**Structuri de date și algoritmi**

Proiect 4: *TAD MultiDicționar (MultiMap) - implementare folosind listă simplu înlănțuită dinamic*

Nume: Bădiță Marin-Georgian

Grupa: 211

# **Enunțul problemei**

În capitala Azerbaijanului, sătul fiind de traficul infernal, primarul dorește să îmbunătățească rețeaua de transport. Având la dispoziție intersecțiile intre care există drumuri, precum și numărul total de vehicule care pot trece prin fiecare drum la un moment dat, primarul dorește să afle:

1. dacă toate mașinile care intră în oraș pot să și iasă
2. dacă rețeaua pe care o are la dispoziție este destul de eficientă astfel încât o mașină să poată parcurge toate drumurile din oraș trecând o singură dată prin fiecare.
3. capacitatea unui drum nou adăugat astfel încât toate mașinile care intră în oraș sa poată ieși
4. numărul minim de drumuri care trebuie adăugate astfel încât să existe un drum cu proprietatea de la b)

Rezolvarea problemei se reduce, de fapt, la interpretarea intersecțiilor ca noduri într-un graf și a drumurilor ca muchiile dintre acestea așa încât în rezolvarea problemei enunțate se vor folosi algoritmi de flux (Ford Fulkerson) și ciclu/lanț eulerian, algoritmi specifici grafurilor, care pot fi foarte bine reprezentați utilizând un MultiDicițonar, cheile acestuia făcând parte din mulțimea nodurilor, iar pentru fiecare cheie (nod) se pot reține vecinii acestuia, păstrând complexitățilie operațiilor cât mai bune specifice listelor înlănțuite.

# **Reprezentarea TAD-ului**

**TPereche: MD: TNod**

c: TCheie prim: ↑TNod e: TPereche

v: TValoare ultim: ↑TNod urm: ↑TNod

# **Specificarea și interfața TAD-ului**

Un MutiDicționar este un container asociativ care conține elemente formate din perechi de tip <TCheie, TValoare>, o cheie putând avea asociate una sau mai multe valori, înseamnă că este posibil să avem mai multe valori care sunt asociate la aceeași cheie, deci cheile nu sunt întotdeauna unice.

**Md = { md | md este un MultiDicționar cu elemente de tip (c, v) unde c ∈ TCheie si v ∈ TValoare}**

**creeaza(md):**

descriere: creează un nou MultiDicționar

pre: True

post: md ∈ MD, (md e un MultiDicționar vid)

**distruge(md):**

descriere: eliberează spațiul alocat pentru md

pre: md ∈ MD

post: md a fost “distrus” (spațiul a fost eliberat)

**adauga(md, c, v):**

descriere: adaugă la md perechia (c, v)

pre: md ∈ MD, c ∈ TCheie, v ∈ TValoare

post: perechea (c, v) a fost adăugată în md

**sterge(md, c, v):**

descriere: șterge din md perechea (c, v)

pre: md ∈ MD, c ∈ TCheie, v ∈ TValoare

post: perechea (c, v) a fost ștearsă din md

**cauta(md, c, l):**

descriere: caută o cheie in MultiDicționar

pre: md ∈ MD, c ∈ TCheie

post: l ∈ L (listă),



true si l conține lista valorilor asociate cu c

cauta ← dacă c este cheie în md

fals și l este lista vidă, altfel

**iterator(md, i):**

descriere: creează un iterator pe md

pre: md ∈ MD

post: i ∈ I, i este iterator pe md

**dim(md):**

descriere: calculeaza dimensiunea MultiDicționarului

pre: md ∈ MD

post: dim ← dimensiunea MultiDicționarului md (numărul de elemente) - număr natural

**vid(md):**

descriere: verifica daca md este MultiDicționarul vid

pre: md ∈ MD

post: vid ← true, dacă md este MultiDicționarul vid

false, altfel

**Iterator Multidicționar**

Un iterator este o structură folosită pentru a parcurge elementele unui container, în cazul nostru elementele unui MultiDicționar

**I = { i | i este un iterator pe multidicționar cu elemente de forma perechi (c, v), cu c ∈** **TCheie si v ∈ TValoare** **}**

**IteratorMultiDicționar - reprezentare:**

**multiDictionar:** MD (MultiDicționar)

**curent:** ↑TNod

**IteratorMultiDicționar - interfață:**

**creeaza(i, md):**

descriere: creează un iterator pe md

pre: md ∈ MD

post: i ∈ I (se creează un iterator pe md. Elementul curent din iterator referă “prima” pereche din MultiDicționar)

**element(i, e):**

descriere: extrage în e perechea referită de curent

pre: i ∈ I, i este valid

post: e ∈ TValoare, e este perechea curentă din iterație (perechea din MultiDicționar referită de curent)

**valid(i):**

descriere: verifică dacă iteratorul este valid

pre: i ∈ I

post: valid ← true, dacă iteratorul referă către un element valid

false, altfel

**urmator(i):**

descriere: trece la perechea următore din iterație

pre: i ∈ I, i este valid

post: curent’ referă ‘următoarea’ pereche din MultiDicționar față de cea referită de curent

# **Interfețe**

Interfață TPereche:

creeaza(p, TCheie, TValoare)

operator=(pereche, pereche1)

operator!=(pereche, perche1)

Interfață TNod:

creeaza(nod, pereche)

Interfață TVector:

creeaza(v)

distruge(v)

push\_bac9k(v, elem)

operator[](v, i)

size(v)

empty(v)

erase(v, it)

begin(v)

end(v)

Interfață MD:

creeaza(md)

adauga(md, c, v)

sterge(md, c, v)

cauta(md, c, l)

dim(md)

vid(md)

iterator(md, i)

Pentru realizarea problemei au mai fost nevoie de urmatoarele:

**Graf:**

flux: Intreg[][]

viz: boolean[]

parent: Intreg[]

num\_intersectii: intreg[]

graph: MD

Functii:

creeaza(gr, inter)

adauga\_muchie(gr, x, y, capacitate)

sterge\_muchie(gr, x, y)

vecini(x)

flux\_maxim(gr, sursa, destinatie)

ciclu\_euler()

capacitate\_drum(gr, sursa, destinatie, flux)

muchii\_euler()

set\_num\_inter(num\_intersectii)

**Muchie:**

x: Intreg

y: intreg

Functii:

creeaza(mc, x, y)

**Node:**

value: TElement

next: ↑Node

Functii:

creeaza(node, val)

**IteratorVector:**

iter\_node: ↑Node

Functii:

creeazaIteratorVector(it, node)

operator=(it1, it2)

operator!=(it1, it2)

urmator(it)

valid(it)

element(it)

**TVector:**

head: ↑Node

Functii:

creeaza(v)

push\_back(v, elem)

operator[](i)

size(v)

empty(v)

begin(v)

end(v)

**ConsoleUI:**

graf: Graf

numar\_masaini: Intreg

Functii:

creeazaUI(ui, num\_inters, num\_mas)

print\_menu(ui)

show\_ui(ui)

adauga\_intersectie(ui)

sterge\_intersectie(ui)

testeaza\_retea\_flux(ui)

testeaza\_retea\_euler(ui)

capacitate\_drum(ui)

drumuri\_noi(ui)

afiseaza\_drumuri(ui)

# **Implementare**

1. **TPereche:**

**subalgoritm creeaza(p, c, v):**

**complexitate: θ(1)**

**pre: c ∈ TCheie, v ∈ TValoare**

**post: p ∈ TPereche, se creeaza perechea cu p.c = c si p.v = v**

p.c ←c

p.v ←v

**sfarsit subalgoritm**

**functie operator=(pereche, pereche1):**

**complexitate: θ(1)**

**pre: pereche ∈ TPereche, pereche1 ∈ TPereche**

**post: operator= ← true, daca pereche.c = pereche1.c si pereche.v = pereche1.v, false altfel**

daca pereche.c = pereche1.c si pereche.v = pereche1.v atunci

operator= ←true

alfel

operator= ←false

**sfarsit functie**

**functie operator!=(pereche, perche1):**

**complexitate: θ(1)**

**pre: pereche ∈ TPereche, pereche1 ∈ TPereche**

**post: operator!= ← true, daca pereche.c != pereche1.c sau pereche.v != pereche1.v, false altfel**

operator!= ←!operator=(pereche, pereche1)

**sfarsit functie**

1. **TNod:**

**subalgoritm creeaza(nod, pereche):**

**complexitate: θ(1)**

**pre: pereche ∈ TPereche**

**post: nod ∈ TNod, creeaza un nod care are ca si informatie utila pereche**

nod.e ←pereche

nod.urm ←NIL

**sfarsit subalgoritm**

**III. TVector**

**subalgoritm creeaza(v):**

**complexitate: θ(1)**

**pre: true**

**post: v ∈ TVector, v este vecorul vid**

v.head ← NIL

**sfarsit subalgoritm**

**subalgoritm distruge(v):**

**complexitate: θ(n)**

**pre: v ∈ TVector**

**post: vectorul v a fost ‘distrus’ (spatiul de memorie alocat a fost eliberat)**

start ←v.head

cat timp start != NIL

next = [start].next

@dealoca start

start = next

sfarsit cat timp

**sfarsit subalgoritm**

**subalgoritm push\_back(v, elem):**

**complexitate: θ(n)**

**pre: v ∈ TVector, elem ∈ TElem**

**post: v’ ∈ TVector, v’ = v + elem**

@aloca node

creeaza(node, elem)

daca v.head = NIL atunci

v.head ←node

altfel

search ←v.head

cat timp [search].next != NIL executa

search ←[search].next

sfarsit cat timp

[search].next = node

sfarsit daca

**sfarsit subalgoritm**

**functie operator[](v, i):**

**Complexitate: θ(n)**

**pre: i ∈ Z(numere intregi), v ∈ TVector**

**post: operator[] ←elementul de pe pozitia i**

n ←0

count ←v.head

cat timp n < i && count != NIL execcuta

count ←[count].next

n ←n + 1

sfarsit cat timp

operator[] ←[count].value

**sfarsit functie**

**functie size(v):**

**complexitate: θ(n)**

**pre: v ∈ TVector**

**post: size ←numarul de elemente din v**

daca v.head = NIL atunci

suze ←0

sfarsit daca

n ←1

node ←v.head

cat timp [node].next != NIL executa

n ←n + 1

node ←[node].next

sfarsit cat timp

size ←n

**sfarsit functie**

**functie empty(v):**

**complexitate: θ(n)**

**pre: v ∈ TVector**

**post: empty ←true daca v este vectorul gol, false altfel**

val ←v.size()

daca val = 0 atunci

empty ←true

altfel

empty ←false

sfarsit daca

**sfarsit functie**

**functie begin(v):**

**complexitate: θ(1)**

**pre: v ∈ TVector**

**post: se returneaza un IteratorVector, la carui curent este setat catre primul element din v**

begin ←creeazaIteratorVector(v.head)

**sfarsit functie**

**functie end(v):**

**complexitate: θ(1)**

**pre: v ∈ TVector**

**post: se returneaza un IteratorVector al carui curent refera catre primul element dupa sfarsitul vectorului**

end ←creeazaIteratorVector(NIL)

**sfarsit functie**

**subalgoritm erase(v, it):**

**complexitate: O(n)**

**pre: v ∈ TVector, it ∈ IteratorVector**

**post: elementul referit de catre it va fi sters din vector**

to\_del ←element(it)

daca !it.valid() atunci

**sfarsit subalgoritm**

altfel

node = v.head

daca to\_del = [node].value atunci

remind ←v.head

v.head ←[v.head].next

@dealoca remind

**sfarsit subalgoritm**

sfarsit daca

cat timp [[node].next].value != to\_del executa

node ←[node].next

sfarsit cat timp

remind ←[node].next

[node].next = [[node].next].next

@dealoca remind

sfarsit daca

**sfarsit subalgoritm**

**IV. IteratorVector**

**subalgoritm creeaza(it, node):**

**complexitate: θ(1)**

**pre: it ∈ IteratorVector, node ∈ Node**

**post: it ∈ IteratorVector, iar it refera catre node**

iter\_node ← node

**sf subalgoritm**

**functie operator=(it1, it2):**

**complexitate: θ(1)**

**pre: it1 ∈ IteratorVector, it2 ∈ IteratorVector**

**post: operator= ←true, daca it1 = it2, false altfel**

operator= ←it1 = it2

**sfarsit functie**

**functie operator!=(it1, it2):**

**complexitate: θ(1)**

**pre: it1 ∈ IteratorVector, it2 ∈ IteratorVector**

**post: operator != ←true, daca it1 != it2, fals altfel**

operator!= ←it1 != it2

**sfarsit functie**

**functie urmator(it):**

**complexitate: θ(1)**

**pre: it ∈ IteratorVector**

**post: curent’ refera la elementul urmator dupa curent in vector, returneaza iteratorl al carui curent refera catre urmatorul element fata de curent**

daca it.iter\_node != NIL atunci

iter\_node ←[iter\_node].next

sfarsit daca

urmator ←it

**sfarsit functie**

**functie valid(it):**

**complexitate: θ(1)**

**pre: it ∈ IteratorVector**

**post: valid ←true, daca iteratorul este valid, fals altfel**

daca it.inter\_node = NIL atunci

valid ←false

altfel

valid ←true

sfarsit daca

**sfarsit functie**

**functie element(it):**

**complexitate: θ(1)**

**pre: it ∈ IteratorVector**

**post: element ←valoare stocata in e elementul catre care efera it**

element ←[it.iter\_node].value

**sfarsit functie**

**V. MD**

**subalgoritm creeaza(md):**

**complexitate: θ(1)**

**descriere: creează un nou MultiDicționar**

**pre: True**

**post: md ∈ MD, (md e un MultiDicționar vid)**

md.prim ←NIL

md.ultim ←NIL

**sfarsit subalgoritm**

**distruge(md):**

**complexitate: θ(n)**

**descriere: eliberează spațiul alocat pentru md**

**pre: md ∈ MD**

**post: md a fost “distrus” (spațiul a fost eliberat)**

start ←md.prim

cat timp start != NIL executa

urmator ←[start].urm

@dealoca start

start ←urmator

sfarsit cat timp

**sfarsit subalgoritm**

**subalgoritm adauga(md, c, v):**

**complexitate: O(n)**

**descriere: adaugă la md perechea (c, v)**

**pre: md ∈ MD, c ∈ TCheie, v ∈ TValoare**

**post: perechea (c, v) a fost adăugată în md**

@aloca node

creeaza(p, c, v)

creeaza(node, p)

daca md.prim = NIL atunci

[node].urm = md.prim

md.ultim ←node

md.prim ←md.ultim

[md.ultim].urm ←NIL

altfel

start ←md.prim

cat timp start != NIL && [start].e.c != [node].e.c executa

start ←[start].urm

sfarsit cat timp

daca start = NIL

[node].urm ←md.prim

md.prim ←node

altfel

[node].urm = [start].urm

[start].urm = node

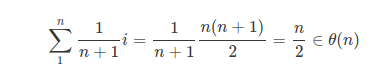
sfarsit daca

sfarsit daca

**sfarsit subalgoritm**

**subalgoritm sterge(md, c, v):**

**complexitate: Best case -θ(1) - cand elementul de sters este primul element din md**

**Worst case - θ(n) - cand elementul de sters nu exista in md**

**Average**

**complexitate overall: O(n)**

**descriere: șterge din md perechea (c, v)**

**pre: md ∈ MD, c ∈ TCheie, v ∈ TValoare**

**post: perechea (c, v) a fost ștearsă din md**

daca md.prim = NIL atunci

**sfarsit subalgoritm**

sfarsit daca

creeaza(p, c, v)

daca [md.prim].e = p atunci

de\_sters ←md.prim

md.prim ←[md.prim].urm

@dealoca de\_sters

**sfarsit subalgoritm**

sfarsit daca

start ←md.prim

cat timp [[start].urm].e != node executa

daca [start].urm = NIL atunci

@termina executia cat timp

sfarsit daca

start ←[start].urm

sfarsit cat timp

daca [start].urm = NIL atunci

**sfarsit subalgoritm**

sfarsit daca

de\_sters ←[start].urm

[start].urm ←[[start].urm].urm

@dealoca de\_sters

**sfarsit subalgoritm**

**functie cauta(md, c):**

**complexitate: θ(n)**

**descriere: caută o cheie in MultiDicționar**

**pre: md ∈ MD, c ∈ TCheie**

**post: cauta ←lista valorilor asociate cheii c dac exista, lista vida altfel**

creeaza(rez)

daca md.prim = NIL atunci

cauta ←rez

altfel

start ←md.prim

cat timp start != NIL executa

daca [start].e.c = c atunci

rez.push\_back([start].e.v);

sfarsit daca

start = [start].urm;

sfarsit cat timp

sfarsit daca

cauta ←rez

**sfarsit functie**

**subalgoritm iterator(md, i):**

**complexitate: θ(1)**

**descriere: creează un iterator pe md**

**pre: md ∈ MD**

**post: i ∈ I, i este iterator pe md**

i ←creeazaIteratorMD(md)

**sfarist functie**

**functie dim(md):**

**complexitate: θ(n)**

**descriere: calculeaza dimensiunea MultiDicționarului**

**pre: md ∈ MD**

**post: dim ← dimensiunea MultiDicționarului md (numărul de elemente) - număr natural**

daca md.prim = NIL atunci

dim ←0

sfarsit daca

lungime ←1

nod ←md.prim

cat timp [nod].urm != NIL executa

lungime ←lungime + 1

nod ←md.prim

sfarsit cat timp

dim ←lungime

**sfarsit functie**

**functie vid(md):**

**complexitate: θ(n)**

**descriere: verifica daca md este MultiDicționarul vid**

**pre: md ∈ MD**

**post: vid ← true, dacă md este MultiDicționarul vid**

**false, altfel**

val ←md.dim()

daca val = 0 atunci

vid ←true

altfel

vid ←false

**sfarsit functie**

**VI. Graf**

**subalgoritm creeazaGraf(gr, inter):**

**complexitate: θ(MAX\_INTERS^2)**

**pre: inter ∈ Z(numar intreg)**

**post: gr ∈ Graf, numarul de intersectii se actualizeaza, iar fluxul se initializeaza cu 0**

gr.num\_intersectii ←inter

pentru i←1,MAX\_INTERS executa

pentru j ←1, MAX\_INTERS executa

gr.flux[i][j] = 0

sfarsit pentru

sfarsit pentru

**sfarsit subalgroitm**

**functie bfs(gr, s, t):**

**complexitate: θ(n + m), n - nr intersectii, m - nr drumuri**

**pre: gr ∈ Graf, s ∈ Z(numar intreg), t ∈ Z (numar intreg)**

**post: bfs ←true daca se gaseste drum de la s la t in graf, fals altfel**

@seteaza elementele lui gr.viz la false

@seteaza elementele lui gr.parent la 0

gr.viz[s] = true

g.parent[s] = -1

q: int[]

prim ←0

ultim ←0

q[ultim] ←0

cat timp prim <= ultim executa

node ←q[prim]

prim ←prim + 1

daca nod ← t atunci

@sari peste iteratie

sfarsit daca

pentru adj in ←gr.graph[node] executa

daca !gr.viz[adj] && gr.flux[node][adj] > 0 atunci

ultim ←ultim + 1

q[ultim] ←adj

gr.viz[adj] = true

gr.parent[adj] = node

sfarsit daca

sfarsit pentru

sfarsit cat timp

bfs ←viz[t]

**sfarsit functie**

**subalgoritm adauga\_muchie(gr, x, y, cost):**

**complexitate: θ(n)**

**pre: x ∈ Z, y ∈ Z, cost ∈ Z, gr ∈ Graf**

**post: muchia de la x la y de cost cost va fi adaugata in graf**

adauga(gr.graph, x, y, cost)

adauga(gr.graph, y, x, cost)

gr.flux[x][y] ←gr.flux[x][y] + cost

**sfarsit subalgoritm**

**subalgoritm sterge\_muchie(gr, x, y, cost):**

**complexitate: θ(n)**

**pre: gr ∈ Graf, x ∈ Z, y ∈ Z, cost ∈ Z**

**post: muchia de la x la y va fi stearsa, daca exista**

sterge(gr, x, y)

**sfarsit subalgoritm**

**functie vecini(gr, x):**

**complexitate: O(n)**

**pre: x ∈ Z, gr ∈ Graf**

**post: vecini ←lista reprezentand vecinii nodului x**

vecini ←gr.graph[x]

**sfarsit functie**

**functie flux\_maxim(gr, sursa, destinatie):**

**complexitate: O(n\*m^2), n - nr de intersectii, m - nr de drumuri**

**pre: gr ∈ Graf, sursa ∈ Z, destinatie ∈ Z**

**post: flux\_maxim ←valoarea fluxului maxim ce poate fi trimis de la sursa la destinatie**

max\_flow ←0

min\_cap ←0

cat timp bfs(gr, sursa, destinatie) = true executa

pentru curr\_node in : gr.graph[destinatie] executa

daca !gr.viz[curr\_node] || gr.flux[curr\_node][destinatie] <= 0 atunci

@sari peste iteratie

sfarsit daca

gr.parent[destinatie] = curr\_node

min\_cap ←INF

pentru i ←destinatie, i != sursa, i = gr.parent[i] executa

min\_cap = min(min\_cap, flow[gr.parent[i]][i])

sfarsit pentru

pentru i ←destinatie, i != sursa, i = gr.parent[i] executa

gr.flux[gr.parent[i]][i] ←gr.flux[gr.parent[i]][i] - min\_cap

gr.flux[i][gr.parent[i]] ←gr.flux[i][gr.parent[i]] + min\_cap

sfarsit pentru

max\_flow ←max\_flow + min\_cap

sfarsit pentru

sfarsit cat timp

flux\_maxim ←max\_flow

**sfarsit functie**

**functie verifica\_euler(gr):**

**complexitate: O(n^2)**

**pre: gr ∈ Graf**

**post: verifica\_eule ←numar ∈ Z > 0 daca graful are ciclu/lant eulerian, -1 altfel**

nr\_impar ←0

start\_nod ←1

pentru i ←1, gr.num\_intersectii executa

daca size(gr.graph[i]) % 2 != 0 atunci

nr\_impar ←nr\_impar + 1

start\_nod ←i

sfarsit daca

sfarsit pentru

daca nr\_impar > 2 atunci

verifica\_euler() ←-1

altfel

verifica\_euler() ←start\_nod

sfarsit daca

**sfarsit functie**

**functie ciclu\_euler(gr)**

**complexitate: O(n^2 + n\*m)**

**pre: gr ∈ Graf**

**post: ciclu\_eule ←ciclul /lantul eulerian daca exista, lista vida altfel**

creeaza(res)

start\_nod ←gr.verifica\_euler()

daca start\_nod = -1 atunci

ciclu\_euler ←res

altfel

st: int[]

st[st[0]] ←st[st[0]] + 1

st[st[0]] ←start\_nod

cat timp st[st[0]] > 0 executa

nod = st[st[0]]

daca !empty(gr.graph[nod]) atunci

nod\_nou ←element(gr.graph[nod].begin())

sterge(gr.graph, nod, nod\_nou)

sterge(gr.graph, nod\_nou, nod)

st[0] ←st[0] + 1

st[st[0]] ←nod\_nou

altfel

st[0] ←st[0] - 1

psuh\_back(res, nod)

sfarsit daca

ciclu\_euler←res

**sfarsit functie**

**functie capacitate\_drum(gr, sursa, destinatie, flux)**

**complexitate: O(n\*m^2)**

**pre: gr ∈ Graf, sursa ∈ Z, destinatie ∈ Z, flux ∈ Z**

**post: capacitate\_drum ←capacitatea unui dum nou adaugat, sau 0 daca nu trebuie adaugat niciun drum**

diff ←flux\_maxim(gr, sursa, destinatie) - flux

daca diff < 0 atunci

capacitate\_drum ←diff

altfel

capacitate\_drum ←0

**sfarsit functie**

**functie muchii\_euler(gr):**

**complexitate: θ(n^2)**

**pre: gr ∈ Graf**

**post: muchii\_euler ←o lista cu muchiile ce trebuie adaugate in graf astfel incat sa existe cilcu/lant eulerian, sau lista vida altfel**

creeaza(impare)

creeaza(res)

pentru i ←1, gr.num\_intersectii executa

daca size(gr.graph[i]) % 2 = 1 atunci

push\_back(impare, i)

sfarsit daca

sfarsit pentru

pentru i ←0, [size(impare) / 2] executa

creeaza(mc, impare[i], impare[size(impare) - i - 1])

push\_back(res, mc)

sfarsit pentru

muchii\_euler ←res

**sfarsit functie**

**subalgoritm set\_num\_inter(gr, num\_intersectii):**

**complexitate: θ(1)**

**pre: num\_intersectii ∈ Z, gr ∈ Graf**

**post: numarul de intersectii va fi actualiat la num\_intersectii**

gr.num\_intersectii ←num\_intersectii

**sfarsit subalgoritm**

**VII. ConsoleUI**

**subalgoritm creeazaUI(ui, num\_inters, num\_mas):**

**complexitate: θ(MAX\_INTERS^2)**

**pre: ui ∈ ConsoleUI, num\_inters ∈ Z, num\_mas ∈ Z**

**post: se creeaza un obiect de tip ConsoleUI**

set\_num\_inter(graf, num\_inters)

ui.numar\_masini ←num\_mas

**sfarsit subalgoritm**

**subalgoritm print\_menu(ui):**

**complexitate: θ(1)**

**pre: ui ∈ ConsoleUI**

**post: se printeaza pe ecran meniul de optiuni**

meniu: char[]

meniu ← "\n\n1. Pentru a adauga un drum nou\n"

"2. Pentru a sterge un drum\n"

"3. Pentru a verifica daca reteaua poate acomoda toate masinile\n"

"4. Pentru a verifica daca exista un ciclu eulerian in reteaua de transport\n"

"5. Pentru a afla capacitatea necesara unui nou drum pentru a acomoda toate masinile\n"

"6. Pentru a afla intersectiile ce trebuiesc legate astfel incat sa existe un ciclu eulerian in retea\n"

"7. Pentru a afisa toate intersectiile\n"

"8. Pentru a iesi din aplicatie\n"

"\tDati va rog o comanda: "

@afiseaza meniu la ecran

**sfarsit subalgoritm**

**subalgoritm show\_ui(ui):**

**pre: ui ∈ ConsoleUI**

**post: true**

cat timp true executa

print\_menu(ui)

@citeste cmd de la tastatura

daca cmd = 1 atunci

adauga\_intersectie(ui)

altfel daca cmd = 2 atunci

sterge\_intersectie(ui)

altfel daca cmd = 3 atunci

testeaza\_retea\_flux(ui)

altfel daca cmd = 4 atunci

testeaza\_retea\_euler(ui)

altfel daca cmd = 5 atunci

capacitate\_drum(euler)

altfel daca cmd = 6 atunci

drumuri\_noi(ui)

altfel daca cmd = 7 atunci

afiseaza\_drumuri(ui)

altfel daca cmd = 8 atunci

@incheie executia ciclului cat timp

altfel

@printeaza mesaj de eroare

sfarsit daca

sfarsit cat timp

**sfarsit subalgoritm**

**subalgoritm adauga\_intersectie(ui):**

**complexitate: θ(n)**

**pre: ui ∈ ConsoleUI**

**post: o noua intersectie va fi adaugata**

@printeaza mesaj de citire

@citeste x de la tastatura

@printeaza mesaj de citire

@citeste y de la tastatura

@printeaza mesaj de citire

@citeste cost de la tastatura

adauga\_muchie(ui.graf, x, y, cost)

**sfarsit subalgoritm**

**subalgoritm sterge\_intersectie(ui)**

**complexitate: θ(n)**

**pre: ui ∈ ConsoleUI**

**post: se va sterge o muchie existenta din graf**

@printeaza mesaj de citire

@citeste x de la tastatura

@printeaza mesaj de citire

@citeste y de la tastatura

sterge\_muchie(ui.graf, x, y)

sterge\_muchie(ui.graf, y, x)

**sfarsit subalgoritm**

**subalgoritm testeaza\_retea\_flux(ui)**

**complexitate: O(n\*m^2)**

**pre: ui ∈ ConsoleUI**

**post: testeaza daca reteaua de transport poate acomoda numarul de masini**

@printeaza mesaj de citire

@citeste x de la tastatura

@printeaza mesaj de citire

@citeste y de la tastatura

rasp ←flux\_maxim(ui.graf, x, y)

@ printeaza numarul de masini ce pot ajunge din x in y

daca rasp >= ui.numar\_masini atunci

@printeaza mesaj afirmativ, toate masinile pot ajunge din x in y

altfe

@printeaza mesaj negativ, nu toate masinile pot ajunge din x in y

sfarsit daca

**sfarsit subalgoritm**

**subalgoritm testeaza\_retea\_euler(ui):**

**complexitate: O(n^2 + n\*m)**

**pre: ui ∈ ConsoleUI**

**post: testeaza daca reteaua are un ciclu/lant eulerian**

rez ←ciclu\_euler(ui.graf)

daca rez.empty() = true atunci

@printeaza mesaj negativ, nu exista ciclu/lant eulerian

altfel

@printeaza mesaj afirmativ, exista ciclu/lant eulerian

pentru elem in rez executa

@printeaza elem

sfarsit pentru

sfarsit daca

**sfarsit subalgoritm**

**subalgoritm capacitate\_drum(ui):**

**complexitate: O(n\*m^2)**

**pre: ui ∈ ConsoleUI**

**post: se vor printa drumurile ce trebuie adaugate astfel incat sa existe un lant/ciclu eulerian in retea**

@printeaza mesaj de citire

@citeste x de la tastatura

@printeaza mesaj de citire

@citeste y de la tastatura

capacitate ←capacitate\_drum(ui.graf, x, y, ui.numar\_masini)

daca capacitate = 0 atunci

@printeaza mesaj afirmativ, nu mai trebuie drumuri adaugate

altfel

@printeaza mesaj negativ si capacitatea drumului nou adaugat

sfarsit daca

**sfarsit subalgoritm**

**subalgoritm drumuri\_noi(ui):**

**complexitate: θ(n^2)**

**pre: ui ∈ ConsoleUi**

**post: se afiseaza drumurile ce trebuie adaugate astfel incat graful sa fie contina un lant/ciclu eulerian**

rez ←muchii\_euler(ui.graf)

daca rez.empty() = true atunci

@printeaza mesaj afirmativ, nu mai trebuie drumuri adaugate

altfel

@printeaza mesaj negativ, mai trebuie drumuri adaugate

pentru muc in rez executa

@printeaza muchia din rezultat

sfarsit pentru

sfarsit daca

**sfarsit subalgoritm**

**subalgoritm afiseaza\_drumuri(ui)**

**complexitate: θ(n)**

**pre: ui ∈ ConsoleUi**

**post: se afiseza toate muchiile grafului utilizand interatorul**

iterator(ui.graf.graph, it)

cat timp valid(it) executa

element(it, e)

@printeaza e

urmator(it)

sfarsit cat timp

**sfarsit subalgoritm**

**VIII. Muchie**

**subalgoritm creeaza(mc, x, y):**

**complexitate: θ(1)**

**pre: mc ∈ Muhcie, x ∈ Z, y ∈ Z**

**post: creeza o muchie ce contine x, y ca si extremitati**

mc.x ←x

mc.y ←y

**sfarsit subalgoritm**

**IX. IteratorMD**

**subalgoritm creeazaIteratorMd(it, md):**

**complexitate: θ(1)**

**descriere: creeaza un iterator pe md**

**pre: md ∈ MD**

**post: i ∈ I (se creeaza un iterator pe md. Elementul curent din iterator refera “prima” pereche din MultiDictionar)**

it.md ←md

curent ←md.prim

**sfarsit subalgoritm**

**functie element(it, e)**

**complexitate: θ(1)**

**descriere: extrage in e perechea referita de curent**

**pre: i ∈ I, i este valid**

**post: e ∈ TValoare, e este perechea curenta din iteratie**

e ←[it.curent].e

**sfarsit functie**

**functie valid(it)**

**complexitate: θ(1)**

**descriere: verifica daca iteratorul este valid**

**pre: i ∈ I**

**post: valid ←true, daca iteratorul refera catre un element valid, false altfel**

val ←it.curent

daca val == NIL atunci

valid ←false

altfel

valid ←true

**sfarsit functie**

**subalgoritm urmator(it)**

**complexitate: θ(1)**

**descriere: trece la urmatoarea pereche din iteratie**

**pre: i ∈ I**

**post: curent’ refera ‘urmatoarea’ pereche din MultiDictionar fata de cea referita de curent**

it.curent ←[it.curent].urm

**sfarsit subalgoritm**

**X. Main**

**functie main()**

**pre: true**

**post: main ←0**

@printeaza mesaj de citire

**@**citeste num\_inter

**@**printeaza mesaj de citire

**@**citeste num\_masini

creeaza(ui, num\_inter, num\_masini)

show\_ui(ui)

**main ←0**

**sfarsit functie**

