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 Dinnetz

Kungsholmens Gymnasium

Contents

List of Algorithms

§ 1. Introduction

§ 1.1 Background

There are many situations in every day life, where a person wonders whether he is doing something efficiently. Unforunately, the human brain is not always capable of coming up with an optimal approach when the problem has a lot of factors at play. It is here when a 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 - an output. The efficiency (in all aspects) of the algorithm is dependant on the instructions which make up it. There are many algorithms which have been designed to solve many problems. For problems in computer science, more than one algorithm is usually proposed and used. Optimization problems are where this is frequently the case.

In the fields of computer science and mathematics, optimization problems are problems of finding the best 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 a discrete but (usually large) space.

Typical combinatorial optimization problems are not limited to but 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 $\{1, 2, ..., n\}$, what is the and a collection of m sets whose union equals the universe, what is the smallest sub-collection of 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 the main focus in the research community. It appears in real-world decision-making processes in a variety of fields. Some examples include:

Many of combinatorial optimization problems including the 0-1 Knapsack problem currently do not have deterministic algorithms which are considered fast enough for them to be used on a large-scale basis. Consequently, the research focus has been on approaches that do not necessarily guarantee the best solution but win over deterministic approaches when it comes to time.

Problem statement

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

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

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

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

The rest of this paper will follow the shorthand "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 global-best harmony search algorithm and the binary harmonic multi-scale algorithms perform when implemented for solving KP01?

§ 2. Theory

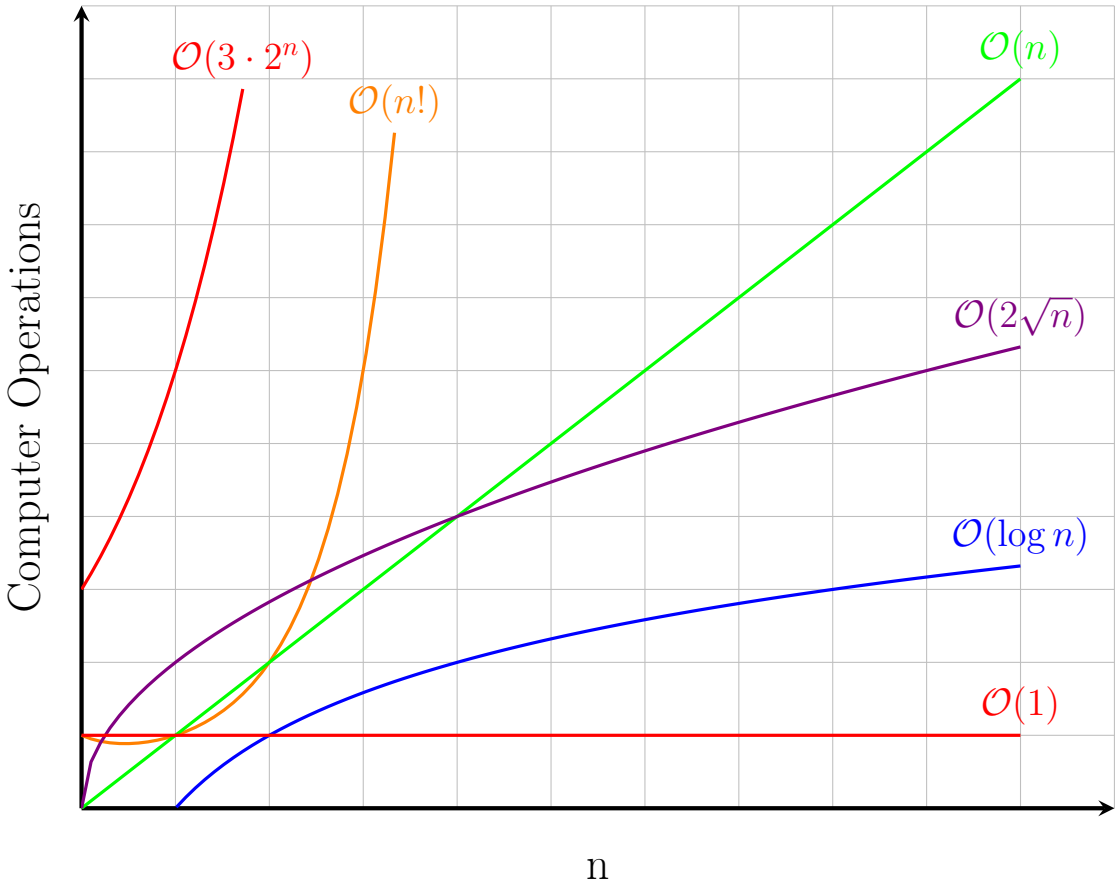
§ 2.1 Computational Complexity

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, notated time complexity and space complexity. 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 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 n numbers $a_1, a_2, a_3, ..., a_n$ are given, and an algorithm checks the existence of an certain element in those numbers, by checking all n elements, 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.

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.



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 ?? as the KP01 problem is NP-complete ??.

Deterministic algorithms

Dynamic programming is a general technique for solving optimization problems. If a problem has optimal substructure and over-lapping subproblems, then dynamic programming is applicable. In computer science a problem has *optimal substructure* if an optimal solution for a problems 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 complicate 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 we

need to get a solution for a sub-problem again, we can just use our previously calculated value.

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.

Heuristic algorithms

When deterministic algorithms prove to be too slow for practical applications, 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 ??:

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 EXPAND. This algorithm was made for continuous search spaces and thus cannot be used for discrete search spaces in combinatorial optimization problems. The Discrete Global-Best harmony search was proposed by EXPAND to overcome this. 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 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

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

12: Replace $\vec{x}_{worst}$ with $\vec{x}_{new}$
13: **end if**
14: $iterator \leftarrow iterator + 1$
15: **end while**
16: 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 you overcome local maxima ??. Similarly, larger values of PAR in the beginning allow for increases in diversity through mutations (item decision bit flips), 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)

1: **for** $i = 1$ to $noItems$ **do**
2: **if** $rand(0,1) <= HMCR(t)$ **then**
3: $newHarmony[i] \leftarrow bestHarmony[i]$
4: **else**
5: Generate a random integer number $a \in \{1, 2, ...HMS\}, a \neq best$
6: $newHarmony[i] \leftarrow Harmony_a[i]$
7: **if** $rand(0,1) <= PAR(t)$ **then**
8: $newHarmony[i] = |newHarmony[i] - 1|$ ▷ Flipping i'th item - mutation
9: **end if**
10: **end if**
11: **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 a items into the knapsack if the total weight is less than the capacity

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 λ_i, and let ind_i denote the original index of each λ_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
    $totalWeight \leftarrow totalWeight - weight[ind_i]$
    $totalProfit \leftarrow totalProfit - profit[ind_i]$
    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
    $\lambda_i = \frac{profit_i}{weight_i}$
  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

26: **if** $newHarmony[ind_i] == 0$ **then**
27: **if** $totalWeight + weight[ind_i] <= maxWeight$ **then**
28: $newHarmony[ind_i] = 1$
29: $totalWeight \leftarrow totalWeight + weight[ind_i]$
30: $totalProfit \leftarrow totalProfit + profit[ind_i]$
31: **end if**
32: **end if**
33: **end for**
34: **end if**

§ 2.1.2 Binary Multi-Scale Quantum Harmonic Oscillator Algorithm

The Multi-Scale Quantum Harmonic Oscillator Algorithm (MQHOA) is called such as it follows a model of a solving a particle's ground state wave function under the harmonic oscillator potential well ?? . 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 fix solutions which 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 m is determined every iteration.
4. If termination constraint is satisfied, output best totalProfit value, else return to Step 2.

Algorithm 5 The BMQHOA algorithm with solution generation

1: Set the number of iterations $ITERATIONS$ and the number of binary vectors (solutions) - BINVEC in memory.
2: Randomly generate the binary vectors and use Algorithm 6 to repair the vectors.
3: **while** $iterator <= ITERATIONS$ **do**
4: Update σ_s
5: $found \leftarrow FALSE$
6: **while** $found == FALSE$ **do**
7: Try to generate a solution
8: Let $solutions_{new}$ be the new generated vector

```
        solutions_new ← solutions_best
        Generate the number of flipped bits m N(0, σ_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] ← solutions_best[rand]
        Repair newly generated solution
        if totalProfit >= totalProfit_worst then
            Replace worst solution with newly generated solution
            solutions_worst ← solutions_new
            found = TRUE
        end if
    end while
end while
```

After generating a new solution, m items are flipped in the hope of increasing diversity in the solutions and increasing the likelihood of finding the optimal solution. They are generated randomly with a normal probability distribution around $\mathbf{0}$, with a mean of 0 and a variable std deviation. 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 corrupting a binary vector closer to the end of the search when close to optimal solutions have (hopefully) been found. 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 $[0 - 3\sigma_s, 0 + 3\sigma_s] \approx [-noItems, noItems]$:

$$STDEV_{max} = noItems/3$$

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

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. Allowing multiple bits to mutate toward current best, this can lead to a premature local maximum, nullifying the rest of the search ?? .

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 weren't 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 more 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. 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: Randomized 100 items

Testcase	1	2	3	4	5
Harmony Mean	4160.33	2561	5288	3535	1747
Harmony StDev.	3.50	0	19.66	0	0
BMQHOA Mean	3926.83	2561	4879	3360.66	1711.5
BMQHOA StDev.	40.27	0	46.18	68.96	25.71

Table 2: Randomized 1000 items

Testcase	1	2	3	4	5
Harmony Mean	156083.83	118603.83	144070	109734.33	126366.16
Harmony StDev.	830.04	1904.69	1343.77	1037.03	1236.73
BMQHOA Mean	145086.5	109876.33	136622.66	101951.33	118291.16
BMQHOA StDev.	1035.45	1540.63	2042.95	1271.88	799.78

Table 3: Randomized 10k items

Testcase	1	2	3	4	5
Harmony Mean	7681910.33	10567302	8014906	9347061.66	9583139.16
Harmony StDev.	51555.39	23956.31	39449.15	56967.66	55816.60
BMQHOA Mean	7438487.16	10276681.83	7774520.83	9075476	9330322.16
BMQHOA StDev.	29887.17	50507.35	34559.10	53168.02	46301.35

Table 4: Uncorrelated Dataset

Testcase	1	2	3	4	5
Harmony Mean	9147	11238	51019.5	99036.16	448975
Harmony StDev.	0	0	437.85	1544.34	1135.46
BMQHOA Mean	8974.5	10976.5	47401.5	91725.83	431259.66
BMQHOA StDev.	147.24	154.37	841.75	1346.23	3049.43

Table 5: Weakly Correlated Dataset

Testcase	1	2	3	4	5
Harmony Mean	1514	1627.66	8907.33	17371.83	82681.33
Harmony StDev.	0	2.42	27.56	126.76	390.98
BMQHOA Mean	1508.16	1613.66	8435.33	16539	80900.66
BMQHOA StDev.	6.82	16.74	60.60	133.95	970.71

Table 6: Strongly Correlated Dataset

Testcase	1	2	3	4	5
Harmony Mean	2391.66	2697	13682.5	26444.66	126448
Harmony StDev.	6.08	0	60.05	240.84	615.61
BMQHOA Mean	2327.33	2658.66	13334.66	25604	123813.83
BMQHOA StDev.	48.55	48.81	182.91	91.32	773.19

§ 5. Appendix

Table 7: Randomized 100 items

Testcase	1	2	3	4	5
Optimal	4169	2561	5309	3535	1747
Harmony	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 8: Randomized 1000 items

Testcase	1	2	3	4	5
Optimal	4169	2561	5309	3535	1747
Harmony	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 9: Randomized 10k items

Testcase	1	2	3	4	5
Optimal	9927473	13743791	10286004	12155538	12446154
Harmony	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 10: Uncorrelated Dataset

Testcase	1	2	3	4	5
Optimal	9147	11238	54503	110625	563647
Harmony	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 11: Weakly Correlated Dataset

Testcase	1	2	3	4	5
Optimal	1514	1634	9052	18051	90204
Harmony	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 12: Strongly correlated Dataset

Testcase	1	2	3	4	5
Optimal	2697	14390	28919	146919	
Harmony	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