size: 16, cap: 256, class: 4, sum val: 3809,
sum weight: 485
[test\input\INPUT_4.txt] >> size: 20, cap: 360, class: 4, sum val: 4909,
```

```
sum weight: 412
[test\input\INPUT_5.txt] >> size: 24, cap: 334, class: 4, sum val: 5476,
sum weight: 604
[test\input\INPUT_6.txt] >> size: 28, cap: 522, class: 3, sum val: 7920,
sum weight: 674
[test\input\INPUT_7.txt] >> size: 32, cap: 536, class: 4, sum val: 8647,
sum weight: 725
[test\input\INPUT_8.txt] >> size: 36, cap: 395, class: 4, sum val: 8473,
sum weight: 967
[test\input\INPUT_9.txt] >> size: 40, cap: 421, class: 5, sum val: 9438,
sum weight: 939
[test\input\INPUT_10.txt] >> size: 100, cap: 3554, class: 9, sum val:
25298, sum weight: 5135
[test\input\INPUT_11.txt] >> size: 200, cap: 6346, class: 8, sum val:
48185, sum weight: 10177
[test\input\INPUT_12.txt] >> size: 300, cap: 11253, class: 7, sum val:
75623, sum weight: 15225
[test\input\INPUT_13.txt] >> size: 400, cap: 13307, class: 9, sum val:
97466, sum weight: 20186
[test\input\INPUT_14.txt] >> size: 500, cap: 18445, class: 8, sum val:
129533, sum weight: 25534
[test\input\INPUT_15.txt] >> size: 600, cap: 18947, class: 5, sum val:
146057, sum weight: 30790
[test\input\INPUT_16.txt] >> size: 700, cap: 23603, class: 9, sum val:
172928, sum weight: 34483
[test\input\INPUT_17.txt] >> size: 800, cap: 28595, class: 10, sum val:
202707, sum weight: 40035
[test\input\INPUT_18.txt] >> size: 900, cap: 33366, class: 7, sum val:
222858, sum weight: 43571
[test\input\INPUT_19.txt] >> size: 1000, cap: 30104, class: 7, sum val:
252272, sum weight: 49394
[test\input\INPUT_20.txt] >> size: 672, cap: 1014, class: 8, sum val:
33607, sum weight: 202333
[test\input\INPUT_21.txt] >> size: 31, cap: 45, class: 4, sum val: 1700,
sum weight: 514
[test\input\INPUT_22.txt] >> size: 1000, cap: 5828, class: 10, sum val:
5469, sum weight: 7465
```

III. Brute Force approach

1. Algorithm

Brute-force is the simplest approach to solve our problem as we will search all the possible states which can happen in the problem. With n objects that we can choose for trying, there will be 2^n possible cases (The number of subsets from the set of n elements). With this approach, we only try with the cases $n \leq 30$, due to the limitation of hardware in

our machine. As we know that, our program can only perform 100 million computations in each stage and it will take more than 3 hours to complete the case $n = 40$. Thus, it is impossible to try some cases such as $n = 50$ or $n = 60$ with this approach as the time complexity of these test cases are very extremely greater than the cases n = 40. (2^{10} and 2^{20} times respectively).

The main coding idea in this method is: "We will generate 2^n subsets of n elements which only consists of 0 and 1 in each set (representing the appearance of some objects among n objects in our set). Next, we will check the total weight of each subset of objects with the maximum weight of the bag, we also calculate the total value that we can receive from putting these objects from each subset. Finally, we will check the class of each set of objects by appending the class of each object in each set into a list and we will set structure in python to guarantee the uniqueness of each element in the set of classes.

The Pseudo code of the Brute Force Approach can be represented as follows:

```
function BRUTE-FORCE(max_cap, total_class, state, current_position):
    if current_position == SIZE(state):
        val ← sum of value of selected items based on state
        cap ← sum of weight of selected items based on state
        class ← number of distinct class of selected item based on state
        if state is valid (cap <= max_cap and class = total_class) and val > best_val:
            best_val ← val
            best_state ← state
        return

    for choice in [not_select, select]:
        state[current_position] ← choice
        Call BRUTE-FORCE(max_cap, total_class, state, current_position + 1)
```

2. Visualization

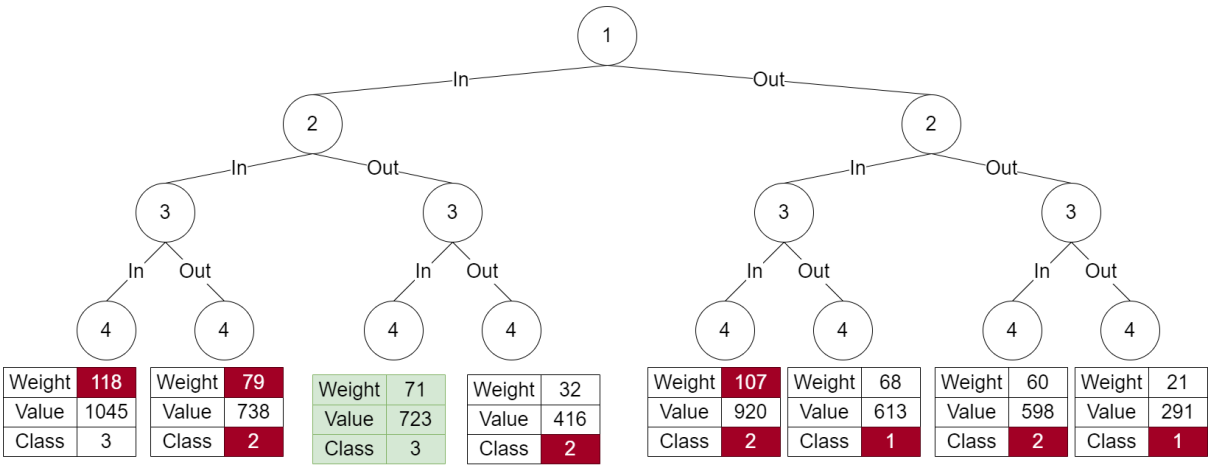
Input:

```
71
3
21, 11, 47, 39
291, 125, 322, 307
1, 2, 1, 3
```

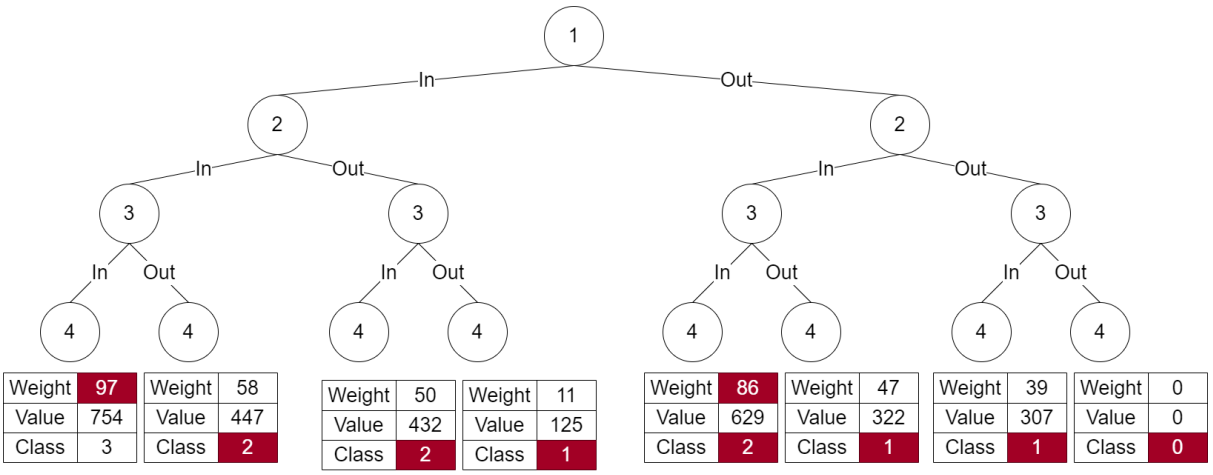
Step:

	Weigh	Value	Class
1	21	291	1
2	11	125	2
3	47	322	1
4	39	307	3

Red is a test condition violation and Green is valid. We divide it into two cases for convenience, one is to get the first item and the other is to not take the first item.
Get the first item:



Don't get the first item:



Get the best result from the valid state.

output:

```
723
1, 1, 0, 1
```

3. Conclusion

This approach is sure to always give optimal results. The weakness of this approach is the time complexity $O(2^n)$.
In the next section, we will introduce a more optimal approach that is improved from this method.

IV. Branch and Bound approach

1. Algorithms

Branch and Bound is an innovation of the Brute Force Approach as it will not check all 2^n cases and it can prune some branches of checking when we represent all cases in the form of a binary tree based on an upper bound which is the maximum value in considered steps of our problem. In the worst case, the complexity of this method is $O(2^n)$, the same as the Brute-force approach.

The main coding idea in this method is: “We will update the result as soon as possible in order to make a bound for checking the profit. We will prune any branch in our tree when we meet one of the following conditions: The current capacity is greater than the capacity of the bag, the total profit of future objects and all current objects are smaller than our bound value, the number of class of set of future objects and current objects is smaller than the number of class in our test case. With these checks, we will minimize unnecessary cases in order to enhance our performance.”

Summary:

- update result as soon as possible
- Stop when total weight > capacity
- Stop if you know that all of the value ahead + current value <= best value
- Stop if you know that all of the class ahead union with current class set <= number of class

Since in this problem, the maximum number of classes is 10, so we will represent the state of the classes in binary form with bit ith = 1 meaning that class i+1 is available and vice versa.

Ex: 0101 = in bag have item from class 1 and class 3

Future value:

To talk about this value, we have many ways to calculate and predict. Our team will cover three ways to calculate this value:

- Method 1: We will assume we take all the values from i to N, so the future value will equal suffix sum of weight.
- Method 2: Moreover we found that an item is the best if its value/weight is the highest (we will ignore its weight). Hence, the best possible future value equal remain capacity in bag * best value/weight at suffix.
- Method 3: From method 2, we see that we have made excellent assumptions as it is possible to get good items to fill in the vacant position. But in practice it's not that great, so we will use the greedy method of sorting the value/weight value in descending order with the suffix part and then drop those items one by one until the total weight is greater or equal to the remaining capacity in the bag.

For optimal performance of the algorithm, we will rearrange the original sequence descending by value/weight so that methods 2 and 3 will proceed to smaller future values.

The method 3 also doesn’t need to sort descending order and we can applied binary search to find range fit optimally.

The Pseudo code of Branch and Bound Approach can be represented as follows:

First is **pre-calculation**

Explanation: In this case, suffixSumVal will represent the total value of objects in the future branch and suffixNumClass will be represented for the bit mask of the number of classes of all objects in the future chosen. The bitmask here is represented as we will create the number with the form (1...0) represented for the object i (As it will have i number 0 before the number 1). And this bitmask is calculated in base 2. The time complexity for this step is O(size)

```
function pre_calculation():
        size = the number of objects
        Array suffixSumVal is suffix sum of value array
        Array suffixNumClass is suffix set of class array
        Array suffixRate is suffix max of value/weight

        for i from size down to 0:
                if i < size - 1:
                        suffixNumClass[i] = suffixNumClass[i + 1]
                        suffixSumVal[i] = suffixSumVal[i + 1]
                        suffixRate [i] = suffixRate [i + 1]

                suffixSumVal[i] += value of the object i
                suffixNumClass[i] = suffixNumClass[i] | (1 << the index of class of object i)
(Use or function for bitmask here)
                suffixRate[i] = max(suffixRate[i], Value[i]/Weight[i])
```

Rearrange array:

```
sort array descending order base on value/weight
```

Method 3 function:

future:

```
function future(cur : position, w : remain capacity)
        d : list of value
        for i from cur to size:
                d.append([ value[i] / weight[i],   weight[i]])
        i = 0
        res = 0 : future value
        while w > 0 and i < len(d):
                res += d[i][0] * d[i][1]
                w -= d[i][1]
                i += 1
        return res
```

```
function BRANCH-BOUND(num_Object_in_subset, cap_current = 0, value = 0,
bitMaskClas = 0):
          if current capacity > maximum bag capacity:
                    return
```

```
        if countBit1(bitMaskClass) == number of class and valuable > current best profit:
                update the current best profit by valuable
                update the list of item which can create the greatest profit

        if num_Object_in_subset == number of given objects:
                return

        # Future value + Current value <= Optimal value
        # Method 1
        if val + suffixSumVal[cur] <= best:
                return
        # Method 2
        if val + suffixRate[cur]*(capacity-cap) <= best:
                 return
        # Method 3
        if future(cur, capacity-cap) + val <= best:
                 return


        if Future number of class + Current number of class < number of classes:
                return

        for i = 0 to 1 do:
                mask the current object by i depending on the loop
                BRANCH-BOUND(num_Object_in_subset + 1, cap_current + i * weight of
current object, update the bitMaskClass) //Recursion step here
                mask the current object by 0
```

In the final function to solve the problem, we will call the function pre-calculation() Try(0) and generate some global variables such as number of objects, max capacity, number of classes, weights of all objects, values of all objects, values of class of all objects, best profit and bestChoosing.

2. Visualization

To make this approach more clear, we will give you an example and we will present it by hand. We will show the simulation with method 1 in finding the future value and the array in this case is also unsorted.

Input:

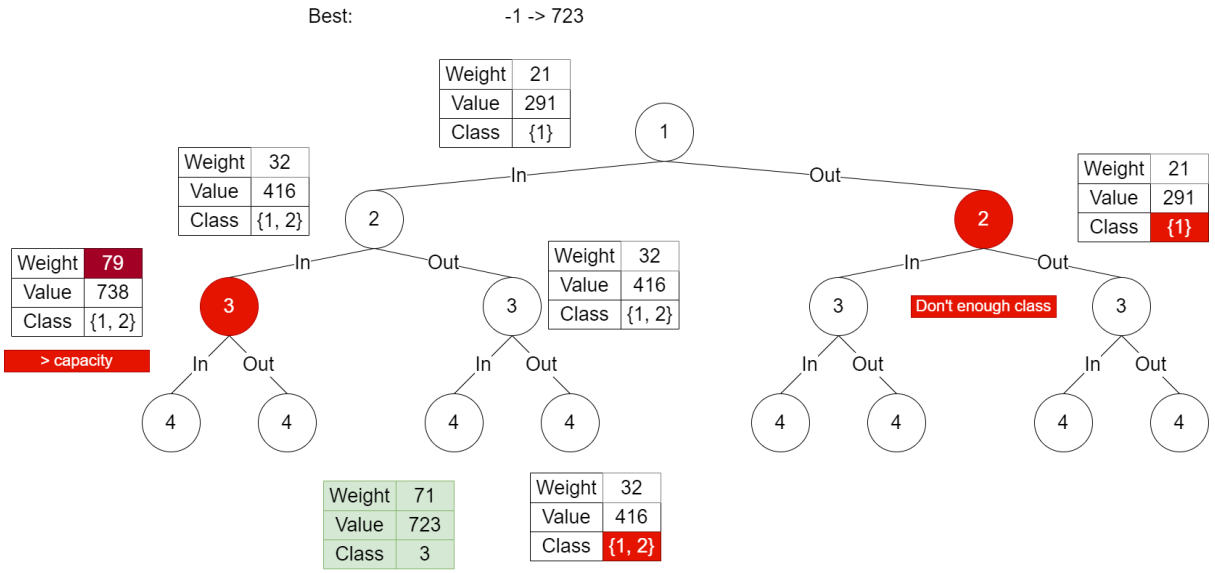
```
71
3
21, 11, 47, 39
291, 125, 322, 307
1, 2, 1, 3
```

	Weigh	Value	Class
1	21	291	1
2	11	125	2
3	47	322	1
4	39	307	3

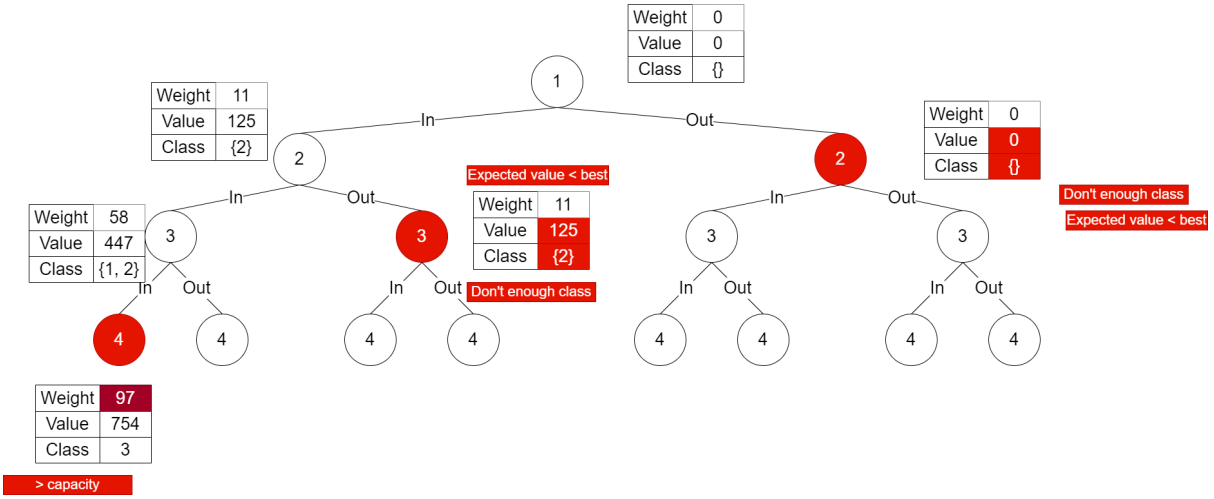
Pre-calculate:

	Value	Class
1 -> 4	1045	{1, 2, 3}
2 -> 4	754	{1, 2, 3}
3 -> 4	629	{1, 3}
4 -> 4	307	{3}

Red is a test condition violation and stop at that node. Green is valid. We divide it into two cases for convenience, one is to get the first item and the other is to not take the first item.
Get the first item:



Don't get the first item:



prune at red node

723
1, 1, 0, 1

3. Conclusion

This method achieves more optimal results in time than brute force. However, it is still unable to solve all problems of too large size (>=50) because the approach is still to find the best solution among all solutions.
In the next section, we will introduce two heuristic algorithms to find the optimal (not the best) solution in a limited time.

V. Local Beam approach

1. Algorithm

This approach is a heuristic to determine which choice can be an optimal solution to achieve the main goal of our problem. Instead of considering all 2^n possible cases, this approach mainly focuses on the set of best k states at each run. As we know that with k being large, our result can be more optimal but it will consume more time to perform our search. With the case k = 1, our approach will become the greedy approach and this situation gives us a non-optimal result in our problem. By filtering k-best states from each step, we can eliminate some states which could lead to a worse situation (exceed max capacity or have large total weight but small total value). The main coding idea of this heuristic is: "We use fitness score to find the k-best states from each step and after a number of steps we can get the list of the items which we can put in the bag and this list satisfies the constraint of our problem.

```
function LOCAL-BEAM(population, max_depth, FITNESS-FN):
        return an individual
        inputs: population, a set of individuals
                FITNESS-FN, a function that measures the fitness of an individual

        for depth from 1 to max_depth:
                successors ← empty set
                for individual in population:
                        add SUCCESSORS(individual) to successors
                population ← best K successors based on FITNESS-FN
        return the best individual at all depth
```

The function starts with a random population of size K, and searches at depth not further than max_depth. This also uses 2 more functions SUCCESSORS and FITNESS-FN as below.

a. Successors

In this function we just generate the successors which are different from its parent 1 position. This does not work well with random starting, because the random state comes with many selected items, changing just one of them does not make a significant difference, and the algorithm may just go around that state. Realizing that, we try to start with an empty state (no items are selected), the algorithm provides much better results (almost optimal) but it takes more time to run.

```
function SUCCESSORS(individual):
```

```
    return a set of individuals
    input: an individual

    successors ← empty set
    for i from 1 to SIZE(individual):
        successor ← individual with different value at i
        add successor to successors

    return successors
```

b. Fitness function

The fitness function we use in this algorithm is:

$$fitness = v \times penalty$$
$$penalty = c \times N + n + v/w$$

- n is the number of objects in current state
- v is the total profit of all selected objects in the subset of n objects
- w is the total weight of all selected objects in the subset of n objects
- c is the number of distinct class of all objects in the subset of n objects
- N is the total number of objects

The meaning of the function is that we want to reach the class number constraint first, since it's very small and easy to achieve. Then we want the state to keep adding items instead of removing, to avoid the algorithm just going around some state. The rate v/w show how much profit in each weight. We want to prioritize the class number first, so we multiply it by N to outscale the other variables.

```
function FITNESS-FN(individual):
    return score of current state
    input: current state

    v ← sum value of selected item in current state
    w ← sum weight of selected item in current state
    c ← number of distinct class of selected item in current state
    n ← number of item in current state
    if weight = 0 or weight > capacity:
        return 0

    score ← c*SIZE(state) + n + v/w
    return score
```

2. Visualization

To make this approach more clearly, we will give you an example and visualize it:
Input:

```
124
3
12, 3, 12, 34, 15, 29, 32, 47
374, 483, 480, 364, 320, 393, 275, 347
```

```
1, 2, 1, 3, 3, 2, 2, 2
```

Initial: empty state, max survive = 5

```
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
```

Level 1:

```
[0, 0, 0, 0, 0, 0, 0, 0] >> parent state, successors below
[1, 0, 0, 0, 0, 0, 0, 0] >> fitness: 15022.33, val: 374, weight: 12, class: 1
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[0, 0, 1, 0, 0, 0, 0, 0] >> fitness: 23520.00, val: 480, weight: 12, class: 1
[0, 0, 0, 1, 0, 0, 0, 0] >> fitness: 7172.94, val: 364, weight: 34, class: 1
[0, 0, 0, 0, 1, 0, 0, 0] >> fitness: 9706.67, val: 320, weight: 15, class: 1
[0, 0, 0, 0, 0, 1, 0, 0] >> fitness: 8862.83, val: 393, weight: 29, class: 1
[0, 0, 0, 0, 0, 0, 1, 0] >> fitness: 4838.28, val: 275, weight: 32, class: 1
[0, 0, 0, 0, 0, 0, 0, 1] >> fitness: 5684.89, val: 347, weight: 47, class: 1

qualify:
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[0, 0, 1, 0, 0, 0, 0, 0] >> fitness: 23520.00, val: 480, weight: 12, class: 1
[1, 0, 0, 0, 0, 0, 0, 0] >> fitness: 15022.33, val: 374, weight: 12, class: 1
[0, 0, 0, 0, 1, 0, 0, 0] >> fitness: 9706.67, val: 320, weight: 15, class: 1
[0, 0, 0, 0, 0, 1, 0, 0] >> fitness: 8862.83, val: 393, weight: 29, class: 1

[local beam step 1] >> current found: value: -1, weight: 0, class: 0
```

Level 2:

```
[0, 1, 0, 0, 0, 0, 0, 0] >> parent state, successors below
[1, 1, 0, 0, 0, 0, 0, 0] >> fitness: 64389.27, val: 857, weight: 15, class: 2
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
[0, 1, 1, 0, 0, 0, 0, 0] >> fitness: 79158.60, val: 963, weight: 15, class: 2
[0, 1, 0, 1, 0, 0, 0, 0] >> fitness: 34635.43, val: 847, weight: 37, class: 2
[0, 1, 0, 0, 1, 0, 0, 0] >> fitness: 50276.72, val: 803, weight: 18, class: 2
[0, 1, 0, 0, 0, 1, 0, 0] >> fitness: 32740.50, val: 876, weight: 32, class: 1
[0, 1, 0, 0, 0, 0, 1, 0] >> fitness: 23996.11, val: 758, weight: 35, class: 1
[0, 1, 0, 0, 0, 0, 0, 1] >> fitness: 22078.00, val: 830, weight: 50, class: 1

[0, 0, 1, 0, 0, 0, 0, 0] >> parent state, successors below
[1, 0, 1, 0, 0, 0, 0, 0] >> fitness: 38928.17, val: 854, weight: 24, class: 1
[0, 1, 1, 0, 0, 0, 0, 0] >> fitness: 79158.60, val: 963, weight: 15, class: 2
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
[0, 0, 1, 1, 0, 0, 0, 0] >> fitness: 30677.57, val: 844, weight: 46, class: 2
[0, 0, 1, 0, 1, 0, 0, 0] >> fitness: 38103.70, val: 800, weight: 27, class: 2
[0, 0, 1, 0, 0, 1, 0, 0] >> fitness: 34302.51, val: 873, weight: 41, class: 2
[0, 0, 1, 0, 0, 0, 1, 0] >> fitness: 26545.11, val: 755, weight: 44, class: 2
[0, 0, 1, 0, 0, 0, 0, 1] >> fitness: 26478.02, val: 827, weight: 59, class: 2

[1, 0, 0, 0, 0, 0, 0, 0] >> parent state, successors below
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
```

[1, 1, 0, 0, 0, 0, 0, 0] >> fitness: 64389.27, val: 857, weight: 15, class: 2
[1, 0, 1, 0, 0, 0, 0, 0] >> fitness: 38928.17, val: 854, weight: 24, class: 1
[1, 0, 0, 1, 0, 0, 0, 0] >> fitness: 25124.09, val: 738, weight: 46, class: 2
[1, 0, 0, 0, 1, 0, 0, 0] >> fitness: 30330.37, val: 694, weight: 27, class: 2
[1, 0, 0, 0, 0, 1, 0, 0] >> fitness: 28154.51, val: 767, weight: 41, class: 2
[1, 0, 0, 0, 0, 0, 1, 0] >> fitness: 21254.75, val: 649, weight: 44, class: 2
[1, 0, 0, 0, 0, 0, 0, 1] >> fitness: 21788.86, val: 721, weight: 59, class: 2

[0, 0, 0, 0, 1, 0, 0, 0] >> parent state, successors below
[1, 0, 0, 0, 1, 0, 0, 0] >> fitness: 30330.37, val: 694, weight: 27, class: 2
[0, 1, 0, 0, 1, 0, 0, 0] >> fitness: 50276.72, val: 803, weight: 18, class: 2
[0, 0, 1, 0, 1, 0, 0, 0] >> fitness: 38103.70, val: 800, weight: 27, class: 2
[0, 0, 0, 1, 1, 0, 0, 0] >> fitness: 16388.08, val: 684, weight: 49, class: 1
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
[0, 0, 0, 0, 1, 1, 0, 0] >> fitness: 24387.84, val: 713, weight: 44, class: 2
[0, 0, 0, 0, 1, 0, 1, 0] >> fitness: 18242.45, val: 595, weight: 47, class: 2
[0, 0, 0, 0, 1, 0, 0, 1] >> fitness: 19181.63, val: 667, weight: 62, class: 2

[0, 0, 0, 0, 0, 1, 0, 0] >> parent state, successors below
[1, 0, 0, 0, 0, 1, 0, 0] >> fitness: 28154.51, val: 767, weight: 41, class: 2
[0, 1, 0, 0, 0, 1, 0, 0] >> fitness: 32740.50, val: 876, weight: 32, class: 1
[0, 0, 1, 0, 0, 1, 0, 0] >> fitness: 34302.51, val: 873, weight: 41, class: 2
[0, 0, 0, 1, 0, 1, 0, 0] >> fitness: 22722.02, val: 757, weight: 63, class: 2
[0, 0, 0, 0, 1, 1, 0, 0] >> fitness: 24387.84, val: 713, weight: 44, class: 2
[0, 0, 0, 0, 0, 0, 0, 0] >> fitness: 0.00, val: 0, weight: 0, class: 0
[0, 0, 0, 0, 0, 1, 1, 0] >> fitness: 13995.15, val: 668, weight: 61, class: 1
[0, 0, 0, 0, 0, 1, 0, 1] >> fitness: 14605.26, val: 740, weight: 76, class: 1

qualify:
[0, 1, 1, 0, 0, 0, 0, 0] >> fitness: 79158.60, val: 963, weight: 15, class: 2
[1, 1, 0, 0, 0, 0, 0, 0] >> fitness: 64389.27, val: 857, weight: 15, class: 2
[0, 1, 0, 0, 1, 0, 0, 0] >> fitness: 50276.72, val: 803, weight: 18, class: 2
[1, 0, 1, 0, 0, 0, 0, 0] >> fitness: 38928.17, val: 854, weight: 24, class: 1
[0, 0, 1, 0, 1, 0, 0, 0] >> fitness: 38103.70, val: 800, weight: 27, class: 2

[local beam step 2] >> current found: value: -1, weight: 0, class: 0

Level 3:

[0, 1, 1, 0, 0, 0, 0, 0] >> parent state, successors below
[1, 1, 1, 0, 0, 0, 0, 0] >> fitness: 91609.26, val: 1337, weight: 27, class: 2
[0, 0, 1, 0, 0, 0, 0, 0] >> fitness: 23520.00, val: 480, weight: 12, class: 1
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[0, 1, 1, 1, 0, 0, 0, 0] >> fitness: 71766.33, val: 1327, weight: 49, class: 3
[0, 1, 1, 0, 1, 0, 0, 0] >> fitness: 89510.63, val: 1283, weight: 30, class: 3
[0, 1, 1, 0, 0, 1, 0, 0] >> fitness: 67553.45, val: 1356, weight: 44, class: 2
[0, 1, 1, 0, 0, 0, 1, 0] >> fitness: 56131.45, val: 1238, weight: 47, class: 2
[0, 1, 1, 0, 0, 0, 0, 1] >> fitness: 52569.03, val: 1310, weight: 62, class: 2

[1, 1, 0, 0, 0, 0, 0, 0] >> parent state, successors below
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[1, 0, 0, 0, 0, 0, 0, 0] >> fitness: 15022.33, val: 374, weight: 12, class: 1
[1, 1, 1, 0, 0, 0, 0, 0] >> fitness: 91609.26, val: 1337, weight: 27, class: 2

```
[1, 1, 0, 1, 0, 0, 0, 0] >> fitness: 63392.33, val: 1221, weight: 49, class: 3
[1, 1, 0, 0, 1, 0, 0, 0] >> fitness: 77956.63, val: 1177, weight: 30, class: 3
[1, 1, 0, 0, 0, 1, 0, 0] >> fitness: 59261.36, val: 1250, weight: 44, class: 2
[1, 1, 0, 0, 0, 0, 1, 0] >> fitness: 48772.34, val: 1132, weight: 47, class: 2
[1, 1, 0, 0, 0, 0, 0, 1] >> fitness: 46256.90, val: 1204, weight: 62, class: 2

[0, 1, 0, 0, 1, 0, 0, 0] >> parent state, successors below
[1, 1, 0, 0, 1, 0, 0, 0] >> fitness: 77956.63, val: 1177, weight: 30, class: 3
[0, 0, 0, 0, 1, 0, 0, 0] >> fitness: 9706.67, val: 320, weight: 15, class: 1
[0, 1, 1, 0, 1, 0, 0, 0] >> fitness: 89510.63, val: 1283, weight: 30, class: 3
[0, 1, 0, 1, 1, 0, 0, 0] >> fitness: 48363.17, val: 1167, weight: 52, class: 2
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[0, 1, 0, 0, 1, 1, 0, 0] >> fitness: 53158.38, val: 1196, weight: 47, class: 2
[0, 1, 0, 0, 1, 0, 1, 0] >> fitness: 43723.68, val: 1078, weight: 50, class: 2
[0, 1, 0, 0, 1, 0, 0, 1] >> fitness: 42196.15, val: 1150, weight: 65, class: 2

[1, 0, 1, 0, 0, 0, 0, 0] >> parent state, successors below
[0, 0, 1, 0, 0, 0, 0, 0] >> fitness: 23520.00, val: 480, weight: 12, class: 1
[1, 1, 1, 0, 0, 0, 0, 0] >> fitness: 91609.26, val: 1337, weight: 27, class: 2
[1, 0, 0, 0, 0, 0, 0, 0] >> fitness: 15022.33, val: 374, weight: 12, class: 1
[1, 0, 1, 1, 0, 0, 0, 0] >> fitness: 48720.00, val: 1218, weight: 58, class: 2
[1, 0, 1, 0, 1, 0, 0, 0] >> fitness: 57646.41, val: 1174, weight: 39, class: 2
[1, 0, 1, 0, 0, 1, 0, 0] >> fitness: 53032.79, val: 1247, weight: 53, class: 2
[1, 0, 1, 0, 0, 0, 1, 0] >> fitness: 44212.45, val: 1129, weight: 56, class: 2
[1, 0, 1, 0, 0, 0, 0, 1] >> fitness: 43134.51, val: 1201, weight: 71, class: 2

[0, 0, 1, 0, 1, 0, 0, 0] >> parent state, successors below
[1, 0, 1, 0, 1, 0, 0, 0] >> fitness: 57646.41, val: 1174, weight: 39, class: 2
[0, 1, 1, 0, 1, 0, 0, 0] >> fitness: 89510.63, val: 1283, weight: 30, class: 3
[0, 0, 0, 0, 1, 0, 0, 0] >> fitness: 9706.67, val: 320, weight: 15, class: 1
[0, 0, 1, 1, 1, 0, 0, 0] >> fitness: 44327.41, val: 1164, weight: 61, class: 2
[0, 0, 1, 0, 0, 0, 0, 0] >> fitness: 23520.00, val: 480, weight: 12, class: 1
[0, 0, 1, 0, 1, 1, 0, 0] >> fitness: 57626.16, val: 1193, weight: 56, class: 3
[0, 0, 1, 0, 1, 0, 1, 0] >> fitness: 48611.86, val: 1075, weight: 59, class: 3
[0, 0, 1, 0, 1, 0, 0, 1] >> fitness: 48747.50, val: 1147, weight: 74, class: 3

qualify:
[1, 1, 1, 0, 0, 0, 0, 0] >> fitness: 91609.26, val: 1337, weight: 27, class: 2
[0, 1, 1, 0, 1, 0, 0, 0] >> fitness: 89510.63, val: 1283, weight: 30, class: 3
[0, 1, 0, 0, 0, 0, 0, 0] >> fitness: 82110.00, val: 483, weight: 3, class: 1
[1, 1, 0, 0, 1, 0, 0, 0] >> fitness: 77956.63, val: 1177, weight: 30, class: 3
[0, 1, 1, 1, 0, 0, 0, 0] >> fitness: 71766.33, val: 1327, weight: 49, class: 3

[local beam step 2] >> current found: value: 1327, weight: 49, class: 3
```

Output: The complete run will produce the following output

```
2414
1, 1, 1, 1, 1, 1, 0, 0
```

Note:

- RED presents the change position and YELLOW presents the best K score.

3. Discussion

The empty starting method can produce a nearly optimal solution, much better than the random starting method. But in fact, this performance is not caused by the initial states, but by the successor's generate method and also the fitness function.

The random starting method makes the states jump to a bad condition instantly (exceed capacity, or high weight but low profit). Those states need to change at many positions to see the difference between them and their successors, and then can escape the bad condition. Changing just 1 position as the current method will lead the algorithm to be stuck around the random state.

The fitness function also has a problem. When we try to focus on some attributes, it makes the algorithm hard to turn back and go the other routes. This is why random starting fails. In our function, we define that the higher number of items is better to avoid the algorithm stuck in an add-remove loop. The algorithm will suffer if it starts in a bad condition.

Those are the reasons why the current methods fit with the empty starting method. It does not start in a bad condition and does not need to go back or find another route.

To make the algorithm work with random starting, we need to change both successors' generate method and fitness function. As our team have discussed and we have some suggestions for this algorithm:

- We can use greedy approach in the successors' generate method to find a further state.
- Define a fitness function that does not depend on the number of current items or total value of profit and weight. We could use rate instead of sum to make the algorithm not be biased at some already large profit but not is the optimal solution.

VI. Genetic approach

1. Algorithm

The genetic algorithm is a variant of random beam search in which successive states are generated by combining two original states rather than modifying a single state. The process of reproduction is sexual rather than asexual. Instead of going through all possible cases, this method starts with a set of k randomly states. It generates the next states based on a random combination of the two individuals in the previous state. The random selection rate will be based on the fitness function.

function GENETIC-ALGORITHM(*population*, numGen, FITNESS-FN)

returns an individual

inputs: *population*, a set of individuals

FITNESS-FN, a function that measures the fitness of an individual

```
        for gen = 1 to numGen do
            new_population ← empty set
            for i = 1 to SIZE(population) do
                x ← RANDOM-SELECTION(population, FITNESS-FN)
                y ← RANDOM-SELECTION(population, FITNESS-FN)
                child ← REPRODUCE(x , y)
                if (small random probability) then child ← MUTATE(child)
                add child to new_population
                population ← new_population
    return the best individual in population at every generation
```

a. Fitness function

For this knapsack problem, there are two conditions that we should care about: the total weight taken in does not exceed the capacity of the bag and at least one object of each class should be taken. As the question requires, we can see that the condition of getting at least one item in each class is an important condition and it is also easy to achieve because the maximum number of classes in this problem is 10. As for the requirement “not to exceed the capacity”, when the capacity is already exceeded, the results obtained are no longer important and the subsequent generations generated from this generation may not achieve valid results. Hence, based on the above 2 analyses, we tested the following two scoring methods.

The per-computation part is the same for the methods

```
function FITNESS-FN(individual)
    return score
    inputs: individual
    class_count = 0
    value = 0
    total_weight = 0
    for i:=0 to SIZE(individual) do
        if (item i is selected) then
```

```
        value += value[i]
        total_weight += weight[i]
        if (class[i] is new) then
            class_count+=1
            class[i] is old
```

Method 1: The value of an individual is equal to the $value$ of it if it is valid.

```
if (total_weight <= capacity) and (class_count = number_of_class) then
    return value
return -INF
```

Method 2: The value of an individual is equal to the $value^2/cap$ of it if it is valid. The basis for this can be represented as that: “if a way of getting the item is optimal, then the value for the same weight will also be optimal, meaning the value/cap is the highest, but to keep from getting too focused on value/weight search without optimizing value, we take the squared value to keep the end goal as high as possible”

*__*Note*__*: If cap = 0 then value = 0 we return for that case.

```
if (total_weight <= capacity) and (class_count = number_of_class) then
    return value*value/cap
return -INF
```

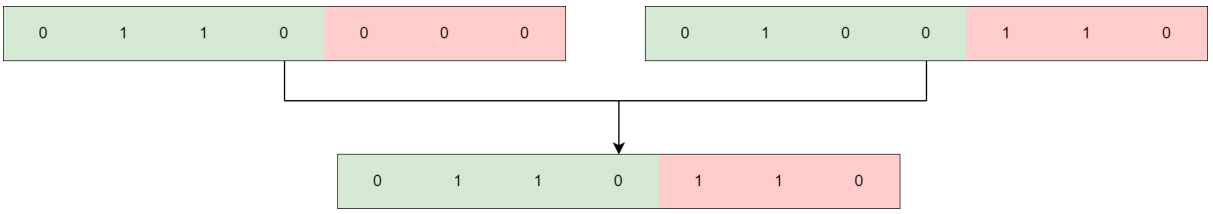
b. Random selection

In this problem, it is easy to see that the value is always >=0 for all cases. So we will use 0.001 instead of -INF to make it easier to calculate the percentage and also add 0.001 for the result to avoid cases when score = 0.

The rate to pick an individual in the set will be FITNESS/sum of all (FITNESS).

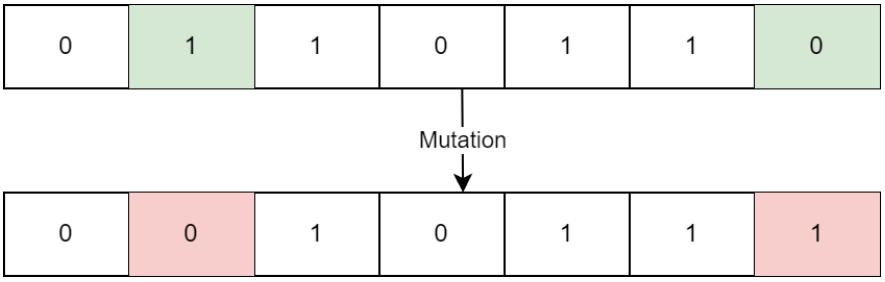
c. Re-procedure

Between 2 randomly selected individuals, choose a random location to cut those 2 individuals and then reassemble the 2 parts together.



d. Mutation

Mutation occurs at a very small rate at each selection of each item, if mutate the selection of the ith item or not will be reversed.



2. Visualization

Let's see how the algorithm works through the following example:

We will use:

- size of population = 4
- number of new generation = 2
- mutation rate = 0.5
- Use the method 2 : value^2 / total_weight

Input:

```
124
3
12, 3, 12, 34, 15, 29, 32, 47
374, 483, 480, 364, 320, 393, 275, 347
1, 2, 1, 3, 3, 2, 2, 2
```

Step:

```
---------------------------------
Generation 0:
[0, 0, 0, 0, 1, 1, 1, 1] , Score:  0.001
[1, 0, 1, 1, 0, 0, 0, 0] , Score:  0.001
[1, 1, 1, 0, 1, 0, 0, 0] , Score:  65372.59523809524
[0, 0, 1, 1, 1, 0, 1, 0] , Score:  22265.817204301075
Best result:  1657 , [1, 1, 1, 0, 1, 0, 0, 0]
*******
Choose:  [1, 1, 1, 0, 1, 0, 0, 0] [1, 1, 1, 0, 1, 0, 0, 0]
```

```
Cut at: 5
Child:              [1, 1, 1, 0, 1, 0, 0, 0]
After mutation:  [0, 1, 0, 0, 0, 1, 0, 0]
*******
Choose:  [1, 1, 1, 0, 1, 0, 0, 0] [0, 0, 1, 1, 1, 0, 1, 0]
Cut at: 6
Child:              [1, 1, 1, 0, 1, 0, 1, 0]
After mutation:  [0, 1, 1, 0, 0, 1, 0, 1]
*******
Choose:  [1, 1, 1, 0, 1, 0, 0, 0] [0, 0, 1, 1, 1, 0, 1, 0]
Cut at: 4
Child:              [1, 1, 1, 0, 1, 0, 1, 0]
After mutation:  [0, 1, 1, 0, 0, 0, 0, 1]
*******
Choose:  [1, 1, 1, 0, 1, 0, 0, 0] [1, 1, 1, 0, 1, 0, 0, 0]
Cut at: 2
Child:              [1, 1, 1, 0, 1, 0, 0, 0]
After mutation:  [0, 0, 0, 0, 0, 0, 1, 1]
---------------------------------
Generation 1:
[0, 1, 0, 0, 0, 1, 0, 0] , Score:  0.001
[0, 1, 1, 0, 0, 1, 0, 1] , Score:  0.001
[0, 1, 1, 0, 0, 0, 0, 1] , Score:  0.001
[0, 0, 0, 0, 0, 0, 1, 1] , Score:  0.001
Best result:  1657 , [1, 1, 1, 0, 1, 0, 0, 0]
*******
Choose:  [0, 1, 1, 0, 0, 1, 0, 1] [0, 1, 0, 0, 0, 1, 0, 0]
Cut at: 4
Child:              [0, 1, 1, 0, 0, 1, 0, 0]
After mutation:  [1, 0, 1, 1, 1, 1, 1, 1]
*******
Choose:  [0, 0, 0, 0, 0, 0, 1, 1] [0, 1, 1, 0, 0, 1, 0, 1]
Cut at: 6
Child:              [0, 0, 0, 0, 0, 0, 0, 1]
After mutation:  [0, 0, 1, 0, 0, 0, 1, 0]
*******
Choose:  [0, 0, 0, 0, 0, 0, 1, 1] [0, 1, 0, 0, 0, 1, 0, 0]
Cut at: 6
Child:              [0, 0, 0, 0, 0, 0, 0, 0]
After mutation:  [0, 0, 0, 1, 1, 1, 1, 1]
*******
Choose:  [0, 1, 0, 0, 0, 1, 0, 0] [0, 0, 0, 0, 0, 0, 1, 1]
Cut at: 8
Child:              [0, 1, 0, 0, 0, 1, 0, 0]
After mutation:  [1, 1, 1, 0, 1, 0, 1, 1]
---------------------------------
Generation 2:
[1, 0, 1, 1, 1, 1, 1, 1] , Score:  0.001
[0, 0, 1, 0, 0, 0, 1, 0] , Score:  0.001
[0, 0, 0, 1, 1, 1, 1, 1] , Score:  0.001
[1, 1, 1, 0, 1, 0, 1, 1] , Score:  42924.305785123965
Best result:  2279 , [1, 1, 1, 0, 1, 0, 1, 1]
```

Output:

2279
1, 1, 1, 0, 1, 0, 1, 1

Ans:

2414
1, 1, 1, 1, 1, 1, 0, 0

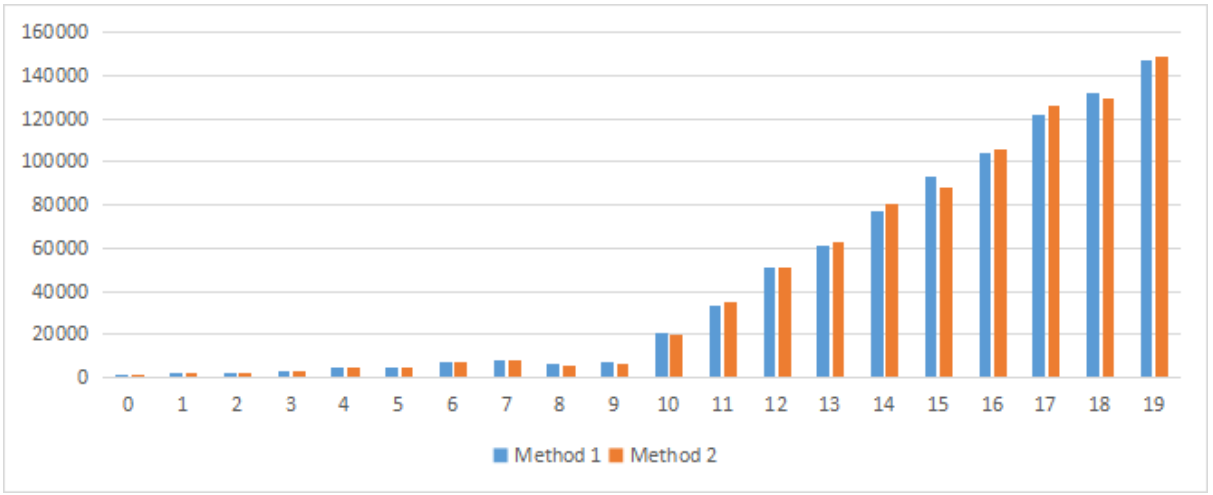
Comment:

- Green and Red represent the taken position of the 2 individuals to create a new individual and Yellow represents the mutated position.
- From generation 0, we see that the result is 1657 and at the end the generation 2 the result is 2279.
- At generation 1, it can be seen that the individuals of this generation do not satisfy the conditions, but from these individuals, it is possible to create generation 2 with better results than generation 1.

3. Experiment

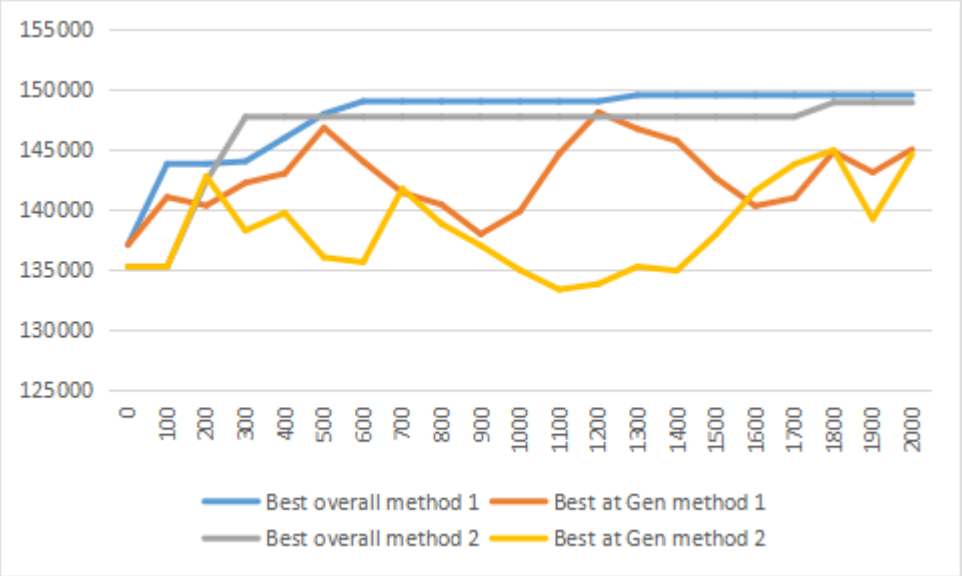
In this project, we will use the algorithm with the following parameters:

- Population size = 100
- Number of generations = 100
- Rate of mutation at each choice of the ith item = 0.001



The difference in results across 20 test cases of the two methods is not significant.

We have a chart of the run results with population size = 50, mutation = 0.001, and number of generations = 2000.



From the experiment, we see that between the two methods after a very large number of generations, the results will not differ much.

Because we can stop at the generation where individuals do not achieve good results. This chart shows why we should get the best results from all the individuals in all generations instead of the last generation.

4. Conclusion

It is easy to see that this is a Heuristic approach so the result will not be optimal. Besides that, the following generations are created from the first generation based on scores. If the first generations are not good or the FITNESS function is not good, then the results of the following generations are also not good.

This approach to the problem is only suitable when the number of results passes the condition of capacity and class exists in a large number. For cases where only very few results meet the condition, this approach will face many difficulties.

5. Some suggested improvements of our team

It should be noted that the number of classes is quite small and we can approach the problem with other directions to ignore the condition of the class.

- When creating new individuals, we always ensure there is at least one object in each class.

- Divide the problem into individual backpack problems for each class, then use the genetic algorithm for each part of that class and continue to ensure the condition of the class.

Another approach regarding capacity is that: “If a new individual has a total weight exceeding its capacity, some greedy method can be used to remove duplicate items from a class.”

VII. Comparison

The brute force and branch-bound algorithms always provide the optimal solution, but it will take time because they search almost every possible state of the problem. On the other hand, local beam and genetic search algorithms are faster but will not assure the solution is optimal because they just search for some states which have a good score based on their self-defined fitness function.

Below is a comparison of some outputs between the algorithms. On request, **weight can be a float number**, but for the convenience of testing the algorithm, we will **use weight as an integer** so that we can use an optimal solution (using dynamic programming) for comparison.

Note:

- Genetic algorithm use size of population = 500, number of generations = 500, rate of mutation = 0.001.
- Local beam (empty start) use size of population = 5~20, number of step =500.
- Local beam (random start) use size of population 50, number of step 50.
- -1 mean can't find solution.
- Empty mean it take too much time to run.
- Optimal solution is from dynamic programming approach.

We use 22 test case we have generated for comparison.

Test case (INPUT_x)	Optimal	Brute force	Branch-Bound	Local beam (random start)	Local beam (empty start)	Genetic 1 (val)	Genetic 2 (val^2/cap)
0	723	723	723	723	723	723	723
1	2414	2414	2414	2414	2414	2414	2414
2	2230	2230	2230	1803	2230	2230	2230
3	3086	3086	3086	2872	3084	3086	3084
4	4771	4771	4771	4757	4771	4771	4771
5	4656	4656	4656	3544	4656	4656	4656
6	7396	7396	7396	7310	7396	7396	7273
7	8234		8234	7213	8234	8234	8206
8	6126		6126	-1	5999	6126	6084
9	7259		7259	6554	7004	7228	7195
10	23727		23727	20500	23704	23251	21382
11	43371		43371	35400	42720	39157	40426
12	70999		70999	65705	70385	61961	63038
13	88550		88550	74911	86416	74125	78855

14	121938		121938	116062	119870	102127	97544
15	129243		129243	96831	123002	102440	107538
16	159659		159659	137505	155467	125765	123711
17	189586		189586	167991	172874	140928	142658
18	214341		214341	200873	179422	150761	156961
19	223322		223322	177583	190104	165885	170584
20	2439		2439	-1	987	-1	-1
21	436		436	-1	411	-1	436
22	5125			4881	3942	3698	3678

Analysis:

Brute force:

Brute force take much time when size is bigger

Branch and bound:

We can see that this algorithm is very fast compared to brute force.

The INPUT_22 is create to counter this algorithm where the algorithm stuck into find the way to stop because of the future value is very good.

Local beam random start:

Local beam random start method is not good in small test cases, but in large test cases, its result is approximative with genetic.

Local beam empty start:

Local beam empty start gives the result very good and approximates to the optimal solution.

Genetic 1 (val):

For the Genetic method 1 algorithm, see that when the test case is small (<=40) the result is close to the optimal solution and when the test case is much bigger the result gradually differs more from the optimal solution.

The INPUT_20 and INPUT_21 is create to counter this algorithm. INPUT_21 is a small test case but the algorithm still can't find the solution. This is the weaknes of the algorithm when the number of available solutions is small. If we run it longer or with the size of population bigger maybe the algorithm can find the solution.

Genetic 2 (val^2/cap):

The Genetic method 2 Algorithm does not give a solution for small test cases as well as method 1. But in large test cases, this method sometimes gives a much more efficient solution than method 1. This algorithm also can give the result for INPUT_21.

VIII. Conclusion

We have introduced up to 4 methods to solve the class-constrained knapsack problem. With 2 methods brute force and its improved version branch and bound will give the best results but have a big weakness in terms of time complexity. The remaining two methods, Beam

and Genetic, approach heuristics, so they will achieve the optimal side of time, but the trade-off is that the results may not be optimal.

In the comparison, we can see a clear difference between the results. If the exact search methods are Brute Force and Branch and Bound, there will be time problems, although Branch and Bound has brought a lot of improvements, it is still not enough when the results are larger. In heuristic methods like Local Beam and Genetics, they aim to find a suitable solution and gradually improve that solution. In these two methods what is important is how the new generation is generated, because both methods are random and in the worst case we may not find a possible solution. With Heuristic methods, the more memory and time resources we spend, the more optimal the results will be.

IX. References

1. A Survey of the Knapsack Problem
2. A branch and bound algorithm for solving the multiple-choice knapsack problem
3. A hybrid dynamic programming/branch-and-bound algorithm for the multiple-choice knapsack problem
4. Implementation of 0/1 Knapsack using Branch and Bound
5. Binary Search - GeeksforGeeks