# COMPARISON OF HEURISTIC ALGORITHMS FOR KP01

A comparison of heuristic algorithms for solving the 0-1 Knapsack Problem

Raunak Redkar

Supervisors: Per-Olof Freerks and Felicia Dinnétz

Kungsholmens Gymnasium

# Contents



# List of Algorithms

# 1. Introduction

## 1.1 Background

There are many situations in every day life, where people wonder whether they are doing something efficiently. Unforunately, the human brain is not always capable of coming up with an optimal approach when the problem has includes several factors to be accounted for. It is here when an external system is used, and the true power of computing can be recognised.

To solve such problems, a computer is provided all the information, from which it will produce an optimal answer. For the system to process all the information, certain instructions must be written into the system so that it knows how to handle the information. This set of instructions is called an algorithm. The system or computer uses algorithms to take in information (the input), and produce an answer - the output. The efficiency (in all aspects) of the algorithm is dependant on the steps that make it up. Most problems have several algorithms which can solve them. For problems in computer science, more than one algorithm is usually proposed and used. This is usually the case for optimization problems.

In the fields of computer science and mathematics, optimization problems are problems of finding the optimal solution, from a range of many feasible solutions. They are usually categorized into 2 categories: discrete optimizations and continuous optimizations, depending on whether the variables are discrete or continuous respectively. Combinatorial optimization problems are a subset of optimization problems that fall into the discrete. Combinatorial optimization involves searching for a maxima or minima for an objective function whose search space domain is discrete but usually large.

Typical combinatorial optimization problems include:

- General Knapsack Problem - Given a set of items, each with weight and profit value and a knapsack capacity, what is the best way to choose the items while respecting the knapsack capacity?
- Traveling Salesman Problem- Given a list of cities, what is the shortest possible path that visits each city exactly once and returns to the origin?
- Set Cover - Given a set of elements $\{1, 2, ..., n\}$ called the universe, and a collection of $m$ sets whose union equals the universe, what is the smallest sub-collection of those m sets whose union is the universe?

Combinatorial optimization problems show up in an array of different fields. The Knapsack Problem in particular has many variants which include the 0-1 knapsack problem, the bounded and unbounded knapsack problems, the multidimensional knapsack problem, the discounted knapsack problem, etc. The 0-1 Knapsack Problem is the simplest form of the knapsack problem and thus has also been in the main focus within the research community. It appears in real-world decision-making processes in a variety of fields. Some examples include: financial modelling and investments [1], cryptography, and resource allocation in networking [2] (all considered high-scale instances of the knapsack problem; large amount of data to process).

Exact deterministic algorithms are those which will always produce the same optimal result every time. Many of combinatorial optimization problems including the 0-1 Knapsack problem currently do not have exact deterministic algorithms which are considered fast enough for them to be used in high-scale data situations. Consequently, the research focus has been on algorithms that do not necessarily guarantee the best solution but win over deterministic algorithms when it comes to time.

#### Problem statement

In the 0-1 Knapsack Problem, there are $n$ items and a maximum weight capacity $W$. Each item has a profit value $p_i$ and a weight value $w_i$. One must then find the optimal selection of items which maximizes the profit value while respecting the max weight value. The problem can be mathematically represented as such:

$$\text{Maximize } f(\vec{x}) = \sum_{i=1}^{n} p_i x_i \tag{1}$$

$$\text{subject to} \sum_{i=1}^{n} w_i x_i <= W \tag{2}$$

$$x \in \{0, 1\} \tag{3}$$

The rest of this paper will use the abbreviation "KP01" for the 0-1 Knapsack Problem.

## 1.2 Aim

The aim of this paper is to compare algorithms in order to investigate the strength of commonly used techniques in for solving optimization problems.

## 1.3 Research Question

How do the Discrete Global-Best Harmony Search Algorithm and the Binary Harmonic Multi-Scale Algorithm perform when implemented for solving KP01?

# 2. Theory

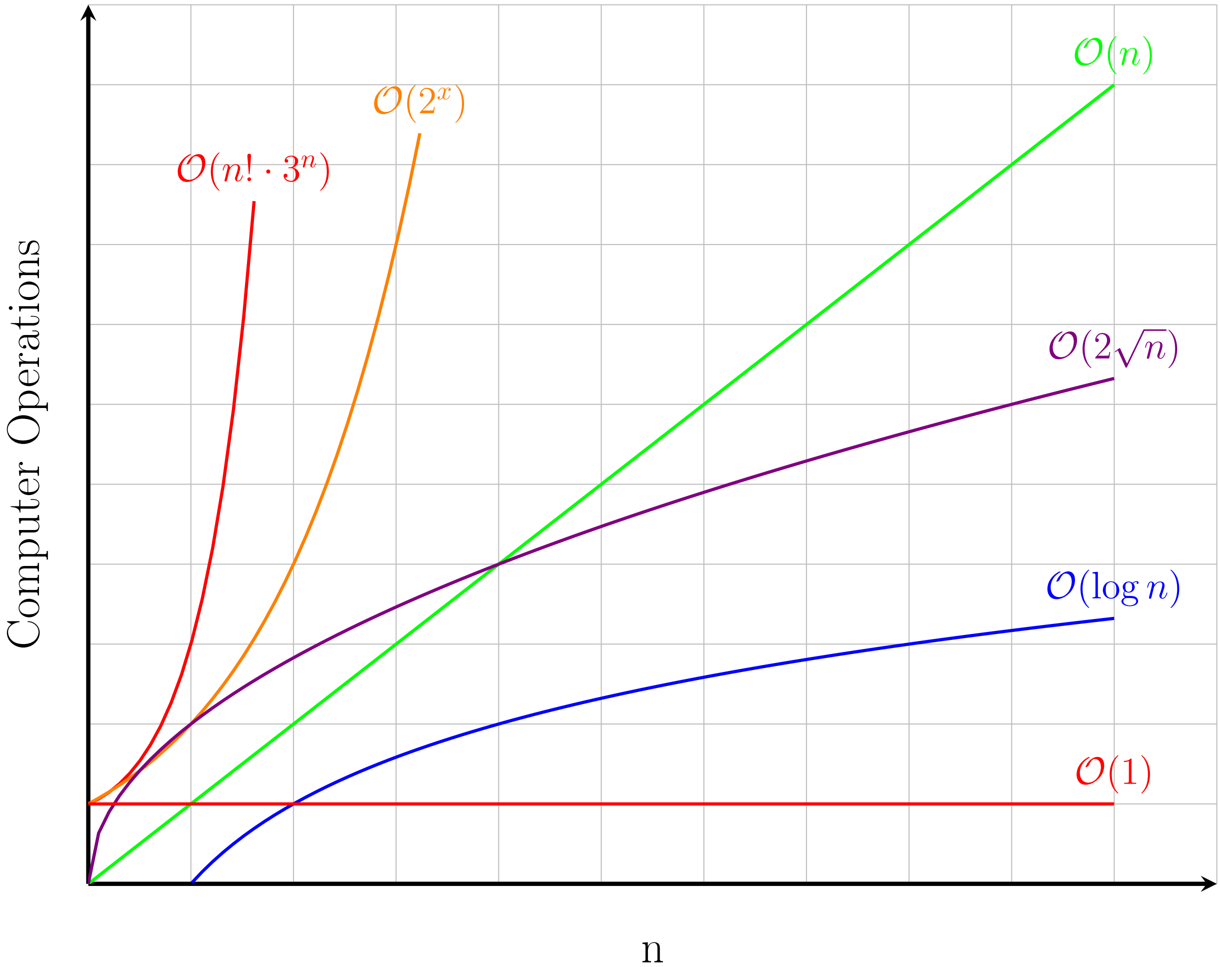
## 2.1 Computational Complexity

In computer science, computational complexity is the measure of how expensive it is to the run an algorithm; the amount of resources required to run the algorithm. The 2 resources are time and memory usage by a computer. As there are many low-level computer operations which happen, it is hard to exactly know how much time a program will take to run, however since these low-level operations always occur, the time taken for a program to run is usually some constant times some function of the size of the input - the time complexity. For this reason, the time and memory are usually given as time and space complexities which are given in terms of the size of the input. As the memory complexity is dependent on the time complexity, the time complexity is the limiting factor for the efficiency of an algorithm and is usually the one focused on. Computable problems' time complexities can be factorial, exponential, polynomial, logarithmic, etc.

### Big O notation

The time complexity of an algorithm is a function of the size of the input of that algorithm. For example, if a set of $n$ numbers $a_1, a_2, a_3, ..., a_n$ are given, and an algorithm checks the existence of an certain number in that set, by checking all n elements of the set for equality, then the time complexity is some constant times $n$ - the size of the input. Computer scientists would denote this using Big O notation as $\mathcal{O}(n)$. Big O notation is a system developed by mathematicians and computer scientists to describe an function/algorithm's asymptotic limiting factor. [3]

Computer scientists usually categories the time complexity of an algorithm into polynomial time complexities and non polynomial time complexities. This is because non-polynomial functions like factorial and exponential functions tend to grow quicker than polynomial functions and thus are usually considered nonviable for large data. One can see thus in the graph below before that the *non-polynomial* functions like $f(x) = x! \cdot 3^x$ or $f(x) = 2^x$ grow much faster than the other *polynomial* or *subpolynomial* functions.




Complexity Classes

All problems in computer science are classified into a complexity class, which describes how difficult it is to find a solution for the problem on demand [4]. For example the class $P$ is is the set of problems which can be solved in polynomial time. The problems of the $NP$ class are those where a proposed solution to an instance of the problem can be verified in polynomial time, but a way of algorithmically determining an optimal solution in polynomial time is not (yet) known. $NP$-complete problems are a subset of $NP$ problems for which it is possible to reduce any other problem in $NP$ to a problem in the subset $NP$-complete in some polynomial time process. This is important, because then if it is then shown that the Knapsack Problem (an NP-complete problem) is actually in $P$ (a method of determining an optimal solution quickly exists), then all NP problems will be in $P$, proving $P = NP$, which is sometimes considered to be the most important problem in computer science [5].

**Solving the 0-1 Knapsack Problem**

There are 2 main approaches to solving KP01: An exact solution using deterministic algorithms, and probabilistic approaches involving heuristic algorithms. A small-scale KP01 can be solved with deterministic approaches, but for high-scale situations it is not realistic to get optimal solutions with exact approaches [6] as the KP01 problem is NP-complete [7].

## Deterministic algorithms

Dynamic programming is a general technique for solving optimization problems. If a problem has an optimal substructure and over-lapping subproblems, then dynamic programming is applicable [8]. In computer science a problem has *optimal substructure* if an optimal solution for a problem can be constructed from optimal solutions of its sub-problems, and *over-lapping subproblems* is when a problem can be decomposed into sub-problems which are reused. Dynamic programming breaks down a complicated problem into smaller sub-problems in a recursive manner, while also using some memory to save the solutions of the sub-problems (usually in tabular form). This way, when one needs to get a solution for a sub-problem again, one can just use previously calculated values.

KP01 is solved using dynamic programming using a table based computation:

**Algorithm 1** Solving 0-1 Knapsack with Dynamic Programming

```
for i = 0 to noItems do                                   ▷ If no items, then profit = 0
    Table[i][0] ← 0
end for
for k = 0 to maxCapacity do                               ▷ If no capacity, then profit = 0
    Table[0][k] ← 0
end for
for i = 0 to noItems do
    for k = 0 to maxCapacity do
        Table[i][k] ← Table[i − 1][k]
        if k − weights[i] >= 0 then
            Table[i][k] ← max(Table[i − 1][k − weights[i]] + profits[i], Table[i][k])
        end if
    end for
end for
Print Table[noItems][maxCapacity]
```

This implementation of dynamic programming method has a time complexity of $\mathcal{O}(N \cdot W)$, where N is the number of items, and W is max capacity. The dynamic programming algorithm does not end until the entire table is built. This proves to be inefficient very quickly as the maximum capacity and the number of items in the knapsack increases. So, with the increase in the scale, the feasibility of dynamic programming decreases which has incentivized research in the subfield of heuristic algorithms [6].

## Heuristic algorithms

When deterministic algorithms prove to be too slow for practical applications such as online resource allocation, heuristic algorithms are chosen for their (usually) near optimal outputs and their speed. Heuristic algorithms should not be confused with approximation algorithms. Approximation algorithms guarantee a maximum margin of error; a constant factor off of the optimal. On the other hand, a heuristic algorithm does not guarantee anything, so it can perform better or worse than an approximation algorithm.

Heuristic algorithms for combinatorial optimization are usually designed in a specific way [9]:

1. Create a potential solution which is current best.
2. Generate new solution. (usually in polynomial time)
3. If new solution better than current best, swap out the best with the newly generated.
4. If satisfied, output best solution, else continue.

Both the Discrete Global-Best Harmony Search (DGHS), and Binary Multi-Scale Quantum Harmonic Oscillator (BMQHOA) have a similar framework.

### 2.1.1 Discrete Global-Best Harmony Search Algorithm

The Harmony Search algorithm (HS) was developed in 2001 based on the improvisation of music players who try to create better harmonies by adjusting previously known harmonies, hoping to produce more desirable ones [10]. This algorithm was made for continuous search spaces and thus cannot be used for discrete search spaces in combinatorial optimization problems, as in KP01, items are either fully chosen or not, meaning one cannot add a fraction of the weight and the profit value of a particular item into the knapsack. The DGHS was proposed by Wan-li Xiang et al to overcome this by rounding decimals to the nearest integer value to discretize the search space [11]. The DGHS for KP01 also has a repair-operator and a greedy selection mechanism. A harmony in this algorithm refers to a candidate solution, or a certain selection of items to be put in the knapsack.

It consists of 4 main parts:

1. Create a harmony memory HM - a fixed size (HMS) of randomly generated harmonies. Initialize parameters harmony memory consider rate (HMCR), and pitch adjusting rate (PAR), and calculate profit-density vector for usage in the repair-operator.
2. Create a new harmony using the current HM - Algorithm 3.
3. If generated harmony is better than the best in the harmony, replace it. If not, check if it is better than then worst harmony, and replace it if so.

4. If maximum number of iterations has been met, output the best sum profit value found so far.

**Algorithm 2** The DGHS algorithm

Set the harmony memory size $HMS$, the number of iterations $ITERATIONS$, and the minimum and maximum values of parameters $PAR$ and $HMCR$.
Initialize the HM through a randomized process, and use Algorithm 4 to the generated harmonies. Calculate the totalProfit and totalWeight values for each harmony in HM.
$iterator \leftarrow 1$
**while** $iterator <= ITERATIONS$ **do**
    Record indices of the best and the worst harmonies in HM.
    Calculate parameters HMCR and PAR for the current iteration.
    Perform Algorithm 3 to produce a new harmony $\vec{x}_{new}$
    Perform Algorithm 4 to repair the new harmony $\vec{x}_{new}$
    **if** $\vec{x}_{new}$ is better than or equal to $\vec{x}_{best}$ **then**
        Replace $\vec{x}_{best}$ with $\vec{x}_{new}$
    **else if** $\vec{x}_{new}$ is better than or equal to $\vec{x}_{worst}$ **then**
        Replace $\vec{x}_{worst}$ with $\vec{x}_{new}$
    **end if**
    $iterator \leftarrow iterator + 1$
**end while**
Output profit of best harmony

The parameters HMCR and PAR are used when generating new harmonies. They determine the likelihood of heading toward the current best harmony, and the likelihood of randomly flipping a decision variable. The idea is that current best harmony should have an influence on the generation of a new harmony, and that a decision variable (whether or not an item is put in the knapsack) should be flipped to allow for diversity in the HM, making it easier to overcome local maxima. These parameters are determined in terms of the current iteration like so:

$$HMCR(t) = HMCR_{max} - \frac{HMCR_{max} - HMCR_{min}}{ITERATIONS} t \tag{4}$$

$$PAR(t) = PAR_{max} - \frac{PAR_{max} - PAR_{min}}{ITERATIONS} t \tag{5}$$

The dynamic updating of the parameters is designed in this way to let larger values of HMCR help accelerate the convergence of the harmonies early on with the the help of the best individual harmony, while smaller values of HMCR can help to overcome local maxima [11]. Similarly, larger values of PAR in the beginning allow for increases in diversity through mutations (item decision bit flips - deciding to take an item when having excluded it before or vice versa), when there is time to "explore" different variants, smaller values allow convergence at the end of the search.

With these parameters one generates a new harmony:

**Algorithm 3** Generating a new harmony during iterative part (part 2)

**for** $i = 1$ to $noItems$ **do**
  **if** $rand(0, 1) <= HMCR(t)$ **then**
    $newHarmony[i] \leftarrow bestHarmony[i]$
  **else**
    Generate a random integer number $a \in \{1, 2, ...HMS\}, a \neq best$
    $newHarmony[i] \leftarrow Harmony_a[i]$
    **if** $rand(0, 1) <= PAR(t)$ **then**
      $newHarmony[i] = |newHarmony[i] - 1|$ ▷ Flipping i'th item - mutation
    **end if**
  **end if**
**end for**

Any time a harmony is generated, during Part 1 or Part 2, it is "repaired" in case the selection of items exceeds the capacity of the knapsack. The new generated harmony is also repaired if it can fit more items, without exceeding the capacity of the knapsack. The repair-operator consists of 2 phases:

1. Drop phase - repairing a harmony if it violates the constraint
2. Add phase - adding items into the knapsack if the new total weight is less than the capacity (the total weight of the knapsack stays below a previously decided value)

The add phase is always done after the drop phase, as a harmony previously infeasible becomes feasible after the drop phase, but its total weight may be less than the capacity of the knapsack. Here is the pseudocode for the repair-operator:

**Algorithm 4** Repair-operator for DGHS

**if** $totalWeight > maxWeight$ **then**
  **for** $i = 1$ to $N$ **do** ▷ DROP phase
    $\lambda_i = \frac{profit_i}{weight_i}$
  **end for**
  Sort items in increasing order of $\lambda_i$, and let $ind_i$ denote the original index of each $\lambda_i$
  **for** $i = 1$ to $noItems$ **do**
    Remove the ones with the least profit-density values greedily
    **if** $\lambda_i == 0$ **then**

```
            Continue
        end if
        newHarmony[ind_i] = 0                                        ▷ Unload the item
        totalWeight ← totalWeight − weight[ind_i]
        totalProfit ← totalProfit − profit[ind_i]
        if totalWeight <= maxWeight then
            Break                                                    ▷ Terminate DROP phase
        end if
    end for
end if
if totalWeight < maxWeight then
    for i = 1 to N do                                                ▷ ADD phase
        λ_i = profit_i / weight_i
    end for
    Sort items in increasing order of λ_i, and let ind_i denote the original index of each λ_i
    for i = 1 to noItems do
        Add the ones with the greatest profit-density if possible
        if newHarmony[ind_i] == 0 then
            if totalWeight + weight[ind_i] <= maxWeight then
                newHarmony[ind_i] = 1
                totalWeight ← totalWeight + weight[ind_i]
                totalProfit ← totalProfit + profit[ind_i]
            end if
        end if
    end for
end if
```

#### 2.1.2 Binary Multi-Scale Quantum Harmonic Oscillator Algorithm

The Multi-Scale Quantum Harmonic Oscillator Algorithm (MQHOA) is called as such as it follows a model of a solving a particle's ground state wave function under the harmonic oscillator potential well [12] . In MQHOA, candidate solutions are generated by sampling points in a Gaussian distribution within a certain distance of something. The Binary Multi-Scale Quantum Harmonic Oscillator Algorithm (BMQHOA) discretizes this by defining the number of bits between solutions (item decision variable) as the distance between solutions, so it becomes a discrete search space. Similar to the DGHS algorithm, a repair-operator is added to fix solutions that violate the capacity constraint.

The BMQHOA algorithm consists of 4 parts:

1. Random binary vector generation -> repair
2. Generating a solutions by flipping $m$ items, mutating, and repairing
3. Reducing the standard deviation value for the normal distribution from which the value for $m$ is determined during and for each iteration.
4. If termination constraint is satisfied, output the best total profit value found so far, else return to Step 2.

---

**Algorithm 5** The BMQHOA algorithm with solution generation

Set the number of iterations $ITERATIONS$ and the number of binary vectors (solutions) - BINVEC in memory.
Randomly generate the binary vectors and use Algorithm 6 to repair the vectors.
**while** $iterator <= ITERATIONS$ **do**
  Update $\sigma_s$
  $found \leftarrow FALSE$
  **while** $found == FALSE$ **do**
    Try to generate a solution
    Let $solutions_{new}$ be the new generated vector
    $solutions_{new} \leftarrow solutions_{best}$
    Generate the number of flipped bits $m$ $N(0, \sigma_s)$
    Treat $solutions_{new}$ as a circular array
    Randomly select a position in $solutions_{new}$ and flip the next $m$ items.
    Mutate a random bit towards current best solution (flip an item)
    $solutions_{new}[rand] \leftarrow solutions_{best}[rand]$
    Repair newly generated solution
    **if** $totalProfit >= totalProfit_{worst}$ **then**
      Replace worst solution with newly generated solution
      $solutions_{worst} \leftarrow solutions_{new}$
      $found = TRUE$
    **end if**
  **end while**
**end while**

---

After generating a new solution, $m$ items are flipped in the aspiration of increasing diversity of the solutions and increasing the likelihood of finding the optimal solution. A normal distribution is defined by a mean value and a standard deviation. The value for $m$ is generated randomly by using a normal probability distribution which has a mean of 0 and a Std. Deviation value which is determined by the iteration the algorithm is on. The Std. Deviation decreases with each new-solution-generation iteration so as to increase diversity in the beginning of the search, while reducing the likelihood of deteriorating a binary vector near the end of the search when close to optimal solutions have probably been found [12]. Initially the value of the standard deviation is set to $noItems/3$, so that there is a $\sim 99.7\%$ chance that the generated number of flipped items $m$ are in the range $[0 - 3\sigma_s, 0 + 3\sigma_s] \approx [-noItems, noItems]$, however in the case that $m \notin [-noItems, noItems]$, the value is reduced mod $noItems$:

$$STDEV_{max} = noItems/3$$
$$\sigma_s = 1 - \frac{t}{ITERATIONS} STDEV_{max}$$

The normal distribution probability density function generator is implemented with the C++ Standard Library class normal_distribution. An item's decision value is also made to match the corresponding decision in the best solution, which gives a slow mutation towards the current best solution allowing for diversity while still allowing the best solution to influence the process. This is because allowing multiple bits to mutate toward current best can lead to a premature local maximum, nullifying the rest of the search [12] .

The repair-operator of the BMQHOA algorithm has 3 phases:

1. Density-first stage: The already selected items are sorted based on their profit to weight ratio in non-increasing order and then greedily selected while respecting the weight constraint.
2. Minimum-weight-first stage: Out of the items that were not selected in the first stage are sorted based on their weight values in non-decreasing order, then greedily selected while respecting the weight constraint.

$Q_1[1, 2, ..., n]$ is the index for the items in the density ratio array in the original vector. $Q_2[1, 2, ..., n]$ is index of the items in the minimum weight sorted array in the original vector.

---

**Algorithm 6** Repair-Operator for BMQHOA

Let $x$ be the current array/vector.
$totalWeight = 0, temp = 0, i = 0, b = 0$
Stage 1: Density first stage
**while** $temp < maxWeight$ **do**
    $totalWeight = temp$
    $i+ = 1$
    $temp = temp + weight[Q_1[i]]$
**end while**
**for** $j = i$ to $n$ **do**
    $x[Q_1[j]] = 0$
    $totalWeight+ = x[Q_1[j]]$
**end for**

Stage 2: Minimum weight first stage

```
while totalWeight < C do
    b+ = 1
    if totalWeight + weight[Q_2[b]] <= C then
        x[Q_2[b]] = 1
        totalWeight+ = weight[Q_2[b]]
    end if
end while
```

# 3. Not finished - Methodology

Testdata is made up of different testcases, each of which is an input on which every algorithm is run.

## 3.1 Generating testdata

3 testgroups of 5 testcases were made using a random number generator, where a random number was generated using the Mersenne Prime Twister (mt19937). A computer cannot generate a truly random number. A pseudorandom number is what is generated, which is a number which appears to be statistically random, but has been generated using a deterministic process. The 3 groups had item counts of 100, 1000, and 10000. The number of solutions in memory (for the heuristic algorithms) was put in the input file which was kept constant, 20, in all testcases and algorithms for this report. In the random number generated testcases, the number of items $n$ was chosen to either be 100, 1000 or 10000. The profit and weight values were then randomly generated in a range of $[1, noItems]$ while a sum-of-weights variable $sum$ was kept. The max weight was then randomly generated in a range of $[\frac{\text{sum}}{100}, \frac{\text{sum}}{10}]$, so that the knapsack wouldn't be able to hold all the items. To avoid any overflow errors, all generated numbers (and the sum of the profit/weight values) were kept below the max value of an $int$ which is $2^{31} - 1$ in many programming languages. Then $n$, $W$ (maxWeight), and each item (profit and weight values) was put in an input file. To make a fair comparison the number of $ITERATIONS$ (how many times a candidate solution is generated) was set to 100 for both algorithms.

3 groups of five testcases were taken from a database ??. These 3 "testgroups" are mentioned/used several times in the literature. The 3 testgroups are uncorrelated, weakly correlated, and strongly correlated; the correlation being between the items' profit and weight values. The number of solutions in memory (20) was added to each of these testcases taken from ??.

These input files were then saved for later testing of the algorithms.

Each testcase was then run on each algorithm and saved in a spreadsheet producing a csv table, refer to appendix.

# 4. Not finished - Results

## 4.1 Raw Data

Refer to the appendix for the raw data for the algorithms running on the six testgroups: 100, 1000, 10k randomized numbers respectively and 3 differently correlated datasets.

## 4.2 Processed Data

The mean and standard deviations for each group can be viewed in the following corresponding tables and graphs:

Table 1: Mean and StDev. of 100 Randomized items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 4160.33±3.50 | 2561±0 | 5288±19.66 | 3535±0 | 1747±0 |
| Mean Percent deviation from optimal | 0.21% | 0% | 0.40% | 0% | 0% |
| BMQHOA Mean | 3926.83±40.27 | 2561.00±0 | 4879±46.18 | 3360.66±68.96 | 1711.5±25.71 |
| Mean Percent deviation from optimal | 5.80% | 0% | 8.09% | 4.93% | 2.03% |

Table 2: Mean and StDev. of 1000 Randomized items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 156083.83±830.04 | 118603.83±830.04 | 144070±1343.77 | 109734.33±1037.03 | 126366.16±1236.73 |
| Mean Percent deviation from optimal | 13.71% | 11.76% | 14.12% | 10.77% | 13.27% |
| BMQHOA Mean | 145086.5±1035.45 | 109876.33±1540.63 | 136622.66±2042.95 | 101951.33±1271.88 | 118291.16±799.78 |
| Mean Percent deviation from optimal | 19.79% | 18.25% | 18.56% | 17.10% | 18.82% |

Table 3: Mean and StDev. of 10k Randomized items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 7681910±51555 | 10567302±23956 | 8014906±39449.15 | 9347061±56967.66 | 9583139±55816.60 |
| Mean Percent deviation from optimal | 22.61% | 23.11% | 22.07% | 23.10% | 23.00% |
| BMQHOA Mean | 7438487±29887.17 | 10276681±50507.35 | 7774520±34559.10 | 9075476±53168.02 | 9330322±46301.35 |
| Mean Percent deviation from optimal | 25.07% | 25.22% | 24.41% | 25.33% | 25.03% |

Table 4: Mean and StDev. of Uncorrelated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 9147±0 | 11238±0 | 51019.50±437.85 | 99036.16±1544.34 | 448975±1135.46 |
| Mean Percent deviation from optimal | 0% | 0.00% | 6.39% | 10.47% | 20.34% |
| BMQHOA Mean | 8974.5±147.24 | 10976.50±154.37 | 47401.50±841.75 | 91725.83±1346.23 | 431259.66±3049.43 |
| Mean Percent deviation from optimal | 1.88% | 2.32% | 13.02% | 17.08% | 23.48% |

Table 5: Mean and StDev. of Weakly Correlated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 1514±0 | 1627.66±2.42 | 8907.33±27.56 | 17371.83±126.76 | 82681.33±390.98 |
| Mean Percent deviation from optimal | 0% | 0.38% | 1.59% | 3.76% | 8.33% |
| BMQHOA Mean | 1508.16±6.82 | 1613.66±16.74 | 8435.33±60.60 | 16539±133.95 | 80900.66±970.71 |
| Mean Percent deviation from optimal | 0.38% | 1.24% | 6.81% | 8.37% | 10.31% |

Table 6: Mean and StDev. of Strongly Correlated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| DGHS Mean | 2391.66±6.08 | 2697±0 | 13682.5±60.05 | 26444.66±240.84 | 126448±615.61 |
| Mean Percent deviation from optimal | 0.22% | 0% | 4.91% | 8.55% | 13.93% |
| BMQHOA Mean | 2327.33±48.55 | 2658.66±48.81 | 13334.66±182.91 | 25604±91.32 | 123813.83±773.19 |
| Mean Percent deviation from optimal | 2.90% | 1.42% | 7.33% | 11.46% | 15.72% |

Table 7: Mean percentage deviation from optimal across all datasets

| | DGHS mean percentage deviation | BMQHOA mean percentage deviation |
|---|---|---|
| 100 randomized items | 0.12% | 4.17% |
| 1000 randomized items | 12.73% | 18.50% |
| 10k randomized items | 22.78% | 25.01% |
| Uncorrelated dataset | 7.44% | 11.56% |
| Weakly Correlated dataset | 2.81% | 5.42% |
| Strongly Correlated Dataset | 5.52% | 7.77% |

Mean percentage deviation from the optimal for DGHS and BMQHOA across all datasets

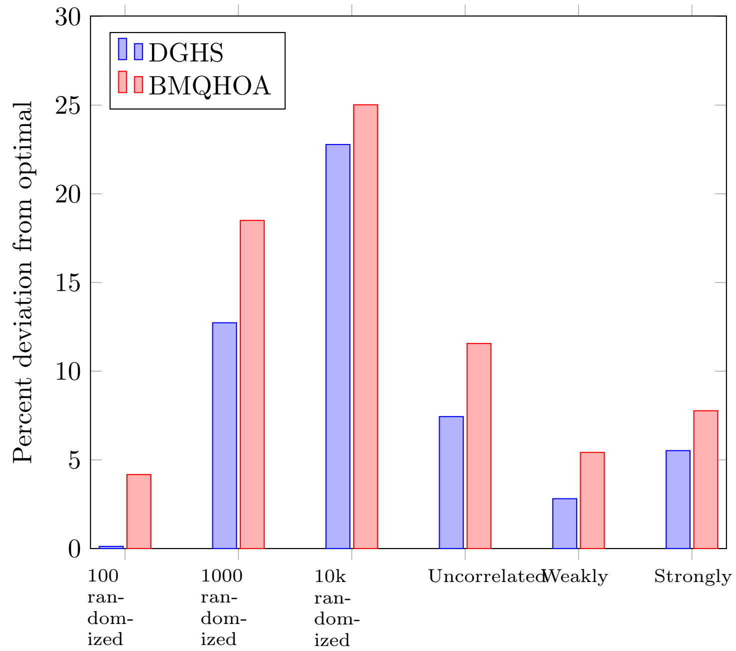



Figure 1: Mean percentage deviation from optimal across all datasets

Means of total profit values generated by DGHS and BMQHOA for 100 randomized items

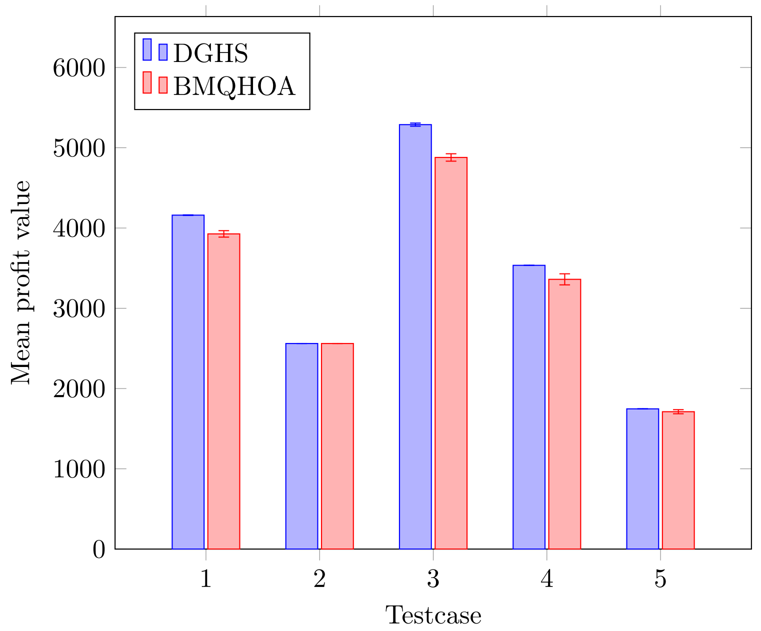



Figure 2: Shows the means of profit values for 100 randomized items

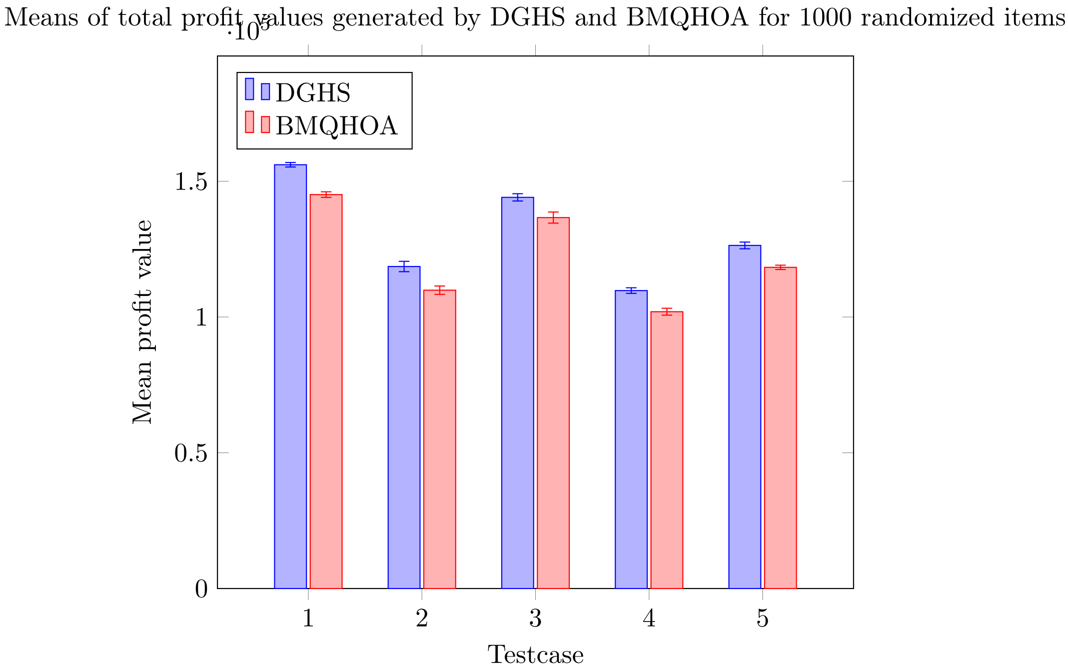



Figure 3: Shows the means of profit values for 1000 randomized items

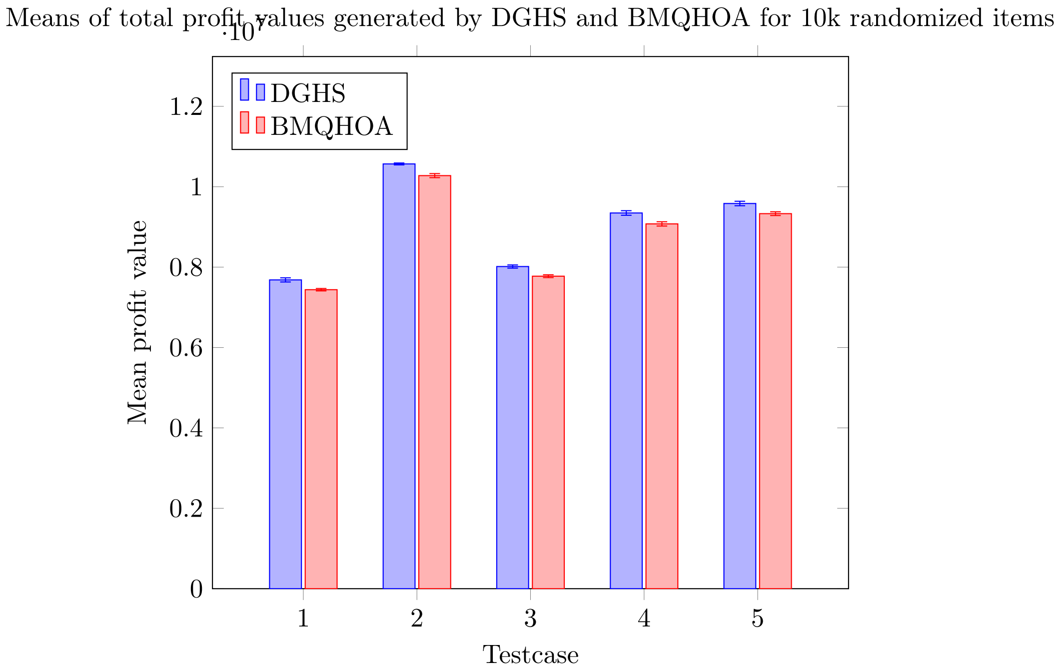



Figure 4: Shows the mean of profit values for 10k randomized items

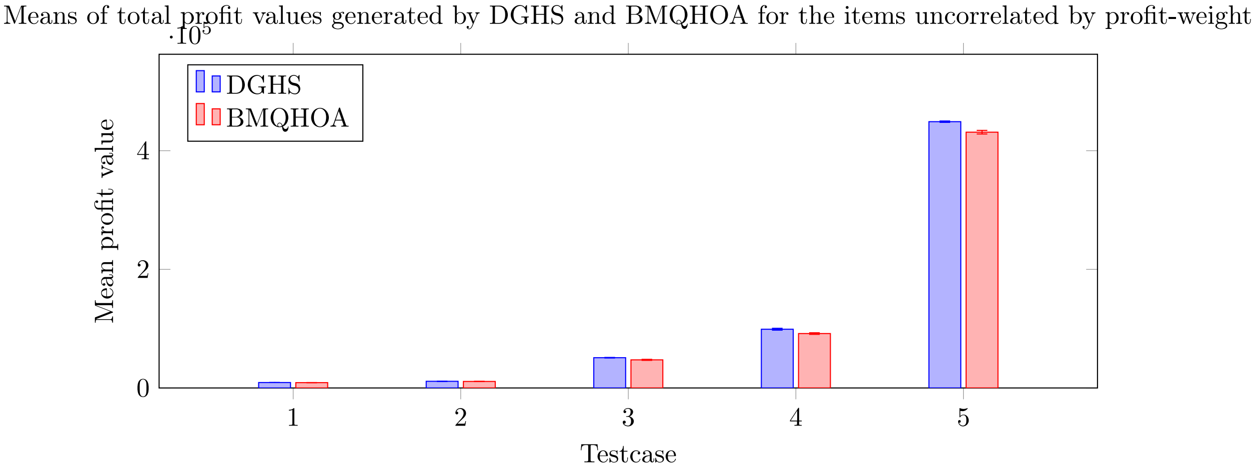



Figure 5: Shows the means of profit values for uncorrelated items

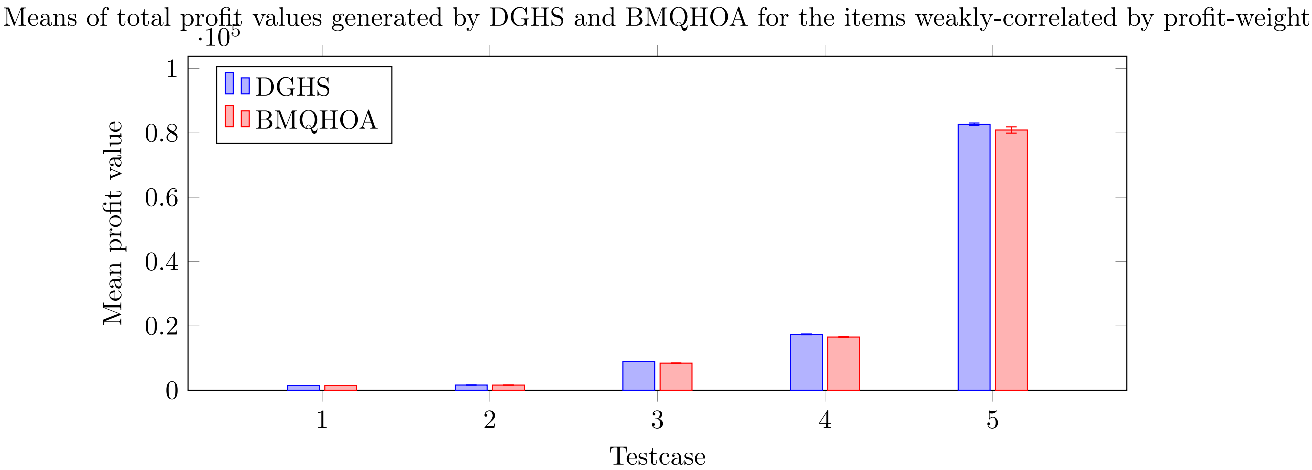



Figure 6: Shows the means of profit values for weakly correlated items

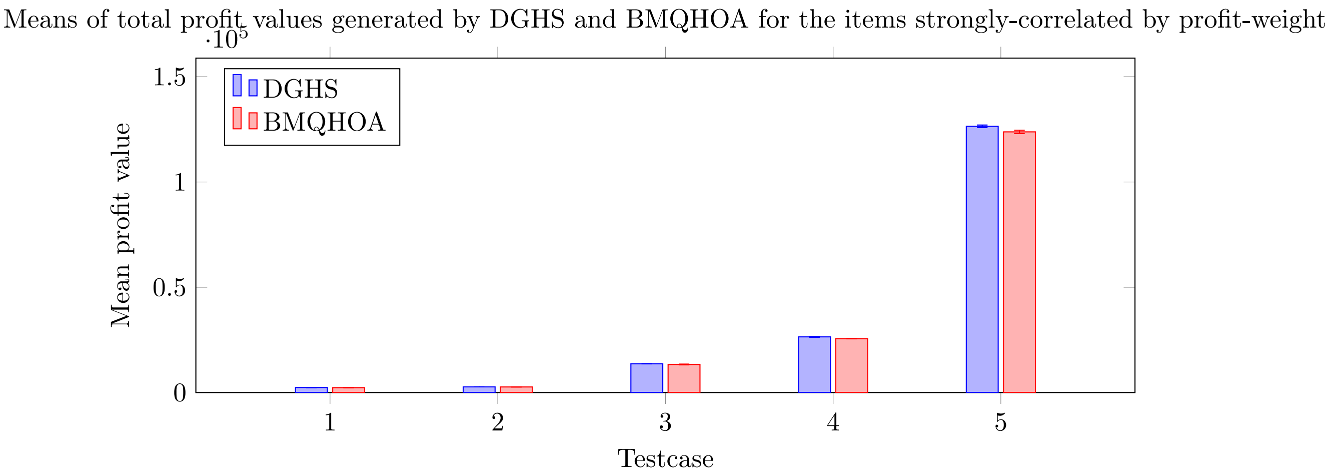





Figure 7: Shows the means of profit values for strongly correlated items

# 5. Not finished - Discussion

As one can see in Figure 1, the DGHS algorithm deviates more than the BMQHOA algorithm in all datasets. One can draw the conclusion that the DGHS algorithm is superior.

In Table 7 and the corresponding Figure 1, the data shows that testcases with a greater number of items have resulted in greater deviations from the optimal. Both algorithms are probabilistic and work by generating new candidate solutions, and replacing worse solutions with better ones. As the generation process is only run 100 times in both algorithms, and as there are $2^{noItems}$ different possible solutions (possibly violating the constraints), the likelihood of finding optimal or near-optimal solutions decreases.

The BMQHOA algorithm only flips 1 item toward the corresponding item in the current best solution during the solution generation process. The DGHS harmony generation, on the other hand, probabilistically considers every item for flipping toward the best solution, depending on the current iteration's HMCR value. As the BMQHOA has produced worse profit values, it can be suggested that a greater level of evolution toward current best solutions is advantageous - so flipping more items toward the best solution in the BMQHOA algorithm might have resulted in better profit values.

## References


[1] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Knapsack Problems.* 01 2004.

[2] Bo Sun, Lin Yang, Mohammad Hajiesmaili, Adam Wierman, John C.s Lui, Don Towsley, and Danny Tsang. The online knapsack problem with departures, 09 2022.

[3] Andrew Ellinor, Ivan Koswara, and Agnishom Chattopadhyay. Bigo notation, 2023.

[4] Scott Aaronson. P ?= np. *Weizmann Institute of Science*, January 2017.

[5] William L.Hosch. P vs np problem. *Encyclopedia Britannica*, February 2023.

[6] Yangjun Gao, Fengming Zhang, Yu Zhao, and Chao Li. Quantum-inspired wolf pack algorithm to solve the 0-1 knapsack problem. *Mathematical Problems in Engineering*, 2018:1–10, 06 2018.

[7] Subham Datta. 0-1 knapsack: A problem with np-completeness and solvable in pseudo-polynomial time, November 2022.

[8] Eric Grimsom and John Guttag. Dynamic programming: Overlapping subproblems, optimal substructure. Technical report, Massachusetts Institute of Technology, 2008.

[9] Anmol Singh. Heuristic algorithms. Technical report, Cornell University, 2020.

[10] Zong Woo Geem, Joong Kim, and G.V. Loganathan. A new heuristic optimization algorithm: Harmony search. *Simulation*, 76:60–68, 02 2001.

[11] Wan-li Xiang, Mei-qing An, Yinzhen Li, Rui-chun He, and Jing-fang Zhang. A novel discrete global-best harmony search algorithm for solving 0-1 knapsack problems. *Discrete Dynamics in Nature and Society*, 2014:1–12, 04 2014.

[12] Yan Huang, Peng Wang, Jianping Li, Xiuhong Chen, and Tao Li. A binary multi-scale quantum harmonic oscillator algorithm for 0–1 knapsack problem with genetic operator. *IEEE Access*, PP:1–1, 09 2019.


# 6. Appendix

Table 8: Randomized 100 items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 4169 | 2561 | 5309 | 3535 | 1747 |
| DGHS | 4163 | 2561 | 5254 | 3535 | 1747 |
| | 4163 | 2561 | 5292 | 3535 | 1747 |
| | 4161 | 2561 | 5307 | 3535 | 1747 |
| | 4163 | 2561 | 5282 | 3535 | 1747 |
| | 4155 | 2561 | 5307 | 3535 | 1747 |
| | 4157 | 2561 | 5286 | 3535 | 1747 |
| BMQHOA | 3895 | 2561 | 4952 | 3398 | 1734 |
| | 3939 | 2561 | 4871 | 3380 | 1728 |
| | 3870 | 2561 | 4890 | 3298 | 1705 |
| | 3948 | 2561 | 4829 | 3435 | 1718 |
| | 3984 | 2561 | 4832 | 3255 | 1663 |
| | 3925 | 2561 | 4900 | 3398 | 1721 |

Table 9: Randomized 1000 items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 4169 | 2561 | 5309 | 3535 | 1747 |
| DGHS | 156898 | 119888 | 141878 | 110608 | 128009 |
| | 155104 | 117086 | 143322 | 108732 | 124714 |
| | 155849 | 117110 | 145381 | 108973 | 125400 |
| | 155171 | 116958 | 145411 | 109414 | 126089 |
| | 156929 | 118957 | 143984 | 109292 | 126532 |
| | 156552 | 121624 | 144444 | 111387 | 127453 |
| BMQHOA | 146187 | 110066 | 136620 | 103073 | 117832 |
| | 146215 | 107880 | 133731 | 101385 | 117207 |
| | 145231 | 109374 | 134981 | 99977 | 119272 |
| | 143511 | 108754 | 137715 | 102749 | 117836 |
| | 144522 | 112065 | 137185 | 101318 | 118550 |
| | 144853 | 111119 | 139504 | 103206 | 119050 |

Table 10: Randomized 10k items

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 9927473 | 13743791 | 10286004 | 12155538 | 12446154 |
| DGHS | 7667918 | 10556718 | 8064941 | 9425334 | 9545542 |
| | 7738704 | 10587645 | 8059383 | 9305736 | 9547723 |
| | 7590747 | 10523899 | 7986766 | 9304266 | 9679149 |
| | 7680907 | 10571613 | 8018689 | 9377397 | 9621182 |
| | 7719598 | 10584319 | 7983856 | 9386494 | 9538807 |
| | 7693588 | 10579618 | 7975801 | 9283143 | 9566432 |
| BMQHOA | 7483414 | 10312486 | 7823273 | 9090434 | 9368506 |
| | 7398497 | 10196292 | 7796298 | 9030552 | 9332763 |
| | 7429280 | 10328466 | 7768443 | 9107012 | 9246868 |
| | 7452071 | 10285331 | 7786678 | 9064793 | 9351821 |
| | 7450200 | 10236135 | 7737071 | 9006745 | 9312696 |
| | 7417461 | 10301381 | 7735362 | 9153320 | 9369279 |

Table 11: Uncorrelated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 9147 | 11238 | 54503 | 110625 | 563647 |
| DGHS | 9147 | 11238 | 51166 | 100770 | 450386 |
| | 9147 | 11238 | 50911 | 98217 | 448025 |
| | 9147 | 11238 | 50674 | 100153 | 449887 |
| | 9147 | 11238 | 51168 | 100285 | 447368 |
| | 9147 | 11238 | 51720 | 97570 | 448856 |
| | 9147 | 11238 | 50478 | 97222 | 449328 |
| BMQHOA | 8990 | 11005 | 47569 | 91878 | 429245 |
| | 9147 | 11031 | 47106 | 90966 | 437054 |
| | 8911 | 10759 | 47773 | 93434 | 432253 |
| | 9147 | 11168 | 47302 | 91491 | 429942 |
| | 8810 | 10826 | 48608 | 89694 | 429823 |
| | 8842 | 11070 | 46051 | 92892 | 429241 |

Table 12: Weakly Correlated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 1514 | 1634 | 9052 | 18051 | 90204 |
| DGHS | 1514 | 1627 | 8917 | 17251 | 82712 |
| | 1514 | 1629 | 8898 | 17224 | 82571 |
| | 1514 | 1629 | 8899 | 17409 | 82348 |
| | 1514 | 1629 | 8892 | 17519 | 83330 |
| | 1514 | 1623 | 8880 | 17321 | 82254 |
| | 1514 | 1629 | 8958 | 17507 | 82873 |
| BMQHOA | 1512 | 1633 | 8516 | 16578 | 80645 |
| | 1512 | 1625 | 8493 | 16493 | 80431 |
| | 1514 | 1604 | 8412 | 16415 | 80591 |
| | 1501 | 1626 | 8385 | 16495 | 80443 |
| | 1498 | 1604 | 8362 | 16463 | 82873 |
| | 1512 | 1590 | 8444 | 16790 | 80421 |

Table 13: Strongly correlated Dataset

| Testcase | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Optimal | 2397 | 2697 | 14390 | 28919 | 146919 |
| DGHS | 2390 | 2697 | 13586 | 26711 | 126318 |
| | 2390 | 2697 | 13690 | 26708 | 127616 |
| | 2397 | 2697 | 13673 | 26311 | 126212 |
| | 2381 | 2697 | 13686 | 26513 | 126513 |
| | 2396 | 2697 | 13685 | 26114 | 126212 |
| | 2396 | 2697 | 13775 | 26311 | 125817 |
| BMQHOA | 2294 | 2596 | 13183 | 25712 | 124918 |
| | 2296 | 2681 | 13090 | 25583 | 123503 |
| | 2390 | 2696 | 13570 | 25706 | 123613 |
| | 2297 | 2689 | 13490 | 25498 | 123217 |
| | 2390 | 2596 | 13288 | 25515 | 123017 |
| | 2297 | 2694 | 13387 | 25610 | 124615 |