HOMEWORK 3 – KNAPSACK ALGORITHMS: EXPERIMENTAL EVALUATION

BY ANDRÉS RUBIO RAMOS

ALGORITHMS USED

Brute Force

This Java algorithm is an iterative implementation of the Brute Force method for solving the 0/1 Knapsack Problem. It explores all combination of variables selected and not selected to find the optimum solution which is stored in the variable "bestSol" with its "bestCost".

```
private static Tuple bruteForce(Knapsack dk) {
    int n=dk.n;
    int maxComb= (int) Math.pow(2, n)-1;
    int bestValue=-1;
    List<Integer> bestSol=new ArrayList<>();
    for (int i=maxComb; i>=0;i--) {
        int currentWeight = 0;
        int currentValue = 0;
        List<Integer> sol=new ArrayList<>();
        for (int j=0; j<n; j++) {
            if ((i & (1<<j)) != 0) {
                if (currentWeight + dk.items().get(j).weight() <= dk.M) {
                    sol.add(1);
                    currentWeight += dk.items().get(j).weight();
                    currentValue += dk.items().get(j).cost();
                } else sol.add(0);
            } else sol.add(0);
        }
        if (currentValue > bestValue) {
            bestValue=currentValue;
            bestSol=sol;
        }
    }
    return new Tuple(bestSol,bestValue);
}
```

Branch & Bound

This Java algorithm is a recursive implementation of the Branch and Bound method for solving the 0/1 Knapsack Problem. It seeks to find the optimal solution by systematically exploring different combinations of items while pruning branches that are guaranteed not to lead to a better solution. The "initBranchAndBound" function initializes the process, while "branchAndBound" is the recursive function that explores various item inclusion/exclusion possibilities. The algorithm maintains "maxValue" to keep track of the maximum value found so far and "solOpt" to store the corresponding solution. The algorithm terminates when all items have been considered, and it returns the best solution and its value.

```
static int maxValue;
static List<Integer> solOpt;
private static Tuple initBranchAndBound(Knapsack k) {
    maxValue=Integer.MIN_VALUE;
    solOpt=new ArrayList<Integer>();
    branchAndBound(k, 0, k.M, 0, new ArrayList<Integer>());
    for (int i=solOpt.size(); i<k.n;i++) solOpt.add(0);
    return new Tuple(solOpt, maxValue);
}
private static void branchAndBound(Knapsack k, int index,
        int capacity, int value, List<Integer> list) {
    if (capacity<0) return;
    if (index==k.n) {
        if (value>maxValue) {
            maxValue=value;
            solOpt=list;
        }
        return;
    }
    Integer costBound=value+k.items().subList(index, k.n()).
            stream().mapToInt(x->x.cost()).sum();
    if (costBound>maxValue) {
        List<Integer> listYes=new ArrayList<Integer>(list);
        listYes.add(1);
        branchAndBound(k, index+1, capacity-k.items().get(index).
                weight(), value+k.items().get(index).cost(), listYes);
        List<Integer> listNo=new ArrayList<Integer>(list);
        listNo.add(0);
        branchAndBound(k, index+1, capacity, value, listNo);
    }
    return;
}
```

cost/weight heuristic

This Java algorithm represents a simple heuristic approach to solving the Knapsack Problem. It initializes a solution list with all zeros and then sorts the items in the knapsack by their cost-to-weight ratio in descending order. The algorithm iterates through the sorted items, selecting them if their weight can fit within the remaining capacity, updating the total cost and solution list accordingly. Finally, it returns a Tuple containing the solution and the total cost of the selected items.

```
private static Tuple simpleHeuristic(Knapsack k) {
    Integer totalCost=0;
    Integer capacity=k.M();
    List<Integer> sol=new ArrayList<Integer>();
    for (int i=0;i<k.n();i++) sol.add(0);
    List<Item> ordKnapsack=k.items().stream().
            sorted(Comparator.comparingDouble(
                    (Item x)->1.*x.cost()/x.weight()).
                    reversed()).toList();
    for (Item it:ordKnapsack)
        if (it.weight()<=capacity) {
            totalCost+=it.cost();
            capacity-=it.weight();
            sol.set(it.index(), 1);
        }
    return new Tuple(sol, totalCost);
}
```

Dynamic programming (capacity)

This Java algorithm is a recursive implementation of dynamic programming decomposition by capacity for solving the Knapsack Problem:

The algorithm utilizes a matrix called “solMatrix” to store computed solutions and avoid redundant calculations.

It starts with the “initDynaByCapacity” function, converting item weights and costs into lists “W” and “C”.

The “dynaByCapacity” function recursively calculates the optimal solution using dynamic programming. It checks if a solution for the given parameters exists in the “solMatrix” and returns it if available. If not, it considers the current item's inclusion and exclusion, updating the solution accordingly.

The algorithm includes utility functions “isTrivial”, “trivialKNAP”, and “removeLast” to handle trivial cases, initialize a solution for trivial cases, and remove the last element from a list, respectively.

This dynamic programming approach systematically explores and calculates the optimal solution while reusing previously computed subproblems, making it an efficient way to solve the Knapsack Problem.

```
static Solution[][] solMatrix;
public static Solution initDynaByCapacity(Knapsack k) {
    solMatrix=new Solution[k.n()+1][k.M()+1];
    List<Integer> W=new ArrayList<>();
    List<Integer> C=new ArrayList<>();
    for (Item it:k.items()) {
        W.add(it.weight());
        C.add(it.cost());
    }
    return dynaByCapacity(W, C, k.M());
}
private static Solution dynaByCapacity(List<Integer> W, List<Integer> C, int M) {
    if (solMatrix[W.size()][M]!=null) return solMatrix[W.size()][M];
    Solution result;
    if (isTrivial(W, C, M)) return trivialKNAP(W, C, M);
    Solution withoutN = dynaByCapacity(removeLast(W), removeLast(C), M);
    if (W.get(W.size() - 1) <= M) {
        Solution withN = dynaByCapacity(removeLast(W), removeLast(C), M - W.get(W.size() - 1));
        if (withN.c + C.get(C.size() - 1) > withoutN.c) {
            List<Integer> newX = new ArrayList<>(withN.X);
            newX.add(1);
            result=new Solution(newX, withN.c + C.get(C.size() - 1), withN.m + W.get(W.size() - 1));
            solMatrix[W.size()][M]=result;
            return result;
        }
    }
    List<Integer> newX = new ArrayList<>(withoutN.X);
    newX.add(0);
    result=new Solution(newX, withoutN.c, withoutN.m);
    solMatrix[W.size()][M]=result;
    return result;
}



private static boolean isTrivial(List<Integer> W, List<Integer> C, int M) {
    return W.isEmpty() || M<=0;
}
private static Solution trivialKNAP(List<Integer> W, List<Integer> C, int M) {
    List<Integer> X = new ArrayList<>(W.size());
    for (int i = 0; i < W.size(); i++) X.add(0);
    int c = 0;
    int m = 0;
    return new Solution(X, c, m);
}
private static List<Integer> removeLast(List<Integer> list) {
    if (!list.isEmpty()) {
        List<Integer> result = new ArrayList<>(list);
        result.remove(list.size() - 1);
        return result;
    }
    return new ArrayList<>();
}
```

In these experiments I have established as base parameters:

-500 instances.

-42 items.

-250 maximum weight.

-2500 maximum cost.

-0.8 capacity to the total weight ratio.

In each individual study I will vary one parameter of the previous ones, and depending on the results, so that they are better appreciated, I may modify another separate parameter that I will also indicate.

ROBUSTNESS OF THE ALGORITHMS

Result quality

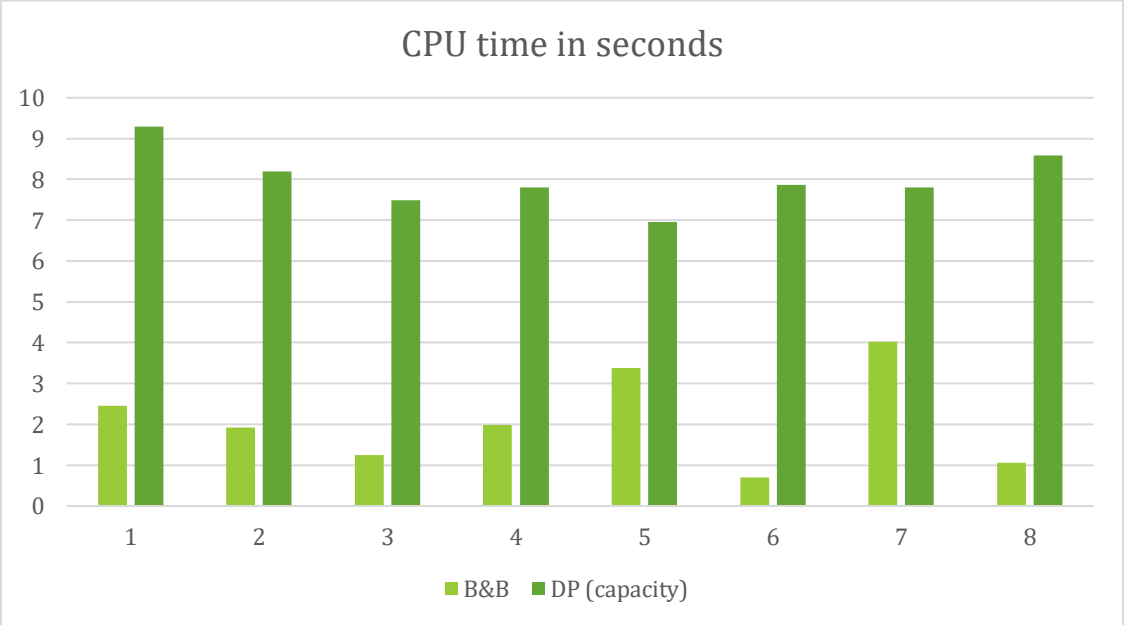
For c/w heuristic the average and maximum error for various permutations is 0.02311 and 0.02311.

This implies that he results are always the same for any permutation, since the c/w heuristic algorithm orders the items by itself, so it is very difficult for the result to change. A change could occur if two elements have the exact same c/w and in different permutations they go in different order. This is very difficult to happen and if it occurs it must also happen that in the optimal solution of the algorithm only one of the two is added, since if neither is ever added or if both are always added then the result would not change.

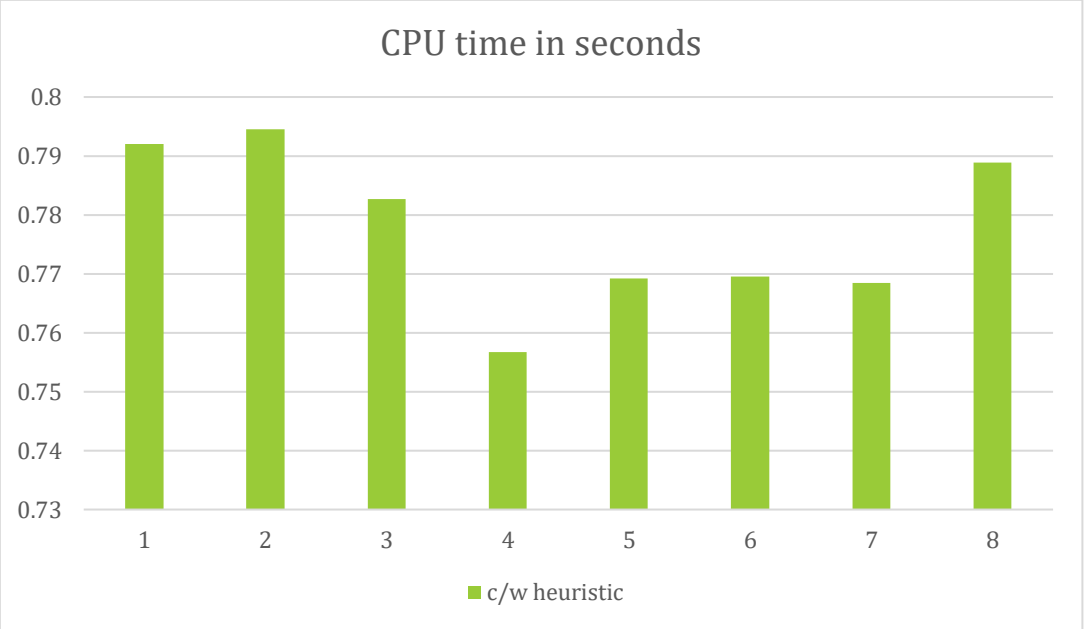
Computational complexity

These are the results of 8 random permutations of the same initial instance with the previously established parameters. For each instance, I have calculated the CPU time in seconds to compute the same instance 500 times to obtain a temporal value for better analysis. For the branch and bound algorithm, we observe its complete dependence on the order of items in the instances, ranging from 0.7 seconds to almost 4 seconds, more than 5 times worse than the best time. This occurs because the pruning process is heavily reliant on the order in which we traverse the items, leading to a drastic change in execution.

On the other hand, for dynamic programming decomposition by capacity, the order of items does not impact the algorithm as much compared to B&B, as it consistently performs the same number of calculations. This stability arises because neither the number of items nor the maximum capacities have been modified during the permutations.



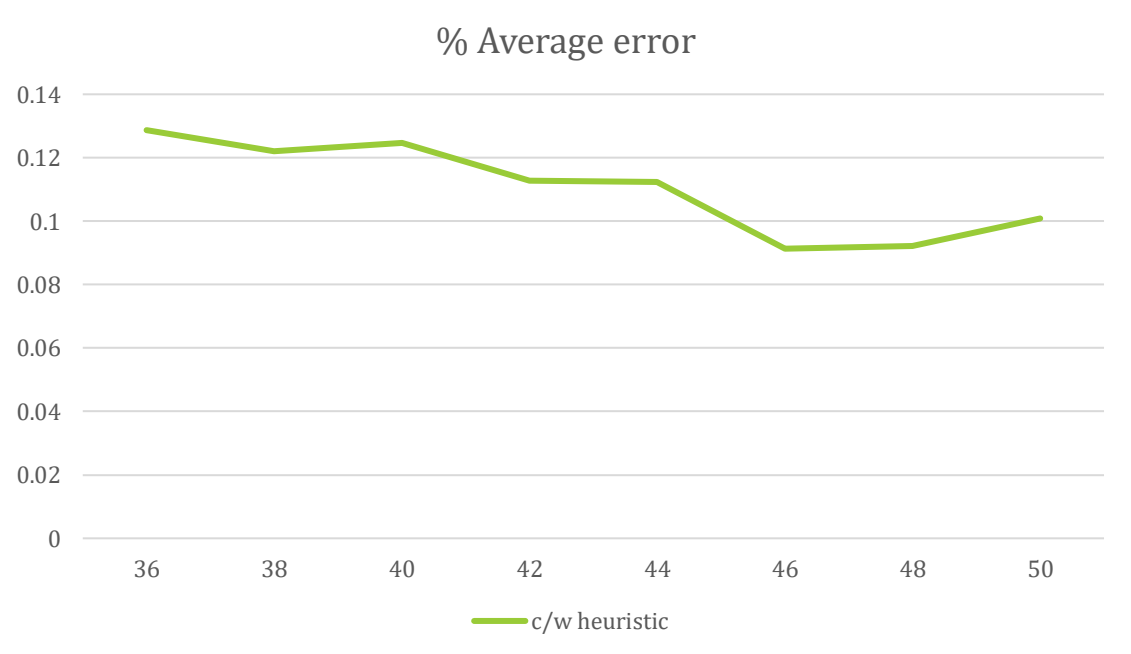
For the c/w heuristic algorithm, instead of 500 times, I have repeated the calculation of the same instance 30,000 times. The only variation in execution time comes from the algorithm's sorting process for the 42 items in different permutations. Surprisingly, this sorting time doesn't seem to have significant relevance, as the execution times are consistently similar across all permutations.



INSTANCE SIZE DEPENDENCE

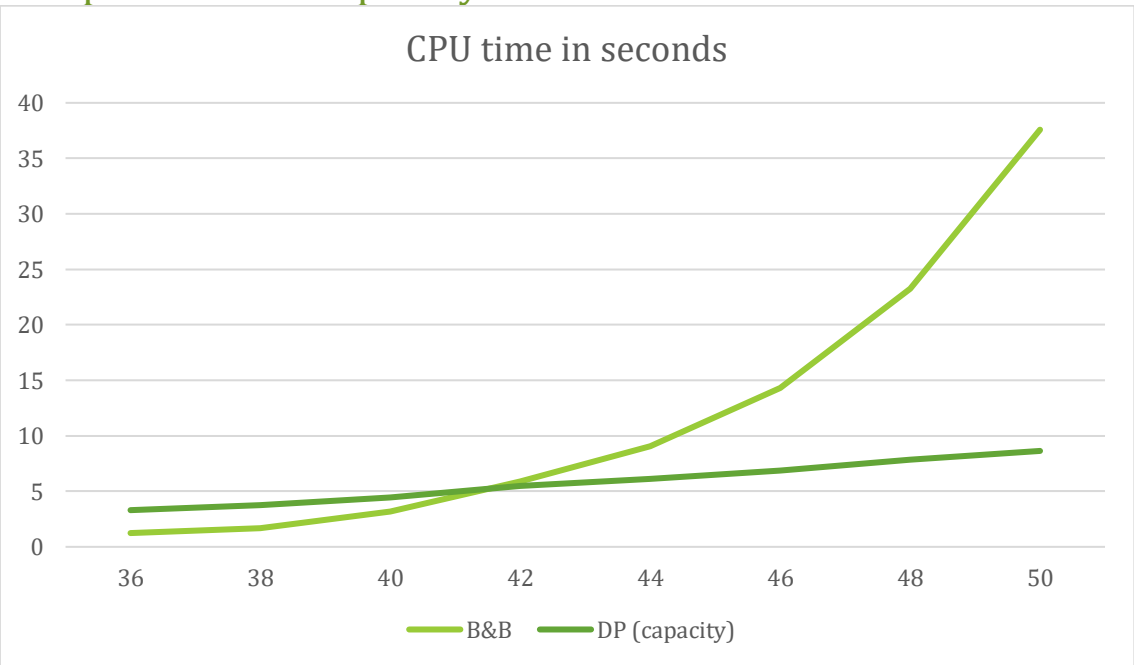
The results of the relative errors and CPU times have been calculated for the groups of instances varying the number of items between 36 and 50.

Result quality

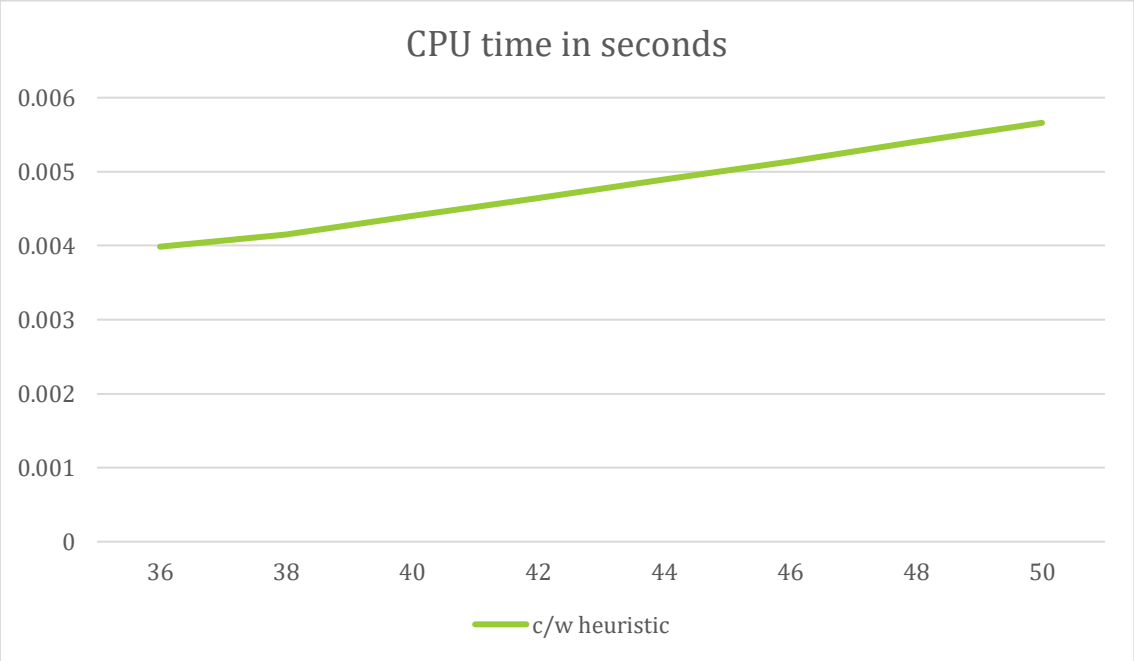


We observe that the results for the c/w heuristic algorithm do not exhibit any discernible trend, aligning with our expectations. This is because the ratio of capacity to the total weight remains unchanged, consistently accommodating a similar proportion of items in the knapsack. Consequently, this leads to relatively consistent relative errors among different permutations, as the heuristic tends to maintain a stable distribution of items in the knapsack without altering the underlying capacity-to-weight relationship.

Computational complexity



Examining the CPU time in seconds for a comparison between Branch and Bound (B&B) and Dynamic Programming (DP), we reaffirm observations from previous homework assignments. For B&B, as the instance size increases, the computational complexity grows exponentially. In contrast, Dynamic Programming, employing a decomposition by capacity, exhibits pseudopolynomial complexity with respect to the number of items (O(nM)), offering a more manageable growth pattern as the instance size expands.

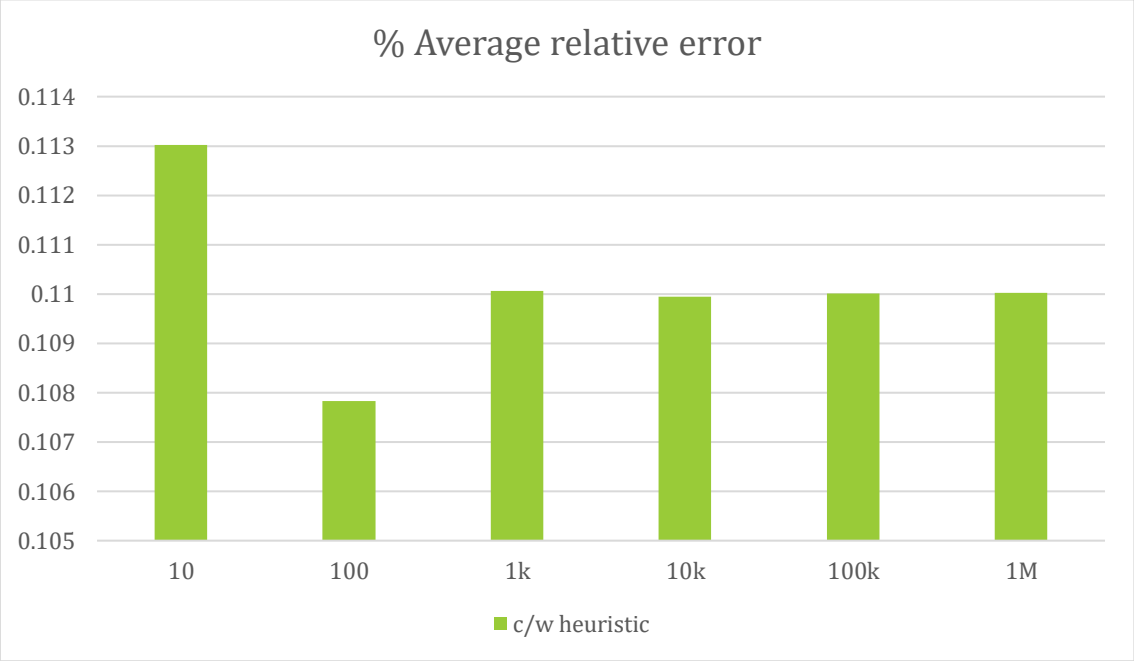


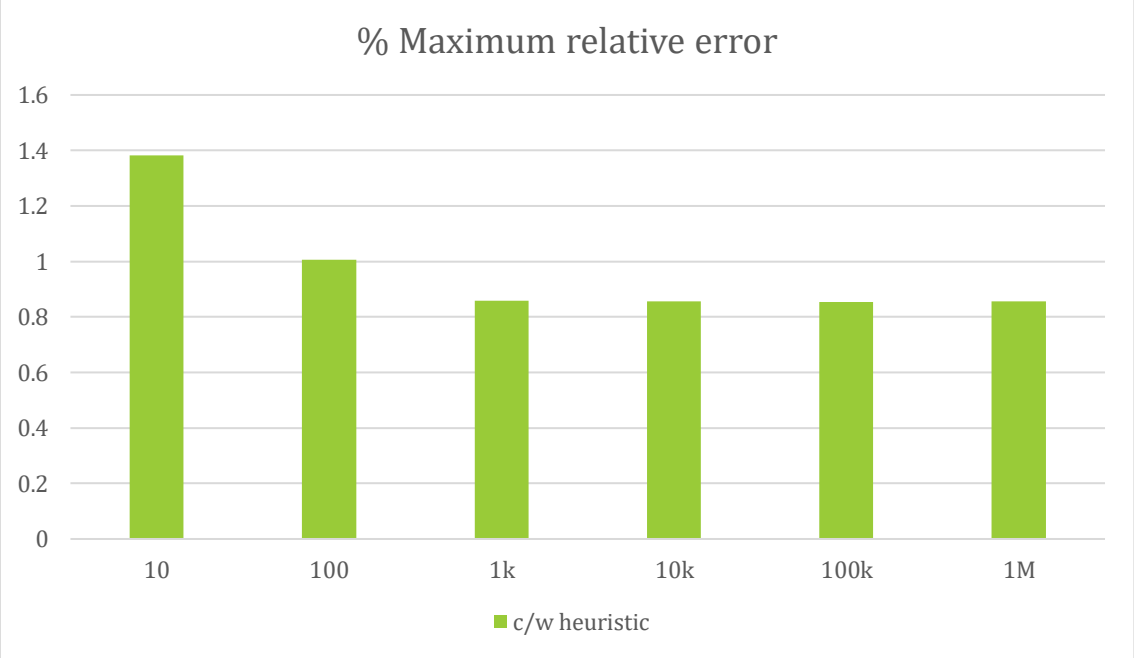
For the c/w heuristic algorithm, which also demonstrates polynomial complexity with respect to the number of items, its linear behavior is evident in the graph, aligning with our expectations. This linearity stems from the algorithm's reliance on cost-to-weight ratios, resulting in a predictable and consistent trend as the number of items increases.

MAXIMUM COST DEPENDENCE

For these experiments I have varied the maximum cost in powers of 10 only. I have opted for bar graphs since the x-axis differences are not linear.

Result quality

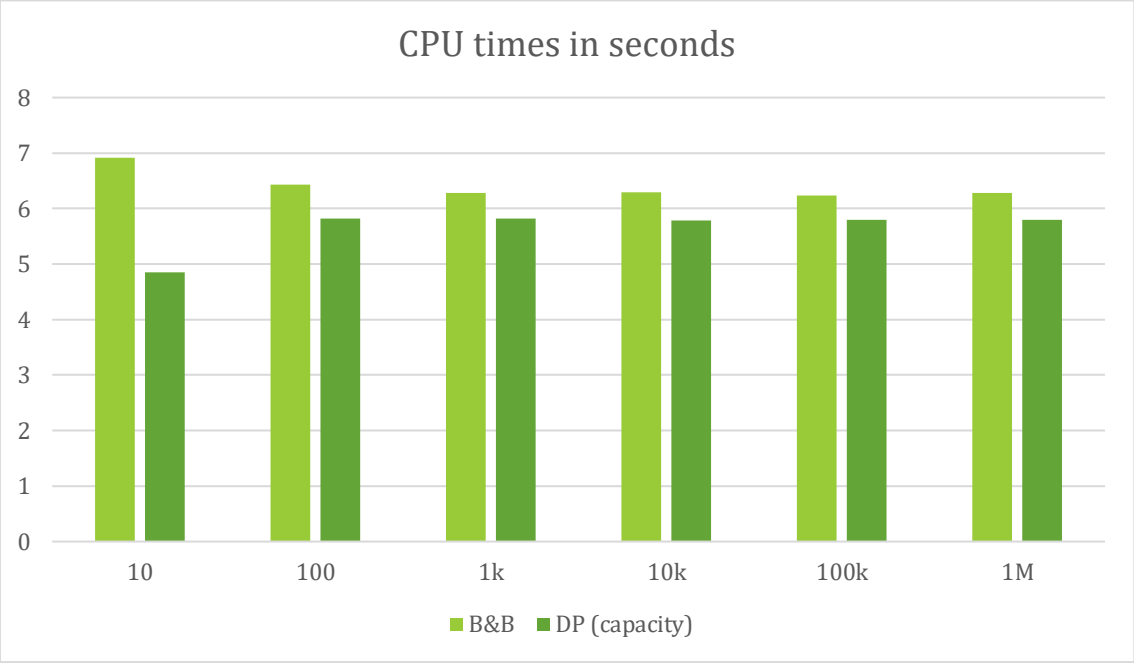


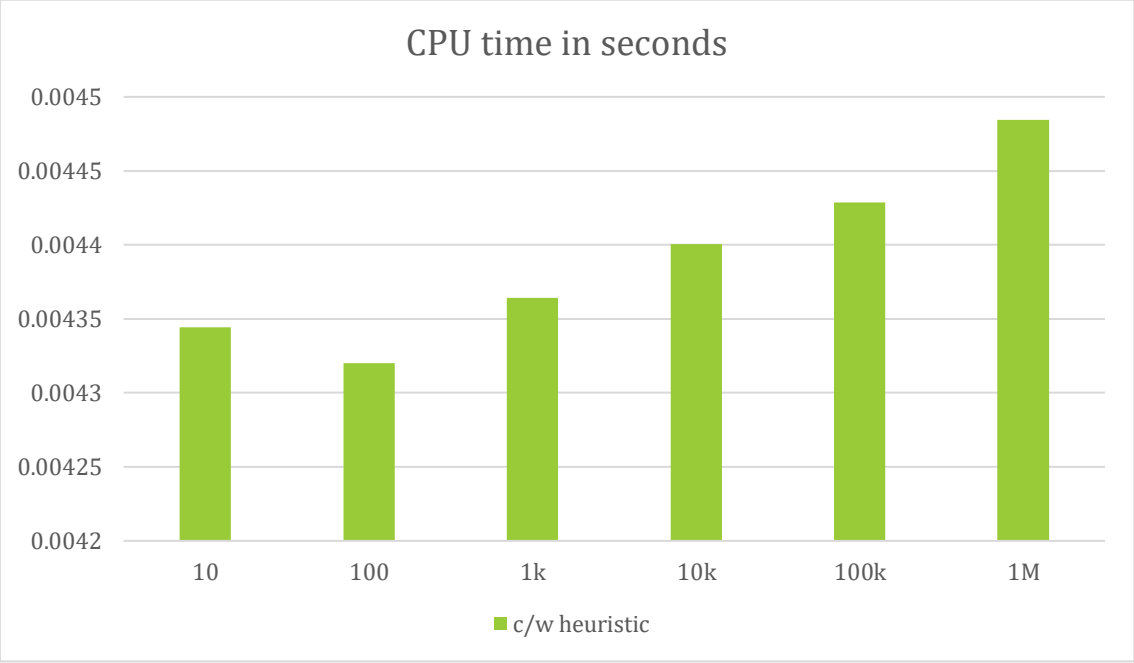


The effectiveness of this algorithm relies on the relationship between the cost and weight of the items and how it correlates with the knapsack's capacity. If there is a clear correlation, and items with a better cost-to-weight ratio are generally more valuable, the algorithm can provide acceptable solutions.

However, the algorithm's performance is not directly affected by the error in the maximum cost of the instances. The solution's quality will depend on the distribution of cost and weight values of the items and how they interact with the knapsack's capacity. This observation is supported by the graphical analysis, as no significant differences have been observed for the substantial variations made in the maximum cost.

Computational complexity





For the 3 algorithms it can be seen that there is no dependence on the computational complexity on the maximum cost. It seems that for c/w heuristic there is a trend but if we look at the values the differences between them are

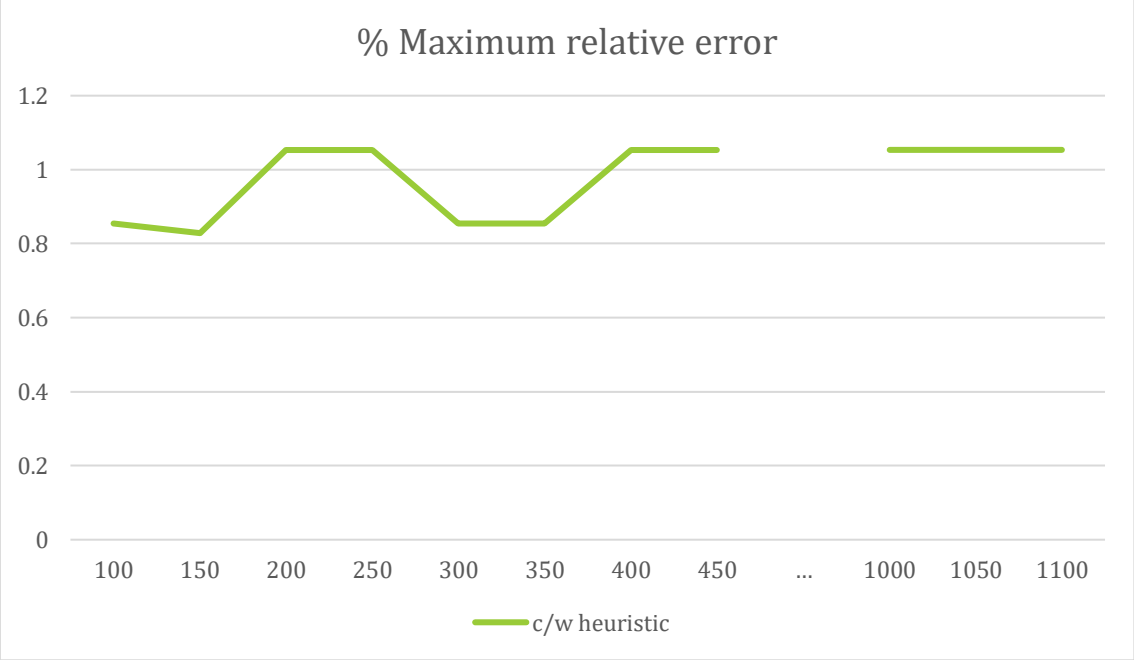
almost insignificant and could simply be due to the difficulty of the processor working with larger numbers each time.

MAXIMUM WEIGHT DEPENDENCE

For these experiments I have simply varied the maximum weight between 100 and 450 and also between 1000 and 1100 to see its trend in certain cases.

Result quality

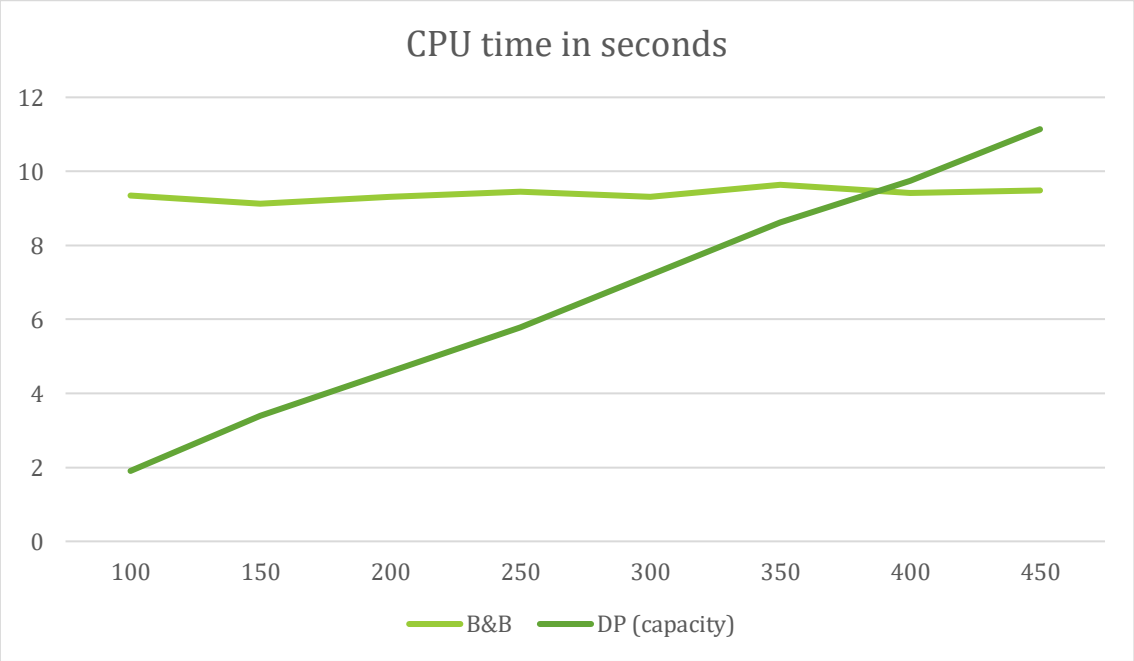
For the error in the c/w heuristic algorithm we see that there is no dependence on the maximum weight of the items.



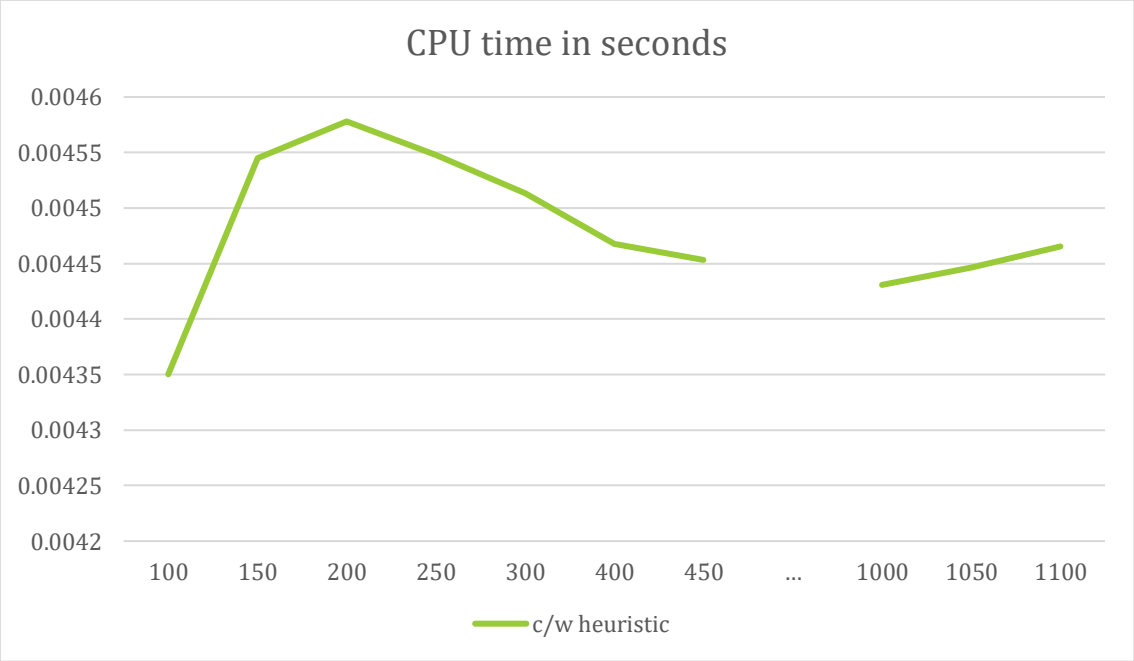
Computational complexity

For B&B we see that the computational complexity does not depend at all on the maximum weight of each item.

However, for dynamic programming by capacity it depends linearly on the capacity of the backpack, which is increased when we increase the maximum weight of the items. This happens since the ratio of capacity to the sum of the weights remains at 0.8, and if we increase the weights then the capacity will have to increase.



For the c/w heuristic algorithm we do not see any dependency in the graph, since we do not modify the number of items or anything similar.

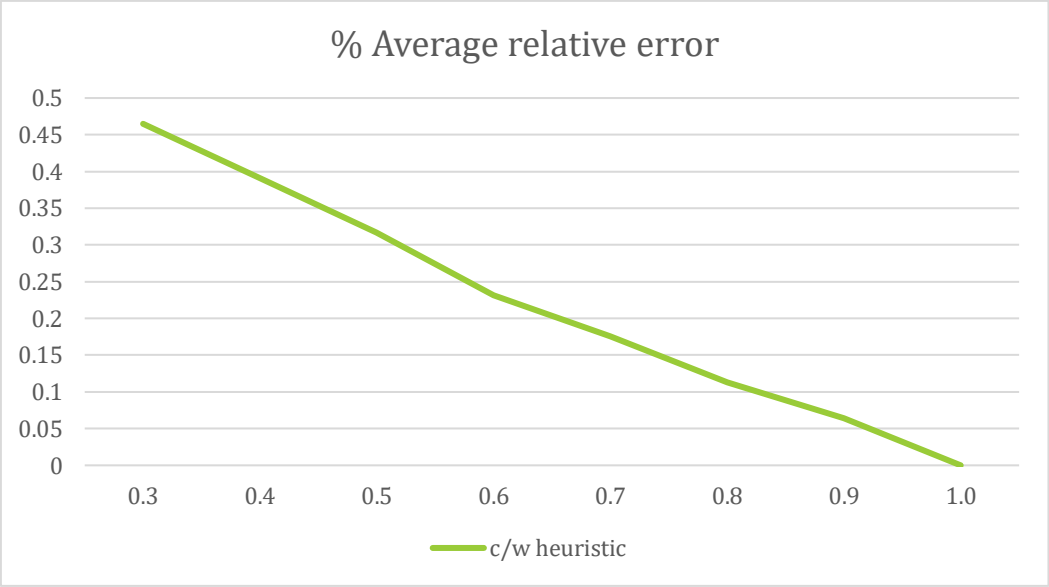


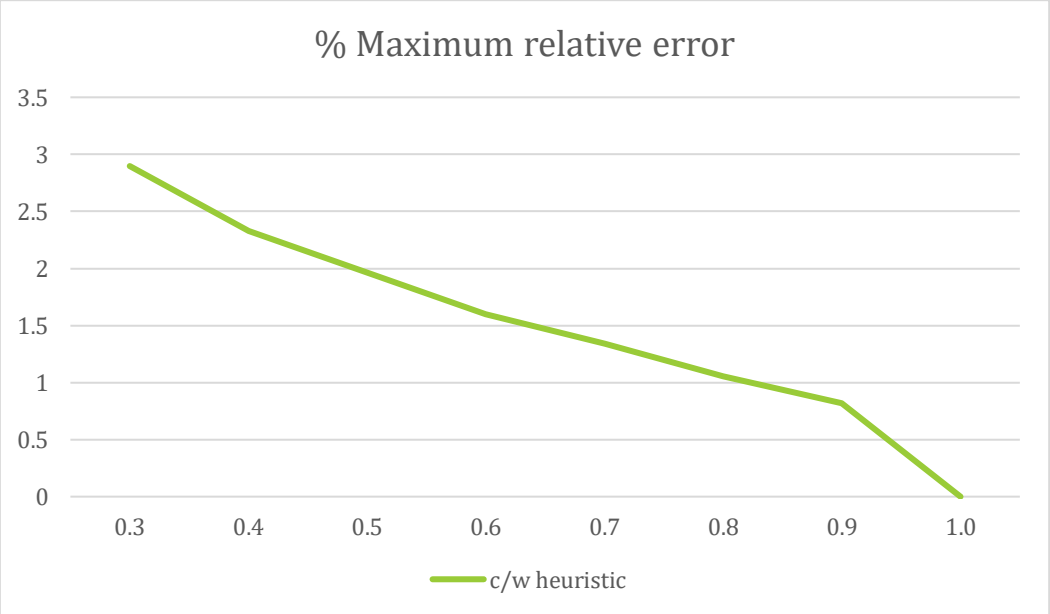
KNAPSACK CAPACITY TO THE TOTAL WEIGHT RATIO

For these experiments I have simply varied the capacity to the total weight ratio between 0.3 and 1.0.

Result quality

As we could suppose and we also see in the graph, the error for the ratio of 1 is 0. This is because all the items always fit in the knapsack, so the solution is always the same and does not give rise to error. But when the ratio goes down the knapsack capacity is reduced relative to the total weight of the items, the C/W heuristic algorithm may face challenges in optimally selecting items. This is because the heuristic prioritizes items with a better cost-to-weight ratio, and if the knapsack capacity is severely limited, it might lead to excluding valuable items due to their higher weight. That is why we observe the linear increase in the error when we decrease the ratio.



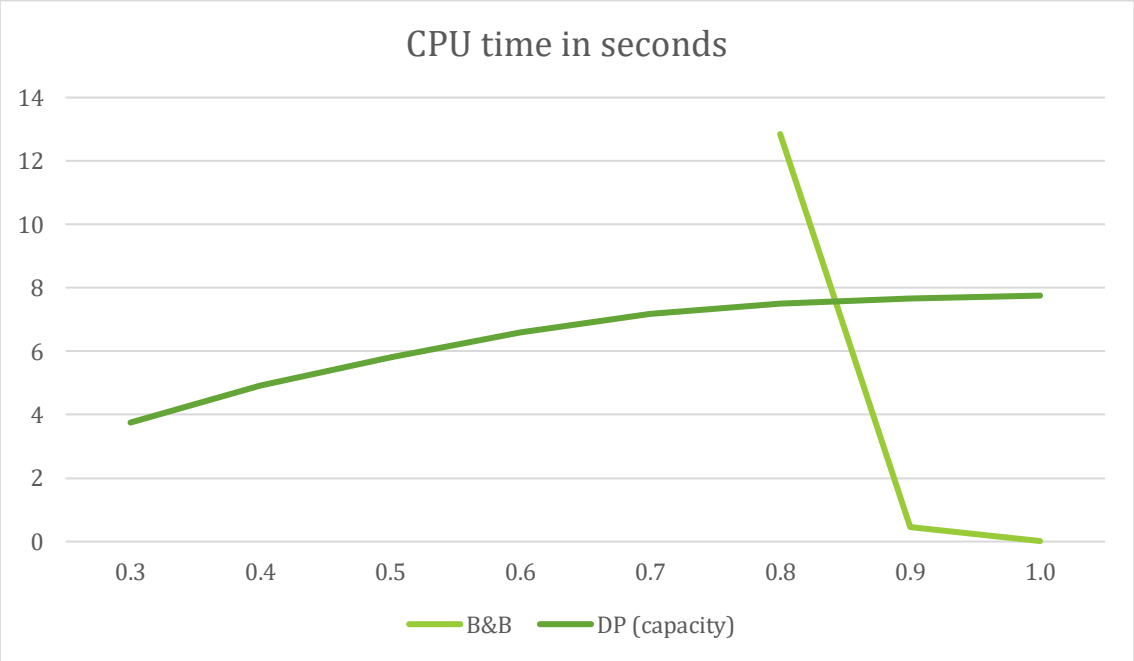


Computational complexity

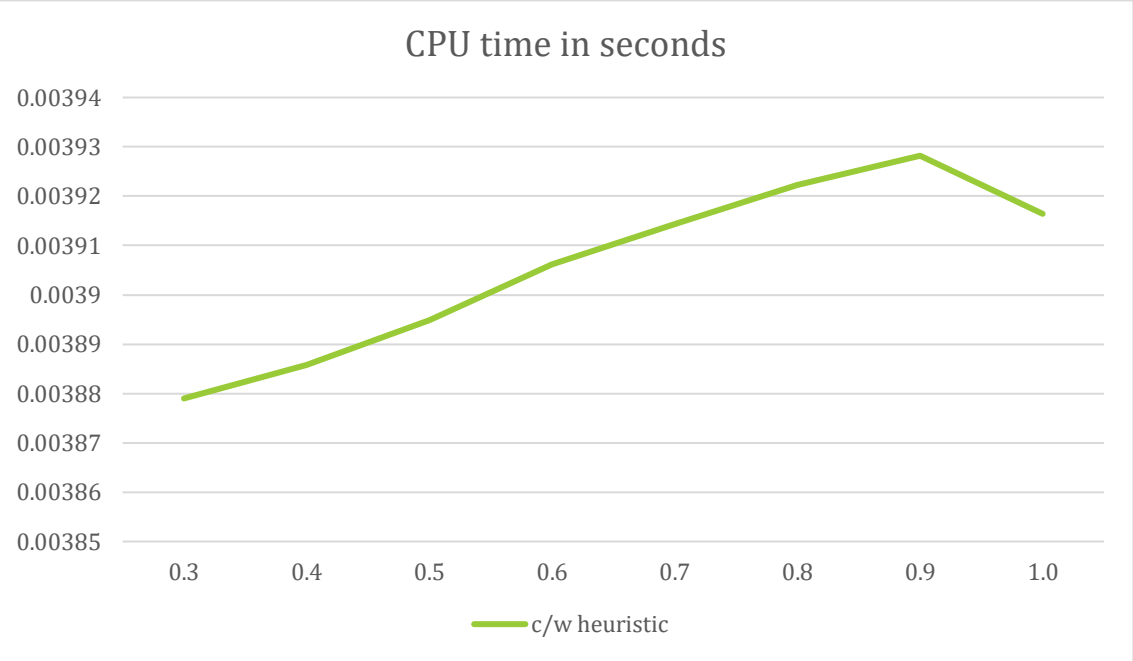
As we can see in the graph for B&B, when we decrease the ratio, the execution time considerably worsens, reaching 350 CPU seconds when the ratio is at 0.7, for example.

Reducing the ratio of knapsack capacity to total item weight in the Branch and Bound algorithm significantly worsens execution time due to a larger and more complex search space, resulting in increased tree depth and node generation. The expanded search space leads to longer paths in the search tree, generating more nodes and increasing computational overhead. The problem becomes more challenging, impacting algorithm performance as it seeks optimal solutions.

However, for dynamic programming by capacity the opposite happens, since by increasing the ratio we are indirectly increasing the total capacity of the knapsack, which causes the complexity to increase.



For the c/w heuristic algorithm, as we can see, the issue of computational complexity is not affected significantly by increasing or decreasing the capacity to the total weight ratio. Since nothing in the algorithm changes, simply fewer items will be selected for the knapsack.

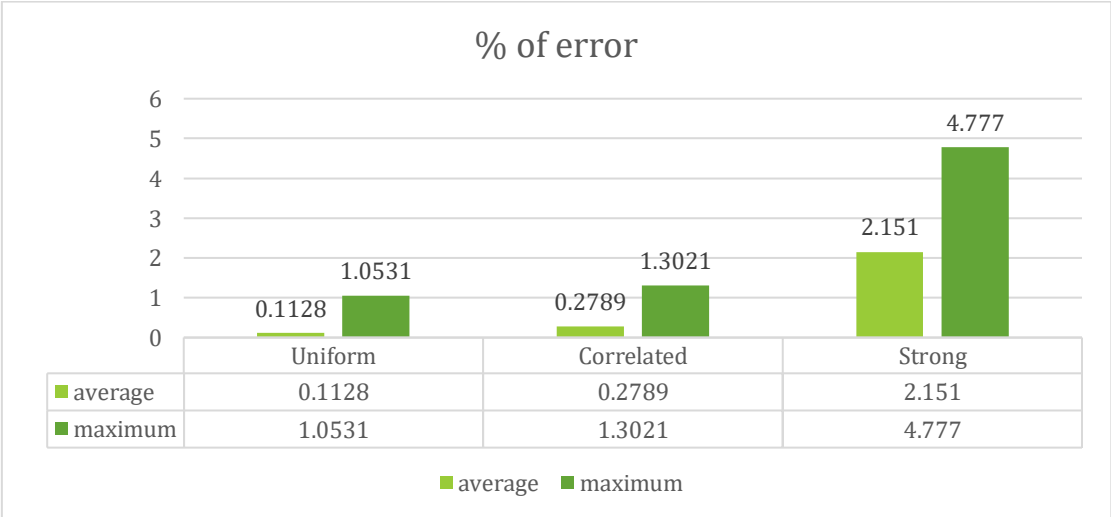


COST/WEIGHT CORRELATION

For these experiments I have varied the correlation and, where indicated, the number of items.

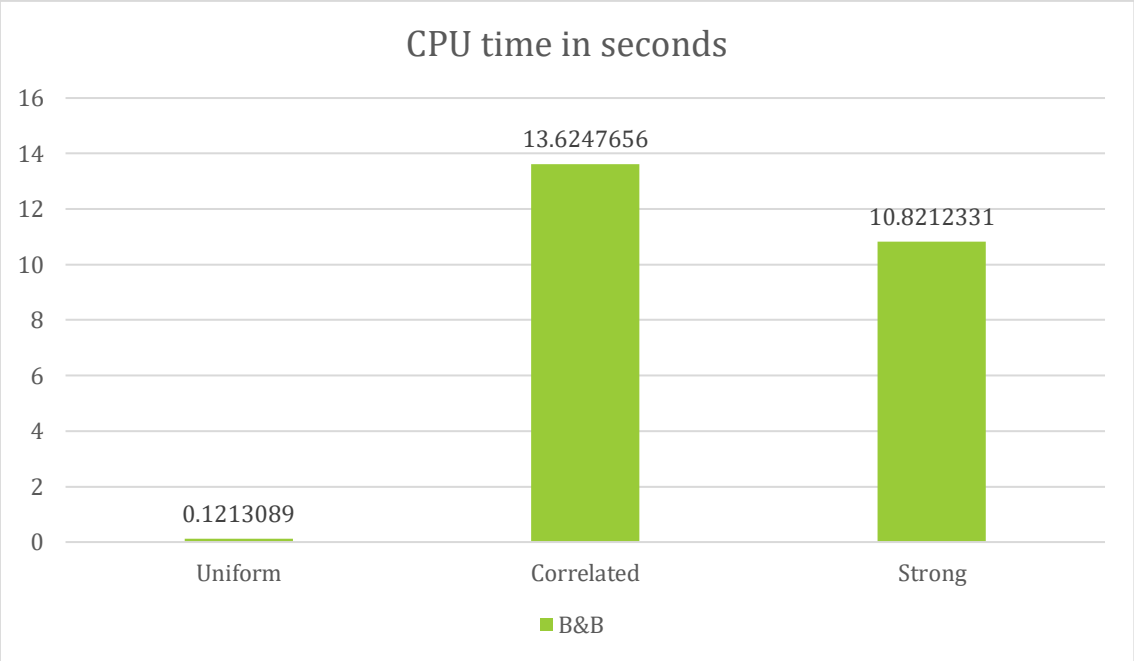
Result quality

When we increase the cost/weight correlation in the c/w heuristic algorithm, the error tends to increase, as we can see in the graph, due to a greater influence of the element order on the resulting solution. The c/w heuristic prioritizes element selection based on their cost-to-weight ratio, and an increase in correlation means that the relative arrangement of elements becomes.



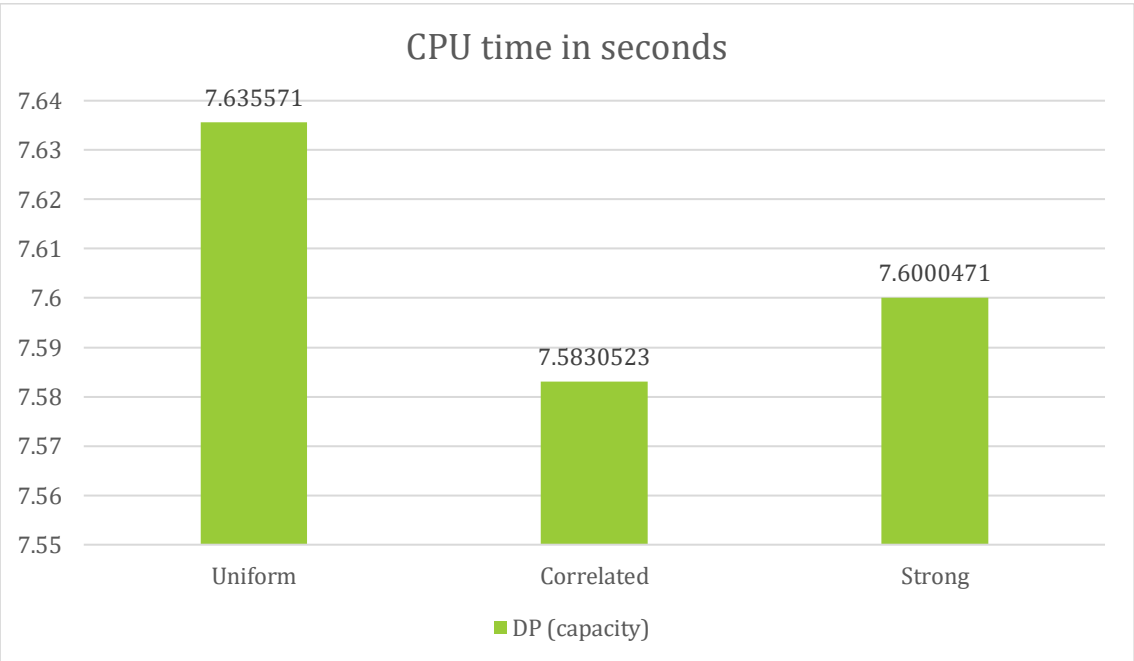
Computational complexity

For 25 items:

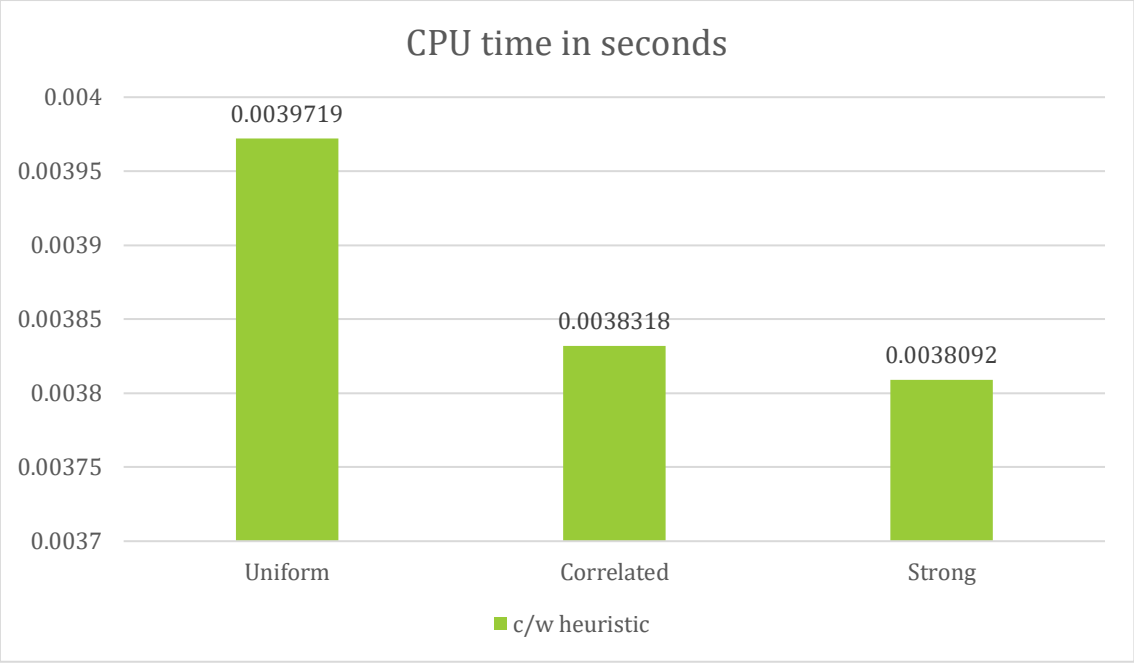


As we can see when we increase the cost/weight correlation in the Branch and Bound algorithm, the computational complexity tends to increase because the dependence between cost and weight affects the efficiency of pruning and elevates the complexity of the search space. Higher correlation implies increased sensitivity to the order of elements, resulting in longer paths in the search tree and more nodes that need to be explored before reaching an optimal solution. This leads to an overall increase in runtime and problem complexity.

For 42 items:



The dynamic programming decomposition by capacity remains resilient to cost/weight correlation because it efficiently explores all relevant subproblems based on capacity, leading to a more consistent performance across different correlation scenarios.



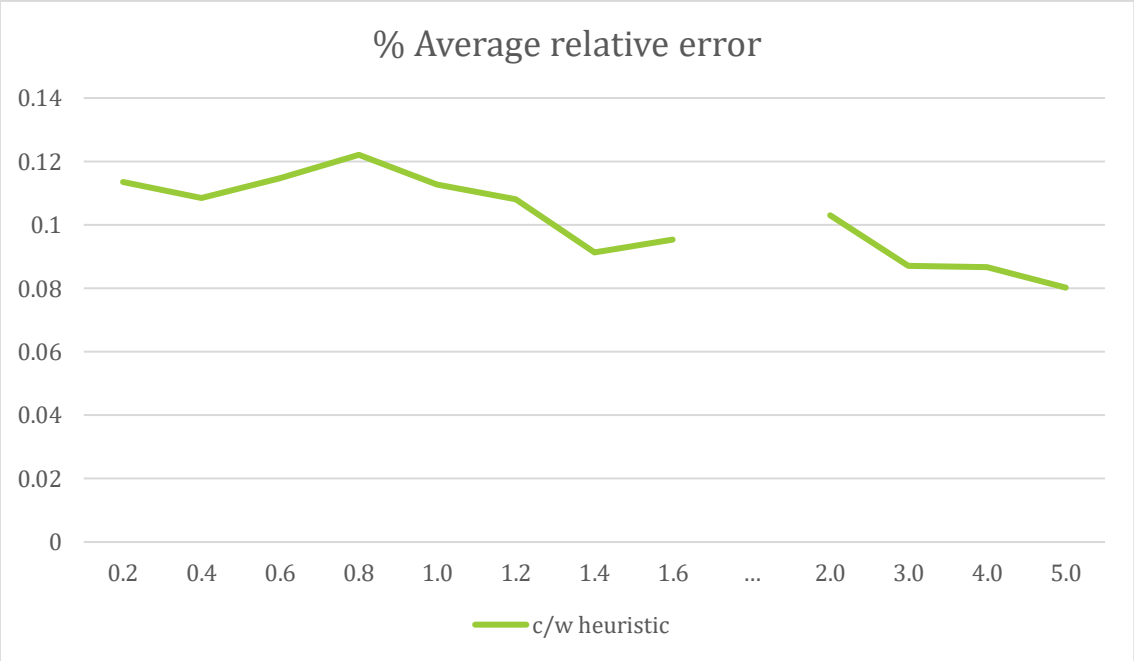
The computational complexity of the c/w heuristic algorithm is also not affected by said correlation due to its simplicity and reliance on sorting items based on their cost-to-weight ratio.

GRANULARITY

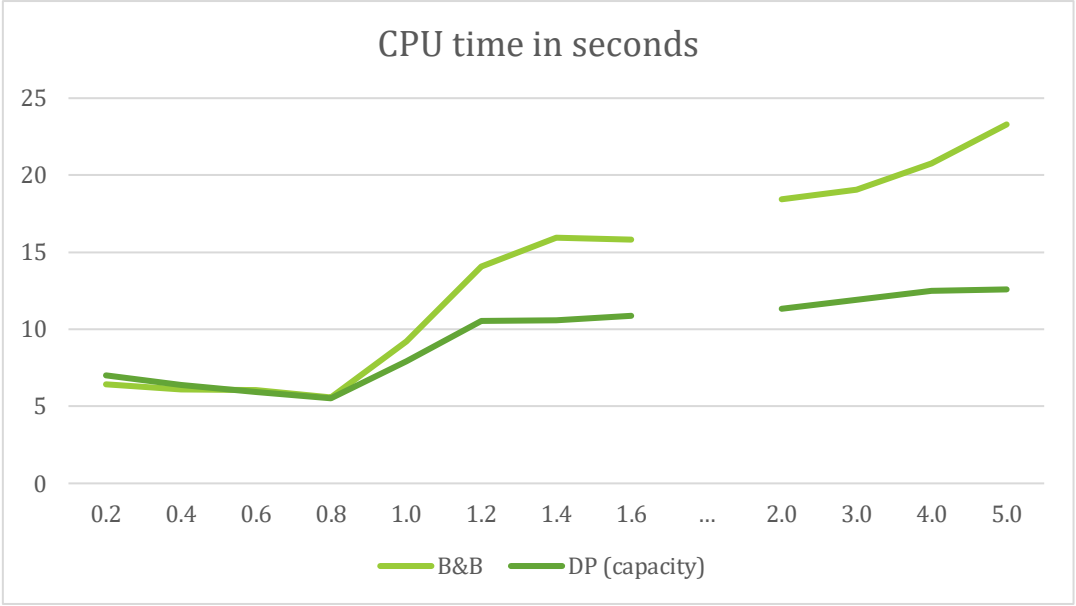
For these experiments I have varied the granularity exponent between 0.2 and 1.6 and then to much larger values such as 2, 3, 4 or 5 to better see the behavior of the algorithms.

Result quality

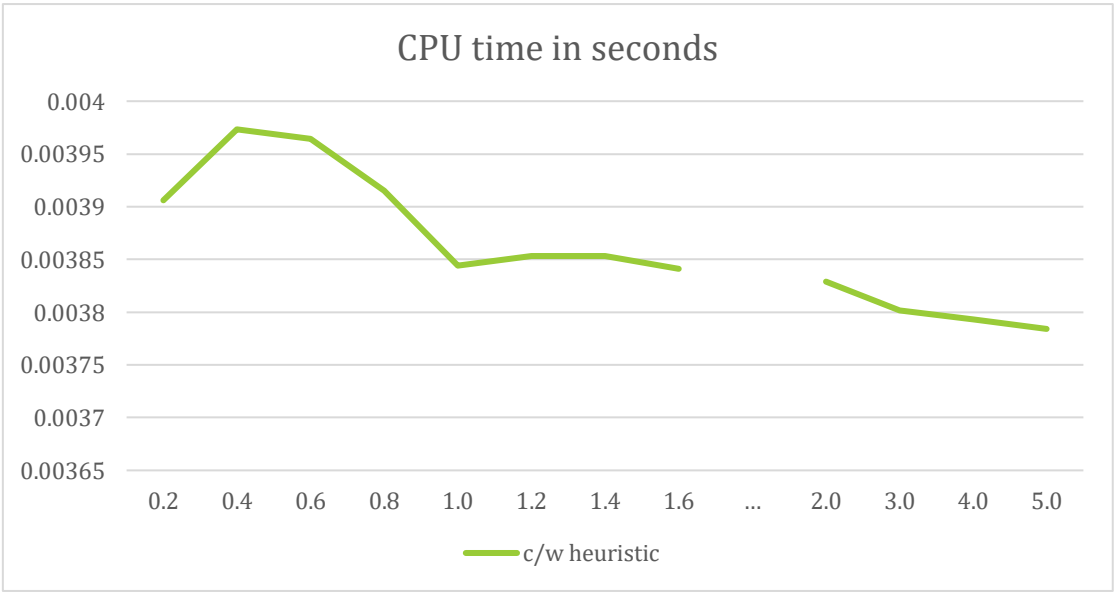
As we can see for the c/w heuristic algorithm it does not have a clear dependence on the granularity. Since the c/w heuristic is based on the cost-weight ratio of elements, variation in granularity could affect the precision with which these ratios are calculated, but not necessarily the final result in terms of error.



Computational complexity



When we increase the granularity in the Branch and Bound (B&B) and Dynamic Programming Decomposition by Capacity algorithms, the computational complexity tends to increase as we can see. This is because greater granularity implies a greater number of subproblems or nodes in the case of Branch and Bound, and greater resolution of subproblems in the case of Dynamic Programming. These factors result in an increase in the execution time and overall computational complexity of the algorithms.



However, the execution time of the c/w heuristic algorithm is not affected by the variation of the granularity since neither the ordering as we have said before nor the selection of items changes.

CONCLUSIONS

When we talk about robustness, DP is better since it is much more stable than B&B.

Varying the number of items for small quantities it is better to use B&B but when we increase the difficulty a little, DP is much more effective.

For the dependence on the maximum cost we have hardly observed any variations in the 3 algorithms.

When we talk about varying the maximum weight, it happens the opposite of what happened to us when we varied the number of items. This time is better for lower weights DP and for higher weights B&B.

When we vary the ratio of the capacity with the weights and the c/w correlation, the behavior of DP is better due to its greater stability.

Finally for granularity, the truth is that the 3 algorithms behave in a similar way.

For the c/ heuristic algorithm we could say that the error varies more with the ratio of the capacity to the weights. For the computational complexity there are more changes when we vary the number of items in the knapsack.

The brute force algorithm has not been studied since all the time complexity results would be the same since my implementation always goes through all the possibilities.