МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

СВЕДЕНИЕ ЗАДАЧИ ОБ УЗЕЛЬНОМ ПОКРЫТИИ К ЗАДАЧЕ О МНОЖЕСТВЕ РЁБЕР, РАЗРЕЗАЮЩИХ ЦИКЛЫ

ОТЧЕТ

студента 3 курса 331 группы
специальности 10.05.01 — Компьютерная безопасность
факультета КНиИТ
Бородина Артёма Горовича

Проверил
к. ф.-м. н. ______________ А. Н. Гамова

Саратов 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной лабораторной работы служит рассмотрение вопроса сводимости задачи о вершинном покрытии к задаче о нахождении множества рёбер, разрезающих циклы. Для сведённой задачи требуется осуществить программную реализацию с эффективной (полиномиальной) временной сложностью.

Решение вопроса сводимости

Теорема. Задача об узельном покрытии полиномиально трансформируема (сводима) в задачу о множестве рёбер, разрезающих циклы. Поэтому задача о множестве рёбер, разрезающих циклы, NP-полна.

Доказательство. Пусть $G = (V, E)$ – неориентированный граф, а $D = (V \times \{0, 1\},\ E')$ – ориентированный граф, где E' состоит из

1) $[v, 0] \to [v, 1]$ для каждого $v \in V$ и

2) $[v, 1] \to [w, 0]$ и $[w, 1] \to [v, 0]$ для каждого неориентированного ребра $(v, w) \in E$.

Пусть $F \subset E'$ – множество ребер графа D, содержащих по крайней мере одно ребро из каждого цикла в D. Заменим каждое ребро из F, имеющее вид $[v, 1] \to [w, 0]$, ребром $[w, 0] \to [w, 1]$. Полученное множество обозначим через F' . Утверждается, что $|F'| \leq |F|$ и F' содержит по крайней мере одно ребро из каждого цикла (единственное ребро, выходящее из $[w, 0]$, идёт в $[w, 1]$, так что $[w, 0] \to [w, 1]$ принадлежит любому циклу, содержащему $[v, 1] \to [w, 0]$).

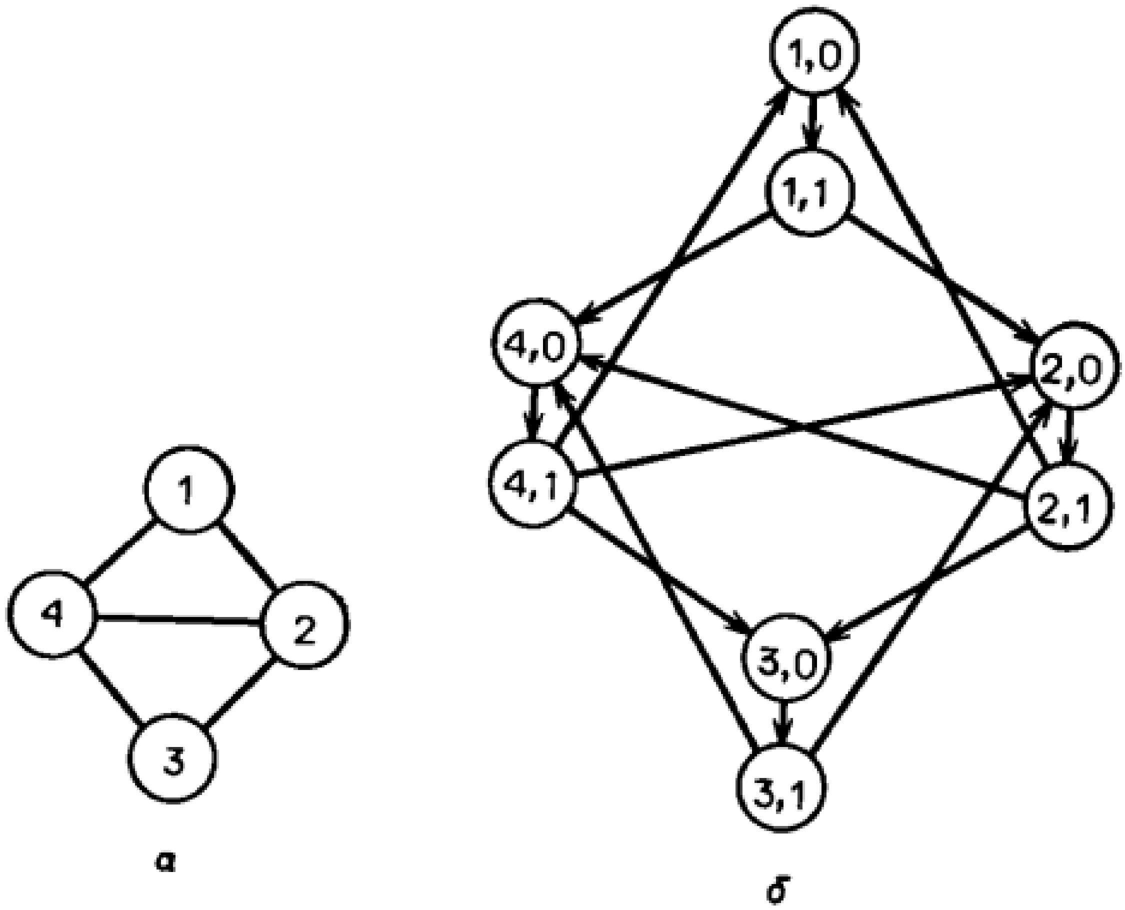

Рисунок 1 – **а** – неориентированный граф G; **б** – соответствующий ориентированный граф D. Узельное покрытие 2, 4 соответствует множеству [2, 0] $\rightarrow$ [2, 1], [4, 0] $\rightarrow$ [4, 1] рёбер, разрезающих циклы.

Ради общности, будем считать, что

$$F' = \{[v_i, 0] \rightarrow [v_i, 1] \mid 1 \leq i \leq k\}$$

для некоторого k. Тогда каждый цикл в D содержит ребро $[v_i, 0] \rightarrow [v_i, 1]$ для некоторого $i,\ 1 \leq i \leq k$. Заметим, что если (x, y) – произвольное ребро в G, то $[x, 1], [y, 0], [y, 1], [x, 0], [x, 1]$ образуют цикл в D. Поэтому каждое ребро в G инцидентно некоторому узлу $v_i,\ 1 \leq i \leq k$, и, значит, $v_1, \ldots, v_k$ – узельное покрытие графа G.

Обратно, можно показать, что если S – узельное покрытие размера k, то $\{[v, 0] \rightarrow [v, 1] \mid v \in S\}$ – множество ребер, разрезающих циклы в D. Чтобы узнать, содержит ли G узельное покрытие размера k, надо построить за полиномиальное время граф D и выяснить, есть ли в нем k-элементное множество ребер, разрезающих циклы. $\Box$

Программная реализация

Сначала приведём обоснование алгоритма. Пусть есть ориентированный граф G и множество его вершин упорядочено: $v_1, v_2, \ldots, v_n$. Будем называть такую перестановку *последовательностью вершин* и обозначать $s = v_1 v_2 \ldots v_n$. Каждая последовательность вершин s порождает набор рёбер, разрезающих циклы $R(s)$, состоящий из всех дуг вида $v_j v_i$, $j > i$. Более того, каждому разрезающему циклы набору рёбер соответствует некоторая последовательность вершин. Таким образом, задача поиска множества рёбер, разрезающих циклы эквивалентна поиску такой последовательности вершин s, что $R(s) = R(G)$.

Алгоритм GR. Алгоритм жадным образом удаляет из G вершины (и инцидентные им рёбра), которые являются стоками или истоками или удовлетворяют следующему свойству: пусть для всякой вершины $u \in V$ $d(u)$ обозначает степень u, $d^+(u)$ – степень исхода, а $d^-(u)$ – степень входа. Тогда $d(u) = d^+(u) + d^-(u)$. На каждом шаге работы алгоритма после удаления стоков и истоков удаляется единственная вершина, для которой $\delta(u) = d^+(u) - d^-(u)$ максимально. Если удалённая вершина u является стоком, то она добавляется в последовательность вершин s_2; иначе, u добавляется в последовательность вершин s_1. Алгоритм работает до тех пор, пока из графа не будут удалены все его вершины. Результирующая последовательность вершин получается как результат конкатенации последовательностей s_1 и s_2.

```
procedure GR (G : DiGraph; var s : VertexSequence);
s₁ ← ∅; s₂ ← ∅
while G ≠ ∅ do
  {while G содержит сток do
       {выбрать сток u; s₂ ← us₂; G ← G − u};
   while G содержит исток do
       {выбрать исток u; s₁ ← s₁u; G ← G − u};
   выбрать вершину u, для которой δ(u) максимально;
   if (u - сток) s₂ ← us₂
   else s₁ ← s₁u;
   G ← G − u;}
s ← s₁s₂.
```

Листинг 1: Алгоритм GR.

По заданному псевдокоду реализуем программу:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int numberOfVertices, numberOfEdges;
vector<vector<int>> graph;
vector<int> vertices;

void displayGraphMatrix (vector<vector<int>> matrix)
{
  int i, j;

  for (i = 0; i < matrix.size (); ++i)
  {
    for (j = 0; j < matrix[i].size (); ++j)
    cout << matrix[i][j] << " ";
    cout << "\n";
  }
}

void getGraphMatrix ()
{
  int i, u, v;

  cout << "Input the number
           of vertices and the number of edges: ";
  cin >> numberOfVertices >> numberOfEdges;
  graph.assign (numberOfVertices,
                vector<int> (numberOfVertices, 0));

  cout << "Now input your edges:\n";
  for (i = 0; i < numberOfEdges; ++i)
  {
    cin >> u >> v;
    graph[--u][--v] = 1;
  }
}

```

```
bool inGraph (int vertex)
{
  int sumRow = 0, sumColumn = 0, i,
      negSize = graph.size () - 2 * graph.size ();

  for (i = 0; i < graph.size (); ++i)
    sumRow += graph[vertex][i], sumColumn += graph[i][vertex];

  return (!((sumRow == sumColumn) && (sumColumn == negSize)));
}

int extractSink (vector<vector<int>> matrix)
{
  int i, j;
  bool has_sink;

  for (i = 0; i < matrix.size (); ++i)
  {
    has_sink = true;
    for (j = 0; j < matrix[i].size (); ++j)
      if (matrix[i][j] == 1)
      {
        has_sink = false;
        break;
      }
    if (has_sink && inGraph (i))
      return (i);
  }
  return (-1);
}

void reduceMatrix (int vertex)
{
  int i;

  for (i = 0; i < graph.size (); ++i)
    graph[vertex][i] = graph[i][vertex] = -1;
}

int extractSource (vector<vector<int>> matrix)
{
```

```
  int i, j;
  bool has_source;

  for (i = 0; i < matrix.size (); ++i)
  {
    has_source = true;
    for (j = 0; j < matrix[i].size (); ++j)
    if (matrix[j][i] == 1)
    {
      has_source = false;
      break;
    }
    if (has_source && inGraph (i))
    return (i);
  }
  return (-1);
}

int getDelta (int vertex)
{
  int outdegree = 0, indegree = 0, i;

  for (i = 0; i < graph.size (); ++i)
  {
    if (graph[vertex][i] == 1)
    ++outdegree;
    if (graph[i][vertex] == 1 && vertex != i)
    ++indegree;
  }
  return (outdegree - indegree);
}

int getMaxDeltaVertex ()
{
  int maxDeltaVertex = 0, maxDelta = getDelta (0), i;

  for (i = 1; i < graph.size (); ++i)
  {
    int curDelta = getDelta (i);
    if (curDelta >= maxDelta && inGraph (i))
    maxDelta = curDelta, maxDeltaVertex = i;
```

```
  }
  return (maxDeltaVertex);
}

int adjacencySum ()
{
  int sum = 0, i, j;

  for (i = 0; i < graph.size (); ++i)
  for (j = 0; j < graph[i].size (); ++j)
  sum += graph[i][j];

  return (sum);
}

void feedbackArcSetProblem ()
{
  vector<int> seqOne, seqTwo;
  int vertexPlaceholder;
  getGraphMatrix ();
  int grSize = graph.size ();
  int negGrSizeSq = -(grSize * grSize);

  while (adjacencySum () != negGrSizeSq)
  {
    while ((vertexPlaceholder = extractSink (graph)) != -1)
    {
      seqTwo.push_back (vertexPlaceholder);
      reduceMatrix (vertexPlaceholder);
    }
    if (adjacencySum () == negGrSizeSq)
    break;

    while ((vertexPlaceholder = extractSource (graph)) != -1)
    {
      seqOne.push_back (vertexPlaceholder);
      reduceMatrix (vertexPlaceholder);
    }
    if (adjacencySum () == negGrSizeSq)
      break;

```

```
    vertexPlaceholder = getMaxDeltaVertex ();
    seqOne.push_back (vertexPlaceholder);
    reduceMatrix (vertexPlaceholder);
  }
  reverse (seqTwo.begin (), seqTwo.end ());

  cout << "\nThis is the first sequence (s_1): ";
  for (int vertex : seqOne)
    cout << ++vertex << " ";
  cout << "\n";
  cout << "This is the second sequence (s_2): ";
  for (int vertex : seqTwo)
    cout << ++vertex << " ";
  cout << "\n";
  cout << "This is the required sequence (s = s_1 + s_2): ";
  for (int vertex : seqOne)
    cout << ++vertex << " ";
  for (int vertex : seqTwo)
    cout << ++vertex << " ";
  cout << "\n";

  for (int i = 0; i < copyGraph.size (); ++i)
    for (int j = 0; j < copyGraph[i].size (); ++j)
    if (copyGraph[i][j] == 1)
      if (vertices[i] > vertices[j])
        cuttingEdges.push_back (make_pair (i, j));
  cout << "Cutting edges are: ";
  for (int i = 0; i < cuttingEdges.size (); ++i)
    cout << ++cuttingEdges[i].first << "-" <<
            ++cuttingEdges[i].second <<
            (i == cuttingEdges.size () - 1 ? "\n" : ", ");

}

int main ()
{
  feedbackArcSetProblem ();

  return (EXIT_SUCCESS);
}
```

Листинг 2: Программная реализация алгоритма GR.

Примеры запуска

Рассмотрим результат работы программы на некоторых входных данных. Пусть задан следующий граф:

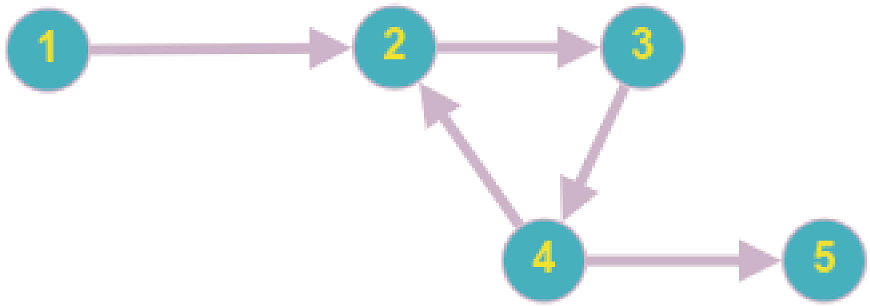

Рисунок 2 – Некоторый граф, содержащий цикл.

Результатом работы программы должна быть последовательность вершин, позволяющая восстановить набор рёбер, разрезающих циклы.

Результатом работы программы на этих входных данных будет являться следующая последовательность вершин и соответствующий ей набор рёбер, разрезающих циклы.

```
Input the number of vertices and the number of edges: 5 5
Now input your edges:
1 2
2 3
3 4
4 2
4 5

This is the first sequence (s_1): 1 4
This is the second sequence (s_2): 2 3 5
This is the required sequence (s = s_1 + s_2): 1 4 2 3 5
Cutting edges are: 2-3
```

Рисунок 3 – Полученная последовательность вершин и набор рёбер.

Теперь рассмотрим граф с большим числом циклов:

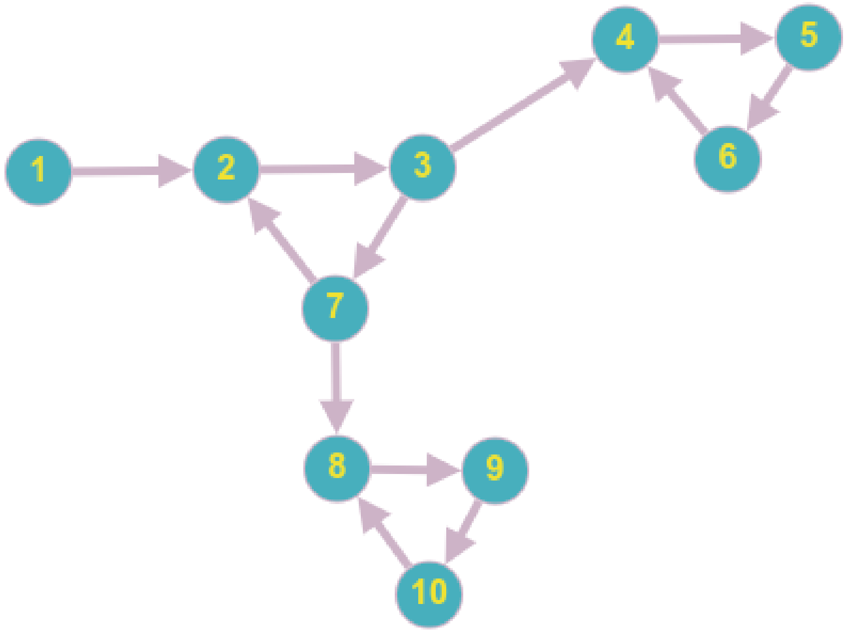

Рисунок 4 – Некоторый граф, содержащий циклы.

Результатом работы программы на этих входных данных будет являться следующая последовательность вершин и соответствующий ей набор рёбер, разрезающих циклы.

```
Input the number of vertices and the number of edges: 10 12
Now input your edges:
1 2
2 3
3 7
7 2
3 4
4 5
5 6
6 4
7 8
8 9
9 10
10 8

This is the first sequence (s_1): 1 7 2 3 10 6
This is the second sequence (s_2): 4 5 8 9
This is the required sequence (s = s_1 + s_2): 1 7 2 3 10 6 4 5 8 9
Cutting edges are: 2-3, 5-6, 6-4, 10-8
```

Рисунок 5 – Полученная последовательность вершин и набор рёбер.

Видно, что в набор рёбер, разрезающих циклы, попало два ребра из одного цикла. Возникающие в процессе вычисления неточности являются следствием использования приближённого алгоритма.

Полученные в результате применения приближённого алгоритма ациклические подграфы исходного графа могут отличаться от максимального ациклического подграфа не более, чем в два раза.

Оценка сложности алгоритма

Оценим временную сложность полученного приближённого алгоритма. Будем оценивать временную сложность каждого шага, проделываемого алгоритмом:

Шаг 1.1. Поиск и выбор стока в графе. Рассматриваемый блок кода:

```
while ((vertexPlaceholder = extractSink (graph)) != -1)
```

Листинг 3: Блок кода, отвечающий за поиск стока.

Стоком в графе называется вершина, не имеющая выходящих из неё рёбер. В случае представления графа матрицей это означает, что строка этой вершины в матрице смежности содержит только нули (поскольку граф невзвешенный, то единица в матрице смежности означает наличие между вершинами ребра, а ноль – его отсутствие). В худшем случае для поиска стока необходимо осуществить полный проход по матрице, что указывает на временную сложность $O(n^2)$, где n – число вершин в графе.

Шаг 1.2. Редукция матрицы. Рассматриваемый блок кода:

```
reduceMatrix (vertexPlaceholder);, расположенный внутри
while ((vertexPlaceholder = extractSink (graph)) != -1)
```

Листинг 4: Блок кода, отвечающий за редукцию матрицы.

После нахождения стока необходимо обновить матрицу смежности графа так, чтобы найденная вершина не использовалась в дальнейших вычислениях. Это достигается путём помещения значения -1 в строку и столбец найденного стока. Общее число операций для первого найденного стока составляет $2n - 1$ и, следовательно, оценка сложности – $O(n)$.

Шаг 1.3. Промежуточная проверка графа на пустоту. Рассматриваемый блок кода:

```
if (adjacencySum () == negGrSizeSq)
  break;
```

Листинг 5: Блок кода, отвечающий за поиск стока.

В данной реализации граф считается пустым, если каждая ячейка его матрицы смежности содержит -1. Для вычисления необходимого значения нужно пройти каждую ячейку матрицы и добавить значение, хранящееся в ней, в аккумулятор. Временная сложность – $O(n^2)$.

Шаг 2.1. Поиск и выбор истока в графе. Рассматриваемый блок кода:

```
while ((vertexPlaceholder = extractSource (graph)) != -1)
```

Листинг 6: Блок кода, отвечающий за поиск стока.

Истоком в графе называется вершина, не имеющая входящих в неё рёбер. В случае представления графа матрицей это означает, что столбец этой вершины в матрице смежности содержит только нули. Дальнейшие рассуждения аналогичны полученным на шаге 1.1. Временная сложность – $O(n^2)$.

Шаг 2.2. Редукция матрицы. Аналогично шагу 1.2.

Шаг 2.3. Промежуточная проверка графа на пустоту. Аналогично шагу 1.3.

Шаг 3.1. Получение вершины с оптимальным δ. Рассматриваемый блок кода:

```
vertexPlaceholder = getMaxDeltaVertex ();
```

Листинг 7: Блок кода, отвечающий за поиск стока.

Осуществляется проходом по непустой части графа. Для каждой вершины просматривается соответствующие этой вершине столбец и строка. В худшем случае для каждой из n вершин необходимо просмотреть $2n - 1$ значений ячеек её столбца и строки. Временная сложность – $O(2n^2 - n) = O(n^2)$.

Шаг 3.2. Редукция матрицы. Аналогично шагам 1.2 и 2.2.

Получаем, что общая временная сложность внутренней части цикла равна $O(n^3 + n^3 + n^2 + n^2) = O(n^3)$. Однако сама проверка выполнения условия цикла требует выполнения n^2 операций, из чего следует, что временная сложность приближённого алгоритма составляет $O(n^5)$ – полиномиальная временная сложность.

ЗАКЛЮЧЕНИЕ

В ходе данной лабораторной работы был рассмотрен вопрос о сведении задачи о вершинном покрытии к задаче о нахождении множества рёбер, разрезающих циклы. Был рассмотрен псевдокод приближённого алгоритма и осуществлена его программная реализация. Также была дана оценка временной сложности приближённого алгоритма.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Eades, P., Lin, X. and Smyth, W.F. (1993) A fast and effective heuristic for the feedback arc set problem. Information Processing Letters, 47 (6). pp. 319-323.