Vietnam National University, Ho Chi Minh City

UNIVERSITY OF SCIENCE

Artificial Intelligence

Project 01 - Searching

Students: 21127378 – Lê Chính Nhân

21127682 – Vũ Minh Quỳnh

21127731 – Nguyễn Trọng Tín

Class: 21CLC10

Guide Teacher: Nguyễn Tiến Huy

Nguyễn Trần Duy Minh

Bùi Duy Đăng

Ho Chi Minh City – 2023

TABLE OF CONTENTS

I. TEAM INFORMATION

Student ID	Full Name	Email
21127378	Lê Chính Nhân	lcnhan21@clc.fitus.edu.vn
21127682	**Vũ Minh Quỳnh**	**vmquynh21@clc.fitus.edu.vn**
21127731	Nguyễn Trọng Tín	nttin21@clc.fitus.edu.vn

II. ASSIGNMENT PLAN

ID	Description	Responsibility
1	Analyzing Algorithm 1 - Brute force searching	Tín
2	Analyzing Algorithm 2 - Branch and bound	Nhân
3	Analyzing Algorithm 3 - Local beam search	Quỳnh
4	Analyzing Algorithm 4 - Genetic algorithms	Quỳnh
5	Generating 5 small datasets of sizes 10 - 40.	Quỳnh
6	Generating 5 large datasets of sizes 50 - 1000	Quỳnh
7	Making videos to demo the implementation of each algorithm	All members

III. SELF-ASSESSMENT

ID	Description	Responsibility	Complete (%)
1	Algorithm 1 - Brute force searching	Tín	%
2	Algorithm 2 - Branch and bound	Nhân	100%
3	Algorithm 3 - Local beam search	Quỳnh	100%
4	Algorithm 4 - Genetic algorithms	Quỳnh	100%
5	Generate 5 small datasets of sizes 10 - 40.	Quỳnh	100%
6	Generate 5 large datasets of sizes 50 - 1000	Quỳnh	100%
7	Making videos to demo implementation for each algorithm	All members	100%

IV. WORK REPORT

1. Brute force searching

The brute force algorithm for solving the Knapsack Problem explores all possible solutions by exhaustively generating and evaluating each candidate solution. It generates all possible subsets of the items using the powerset function and computes the total weight and value of each subset. The algorithm selects the subset with the highest value that fits within the capacity constraint. This approach guarantees that the optimal solution

will be found, but it can be computationally expensive for large problem instances with many items.

The brute force algorithm for the Knapsack Problem does not utilize any heuristics or domain knowledge to guide the search process. It systematically examines all possible combinations of items without any optimization or pruning techniques. This makes it less efficient compared to heuristic-based algorithms, as it tends to have high time and space complexity, especially for large problem instances.

The brute force algorithm evaluates each solution individually, without taking into account any partial solutions or intermediate states. It employs a systematic traversal strategy, such as depth-first search or breadth-first search, to explore the entire solution space. This exhaustive exploration ensures that no potential solution is overlooked, but it can be computationally expensive.

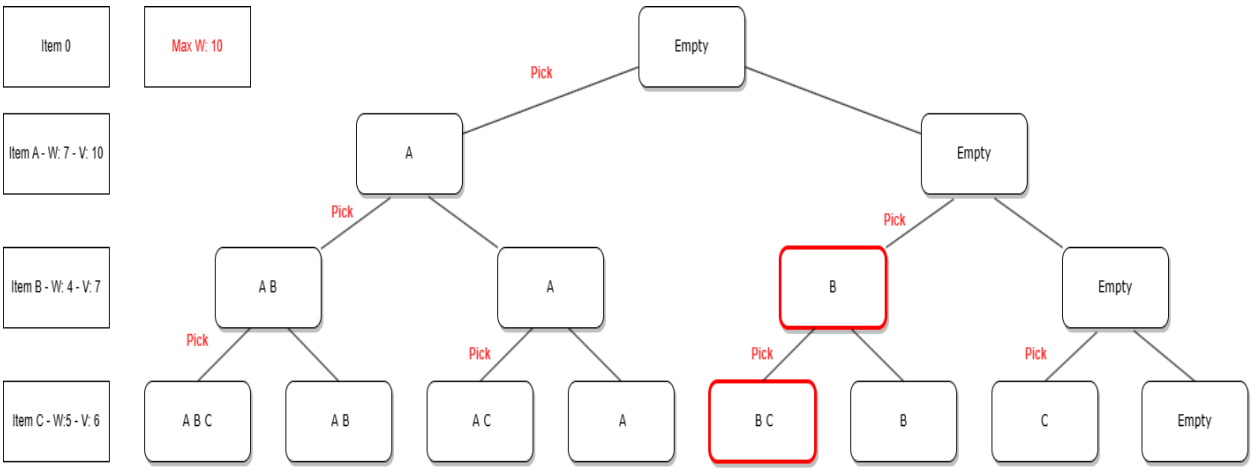
Although the brute force algorithm for the Knapsack Problem guarantees that it will find the optimal solution if one exists within the search space, it is not optimal in terms of finding the best solution. It does not prioritize or consider the quality or optimality of the solutions during the search process. Therefore, it may require considerable computational resources and time to evaluate all possible solutions, especially when the search space is large.

In summary, the brute force algorithm for solving the Knapsack Problem exhaustively explores all possible combinations of items to find the optimal solution. It guarantees completeness but lacks efficiency and optimality. The algorithm systematically generates and evaluates all possible solutions, making it suitable for problems with small search spaces but less practical for larger or more complex problems.

Pseudocode:

```
function knapsack_brute_force(W, m, weights, values, classes):
    n = length(weights)
    max_value = 0
    best_combination = []

    // Ensure at least one item is selected from each class
    for i = 1 to m:
        if i not in classes:
            return "At least one item is missing from class " + str(i)

    // Iterate over all possible combinations of items
    for i = 0 to 2^n - 1:
        combination = binary_representation(i, n)
        current_weight = 0
        current_value = 0
        class_counter = [0] * m

        // Calculate the weight, value, and class count of the current combination
        for j = 0 to n - 1:
            if combination[j] == '1':
                current_weight += weights[j]
                current_value += values[j]
                class_counter[classes[j]-1] += 1

        // Check if the combination satisfies the constraints
        if current_weight <= W and all(count > 0 for count in class_counter):
            // Update the maximum value and best combination if necessary
            if current_value > max_value:
                max_value = current_value
                best_combination = combination

    return max_value, best_combination
```

Visualization



Explanation

Brute force is a method for solving optimization problems that involve generating all possible solutions and selecting the one that maximizes or minimizes the objective function while satisfying the problem constraints.

Suppose we have a knapsack with a maximum weight capacity of 10, and we have the following items with their weights and values:

Item A: weight 7, value 10

Item B: weight 4, value 7

Item C: weight 5, value 6

For each combination, we calculate the total weight and value of the items included in the knapsack. We then check if the total weight is less than or equal to the maximum capacity of the knapsack. If it is, we update the maximum value and the items to include in the knapsack if the total value is greater than the current maximum value.

Item A	Item B	Item C	Total Weight	Total Value	Max Value	Max Items
0	0	0	0	0	0	[]
0	0	1	5	6	6	[C]
0	1	0	4	7	7	[B]
0	1	1	**9**	13	13	**[B],[C]**
1	0	0	7	10	13	[A]
1	0	1	12	16	16	[A],[C]
1	1	0	11	17	17	[A],[B]
1	1	1	16	23	23	[A],[B],[C C]

As we can see from the table, the maximum weight capacity of the knapsack is 10, and the maximum weight that can be carried is 9 by selecting items 2 and 3. Therefore, the optimal selection of items to maximize the total value while not **exceeding** the weight capacity of the knapsack is items 2 and 3, with a total value of 13 and a total weight of 9.

2. Branch and bound

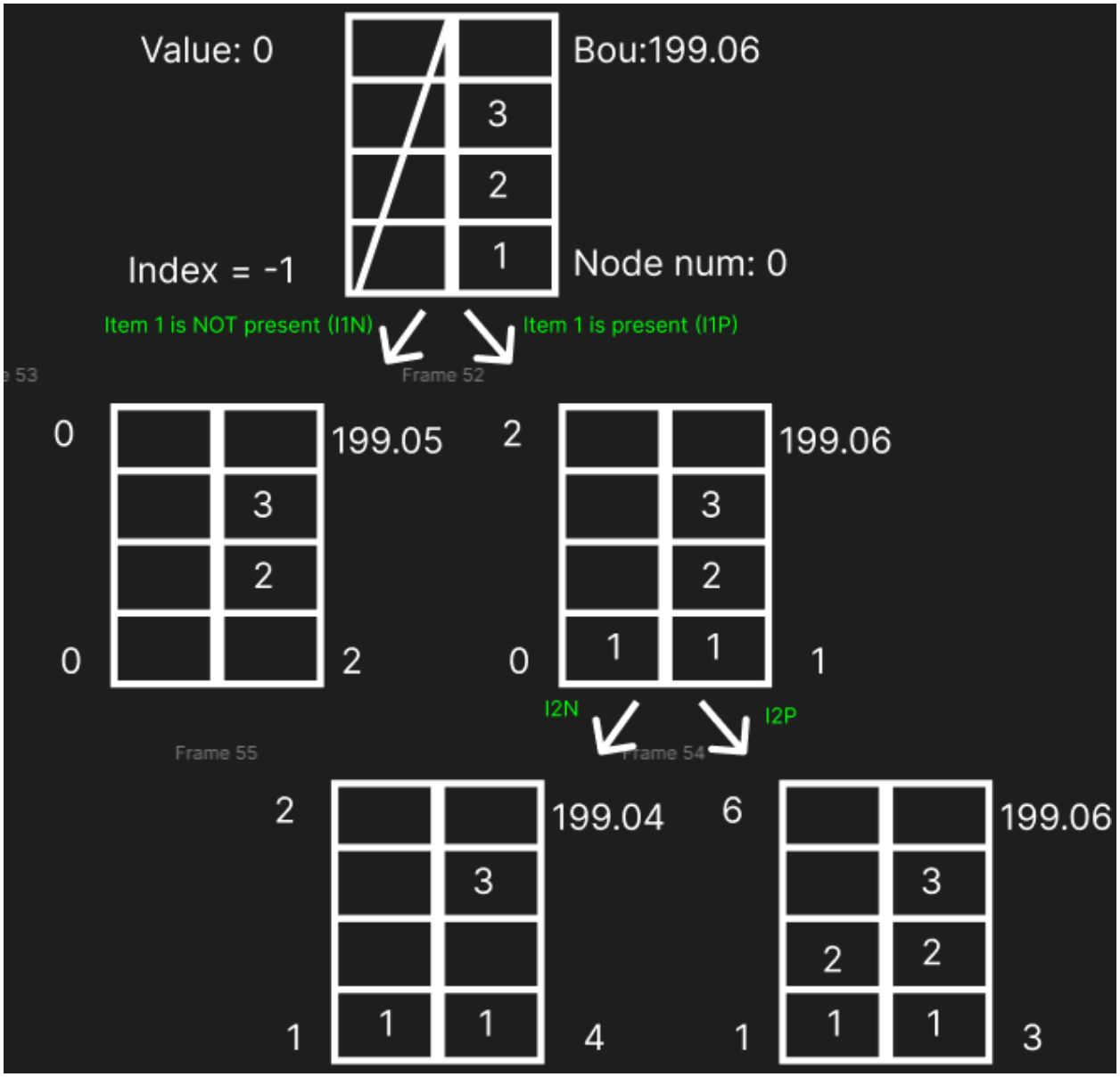
The Branch and Bound (B&B) algorithm is a powerful technique used to solve combinatorial optimization problems, including the well-known Knapsack Problem. The Knapsack Problem involves selecting a subset of items to maximize the total value within a given capacity constraint. The B&B algorithm systematically explores the solution space by generating partial solutions (nodes) and efficiently pruning branches that are guaranteed to not lead to an optimal solution. At each node, an upper bound (also known as a bound or estimate) is computed based on the current partial solution and the remaining items. This bound provides an optimistic estimate of the maximum achievable value, allowing the algorithm to make informed decisions about which branches to explore. By comparing the bound of a node with the current best solution value, the algorithm can selectively explore promising branches and discard unpromising ones. This process continues until all nodes are examined, and the algorithm terminates with the best solution found. The B&B algorithm is particularly effective for solving the Knapsack Problem as it balances thorough exploration of the solution space with intelligent pruning, leading to improved computational efficiency and the ability to find optimal or **near-optimal solutions.**

Pseudocode

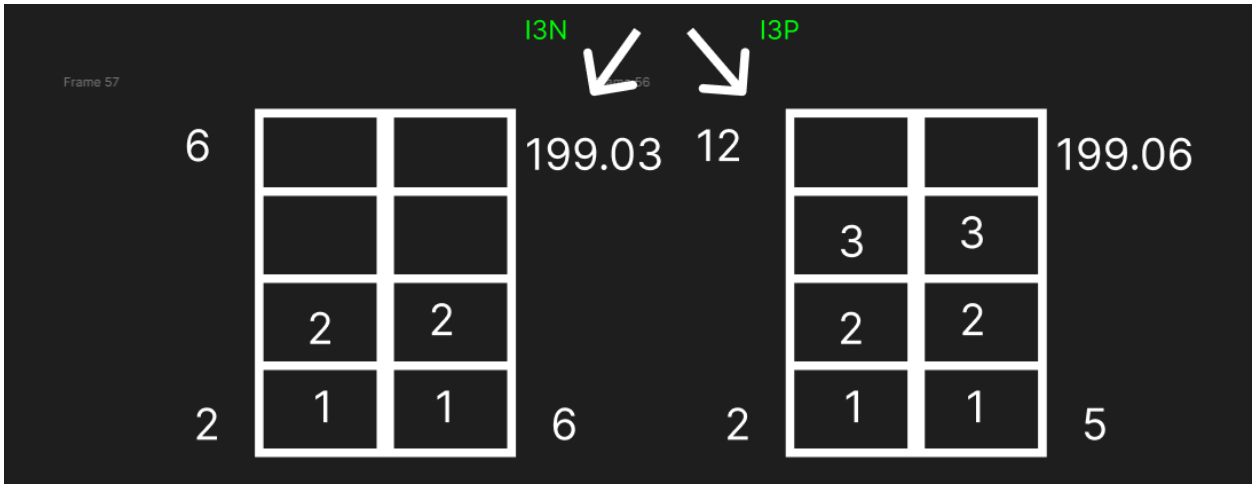
```
knapsack_branch_and_bound(weights, values, capacity):
    items = Create Item objects for each item in the problem
    Sort items based on a value-to-weight ratio in descending order

    priority_queue = Create an empty priority queue
    Create the root node with weight 0, value 0, and index -1
    root.bound = Compute an initial bound value for the root node
    Add root to the priority queue

    while priority_queue is not empty:
        Get the node with the highest bound value from the priority queue

        if node.bound > max_value:
            Update max_items with the included item
            Update max_value with the value of the included item

            level = Determine the level of the next node
            if level is within the item range:
                Calculate the weight of including the next item
                Calculate the value of including the next item

                if new_weight does not exceed the capacity and the bound value is
higher than max_value:
                    Create a new node for including the next item with new_weight,
new_value, and level
                        Compute a bound value for the new node
```

Two nodes are **either put in `priority_queue`** to compare the bound, or the **highest will the taken out** to expand further node num 1 will have a bound value is 199.06 while node num 2 will have a bound value of only 199.05 due to the absence of the first item (the bound calculation of this node will not be demonstrated, as its formula was mentioned above). Therefore, after **comparing the bound value of two leaf nodes**, the node no one will be chosen to continue to find an optimal solution.

Node 1 **will be divided into two other nodes** num 3 and node num 4 which will have item 1 and 2 while node num 4 will not have item 2. Therefore, node num 3 will have a bound value is 199.06 and a node value is 6 while node num 4 only has a bound value is 199.04 due to the absence of item 2. (2+6)+(100-4)*1.99 = 199.04



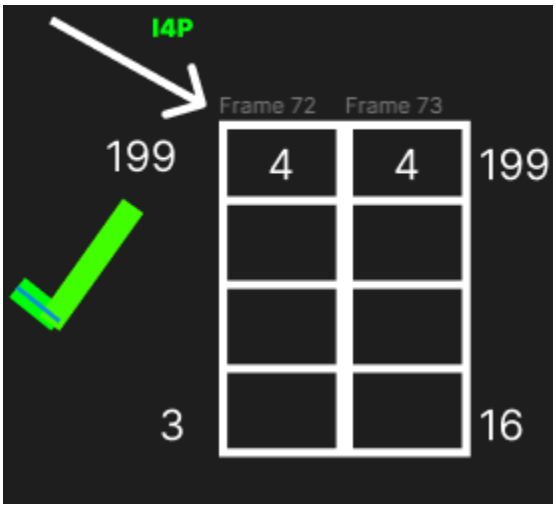
The node num 3 again will be expanded into two new leaf nodes which are node 5 and node 6 one node will have item 3 and the doesn't. Again, the bound value will be calculated and the result will be shown in below image



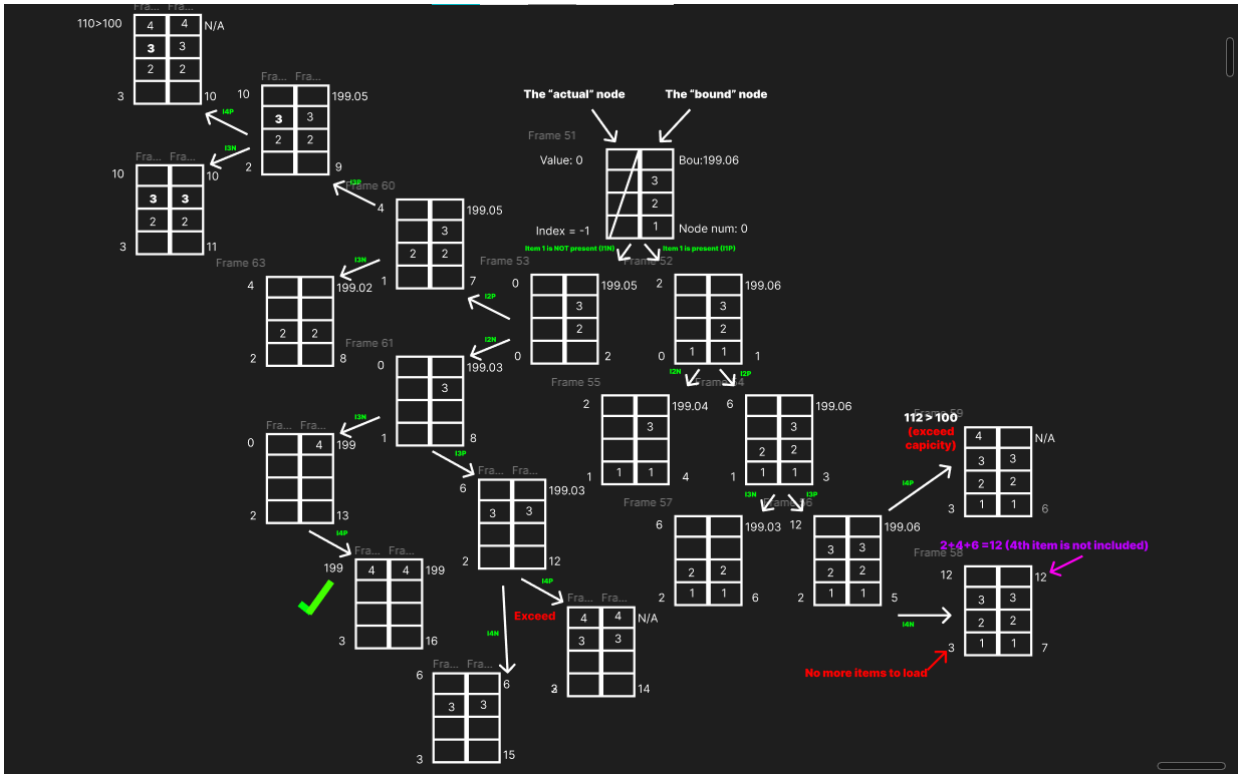
Then, the algorithm will keep calculating the node num 5…

From there, no more items can be included because we already got all the items and it seems that the max value is 12. However, this is when `priority_queue` comes in. The node num 2 is still in `priority_queue` and it now has the highest bound value is 199.05. Therefore, node num 2 will be expanded into two nodes and the cycle will be repeated.

After multiple nodes are checked, the algorithm will stop at the 16th as it found the perfect solution. The 4th item, which weighs 100kg, fits the knapsack and with the highest value is 199 is the best item to be put in the knapsack.



Here are all the nodes that are checked by the algorithm, as shown below:



3. Local beam search

Local beam search is a heuristic search algorithm that explores a graph by expanding the most optimistic node in a limited set. This is a better, and more efficient version of the best-first search algorithm because it reduces the space complexity.

Best-first search algorithm takes all partial solutions base on some heuristic while local beam search only chooses the best partial solutions to store and calculate the next step, thereby making it a greedy algorithm.

Beam search uses breadth-first search to build its search tree. At each level of the tree, it generates all the next nodes of nodes at the current level, sorting them in increasing order of heuristic cost. However, it only stores a number of the best nodes at each level called the **beamwidth (B)**. Only those nodes are expanded next.

Since a goal node could potentially be pruned, local beam search sacrifices completeness (meaning that the algorithm will terminate if it found a solution). Local beam search is not optimal, which means there is no guarantee that it will find the best solution.

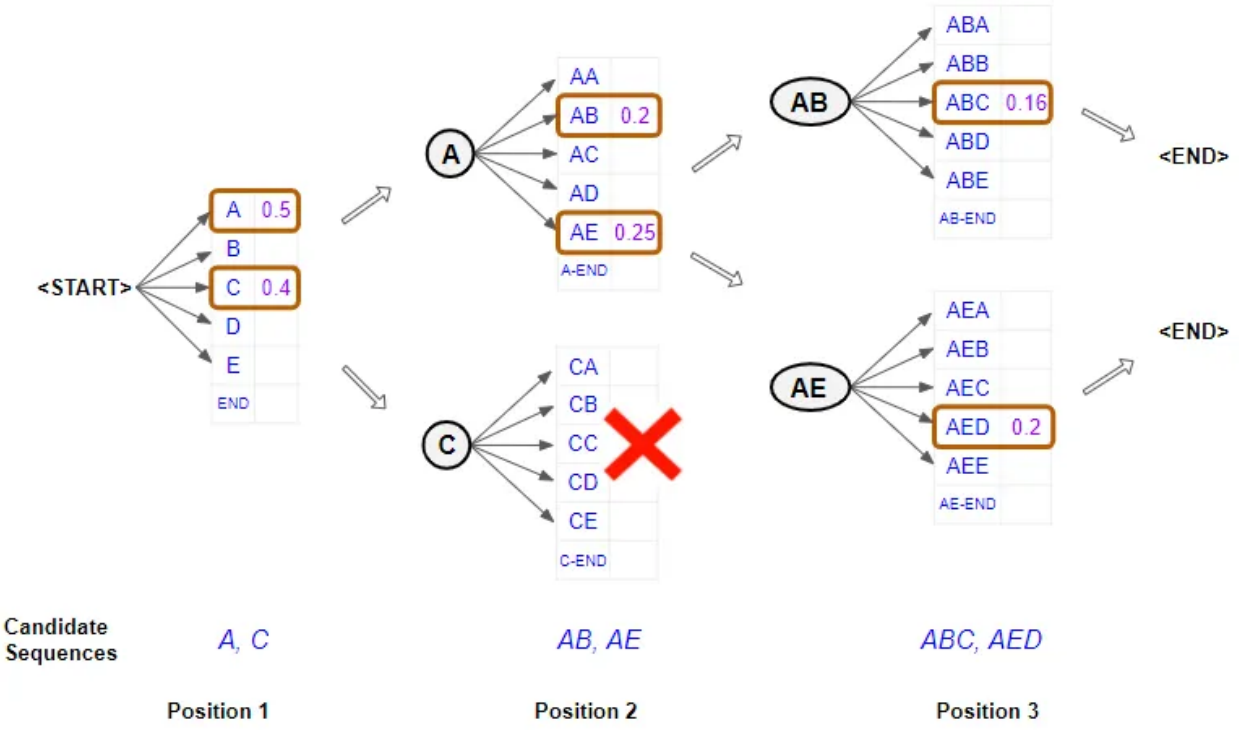
Local beam search algorithm will return the first solution found, so when it reaches the maximum depth, the algorithm will evaluate the nodes during the tree traversal process and take the one with the highest heuristic value.

The **beamwidth (B)** can either be fixed or variable. One of the possible approaches is using a minimum beamwidth at the start. If no solution is found, the beamwidth will increase and the process can be repeated.

Pseudocode

```
beamSearch(problemSet, ruleSet, memorySize)
     openMemory = new memory of size memorySize
     nodeList = problemSet.listOfNodes
     node = root or initial search node
     Add node to openMemory;
     while (node is not a goal node)
          Delete node from openMemory;
          Expand node and obtain its children, evaluate those children;
          If a child node is pruned according to a rule in ruleSet, delete it;
          Place remaining, non-pruned children into openMemory;
          If memory is full and has no room for new nodes, remove the worst
              node, determined by ruleSet, in openMemory;
          node = the least costly node in openMemory;
```

Visualization



Beam Search example, with beamwidth = 2

Explanation

In the beginning, the algorithm starts with the "<START>" icon and gets probabilities for each character (A, B, C, D, E). Then, it will select the two best characters, "A" and "C", and remove the remaining.

After that, "A" and "C" will form two different branches and the algorithm will generate two sets of probabilities corresponding to two branches. The algorithm then picks the overall two best characters based on the probability of branches "A" and "C" and removes the remaining. In this case, it will pick "AB" and "AE" in the branch "A".

Finally, it repeats the process, generating sets of probabilities for each branch and choosing the two best characters. "ABC" in the first branch and "AED" in the second branch are selected to take the next step and all the remaining will be removed.

It will repeat the process like that until it meets an “<END>” icon and it will stop and return the solution.

4. Genetic algorithms

A genetic algorithm is an adaptive heuristic search algorithm inspired by "Darwin's theory of evolution in Nature." This algorithm reflects the process of natural selection where the fittest individuals are selected for reproduction in order to produce offspring of the next generation. Genetic Algorithms are being widely used in different real-world applications, for example, Designing electronic circuits, code-breaking, image processing, and artificial creativity.

The process of natural selection starts with the selection of the fittest individuals from a population. They produce offspring which inherit the characteristics of the parents and will be added to the next generation. If parents have better fitness, their offspring will be better than parents and have a better chance of surviving. This process keeps on iterating and in the end, a generation with the fittest individuals will be found.

Pseudocode

```
Initialize mutationRate, populationRate
Initialize population base on mutationRate and populationRate
Calculate fitness of each individual

repeat until termination condition is met:
    Phase 1: Fitness
    Calculate the fitness of each individual in the population

    Phase 2: Selection
    Select parents from the population based on their fitness (using selection
methods like tournament selection, roulette wheel selection, or rank-based
selection)

    Phase 3: Crossover
    Create offspring through crossover (using crossover methods like single-point
crossover, two-point crossover, or uniform crossover)

    Phase 4: Mutation
```

```
    Apply mutation to the offspring (using mutation methods like bit-flip mutation
or swap mutation)

    Phase 5: Evaluation
    Evaluate the fitness of the offspring

    Phase 6: Replacement
    Select individuals from the population and the offspring to form the next
generation (using replacement strategies like elitism, generational replacement, or
steady-state replacement)

    Calculate fitness of each individual

return best individual
```

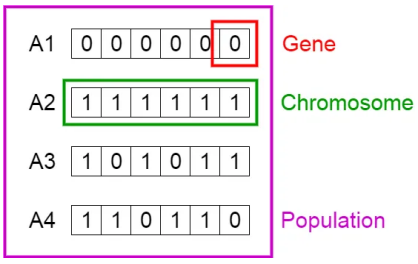
Five phases are considered in a genetic algorithm.

Initial Population

The process begins with a set of individuals which is called a **Population**. Each individual is a solution to the problem you want to solve.

An individual is characterized by a set of parameters (variables) known as Genes. Genes are joined into a string to form a **Chromosome** (solution).

In a genetic algorithm, the set of genes of an individual is represented using a string, in terms of an alphabet. Usually, binary values are used (a string of 1s and 0s). We say that we encode the genes in a chromosome.



Fitness Function

The fitness function determines how fit an individual is (the ability of an individual to compete with other individuals). It gives a fitness score to each individual. The

probability that an individual will be selected for reproduction is based on its fitness score.

Selection

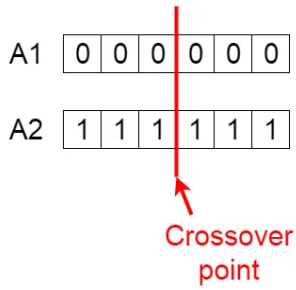
The idea of the selection phase is to select the fittest individuals and let them pass their genes to the next generation.

Two pairs of individuals (parents) are selected based on their fitness scores. Individuals with high fitness have more chances to be selected for reproduction.

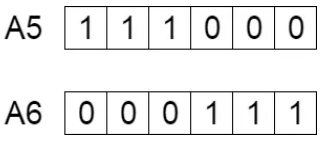
Crossover

Crossover is the most significant phase in a genetic algorithm. For each pair of parents to be mated, a crossover point is chosen at random from within the genes.

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



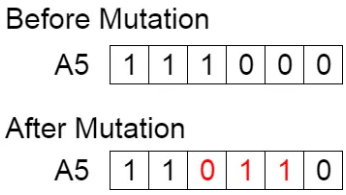
Offspring are created by exchanging the genes of parents among themselves until the crossover point is reached.



The new offspring are added to the population.

Mutation

In certain new offspring formed, some of their genes can be subjected to a mutation with a low random probability. This implies that some of the bits in the bit string can be flipped.



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

5. Evaluations

All the algorithms will be tested on 10 different test cases. There are 5 small test cases and 5 large test cases

Brute force and **Brand and Bouch** algorithms will be evaluated based on the time running and these algorithms will also have a limited time to execute the large test cases while **Local beam search** and **Genetic algorithm** will be evaluated based on the result, meaning that we will evaluate the global extreme of **Local beam search** and **Genetic algorithm** for each test case and each initialization.

Testcases

The first 5 test cases will be the small test cases while the latter will be the larger ones.

Below is the summary table of 10 test cases.

ID	Capacity (W)	Classes (m)	Size (n)	Note
0	292	3	10	Normal case
1	151	3	15	Normal case

2	460	5	20	Take all items
3	628	3	40	Normal case
4	941	5	40	Normal case
5	1194	5	50	Normal case
6	3200	5	100	Take all items
7	3688	5	200	Normal case
8	18000	5	500	Take all items
9	23246	5	1000	Normal case

a) Brute Force

Advantages: The brute force algorithm for solving the knapsack problem has a number of advantages. Firstly, the algorithm is simple and easy to understand, with no specialized knowledge or expertise required for implementation. Secondly, the algorithm guarantees that the optimal solution will be found, as it examines all possible combinations of items. This is especially useful for decision-making, as accuracy and precision are critical. Finally, the brute force algorithm is flexible and can be adapted to different variations of the knapsack problem, such as the bounded or unbounded knapsack problem.

Disadvantages: While the brute force algorithm has several advantages, it also has certain limitations. Firstly, the algorithm is computationally expensive, as it generates and evaluates all possible combinations of items. This can be time-consuming and impractical for large problem instances. Additionally, the number of possible combinations of items grows exponentially with the number of items, making the algorithm inefficient for large problem instances. Finally, the algorithm requires significant memory, as all possible combinations of items must be generated and stored. This can be problematic for large problem instances where memory resources are limited.

Results

	Small data set					Large data
Input num	0	1	2	3	4	**N/A**
Max value	350	287	500	N/A	N/A	**N/A**
Runtime (ms)	1	61	2539	N/A	N/A	**N/A**

Comments

- The Brute Force algorithm is a simple and easy-to-implement method for solving the Knapsack problem. The algorithm evaluates all possible ways of selecting and not selecting items to put in the knapsack, ensuring that the optimal solution is always found. However, the algorithm's susceptibility to an exponential explosion makes it impractical for solving moderately sized and larger Knapsack problems.
- For solving very small problem sizes (**n < 20**), the Brute Force algorithm remains a reasonable option, as it delivers an acceptable run time and guarantees the optimal solution.
    - **Space complexity:** O(n) reflecting the maximum length of the combination.
    - **Time complexity:** O(2^n) as the algorithm will consider all possible combinations of n.

b) Branch and Bound

Advantages: The B&B algorithm has a number of benefits when tackling combinatorial optimization issues. In order to ensure that the best possible solution is obtained within

the given constraints, it first guarantees optimality or provides solutions that are close to optimal. This is especially useful when making decisions because accuracy and precision are so important. Additionally, the B&B algorithm effectively prunes unproductive branches to condense the search space and boost computational speed. It can significantly reduce the time and resources needed to find a solution, especially for large-scale instances of problems like the Knapsack Problem, by intelligently exploring the most promising branches. The branching and bounding strategies of the B&B algorithm can be changed to adapt to different problem domains, which enables customization and optimization based on particular problem characteristics.

Disadvantages: Despite its advantages, the B&B algorithm also has certain limitations. Firstly, the algorithm's efficiency heavily depends on the quality of the bounding function. If the bounding function provides weak estimates of the upper bounds, the algorithm may need to explore more branches, reducing its effectiveness in pruning the search space. Additionally, for highly constrained problems or instances with a large number of potential solutions, the B&B algorithm may still require significant computational resources to reach an optimal or near-optimal solution. The algorithm's performance is inherently limited by the exponential nature of combinatorial optimization problems, as the number of possible solutions grows exponentially with the problem size. Therefore, although the B&B algorithm offers improved efficiency compared to brute-force approaches, it may still face challenges in solving very large-scale instances within practical timeframes.

Results

	Small data set						Large data
Input num	0	1	2	3	4	5	**N/A**
Max value	350	287	500	929	1160	1291	**N/A**

Runtime (ms)	~0	60	~0	128240	60	60	**N/A**

Comments

- The first three examples, where the input sizes are quite tiny, appear to show good performance from the algorithm. However, in the input3, something wrong is that it makes the algorithm recursive so many times, the algorithm's execution time, however, skyrocket noticeably. As a result, it appears that the method performs worse as the size of the input rises, which is to be expected given its exponential time and space complexity.

- **Compared to brute force:** Branch and bound outperform brute force in terms of speed, but both have the same time complexity. As a result, when there are many items, the branch, and bound algorithm should be chosen over the brute force algorithm because it can drastically reduce the search space and speed up processing.

- **Compared to local beam:** The B&B algorithm guarantees optimality but can be computationally expensive, while the Local Beam Search algorithm is a heuristic algorithm that provides good-quality solutions more efficiently but does not guarantee optimality. The choice between the two algorithms depends on the specific requirements of the problem, the size of the problem instance, and the available computational resources.

- **Compared to genetic algorithms:** the B&B algorithm has the advantage of guaranteeing optimality for the knapsack problem and providing an interpretable solution. It is particularly effective for small to medium-sized instances where an exhaustive search is feasible. On the other hand, Genetic Algorithms excel in handling large problem instances, providing good-quality solutions within reasonable time and resources, although without the guarantee of optimality.

- **Space complexity:** There are no more than k best items in the priority queue. Each item has a set that contains a maximum of n unique classes. The complexity of space is therefore **O(k*n).**
- **Time complexity: O(2n)**, where 2 is the branching factor (each item can be selected or not) and n is the number of items. Therefore, as the number of items increases, the execution time of the algorithm will also increase exponentially.

c) Local beam search

Advantages:

- It allows for parallelism by exploring multiple search paths concurrently, leading to faster convergence and efficient search space exploration.
- It also helps overcome the problem of getting stuck in local optima by maintaining multiple beams and exploring alternative paths.
- The flexibility to adjust the beam size dynamically enables thorough exploration or focus on promising solutions.
- Local beam search is memory-efficient as it doesn't require storing a complete search tree.
- It produces good-quality solutions quickly, even in complex search spaces, and is relatively easy to implement.

Disadvantages:

- In general, the Beam Search Algorithm is not complete. Despite these disadvantages, beam search has found success in the practical areas of speech recognition, vision, planning, and machine learning.

- The main disadvantages of a beam search are that the search may not result in an optimal goal and may not even reach a goal at all after being given unlimited time and memory when there is a path from the start node to the goal node.
- The beam search algorithm terminates for two cases: a required goal node is reached, or a goal node is not reached, and there are no nodes left to be explored.
- A more accurate heuristic function and a larger beam width can improve Beam Search's chances of finding the goal.

Results

Beamwidth	K = 5	K = 10	K = 20
input0	350	350	350
input1	287	287	287
input2	500	500	500
input3	906	833	888
input4	1160	1160	1157
input5	1291	1291	1291
input6	4900	4900	4900
input7	4695	4617	4697
input8	**TLE**	**TLE**	**TLE**
input9	**TLE**	**TLE**	**TLE**

Comments

- The initial states are generated randomly. In the small test cases, most of them return a global extreme value. But there are some exceptions which are input3 and input4, the results of “K=5” in both inputs are better than the value for “K=10” and “K=20”. It means that the larger K does not mean a better result

- In large test cases, we can also see a similar pattern. The results for input5 and input6 are the global extreme values. In input7, a similar pattern in input3 and input4 can be seen again. In this input, the result for "K=10" is worse than it for "K=5", and the result for "K=20" is the best.

- We can also see that input8 and input9 are the larger test cases with about 1000 items. So with the Local beam search, with will consume a big amount of time to return a result.

- In conclusion, the Local beam search algorithm has to rely heavily on the initial states. If the initial states are good enough, it will return a better result. Moreover, the beamwidth does not affect much on the result.

- **Space complexity: O(b*d*n)**, b is the beamwidth, d is the beamdepth and n is the number of items

- **Time complexity: $O(b*d*n^2)$**, b is the beamwidth, d is the beamdepth and n is the number of items.

d) Genetic algorithm

Advantages:

- **Exploration of Search Space** – Genetic algorithms are designed to explore a wide range of potential solutions to a problem. They use a process of "evolution" to generate and evaluate a large number of candidate solutions, allowing them to search a large portion of the problem space.

- **Flexibility** – Genetic algorithms can be applied to a wide range of problems, from scheduling to data mining to machine learning. They are versatile tools that can be used in many different contexts.

- **Adaptability** – Genetic algorithms are able to adapt to changes in the problem or the environment. They can evolve and improve their solutions over time, even when the problem is complex or dynamic.
- **Parallel Processing** – Genetic algorithms can be run in parallel on multiple processors or nodes, allowing them to process large amounts of data more quickly and efficiently.
- **Global Optimization** – Genetic algorithms are capable of finding globally optimal solutions to a problem, rather than just local optima. This makes them a powerful tool for optimization problems where finding the best solution is critical.

Disadvantages:

- **Computational Complexity** – Genetic algorithms require significant computational resources to run, particularly when dealing with large datasets or complex problems. This can make them slow and computationally expensive.
- **Difficulty in Tuning Parameters** – Genetic algorithms rely on several parameters, such as population size, mutation rate, and crossover rate, which can be difficult to tune to the specific problem at hand. Setting these parameters incorrectly can lead to poor performance.
- **Dependence on Randomness** – Genetic algorithms use a random process to generate and evaluate candidate solutions. This can lead to unpredictable results and make it difficult to compare the performance of different algorithms.
- **Risk of Premature Convergence** – Genetic algorithms can sometimes converge too quickly to a suboptimal solution, particularly when the population size is small or the mutation rate is too low.

- **Limited Understanding of Results** – Genetic algorithms can produce results that are difficult to interpret or understand. This can make it difficult to determine whether the algorithm has found the optimal solution or not.

Results

	maxGen = 50	maxGen = 100	maxGen = 200
input0	350	350	350
input1	287	287	287
input2	500	500	500
input3	929	929	929
input4	1158	1160	1160
input5	1290	1291	1291
input6	4900	4900	4900
input7	4574	4719	4676
input8	17544	19436	21199
input9	19691	21477	23216

***Note:** **maxGen** is the predefined number of generations to be executed

Comments

- The algorithm can calculate the optimal value for most test cases although there are some differences between maxGen values.
- In the small test cases, the fittest value is equal to the global extreme because of its reasonable initial population when the size of the population is small.
- In the large test cases, nevertheless, the difference between maxGen numbers has a significant impact on the fittest value. We can see clearly that from inputs 7 to 9, a

different maxGen will return a different result. This is because the population size of these inputs is larger than the others.

- Although there are some differences in some cases, the fittest value is still acceptable because we use Crossover and Mutation with random choice. Therefore, the results are still optimal values.
- In conclusion, the Genetic algorithm heavily relies on the size of the problem and the maxGen values. If these values are too large, it will consume a big amount of time to solve the problem. But in the opposite direction, if these values are too small, the results are not optimal enough.
- **Space complexity: O(p*n)**, p is the population size, n is the size of each individual
- **Time complexity: O(g*p*f)**, g is the maxGen values, p is the population size, f is the time complexity of fitness evaluation function (In this solution, f is equal to the number of individuals in the test case).

6. Conclusions

In summary, the knapsack problem can be solved by different algorithms: Brute force, Branch and Bound, Local beam search, and Genetic algorithm. Each algorithm has its own cons and pros. Therefore, depending on the case, each algorithm can show its advantages and be considered the best based on the problem size, time, and constraint.

7. Videos demo the implementations

Artificial Intelligence - Project1 - Searching - YouTube

V. REFERENCES

[1] Define Beam Search - Javatpoint

[2] Beam Search Algorithm in Artificial Intelligence

[3] Python Beam Search Algorithm – Be on the Right Side of Change

[4] Genetic Algorithm in Machine Learning - Javatpoint

[5] Knapsack-Artificial-Intelligent: Knapsack problem solving with various algorithm for study in AI

[6] Foundations of NLP Explained Visually: Beam Search, How it Works

[7] What is branch and bound techniques?

[8] 0/1 Knapsack using Branch and Bound with example

[9] 0/1 Knapsack Problem to print all possible solutions

[10] Brute Force Approach

[11] How to solve Knapsack Problem using Exhaustive Search | Brute Force

[12] Branch and bound, how to and implement

[13] Strip in Map Function

[14] Read and Return Text File