ΠΑΝΕΠΙΣΤΗΜΙΟ ΙΩΑΝΝΙΝΩΝ

ΣΧΟΛΗ: ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ: ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΔΙΕΡΕΥΝΗΣΗ ΠΡΟΣΕΓΓΙΣΕΩΝ ΕΠΙΛΥΣΗΣ ΤΟΥ ΠΡΟΒΛΗΜΑΤΟΣ 0-1 ΣΑΚΙΔΙΟΥ

ΜΑΘΗΜΑ: ΑΛΓΟΡΙΘΜΟΙ ΚΑΙ ΠΡΟΧΩΡΗΜΕΝΕΣ ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

ΜΕΤΑΠΤΥΧΙΑΚΟΣ ΦΟΙΤΗΤΗΣ: ΜΠΕΚΡΗΣ ΓΕΩΡΓΙΟΣ

ΕΞΑΜΗΝΟ 1ο

ΚΑΘΗΓΗΤΗΣ: ΓΚΟΓΚΟΣ ΧΡΗΣΤΟΣ

ΗΜΕΡΟΜΗΝΙΑ ΥΠΟΒΟΛΗΣ ΕΡΓΑΣΙΑΣ: 07/01/2020

ΑΚΑΔ. ΕΤΟΣ 2019-2020

Πίνακας περιεχομένων

Περίληψη

Στην παρούσα εργασία έγινε η επίλυση του προβλήματος 0-1 σακιδίου (0-1 knapsack) για διάφορα στιγμιότυπα με τέσσερις διαφορετικούς τρόπους (άπληστα, εξαντλητικά, με διακλάδωση και φραγή και με δυναμικό προγραμματισμό). Επιπλέον, συντάχθηκε τεχνική αναφορά που περιγράφει εν συντομία τις προσεγγίσεις επίλυσης και τα αποτελέσματα. Τέλος, ο κώδικας της εφαρμογής (σε γλώσσα c++) αναρτήθηκε σε αποθετήριο και κατασκευάστηκε μια ιστοσελίδα μέσω του github που εμφανίζει τα κύρια αποτελέσματα της εργασίας.

1. Περιγραφή του προβλήματος

Το πρόβλημα 01 σακιδίου (01 knapsack) αφορά ένα σύνολο από αντικείμενα για το καθένα από τα οποία γνωρίζουμε το βάρος και την αξία του. Ζητείται η επιλογή ενός υποσυνόλου των αντικειμένων έτσι ώστε το συνολικό βάρος από τα επιλεχθέντα αντικείμενα να μην ξεπερνά μια συγκεκριμένη τιμή βάρους και ταυτόχρονα να επιτυγχάνεται η μεγαλύτερη δυνατή αξία. Το πρόθεμα 01 στο όνομα του προβλήματος υποδηλώνει ότι κάθε αντικείμενο μπορεί είτε να επιλεχθεί, είτε να μην επιλεχθεί στο σύνολό του και όχι τμηματικά.

Για την επίλυση του προβλήματος έγινε χρήση του προγράμματος **Visual Studio Code**.

2. Δημιουργία στιγμιότυπων του προβλήματος

Αρχικά δημιουργήθηκαν στιγμιότυπα προβλημάτων 01 σακιδίου με τη χρήση του generator και του κώδικα που βρίσκεται στη διεύθυνση (http://hjemmesider.diku.dk/~pisinger/generator.c).

Στη συνέχεια έγινε compile του κώδικα:

```
g++ -o generator generator.c
```

Δημιουργήθηκαν από 5 στιγμιότυπα για κάθε συνδυασμό των ακόλουθων παραμέτρων: n={10,50,100,500}, r={50,100,500,1000} και type={1,2,3,4}, δηλαδή σύνολο 5 × 4 × 4 × 4 = 320 στιγμιότυπα.

Επομένως για να δημιουργηθούν τα παραπάνω στιγμιότυπα έτρεχε ο παρακάτω κώδικας:

```
generator n r type i S
```

όπου n: ο αριθμός των αντικειμένων
r: το εύρος των συντελεστών pi και wi
type: ο τύπος των στιγμιότυπων
i: το στιγμιότυπο όπου (i ϵ [1,…., S])
S: Το σύνολο των στιγμιότυπων
Το αποτέλεσμα του κώδικα αποθηκεύεται σε ένα αρχείο test.in.

Για παράδειγμα το αρχείο “problem_10_50_1_1_5.txt” είναι το αρχείο που δημιουργήθηκε με n=10, r=50, t=1 και είναι το πρώτο από τα συνολικά πέντε αρχεία που δημιουργήθηκαν με τις ίδιες παραμέτρους.

Άρα για να δημιουργηθεί το παραπάνω αρχείο θα πρέπει να τρέξει ο παρακάτω κώδικας:

```
generator 10 50 1 1 5

mv test.in problem_10_50_1_1_5.txt
```

Η ίδια διαδικασία ακολουθήθηκε και για τα υπόλοιπα στιγμιότυπα.

3. Επίλυση του προβλήματος

Ο κώδικας του προβλήματος αποθηκεύτηκε σε αρχείο με όνομα knapsack.cpp

Η επίλυση του προβλήματος έγινε με τους ακόλουθους τέσσερις διαφορετικούς τρόπους:

Άπληστη μέθοδος

Στην άπληστη μέθοδο (greedy approach) υπολογίζεται για κάθε αντικείμενο ο λόγος αξία προς βάρος και δίνεται προτεραιότητα στην εισαγωγή των αντικειμένων με τις μεγαλύτερες τιμές και μέχρι να μην μπορεί να προστεθεί άλλο αντικείμενο στο σακίδιο.

Δημιουργήθηκε συνάρτηση vector<item>greedy_solver(knapsack_problem &ks) όπως φαίνεται στον παρακάτω κώδικα:

```
std::vector <item> greedy_solver(knapsack_problem &ks)
{
std:: sort(ks.items.begin(), ks.items.end(), [](item &item1, item &item2)
{
return (double) item1.profit/(double) item1.weight > (double) item2.profit/ (double) item
2.weight;
});
int total_weight = 0;
std:: vector <item> solution;

for (int i=0; i<ks.items.size(); i++)
{
if (total_weight + ks.items[i].weight >ks.capacity)
break;
solution.push_back(ks.items[i]);
total_weight +=ks.items[i].weight;
}
std:: sort(solution.begin(), solution.end(), [](item &item1, item &item2)
{
return (item1.id < item2.id);
});
return solution;
}
```

Στη συνέχεια για να δημιουργηθεί το κάθε αρχείο με το αποτέλεσμα της άπληστης μεθόδου (συνολικό κέρδος, συνολικό βάρος, σύνολο επιλεγέντων αντικειμένων) έγινε κλήση της main (int argc, char **argv) ως εξής:

```
g++ -o knapsack knapsack.cpp

./knapsack 1
```

```
case 1:

     for (int n :{10, 50, 100, 500})
   for (int r: {50, 100, 500, 1000})
   for (int type: {1,2,3,4})
   for (int instance_id=1; instance_id<=5; instance_id++)
  { string problem_instance="problem_" + to_string(n) + "_" + to_string(r) + "_" + to_str
ing(type) +"_" + to_string(instance_id);
    string out= problem_instance;
    out.append("_GR");
    string path_fn="../helloworld/";
    path_fn.append(problem_instance);
    path_fn.append("_5.txt");
    knapsack_problem ks =read_data(path_fn);
    print_knapsack_problem_info(ks);
    std::vector<item> solution;
    solution=greedy_solver(ks);
    get_profit(ks,solution);
    export_solution(ks,solution,out);

   }
break;
```

Άρα δημιουργήθηκαν 320 αρχεία με τα αποτελέσματα της άπληστης μεθόδου. Για παράδειγμα για το στιγμιότυπο “problem_10_50_1_1_5.txt” δημιουργήθηκε το αρχείο problem_10_50_1_1_GR.

Εξαντλητική απαρίθμηση συνδυασμών

Στην εξαντλητική απαρίθμηση (brute force full enumeration) δοκιμάστηκαν όλοι οι πιθανοί συνδυασμοί τοποθέτησης αντικειμένων στο σακίδιο (ανά ένα αντικείμενο, ανά δύο αντικείμενα, ανά τρία αντικείμενα κ.ο.κ.) έτσι ώστε να επιλεγεί η πλέον συμφέρουσα. Η συγκεκριμένη προσέγγιση δεν μπορεί να λειτουργήσει για μεγάλα στιγμιότυπα προβλημάτων, οπότε εάν το πρόβλημα είναι επαρκώς μεγάλο θα πρέπει να επιστρέφει το καλύτερο αποτέλεσμα που εντοπίζει με μέγιστο χρονικό περιθώριο εκτέλεσης τα 10 δευτερόλεπτα.

Δημιουργήθηκε συνάρτηση vector<item>brute_force_solver(knapsack_problem &ks) όπως φαίνεται στον παρακάτω κώδικα:

```
vector <item> brute_force_solver(knapsack_problem &ks)
{
    high_resolution_clock :: time_point t1 =high_resolution_clock::now();
    vector <item> result;
    int max_profit=-1;
    vector <vector<item>>sets;
 for (item an_item:ks.items)
    {
        vector <vector<item>>new_sets;
        new_sets.push_back({an_item});
```

```
        for (vector<item> a_set : sets)
        {
            a_set.push_back(an_item);
            new_sets.push_back(a_set);
        }

    }
    int n=ks.items.size();
    int total=1<<n;
    if (n>=31)
    total=numeric_limits<int>::max();
    for (int i=0; i<total; i++)
    {
        vector<item> sol;
        for (int j=0; j<n; j++)
        {
            if ((i>>j) & 1)
            sol.push_back(ks.items[j]);
        }
        int profit=get_profit(ks,sol);
        if (profit> max_profit)
        {
        max_profit = profit;
        result=sol;
        }
    }

        high_resolution_clock :: time_point t2 =high_resolution_clock::now();
      auto duration = duration_cast<microseconds>(t2-t1).count();
      if (duration>ks.time_limit*1000)
      {
        cout<<"Time elapsed:"<<duration<<"microseconds"<<duration/1000.0<<"seconds"<<endl
;
      }
       return result;

}
```

Κατά τον ίδιο τρόπο, όπως και στην άπληστη μέθοδο, για να δημιουργηθεί το κάθε αρχείο με το αποτέλεσμα της εξαντλητικής απαρίθμησης συνδυασμών (συνολικό κέρδος, συνολικό βάρος, σύνολο επιλεγέντων αντικειμένων) έγινε κλήση της main (int argc, char **argv) ως εξής:

```
g++ -o knapsack knapsack.cpp

./knapsack 2
```

Δημιουργήθηκαν 320 αρχεία με τα αποτελέσματα της εξαντλητικής απαρίθμησης συνδυασμών. Για παράδειγμα για το στιγμιότυπο “problem_10_50_1_1_5.txt” δημιουργήθηκε το αρχείο problem_10_50_1_1_BF.

Δυναμικός προγραμματισμός

Για την προσέγγιση του δυναμικού προγραμματισμού (dynamic programming) υλοποιήθηκε ο παρακάτω κώδικας που χρησιμοποιώντας αναδρομή, επίλυση μικρότερων προβλημάτων και καταγραφή ενδιάμεσων αποτελεσμάτων σε έναν πίνακα οδηγούσε σε λύση του προβλήματος (συνάρτηση vector<item> dynamic_programming_solver(knapsack_problem &ks)):

```
vector<item> dynamic_programming_solver(knapsack_problem &ks)
 {
   int i,w;
   int n=ks.items.size();
   int W= ks.capacity;
   int maxValue;

  vector<vector<int>>table(n+1, vector<int>(W+1,0));

      for (w = 0; w <= W; w++)
        {
            table[0][w]=0;
        }
      for (i=1; i<=n; i++)
        {
                for (w = 0; w <= W; w++)
            {
                    if(ks.items[i].weight> w)
        {
                table[i][w] = table[i-1][w];

                }

                else
                {
        table[i][w] = max( table[i-1][w], ks.items[i].profit + table[i-1][w-
ks.items[i].weight] );
                  }
            }
        }

          vector <item> solution;

      int j= W;
      for(i=n; i>0; i--)

     {
     if (ks.items[i].weight<=j && ks.items[i].profit + table[i-1][j-
ks.items[i].weight] >=  table[i-1][j])
      {

      solution.push_back(ks.items[i]);
         j = j-ks.items[i].weight;

      }
```

```
      }
       maxValue = table[n][W];
       return solution;
}
```

Για τη δημιουργία του κάθε αρχείου με το αποτέλεσμα του δυναμικού προγραμματισμού (συνολικό κέρδος, συνολικό βάρος, σύνολο επιλεγέντων αντικειμένων) έγινε κλήση της main (int argc, char **argv) ως εξής:

```
g++ -o knapsack knapsack.cpp

./knapsack 3
```

Δημιουργήθηκαν 320 αρχεία με τα αποτελέσματα του δυναμικού προγραμματισμού. Για παράδειγμα, για το στιγμιότυπο “problem_10_50_1_1_5.txt” δημιουργήθηκε το αρχείο problem_10_50_1_1_DP.

Διακλάδωση και φραγή

Η μέθοδος διακλάδωσης και φραγής (branch and bound) αποτελεί βελτίωση της μεθόδου πλήρους απαρίθμησης.

Δημιουργήθηκε κώδικας για τη συγκεκριμένη μέθοδο όπως φαίνεται στον παρακάτω κώδικα:

```
struct Node
{
    int level, value, bound, weight;
};


int bound(Node u, knapsack_problem &ks)
{
  int j=0;
  int k=0;
  int value_bound=0;
  int totweight=0;
    if (u.weight >= ks.capacity)
    {
    return 0;
    }
    else
    {
    value_bound=u.value;
    j = u.level +1;
    totweight=u.weight;

    while ((j<ks.items.size()) && (totweight + ks.items[j].weight <=ks.capacity))
    {
```

```
        totweight +=ks.items[j].weight;
        value_bound += ks.items[j].profit;
        j++;
    }
    k=j;
    if (k < ks.items.size())
    {
    value_bound += (ks.capacity - totweight) * ks.items[j].profit / ks.items[j].weight;
    }
    return value_bound;
    }
}

vector<item> branch_and_bound_solver(knapsack_problem &ks)
{

vector<item> solution;
int n=ks.items.size();
int total_weight=0;

    std:: sort(ks.items.begin(), ks.items.end(), [](item &item1, item &item2)
{
return (double) item1.profit/(double) item1.weight > (double) item2.profit/ (double) item
2.weight;
});

for (int i=0; i<n; i++)
{
if (total_weight + ks.items[i].weight <=ks.capacity)
{

solution.push_back(ks.items[i]);
total_weight +=ks.items[i].weight;
}
}
    queue<Node> Q;
    Node u,v;
    Q.empty();
    u.level=-1;
    u.value=u.weight=0;
    Q.push(u);
    int maxProfit=0;

    while (!Q.empty())
    {
        u=Q.front();
        Q.pop();
        if (u.level == -1)
        {
        v.level =0;
        }
```

```
        else if (u.level != ks.items.size()-1)
        {

        v.level = u.level + 1;
        }
        v.weight = u.weight + ks.items[v.level].weight;
        v.value = u.value + ks.items[v.level].profit;
        v.id=ks.items[v.level].id;

        if (v.weight <=ks.capacity && v.value > maxProfit)
        {
          maxProfit = v.value;
          total_weight=v.weight;
        }
        v.bound = bound(v, ks);
          if (v.bound > maxProfit)
          {
          Q.push(v);
          }
          v.weight = u.weight;
          v.value = u.value;
          v.id=u.id;
          v.bound = bound(v,ks);
          if (v.bound > maxProfit)
          {
          Q.push(v);
          }
    }
 return solution;
}
```

Για τη δημιουργία του κάθε αρχείου με το αποτέλεσμα της διακλάδωσης και φραγής (συνολικό κέρδος, συνολικό βάρος, σύνολο επιλεγέντων αντικειμένων) έγινε κλήση της main (int argc, char **argv) ως εξής:

```
g++ -o knapsack knapsack.cpp

./knapsack 4
```

Δημιουργήθηκαν 320 αρχεία με τα αποτελέσματα της διακλάδωσης και φραγής. Για παράδειγμα για το στιγμιότυπο “problem_10_50_1_1_5.txt” δημιουργήθηκε το αρχείο problem_10_50_1_1_BNB.

4. Πειράματα

Για τις τέσσερις μεθόδους επίλυσης του προβλήματος δημιουργήθηκαν csv αρχεία ως εξής:

Δημιουργήθηκε csv αρχείο με τίτλο resultsGREEDY.csv με στήλες το συνολικό κέρδος, το συνολικό βάρος και τον χρόνο που απαιτήθηκε για την άπληστη μέθοδο σύμφωνα με τον κώδικα της παρακάτω συνάρτησης (void solve_GR()):

```
void solve_GR()
{
string csv_path = "../helloworld/resultsGREEDY.csv";
remove(csv_path.c_str());
std::ofstream file;
file.open(csv_path, std::ofstream::out);
string header="INSTANCE";
header.append(";VALUE_GR;WEIGHT_GR;EXECUTION_TIME_GR");
header.append("\n");
file<<header;
for (int n :{10, 50, 100, 500})
    for (int r: {50, 100, 500, 1000})
        for (int type: {1,2,3,4})
           for (int instance_id=1; instance_id<=5; instance_id++)
  { string problem_instance="problem_" + to_string(n) + "_" + to_string(r) + "_" + to_str
ing(type) +"_" + to_string(instance_id);
    string out= problem_instance;
    out.append(";");
file<<out;
string path_fn="../helloworld/";
    path_fn.append(problem_instance);
    path_fn.append("_5.txt");
    knapsack_problem ks =read_data(path_fn);
    std::vector<item> solution;
    auto t1 =chrono::high_resolution_clock::now();
    solution=greedy_solver(ks);
    string output;
      int total_weight=0;
    int total_profit=0;
    for (int i=0; i<solution.size(); i++)
{
total_weight += solution[i].weight;
total_profit +=solution[i].profit;

}
output=std:: to_string(total_profit) + ";" + std:: to_string(total_weight) +";";
   auto t2 =chrono::high_resolution_clock::now();
chrono::duration<double,milli>duration = t2-t1;
   string x= to_string(duration.count() / 1000.0);
  output.append(x);
   file<<output<<endl;
  }
  file.close();
}
```

Επίσης δημιουργήθηκε csv αρχείο με τίτλο resultsBRUTEFORCE.csv με στήλες το συνολικό κέρδος, το συνολικό βάρος και το χρόνο που απαιτήθηκε για την εξαντλητική απαρίθμηση συνδυασμών σύμφωνα με τον κώδικα της συνάρτησης (void solve_BF()).

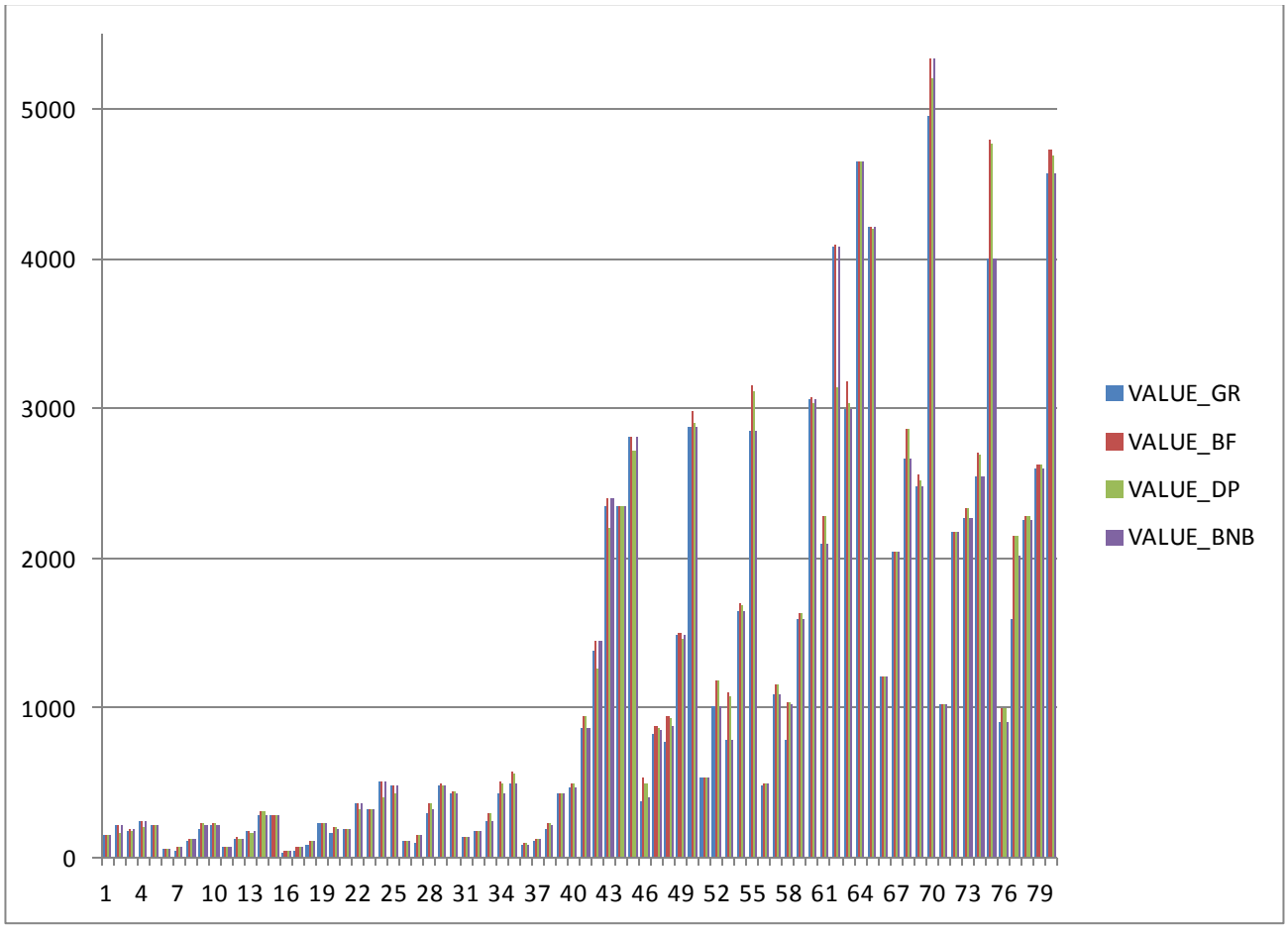
Δημιουργήθηκε csv αρχείο με τίτλο resultsDYNAMICPROG.csv με στήλες το συνολικό κέρδος, το συνολικό βάρος και τον χρόνο που απαιτήθηκε για τον δυναμικό προγραμματισμό σύμφωνα με τον κώδικα της συνάρτησης (void solve_DP()).

Δημιουργήθηκε csv αρχείο με τίτλο resultsBRANCHBOUND.csv με στήλες το συνολικό κέρδος, το συνολικό βάρος και τον χρόνο που απαιτήθηκε για την διακλάδωση και φραγή σύμφωνα με τον κώδικα της συνάρτησης (void solve_BNB()).

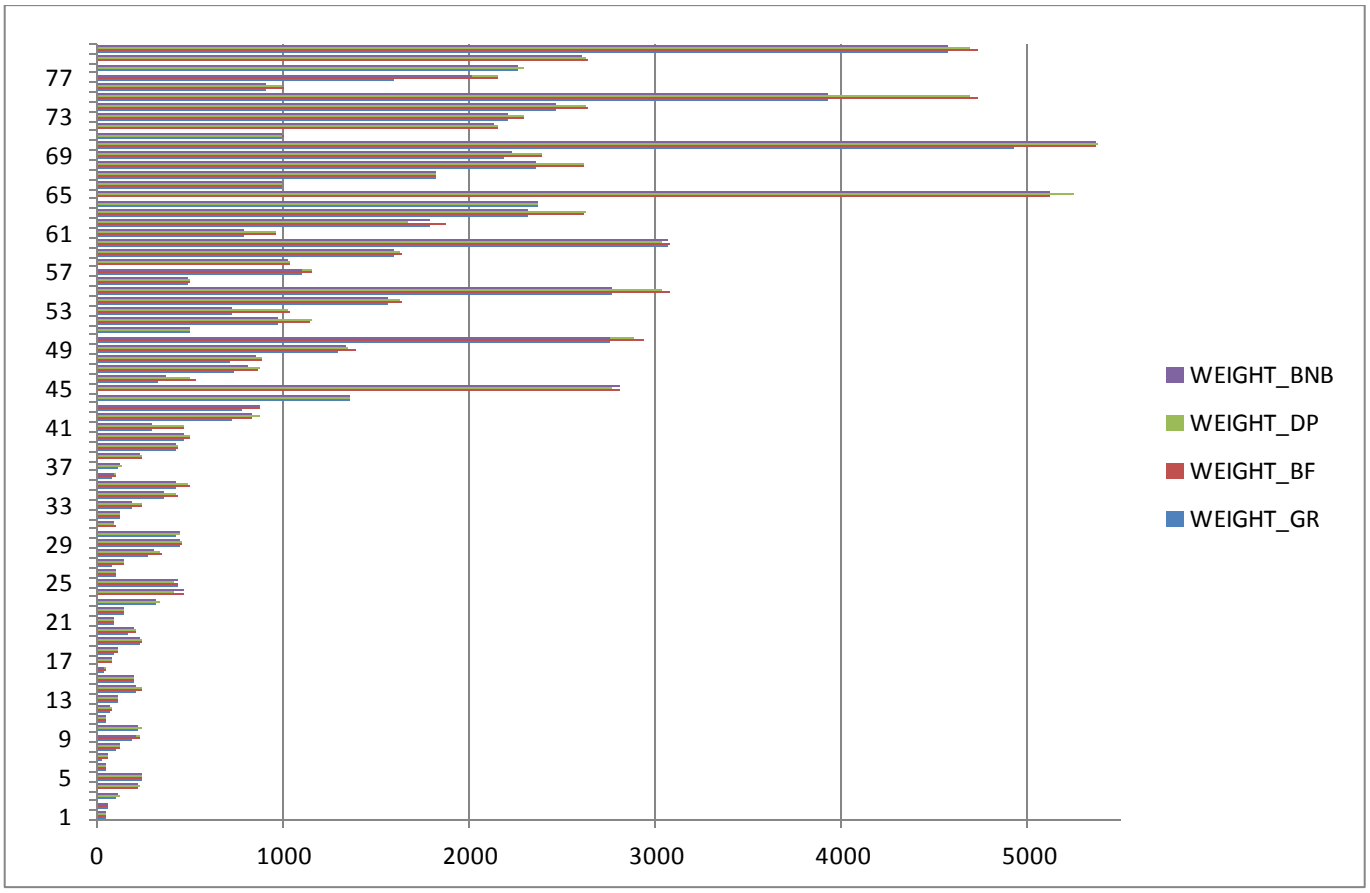
Τέλος, δημιουργήθηκε ένα συνολικό csv αρχείο results.csv με τα αποτελέσματα και των τεσσάρων επιλυτών του προβλήματος.

5. Πίνακες- Γραφήματα

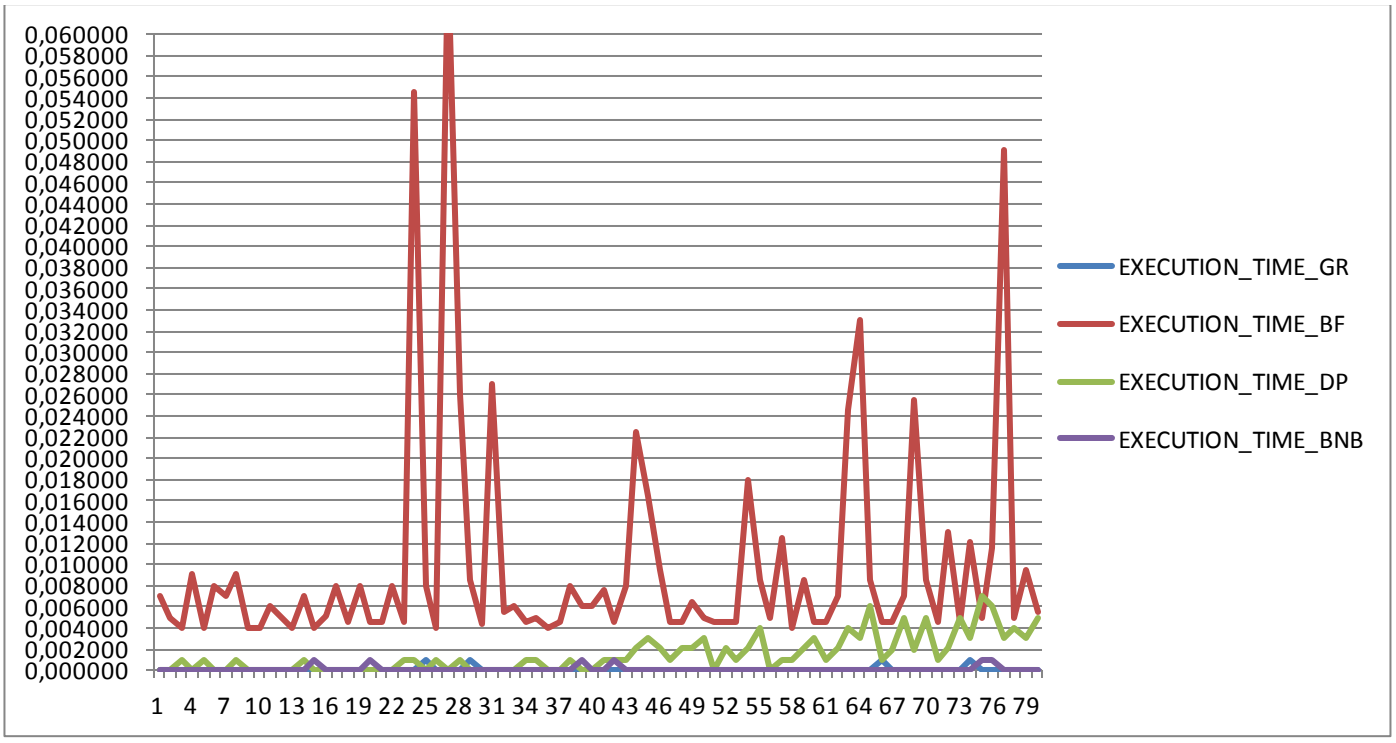
1) Στιγμιότυπα με n=10, r={50,100,500,1000}, t={1,2,3,4} – Συνολική αξία (value) και για τους τέσσερις επιλυτές του προβλήματος (80 στιγμιότυπα)



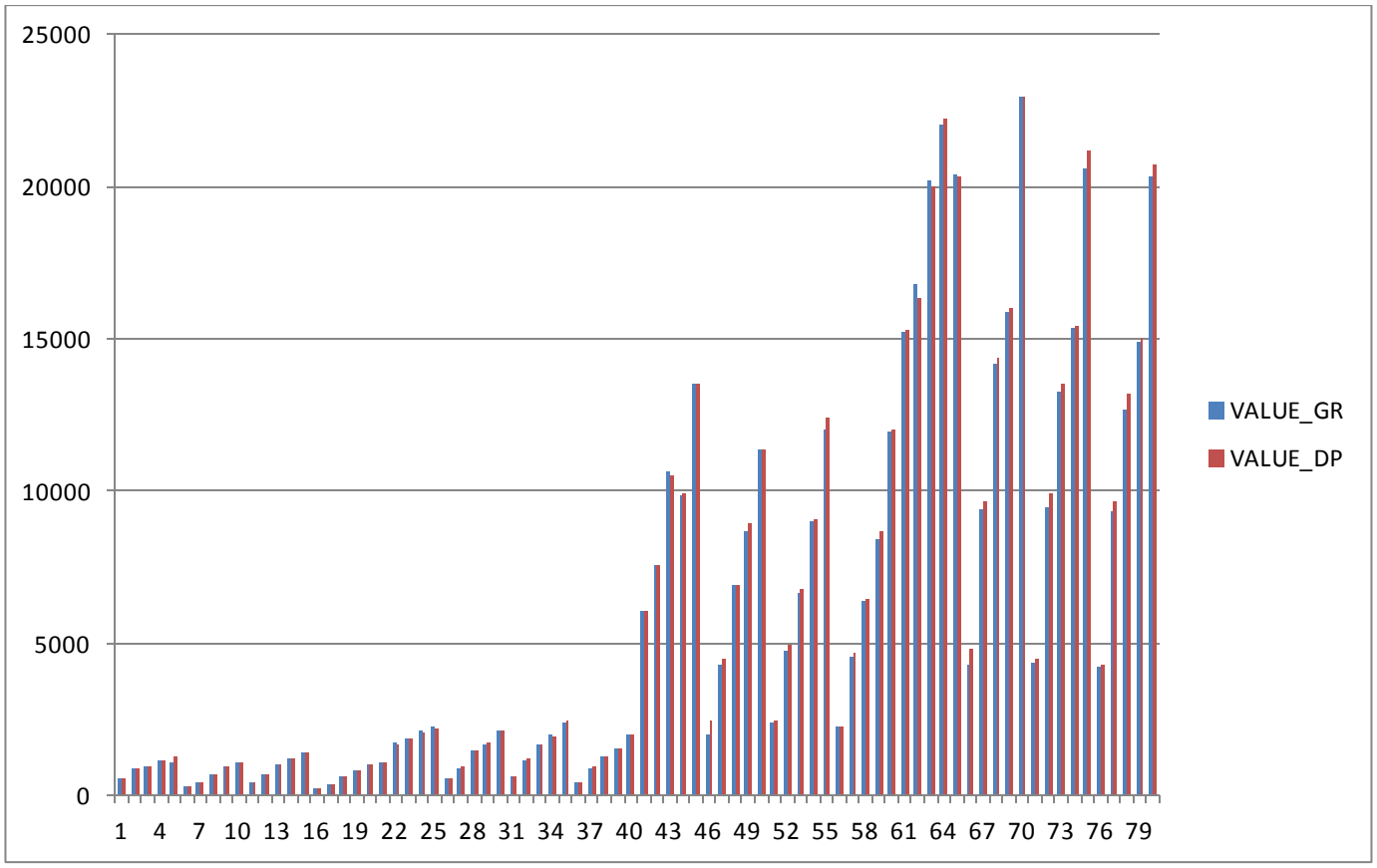
2) Στιγμιότυπα με n=10, r={50,100,500,1000}, t={1,2,3,4} – Συνολικό βάρος (weight) και για τους τέσσερις επιλυτές του προβλήματος (80 στιγμιότυπα)



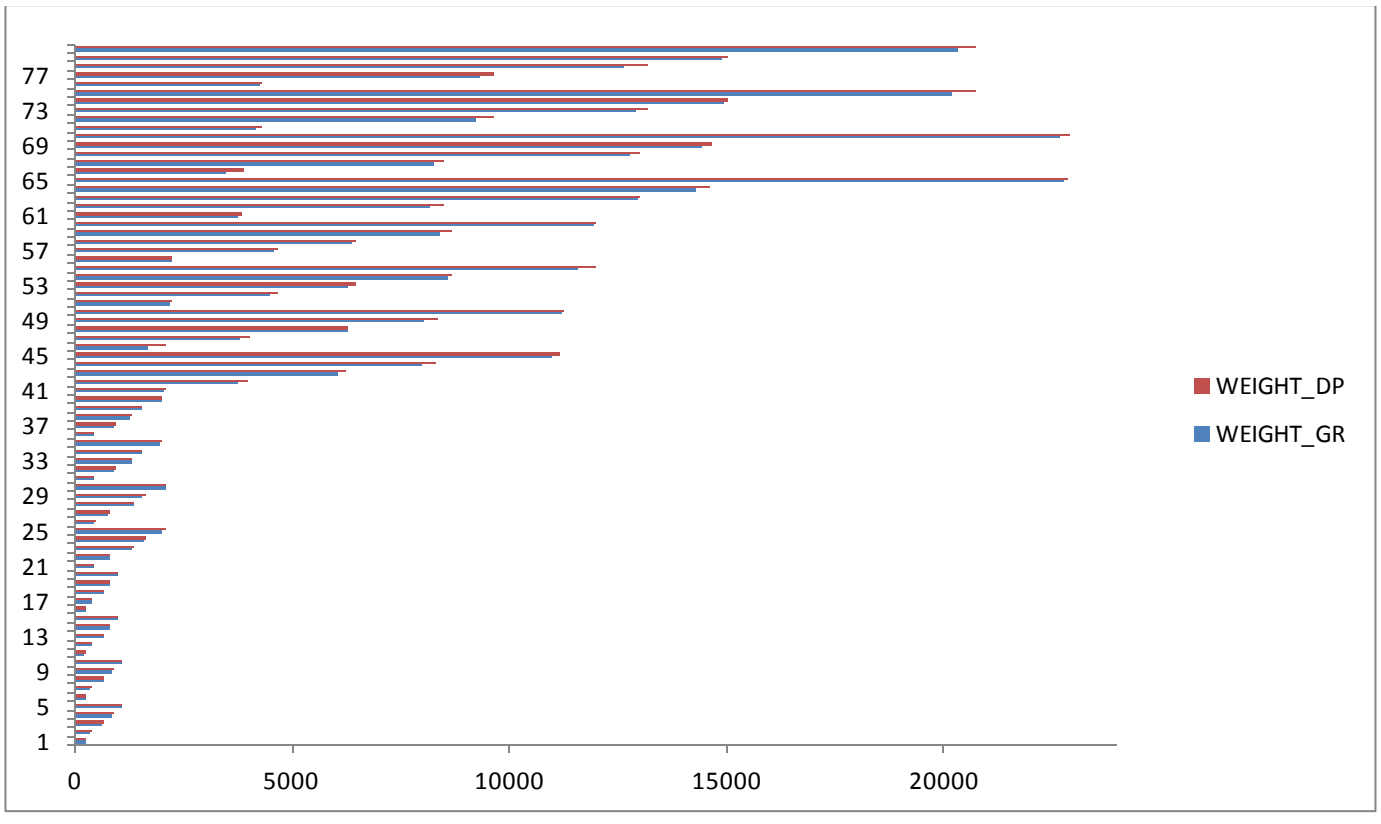
3) Στιγμιότυπα με n=10, r={50,100,500,1000}, t={1,2,3,4} – Συνολικός χρόνος που απαιτήθηκε (execution time) και για τους τέσσερις επιλυτές του προβλήματος (80 στιγμιότυπα)



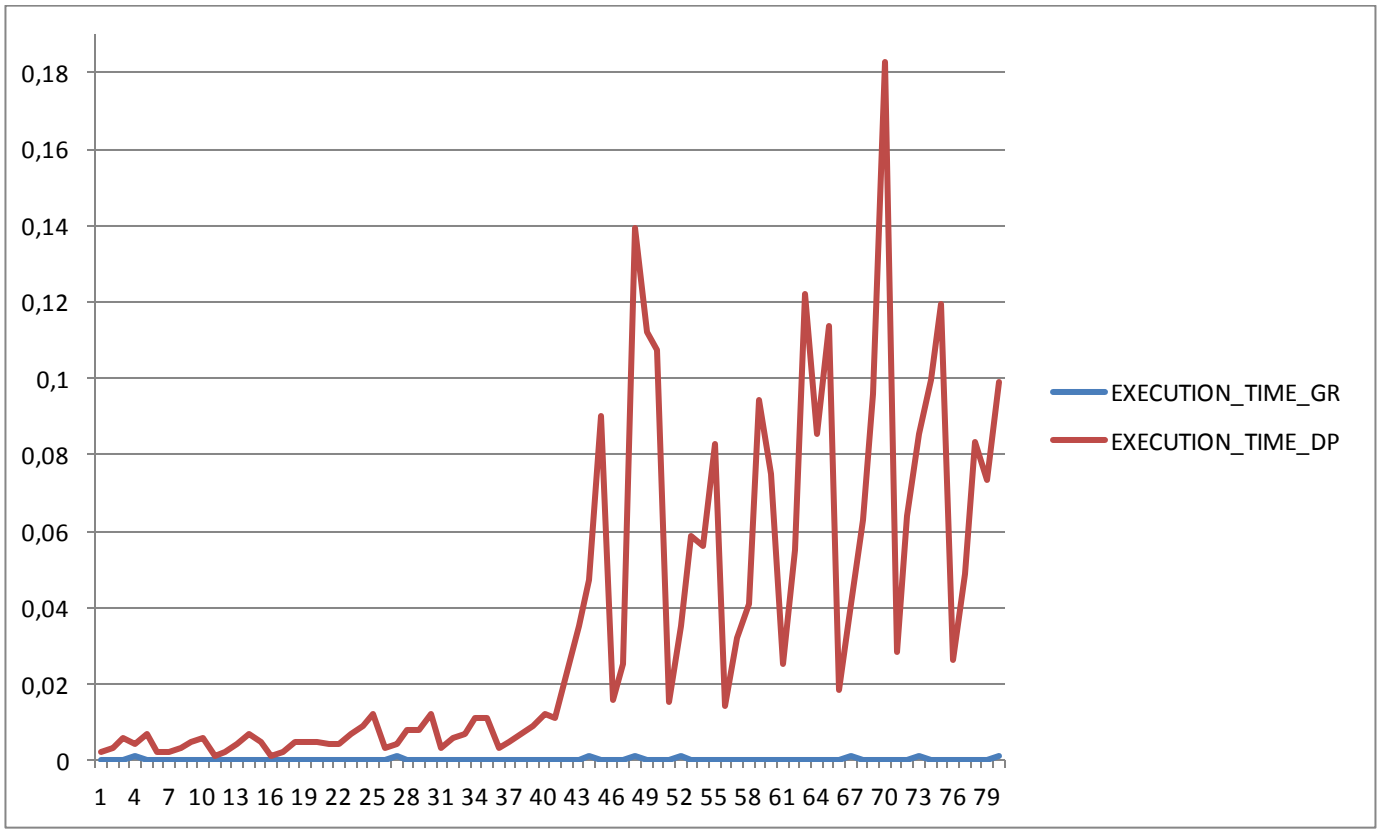
4) Στιγμιότυπα με n=50, r={50,100,500,1000}, t={1,2,3,4} – Συνολική αξία (value) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



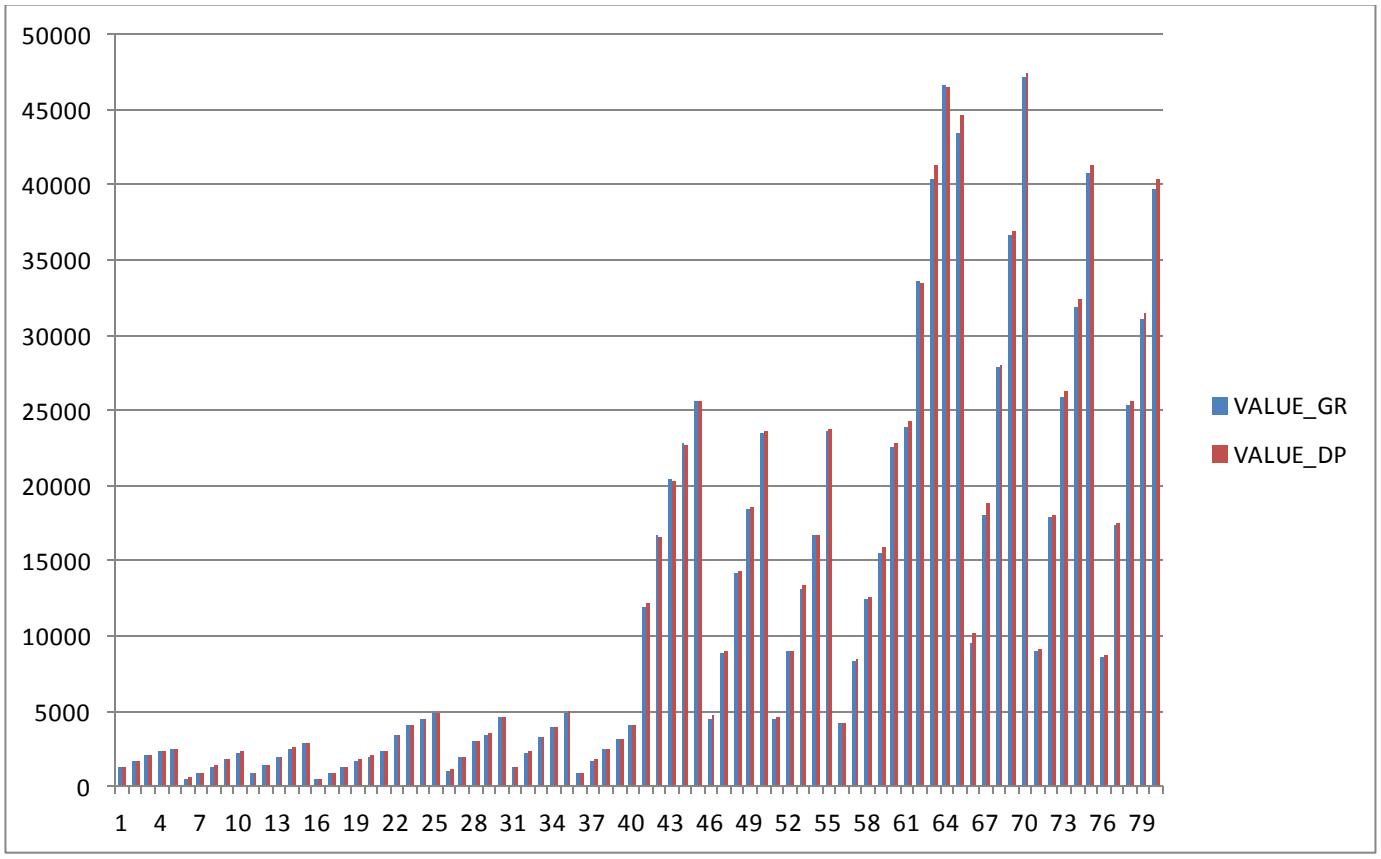
5) Στιγμιότυπα με n=50, r={50,100,500,1000}, t={1,2,3,4} – Συνολικό Βάρος (weight) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



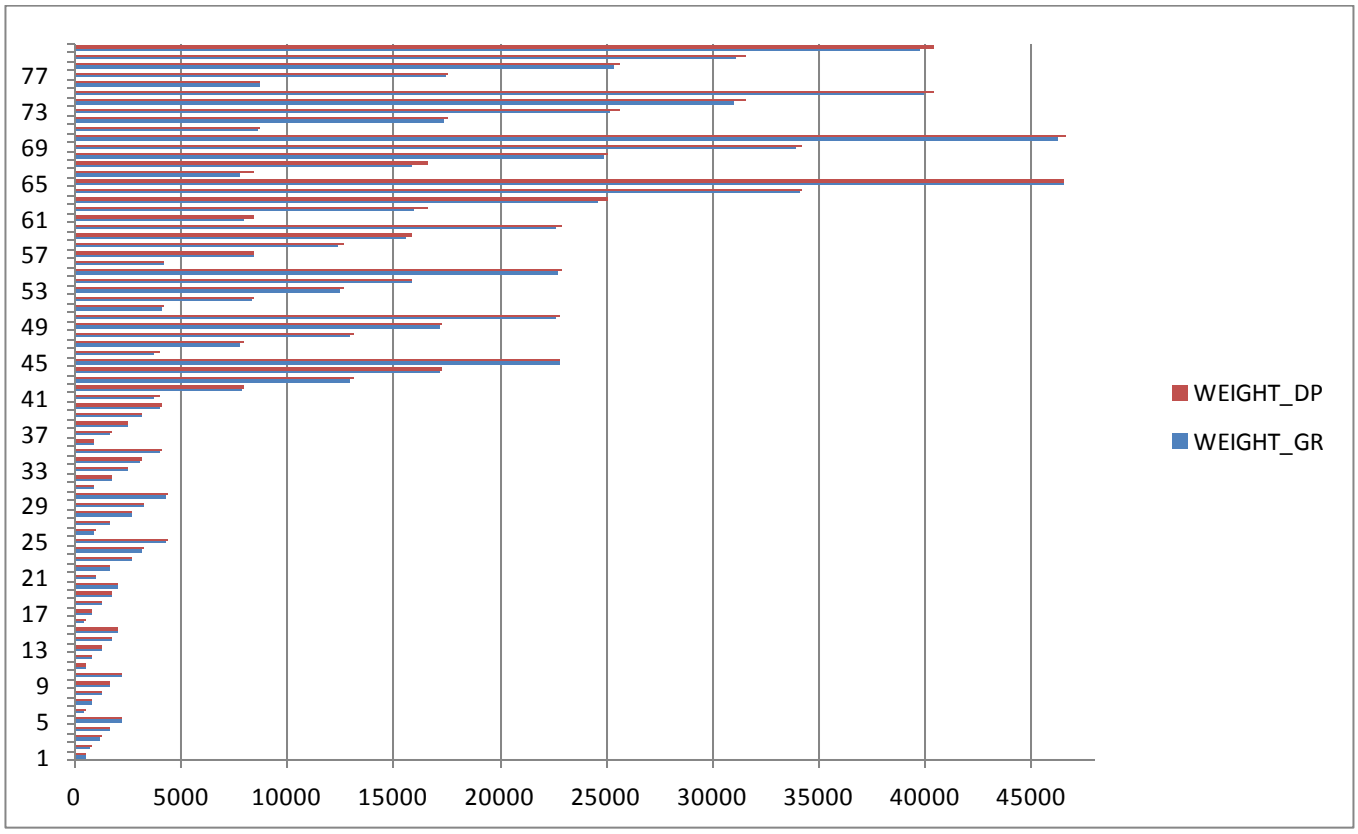
6) Στιγμιότυπα με n=50, r={50,100,500,1000}, t={1,2,3,4} – Συνολικός χρόνος που απαιτήθηκε (execution time) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



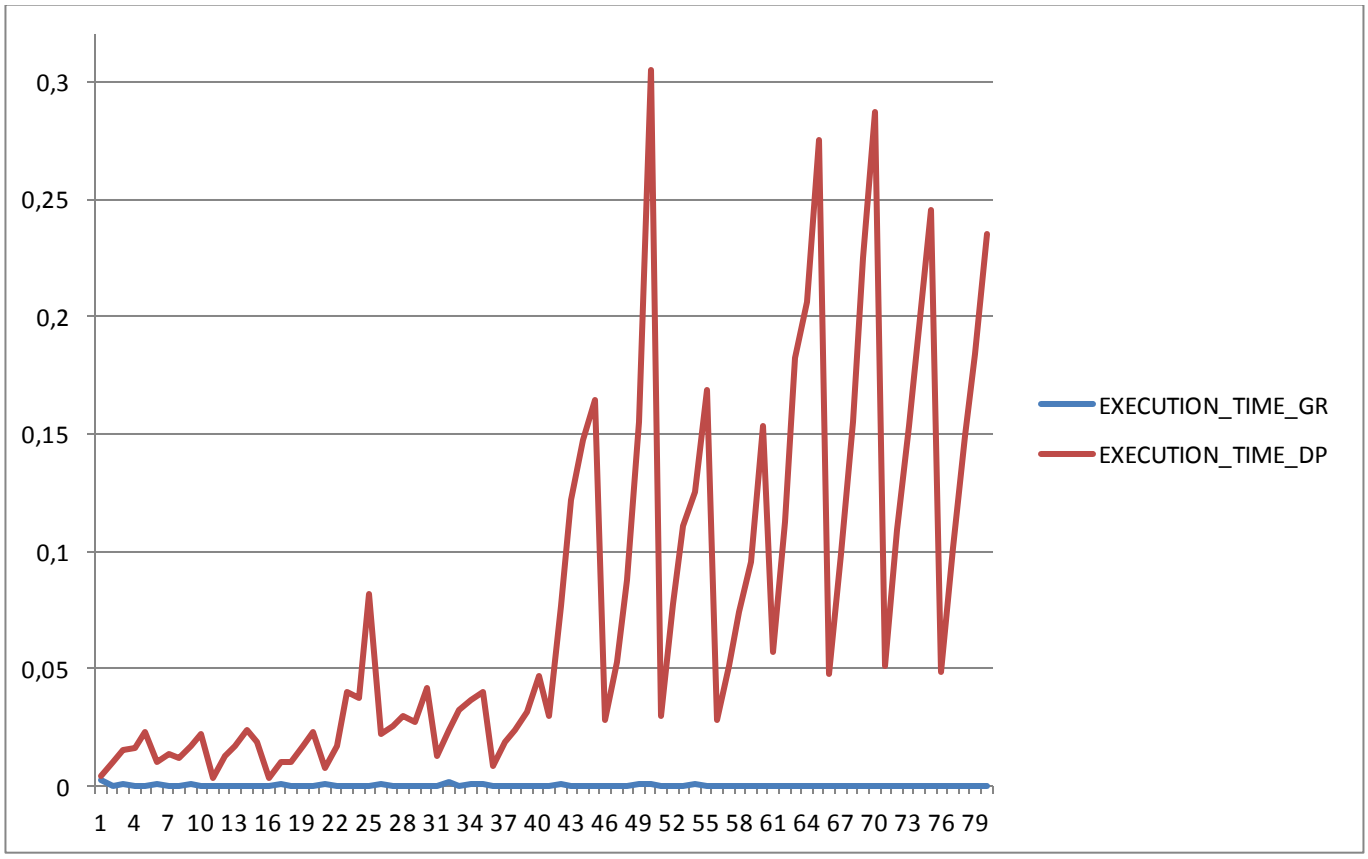
7) Στιγμιότυπα με n=100, r={50,100,500,1000}, t={1,2,3,4} – Συνολική αξία (value) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



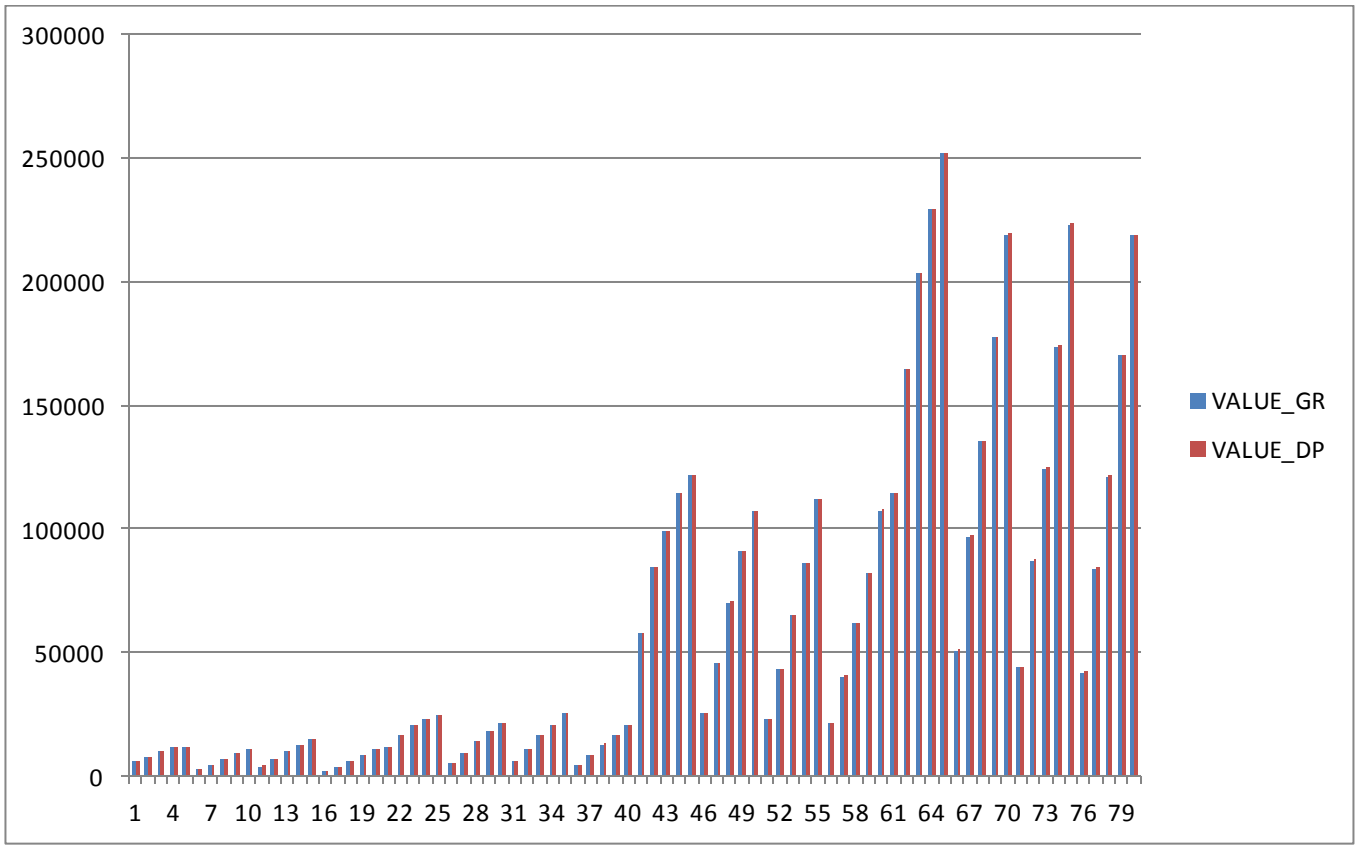
8) Στιγμιότυπα με n=100, r={50,100,500,1000}, t={1,2,3,4} – Συνολικό Βάρος (weight) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



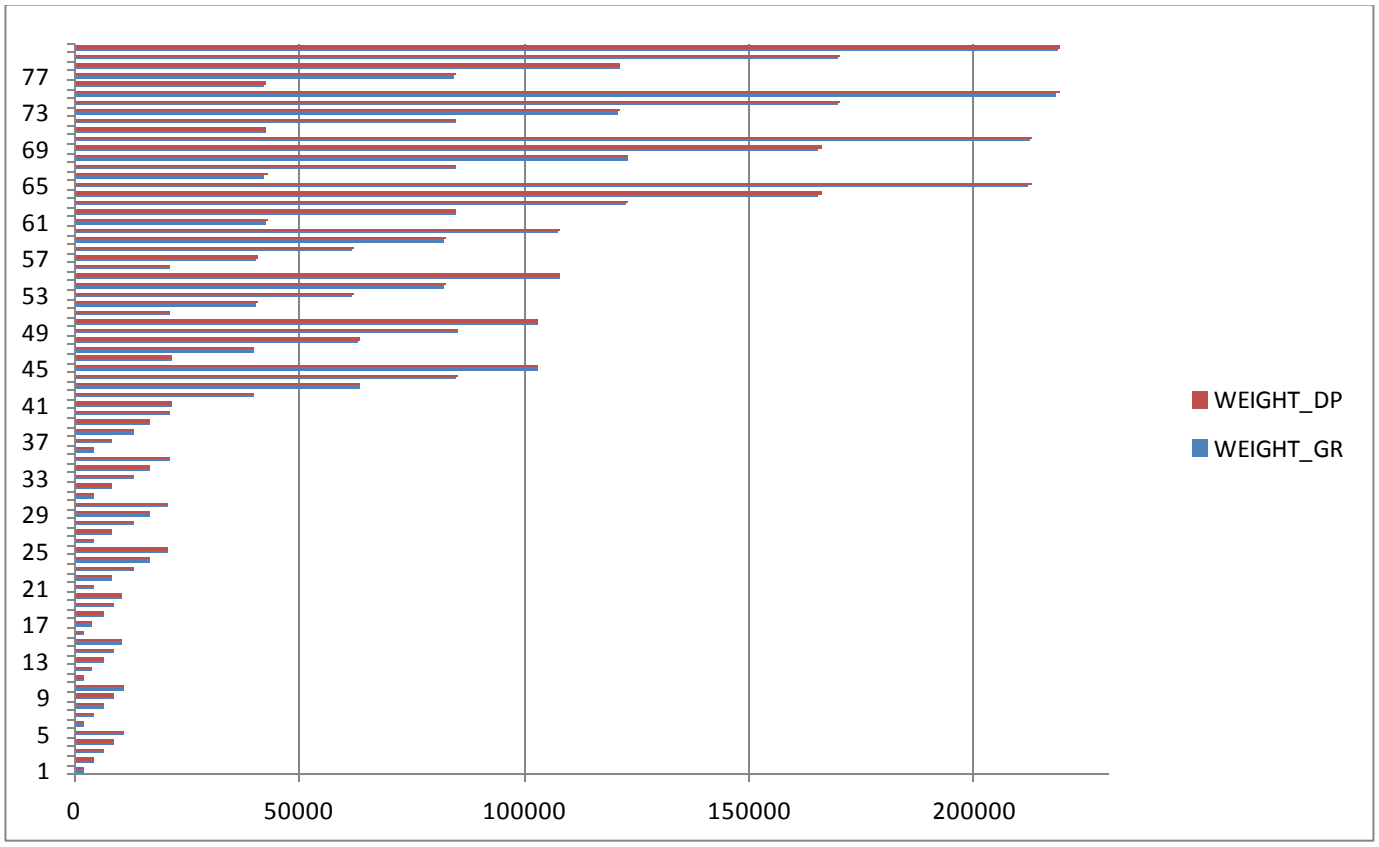
9) Στιγμιότυπα με n=100, r={50,100,500,1000}, t={1,2,3,4} – Συνολικός χρόνος που απαιτήθηκε (execution time) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



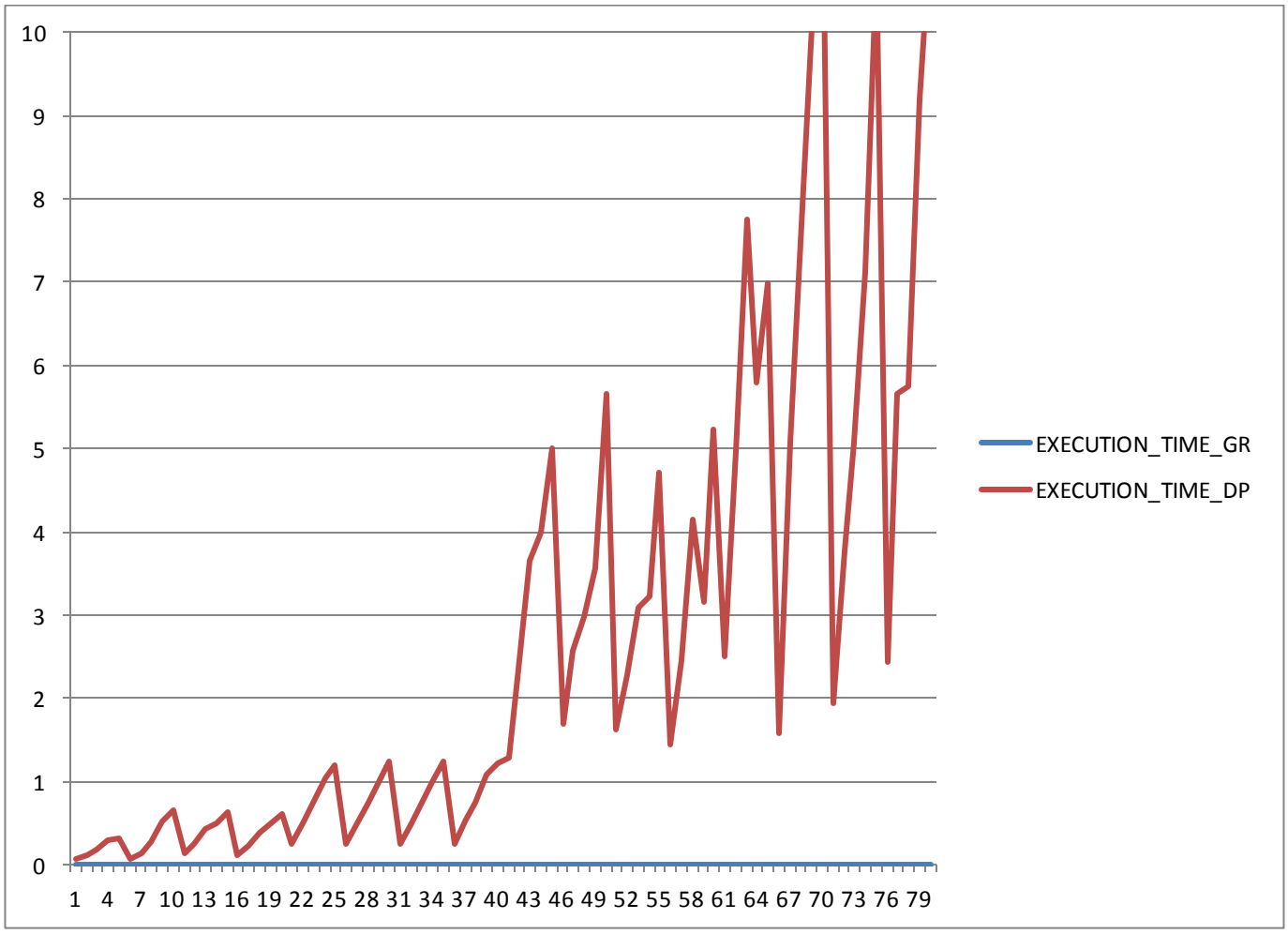
10) Στιγμιότυπα με n=500, r={50,100,500,1000}, t={1,2,3,4} – Συνολική αξία (value) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



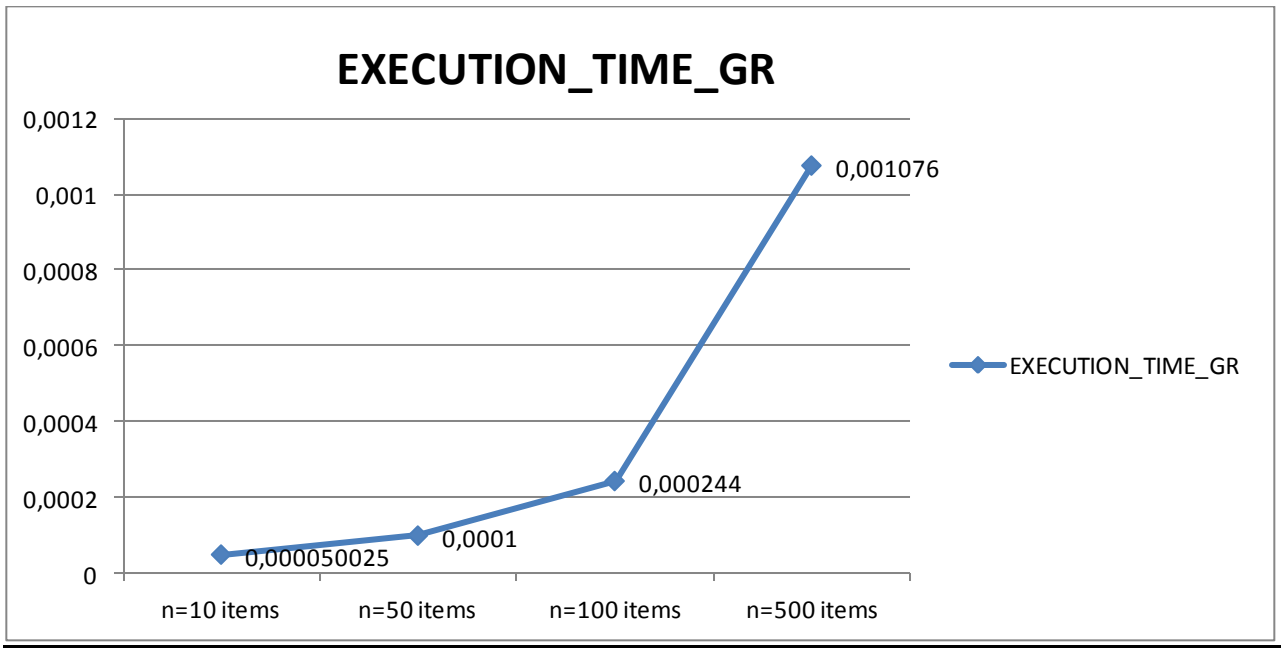
11) Στιγμιότυπα με n=500, r={50,100,500,1000}, t={1,2,3,4} – Συνολικό Βάρος (weight) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



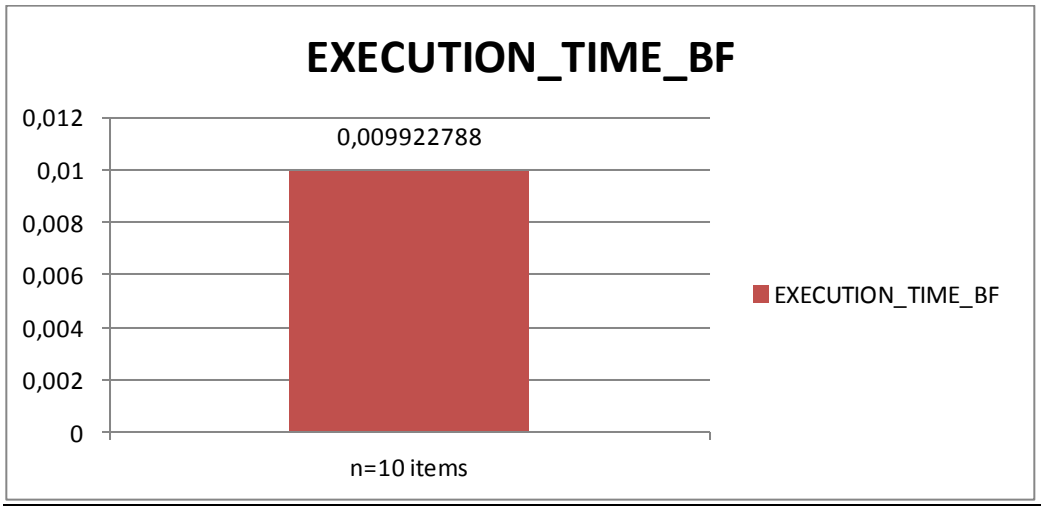
12) Στιγμιότυπα με n=500, r={50,100,500,1000}, t={1,2,3,4} – Συνολικός χρόνος που απαιτήθηκε (execution time) για τους δύο επιλυτές του προβλήματος (80 στιγμιότυπα)



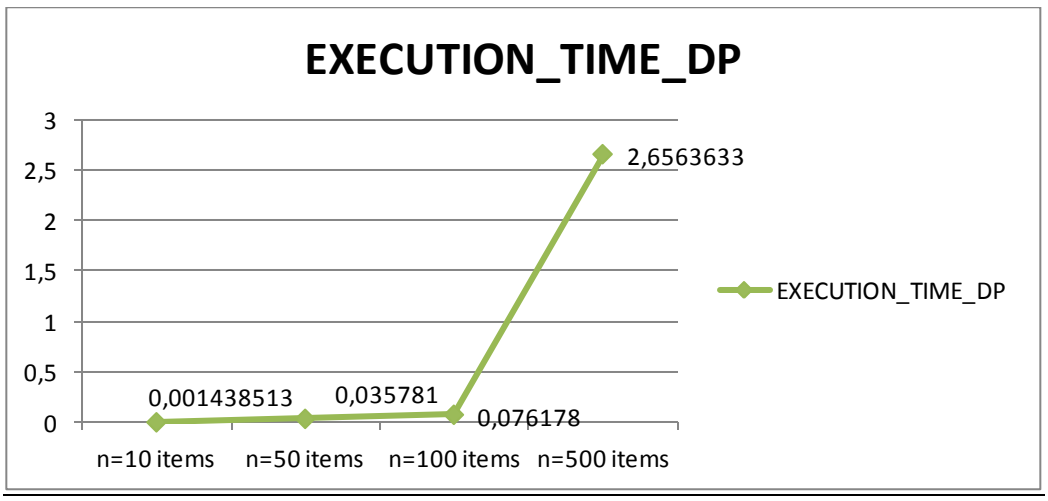
13) Μέσος χρόνος που απαιτήθηκε για την άπληστη μέθοδο (n={10,50,100,500} αντικείμενα)



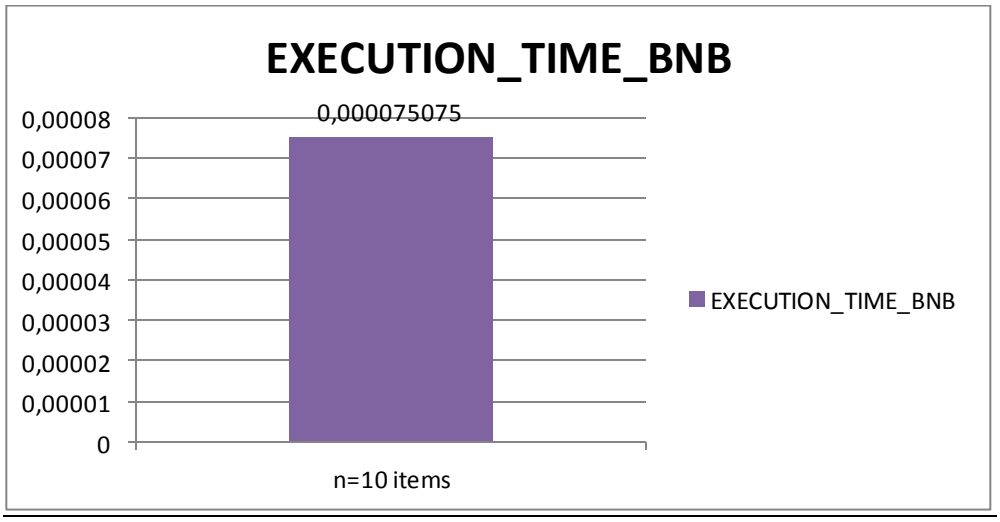
14) Μέσος χρόνος που απαιτήθηκε για την εξαντλητική απαρίθμηση συνδυασμών (n={10,50,100,500} αντικείμενα)



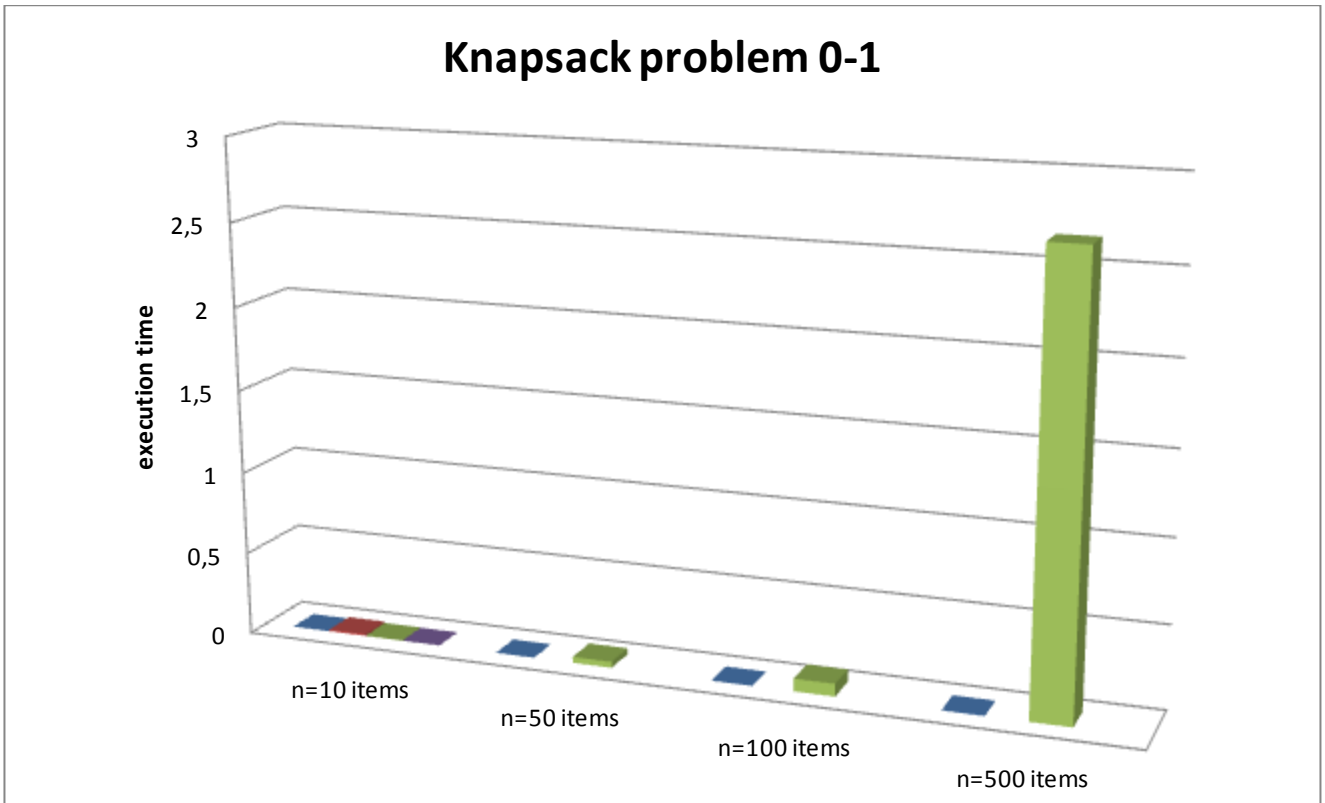
15) Μέσος χρόνος που απαιτήθηκε για τον δυναμικό προγραμματισμό (n={10,50,100,500} αντικείμενα)



16) Μέσος χρόνος που απαιτήθηκε για την διακλάδωση και φραγή (n={10,50,100,500} αντικείμενα)



17) Μέσος χρόνος που απαιτήθηκε και για τους τέσσερις επιλυτές του προβλήματος (n={10,50,100,500} αντικείμενα)



	n=10 items	n=50 items	n=100 items	n=500 items
EXECUTION_TIME_GR	0,000050025	0,0001	0,000244	0,001076
EXECUTION_TIME_BF	0,009922788			
EXECUTION_TIME_DP	0,001438513	0,035781	0,076178	2,6563633
EXECUTION_TIME_BNB	0,000075075			

6. Συμπεράσματα

Συνολικά, στην παρούσα εργασία αναπτύχθηκε και παρουσιάστηκε η επίλυση του προβλήματος 0-1 σακιδίου (Knapsack problem) με τους τέσσερις από τους συνολικά έξι προτεινόμενους τρόπους (άπληστη μέθοδος, εξαντλητική μέθοδος, δυναμικός προγραμματισμός, διακλάδωση και φραγή) και τα αποτελέσματα της μελέτης κατέδειξαν τα πλεονεκτήματα και τα μειονεκτήματα του καθενός.

Η άπληστη μέθοδος είναι πολύ πιο γρήγορη από τη εξαντλητική απαρίθμηση συνδυασμών και είναι μια καλή προσέγγιση του προβλήματος. Το συγκεκριμένο πρόβλημα μπορεί να αντιμετωπιστεί με αυτή τη μέθοδο γιατί δεν υπάρχουν προβλήματα μεγάλου χρόνου εκτέλεσης όπως φαίνεται στο γράφημα 13. Όμως αυτή η μέθοδος δεν βρίσκει πάντα τη βέλτιστη λύση, αλλά την προσεγγίζει ικανοποιητικά πολλές φορές. Η άπληστη μέθοδος δίνει άμεσα αποτελέσματα, αλλά ενδεχομένως να μη δίνει τη βέλτιστη απόδοση.

Η εξαντλητική απαρίθμηση συνδυασμών είναι ο πιο προφανής τρόπος επίλυσης ενός προβλήματος και έχει μειονεκτήματα και πλεονεκτήματα. Το βασικό μειονέκτημα της είναι ότι όσο

προσθέτουμε παραμέτρους στο πρόβλημα τόσο πιο αργή γίνεται σαν μέθοδος (γραφήματα 3, 14). Έχει δηλαδή μεγάλη πολυπλοκότητα γιατί ελέγχονται όλες οι δυνατές επιλογές. Άρα στιγμιότυπα με αριθμό αντικειμένων 50, 100 και 500 χρειαζόταν χρονικό διάστημα πολύ μεγαλύτερο από το χρονικό περιθώριο εκτέλεσης των 10 δευτερολέπτων για να βρεθεί το καλύτερο αποτέλεσμα και γι' αυτό δεν πήραμε τιμές για το συνολικό κέρδος, το συνολικό βάρος και το συνολικό χρόνο που απαιτήθηκε. Το πλεονέκτημα της τεχνικής αυτής είναι ότι είναι εύκολα υλοποιήσιμη, κατανοητή και δίνει βέλτιστη λύση. Επειδή αποτελεί τον πιο απλό τρόπο επίλυσης ενός προβλήματος τέτοιου είδους, είναι και εύκολο να γίνει εφαρμόσιμη.

Ο δυναμικός προγραμματισμός βασίζεται στην επίλυση υποπροβλημάτων για να επιλύσει το αρχικό πρόβλημα και εγγυάται βέλτιστη λύση αλλά κοστίζει σημαντικά σε υπολογιστικό χρόνο αν και σε κάθε περίπτωση είναι σίγουρα αρκετά μικρότερος από τη μέθοδο της εξαντλητικής απαρίθμησης συνδυασμών, στην οποία περιλαμβάνονται όλες οι δυνατές λύσεις ενός προβλήματος. Βέβαια όπως φαίνεται και στα γραφήματα 12, 15 και 17 για στιγμιότυπα με 500 αντικείμενα αυξάνονταν σημαντικά το χρονικό διάστημα για την εύρεση της βέλτιστης λύσης.

Η μέθοδος Branch and Bound είναι καλύτερη από τον δυναμικό προγραμματισμό και βρίσκεται στην κατηγορία με τους ακριβείς αλγορίθμους (γραφήματα 1, 2 , 3). Μια χονδρική εκτίμηση μπορεί να γίνει πολύ γρήγορα αλλά δεν περιορίζει αρκετά τον αριθμό των ανιχνευμένων κόμβων. Όμως μια ακριβής εκτίμηση ενώ μας επιτρέπει να αποφύγουμε μεγάλο μέρος του χώρου αναζήτησης, ο υπολογισμός της επιβαρύνει σημαντικά τον συνολικό χρόνο. Και αυτό είχε ως αποτέλεσμα να μην πάρουμε τιμές για το συνολικό κέρδος, το συνολικό βάρος και το συνολικό χρόνο για στιγμιότυπα με αριθμό αντικειμένων 50, 100 και 500 που χρειαζόταν χρονικό διάστημα πολύ μεγαλύτερο από το χρονικό περιθώριο εκτέλεσης των 10 δευτερολέπτων.

Δυστυχώς, το πρόβλημα του 0-1 σακιδίου δεν επιλύθηκε ούτε με ακέραιο προγραμματισμό όπου θα γινόταν χρήση του επιλυτή προβλημάτων ακέραιου προγραμματισμού GLPK [GNU19] μέσω του λογισμικού ORTools, ούτε με τον εξειδικευμένο επιλυτή που διαθέτει το λογισμικό ORTools [PF19c]. Πραγματοποιήθηκε η εγκατάσταση του Google OR-Tools (κατέβασμα του zip αρχείου και αποσυμπίεση), η μεταφορά του αποσυμπιεσμένου καταλόγου (or-tools_VisualStudio2017-64bit_v7.4.7247) στο κατάλογο στον οποίο είχε κατέβει το αποθετήριο (C:\git\uoi_algorithms_and_complexity) και η μετονομασία του or-tools_VisualStudio2017-64bit_v7.4.7247 σε ortools (για Windows). Όμως υπήρξε πρόβλημα κατά την εγκατάσταση του Visual Studio Community 2017. Μετά την ολοκλήρωση εγκατάστασής του στον υπολογιστή, όταν άνοιγε η εφαρμογή για να τρέξουν τα demo και ο κώδικας κολλούσε συνεχώς με αποτέλεσμα να μην καταστεί δυνατή η επίλυση του προβλήματος με τους συγκεκριμένους επιλυτές.

Επίσης, υπήρξαν δυσκολίες στη δημιουργία των στιγμιότυπων με τη βοήθεια του generator.c αρχείου. Δεν ήταν εφικτή η δημιουργία κώδικα σε c++ για την αυτόματη δημιουργία όλων των στιγμιότυπων (χρήση εμφωλευμένων for και του generator.c αρχείου) με αποτέλεσμα τα στιγμιότυπα να δημιουργηθούν το καθένα ξεχωριστά όπως περιγράφεται στην αρχή της εργασίας.

Ακόμη, υπήρξαν πολλά προβλήματα κατά τη δημιουργία του csv αρχείου με τα αποτελέσματα του κάθε επιλυτή. Δεν ήταν δυνατόν να δημιουργηθεί μια συνάρτηση όπου θα συμπεριλαμβάνει τα αποτελέσματα και των τεσσάρων επιλυτών σε ένα csv αρχείο, πρόβλημα το οποίο επιλύθηκε με τη δημιουργία τεσσάρων ξεχωριστών συναρτήσεων για κάθε επιλυτή.

Τέλος, δυσκολίες υπήρξαν στη δημιουργία των γραφημάτων έτσι ώστε να παρουσιαστούν τα αποτελέσματα και των τεσσάρων επιλυτών. Έπρεπε να γίνει ομαδοποίηση των στιγμιότυπων κατά τέτοιο τρόπο έτσι ώστε να είναι εμφανείς όσο είναι δυνατόν οι διαφορές στην συνολική αξία, στο συνολικό βάρος και στο συνολικό χρόνο των τεσσάρων επιλυτών. Όσον αφορά το χρόνο υπολογίστηκε και ο μέσος χρόνος που απαιτήθηκε από τους επιλυτές για τις τέσσερις περιπτώσεις δηλαδή για στιγμιότυπα με 10, 50 ,100 και 500 αντικείμενα όπως φαίνεται στα γραφήματα 13, 14, 15, 16 και 17.

Αν ήταν εφικτή η αντιμετώπιση των παραπάνω δυσκολιών θα μπορούσαν να βγουν πιο ασφαλή και χρήσιμα συμπεράσματα μέσω της σύγκρισης και των έξι επιλυτών. Παρόλα αυτά καταβλήθηκε κάθε δυνατή προσπάθεια για να γίνει μια σαφής και ολοκληρωμένη παρουσίαση των δυνατοτήτων επίλυσης του προβλήματος 0-1 σακιδίου.