University of Science

Viet Nam National University Ho Chi Minh

PROJECT REPORT

SEARCHING FOR THE KNAPSACK

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

Class: 21CLC02

Tutors: NGUYEN TIEN HUY
NGUYEN TRAN DUY MINH

Group members:

21127267 - PHAN VAN BA HAI

21127556 - DO QUOC TRI

21127579 - PHAM BUI TUAN ANH

21127657 - NGUYEN KHANH NHAN

Contents

1 Project Information

1.1 Group Members

- 21127267 - Phan Van Ba Hai: Algorithm 3 + report
- 21127556 - Do Quoc Tri: Algorithm 4 + report
- 21127579 - Pham Bui Tuan Anh: Algorithm 1 + report + output
- 21127657 - Nguyen Khanh Nhan: Algorithm 2 + report + input

1.2 General information

Based on the released topic, there are 4 algorithms need to be done. They are: Brute force searching, Branch-Bound, Local beam search and Genetic algorithm. After meeting, we agreed on some matters.

- Unified the coding style of group and the form of input/output files.
- Split source code into 4 files as 4 different algorithms.
- Decided report content and demos' script.

1.3 Assessment

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

Table 1: Completion level for requirements

2 Brute force searching

2.1 Algorithm's idea

The brute force approach used in the Knapsack Problem software includes producing all feasible combinations of the provided items and then checking each subset to see if it satisfies the problem's requirements.

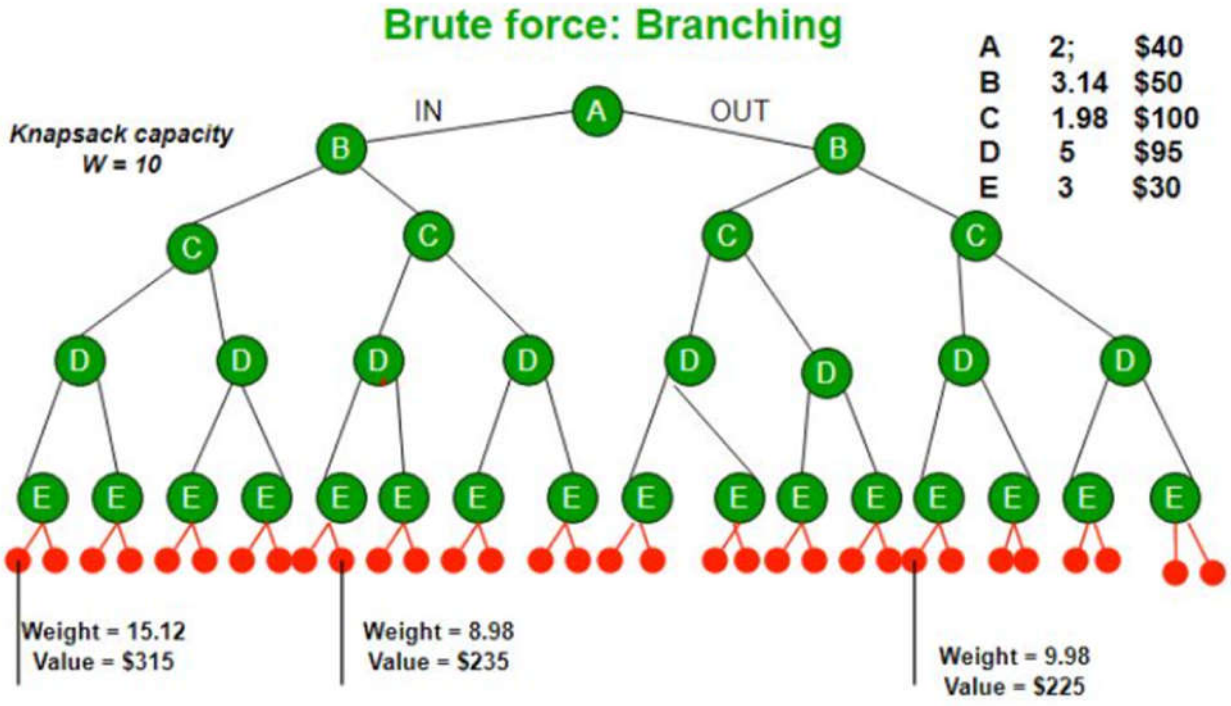

Figure 1: Idea of brute force algorithm in Knapsack problem

More specifically, the algorithm creates every feasible binary combination of length n, where n is the number of items, using the "itertools.product function".

A value of 1 in the i^{th} place indicates that the i^{th} item is included in the combination, whereas a value of 0 indicates that the i^{th} item is not included. Each binary combination represents a subset of the items.

Calculating the combined weight and value of the components in each combination. Determine whether the overall weight of the combination is less than or equal to the knapsack's capacity. If it is, compare the total value of this combination to the highest value found thus far. If the total value is greater than the maximum value found thus far, the maximum value and the accompanying combination should be updated.

The ideal answer to the problem is the combination with the highest overall value that yet fits inside the capacity of the knapsack.

Because the technique generates all potential subsets, its time complexity grows exponentially with the number of elements, making it only practicable for modest issue sizes.

This is a pseudo code to explain the idea of this algorithm:

```
def Brute_Force(capacity, weights[], profits[], classes[], number_class):
    n = number of items
    for each combination in intertools.product:
        weight of current combination = sum(combination[i]*weights[i] for i in range(n))
        if weight of current combination <= capacity:
            combination_value = sum(combination[i]*profits[i] for i in range(n))
            if combination_value > max_val:
                class_check = [0 for i in range(number_class)]
                for i in range(n):
                    if combination[i] == 1:
                        class_check[classes[i] - 1] = 1
                if at least 1 item from each class has been selected:
                    max_val = combination_value
                    best_comb = combination
    return best_comb, max_val
```

2.2 Explaining code

Class "Brute_Force": This will be the class used to execute the algorithm and return the results.

We will filter the data from the input file to get the maximum capacity, the number of classes that the final result needs as well as the list of items with their value, weight and class thanks to the "get_Data()" and the constructor of the Brute_Force class.

First initialize 2 variables "max_value" and "best_comb" to save the result. Variable "n" to store the total number of items in the input.

Use a for loop to test each combination. Skip for next combination if the weight of the current combination is greater than the maximum weight or the value of the combination is less than "max_val". If a combination satisfies the above two conditions, then check that combination to see if at least one item belongs to each class. Then assign the values of combination to "max_val" and "best_comb".

2.3 Visualization and Test case

*Visualization: Brute force explaining and testing video

*Small data set:

File name	Size	Result	Running time (ms)
INPUT_1.txt	30	N/A	N/A
INPUT_2.txt	10	358	4.000663757324219ms
INPUT_3.txt	20	951	3722.8355407714844ms
INPUT_4.txt	25	1012	183426.34391784668ms
INPUT_5.txt	40	N/A	N/A

Table 2: Brute force - small data set

*Large data set:

File name	Size	Result	Running time (ms)
INPUT_6.txt	800	N/A	N/A
INPUT_7.txt	200	N/A	N/A
INPUT_8.txt	400	N/A	N/A
INPUT_9.txt	600	N/A	N/A
INPUT_10.txt	1000	N/A	N/A

Table 3: Brute force - large data set

(Detailed results will be present in output files "OUTPUT_X.txt" and visualization video demo.)

2.4 Conclusion about the algorithm

2.4.1 Evaluation

In this algorithm we always have to go through all possible cases, so the complexity is always $O(n^2)$. So with a large data it will need a very long time to get a result.

Brute Force algorithm has a relatively large complexity and based on our tests our machine can only run up to input with item's size equal to 20. Brute Force is a simple and accessible algorithm that can also handle small data, but it is not optimal and cannot handle large amounts of data.

2.4.2 Pros and Cons

*Advantages:

- The brute force approach is a guaranteed way to find correct solution by listing all the possible candidate solutions for the problem.
- Brute force is a generic method and do not limit to any specific domain of problems. So this is an ideal algorithm for solving small and simpler problems.
- It is known for its simplicity and serve as a comparison benchmark.

*Disadvantages:

- The brute force approach is inefficient. For real-time problems, algorithm analysis often goes above the $O(2^n)$ order of growth. So brute force algorithms are very slow.
- This method relies more on compromising the power of a computer system for solving a problem than on a good algorithm design.
- Brute force algorithms are not constructive or creative compared to algorithms that are constructed using some other design paradigms.

Brute Force is a recommending algorithm only when you have no idea with the problem which has small inputs because of its complexity. Brute force is definitely not an optimizing algorithm in Knapsack problem.

3 Branch and bound

3.1 Algorithm's idea

Similar to Brute Force algorithm, Branch and Bound is a commonly used algorithm design model for solving combinatorial optimization problems. These problems often have an exponential complexity in time and may require exploring all possible permutations in the worst case. However, what is special about Branch and Bound is that it can help us reduce the time and memory space needed by eliminating "bad" cases and selecting the optimal case among the remaining good cases instead of traversing through all possible cases before selecting the optimal one.

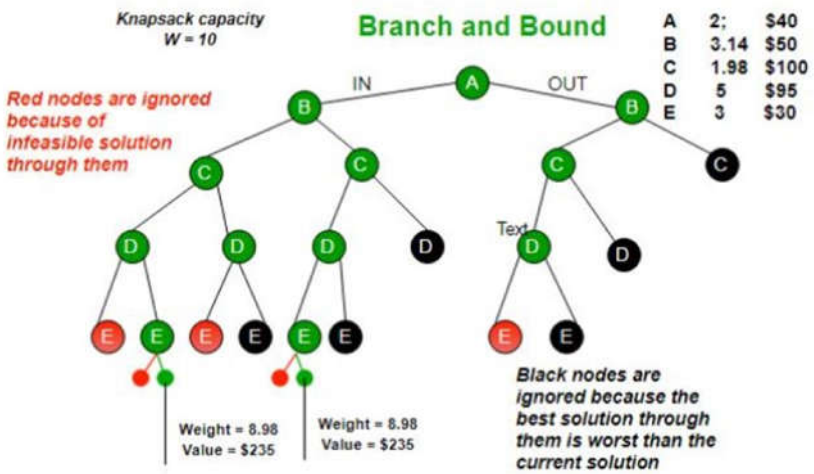

Figure 2: Idea of branch and bound algorithm in Knapsack problem

Specifically, in the Knapsack problem, we will solve the problem requirements by modeling the problem as a binary tree. Each node of the tree show a state of the bag: which items it contains, the total value it contains, the total weight of the items it contains, and the classes of the items in the bag. The first node at level 0 of the tree corresponds to an empty bag. From here, we decide whether or not to put the first item in the bag, which means we will create 2 new nodes at level 1 representing the state of the bag with or without item 1. This way, with n given items, we can construct a binary tree with n levels. And

the result we need to find will be one of the nodes at level n of the tree.

Creating the entire binary tree and then traversing all leaf nodes (i.e., nodes at level n) has been done in the Brute Force algorithm, and we see that it consumes an enormous amount of memory and time because it requires creating a complete binary tree with n levels. Fortunately, Branch and Bound has overcome all these weaknesses.

Instead of traversing the entire tree, now Branch and Bound will only expand a branch if it sees that the nodes on that branch have the potential to lead to a good case, i.e., from that branch, it will lead to nodes with a total weight not exceeding the maximum weight allowed and containing all items belonging to all classes. If either of the two requirements is not met, the branch will be immediately "cut off." This helps our algorithm save time and memory because it doesn't have to expand bad nodes.

Finally, we will filter out the nodes that meet both requirements above and select a node with the best total item value. This is also the answer to the Knapsack problem.

This is a pseudo code to explain the idea of this algorithm:

```
# function to solve a problem
func solve(capacity, number_class, items):
    sort(items, key = items.prio, reverse = True)
    stack = [{items: [], total_v = 0, total_w = 0, total_class = []}]
    resultNode = [{items: [], total_v = 0, total_w = 0, total_class = []}]
    While stack is not empty:
        curNode = stack.pop()
        i = level of curNode
        if curNode is a leaf and number of classes in curNode = number_class:
            if total value in curNode > total value in resultNode:
                resultNode = curNode
        else:
            NotAddItem_Node = curNode
            AddItem_Node = curNode.create(items[i])
            if isPromissing(NotAddItem_Node, capacity,
```

```
    number_class, items, resultNode):
16                  stack.push(NotAddItem)
17              if isPromissing(AddItem_Node, capacity,
    number_class, items, resultNode):
18                  stack.push(AddItem)
19      return resultNode
20
21  # function to check if a branch is promissing to be expanded
22  func isPromissing(Node, capacity, number_class, items,
    resultNode):
23      if  total weight in Node <= capacity
24      and getBound(Node, capacity, items) > total value in
    resultNode
25      and Check_Class(Node, items, number_class) = True:
26          return True
27      return False
28
29  # function to get the bound
30  func getBound(Node, capacity, items):
31      bound = total value in Node
32      remaining_w = capacity - total weight in Node
33      for all remaining items:
34          if weight of item <= remaining_w:
35              remaining_w -= weight of item
36              bound += value of item
37          else:
38              bound += item.prio * remaining_w
39      return bound
40
41  # function to check if all the class is used
42  func Check_Class(Node, items, number_class):
43      for all remaining items:
44          number of classes in Node += number of different
    classes
45          if number of classes in Node == number_class:
46              return True
47          else:
48              return False
```

3.2 Explaining code

Class Item: (Each item has these atributes)

- Weight - weight of an item.
- Value - value of an item.
- Class - class(type) of an item.
- Prio - priority of an item (the less weight the more priority, the more value the more priority).
- Pre_position - item's position before the sorting by priority (use to check if an item is picked in the final result).
- New_postion - item's position after the sorting by priority.

Class Node: (Each node represent a status of the knapsack)

- Total_weight - total weight of items in the knapsack.
- Total_value - total value of items in the knapsack.
- Total_class - a list to check if a class has appeared in the knapsack.
- Total_item - a list of binary value (0, 1), which is used to show if an item is picked or not.

Class Branch_Bound is used to execute the algorithm and return answer, which is the total_item atribute of class Node.

At first, we read the input file to get the maximum weight allowed, number of classes and item list. Then use Branch_Bound's constructor to init all the data. Next, sort the item list based on "prio", update "pre_pos" and "new_pos" to identify item's position which will be used to calculate final result. Generate an empty status of the knapsack.

Next, we initiate the first element in the stack is the node with empty status. Then start to scan entire stack until no element left. This means that we scanned all leaf nodes (a leaf node is identified if variable "i"=number of income items). Lastly, we compare this leaf node with result node. Update the result node by that leaf node when its value is better than the current result.

Every single time we do a loop, pop out the first element from stack.

From this moment, 2 kid-nodes is "AddItem" and "NotAddItem" appear. Each node will be added to stack if its "isPromissing()" (this function will be shown below) return true result. On the other hand, 1 or 2 nodes will be pass through, it looks like cutting a branch. We choose to push "NotAddItem" node before "AddItem" node because we are using stack, so the "AddItem" branch will run first and it has the higher chance to reach the final result quickly.

So why we choose to use stack instead of queue? The answer is that we need LIFO organization of stack to do a depth-first-search. So that we can reach a single leaf node as much faster than using queue. This first leaf node is used as a best result, by using this, we can start cutting off branches sooner.

By the time the loop end, the result is returned and it is exactly the best node we have after expand search tree.

Function "isPromissing()" will decide to expand a node if it can satisfy 3 of these conditions:

- Does total weight of all items in that node is equal to or higher than maximum weight
- Based on total weight of items in the node, we will calculate the peak weight that we can reach (in theory). If that is higher than the current weight, we should try to expand this branch.
- Check if left over items' classes is not enough, we will decide to cut off this branch.

Finally, we can reach the best node with the highest value. Now, with 'pre_position' and 'new_position' we can consider that which item is picked and which one is not. Then, we have the final result.

3.3 Visualization and Test case

*Visualization: Branch and Bound explaining and testing video

*Small data set:

File name	Size	Result	Running time (ms)
INPUT_1.txt	30	752	0.9999752
INPUT_2.txt	10	358	1.0004043
INPUT_3.txt	20	951	1.0025501
INPUT_4.txt	25	1012	0.9994506
INPUT_5.txt	40	1922	1.0128021

Table 4: Branch and Bound - small data set

*Large data set:

File name	Size	Result	Running time (ms)
INPUT_6.txt	800	25383	158.235078
INPUT_7.txt	200	9518	28.0065536
INPUT_8.txt	400	19578	96.0221290
INPUT_9.txt	600	27888	219.046831
INPUT_10.txt	1000	46802	844.697475

Table 5: Branch and Bound - large data set

(Detailed results will be present in output files "OUTPUT_X.txt" and visualization video demo.)

3.4 Conclusion about the algorithm

3.4.1 Evaluation

In the best case, the algorithm only needs to follow a single path and the leaf node of that path is the result, so we will "cut off" almost the entire remaining branch, resulting in a complexity of O(n). On the other hand, in the worst case, we need to traverse the entire tree, resulting in a complexity of $O(2^n)$.

There are many ways to execute this algorithm(using queue or recursion). However, for large data, using a queue will be slower because there will very few branches be cut off, and recursion will be limited. Therefore, using loops and stacks will greatly optimize our algorithm.

In addition, we also apply another method to optimize the algorithm, which is to sort the item list by decreasing "prio" value. This value is in direct proportional to the value of an item and inversely proportional to the weight of an item. This helps us prioritize to put items with high value and light weight into the bag.

Finally, for more optimization, we can define more conditions in cutting off the branches instead of 3 ones we have delivered above.

3.4.2 Pros and Cons

For test cases with a small number of items and classes, the Branch and Bound algorithm always produces the optimal result in an extremely short time. This demonstrates the superior productivity of Branch and Bound over Brute Force. Thanks to the "cutting off" bad nodes/branches that the execution of this algorithm is always fast and delivers optimal results.

Even with large input data sizes, Branch and Bound still produces optimal results and executes extremely quickly. This is an ability that Brute Force cannot achieve. With such large test cases, Brute Force takes a very long time to execute.

*Advantages:

- We do not explore all of the nodes in the tree in a branch and bound algorithm. As a result, the branch and the bound algorithm have a lower time complexity when compared to other algorithms.
- The algorithm will almost always find the optimal solution in a very short time if the pruning conditions are optimal.
- The branch and the bound algorithm find the shortest path to the best solution for a given problem. While exploring the tree, it does not repeat nodes.
- One significant advantage of branch-and-bound algorithms is that we can control the quality of the expected solution, even if it has not yet been found. The cost of an optimal solution is only up to the cost of the computed solution.

*Disadvantages:

- The branch and bound algorithm take time. The number of nodes in the tree may be too large in the worst case, depending on the size of the given problem.
- Also, parallelism in the branch and bound algorithm is complicated.

Overall, the code is well-structured and easy to follow, and the algorithm is correctly implemented. The algorithm is more optimal because it checks the potential value and weight ratios of the items before selecting them.

4 Local beam search

4.1 Algorithm's idea

Beam Search algorithm is a heuristic search algorithm that starts with a set of n initial states. Particularly in the Knapsack problem, each state represents the state of the bag after searching through n items, where value is 1 (select that item), or 0 in the opposite. It then generates next generation from these states by flipping the values from 1 to 0 or 0 to 1 at each position, so that from one state, it produces as many descendants as the number of items.

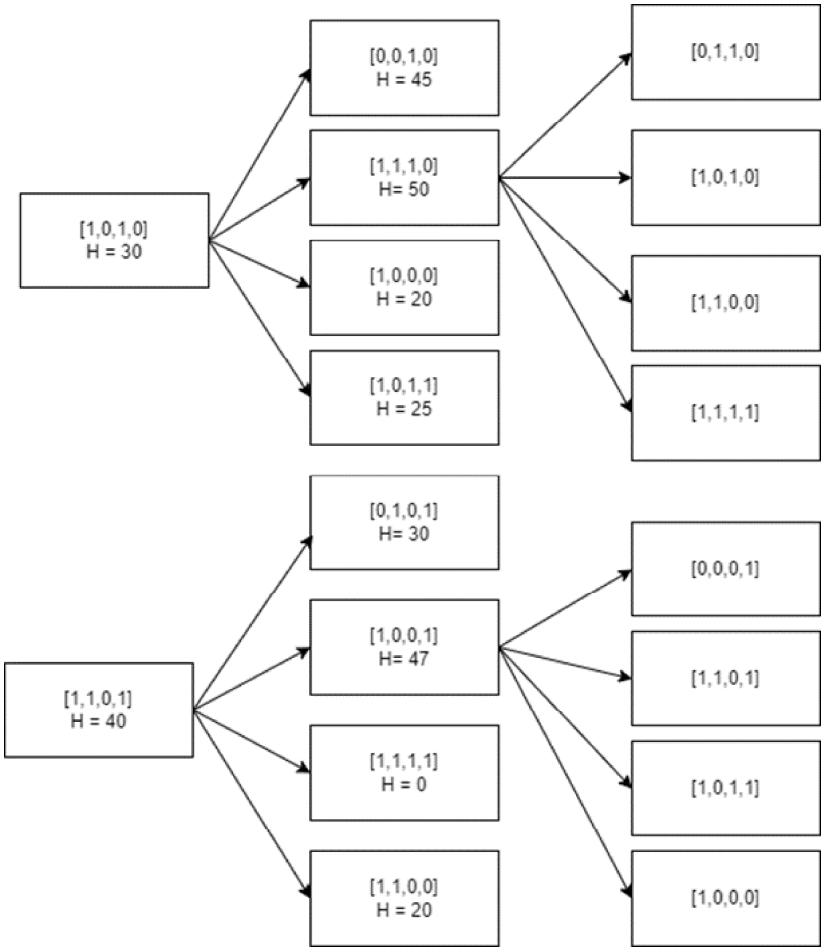

Figure 3: Example of beam search in Knapsack problem

A heuristic search algorithm that examines a graph by extending the most promising node in a limited set is known as beam search.

Beam search is a heuristic search technique that always expands the W number of the best nodes at each level. It progresses level by level and moves downwards only from the best beam_width nodes at each level. Beam Search uses breadth-first search to build its search tree. Beam Search constructs its search tree using breadth-first search. It generates all the successors of the current level's state at each level of the tree. However, at each level, it only evaluates a beam_width number of states. Other nodes are not taken into account.

The heuristic cost associated with the node is used to choose the best nodes. The width of the beam search is denoted by W. If B is the branching factor, at every depth, there will always be W × B nodes under consideration, but only W will be chosen. More states are trimmed when the beam width is reduced.

However, in many cases the knapsack problem exhibits certain partial assignments that are intuitively desired to appear in the final solution. So with a good assignment of initial states we should be able to converge to an optimal solution. This is a pseudo code to explain the idea:

```
# function to solve
func solve(max_loop):
    loop_times = 0
    current_nodes = Initialize_Nodes()
    best_node = max(current_nodes, key = Heuristic())
    best_value = Heuristic(best_node)
    while loop_times <= max_loop:
        loop_times += 1
        for node in current_nodes:
            next_nodes += Create_Successors(node)
        current_nodes = sorted(next_nodes, key = Heuristic(),
    reverse = True)
        best_current_node = max(current_nodes, key =
   Heuristic())
        if Heuristic(best_current_node) > best_value:
            best_value = Heuristic(best_current_node)
            best_node = best_current_node, loop_time = 0
    return best_node, best_value

# function to find heuristic value
```

```
func Heuristic(node):
    node_w, node_v, node_class = 0, 0, [0, 0, 0,...]
    for i in range(n):
        if node[i] == 1:
            node_w += item[i].weight
            node_v += item[i].value
            node_class[item[i].class - 1] += 1
    if node_w > capacity or not all( c > 0 for c in
   node_class): return -1
    return node_v

# function to init a node
func Initialize_Nodes():
    nodes = []
    for i in range(beam_width):
        node = [random(0, 1) for n times]
        nodes.append(node)
   return nodes

# function to create final status node
func Create_Successors(node):
    successor_list = []
    for i in range(n):
        successor_node = node
        successor_node[i] = 1 - successor_node[i]
        successor_list.append(successor_node)
   return successor_list
```

4.2 Explaining code

Class "Beam_Search": This is a class to execute the algorithm. Here is some attributes of it

- self.k - beam width
- self.n - quantity of items
- self.max_loop - a bound to limit loop times when next generation is worse than itself. After "self.max_loop" loop times, we will return the best possible result (may not be the most optimized result).

Function "Heuristic(Node)": This is used to calculate Heuristic value of a node/status. A valid state's Heuristic value is equal to its total

value and -1 for invalid one.

Function "Create_Successors(Node)": This is for creating "n"(beam width) next generation states of a node. This function will reverse value of each item in the item list. For example [1,0,1] will create 3 next-gen states is [0,0,1], [1,1,1], [1,0,0].

Function "Initialize_Nodes()": This function is to initiate "n"(beam width) start status for preparation.

Function "solve()": This is an execution function. First, from the initial states, we will find the state with the best heuristic value and heuristic score to assign to the two result variables "best_node" and "best_value". We will continue looping until we do not find any state better than "best_node" after "max_loop" iterations. In each iteration, we will generate descendant states from the "current_nodes". Then select the states with the best heuristic value to replace the states in "current_nodes" and also compare the state with the best heuristic value with "best_node" and "best_value". If it is better, we will update them. At the end of the loop, we will return "best_node" and "best_value", which are the solutions to the problem.

4.3 Visualization and Test case

*Visualization: Local beam search explaining and testing video

*Small data set: (beam_width = 200, max_loop = 500)

File name	Size	Result	Running time (ms)
INPUT_1.txt	30	752	9927.850723
INPUT_2.txt	10	358	2156.4846
INPUT_3.txt	20	951	7287.63532
INPUT_4.txt	25	1012	10213.23966
INPUT_5.txt	40	1896	23702.15487

Table 6: Local Beam search - small data set

*Large data set: (beam_width = 300, max_loop = 1200)

File name	Size	Result	Running time (ms)
INPUT_6.txt	800	N/A	N/A
INPUT_7.txt	200	9501	1748820.29581
INPUT_8.txt	400	N/A	N/A
INPUT_9.txt	600	N/A	N/A
INPUT_10.txt	1000	N/A	N/A

Table 7: Local Beam search - large data set

(Detailed results will be present in output files "OUTPUT_X.txt" and visualization video demo.)

4.4 Conclusion about the algorithm

4.4.1 Evaluation

Through the above test cases, we can easily find out that the Beam Search works well in small-scale data sets. When facing the large-scale data sets, because the Beam Search will have to generate all possible successor_states so it can choose the k best one, it has a high chance to cause data overflow.

When beam width = 1, the search becomes a hill-climbing search in which the best node is always chosen from the successor nodes. No states are pruned if the beam width is unlimited, and the beam search is identified as a breadth-first search.

The beam_width bounds the amount of memory needed to complete the search, but it comes at the cost of completeness and optimization (possibly that it will not find the best solution).

4.4.2 Pros and Cons

*Advantages:

- The Beam Search algorithm is pretty fast compared to other variants of greedy search.
- The Beam Search will also work well when given small-scale data sets or when it can create initial nodes has good heuristic value.

*Disadvantages:

- The cost of completeness and optimal solution is the running time of the algorithm.
- If the running time is low, it usually won't give the optimal solution.
- If we want the optimal solution, we must adjust the beam width or "max_loop", which may take a lot of time to wait for the solution.
- So for the large data sets, we should not use local beam search.

Local Beam Search is a heuristic search algorithm that relies on the heuristic value to determine the best result. However, finding the best result is not necessarily the optimal result, and optimizing the result requires increasing the search time. This is a trade-off that is not always feasible and acceptable. Therefore, in summary, the Local Beam Search algorithm should only be applied to small data sets, while for large data sets, you must accept the time loss to get the closest optimal result or find a way to create good enough initial states to quickly reach the goal state.

5 Genetic algorithm

5.1 Algorithm's idea

When it comes to optimization algorithms, we are trying to replace a brute-force search over the whole problem space with a local search that filters out as many points in the problem space as possible: finding the same result by searching a much smaller domain.

Nature has been a great source of inspiration for mathematicians and computer scientists. Generic algorithms find the individual with the best fitness after a certain number of generations evolved from the original population. Some biology terms such as natural selection, crossover, mutation are applied in these algorithms as functions to solve searching problems.

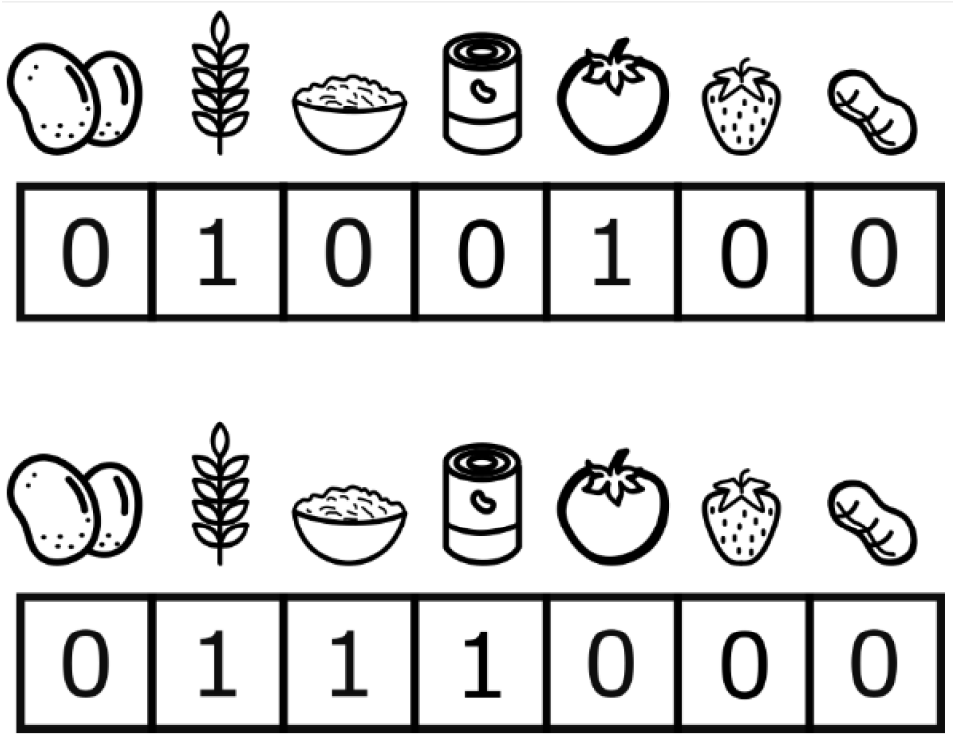

Figure 4: Idea of genetic algorithm in Knapsack problem

In nature, individuals of the same species reproduce their offsprings by crossing over the parents' chromosomes with a small rate of mutation. The children with the best fitness survive and reproduce the next

generation while the less fit children have lower chances to compete with others or die due to the lack of adaptation. Mutation plays an important role in giving the population variety and competitions help it find the best genes for the next generation.

Applying all of them in solving the 0-1 Knapsack problem with a condition that at least one item of each class has to be taken, the answer is the chromosome of the fittest individual which holds the value of zeros and ones represent whether an item is taken. The fitness of each individual is determined by the sum of value of every taken item if the total weight is not greater than the knapsack weight limit and every class has at least one item taken. This is a pseudo code to explain the idea:

```
# Class to store a single state
class Individual:
    constructor(knapsack, item_list):
        list_point = [] -> sort by optimal_point
        median = list_point[len / 2]
        for i -> number of items:
            if optimal_point < median:
                init (high rate)
            else:   init (low rate)
        if value valid:
            fitness = value
        else:   fitness = 0

    func update(chromosome, knapsack, item_list):
        for first_item -> last_item:
            if gene = 1:
                value += item's value
                total_weight += item's weight
        if total_weight invalid:
            total_weight = 0

# Class to get entire population of a generation
class Population:
  constructor(pop_size, knapsack, items[]):
    while this.pop.size < pop_size:
      this.pop.append(new Individual(knapsack, items[]))

  func get():
```

```
29      return this.pop
30
31  # Class to create children for next population
32  class Crossover
33    func apply(mom_chromosome, dad_chromosome) ->
       child_chromosome:
34      child_chromosome = []
35      cross_point = randint(0, len(mom_chromosome) - 1)
36
37      if randfloat(0, 1) < 0.5:
38        child_chromosome = mom_chromosome[begin : cross_point]
       + dad_chromosome[cross_point : end]
39      else:
40        child_chromosome = dad_chromosome[begin : cross_point]
       + mom_chromosome[cross_point : end]
41      return child_chromosome
42
43  # Class to create a mutation for diversity
44  class Mutation:
45    contructor(rate_mutation)
46
47    func apply(individual):
48      for gene in individual.chromosome:
49        if randfloat(0, 1) < this.rate:
50          gene = 1 - gene
51
52  # Class to filt top "strongest" gene for breeding
53  class Elitisim:
54    contructor(rate):
55      this.rate = rate
56
57    func apply(population) -> population:
58          sort -> population(by fitness)
59          new_pop = population[0 -> rate * population]
60
61          for i -> new_pop_length:
62              if chromosome dupplicated:
63                  remove all dupplicates
64                  add other chromosome
65    return new_pop
66
```

```
# Class to do natural selection
class Nature:
  contructor(elitism, select4mating, crossover, mutation)

  func select(population, knapsack, items[]) -> new
   population
    new_pop = this.elitism.apply(population)
    while new_pop.size < population.size:
      valid = False
      child = population[0]
      while valid == False:
        mom = new_pop[random]
        dad = new_pop[random]
        child = crossover(mom, dad)
                child -> mutation
                child -> update_value()
        return new_pop

# Class to solve the problem
class GeneticAlgo:
    constructor(generation, population, nature)

    func solve(knapsack, item_list) -> best chromosome
        population.constructor(population, knapsack,
   item_list)
        for i -> generation:
            population = nature(population, knapsack,
   item_list)
        sort -> population
        return population[0]
```

5.2 Explaining code

Class “Individual”: An naive approach for initializing an individual is to random between zero and one at the rate of 0.5 to decide whether an item is taken or not. As a result, the first generation has poor fitness leading to the searching process being time consuming and the final result has a large difference from the optimal solution when the data set has a numerous amount of low-value-high-weight or high-value-low-weight individuals. Therefore, the rate decides if an item is taken or not has to be depended on both the weight and the value of an item.

There is a variable "optimal_ratio" (equal to value divide by weight). Based on this attribute, we can find the median ratio, which will affect the appear rate of a unique item.

Class "Population": Population generator initializes a population at a given size passed in the constructor of the class. All individuals are considered "good" because of their construction process mentioned in the previous part.

Class "Crossover": To find, reserve and create the good answer (chromosome) through many generations, crossover operation randomly combines the first and second parts of chromosomes from different parents selected by tournaments.

Class "Mutation": To enhance the variety of the population, a small percentage of mutation is applied. Whenever a child is born, the algorithm traverses its chromosome and creates a random number in the range of 0 and 1. If the random number is lower than the mutation rate, it flips the bit at that gene. Although mutation occurs rarely, it still affects the population when the number of generations is large enough.

Class "Nature": Just like its name, the nature class represents nature in the program that provides the Select operation. By passing in the number of generations and the population size, the programmer has the ability to reduce the number of loops being performed so the runtime can be altered from short to long. The more generations being created the more accurate the algorithm reaches.

Class "Elitism": A method to filter a unique amount of "strong" population to support the crossover method. Thanks to this class, the next generation population will have a better "gene".

Class "Genetic_Algorithm": With all of the following features, the generic algorithm search for the answer.

5.3 Visualization and Test case

*Visualization: Genetic algorithm explaining and testing video

As this algorithm working based on randomization, all the data collected is the average value of 10 latest test. The executable time is also affected by many variables (population, generation, mutation rate, etc) so every statistics is only for reference.

*Small data set: (generation = 20, population = 200)

File name	Size	Result	Running time (ms)
INPUT_1.txt	30	752	669
INPUT_2.txt	10	358	264
INPUT_3.txt	20	947	471
INPUT_4.txt	25	1012	587
INPUT_5.txt	40	1909	910

Table 8: Genetic algorithm - small data set

*Large data set: (generation = 50, population = 500)

File name	Size	Result	Running time (ms)
INPUT_6.txt	800	21403	29018
INPUT_7.txt	200	9216	6252
INPUT_8.txt	400	18375	12965
INPUT_9.txt	600	25136	21026
INPUT_10.txt	1000	37545	37926

Table 9: Genetic algorithm - large data set

(Detailed results will be present in output files "OUTPUT_X.txt" and visualization video demo.)

5.4 Conclusion about the algorithm

5.4.1 Evaluation

The time complexity of the whole algorithm is really hard to evaluate so we evaluate the complexity of each operation performed in the algorithm then the whole algorithm. The input has N items and each of them is in one of C classes.

Individual initialization (Indiv) traverses the answer list and calculates the total weight, total value and sets the bits so the time complexity is O(N). After that, it checks whether all the classes are included so the time complexity is O(C).

The total time complexity (Indiv): O(N + C) = O(N) when N >> C

The population initialization (Pop) creates a population of size S. For each loop, the population appends a new individual.

The total time complexity (Pop): O(S.N)

The crossover operation (Cross) copies 2 parents' chromosomes to the child's.

The time complexity (Cross): O(N)

The mutation (Mut) traverses the chromosome of each new child.

The time complexity (Mut): O(N)

The elite selection (Eli) in the original population has a sort operation on the population of size S having the time complexity of O(S.logS) and copies the top T% DISTINCT individuals to the new population having the time complexity of O(T%.S^2)for having 2 loops (the outer one traverses original population of size S, the inner one traverses the new population of size T%.S).

Time complexity (Eli): O(S.logS + T%.S^2) = O(S^2), T = const

The nature selection (NS) creates a new population from the original population with a size of S individuals. At the beginning, select the elite with O(S^2) time complexity. Then, it starts a loop to fill the new population with the remaining size of (1-T%).S until the size of S is satisfied. At each loop, crossover, mutation and individual initialization are applied so the time complexity is O(N + N + N) = O(N).

Time complexity (NS):

O(S^2 + (1-T%).S.N) = O(S^2 + S.N), T = const

Finally, the whole generic algorithm generates G generations and finds the best individual with the highest fitness in the population with S individuals.

Total time complexity: **O(G.(S^2 + S.N) + S) = O(G.S^2 + G.S.N)**

G: number of generations, S: size of population, N: number of items

5.4.2 Pros and Cons

*Advantages:

- Running time and answer accuracy might be adjusted by the programmer.
- After a certain number of generations and reasonable population size, a good answer (might not be optimal) could be expected.
- Works well with large data sets for "good" answers.

*Disadvantages:

- Finding the best solution might take plenty of time.
- Good answers are given instead of the most optimal solution.

(Future development) There are plenty of parameters in the algorithms that can be adjusted such as the number of generations (G), population size (S), mutation rate (S), taken rate (R) of items so we may find the optimal vector (G, S, M, R,...) that reduce the time taken for solving this specific problem.

6 References

A. Branch and Bound algorithm - Idea

B. Brute force algorithm - Intertools idea

C. Branch and Bound algorithm - Idea

D. Branch and Bound algorithm - Example running

E. Local beam search - Idea

F. Local beam search - Example running

G. Genetic algorithm - Idea