UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

PROJECT 1: Searching For The Knapsack

SUBJECT: Introduction to Artificial Intelligence

Class: 21CLC03

Students: Trần Nguyên Huân – 21127050, Lê Huỳnh Phúc – 21127392
Nguyễn Hoàng Phúc – 21127671, Nguyễn Phú Trọng – 21127708

Instructors: Nguyễn Ngọc Thảo, Lê Ngọc Thành,
Nguyễn Trần Duy Minh, Nguyễn Hải Đăng

HO CHI MINH CITY – 2023

Contents

I. Tasks assignment table, completion level

1. Student's information and Tasks assignment, completion level

Full Name	Student Id	Tasks	Completion Level
Trần Nguyên Huân	21127050	+ Do research about 4 algorithms. + Implement brute force search and local beem search algorithms.	100%
Lê Huỳnh Phúc	21127392	+ Do research about 4 algorithms. + Record video process of running algorithms with small datasets and big datasets. + Write report part test case	100%
Nguyễn Hoàng Phúc	21127671	+ Do research about 4 algorithms. + Implement branch and bound, genetic algorithms	100%
Nguyễn Phú Trọng	21127708	+ Do research about 4 algorithms. + Write report part algorithm description and performance of those algorithms	100%

2. Evaluate the level of accomplishment

No.	Criteria	Completed
1	Algorithm 1	✓
2	Algorithm 2	✓

3	Algorithm 3	✓
4	Algorithm 4	✓
5	Generate at least 5 small datasets of sizes 10-40. Compare with performance in the experiment section. Create videos to show your implementation.	✓
6	Generate at least 5 large datasets of sizes 50-1000 with number of classes of 5-10. Compare with performance in the experiment section. Create videos to show your implementation.	✓
7	Report your algorithms, experiments with some reflection or comments.	✓

II. Algorithms Description

1. Brute Force Searching

1.1. Description

- Brute force searching is a method of solving problems or searching for a solution by exhaustively trying every possible option until a satisfactory result is found. It involves systematically generating and evaluating all possible solutions without relying on any particular algorithm or optimization technique.

1.2. Pseudo code

```
1. Initialize max_value and max_subset to 0 and None respectively
2. Loop through all possible subsets of items, including the empty subset
    a. Check if the subset contains at least one item from each class. If
       not, skip to the next subset.
    b. Calculate the total value and weight of the subset.
    c. If the total weight is less than or equal to the capacity and the total
       value is greater than the current max_value, update max_value and
       max_subset with the values of the current subset.
3. Return max_value and max_subset.
```

1.3. Explanation each function in algorithm

- **knapsack_brute_force_searching(values, weights, capacity, class_labels)**: This function takes as input the lists of item values, weights, class labels, and the capacity of the knapsack. It uses brute force searching to find the subset of items that maximizes the total value of items in the knapsack, subject to the capacity constraint and the requirement that at least one item from each class must be included. It returns the maximum value, the indices of the items in the optimal subset, and the class labels of the items in the subset.

- **read_input_file(filename)**: This function reads the input file specified by the filename argument and returns the capacity, number of classes, weights, values, and class labels of the items in the knapsack.
- **write_output_file(filename, max_value, max_subset, num_items)**: This function writes the output to the file specified by the filename argument. It writes the maximum value, the indices of the items in the optimal subset, and the number of items to the file.
- **For example, let's say we have a Knapsack that can hold a maximum weight of 10 and the following items:**

1.4. Visualization algorithm

- For example, let's say we have a Knapsack that can hold a maximum weight of 10 and the following items:

Item	Weight	Value	Class Label
A	5	10	1
B	3	7	2
C	2	5	1
D	1	3	3

- To apply the brute force algorithm with the additional class label constraint, we would consider all possible combinations of items that can fit in the Knapsack and contain at least one item from each class label:

Subset	**Items**	**Weight**	**Value**
{}		0	0
{A, B, C}	A, B, C	10	22
{A, B, D}	A, B, D	9	20
{A, C, D}	A, C, D	8	18

{B, C, D}	B, C, D	6	15
{A, B, C, D}	A, B, C, D	11	25

➔ So, we have generated all possible subsets satisfying the constraint at least one item from each class and calculated the value and weight of each subset. The largest-valued subset and and does not exceed the value of capacity is {A,B,C} with a value of 22.

2. Branch And Bound

2.1. Description

- The branch and bound algorithm is a commonly used method for solving combinatorial optimization problems, including the knapsack problem. The basic idea behind the algorithm is to explore the search space of possible solutions by systematically partitioning it into smaller and smaller subsets, until an optimal solution is found or it is determined that no better solution can be found.

2.2. Pseudo code

```
1. Sort items in descending order of value-to-weight ratio.

2. Initialize an empty heap queue (min-heap) and create the root node with
   level = -1, weight = 0, value = 0, bound = 0, item_set = [], and
   item_classes = set().

3. Compute the bound of the root node and push it onto the heap queue.

4. Initialize max_value = 0 and best_set = [].

5. While the heap queue is not empty:
   a. Pop the node with the lowest bound from the heap queue.
   b. If the bound of the current node is greater than max_value and the level
   of the current node is less than len(items) - 1:
         i. Increment the level by 1 and get the next item.
         ii. Create a child node by adding the item to the current node and
   update its weight, value, and item_set accordingly.
         iii. If the child node satisfies the constraints (weight <= W and
   len(item_classes) == m) and its value is greater than max_value, update
   max_value and best_set.
```

iv. Compute the bound of the child node and push it onto the heap queue if its bound is greater than max_value.

v. Create another child node by not adding the item to the current node.

vi. Compute the bound of the second child node and push it onto the heap queue if its bound is greater than max_value.

6. Return max_value and best_set.

2.3. Explanation each function in algorithm

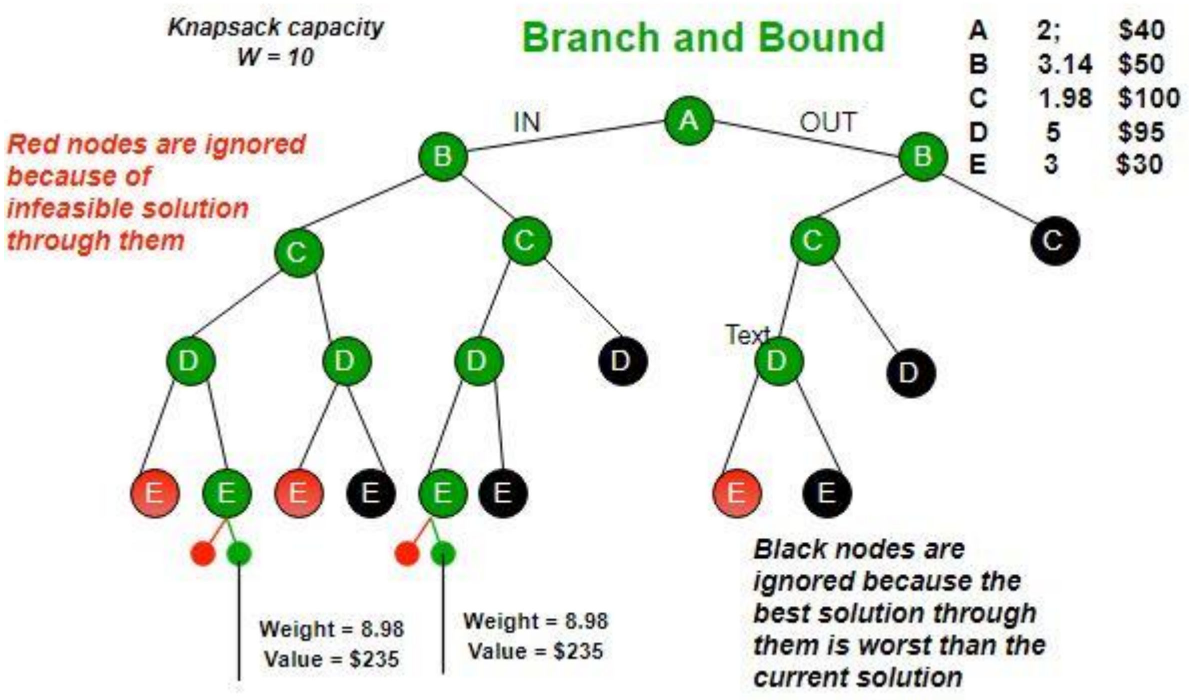
- **Item**: A named tuple that represents an item in the knapsack problem. It has four fields: index, weight, value, and item_class. The index is a unique identifier for the item, the weight is its weight, the value is its value, and the item_class is a category that the item belongs to.
- **Node**: A class that represents a node in the search tree. It has five fields: level, weight, value, bound, item_set, and item_classes. The level is the level of the node in the search tree (i.e., the index of the item that the node represents), the weight is the total weight of the items in the node's item_set, the value is the total value of the items in the node's item_set, the bound is the node's upper bound, the item_set is a list of the indices of the items in the node, and the item_classes is a set that contains the classes of the items in the node.
- **__lt__**: A method of the Node class that defines the less than operation for nodes. This is used to compare nodes in the heap queue.
- **bound**: A function that computes the upper bound of a node. It takes a node, the capacity of the knapsack, and a list of items as arguments. The function calculates the maximum possible value of the node by adding items to the node's item_set until the knapsack is full. If the node is not at the last level of the search tree, the function also includes a fractional item based on the value-to-weight ratio of the next item in the list. The function returns the calculated upper bound.
- **branch_and_bound**: The main function that performs the branch and bound algorithm. It takes the capacity of the knapsack, the number of distinct item classes, and a list of items as arguments.

  + The function first sorts the items in descending order of value-to-weight ratio. It then initializes a heap queue and creates the root node. The function computes the upper bound of the root node and pushes it onto the heap queue. The function also initializes max_value and best_set to 0 and an empty list, respectively.

+ The function then enters a loop where it pops the node with the lowest bound from the heap queue. If the bound of the current node is greater than max_value and the level of the current node is less than len(items) - 1, the function creates two child nodes: one with the next item added to the node's item_set and one without the next item. If the child node with the next item satisfies the constraints and has a greater value than max_value, the function updates max_value and best_set. The function then calculates the upper bounds of both child nodes and pushes them onto the heap queue if their bounds are greater than max_value.

+ The loop continues until the heap queue is empty. The function returns the maximum value and the set of indices of the items that achieve the maximum value.

2.4. Visualization algorithm


Knapsack capacity
W = 10
Branch and Bound
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
IN
OUT
A
B
C
D
E
Text
Red nodes are ignored because of infeasible solution through them
Weight = 8.98
Value = $235
Weight = 8.98
Value = $235
Black nodes are ignored because the best solution through them is worst than the current solution

Source: https://www.geeksforgeeks.org/branch-and-bound-algorithm/

3. Local Beam Search

3.1. Description

- Local beam search is a heuristic search algorithm used for solving optimization problems. It is similar to the hill-climbing algorithm but has the advantage of being able to explore multiple paths in the search space simultaneously.

3.2. Pseudo code

1. Define the local_beam_search function with the parameters W, m, weights, values, classes, k=10, max_iterations=100.
2. Generate k initial states randomly using the "generate_initial_states" function.
3. Evaluate the initial states using the "evaluate" function and determine the best state and its value.
4. While the number of iterations is less than max_iterations, do the following:
    a. Generate the neighbors of the current states using the "get_neighbors" function.
    b. Evaluate the neighbors using the evaluate function and select the k best states.
    c. Update the current states with the k best states.
    d. Determine the current best state and its value.
    e. If the current best value is greater than the best value, update the best state and its value.
5. Return the best value and best state.

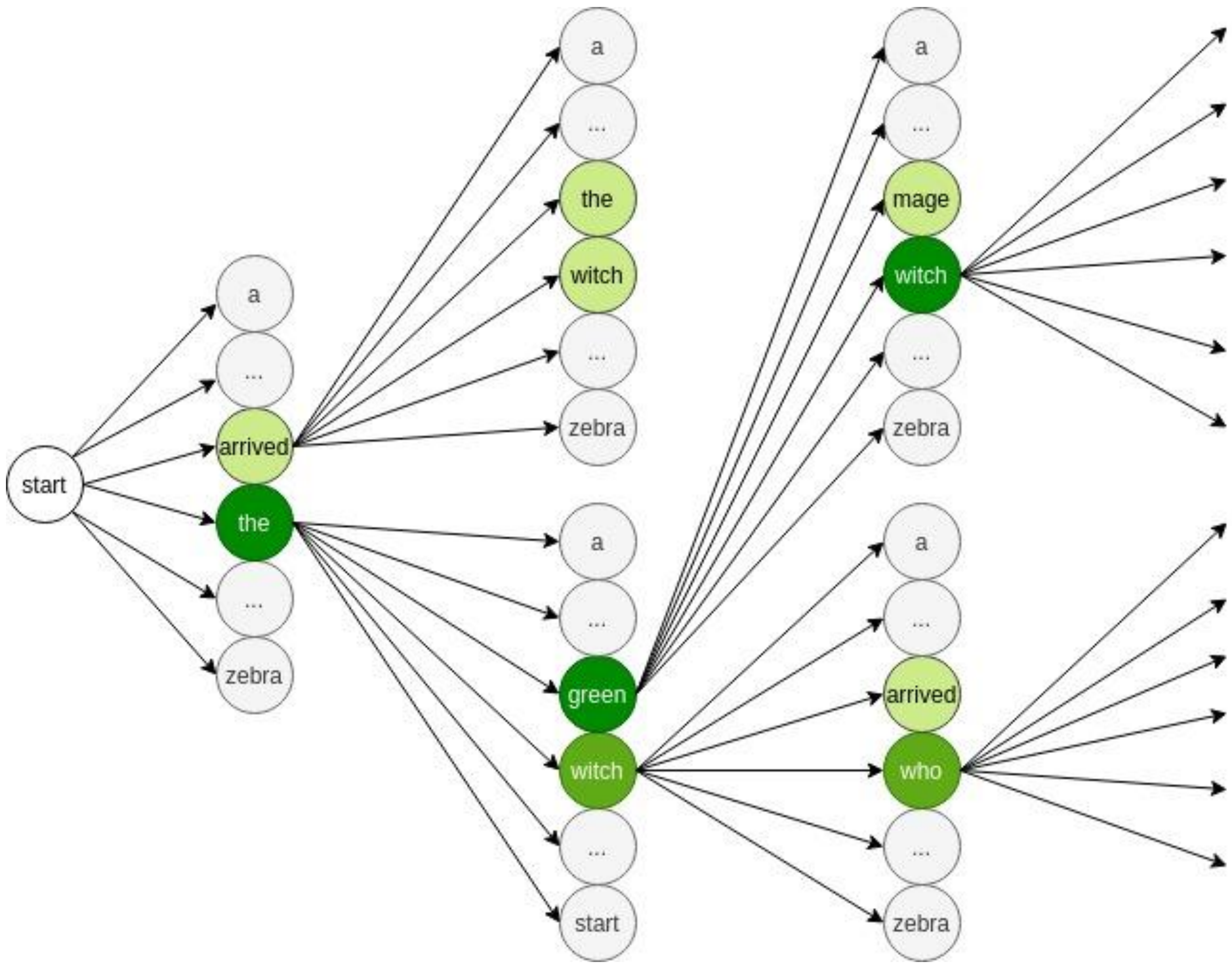
3.3. Explanation each function in algorithm

- **local_beam_search(W, m, weights, values, classes, k=10, max_iterations=100)**: This function implements the Local Beam Search algorithm. It takes as input the knapsack capacity (W), the number of classes (m), the weights of the items (weights), the values of the items (values), the class labels of the items (classes), the number of beams (k), and the maximum number of iterations (max_iterations). It generates k random initial states and evaluates their fitness using the evaluate function. Then, it generates the next states by flipping one bit at a time in the current states and selects the k best states based on their fitness. It repeats this process for max_iterations iterations or until the best state has not improved in the last k iterations. Finally, it returns the fitness value and the state of the best solution found.

- **generate_initial_states()**: This function generates k random initial states. It flips each bit with a probability of 0.5.
- **evaluate(state)**: This function evaluates the fitness of a given state. It calculates the total weight and total value of the selected items and checks whether each class has at least one item selected. If the total weight exceeds the knapsack capacity or any class has no item selected, the fitness is 0. Otherwise, it returns the total value.
- **get_neighbors(state)**: This function generates the neighboring states of a given state. It flips one bit at a time in the state and returns the resulting states.
- **read_input_file(file_path)**: This function reads the input file and returns the relevant parameters. It takes as input the path to the input file and returns the knapsack capacity (W), the number of classes (m), the weights of the items (wi), the values of the items (vi), and the class labels of the items (ci).
- **write_output_file(file_path, max_value, items_in_knapsack)**: This function writes the output file. It takes as input the path to the output file, the maximum fitness value (max_value), and the state of the best solution found (items_in_knapsack). It writes the maximum fitness value and the indices of the selected items separated by commas.

3.4. Visualization algorithm

- To understand, let's try a simplified sample decoding process step by step.



Source: https://www.baeldung.com/cs/beam-search

- To decode a sequence, we can use a heuristic function that calculates the probabilities of words and phrases in the target language.
- We start by initializing a sequence with a beginning token and selecting the B most probable words from the vocabulary. For example, if B = 2, we might choose the words "arrived" and "the" as our initial words.
- Next, we expand the sequence by adding new words based on their probabilities. The words with the highest probabilities will be ({arrived the, arrived witch, the green, the witch, …}). From these possible paths, we choose the two most probable ones ({the green, the witch}). Now we expand these two and get other possible combinations ({the green witch, the green mage, the witch arrived, the witch who}). Once again, we select two words that maximize the probability of the current sequence ({the green witch, the witch who}). From the possible paths, we choose the B most probable ones and continue to expand them until we reach the end of the sequence.

➔ By choosing a small B, we might overlook some more likely paths due to long-term dependencies in natural languages. To address this issue, we can increase the beam width and consider more possibilities. Ultimately, we select the most probable translation sequence based on our heuristic function.

4. Genetic Algorithm

4.1. Description

- Genetic algorithm is a type of optimization algorithm that is inspired by the process of natural selection in genetics. It is used to find approximate solutions to optimization problems where finding the exact solution is computationally infeasible or impossible.

4.2. Pseudo code

1. Initialize population:
- Randomly generate a set of chromosomes (potential solutions) with random values for decision variables.
2. Evaluate fitness:
- Evaluate the fitness function for each chromosome to determine how well it solves the problem.
3. Selection:
- Select the best chromosomes (based on fitness) to reproduce and create the next generation.
4. Crossover:
- Create new offspring by combining segments of chromosomes from selected parents.
5. Mutation:
- Introduce random mutations in the chromosomes to create additional diversity in the population.
6. Evaluation:
- Evaluate the fitness of the new offspring.
7. Replacement:
- Replace weaker members of the population with the new offspring.
8. Termination:
- Repeat steps 2-7 for multiple generations or until a satisfactory solution is found.

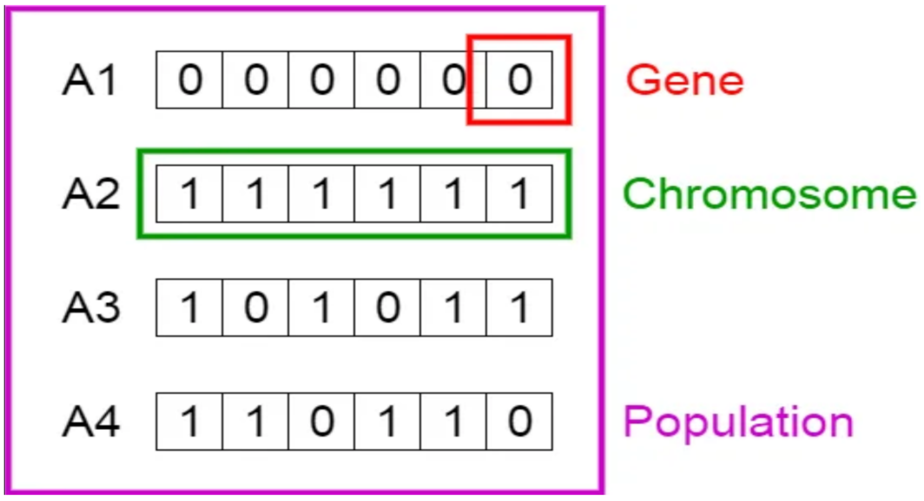
4.3. Explanation each function in algorithm

- **Item**: This is a simple class to hold the weight, value, and class of an item in the knapsack.
- **read_input_file(file_name)**: This function reads the input file and returns the knapsack capacity, the number of classes, and a list of Item objects.
- **fitness(solution, W, m, items)**: This function calculates the fitness value of a solution, which is the total value of the selected items if their total weight is less than or equal to the knapsack capacity and the number of classes they belong to is at least m. If the solution is invalid, the function returns -1.
- **generate_population(population_size, n)**: This function generates a population of population_size solutions, each with n bits randomly set to 0 or 1.
- **selection(population, W, m, items)**: This function performs fitness-based selection on the population, returning the top half of solutions sorted by fitness value.
- **crossover(parents)**: This function performs single-point crossover on pairs of adjacent parents in the input list. It returns a list of offspring solutions.
- **mutation(offspring, mutation_rate)**: This function applies a bit-flip mutation to each bit in each offspring solution with a probability of mutation_rate. It returns the mutated offspring solutions.

4.4. Visualization algorithm

❖ **Initial Population**

- A population is the initial set of individuals in a genetic algorithm, where each individual represents a possible solution to the problem being solved.
- An individual is defined by a set of parameters, known as genes, which are combined into a chromosome that represents the solution.
- An individual is defined by a set of parameters, known as genes, which are combined into a chromosome that represents the solution.

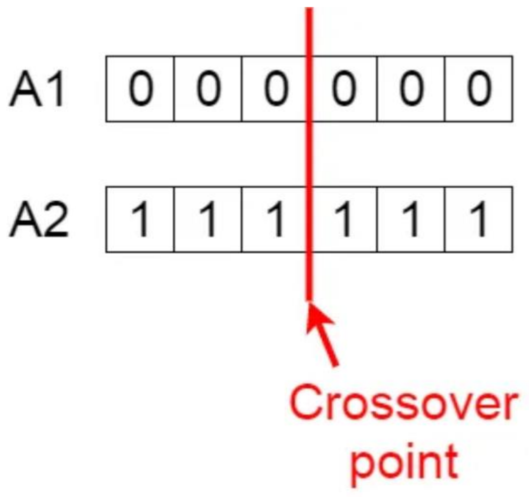


Population, Chromosomes and Genes

Source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

- ❖ **Fitness Function:** The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The probability that an individual will be selected for reproduction is based on its fitness score.
- ❖ **Selection:** The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.
- ❖ **Crossover:** Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes. rewrite the following paragraph.

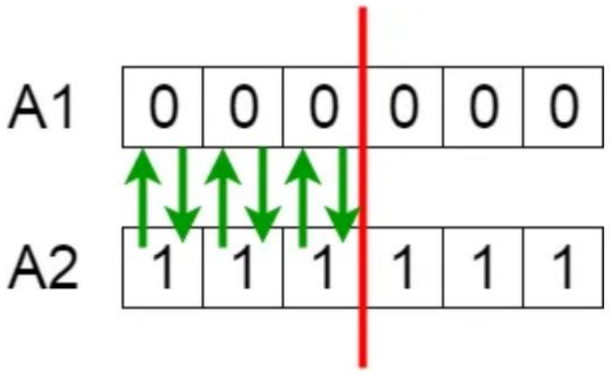
➢ For example, consider the crossover point to be 3 as shown below.



Crossover point

Source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

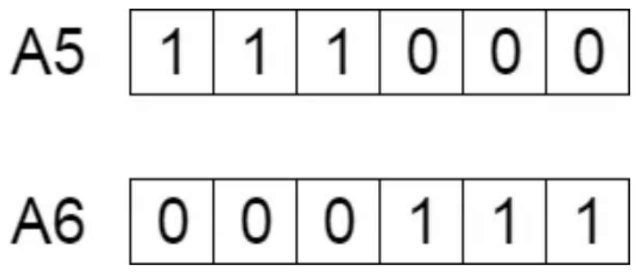
❖ **Offspring** are created by exchanging the genes of parents among themselves until the crossover point is reached.



Exchanging genes among parents

Source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

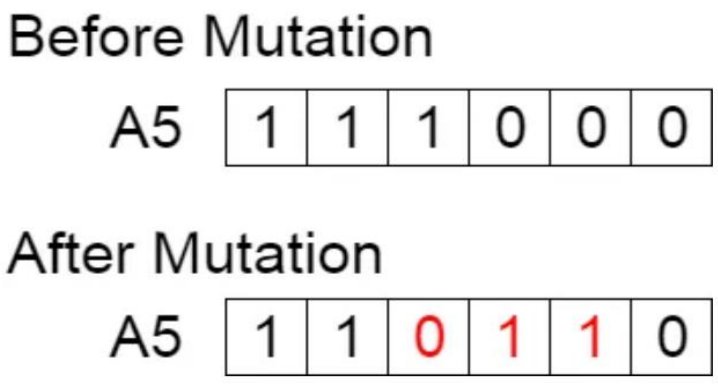
➢ The new offspring are added to the population.



New offspring

Source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

- **Mutation:** Newly produced offspring may undergo a random mutation that affects some of their genes. This mutation occurs with a low probability and can result in certain bits within the corresponding bit string being flipped.



Mutation: Before and After

Source: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

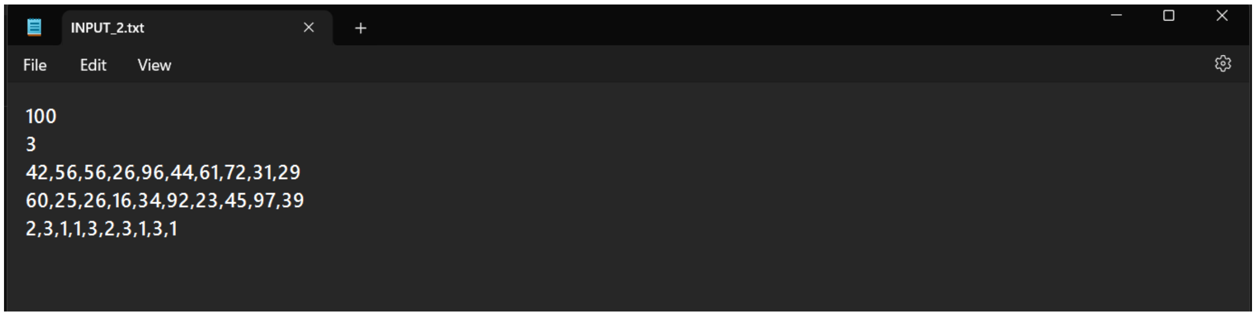
- Mutation occurs to maintain diversity within the population and prevent premature convergence.

- **Termination**: If the population reaches a state where newly produced offspring are no longer significantly different from the previous generation, the algorithm will stop running as it has converged. At this point, it can be concluded that the genetic algorithm has generated a set of solutions to the given problem.

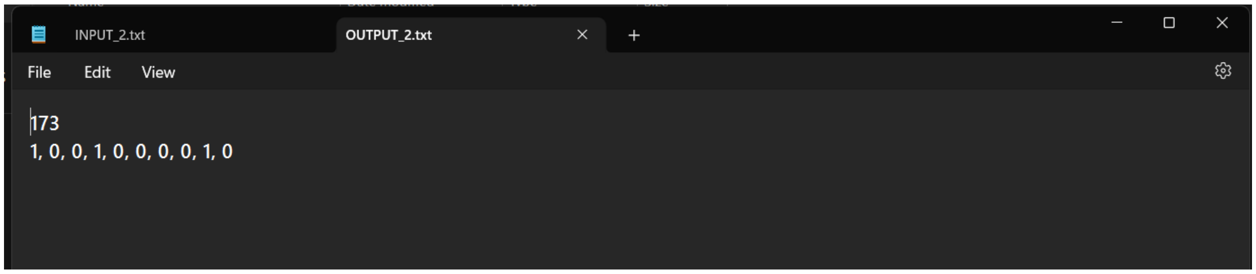
III. Test cases

1. Brute Force Searching

1.1. Test algorithm with small datasets (sizes 10 - 40)

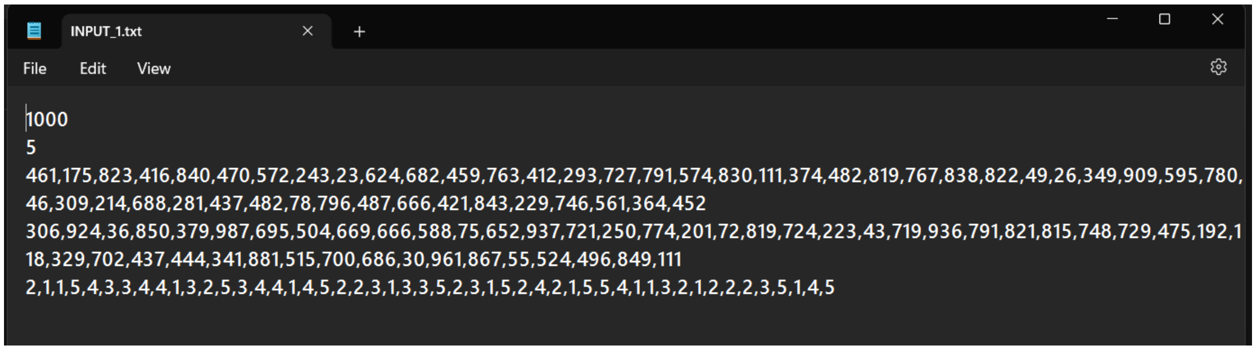


```
100
3
42,56,56,26,96,44,61,72,31,29
60,25,26,16,34,92,23,45,97,39
2,3,1,1,3,2,3,1,3,1
```



```
173
1, 0, 0, 1, 0, 0, 0, 0, 1, 0
```

1.2. Test algorithm with big datasets (sizes 50 - 1000)



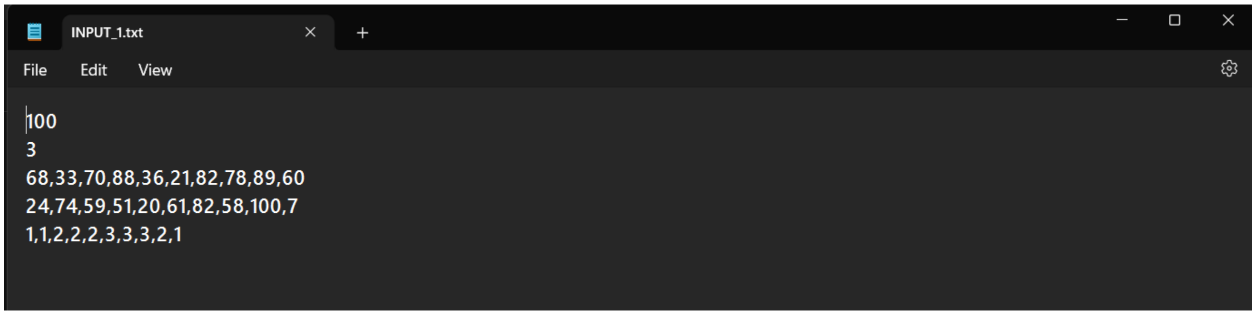
```
1000
5
461,175,823,416,840,470,572,243,23,624,682,459,763,412,293,727,791,574,830,111,374,482,819,767,838,822,49,26,349,909,595,780,
46,309,214,688,281,437,482,78,796,487,666,421,843,229,746,561,364,452
306,924,36,850,379,987,695,504,669,666,588,75,652,937,721,250,774,201,72,819,724,223,43,719,936,791,821,815,748,729,475,192,1
18,329,702,437,444,341,881,515,700,686,30,961,867,55,524,496,849,111
2,1,1,5,4,3,3,4,4,1,3,2,5,3,4,4,1,4,5,2,2,3,1,3,3,5,2,3,1,5,2,4,2,1,5,5,4,1,1,3,2,1,2,2,2,3,5,1,4,5
```

***OUTPUT:**

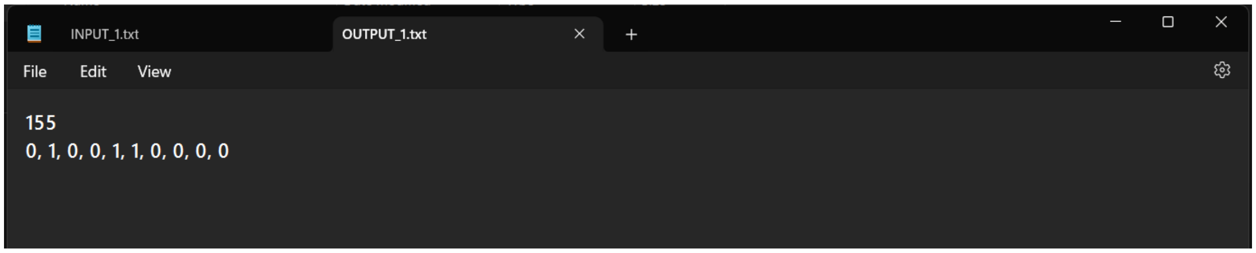
Intractable

2. Branch And Bound

2.1. Test algorithm with small datasets (sizes 10 - 40)

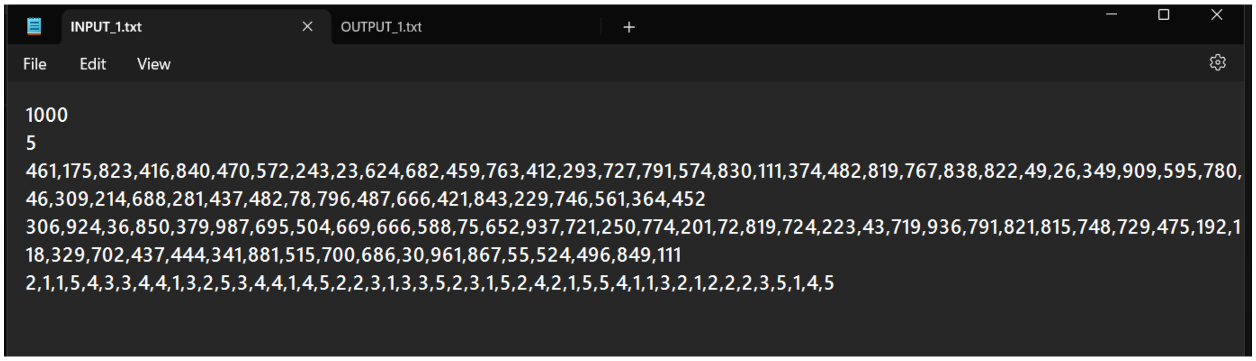


```
100
3
68,33,70,88,36,21,82,78,89,60
24,74,59,51,20,61,82,58,100,7
1,1,2,2,2,3,3,3,2,1
```

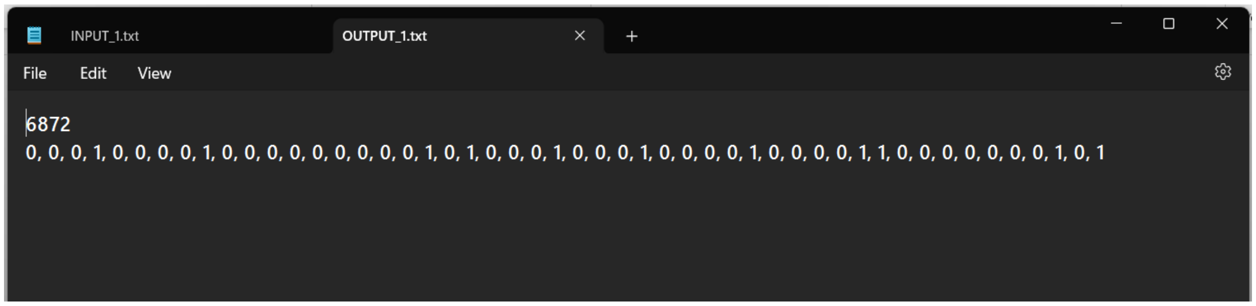


```
155
0, 1, 0, 0, 1, 1, 0, 0, 0, 0
```

2.2. Test algorithm with big datasets (sizes 50 - 1000)



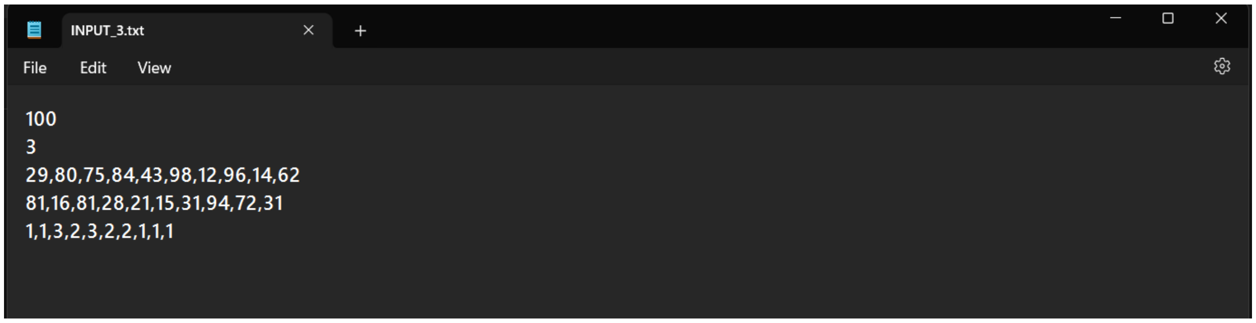
```
1000
5
461,175,823,416,840,470,572,243,23,624,682,459,763,412,293,727,791,574,830,111,374,482,819,767,838,822,49,26,349,909,595,780,
46,309,214,688,281,437,482,78,796,487,666,421,843,229,746,561,364,452
306,924,36,850,379,987,695,504,669,666,588,75,652,937,721,250,774,201,72,819,724,223,43,719,936,791,821,815,748,729,475,192,1
18,329,702,437,444,341,881,515,700,686,30,961,867,55,524,496,849,111
2,1,1,5,4,3,3,4,4,1,3,2,5,3,4,4,1,4,5,2,2,3,1,3,3,5,2,3,1,5,2,4,2,1,5,5,4,1,1,3,2,1,2,2,2,3,5,1,4,5
```



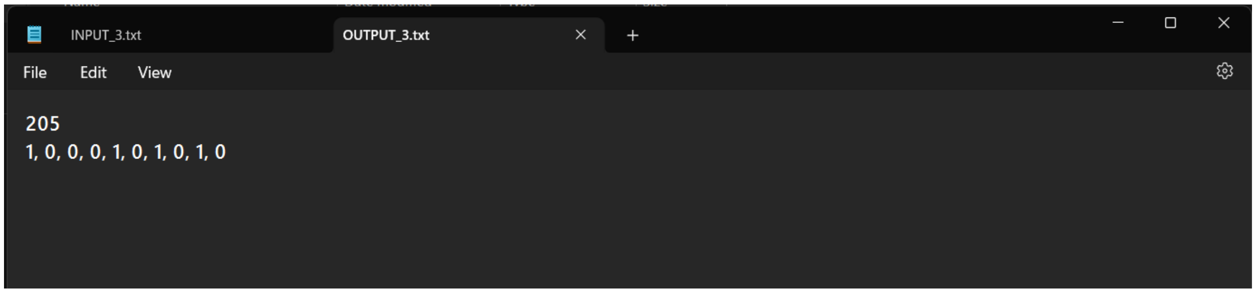
```
6872
0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1
```

3. Local Beam Search

3.1. Test algorithm with small datasets (sizes 10 - 40)

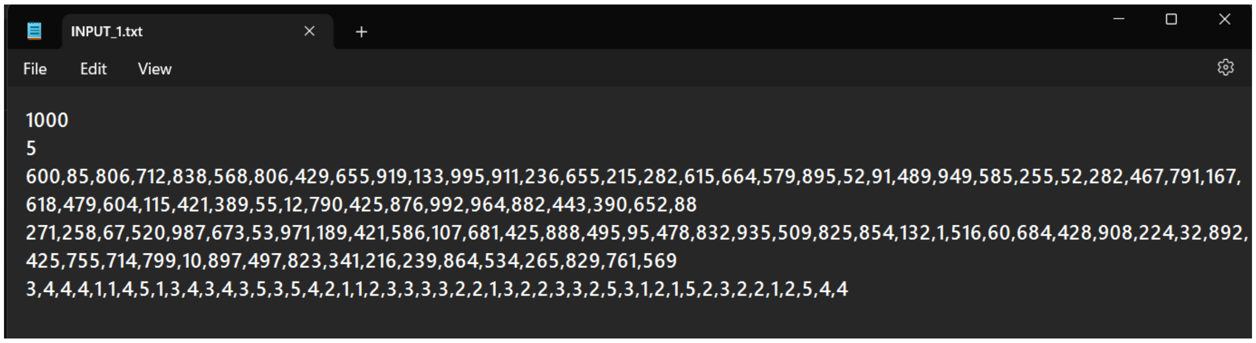


```
100
3
29,80,75,84,43,98,12,96,14,62
81,16,81,28,21,15,31,94,72,31
1,1,3,2,3,2,2,1,1,1
```



```
205
1, 0, 0, 0, 1, 0, 1, 0, 1, 0
```

3.2 Test algorithm with big datasets (sizes 50 - 1000)



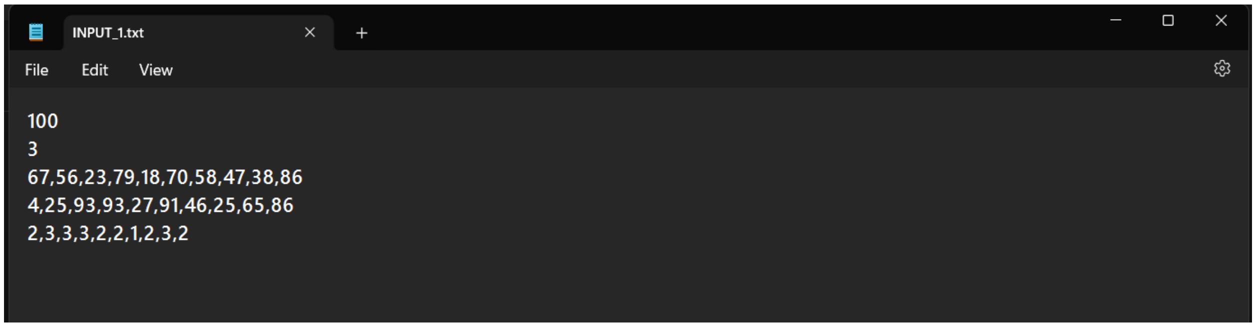
```
1000
5
600,85,806,712,838,568,806,429,655,919,133,995,911,236,655,215,282,615,664,579,895,52,91,489,949,585,255,52,282,467,791,167,
618,479,604,115,421,389,55,12,790,425,876,992,964,882,443,390,652,88
271,258,67,520,987,673,53,971,189,421,586,107,681,425,888,495,95,478,832,935,509,825,854,132,1,516,60,684,428,908,224,32,892,
425,755,714,799,10,897,497,823,341,216,239,864,534,265,829,761,569
3,4,4,4,1,1,4,5,1,3,4,3,4,3,5,3,5,4,2,1,1,2,3,3,3,3,2,2,1,3,2,2,3,3,2,5,3,1,2,1,5,2,3,2,2,1,2,5,4,4
```

***OUTPUT:**

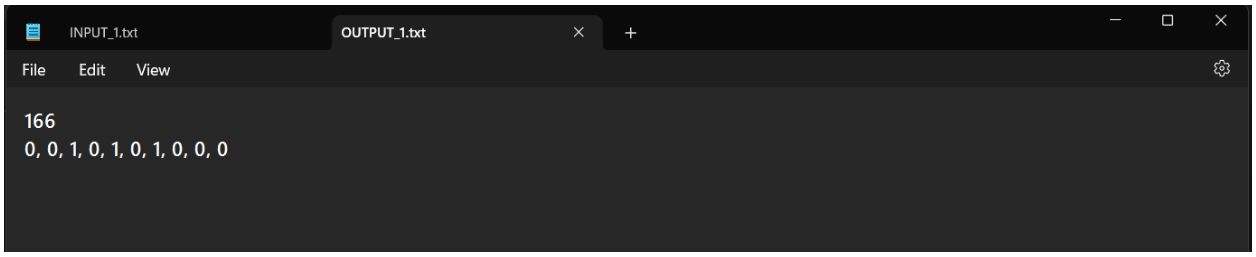
Intractable

4. Genetic Algorithms

4.1. Test algorithm with small datasets (sizes 10 - 40)

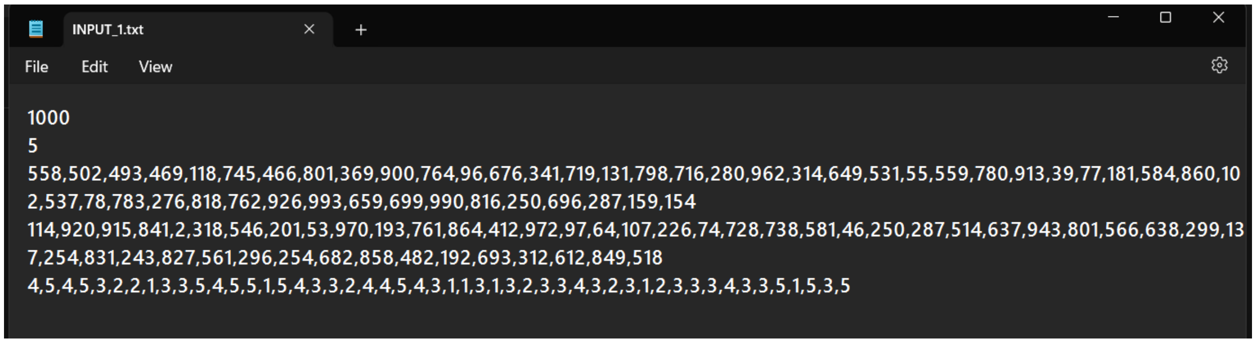


```
100
3
67,56,23,79,18,70,58,47,38,86
4,25,93,93,27,91,46,25,65,86
2,3,3,3,2,2,1,2,3,2
```



```
166
0, 0, 1, 0, 1, 0, 1, 0, 0, 0
```

4.2. Test algorithm with big datasets (sizes 50 - 1000)



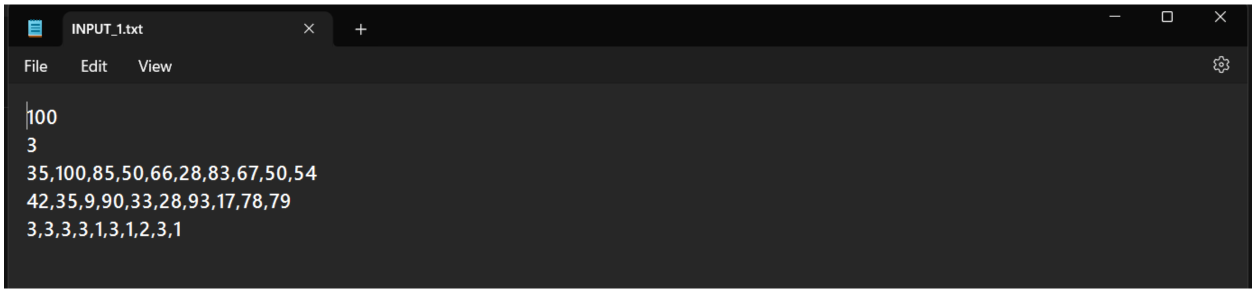
```
1000
5
558,502,493,469,118,745,466,801,369,900,764,96,676,341,719,131,798,716,280,962,314,649,531,55,559,780,913,39,77,181,584,860,10
2,537,78,783,276,818,762,926,993,659,699,990,816,250,696,287,159,154
114,920,915,841,2,318,546,201,53,970,193,761,864,412,972,97,64,107,226,74,728,738,581,46,250,287,514,637,943,801,566,638,299,13
7,254,831,243,827,561,296,254,682,858,482,192,693,312,612,849,518
4,5,4,5,3,2,2,1,3,3,5,4,5,5,1,5,4,3,3,2,4,4,5,4,3,1,1,3,1,3,2,3,3,4,3,2,3,1,2,3,3,3,4,3,3,5,1,5,3,5
```

***OUTPUT:**

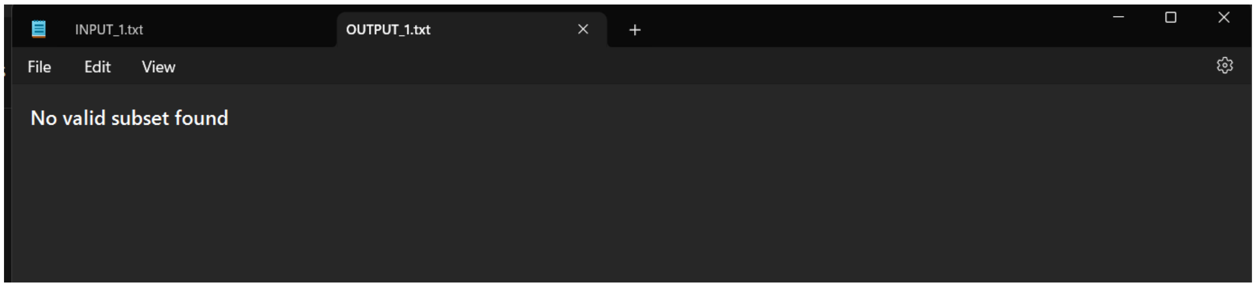
Intractable

5. Error Test Case ("No valid subset found")

- Case with Local Beam: could not find a satisfactory subset in the search range



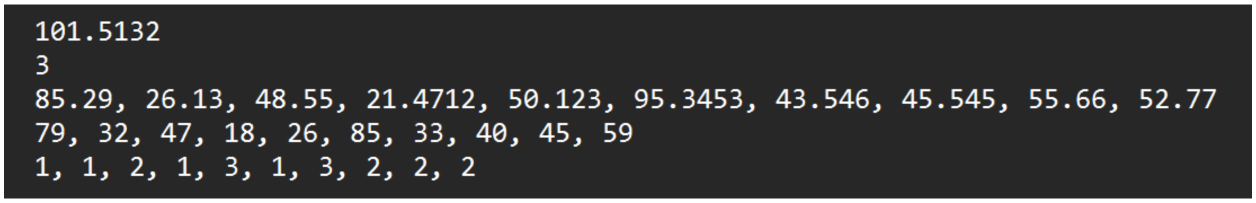
```
100
3
35,100,85,50,66,28,83,67,50,54
42,35,9,90,33,28,93,17,78,79
3,3,3,3,1,3,1,2,3,1
```



```
No valid subset found
```

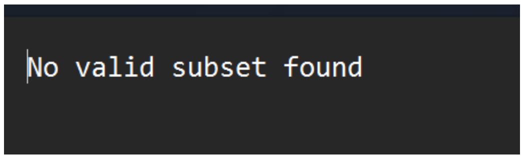
- Case: there is no sum of subsets where at least one element from each class is less or equal than weight

* Input:



```
101.5132
3
85.29, 26.13, 48.55, 21.4712, 50.123, 95.3453, 43.546, 45.545, 55.66, 52.77
79, 32, 47, 18, 26, 85, 33, 40, 45, 59
1, 1, 2, 1, 3, 1, 3, 2, 2, 2
```

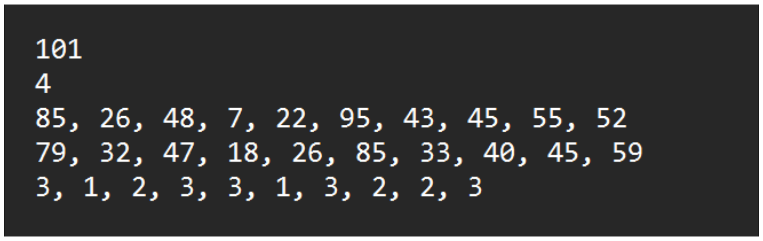
* Output:



```
No valid subset found
```

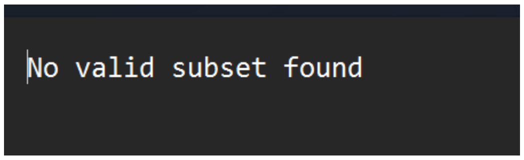
- ➢ Case: not enough classes have class_label from 1 to m (num_classes) (for example in below m = 4 while class_label has value from 1 to 3)

* Input:



```
101
4
85, 26, 48, 7, 22, 95, 43, 45, 55, 52
79, 32, 47, 18, 26, 85, 33, 40, 45, 59
3, 1, 2, 3, 3, 1, 3, 2, 2, 3
```

* Output:



```
No valid subset found
```

6. Video

https://youtu.be/Pc2bbwwQ9gM

IV. Performance of those algorithms

1. Comparision

Algorithm	Time Complexity	Space Complexity	Exact/ Approximate	Strengths	Weaknesses
Brute Force	O(2^n)	O(1)	Exact	Simple to implement	Impractical for large

					problem instances
Brand and Bound	O(2^n)	O(n)	Exact	Can provide exact solution for small instances	Time complexity can still be exponential
Local Beam	O(kn)	O(kn)	Approximate	Can converge quickly to local optimum	May miss global optimum
Genetic	O(gkn)	O(gkn)	Approximate	Can explore search space efficently	Convergence to global optimum not guaranteed

- "n" represents the number of items.
- "k" represents the population size, or the number of candidate solutions in each generation.
- "g" represents the number of generations in the genetic algorithm.

2. Evalution

- To evaluate the performance of the algorithms for solving the Knapsack problem, several metrics can be used, including the quality of the solution, the time taken to find the solution, and the memory used by the algorithm (as shown in the table)
  - Brute Force: The brute force algorithm is simple to implement, but its time complexity grows exponentially with the number of items in the Knapsack, which makes it impractical for large problem instances. However, for small problem instances, it provides an exact solution to the Knapsack problem. The quality of the solution obtained by the brute force algorithm is guaranteed to be optimal, but the time taken to find the solution may be prohibitively long.

  - Branch and Bound: The branch and bound algorithm can provide an exact solution to the Knapsack problem for small problem instances by pruning the search space. However, its time complexity can still be exponential in the worst case, which limits its usefulness for large problem instances. The quality of the solution obtained by the branch and bound algorithm is optimal, but the time taken to find the solution can be high.

- ➢ Local Beam: The local beam search algorithm can converge quickly to a local optimum, but it may miss the global optimum. Its time complexity is polynomial, making it more efficient than brute force and branch and bound for large problem instances. However, the quality of the solution obtained by the local beam search algorithm is approximate, and the algorithm may get stuck in local optimal.

- ➢ Genetic: The genetic algorithm can explore the search space efficiently and converge to a good solution for large problem instances. Its time complexity is also polynomial, making it suitable for large problem instances. However, the quality of the solution obtained by the genetic algorithm is approximate, and convergence to the global optimum is not guaranteed. The memory used by the genetic algorithm can also be high due to the population size and the number of generations.

3. Comments

- ❖ Brute force searching is a simple and straightforward algorithmic approach to solving problems that involves generating all possible solutions and selecting the best one.
    - It is a generic method and not limited to any specific domain of problems.
    - The main advantage of the brute force approach is that it guarantees finding the optimal solution since it explores the entire search space.
    - Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.
    - However, the major disadvantage is its computational complexity, which grows exponentially with the problem size. As a result, brute force searching is only practical for problems with small input sizes or when the time complexity is not a critical factor.

      ➔ Despite its limitations, brute force searching is still widely used in practice, especially when dealing with small datasets or when the problem has a limited number of solutions.
- ❖ Branch and bound is an improvement over the brute force algorithm for solving the Knapsack problem.

- The time complexity of branch and bound is O(2^n), the same as brute force. However, the algorithm can provide an exact solution for small problem instances by efficiently pruning the search space.
- The space complexity of branch and bound is O(n) because the algorithm needs to keep track of the current subset and the remaining items at each node.
- The quality of the solution obtained by the Branch and Bound algorithm depends on the quality of the upper bounds used to prune the search space. Better upper bounds can improve the efficiency of the algorithm by pruning more nodes and reducing the search space.

➔ Branch and bound is very useful technique for searching a solution but in worst case, we need to fully calculate the entire tree. At best, we only need to fully calculate one path through the tree and prune the rest of it. (geeksforgeeks)

❖ The Local Beam Search algorithm is a heuristic search algorithm for solving optimization problems.
  - The time complexity of Local Beam Search is O(kn) per iteration, where k is the size of the beam and n is the number of items in the Knapsack. The space complexity is also O(kn) because the algorithm needs to store the current beam and its neighborhood.
  - One advantage of Local Beam Search over other algorithms like brute force or branch and bound is that it can converge quickly to a local optimum, which can be a good enough solution for many practical applications. Another advantage is that the algorithm can explore the search space efficiently by generating multiple candidate solutions in parallel.
  - However, one weakness of Local Beam Search is that it may miss the global optimum because it can get stuck in a local optimum and not explore other regions of the search space.

    ➔ The performance of the algorithm also depends on the quality of the initial beam and the choice of the neighborhood function.

❖ Genetic Algorithms (GA) are a class of metaheuristic algorithms that are inspired by the process of natural selection and evolution.
  - The space complexity is also O(gkn) because the algorithm needs to store the current population and its genetic information.
  - One advantage of the Genetic Algorithm over other algorithms like brute force or branch and bound is that it can explore the search space efficiently by generating multiple candidate solutions in parallel and searching for

multiple local optima. The algorithm is also flexible and can be adapted to different types of Knapsack problems, such as the multiple Knapsack problem or the 0/1 Knapsack problem with additional constraints.

- However, one weakness of the Genetic Algorithm is that it is not guaranteed to converge to the global optimum, and the quality of the solutions obtained depends on the quality of the initial population and the choice of the genetic operators. The algorithm may also suffer from premature convergence or stagnation in certain regions of the search space, which can limit its ability to find better solutions.

➔ The Genetic Algorithm is a useful algorithm to solve the Knapsack problem when an approximate solution is acceptable, and the search space is large and complex. The algorithm can be optimized by using various genetic operators and selection strategies to improve the diversity and quality of the population and reduce the time and space complexity.

4. Conclusion

- The knapsack problem is a classic optimization problem that can be solved using various search algorithms, including brute force, branch and bound, local beam, and genetic algorithms. Each algorithm has its own strengths and weaknesses, and the choice of algorithm depends on the problem size, the required solution quality, and the available computational resources. The brute force algorithm is simple but inefficient for large instances of the problem, the branch and bound algorithm is an improvement over brute force and can provide an exact solution for small and medium-sized instances of the problem, the local beam algorithm is a stochastic algorithm that can converge quickly to a local optimum, and the genetic algorithm is a metaheuristic algorithm that can explore the search space efficiently and converge to a good solution in a reasonable amount of time. Ultimately, selecting the most appropriate algorithm for a specific instance of the knapsack problem requires careful consideration of the trade-offs between solution quality, computational resources, and algorithm complexity.

V. References

(1): brute force algorithm: https://www.javatpoint.com/brute-force-approach

(2): branch and bound algorithm: https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/

(3): local beam search algorithm: https://www.baeldung.com/cs/beam-search

(4): genetic algorithm: https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3