

# ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΕΣ ΕΡΓΑΣΙΕΣ ΑΛΓΟΡΙΘΜΟΙ

## ΙΑΚΩΒΟΣ ΕΥΔΑΙΜΩΝ 3130059

ΣΗΜΕΙΩΣΗ: IN ECLIPSE FILE-> IMPORT->Existing Projects into workspace-> Select root directory (Αχρείστη σημείωση αλλά καλού κακού την αναφέρω)

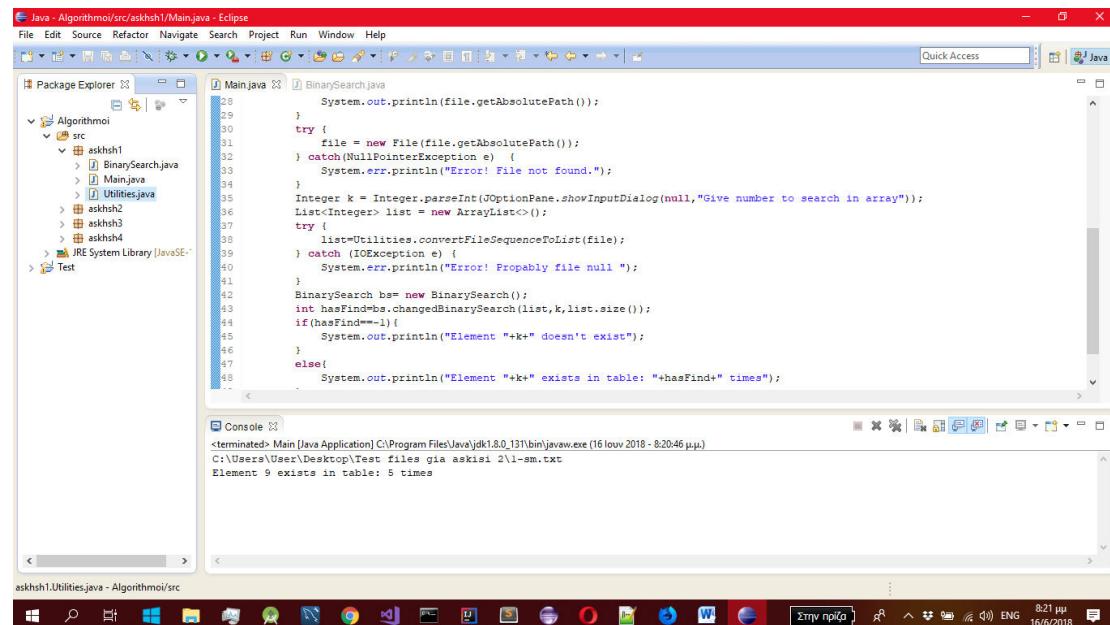
### ΑΣΚΗΣΗ 1

Για το σκοπό αυτής της άσκησης υλοποιούμε μια τροποποιημένη δυαδική αναζήτηση αφού διαβάσουμε το δοθέν αρχείο. Το αρχείο αποθηκεύεται σε μία List. Το αρχείο που μας δίνεται είναι ταξινομημένο οπότε αποθηκεύεται και στην List ταξινομημένο και μετά εφαρμόζουμε μία λίγο τροποποιημένη δυαδική αναζήτηση η οποία βρίσκει την πρώτη θέση στην λίστα του στοιχείου που έχει δοθεί από τον χρήστη και αφού βρει την πρώτη θέση στην οποία εμφανίζεται το δοθέν στοιχείο και αφού όντως υπάρχει στην λίστα μετά εντοπίζει και την τελευταία θέση στην οποία βρίσκεται το δοθέν στοιχείο μέσα στη λίστα μας. Τέλος επιστρέφει το αποτέλεσμα στον χρήστη με το πλήθος των φορών που βρίσκεται στη λίστα το στοιχείο που δόθηκε από τον χρήστη προς διερεύνηση. Η πολυπλοκότητα του αλγορίθμου μας είναι  $O(\log N)$  για να βρει το πλήθος των φορών που εμφανίζεται μέσα στη λίστα το στοιχείο που διερευνάμε. Είναι  $O(\log N)$  καθώς ουσιαστικά εφαρμόζουμε 2 δυαδικές αναζητήσεις για την πρώτη θέση που εμφανίζεται το στοιχείο και για την τελευταία θέση  $O(\log N + \log N) = O(\log N)$ . Το γεγονός ότι το αρχείο που διαβάζουμε μας δίνει έναν πίνακα(λίστα) ταξινομημένο μας βοηθάει να κάνουμε δυαδική αναζήτηση και να γνωρίζουμε ότι το στοιχείο που αναζητούμε θα ξαναεμφανίζεται στον πίνακα(λίστα) ακριβώς μετά από την θέση του πίνακα(λίστα) που το πρωτοεντοπίσαμε. Για την μέθοδο που υλοποιεί το διάβασμα του αρχείου έχουμε πολυπλοκότητα  $O(N)$ , όπου  $N$  ουσιαστικά είναι το πλήθος των στοιχείων του πίνακα καθώς ουσιαστικά διαβάζουμε την 1<sup>η</sup>

γραμμή του αρχείου και μετά κάθε στοιχείο της 1<sup>ης</sup> γραμμής το εισάγουμε στην λίστα.

## ΑΠΟΤΕΛΕΣΜΑΤΑ

Έστω ότι ο χρήστης έδωσε τον στοιχείο με τιμή 9.



```
28      System.out.println(file.getAbsolutePath());
29  }
30  try {
31      file = new File(file.getAbsolutePath());
32  } catch (NullPointerException e) {
33      System.err.println("Error! File not found.");
34  }
35  Integer k = Integer.parseInt(JOptionPane.showInputDialog(null, "Give number to search in array"));
36  List<Integer> list = new ArrayList<>();
37  try {
38      list=Utilities.convertFileSequenceToList(file);
39  } catch (IOException e) {
40      System.err.println("Error! Probably file null ");
41  }
42  BinarySearch bs= new BinarySearch();
43  int hasFind=bs.changedBinarySearch(list,k,list.size());
44  if (hasFind==1){
45      System.out.println("Element "+k+" doesn't exist");
46  }
47  else{
48      System.out.println("Element "+k+" exists in table: "+hasFind+" times");
49  }
```

Console Output:

```
<terminated> Main [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (16 Ιουν 2018 - 8:20:46 μ.μ.)
C:\Users\User\Desktop\Test files gia askis1 2\1-sm.txt
Element 9 exists in table: 5 times
```

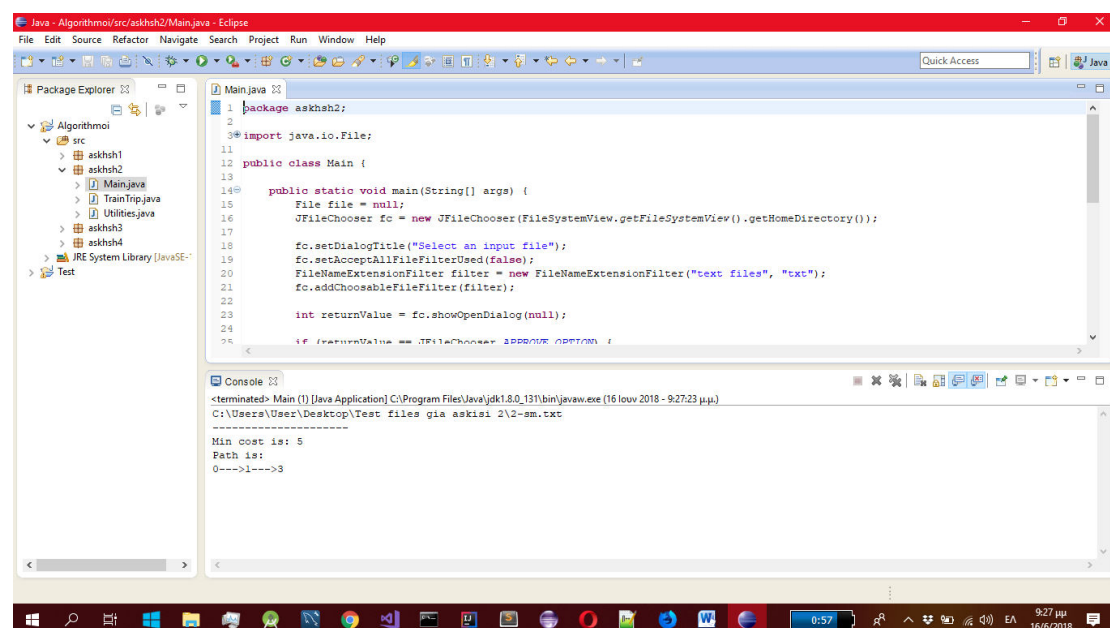
Υ.Γ. Για την επιλογή του αρχείου που θα διαβάσει το πρόγραμμά μας καθώς και για το ποιο στοιχείο θα δώσει στο πρόγραμμα ο χρήστης για να το αναζητήσει στη λίστα μας χρησιμοποιώ γραφικό περιβάλλον. Ενώ το αποτέλεσμα του αλγορίθμου εκτυπώνεται στην console.

## ΑΣΚΗΣΗ 2

Για την υλοποίηση του αλγορίθμου αυτού αρχικά διαβάζουμε το αρχείο εισόδου με τα κόστη να πάμε από τον σταθμό  $i$  στον  $j$ , όπου  $i < j$ . Όπως είναι λογικό για τις θέσεις του πίνακα όπου  $i \geq j$  οι τιμές του πίνακα είναι 0. Διαβάζουμε τον δυσδιάστατο πίνακα όπου μας δίνεται σε μορφή αρχείου txt και το τοποθετούμε σε μία `List<List<Integer>>` όπου κάθε θέση της λίστας έχει μία άλλη λίστα με τις τιμές για το κόστος της μετάβασης από τον σταθμό  $i$  στον  $j$ . Για να βρούμε το ελάχιστο κόστος για την μετακίνηση από τον σταθμό 1 στον  $N$  (ουσιαστικά από τον 0 στον  $N-1$ ) χρησιμοποιούμε έναν μονοδιάστατο πίνακα μεγέθους  $N$  όπου κρατάει την μικρότερη τιμή για να μετακινηθούμε προς τον σταθμό  $i$  με  $0 \leq i \leq N$ . Αρχικά θέτουμε όλες τις τιμές αυτού του μονοδιάστατου

πίνακα ίσες με άπειρο και μετά σαρώνουμε την λίστα μας για να φτάσουμε τον μονοδιάστατο πίνακα που θα έχει τα μικρότερα κόστος για την μετάβαση προς τον σταθμό  $i$ . Τέλος, σαρώνουμε αυτόν τον μονοδιάστατο πίνακα για να βρούμε το μονοπάτι της μετάβασης από τον σταθμό 1 προς τον  $N$  (ουσιαστικά 0 προς  $N-1$ ) όπου ελέγχουμε και προσθέτουμε στο μονοπάτι τις θέσεις του μονοδιάστατου πίνακα όπου οι τιμές τους είναι μικρότερες από ότι η τιμή του πίνακα στη θέση που αφορά τον σταθμό προορισμού μας. Μετά από αυτή την διαδικασία επιστρέφουμε αυτό το μονοπάτι και το ελάχιστο δυνατό κόστος προς τον σταθμό προορισμού μας. Για να θέσουμε τιμές στον μονοδιάστατο πίνακα που κρατάει τα ελάχιστα κόστη προς κάθε σταθμό γίνεται με την εξής λογική. Το ελάχιστο κόστος για τον σταθμό 1 (θέση 0 του πίνακα) είναι 0 για τον σταθμό 2 (θέση 1 του πίνακα) είναι το κόστος από τον 1 να πάμε στον 2 για τον σταθμό 3 (θέση 2 του πίνακα) είναι το ελάχιστο είτε του κόστους να πάμε από τον 1 στον 3 είτε το ελάχιστο κόστος προς τον 2 και μετά το κόστος να πάμε από τον 2 στον 3. Το ίδιο γίνεται και με τους υπόλοιπους σταθμούς. Η πολυπλοκότητα του αλγορίθμου είναι  $O(N)$  για να τα θέσουμε όλες τις τιμές του πίνακα άπειρο,  $O(N)$  για να βρούμε το μονοπάτι και  $O(N^2)$  για να θέσουμε τις σωστές τιμές στον μονοδιάστατο πίνακα που κρατάει τα ελάχιστα κόστη προς κάθε σταθμό. Για να διαβάσουμε το αρχείο η πολυπλοκότητα της μεθόδου που χρησιμοποιούμε είναι επίσης  $O(N^2)$  καθώς είναι  $N$  οι γραμμές που διαβάζουμε με  $N$  στοιχεία σε κάθε γραμμή. Άρα η συνολική πολυπλοκότητα του αλγορίθμου μας είναι  $O(N^2)$ .

## ΑΠΟΤΕΛΕΣΜΑΤΑ



The screenshot shows the Eclipse IDE with a Java project named 'Algorithmmoi'. The 'Main.java' file is open, displaying the following code:

```
1 package askhsh2;
2
3 import java.io.File;
4
5 public class Main {
6
7     public static void main(String[] args) {
8         File file = null;
9         JFileChooser fc = new JFileChooser(FileSystemView.getFileSystemView().getHomeDirectory());
10
11         fc.setDialogTitle("Select an input file");
12         fc.setAcceptAllFileFilterUsed(false);
13         FileNameExtensionFilter filter = new FileNameExtensionFilter("text files", "txt");
14         fc.addChoosableFileFilter(filter);
15
16         int returnValue = fc.showOpenDialog(null);
17
18         if (returnValue == JFileChooser.APPROVE_OPTION) {
19             file = fc.getSelectedFile();
20         }
21     }
22 }
```

The console output shows the execution of the program:

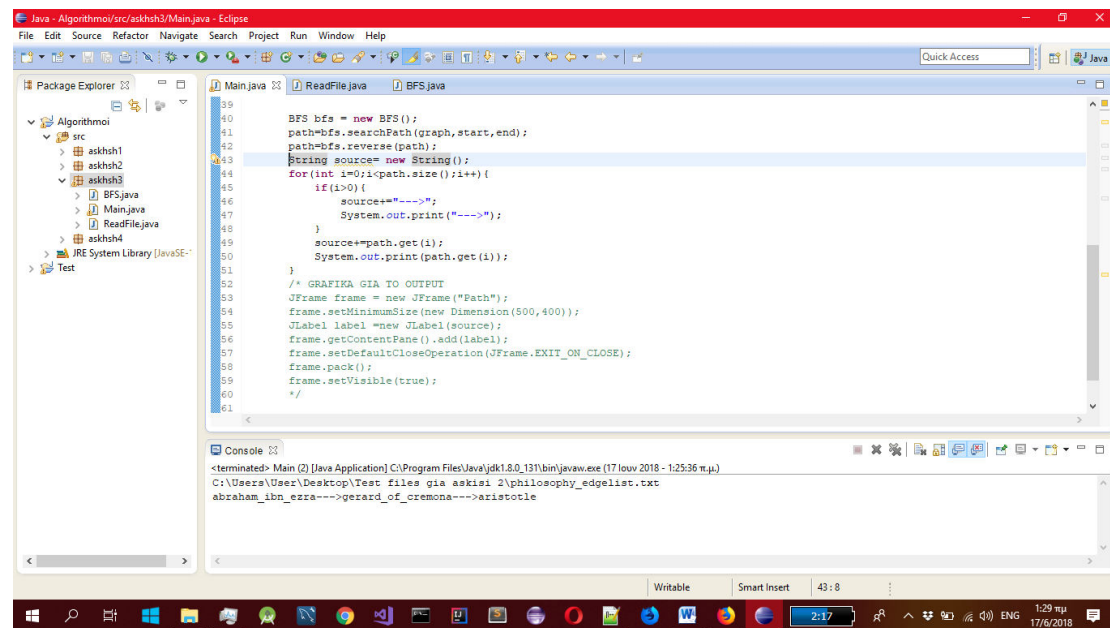
```
<terminated> Main (1) [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (16 Ιουν 2018 - 9:27:23 μ.μ.)
C:\Users\User\Desktop\Test files\askhsh2\2-sm.txt
Min cost is: 5
Path is:
0--->1--->3
```

Υ.Γ. Για την επιλογή του αρχείου που θα διαβάσει το πρόγραμμά μας χρησιμοποιώ γραφικό περιβάλλον. Ενώ το αποτέλεσμα του αλγορίθμου εκτυπώνεται στην console.

### ΑΣΚΗΣΗ 3

Για τον σκοπό της εργασίας 3 φτιάχνουμε μία κλάση που διαβάζει το αρχείο μας και το αποθηκεύει σε ένα hash map με κλειδί το όνομα του κόμβου (τύπου String ) και τιμή τη λίστα των γειτόνων του κόμβου αυτού (τύπου List<String>). Οπότε ουσιαστικά διαβάζουμε το αρχείο γραμμή-γραμμή και δημιουργούμε τον γράφο όπου όποιου κόμβου το κλειδί δεν υπάρχει ήδη τον δημιουργεί και άμα υπάρχει ήδη του προσθέτει στη λίστα γειτνίασής του τον κόμβο με τον οποίο εννώνεται. Επιπλέον, πρέπει να τονίσω ότι όταν διαβάζει μία γραμμή από το αρχείο δημιουργεί τους κόμβους και για τους δύο κόμβους που διαβάζει σε κάθε γραμμή(άμα δεν υπάρχουν) και προσθέτει και στους δύο στην λίστα γειτνίασής τους τον κατάλληλο κόμβο. Έπειτα, για τους σκοπούς της άσκησης υλοποιούμε τον αλγόριθμο BFS για να βρούμε το συντομότερο μονοπάτι ανάμεσα σε δύο δοθέντες κόμβους(αρχικό κόμβο-κόμβο προορισμού) από τον χρήστη. Για κάθε κόμβο που ανακαλύπτουμε τον βάζουμε σε ένα hash map με κλειδί το παιδί και τιμή τον γονιό. Με αυτό τον τρόπο όταν βρεθεί το μονοπάτι μας θα ξεκινώντας από τον κόμβο προορισμού θα βρούμε τον γονιό του και μετά τον γονιό του γονιού του και ούτε καθεξής μέχρι να φτάσουμε στον κόμβο από όπου ξεκινήσαμε παίρνοντας έτσι το μονοπάτι μας. Ο BFS λόγω ότι εξερευνεί τον γράφο κατά πλάτος εγγυάται ότι θα βρει το συντομότερο μονοπάτι. Για τον BFS χρησιμοποιούμε μία ουρά και ένα hash map που μαρκάρει όσους κόμβους έχουμε επισκεφτεί για να μην τους ξαναεξερευνήσουμε και πέσουμε σε κύκλους. Η πολυπλοκότητα του αλγορίθμου μας είναι  $O(|V|+|E|)$  λόγω του BFS και  $O(V)$  για να αρχικοποιήσουμε τον hash map visited. Επίσης, θέλω  $O(V)$  και για να βρω το μονοπάτι. Επιπροσθέτως θέλω χρόνο  $O(N)$ , όπου  $N$  οι γραμμές του αρχείου, για να διαβάσω το αρχείο. Οπότε, ο συνολικός μου χρόνος είναι  $O(|V|+|E|)$ .

## ΑΠΟΤΕΛΕΣΜΑΤΑ



The screenshot shows the Eclipse IDE with a Java project named 'Algorithmmoi'. The 'Main.java' file is open, displaying a BFS algorithm implementation. The code includes a 'BFS' class, a 'searchPath' method, and a 'main' method that reads a file 'philosophy\_edgelist.txt' and prints the resulting path. The console output shows the path: 'abraham\_ibn\_ezra-->gerard\_of\_cremone-->aristotle'.

```
39 BFS bfs = new BFS();
40 path=bfs.searchPath(graph,start,end);
41 path=bfs.reverse(path);
42 String source= new String();
43 for(int i=0;i<path.size();i++){
44     if(i>0){
45         source+="->";
46         System.out.print("->");
47     }
48     source+=path.get(i);
49     System.out.print(path.get(i));
50 }
51 /* GRAFIKA GIA TO OUTPUT
52 JFrame frame = new JFrame("Path");
53 frame.setMinimumSize(new Dimension(500,400));
54 JLabel label =new JLabel(source);
55 frame.getContentPane().add(label);
56 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
57 frame.pack();
58 frame.setVisible(true);
59 */
60
61
```

Console Output:

```
<terminated> Main (2) [Java Application] C:\Program Files\Java\jdk1.8.0_131\bin\javaw.exe (17 Iou 2018 - 1:25:36 π.μ.)
C:\Users\User\Desktop\Test files gia askisi 2\philosophy_edgelist.txt
abraham_ibn_ezra-->gerard_of_cremone-->aristotle
```

Υ.Γ. Για την επιλογή του αρχείου που θα διαβάσει το πρόγραμμά μας καθώς και για το ποιο αρχικό κόμβο και κόμβο προορισμού θα δώσει στο πρόγραμμα ο χρήστης για να το αναζητήσει στη λίστα μας χρησιμοποιώ γραφικό περιβάλλον. Ενώ το αποτέλεσμα του αλγορίθμου εκτυπώνεται στην Main console. Σε αυτήν την άσκηση έχω αφήσει σε σχόλια ώστε το αποτέλεσμα να μπορεί να εκτυπωθεί και σε γραφικό περιβάλλον.

## ΑΣΚΗΣΗ 4

Για την άσκηση αυτή χρησιμοποιούμε για το διάβασμα του αρχείου την ίδια υλοποίηση που χρησιμοποιήσαμε και στην άσκηση 3 για να διαβάσουμε το αρχείο. Μετά υλοποιούμε 2 διαφορετικές προσεγγίσεις για λύσουμε το πρόβλημά μας. Η πρώτη προσέγγιση αποτελεί μία προσέγγιση εξαντλητικής αναζήτησης ενώ η δεύτερη προσέγγιση αποτελεί μία άπλειστη προσέγγιση. Για την πρώτη προσέγγιση αρχικά ελέγχουμε άμα έστω ένας κόμβος εννώνεται με όλους τους υπόλοιπους κόμβους(vertex cover). Αν δεν εννώνεται θέτουμε μία μεταβλητή k=2 και μέσα σε έναν επαναληπτικό βρόγχο δημιουργούμε όλους τους

δυνατούς συνδυασμούς όλων κόμβων από το Hash map(γράφο μας) συνδυάζοντας κάθε φορά  $k$  μεταξύ τους και ελέγχουμε άμα αυτό το υποσύνολο των  $k$  κόμβων συνδέεται με ακμές με όλους τους άλλους κόμβους. Άμα δεν συνδέεται παίρνουμε τον επόμενο συνδυασμό των  $k$  κόμβων και ελέγχουμε και αυτό. Άμα, πάλι κανένα από τα υποσύνολα των  $k$  κόμβων δεν συνδέεται με ακμές με όλους τους άλλους κόμβους τότε αυξάνουμε κατά ένα το  $k$  και ξαναεπαναλαμβάνουμε τις προηγούμενες διαδικασίες. Συνεχίζουμε κατά αυτό τον τρόπο μέχρι να βρούμε ένα υποσύνολο που συνδέεται με ακμές με όλους τους κόμβους ή μέχρι το  $k$  να γίνει ίσο με τον αριθμό των κόμβων. Για την δεύτερη προσέγγιση αρχικά παίρνουμε τον κόμβο με τους περισσότερους γείτονες και αφαιρούμε αυτό τον κόμβο από το Hash map(γράφο) και τον αφαιρούμε επίσης και από τις λίστες γειτνίασης όλων των κόμβων. Από τις λίστες γειτνίασης όλων των κόμβων αφαιρούμε επίσης και τους γείτονες του κόμβου που μόλις εξερευνήσαμε. Προσθέτουμε σε μία ArrayList τον κόμβο και ελέγχουμε άμα αυτός ο κόμβος συνδέεται με ακμές με όλους τους άλλους κόμβους μας. Άμα συνδέεται τον επιστρέφουμε αλλιώς παίρνουμε τον επόμενο κόμβο που έχει τους περισσότερους γείτονες, τον αφαιρούμε αυτό τον κόμβο από το Hash map(γράφο) και τον αφαιρούμε επίσης και από τις λίστες γειτνίασης όλων των κόμβων. Από τις λίστες γειτνίασης όλων των κόμβων αφαιρούμε επίσης και τους γείτονες του κόμβου που μόλις εξερευνήσαμε. Προσθέτουμε στην ArrayList τον κόμβο και ελέγχουμε άμα οι κόμβοι που έχουμε στην ArrayList συνδέονται με ακμές με όλους τους άλλους κόμβους μας. Άμα συνδέονται επιστρέφουμε την ArrayList αλλιώς συνεχίζουμε την ίδια διαδικασία μέχρι να βρούμε έναν συνδυασμό κόμβων που εννώνονται με ακμές με όλους τους άλλους κόμβους του Hash map(γράφου). Στην πρώτη προσέγγιση λόγω όλων των συνδυασμών, που έχουμε όπου για κάθε  $k$  έχουμε  $V!/(k!(V-k)!)$  συμδυασμούς, η πολυπλοκότητα θα είναι  $O((V!/(k!(V-k)!))^k)$ . Ενώ για την δεύτερη προσέγγιση θα έχουμε πολυπλοκότητα  $O(V^2 * E)$ .

## ΑΠΟΤΕΛΕΣΜΑΤΑ

```
HashMap<String,List<String>> graph = rf.getGraph();
System.out.println("Size of graph: "+graph.size());
System.out.println("-----BRUTE FORCE APPROACH-----");
long startTimeBrute = System.nanoTime();
BruteForceSearch brute = new BruteForceSearch();
brute.search(graph);

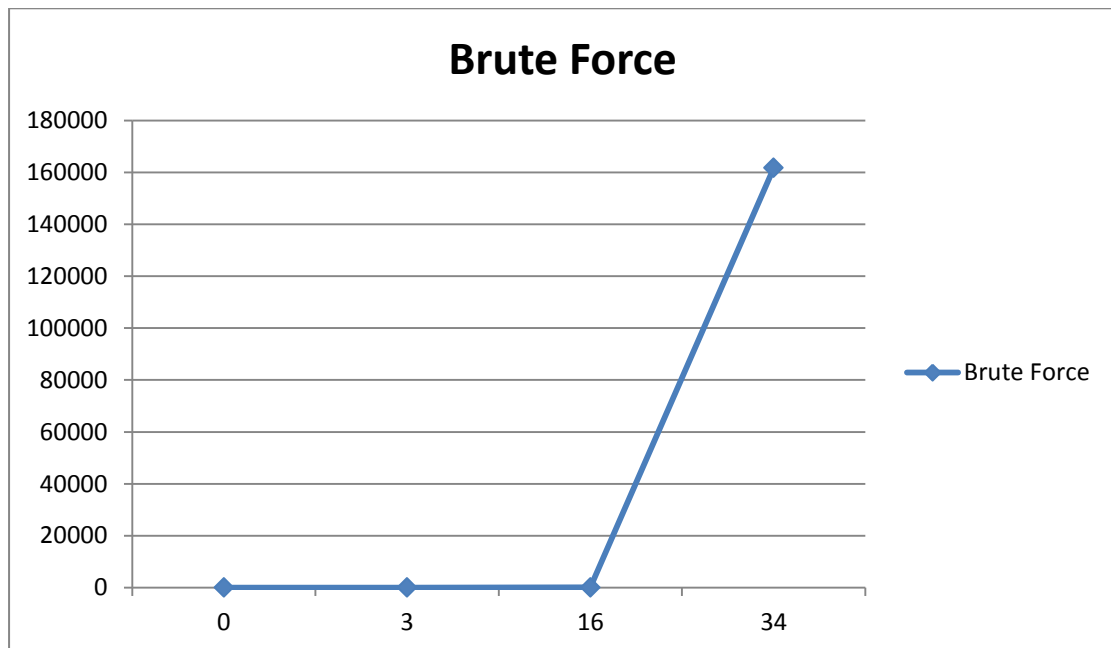
-----BRUTE FORCE APPROACH-----
NODES:
25
34
17
1
Elapsed time in milliseconds of brute force: 117.992821 milliseconds
-----END OF BRUTE FORCE-----
-----GREEDY APPROACH-----
NODES:
34
1
32
6
Elapsed time in milliseconds of greedy: 1.469286 milliseconds
-----END OF GREEDY-----
```

Ο άξονας Χ(οριζόντιος) είναι το πλήθος των κόμβων και ο άξονας Υ(κατακόρυφος) είναι ο χρόνος εκτέλεσης του αλγορίθμου σε milliseconds.

## Ο ΑΠΛΗΣΤΟΣ

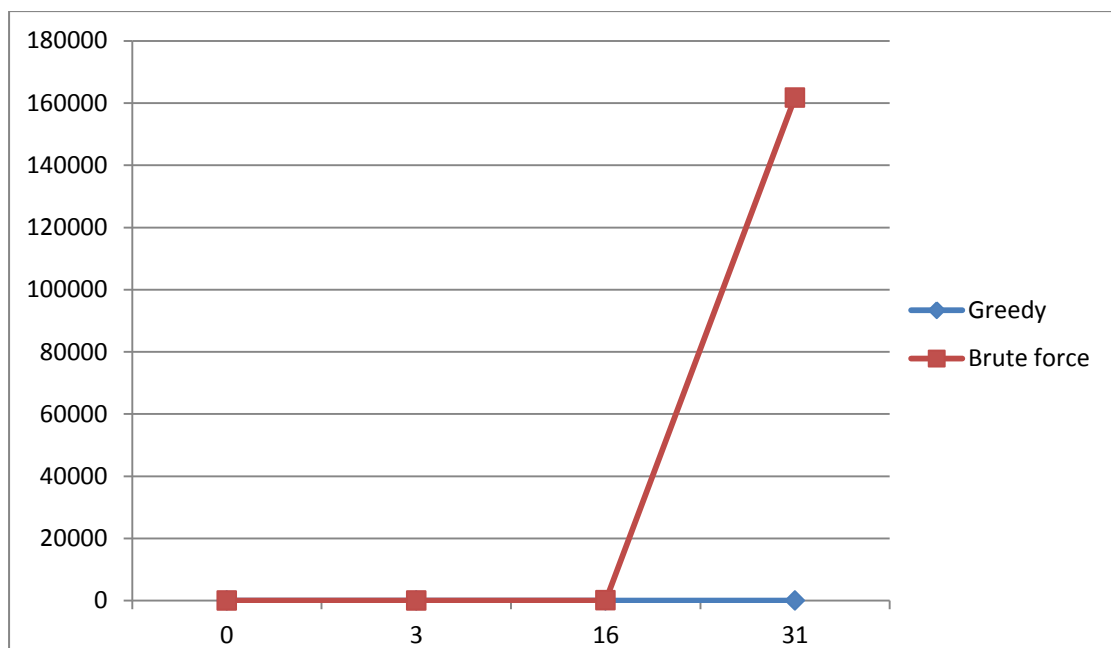


Ο αλγόριθμος εξαντλητικής αναζήτησης



Για τις τιμές μεγαλύτερες του μηδενός και μέχρι 16 του άξονα X οι τιμές για τον Y δεν είναι 0 απλά φαίνονται έτσι στο διάγραμμα. Κανονικά είναι για X: 3 και 16 έχει Y : 3,4 και 62,1 αντίστοιχα.

Μαζί και οι 2 αλγόριθμοι





Όπως είναι λογικό και φαίνεται και από το παραπάνω διάγραμμα η άπληστη προσέγγιση του αλγορίθμου θέλει μικρότερο χρόνο εκτέλεσης για την ίδια είσοδο από ότι η προσέγγιση της εξαντλητικής αναζήτησης. Επίσης, ο χρόνος του άπληστου αλγορίθμου παρά την αύξηση της εισόδου παραμένει σε μικρά επίπεδα ενώ της εξαντλητικής αναζήτησης ο χρόνος αυξάνεται πάρα πολύ με την αύξηση της εισόδου. Ο χρόνος του άπληστου αλγόριθμου είναι πολυωνυμικού χρόνου σε σχέση με την είσοδο ενώ της εξαντλητικής αναζήτησης ο χρόνος είναι παραγοντικού χρόνου σε σχέση με την είσοδο. Στον αλγόριθμο εξαντλητικής αναζήτησης ο χρόνος εξαρτάται αρκετά και από την μορφή του γράφου καθώς και άμα χρειάζεται ένα αρκετά μεγάλο  $k$ , που θα δημιουργήσει ένα υποσύνολο  $k$  κόμβων που θα αποτελέσει vertex coverage, τότε ο χρόνος θα αυξηθεί αρκετά επίσης. Ενώ στον άπληστο αλγόριθμο η μορφή του γράφου δεν έχει τόση σημασία. Πάντως και οι δύο αλγόριθμοι δίνουν μία σωστή λύση αν και η άπληστη προσέγγιση είναι πολύ πιο αποδοτική λόγω του χρόνου που χρειάζεται σε σχέση με την εξαντλητική αναζήτηση. Παρ'όλα αυτά η λύση που δίνει η εξαντλητική αναζήτηση δίνει μία πιο ποιοτική λύση.

Υ.Γ. Για την επιλογή του αρχείου που θα διαβάσει το πρόγραμμά μας χρησιμοποιώ γραφικό περιβάλλον. Ενώ το αποτέλεσμα του αλγορίθμου εκτυπώνεται στην console.