Vietnam National University - HCM City

University of Science

Project 01:
Searching for the Knapsack

Course: CSC14003 - Artificial Intelligence

Class:	21CLC03
Student:	21127076 – Doãn Anh Khoa
	21127240 – Nguyễn Phát Đạt
	21127322 – Hoàng Xuân Khôi
	21127388 – Tăng Đức Phong
Lecturer:	Nguyễn Ngọc Thảo
	Nguyễn Trần Duy Minh
	Lê Ngọc Thành
	Nguyễn Hải Đăng

TPHCM, April 2023

Table of Contents

I. Information

Class: 21CLC03

ID	Name
21127076	Doãn Anh Khoa
21127240	Nguyễn Phát Đạt
21127322	Hoàng Xuân Khôi
21127388	Tăng Đức Phong

II. Assignment plan

Work	Student
Algorithm 1: Brute force algorithm	21127388 – Tăng Đức Phong
Algorithm 2: Branch and bound algorithm	21127240 – Nguyễn Phát Đạt
Algorithm 3: Local beam search algorithm	21127322 – Hoàng Xuân Khôi
Algorithm 4: Genetic algorithm	21127076 – Doãn Anh Khoa
Generate test cases	21127240 – Nguyễn Phát Đạt
Comparison	21127240 – Nguyễn Phát Đạt

III. Self-assessment

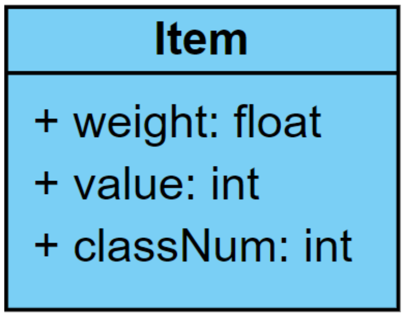
Completion: 100%

Requirement	Completion
Algorithm 1	100%
Algorithm 2	100%
Algorithm 3	100%
Algorithm 4	100%
Generate at least 5 small datasets of sizes 10-40. Compare with performance in the experiment section. Create videos to show implementation.	100%
Generate at least 5 large datasets of sizes 50-1000 with several classes of 5-10. Compare with performance in the experiment section. Create videos to show implementation.	100%
Report algorithms, and experiments with some reflection or comments.	100%

IV. Works

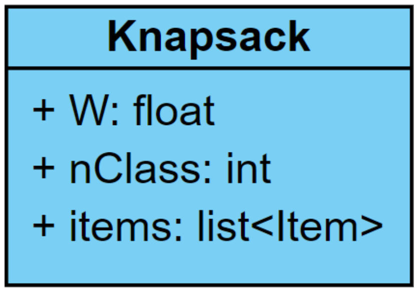
1. Preliminaries

Firstly, we define the Item class as follows:



* weight: the weight of a item
* value: the value of a item
* classNum: the class number of a item (start from 1 to nClass)

Secondly, the Knapsack class is defined as follows:



* W: the capacity of the knapsack
* nClass: the number of classes
* items: the list of *Item* object

Notice that we use all public attributes for each class to simplify the operations.

2. Test cases

- There are 5 small and normal datasets:

Case	Capacity	Number of items	Number of classes	Features	Solution
INPUT_0	105	10	2	Random	109
INPUT_1	382.52	10	3	Random	331
INPUT_2	878	20	3	Random	1024
INPUT_3	975.64	28	3	Random	1195
INPUT_4	1449.42	40	4	Random	1890

- Note that for small datasets, we just generated them randomly since they are so small that inserting more features won't make any significant impact.

- 5 big datasets with different features: each dataset has a random (normal) case used for comparing the average case of the 4 algorithms.

Case	**Capacity**	**Number of items**	**Number of classes**	**Features**	**Solution**
INPUT_5	1944.78	50	5	Random	1913
INPUT_6	8965.09	230	6	Random	11477
INPUT_7	15781.65	450	7	Random	20463
INPUT_8	16456.34	630	8	Random	26677
INPUT_9	26959.06	1000	10	Random	41969

- The remaining test cases are some special cases for comparing and testing purposes:

Case	**Capacity**	**Number of items**	**Number of classes**	**Features**	**Solution**
INPUT_10	15	1	1	Only one item appears	2
INPUT_11	11984.24	230	6	The capacity is very large	12078
INPUT_12	7703.52	450	7	The capacity is small	15114
INPUT_13	1	500	7	The capacity is not large enough	No solution
INPUT_14	14538.3	500	7	Not enough classes	No solution

INPUT_15	12453.38	500	500	Too many classes require	No solution

- INPUT_10: The solution only exists if this item's weight is smaller or equal to the capacity. Here the weight of this only item is 1 < capacity = 15. The solution here is this item's value, which is 2.
- INPUT_11 (has the same item set with INPUT_6): The knapsack's capacity is very large (larger than the sum of every item's weight). This can make Branch and bound algorithms harder to find a solution as most of the nodes won't be pruned (all nodes won't be pruned by the overweight case).
- INPUT_12 (has the same item set with INPUT_7): The knapsack's capacity is small, so the algorithm must carefully choose the right items. The Local beam search and Genetic algorithm randomly choose items, which can make it harder to find the solution since most of the time, the selected items won't be the solution.
- INPUT_13: The knapsack's capacity is so small that they can't hold a solution (smaller than every item's weight). This will significantly worsen the LBS and GA algorithms.
- INPUT_14: One class label is missing (class 1), so there is no solution for this problem. Branch and bound encounter the worst case, since no branches will be pruned.
- INPUT_15: There are 500 classes, so it requires at least 500 items, since their total weight is larger than the capacity, there is no solution. Impact: same as above.

3. Brute force algorithm

3.1. Algorithm's description

Brute Force algorithms are straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency.

Brute Force is the simplest method to solve the 0/1 Knapsack problem with multiple classes of items. If there are ***n*** items to choose from, then there will be 2^n possible combinations of items for the knapsack (An item is either chosen or not chosen). There is a list containing only 0's and 1's with length ***n*** represents the number of chosen items from the available items of the problem. If the ith item in the list is 0, then the ith item is not chosen and if it is 1, then the ith item is chosen.

3.2. Pseudocode

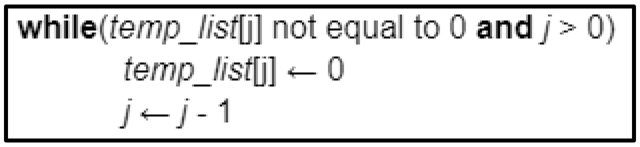
```
Function BruteForce(problem): return best value and list of chosen items
    temp_list ← initialized with n (number of items) 0s
    bestvalue ← -1
    item_list ← empty list
    for i = 1 to $2^n$ do
        j ← n
        tempweight ← 0
        tempvalue ← 0
        while(temp_list[j] not equal to 0 and j > 0)
            temp_list[j] ← 0
            j ← j - 1
        if tempvalue > bestvalue and tempweight <= problem.capacity and
                        Class_Constraint(temp_list, problem) then
            bestvalue ← tempvalue
            item_list ← temp_list
    return bestvalue, item_list
```

```
Function Class_Constraint(item_list, problem): return True or False
    check_list ← initialized with n (number of classes) 0s
    for i = 1 to len(problem.items) do
        if item_list equal to 1 then
            check_list[problem.items[i].classNum - 1] ← 1
            (classNum = The class number of an item)
    for i = 1 to n do
        if check_list[i] equal to 0 then return False
    return True
```

3.3. Explanation

BruteForce(problem): Implementation of Brute Force algorithm on Knapsack problem with multiple classes of items.

- First, *temp_list* is a temporary list to represent chosen items, variable *bestvalue* is assigned with -1 (holds the solution value of the problem, if *bestvalue* = -1 after 2^n items combinations then it means there is no solution to the problem), *item_list* holds the solution list of chosen items, *j* is initialized with number of items in problem.



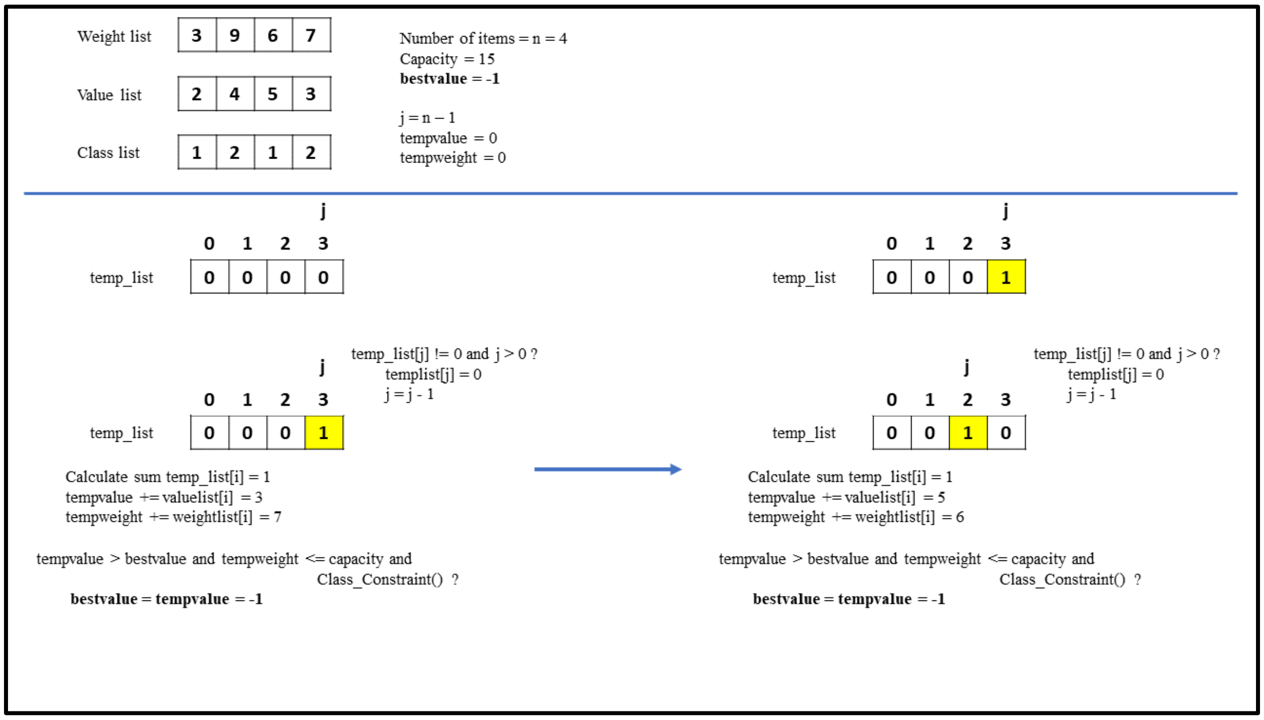
```
while(temp_list[j] not equal to 0 and j > 0)
        temp_list[j] ← 0
        j ← j - 1
```

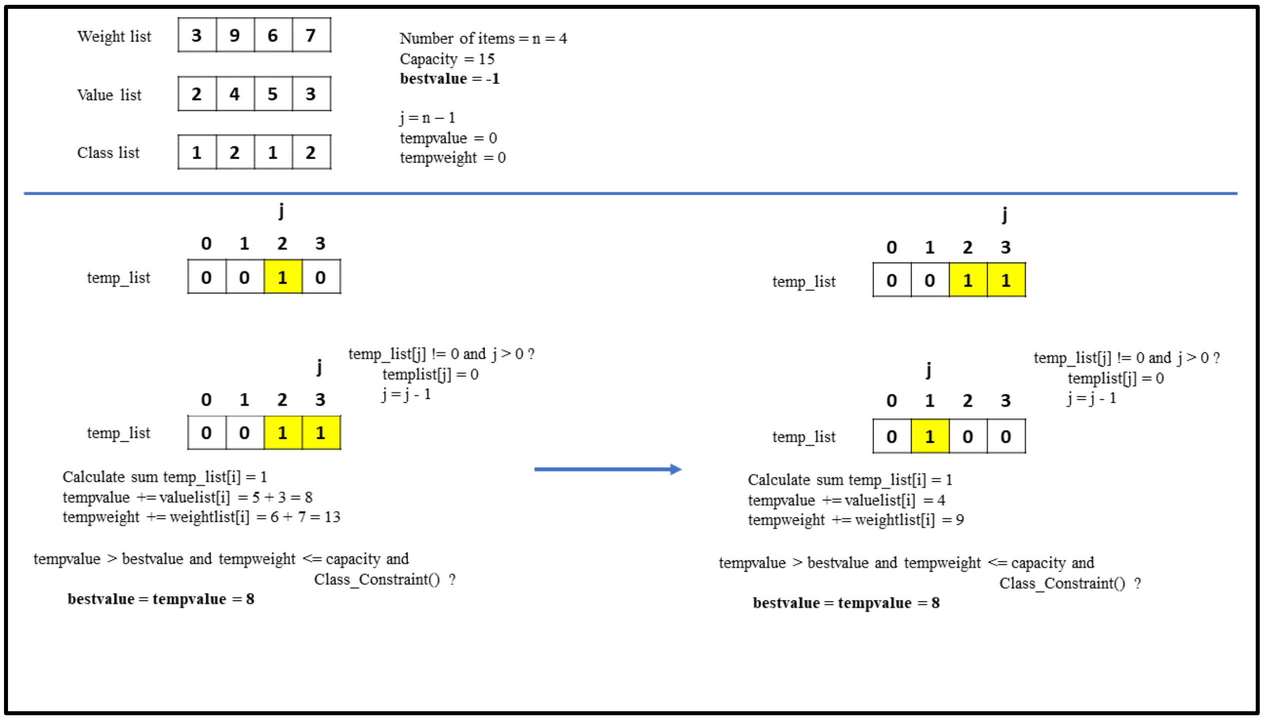
- This *while loop* is used for pairing the items together from the back. For example, problem item list [1, 2, 3, 4] → the pairs are [{1}, {2}, {3}, {4}, {1, 2}, {1, 3}, {1, 4}, {2, 3}, {2, 4}, {3, 4}, {1, 2, 3}, {1, 2, 4}, {1, 3, 4}, {2, 3, 4}, {1, 2, 3, 4}].
- After pairing, the algorithm calculates the pair's weight and value and saves into variables *tempweight*, *tempvalue*. Then it checks if the *tempvalue* is better than the *bestvalue* and *tempweight* does not exceed the knapsack capacity and if there are enough classes in the pair (Class_*Constraint()* function). If all the above conditions are satisfied, then *bestvalue* is assigned with *tempvalue,* and *item_list* is assigned with *temp_list*. The operation of pairing then checks if the conditions are satisfied or not and is repeated until 2^n times, then it returns the *bestvalue* and *item_list*

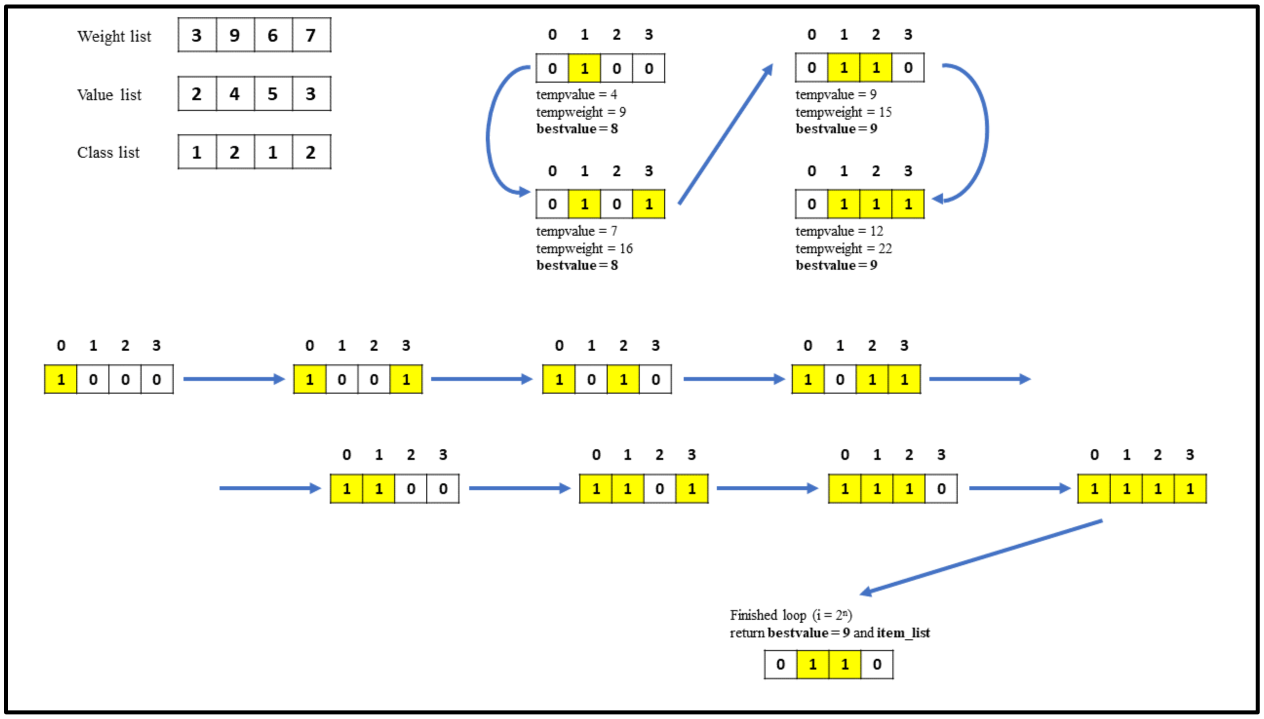
Class_Constraint(item_list, problem): Check if the *item_list* has at least one item of each class or not.

- The idea is to create a *check_list* is a list with length of *n* (number of classes of the problem) initialized with 0s, then for each item in *item_list,* the class of each item is the index of the check list and the check list at that index will have the value 1. For example, list of items class (3 classes) [1, 1, 2, 1, 2] then the check list will be [1, 1, 0]

3.4. Visualization


Weight list 3 9 6 7
Value list 2 4 5 3
Class list 1 2 1 2
Number of items = n = 4
Capacity = 15
bestvalue = -1
j = n – 1
tempvalue = 0
tempweight = 0
j
0 1 2 3
temp_list 0 0 0 0
temp_list[j] != 0 and j > 0 ?
templist[j] = 0
j = j - 1
temp_list 0 0 0 1
Calculate sum temp_list[i] = 1
tempvalue += valuelist[i] = 3
tempweight += weightlist[i] = 7
tempvalue > bestvalue and tempweight <= capacity and Class_Constraint() ?
bestvalue = tempvalue = -1
temp_list 0 0 0 1
temp_list[j] != 0 and j > 0 ?
templist[j] = 0
j = j - 1
temp_list 0 0 1 0
Calculate sum temp_list[i] = 1
tempvalue += valuelist[i] = 5
tempweight += weightlist[i] = 6
tempvalue > bestvalue and tempweight <= capacity and Class_Constraint() ?
bestvalue = tempvalue = -1


Weight list 3 9 6 7
Value list 2 4 5 3
Class list 1 2 1 2
Number of items = n = 4
Capacity = 15
bestvalue = -1
j = n – 1
tempvalue = 0
tempweight = 0
j
0 1 2 3
temp_list 0 0 1 0
temp_list[j] != 0 and j > 0 ?
templist[j] = 0
j = j - 1
temp_list 0 0 1 1
Calculate sum temp_list[i] = 1
tempvalue += valuelist[i] = 5 + 3 = 8
tempweight += weightlist[i] = 6 + 7 = 13
tempvalue > bestvalue and tempweight <= capacity and Class_Constraint() ?
bestvalue = tempvalue = 8
temp_list 0 0 1 1
temp_list[j] != 0 and j > 0 ?
templist[j] = 0
j = j - 1
temp_list 0 1 0 0
Calculate sum temp_list[i] = 1
tempvalue += valuelist[i] = 4
tempweight += weightlist[i] = 9
tempvalue > bestvalue and tempweight <= capacity and Class_Constraint() ?
bestvalue = tempvalue = 8



3.5. Test cases

Input	Executed time (s)	Best Value
INPUT_0 (10 items)	0.00128	109
INPUT_1 (10 items)	0.00126	331
INPUT_2 (20 items)	2.13116	1024
INPUT_3 (28 items)	729.03113	1195
Input with 30 items	3057.295	X
Further	Intractable	X

3.6. Evaluation

The complexity of Brute Force algorithm is ***O*(*n*2*ⁿ*)**. Since the complexity of this algorithm grows exponentially, it can only be used to solve the Knapsack problem with a small number of items. So, 30 is the maximum number of items that can be solved in an acceptable amount of time, as it takes a lot of time to solve for each combination of items with a big ***n***. Nonetheless, it is enough to prove that the algorithm successfully generates the correct solution to the problem.

3.7. Comments

The Brute Force approach is easy to implement and does not require much programming skill. Therefore, the efficiency not very bright (the complexity is ***O*(*n*2*ⁿ*)**). It totally relies on pure computing power and takes a lot of time as it tries every possible solution.

3.8. Conclusions

Although the algorithm provides the correct solution. However, it is the worst approach to solve the Knapsack problem as the time it takes to solve grows exponentially. The bigger n means the longer time it takes to solve the problem.

4. Branch and bound algorithm

4.1. Algorithm's description

Branch and bound (BB, B&B, or BnB) is a method for solving optimization problems by breaking them down into smaller sub-problems and using a bounding function to eliminate sub-problems that cannot contain the optimal solution [5].

For the knapsack problem: The algorithm utilizes a strategy that looks for an ideal combination of items that can fit in a knapsack given weight constraints. It works by generating a set of potential solutions, or nodes, and then branching off from each one to explore all possible sub-solutions in the search space [7].

How it works: The algorithm estimates the highest possible value of the remaining items that could be added to the knapsack at each node, then determines an upper bound on the potential value of the solution at each node. The node will be pruned, and the algorithm moves on to the following potential solution if the upper bound is less than the current best solution.

If the upper bound is higher than the current best solution, the algorithm branches off into two sub-nodes, one in which the following item is added to the knapsack and one in which it is not. The algorithm then recursively explores these sub-nodes, pruning any lower upper bound than the current best solution, until it finds the optimal combination of items that meet the weight constraint.

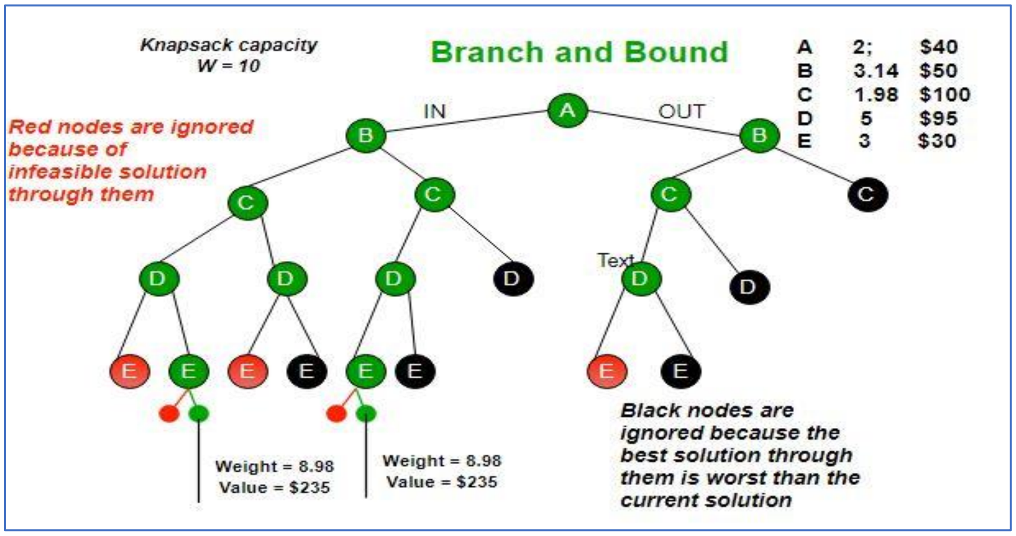

Knapsack capacity
W = 10
Branch and Bound
A 2; $40
B 3.14 $50
C 1.98 $100
D 5 $95
E 3 $30
IN
OUT
Red nodes are ignored because of infeasible solution through them
Text
Weight = 8.98
Value = $235
Weight = 8.98
Value = $235
Black nodes are ignored because the best solution through them is worst than the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

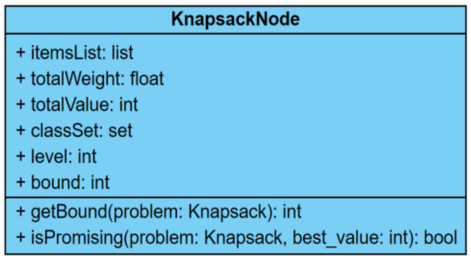
Figure 1: Branch and Bound algorithm (https://www.geeksforgeeks.org/branch-and-bound-algorithm)

Keep in mind that the algorithm depends on the accurate estimation of the upper bounds of each branch in the search space. If the calculation of bounds is not accurate and not effective, the algorithm degenerates into an exhaustive search (like the Brute force algorithm). [5].

Overall, the branch and bound algorithm efficiently solves the knapsack problem by systematically exploring the search space and quickly eliminating suboptimal solutions.

4.2. Pseudocode

First, we define each node of the search tree as follows:



* *itemList:* a list of selected items.
* *totalWeight:* the total weight of selected items.
* *totalValue:* the total value of selected items.
* *classSet:* a set of the class number of selected items (unique).
* *level*: the depth of a node.
* *bound:* the maximum value a node will have.
* getBound()*:* calculate the bound value.
* isPromising(): check if a node is worth consider and push to the stack or not.

Table 1: Pseudo code for the Branch and bound algorithm

```
function getBound(self, problem: Knapsack) returns the bound value of a node
    remainingWeight ← problem.capacity - self.totalWeight
    self.bound ← self.totalValue
    loop with i from self.level + 1 to problem.items.length - 1
        item ← problem.items[i]
        if remainingWeight >= problem.capacity then
            self.bound ← self.bound + item.value
            remainingWeight ← remainingWeight - item.weight
        else
            self.bound ← self.bound + remainingWeight * item.value / item.weight
            break
    return self.bound
```

```
function isPromising(self, problem: Knapsack, bestValue) return true or false
    return self.totalWeight <= problem.capacity and self.bound > bestValue
```

```
function BranchAndBound(problem: Knapsack) returns a solution, or failure
    sort problem.items in decreasing order with key is the item.value / item.weight
    myStack ← an empty LIFO stack
    myStack.PUSH(a KnapsackNode with empty itemsList and classSet, totalWeight,
    totalValue, bound ← 0,  level ←  -1)
    solution ← None
    best_value ← -1
    promisingLeft ← False
    promisingRight ← False
    while not myStack.EMPTY()
```

```
        u ← myStack.POP()

        if u is not a leaf node (u.level ≠ problem.items.length - 1) then
            curItem ← problem.items[u.level + 1]

            childWithItem ← (a KnapsackNode with
                    itemsList ← u.itemsList append with curItem,
                    classSet ← u.classSet add curItem.classNum,
                    totalWeight ← u.totalWeight + curItem.weight,
                    totalValue ← u.totalValue + curItem.value,
                    level ← u.level + 1,
                    bound ← calculate with getBound() function)
            if childWithItem.isPromising(problem, best_value) then
                promisingLeft ← True
                if childWithItem.totalValue > best_value and
                childWithItem.classSet.length = problem.nClass then
                    bestValue ← childWithItem.totalValue
                    solution ← childWithItem.itemsList
            else promisingLeft ← False

            childWithoutItem ← (a KnapsackNode with
                    itemsList ← u.itemsList,
                    classSet ← u.classSet,
                    totalWeight ← u.totalWeight,
                    totalValue ← u.totalValue,
                    level ← u.level + 1,
                    bound ← calculate with getBound() function)
            if childWithoutItem.isPromising(problem, best_value) then
                promisingRight ← True
                if childWithoutItem.totalValue > best_value and
                childWithoutItem.classSet.length = problem.nClass then
                    bestValue ← childWithoutItem.totalValue
                    solution ← childWithoutItem.itemsList
            else promisingRight ← False

            if promisingRight then myStack.PUSH(childWithoutItem)
            if promisingLeft then myStack.PUSH(childWithItem)

    if solution is None then
        return failure
    else return best_value, solution
```

4.3. Explanation

- **getBound()** method (of the KnapsackNode class) takes a Knapsack problem as input and will replace and returns the bound value of the knapsack node. It calculates this value by loop through every remaining (not taken) item in the item list of the problem and continue to add the value of remaining items to the bound, which is initialize with the node's total value, until the remaining total item's weight is less than the weight of a 'x' item. Finally, it will add the ratio of the 'x' item times the remaining weight to fill up the capacity as much as possible.

- **isPromising()** method (of the KnapsackNode class) will return true if a node is worth considering when its total weight is less than or equal to the knapsack's capacity and its bound value is larger than the current best value so far or return false otherwise. Explaination: the bound value is the potential of the future total value of a node, that's why when its bound value is less than the current best value, we don't need to consider or push it to the stack since its value will never be the solution, in other words, we prune this node's branch from the search tree.

- **BranchAndBound()** function takes the knapsack problem as input and will return a solution or failure. The algorithm will consider every node in the search tree, check if a leaf node is a solution or not and pruning every non potential node.
  (Here, I will call the child with the new item is the left child and the child without the new item is the right child for short)
  - First, we **sort** the list of items of the knapsack problem in **decreasing order** with the key is its **ratio** (value / weight). By doing this, we can make the algorithm consider the best item first (highest value and lowest weight) since we want to maximize the total value of the chosen items and minimize the total weight of them, then the second best and so on.

  - Here I use the ***depth-first*** approach (kind of) to traverse over the search tree by using the **stack**. After we sort the item list of the problem in decreasing order, we can make sure that the algorithm often chooses the top best items for the next child nodes first, so with the stack, these top best items will be choosen earlier. Hence, the *best_value* variable will be updated more often, and more branches will be pruned. There are some other types of traverse like ***breadth-first*** (by using the **queue**) or ***best-first*** (by using the **priority queue**).

  - Moreover, we can see that most of the time, the left child has total value higher than the right child (sometimes, the left child cannot be a solution and we will use the right child). So, the left child is more likely to be the solution, then we will want to pop it out and consider it first (*because the stack has LIFO behavior, we need to push the right child first and then the left child*).

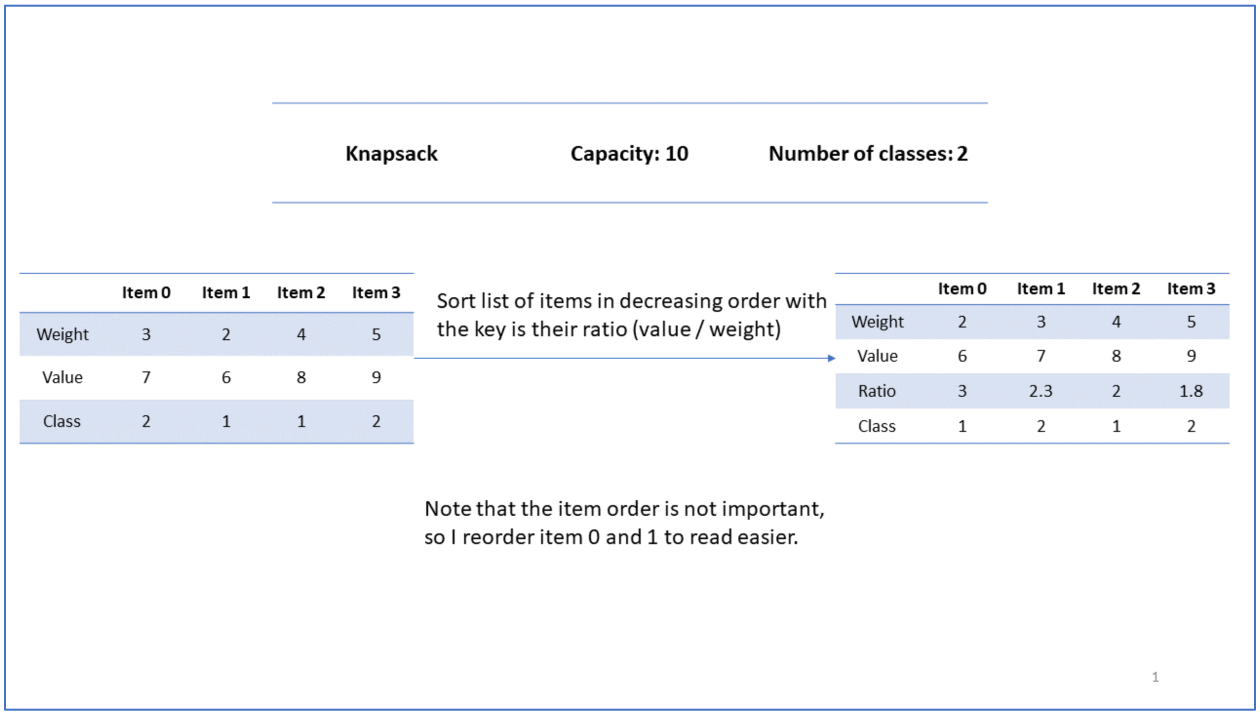
  - The reason I initialize the first (to put in the stack) node's level with the value of **-1** instead of 0 is because this node is just a dummy node (not appear on the search tree), we will not consider this node since it doesn't contain any item and just use it to create its children, which can contain items.

- We can check if a node is the **leaf node** or not by checking if its level or its number of items is equal to the number of the problem's items. Notice that I ignore the leaf nodes since the generated children have been checked if they are the solution or not and update the best value if neccessary, by doing this, we can prune more branch as the algorithm doesn't always have to reach the leaf node to update the best value and the solution.

- The reason I use **node level** and consider the **child node without new item** is to consider the case when a node has contained *enough* items (there is no more node to push in the item list to make total weight of them less than or equal to the knapsack's capacity). If I don't do it, the algorithm won't know when to stop (unless the solution is all the node in the problem), and the node doesn't have the number of items equal to the problem's number of items will never be considered as a solution. What makes it even worse is the right child will continuously be put into the stack and the algorithm will encounter an infinite loop unless we increase its level.

- Note that by using the **set** of numbers of classes in each node, we can check whether the solution contains at least one item for each class. When a node is a leaf node, we check if its value is higher than the current best value so far, but to consider it's a solution we need the length of the class set must be equal to the number of classes of the problem. By using the add method, we ensure that there is no dupplicate class of items inside the set, so its length can only be equal to the problem's number of classes when its items have enough class.

4.4. Visualization



Knapsack **Capacity: 10** **Number of classes: 2**

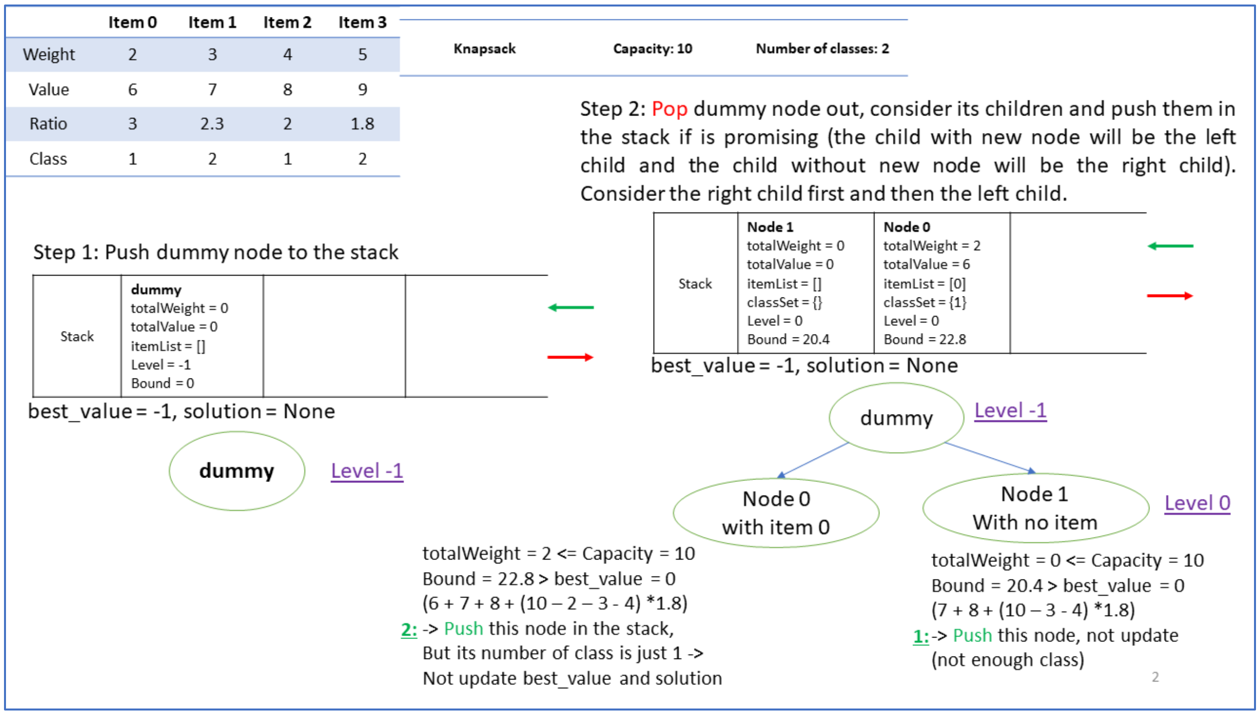
	Item 0	Item 1	Item 2	Item 3
Weight	3	2	4	5
Value	7	6	8	9
Class	2	1	1	2

Sort list of items in decreasing order with the key is their ratio (value / weight)

	Item 0	Item 1	Item 2	Item 3
Weight	2	3	4	5
Value	6	7	8	9
Ratio	3	2.3	2	1.8
Class	1	2	1	2

Note that the item order is not important, so I reorder item 0 and 1 to read easier.

1



	Item 0	Item 1	Item 2	Item 3
Weight	2	3	4	5
Value	6	7	8	9
Ratio	3	2.3	2	1.8
Class	1	2	1	2

Knapsack **Capacity: 10** **Number of classes: 2**

Step 1: Push dummy node to the stack

Stack	**dummy**
	totalWeight = 0
	totalValue = 0
	itemList = []
	Level = -1
	Bound = 0		

best_value = -1, solution = None

Step 2: Pop dummy node out, consider its children and push them in the stack if is promising (the child with new node will be the left child and the child without new node will be the right child). Consider the right child first and then the left child.

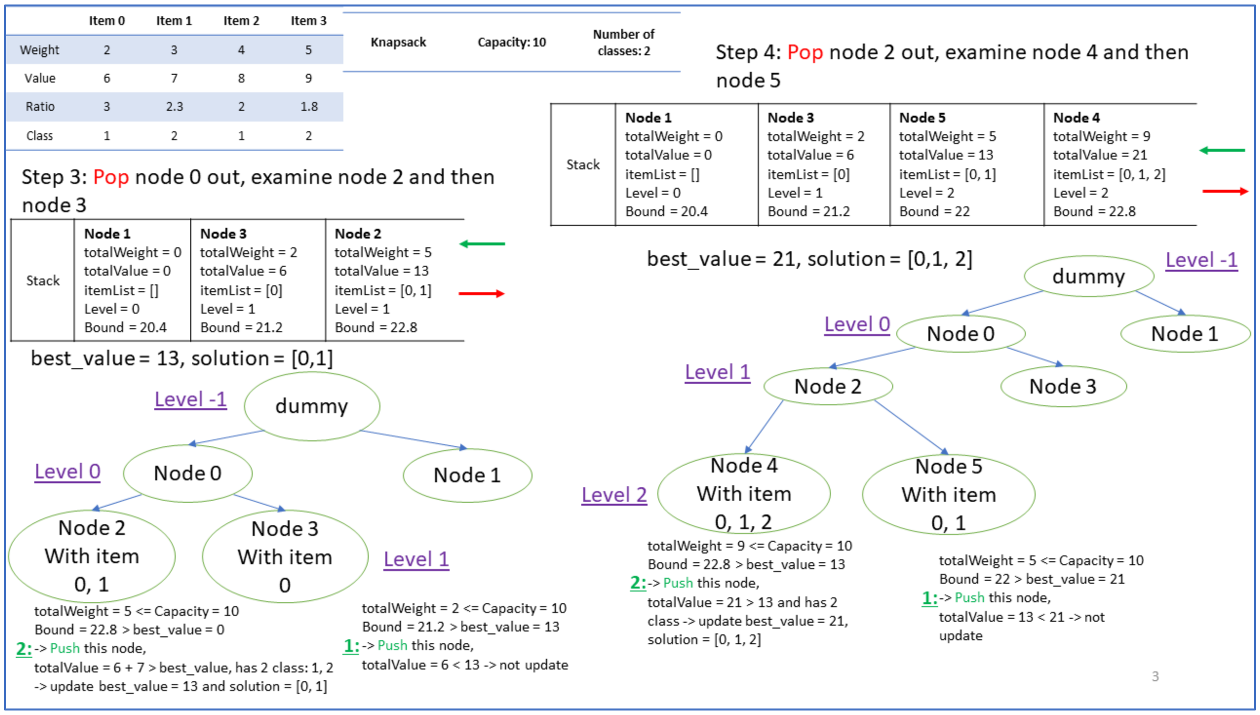
Stack	**Node 1**
	totalWeight = 0
	totalValue = 0
	itemList = []
	classSet = {}
	Level = 0
	Bound = 20.4
		Node 0
		totalWeight = 2
		totalValue = 6
		itemList = [0]
		classSet = {1}
		Level = 0
		Bound = 22.8	

best_value = -1, solution = None

totalWeight = 2 <= Capacity = 10
Bound = 22.8 > best_value = 0
(6 + 7 + 8 + (10 – 2 – 3 - 4) *1.8)
2: -> Push this node in the stack,
But its number of class is just 1 ->
Not update best_value and solution

totalWeight = 0 <= Capacity = 10
Bound = 20.4 > best_value = 0
(7 + 8 + (10 – 3 - 4) *1.8)
1: -> Push this node, not update
(not enough class)

2



	Item 0	Item 1	Item 2	Item 3
Weight	2	3	4	5
Value	6	7	8	9
Ratio	3	2.3	2	1.8
Class	1	2	1	2

Knapsack	Capacity: 10	Number of classes: 2

Step 3: Pop node 0 out, examine node 2 and then node 3

Stack	Node 1	Node 3	Node 2

	totalWeight = 0
	totalValue = 0
	itemList = []
	Level = 0
	Bound = 20.4
		totalWeight = 2
		totalValue = 6
		itemList = [0]
		Level = 1
		Bound = 21.2
			totalWeight = 5
			totalValue = 13
			itemList = [0, 1]
			Level = 1
			Bound = 22.8

best_value = 13, solution = [0,1]

totalWeight = 5 <= Capacity = 10
Bound = 22.8 > best_value = 0
2: -> Push this node,
totalValue = 6 + 7 > best_value, has 2 class: 1, 2
-> update best_value = 13 and solution = [0, 1]

totalWeight = 2 <= Capacity = 10
Bound = 21.2 > best_value = 13
1: -> Push this node,
totalValue = 6 < 13 -> not update

Step 4: Pop node 2 out, examine node 4 and then node 5

Stack	Node 1	Node 3	Node 5	Node 4

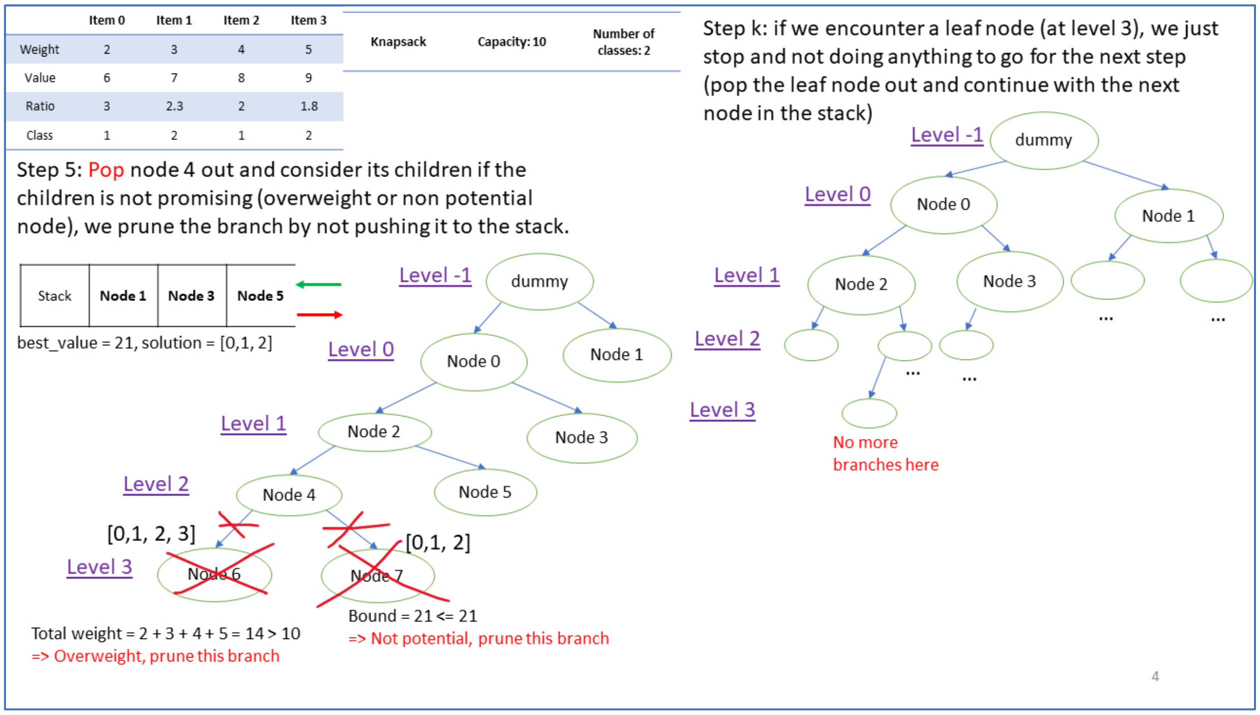
	totalWeight = 0
	totalValue = 0
	itemList = []
	Level = 0
	Bound = 20.4
		totalWeight = 2
		totalValue = 6
		itemList = [0]
		Level = 1
		Bound = 21.2
			totalWeight = 5
			totalValue = 13
			itemList = [0, 1]
			Level = 2
			Bound = 22
				totalWeight = 9
				totalValue = 21
				itemList = [0, 1, 2]
				Level = 2
				Bound = 22.8

best_value = 21, solution = [0,1, 2]

totalWeight = 9 <= Capacity = 10
Bound = 22.8 > best_value = 13
2: -> Push this node,
totalValue = 21 > 13 and has 2 class -> update best_value = 21, solution = [0, 1, 2]

totalWeight = 5 <= Capacity = 10
Bound = 22 > best_value = 21
1: -> Push this node,
totalValue = 13 < 21 -> not update

3



	Item 0	Item 1	Item 2	Item 3
Weight	2	3	4	5
Value	6	7	8	9
Ratio	3	2.3	2	1.8
Class	1	2	1	2

Knapsack	Capacity: 10	Number of classes: 2

Step 5: Pop node 4 out and consider its children if the children is not promising (overweight or non potential node), we prune the branch by not pushing it to the stack.

Stack	Node 1	Node 3	Node 5

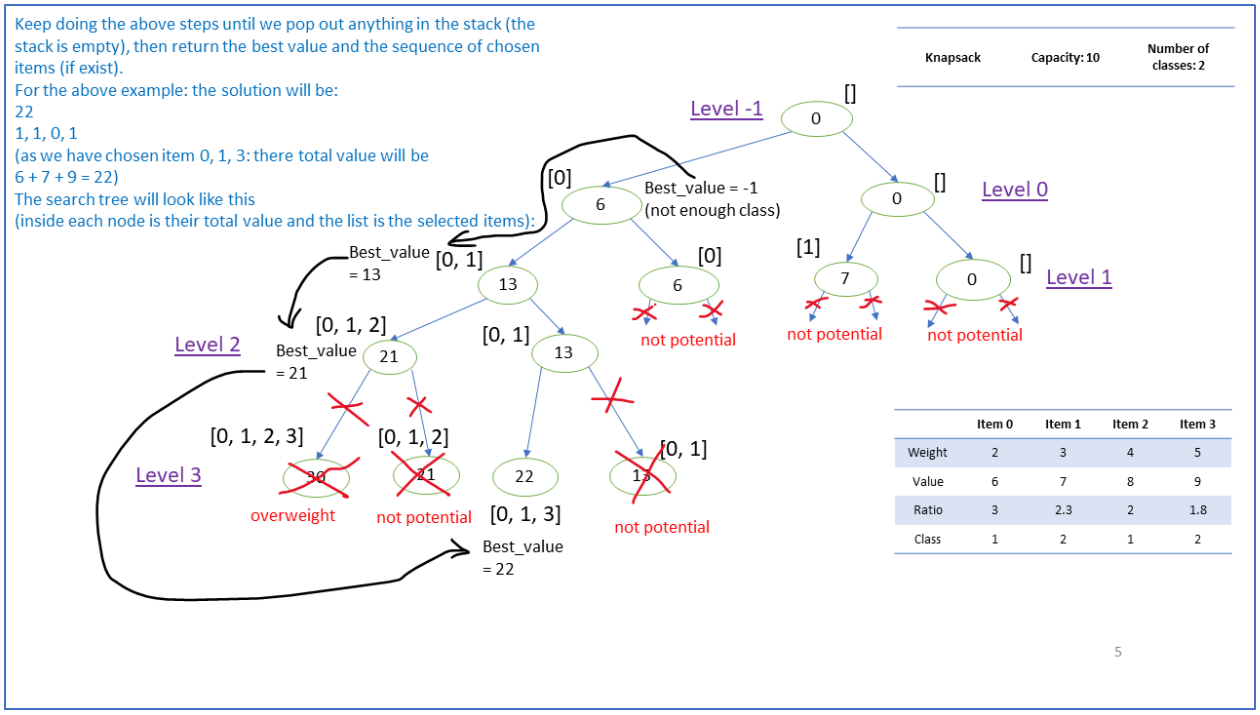
best_value = 21, solution = [0,1, 2]

Total weight = 2 + 3 + 4 + 5 = 14 > 10
=> Overweight, prune this branch

Bound = 21 <= 21
=> Not potential, prune this branch

Step k: if we encounter a leaf node (at level 3), we just stop and not doing anything to go for the next step (pop the leaf node out and continue with the next node in the stack)

4



Keep doing the above steps until we pop out anything in the stack (the stack is empty), then return the best value and the sequence of chosen items (if exist).
For the above example: the solution will be:
22
1, 1, 0, 1
(as we have chosen item 0, 1, 3: there total value will be 6 + 7 + 9 = 22)
The search tree will look like this
(inside each node is their total value and the list is the selected items):

Knapsack	Capacity: 10	Number of classes: 2

	Item 0	Item 1	Item 2	Item 3
Weight	2	3	4	5
Value	6	7	8	9
Ratio	3	2.3	2	1.8
Class	1	2	1	2

4.5. Test cases

INPUT	Executed time (s)	Best value
INPUT_0 (10 items)	0.00019	109
INPUT_1 (10 items)	0.00011	331
INPUT_2 (20 items)	0.000305	1024
INPUT_3 (28 items)	0.000472	1195
INPUT_4 (40 items)	0.00118	1890
INPUT_5 (50 items)	0.00121	1913
INPUT_6 (230 items)	0.013	11477
INPUT_11 (230 items)	0.022	12078
INPUT_7 (450 items)	0.049	20463

INPUT_12 (450 items)	0.026	15114
INPUT_13 (500 items)	0.0014	No solution
INPUT_8 (630 items)	0.0619	26677
INPUT_9 (1000 items)	0.121	41969
INPUT_14 (500 items)	Intractable	Intractable
INPUT_15 (500 items)	Intractable	Intractable

4.6. Evaluation

The branch and bound algorithm approach makes it possible to solve some large and variance datasets. However, in the worst case, it still has an exponential complexity like any other exhaustive search (Brute force algorithm).

Space-complexity: It is proportional to the size of the search space. For the Knapsack problem, the search space can be represented as a binary tree (the visualization section above), with each node corresponding to a chosen subset of items of the problem. The maximum number of nodes in this search tree is equal to **2^n**, where **n** is the number of items.

- ⇨ The space complexity of this algorithm in the worst case (just a few nodes be pruned from the search tree) can reach **$O(2^n)$**, which is impractical and intractable for very large dataset.

Time-complexity: Depending on the the number of subproblems that need to be examined (subset of items). In practice, many subproblems can be pruned easily and early based on the upper bound, which can speed up the algorithm greatly. On average, the executed time of the branch and bound algorithm is typically much faster than many exhaustive approaches.

- ⇨ In the best case, only one path from the root to the leaf node in the search tree will be explored (the algorithm has pruned a lot of nodes). But, before this, we need to sort the item list with the average time-complexity of **O(nlogn)**. Hence, its time-complexity will be **O(nlogn)**.
- ⇨ In the worst case, where the algorithm can't prune any node, in other words, all the subproblems must be examined. The *getBound()* function has time-complexity of **O(n),** and in the worst case, we need to consider **2^n** nodes. Since

we need to calculate the upper bound value of each node the time-complexity of the algorithm can reach up to **$O(n2^n)$**.

For 3 cases (INPUT_14, INPUT_15), the algorithm encounters the worst case, which makes its complexity so bad that it can't tell whether the problem has a solution or not (in fact, there is not) in a practical amount of time and memory.

Moreover, the performance of this algorithm depends strongly on the knapsack's capacity: The smaller the knapsack's capacity is, the faster the algorithm runs and vice versa. Because when the capacity is small, more branches will be pruned earlier.

4.7. Comments

Advantages:

- Branch and bound algorithms can always yield an optimal solution in an acceptable amount of time if its memory requirements are satisfied (usually not a very big deal).
- The algorithm always runs stably and does not depend on any random factors.
- It can handle large problems, which is impractical for exhaustive approaches (Brute force algorithm).
- It's a flexible algorithm that can be used to solve a multiple constraint problem (e.g., this Knapsack problem).
- Performs very well on the problem in which the knapsack has small capacity (INPUT_12 is significantly faster than INPUT_7).

Disadvantages:

- Branch and bound algorithm can be very computationally expensive and impractical since its complexity in the worst case is exponential large.
- Still requires a lot of memory to keep track of nodes (a node must contain many attributes to make the algorithm work).
- Although this algorithm can solve multiple constraints problems, when the constraints become numerous and complex, branch and bound algorithm can not solve it efficiently anymore (for this knapsack problem, the class constraint make it runs not efficently as normal since fewer nodes will be pruned).
- This algorithm cannot foresee the case when a problem has no solution (INPUT_14, 15): it must try and consider every node in the search tree. Because when there is no solution, the current *best_value* will never be updated, so no branches will be pruned, and the algorithm will become even worse than the exhaustive search since it takes more step.
- The same thing happens when the knapsack's capacity is too large: just a few branches will be pruned, its complexity will be higher than usual. For example: in

the case INPUT_11 and INPUT_6 (both have the same ammount of items), but in the case INPUT_11, it runs much slower than usual.

4.8. Conclusions

In conclusion, the branch and bound algorithm is an effective tool for resolving problems like the Knapsack problem. It's kind of a *divide and conquer* algorithm that is based on breaking the problem into smaller subproblems and solving them independently. By using this algorithm, it's possible to reduce the search space and remove unnecessary computation (by predicting its future value) and leading to faster and achivable results. Although can be impractical in abnormal problems (some special cases like too large capacity, not enough class…), on average, the branch and bound algorithm continues to be a significant and outstanding method for solving optimazation problems.

5. Local beam search

5.1. Algorithm's description

Firstly, **Beam search** is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. Beam search is an optimization of best-first search that reduces its memory requirements.[12]

In the context of **Local search**, we call **local beam search** a specific algorithm that begins with k randomly generated states. At each step, all the successors of all k states are generated. If any of them is a goal, the algorithm halts. Otherwise, it selects the k best successors from the complete list and repeats.[1]

5.2. Pseudocode

```
Function localBeamSearch (problem, numIteration, populationSize, beamWidth)
    Initialize:     numItem = number of problem's items
                    population = generate the stater state randomly
                    bestIndivi = None        # to mark the current best individual
                    currIndivi = an empty list which contains current individuals
    Do a loop in range of numIteration
        currIndivi = selectBestIndivi (population, problem, beamWidth)
        fitnessOfCurr_0 = fitness (problem, currIndivi[0])
        # Get the value of the current best individual in individuals list
        if fitnessOfCurr_0 is not equal to 0
            if the bestIndivi is None or fitness(problem, bestIndivi) < fitness(problem, currIndivi[0])
                bestIndivi ← currIndivi[0]
        population = createPopulationNearby (populationSize, currIndivi, numItem)
    end loop
    return bestIndivi
```

```
Function fitness (problem, individual)
    Initialize: sum_value = 0, sum_weight = 0, a checkClass to count if there is a class
    that is not in the individual
    For i in range length of individual
        sum_value += problem.items[i].value
        sum_weigth += problem.items[i].weigth
        checkClass[problem.items[i].ci - 1] += 1
    if (sum_weigth < problem.W) or (there is a class that is not in individual)
        return 0
    return sum_value
```

```
Function selectBestIndivi (population, problem, beamWidth)
# Select some good states from the current states in population
    return beamWidth elements from the sorted list of elements in population
```

```
Function createPopulationNearby (populationSize, currIndivi, numItem)
    newList = an empty list
    i = 0
    loop while i < populationSize and i < length of currIndivi
        temp = a copy list of currIndivi[i]
        newList ← add temp
        numGen = a random integer number from (numItem / 3) to (numItem / 2)
        loop in range of numGen
            pos = a random interger number from 0 to (numItem – 1)
            temp[pos] = 1 – temp[pos]
            newList ← add temp
        i += 1
    return newList
```

5.3. Explanation

- For the Local beam search to run efficiently, firstly, there are parameters we must define, which are: *population size, beam width, number of iterations*.
    - The above-mentioned parameters have a significant effect on performance and running time of the algorithm. However, how to choose resonably the values of the parameters depend on the size of the items to be solved.
    - If the number of items in the problem is small, using a larger *Population size* will make the algorithm more searchable. In this case, the *Beam width* should be chosen smaller to reduce the computation time and increase the performance of the algorithm. The *Number of iterations* should also be

chosen at a moderate value to avoid the algorithm being repeated many times without reaching the optimal solution.

- In contrast, if the number of items in the problem is large, using a *Population size* that is too large can lead to memory overflow and decrease the performance of the algorithm. Instead, *Beam width* should be chosen to be large so that the algorithm is able to search a larger area of the search space. The *Number of iterations* should also be chosen to a larger value to ensure that the search is performed as accurately and optimally as possible.

⇨ In the algorithm that I have implemented, I use the appropriate parameters for finding 20 to 30 items efficiently.

- Next, for the **Function** *Local beam search* implement:
    - Do a loop in range of number of iterations.
    - Firstly, choose some good individuals and add them to list *currIndivi* which is sorted by the *selectBestIndivi* function which is based on *fitness value* before.
    - Then, calculate the fitness of the first individual in list *currIndivi,* if its fitness is higher than the current best individual which is contain in variable named *bestIndivi,* update *bestIndivi* by the first individual in list *currIndivi.* Furthermore, generate a new list of population by returning some new lists of individuals which are based on some good individuals.

- Furthermore, **Function** *fitness* does:
    - Returns the sum of value of an individual when the following conditions are satisfied:
        - → The sum of weight of a given individual must be lower than or equal to the given total weight.
        - → All the classes of item must have at least one in that individual.
    - Returns 0 if one of the above conditions is not satisfied.
- **Function** *createPopulationNearby*:
    - Choose a few good individuals in the current population, then generate randomly a *k* new individuals (successors) and return these individuals.
    - In more detail, the algorithm does a loop in range *Population size* and chooses a group of individuals at the begining of the list of current individuals, which are the good individuals.
    - Then, the algorithm generates *numGen* individuals from the given good individuals and return them to create a new population which contains these good individuals. The reason why *numGen* is a random number from (numItem / 3) to (numItem / 2) is the number of individuals that generated is

moderate but certainly not greater than the current number of items (so as not to generate too many bad individuals if unfortunately).

- However, sometimes generating a new population based solely on the best current individuals can lead to a state of local optimum. Therefore, the selection of individuals is also very important, and should be done carefully for the most optimal algorithm.

5.4. Visualization

- Each individual will be represented as the table below:

Weigth	85	26	48	21	22	95	43	45
Value	79	32	47	18	26	85	33	40
Class	1	1	2	1	2	1	1	2
Individual state	0	1	0	1	1	0	0	0

⇨ 1s is representing a chosen items and in contrast, 0s is not chosen.

Assume that total weigth is 100.

- Following the given table, **fitness** function will calculate:

sum_w = 26 + 21 + 22 = 69
sum_v = 32 + 18 + 26 = 76
All the classes are included

⇨ This individual satisfies all conditions.

- Assume that the above individual is the only one individual in current population and number of individuals that could be generated is 3, **createPopulationNearby** function will return (for instance individual 0 is the above individual):

Individual 0	0	1	0	1	1	0	0	0
Individual 1	0	1	1	1	0	0	0	0
Individual 2	1	1	1	1	1	0	1	0
Individual 3	0	0	0	1	0	1	0	0

5.5. Test cases

In the cases whose number of items is lower than 50, I will run three times to find the most optimal solution. In the other cases, because the number of items is large, which leads to the running time takes a lot of time, so I just run one time for each case. For details, watch the video by following the link.

The parameters will be implemented as followed conditions:

- If the number of items <= 30: popSize = number of items, beamWidth = popSize / 2, number of iterations = 7000.
- If the number of items <= 50: popSize = number of items / 2, beamWidth = popSize, number of iterations = 10000.
- If the number of items <= 300: popSize = number of items / 4, beamWidth = popSize, number of iterations = 10000.
- If the number of items < 450: popSize = number of items / 10, beamWidth = popSize, number of iterations = 10000.
- Else in other cases: popSize = number of items / 100, beamWidth = popSize, number of iterations = 10000.

Note: The limit time is 1 hour, if the test case running time is over than 1 hour, there is not a solution. For details, watch the video by following the link.

INPUT	**Executed time (s)**	**Best value**
INPUT_0 (10 items)	0.55	109
INPUT_1 (10 items)	0.58	331
INPUT_2 (20 items)	3.17	1024
INPUT_3 (28 items)	6.91	1195
INPUT_4 (40 items)	26.15	1890
INPUT_5 (50 items)	48.02	1906
INPUT_6 (230 items)	2253.32	11138
INPUT_7 (450 items)	3158.04	19590
INPUT_8 (650 items)	715.79	23522
INPUT_9 (1000 items)	Intractable	Intractable
INPUT_10 (1 item)	0.03	2
INPUT_11 (230 items)	2428.05	11635
INPUT_12 (450 items)	No solution	No solution

INPUT_13 (500 items)	No solution	No solution
INPUT_14 (500 items)	No solution	No solution

5.6. Evaluation

We have two discussion to talk when running Local Beam search are Space complexity and Time complexity

- Space complexity = O ((Population size) * (Beam width))
    - Space complexity is base on two factors which are number of individuals which is known as Population size and Beam width.
    - When applying Local beam search, we use a list of individuals held in memory. Depending on the Beam width, the number of individuals will vary.
    - Thus, to reduce space complexity, we can reduce the Beam width or reduce the size of each state. However, this can affect the performance of the algorithm.
- Time complexity = O((I) * (S)^2 * (N))
    - I = Number of iterations, S = Population size, N = Number of items
    - These parameters will not be changed beacause they are constant numbers which is set up by user before running the algorithm.
    - The Local beam search algorithm will run a for loop whose range if Number of iterations. In each loop, we have a function to get the best individual in the current individuals which is O(S)

5.7. Comments

- **Advantage:**
    - Minimizing computational cost: Local beam search evaluates only a limited number of children of a node. Therefore, it helps to reduce the total number of evaluation nodes, thereby reducing the computational cost.
    - Minimizing computational cost: Local beam search evaluates only a limited number of children of a node. Therefore, it helps to reduce the total number of evaluation nodes, thereby reducing the computational cost.
    - The essence of local beam search is to search locally at each step. It helps to find the best results within the parent node's internal range.
    - Use a statical amount of memory for the whole process. Because the algorithm only stores a certain number of predefined states.
- **Disadvantage:**

- During the search, the local beam search can get stuck in the optimal local positions without being able to go forward to find a better solution.
- Local beam search usually only finds a good approximation, not a fully optimal solution. Due to the limitation of the number of states traversed in each loop, local beam search is not guaranteed to find the optimal solution. This can lead to suboptimal solutions being found, particularly if the beam width is set too narrow.
- Local beam search does not support optimization of resource allocation between search tasks, which can lead to excess or shortage of resources.
- The performance of local beam search is sensitive to the initialization of the search. If the initial set of paths explored is not diverse, it can lead to suboptimal or premature convergence.

5.8. Conclusions

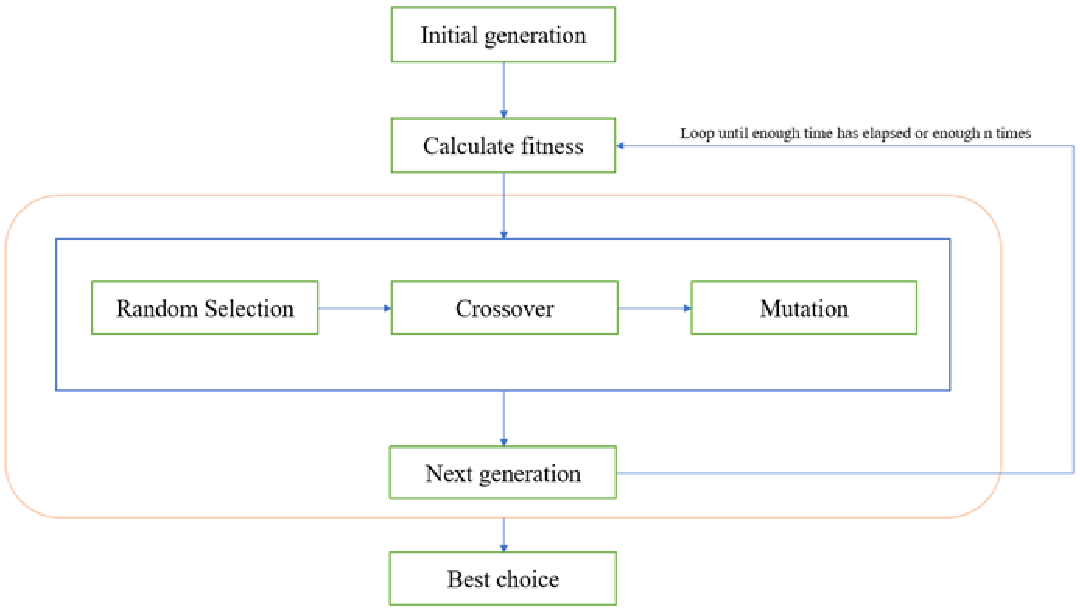
This algorithm is an advanced algorithm and is often considered in problems similar to the Knapsack problem, with advantages such as not consuming too much memory, find the solution quickly if the parameters are reasonable. But there will also be cases where if the number of items is too large or the parameters are not optimal enough, it can lead to very long running times and the result is only an approximation, not the most optimal solution.

In order for the solution to be the most optimal, we have to experiment quite a few times to derive the most optimal parameters and require a fairly accurate calculation.

6. Genetic algorithms

6.1. Algorithm’s description

- Genetic Algorithms (GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. Genetic programming can provide an alternative method for finding a solution to the knapsack problem. By using genetic programming, it is possible to quickly find a solution that is “good enough” for the given problem. It can also be used to optimize and improve upon solutions.
- Initial generation is randomly generated and then evaluated using a set of criteria. The solutions that meet the requirements are chosen, and genetic mutations are used to generate new solution variants. After evaluating the new generation of variants, the process is repeated until a satisfactory solution is discovered. The procedure is repeated until an optimal, or best "good enough", answer is discovered.
- A workflow of Genetic Algorithms:



6.2. Pseudocode

```
**Function** generatePopulation(size,items) **returns** a set of possible solutions randomly generated
    population ← an empty array
    genes ← an array containing value 0s and 1s
    for i = 1 to size do
        chromosome ← an empty array
        for j = 1 to size of items do
            val ← randomly choose one value in genes
            add val to chromosome
        add chromosome to population
    return population
```

```
Function calculateFitness(Chromosome, Backpack) returns total value of chromosome
    total_weight ← 0
    total_value ← 0
    check ← an empty set
    for index,value in enumerate(Chromosome) do
        if value = 1then
            total_weight ← total_weight +  weight of the index^th items of class
            total_value ← total_value + value of the index^th items of class
            add classNum of the index^th items of class to check
    if total_weight > Backpack's weight then
        return 0
    if len(check) != Backpack's number of Class then
        return 0
    return total_value
```

```
Function Crossover(Parent1, Parent2) returns two new chromosomes
    n ← the size of chromosome Parent1
    index ← randomly choose the value from 0 to n – 1
    Child1 ← SUBSTRING(Parent1, 0, index), SUBSTRING(Parent2, index , n - 1)
    Child2 ← SUBSTRING(Parent2, 0, index), SUBSTRING(Parent1, index , n - 1)
    return Child1,Child2
```

```
Function selectionChromosomes (Population,Backpack:Knapsack) returns two chromosomes are chosen
    fitness_values ← an empty array
    for each Chromosome in Population do
        add calculateFitness(Chromosome, Backpack)) to fitness_values
    total ← sum of all values in fitness_values
    if total != 0 then
        fitness_values ← an array with every value is assigned to fitness_values / total
        Parent1, Parent2 ← randomly choose chromosome in population based on values in fitness_values
        return Parent1, Parent2
    return -1, -1
```

```
Function Mutation(Chromosome) returns new chromosome
    index ← random value from 0 to size of Chromosome – 1
    if index^th value in Chromosome = 1then
        index^th value in Chromosome ← 0
    else
        index^th value in Chromosome ← 1
    return Chromosome
```

```
Function createNextGeneration (Population, Backpack) returns next generation
    MUTATION_PROBABILITY ← 0.4
    newGen ← an empty array
    while size of newGen < size of Population
        Parent1,Parent2 ← selectionChromosomes(Population, Backpack)
        if Parent1 = -1 or Parent2 = -1 then break
        Child1, Child2 ← Crossover(Parent1, Parent2)
        if value that randomly chosen from 0 to 1 < MUTATION_PROBABILITY then
            Child1 ← Mutation(Child1)
        if value that randomly chosen from 0 to 1 < MUTATION_PROBABILITY then
            Child2 ← Mutation(Child2)
        add Child1 to newGen
        add Child2 to newGen
    return newGen
```

```
Function GeneticAlgorithm(Backpack) returns best value and solution
```

```
Population ← generatePopulation(200, Backpack.items)
best_choice ← an empty array
fitness_values ← an empty array
indexOfBest ← 0
val ← 0
for i = 1 to 1000 do
    if size of population = 0 then return -1, []
    fitness_values ←  an array containing total value of every chromosome in
population
    indexOfBest ← the index of max value in fitness_values
    if val < fitness_values[indexOfBest] then
        best_choice ← Population[indexOfBest]
        val←fitness_values[indexOfBest]
    if i = size of Backpack.items * 10 then break
    else Population ← createNextGeneration(Population, Backpack)
return val, sol
```

6.3. Explanation

- generatePopulation(size,items): function creates a set of an array containing only the values 0s and 1s. The above values indicate solution of Knapsack problem that items are chosen to load in backpack. Input of function: the size of initial population and items from Knapsack problem. Function using loop to create each array and add it to population. Creating an array is to use loop add each value that randomly choose 0s and 1s to array. The size of array will be the numbers of items.
    - E.g:

An array of items from problem	A	B	C	D	E	F	G
An array in population	0	1	0	1	0	1	0

=> Result indicates that items are chosen: B, D, F.

- calculateFitness(Chromosome, Backpack): function returns total value from values of the chosen items. Input of function: a chromosome (an array) in population and information of backpack from Knapsack problem. If a chromosome has total weight greater than backpack's weight or chromosome doesn't have number of type of items that problem requires, function will return 0.
- selectionChromosomes (Population,Backpack:Knapsack): Input of function: population and information of Backpack. Function selects 2 chromosomes in

population randomly. Selecting is based on values (using function calculateFitness) of all chromosomes in population.

- Crossover(Parent1, Parent2): Input of function: 2 individuals in population. Crossover is an evolutionary operation between two individuals, and it generates children having some parts from each parent. For our knapsack problem, we keep it simple and create two children from two fit parents such that both get one part (position is randomed) from each parent and the other part from the other parent.
- Mutation(Chromosome): The mutation is an evolutionary operation that randomly mutates an individual. We define a parameter called the *MUTATION_PROBABILITY,* which is very low given that mutation rate in nature. For Knapsack problem, we choose position randomly to flip bits of the child (chromosome).
- createNextGeneration (Population, Backpack): combine the above functions based on workflow of GeneticAlgorithm: selectionChromosomes ➝ Crossover ➝ Mutation. The purpose of this function is to create the next generation with the number of individuals equal to the number of ones in population. Using loop to add each individual to new generation.
- GeneticAlgorithm(Backpack): function using loop to create generations to find the best solution accommodate required Knapsack problem. Each generation is created, we choose the best solution that generation and add it to an array (best_choice). When having enough time has elapsed or enough n times loop, function choose the best solution in an array (best_choice) to return. (With the number of items is greather than 100, the number of generation created is 500.)

6.4. Visualization

- Example for Knapsack problems:

```
101
2
85, 26, 48, 21, 22, 95, 43, 45, 55, 52
79, 32, 47, 18, 26, 85, 33, 40, 45, 59
1, 1, 2, 1, 2, 1, 1, 2, 2, 2
```

- generatePopulation function

0 1 0 1 0 1 0 1 1 1
0 1 1 1 0 1 0 0 0 0
1 1 1 0 0 0 1 1 1 0
0 1 1 0 0 1 1 0 0 1

➝ Randomly selected

Intial generation

- calculateFitness function:

0 1 0 1 0 1 0 1 1 1

➝ total_weight = 26 + 21 + 95 + 45 + 55 + 52 = 294 > 101 (Backpack's weight)

⇨ Fitness = 0

0 1 0 0 1 0 0 0 0 1

total_weight = 26 + 22 + 52 = 100 < 101 (Backpack's weight)
check = {1,2} → enough the number of class
⇨ Fitness = 32 + 26 + 59 = 117

0 1 0 1 0 0 1 0 0 0

total_weight = 26 + 21 + 43 = 90 < 101 (Backpack's weight)
check = {1} → enough the number of class
⇨ Fitness = 0

- selectionChromosomes function:

0 1 0 1 0 1 0 1 1 1	0
0 1 0 0 1 0 0 0 0 1	117
…	…
0 1 0 1 0 0 1 0 0 0	0

Select indivual randomly based on fitness:

0 1 0 0 1 0 0 0 0 1
0 1 0 0 0 1 0 0 0 1

- Crossover function:

Parent 1	0 1 0 1 0 1 0 1 1 1
Parent 2	0 1 1 1 0 1 0 0 0 0

Child 1	0 1 1 1 0 1 0 1 1 1
Child 2	0 1 0 1 0 1 0 0 0 0

- Mutation function:

Before	0 1 0 1 0 0 1 0 0 0

After	0 1 0 1 0 1 1 0 0 0

6.5. Test cases

Case	Number of items	Executed time (s)	Best Value
INPUT_10	1	0.0190	2
INPUT_0	10	3.6009	109
INPUT_1	10	5.5304	331
INPUT_2	20	22.4280	1024
INPUT_3	28	37.4344	1193
INPUT_4	40	90.9931	1888
INPUT_5	50	119.9597	1902
INPUT_6	230	1451.2985	10367

INPUT_11	230	1313.3461	10528
INPUT_7	450	1005.6373	15816
INPUT_12	450	0.1082	No solution
INPUT_13	500	0.0906	No solution
INPUT_14	500	0.0844	No solution
INPUT_15	500	0.0889	No solution
INPUT_8	630		
INPUT_9	1000		

6.6. Evaluation

- The run-time complexity of this algorithm to generate a high-quality solution for the Knapsack problem is polymonial. The size of population is P, use loop to create G generations, and F is the run-time complexity of calculateFitness function => the overall complexity of this algorithm is O (P.G.F).
- The efficiency of Genetic Algorithm: there are multiple parameters that increase or decrease the efficiency of this algorithm. The size of the initial population is critical in achieving high efficiency. GA operates on a much larger search space and if the initial population size does not affect the execution time of the algorithm, it converges faster with a larger population than a smaller one. Besides, Crossover, we can use function Single Point Crossover, Two-point Crossover or Multi-point Crossover. Which crossover function would work better for a problem depends totally on the problem at hand. It is generally observed that the two-point crossover results in the fastest convergence. Finally, this algorithm is dependent on creating factors randomly in order to determine optimal solution and take measurement about running time so hard.

6.7. Comments

- The algorithm performed best with large items relative to capacity. This is likely because it is a much easier case with less variance amongst solutions. The algorithm seemed to perform worst with small items relative to capacity. This too was as we predicted. In such hard problems the genetic algorithm would find it difficult to emulate the solution space as it would be inherently much larger due to the several different possible solutions this type of problem allows for.

6.8. Conclusions

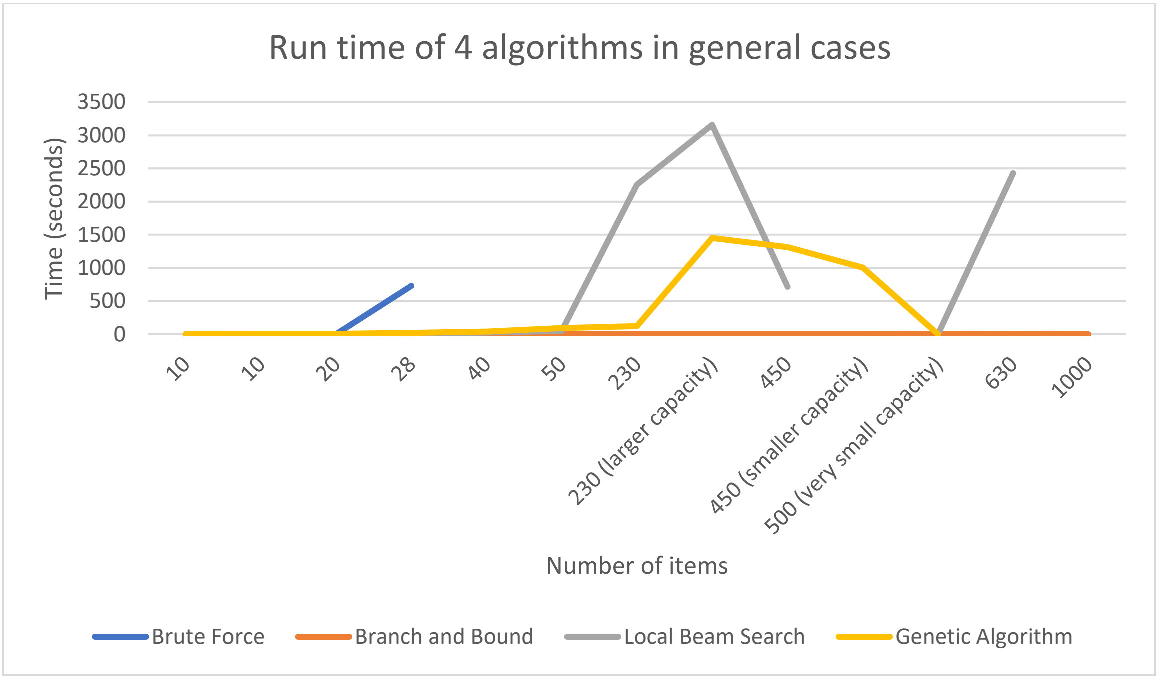
- Genetic Algorithms are powerful meta-algorithms inspired by natural selection that can be used for solving complex optimization problems. The algorithm is general enough to be used in other fields of optimization where a utility ratio (like the greedy estimate) is available. The application of GA to one instance of constrained optimization: 'knapsack problem'. In this experiment, it is demonstrated that GA can be applied to yield a higher benefit or total profit.

7. Experiment

Comparison of performances between 4 algorithms:

- ***Brute Force***: Since its complexity is very high **O(n2^n)**, its run time is the worst. Moreover, this algorithm has exponential complexity in all cases, which makes it intractable even for small datasets (n > 30). However, this algorithm can ensure that this will return the optimal solution (if its required amount of memory and times are satisfied).
- ***Branch And Bound***: This algorithm outperforms every other algorithms for normal cases when using the *depth-first* trarversal method. However, notice that, in the worst case (for case INPUT_14 or 15), the algorithm becomes even worse than the Brute Force as it has exponential complexity with even more uneccessary steps (no nodes will be pruned but it must do a lot of computation to check its). Moreover, the algorithm can return the optimal solution in all cases if it doesn't encounter the worst case.
- ***Local Beam Search***: This algorithm depends on many random factors and many predefined constants like the number of iterations, population size, beam width. Depending on the value of these constants, the smaller these values are, the faster it runs but with more deviation from the optimal solution (most of the time, this algorithm can't find an exactly optimal solution).
- ***Genetic Algorithms***: Like the *local beam search*, this algorithm depends on random factors and predefined constants: population size, number of generations, mutation probability. With more generation, the algorithm generates better solutions close to the optimal one, but it will take more time to execute the algorithm.

In conclusion, we shouldn't use such a highly complex algorithm like Brute force for this problem with large datasets. The LBS and GA depend on many random factors, so most of the time, it won't return the optimal solution (unless we increase the predefined constants), but their required ammount of space and time is stable in all case and its run time won't depend much on the items order. The BNB algorithm is very good for this problem as it just considers a few nodes in its execution. However, BNB is the worst if it encounters the worst case (INPUT_14, 15) => We should use LBS and GA for this case.



For LBS and GA, at the large data sets, it takes a lot of time to run each iteration, so we decrease the number of iterations to make the algorithm return a solution in acceptable amount of time. However, the deviation between the returned solution and the optimal solution becomes worse.

V. Link videos:

https://drive.google.com/drive/folders/1PA2kfORU9pgeOMCSiHvRnTvxrfxITGAQ?fbclid=IwAR3McxakNqgMoeOILEpvOQU24IERz0efd1j4D97f3g6J4l0Ulk81KUTbCqo

Special case: INPUT_10: just 1 item, INPUT_13: no solution exists.

Notice that, for Branch and bound algorithms, I ran from the INPUT_0 to INPUT_13 since for INPUT_14, 15, the algorithm encounters the worst case which makes the algorithm become intractable.

VI. References

[1] Russell, S. J., & Norvig, P. (2010). "Artificial Intelligence: A Modern Approach" (3rd ed.). Upper Saddle River, NJ: Prentice Hall.

[2] *Knapsack problem: https://en.wikipedia.org/wiki/Knapsack_problem*

[3] *Comparison of different approaches to solving the 0/1 Knapsack problem:* https://micsymposium.org/mics_2005/papers/paper102.pdf

[4] *Brute force algorithm idea:* https://www.youtube.com/watch?v=pToZp75IYF0

[5] *Branch and bound algorithm idea:* https://codecrucks.com/knapsack-problem-using-branch-and-bound/

[6] https://en.wikipedia.org/wiki/Branch_and_bound#:~:text=A%20branch%2Dand%2Dbound%20algorithm,subsets%20of%20the%20solution%20set.

[7] *Branch and bound approach for knapsack problem:* https://medium0.com/@leenancyparmar1999/knapsack-problem-branch-and-bound-approach-1fdab6d9a241

[8] Kellerer, H., Pferschy, U., & Pisinger, D. (2004). Knapsack problems (Vol. 2). Springer Science & Business Media.

[9] *BNB implementation:* https://www.geeksforgeeks.org/implementation-of-0-1-knapsack-using-branch-and-bound/

[10] https://www.tutorialspoint.com/0-1-knapsack-using-branch-and-bound-in-cplusplus

[11] *Kolesar branch and bound:* https://www0.gsb.columbia.edu/mygsb/faculty/research/pubfiles/4407/kolesar_branch_bound.pdf

[12] *Some typical test cases: https://www.researchgate.net/figure/Five-typical-test-cases-of-0-1-knapsack-problems_tbl1_349878676*

[13] https://en.wikipedia.org/wiki/Beam_search#:~:text=In%20the%20context%20of%20a,until%20it%20reaches%20a%20goal.

[14] https://www.geeksforgeeks.org/genetic-algorithms/

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

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

[17] https://darrenhwang1.github.io/Genetic-Algorithms-on-Knapsack-0-1/

[18] https://www.dataminingapps.com/2017/03/solving-the-knapsack-problem-with-a-simple-genetic-algorithm/