

**Algorithms Spring 2024**

**Project Report**

*Δημήτρης Παπαδημητρίου-Βασίλης Μαδέσης*

*AEM: 03750-03748*

### **Τρόπος αποθήκευσης γράφων και δομές δεδομένων:**

Για την αποθήκευση του γράφου χρησιμοποιούμε έναν πίνακα απο λίστες γειτνίασης. Κάθε θέση του πίνακα δηλαδή αντιστοιχεί σε έναν κόμβο του γράφου, και περιέχει μία απλά συνδεδεμένη λίστα, κάθε στοιχείο της οποίας περιέχει έναν κόμβο με τον οποίο ο αρχικός συνδέεται. Γενικά οι λίστες γειτνίασης χρησιμοποιούνται για αλγορίθμους γράφων όπως Breadth First Search που χρησιμοποιήσαμε και στην άσκηση, και είναι πολύ αποδοτικές σε αραιούς γράφους, όπως αυτούς που δώθηκαν στα παραδείγματα. Άλλες δομές δεδομένων που χρησιμοποιήθηκαν στο πρόγραμμα είναι οι πίνακες, για να αποθηκεύσουμε για παράδειγμα την απόσταση καθώς και τις χρήσεις σε συντομότερα μονοπάτια για κάθε ακμή, και οι ούρες, που αποτελούν κομμάτι του Breadth First Search αλγόριθμου.

### **Εύρεση CPL ενός γράφου:**

Για να βρούμε το CPL ενός γράφου χρησιμοποιήσαμε Breadth First Search ώστε να υπολογίσουμε όλα τα μήκοι των σύντομων μονοπατιών μεταξύ όλων των κόμβων. Συγκεκριμένα υλοποιούμε μια συνάρτηση η οποία λαμβάνει ως input το source node και έχει ένα πίνακα που κρατάει τις αποστάσεις όλων των κόμβων από τον κόμβο source μαζί με ένα πίνακα με τους γονείς κάθε κόμβου. Ο πίνακας γονιών θα χρησιμοποιηθεί αργότερα για το B ερώτημα όπου θα εξηγήσουμε την χρήση του. Τελός η συνάρτηση αυτή επιστρέφει τους πίνακες αυτούς ενώ αν της δωθεί συγκεκριμένος destination κομβός τότε θα τρέχει μέχρι να βρεί το κόμβο αυτό επιστρέφοντας επιτυχία με 1 και αποτυχία με 0.

Για την υλοποίηση της Breadth First Search χρησιμοποιούμε μια queue (FIFO ούρα) ώστε να τοποθετούμε μέσα της τους κόμβους που πρέπει να επισκεφτούμε μετά από μια επίσκεψη κόμβου σε κάθε επανάληψη.

Η συνάρτηση λειτουργεί ως εξής:

---

```
pathSearch()
distance[size of graph] -> 0

Insert in queue node S
while(Queue is not empty)
    parent node <- remove from queue the first element

    for all child nodes of parent node
        if(distance[child node] is 0 && child node != S)
            Put child node into queue
            distance[child node] = distance[parent node] + 1
            parent[child node] = parent node
        else if(distance[child node] == distance[parent node] + 1)
            parent[child node] -> add parent to the list of parents for
                child node
        if (distance[child node] == distance[parent node] + 1 && child
            node == destination node)
            return 1
    end
return 0
end
```

---

Για να βρούμε το CPL εκτελούμε κατα επανάληψη την προηγούμενη συνάρτηση για όλους τους κόμβους και μηδενίζουμε τις επαναλαμβανόμενες αποστάσεις που προκύπτουν στον πίνακα αποστάσεων που μας επιστρέφει η συνάρτηση και έπειτα προσθέτουμε στο συνολικό άθροισμα, το άθροισμα των CPL του πίνακα αυτού.

---

```

Sum of all cpls = 0;

for all nodes inside the graph
    distances array = execute pathSearch
    distances array -> distance array with zeroed the recurring cpls
    Sum of all cpls = distances array + Sum of all cpls
end

cpl = Sum of all cpls / binomial coefficient(number of nodes,2)

```

---

Για να υπολογίσουμε τον συνολικό αριθμό των συντομοτέρων μονοπατιών (SPs) μεταξύ κάθε δυνατού ζεύγους κόμβων  $j$  και  $k$  στα οποία μεσολαβεί κάθε ακμή  $e$  του γραφήματος. Θα χρησιμοποιήσουμε τον πίνακα γονέων που μας επιστρέφει η συνάρτηση `pathSearch()`.

Ο πίνακας γονέων είναι ένας πίνακας που σε κάθε θέση  $i$  έχει τον γονέα του κόμβου  $i + 1$ . Γονέας ενός κόμβου είναι ο γειτονικός κόμβος από τον οποίο διαβάστηκε και μπήκε στον πίνακα αποστάσεων ο κόμβος παιδί (στην περιπτωσή μας ο κόμβος  $i + 1$ ). Σε περίπτωση ύπαρξης δύο ή περισσότερων σύντομων μονοπατιών μεταξύ δύο κόμβων ο κόμβος destination θα έχει δύο γονείς καθώς και οι δύο είναι μέρος σύντομου μονοπατιού.

Οι συμπλήρωση του πίνακα γονέων φαίνεται στο παρακάτω code snippet της συνάρτησης `pathSearch`:

---

```

if (distance[child node] is 0 && child node != S)
    Put child node into queue
    distance[child node] = distance[parent node] + 1
    parent[child node] = parent node
else if (distance[child node] == distance[parent node] + 1)
    parent[child node] -> add parent to the list of parents for
    child node

```

---

Αν ο κόμβος έχει παραπάνω από έναν γονέα τότε δημιουργείται μια λίστα με του γονείς του κόμβου αυτού στην αντίστοιχη θέση του πίνακα.

Προσθέτοντας στον προηγούμενο κώδικα, για κάθε επανάληψη της 'λούπας' παίρνουμε τον πίνακα γονέων και για κάθε ζευγάρι κόμβων source-destination διατρέχουμε το πίνακα ξεκινώντας από το destination και ακολουθώντας τον γονέα του και από εκεί ακολουθώντας τον γονέα του γονέα του, φτάνουμε στο source καταγράφοντας τα edges που συμμετείχαν στο δρόμο μέχρι το source.

Το path για κάθε ζεύγος καθώς και τα edges που συμμετέχουν σε αυτό βρίσκεται από τον ψευδοκώδικα:

---

```
countEdgesPath()
  current node = destination node
  while(parent of current node != Source)
    array of edges [parent node][current node] ++
    array of edges [parent node][current node] ->add info about the edge
  end

  if (there are more than one parents for destination node)
    repeat the process for the other parent of destination node
  end
end
```

---

Ο συνολικός ψευδοκώδικας προκύπτει:

---

```
Sum of all cpls = 0;

for all nodes inside the graph
  distances array = execute pathSearch
  distances array -> distance array with zeroed the recurring cpls
  Sum of all cpls = distances array + Sum of all cpls

  for all combinations of sources and destination nodes inside a graph
    countEdgesPath() fill the edge array
  end
end

cpl = Sum of all cpls / binomial coefficient(number of nodes,2)
```

---

Τέλος έξω από την επανάληψη κάνουμε merge sort το πίνακα edge array και επιστρέφουμε το πρώτο edge του πίνακα το οποίο και θα αφαιρέσουμε.Το δύο προηγούμενα ερωτήματα τα εκτελεί η συνάρτηση cpl sp.

---

```

cpl_sp()
Sum of all cpls = 0;

for all nodes inside the graph
    distances array = execute pathSearch
    distances array -> distance array with zeroed the recurring cpls
    Sum of all cpls = distances array + Sum of all cpls

    for all combinations th source and destination nodes inside the graph
        countEdgesPath() fill the edge array
    end
end

cpl = Sum of all cpls / binomial coefficient(number of nodes,2)
mergesort(edge array)

return top element in edge array
end

```

---

Συνεπώς το complexity της συνάρτησης προκύπτει ως εξής:

Το complexity της αναζήτησης είναι  $O(V + E)$  όπου  $V$  κόμβοι και  $E$  ακμές.

Για την συνάρτηση μηδενισμού έχουμε complexity  $O(V)$  όπως το ίδιο και για την συνάστηση αθροίσματος.

Η συνάστηση επαναληπτική εκτέλεση της συνάστησης `countEdgesPath()` έχει complexity  $O(V * BiggestPathLength)$  καθώς εκτελείται περίπου  $O(V)$  φορές για ένα δεδομένο πίνακα γονέων και η διάτρεξη του πίνακα γονέων έχει στην μεγαλύτερη περίπτωση μεταξύ όλων των  $O(BiggestPathLength)$  πράξεις.

Άρα το συνολικό complexity εφόσον γίνονται  $V$  επαναλήψεις είναι  $O(V) * [O(V + E) + O(V) + O(V) + O(V * BiggestPathLength)] = O(V^2 + VE)$

Εφόσον έχουμε εκτελέσει την συνάρτηση `cpl sp` τότε κάνουμε αφαίρεση της ακμής μεταξύ των δύο κόμβων αφαιρώντας τις μεταξύ ακμές από τις αντίστοιχες λίστες γειτνίασης τους.

### Αφαίρεση της ακμής, έλεγχος συνεκτικότητας του γράφου:

Για την αφαίρεση της ακμής που ενώνει δύο κόμβους  $i, j$ , θα χρειαστεί απλώς να αφαιρέσουμε τον κόμβο  $j$  από τη λίστα γειτνίασης του κόμβου  $i$  και αντίστροφα. Θα χρειαστεί όμως, όταν αφαιρεθεί η ακμή, να ελέγξουμε αν ο γράφος είναι συνεκτικός, η αν έχει χωριστεί σε δύο διαφορετικές συνιστώσες. Η περιπλοκότητα αυτής της πράξης θα κριθεί από τον μέγιστο μέγεθος μίας λίστας γειτνίασης, δηλαδή είναι ίση με  $O(V)$ . Όμως επειδή οι γράφοι στα παραδείγματα είναι αραιοί, η περιπλοκότητα εδώ θα είναι στη πραγματικότητα αμελητέα. Όμως, αν ο γράφος έχει πράγματι χωριστεί, σημαίνει πως η ακμή που αφαιρέσαμε ήταν ο μόνος συνεκτικός "κρίκος" ανάμεσα στους δύο γράφους. Δηλαδή μπορούμε να διαπιστώσουμε αν οι γράφοι χωρίστηκαν, αρχίζοντας από τον ένα κόμβο και ψάχνοντας τουλάχιστον ένα μονοπάτι που θα οδηγήσει στον άλλον. Προφανώς αν το μονοπάτι αυτό δεν υπάρχει, τότε σημαίνει πως οι γράφοι πράγματι χωρίστηκαν. Η περιπλοκότητα της αναζήτησης αυτής είναι ίση με  $O(V + E)$ . Η συνάρτηση `Path search()` μπορεί να χρησιμοποιηθεί για την εύρεση αυτού του μονοπατιού, που επιστρέφει ανάλογη τιμή αν βρεί τον destination node. Θα μας δώσει επίσης τον πίνακα με τις αποστάσεις από τον αρχικό κομβο (source node), που θα φανεί ιδιαίτερα χρήσιμο στο επόμενο βήμα του αλγορίθμου.

### Αναδρομική συνάρτηση:

Εδώ αξίζει να σημειωθεί η δομή της συνάρτησης που χρησιμοποιείται, η οποία αποτελείται από μία επανάληψη μέσα στην οποία καλούνται οι συναρτήσεις για την εύρεση του cpl, την αφαίρεση ενός κόμβου, τον έλεγχο συνεκτικότητας του γράφου καθώς και για την δημιουργία των δύο υπογράφων και την αναδρομική κλήση του εαυτού της για καθέναν από τους υπογράφους, αν χρειαστεί. Η αναδρομή τερματίζει όταν μια συνιστώσα αποτελείται από έναν μονάχα κόμβο. Αυτό φαίνεται και στον παρακάτω ψευδοκώδικα:

---

```
void analyseGraph()
    if numVertices = 1
        return

    while 1:
        MostUsedEdge = cpl_sp()
        removeEdge(MostUsedEdge)

        is graph connected = pathSearch()

        if graph is not connected:

            subgraphs = splitGraph()
            analyseGraph(bigger subgraph)
            analyseGraph(smaller subgraph)
            return
```

---

Για να δημιουργήσουμε τους υπογράφους, ιδιαίτερα χρήσιμος θα φανεί ο πίνακας αποστάσεων της `pathSearch`, καθώς παρέχει έναν εύκολο τρόπο να ξεχωρίσουμε τους κόμβους που θα είναι μέρος κάθε υπογράφου. Συγκεκριμένα, κάθε κόμβος που αντιστοιχεί σε θέση του πίνακα με απόσταση διάφορη του μηδενός, συμπεριλαμβανομένου του `source node`, θα ανήκουν στον πρώτο γράφο, και τα υπόλοιπα θα ανοίχουν στον 2ο γράφο.