VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY

UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

REPORT
PROJECT 01: SEARCHING FOR THE KNAPSACK

Under The Guidance Of:
Lecturer: Dr. Nguyen Ngoc Thao
Lecturer: Dr. Le Ngoc Thanh

Ho Chi Minh City, April 2023

TABLE OF CONTENTS

I. MEMBERS' INFORMATION

	ID	Full name
1	21127234	Nguyễn Lê Anh Chi
2	21127235	Nguyễn Xuân Quỳnh Chi
3	21127495	Lê Ngô Song Cát
4	21127659	Bùi Ngọc Kiều Nhi

II. ASSIGNMENT PLAN

	Nguyễn Lê Anh Chi (21127234)	Nguyễn Xuân Quỳnh Chi (21127235)	Lê Ngô Song Cát (21127495)	Bùi Ngọc Kiều Nhi (21127659)
15/3	Meeting to share work			
16 - 21/3	Genetic algorithms	Local beam search	Brute force search	Branch and bound
22/3	Meeting to track progress and divide work			
23 - 25/3	Creating test cases		Editing source and test	
26 - 30/3	Writing the report			
31/3	Meeting to track progress and divide work			
1 - 3/4	Record the demo video	Editing the report		Record the demo video
4/4	Meeting to edit and comment			

III. SELF-ASSESSMENT

	Algorithm	Evaluate	ID
1	Brute force search	100%	21127495
2	Branch and bound	100%	21127659
3	Local beam search	100%	21127235
4	Genetic algorithms	100%	21127234

IV. ALGORITHMS' DESCRIPTIONS

1. Brute force search:

1.1. Description:

- Brute force is a simple technique that can be used to solve several searching problem. This algorithm generates all possible solution in problem space and choose the best solution. Although it can be effective for some small-size problems, using brute force can be challenging for solving larger problem sizes because of the exponential growth of search space. As a result, this can lead to massive amount of computation time and memory usage.

1.2. Explanation:

- Brute force search can be used to solve knapsack problem by generating all possible sublists of items, checking whether each sublist satisfies tthe weight and class contraints or not, and choose the maximum-value sublist. Below is a step-by-step process of using brute force search to solve knapsack problem:

  + Generate all possible sublists of items: Use binary representation to represent item. For example, if the sublist is [0,1,0,1], the chosen items are the items at index 1 and 3.
  + Loop through all the sublists from 0 to $2^n - 1$, where n is the number of items.
  + Calculate the weight, the number of items in each class, and value for each sublist.
  + Check weight and class constraints: if the weight is less than or equal to the capacity of knapsack problem, satisfies class constraints and the value of this current sublist is greater than the current best value then updates this sublist as the best solution. Do this until there is no possible sublists and return the best solution found.

1.3. Pseudo-code:

```
1. function BruteForce(Capacity, NumberofClasses, Items):
2. #Items: list of items, each item: (weight, value, class)
3.   Initalize all possible string containing 0 and 1 bits (loop 2^n times)
4.      Calculate weight of each item and class constraints:
5.         if (weight <= Capacity && every class has at least 1 item && value > currentBestValue)
6.             then Update Best Solution
7.   return Best Solution
```

1.4. Pseudo-code:

- Here is an instance of how to solve the Knapsack problem using brute force search:
- Suppose that the Knapsack has a capacity of 7 and list of following items:

	Weight	Value	Class
Item 1	2	3	1
Item 2	3	4	1
Item 3	4	5	2
Item 4	5	6	2

- As we have 4 items, there are 16 possible sublists of items to be consider:

Sublist	Weight	Value	Class 1	Class 2	Evaluation
{}	0	0	0	0	-
{1}	2	3	1	0	Violate Class Constraint
{2}	3	4	1	0	Violate Class Constraint
{1,2}	5	7	2	0	Violate Class Constraint
{3}	4	5	0	1	Violate Class Constraint
{1,3}	6	8	1	1	
{2,3}	7	9	1	1	
{1,2,3}	9	12	2	1	Violate Weight Constraint
{4}	5	6	0	1	Violate Class Constraint
{1,4}	7	9	1	1	
{2,4}	8	10	1	1	**Best Solution**
{1,2,4}	10	13	2	1	Violate Weight Constraint
{3,4}	9	11	0	2	Violate Weight and Class Constraint
{1,3,4}	11	14	1	2	Violate Weight Constraint
{2,3,4}	12	15	1	2	Violate Weight Constraint
{1,2,3,4}	14	18	2	2	Violate Weight Constraint

1.5. Pseudo-code:

Test case	Size	Number of classes	Time taken (ms)	Memory Consumed (MB)
1	10	2	3.985	0.0050
2	10	3	3.019	0.0001
3	12	2	17.449	0.0008
4	15	4	129.613	0.0010
5	20	3	6535.836	0.0005
6	50	5	Intractable	Intractable
7	55	5	Intractable	Intractable
8	60	6	Intractable	Intractable
9	65	6	Intractable	Intractable
10	70	7	Intractable	Intractable

2. Branch and bound:

2.1. Description:

- *Branch and bound* is an improved method from *backtracking* method, which works better than *brute force* by ignoring impossible solutions. We calculate the limit (best solution) for every node and compare the limit with the current best solution before discovering the node. If the best in the subtree is worse than the current best, we can ignore this node and its subtrees.

- In the good case, we only need to compute a path through the tree and prune the remaining branches, but in the worst case we need to compute the entire tree.

2.2. Explanation:

- Branch and bound is used to find solutions of optimization problems. The optimization problem here is the Knapsack problem, given a set of items, each with its own weight, value, and classification, the goal is to determine which items are included in the knapsack so that the total weight is less or equal to the maximum weight of the knapsack and whose total value is as large as possible.

(1) Calculate the unit price for each item then sort the items in descending order.

(2) Initially, when there are no items, the bag has a total value of 0, the weight is equal to the maximum weight the bag can hold, the upper bound is calculated as The current weight = Total value + Weight*The unit price of the largest item.

(3) Select items to add to the bag by prioritizing the items with the largest unit price that Weight < The remaining weight of the bag. If the weight is exceeded, consider the next object until no one can be selected, then go to step 7.

(4) After selecting the item, recalculate the total value of the bag by the value of the added item. The remaining weight of the bag is equal to the previous value - Added weight, the top of the bag is equal to The total value + Weight*Unit price of the next item.

(5) If you don't choose an object, keep the previous total value and weight, the upper limit is equal to the old upper limit minus the total value + Weight * Unit price of the object.

(6) Select the branch whose bound is the largest and go back to step 3.

(7) Returns the maximum value and selected objects..

2.3. Pseudo-code:

```
#Initialize the root and put into the queue
    Queue.enqueue(root)
    BestValue = 0
    BestItems = []
while (Queue is not empty)
    node = Queue.pop(0)
        if(node is leaf node):
            #Update best solution
            BestValue = node.value
            BestItems = node.items
        if(bestValue > bound):
            Continue with next node
#Add the child nodes to queue if they are promising
if(leftChild.bound >= bestValue and leftChild.weight <= Capacity):
    Queue.append(leftChild)
if(rightChild.bound >= bestValue and rightChild.weight <= Capacity):
    Queue.append(rightChild)
return BestValue, BestItems
```

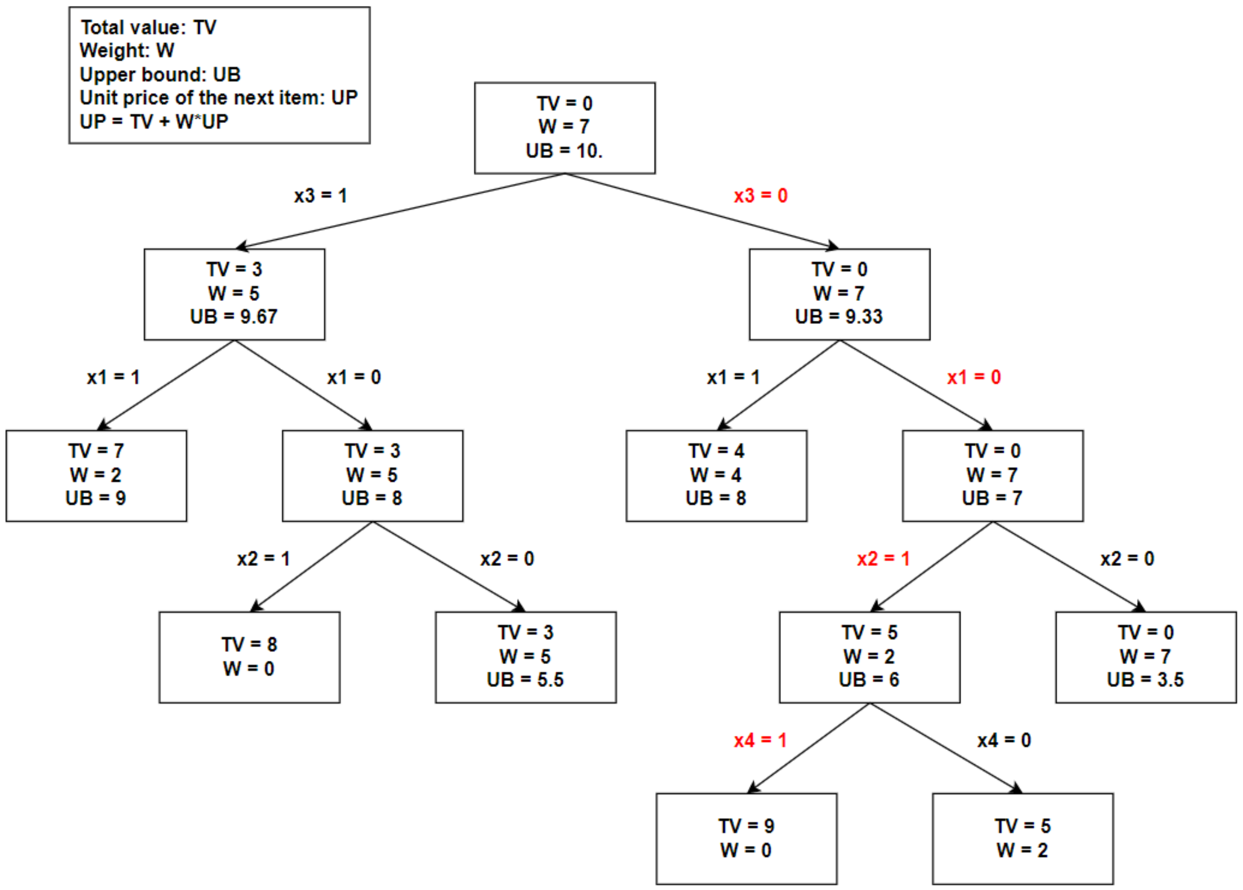
2.4. Visualization:

- Here is an instance of how to solve the Knapsack problem using branch and bound.
- Suppose that the Knapsack has a capacity of 7 and list of following items:

	Weight	Value	Class	Unit price
Item 1	3	4	1	4/3
Item 2	5	5	1	1
Item 3	2	3	2	1.5
Item 4	4	2	2	0.5

- After sorting:

	Weight	Value	Class	Unit price
Item 3	2	3	2	1.5
Item 1	3	4	1	4/3
Item 2	5	5	1	1
Item 4	4	2	2	0.5



2.5. Testing:

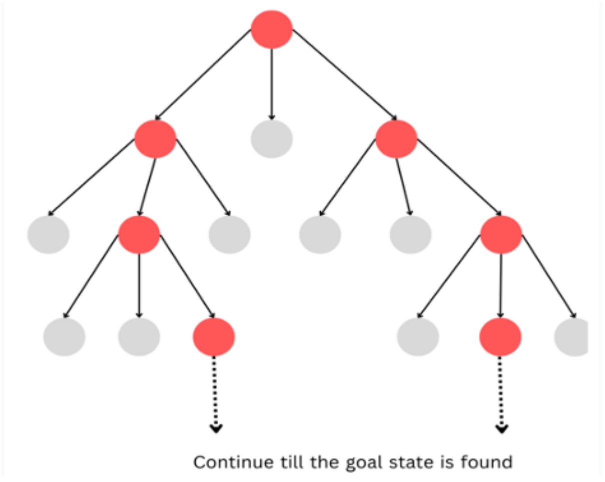
Test case	Size	Number of classes	Time taken (ms)	Memory Consumed (MB)
1	10	2	0.00001	0.0128
2	10	3	31.295	0.0088

3	12	2	158.367	0.0095
4	15	4	2050.211	0.0105
5	20	3	Intractable	Intractable
6	50	5	Intractable	Intractable
7	55	5	Intractable	Intractable
8	60	6	Intractable	Intractable
9	65	6	Intractable	Intractable
10	70	7	Intractable	Intractable

3. Local beam search:

3.1. Description:

- Local Beam Search is a heuristic search algorithm. It starts with a set of initial states generated randomly, then it creates all the descendants from these states. After that, the algorithm filters and retains suitable forms from them based on a specific metric. Gradually, the states will converge to an expected optimal point which can be locally optimal, globally optimal or the states can be stuck into the positions that cannot converge.



3.2. Explanation:

- The algorithm starts by generating a set of initial valid states (Each class must have at least one item in a certain state) and then calculate the value of each state to choose the best state whose value is the largest one.
- At first, calculate: The weight of a state = total weights of all items in the state. The value of a state = total value of all items in the state. If weight > W – capacity (violate the constraint), set value = 0.
- After that, generate the descendants of all initial states. Calculate their weight and value and then compare them to find out the best new state of all child states. Compare best new state with best state. Keep generating child states until reaching the best solution.

3.3. Pseudo-code:

```
#The chosen items in a solution are a sequence of character ‘0’ and ‘1’ separated by comma “,”. E.g., 1, 0, 1, 0, 1, 1
#An item is a label (including weight, value, class)

#Calculate the heuristic of each state
Method calculate_heuristic(state):
  for each item in the state:
    if item is chosen: #The chosen item is marked with 1
      calculate total value of state += value of item
      calculate total weight of state += weight of item
  if total weight of state > Storage capacity W:
    total value = 0
  return total value

#Generate descendants of a solution
Method generate_states(sub_state): #sub_state is the sub-optimal solution
  children is a list of child solutions of sub_state
  for each interation in range(number of given items): #Generate all child solutions
    child is a child solution of sub_state, including a set of items
    if an item is in the sub_state:
      pick item our from child #Mark it with 0
    else:
      put item into child #Mark it with 1
    append child to children
  return children

#Randomly generate a solution
Method states():
  random_state is a randomly generated solution with the largest heuristic
  # from ‘1’ and ‘0’, generate a set of sequences which are solutions
  #find the maximum value between these sequences
  return random_state

#Run the algorithm
Method algorithm(k=number of descendants, max_interations):
  random_states is a list of states generated by method states()
  best_state is the largest value in random_states, based on calculate_heuristic(state)
  best_value is the value of best_state

  for each interation in range(max_interations):
    children is a list of all descendants of parent solutions
    for each state in sub_states:
      children += generate_states(state) #Push child solutions generated into children

    sort sub_states in descending order, based on returned value of calculate_heuristic(state)
    new_best_state is largest value a state in sorted sub_states
    new_best_value is the value of new_best_state
    if new_best_value > best_value:
        best_state = new_best_state
        best_value = new_best_value
chosen_items is a list of positions of items in best_state
int_state is the best state, initialized with a sequence of '0'
for each position in chosen_items:
    int_state[position] = 1
return best_value, int_state
```

3.4. Visualization:

- Here is an instance of how to solve the Knapsack problem using local beam search:
- Suppose that the Knapsack has a capacity of 7 and list of following items:

	Weight	Value	Class
Item 1	3	4	1
Item 2	5	5	2
Item 3	2	3	1
Item 4	4	2	2

	Solution	Weight	Value	Number of classes	Descendant	Weight	Value	Number of classes
1	1-1-0-0	8 (violate)	0	2	0-0-1-1	6	5	2
2	1-0-0-1	7	6	2	**0-1-1-0**	7	8	2
3	1-1-0-1	12 (violate)	0	2	0-0-1-0			1 (violate)
4	1-0-1-1	9 (violate)	0	2	0-1-0-0			1 (violate)
5	0-1-1-1	11 (violate)	0	2	1-0-0-0			1 (violate)

Notes: Since the weight of solution 1, 3, 4, 5 is larger than the capacity W=7, their values will be set 0.

→ 0-1-1-0 is the best solution to the problem after running the algorithm.

3.5. Testing:

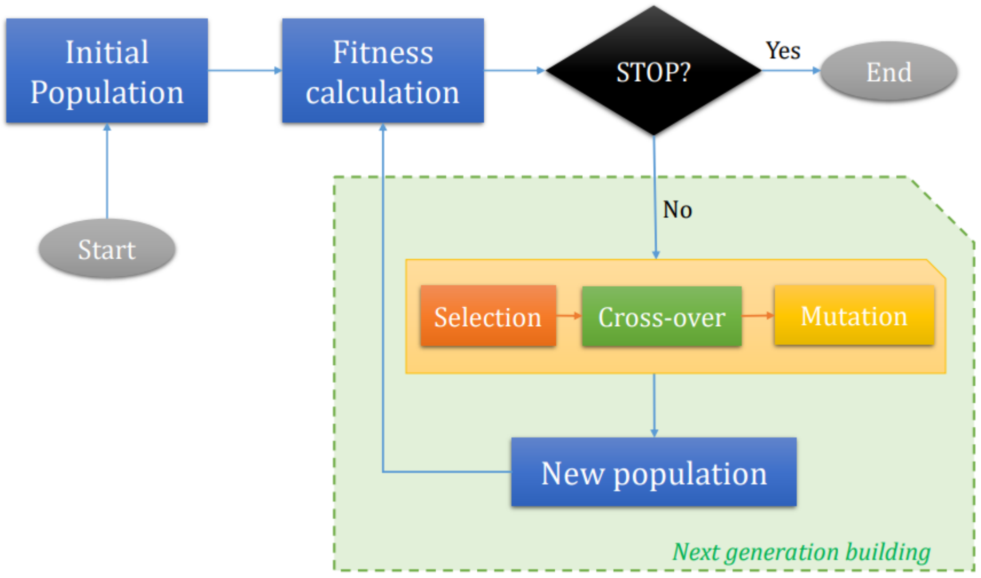
Test case	Size	Number of classes	Time taken (ms)	Memory Consumed (MB)
1	10	2	527.298450	0.0139
2	10	3	580.708504	0.0004
3	12	2	1407.910824	0.0005
4	15	4	1845.561504	0.0006
5	20	3	3206.113338	0.0007
6	50	5	24857.117891	0.0023

7	55	5	33752.249241	0.0007
8	60	6	39245.891809	0.0008
9	65	6	49392.850161	0.0008
10	70	7	59959.122896	0.0007

4. Genetic algorithm:

4.1. Description:

- In genetic programming, a set of possible solutions (or initial generation) are randomly generated, and then evaluated based on a set of criteria. Those solutions that best fit the criteria are then selected, and genetic mutations are applied to create new solution variants (or subsequent generations). This new generation of variants is then evaluated and the process is repeated until a satisfactory solution is found. The process is repeated until an optimal, or best "good enough", solution is found.
- To generate the next generation, the current generation undergoes natural selection through mini-tournaments, and the ones who are fittest reproduce to create offspring. The offspring are either copies of the parent or undergo crossover where they get a fragment from each parent, or they undergo an abrupt mutation. These steps mimic what happens in nature.



- The advantage of using genetic programming to solve the knapsack problem is that a good enough solution can be found quickly without having to exhaustively search through all possible solutions. This makes it a much more efficient approach than traditional algorithms, and allows for a much faster solution to be found.

4.2. Pseudo-code:

```
function ga(population, fitness)
  returns an individual
  inputs: population, a set of individuals
  fitness, a function that measures the fitness of an individual
  repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
      x ← selection(population, fitness)
      y ← selection(population, fitness)
      child ← crossover(x , y)
      if (small random probability) then child ← mutation(child)
      add child to new_population
      population ← new_population
    until some individual is fit enough, or enough time has elapsed
    return the best individual in population, according to fitness

function crossover(x , y) returns an individual
  inputs: x , y, parent individuals
  n ← length(x); c ← random number from 1 to n
  return append(SUBSTRING(x , 1, c), SUBSTRING(y, c + 1, n))

function genetic(number of classes, max_weight, items) return a value and a list
  for i = 0 to number of classes do
    ga(population, fitness)
  return maximum value and list of items chosen
```

4.3. Explanation:

i. Initialize the variables and lists

- This function initialize the variables needed and generates a random population of a given size. For generating population, it uses a for loop to iterate through the given size, and for each iteration it creates a individual. This individual is a list of 0s and 1s, which is generated using the random.getrandbits(length of an individual) function. The individual is then added to the population list. This function is useful for creating a population of individuals for genetic algorithms.

ii. Calculate individual fitness

- This function is used to calculate the fitness of a individual. It takes a individual as an argument and iterates through it. If the value of the individual at a given index is 1, it adds the corresponding item's weight and value to the total weight and total value respectively. If the bag is overweighted, the fitness is set to 0. Otherwise, the fitness is set to the total value. This function is used in genetic algorithms to determine the fitness of a given individual.

iii. Select individuals

- This function is used to select two individuals from a population for crossover. It first calculates the fitness values of each individual in the population using the Fitness() function. It then normalizes the fitness values by dividing each value by the sum of all fitness values. Finally, it randomly selects two individuals from the population based on the normalized fitness values so as to be the parent individuals for crossover.

iv. Perform 2-point crossover

- This function performs 2-point crossover between two individuals. It takes two parent individuals as input and randomly selects two crossover points. The first part of the first parent's genome up to the first crossover point is extracted, and the second part of the genome is from the second crossover point up to the end of the genome. The | operator is used combines these two parts to create a mask that is then applied to the parent's genome using the & operator to extract the desired bits.
- The first and the second part of the second parent's genome is the same formula as the the first parent's genome. Use ~ operator inverts the mask created in the previous step to exclude the bits already selected from the first parent's genome. The | operator combines these two parts to create a mask that is then applied to the parent's genome using the & operator to extract the desired bits. The | operator combines the selected bits from both parents to create the first child's genome. The ^ operator performs a

v. Perform mutation

- This function performs a mutation on a individual. It takes in a individual as an argument and uses the random module to generate a random number between 0 and the length of the individual. If the value at the mutation point is 0, it is changed to 1, and if it is 1, it is changed to 0. The function then prints a message and returns the mutated individual.

vi. Get best individual

- This function takes in a population of individuals and returns the best individual from the population. It does this by first creating a list of fitness values for each individual in the population. It then finds the maximum fitness value and its corresponding index in the list. Finally, it returns the individual at the index of the maximum fitness value. This function is useful for finding the best individual from a population of individuals in order to use it for further operations.

4.4. Visualization:

- Here is an instance of how to solve the Knapsack problem using genetic algorithm:
- Suppose that the Knapsack has a capacity of 12 and list of following items:

	Weight	**Value**	**Class**
Item 1	2	3	1
Item 2	3	4	1
Item 3	4	5	2
Item 4	5	6	2

- Generated a random population of size 200, calculate the fitnesses of the population.
- For example, from the table up there, we got 7 states from generating randomly (there are more states but these states are just chosen for illustration purpose).

States	Weight	Value	Class 1	Class 2	Fitness	Evaluation
[0, 1, 1, 0]	7	9	1	1	9	Win
[0, 1, 0, 0]	2	3	1	0	-	Violate Class Constraint
[1, 0, 1, 0]	6	8	1	1	8	Lose
[1, 0, 0, 1]	7	9	1	1	9	Lose
[1, 0, 1, 1]	11	14	1	2	14	Not picked
[1, 1, 0, 1]	10	13	2	1	13	Win
[0, 1, 1, 1]	12	15	1	2	-	Violate Weight Constraint

- For the next step, we will go through tournament selections. Two states will be chosen randomly to through a tournament where the winner is the state has higher fitness. If the pair of states in blue is picked, the one with the fitness value of 9, which is [0, 1, 1, 0], will be the winner and become the new parent. The same process for the pair of states colored green, the [1, 1, 0, 1] is the winner with the higher value and become the new parent.
- We then crossed the parents over and then we will go through mutation and mutation has flipped one of the bits from one to a zero so these are now our two completed children.

Parent	Crossover	Mutation
[0, 1, 1, 0]	[0, 1, 0, 1]	[0, 1, 0, 1]
[1, 1, 0, 1]	[1, 1, 1, 0]	[1, 0, 1, 0]

- We have already calculated the fitnesses of every single member of the population so we just generate another random tournament.
- Once we have our children here, repeat the process until get an equivalent amount of children that were our original population. Once we have swapped over the population of the children we have created with the old population, this is 1 generation. According to our algorithm, we are going to run this for 500, which is the max generations and as the program going, we should see the average fitness value gets higher as well as the best value. That shall be a good time to stop the algorithm.
- However, if we look closely to the table generating random states, the state which has the highest value, [1, 0, 1, 1], was not chosen at all. It was just randomly not picked but there is a chance it might not be chosen at all and would have not been

passed to the next generation. To overcome this problem, we will choose the best solution from the current population and put it into the new population. For doing this, we will remove one of the new states that we had in new population and replace with this solution → function findBestIndividual.

4.5. Testing

Test case	**Size**	**Number of classes**	**Time taken (ms)**	**Memory Consumed (MB)**
1	10	2	13911.301136	0.0180
2	10	3	24517.560482	0.0005
3	12	2	19594.843626	0.0006
4	15	4	37029.013634	0.0007
5	20	3	53214.582205	0.0008
6	50	5	260324.881315	0.0021
7	55	5	289970.123485	0.0005
8	60	6	261013.999462	0.0005
9	65	6	267957.853317	0.0005
10	70	7	258031.573462	0.0005

5. Test cases:

Test case	**Size**	**Number of classes**
1	10	2
2	10	3
3	12	2
4	15	4
5	20	3
6	50	5
7	55	5
8	60	6
9	65	6
10	70	7

V. COMMENTS AND CONCLUSIONS

1. Comments:

- **Brute force search:**

Time complexity: the time complexity of brute force technique for solving the knapsack problem with classes is exponential in the number of given items, as it generates all possible sublists of items representing as 0 and 1 bits, and checks if they satisfy class and weight constraints: every class has at least 1 chosen item, and the total weight cannot greater than the given capacity.

Actions	Time complexity in worst case
Generating sublists of items	O(2^n)
Checking constraints	O(n)
Overall	O(n.2^n)

Space complexity: the space complexity of brute force technique for solving the knapsack problem with classes is also exponential in the number of given items, because it stores all possible sublists and to choose the best solution.

Actions	Space complexity in all cases
Generating sublists of items	O(2^n)

Completeness: Brute force algorithm guaranteed to find the best solution, as it compares all sublists and select the optimal one.

Time complexity	Space Complexity	Completeness
O(n.2^n)	2^n	Yes

- **Branch and bound:**

Time complexity: In the worst case, branch and bounf technique will generate to all nodes. Hence, the branch tree will have 2^{n-1} - 1 nodes and have the exponential time complexity.
Space Complexity: Space complexity depends on the possible length that the queue can reach to.
Completeness: Branch and bound algorithm guaranteed to find the best solution, as it compares all sublists and select the optimal one.

Time complexity	Space Complexity	Completeness
2^n	Depends	Yes

- **Local beam search**:

Time complexity: It depends on the following things:
+ The accuracy of the heuristic function.
+ In the worst case, the heuristic function leads Beam Search to the deepest level in the search tree.

+ The worst-case time = ***O(B*m)***
(B is the beam width, and ***m*** is the maximum depth of any path in the search tree.)
Space complexity: It depends on the following things:
+ Beam Search's memory consumption is its most desirable trait.
+ Since the algorithm only stores B nodes at each level in the search tree.
+ The worst-case space complexity = ***O(B*m)***
(B is the beam width, and m is the maximum depth of any path in the search tree.)
+ This linear memory consumption allows Beam Search to probe very deeply into large search spaces and potentially find solutions that other algorithms cannot reach.
Completeness: Beam search is an optimization of best-first-search to build its search true, which reduces its memory requirements. In beam search, only a predetermined number of best partial solutions are kept as candidates. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost. However, it only stores a predetermined number of best states at each level. Only those states are expanded next. The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and beam search is identical to breadth-first search. Since a goal state could potentially be pruned, beam search sacrifices completeness. Thus, beam search is not optimal, that is, there is no guarantee that it will find the best solution)

- **Genetic algorithm:**
  For small data input, specifically in the some of the first test cases, it is clearly seen that genetic algorithm spent much more time to find the solutions than other algorithms. The same goes for memory spent but the gaps are not as huge as the time spent by using this algorithm. However, this algorithm is kind of stable since there is not much difference between the time spent to find the solution for the first and the last test case even though the data was more gigantic. Furthermore, the memory used by this algorithm decreases gradually as test cases goes on.

2. Conclusions:

- **Brute force search:** Because the technique has exponential complexity, it can only be used for small-size test cases. On the other hand, it is easy to understand with no complex programming technique.
- **Branch and bound:** Quite similar to brute force search, branch and bound also has exponential time complexity. Therefore, it is suitable for small-size datasets.
- **Local beam search**: Beam search is most often used to maintain tractability in large systems with insufficient amount of memory to store the entire search tree.
- **Genetic algorithm:** Overall, genetic algorithm is a strong algorithm when it comes to find solutions for difficult problems with big data input. The average time and memory it used for finding solutions was lower than the other algorithms. By generating population randomly, it can explore to find the results which local is not capable of.

VI. DEMO

Drive:
https://drive.google.com/file/d/1Opl7ksHPatC_2Ef4Tcio8PK9SlzO510Q/view?usp=sharing

Youtube:
Demo Knapsack - Nhóm 11 - 21CLC04 - HCMUS

VII. REFERENCES

[1] https://arpitbhayani.me/blogs/genetic-knapsack

[2] https://www.kdnuggets.com/2023/01/knapsack-problem-genetic-programming-python.html

[3] https://www.geeksforgeeks.org/0-1-knapsack-using-branch-and-bound/

[4] Dr. Nguyen Ngoc Thao Lecture's Slides

[5] https://micsymposium.org/mics_2005/papers/paper102.pdf

[6] https://en.wikipedia.org/wiki/Beam_search

[7] https://www.javatpoint.com/define-beam-search