САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

КАФЕДРА МОДЕЛИРОВАНИЯ ЭЛЕКТРОМЕХАНИЧЕСКИХ И КОМПЬЮТЕРНЫХ СИСТЕМ

Павлов Илья Николаевич

Выпускная квалификационная работа бакалавра

Алгоритмы и программное обеспечение моделирования и расчетов систем с помощью теории графов

Направление 010400

Прикладная математика и информатика

Научный руководитель,
доктор техн. наук,
профессор
Карпов А. Г.

Санкт-Петербург

2017

Оглавление

Введение

Актуальность работы

Теория графов является мощным инструментом моделирования и расчета систем различной физической природы. При этом применение методов теории графов встречает некоторые трудности.

Во-первых, отсутствует прямой метод определения физической реализуемости системы заданной графом, а именно выполнения фундаментальных законов сохранения последовательной переменной (закона сохранения импульса в механике и закона сохранения заряда в электричестве). В теории физическая реализуемость системы, заданной графом, отражается в так называемой гамильтоновости графа. Граф является гамильтоновым, если он обладает гамильтоновым циклом (или контуром). Гамильтоновым называется цикл графа, проходящий через каждую его вершину без повторения вершин (кроме, быть может, совпадающих концевых вершин) и ребер. Гамильтонов цикл не обязательно содержит все ребра графа. Очевидно, что гамильтоновым может быть только связный граф. К сожалению, в отличие от графов Эйлера, где имеется конкретный критерий для графа быть эйлеровым, для гамильтоновых графов подобного критерия нет. Более того, задача проверки существования гамильтонова цикла оказывается NP-полной. Большинство известных утверждений имеет вид: «если граф имеет достаточное количество ребер, то граф является гамильтоновым». Таковыми являются теорема Дирака, теорема Оре [1], условие Поша [1], условие Бонди-Хватала [2]. Задача отыскания гамильтонова цикла или эквивалентная задача коммивояжера являются практически востребованными, но для нее неизвестен (и, вероятно, не существует) эффективный общий алгоритм решения.

Во-вторых, известные методики построения математических моделей с использованием, например, системы уравнений независимых контуров, на практике используют ручные операции и не могут быть применены для сложных систем.

Таким образом, тема квалификационной выпускной работы является, несомненно, актуальной.

Цель работы

Целью работы является создание алгоритмического и программного обеспечения для получения математической модели системы, заданной графом. Исходной информацией служит матрица смежности **A** связного ориентированного графа G с множеством вершин $V = \{0, ..., m-1\}$. В ходе построения математической модели необходимо определить физическую реализуемость системы, заданной графом.

Постановка задачи

Анализ цели работы показал, что для ее достижения необходимо решить следующую совокупность взаимосвязанных и взаимодополняющих задач:

1. Разработка и реализация алгоритма построения дерева графа по матрице смежности.

2. Разработка и реализация алгоритма построения совокупности независимых контуров.

3. Разработка и реализация метода и соответствующего алгоритма определения физической реализуемости системы, заданной графом.

4. Разработка и реализация алгоритма построения системы уравнений независимых контуров.

Путем объединения решений поставленных задач будет получен алгоритм, результатом работы которого будет математическая модель системы в виде системы уравнений независимых контуров. В дальнейшем на основе системы уравнений независимых контуров может быть построена система уравнений с учетом характеристик элементов физической системы, заданной графом [3].

Структуры данных

Высокая скорость работы любого алгоритма достигается за счет оптимизации операций, которые выполняются на каждом шаге. Кроме того, для повышения производительности необходимо уделить отдельное внимание структурам данных, которыми оперирует алгоритм. В данной работе основными структурами данных являются матрицы(двумерные массивы), массивы и списки. Массивы лучше использовать тогда, когда структура данных остается неизменной во время работы алгоритма, а доступ к

определенному элементу необходимо выполнять достаточно быстро. Списки же будем использовать в том случае, если структура данных во время работы алгоритма изменяется динамически, например, если добавляются новые элементы. При этом скорость доступа к элементу списка, отличного от начального и последнего, не представляет интереса. Стоит отметить, что в данной работе практически везде список может быть заменен на стек, так как в основном списки используются для вставки элементов в конец списка и удаления элементов из конца списка. Операциями над списками будут следующие: операция вставки элементов в конец списка($push$), операция извлечения элемента из конца списка(pop), операция вставки элементов в начало списка($unshift$) и операция удаления элементов из начала списка($shift$). Так как список является мультимножеством, сохраняющим порядок вхождения элементов, то для упрощения описания алгоритмов будем использовать язык теории множеств. Например, для инициализации списка S будем использовать запись вида $S := \emptyset$, которая означает создание пустого списка, где $|S|$ - количество элементов в списке. Так же будем использовать кванторы теории множеств при выборке элементов из массива или списка. Например, выражение $R := \{\forall u \in V : color[u] = WHITE, a_{vu} \neq 0\}$ следует понимать так: «выбрать все элементы u из множества V, для которых выполняется слудующее: $color[u] = WHITE$ и $a_{vu} \neq 0$; результат сохранить в список R». Стоит отметить, что если R не было определено на этапе инициализации алгоритма, то подобная запись используется для компактности, чтобы в одном выражении не было слишком много условий. Это означает, что при программной реализации не обязательно создавать список R, а достаточно задать условие $u \in V \wedge color[u] = WHITE \wedge a_{vu} \neq 0$ в условном операторе. Структуры данных будем обозначать прописными латинскими буквами. Для определенного элемента будем использовать соответствующие строчные буквы с индексами. Например, если список обозначается S, то его i-тый элемент s_i. Для правого конца списка S будем использовать обозначение s_{top}. Матрицы будем обозначать жирными буквами. Элементы матрицы $\mathbf{A}$, обозначаются через $a_{i,j}$. Для обозначения i-й строки матрицы $\mathbf{A}$ будем использовать $a_{i,*}$, а для j-го столбца $a_{*,j}$.

Глава 1. Построение дерева графа

Для построения дерева графа необходим алгоритм обхода графа. Однако большинство известных алгоритмов, выполняющих указанную процедуру, работают рекурсивно, что при программной реализации снижает производительность, а при больших графах приводит к переполнению стека вызовов функций [4]. В данной работе предлагается алгоритм на основе нерекурсивного поиска в глубину [2]. Дополнительно этот алгоритм выполняет операции, необходимые для дальнейших этапов решения поставленной задачи.

1.1. Алгоритм

Рассмотрим алгоритм построения дерева графа $Tree(\mathbf{A}_{m\times m}, \mathbf{D}, B, C)$. Входным данными для него являются матрица смежности $\mathbf{A}_{m\times m}$ связного графа G; нулевая матрица $\mathbf{D}_{m\times m}$, в которой в результате работы алгоритма будет находиться матрица смежности дерева графа; в изначально пустых списках B и C в результате работы алгоритма будут находиться ветви и хорды графа соответственно. Приведем сам алгоритм:

1. Инициализация.
   (a) $\mathbf{D} := \mathbf{G}$,
   (b) $B := C := \emptyset$,
   (c) $v := 0$,
   (d) $color[v] := RED$,
   (e) $push\ S,\ v$.
2. Основной цикл.
   (a) $|S| > 0$? Перейти к шагу 3; иначе перейти к шагу 6.
3. Необходимые действия
   (a) $v := s_{top}$,
   (b) $pop\ S$,

(c) $color[v] := BLACK$.

4. Добавление хорд.

   (a) $R := \{\forall u \in V : color[u] = RED, a_{v,u} \neq 0\}$,

   (b) $\forall u \in R \Rightarrow push\ C, (v, u)$, если $a_{v,u} > 0$; иначе $push\ C,\ (u, v)$,

   (c) $\forall u \in R \Rightarrow d_{u,v} := d_{v,u} := 0$.

5. Добавление ветвей.

   (a) $R := \{\forall u \in V : color[u] = WHITE, a_{v,u} \neq 0\}$,

   (b) $push\ S, R$,

   (c) $\forall u \in R \Rightarrow push\ B,\ (u, v)$,

   (d) $\forall u \in R \Rightarrow color[u] := RED$,

   (e) Перейти к шагу 2.

6. Конец.

   (a) Закончить выполнение алгоритма. B, C — списки хорд и ветвей графа G. $\mathbf{D}$ — матрица смежности дерева графа G.

1.2. Обсуждение алгоритма

Данный алгоритм основан на нерекурсивном поиске в глубину, поэтому скорость его работы пропорциональна $O(m + n)$, где m — число вершин, а n — число ребер. Основным отличием предложенного алгоритма от приведенного в [2] является то, что используются дополнительные красная окраска вершин и список (стек) для хранения вершин, с которых необходимо начать следующий проход алгоритма. Цвет каждой вершины хранится в массиве *color* в ячейке, номер которой совпадает с номером вершины. По умолчанию в начале работы алгоритма все вершины окрашены в белый цвет. Построение дерева всегда начинается с нулевой вершины. Во время работы алгоритма необходимо определить, какое ребро будет хордой, а какое ветвью. Условие добавления ребра в множество ветвей заключается в том, что одна из его вершин окрашена в белый цвет, а другая в черный. После идентификации ветви ее белая вершина окрашивается в красный цвет и сохраняется в списке S. Определение хорды основано на том, что если у красной вершины есть смежная ей красная вершина, то эти две вершины имеют общую вершину-предка, а ребро, которое соединяет эти вершины, замыкает простой цикл. Другими словами, они принадлежат одному контуру, поэтому соединяющее их ребро будет хордой. Стоит отметить, что поскольку граф G ориентированный, то добавляемое в список

хорд или список ветвей ребро (u, v) отличается от ребра (v, u) ориентацией. Для удобства дальнейших действий (см. главу 4) в список C добавляются положительно-ориентированные хорды.

1.3. Пример работы алгоритма

Рассмотрим работу алгоритма построения деррева графа на примере электрической цепи, изображенной на рис. 1 *a*. Для того, чтобы построить граф этой цепи необходимо каждый элемент цепи заменить на ребро, а узлы соединения элементов на вершины. На рис. 1 *b* изображен граф G электрической цепи на рис. 1 *a*.

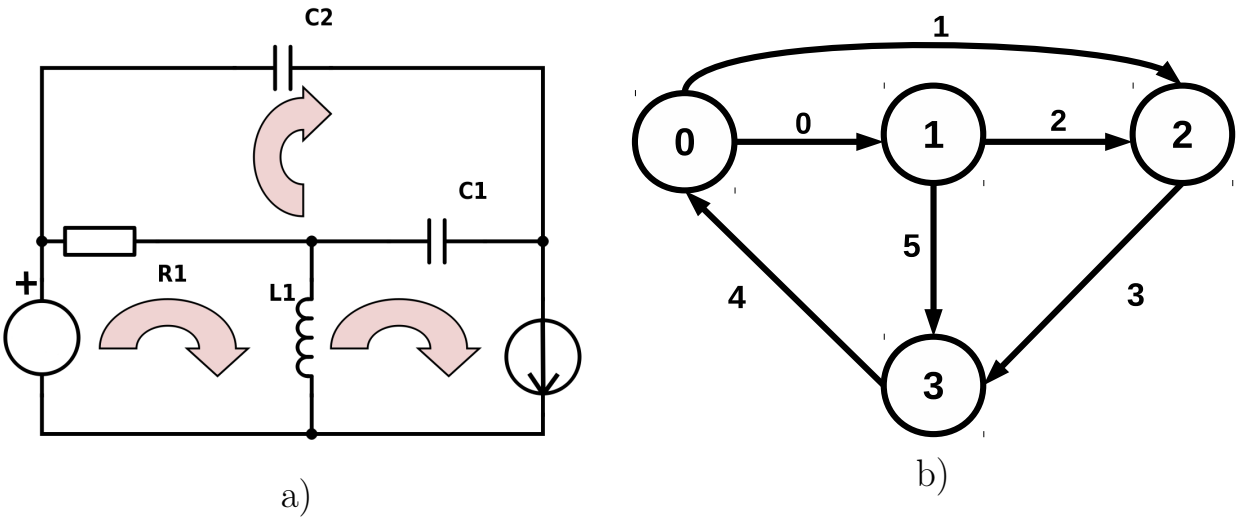

Рис. 1: Электрическая цепь и характерезующий ее граф.

Составим матрицу смежности $\mathbf{A}$ графа G.

$$\mathbf{A} = \begin{pmatrix} 0 & -1 & -1 & 1 \\ 1 & 0 & -1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & 1 & 1 & 0 \end{pmatrix}$$

Этапы работы алгоритма показаны на рисунке 2.

На рис. 1.2(a) изображен этап инициализации алгоритма. Вершина с номером 0 добавлена в список S и окрашена в красный цвет.

На рис. 1.2(b) показан результат первого прохода основного цикла алгоритма. Вершина с номером 0 перекрашена в черный цвет и удалена из списка S. На данном этапе состояние переменных алгоритма будет следующее: $S = \{1, 2, 3\}$, $B = \{(0, 1),\ (0, 2), (0, 3)\}$. Хорд на данном этапе найдено не было, так как на предыдущем этапе была всего одна красная вершина.

На рис. 1.2(c) показан результат второго прохода основного цикла алгоритма. Данный проход начинается с последней вершины списка S — 3. Новых вершин найдено не было, поэтому количество вершин списка S умень-

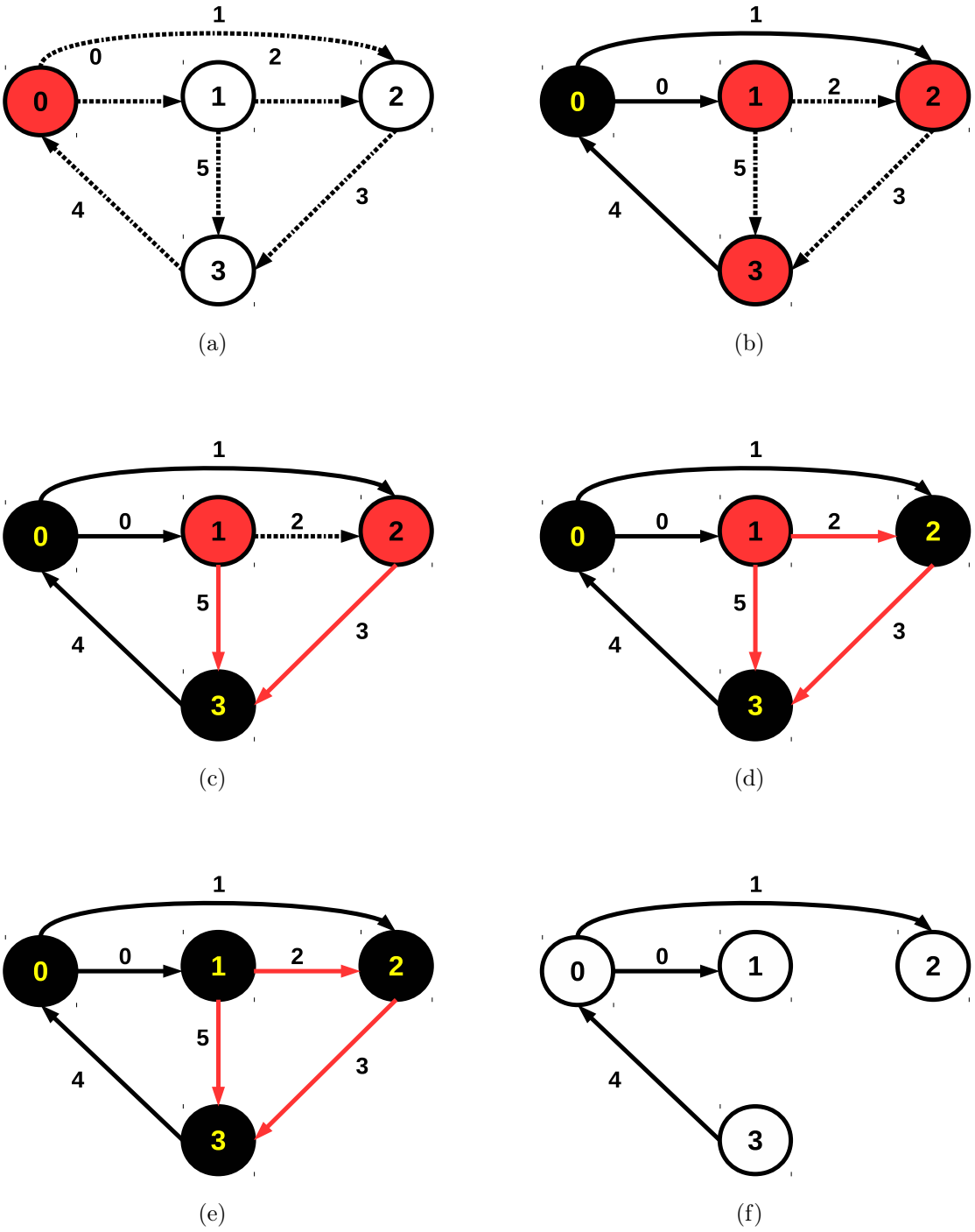

Рис. 2: Основные этапы работы алгоритма построения дерева графа.

шилось, а новых ветвей добавлено не было. На данном этапе состояние переменных алгоритма будет следующее: $S = \{1, 2, 3\}$, $B = \{(0, 1),\ (0, 2),\ (0, 3)\}$, $C = \{(1, 3),\ (2, 3)\}$.

На рис. 1.2(d) показан результат третьего прохода основного цикла алгоритма. Данный проход начинался с вершины под номером 2. Была найдена одна хорда. На данном этапе состояние переменных алгоритма будет следующее: $S = \{1\}$, $B = \{(0, 1),\ (0, 2),\ (0, 3)\}$, $C = \{(1, 3),\ (2, 3),\ (1, 2)\}$.

На рис. 1.2(e) показан результат четвертого завершающего прохода основного цикла алгоритма. Так как в списке S оставалась одна вершина с номером 1, а смежных белых вершин у нее не было, то данная верши-

на просто перекрашивается в черный цвет, и из-за отсутствия элементов в списке S программа перейдет к шагу 6 и завершит выполнение.

Построенное дерево графа G изображено на рис. 1.2(f), а его матрица смежности $\mathbf{D}$ после выполнения алгоритма будет следующей:

$$\mathbf{D} = \begin{pmatrix} 0 & -1 & -1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{pmatrix}$$

Глава 2. Построение независимого контура

Из определения отсечения следует, что построить систему уравнений отсечений можно только с помощью полного перебора всевозможных разбиений множества вершин дерева графа, что весьма трудоемко при компьютерной реализации. Иначе обстоит дело с построением независимых контуров и составлением соответствующей системы уравнений. Однако большинство из известных алгоритмов построения пути из одной вершины графа в другую работает со взвешенными неориентированными графами, что заставляет выполнять лишние операции с матрицей смежности графа. Кроме того, как было отмечено ранее, большинство таких алгоритмов работают рекурсивно. В данной работе предлагается алгоритм построения независимого контура, основанный на нерекурсивном поиске в глубину. По-существу этот алгоритм является модификацией алгоритма, приведенного в главе 1.

2.1. Алгоритм

Рассмотрим алгоритм построения независимого контура $IndependentLoop\ (\mathbf{D}_{m\times m}, v_{start}, v_{end}, S)$. Входной информацией для даннго алгоритм являются матрица смежности дерева $\mathbf{D}$ графа G; v_{start} и v_{end} — начальная и конечная вершины поиска. В списке S будет получена последовательность вершин, которые необходимо пройти в порядке их следования для построения независимого контура.

1. Инициализация.
   (a) $v := v_{start}$,
   (b) $S := \emptyset$,
   (c) $push\ S,\ v$,
   (d) $P := \emptyset$,
   (e) $color[v] := RED$.
2. Выделение «висячей» цепи.
   (a) $deg(v_{end}) <= 2$?

i. $unshift\ P,\ v_{end}$,
ii. $color[v_{end}] := BLACK$,
iii. $v_{end} = v_{start}$?
A. $S := P$,
B. Перейти к шагу 8.
iv. $v_{end} := u,\ u : a_{uv_{end}} \neq 0 \wedge color[u] = WHITE$,
v. Перейти к шагу 2(a).

(b) Перейти к шагу 3.

3. Начало цикла обработки текущей вершины.

(a) $color[v] \neq RED$? Если да, то перейти к шагу 4; если нет, перейти к шагу 5.

4. Удаление рассмотренной вершины.

(a) $v := s_{top}$,
(b) $pop\ S,\ v$,
(c) Перейти к шагу 3.

5. Пометить опорную вершину как рассмотренную.

(a) $color[v] := BLACK$,

6. Проверка достижения конечной вершины.

(a) $v = v_{end}$? Если да, то перейти к шагу 8;
иначе перейти к шагу 7.

7. Раскрасить опорные вершины.

(a) $R := \{\forall u \in V : color[u] = WHITE, a_{vu} \neq 0\}$,
(b) $v_{end} \in R$?
i. $color[v_{end}] := BLACK$,
ii. Перейти к шагу 8.
(c) $push\ S,\ R$,
(d) $\forall u \in R \Rightarrow color[u] := RED$,
(e) Перейти к шагу 3.

8. Выделение искомой последовательности вершин.

(a) $S := S \setminus \{u : u \in V, color[u] = RED\}$,

(b) $unshift\ S,\ P$,

9. Конец.

   (a) Окончание выполнение алгоритма. S — искомая последовательность вершин независимого контура.

2.2. Обсуждение алгоритма

Скорость работы данного алгоритма, как и алгоритма, рассмотренного в главе 2, пропорциональна $O(m+n)$, где m — число ребер, а n — число вершин. Данный алгоритм имеет два существенных отличия от обычного нерекурсивного поиска в глубину:

1. Текущая рассматриваемая опорная вершина не удаляется из стека S сразу, а только если выяснится, что она принадлежит пути, не приводящему в вершину v_{end} .

2. Введен дополнительный шаг выделения «висячей» цепи.

Первое необходимо для того, чтобы сохранять пройденный путь, который помечается черными вершинами. В стеке присутствуют и красные вершины, которые данному пути не принадлежат, поэтому их необходимо исключить из стека S. Стоит отметить, что выбор списка для хранения данных позволяет получать вершины в порядке их добавления.

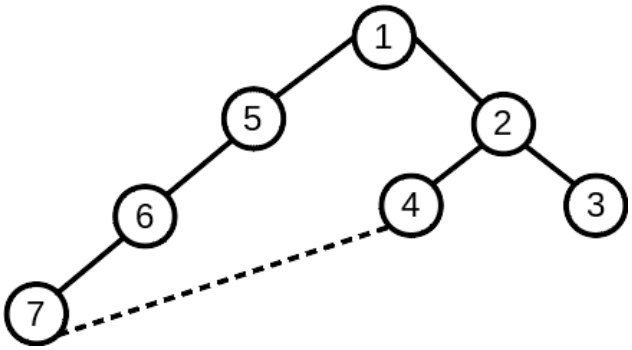

Рис. 3: Пример «висячей» цепи.

Шаг выделения «висячей» цепи является модификацией поиска в глубину. Ее идея основана на том, что если конечная вершина хорды является так называемой «висячей» вершиной, то указатель на конечную вершину можно переместить на смежную вершину. Более того, специфика, например, электрических цепей такова [5], что очень часто в дереве может существовать последовательное соединение ребер, каждая вершина которых имеет степень 2, кроме начальной и конечной вершин цепи. Для такого случая можно сместить указатель окончания поиска v_{end} на вершину начала

такой последовательности, предварительно запомнив последовательность вершин этой цепи. Так, если для графа на рис. 3 необходимо построить независимый контур для хорды $(4, 7)$, то вершины $2, 1, 5, 6, 7$ образуют висячую цепь, поэтому указатель на конечную вершину можно переместить с вершины 7 на вершину 2. В простейшем случае, когда каждая вершина дерева имеет степень 2, за исключением двух вершин степени 1, данная процедура перенесет указатель конечной вершины на начальную вершину, тем самым, построив независимый контур. Предложенная модификация может существенно ускорить поиск конечной вершины при большой «ветвистости» дерева (при высоких степенях вершин дерева). Лишь в самом неблагоприятном случае скорость алгоритма останется прежней.

2.3. Пример работы алгоритма

Для иллюстрации работа алгоритма воспользуемся результатами предыдущей главы. Построим независимый контур для хорды $(2, 3)$. Основные этапы работы приведены на рис. 4.

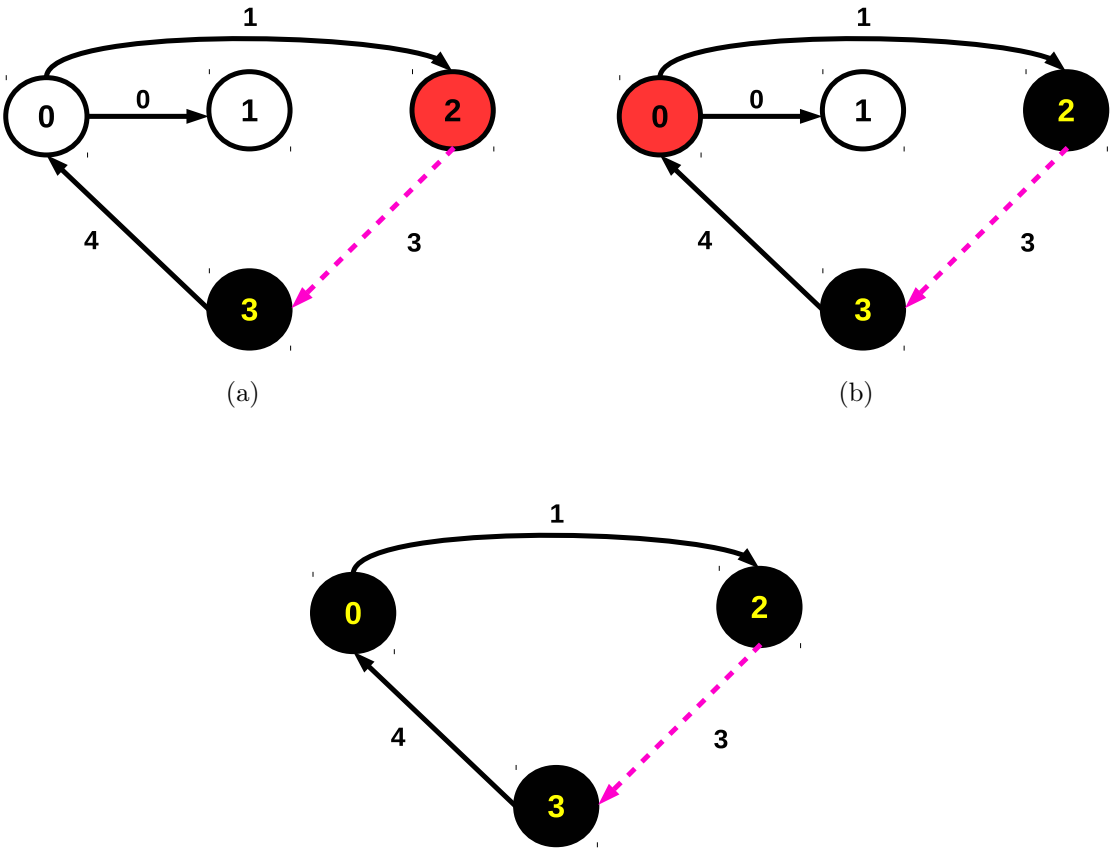

Рис. 4: Основные этапы работы алгоритма построения независимого контура.

На рис. 2.4(а) отображено состояние переменных алгоритма после ша-

гов инициализации и поиска «висячей» вершины. Так, была найдена висячая цепь из одной вершины (лист) под номером 3. Данная вершина помещена в список P, а указатель на конечную вершину поиска v_{end} перемещен на вершину с номером 0. Вершина 2 была окрашена в красный цвет и помещена в список S. После этого начинается выполнение основного цикла алгоритма.

На рис. 2.4(b) показан результат первого прохода основного цикла алгоритма. На данном этапе красным цветом помечается вершина с номером 0 и обнаруживается, что она является вершиной окончанием поиска. Вершина 0 окрашивается в черный цвет, добавляется в список S и алгоритм переходит к шагу 8. На данном этапе состояние переменных алгоритма будет следующее: $S = \{2, 0\}$, $P = \{3\}$.

На рис. 2.4(c) показан результат завершающих шагов алгоритма. После окончания работы алгоритма список S будет содержать значения $\{2, 0, 3\}$. Чтобы построить независимый контур нам необходимо пройти вершины из этого списка в порядке их следования. Не трудно убедиться, что для хорды $(1, 2)$ список S будет содержать значения $\{1, 0, 2\}$, а для хорды $(1, 3)$ $S = \{1, 0, 3\}$.

Глава 3. Проверка физической реализуемости системы, заданной графом

В этой части работы основная идея состоит в том, что физическая реализуемость системы проверяется не в начале выполнения программы, а в процессе построения системы уравнений независимых контуров, что позволяет сбалансировать время работы программы и результат ее выполнения. Например, может оказаться так, что физически система не имеет смысла, но программа все равно начнет построение независимых контуров — самую затратную процедуру. С другой стороны, если система действительно физически реализуема, то ожидание имеет смысл, но время ожидания будет значительно меньше.

3.1. Метод проверки физической реализуемости

Физическая реализуемость системы подразумевает выполнение фундаментальных законов сохранения последовательных переменных, что в теории графов эквивалентно гамильтоновости. После выполнения предыдущего алгоритма построены всевозможные независимые контуры. Условие того, что граф не является гамильтоновым, состоит в том, что в графе присутствует ветвь, которая не вошла ни в один независимый контур. Действительно, наличие подобной ветви означает незамкнутость системы. Другим условием негамильтоновости является наличие среди независимых контуров так называемых «петель». Под *петлей* будем понимать такой независимый контур, в котором каждая ветвь принадлежит только одному этому независимому контуру. Наличие петель противоречит условию гамильтоновсти графа. Следовательно, критерием физической реализуемости является отсутствие в графе петель и висячих цепей. Для реализации данного метода предлагается составить вектор (массив), каждая ячейка которого соответствует определенной ветви и содержит количество независимых контуров, в которых данная ветвь присутствует. Для этого надо пронумеровать все ребра графа. Для удобства проведения операций с ребрами полезно провести нумерацию ребер следующим образом: номера $0 \dots |B| - 1$ присваиваются ветвям, а номера $|B| \dots |C| - 1$ — хордам [3]. Нумерация ребер с нуля удобна для компьютерной реализации алгоритмов, поскольку в большинстве современных языках программирования нумерация масси-

вов начинается именно с нулевого индекса. Стоит отметить, что задача присвоения ребрам графа номеров является довольно «жадной» в плане потребления памяти, поскольку необходимо сохранить в памяти каждое ребро и соответствующий ему номер. В качестве возможной реализации такого хранилища предлагается создать ассоциативный массив (карту) M, где ключом будет ребро графа, а значением соответствующий ребру номер. Элементы карты M обозначим через m_{ij}, где i и j образуют ребро (i, j). Стоит отметить, что если существует ребро (i, j), то элемент карты m_{ij} будет совпадать с элементом m_{ji}. Поэтому для уменьшения потребления памяти в ассоциативном массиве M следует хранить только одну из этих пар, а для доступа к номеру ребра использовать следующую функцию:

$$getNum(i, j) = \begin{cases} m_{ij} & , i \leq j \\ m_{ji} & , i > j \end{cases}$$

Теперь в памяти будут храниться только ребра, у которых номер первой вершины не превосходит номер второй вершины. Однако если не планируется дальнийших операций с матрицей смежности графа, за исключением чтения, то наиболее экономным решением будет запись номера ребра в ячейку, находящуюся на пересечении соответствующих столбца и строки вершин ребра. Также необходимо присвоить номеру ребра знак находящегося до этого значения ячейки.

Для дальнейших действий необходимо последовательность вершин S перевести в последовательность ребер (пар вершин). Обозначим данную последовательность через $P := \{(S_{i-1}, S_i)\} \forall i \in \{1...(|S| - 1)\}$. При этом предполагается, что все независимые контуры сохранены в списке I.

3.2. Алгоритм

Алгоритм определения физической реализуемости графа состоит в следующем.

1. Инициализация.
   (a) $Y := (0, \ldots, 0)_{1 \times |B|}$
2. Подсчет вхождений ветвей в контуры
   (a) $\forall e \in P \Rightarrow Y[getNum(e)] := Y[getNum(e)] + 1.$
3. Проверка количества вхождений для каждого контура.
   (a) $\forall i \in I \Rightarrow$ перейти к шагу 4 если $sum(i) \geq |i|$,
   (b) Перейти к шагу 5.

4. Аварийная остановка

    (a) Закончить выполнение программы. Система физически не реализуема.

5. Успешное завершение.

    (a) Закончить выполнение программы. Система физически реализуема.

Глава 4. Составление системы уравнений независимых контуров

После проведения операций, описанных выше, остается составить систему уравнений независимых контуров.

4.1. Составление уравнения независимого контура

После построения дерева графа, представленного в главе 1, имеются множество ветвей и множество хорд H графа G. Элементами обоих множеств являются ребра графа. После работы предыдущего алгоритма получена последовательность вершин S, которые необходимо пройти в порядке их следования для построения независимого контура. Уравнение независимого контура получается с помощью ребер графа, поэтому необходимо последовательность вершин S перевести в последовательность ребер (пар вершин). Обозначим данную последовательность через $P := \{(S_{i-1}, S_i)\} \forall i \in \{1...(|S|-1)\}$ Для составления уравнения независимого контура необходимо, чтобы все ребра графа были пронумерованы, для этого используются положения и операции, описанные в предыдущей главе. Для составления уравнения независимого контура необходимо коэффициенту переменной, соответствующей ребру, присвоить 1, если ребро ориентировано по направлению обхода построенного пути независимого контура. Если ребро ориентировано противоположно направлению обхода, то коэффициент будет равняться -1. Если ребро не входит в указанный путь, то его коэффициент равен нулю. Стоит отметить, что при разбиении графа G на множества ветвей и хорд, описанном в главе 1, хорды добавлялись в список C исключительно с положительной ориентацией. Это используется для определения направления обхода всех вершин списка $|S|$. Рассмотрим алгоритм составления уравнения независимого контура $ILoopCoef(S, K)$, где K — массив коэффициентов переменных уравнения независимого контура, в котором номер ячейки соответствует номеру ребра графа, коэффициент переменной которого хранится в этой ячейке. Обозначим через o ориентацию текущего ребра. Алгоритм составления уравнения независимого контура состоит в следующем:

1. Инициализация.
   (a) $o := 1$,
   (b) $i := 1$.
2. Цикл.
   (a) $a_{S[i-1],S[i]} \neq o$?
      i. $o := -o$,
   (b) $K[getNum(S[i-1], S[i])] := o$,
   (c) $i \neq |S| - 1$?
      i. $i := i + 1$,
      ii. Перейти к шагу 2(a).
3. Окончание работы алгоритма.
   (a) K — массив коэффициентов уравнения независимого контура.

4.2. Составление системы уравнений независимых контуров

Для составления системы уравнений независимых контуров надо последовательно составить уравнения для каждого полученного независимого контура (для каждой хорды). Перейдем к алгоритму, реализующему весь процесс построения системы уравнений независимых контуров. Построение независимого контура и составление соответствующего уравнения - процессы, последовательно выполняющиеся для каждой хорды, поэтому будет логичным их объединить в одну процедуру. При этом алгоритмы построения независимого контура и составления соответствующего уравнения выполняются последовательно для каждой индивидуальной хорды. Поэтому их можно было бы объединить в одну процедуру, которая сначала выполняет $IndependentLoop$, а потом $ILoopCoef$. Кроме того, при программной реализации всего комплекса алгоритмов при наличии многопроцессорной системы под эту процедуру можно выделить собственный процесс для повышения производительности работы программы. Итак, процедура, назовем ее $IndependentLoopEquation$, в качестве исходных данных принимает матрицу смежности $\mathbf{D}_{m\times m}$ дерева графа G, вершины $v_{start}v_{end}$, инцидентные хорде, в массивK будут записаны коэффициенты уравнения независимого контура. Алгоритм процедуры будет выглядеть следующим образом:

1. Инициализация
   (a) $S := \emptyset$.

(b) $K = (0, \ldots, 0)_{1 \times m}$

2. Построение независимого контура.

   (a) $IndependentLoop(\mathbf{D}, v_{start}, v_{end}, S)$,

3. Составление уравнения независимого контура.

   (a) $ILoopCoef(S, K)$,

4. Окончание работы процедуры.

   (a) K — массив коэффициентов уравнения независимого контура.

Тогда полный алгоритм построения системы уравнений независимых контуров будет следующим:

1. Инициализация.

   (a) $\mathbf{D} := \mathbb{O}_{m \times m}$,
   (b) $B := C := \emptyset$.

2. Построение дерева графа.

   (a) $Tree(\mathbf{D}, B, C)$.

3. Создание матрицы коэффициентов системы уравнений независимых контуров.

   (a) $\mathbf{Z} := \mathbb{O}_{|B| \times m}$.
   (b) $i := 0$

4. Составление системы уравнений независимых контуров.

   (a) Если $i \geq |C|$, то перейти к шагу 5.
   (b) $v_{start} := C_{i_{start}}, v_{end} := C_{i_{end}}$,
   (c) $IndependentLoopEquation(\mathbf{D}, v_{start}, v_{end}, z_{i,*}, m)$,
   (d) $i := i + 1$,
   (e) Перейти к шагу 4(a).

5. Окончание работы алгоритма.

   (a) Z — матрица коэффициентов системы уравнений независимых контуров.

4.3. Пример составления системы уравнений независимых контуров

Составим систему уравнений независимых контуров для примера из главы 1. Для начала необходимо перенумеровать ребра графа в соответствии с условием, описанным выше. Вариант новой нумерации приведен на рис. 5.

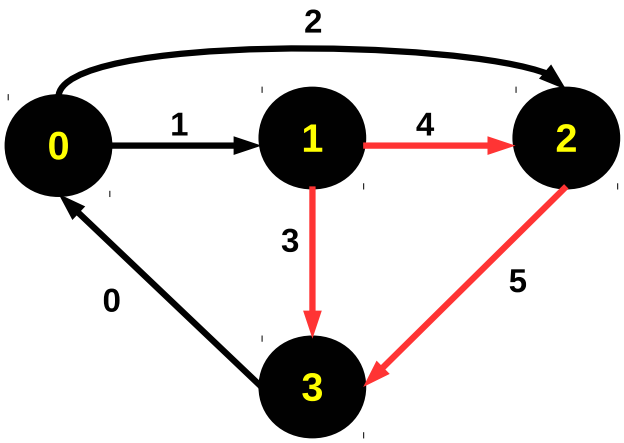

Рис. 5: Новая нумерация ребер.

Матрица коэффициентов для этого графа будет выглядеть следующим образом:

	0	1	2	3	4	5
3	1	1	0	1	0	0
4	0	1	−1	0	1	0
5	0	0	0	−1	1	1

А соответствующая система уравнений независимых контуров примет вид:

$$\begin{cases} v_0 + v_1 + v_3 = 0 \\ v_1 - v_2 + v_4 = 0 \\ v_4 - v_3 + v_5 = 0 \end{cases}$$

Глава 5. Заключение

В результе выполнения квалификационной выпускной работы были предложены алгоритмы и созданно программное обеспечение, позволяющие решать следующие задачи:

1. Построение дерева графа по матрице смежности.
2. Построение совокупности независимых контуров.
3. Определение физической реализуемости системы, заданной графом.
4. Построение системы уравнений независимых контуров.

Программное обеспечение было разработано на языке C++. Данная программа работает в многопоточном режиме, что при больших графах позволяет ускорить вычисления. Программа была протестирована на компьютере с процессором Intel Core i7-4770 3.9GHz и оперативной памятью размером 8192 Gb для полного орграфа с 5000 вершинами. Построение системы уравнений независимых контуров для соответствующего числа потоков происходило за следующее время:

1. 217795278 мс.
2. 201436595 мс.
3. 205971920 мс.
4. 197777720 мс.

Литература

[1] Харари Ф. Теория Графов. М.: Мир, 1973, 300 с.

[2] Свами М., Тхуласираман К. Графы, сети и алгоритмы. М.: Мир, 1984. 455 с.

[3] Львович А. Ю. Основы электромеханических систем. Л.: Изд-во Ленингр. ун-та, 1973. 196 с.

[4] Кормен Томас Х., Лейзерсон Чарльз И., Ривест Рональд Л., Штайн Клиффорд. Алгоритмы: построение и анализ, 2-е изд. М.: Издательский дом "Вильямс" , 2005. 1296 с.

[5] Корис Р., Шмидт-Вальтер Х. Справочник инженера-схемотехника. М.: Техносфера, 2008. 608 с.

Приложение

```
/*
Copyright 2017 Pavlov Ilja

Redistribution and use in source and binary forms, with or without
    modification, are permitted provided that the following conditions are met
    :

1. Redistributions of source code must retain the above copyright notice, this
     list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
    this list of conditions and the following disclaimer in the documentation
    and/or other materials provided with the distribution.

3. Neither the name of the copyright holder nor the names of its contributors
    may be used to endorse or promote products derived from this software
    without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
    AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
    IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
     ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
     LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
    CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
    SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
    INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
    CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
    ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
     POSSIBILITY OF SUCH DAMAGE.
*/
#include "graph_methods.h"
#include <cstdio>
#include <cstdlib>
#include <list>
#include <vector>
#include <utility>
#include <iterator>
#include <iostream>
#include <omp.h>
#include <ctime>

using namespace std;

template <typename T> int sgn(T val) { return (T(0) < val) - (val < T(0)); }

int main(int argc, char* argv[]) {
    unsigned int start_time =  clock();
    int** adjacency_matrix;
    int vertex_count;
```

```
    read_matrix("big_matrix.txt", &adjacency_matrix, &vertex_count);

    list<int> loop, edgesLoop;
    vector< pair<int, int> > chords;
    int** tree;
    int edgesCount;
    dfs_not_recursive(0, adjacency_matrix, &tree, vertex_count, chords,
   edgesCount);


    vector< list<int> > loops(chords.size());

#pragma omp parallel for shared(loops, tree, adjacency_matrix) private(loop,
   edgesLoop)
    for(int i = 0; i < chords.size(); i++){
        get_closed_loop(chords[i].first, chords[i].second, tree, vertex_count,
    loop);

        int orientation = sgn( adjacency_matrix[chords[i].first][chords[i].
   second] );
        edgesLoop.push_back(getNum(chords[i].first, chords[i].second,
   adjacency_matrix));

        list<int>::iterator it = loop.begin();
        for (it++; it != loop.end(); it++){
            int prev = *(--it);
            int cur  = *(++it);
            if ( sgn( adjacency_matrix[prev][cur] ) != orientation) {
                orientation *= -1;
            }
            edgesLoop.push_back(getNum(prev, cur, adjacency_matrix)*
   orientation);
        }

        loops[i] = edgesLoop;
        edgesLoop.clear();
        loop.clear();
    }
    unsigned int end_time = clock();
    unsigned int search_time = end_time - start_time;
    cout << search_time << endl;

    return 0;
}
```

./src/demo.cpp

```
#ifndef GRAPH_METHODS_H
#define GRAPH_METHODS_H
#include <list>
#include <utility>
#include <vector>

using namespace std;

void read_matrix(char* filename, int** matrix[], int* vertex_count);

void dfs_not_recursive(int vertex, int** adjacency_matrix, int** tree_matrix
   [], int vertex_count,
        vector< pair<int, int> > &chords, int& edgesCount);
```

```
void get_closed_loop(int start_vertex, int end_vertex, int**
    adjacency_matrix_tree, int vertex_count,
        list<int> &loop);

bool checkPhysicalFeasibility(const list< pair<int, int> > &branchs,
        const vector< list< pair<int, int> > > &loops, int** adjacency_matrix)
    ;

unsigned int getNum(int a, int b, int** adjacency_matrix_tree);

#endif
```

./src/graph_methods.h

```
#include <cstdio>
#include <cstdlib>
#include <utility>
#include <iterator>
#include <stack>
#include <vector>
#include <iostream>
#include <cmath>
#include "graph_methods.h"

using namespace std;

/*format file:
vertex_count
row(0)
...
row(vertex_count-1)
 */
enum {
    WHITE, GREY, BLACK, RED
};

void read_matrix(char* filename, int**matrix[], int* vertex_count) {
    FILE* in = fopen(filename, "rt");
    fscanf(in, "%d", vertex_count);
    int v = *vertex_count;
    *matrix = (int**) malloc(v * sizeof (int*));
    for (int i = 0; i < *vertex_count; i++) {
        (*matrix)[i] = (int*) malloc(v * sizeof (int));
        for (int j = 0; j < v; j++) {
            fscanf(in, "%d", &((*matrix)[i][j]));
        }
    }

    fclose(in);
}

void dfs_not_recursive(int vertex, int** adjacency_matrix, int** tree_matrix
    [], int vertex_count,
        vector< pair<int, int> > &chords, int& edgesCount){

    *tree_matrix = (int**) malloc(vertex_count * sizeof (int*));
    for (int i = 0; i < vertex_count; i++){
        (*tree_matrix)[i] = (int*) calloc(vertex_count, sizeof (int));
    }
```

```
    int* vertex_color = (int*) calloc(vertex_count, sizeof (int));
    unsigned int enumerator = 1;
    stack<int> grey_vertexes;
    grey_vertexes.push(vertex);
    vertex_color[vertex] = GREY;
    edgesCount = 0;

    while (!grey_vertexes.empty()) {
        vertex = grey_vertexes.top();
        grey_vertexes.pop();
        vertex_color[vertex] = BLACK;
        for (int i = 0; i < vertex_count; i++) {
            if (adjacency_matrix[vertex][i] != 0) {
                if (vertex_color[i] == WHITE) {
                    vertex_color[i] = GREY;
                    grey_vertexes.push(i);
                    (*tree_matrix)[vertex][i] = adjacency_matrix[vertex][i];
                    (*tree_matrix)[i][vertex] = adjacency_matrix[i][vertex];

                    adjacency_matrix[vertex][i] *= enumerator;
                    adjacency_matrix[i][vertex] *= enumerator++;

                    edgesCount++;

                } else if (vertex_color[i] != BLACK) {
                    if (adjacency_matrix[vertex][i] < 0) {
                        chords.push_back(make_pair(vertex, i));
                    } else {
                        chords.push_back(make_pair(i, vertex));
                    }
                    edgesCount++;
                }
            }
        }
    }

    for (pair<int, int> chord : chords){
        adjacency_matrix[chord.first][chord.second] *= enumerator;
        adjacency_matrix[chord.second][chord.first] *= enumerator++;
    }

    free(vertex_color);
}

void get_closed_loop(int start_vertex, int end_vertex, int**
    adjacency_matrix_tree, int vertex_count, list<int> &loop) {
    int* vertex_colors = (int*) calloc(vertex_count, sizeof (int));
    list<int> black_vertexes;
    int vertex = start_vertex;
    black_vertexes.push_back(vertex);
    vertex_colors[vertex] = RED;

    loop.push_back(end_vertex);
    while (true) {
        int count_adjacency_vertex = 0;
        int last_adjacency_vertex;
        for (int i = 0; i < vertex_count; i++) {
            if (adjacency_matrix_tree[end_vertex][i]) {
                count_adjacency_vertex++;
```

```
                last_adjacency_vertex = i;
            }
        }
        if (count_adjacency_vertex > 1) {
            break;
        }
        loop.push_front(last_adjacency_vertex);
        end_vertex = last_adjacency_vertex;
        if (end_vertex == start_vertex) {
            free(vertex_colors);
            return;
        }
    }


    while (true) {
        while (!black_vertexes.empty() && vertex_colors[vertex =
    black_vertexes.back()] != RED) {
            black_vertexes.pop_back();
        }
        vertex_colors[vertex] = BLACK;

        for (int i = 0; i < vertex_count; i++) {
            if (adjacency_matrix_tree[vertex][i] != 0 && vertex_colors[i] ==
    WHITE) {
                if (i == end_vertex){
                    black_vertexes.remove_if([&vertex_colors](int v){return
    vertex_colors[v] == RED;});
                    loop.splice(loop.begin(), black_vertexes);
                    black_vertexes.clear();
                    free(vertex_colors);
                    return;
                }
                black_vertexes.push_back(i);
                vertex_colors[i] = RED;

            }
        }
    }
}

bool checkPhysicalFeasibility(const vector< pair<int, int> > &branchs,
        const vector< list< pair<int, int> > > &loops,
        int** adjacency_matrix){
    int* occurences = (int*) calloc(branchs.size(), sizeof (int));
    int emptyBranchs = branchs.size();
    for (list< pair<int, int> > loop : loops){
        for (list< pair<int, int> >::iterator edge = loop.begin(); edge != --
    loop.end(); edge++){
            int index = getNum((*edge).first, (*edge).second, adjacency_matrix
    );
            if (!occurences[ index ]){
                emptyBranchs--;
            }
            occurences[ index ] += 1;
        }
    }

    if (emptyBranchs){
        free(occurences);
```

```
            return false;
        }

        if (loops.size() > 1){
            cout << loops.size();
            for (list< pair<int, int> > loop : loops){
                bool ok = true;
                for (list< pair<int, int> >::iterator edge = loop.begin(); edge !=
        --loop.end() && ok; edge++){
                    if (occurences[ getNum((*edge).first, (*edge).second,
        adjacency_matrix) ] > 1){
                        ok = false;
                    }
                }
                free(occurences);
                return false;
            }
        }
        free(occurences);
        return true;
    }

    unsigned int getNum(int a, int b, int** adjacency_matrix){
        return abs(adjacency_matrix[a][b])-1;
    }
```

./src/graph_methods.cpp