Κωνσταντίνος Σκορδούλης

ΑΜ : 1115 2016 00155

Η εργασία έχει υλοποιηθεί σε C++. Οι δομές δεδομένων (που αντιστοιχούν σε κλάσεις) που έχω χρησιμοποιήσει υπάγονται στην απλή συνδεδεμένη λίστα (single linked list).

Η δομή του γράφου μου έχει ως εξής:

- Class **Graph:** Περιέχει δείκτη σε απλή συνδεδεμένη λίστα(VList), που οι κόμβοι της(VNodes) περιέχουν δείκτη σε κορυφές (Vertices)
- Class **VList**: Περιέχει ένα δείκτη στον πρώτο κόμβο(VNode* head) και έναν δείκτη στον τελευταίο κόμβο(VNode* tail)
- Class **VNode**: Περιέχει ένα δείκτη, δείχνει στην κορυφή(Vertex* vertex) και έναν δείκτη στον επόμενο κόμβο (VNode* next)
- Class **Vertex**: Περιέχει το όνομα του(char* name) , έναν δείκτη σε λίστα εισερχόμενων ακμών (EList* incoming) και ένα δείκτη σε λίστα εξερχόμενων ακμών(EList* outgoing)
- Class **EList**: Περιέχει ένα δείκτη στον πρώτο κόμβο(ENode* head) και έναν δείκτη στον τελευταίο κόμβο(ENode* tail)
- Class **ENode**: Περιέχει ένα δείκτη, δείχνει στην ακμή(Edge* edge) και έναν δείκτη στον επόμενο κόμβο (ENode* next)
- Class **Edge**: Περιέχει το βάρος (int weight), δείχνει στην κορυφή προέλευσης (Vertex* source) και έναν δείκτη στην κορυφή προορισμού (Vertex* dest). Με αυτόν τον τρόπο ορίζω την κατεύθυνση του κόμβου.

Constructors/Destructors

1) Όταν δημιουργώ έναν γράφο **Graph**(σαν object), δεσμεύω χώρο μνήμης για την ***VList***, καλώντας τον constructor της, που αρχικοποιεί : head = NULL και tail = NULL. Από εδώ και πέρα, η εισαγωγή μιας κορυφής **Vertex**, απαιτεί την δημιουργία(δέσμευση μνήμης) ενός κόμβου **VNode**, ο οποίος εισάγετε στην **VList** .
2) Αν θέλω να εισάγω μια ακμή **Edge**, της μορφής V1->w->V2 , θα πρέπει να δεσμεύσω χώρο μνήμης για τη δημιουργία ενός κόμβου **ENode**, ο οποίος θα εισάγεται : α) στην **EList* incoming** του V2, β) στην **EList* outgoing** του V1
3) Η δημιουργία ενός **ENode, VNode** δεν συνεπάγεται **πάντα** την ταυτόχρονη δημιουργία **κορυφής ή ακμής**. Δηλαδή μπορεί να δημιουργηθεί ένα ENode και να δείχνει σε υπάρχουσα ακμή, αντίστοιχα και για την VNode.
4) Όταν κληθεί η **delete Graph**, θα γίνει chain calling destructors:
Graph -> VList ->(each) VNode -> Vertex -> 2 ELists -> (each) ENode -> Edge.
5) Έχουν παρθεί οι αναγκαίες προφυλάξεις για να μην προκύψει κάτι απρόοπτο, πχ αν σβήσουμε μια ακμή να μην κληθούν οι destructors των κορυφών.

Delete

1) Για να διαγράψω μια **ακμή (Edge)** V1->w->V2**,** πρέπει πρώτα να ενημερώσω την **EList *Incoming** του V2. Δηλαδή να βρω το **ENode** του V2, το οποίο δείχνει σε αυτήν την ακμή, και να το αφαιρέσω από την **Incoming EList**(χωρίς να διαγράψω την ακμή ακόμα).Τέλος γυρνάω στο V1 και αφαιρώ και το ENode και την ακμή. Γενικεύοντας έχουμε τα εξής βήματα :
   - **a.** Βρίσκω το ENode της V1, που δείχνει σε αυτή την ακμή
   - **b.** Μέσω αυτής της ακμής περνάω απέναντι στο V2(Vertex* destination or Vertex* source ανάλογα)
   - **c.** Ψάχνω το ENode της V2, που δείχνει σε αυτήν την ακμή, και το αφαιρώ από την **EList**, **χωρις να σβήσω την ακμή**.
   - **d.** Γυρνάω στο V1 , αφαιρώ το ENode από την EList, σβήνοντας και την ακμή
2) Για την deleteVertex, απλά σβήνω όλες τις εισερχόμενες/εισερχόμενες ακμές (όπως είπα πάνω) και μετά σβήνω το Vertex.
3) DeleteEdges(V1,V2) , δηλαδή σβήνω όλες τις ακμές μεταξύ των δυο (όπως είπα πάνω)

ModEdge/Receiving

1) **ModEdge**: Βρίσκω το ENode , άρα και την ακμή , και αλλάζω το weight.
2) **Receiving**: Εκτυπώνω την **Incoming EList** της κορυφής Vertex.

Circlefind/Findcircles/Traceflow

1) Επειδή υπήρχε μια γενική σύγχυση περί κύκλων, εγώ κατάλαβα τα εξής :
   a. **Simple cycles:** δεν επιτρέπεται η επανάληψη κορυφών και ακμών. Άρα για το μονοπάτι ισχύει :
      i. ***No** duplicate* **Vertices** (can't revisit any Vertex) Η μόνη επανάληψη κόμβων είναι η αρχή και το τέλος (δεν επιτρέπεται να έχουμε A->100-> B->200 -> A-> 300-> A, θα εκτυπωθεί ως A->100->B->200->A και A->300->A)**.**
      ii. ***No** duplicate* **Edges** (αυτό ισχύει γενικά για τους κύκλους).
   b. **Cycles:** δεν επιτρέπεται η επανάληψη ακμών, επιτρέπεται η επανάληψη κόμβων. Άρα για το μονοπάτι ισχύει :
      i. ***Yes** duplicate* **Vertices** (can revisit any Vertex). Εδώ μας ενδιαφέρει μόνο μην υπάρχει επανάληψη ακμών και ο αρχικός κόμβος να είναι ίδιος με το τελικό.

ii. ***No*** *duplicate* **Edges** (αυτό ισχύει γενικά για τους κύκλους)

2) ***Circlefind***: βρίσκει τους απλούς κύκλους μόνο.
3) ***FindCircles:*** βρίσκει όλους τους κύκλους (συμπεριλαμβάνονται και οι απλοί)
4) ***Traceflow(V1,V2,l)***: Βρίσκει (μακρύτατο)μονοπάτι από το V1,V2. Δεν επιτρέπεται η επανάληψη ακμών. Δύο φορές γίνεται έλεγχος:
    - ***a.*** Όταν l = 0, τσεκάρουμε αν η κορυφή που βρισκόμαστε είναι η V2. Αν ναι , εκτυπώνουμε το μονοπάτι. Αν όχι επιστρέφουμε στην προηγούμενη κορυφή (more on this later).
    - ***b.*** Όταν βρεθούμε σε dead end, δηλαδή όταν δεν μπορούμε να περάσουμε από καμία ακμή.

Υλοποίηση των παραπάνω -----→ DFS with Backtracking

https://stackoverflow.com/questions/546655/finding-all-cycles-in-a-directed-graph

Η λύση μου είναι εμπνευσμένη από αυτόν τον ψευδοκώδικα , αλλά η **υλοποίηση** είναι καθαρά δική μου.

Ουσιαστικά, χρησιμοποιώ μια **EList**(και μια **VList** όταν ψάχνω για **simple cycles**), ανεξάρτητη/(τες) από το γράφο, για να κρατάω το **path** (και τους **visited Vertices**) μου και να μην έχω επανάληψη ακμών (και κορυφών).

Κάθε μια παραπάνω συνάρτηση έχει τη δική της DFS συνάρτηση (circlefind -> DFS, findcircles ->DFS1, traceflow -> DFS2)

Και οι 3 DFS είναι ***αναδρομικές***.

Και οι 3 DFS έχουν ***backtracking*** . Διαφέρουν στα σημεία ελέγχου, και σε κάποιες παραμέτρους (πχ η **k στην findcircles και το l στην traceflow**)

Πιστεύω πως τα σχόλια του κώδικα , θα καλύψουν όποιες απορίες προκύψουν.

Exit

Καλούμε *delete G* όπου G = Graph*, και έχουμε **chain calling destructors** (περιγράφεται παραπάνω).

Για να φανεί καλύτερα , η πορεία δέσμευσης/αποδέσμευσης της μνήμης (δηλαδή των destructors) , βγάλτε τα **//** τις **cout** στους constructors/destructors.

Κάνουμε και delete buffer, αποδεσμεύοντας τον χώρο μνήμης.