Γεώργιος Δημόπουλος

Α.Μ. 2964

Άσκηση 1 (Διερεύνηση γραφημάτων)

Θα χρησιμοποιήσω τον αλγόριθμο DFS επειδή αυτός διερευνά πρώτα τους κόμβους που βρίσκονται σε όσο δυνατόν μεγαλύτερη απόσταση από τον κόμβο εκκίνησης, κάτι το οποίο χρειαζόμαστε και για την άσκηση. Θα χρησιμοποιήσω την υλοποίηση του αλγορίθμου με την στοίβα. Οπότε αρχικά αυτό που κάνω είναι ότι θέτω όλους τους κόμβους έστω x ένας κόμβος min(x)=x. Αυτό μας παίρνει χρόνο O(|V|) Τώρα τρέχω τον αλγόριθμο και ξεκινάω με ένα τυχαίο κόμβο a. Αρχικά η στοίβα μου είναι κενή και τοποθετώ στην στοίβα μου τον κόμβο a. Έπειτα για ΚΑΘΕ νέο κόμβο που θέλει να μπει στην στοίβα (γιατί μπορεί να μην μπει τελικά γιατί έχει ήδη διερευνηθεί) κάνω τον εξής έλεγχο: του λέω εάν το w(min(κόμβος που θέλει να εισαχθεί))< w(min(κόμβος που είναι στην κορυφή της στοίβας)), τότε σε ΟΛΟΥΣ τους κόμβους της στοίβας θέσε για min(κάθε κόμβος λίστας)=κόμβος που θέλει να μπει στην στοίβα. Αλλιώς άσε τα πράγματα όπως είναι. Όταν η στοίβα αδειάσει επιστρέφω τα min του κάθε κόμβου. Έπειτα βρίσκω τον επόμενο κόμβο που δεν έχω επισκεφτεί και κάνω το ίδιο. Περιμένω έτσι μέχρι να τελειώσει ο αλγόριθμος καθοδικής διερεύνησης και να μας δώσει τα αποτελέσματα. Ο συνολικός χρόνος που θα μας πάρει για να γίνει αυτή η διαδικασία είναι όσο χρόνο κάνει για να εκτελεστεί ο DFS γιατί ουσιαστικά εκτελώ τον DFS απλά βάζω και έναν έλεγχο για κάθε κόμβο κάτι το οποίο δεν κοστίζει. Έτσι λοιπόν ο συνολικός χρόνος είναι $O(|V| + |E|)$, δηλαδή γραμμικός χρόνος. Όμως έτσι έχω κάνει ένα σημαντικό λάθος. Στο παράδειγμα σας έχουμε ότι min(d)=d κάτι το οποίο αν τρέξουμε τον δικό μου αλγόριθμο δεν θα ισχύει. Οπότε αυτό που έχω να κάνω μόνο είναι να ελέγξω μια φορά ΜΟΝΟ τους κόμβους. Έτσι ο έλεγχος που κάνω για κάθε κόμβο είναι ο εξής: εάν το w(min(κόμβου))>w(κόμβου) τότε θέσε min(κόμβου)=κόμβος. Αλλιώς άστο έτσι όπως είναι. Κάπου εδώ ολοκληρώνεται ο αλγόριθμος. Ο χρόνος που μου παίρνει να κάνω αυτή την τελευταία δουλειά είναι $O(|V|)$, δηλαδή γραμμικός χρόνος. Άρα ο τελικός συνολικός χρόνος για την άσκηση 1 είναι: **$O(|V| + |E|)+ O(|V|)+O(|V|)=O(3|V|+|E|)=O(|V|+|E|)$** (γραμμικός χρόνο εκτέλεσης)**.**

Άσκηση 2 (Τοπολογική ταξινόμηση)

Σε ένα κατευθυνόμενο άκυκλο γράφημα G οι κόμβοι του γραφήματος μπορούν να τοποθετηθούν σε μια σειρά, η οποία ονομάζεται τοπολογική ταξινόμηση. Το κατευθυνόμενο άκυκλο γράφημα έχει σίγουρα ένα τουλάχιστον κόμβο με in-degree=0 (δηλαδή δεν έχει εισερχόμενες ακμές) και έναν τουλάχιστον κόμβο με out-degree=0 (δηλαδή δεν έχει εξερχόμενες ακμές). Εγώ στον αλγόριθμο μου θα ασχοληθώ με το in-degree=0 αλλά το ίδιο ακριβώς ισχύει και για το out-degree=0.

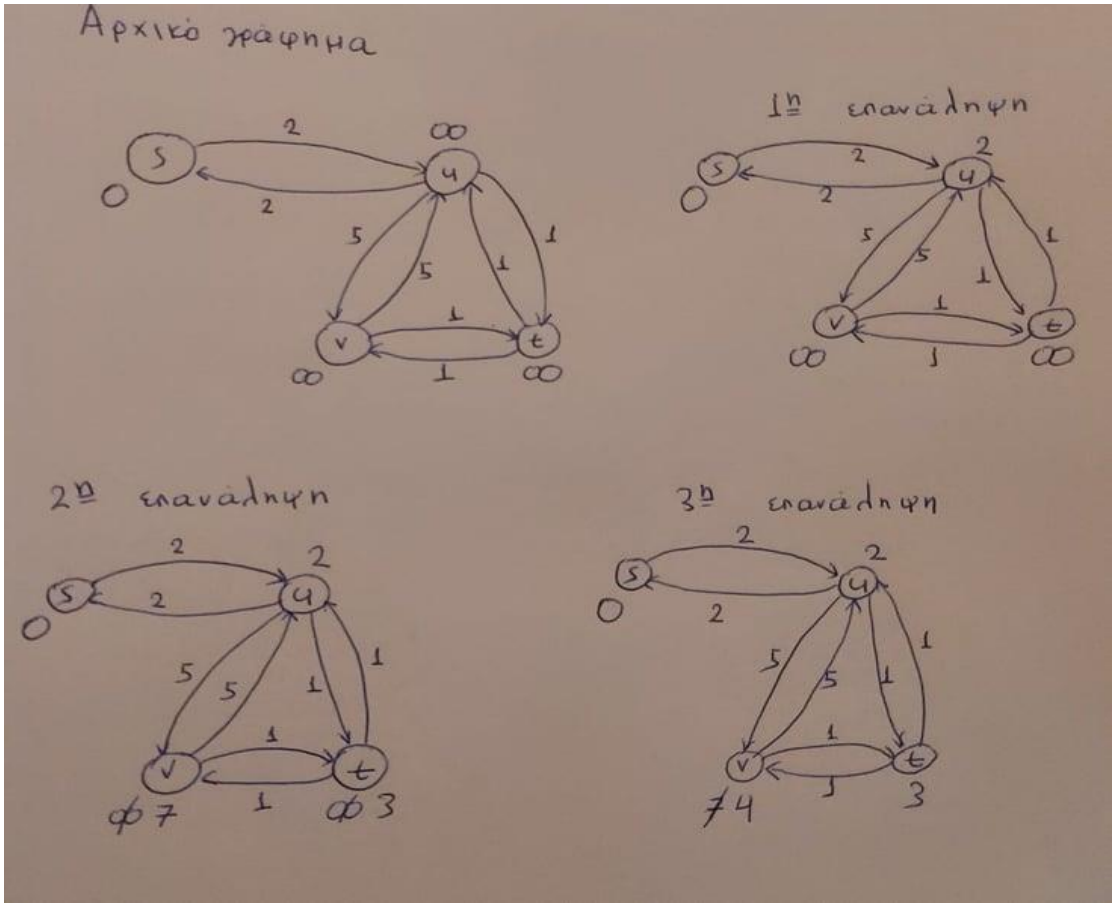
Αρχικά παίρνουμε έναν κόμβο χωρίς εσωτερικές ακμές και τον τοποθετούμε στην επόμενη θέση της διάταξης. Έπειτα τον αφαιρούμε από το γράφημα μας μαζί με τις ακμές του. Αν τώρα έχουμε παραπάνω από έναν κόμβους που δεν έχουν εσωτερικές ακμές τότε ο αλγόριθμος επιλέγει κάποιον τυχαία και τον τοποθετεί στην επόμενη θέση της διάταξης. Αυτό αυτομάτως σημαίνει ότι ΔΕΝ έχουμε μια και μοναδική τοπολογική διάταξη. Άρα αυτό που κάνουμε είναι σε κάθε βήμα του αλγορίθμου τοπολογικής διάταξης να ελέγχουμε αν έχουμε παραπάνω από έναν κόμβο με in-degree =0. Αυτό θα σημαίνει ότι έχουμε πάνω από μια τοπολογική διάταξη. Τώρα ο χρόνος που απαιτείται για αυτό είναι γραμμικός χρόνος και ίσος με: **O(|V| + |E|)** όπου V το πλήθος κόμβων και E το πλήθος των ακμών.

Άσκηση 3 (Ελαφρύτερες διαδρομές)

Σε ένα συνεκτικό γράφημα για κάθε κόμβο του γραφήματος υπάρχει μονοπάτι προς κάθε άλλο κόμβο του γραφήματος. Εγώ τώρα θα εκτελέσω τον αλγόριθμο Bellman-Ford για δύο γραφήματα και μετά θα σας εξηγήσω γιατί δεν δουλεύει για τον ένα από αυτά τα γραφήματα. (Δυστυχώς θα σας τα σχεδιάσω χειρόγραφα γιατί δεν μπόρεσα να τα σχεδιάσω ηλεκτρονικά).

Θα εκτελέσω Bellman-Ford:

Σειρά επεξεργασίας ακμών: {(t,v),(v,t),(u,v),(v,u),(u,t),(t,u),(s,u),(u,s)}

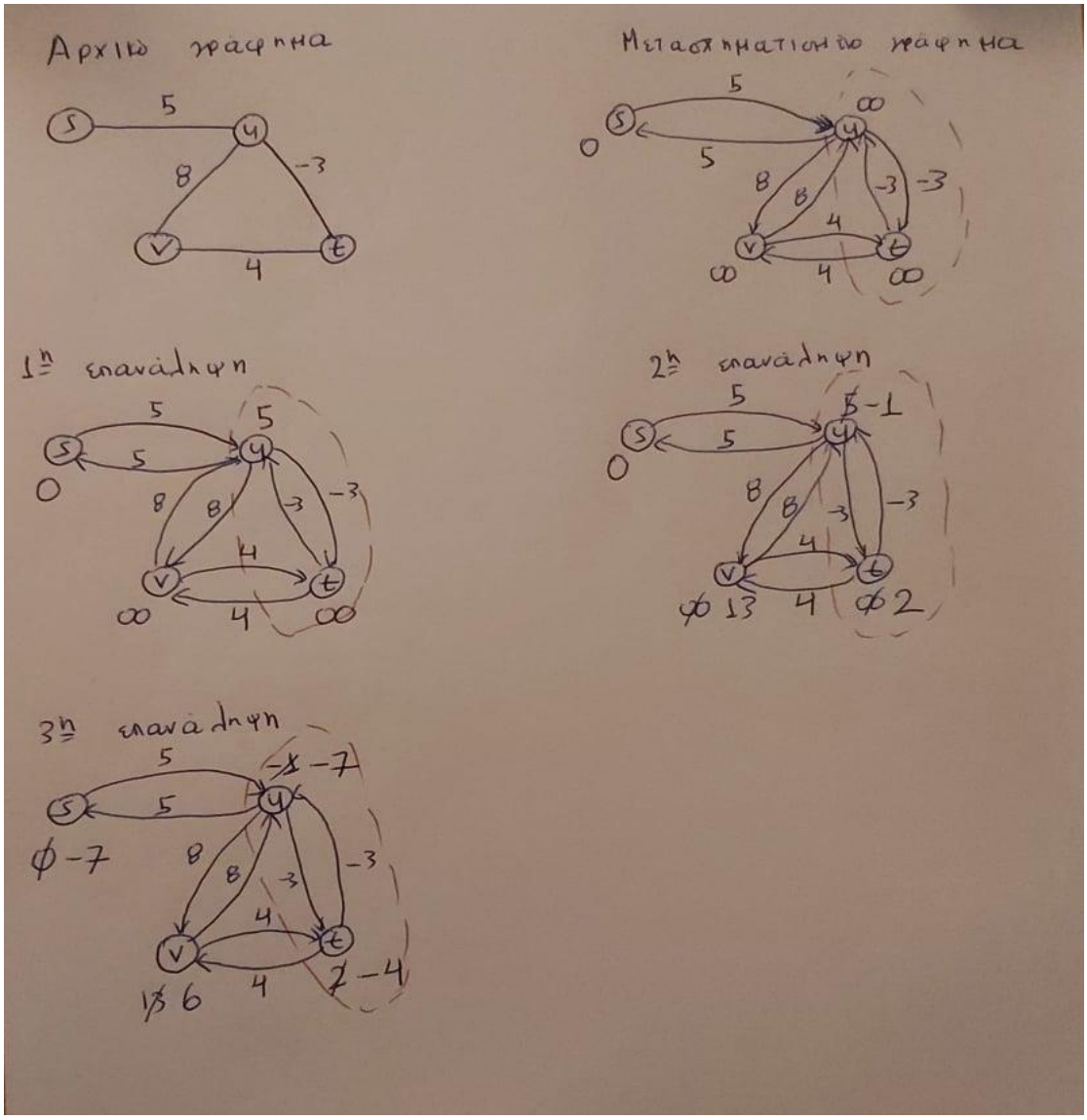


Στο παραπάνω παράδειγμα βλέπουμε ότι ο αλγόριθμος μας λειτουργεί κανονικά. Κάνω ενημέρωση τις ακμές 3 φορές, δηλαδή όσο το πλήθος των κόμβων που

έχω -1.

Θα εκτελέσω Bellman-Ford:

Σειρά επεξεργασίας ακμών: {(t,v),(v,t),(u,v),(v,u),(u,t),(t,u),(s,u),(u,s)}



Στο παραπάνω γράφημα κάνω ενημέρωση τις ακμές 3 φορές, δηλαδή όσο το πλήθος των ακμών που έχω -1. Αν έκανα Bellman-Ford στο αρχικό γράφημα τότε θα εκτελούταν κανονικά και θα μας έβγαζε τις ελαφρύτερες διαδρομές. Όμως όταν εκτελώ Bellman-Ford στο μετασχηματισμένο σχήμα παρατηρώ ότι δημιουργείται κατευθείαν κύκλος αρνητικού βάρους. Έτσι βλέπω ότι ο αλγόριθμος μου δεν τρέχει σωστά και έτσι δεν υπολογίζονται σωστά οι ελαφρύτερες διαδρομές. Το πρόβλημα μας λοιπόν είναι ότι αν στο γράφημα G έχω αρνητική ακμή αλλά όχι κύκλο αρνητικού βάρους, στο γράφημα G' θα έχω σίγουρα κύκλο αρνητικού βάρους και έτσι η προσέγγιση της άσκησης αποτυγχάνει.

Άσκηση 4 (Εναλλακτικές διαδρομές)

Αυτό που θέλω να κάνω για αυτή την άσκηση είναι ουσιαστικά να τρέξω τον αλγόριθμο Dijkstra στο αρχικό γράφημα G από τον s στον t. Θέλω να τρέξω τον πιο γρήγορο Dijkstra οπότε θα επιλέξω αυτόν που χρησιμοποιεί τον σωρό Fibonacci και εκτελείται σε χρόνο *O(|n| + |m|*log|m|)*. Όμως θέλω να προσθέσω και κάτι άλλο στον αλγόριθμο Dijkstra ώστε να τον κάνω πιο προσαρμόσιμο στην άσκηση. Αυτό που θέλω να κάνω είναι να υπολογίσω την ελαφρύτερη διαδρομή από τον s στο t με τις ΛΙΓΟΤΕΡΕΣ ακμές. Αυτό για να το κάνω δεν κοστίζει τίποτα απλά κάνω έναν παραπάνω έλεγχο στην χαλάρωση. Προσθέτω απλά για κάθε κόμβο ένα επιπλέον πεδίο έστω pathlength και το αυξάνω όταν ελέγχω αυτόν τον κόμβο. Έστω ότι ένας κόμβος θέλει να οριστικοποιηθεί αλλά μπορεί από δύο ακμές, συμβουλεύομαι το pathlength και οριστικοποιώ την διαδρομή με τις λιγότερες ακμές. Ακόμη θέλω ο αλγόριθμος να μου επιστρέφει και το συνολικό βάρος του συντομότερου μονοπατιού. Βάζω και σε μια αρχική λίστα που την ονομάζω all_edge_list όλες τις ακμές του γραφήματος, κάτι το οποίο θα με πάρει χρόνο *O(|m|)*. Επίσης έχω σε μια λίστα τις ακμές που έχει η ελαφρύτερη διαδρομή από τον s στον t και την ονομάζω dijkstra_list. Τώρα αφαιρώ από την all_edge_list όλες τις ακμές που περιέχονται στην dijkstra_list, κάτι το οποίο θα με πάρει χρόνο *O(|m|)*. Μετά την διαγραφή για οποιαδήποτε ακμή e στην all_edge_list, συμβολίζουμε με G-e το γράφημα που προκύπτει από το G με την διαγραφή της e. Για κάθε ακμή e που ανήκει στο all_edge_list ορίζω το συντομότερο μονοπάτι από τον s στον t στο G-e να είναι αυτό που έχει η dijkstra_list και έχει συνολικό βάρος αυτό που θα μου επιστρέψει ο dijkstra. Αυτό θα με πάρει χρόνο *O(|m|)* αλλά θα το κάνω ταυτόχρονα με την διαγραφή οπότε θα χρειαστεί μόνο μια φορά να διατρέξω την λίστα all_edge_list. Τώρα μένει να βρούμε το συντομότερο μονοπάτι στο G-e εάν διαγράψουμε μια ακμή από το dijkstra_list. Εδώ δεν μπορούμε να κάνουμε κάτι απλά θα τρέξω τόσες φορές dijkstra όσο το μέγεθος του dijkstra_list. Για αυτό ακριβώς θέλω αρχικά να υπολογίσω την ελαφρύτερη διαδρομή με τις λιγότερες ακμές, ώστε να τρέξω όσο τον δυνατόν λιγότερες φορές dijkstra. Άρα λοιπόν για οποιαδήποτε ακμή e στο dijkstra_list, συμβολίζουμε με G-e το γράφημα που προκύπτει από το G με την διαγραφή της e. Άρα για κάθε ακμή e που ανήκει στο dijkstra_list, το συντομότερο μονοπάτι καθώς και το συνολικό βάρος του μονοπατιού από τον s στον t στο G-e προκύπτει αν τρέξουμε dijkstra στο G-e. Αυτό θα με πάρει χρόνο:

O([length(dijkstra_list)][|n| + |m|*log|m|])*. Έτσι λοιπόν τα σενάρια που προκύπτουν αν αφαιρέσω μια ακμή από το συντομότερο μονοπάτι είναι: Απλά να υπάρχει άλλη διαδρομή με το ίδιο βάρος με την αρχική, ή να μην υπάρχει διαδρομή από τον s στον t γιατί η ακμή ήταν γέφυρα και τελευταία περίπτωση είναι να υπάρχει μια διαδρομή απλά να έχει μεγαλύτερο βάρος από την ελαφρύτερη διαδρομή. Ο συνολικός χρόνος για την άσκηση είναι: **O(|n| + |m|*log|m|)+O([length(dijkstra_list)]*[|n| + |m|*log|m|]) + O(|m|)+ O(|m|)= O([length(dijkstra_list)]*[|n| + |m|*log|m|]).** Η μόνη βελτίωση που δέχεται είναι στην χειρότερη περίπτωση, όπου αν μετά την διαγραφή η all_edge_list είναι κενή τότε οποιαδήποτε ακμή είναι γέφυρα άρα όποια και ακμή να διαγράψουμε στο G-e δεν θα έχουμε μονοπάτι από τον s στον t.