HOMEWORK 2 – EXACT AND HEURISTIC SOLUTIONS OF THE KNAPSACK PROBLEM

BY ANDRÉS RUBIO RAMOS

ALGORITHMS USED

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;
}
```

Simple 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);
}
```

Extended cost/weight heuristic

This Java algorithm is an extended heuristic approach for solving the Knapsack Problem. It involves two parts:

-1. The first part uses a simple heuristic approach, where items are sorted by their cost-to-weight ratio in descending order. It iterates through these items, selecting those that can fit within the remaining capacity of the knapsack. The total cost and solution list are updated accordingly.

-2. The second part aims to find the most expensive item that can fit in the knapsack without considering weight constraints. It sorts the items by cost in descending order and selects the first item that fits.

The algorithm returns the solution and total cost from the simple heuristic if it's greater or equal to the total cost from the expensive item approach, or it returns the solution and total cost from the expensive item approach if that results in a higher total cost.

```
private static Tuple extendedHeuristic(Knapsack k) {
    Integer costSimple=0; //init simple heuristic part
    Integer capacity=k.M();
    List<Integer> solSimple=new ArrayList<Integer>();
    for (int i=0;i<k.n();i++) solSimple.add(0);
    List<Item> ratioKnapsack=k.items().stream().
            sorted(Comparator.comparingDouble((Item x)->1.*x.cost()/x.weight()).reversed()).toList();

    Integer costExpensive=0; //init most expensive item solution
    List<Integer> solExpensive=new ArrayList<Integer>(solSimple);
    List<Item> costKnapsack=k.items().stream().
            sorted(Comparator.comparingDouble((Item x)->x.cost()).reversed()).toList();
    for (Item it:ratioKnapsack)
        if (it.weight()<=capacity) {
            costSimple+=it.cost();
            capacity-=it.weight();
            solSimple.set(it.index(), 1);
        }
    for (Item it:costKnapsack)
        if (it.weight()<=k.M()) {
            costExpensive+=it.cost();
            solExpensive.set(it.index(), 1);
            break;
        }
    return costSimple>=costExpensive ?
            new Tuple(solSimple, costSimple) : new Tuple(solExpensive, costExpensive);
}
```

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<>();
}
```

Dynamic programming (cost)

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

Based on minimizing the total cost while staying within a given weight limit. It initializes a matrix to store intermediate results and iterates through subproblems to calculate the minimum cost for different combinations of items and weights. The final result is the maximum cost achievable within the specified weight limit for the given set of items.Since the last time I had problems with this algorithm and I have changed things like memory. Before I used a map and now I have opted for a matrix and the results in efficiency are as expected.

```
private static Integer initDynaByCost(Knapsack k) {
    //memory to remember the results of already solved subproblems.
    Map<IntTuple, Integer> mapWeights=new HashMap<>();
    List<Integer> W=new ArrayList<>();
    List<Integer> C=new ArrayList<>();
    Integer sumCosts=0;
    for (Item it:k.items()) {
        W.add(it.weight());
        C.add(it.cost());
        sumCosts+=it.cost();
    }
    int n = W.size();
    Integer result=0;
    Integer sol;
    for (int c=sumCosts;c>=0;c--) {
        sol=dynaByCost(n, c, W, C, mapWeights);
        if (sol<=k.M()) { // we chose the one where W(n, c) < M for the highest c
            result=c;
            break;
        }
    }
    return result;
}

private static Integer dynaByCost(Integer index, Integer currentCost,
        List<Integer> W, List<Integer> C, Map<IntTuple, Integer> mapWeights) {
    IntTuple key=new IntTuple(index,currentCost);
    if (mapWeights.containsKey(key)) return mapWeights.get(key);

    Integer result;
    if (index==0 && currentCost==0) { //base subproblems
        result=0;
    } else if (index==0) {
        result=10000000;
    } else if (currentCost-C.get(index-1)>=0) { //recursion
        result=Math.min(dynaByCost(index-1, currentCost, W, C, mapWeights),
                dynaByCost(index-1, currentCost-C.
                        get(index-1), W, C, mapWeights)+W.get(index-1));
    } else {
        result=dynaByCost(index-1, currentCost, W, C, mapWeights);
    }
    mapWeights.put(key, result);
    return result;
}
```

FPTAS approximation algorithm

This Java algorithm implements the Fully Polynomial-Time Approximation Scheme (FPTAS) for the knapsack problem.

It adjusts the costs of items to create an approximate solution with a specified maximum error. The algorithm uses dynamic programming to find the optimal solution for the modified costs, considering a reduced set of items that meet the weight constraint. The final result is the total cost of the selected items in the knapsack, accounting for the adjustments made to achieve the desired approximation.

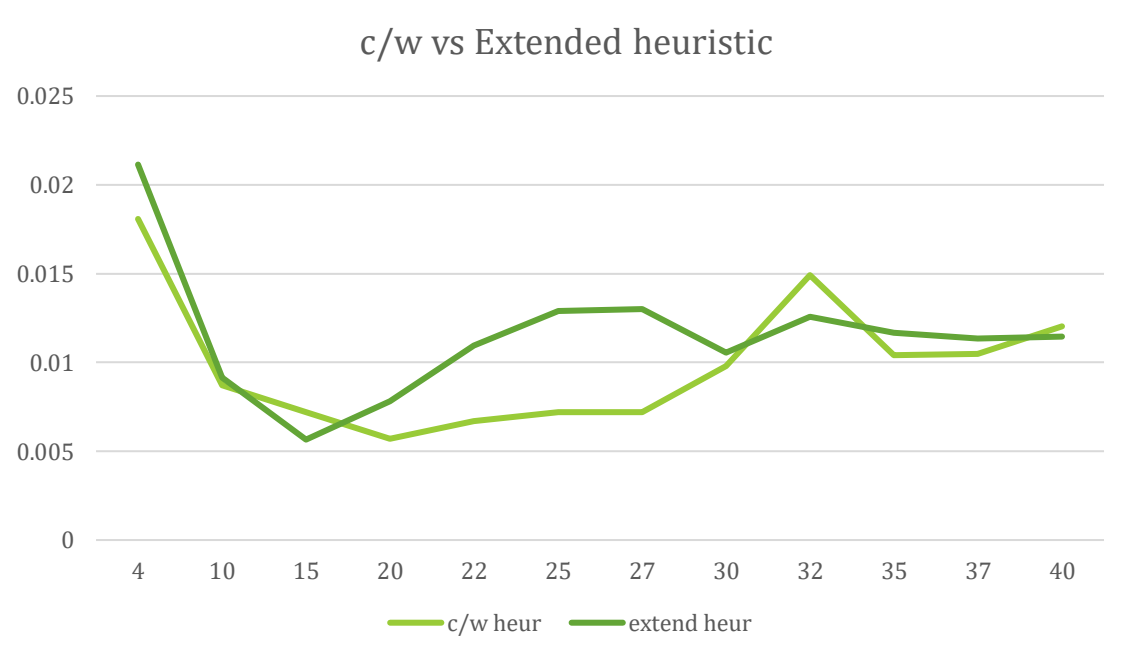
```
private static Integer initFPTAS(Knapsack k, Double maximumError) {
    List<Integer> W=new ArrayList<>();
    List<Integer> C=new ArrayList<>();
    List<Integer> CPrime=new ArrayList<>();
    Integer sumCosts=0;
    int n = k.n(); //FPTAS changes
    Integer maxC=k.items().stream().mapToInt(x->x.cost()).max().getAsInt();
    Double K=maximumError*maxC/n;
    for (Item it:k.items())
        if (it.weight()<=k.M()) {
            Integer cPrime=(int) (it.cost()/K);
            W.add(it.weight());
            C.add(it.cost());
            CPrime.add(cPrime); //List of alternative costs
            sumCosts+=cPrime;
        } else n--;
    //matrix to remember the results of already solved subproblems.
    int[][] matrixWeights=new int[n+1][sumCosts+1];
    Integer costOptFPTAS=dynaByCost(n, k.M(), W, CPrime, sumCosts, matrixWeights);
    Integer result=0;
    for (int i=n;i>0;i--) //Construct the total real cost of the solution
        if (matrixWeights[i][costOptFPTAS]!=matrixWeights[i-1][costOptFPTAS]) {
            costOptFPTAS-=CPrime.get(i-1);
            result+=C.get(i-1);
        }
    return result;
}
```

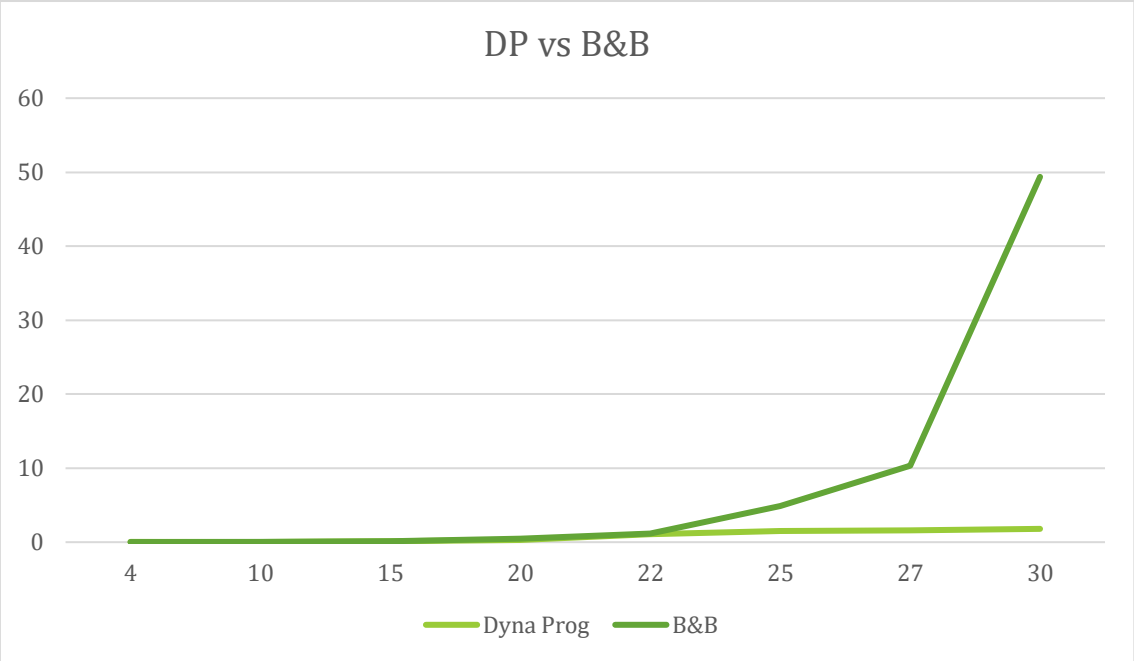
COMPARISON OF RUN-TIMES

NK instances

For these instances these would be the time results in seconds obtained for the different sizes:

Items	Branch and Bound	Dynamic(capacity) programming	Cost/weight heuristic	Extended heuristic
4	0.0174229	0.019101	0.0180784	0.0211422
10	0.0279103	0.1016763	0.0087172	0.0091627
15	0.0900123	0.3359661	0.00721	0.0056562
20	0.4741521	1.0483254	0.0057028	0.0078177
22	1.1407103	1.5140273	0.0066919	0.0109409
25	4.86854	1.6059795	0.0072102	0.0128858
27	10.3082464	1.7960927	0.0071925	0.0130013
30	49.3699149	2.3620575	0.0098062	0.0105582
32	126.6918067	2.9417519	0.0149244	0.0125744
35	445.5081343	3.6422924	0.0104024	0.0116867
37	1204	4.7890909	0.0104885	0.0113528
40		5.661964	0.0120272	0.0114516

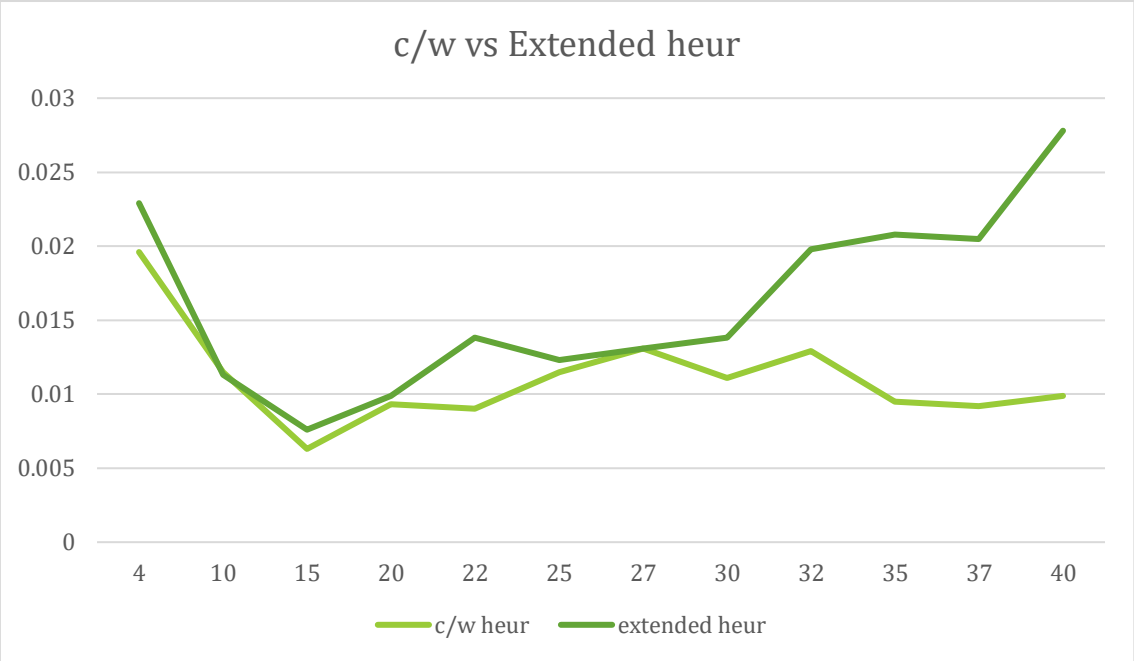


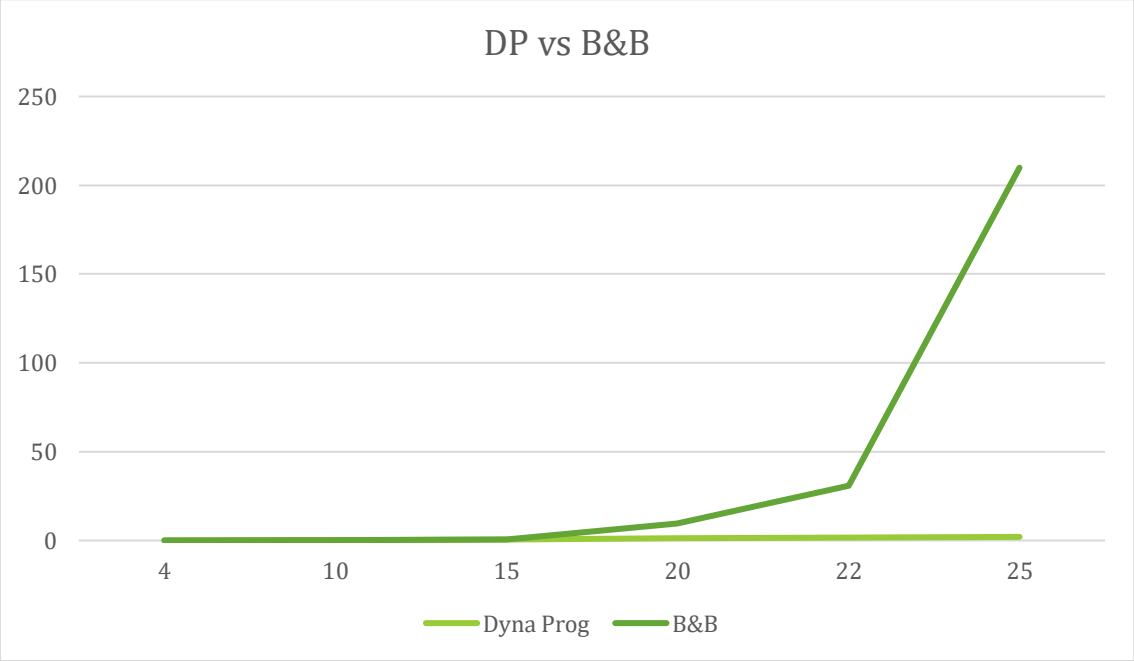


ZKC instances

For these instances these would be the time results in seconds obtained for the different sizes:

Items	Branch and Bound	Dynamic programming	Cost/weight heuristic	Extended heuristic
4	0,0181	0.0173379	0.0196	0.0229
10	0.0583	0.0967509	0.0115	0.0113
15	0.5152	0.3873794	0.0063	0.0076
20	9.6905	1.1303484	0.0093	0.0099
22	30.8602	1.5180133	0.0090	0.0138
25	209.9095	2.0293051	0.0115	0.0123
27	712.66	2.2618395	0.0131	0.0131
30		3.2034054	0.0111	0.0138
32		4.382569	0.0129	0.0198
35		5.3001313	0.0095	0.0208
37		5.6908202	0.0092	0.0205
40		7.7016878	0.0099	0.0278

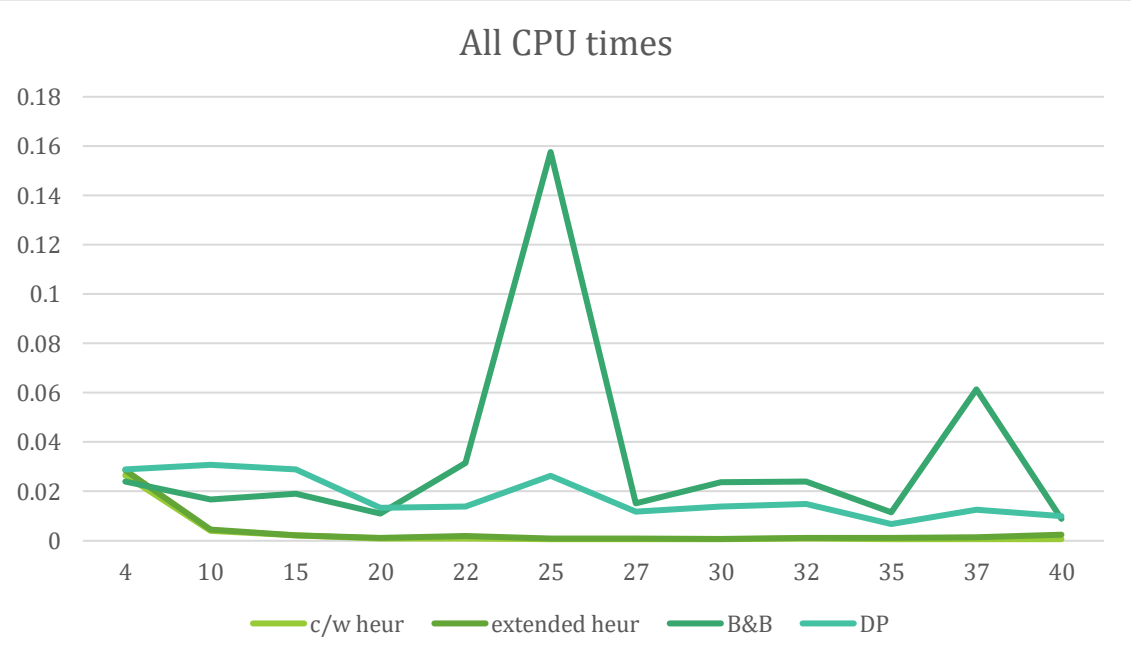

c/w vs Extended heur
0.03
0.025
0.02
0.015
0.01
0.005
0
4
10
15
20
22
25
27
30
32
35
37
40
c/w heur
extended heur


DP vs B&B
250
200
150
100
50
0
4
10
15
20
22
25
Dyna Prog
B&B

ZKW instances

For these instances these would be the time results in seconds obtained for the different sizes:

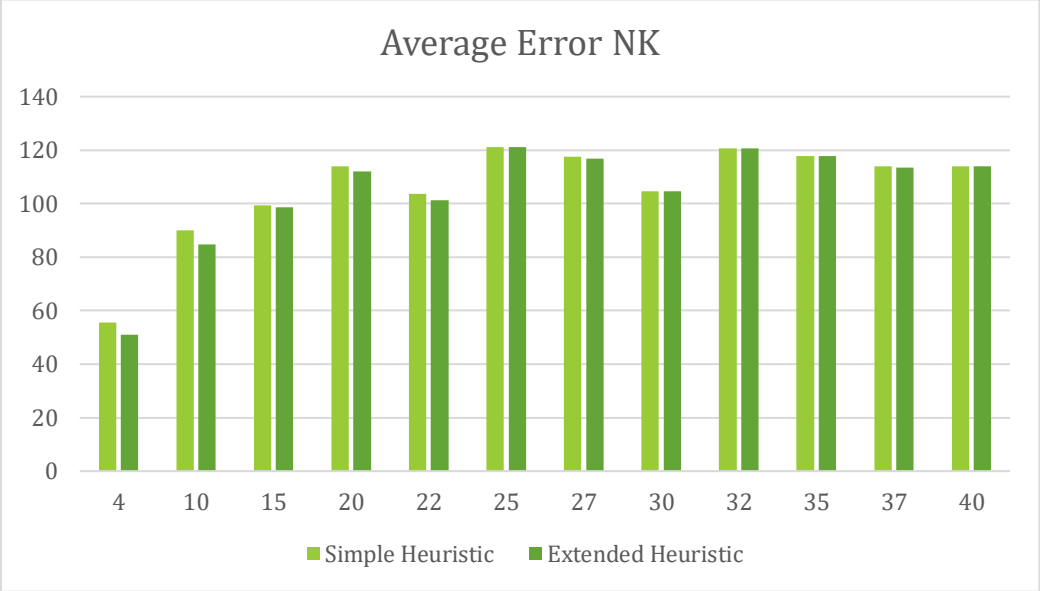
Items	Branch and Bound	Dynamic programming	Cost/weight heuristic	Extended heuristic
4	0.0239	0.029	0.02652	0.02859
10	0.0167	0.030853	0.00412	0.0046612
15	0.0192	0.02909	0.00218	0.0023019
20	0.0111	0.0134	0.0010207	0.001262
22	0.0316	0.014	0.0010343	0.0018731
25	0.1576	0.0264	0.0007587	0.000928
27	0.0151	0.0117	0.0007056	0.0009263
30	0.0238	0.0139	0.0007178	0.000754
32	0.0239	0.0150	0.0008243	0.0010565
35	0.0116	0.0068	0.0007157	0.001214
37	0.0613	0.0127	0.0006642	0.0014731
40	0.0089	0.0099	0.0007437	0.0024284

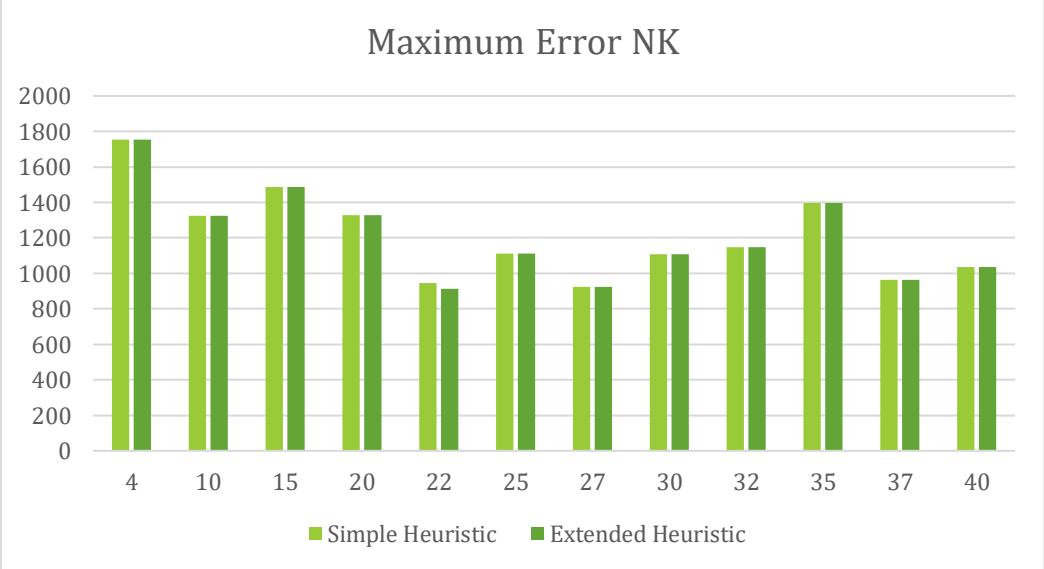


COMPARISON OF AVERAGE AND MAXIMUM ERROR

NK instances

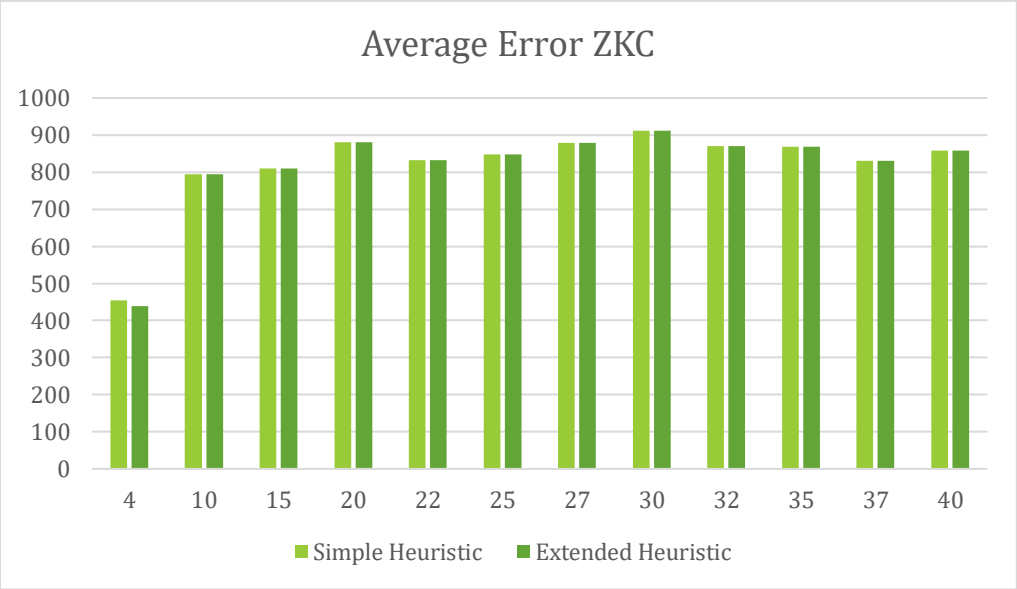
Items	c/w heuristic Average	Extended heuristic Average	c/w heuristic maximum	Extended heuristic maximum
4	55.564	50.964	1752	1752
10	89.98	84.714	1324	1324
15	99.376	98.55	1486	1486
20	114.044	111.972	1329	1329
22	103.606	101.192	945	913
25	121.03	121.03	1112	1112
27	117.492	116.714	923	923
30	104.664	104.664	1106	1106
32	120.604	120.604	1147	1147
35	117.79	117.79	1398	1398
37	113.848	113.448	965	965
40	113.84	113.84	1036	1036

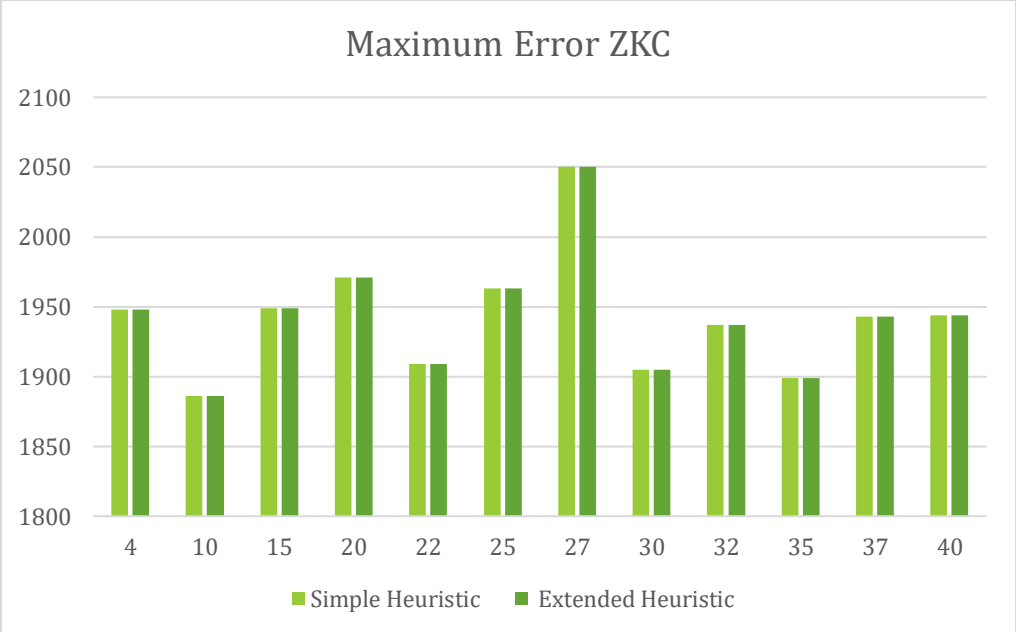




ZKC instances

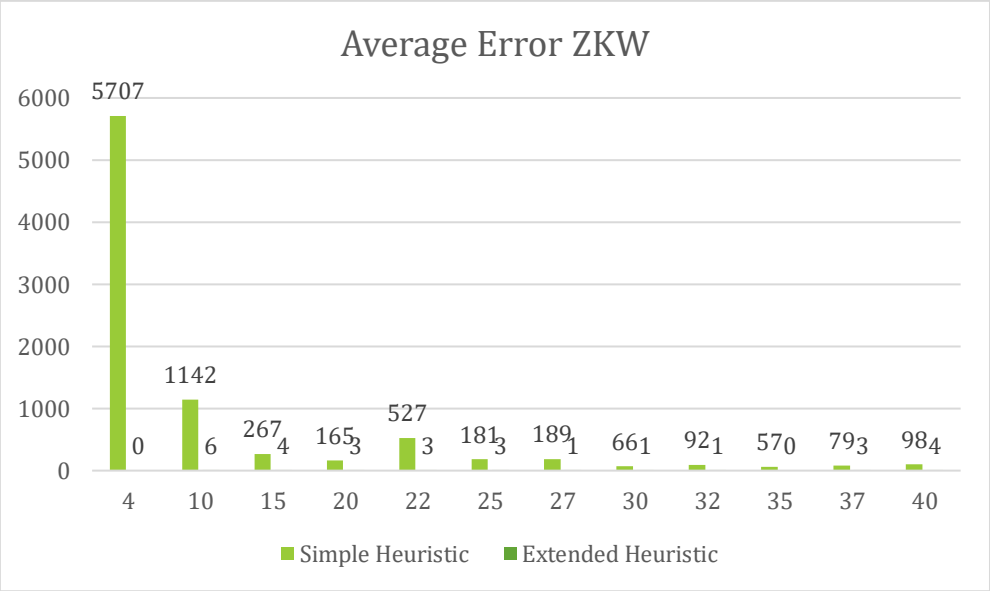
Items	c/w heuristic Average	Extended heuristic Average	c/w heuristic maximum	Extended heuristic maximum
4	454.436	439.426	1948	1948
10	794.346	794.346	1886	1886
15	810.326	810.326	1949	1949
20	881.34	881.34	1971	1971
22	831.516	831.516	1909	1909
25	847.174	847.174	1963	1963
27	878.682	878.682	2050	2050
30	911.27	911.27	1905	1905
32	870.284	870.284	1937	1937
35	869.2	869.2	1899	1899
37	830.13	830.13	1943	1943
40	858.37	858.37	1944	1944

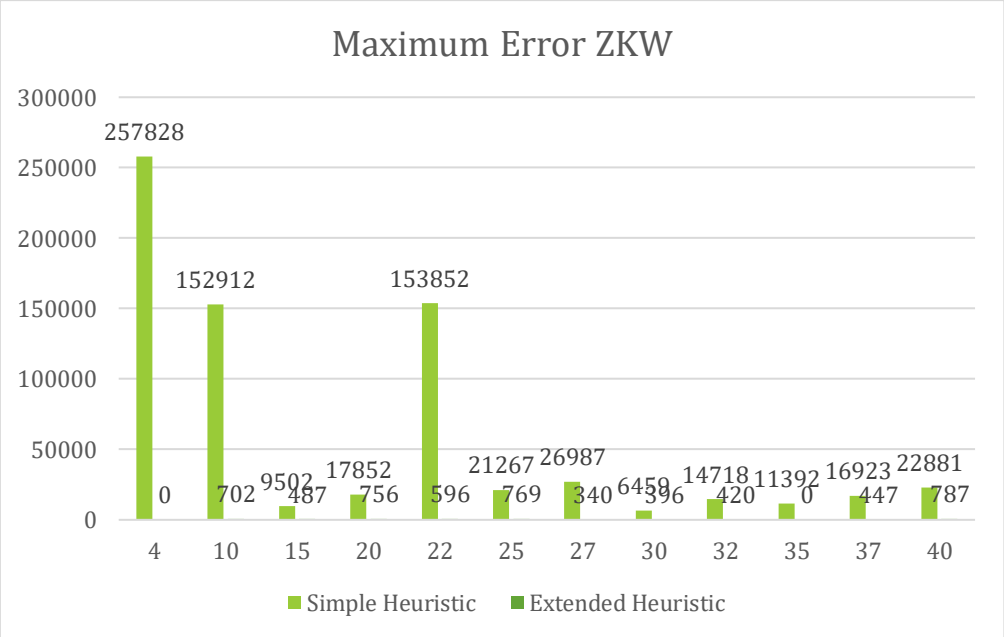




ZKW instances

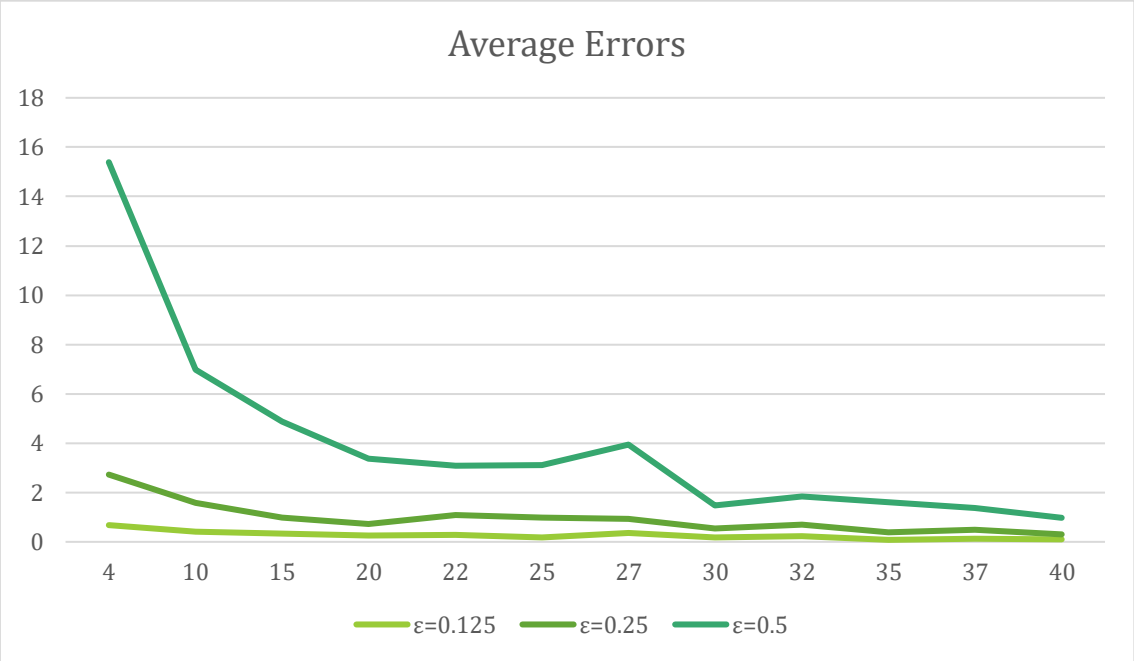
Items	c/w heuristic Average	Extended heuristic Average	c/w heuristic maximum	Extended heuristic maximum
4	5707.05	0	257828	0
10	1142.638	5.884	152912	702
15	267.162	3.758	9502	487
20	165.904	2.964	17852	756
22	527.688	2.77	153852	596
25	181.646	3.452	21267	769
27	189.37	0.68	26987	340
30	66.836	0.792	6459	396
32	92.384	1.434	14718	420
35	57.594	0	11392	0
37	79.59	3.48	16923	447
40	98.386	3.684	22881	787

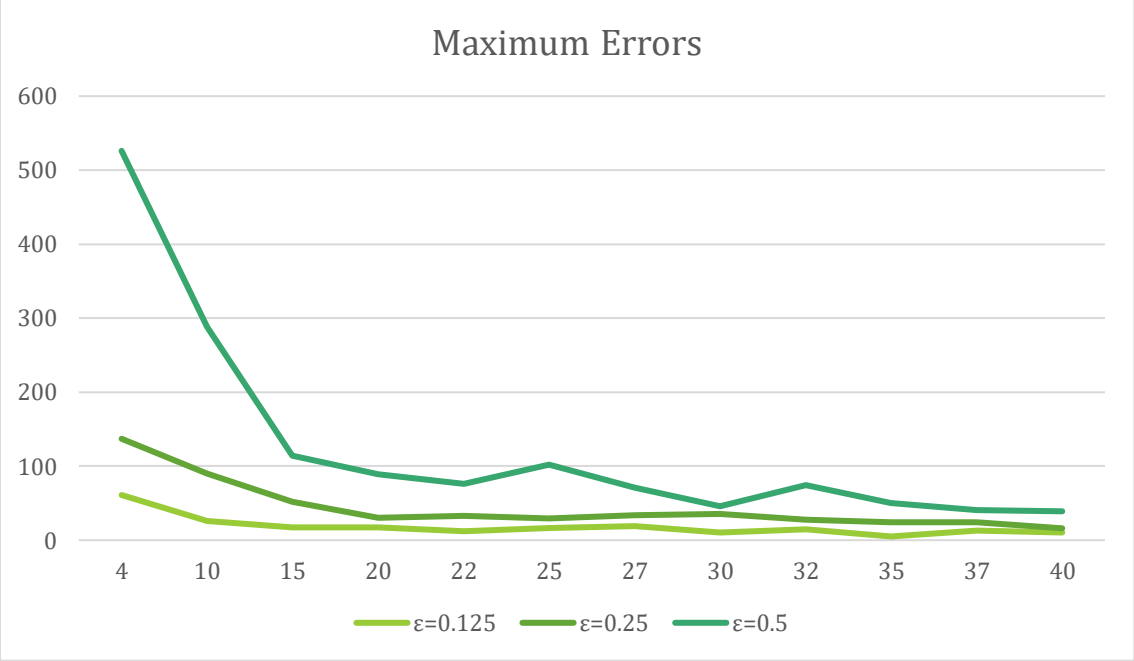


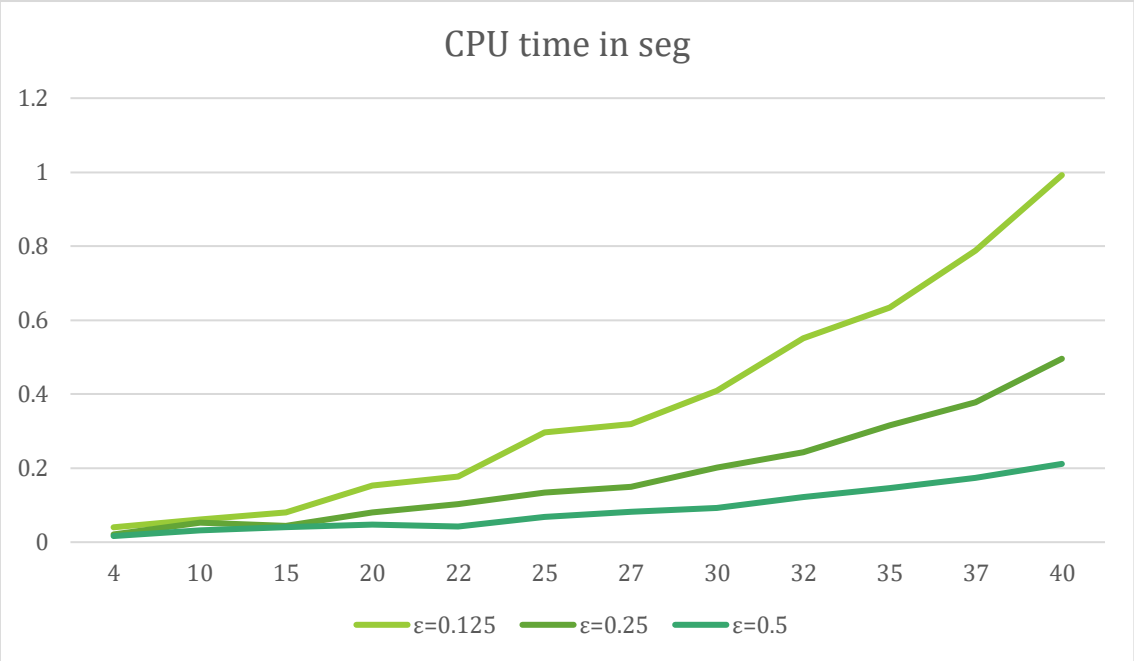


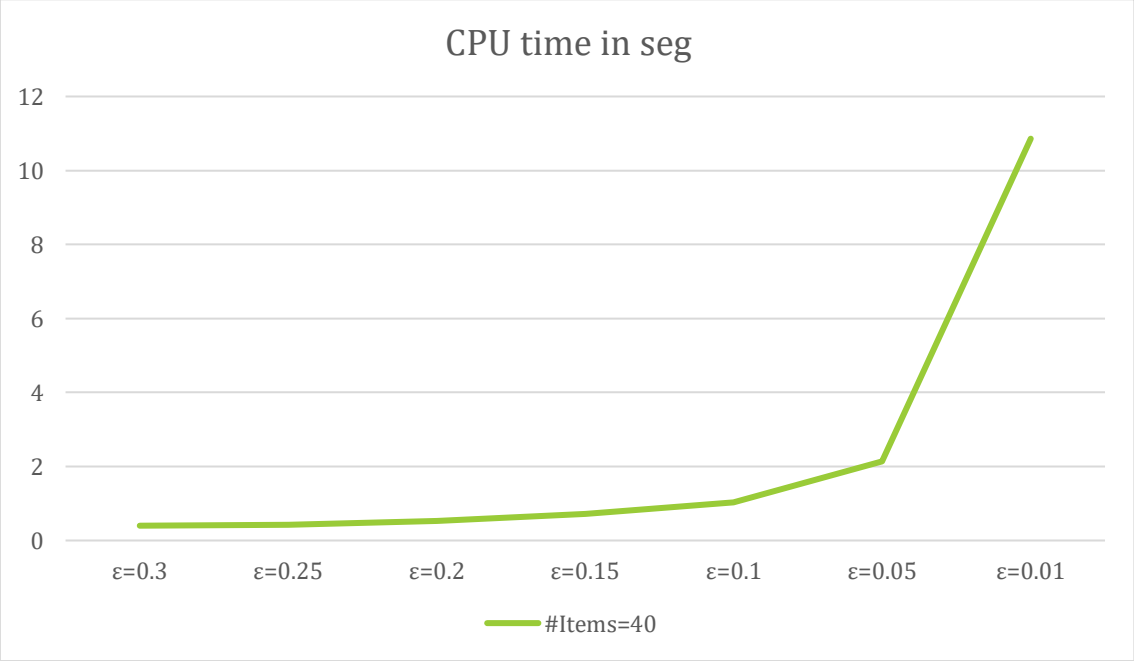
FPTAS study

NK instances

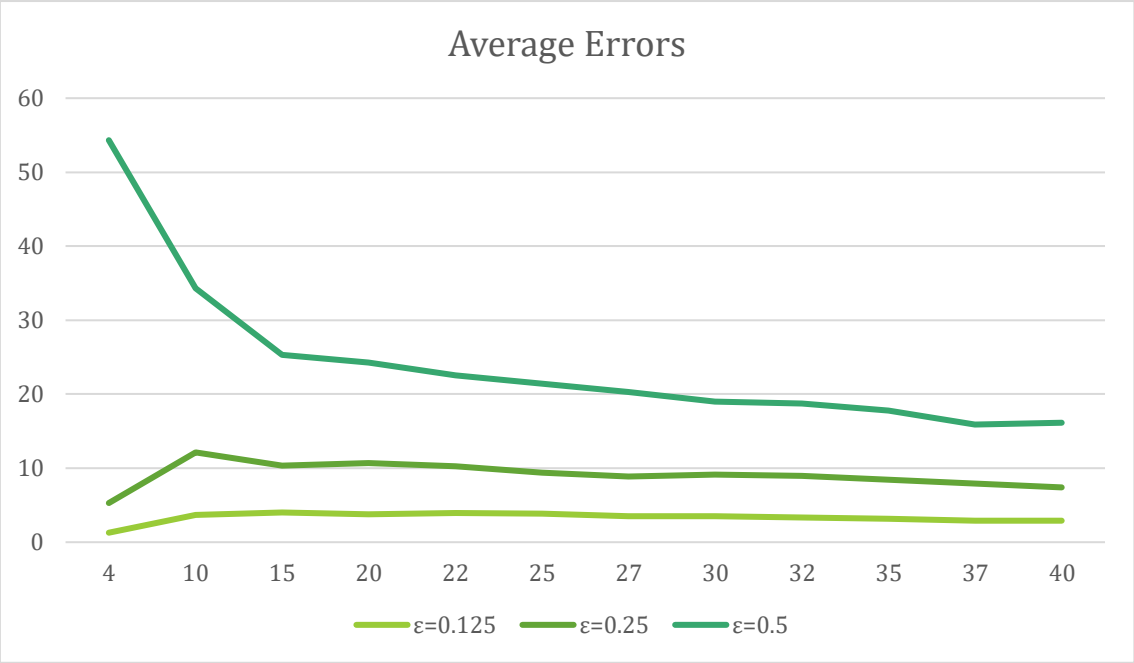


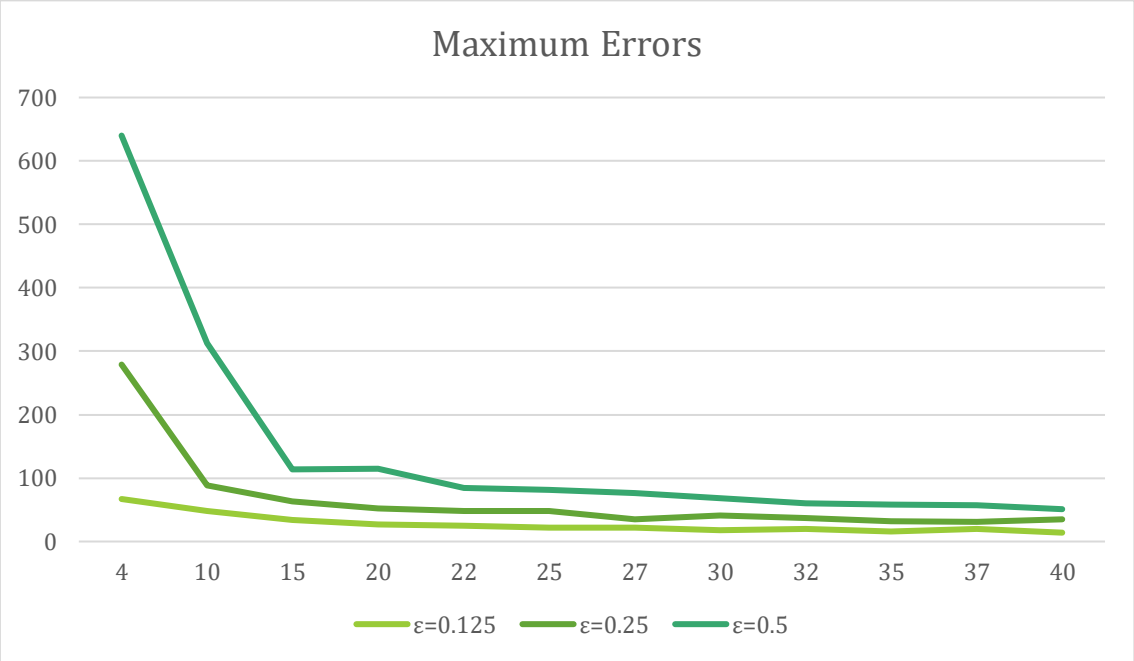


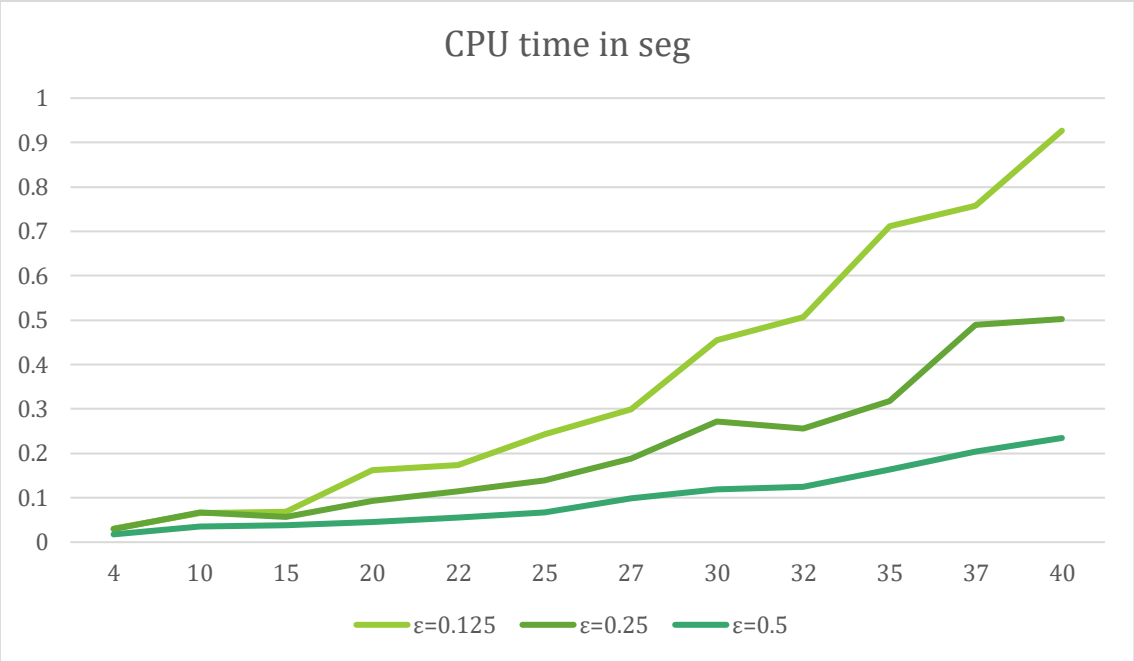

CPU time in seg
1.2
1
0.8
0.6
0.4
0.2
0
4
10
15
20
22
25
27
30
32
35
37
40
ε=0.125
ε=0.25
ε=0.5

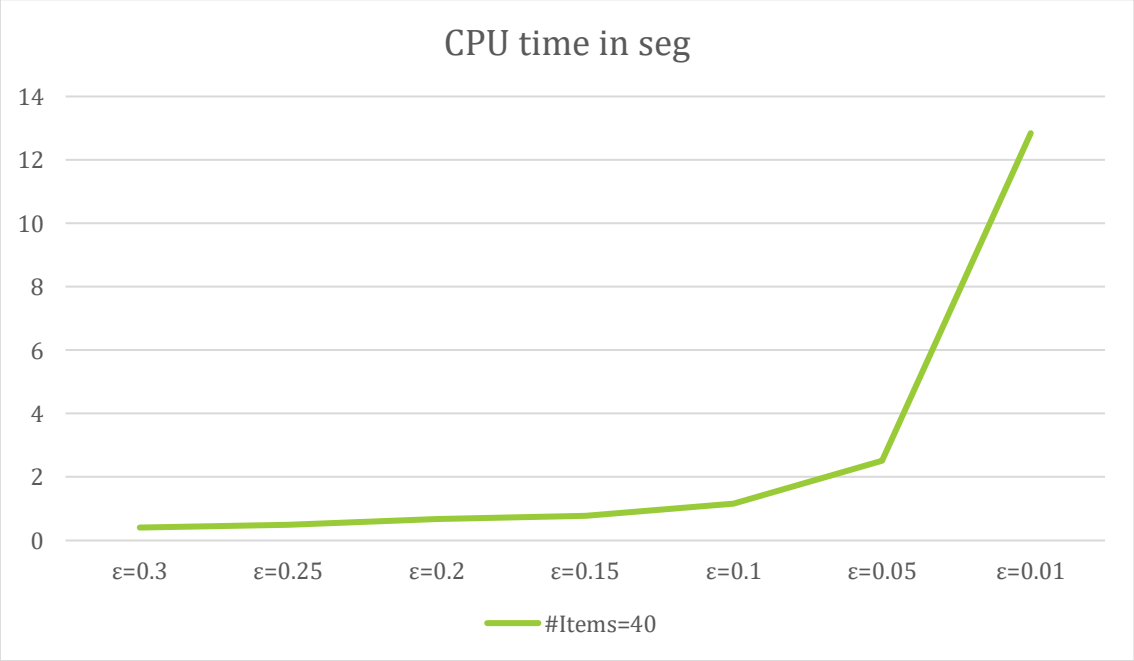

CPU time in seg
12
10
8
6
4
2
0
ε=0.3
ε=0.25
ε=0.2
ε=0.15
ε=0.1
ε=0.05
ε=0.01
#Items=40

ZKC instances

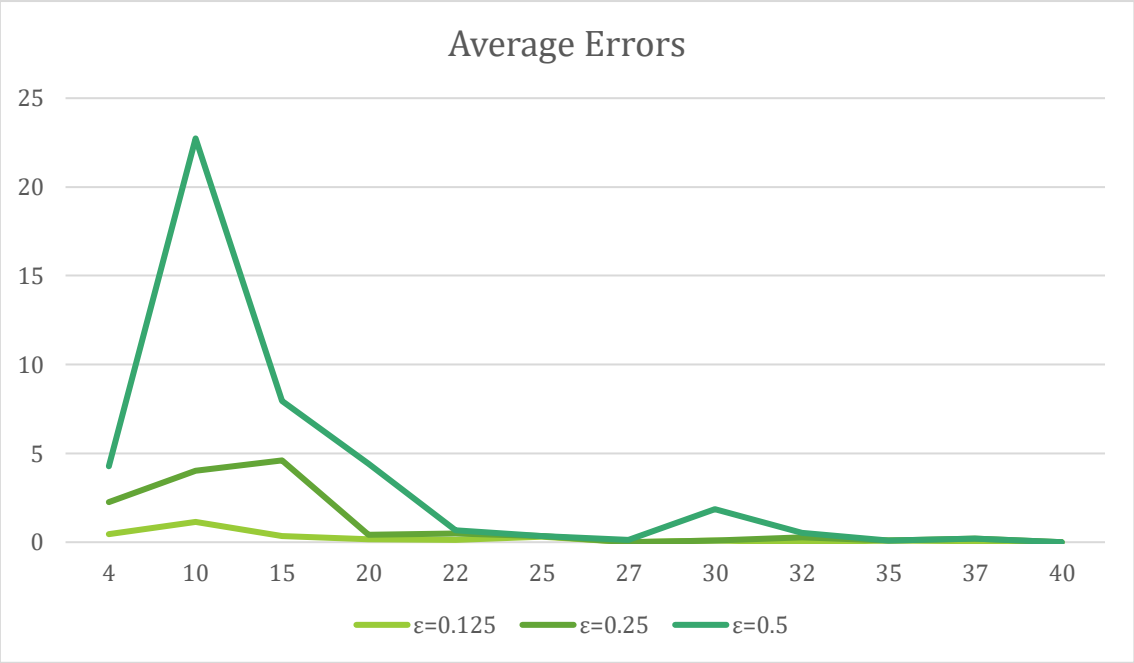


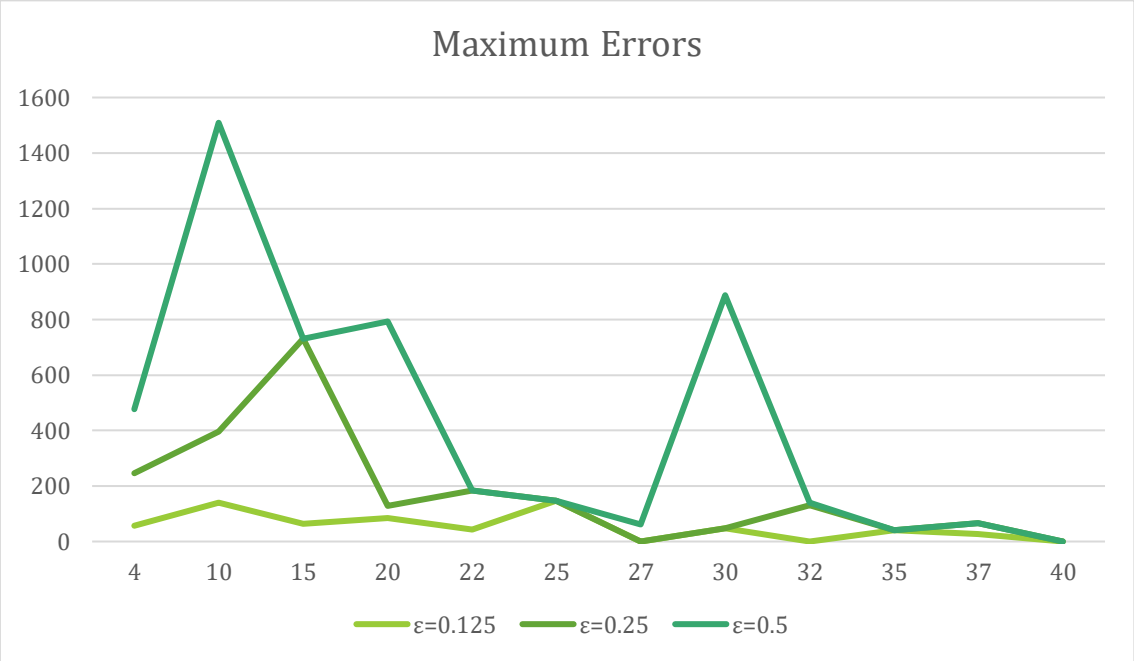


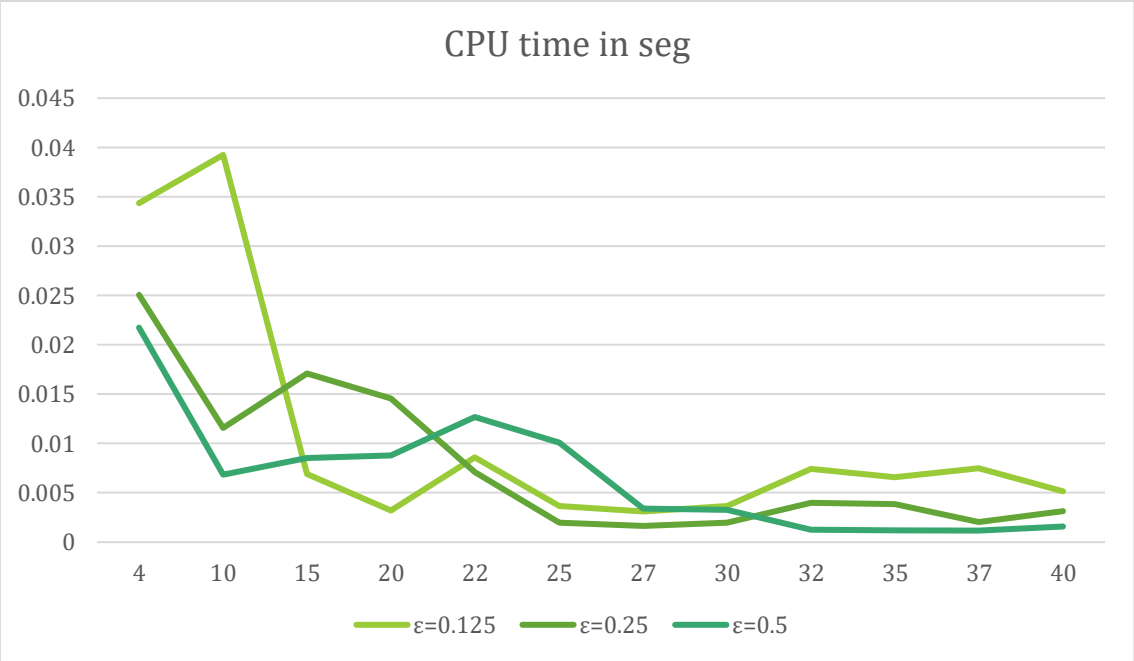

CPU time in seg
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
4
10
15
20
22
25
27
30
32
35
37
40
ε=0.125
ε=0.25
ε=0.5

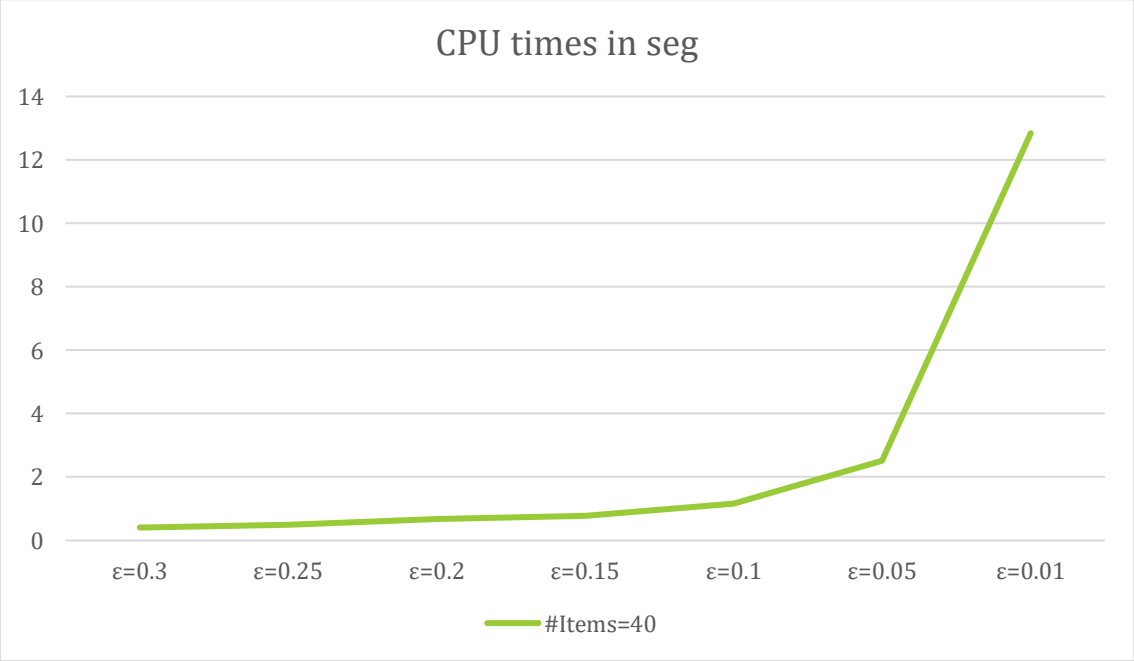

CPU time in seg
14
12
10
8
6
4
2
0
ε=0.3
ε=0.25
ε=0.2
ε=0.15
ε=0.1
ε=0.05
ε=0.01
#Items=40

ZKW instances






CPU time in seg
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
4
10
15
20
22
25
27
30
32
35
37
40
ε=0.125
ε=0.25
ε=0.5


CPU times in seg
14
12
10
8
6
4
2
0
ε=0.3
ε=0.25
ε=0.2
ε=0.15
ε=0.1
ε=0.05
ε=0.01
#Items=40