

Εθνικό Μετσόβιο Πολυτεχνείο



Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Θεμελιώδη Θέματα Επιστήμης Υπολογιστών 2016-2017
2η Σειρά Γραπτών Ασκήσεων

Όνομα: Παναγιώτης Κωστοπαναγιώτης
Α.Μ: 03115196

Άσκηση 1. (Αναδρομή-Επανάληψη-Επαγωγή)

(α) Ας συμβολίσουμε ως $H(n)$ το πλήθος των κινήσεων που κάνει ο αναδρομικός αλγόριθμος για να μετακινήσει η δίσκους από τον έναν στύλο στον άλλον. Από τον ορισμό του αναδρομικού αλγορίθμου, καθώς και από το γεγονός ότι τα υποπροβλήματα που επιλύει ο αναδρομικός αλγόριθμος είναι συμμετρικά μεταξύ τους, συμπαιχένουμε πως ισχύει η ακόλουθη αναδρομική σχέση:

$$H(n) = 2H(n-1) + 1, \quad H(1) = 1$$

Θα δείξουμε, μέσω της μεθόδου της μαθηματικής επαγωγής ότι είναι $H(n) = 2^n - 1$.

Απόδειξη:

- Για $n = 1$, ισχύει $H(1) = 2^1 - 1 = 1$.
- Υποθέτουμε πως η υπόθεση ισχύει από 1, μέχρι και $n - 1$. Θα δείξουμε ότι ισχύει και για n .Έχουμε $H(n) = 2H(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 2 + 1 = 2^n - 1$.

Συνεπώς, οι συνολικές κινήσεις που κάνει ο αλγόριθμος για n δίσκους είναι $2^n - 1$.

(β) Ο επαναληπτικός αλγόριθμος για τον πύργο του Hanoi, αρχικά μετακινεί τον πύργο μεγέθους $n - 1$ στον βοηθητικό στύλο, έπειτα μετακινεί τον τελευταίο δίσκο στον διαθέσιμο στύλο και εντελώς συμμετρικά, όπως στο πρώτο βήμα, επαναποθετεί τον πύργο μεγέθους $n - 1$ στον τρίτο στύλο.

Παρατηρούμε πως η διαδικασία είναι ακριβώς ανάλογη με του αναδρομικού αλγορίθμου και συνεπώς εκτελούνε ακριβώς τον ίδιο αριθμό κινήσεων. Όπως και πάνω, μπορούμε να εφαρμόσουμε ακριβώς την ίδια επαγωγή για να αποδείξουμε και αυστηρά την εγκυρότητα του επιχειρήματός μας.

(γ) Αρχικά, είναι προφανές ότι η βέλτιστη λύση για να μετακινήσουμε έναν δίσκο είναι 1 κίνηση.

Έστω πως γνωρίζαμε την βέλτιστη λύση για να μετακινήσουμε n δίσκους, έστω $opt(n)$, μπορούμε να εφαρμόσουμε την βέλτιστη λύση αυτή προκειμένου να βρούμε βέλτιστη λύση για $n+1$ δίσκους.

Γνωρίζουμε πως ο τελευταίος δίσκος σίγουρα πρέπει να μετακινηθεί στον τελευταίο στύλο, αυτό όμως απαιτεί οι n δίσκοι που βρίσκονται από πάνω του να μετακινηθούν, προκειμένου να τον απεγκλωβίσουν και να ξαναμετακινηθούν από πάνω του. Ο τρόπος με τον οποίο θα μετακινήσουμε τους n δίσκους είναι προφανώς η βέλτιστη στρατηγική την οποία γνωρίζουμε.

Λαμβάνουμε συνεπώς: $opt(n+1) = 2opt(n) + 1$ και $opt(1) = 1$.

Αυτή όμως είναι η αναδρομική σχέση που αντιμετωπίσαμε παραπάνω. Συνεπώς ο βέλτιστος αριθμός κινήσεων που απαιτούνται είναι $2^n - 1$.

(δ) Μπορούμε να εφαρμόσουμε τον γενικευμένο αλγόριθμο για τους πύργους του Hanoi, Frame-Stewart. Θέτοντας όπου $r = 4$ και με κατάλληλη επιλογή για το k μπορούμε να μειώσουμε το συνολικό πλήθος κινήσεων σημαντικά, καταλήγοντας σε έναν πολυωνυμικό αριθμό κινήσεων ως προς τους συνολικούς δίσκους.

Ωστόσο, δεν είναι γνωστό ακόμα κατά πόσο ο παραπάνω αλγόριθμος επιτυγχάνει τον ελάχιστο αριθμό κινήσεων ούτε καν για μικρό πλήθος πασσάλων.

Πηγές:

1. https://en.wikipedia.org/wiki/Tower_of_Hanoi#Frame_E2.80.93_Stewart_algorithm
2. https://www2.bc.edu/~grigsbyj/Rand_Final.pdf

Άσκηση 2. (Λογική και Αλγόριθμοι)

Προκειμένου να καταλήξουμε σε αλγόριθμο που λωνυμικού χρόνου για το πρόβλημα της ικανοποιησιμότητας DNF, αρχεί να αποδείξουμε το παρακάτω λήμμα.

Λήμμα: Ένας τύπος σε μορφή DNF είναι ικανοποιήσιμος, αν και μόνο αν υπάρχει clause στο οποίο δεν εμφανίζεται κάποιο literal μαζί με την άρνησή του.

Απόδειξη:

- Ευθύ: Αν υπάρχει κάποιο γινόμενο, στο οποίο δεν εμφανίζεται κάποιο literal με την άρνησή του, μπορούμε να δώσουμε αληθή τιμή σε όποια literals εμφανίζονται ως κατάφαση και ψευδή τιμή σε όσα εμφανίζονται ως άρνηση, δίνοντας έτσι στο γινόμενο τιμή αληθή. Στα υπόλοιπα literals που δεν εμφανίζονται στο γινόμενο δίνουμε αυθαίρετες τιμές. Εφ' όσον η τιμή του γινομένου αυτού είναι αληθής, όλος ο τύπος μας είναι αληθής και άρα έχουμε το ζητούμενο.
- Αντίστροφο: Αν σε κάθε γινόμενο εμφανίζεται κάποιο literal με την άρνησή του, έστω x_i τότε δεν είναι δυνατόν να είναι ικανοποιήσιμος, αφού για να ήταν θα έπρεπε ο τύπος $x_i \wedge \neg x_i$ να ήταν ικανοποιήσιμος το οποίο προφανώς δεν αληθεύει.

Εφαρμόζουμε λοιπόν την κατασκευή που προτείναμε στην ευθεία απόδειξη. Δοκιμάζουμε κάθε clause ξεχωριστά και ελέγχουμε αν κάθε literal εμφανίζεται χωρίς την άρνησή του. Άμα ισχύει αυτό, δοκιμάζουμε το επόμενο διαδοχικά, μέχρι να καταλήξουμε στο τέλος. Αν δεν βρούμε clause στο οποίο να ισχύει η ιδιότητα αποφαινόμαστε πως ο τύπος δεν είναι ικανοποιήσιμος, αλλιώς αποδίδουμε τιμές στο πρώτο γινόμενο για το οποίο ισχύει η ιδιότητα αυτή με τον τρόπο που προτείναμε παραπάνω.

Η συνολική πολυπλοκότητα του αλγορίθμου μας είναι $O(n^2)$, αφού σε κάθε γινόμενο ελέγχουμε κάθε literal με όλα τα υπόλοιπα.

Αλγόριθμος: DNF-SAT

Input: A logical type in DNF

Output: YES/NO, in case of YES a valid assignment

split the expression by \wedge clauses

\forall clause

status \leftarrow TRUE

\forall literal $x_i \in$ clause

\forall literal $x_j \in$ clause

if $x_i = \text{not } x_j$

status \leftarrow FALSE

if status = TRUE

\forall literal $x_i \in$ clause

if x_i contains $\neg i$

assign $x_i \leftarrow$ FALSE

else

assign $x_i \leftarrow$ TRUE

\forall literal $x_i \notin$ clause

randomly assign x_i

return YES, x_1, x_2, \dots, x_n

return NO, \emptyset

Άσκηση 3. (Αλγόριθμοι γράφων/δικτύων)

Εφαρμόζουμε τον αλγόριθμο του Fleury. Ελέγχουμε αρχικά τον βαθμό των κορυφών. Αν υπάρχουν ακριβώς 2 κορυφές με περιττό βαθμό, τότε υπάρχει μονοπάτι euler από την μα στην άλλη. Αν όλες οι κορυφές είναι άρτιους βαθμούς, τότε ο γράφος μας έχει κύκλο euler. Σε διαφορετική περίπτωση, ο γράφος μας δεν περιέχει τίποτα από τα 2.

Για το κομμάτι του αλγορίθμου που βρίσκει το euler path/cycle εφαρμόζουμε απλά μια dfs η οποία διαγράφει μια μια τις ακμές και καλέι αναδρομικά τους γείτονες του κόμβου που εξετάζουμε.

Αλγόριθμος: Euler tour-circuit

Input: $G(V, E)$

Output: $V = v_1, v_2, \dots, v_n$

circuit = \emptyset

dfs(node u)

$\forall v: M[u][v] = \text{TRUE}$

$M[u][v] = \text{FALSE}$

$M[v][u] = \text{FALSE}$

dfs(v)

append node u \rightarrow *circuit*

euler-finder(G)

compute degree[i] $\forall i \in V$

find nodes for which degree[i] is odd

if (number > 2)

return \emptyset

if (number = 2)

pick one of these nodes s

dfs(s)

if (number = 0)

pick random node s

dfs(s)

return circuit

Η πολυπλοκότητα του αλγορίθμου μας είναι $O(V^2)$ εφ' όσον διασχίζουμε κάθε ακμή μια φορά, αφού στην χειρότερη περίπτωση που ο γράφος μας είναι πυκνός, το πλήθος των ακμών είναι τετραγωνικό ως προς το πλήθος των κόμβων.

Άσκηση 4. (Αλγόριθμοι γράφων/δικτύων)

Προκειμένου να αποκτήσουμε καλύτερη κατανόηση του προβλήματος, με σκοπό να καταλήξουμε σε μια παραλλαγή του αλγορίθμου Ford-Fulkerson είναι χρήσιμο να εκφράσουμε το αντίστοιχο πρόβλημα ροής σαν γραμμικό πρόγραμμα. Η περιγραφή αυτή είναι η παρακάτω:

- $f_{u,v} \geq c_{u,v} \quad \forall (u,v) \in E$
- $\sum_{v: u \in adj_v} f_{v,u} = \sum_{v' \in adj_u} f_{u,v'} \quad , \text{ flow conservation property}$
- $\min(\sum_{u \in adj_s} f_{s,u}) \quad , \text{ objective function}$

Βάση της παραπάνω έκφραση καταλήγουμε στον εξής αλγόριθμο. Η βασική διεργασία για την εύρεση augmenting path δεν αλλάζει, καθώς και όλο το residual network παραμένει ίδιο με το αντίστοιχο πρόβλημα μέγιστης ροής. Η αλλαγή έγκειται στο ότι μόλις βρούμε ένα augmenting path, αυξάνουμε το flow σε κάθε ακμή του κατά το μέγιστο δυνατό, ώστε όλες οι ακμές του μονοπατιού να ικανοποιούν τον πρώτο περιορισμό του γραμμικού προγράμματος.

Η περιγραφή του αλγορίθμου λοιπόν είναι κατά βάση η εξής. Κατασκευάζουμε το residual network και κάθε φορά ψάχνουμε να βρούμε ένα augmenting path. Μόλις το βρούμε αναζητάμε την ακμή μέγιστου βάρους στο residual network και αυξάνουμε το flow κατά μήκος του μονοπατιού κατά το βάρος αυτό. Αν κάποια στιγμή δεν βρούμε augmenting path τερματίζουμε τον αλγόριθμο.

Ο ψευδοκώδικας του αλγορίθμου φαίνεται παρακάτω:

Αλγόριθμος: Minimum flow with constraints
Input: $G(V,E)$
Output: minimum flow, f , satisfying constraints

```

 $f=0$ 
while augmenting-path( $G$ )=TRUE
     $aug=0$ 
     $\forall e(u,v) \in$  augmenting path
         $aug=\max(aug, c_{u,v} - f_{u,v})$ 
     $f=f+aug$ 
     $\forall e(u,v) \in$  augmenting path
         $f_{u,v}=f_{u,v}+aug$ 
         $f_{v,u}=f_{v,u}-aug$ 

```

Η εύρεση augmenting-path γίνεται απλά με μια BFS στο residual network, η συνολική πολυπλοκότητα του αλγορίθμου μας είναι αντίστοιχη με του αλγορίθμου Edmonds-Karp αφού οι μόνες αλλαγές στον αλγόριθμο Edmonds-Karp είναι σταθερής πολυπλοκότητας. Συνεπώς η πολυπλοκότητα του αλγορίθμου μας είναι $O(VE^2)$.

Στο δίκτυο που περιγράψαμε παραπάνω ισχύει το θεώρημα ελάχιστης ροής-μέγιστης τομής.

Θεώρημα: Η τιμή της μέγιστης τομής ταυτίζεται με την τιμή της ελάχιστης ροής στο παραπάνω δίκτυο.

Άσκηση 5. (Αλγόριθμοι γράφων/δικτύων)

(α) Θα χρησιμοποιήσουμε την μέθοδο της μαθηματικής επαγωγής. Θα δείξουμε ότι, σε κάθε βήμα i , ο αλγόριθμος υπολογίζει όλα τα συντομότερα μονοπάτια μεγέθους i και άρα στο τελευταίο βήμα n θα έχει υπολογίσει όλα τα συντομότερα μονοπάτια μεγέθους m μέχρι και $n-1$ που είναι το μέγιστο μήκος απλού μονοπατιού σε γράφο με n κόμβους.

- **Βάση της επαγωγής:** Στο πρώτο βήμα, κάνει relax μια-μια τις ακμές ξεχωριστά και συνεπώς υπολογίζει τα συντομότερα μονοπάτια μεγέθους τουλάχιστον 1.
- **Επαγωγικό βήμα:** Έστω πως ο αλγόριθμος μας έχει υπολογίσει όλα τα συντομότερα μονοπάτια μεγέθους τουλάχιστον k . Στο επόμενο βήμα θα κάνει relax όλες τις ακμές και συνεπώς θα επεκτείνει όλα τα μονοπάτια που είναι υπολογισμένα μέχρι στιγμής κατά 1 ακμή. Συνεπώς, στο τέλος του βήματος θα έχει υπολογίσει όλα τα συντομότερα μονοπάτια μεγέθους $k+1$.

(β) Η τροποποίηση που πρέπει να κάνουμε στον αλγόριθμό Bellman-Ford προκειμένου να εντοπίζει αρνητικούς κύκλους είναι να προσθέσουμε στο τέλος ένα ακόμα βήμα στο οποίο να ελέγχει αν υπάρχει ακμή η οποία μπορεί να γίνει relaxed. Αν υπάρχει, τότε θα αυτό συνεπάγεται αυτόματα την

ύπαρξη αρνητικού κύκλου.

Παρακάτω φαίνεται ο τροποποιημένος αλγόριθμος Bellman-Ford.

Αλγόριθμος: Negative Cycle Detection

Input : $G(V, E)$

Output : TRUE / FALSE

```
relax(edge(u, v, w))
    if (dist[v] > dist[u] + w)
        dist[v] ← dist[u] + w
    return TRUE
return FALSE
```

negative-cycle(G(V, E), node s)

$\forall v \in V$

$dist[v] = \infty$

$dist[s] = 0$

for $1 \leq i \leq V$

$\forall e \in E$

$relax(e)$

$\forall e \in E$

 if $relax(e) = \text{TRUE}$

 return TRUE

 return FALSE

Η πολυπλοκότητα του παραπάνω αλγορίθμου με την προσθήκη είναι ακριβώς όση και του απλού Bellman-Ford, αφού προσθέσαμε απλά μια γραμμική διαδικασία. Συνεπώς, έχουμε πολυπλοκότητα O(VE).

Η ορθή λειτουργία του παραπάνω αλγορίθμου επαφίεται άμεσα στην ορθή λειτουργία του αλγορίθμου Bellman-Ford. Έχουμε από το (α) πως ο αλγόριθμος Bellman-Ford, μόλις τερματίζει έχει υπολογίσει σωστά τις ελάχιστες αποστάσεις από τον κόμβο s . Συνεπώς, άμα μετά κάποια ακμή (u, v) μπορεί να γίνει relaxed, η απόσταση $dist[v]$ δεν θα αποτελεί απόσταση πάνω σε απλή διαδρομή διότι ο αλγόριθμος Bellman-Ford έχει ήδη υπολογίσει όλες τις αποστάσεις πάνω σε απλές διαδρομές. Συνεπώς, η απόσταση αυτή αντιστοιχεί σε κάποιο κύκλο αρνητικού μήκους, αφού σε διαφορετική περίπτωση, οι απλές διαδρομές πάντα δίνουν καλύτερες αποστάσεις.

Τελικά, συμπαιρένουμε ότι ο αλγόριθμος Bellman-Ford, με την παραπάνω προσθήκη αποφαίνεται ορθά για την ύπαρξη ή μη αρνητικού κύκλου.

Άσκηση 6. (Δυναμικός Προγραμματισμός)

(α) Για $k = 1$, μπορούμε να δοκιμάσουμε μέχρι να σπάσουμε την μπάλα μια φορά, συνεπώς το καλύτερο που έχουμε να κάνουμε είναι γραμμική αναζήτηση από πάνω προς τα κάτω. Για $k = 2$ ωστόσο, μπορούμε να κάνουμε κάτι καλύτερο, εξετάζουμε τα πολλάπλασια του $\sqrt{n} = 10$, μόλις η μπάλα μας σπάσει εξετάζουμε από το προηγούμενο πολλαπλάσιο μέχρι το τωρινό. Έτσι, στην χειρότερη περίπτωση θα χρειαστούμε $2 * \sqrt{n} - 1 = 19$ δοκιμές.

(β) Έστω ότι έχουμε στην διάθεσή μας k μπάλες και n ορόφους, επιλέγοντας έναν όροφο, έστω i , έχουμε 2 πιθανές επιλογές, είτε η μπάλα θα σπάσει, το οποίο σημαίνει ότι πρέπει πλέον να ελέγχουμε $i - 1$ ορόφους χρησιμοποιώντας $k - 1$ μπάλες, είτε με τον ίδιο αριθμό μπαλών να δοκιμάσουμε $n - i$ ορόφους. Σκοπός μας είναι να πετύχουμε το i το οποίο μας ελαχιστοποιεί τον μέγιστο αριθμό δοκιμών στην χειρότερη περίπτωση.

Γενικά, ας ορίσουμε $dp[n, k]$ ως τον ελάχιστο αριθμό δοκιμών που θα χρειαστεί να κάνουμε στην χειρότερη περίπτωση αν θέλουμε να ελεγξουμε η ορόφους με k μπάλες. Δεδομένης της παραπάνω περιγραφής αναδρομικής σχέσης καθώς και της αρχής της βέλτιστητας, καταλήγουμε στην ακόλουθη αναδρομική σχέση:

base case: $dp[0, 0] = 0$

αναδρομική σχέση: $dp[n, k] = \min_{1 \leq i \leq n} (\max(dp[i-1, k-1], dp[n-i, k]))$

Η ορθότητα της αναδρομικής σχέσης οφείλεται στο εξής επιχείρημα. Έστω πως γνωρίζαμε μια ακολουθία δοκιμών η οποία μας ελαχιστοποιεί το πλήθος των δοκιμών στην χειρότερη περίπτωση. Αυτή η ακολουθία δοκιμών, πρέπει σαφώς να έχει μια πρώτη κίνηση, η οποία δεν μπορεί παρά να είναι ένας όροφος εκ των 1, 2, ..., n .

Έστω ότι μπορούσαμε να μαντέψουμε αυτή την πρώτη κίνηση (έστω i). Τότε, έχουμε 2 περιπτώσεις, όπως αναφέραμε παραπάνω, από τις οποίες πρέπει να επιλέξουμε την μεγαλύτερη, αφού αναζητάμε την χειρότερη περίπτωση. Δοκιμάζοντας κάθε πιθανό όροφο σαν πρώτη κίνηση και λαμβάνοντας αναδρομικά την απάντηση για κάθε περίπτωση θα πάρουμε τελικά το βέλτιστο.

Η τελική μας απάντηση είναι το $dp[n, k]$.

Η συνολική πολυπλοκότητα του αλγορίθμου μας είναι $O(kn^2)$, όπου k ο αριθμός των μπαλών και n το πλήθος των ορόφων

(γ) Το πρόγραμμα που υλοποιεί τον παραπάνω αλγόριθμο βρίσκεται στο balls.cpp. Στον κώδικα υπολογίζουμε την απάντηση μέχρι και για 300 ορόφους και 300 μπάλες. Ωστόσο, στο output τυπώνεται ένας τετραγωνικός πίνακας μεγέθους 25x25 λόγω χώρου.

Σε περίπτωση ελέγχου του προγράμματος τα όρια αυτά μπορούν να αυξηθούν ανάλογα.

Άσκηση 7. (Αναγωγές προς επίλυση προβλημάτων)

(α) Κατασκευάζουμε έναν εξαιρετικά απλό λαβύρινθο με μια είσοδο, μια έξοδο και καθόλου εσωτερικούς τοίχους. Το πρόβλημα φαίνεται παρακάτω:

```
4 5 0
1 1
4 5
```

Η είσοδος βρίσκεται στο κελί (1,1) και η έξοδος στο κελί (4,5)

(β) Το πρόβλημα το οποίο μας ζητείται να λύσουμε παρομοιάζει με το πρόβλημα εξερεύνησης γράφου. Συνεπώς, πρέπει να ανάγουμε το πρόβλημα μας σε κάποιο πρόβλημα που αφορά γράφους.

Θεωρούμε πως κάθε κόμβος αποτελείται από 2 στοιχεία που ορίζουν την γραμμή και την στήλη του αντίστοιχα. Για κάθε κόμβο κρατάμε έναν δυαδικό αριθμό, 4 ψηφίων, όπου κάθε ψηφίο του εκφράζει αν ο κόμβος συνδέεται με τόν αριστερά του, τον από κάτω, τον δεξιά και τον πάνω αντίστοιχα. Αρχικά θεωρούμε πως έχουμε όλες τις συνδέσεις και όσο διαβάζουμε τοίχους αφαιρούμε ακμές.

Έτσι, έχουμε έναν γράφο στον οποίο υπάρχει ισοδυναμία ως προς την συνεκτικότητα με τον αρχικό μας λαβύρινθο. Τώρα, αρχεί να εκτελέσουμε έναν αλγόριθμο αναζήτησης (πχ DFS), από τον κόμβο s και να ελέγξουμε αν ο κόμβος t είναι visited στο τέλος, όπου οι κόμβοι s και t αντίστοιχον στην είσοδο και στην έξοδο του λαβύρινθου αντίστοιχα. Σε αυτή την περίπτωση, θα κρατάμε και το dfs δέντρο, οπότε θα εμφανίσουμε και το μονοπάτι, αυτό καθ' αυτό.

Πολυπλοκότητα: Στην χειρότερη περίπτωση θα χρειαστεί να διατρέξουμε ολόκληρο τον λαβύρινθο (πχ σε περίπτωση που το μονοπάτι από την είσοδο στην έξοδο είναι μονοπάτι hamilton) και άρα η DFS έχει πολυπλοκότητα χειρότερης περίπτωσης $O(nm)$. Επειδή πρέπει να διαβάσουμε και την

είσοδο η συνολική πολυπλοκότητα του αλγορίθμου μας είναι $O(nm + k)$. Παρατηρούμε πως αυτή είναι και η βέλτιστη πολυπλοκότητα χειρότερης περίπτωσης αφού πάντα, ανεξάρτητα το τι θα κάνει ο αλγόριθμός μας θα χρειαστεί να διαβάσουμε το input (μεγέθους k) και να εκτυπώσουμε το μονοπάτι (μεγέθους nm στην χειρότερη περίπτωση).

(γ) Παρατηρούμε ότι τα βάρη των ακμών του γράφου μας είναι μοναδιαία. Συνεπώς, η μόνη διαφοροποίηση στον αλγόριθμό μας είναι πως αντί για DFS θα χρειαστεί να χρησιμοποιήσουμε BFS για να βρούμε τις ελάχιστες διαδρομές. Εφ' όσον η πολυπλοκότητα του αλγορίθμου BFS είναι ίδια με αυτή της DFS η πολυπλοκότητα του αλγορίθμου μας παραμένει $O(nm + k)$.

(δ) Στα αρχεία dfs_maze.cpp, bfs_maze.cpp μπορούμε να βρούμε το πηγαίο αρχείο ακόδικα που υλοποιεί τον αλγόριθμο του ερωτήματος (γ) και (δ) αντίστοιχα.

Δεπτομέρειες υλοποίησης: Για την διευκόλυνσή μας στην συγγραφή του κώδικα χρησιμοποιούμε τις δομές της std::queue και std::stack για την λειτουργία της bfs καθώς και για την εκτύπωση του μονοπατιού. Τα όρια του λαβυρίνθου έχουν τεθεί περίπου 2000, που είναι ένα καλό όριο αν θέλουμε ο αλγόριθμός μας να τερματίζει εντός των 1 δευτερολέπτου. Προκειμένου να γίνει ο έλεγχος του προγράμματος τα όρια αυτά μπορούν να αυξηθούν.

Άσκηση 8. (Αναγωγές προς απόδειξη δυσκολίας)

Αρχικά, θα δείξουμε ότι το πρόβλημα Hamilton Path είναι στο NP. Πράγματι, αν θέλουμε να ελέγξουμε αν ένα δεδομένο μονοπάτι είναι μονοπάτι Hamilton, αρκεί να ελέγξουμε αν περιέχει όλους τους κόμβους καθώς και αν είναι έγκυρο μονοπάτι του γράφου. Ο παραπάνω έλεγχος είναι γραμμικού χρόνου και άρα το Hamilton Path είναι στο NP.

Αρκεί τώρα να δείξουμε ότι το Hamilton-Cycle, το οποίο γνωρίζουμε πως είναι NP-Complete, ανάγεται στο Hamilton-Path. Δηλαδή θα δείξουμε ότι, για κάθε γράφο, υπάρχει μετασχηματισμός στον οποίο υπάρχει Hamilton-Path αν και μόνο αν υπάρχει Hamilton-Cycle στον αρχικό γράφο. Αν καταφέρουμε να το δείξουμε αυτό, η απόδειξή μας έχει ολοκληρωθεί, αφού σε αυτή την περίπτωση το Hamilton-Path είναι NP-Hard, αλλά και στο NP, συνεπώς NP-Complete. Σημειώνεται ότι προκειμένου να ισχύουν τα παραπάνω πρέπει ο μετασχηματισμός να γίνεται σε πολυωνυμικό χρόνο ως προς το μέγεθος της εισόδου.

Αναγωγή: Έστω ένας τυχόν γράφος με n τα πλήθος κόμβους. Επιλέγουμε έναν από αυτούς, έστω τον κόμβο 1, και γι' αυτόν παίρνουμε όλους τους γειτονές του, έστω u_1, u_2, \dots, u_k . Για κάθε έναν από αυτούς, έστω u_i , εισάγουμε τις εξής νέες ακμές $(1, s), (u_i, v), (v, t)$, όπου $s, t, v \notin V$. Ο παραπάνω μετασχηματισμός απαιτεί πολυωνυμικό χρόνο και ικανοποιεί τις συνθήκες που απαιτήσαμε παραπάνω.

Απόδειξη:

- **Ευθύ:** Θα δείξουμε ότι για κάθε γράφο G που περιέχει Hamilton Cycle, ο μετασχηματισμένος γράφος G' περιέχει Hamilton Path από τον κόμβο s στον κόμβο t . Πράγματι, αν ο γράφος περιέχει κύκλο Hamilton, μπορούμε να θεωρήσουμε χωρίς βλάβη της γενικότητας ότι ο κύκλος ξεκινάει από τον κόμβο 1, αυτός ο κύκλος θα περιέχει έναν τελευταίο κόμβο, οποίος θα είναι γείτονας του κόμβου 1. Ας συμβολίσουμε αυτον τον κόμβο v' . Αφού είναι γείτονας του κόμβου 1, θα υπάρχει ακμή (v', v) και έπειτα ακμή (v, t) . Επίσης, έχουμε την ακμή $(s, 1)$, συνεπώς το μονοπάτι $(s, 1)$, hamilton-cycle, $(v', v), (v, t)$ είναι μονοπάτι hamilton από τον κόμβο s στον κόμβο t , το οποίο είναι και το ζητούμενο.
 - **Αντίστροφο:** Θα δείξουμε επιπλέον ότι αν ο μετασχηματισμένος γράφος G' περιέχει μονοπάτι hamilton, τότε ο αρχικός G θα περιέχει κύκλο hamilton. Έστω ότι υπάρχει μονοπάτι μονοπάτι hamilton στον γράφο G' , έστω $(s, 1), (1, u_1), \dots, (u_i, v'), (v', v), (v, t)$. Εφ' όσον υπάρχει ακμή από τον v' στον v αυτό σημαίνει ότι ο v' είναι γείτονας του κόμβου 1, αφού μόνο στους γείτονες του 1 προστέθηκε ακμή στον v . Συνεπώς υπάρχει η ακμή $(v', 1)$.

Επιπλέον, αν αφαιρέσουμε την ακμή $(s,1)$ και τις ακμές $(v',v),(v,t)$ αυτό που απομένει είναι μονοπάτι hamilton από τον κόμβο 1 στον κόμβο v' του αρχικού γράφου, συνεπώς με την προσθήκη της ακμής $(v',1)$, λαμβάνουμε έναν κύκλο hamilton στον αρχικό γράφο.

Άσκηση 9. (Κρυπτογραφία)

(α) Για την υλοποίηση του τεστ του Fermat χρησιμοποιούμε την γλώσσα προγραμματισμού python. Η επιλογή αυτή έγινε διότι η python υποστηρίζει big numbers, το οποίο μας απαλλάσσει από το να υλοποιήσουμε δικιές μας συναρτήσεις για χειρισμό μεγάλων αριθμών. Ο υπολογισμός των δυνάμεων γίνεται μέσω του αλγορίθμου modular exponentiation.

Το αρχείο κώδικα που υλοποιεί τον αλγόριθμο βρίσκεται στο fermat.py.

Δοκιμές: Παρατηρούμε πως για μεγάλους πρώτους καθώς και για ημιπρώτους το πρόγραμμά μας βγάζει σωστά αποτελέσματα. Ωστόσο, δοκιμάζοντας αριθμούς Carmichael, παρατηρούμε πως το πρόγραμμά μας βγάζει λάθος αποτέλεσμα για κάθε έναν από αυτούς, ακόμα και με αυξημένο πλήθος δοκιμών.

Για να αποφύγουμε τα παραπάνω προβλήματα υλοποιούμε τον αλγόριθμο Miller-Rabin.

(β) Η υλοποίηση του αλγορίθμου Miller-Rabin βρίσκεται στο miller_rabin.py

Δοκιμάζοντας τον αλγόριθμο τόσο με μεγάλους πρώτους, όσο και με αριθμούς Carmichael παρατηρούμε πως εμφανίζει σωστά αποτελέσματα, σε αντίθεση με το τεστ του Fermat.