Αλγόριθμοι και Δομές Δεδομένων

Σεπτέμβρης 2018

Διδάσκων: Αν. Καθηγητής Π. Λουρίδας

Πόσα Ταξί Χρειάζονται;

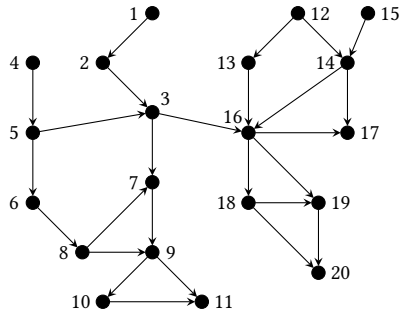
Σημαντικό μέρος των μεταφορών σε μία μεγάλη πόλη γίνεται με ταξί. Το καθεστώς αδειοδότησης των ταξί διαφέρει από πόλη σε πόλη, συνήθως όμως υπάρχει ένας συγκεκριμένος αριθμός αδειών ταξί που χορηγείται.

Από τη μία μεριά, ένας μεγάλος αριθμός ταξί μπορεί να διευκολύνει το επιβατικό κοινό, καθώς όσο πιο πολλά ταξί υπάρχουν τόσο πιο εύκολο είναι να βρει κάποιος ένα ταξί όταν το χρειάζεται. Από την άλλη μεριά, τα ταξί συνεισφέρουν στην κίνηση της πόλης και μπορεί να εκπέμπουν καυσαέρια· επίσης, οι κάτοιχοι αδειών ταξί δεν επιθυμούν να πέσει η αξία της άδειάς τους ως αποτέλεσμα της αύξησης των ταξί που κυκλοφορούν.

Μπορούμε λοιπόν να διερευνήσουμε κατά πόσο είναι δυνατόν να *μειωθεί* ο αριθμός των ταξί σε μία πόλη, χωρίς αρνητικές επιπτώσεις στο επιβατικό κοινό. Είναι, δηλαδή, δυνατόν να εξυπηρετηθεί το επιβατικό κοινό το ίδιο καλά με λιγότερα ταξί; Αυτό είναι το *πρόβλημα του ελάχιστου στόλου* (minimum fleet problem): να βρεθεί ο ελάχιστος αριθμός οχημάτων ο οποίος μπορεί να εξυπηρετήσει ένα συγκεκριμένο σύνολο διαδρομών.

Για να το κάνουμε αυτό, συλλέγουμε στοιχεία για τις διαδρομές που κάνουν τα ταξί μιας πόλης σε ένα χρονικό διάστημα (π.χ. ένα χρόνο). Στη συνέχεια, μαζεύουμε όλες τις διαδρομές που γίνονται σε κάποιο συγκεκριμένο χρονικό παράθυρο (π.χ. μία ώρα). Με βάση αυτές κατασκευάζουμε ένα γράφο, οι κορυφές του οποίου είναι οι διαδρομές των ταξί. Δύο κορυφές u και v θα συνδέονται μεταξύ τους αν και μόνο αν: (1) το σημείο τέλους της διαδρομής u είναι εντός μιας ακτίνας r από το σημείο εκκίνησης της διαδρομής v και (2) η διαδρομή u τελειώνει τη χρονική στιγμή t_u^e, η διαδρομή v ξεκινάει τη χρονική στιγμή t_v^s, και $t_v^s - t_u^e < \delta$. Τα r και δ τα ορίζουμε εμείς: για παράδειγμα, μπορεί να θέσουμε $r = 100\,m$ και $\delta = 15\text{mn}$.

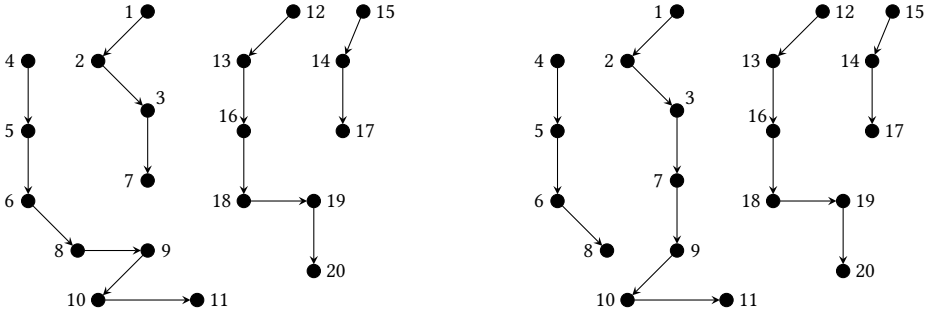
Ο γράφος που προκύπτει ονομάζεται *δίκτυο διαμοιρασμού οχημάτων* (vehicle-shareability network), διότι μας δείχνει πώς μπορούμε να χρησιμοποιήσουμε έναν αριθμό οχημάτων για να καλύψουμε όλες τις διαδρομές. Δείτε τον παρακάτω γράφο:



Ο γράφος αυτός δείχνει 20 διαδρομές ταξί, οι οποίες υπόκεινται στους περιορισμούς που αναφέραμε. Προφανώς, μπορούμε να χρησιμοποιήσουμε 20 ταξί για να τις πραγματοποιήσουμε. Μπορούμε όμως να χρησιμοποιήσουμε και λιγότερα ταξί, αφού το ίδιο ταξί μπορεί να πραγματοποιήσει μια διαδρομή και στη συνέχεια να πραγματοποιήσει μια άλλη διαδρομή, η οποία θα είναι γείτονας της προηγούμενης στο γράφο. Ένα ταξί μπορεί να πραγματοποιήσει τη διαδρομή 1, μετά το ίδιο ταξί μπορεί να πραγματοποιήσει τη διαδρομή 2, κ.ο.κ.

Η εύρεση του ελάχιστου αριθμού ταξί που μπορεί να καλύψει όλες τις διαδρομές στο χρονικό παράθυρο που ερευνούμε αντιστοιχεί στην εύρεση της *βέλτιστης κάλυψης μονοπατιών* (optimal path cover) στο γράφο διαμοιρασμού οχημάτων. Η βέλτιστη κάλυψη μονοπατιών είναι ένα ελάχιστο σύνολο μονοπατιών, χωρίς κοινές κορυφές, που να καλύπτουν όλες

τις κορυφές ενός γράφου. Στην επόμενη εικόνα βλέπετε δύο τέτοια σύνολα μονοπατιών, που δείχνουν ότι όλες οι διαδρομές μπορούν να πραγματοποιηθούν με τέσσερα μόνο ταξί.

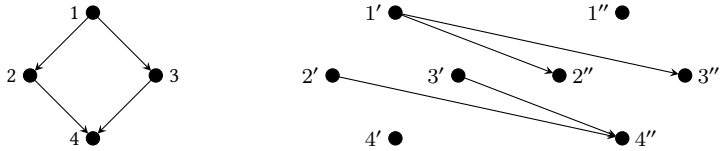


Μια ομάδα ερευνητών καταπιάστηκε με τη βελτιστοποίηση του στόλου ταξί της πόλης της Νέας Υόρκης. Εκεί υπήρχαν 13.586 ταξί το έτος 2011. Χρησιμοποιώντας την παραπάνω μέθοδο, οι ερευνητές επεξεργάστηκαν πάνω από 150 εκατομμύρια διαδρομές ταξί που έγιναν μέσα στο έτος, ομαδοποιώντας τες ανά ώρα στο εικοσιτετράωρο ώστε να βρουν το βέλτιστο αριθμό ταξί που απαιτούνται ανά ώρα. Προέκυψε ότι από 7.748 ταξί που κυκλοφορούν ανά ώρα στην πόλη, οι επιβάτες θα μπορούσαν να εξυπηρετηθούν με μόνο 4.627, μια μείωση 40%!

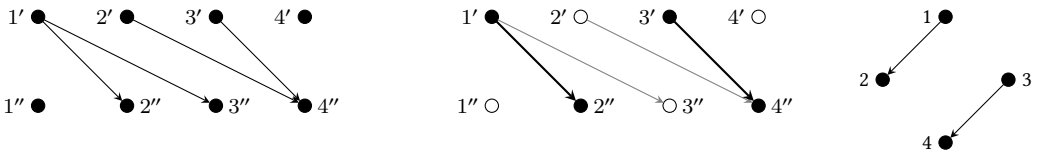
Τρόπος Επίλυσης

Το ζήτημα τώρα είναι, πώς μπορούμε να βρούμε μία ελάχιστη κάλυψη μονοπατιών σε ένα γράφο; Δεν υπάρχει αποτελεσματικός αλγόριθμος για όλων των ειδών τους γράφους, αλλά ευτυχώς υπάρχει αποτελεσματικός αλγόριθμος για άκυκλους κατευθυνόμενους γράφους (directed acyclig graphs), όπως αυτός στο παράδειγμά μας.

Έστω ότι έχουμε έναν άκυκλο κατευθυνόμενο γράφο $G = (V, E)$, όπου V είναι οι κορυφές και E είναι οι σύνδεσμοι. Φτιάχνουμε έναν άλλο γράφο G', ο οποίος για κάθε κορυφή $v \in V$ έχει δύο κορυφές, v' και v''. Για κάθε μία ακμή (u, v) του γράφου G δημιουργούμε μία ακμή, (u', v'') στο γράφο G'. Στην εικόνα που ακολουθεί, αριστερά έχουμε έναν άκυκλο κατευθυνόμενο γράφο και δεξιά έχουμε το γράφο τον οποίο κατασκευάζουμε:



Ο γράφος G' που προκύπτει είναι ένας *διμερής γράφος* (bipartite graph): αν προσέξετε, όλοι οι σύνδεσμοι είναι από κορυφές u' σε κορυφές v''. Μπορούμε σε αυτόν το διμερή γράφο να βρούμε μια ένα *μέγιστο ταίριασμα* (maximum matching), δηλαδή μια αντιστοίχιση όσων περισσότερων κόμβων μπορούμε χρησιμοποιώντας συνδέσμους του γράφου, έτσι ώστε κάθε κόμβος να αντιστοιχίζεται το πολύ σε έναν άλλο κόμβο. Το ταίριασμα αυτό θα μας δώσει μια βέλτιστη κάλυψη μονοπατιών και στον αρχικό γράφο. Παρακάτω αριστερά βλέπετε το γράφο G' σχεδιασμένο έτσι ώστε να φαίνεται πιο καθαρά ότι είναι διμερής· στη μέση βλέπετε ένα ταίριασμα και δεξιά βλέπετε πώς αυτό αντιστοιχεί στον αρχικό μας γράφο.

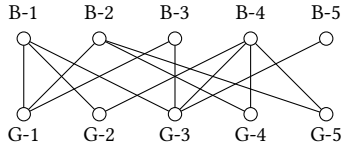


Η εύρεση ενός μέγιστου ταιριάσματος σε ένα διμερή γράφο έχει πολλές εφαρμογές, εκτός αυτής που εξετάζουμε εδώ:

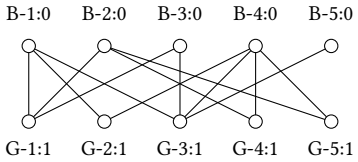
- Ο διμερής γράφος μπορεί να αποτελείται από κόμβους που αντιστοιχούν σε άτομα και κόμβους που αντιστοιχούν σε εργασίες. Ένα μέγιστο ταίριασμα θα μας δώσει ένα βέλτιστο τρόπο να αντιστοιχίσουμε άτομα σε εργασίες.

- Το ένα σύνολο των κόμβων μπορεί να είναι υποψήφιοι για την κάλυψη θέσεων και το άλλο σύνολο των κόμβων μπορεί να είναι οι διαθέσιμες θέσεις. Οι σύνδεσμοι αντιστοιχούν σε θέσεις για τις οποίες έχει αιτηθεί ο κάθε υποψήφιος. Ένα μέγιστο ταίριασμα θα μας δώσει ένα βέλτιστο τρόπο αντιστοίχισης υποψηφίων σε θέσεις.

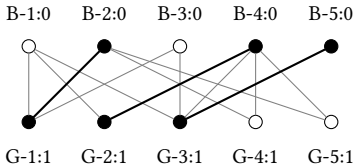
Για να βρούμε ένα μέγιστο ταίριασμα σε ένα διμερή γράφο χρησιμοποιούμε τον αλγόριθμο Hopcroft-Karp. Ο αλγόριθμος αυτός προχωράει σε στάδια. Το κάθε στάδιο αποτελείται από μία κατά πλάτος αναζήτηση και μια σειρά κατά βάθος αναζητήσεων. Ο καλύτερος τρόπος να δούμε πώς λειτουργεί είναι με ένα παράδειγμα. Θα χρησιμοποιήσουμε τον παρακάτω διμερή γράφο, όπου με λευκό κύκλο σημειώνουμε τους ελεύθερους κόμβους (στην αρχή είναι όλοι ελεύθεροι). Οι κόμβοι του γράφου αντιστοιχούν σε κιθαρίστες (τα ονόματα αρχίζουν από G) και σε μπασίστες (τα ονόματα αρχίζουν από B). Οι σύνδεσμοι δείχνουν πιθανές συνεργασίες. Σκοπός μας είναι να ταιριάξουμε όσους περισσότερους κιθαρίστες και μπασίστες μπορούμε.



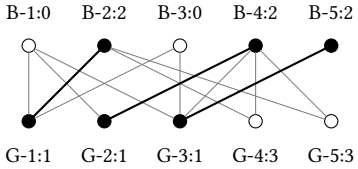
Κάνουμε μια κατά πλάτος διάσχιση ξεκινώντας από τους ελεύθερους μπασίστες, προκειμένου να βρούμε τις αποστάσεις των υπόλοιπων κόμβων από αυτούς. Όλοι οι μπασίστες θα έχουν απόσταση μηδέν και όλοι οι κιθαρίστες θα έχουν απόσταση ένα από τους μπασίστες:



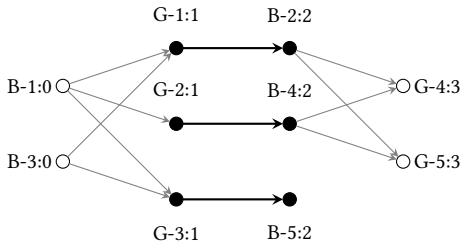
Στη συνέχεια κάνουμε μια κατά βάθος αναζήτηση από κάθε ελεύθερο μπασίστα μέχρι να βρούμε έναν ελεύθερο κιθαρίστα, ακολουθώντας όμως ένα σύνδεσμο μόνο αν μας πηγαίνει από μια απόσταση d σε μία απόσταση $d + 1$. Κάθε σύνδεσμο από μπασίστα σε κιθαρίστα που συναντάμε τον προσθέτουμε στο ταίριασμά μας. Αν κάνουμε τις κατά βάθος αναζητήσεις ξεκινώντας από τους κόμβους B-2, B-4, B-5, B-1, B-3, θα πάρουμε το παρακάτω αρχικό ταίριασμα:



Τώρα επαναλαμβάνουμε τις κατά πλάτος αναζητήσεις από κάθε ελεύθερο μπασίστα, δηλαδή μία από τον B-1 και μία από τον B-3. Προσέχουμε όμως τον παρακάτω κανόνα: αν είμαστε σε μπασίστα, μπορούμε να ακολουθήσουμε ένα σύνδεσμο μόνο αν δεν ανήκει στο ταίριασμα που έχουμε βρει μέχρι τώρα, ενώ αν είμαστε σε κιθαρίστα, μπορούμε να ακολουθήσουμε ένα σύνδεσμο μόνο αν ανήκει στο ταίριασμα που έχουμε βρει μέχρι τώρα. Οι αποστάσεις θα αλλάξουν ως εξής:

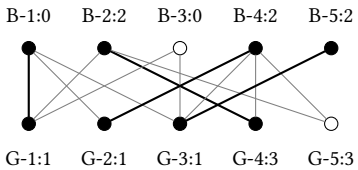


Γιατί το κάνουμε αυτό με τις αποστάσεις; Γιατί θέλουμε να κάνουμε κατά βάθος αναζητήσεις στο διμερή γράφο σαν να τον είχαμε χωρίσει σε στρώματα, ανάλογα με τις αποστάσεις των κόμβων. Τα επίπεδα θα αποτελούνται εναλλάξ από μπασίστες και κιθαρίστες. Ο διαστρωματωμένος αυτός γράφος έχει συνδέσμους μόνο από το ένα επίπεδο στο επόμενό του, και μόνο αν πηγαίνουμε από μπασίστα σε κιθαρίστα και ο σύνδεσμος δεν είναι μέρος του ταιριάσματος, ή πηγαίνουμε από κιθαρίστα σε μπασίστα και ο σύνδεσμος είναι μέρος του ταιριάσματος. Στην περίπτωσή μας, ο διαστρωμένος γράφος θα είναι:

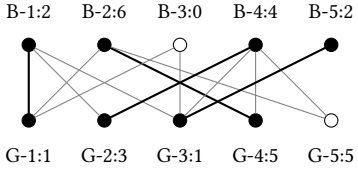


Στο διαστρωματωμένο γράφο αναζητούμε μονοπάτια, μέσω κατά βάθος αναζητήσεων, από τους ελεύθερους μπασίστες (στα αριστερά) στους ελεύθερους κιθαρίστες (στα δεξιά). Ένα τέτοιο μονοπάτι ονομάζεται *επαυξημενο μονοπάτι* (augmenting path). Θα ξεκινάει, από έναν ελεύθερο μπασίστα, θα καταλήγει σε έναν ελεύθερο κιθαρίστα, και θα περνάει εναλλάξ από κιθαρίστες και μπασίστες. Όταν βρίσκουμε ένα τέτοιο μονοπάτι τότε προσθέτουμε στο ταίριασμα που έχουμε βρει μέχρι τώρα κάθε σύνδεσμο του μονοπατιού που οδηγεί από έναν μπασίστα σε έναν κιθαρίστα.

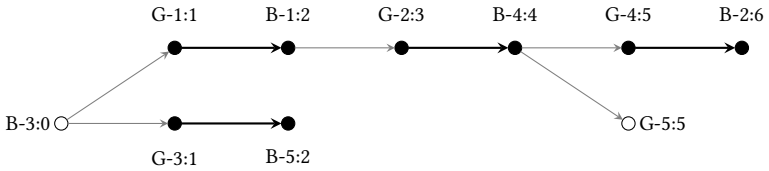
Στο παράδειγμά μας, ξεκινάμε μια κατά βάθος αναζήτηση από τον κόμβο B-1 και βρίσκουμε το επαυξημένο μονοπάτι B-1, G-1, B-2, G-4. Προσθέτουμε λοιπόν στο ταίριασμά μας τις αντιστοιχίσεις B-1: G-1 και B2: G-4. Προσέξτε ότι αυτό σημαίνει ότι η αντιστοίχιση B-2: G-1 που είχαμε πριν δεν υπάρχει πια, όπως μπορείτε να δείτε στο διμερή γράφο παρακάτω. Αυτό δεν μας πειράζει, γιατί από εκεί που είχαμε ένα ταίριασμα με τρεις αντιστοιχίσεις, τώρα έχουμε ένα ταίριασμα με τέσσερεις αντιστοιχίσεις:



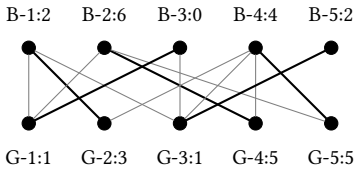
Επιστρέφουμε στη διαδικασία των κατά πλάτος αναζητήσεων. Ακολουθούμε πάλι τον κανόνα σύμφωνα με τον οποίο αν είμαστε σε μπασίστα, μπορούμε να ακολουθήσουμε ένα σύνδεσμο μόνο αν δεν ανήκει στο ταίριασμα που έχουμε βρει μέχρι τώρα, ενώ αν είμαστε σε κιθαρίστα, μπορούμε να ακολουθήσουμε ένα σύνδεσμο μόνο αν ανήκει στο ταίριασμα που έχουμε βρει μέχρι τώρα. Αυτή τη φορά έχουμε μόνο έναν ελεύθερο μπασίστα, τον B-3, οπότε η αναζήτηση αυτή θα ενημερώσει τις αποστάσεις ως εξής:



Ο γράφος μας τώρα αντιστοιχεί σε έναν νέο διαστρωμένο γράφο:



Προσπαθούμε στο διαστρωμένο γράφο να βρούμε επαυξημένα μονοπάτια. Αφού έχουμε έναν ελεύθερο μπασίστα και έναν ελεύθερο κιθαρίστα, θα προσπαθήσουμε να βρούμε ένα επαυξημένο μονοπάτι από τον B-3 στον G-5. Ξεκινώντας κατά βάθος αναζήτηση από τον B-3 βρίσκουμε το μονοπάτι B-3, G-1, B-1, G-2, B-4, G-5. Με βάση αυτό το μονοπάτι, προσθέτουμε στο ταίριασμα τις αντιστοιχίσεις από μπασίστα σε κιθαρίστα, δηλαδή τις B-3: G-1, B-1:G-2, B-4:G-5. Αυτό σημαίνει ότι ταυτόχρονα αφαιρούνται οι αντιστοιχίσεις B1:G1 και B4:G2 που είχαμε πριν, αλλά δεν μας πειράζει, γιατί αυξήσαμε πάλι το ταίριασμά μας.



Στο συγκεκριμένο παράδειγμα δεν συνεχίζουμε διότι βρήκαμε ένα μέγιστο ταίριασμα. Γενικότερα όμως, ο αλγόριθμος Hopcroft-Karp συνεχίζει με εναλλαγές κατά πλάτους και κατά βάθους αναζητήσεων μέχρι να μην μπορεί να βρει κανένα επαυξημένο μονοπάτι, οπότε σταματάει αναφέροντας το ταίριασμα το οποίο έχει βρει, το οποίο θα είναι μέγιστο. Δεν είναι απαραίτητο ότι ένα μέγιστο ταίριασμα θα καλύπτει όλους τους κόμβους του διμερούς γράφου· σε κάθε περίπτωση ο αλγόριθμος Hopcroft-Karp θα βρει το μέγιστο.

Ο τρόπος που λειτουργούν οι κατά πλάτος και κατά βάθος αναζητήσεις στον αλγόριθμο δεν διαφέρουν πολύ από τους γνωστούς αλγορίθμους κατά πλάτος και κατά βάθος αναζήτησης. Ο ψευδοκώδικας για την κατά πλάτος αναζήτηση παρατίθεται στη συνέχεια. Ο αλγόριθμος παίρνει ως είσοδο τον γράφο, ένα σύνολο, το οποίο δείχνει ποιοι κόμβοι είναι οι μπασίστες, και μία απεικόνιση (λεξικό), που δείχνει το τρέχον ταίριασμα. Επιστρέφει μια άλλη απεικόνιση η οποία δείχνει τις αποστάσεις των κόμβων από τους ελεύθερους μπασίστες. Οι αποστάσεις αρχικοποιούνται στο μηδέν, για τους ελεύθερους μπασίστες, και στο άπειρο, για τους υπόλοιπους κόμβους. Όπως είδαμε στην περιγραφή του αλγόριθμου, θέλουμε η κατά βάθος αναζήτηση να προχωράει από μπασίστα σε κιθαρίστα, αν ο σύνδεσμος δεν ανήκει στο

παρόν ταίριασμα, ή από κιθαρίστα σε μπασίστα, αν ο σύνδεσμος ανήκει στο παρόν ταίριασμα. Αυτό επιτυγχάνεται με τις συνθήκες στη γραμμή 13. Στις γραμμές 14–15 ενημερώνουμε την απόσταση του κάθε γειτονικού κόμβου που βρίσκουμε.

```
BFS(G, B, M)
    Input: G = (V, E), a graph
           B, a set such that v ∈ B if v is among the bassists nodes
           M, a map; keys are nodes and the value for each key is the node to which it is
           matched, or NULL if the node is unmatched
    Result: D, a map; keys are nodes and the value for each key is the distance of the
            node from a free upper node
1   Q ← CreateQueue()
2   D ← CreateMap()
3   foreach v in V do
4       if v ∈ B and LookUp(M, v) = NULL then
5           InsertInMap(D, v, 0)
6           Enqueue(Q, v)
7       else
8           InsertInMap(D, v, ∞)
9   while not IsQueueEmpty(Q) do
10      c ← Dequeue(Q)
11      foreach v in AdjacencyList(G, c) do
12          if LookUp(D, v) = ∞ then
13              if (c ∈ B and v ≠ LookUp(M, c)) or (c ∉ B and v = LookUp(M, c)) then
14                  d ← LookUp(D, c) + 1
15                  InsertInMap(D, v, d)
16                  Enqueue(Q, v)
17  return D
```

Με αντίστοιχο τρόπο λειτουργούμε στην κατά βάθος αναζήτηση, στον ψευδοκώδικα που ακολουθεί. Στη γραμμή 5 βλέπουμε πάλι τον ίδιο κανόνα όπως και στην κατά πλάτος αναζήτηση για να είμαστε σίγουροι ότι προχωράμε στο γράφο όπως θέλουμε, ενώ στη γραμμή 6 εξασφαλίζουμε ότι προχωράμε από ένα επίπεδο στο επόμενο επίπεδο του διαστρωμένου γράφου. Με τον τρόπο αυτό, στην πραγματικότητα δεν κατασκευάζουμε ποτέ τον διαστρωμένο γράφο! Απλώς εξασφαλίζουμε ότι διασχίζουμε το διμερή γράφο ακολουθώντας τους σωστούς κανόνες, σαν να διασχίζαμε το διαστρωμένο γράφο.

```
DFS(G, s, B, M, D)
    Input: G = (V, E), a graph
           s, the starting bassist node
           B, a set such that v ∈ B if v is among the bassists nodes
           M, a map; keys are nodes and the value for each key is the node to which it is
           matched, or NULL if the node is unmatched
           D a map; keys are nodes and values are the distances of each node from the
           free nodes
    Result: TRUE if a path was found from s to a free guitarist node, FALSE otherwise
1   if s ∉ B and LookUp(M, s) = NULL then
2       InsertInMap(D, s, ∞)
3       return TRUE
4   foreach v in AdjacencyList(G, s) do
5       if (s ∈ B and v ≠ LookUp(M, s)) or (s ∉ B and v = LookUp(M, s)) then
6           if LookUp(D, v) = LookUp(D, s) + 1 then
7               if DFS(G, v, U, M, D) then
8                   if s ∈ B then
9                       InsertInMap(M, s, v)
10                      InsertInMap(M, v, s)
11                  return TRUE
12  InsertInMap(D, s, ∞)
13  return FALSE
```

Ο ψευδοκώδικας για τον αλγόριθμο Hopcroft-Karp καλεί κατ' επανάληψη τους αλγορίθμους της κατά πλάτος και της κατά βάθος αναζήτησης μέχρι να μπει μπορεί να βρεθεί άλλο επαυξημένο μονοπάτι. Αρχικοποιεί μία απεικόνιση στην οποία θα αποθηκεύσουμε το ταίριασμα, και εκτελεί τις επαναλήψεις στις γραμμές 4–11.

```
HopcrofKarp(G, B)
    Input: G = (V, E), a graph
           B, a set such that v ∈ U if v is among the bassists nodes
    Result: M, a map; keys are nodes and the value for each key is the node to which it
            is matched, or NULL if the node is unmatched
1   M ← CreateMap()
2   foreach v in V do
3       InsertInMap(M, v, NULL)
4   augmented ← TRUE
5   while augmented = TRUE do
6       augmented ← FALSE
7       D ← BFS(G, B, M)
8       foreach v in B do
9           if LookUp(M, v) = NULL then
10              if DFS(G, v, B, M, D) then
11                  augmented ← TRUE
12  return M
```

Απαιτήσεις Προγράμματος

Κάθε φοιτητής θα εργαστεί σε αποθετήριο στο GitHub. Για να αξιολογηθεί μια εργασία θα πρέπει να πληροί τις παρακάτω προϋποθέσεις:

- Για την υποβολή της εργασίας θα χρησιμοποιηθεί το ιδιωτικό αποθετήριο του φοιτητή που δημιουργήθηκε για τις ανάγκες του μαθήματος και του έχει αποδωθεί. Το αποθετήριο αυτό έχει όνομα του τύπου `username-algo-assignments`, όπου username είναι το όνομα του φοιτητή στο GitHub. Για παράδειγμα, το σχετικό αποθετήριο του διδάσκοντα θα ονομαζόταν `louridas-algo-assignments` και θα ήταν προσβάσιμο στο `https://github.com/dmst-algorithms-course/louridas-algo-assignments`. Τυχόν άλλα αποθετήρια απλώς θα αγνοηθούν.
- Μέσα στο αποθετήριο αυτό θα πρέπει να δημιουργηθεί ένας κατάλογος με το όνομα `assignment-2018-4`.
- Μέσα στον παραπάνω κατάλογο το πρόγραμμα θα πρέπει να αποθηκευτεί με το όνομα `pc.py`.
- Δεν επιτρέπεται η χρήση έτοιμων βιβλιοθηκών γράφων ή τυχόν έτοιμων υλοποιήσεων των αλγορίθμων, ή τμημάτων αυτών, εκτός και αν αναφερθεί ότι επιτρέπεται. Επιτρέπεται η χρήση δομών δεδομένων της Python όπως στοίβες, λεξικά, κ.λπ.
- Αυτό ισχύει και για υλοποιήσεις του αλγορίθμου Hopcroft-Karp που μπορεί να βρεθούν στο διαδίκτυο. Θα πρέπει να υλοποιήσετε τον αλγόριθμο με βάση τον ψευδοκώδικα που περιλαμβάνεται στην εκφώνηση, αλλιώς η εργασία δεν θα βαθμολογηθεί.

Το πρόγραμμα θα πρέπει να είναι γραμμένο σε Python 3.

Το πρόγραμμα θα το καλεί ως χρήστης ως εξής (όπου `python` η κατάλληλη εντολή στο εκάστοτε σύστημα):

```
python pc.py <input_file>
```

Το `input_file` θα είναι ένα αρχείο που θα περιέχει έναν γράφο που θα περιγράφει ένα δίκτυο διαμοιρασμού οχημάτων. Κάθε αρχείο θα αποτελείται από γραμμές, όπου η κάθε γραμμή θα αναπαριστά ένα σύνδεσμο στο γράφο. Το πρόγραμμά σας θα πρέπει να:

1. Να κατασκευάζει το γράφο διαβάζοντάς τον από το αρχείο.
2. Να κατασκευάζει το διμερή γράφο, όπως περιγράφηκε.
3. Να βρίσκει ένα μέγιστο ταίριασμα στο διμερή γράφο.
4. Να επιστρέφει την ελάχιστη κάλυψη μονοπατιών που περιγράφεται από το μέγιστο ταίριασμα που βρήκε. Τα μονοπάτια θα εμφανίζονται ταξινομημένα, ένα ανά γραμμή.

Παραδείγματα Εκτέλεσης

Παράδειγμα 1

Αν ο χρήστης καλέσει το πρόγραμμα για το αρχείο `https://github.com/dmst-algorithms-course/assignment-2018-4/blob/master/rhombus.txt`, το οποίο περιγράφει το παράδειγμα με τον ρόμβο που είδαμε:

```
python pc.py rhombus.txt
```

η έξοδος του προγράμματος θα είναι μία μέγιστη κάλυψη μονοπατιών όπως:

```
['0', '1']
['2', '3']
```

ή

```
['0', '1', '3']
['2']
```

Η δεύτερη λύση είναι σωστή, γιατί δεχόμαστε και μονοπάτια μηδενικού μήκους. Εξάλλου, και στις δύο λύσεις προκύπτει ότι θέλουμε δύο ταξί για να εξυπηρετήσουμε το σύνολο των διαδρομών.

Παράδειγμα 2

Αν ο χρήστης καλέσει το πρόγραμμα για το αρχείο `https://github.com/dmst-algorithms-course/assignment-2018-4/blob/master/example.txt`, το οποίο περιγράφει το παράδειγμα με τις 20 διαδρομές:

```
python pc.py example.txt
```

η έξοδος του προγράμματος θα είναι μία μέγιστη κάλυψη μονοπατιών όπως:

```
['1', '2', '3', '7']
['12', '13', '16', '18', '19', '20']
['15', '14', '17']
['4', '5', '6', '8', '9', '10', '11']
```

Αφού δεν υπάρχει μοναδική λύση, αρκεί το πρόγραμμά σας να επιστρέφει μία λύση—εξάλλου είναι εύκολο να γράψετε ένα βοηθητικό προγραμματάκι με το οποίο θα μπορείτε να διαπιστώνετε αμέσως αν η λύση είναι σωστή ή όχι.

Βιβλιογραφία

Το άρθρο το οποίο διερευνά την περίπτωση των ταξί στη Νέα Υόρκη είναι το: M. M. Vazifeh, P. Santi, G. Resta, S. H Strogatz, and C. Ratti. Addressing the minimum fleet problem in on-demand urban mobility. *Nature*, 557:534–538, 2018.

Ο αλγόριθμος των Hopcroft-Karp παρουσιάστηκε στο John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM Journal of Computing*, 2 (4):225–231, December 1973.

Η επίλυση του προβλήματος της βέλτιστης κάλυψης μονοπατιών με χρήση του αλγόριθμου Hopcroft-Karp προτάθηκε στο πλαίσιο της βελτιστοποίησης κώδικα, στο F. T. Boesch and J. F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the Association for Computing Machinery*, 24(2):192–198, April 1977.

Καλή Επιτυχία!