CS420 - Project 01

Searching for the Knapsack

20125074 - Dương Hoàng Huy

20125123 - Hoàng Thanh Tùng

20125051 - Trần Nguyễn Đăng Tâm

20125107 - Hồ Văn Quân

Table of contents

Assignment Plan

Member	Tasks	Completion level
20125051 - Trần Nguyễn Đăng Tâm	- Research for Brute force searching	95%
	- Do experiment & evaluation for Brute force searching	95%
20125123 - Hoàng Thanh Tùng	- Research for Branch and Bound	95%
	- Do experiment & evaluation for Branch and Bound	95%
	- Create video for experiments	100%
20125107 - Hồ Văn Quân	- Local Beam Search	95%
	- Do experiment & evaluation for Local Beam Search	95%
20125074 - Dương Hoàng Huy	- Research for Genetic algorithm	95%
	- Do experiment & evaluation for Genetic Algorithm	95%
	- Generate both small & large test cases	100%

Algorithm 1: Brute Force Searching

Algorithms' description

A highly common approach to issue solving, known as "brute-force search," entails systematically listing all potential candidates for the answer and determining if each one matches the problem's statement instead of using advanced techniques to do the task more efficiently. Therefore, it could be rather inefficient, take a long time to run and the costs are proportional to the number of candidate solutions.

```
forS i = 1 to 2^n do
      j ← n
      tempWeight ← 0
      tempValue ←0

      #generate another bit string A to try
      while (A[j]!= 0 and j > 0)
            A[j]←0
            j←j-1
      A[j]←1

      #check at least one item from each class
      #with totalC is sum of different classes in bit string A
      if (len(totalC != m) then continue

      #find out which is the best choice of A
      for k←1 to n do
            if (A[k] = 1 ) then
                  tempWeight ← tempWeight + Weights [k]
                  tempValue ← tempValue + Values [k]
      if ((tempValue > bestValue) AND (tempWeight ≤ Capacity)) then
            bestValue ← tempValue
            bestWeight ← tempWeight
      bestChoice ← A
return bestChoice
```

Explanation

Brute Force Algorithms are a simple approach to solving a problem that depends on sheer computing power and trying every possibility rather than using cutting-edge methods to increase efficiency. For the knapsack problem, if there are n items then we have 2^n possibilities. The solution to the problem would be bit string 0's and 1's where 0's is the unselected item and 1's is the selected item, and the total value is as large as possible.

Complexity

With n is the number of items:

- **Space Complexity O(2^n)**: We have to keep all instances of the solution for future validating
- **Time Complexity O(n*2^n):** Scan through all instances and validating the constraints of each instance

Algorithm 2: Branch and Bound

Algorithms' description

Branch and bound is only used to solve optimization problems. Branch and bound is superior to exhaustive search because, unlike exhaustive search, it constructs candidate solutions one component at a time and evaluates the partially constructed solutions. If none of the remaining component's potential values can lead to a solution, the remaining components are not generated at all.

Explanation

This approach allows for the solution of some large instances of difficult combinatorial problems, but it still has an exponential complexity in the worst case:

```
// Input:Array Weights contains the values of all items
Array Values contains the values of all items
// Output:   An array that contains the best solution and its MaxValue
// Precondition: The items are sorted according to their value-per-weight
ratios

Queue<nodeType> PQ
Node  Type current, temp

Initialize the root
PQ.enqueue(the root)
MaxValue = value(root)

while(PQ is not empty)
    current = PQ.GetMax()
    if (current.nodeBound > MaxProfit)
       Set the left child of the current node to include the next item

    if (the left child has value greater than MaxProfit
        and meet the class constraint) then
       MaxValue = value (left child)

    Update Best Solution

    if (left child bound better than MaxProfit)
        PQ.enqueue(left child)


    Set the right child of the current node not to include the next item
```

```
        if (right child bound better than MaxProfit)
           PQ.enqueue(right child)
    return the best solution and it's maximum valuation.
```

1. **get_bound() : First, let me explain the get_bound(node) first**

To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using the Greedy approach. If we know a bound on the best possible solution subtree rooted with every node. If the best in subtree is worse than the current best, we can simply ignore this node and its subtrees. So we compute the bound (best solution) for every node and compare the bound with the current best solution before exploring the node.

2. **implement get_bound() function s**

We compute the upper bound on the value of any subset by adding the cumulative value of the items already selected in the subset, v, and the product of the remaining capacity of the knapsack (Capacity minus the cumulative weight of the items already selected in the subset, w,) and the best per unit payoff among the remaining items, which is v[i+1] / w[i+1]

$$Upper\ Bound = v + (Capacity - w)*(vi+1 / wi+1)$$

3. **The constraint on classes : we will check class in the section to choose the best node**

When the algorithm scans from beginning to end, we cannot generate max profit because it can accidentally prune branches that have the best results and satisfy class constraints. Therefore, before we find a node that meets the requirements of having enough classes, we will not update the max profit.

Complexity

Even though this method is more efficient than the other solutions to this problem, its worst case time complexity is still given by O(2^n), in cases where the entire tree has to be explored. However, in its best case, only one path through the tree will have to be explored, and hence its best case time complexity is given by O(n). Since this method requires the creation of the state space tree, its space complexity will also be exponential.

Algorithm 3: Local Beam Search

Introduction

In computer science, local beam search is a heuristic search algorithm that explores a graph by expanding the best node in a limited set. Beam Search uses breadth-first search to build its search tree. It generates all the successors of the current level's state at each level of the tree, but only evaluates a B number of states. It expands the B number of the best node, level by level and moves downwards from those best B nodes at the level, periodically.

Pseudo code

```
Start
Take the inputs
Set up initial solution
Put initial solution to expanded list
While not reach the limit iteration:
     Nbhood is empty Then add all neighbors of nodes in expanded list to
nbhood
     Check if each neighbor's weight is valid
     If not valid, remove from nbhood
     Sort nbhood by values of nodes in descending order
     Choose best B neighbors
     If these best neighbors have smaller values than nodes in expanded
list or do not contain sufficient classes, stop searching and return the
expanded list
     Else set the expanded list as the list of these best B neighbors, and
continue the loop
End.
```

Explanation

Expanded: list of solutions waiting for being expanded to its neighborhood

Nbhood: nbhood is a set of all neighbors of nodes in expanded list

Check if each neighbor's weight is valid means it's not greater than maxWeight

```
Sort the nbhood descendent so that best B values is first B nodes in nbhood
```

The algorithm begins selecting the initial generated state. Then, for each level of the search tree, it always considers B new states among all the possible successors of the current ones until it reaches a goal.

Local beam search requires a good initial setting algorithm to perform better results, and it only finds local maxima, not complete and not optimal solution, yet still useful sub-optimal solutions quickly, then goes back and keeps looking for better solutions until it converges to an ideal solution. Its linear memory consumption allows Beam Search to probe very deeply into large search spaces and potentially find solutions that other algorithms cannot reach.

So that, the **drawbacks** of Local Beam Search:

- It might not provide an optimal goal or might even fail to obtain a goal. Local beam search often ends up on local maxima
- Beam Search's challenge of locating the target can be enhanced by a more precise heuristic function and a wider beam (larger beam width).

Complexity

Time Complexity

Depend on

- The heuristic function’s accuracy
- The worst-case scenario is that the heuristic function directs Beam Search to the deepest level of the search tree.
- The worst-case time = ***O(B*m)***

Space Complexity

Depend on

- Reduce memory consumption because it only maintains B nodes at each level
- The worst-case space complexity = ***O(B*m)***
- Beam Search can delve very deeply into broad search fields due to its linear memory consumption, potentially uncovering solutions that other algorithms are unable to.

*Note: ***B*** is the beam width, and ***m*** is the maximum depth of any path in the search tree.

Algorithm 4: Genetic Algorithms

Algorithms' description

A genetic algorithm (GA) is a search technique used in computing to find good solutions to optimization problems. However, GA does not guarantee an optimal solution.

GAs are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as Mutation, Selection, Crossover and Reproduction. The fittest individuals in an environment survive and pass their genotype to the future generations while the weak characteristics disappear long the journey.

Explanation

Here are all global variables' descriptions for reusability. We will not explain again because of the duplication in each function:

```
individual # any potential solution which is a list of bits (0 or 1).
# Each bit at a particular index represents the decision to pick the item
at that index or not.

# GIVEN VARIABLE
W #the capacity (maximum weight) of the knapsack
num_classes #the number of existed classes, start from 1
weights #list of weights with weights[i] is the weight of item ith
values #list of values with values[i] is the value of item ith
classes #list of classes with classes[i] is the class of item ith

# SUPPORT VARIABLE
population = [] # list of individuals
fitness = [0 for i in range(POPULATION_SIZE)], list of fitness coefficient
with fitness[i] is the fitness coefficient of item ith

# CONSTANT
POPULATION_SIZE = 6 # the size of population
INDIVIDUAL_SIZE = len(weights) # the number of item in our problem
CROSSOVER_RATE = 0.85 # the probability of a crossover between two parents
MUTATION_RATE = 0.02 # the probability of a mutation on each bit of
individual
REPRODUCTION_RATE = 0.3 # the probability of a reproduction which passes
```

```
down the parents to the next generation without crossover & mutation
```

Here the the **basic steps** of the algorithm:

1. Start: Generate the initial population with constraints.
2. Calculate the fitness coefficient of each individual in population
3. Select parents for generating the next generation. After that, we have two options:
    a. Reproduction
    b. Crossover & Mutation
4. Repeat those steps at MAX_STEPS.
5. Get the fittest individual of the latest generation.

Generate initial population

We generate a random set of Individuals that would form our initial *population*. We decided to choose six individuals for our population throughout this report. In addition, our problem has an important constraint that we must choose at least an item from each class. `is_at_least_1_item_each_class` will check that constraint for any new generated individual:

```
function is_at_least_1_item_each_class(individual)
    inputs individual, the solution which is validated
    returns boolean, does the individual pass the constraint or not
    lst_classes <- list of appearances of each class (0 or 1)

    for index=0 to INDIVIDUAL_SIZE do
        if individual[index] is 1 # picked
        then lst_classes[class at that item - 1] <- 1

    for appearance in lst_classes do
        if appearance is 0 # which mean it does not appear in this solution
        then return False

    return True
```

Generate the first population:

```
function generate_initial_population()
    while length of population not equal POPULATION_SIZE
              individual <- list of random number 0 to 1 with length equal
INDIVIDUAL_SIZE
```

```
        if individual not in population and is_at_least_1_item_each_class(individual)
                add individual to population

    return population
```

Calculate the fitness coefficient

Fitness coefficient is a non-negative score that shows how fit the individual is for the problem. For our Knapsack Problem, we define the fitness coefficient as the total value of all items picked in an individual with total weight less than or equal to the capacity of the knapsack. For any individual has total weight which exceeds the capacity of the knapsack, the fitness coefficient is 0 (the least potential solution):

```
function cal_fitness()
    for i=0 to POPULATION_SIZE-1 do
        sum_of_weight <- 0
        sum_of_value <- 0
        for j=0 to INDIVIDUAL_SIZE-1 do
            sum_of_weight <- sum_of_weight + population[i][j] * weights[j]
            sum_of_value <- sum_of_value + population[i][j] * values[j]
        if sum_of_weight is greater than W then fitness[i] <- 0
        else fitness[i] <- sum_of_value
```

Selection

In the selection stage, we randomly select a subset of individuals based on their fitness coefficient from the population to generate the next generation. There are many different strategies to select the individuals and Tournament Selection is a good strategy among those. This strategy will randomly select two individuals from the population, the one having a higher fitness score is chosen as a parent for the next stage. In our problem, we randomly select four individuals (for the purpose of not duplicating parents) and run two tournaments for selecting two fittest individuals for parents:

```
function selection()
    returns parents, the list of 2 parents after the tournament selection
    selected_index <- 4 random indexes from population
    parents <- empty list
```

```
    if fitness at selected_index[0] > fitness at selected_index[1]
    then add population at selected_index[0] to parents
    else add population at selected_index[1] to parents

    repeat the above process for selected_index[2] and selected_index[3]

    return parents
```

By using the Tournament selection, the range of population is more diverse and the convergence to the final solution will be slower, which can help to generate a better solution.

Crossover

In the crossover stage, a part of each parent in the chosen parents is potentially exchanged to generate the child at a predefined probability which is called Crossover rate. This rate is relatively in the range of ... , for simulating the crossover in nature which does not always happen.

Again, there are many strategies for crossover. In our problem, we simply exchanged the half part of each parent:

```
function crossover(parents)
    inputs parents, list of 2 parents from selection stage
    returns children, list of 2 children after crossover

    children <- empty list
    if not hit CROSSOVER_RATE then return parents

    first_child <- merge half first parents[0] with half second parent[1]
    second_child <- merge half first parents[1] with half second parent[0]
    add first_child, second_child to children

    return children
```

Mutation

In the mutation stage, every single bit in each generated child has a small probability of mutation which is called Mutation Rate. It means that bit can be flipped, from 0 to 1 and 1 to 0:

```
function mutation(children)
    inputs children, list of 2 children after the crossover
    return children, list of 2 children after the mutation

    for each child in children
        for each bit of the child
            if hit the MUTATION_RATE then flip that bit

    return children
```

Generate the next generation & Reproduction

In reproduction, we also have a Reproduction Rate and it gives parents two options: if the program hits that rate, two parents will be passed down to the next generation. Otherwise, they will go through Crossover and Mutation:

```
function generate_next_population()
      returns next_population, the next generation based on the current
population

    next_population <- empty list
    while size of next_population is not equal the POPULATION_SIZE do
        parents <- selection()
        children <- empty list

        if hit the REPRODUCTION_RATE then children <- parents
        else
            children <- crossover(parents)
            children <- mutation(children)
            for each child in children do
                if not is_at_least_1_item_each_class(child) then continue

        add children to next_population

    return next_population
```

Genetic Algorithm

We need to get the fittest individual from the current population by a simple implementation scanning through the the list fitness to get the index of the highest fitness coefficient:

```
function get_fittest_individual()
    returns fitness[fittest_index], the highest fitness coefficient
                population[fittest_index], the fittest individual in the
population

    fittest_index <- 0
    for i=0 to POPULATION_SIZE-1 do
        if fitness[i] > fitness[fittest_index] then fittest_index <- i
    return fitness[fittest_index] and population[fittest_index]
```

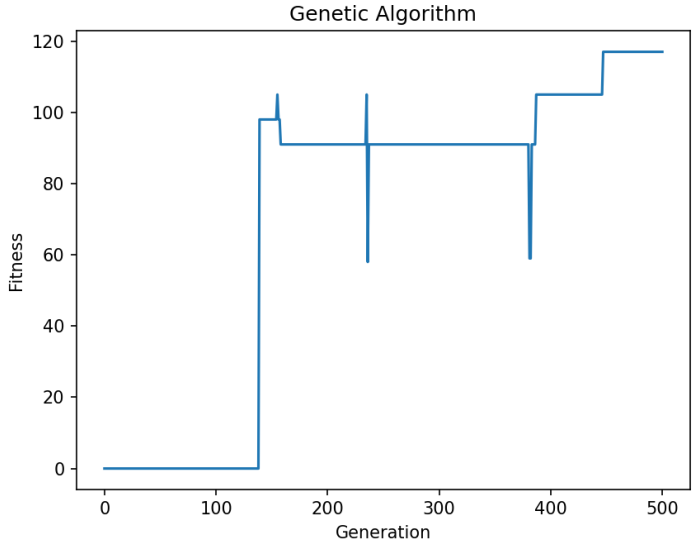
By combining all the above processes with the termination condition, we have implemented the Genetic Algorithm. Because we cannot generate the next population infinitely, we have to limit to 500 iterations and return the fittest individual:

```
function genetic_algorithm_knapsack_problem()
    returns fitnest_individual, the fittest individual (good solution)

    population <- generate_initial_population()
    cal_fitness()

    for i in range(TIME):
        population = generate_next_population()
        cal_fitness()

    fitnest_individual = get_fitnest_individual()
    return fitnest_individual
```



This graph shows the fittest individuals at each iteration with the sample test case of the project. As we see, the generation is getting better and better.

Complexity

Time complexity

The algorithm depends on the size of population (POPULATION_SIZE), the size of individuals (INDIVIDUAL_SIZE) in the population and the number of iterations(MAX_STEPS) to get the next generation. The dominant big O is in the genetic algorithm function in which iteration of MAX_STEPS contains an iteration of population and a loop through individuals inside it. So the time complexity is O(POPULATION_SIZE * INDIVIDUAL_SIZE * MAX_STEPS).

Space complexity

The algorithm creates an array of population, fitness and reuses them. The space complexity is O(POPULATION_SIZE + INDIVIDUAL_SIZE)

Experiment & Evaluation

In the experiment section, we generate 5 small datasets and 5 large datasets increasing in size by random sampling of weights, values and classes with suitable capacity of knapsack.

Testing 1: Small datasets with capacity = 100, number of classes = 2

For the first testing, our purpose is to test the level of optimality of solution for each algorithm. For the performance evaluation, we will not discuss in depth in this section because the cost of each algorithm in small datasets is not significant and discussed later in testing 2.

Brute force gives us the optimal solution, so we use it to evaluate the other algorithm's solutions. For two local searches (local beam search and genetic algorithm), we will generate 10 solutions and get the best of them for finding the best sub-optimal solution. Sometimes, we also get the optimal solution(it is quite frequent in small datasets):

input/INPUT_0.txt with 10 items

Algorithm	Total weight	Total value	Picked items	Processing time(sec)	Memory consumed(Kb)
Brute force	97	**226**	1, 2, 4, 8	0.0061614	155 648
Branch and bound	97	226	1, 2, 4, 8	0.00246559	Too small to track
Local beam search	97	226	1, 2, 4, 8	0.0039	94208
Genetic algorithm	97	226	1, 2, 4, 8	0.0500123	Too small to track

input/INPUT_1.txt with 15 items

Algorithm	Total weight	Total value	Picked items	Processing time(sec)	Memory consumed(Kb)
Brute force	95	**367**	2, 3, 5, 8, 12	0.13278069	258 048

Branch and bound	95	367	2, 3, 5, 8, 12	0.0036374	Too small to track
Local beam search	95	367	2, 3, 5, 8, 12	0.00374829	114 688
Genetic algorithm	95	367	2, 3, 5, 8, 12	0.0614544	Too small to track

input/INPUT_2.txt with 20 items

- Local beam search in this test case has much local maxima, it is hard to find the optimal solution so we decide to set the **beam width = 10** to find the optimal solution:

Algorithm	Total weight	Total value	Picked items	Processing time(sec)	Memory consumed(Kb)
Brute force	98	**523**	1, 2, 6, 8, 10, 11, 12	4.7928263	1 597 440
Branch and bound	98	523	1, 2, 6, 8, 10, 11, 12	0.0061949	Too small to track
Local beam search	98	523	1, 2, 6, 8, 10, 11, 12	0.0166164	98 304
Genetic algorithm	98	523	1, 2, 6, 8, 10, 11, 12	0.07211349	Too small to track

input/INPUT_3.txt with 25 items

- Same as the reason of the above test case, we decide to set the **beam width = 20** to find the optimal solution:

Algorithm	Total weight	Total value	Picked items	Processing time(sec)	Memory consumed(Kb)
Brute force	100	640	0, 1, 3, 7, 10, 15, 16, 20, 24	343.0675448	10 162 176

Branch and bound	100	640	0, 1, 3, 7, 10, 15, 16, 20, 24	0.0346225	Too small to track
Local beam search	100	640	0, 1, 3, 7, 10, 15, 16, 20, 24	0.09976	229 376
Genetic algorithm	100	640	0, 1, 3, 7, 10, 15, 16, 20, 24	0.07188	Too small to track

Brute force does not give reasonable time processing for more than 25 items. From here, we use Branch & Bound which is almost closer to brute force to evaluate the others.

input/INPUT_4.txt with 30 items

Algorithm	Total weight	Total value	Picked items	Processing time(sec)	Memory consumed(Kb)
Branch and bound	100	594	3, 6, 9, 10, 12, 16, 17	0.0439908	Too small to track
Local beam search	100	594	3, 6, 9, 10, 12, 16, 17	0.0578853	229 376
Genetic algorithm	100	594	3, 6, 9, 10, 12, 16, 17	0.0897522	Too small to track

Evaluation for Testing 1

In general, almost all of these algorithms can give the optimal solutions with reasonable time processing and memory usage, except the brute force algorithm.

Brute force

Brute force does not give reasonable **time processing** along with large **memory consumption** for more than 25 items. Even with a small test size, the overall run time was rather long. As a result, this naive approach is the least effective algorithm to solve this knapsack problem.

Branch and bound

Branch & bound does a good job in small datasets which gives us an optimal solution by running only once. Moreover, it has the highest performance in both time processing and memory usage. This algorithm actually makes use of the good heuristic function to get a good result.

Local beam search

In small datasets, by using appropriate ***beam width*** settings, we can get better results with less time because the convergence to the local maxima is faster. However, there is a trade-off between time and memory for controlling more branches. This algorithm has the highest memory usage in experiment (except the brute force).

Genetic algorithm

In small datasets, with the ***setting***: MAX_STEPS = 200, the genetic algorithm gives us optimal solutions with ideal time processing and memory consumption (linear space complexity) by rerun ~ 10 times. These works are still reasonable in small datasets because our purpose is finding the optimal solution and the cost is so cheap.

Testing 2: Large datasets with capacity = 500 with increasing the number of classes

Our purpose in this testing 2 is evaluating the **performance (time, space)** of each algorithm. Branch & Bound is almost closer to brute force so we use its solution to evaluate the others (Because brute force cannot give reasonable time processing for over 25 items, we do not experiment it in this testing).

Only Genetic Algorithm give a small time processing so we will test multiple time and give the range of solutions:

large_input/INPUT_0.txt : 200 items, 5 classes, 4000 capacity

Algorithm	Total weight	Total value	Processing time(sec)	Memory consumed(Kb)
Branch and bound	3999	146 602	26.8962351	348 160
Local	3999	132 246	10.1569948	237 568

beam search				
Genetic algorithm	~ 4000	Between 100 000 ~ 120 000	< 1	< 70 000

large_input/INPUT_1.txt : 300 items, 6 classes, 5500 capacity

Algorithm	Total weight	Total value	Processing time(sec)	Memory consumed(Kb)
Branch and bound	5499	279 857	144.1399765999998	143 360
Local beam search	5500	222 126	31.7021931	1 048 576
Genetic algorithm	~ 5500	180 000 ~ 210 000	< 1.5	< 70 000

large_input/INPUT_2.txt : 400 items, 7 classes, 8000 capacity

Algorithm	Total weight	Total value	Processing time(sec)	Memory consumed(Kb)
Branch and bound	8000	465 363	297.9842375	614 400
Local beam search	7999	389 041	81.6250135	507 904
Genetic algorithm	~ 8000	300 000 ~ 330 000	< 1.5	< 70 000

large_input/INPUT_3.txt : 500 items, 8 classes, 12000 capacity

Algorithm	Total weight	Total value	Processing time(sec)	Memory consumed(Kb)

Branch and bound	11 999	700 875	1 845.8147978 ~ 30 min	15 708 160
Local beam search	12 000	631 236	186.5044367	2 322 432
Genetic algorithm	~12 000	470 000 ~ 500 000	< 2	< 100 000

large_input/INPUT_4.txt : 600 items, 9 classes, 14000 capacity

Algorithm	Total weight	Total value	Processing time(sec)	Memory consumed(Kb)
Branch and bound	NA	NA	NA	NA
Local beam search	14 000	797 249	287.923268	2 527 232
Genetic algorithm	~ 14 000	600 000 ~ 650 000	< 2	< 150 000

Evaluation for testing 2

Time processing

In large datasets, the complexity of Branch and Bound grows exponentially which gives the highest time processing in input 3. Despite the good heuristic function, branch and bound actually exhaustedly scan to get the optimal solution.

Difference from the branch & bound with slower growing in complexity, by changing the setting, we can reduce the exhaustion in Local Beam Search to get the sub-optimal solution. Therefore, the time is much lower than Branch and Bound and this also means a trade-off for accuracy.

In Genetic Algorithm, the complexity does not grow much faster than the two above (population size and individual size) so the time processing is the smallest.

Memory consumption

In the experiment, the memory usage of Branch & Bound grows much faster than of Local Beam Search. However, the Genetic Algorithm's memory usage increases slightly (almost unchanged).

Optimality

It is clearly that we can infer from the experiment the rank for the optimality of solution:

Branch & Bound > Local Beam Search > Genetic Algorithm

This is because in Local Beam & Genetic Algorithm, there are many factors that affect the accuracy of the solution such as limited iterations, beam width, population size. In Genetic Algorithm, there are many constant rates for mutation, reproduction, ... and strategies for selections, ...

Conclusions

It is clear that Brute force is not suitable for solving this problem because of its performance. For some average size of samples, Branch and Bound along with Local Beam Search give a good performance with the highest accuracy. However, the best approximation approaches for this Knapsack Problem Genetic Algorithms which has a good performance and the optimality of solution.

References

Brute force

http://www.micsymposium.org/mics_2005/papers/paper102.pdf

Branch and bound

https://iq.opengenus.org/0-1-knapsack-using-branch-and-bound/

Local beam search

https://www.geeksforgeeks.org/introduction-to-beam-search-algorithm/

https://static.aminer.org/pdf/PDF/000/305/085/penalty_functions_and_the_knapsack_problem.pdf

https://www.javatpoint.com/define-beam-search

Genetic algorithm

Selection (genetic algorithm) - Wikipedia

Genetic Algorithm to solve the Knapsack Problem (arpitbhayani.me)

Genetic Algorithms (GAs) (cmu.edu)

Genetic algorithms: Biologically inspired, fast-converging optimization - Manning