HOMEWORK 1 – SOLVING THE DECISION KNAPSACK PROBLEM EXACTLY

BY ANDRÉS RUBIO RAMOS

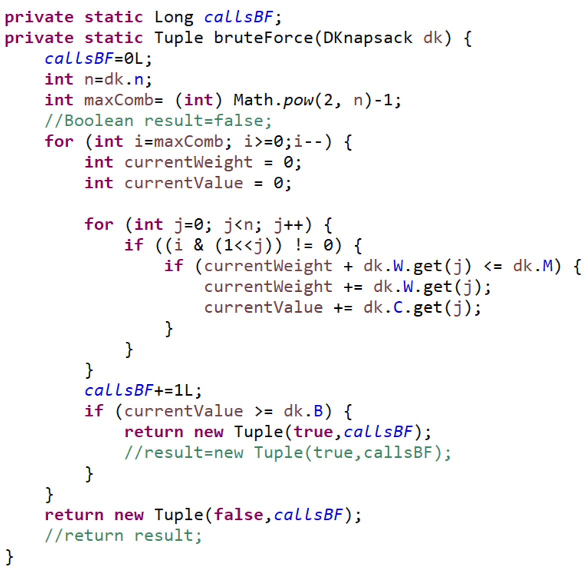
I have added a very good prune by cost to the Branch and Bound algorithm. And in the Brute Force now instead of starting to go through the chosen items 0000, 0001, 0010... I started with 1111, 1110, 1101... since for this specific data the algorithm takes less time.

ALGORITHMS USED

Brute force

I have opted for an iterative solution since it seemed simpler for me. In my version, once a correct configuration is found, the algorithm ends and returns true.

If you wanted to check all the configurations, I have commented what should be changed.



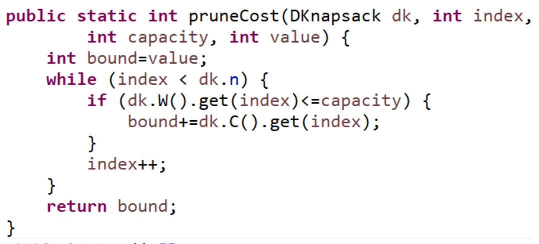
```java
private static Long callsBF;
private static Tuple bruteForce(DKnapsack dk) {
    callsBF=0L;
    int n=dk.n;
    int maxComb= (int) Math.pow(2, n)-1;
    //Boolean result=false;
    for (int i=maxComb; i>=0;i--) {
        int currentWeight = 0;
        int currentValue = 0;

        for (int j=0; j<n; j++) {
            if ((i & (1<<j)) != 0) {
                if (currentWeight + dk.W.get(j) <= dk.M) {
                    currentWeight += dk.W.get(j);
                    currentValue += dk.C.get(j);
                }
            }
        }
        callsBF+=1L;
        if (currentValue >= dk.B) {
            return new Tuple(true,callsBF);
            //result=new Tuple(true,callsBF);
        }
    }
    return new Tuple(false,callsBF);
    //return result;
}
```

Branch & Bound method

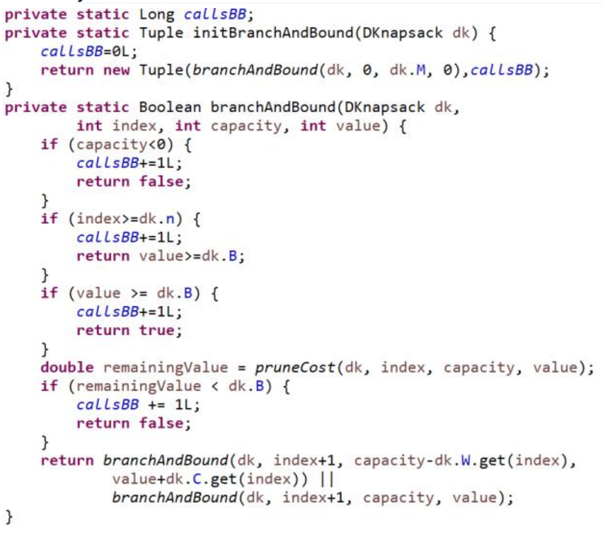
For this method I have implemented a recursive algorithm that consists of exploring the binary tree deciding whether or not to take the next object by its index.

I start with all the available capacity of the knapsack and as I go through the tree I substract the weights of the objects and if I reach negative capacity I discard that branch.

Thanks to the operator || the algorithm ends when it finds the first branch with sum value greater than B.



```java
public static int pruneCost(DKnapsack dk, int index,
        int capacity, int value) {
    int bound=value;
    while (index < dk.n) {
        if (dk.W().get(index)<=capacity) {
            bound+=dk.C().get(index);
        }
        index++;
    }
    return bound;
}
```



```java
private static Long callsBB;
private static Tuple initBranchAndBound(DKnapsack dk) {
    callsBB=0L;
    return new Tuple(branchAndBound(dk, 0, dk.M, 0),callsBB);
}
private static Boolean branchAndBound(DKnapsack dk,
        int index, int capacity, int value) {
    if (capacity<0) {
        callsBB+=1L;
        return false;
    }
    if (index>=dk.n) {
        callsBB+=1L;
        return value>=dk.B;
    }
    if (value >= dk.B) {
        callsBB+=1L;
        return true;
    }
    double remainingValue = pruneCost(dk, index, capacity, value);
    if (remainingValue < dk.B) {
        callsBB += 1L;
        return false;
    }
    return branchAndBound(dk, index+1, capacity-dk.W.get(index),
            value+dk.C.get(index)) ||
            branchAndBound(dk, index+1, capacity, value);
}
```

STUDY OF THE RESULTS

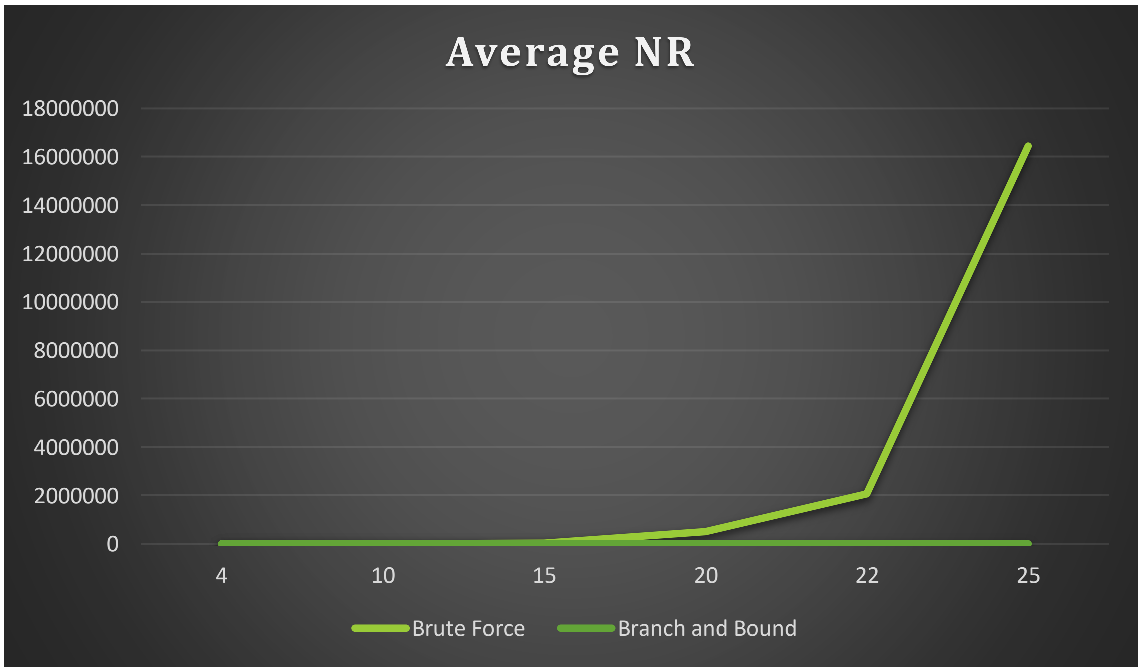
By using the "calls" variable that counts the number of calls to the iterative or recursive function we can get an idea of the computational complexity and we can compare the values depending on the number of items.

NR instances

For these instance I have run up to the number of items equal to 25 and these would be the results obtained with the average and maximum calls (For the average I have put the value of the floor function):

Items	Average BF	Maximum BF	Average B&B	Maximum B&B
4	7	16	1	6
10	502	1024	4	95
15	15937	32768	16	849
20	514515	1048576	98	10986
22	2055710	4194304	230	16779
25	16445873	33554432	1022	60101

Since cost pruning is so good for B&B, the difference with respect to BF is too high. which makes B&B look like a straight line on the graph, but in reality it has an exponential shape. The same happens with the differences between the maximums.

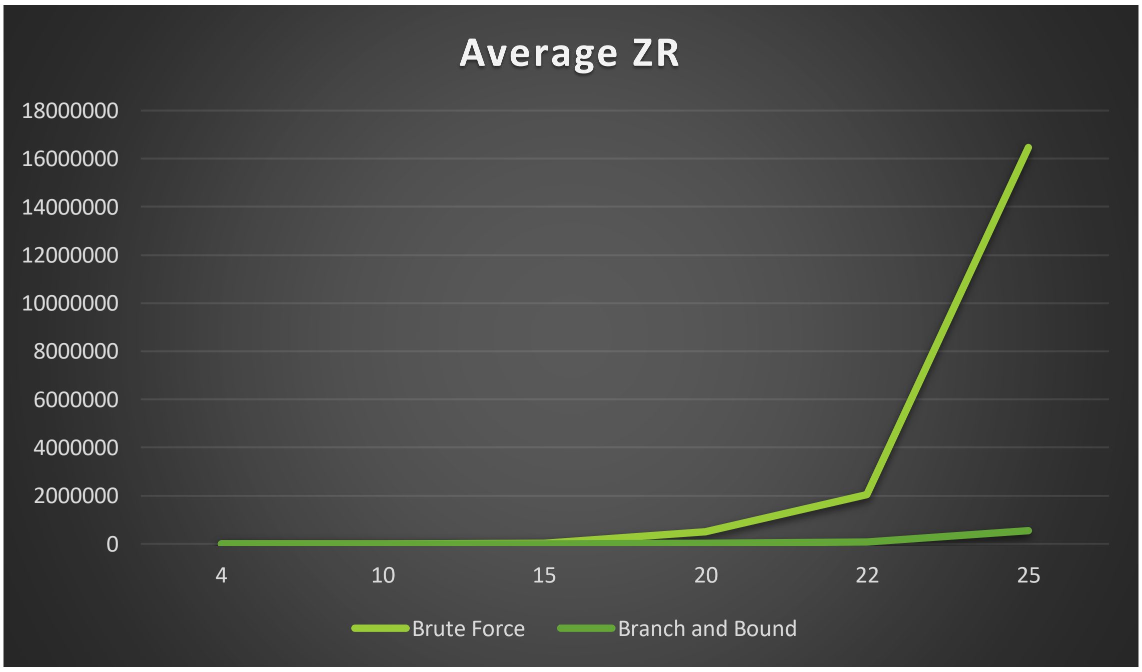


ZR instances

For these instance I have run up to the number of items equal to 25 and these would be the results obtained with the average and maximum calls:

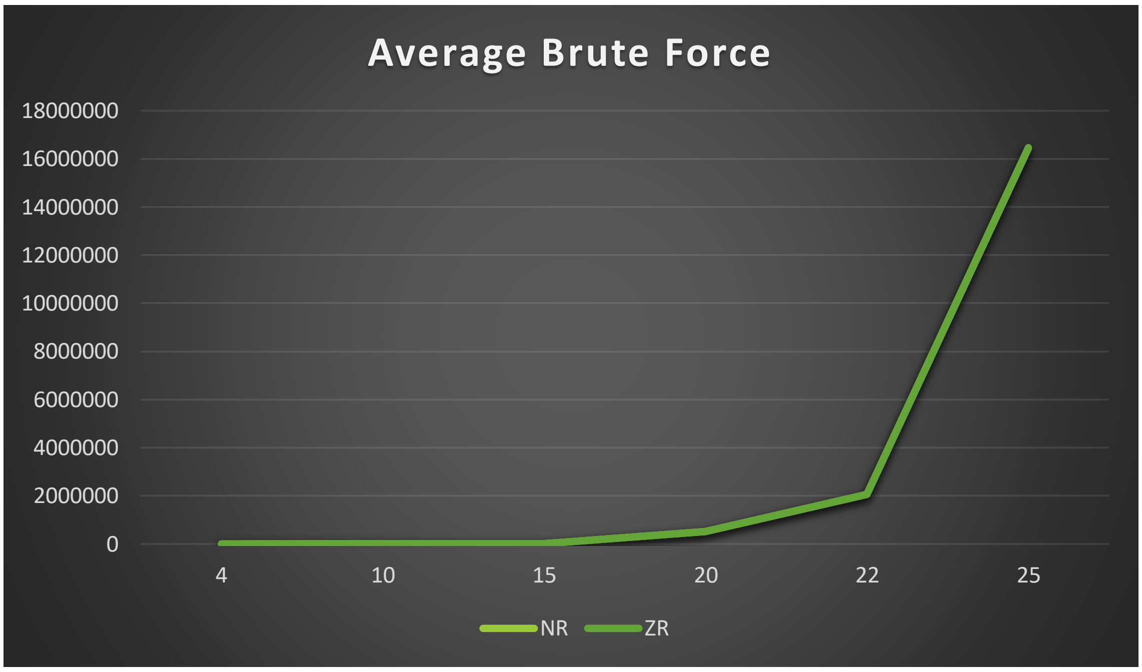
Items	Average BF	Maximum BF	Average B&B	Maximum B&B
4	8	16	3	8
10	508	1024	80	345
15	16130	32768	1391	6055
20	515091	1048576	29168	162101
22	2057802	4194304	89887	514755
25	16466752	33554432	548546	3375435

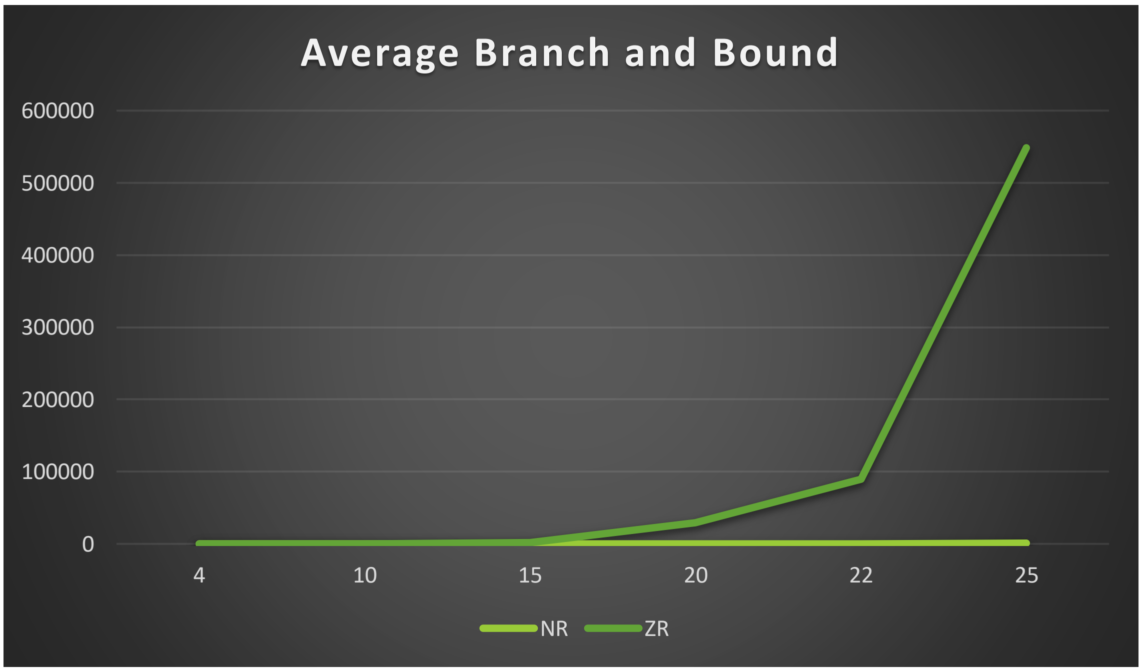
Here being on average some more "difficult" instances so yes for B&B it is not a simple straight line. The same happens with the differences between the maximums.



NR VS ZR instances

We are going to compare both instances in the same graph to see the differences between them.





As we can see, both have exponential behavior, as we could expect from these algorithms since its complexity is 2^n.

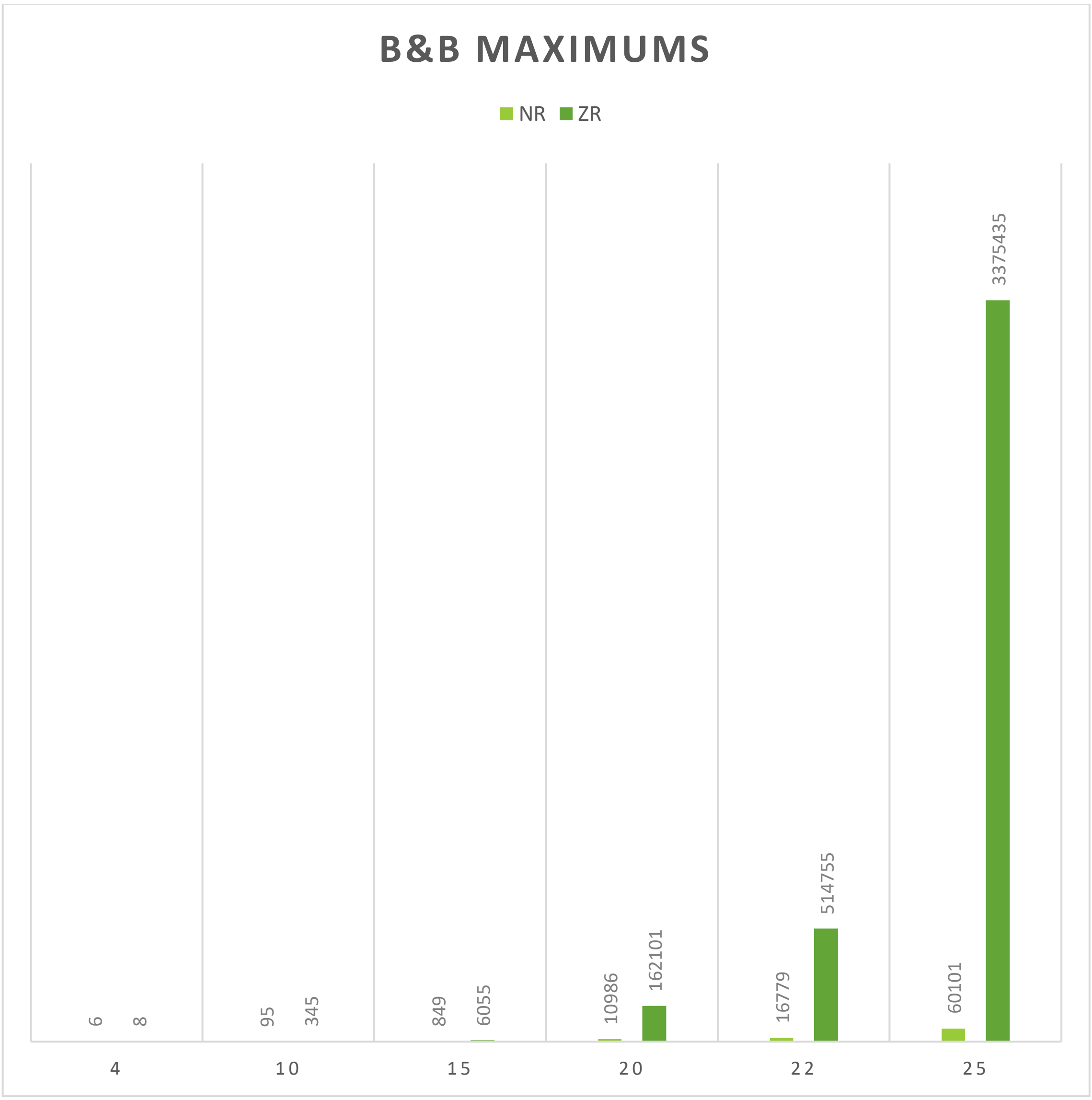
For brute force there is no difference, both results are always very high.

But for Branch and Bound due to the “worse” configuration of the ZR instances, the results in number of calls are higher as seen in the graph when comparing the averages since It cannot take advantage of pruning as much.

When we study the difference between the maxima:

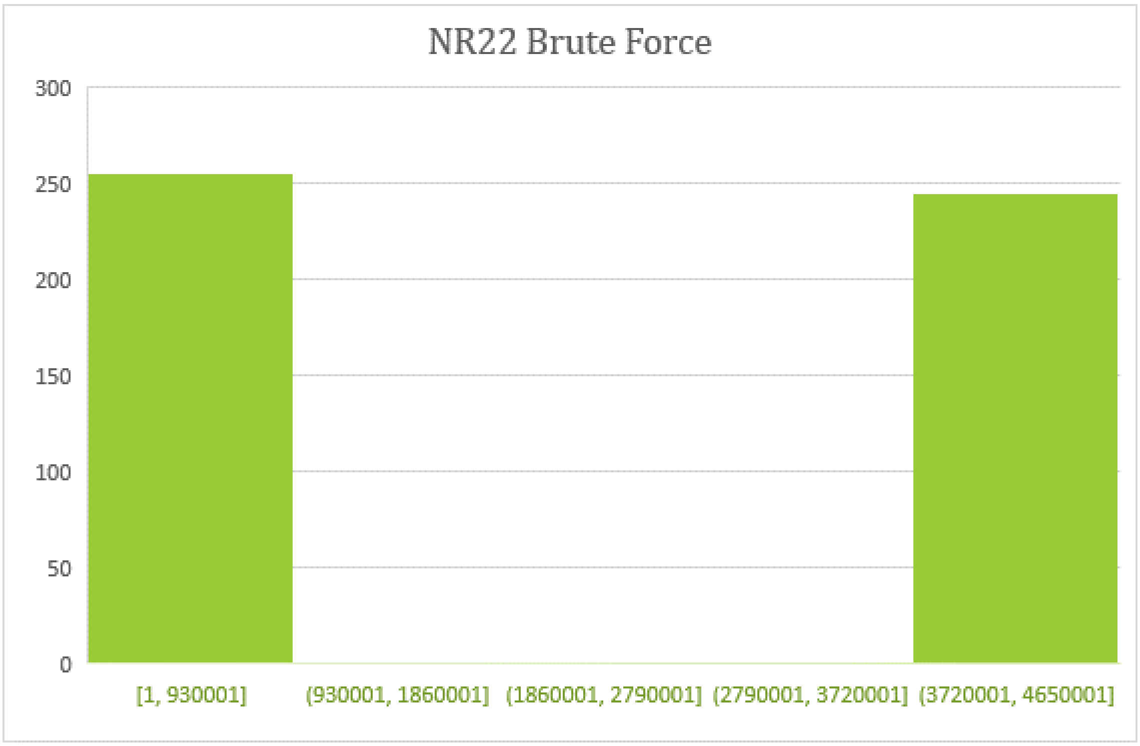
For Brute Force there is hardly any difference when we talk about the maximum calls that have had to be made among the 500 problems of each type. This may be because among the 500 problems in NR there are some that have solutions that go as deep as those in ZR.

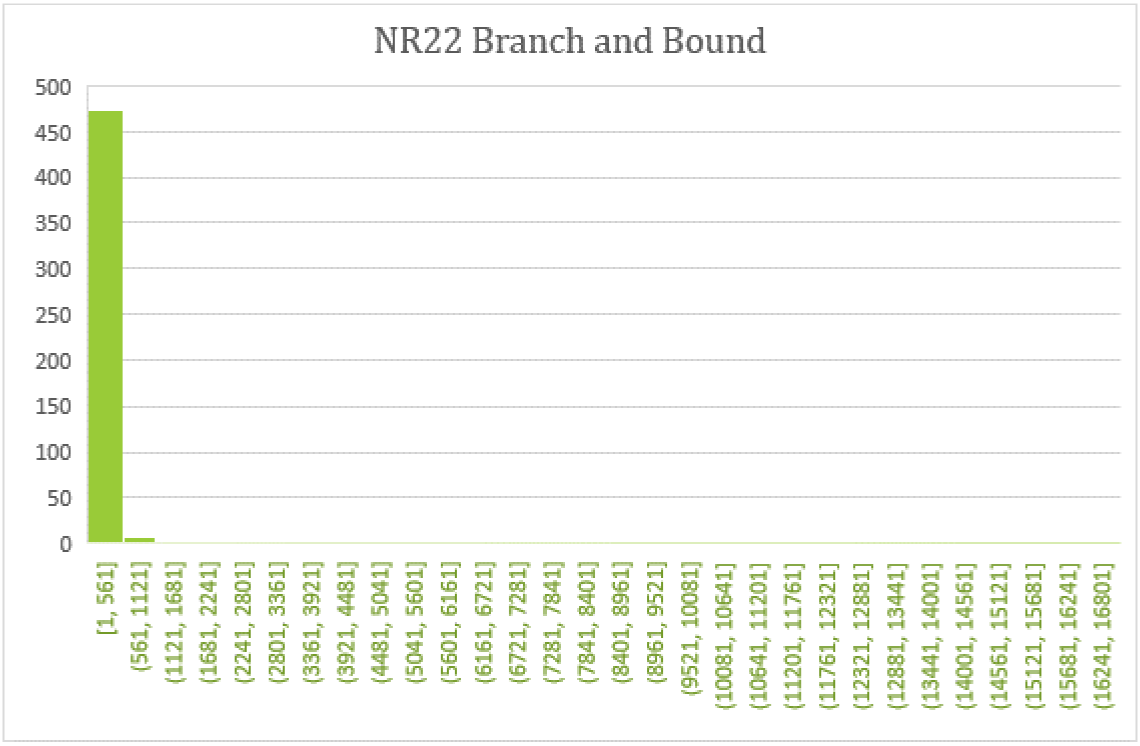
For B&B something similar happens to what has happened with the averages, the difference is very gigantic due to the different possibilities of pruning or not.



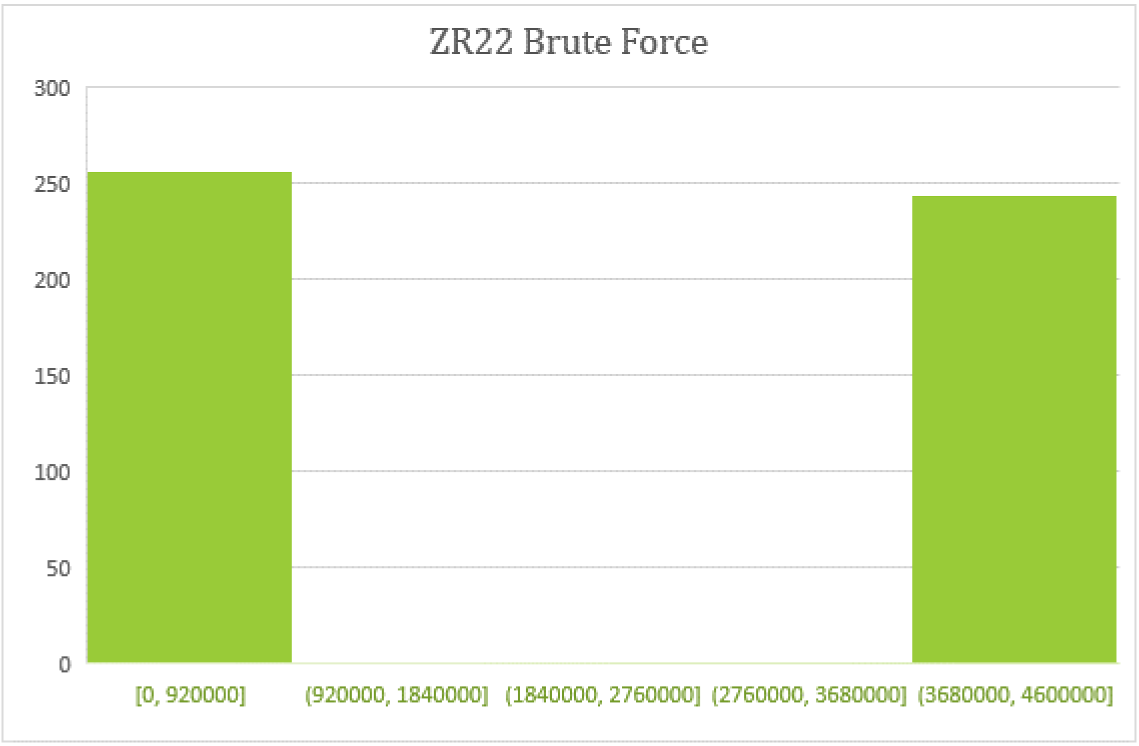
Frequencies for 22 items

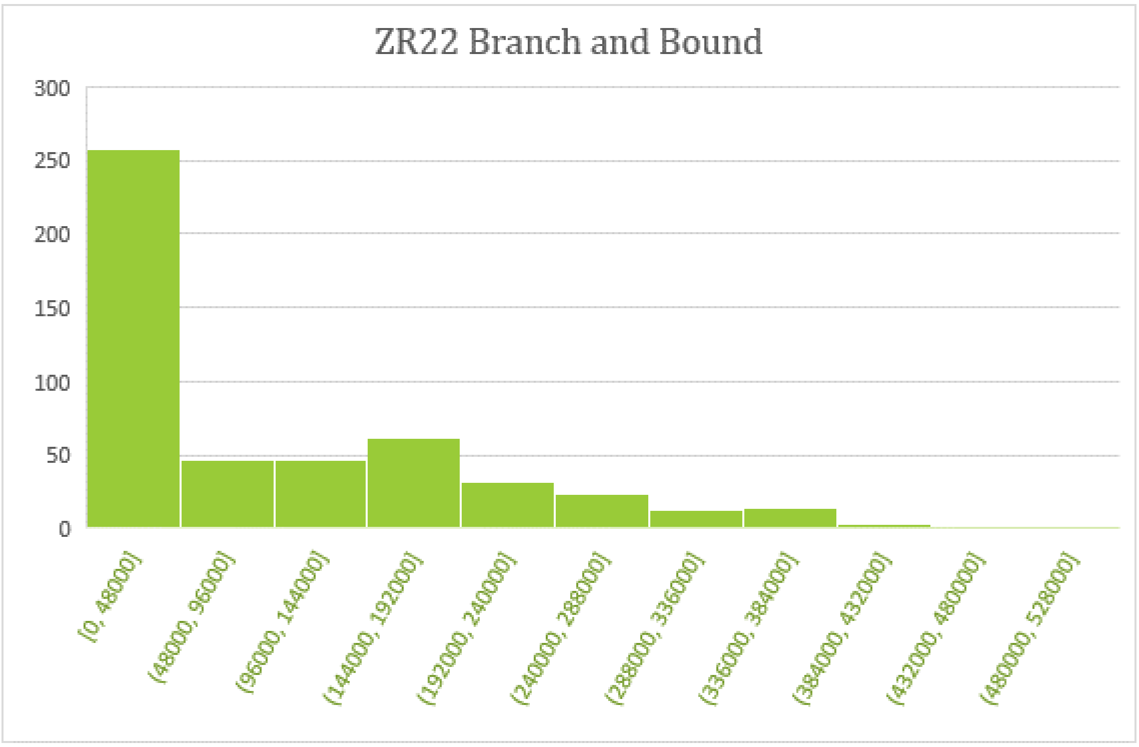
NR instances





ZR instances





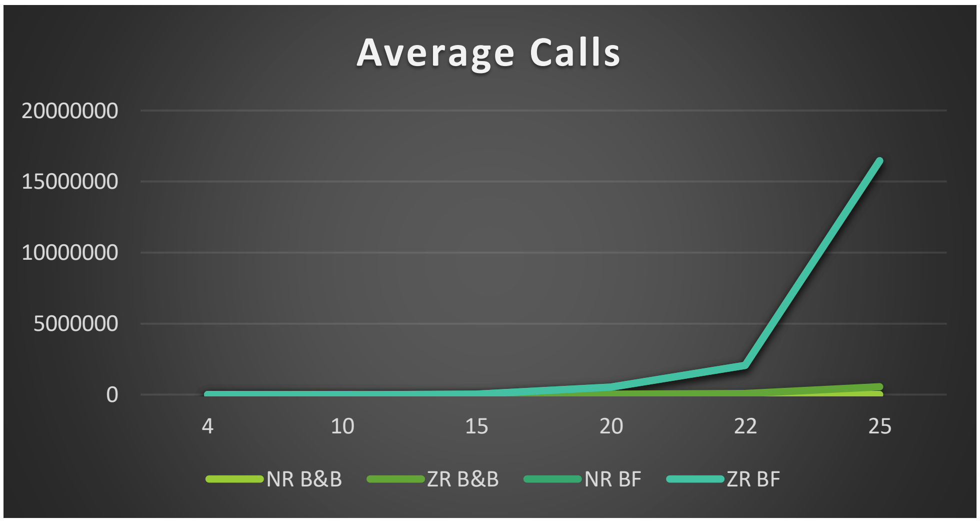
As we can see thanks to the histogram, by Brute Force there aren´t many differences but relative to Branch and Bound the NR instances are closer to the low calls, while the ZR instances do increase the number of calls more considerably and are not concentrated in the minimum as in NR. This means, as we could assume with the previous results, that the number of more “complex” problems is much greater in ZR, although there is a presence of these in both instances.

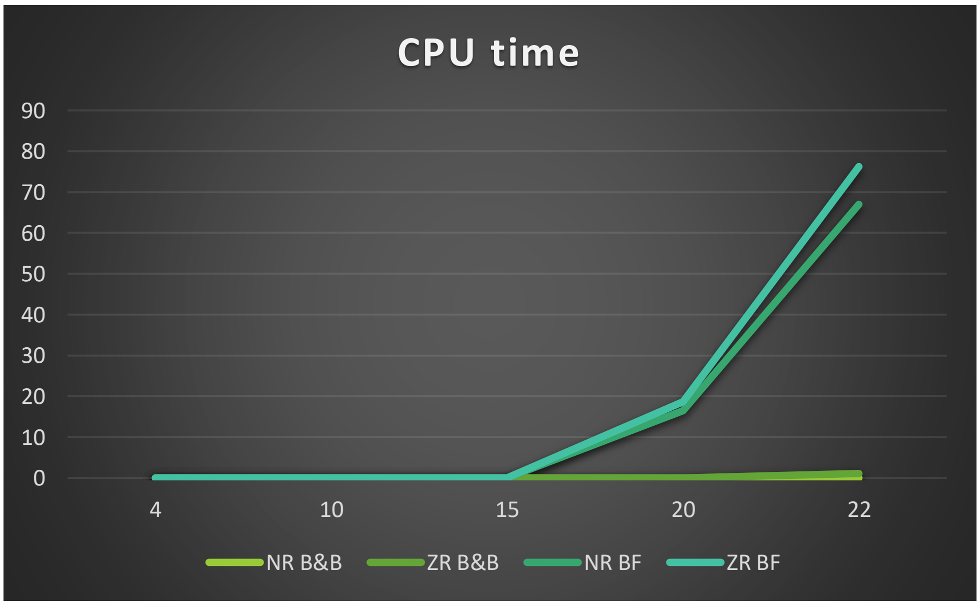
CPU time VS time complexity

Both the CPU time and the average recursive calls in both graphs exhibit exponential behavior and show remarkable uniformity in terms of growth. These parameters maintain consistent proportions in relation to the number of elements analyzed.

To delve deeper into this phenomenon, it is important to highlight that exponential growth means that as the size of the data sets increases, the resources required to complete the task increase exponentially. This can lead to a rapid escalation in CPU time and the average number of recursive calls, which can have significant implications in terms of computational efficiency.

The difference in CPU time between Brute Force and B&B is due to the fact that in B&B in some iterations it is cut when the weight is exceeded, while in Brute Force it is always continued until deciding whether or not to put all the objects in the backpack. In both cases it counts as an iteration or call.


Average Calls
NR B&B
ZR B&B
NR BF
ZR BF


CPU time
NR B&B
ZR B&B
NR BF
ZR BF