VILNIAUS UNIVERSITETAS

MATEMATIKOS IR INFORMATIKOS FAKULTETAS

DUOMENŲ MOKSLAS

Algoritmų teorija

Projektinis darbas

MAKSIMALUS PORAVIMAS GRAFE

Anton Cifirov, Duomenų mokslas 2 grupė

Turinys

Įvadas

Šis darbas yra skirtas susipažinti su Edmondso maksimalaus poravimu neorientuotame grafe algoritmų, bei parašyti programą įgyvendinant šių algoritmų veikimo principą operuojant kombinatoriniais objektais bei išanalizuoti jų veikimą ir įvertinti sudėtingumą. Algoritmas realizuotas kalba c++.

1. Uždavinys

1.1.Uždavinio formuluotė

Maksimalaus poravimo neorientuotame grafe uždavinys

• Duota: Neorientuotas svorinis grafas G, turintis n viršūnių ir m briaunų.

• Rasti: Maksimalų grafo G viršūnių poravimą, t.y. maksimalų briaunų aibės E poaibį E', kurio visos briaunos nėra incidentinės

. 1.2.Uždavinio pavyzdys

• Duota: Neorientuotas grafas G:

Viršūnės: Viršūnės kurios yra prijungtos.

1 ----> 2

2 ----> 3, 6, 8

3 ----> 2, 4, 7

4 ----> 3, 5, 8

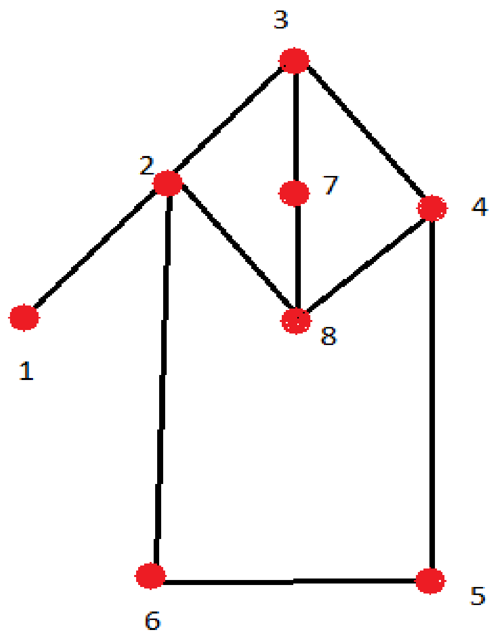
5 ----> 4, 6

6 ----> 2, 5

7 ----> 3, 8

8 ----> 2, 4, 7

Rasti: Grafo G (V,E) maksimalu poravimą M.(Pav 1)



Pav 1

Atsakymas: Maksimalus poravimas M(Pav 2):

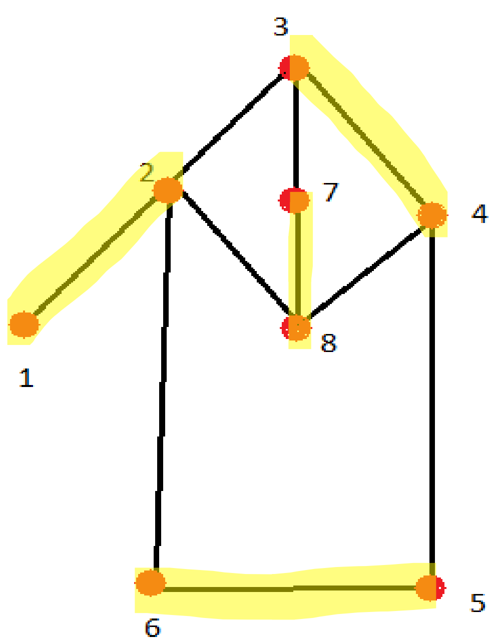
Briaunos:

8 - 7

3 - 4

5 - 6

2 - 1



Pav 2: Grafo G briaunos itrauktos į poravima M pažymėtos geltona spalva.

2 Edmondso Algoritmas.

2.1 Algoritmo aprašymas.

Edmondso algoritmas randa maksimalu duoto neorientuoto grafo G(V,E) poravimą M. Tai yra maksimalaus didžio E poaibį E‘ tokį kad visos briaunos jame nėra incidentinės.

Įvedame apibrėžimus:

- Viršūnė *v* vadinama **neuždengta** poravimu(angl. *covered vertex v*)M, jei M nėra briaunos, kuri yra incidentinė *v*.
- Kelias G yra **kintamasis kelias**(angl. *alternating walk*), jei jo kraštai pakaitomis nepriklauso M ir priklauso M.
- **Didinamasis kelias** P(angl. *augmenting path P*) yra kintamasis kelias, kuris prasideda ir baigiasi neuždengtomis viršūnėmis.
- **Žiedas** B(angl. *blossom*) yra nelyginis ciklas grafe G turintis 2k+1 briaunų lygiai k iš kuriu priklauso poravimui ir turinčia bazė – viršūnė *v*, tokia kad egzistuoja kintamasis kelias iš *v* į neuždengta poravimu viršūnė *w*.

Svarbi Edmondso algoritmui yra ir *Berge Lemma* teigianti jog poravimas M yra maksimalus tada ir tik tada, kai neegzistuoja M-didinančio kelio grafe G. Taigi arba poravimas yra maksimalus, arba ji galima padidinti.

Suformuluokime algoritmą pseudokodu:

Algoritmo Įvestis : Grafas G(V,E) ir poravimas M.

Algoritmo Išvestis: Maksimalus poravimas M‘ grafe G(V,E)

function *find_maximum_matching*(*G*, *M*)

P == *find_augmenting_path(G, M)*

If P is not empty **then**

return *find_maximum_matching*(*G*, augment *M* along *P*)

else

return M

end if

end function

Taigi pilno algoritmo realizavimo raktas yra radimas didinančiųjų keliu(augmenting path)

Galima butu bandyti spręsti ši uždavinį naudojant kintamųjų keliu radimą leidžiant BFS(Breadth First Search), bet tada musu kelias gali nueiti į ciklą kuris lengvai nepašalinamas. Naudojant Edmondso gėlių susitraukimo algoritmą, tai yra pakeičiant visus tokius nelyginius ciklus viena viršūnė.

Trakim X ir Y yra aibes. Tada susitraukimo operacija X/Y apibrėžiam taip:

$$X/Y := X \text{ if } X \cap Y = \emptyset,$$
$$X/Y := (X \setminus Y) \cup \{Y\} \text{ if } X \cap Y \neq \emptyset.$$

Todėl jei G (V, E) yra grafas ir C ⊆ V, tada V/C gaunamas nutrinant visas viršūnės iš C, ir įtraukiant viena nauja viršūnė C. Tada kiekvienai briaunai e iš E, e/C = e jei e neincidentinė C, ir e/C = uC jei e = uv kur u ne iš C, v iš C. Tada visiems F ⊆ E:

$$F/C := \{f/C \mid f \in F\}. \tag{11}$$

Taigi G/C   (V/C, E/C) vėl yra grafas. Sakome kad G/C gauname sutraukdami C grafe G.

Teorema: *Tegu C yra M-žiedas iš G. Tada M yra maksimumas grafe G tada ir tik tada, kai M/C yra maksimumas grafe G/C*

Dabar turime įrankius funkcijos algoritmo pseudokodui sukurti:

Duota: grafas G(V, E) ir jo pradinis poravimas M;

Gražina: maksimalu poravimą N grafe G;

```
WHILE F != Ø DO   //(*nustatom laisvas viršunės F*)
    pick r ∈ F
    queue.push(r)          //(*Paiškos gilyn eile*)
    T ← Ø                  // (*Paiškos gilyn medis*)
    T.add(r)
    WHILE queue != Ø
        v ← queue.pop()
        FOR ALL neighbors w of v DO
            IF w ∉ T AND w matched THEN
                T.add(w)
```

T.add(mate(w))

queue.push(mate(w))

ELSE IF w ∈ T AND even-length cycle detected THEN

CONTINUE

ELSE IF w ∈ T AND odd-length cycle detected THEN

shrink cycle

ELSE IF w ∈ F THEN

expand all contracted nodes

reconstruct augmenting path

invert augmenting path

END

2.1.2. Algoritmo realizacijos programoje aprašymas

Kuriame struktura Grafas ir joje apibrežiame mums reikiamas funkcijas

```
    //             Sekiame kelią iki šaknies
    vector<int> path_to_root(int x)   // Sudetingumas O(n)
    {
        vector<int> xpath_to_root;  // vx - kelias iki saknies nuo x visunes
        while (true)
        {
            while (blossom_containing_vertex[x] != x)
                x = blossom_containing_vertex[x];

            if (!xpath_to_root.empty() && xpath_to_root.back() == x)
                break;

            xpath_to_root.push_back(x);
            x = parent[x];
        }
        return xpath_to_root;
    }

    // Žiedo susitraukimas
    // c - id naujo susitraukiusio žiedo
    // x, y - virsunes brauna tarp kuriu susidaro cikla
    // vx, vy - keliai į šakni nuo x ir y atitinkamai
    // r - žemiausias bendras protėvis
    // Sudetingumas O(n*|#blossom|) kur |blossom| virsuniu/žiedu sutrauktu į c
skaicius
```

```
    void shrinking(int blossom_, int vertex_x, int vertex_y, vector<int>&
xpath_to_root, vector<int>& ypath_to_root)

    {
        vertices_in_blossom[blossom_].clear();
        int r = xpath_to_root.back();
        while (!xpath_to_root.empty() && !ypath_to_root.empty() &&
xpath_to_root.back() == ypath_to_root.back())
        {
            r = xpath_to_root.back();
            xpath_to_root.pop_back();
            ypath_to_root.pop_back();
        }
        vertices_in_blossom[blossom_].push_back(r);
        vertices_in_blossom[blossom_].insert(vertices_in_blossom[blossom_].end(),
xpath_to_root.rbegin(), xpath_to_root.rend());
        vertices_in_blossom[blossom_].insert(vertices_in_blossom[blossom_].end(),
ypath_to_root.begin(), ypath_to_root.end());
        for (int i = 0; i <= blossom_; i++)
        {
            edge_matrix[blossom_][i] = edge_matrix[i][blossom_] = -1;
        }
        for (int z : vertices_in_blossom[blossom_])
        {
            blossom_containing_vertex[z] = blossom_;
            for (int i = 0; i < blossom_; i++)
            {
                if (edge_matrix[z][i] != -1) {
                    edge_matrix[blossom_][i] = z;
                    edge_matrix[i][blossom_] = edge_matrix[i][z];
                }
            }
        }
    }

    // Kelio atstatymas (tai yra visu susitraukiusiu žiedu atstatymas)
    // z - stako virsune
    // w - sekantis elementas uz z
    // i - vertices_in_blossom[z] vektoriaus paskutinios virsunes indeksas
    // j - vertices_in_blossom[z] vektoriaus pirmos virsunes indeksas
    // dif - kryptis kuria turime eiti nuo i iki j kad kelias teisingai alternuotusi
    // Sudetingumas O(n)
    vector<int> lift(vector<int>& xpath_to_root)
    {
        vector<int> A;
        while (xpath_to_root.size() >= 2)
        {
            int top_of_stack = xpath_to_root.back(); xpath_to_root.pop_back();
            if (top_of_stack < n)
            {
                A.push_back(top_of_stack);
                continue;
            }
            int behind_the_top_of_stack = xpath_to_root.back();
            int i = (A.size() % 2 == 0 ?
find(vertices_in_blossom[top_of_stack].begin(),
vertices_in_blossom[top_of_stack].end(),
```

```
edge_matrix[top_of_stack][behind_the_top_of_stack]) -
vertices_in_blossom[top_of_stack].begin() : 0);
            int j = (A.size() % 2 == 1 ?
find(vertices_in_blossom[top_of_stack].begin(),
vertices_in_blossom[top_of_stack].end(), edge_matrix[top_of_stack][A.back()]) -
vertices_in_blossom[top_of_stack].begin() : 0);
            int k = vertices_in_blossom[top_of_stack].size();
            int direction = (A.size() % 2 == 0 ? i % 2 == 1 : j % 2 == 0) ? 1 : k -
1;
            while (i != j)
            {
                xpath_to_root.push_back(vertices_in_blossom[top_of_stack][i]);
                i = (i + direction) % k;
            }
            xpath_to_root.push_back(vertices_in_blossom[top_of_stack][i]);
        }
        return A;
    }



    int find_maximum_matching()
    {
        for (int maximum_match_size = 0; ; maximum_match_size++)
        {
            fill(label.begin(), label.end(), 0);   // uzpildom label vektori nuliais
            queue<int> Queue;
            for (int i = 0; i < m; i++) blossom_containing_vertex[i] = i;
            for (int i = 0; i < n; i++)
            {
                if (mate[i] == -1)
                {
                    Queue.push(i);
                    parent[i] = i;
                    label[i] = 1;
                }
            }
            int blossom_ = n;
            bool augmenting_path_exists = false;
            while (!Queue.empty() && !augmenting_path_exists)
            {
                int vertex_x = Queue.front(); Queue.pop();
                if (blossom_containing_vertex[vertex_x] != vertex_x)
                    continue;
                for (int vertex_y = 0; vertex_y < blossom_; vertex_y++)
                {
                    if (blossom_containing_vertex[vertex_y] == vertex_y &&
edge_matrix[vertex_x][vertex_y] != -1)
                    {
                        if (label[vertex_y] == 0)
                        {
                            parent[vertex_y] = vertex_x;
                            label[vertex_y] = 2;
                            parent[mate[vertex_y]] = vertex_y;
                            label[mate[vertex_y]] = 1;
                            Queue.push(mate[vertex_y]);
                        }
                        else if (label[vertex_y] == 1)
```

```
                        {
                            vector<int> xpath_to_root = path_to_root(vertex_x);
                            vector<int> ypath_to_root = path_to_root(vertex_y);
                            if (xpath_to_root.back() == ypath_to_root.back())
                            {
                                shrinking(blossom_, vertex_x, vertex_y,
xpath_to_root, ypath_to_root);
                                Queue.push(blossom_);
                                parent[blossom_] =
parent[vertices_in_blossom[blossom_][0]];
                                label[blossom_] = 1;
                                blossom_++;
                            }
                            else
                            {
                                augmenting_path_exists = true;
                                xpath_to_root.insert(xpath_to_root.begin(),
vertex_y);

                                ypath_to_root.insert(ypath_to_root.begin(),
vertex_x);

                                vector<int> A = lift(xpath_to_root);
                                vector<int> B = lift(ypath_to_root);
                                A.insert(A.end(), B.rbegin(), B.rend());
                                for (int i = 0; i < (int)A.size(); i += 2) {
                                    match(A[i], A[i + 1]);
                                    if (i + 2 < (int)A.size()) add_edge(A[i + 1],
A[i + 2]);
                                }
                            }
                            break;
                        }
                    }
                }
            }
            if (!augmenting_path_exists)
            {
                return maximum_match_size;
            }
        }
    }

};
```

Musu programos main bloke sukuriame trumpa funkcija leidžianti generuoti grafus mums norimo dydžio algoritmo testavimui.

```
for(int n = 1; n != 11; n++)
    {
        number_of_vertices = n * 1000;
        number_of_edges = 0;
        Grafas grafas(number_of_vertices);
        cout << "Sugeneruotas grafas" << endl;
        for (int i = 0; i != number_of_vertices; i++)
        {
            random_1 = (rand() % 3)+10; // number_of_neighbours from 10 to 12
            for (int a = 0; a != random_1; a++)
            {
                random_2 = rand() % number_of_vertices + 1;
```

```
                grafas.add_edge(i + 1, random_2);
                number_of_edges++;
                //cout << i + 1 <<" "<< random_2 << endl;
            }
        }
```

number_of_vertices – kiek viršūnių grafe bus generuojama

random_1 – atsitiktinis skaičius nuo 10 iki 12. Reiškia kad kiekviena viršūnė turės nuo 10 iki 12 briaunų

random_2 – atsitiktinis skaičius nuo 1 iki number_of_vertices

number_of_edges – jo pagalba skaičiuojame kiek briaunų buvo sukurta musu grafe.

3. Sudėtingumo analizė

Teorinis algoritmo sudėtingumas:

Kiekviena iteracija bloko kuris randa didinančiąja kelia padidina viršūnių turinciu pora 1, todėl iš viso gali būti daugiausia n/2 iteracijų. BFS operacijoms reikės O (n2) laiko. Didinamasis kelias randamas daugiausia vieną kartą per iteraciją ir tai trunka O (n) laiko. Žiedų susitraukimas trunka O(n|bc|).

Bendras žiedų susitraukimų iteracijos laikas yra O(n2). Kadangi kiekvienai iteracijai reikia O(n2) laiko ir iš viso yra O(n) iteracijų, matome, kad algoritmas užtrunka O(n3) laiko.

4. Bandymai

Microsoft Visual Studio Debug Console

```
Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 499

Grafo, turincio 1000 virsuniu ir 4578 briaunu Edmondso algoritmas truko0.202933sekund₨iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 998

Grafo, turincio 2000 virsuniu ir 8922 briaunu Edmondso algoritmas truko1.10275sekund₨iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1497

Grafo, turincio 3000 virsuniu ir 13337 briaunu Edmondso algoritmas truko3.15483sekund₨iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1996

Grafo, turincio 4000 virsuniu ir 17894 briaunu Edmondso algoritmas truko6.13172sekund₨iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 2496

Grafo, turincio 5000 virsuniu ir 22429 briaunu Edmondso algoritmas truko10.1626sekund₨iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 2997

Grafo, turincio 6000 virsuniu ir 26782 briaunu Edmondso algoritmas truko11.861sekund₨iu
```

Microsoft Visual Studio Debug Console

```
Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 3497

Grafo, turincio 7000 virsuniu ir 31429 briaunu Edmondso algoritmas truko19.3887sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 3995

Grafo, turincio 8000 virsuniu ir 35731 briaunu Edmondso algoritmas truko27.1558sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 4492

Grafo, turincio 9000 virsuniu ir 40322 briaunu Edmondso algoritmas truko40.8992sekund₧iu
```

1-as bandymas:

Grafo viršūnių skaičius: 1000;   Vykdymo laikas: 0.202933

2-as bandymas:

Grafo viršūnių skaičius: 2000;   Vykdymo laikas: 1.10275

3-as bandymas:

Grafo viršūnių skaičius: 3000;   Vykdymo laikas: 3.15483

4-as bandymas:

Grafo viršūnių skaičius: 4000;   Vykdymo laikas: 6.13172

5-as bandymas:

Grafo viršūnių skaičius: 5000;   Vykdymo laikas: 10.1626

6-as bandymas:

Grafo viršūnių skaičius: 6000;   Vykdymo laikas: 16.6504

7-as bandymas:

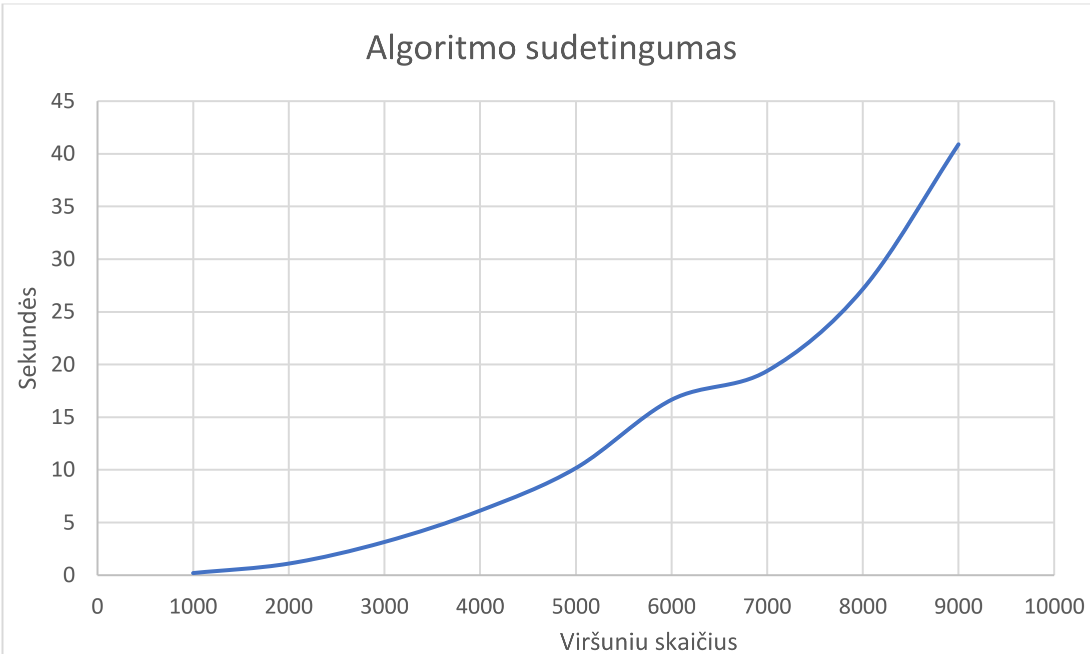
Grafo viršūnių skaičius: 7000;   Vykdymo laikas: 19.3887

8-as bandymas:

Grafo viršūnių skaičius: 8000;   Vykdymo laikas: 27.1558

9-as bandymas:

Grafo viršūnių skaičius: 9000;   Vykdymo laikas: 40.8992

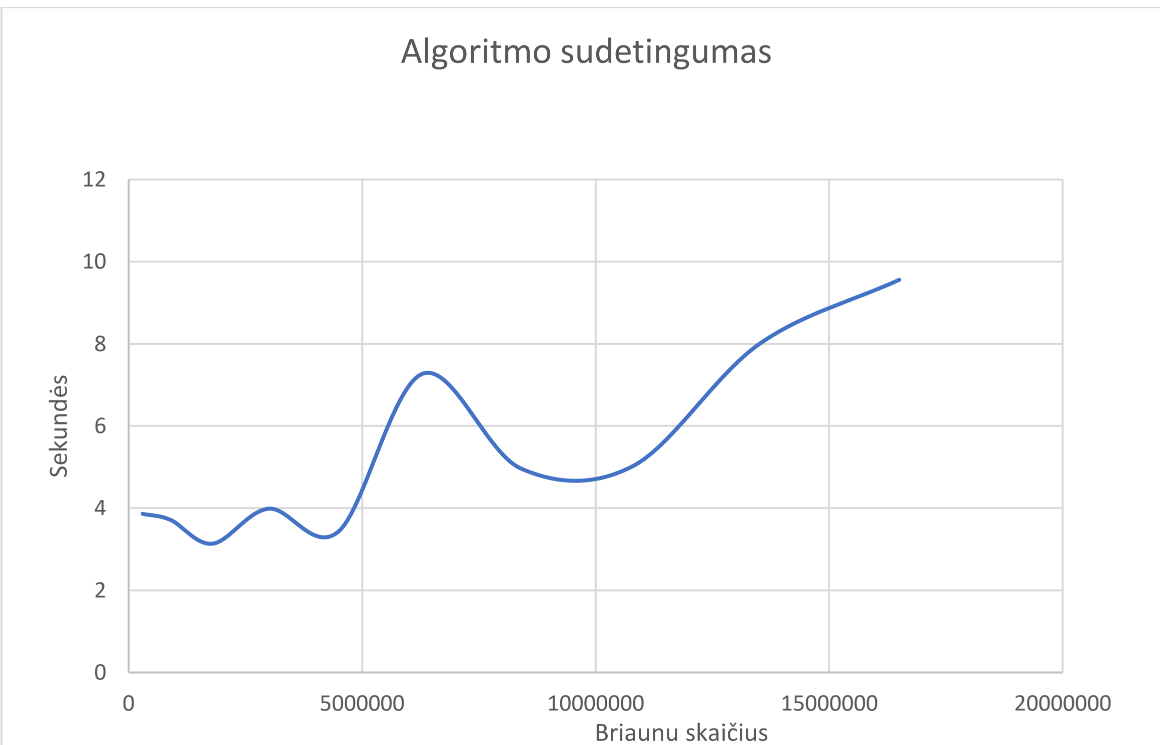


Dabar fiksuojame viršūniu skaičiu(3000) ir nagrinėsime programos veikimo laiko kitimą, didinant briaunų skaičiu.

Microsoft Visual Studio Debug Console

```
Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 300000 briaunu Edmondso algoritmas truko 3.86318sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 900000 briaunu Edmondso algoritmas truko 3.7134sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 1800000 briaunu Edmondso algoritmas truko 3.13596sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 3000000 briaunu Edmondso algoritmas truko 3.98798sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 4500000 briaunu Edmondso algoritmas truko 3.43768sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 6300000 briaunu Edmondso algoritmas truko 7.27508sekund₧iu


Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 8400000 briaunu Edmondso algoritmas truko 4.96556sekund₧iu
```

```
Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 10800000 briaunu Edmondso algoritmas truko 5.02576sekund₧iu

Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 13500000 briaunu Edmondso algoritmas truko 7.99318sekund₧iu

Sugeneruotas grafas

Grafo maximalaus poravimo dydis yra 1499

Grafo, turincio 3000 virsuniu ir 16500000 briaunu Edmondso algoritmas truko 9.55973sekund₧iu
```



5.Programos naudojimo instrukcija.

1. Paleidžiam programa iš Non_bipartite_matching.exe failo

2. Jeigu norime įvesti ranka, spaudžiame 1 ENTER, įvedame viršūnių skaičių, o po to briaunas tokiu pavidalu:

1 2 (spaudžiame Enter)

1 3 (spaudžiame Enter)

2 3 (spaudžiame Enter)

3 4 (spaudžiame Enter)

3. Jei norime sugeneruoti spaudžiame 2 ENTER:

3.1 Įvedame kiek briaunų mes norime sugeneruoti

3.2 Įvedame kiek incidentiniu briaunų turės viena viršūnė. Programa atsitiktinai generuos su kokiomis viršūnėmis bus sujungtos briaunos.

4. Jei norime paleist jau paruošta testavima, spaudžiame 3 ENTER.

5. Norint išeiti iš programos spaudžiame 0 ENTER.

6.Išvados

Šio darbo metu buvo susipažinta su maksimalaus grafo poravimo algoritmu. Buvo realizuotas Edmondso poravimo algoritmas (Edmonds blossom algorithm). Empiriškai testuojant algoritmą ir nubraižęs grafika, buvo įsitikinta, jog algoritmo sudėtingumas yra $O(n^3)$.

Naudotos literatūros sąrašas

https://en.wikipedia.org/wiki/Blossom_algorithm

A Course in Combinatorial Optimization Alexander Schrijver.