



Tallinna Tehnikaülikool
Informaatikainstituut

Kõikide maksimaalsete klikkide leidmine

iseseisev töö #8 õppeaines
Eksperimentaalsed sõresüsteemid
IDN5590

juhendaja :	Rein Kuusik
üliõpilane :	Erki Suurjaak
e-post :	erki@lap.ttu.ee
matrikkel :	970772
õpperühm :	LAP62

Tallinn 2000

Sisukord

Sisukord.....	2
1. Meetod	3
1.1. Tutvustus	3
1.2. Algoritm.....	3
2. Programm	4
2.1. Tutvustus	4
2.2. Programmi kasutajajuhend	5
2.3. Programmi väljund	5
2.4. Programmi listing	5

1. Meetod

1.1. Tutvustus

Algoritm "kõikide maksimaalsete klikkide eraldamine" leiab etteantud graafist kõik maksimaalsed klikid. Algoritmi juures kasutatakse monotoonsete süsteemide teoriat - graafi kujutatakse seosemaatriksina. Analüüsitud graafid on antirefleksiivsed (maatriksi peadiagonaal sisaldab nulle) ja sümmeetrlised (maatriksi element x_{ij} võrdub maatriksi elemendiga x_{ji}).

1.2. Algoritm

Realisatsiooni algoritm on järgmine :

- 1 Arvutame graafi tippude kaalud (kaalufunktsiooniks on tippude valents - tipu kaaluks on tema naabertippude arv)
- 2 Kontrollime, kas me saame graafil kergelt eraldada maksimaalse kliki, kasutades 0-reeglit¹ ja $(n-1)$ -reeglit². Kui mõni neist reeglitest rakendub, siis kontrollime täiendavalt eraldatavata kliki maksimaalsust³.
- 3 Kui graafi kõik tipud on null-kaaluga, siis läheme analüüsi eelmisele tasemele (kui oleme algasemel, siis on analüüs lõppenud).
- 4 Valime juhttipuks suurima kaaluga punkti (või esimese suurima kaaluga punkti, kui sama kaal on mitmel tipul). Teeme tema järgi väljavõtu. Kustutame alggraafil kõik juhttipu kaared.
- 5 Arvutame väljavõtu kaalud. Teeme tagasivõndluse - kui mõne tipu kaalud on samad nii väljavõtul kui alggraafil, siis kustutame alggraafil selle tipu kõik kaared.
- 6 Läheme sügavamale tasemele, rakendades väljavõtule realisatsiooni algoritmi (läheme punkti 1, graafiks on väljavõtt).

¹ **0-reegel** - kui graafil leidub null-kaaluga tipp ja analüüs ei ole algasemel, siis oleme leidnud maksimaalse kliki - see moodustub antud tipust ning analüüsi käigus juhttipuudeks valitud tippudest. Reeglit võib rakendada iga null-kaaluga tipu kohta, s.o. iga selline tipp näitab ühe maksimaalse kliki olemasolu.
NB! 0-reeglit tohib kasutada ainult üks kord iga graafi puhul.

² **(n-1)-reegel** - kui graafil on n tippu, mille kaal on $n-1$, ja ülejäänud graafi tipud on null-kaaluga, siis oleme leidnud maksimaalse kliki - see moodustub antud tippudest ning analüüsi käigus juhttipuudeks valitud tippudest. Reeglit võib rakendada ka algasemel. Reegli rakendumisel kustutatakse antud tippude kõik kaared.

³ **eraldataava kliki maksimaalsuse kontroll** - arvutame esialgse graafi kõikide tippude (v.a. tipud, mis kuuluvad klikki) kaalu kliki tippude suhtes (kaalumisel arvestatakse ainult klikki kuuluvaid tippe). Kui mõne tipu kaal on võrdne eraldatavata kliki tippude arvuga, siis pole klick maksimaalne.

2. Programm

2.1. Tutvustus

Programm on kirjutatud programmeerimiskeeles C++. Programmi levitatav versioon on kompileeritud programmeerimiskeskonna DJGPP v2.03 koosluses oleva kompilaatoriga gcc v2.95 MS-DOS keskkonnas. Programm töötab MS-DOS keskkonnas.

Programmi kiirus ja analüüsitava graafi võimalik suurus ei ole üheselt määratud - see sõltub kasutatava arvuti kiirusest ja mälumahust; eriti aga graafi enda struktuurist. 400-tipuline graaf võib sisaldada 46 000 maksimaalset klikki ja analüüs võib kesta 22 sekundit; 400-tipuline graaf võib ka sisaldada 2 000 000 maksimaalset klikki ja analüüs võib kesta 30 minutit.

Kiirusnäited : arvuti peal, kus on Celeron 433 MHz protsessor ja 128 MB põhimälu, tulid järgmised tulemused :

Graafi tippude arv	Maksimaalsete klikkide arv	Aeg (MM:SS)
10	9	00:00
20	36	00:00
50	398	00:00
100	5 287	00:01
200	84 908	00:34
400	2 094 633	28:55

Kõik graafid olid juhuslikult genereeritud, tõenäosus kaare tekkeks kahe tipu vahel oli valitud 0,4.

Sisendfaili struktuur :

N
N
 $x_{11} \ x_{12} \ \dots \ x_{1N}$
 $x_{21} \ x_{22} \ \dots \ x_{2N}$
 \dots
 $x_{N1} \ x_{N2} \ \dots \ x_{NN}$

N on graafi tippude arv. x_{ij} näitab kaare olemasolu tippude i ja j vahel (1 = on, 0 = pole). Eleemendid x_{ii} peavad olema nullid. Sisendfailis võib peale tabelit olla prahti.

2.2. Programmi kasutajajuhend

Programmile antakse parameetritega ette sisendfaili nimi ja väljundfaili nimi.
Süntaks : cliques.exe sisendfail väljundfail

Programmiga tuleb kaasa graafide genereerimise programm crgraph.exe, mille kasutusjuhend on :

crgraph.exe	väljundfail	tippude_arv	kaare_tõenäosus,	kus
				väljundfaili nimi
		tippude_arv		graafi tippude arv
		kaare_tõenäosus		tõenäosus kahe graafi punkti vahele kaare
				tekkimiseks

2.3. Programmi väljund

Programmi väljundi näide :

```
Analysis started at Sun Mar 19 17:22:54 2000
Input file name : aaa6
Output file name : rrr6

The graph has 10 points.
Maximum cliques are :
{1, 2, 10}
{1, 2, 3, 4}
{4, 6}
{5, 6}
{6, 7, 9}
{3, 5}
{7, 8}
{8, 10}
{9, 10}

Number of maximum cliques found : 9
Analysis finished at Sun Mar 19 17:22:54 2000
```

2.4. Programmi listing

```
/*
Author      : Erki Suurjaak
Created     : 16.03.2000
Last changed : 28.05.2000
```

The program analyzes the graph in the input file and finds all the maximum cliques. Program usage : cliques.exe input_file output_file

```
*/
```

```

#include <iostream>
#include <fstream>
#include <vector>
#include <map>
#include <slist>
#include <time.h>
#include <new.h>
using namespace std;

// ----- Global type-definitions -----

typedef signed short int_type;           // The type of the point number
typedef vector<bool> graph_column;      // The type of a graph table column
typedef vector<graph_column> graph_table; // The type of a graph table
typedef slist<int_type> point_stack;     // The type of stack to hold necessary points

// ----- Global type-definitions -----

// ----- Global constants -----


const char NL = '\n';
const char* const ErrorMessageInputFileError =
    "\ncliques : there was an error reading from the input file.\n";
const char* const ErrorMessageOutputFileError =
    "\ncliques : there was an error writing to the output file.\n";
const char* const ErrorMessageInputFileDataError =
    "\ncliques : there was an error in the input file data structure.\n";
const char* const ErrorMessageNoMoreMemory =
    "\ncliques : memory allocation failed : not enough memory.\n";
const char* const HelpMessage =
    "\nThis program finds all the maximum cliques in a graph.\n"
    "Program usage : cliques.exe input_file output_file\n"
    "Inputfile format : \n"
    "N\n"
    "N\n"
    "x11 x12 .. x1N\n"
    "x21 x22 .. x2N\n"
    "... \n"
    "xN1 xN2 .. xNN\n"
    "where N is the number of points on the graph and element xij shows\n"
    "a connection between points i and j (1 = connection, 0 = no connection).\n"
    "The graph must be antireflexive (the main diagonal must contain zeroes) and\n"
    "symmetrical (xij == xji).\n"
;

// ----- /Global constants -----

```

```

// ----- Global functions -----


// Displays Message and exits the program with errorlevel 255
inline void FatalError (const char* const Message)
{
    if (cerr) cerr << Message;
    exit (255);
    return;
}

// Function to be called when memory allocation fails
inline void NoMoreMemory( )
{
    FatalError (ErrorMessageNoMoreMemory);
    return;
}

// ----- /Global functions -----


// ----- class TGraph -----


class TGraph
{
private :
    // The amount of points on the graph
    int_type
    N;

    // The table containing the graph
    graph_table
    Graph;

    // The array containing the current weights of points. It is used
    // for quicker and simpler program flow. Theoretically, when making
    // a selection based on a heaviest point, we should change the
    // table Graph, eliminating the necessary selections, and then
    // re-weigh the points. Practically, it is easier (and more efficient)
    // to simply decrease the necessary weights - as the weights need to
    // be calculated anyway. This procedure creates the need for a few
    // additional comparisons here and there, but eliminates the need
    // for weighing the points again and again. The additional memory
    // overhead is also not worth mentioning - the main issue is working
    // time.
    vector<int_type>
    Weight;
}

```

```

// These are used to create mappings between a point's number on the
// current graph and its real number on the original graph.
map<int_type, int_type>
CurrentToReal,
RealToCurrent;

// Holds the selected heaviest points that have been used to construct
// lesser graphs. Used while writing a clique to file.
static point_stack
HeaviestPoints;

// The copy of the global graph is needed when determining whether a
// potential maximum clique is really maximum.
static graph_table
OriginalGraph;

// Shows the number of maximum cliques found
static unsigned long
NumberOfCliques;

// This method writes the clique to the file "out". File must be
// open and writable. First the clique is checked whether it is
// really a maximum clique or whether it is already contained in
// some other, bigger clique.
// If the clique is maximum, then the clique is sorted (for aesthetic
// purpose only). The written clique has the following format :
// "{PointNumber1, PointNumber2, ..., PointNumberN}"
void
WriteClique (ofstream& out, point_stack Clique) const;

public :
    // Return the number of cliques found
    inline unsigned long
    GetNumberOfCliques () const { return NumberOfCliques; }

    // This method reads the graph from the file "in". The validity of
    // the graph is checked and, if an error is found, the program exits.
    // The input file must be opened and readable and has the following
    // format :
    // N
    // N
    // x11 x12 x13 .. x1N
    // x21 x22 x23 .. x2N
    // ..
    // xN1 xN2 xN3 .. xNN
    // Returns 0 if there was an error in the input file, otherwise
    // returns the number of points on the graph.
    int_type
    ReadGraph (ifstream& in);

    // This method finds all the maximum cliques and writes them to the
    // file "out" (file must be open and writable). The method uses
    // recursion - during analysis, the graph is broken down into smaller
    // subgraphs, which in turn are analyzed.
    void
    WriteOutAllMaxCliques (ofstream& out);

```

```

};

// This method writes the clique to the file "out". File must be
// open and writable. First the clique is checked whether it is
// really a maximum clique or whether it is already contained in
// some other, bigger clique.
// If the clique is maximum, then the clique is sorted (for aesthetic
// purpose only). The written clique has the following format :
// "{PointNumber1, PointNumber2, ..., PointNumberN}"
void TGraph::WriteClique (ofstream& out, point_stack Clique) const
{
    int_type
    CliqueSize = Clique.size (),           // The size of the clique
    OriginalN = OriginalGraph.size (),    // The size of the original graph
    i;                                     // Just a variable for iterations
    point_stack::iterator ListIterator; // Just a variable for iterations
    bool CliqueIsNotMaximum = false;

    // A map used for checking whether a point is in the clique
    bool PointIsNotInTheClique[OriginalN];
    for (i = 0; i < OriginalN; i++)
        PointIsNotInTheClique[i] = true;

    for (ListIterator = Clique.begin (); ListIterator != Clique.end ();
         ListIterator++)
        PointIsNotInTheClique[*ListIterator - 1] = false;

    // Before writing the clique to file, we must check whether it is
    // really maximum. It goes as follows - we weigh every point on
    // the original graph (except the points in the current clique)
    // to the points in the current clique. If any weight equals the
    // number of points in the current clique, the clique is not
    // maximum. In other words, if a point's weight equals the number of points
    // in the current clique, then the point is connected to all those points
    // and forms a bigger clique.

    for (i = 0; i < OriginalN; i++)
        if (PointIsNotInTheClique[i])
    {
        int_type CurrentWeight = 0;

        for (ListIterator = Clique.begin (); ListIterator != Clique.end ();
             ListIterator++)
            CurrentWeight += OriginalGraph[*ListIterator - 1][i];

        if (CurrentWeight == CliqueSize)
        {
            CliqueIsNotMaximum = true;
            break;
        }
    }

    if (CliqueIsNotMaximum)
        return;
}

```

```

NumberOfCliques++;
Clique.sort ();
out << '{';
out << Clique.front ();
Clique.pop_front ();
while (!Clique.empty ())
{
    int_type CurrentPoint = Clique.front ();
    Clique.pop_front ();
    out << ", " << CurrentPoint;
}
out << '}' << NL;
}

// This method reads the graph from the file "in". The validity of
// the graph is checked and, if an error is found, the program exits.
// The input file must be opened and readable and has the following
// format :
// N
// N
// x11 x12 x13 .. x1N
// x21 x22 x23 .. x2N
// ..
// xN1 xN2 xN3 .. xNN
// Returns 0 if there was an error in the input file, otherwise
// returns the number of points on the graph.
int_type TGraph::ReadGraph (ifstream& in)
{
if (!(in >> N))
    return 0;

int copyOfN;
if (!(in >> copyOfN))
    return 0;

if (N != copyOfN)
    return 0;

int_type i, j; // Just some variables for iterations

for (i = 0; i < N; i++)
{
    graph_column CurrentColumn;
    for (j = 0; j < N; j++)
    {
        bool TempBool;
        if (!(in >> TempBool))
            return 0;
        CurrentColumn.push_back (TempBool);
    };
    Graph.push_back (CurrentColumn);
}

// Validating the graph
for (i = 0; i < N; i++)
    for (j = i; j < N; j++)

```

```

{
  if (i == j)
  {
    if (Graph[i][j]) // Main diagonal must contain zeroes
      return 0;
  }
  // Other elements must be mirrored upon the main diagonal
  else if (Graph[i][j] != Graph[j][i])
    return 0;
};

// Construct the mappings between a point's graph number and
// real (global) number.
for (i = 0; i < N; i++)
{
  CurrentToReal[i] = i + 1;
  RealToCurrent[i + 1] = i;
}

for (i = 0; i < N; i++)      // Weigh all the points
{
  Weight.push_back (NULL);
  for (j = 0; j < N; j++)
    Weight[i] += Graph[i][j];
}

OriginalGraph = Graph; // The copy of the global graph is needed
                      // when determining whether a potential
                      // maximum clique is really maximum.

return N;
}

// This method finds all the maximum cliques and writes them to the
// file "out" (file must be open and writable). The method uses
// recursion - during analysis, the graph is broken down into smaller
// subgraphs, which in turn are analyzed.
void TGraph::WriteOutAllMaxCliques (ofstream& out)
{
  int_type i, j; // Just some variables for iterations

  // When first entering the graph analysis, we must check whether
  // the zero-rule can be applied. That is, whether there are points
  // of zero weight. If there are, then we have found a clique that
  // consists of that point and the points in the LeadPoint stack.

  if (!HeaviestPoints.empty ()) // This rule cannot be applied on the
    for (i = 0; i < N; i++)      // very first level, as the point that
      if (!Weight[i])            // is not connected to anything does
      {
        // not form a clique.
        HeaviestPoints.push_front (CurrentToReal[i]);
        WriteClique (out, HeaviestPoints);
        HeaviestPoints.pop_front ();
      }
}

```

```

// Now starts the main block of the method. It is executed until
// all the weights have been reduced to zero - then we can return
// to the previous level of analysis (or exit the analysis, if we
// are on the base level).

while (1)
{
    // We must check whether the (n-1)-rule can be applied. It goes
    // as follows : if there are n points that have the weight of n-1,
    // and all the other points are of zero weight, then those points
    // (and the points in the HeaviestPoints stack) form a clique.

    int_type SelectedWeight = 0;
    for (i = 0; i < N; i++)           // First we iterate through the
        if (Weight[i])              // weights and find a non-zero
    {
        SelectedWeight = Weight[i];
        break;
    }

    bool NMinus1Applicable = true;      // Then we count the
    int_type PointsWithSelectedWeight = 0; // points of that weight
    for (i = 0; i < N; i++)           // and check whether there
        if (Weight[i])              // are any points of
    {
        if (SelectedWeight != Weight[i]) // different weight. If
        {
            NMinus1Applicable = false; // there are, then the
            break;                  // (n-1)-rule cannot be
        }
        else PointsWithSelectedWeight++;
    }

    if (NMinus1Applicable
        && PointsWithSelectedWeight - 1 == SelectedWeight)
    {
        int PointsPushed = 0;
        for (i = 0; i < N; i++)
            if (Weight[i] == SelectedWeight)
        {
            HeaviestPoints.push_front (CurrentToReal[i]);
            Weight[i] = 0;
            PointsPushed++;
        }
        WriteClique (out, HeaviestPoints);
        for (i = 0; i < PointsPushed; i++)
            HeaviestPoints.pop_front ();
    }

    // Then we check whether all points are of zero weight. If
    // they are, then we can exit the method.
    bool AllWeightsAreZero = true;
    for (i = 0; i < N; i++)

```

```

if (Weight[i]) AllWeightsAreZero = false;

if (AllWeightsAreZero)
    return;

// If there are yet weighted points, we must analyze them. This
// goes as follows : we find the heaviest point (or the first of
// the heaviest points). We construct a selection based on that
// point - the selection graph contains only those points that
// are connected to the heaviest point and are weighted. The
// weight of the heaviest point is reduced to zero on the current
// graph (it has been exhausted by the selection) and the weight
// of all the connected points is reduced by one (as their
// connection to the heaviest point is now discarded). We weigh
// the new graph. We compare the weights of the old and the new
// graph; if some points have equal weight on both graphs, then
// their weights are reduced to zero on the old graph (the reason
// for this is - because their weights did not change, then the
// new graph contains all the cliques they belong to).

// Selecting the heaviest point.
int_type MaxWeight = 0,
HeaviestPoint = N - 1;
for (i = N - 1; i >= 0; i--)
if (Weight[i] >= MaxWeight)
{
    MaxWeight = Weight[i];
    HeaviestPoint = i;
}

TGraph NewSelection;

// Constructing the new graph
j = 0;
for (i = 0; i < N; i++)
if (Graph[i][HeaviestPoint] && Weight[i])
{
    NewSelection.RealToCurrent[CurrentToReal[i]] = j;
    NewSelection.CurrentToReal[j] = CurrentToReal[i];
    graph_column CurrentColumn;
    for (int_type k = 0; k < N; k++)
        if (Graph[HeaviestPoint][k] && Weight[k])
            CurrentColumn.push_back (Graph[i][k]);
    NewSelection.Graph.push_back (CurrentColumn);
    j++;
}

for (i = 0; i < N; i++)
if (Graph[i][HeaviestPoint] && Weight[i])
    Weight[i]--;
}

NewSelection.N = j;

Weight[HeaviestPoint] = 0;

// Weighing the new graph

```

```

for (i = 0; i < NewSelection.N; i++)
{
    NewSelection.Weight.push_back (NULL);
    for (j = 0; j < NewSelection.N; j++)
        NewSelection.Weight[i] += NewSelection.Graph[i][j];
}

// Comparing the weights of the new and the old graph
point_stack PointsToLighten;
for (i = 0; i < NewSelection.N; i++)
{
    int_type PointNumber = RealToCurrent[NewSelection.CurrentToReal[i]];
    if (NewSelection.Weight[i] == Weight[PointNumber])
    {
        Weight[PointNumber] = 0;
        for (j = 0; j < N; j++)
            if (Graph[PointNumber][j] && Weight[j])
                PointsToLighten.push_front (j);
    }
}
while (!PointsToLighten.empty ())
{
    if (Weight[PointsToLighten.front ()])
        Weight[PointsToLighten.front ()]--;
    PointsToLighten.pop_front ();
}

// Add the heaviest point to stack
HeaviestPoints.push_front (CurrentToReal[HeaviestPoint]);

NewSelection.WriteOutAllMaxCliques (out);

HeaviestPoints.pop_front ();

}
return;
}

// Initializing static data members
graph_table TGraph::OriginalGraph;
point_stack TGraph::HeaviestPoints;
unsigned long TGraph::NumberOfCliques = 0;

// ----- /class TGraph -----


int main (int argc, char* argv[])
{
if (argc != 3)
    FatalError (HelpMessage);

set_new_handler (NoMoreMemory);

```

```

ifstream in (argv[1]);
if (!in)
    FatalError (ErrorMessageInputFileError);
ofstream out (argv[2]);
if (!out)
    FatalError (ErrorMessageOutputFileError);

TGraph Graph;

int_type Points = Graph.ReadGraph (in);
in.close ();
if (!Points)
    FatalError (ErrorMessageInputFileDialogError);

time_t now;
time (&now);
out << "Analysis started at " << asctime (localtime (&now));
out << "Input file name : " << argv[1] << NL;
out << "Output file name : " << argv[2] << NL;
out << NL;
out << "The graph has " << Points << " points." << NL;
out << "Maximum cliques are : " << NL;

Graph.WriteOutAllMaxCliques (out);

out << NL;
out << "Number of maximum cliques found : " << Graph.GetNumberOfCliques ()
    << NL;
time (&now);
out << "Analysis finished at " << asctime (localtime (&now));

out.close ();

return 0;
}

```