Ανάλυση και Σχεδιασμός Αλγορίθμων

Προαιρετική εργασία

Team 50

Αξιμιώτης Δημήτριος, 10622
Φραϊδάκης Ιωάννης, 10736

Πρόβλημα 1

1.1)

Τρόπος σκέψης :

Το ζητούμενο είναι να βρούμε αν υπάρχει διαδρομή μεταξύ 2 πόλεων (s και t) του γράφου G. Σκεφτόμαστε να χρησιμοποιήσουμε είτε τον αλγόριθμο DFS είτε τον BFS αφού και οι δύο βρίσκουν όλους τους κόμβους οι οποίοι είναι προσβάσιμοι από έναν αφετηριακό κόμβο s. Οπότε, προφανώς εντοπίζουν και το αν υπάρχει διαδρομή μεταξύ του s και του t. Τα πράγματα όμως δεν είναι τόσο απλά καθώς υπάρχει μια μικρή διαφοροποίηση από ένα κοινό γράφημα. Ενδέχεται να υπάρχουν οδικές αρτηρίες τις οποίες δεν μπορούμε να διασχίσουμε εξαιτίας της χωρητικότητας του ρεζερβουάρ και της έλλειψης βενζινάδικων κατά μήκος των αρτηριών για επαναγέμισμα αυτού.

Με ορολογία γραφημάτων, δεδομένου ενός αφετηριακού κόμβου s και ενός κόμβου προορισμού t, αναζητούμε διαδρομή που να τους συνδέει, χωρίς όμως να λαμβάνουμε υπόψη τις «απαγορευμένες» ακμές. Στην πράξη, εφόσον δεν μπορούμε να διασχίσουμε αυτές τις ακμές, είναι σαν να μην υπάρχουν. Οπότε απλώς τις αφαιρούμε! Και εδώ είναι που θα στηριχθεί όλη η ιδέα του προτεινόμενου αλγορίθμου παρακάτω. Αν θεωρήσουμε επιπλέον πως γεμίζουμε τα καύσιμα σε κάθε πόλη, εύκολα γίνεται αντιληπτό πως όλες οι εναπομείνασες ακμές μπορούν να διασχισθούν. Άρα έχουμε ένα σύνηθες γράφημα.

Περιγραφή αλγορίθμου :

Αρχικά διατρέχουμε όλες τις ακμές του γράφου και κάθε ακμή που εντοπίζουμε με μήκος $l_e > L$ («απαγορευμένη» ακμή), την αφαιρούμε από το γράφημα – χρονική πολυπλοκότητα $\boldsymbol{O}(|\boldsymbol{E}|)$. Πλέον έχουμε έναν κοινό γράφο και εκτελούμε μια Breadth First Search (επιλέχθηκε αυτή λόγω απλότητας), αρχίζοντας από τον κόμβο s και αγνοώντας τα βάρη/αποστάσεις των ακμών αφού 1ον: μπορούμε να τις διασχίσουμε όλες και 2ον: δεν ενδιαφερόμαστε για την εύρεση ελάχιστης απόσταση αλλά μονάχα για την ύπαρξη διαδρομής μεταξύ s και t – χρονική πολυπλοκότητα $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$. Τέλος, αν φτάσουμε σε ένα σημείο όπου θα κάνουμε dequeue τον κόμβο t τότε έχει βρεθεί μια τέτοια διαδρομή, ενώ αν ολοκληρωθεί η BFS αναζήτηση χωρίς να εξετάσουμε/γκριζάρουμε τον κόμβο t, τότε δεν υπάρχει τέτοια. Συνολικά $\boldsymbol{O}(|\boldsymbol{V}| + |\boldsymbol{E}|)$ – αναλυτικότερα βλ. σελίδα 6.

1.2)

Τρόπος σκέψης :

Αναζητούμε την ελάχιστη χωρητικότητα καυσίμων L που απαιτείται για να πάμε από την πόλη s στην πόλη t. Δηλαδή, το ζητούμενο είναι η εύρεση διαδρομής από τον κόμβο s στον t, αν υπάρχει, της οποίας η ακμή με το μεγαλύτερο μήκος να είναι η ελάχιστη απ' όλες τις άλλες διαδρομές s → t . Μοιάζει με πρόβλημα συντομότατων διαδρομών, αφού το ζητούμενο και στις δύο περιπτώσεις είναι η ελαχιστοποίηση κάποιας παραμέτρου σχετιζόμενης με τις ακμές, και εφόσον όλες είναι θετικές, σκεφτόμαστε τον αλγόριθμο δυναμικού προγραμματισμού του Dijkstra (που για κατάλληλη υλοποίηση[1] ικανοποιεί την ζητούμενη χρονική πολυπλοκότητα).

Έστω d_i το ελάχιστο – μέγιστο μήκος – ακμής που χρειάζεται να διασχίσουμε για να φτάσουμε από τον s στον τρέχων κόμβο v_i. Η βασική διαφορά εδώ με τον κλασσικό αλγόριθμο του Dijkstra είναι πως η σειρά επίσκεψης των κόμβων (δηλαδή η ταξινόμηση τους στην ουρά προτεραιότητας) δεν γίνεται με βάση την ελάχιστη απόσταση από τον αφετηριακό κόμβο s αλλά με βάση την μεταβλητή d_i που διατηρεί και ανανεώνει κάθε κόμβος. Εννοείται πως τα βήματα για την μετατροπή του G σε συνήθη γράφο επαναλαμβάνονται όπως στο ερώτημα 1.1)

Περιγραφή αλγορίθμου :

Αρχικά αφαιρούμε τις «απαγορευμένες» ακμές σε χρόνο $\boldsymbol{O(E)}$ όπως προηγουμένως και αρχικοποιούμε όλα τα d_i με άπειρο εκτός από το $d_s = d_0 = 0$. Τώρα χρησιμοποιούμε την ίδια βάση με τον αλγόριθμο Dijkstra (δηλαδή την φιλοσοφία της PriorityQueue) αλλάζοντας μόνο τη συνάρτηση relax. Κάθε φορά που καλείται η relax από έναν κόμβο *i* προς έναν γειτονικό του κόμβο *j,* η ανανέωση του κλειδιού (δηλαδή της προτεραιότητας) του κόμβου *j*, γίνεται ως εξής : Συγκρίνουμε την τρέχουσα τιμή του d_j με το μέγιστο από τα d_i και $e(i,j)$[2] και αν το τρέχων d_j είναι μεγαλύτερο, σημαίνει ότι βρήκαμε μια διαδρομή που οδηγεί στον κόμβο *j* με μικρότερο d και συνεπώς ανανεώνουμε την τιμή του κλειδιού του (d_j) και του πρόγονο του (j.π). Αν κάποια στιγμή η extractMin επιστρέψει τον κόμβο t, τότε ο αλγόριθμος τερματίζεται και η τρέχουσα τιμή d_t είναι η ζητούμενη[3]. Διαφορετικά, δεν υπάρχει διαδρομή από τον s στον t και επιστρέφεται άπειρο[4]. Εφόσον η τροποποιημένη relax εξακολουθεί να γίνεται σε χρόνο $\boldsymbol{O(logV)}$[5], η χρονική πολυπλοκότητα του αλγορίθμου είναι ίδια με αυτή του Dijkstra, δηλαδή $\boldsymbol{O((|V| + |E|) * logV)}$.

---

[1] Υλοποίηση της PriorityQueue με Binary minHeap, οπότε extractMin → $\boldsymbol{O(logV)}$ και insert/decreaseKey → $\boldsymbol{O(logV)}$.

[2] Με e(i,j) συμβολίζουμε το μήκος της ακμής που συνδέει τους δύο κόμβους *i* και *j*.

[3] Η απόδειξη αυτού είναι ίδια με την απόδειξη της ορθότητας του αλγορίθμου του Dijkstra. (βλ. CLRS, έκδοση 3/2021, σελ. 656, Θ 24.6) θέτοντας όπου u.d το u.key και δ(s,u) το d_u .

[4] Προφανώς αυτός ο αλγόριθμος δίνει λύση και για το 1.1) : Αν επιστραφεί άπειρο δεν υπάρχει διαδρομή ενώ για κάθε άλλη τιμή υπάρχει. Ωστόσο η λύση με τον BFS είναι σαφώς γρηγορότερη και παρατέθηκε χάριν πληρότητας.

[5] Στην relax εξακολουθεί να γίνεται μια decreaseKey κάθε φορά που βρίσκουμε συντομότερη διαδρομή για έναν κόμβο.

Πρόβλημα 2

Τρόπος σκέψης[1] :

Θέλουμε να ελαχιστοποιήσουμε το συνολικό χρόνο αναμονής, δηλαδή το άθροισμα των χρόνων αναμονής όλων των πολιτών. Έστω t_j ο χρόνος εξυπηρέτησης του *j* – οστού πολίτη. Τότε ο χρόνος αναμονής του πολίτη με σειρά *k*, έστω T_k, είναι το άθροισμα των χρόνων εξυπηρέτησης των πολιτών που προηγούνται αυτού, δηλ. : $T_k = \sum_{j=1}^{k-1} t_j$. Έτσι, ο συνολικός χρόνος αναμονής όλων το πολιτών, $T_{ολ}$,είναι:

$$T_{o\lambda} = \sum_{k=1}^{n} T_k = \sum_{k=1}^{n} \sum_{j=1}^{k-1} t_j = (n-1)t_1 + (n-2)t_2 + \cdots + t_{n-1}$$

Η λογική ερμηνεία αυτού του τύπου είναι πως, (n-1) πολίτες θα περιμένουν τον 1ο να εξυπηρετηθεί (όλοι εκτός τον 1ο) άρα έχουμε τον όρο $(n-1)t_1$, (n-2) περιμένουν τον 2ο (όλοι εκτός τον 1ο και τον 2ο) άρα έχουμε τον όρο $(n-2)t_2$, κλπ και τέλος μόνο ένας (ο n-οστος) θα περιμένει τον προτελευταίο άρα προκύπτει ο όρος t_{n-1}. Επομένως, το ζητούμενο είναι η ελαχιστοποίηση της παρακάτω έκφρασης : $T_{o\lambda} = (n-1)t_1 + (n-2)t_2 + \cdots + t_{n-1}$. Από ‘δω, παρατηρούμε ότι όσο πιο νωρίς εξυπηρετείται ένας πολίτης, τόσο πιο πολλούς συμπολίτες του θα καθυστερήσει. Γι’ αυτό σκεφτόμαστε πως η βέλτιστη σειρά εξυπηρέτησης των πολιτών θα ταυτίζεται με την κατά αύξουσα σειρά ταξινόμησή τους με βάση τους χρόνους εξυπηρέτησης t_j. Δηλαδή αυτοί με τους μικρότερους χρόνους εξυπηρέτησης, εξυπηρετούνται πρώτα.

Περιγραφή αλγορίθμου :

Απλώς ταξινομούμε τους πολίτες κατά αύξουσα σειρά των χρόνων εξυπηρέτησης του με μια mergeSort, σε χρόνο $\boldsymbol{O(nlogn)}$, και έτσι προκύπτει η ζητούμενη σειρά εξυπηρέτησης.

[1] Για την απόδειξη της ορθότητας του αλγορίθμου βλ. σελίδα 8.

Πρόβλημα 3

Τρόπος σκέψης :

Έχουμε μια συμβολοσειρά, έστω s, και m σημεία διαχωρισμού. Χρησιμοποιώντας δυναμικό προγραμματισμό, θέλουμε να βρούμε την σειρά με την οποία θα γίνουν οι κοπές στην s έτσι ώστε να ελαχιστοποιήσουμε το συνολικό υπολογιστικό κόστος. Μας έρχεται αμέσως στο μυαλό το πρόβλημα «Chain Matrix Multiplication» (Lecture: Dynamic Programming → Problem 5). Πράγματι, η τοποθέτηση παρενθέσεων μεταξύ των πινάκων είναι το ισοδύναμο των σημείων κοπής μεταξύ των στοιχείων της συμβολοσειράς. Η μόνη διαφορά (πέραν προφανώς του τρόπου υπολογισμού του κόστους σε κάθε περίπτωση), έγκειται στο γεγονός ότι εκεί επιλέγαμε εμείς που θα τοποθετήσουμε τις παρενθέσεις ενώ τώρα οι θέσεις κοπής είναι δεδομένες (και δοσμένες). Αυτό απλώς κάνει τα πράγματα πιο εύκολα, αφού πλέον δεν έχουμε να εξετάσουμε κάθε δυνατό συνδυασμό όπως είχαμε στον πολλαπλασιασμό πινάκων αλλά εξετάζουμε μονάχα τις m θέσεις στις οποίες πρέπει να γίνει διάσπαση του s. Επομένως, ο αλγόριθμος θα έχει την ίδια δομή με το CMM problem.

Περιγραφή αλγορίθμου :

Ορίζουμε τον πίνακα $C[i][j]$ ο οποίος αποθηκεύει το ελάχιστο κόστος για την διάσπαση του substring που αρχίζει από το index *i* έως *j* (για τον υπολογισμό, λαμβάνει υπόψη μόνο όσα σημεία τομής του αρχικού προβλήματος βρίσκονται μέσα στο substring)[1]. Προφανώς αν *j<i* ο υπολογισμός είναι αδύνατος (ανύπαρκτο substring) και το συμβολίζουμε με την τιμή -1. Η βάση του DP είναι όταν μεταξύ του *i* και του *j* δεν έχουμε σημείο κοπής, οπότε $C[i][j] = 0$ (δεν θα γίνει ξεχωριστή αρχικοποίηση ως συνήθως, βολεύει να το συμπεριλάβουμε καθώς διατρέχουμε τις θέσεις του πίνακα C, βλ. παρακάτω). Τώρα, όπως και στο CMM, ορίζουμε μία παράμετρο length σε for από 1 έως n, η οποία δηλώνει το μήκος των substring που πρόκειται να εξετάσουμε. Εμφολευμένα σ' αυτή τη for έχουμε άλλη μια for για το σημείο εκκίνησης i, με τιμές από 1 έως $n - length$. Με δεδομένα τα length και i γνωρίζουμε και το πέρας j του substring. Αρχικοποιούμε $C[i][j] = \infty$. Εξετάζουμε αν υπάρχει cutting_point k μέσα στο substring (πρέπει $i \leq k < j$). Αν δεν υπάρχει, βρισκόμαστε στο base case όπως αναφέραμε νωρίτερα. Διαφορετικά, για κάθε τέτοιο k, υπολογίζουμε το κόστος διάσπασης από τον τύπο[2] : $C[i][k] + C[k+1][j] + length$ και αποθηκεύουμε στο $C[i][j]$ το min όλων αυτών. Αφού τελειώσουν οι υπολογισμοί, το ζητούμενο κόστος δίνεται από το $C[1][n]$ που αφορά όλο το αρχικό string s. Η χρονική πολυπλοκότητα είναι $\boldsymbol{O(n)}$ για την for του length, $\boldsymbol{O(n)}$ για την for του i και στην χειρότερη θα εξετάσουμε $\boldsymbol{O(n)}$ σημεία κοπής. Συνολικά $\boldsymbol{O(n^3)}$.

---

[1] Το γιατί δεν θα δούλευε ένας μονοδιάστατος (1D) πίνακας και χρειαζόμαστε 2D, εξηγείται στην σελίδα 10.

[2] Τα C[i][k] και C[k+1][j] που απαιτούνται για τον υπολογισμό, έχουν μήκος k-i+1 και j-k αντίστοιχα και είναι μικρότερα του current length, άρα έχουν ήδη υπολογιστεί και είναι γνωστά.

Πρόβλημα 1.1

```
main:
{
    G = graph representing the road network
    s = starting city
    t = destination city
    L = maximum distance the car can travel with a full tank

    Remove_Unreachable_Edges(G, L)

    if BreadthFirstSearch(G, s, t):
        print "There is a path from city", s, "to city", t
        print("Path: ", printPath(G, s, t))
    else:
        print "There is no path from city", s, "to city", t
}
```

```
struct vertex:
{
    vertex π; // predecessor
    int key; // max edge length among all edges in path from s to city
}
```

```
Remove_Unreachable_Edges(G, L):
{
    for each vertex city in G:
        for each vertex neighbor in G.Adj[city]: // Once per edje - O(|E|)
            if weight(city, neighbor) > L:
                G.Adj[city].remove(neighbor)
}
```

```
BreadthFirstSearch(G, s, t):
{
    visited = array of |V| booleans // Denoting if city is visited
    queue = empty queue // Queue of cities to visit

    visited.add(s)
    s.π = NIL
    queue.push(s)

    while queue is not empty:
    {
        city = queue.pop()

        if city == t:
            return true  // City t is reachable from city s

        for each neighbor in G.Adj[city]:
            if neighbor not in visited:
                visited.add(neighbor)
                neighbor.π = city
                queue.push(neighbor)
    }

    return false  // City t is not reachable from city s
}
```

```
printPath(G, s, t):
{
    path = []
    city = t

    while city != NIL:
    {
        path.add(city)
        city = city.π
    }

    return path.reverse()
}
```

Πρόβλημα 1.2

```
main()
{
    G = graph representing the road network
    s = starting city
    t = destination city
    L = maximum distance the car can travel with a full tank

    Remove_Unreachable_Edges(G, L)

    max_edge_in_path = Dijkstra_Modified(G, weight, s, t) // Find the minimum - max edge length in path from s to t

    if max_edge_in_path == INF:
        print("City t is not reachable from city s")
    else:
        print("City t is reachable from city s with max edge length in path: ", max_edge_in_path)
        print("Path: ", printPath(G, s, t))
}
```

```
Dijkstra_Modified(G, weight, s, t):
{
    visited = array of |V| booleans // Denoting if city is visited
    Iniitialize(G, s, visited) // O(|V|)

    priority_queue = G.V // ordered by key of vertex, |V| insertions - O(|V|log|V|)

    while priority_queue is not empty:
    {
        city = priority_queue.extraxtMin() // extractMin operation, once per vertex - O(|V|log|V|)
        visited[city] = true

        if city == t: // the first path to reach t has the minimum max edge length
            return city.key  // City t is reachable from city s

        for each vertex neighbor in G.Adj[city]:
            if ( !visited[neighbor] ):
                relax( city, neighbor, weight(city, neighbor) ) // decreaseKey operation, max one per ejde - O(|E|log|V|)
    }

    return INF  // City t is not reachable from city s
}
```

```
Iniitialize(G, s, max_edge_in_path, visited):
{
    for each city in G:
        city.π = NIL
        city.key = INF
        visited[city] = false

    s.key = 0
}
```

```
relax(curr_city, neighbor, edge_length):
{
    if neighbor.key > max(curr_city.key, edge_length):
        neighbor.key = max(curr_city.key, edge_length) // decreaseKey operation - O(log|V|)
        neighbor.π = curr_city
}
```

Πρόβλημα 2

Απόδειξη (εις άτοπον απαγωγή) :

Έστω *S* η βέλτιστη σειρά εξυπηρέτησης των πολιτών η οποία ελαχιστοποιεί το $T_{ολ}$. Υποθέτουμε τώρα ότι η *S* δεν ταυτίζεται με την ταξινόμηση κατά αύξοντα χρόνο εξυπηρέτησης. Τότε θα υπάρχουν πολίτες *i*, *j* με $t_i < t_j$ και η σειρά εξυπηρέτησης του *j*, έστω x, προηγείται αυτής του *i*, έστω y, δηλαδή $x < y$. Ορίζουμε μια άλλη σειρά εξυπηρέτησης *S΄* η οποία είναι ίδια με την *S* με τη μόνη διαφορά ότι έχει γίνει swap στους πολίτες *i* και *j* και πλέον ο *i* εξυπηρετείται πριν τον *j*. Υπολογίζουμε τους συνολικούς χρόνους αναμονής :

$$T_{ολ,S} = (n-1)t_1 + \cdots + (n-x)t_j + \cdots + (n-y)t_i + \cdots + t_{n-1}$$

$$και$$

$$T_{ολ,S'} = (n-1)t_1 + \cdots + (n-x)t_i + \cdots + (n-y)t_j + \cdots + t_{n-1}$$

Εφόσον η S είναι η βέλτιστη σειρά εξυπηρέτησης, ισχύει :

$$T_{ολ,S} \leq T_{ολ,S'} \Leftrightarrow T_{ολ,S} - T_{ολ,S'} \leq 0 \overset{(1)}{\Leftrightarrow} (n-x)t_j + (n-y)t_i - (n-x)t_i - (n-y)t_j \leq 0 \Leftrightarrow$$

$$\Leftrightarrow xt_i + yt_j - xt_j - yt_i \leq 0 \Leftrightarrow t_j(y-x) - t_i(y-x) \leq 0 \Leftrightarrow (y-x)\left(t_j - t_i\right) \leq 0 \; και \; επειδή$$

$$(y-x) > 0 \, , τελικά \; έχουμε \left(t_j - t_i\right) \leq 0 \Leftrightarrow t_j \leq t_i \, , άτοπο$$

Επομένως, πράγματι η βέλτιστη σειρά εξυπηρέτησης ταυτίζεται με την σειρά ταξινόμησης των πολιτών κατά αύξουσα σειρά χρόνων εξυπηρέτησης όπως είχαμε προβλέψει εξ αρχής!

```
main():
{
    n = number of citizens
    service_time[n] = array of service time for each citizen

    mergeSort(service_time) // sort citizens on ascending order of service time

    waiting_time = 0; // waiting time of citizen i = sum of service time from 0 to i-1
    total_waiting_time = 0; // total waiting time of all citizens

    for (int i = 0; i < n; i++)
    {
        total_waiting_time += waiting_time; // add waiting time of citizen i to total waiting time
        waiting_time += service_time[i]; // add service time of citizen i to waiting time of all following citizens
    }

    print("Best order of service: ", service_time);
    print("Minimum total waiting time: ", total_waiting_time);
}
```

[1] Όλοι οι υπόλοιποι όροι ταυτίζονται και συνεπώς αλληλοαναιρούνται.

Πρόβλημα 3

```
main():
{
    n = length of the string
    m = number of cuts
    cut_points[] = array of cut points // cut between s[i] and s[i+1] is denoted as i

    print("Minimum cost to cut the string is: ", min_cut_cost(n, m, cut_points))
}
```

```
min_cut_cost(n, m, cut_points):
{
    C[n][n] // 2D array, where C[i][j] stores the minimum cost to cut the string s[i..j]

    for i = 1 to n:
        for j = 1 to n:
            C[i][j] = -1 // symbolizes that the value is undefined (eg if j < i)

        for length = 1 to n: // length of the string
            for i = 1 to n - (length-1): // starting index
            {
                j = i + length - 1 // ending index

                C[i][j] = INF

                for k in cut_points:
                    if i <= k < j: // cut point is within the string
                        C[i][j] = min(C[i][j], C[i][k] + C[k+1][j] + length)

                if(C[i][j] == INF) // no cut point within the string
                    C[i][j] = 0
            }

    return C[1][n] // minimum cost to cut the whole string - s[1..n]
}
```

Γιατί δεν θα δούλευε ένας μονοδιάστατος πίνακας ούτε εδώ αλλά ούτε και στο Chain Matrix Multiplication problem;

Ας υποθέσουμε ότι είχαμε προσπαθήσει να περιορίσουμε τον χώρο των υποπροβλημάτων μας για την διάσπαση του string s σε διάσπαση substring της μορφής (**1**, i). Όπως και πριν, μια βέλτιστη θέση τομής θα διαχώριζε το substring αυτό ανάμεσα στο (1, k) και στο (k+1, i) για κάποιο k τέτοιο ώστε $1 \leq k < i$. Επομένως θα διαπιστώναμε ότι προκύπτουν υποπροβλήματα που δεν είναι αποκλειστικά της μορφής (1, i), δηλαδή μονοδιάστατα (εκτός και αν το k ισούταν πάντα με i-1). Γι' αυτό λοιπόν, και στα δύο αυτά προβλήματα, ήμασταν υποχρεωμένοι να ορίσουμε υποπροβλήματα μεταβλητά και στα δύο άκρα, δηλαδή να θεωρήσουμε μεταβλητούς και τους δύο δείκτες i και j στο υποπρόβλημα (i, j)…

Παράδειγμα εκτέλεσης αλγορίθμου

Example

String : "12345678" || Cut points: 2, 5, 7

i/j	1	2	3	4	5	6	7	8
1	0	0	3	4	5	10	12	16
2		0	2	3	4	9	10	14
3			0	0	0	4	5	9
4				0	0	3	4	8
5					0	2	3	7
6						0	0	3
7							0	2
8								0

(τα κίτρινα υποπροβλήματα είναι αυτά που έχουν τουλάχιστον 2 σημεία τομής)