Лабораторная работа № 8

Графы

Цель работы: изучить основные алгоритмы работы с графами – обход, построение транзитивного замыкания, поиск кратчайших путей, закрепить навыки структурного и объектно-ориентированного программирования.

Задание

1. Используя исходный код, приведенный в приложении, разработать функции (методы) реализующие задания, представленные в таблице 1 в соответствии с вариантом задания. При этом необходимо выбрать тип графа и методы, которые позволят эффективно решить поставленную задачу. Решение аргументировать.

Таблица 1.

Варианты заданий

№	Задание
1	Граф представляет карту дорог. Найти кратчайший путь из города A в город B, минуя город C. Если такой путь не существует, выдать соответствующее сообщение.
2	Граф представляет карту дорог. Найти кратчайший путь из города A в город B, через город C. Если такой путь не существует, выдать соответствующее сообщение.
3	Три графа представляют маршруты трех автобусов (множество вершин считается одинаковым). Определить, возможно ли добраться от остановки A до остановки B.
4	Три графа представляют маршруты трех автобусов (множество вершин считается одинаковым). Найти кратчайший путь от остановки A до остановки B, если такой путь не существует, выдать соответствующее сообщение
5	Найти путь, представляющий диаметр графа.
6	Граф представляет карту дорог. Найти такой город D, что сумма длин путей до городов A, B, C была не меньше заданной.
7	Граф представляет карту дорог. Найти все города, длина пути до которых от города A не превышает заданную.
8	Граф представляет карту дорог. Найти все города, длина пути до которых от города A не меньше заданной.
9	Граф представляет карту дорог. Найти такой город A, сумма длин путей от которого до всех остальных минимальна.
10	Граф представляет собой авиарейсы. Найти такой город A, из которого существует возможность добраться до всех остальных, сделав не более одной пересадки.
11	Граф представляет карту дорог. Найти такой город D, что сумма расстояний до городов A, B, C была минимальна.

Приложение А

```
#include <values.h>
#include <iostream.h>
#include <conio.h>
#include <stdlib.h>
//---------------------------------------------------------------------------
struct Item {
int D;
Item * Next;
  Item():D(-1),Next(0){}
  Item(int V): D(V),Next(0){}
};
//---------------------------------------------------------------------------
class Cont {
protected:
Item * _Head;
Item * _Tail;
public:
    Cont():_Head(0),_Tail(0){}
    bool Empty(void) {return _Head?false:true;}
    virtual void Push(int V) = 0;
    virtual int  Pop(void);
    ~Cont();
};
//---------------------------------------------------------------------------
class CStack: public Cont {
public:
    CStack():Cont(){}
    virtual void Push(int V);
};
//---------------------------------------------------------------------------
class CQueue: public Cont {
public:
    CQueue():Cont(){}
    virtual void Push(int V);
};
//---------------------------------------------------------------------------
class Matrix {
protected:
  double ** _m;
  int _Size;
  virtual  void   Init(double Value);
public:
                  Matrix(int Size);
                  Matrix(const Matrix & M);
           double operator () (int row, int col);
           int    Size(void) {return _Size;}
           void   Set(int row, int col, double Value);
  virtual  void   Print(void);
           void   PrintFloydPath(int From, int To);
           void   PrintFloydPaths();
                  ~Matrix();
};
//---------------------------------------------------------------------------
class Graph: public Matrix {
  virtual void    Init(double Value);
public:
                  Graph(int Size):Matrix(Size){Init(MAXDOUBLE);}
                  Graph(const Graph & G):Matrix(G){}
  virtual void    Random(double Density, double MaxWeight = 1);
  virtual void    Print(void);
  virtual void    AddEdge(int V0, int V1, double Weight = 1);
  virtual void    DeleteEdge(int V0, int V1);
  virtual  int    EdgeCount(void);
           int    VertexCount(void) {return _Size;}
  virtual Graph   ShortestPath(int From, int To);
          void    Visit(int From, Cont & C);
          void    FindMinEdge(int & V0, int & V1);
          Graph   Floyd(Matrix & M);
          Graph   Kruskal(void);
          int     Hamiltonian(int v, int w, int Length, bool * Labelled, Graph & G);
          Graph   HamiltonianPath(int From, int To);
};
```

```
//---------------------------------------------------------------------------
class SGraph: public Graph {
public:
                  SGraph(int Size):Graph(Size){}
                  SGraph(const SGraph & G): Graph(G){}
  virtual void    Random(double Density, double MaxWeight = 1);
  virtual  int    EdgeCount(void) {return Graph::EdgeCount()/2;};
  virtual void    AddEdge(int V0, int V1, double Weight = 1);
};
//---------------------------------------------------------------------------
class WGraph: public Graph {
public:
                  WGraph(int Size):Graph(Size){}
                  WGraph(const WGraph & G): Graph(G){}
  virtual  int    EdgeCount(void) {return Graph::EdgeCount()/2;};
  virtual void    AddEdge(int V0, int V1, double Weight = 1);
};
//---------------------------------------------------------------------------
class OrGraph: public Graph {
public:
                  OrGraph(int Size):Graph(Size){}
                  OrGraph(const OrGraph & G): Graph(G){}
  virtual void    Random(double Density, double MaxWeight = 1);
  virtual void    AddEdge(int V0, int V1, double Weight = 1);
          OrGraph TransClosure(bool PathOrder);

};
//---------------------------------------------------------------------------
class OrWGraph: public Graph {
public:
                  OrWGraph(int Size):Graph(Size){}
                  OrWGraph(const OrWGraph & G): Graph(G){}
  virtual void    Random(double Density, double MaxWeight = 1);
};
//---------------------------------------------------------------------------
struct Deikstra {
bool   Label;
double Path;
int    Vertex;
  Deikstra():Label(false),Path(MAXDOUBLE),Vertex(-1){}
};
//---------------------------------------------------------------------------
int Cont::Pop(void)
{
if(_Head)
  {
  Item * I = _Head;
  _Head = _Head->Next;
  int V = I->D;
  delete I;
  if (!_Head) _Tail = 0;
  return V;
  }
return -1;
}
//---------------------------------------------------------------------------
    Cont::~Cont()
{
while(_Head)
  {
  Item * I = _Head->Next;
  delete _Head;
  _Head = I;
  }
}
//---------------------------------------------------------------------------
void CStack::Push(int V)
{
if(_Head)
  {
  Item * I = new Item(V);
  I->Next = _Head;
  _Head = I;
```

```
  }
else
  _Tail = _Head = new Item(V);
}
//---------------------------------------------------------------------------
void CQueue::Push(int V)
{
if(_Head)
  {
  _Tail->Next = new Item(V);
  _Tail = _Tail->Next;
  }
else
  _Tail = _Head = new Item(V);
}
//---------------------------------------------------------------------------
       Matrix::Matrix(int Size):_Size(0),_m(0)
{
if(Size > 0)
  {
  _Size = Size;
  _m = new double*[_Size];
  for(int i = 0; i < _Size; i++)
    _m[i] = new double [_Size];
  Init(0);
  }
}
//---------------------------------------------------------------------------
       Matrix::Matrix(const Matrix & M)
{
if (&M != this)
  {
  _Size = 0;
  _m = 0;
  if(M._Size > 0)
    {
    _Size = M._Size;
    _m = new double*[_Size];
    for(int i = 0; i < _Size; i++)
      _m[i] = new double [_Size];
    }
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    _m[i][j] = M._m[i][j];
  }
}
//---------------------------------------------------------------------------
void   Matrix::Init(double Value)
{
for(int i = 0; i < _Size; i++)
for(int j = 0; j < _Size; j++)
  _m[i][j] = Value;
}
//---------------------------------------------------------------------------
double Matrix::operator () (int row, int col)
{
if(row < _Size && col < _Size && row >= 0 && col >= 0)
  return _m[row][col];
return MAXDOUBLE;
}
//---------------------------------------------------------------------------
void   Matrix::Set(int row, int col, double Value)
{
if(row < _Size && col < _Size && row >= 0 && col >= 0)
  _m[row][col] = Value;
}
//---------------------------------------------------------------------------
void   Matrix::Print(void)
{
for(int i = 0; i < _Size; i++)
  {
  for(int j = 0; j < _Size; j++)
    printf("%10.2lf",_m[i][j]);
```

```
  printf("\n");
  }
}
//--------------------------------------------------------------------------
void   Matrix::PrintFloydPath(int From, int To)
{
if (From != To)
  {
  printf("Path from %d to %d :  ",From,To);
  int v = From;
  printf("%d ",v);
  do
    {
    v = _m[v][To];
    if (v == -1)
      {
      printf("--> |x|");
      break;
      }
    printf("--> %d ",v);
    }
  while(v != To);
  printf("\n");
  }
}
//--------------------------------------------------------------------------
void   Matrix::PrintFloydPaths()
{
for (int v = 0; v < _Size; v++)
  for (int w = 0; w < _Size; w++)
     PrintFloydPath(v,w);
printf("\n");
}
//--------------------------------------------------------------------------
       Matrix::~Matrix()
{
for(int i = 0; i < _Size; i++)
  delete [] _m[i];
delete [] _m;
}
//--------------------------------------------------------------------------
//--------------------------------------------------------------------------
void   Graph::Init(double Value)
{
for(int i = 0; i < _Size; i++)
for(int j = 0; j < _Size; j++)
  if(i != j) _m[i][j] = Value;
}
//--------------------------------------------------------------------------
void    Graph::AddEdge(int V0, int V1, double Weight)
{
if(V0 >= 0 && V1 >= 0 && V0 < _Size && V1 < _Size && V0 != V1 && Weight >= 0)
  {
  _m[V0][V1] = Weight;
  }
}
//--------------------------------------------------------------------------
void    Graph::DeleteEdge(int V0, int V1)
{
if(V0 >= 0 && V1 >= 0 && V0 < _Size && V1 < _Size && V0 != V1)
  {
  _m[V0][V1] = MAXDOUBLE;
  }
}
//--------------------------------------------------------------------------
void   Graph::Random(double Density, double MaxWeight)
{
if (Density >= 0 && Density <= 1 && MaxWeight >= 0)
  {
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    if(i > j)
      {
```

```
      if(Density >= (double)rand()/RAND_MAX)
        _m[i][j] = MaxWeight*rand()/RAND_MAX;
      else
        _m[i][j] = MAXDOUBLE;
      _m[j][i] = _m[i][j];
      }
    else
      continue;
  }
}
//---------------------------------------------------------------------------
void   Graph::Print(void)
{
for(int i = 0; i < _Size; i++)
  {
  for(int j = 0; j < _Size; j++)
    if(_m[i][j] < MAXDOUBLE)
      printf("%7.1lf",_m[i][j]);
    else
      printf("     oo");
  printf("\n");
  }
printf("\n");
}
//---------------------------------------------------------------------------
Graph Graph::ShortestPath(int From, int To)
{
if (From >= 0 && From < _Size && To < _Size && To >= 0)
  {
  if(From == To) return Graph(0);
  Deikstra * D = new Deikstra[_Size];
  D[From].Path   = 0;
  D[From].Vertex = -1;
  int U, V = From;
  double SPath;
  do
    {
    D[V].Label = true;
    for(U = 0; U < _Size; U++)
      {
      if(D[U].Label || _m[V][U] == MAXDOUBLE) continue;
      if(D[U].Path > D[V].Path + _m[V][U])
        {
        D[U].Path   = D[V].Path + _m[V][U];
        D[U].Vertex = V;
        }
      }
    SPath = MAXDOUBLE;
    V = -1;
    for(U = 0; U < _Size; U++)
      if(!(D[U].Label) && D[U].Path < SPath)
         {
         SPath = D[U].Path;
         V = U;
         }
    }
  while(V != -1 && V != To);
  Graph G(_Size);
  V = To;
  while(D[V].Vertex != -1)
    {
    G.AddEdge(V,D[V].Vertex,_m[V][D[V].Vertex]);
    V = D[V].Vertex;
    }
  delete [] D;
  return G;
  }
return Graph(0);
}
//---------------------------------------------------------------------------
int    Graph::EdgeCount(void)
{
if(_Size)
```

```
  {
  int Count = 0;
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    {
    if(i == j) continue;
    if(_m[i][j] < MAXDOUBLE) Count++;
    }
  return Count;
  }
return 0;
}
//---------------------------------------------------------------------------
Graph   Graph::Floyd(Matrix & M)
{
Graph G(*this);
if(M.Size() == _Size)
  {
  for(int i = 0; i < _Size; i++)
    for(int s = 0; s < _Size; s++)
      M.Set(i, s, G._m[i][s] == MAXDOUBLE ? -1 : s);
  for(int i = 0; i < _Size; i++)
    for(int s = 0; s < _Size; s++)
      if (G._m[s][i] < MAXDOUBLE)
        for(int t = 0; t < _Size; t++)
          if (G._m[i][t] < MAXDOUBLE)
            if(G._m[s][t] > G._m[s][i] + G._m[i][t])
               {
               G._m[s][t] = G._m[s][i] + G._m[i][t];
               M.Set(s, t, M(s, i));
               }
  }
return G;
}
//---------------------------------------------------------------------------
void    Graph::FindMinEdge(int & V0, int & V1)
{
double MinWeight = MAXDOUBLE;
V0 = V1 = -1;
for(int i = 0; i < _Size; i++)
  for(int s = 0; s < _Size; s++)
    {
    if (i == s)  continue;
    if (_m[i][s] < MAXDOUBLE)
      {
      MinWeight = _m[i][s];
      V0        = i;
      V1        = s;
      }
    }
if (MinWeight == MAXDOUBLE)
  V0 = V1 = -1;
}
//---------------------------------------------------------------------------
Graph   Graph::Kruskal(void)
{
Graph MST(_Size);
Graph G(*this);
bool * Labelled = new bool[_Size];
for(int i = 0; i < _Size; i++) Labelled[i] = false;
int V0 = -1, V1 = -1;
while (MST.EdgeCount() < (_Size - 1)*2)
  {
  G.FindMinEdge(V0,V1);
  if(V0 == -1 && V1 == -1)
    break;
  if(!Labelled[V0] || !Labelled[V1])
    {
    MST.AddEdge(V0,V1,_m[V0][V1]);
    MST.AddEdge(V1,V0,_m[V0][V1]);
    Labelled[V0] = true;
    Labelled[V1] = true;
    }
```

```
  G.DeleteEdge(V0,V1);
  G.DeleteEdge(V1,V0);
  }
delete [] Labelled;
return MST;
}
//---------------------------------------------------------------------------
void    Graph::Visit(int From, Cont & C)
{
if (From >= 0 && From < _Size)
  {
  int V = From;
  bool * Labelled = new bool[_Size];
  for(int i = 0; i < _Size; i++)
    Labelled[i] = false;
  C.Push(V);
  Labelled[V] = true;
  do
    {
    V = C.Pop();
    printf("%d ", V);
    for(int U = 0; U < _Size; U++)
      {
      if(!Labelled[U] && _m[V][U] != MAXDOUBLE)
        {
        C.Push(U);
        Labelled[U] = true;
        }
      }
    }
  while(!C.Empty());
  delete [] Labelled;
  printf("\n");
  }
}
//---------------------------------------------------------------------------
int  Graph::Hamiltonian(int v, int w, int Length, bool * Labelled, Graph & G)
{
if (v == w)
  return Length == 0 ? 1 : 0;
Labelled[v] = true;
for (int t = 0; t < _Size; t++)
  if( _m[v][t] < MAXDOUBLE && _m[v][t] != 0 && !Labelled[t])
    if ( Hamiltonian(t, w, Length - 1, Labelled, G))
      {
      G.AddEdge(t,v,_m[t][v]);
      G.AddEdge(v,t,_m[v][t]);
      return 1;
      }
Labelled[v] = false;
return 0;
}
//---------------------------------------------------------------------------
Graph   Graph::HamiltonianPath(int From, int To)
{
bool * Labelled = new bool[_Size];
for (int i = 0; i < _Size; i++) Labelled[i] = false;
Graph G(_Size);
int f = Hamiltonian(From,To, _Size - 1 ,Labelled, G);
delete [] Labelled;
return G;
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void    SGraph::AddEdge(int V0, int V1, double Weight)
{
Graph::AddEdge(V0,V1,1);
Graph::AddEdge(V1,V0,1);
}
//---------------------------------------------------------------------------
void   SGraph::Random(double Density, double MaxWeight)
{
if (Density >= 0 && Density <= 1 && MaxWeight >= 0)
```

```
  {
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    if(i > j)
      {
      if(Density >= (double)rand()/RAND_MAX)
        _m[i][j] = 1.0;
      else
        _m[i][j] = MAXDOUBLE;
      _m[j][i] = _m[i][j];
      }
    else
      continue;
  }
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void    WGraph::AddEdge(int V0, int V1, double Weight)
{
Graph::AddEdge(V0,V1,Weight);
Graph::AddEdge(V1,V0,Weight);
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void    OrGraph::AddEdge(int V0, int V1, double Weight)
{
Graph::AddEdge(V0,V1,1);
}
//---------------------------------------------------------------------------
OrGraph OrGraph::TransClosure(bool PathOrder)
{
OrGraph G(*this);
for(int i = 0; i < _Size; i++)
  for(int s = 0; s < _Size; s++)
     if (G._m[s][i] == 1)
       for(int t = 0; t < _Size; t++)
         G._m[s][t] = G._m[i][t] == 1 ? 1 : G._m[s][t];
if(!PathOrder)
  for(int i = 0; i < _Size; i++)
    G._m[i][i] = 0;
return G;
}
//---------------------------------------------------------------------------
void   OrGraph::Random(double Density, double MaxWeight)
{
if (Density >= 0 && Density <= 1 && MaxWeight >= 0)
  {
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    {
    if(i == j) continue;
    if(Density >= (double)rand()/RAND_MAX)
      _m[i][j] = 1.0;
    else
      _m[i][j] = MAXDOUBLE;
    }
  }
}
//---------------------------------------------------------------------------
//---------------------------------------------------------------------------
void   OrWGraph::Random(double Density, double MaxWeight)
{
if (Density >= 0 && Density <= 1 && MaxWeight >= 0)
  {
  for(int i = 0; i < _Size; i++)
  for(int j = 0; j < _Size; j++)
    {
    if(i == j) continue;
    if(Density >= (double)rand()/RAND_MAX)
      _m[i][j] = MaxWeight*rand()/RAND_MAX;
    else
      _m[i][j] = MAXDOUBLE;
    }
```

```
  }
}
#pragma argsused
//---------------------------------------------------------------------------
int main(int argc, char* argv[])
{
Graph A(5);
A.Random(1, 1);
//Matrix M(A.VertexCount());
//Graph  F(A.Floyd(M));
//Graph  G(A.Kruskal());
Graph H(A.HamiltonianPath(0,1));
A.Print();
H.Print();
//F.Print();
//M.PrintFloydPaths();
//M.Print();
//G.Print();
getch();
return 0;
}
```