МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное автономное образовательное учреждение высшего образования
«Дальневосточный федеральный университет»
(ДВФУ)

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент математического и компьютерного моделирования

РЕФЕРАТ
о практическом задание по дисциплине АИСД
«Алгоритм сжатия цветков»

направление подготовки 09.03.03 «Прикладная информатика»
профиль «Прикладная информатика в компьютерном дизайне»

Выполнил студент
гр. Б9121-09.03.03пикд
Морокова Валерия Александровна
_____
(подпись)
Руководитель практики
Доцент ИМКТ А.С Кленин
(должность, уч. звание)
_____
(подпись)
«_____» _____ 2022г.

Доклад защищен:
С оценкой _____

Рег. № _____

«____» _____ 2023 г.

г. Владивосток
2023

Оглавление

Глоссарий

Граф — математическая система, объекты которой обладают парными связями.

Паросочетание — набор несмежных ребер.

Вершина(узел) — структурная единица графа.

Ребро — соединяет вершины графа.

Свободная вершина — вершина графа, не покрытая паросочетанием.

Не инцидентные ребра — отношение между рёбрами, в котором существует соединяющая их вершина.

Чередующаяся цепь — путь в графе, в котором для любых двух соседних рёбер верно, что одно из них принадлежит паросочетанию M, а другое нет.

Дополняющий(увеличивающий) путь — чередующаяся цепь, которая начинается и кончается свободными вершинами.

Цветок(соцветие/бутон) — нечетный цикл графа.

Стебель — чередующаяся цепь чётной длины.

База — вершина графа, которая принадлежит стеблю и является частью цикла.

Введение

Историческая справка

Алгоритмы поиска максимальных совпадений достаточно сложны. О первом эффективном подходе сообщил Джек Эдмондс в 1960-х годах, что стало важной вехой в истории компьютерных наук. Его «Blossom algorithm» вдохновил на вариации и альтернативы за последние несколько десятилетий. Джек Эдмондс разработал алгоритм в 1961 году и опубликовал в 1965 году.

Постановка задачи

Задача разделяется на следующие пункты:

1. Подобрать и изучить источники по теме: «Алгоритм сжатия цветков».
2. Описать алгоритм «Сжатия цветков» в форме научного доклада.
3. Реализовать алгоритм «Сжатия цветков».
4. Продумать тесты с наибольшим покрытием кода алгоритма.
5. Провести тестирование алгоритма.
6. Сделать выводы о проделанной работе.

Описание алгоритма

Суть и назначение

Алгоритм сжатия цветков (англ. Blossom algorithm) — это алгоритм в теории графов для построения наибольших паросочетаний на графах.

Если дан граф $G = (V, E)$ общего вида, алгоритм находит паросочетание M такое, что каждая вершина из V инцидентна не более чем одному ребру из M и $|M|$ максимально. Паросочетание строится путем итеративного улучшения начального пустого паросочетания вдоль увеличивающих путей графа.

В отличие от двудольного паросочетания ключевой новой идеей было сжатие нечетного цикла в графе (цветка) в одну вершину с продолжением поиска итеративно по сжатому графу.

Основной причиной, почему алгоритм сжатия цветков важен, является то, что он дал первое доказательство возможности нахождения наибольшего паросочетания за полиномиальное время.

Другой причиной является то, что метод приводит к описанию многогранника линейного программирования для многогранника паросочетаний, что приводит к алгоритму паросочетания минимального веса.

Оценка сложности

Пусть n — общее количество вершин, n_1 — количество цветков, n_2 — количество вершин в цветке, m — количество ребер.

Всего имеется n итераций, на каждой из которых выполняется обход в ширину за $O(m)$, кроме того, могут происходить операции сжатия цветков — их может быть $O(n_1)$. Сжатие соцветий работает за $O(n_2)$, стоит отметить $n_1 \equiv n_2$, то есть общая асимптотика алгоритма составит $O(n(m + n^2)) = O(n^3)$.

Нахождение дополняющего(увеличивающего) пути

Пусть зафиксировано некоторое паросочетание M. Тогда простая цепь $P = (v_1, v_2, ..., v_k)$ называется чередующейся цепью, если в ней рёбра по очереди принадлежат — не принадлежат паросочетанию M. Чередующаяся цепь называется увеличивающей, если её первая и последняя вершины не принадлежат паросочетанию. Иными словами, простая цепь P является увеличивающей тогда и только тогда, когда вершина $v_1 \notin M$, ребро $(v_2, v_3) \in M$, ребро $(v_4, v_5) \in M$, ..., ребро $(v_{k-2}, v_{k-1}) \in M$, и вершина $v_k \notin M$.

Можно найти максимальное паросочитание путем инверсии дополняющего пути.

Основная проблема заключается в том, как находить увеличивающий путь. Если в графе имеются циклы нечётной длины, то просто обход в глубину/ширину будет работать некорректно — при попадании в цикл нечётной длины обход может пойти по циклу в неправильном направлении.

Сжатие цветка

Сжатие цветка — это сжатие всего нечётного цикла в одну псевдовершину (соответственно, все рёбра, инцидентные вершинам этого цикла, становятся инцидентными псевдо-вершине).

Теорема Эдмондса

В графе $\overline{G}$ существует увеличивающая цепь тогда и только тогда, когда существует увеличивающая цепь в G.

Доказательство. Итак, пусть граф $\overline{G}$ был получен из графа G сжатием одного цветка (обозначим через B цикл цветка, и через $\overline{B}$ соответствующую сжатую вершину), докажем утверждение теоремы. Вначале заметим, что достаточно рассматривать случай, когда база цветка является свободной вершиной (не принадлежащей паросочетанию). Действительно, в противном случае в базе цветка оканчивается чередующийся путь чётной длины, начинающийся в свободной вершине. Прочередовав паросочетание вдоль этого пути, мощность паросочетания не изменится, а база цветка станет свободной вершиной. Итак, при доказательстве можно считать, что база цветка является свободной вершиной.

Доказательство необходимости. Пусть путь P является увеличивающим в графе G. Если он не проходит через B, то тогда, очевидно, он будет увеличивающим и в графе $\overline{G}$. Пусть P проходит через B. Тогда можно, не теряя общности, считать, что путь P представляет собой некоторый путь P_1, не проходящий по вершинам B, плюс некоторый путь P_2, проходящий по вершинам B и, возможно, другим вершинам. Но тогда путь $P_1 + \overline{B}$ будет являться увеличивающим путём в графе $\overline{G}$, что и требовалось доказать.

Доказательство достаточности. Пусть путь $\overline{P}$ является увеличивающим путём в графе $\overline{G}$. Снова, если путь $\overline{P}$ не проходит через $\overline{B}$, то путь $\overline{P}$ без изменений является увеличивающим путём в G, поэтому этот случай рассматриваться не будет.

Рассмотрим отдельно случай, когда $\overline{P}$ начинается со сжатого цветка $\overline{B}$, т.е. имеет вид $(\overline{B}, c, \ldots)$. Тогда в цветке B найдётся соответствующая вершина v, которая связана (ненасыщенным) ребром с c. Осталось только заметить, что из базы цветка всегда найдётся чередующийся путь чётной длины до вершины v. Учитывая всё вышесказанное, получаем, что путь $P = (b, \ldots, v, c, ...)$ является увеличивающим путём в графе G.

Пусть теперь путь $\overline{P}$ проходит через псевдо-вершину $\overline{B}$, но не начинается и не заканчивается в ней. Тогда в $\overline{P}$ есть два ребра, проходящих через $\overline{B}$, пусть это $(a, \overline{B})$ и $(\overline{B}, c)$. Одно из них обязательно должно принадлежать паросочетанию M, однако, т.к. база цветка не насыщена, а все остальные вершины цикла цветка B насыщены рёбрами цикла, то приходим к противоречию. Таким образом, этот случай просто невозможен.

Во всех случаях теорема доказана.

Общая схема алгоритма

N – количество вершин

i – вершина

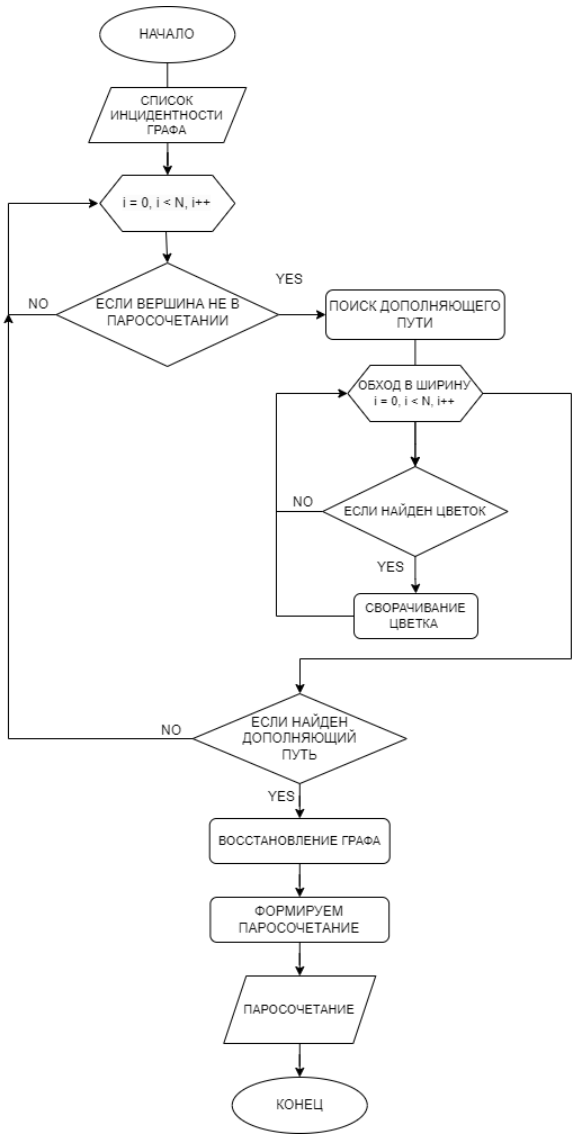

Рисунок 1-Общая схема алгоритма

Пример работы алгоритма

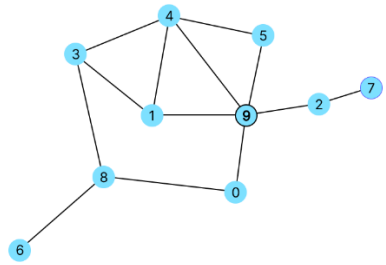

Рисунок 2-Исходный граф

Начинаем рассматривать граф со свободной вершины 7. Двигаясь по ребрам, обнаруживаем стебель: ребра 7-2 и 2-9. Следовательно, предполагаемый дополняющий путь будет начинаться со свободной вершины 7, ребро 2-9 — паросочетание. Тогда вершина 9 является базой. С помощью BFS идем дальше по графу: ребро 9-5, ребро 5-4, ребро 4-9 — составляют нечетный цикл — цветок.

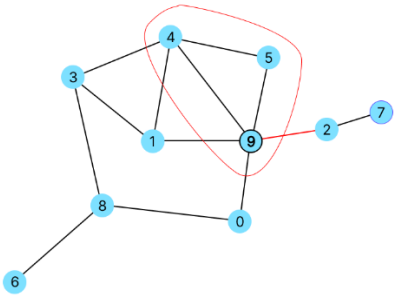

Рисунок 3- Нахождение первого цветка

Вершины цикла сжимаем в базу. Получаем следующий граф, который продолжаем обрабатывать по тому же алгоритму. С помощию BFS идем дальше по графу: ребро 9-3, ребро 3-1, ребро 1-9 — составляют нечетный цикл — цветок.

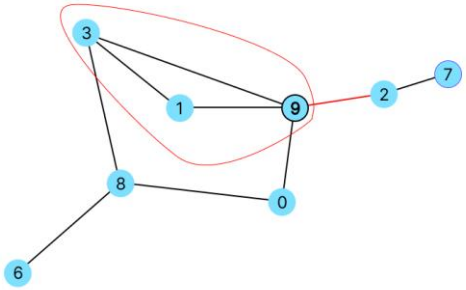

Рисунок 4-Нахождение второго цветка

Вершины цикла сжимаем в базу. Новый граф снова обрабатываем. Ребра 9-8 8-0 и 0-9 — составляют нечетный цикл — цветок.

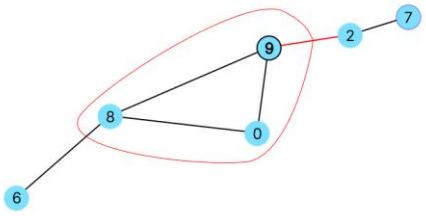

Рисунок 5-: Нахождение третьего цветка

Сжимаем цветок и продолжаем анализировать граф. После базы 9 идет только одно ребро 9-6, окончание которого — свободная вершина 6. Следовательно можем утверждать, что мы нашли дополняющий путь: 7-2, 2-9, 9-6.

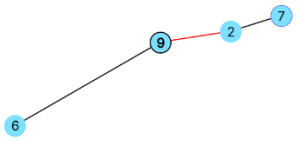

Рисунок 6-Дополняющий путь

Обращаемся к теореме Эдмондса: В графе G существует увеличивающая цепь тогда и только тогда, когда существует увеличивающая цепь в G. Значит мы можем приступить к восстановлению графа путем последовательного возвращения цветков. Для нахождения максимального паросочетания начнём с инверсии дополняющего пути. Начинаем восстановление с последнего сжатия цветка, инвертируя путь. При этом инвертрование пути подразумевает переопределение паросочетания. В цветке мы определяем паросочетание "в обратной последовательности": начиная с 9-0, переходя к 0-8. Так как в дополняющем пути база 9 была соединена с вершиной 6, то дойдя до узла, соединенного с вершиной 6, мы продолжаем переопределять паросочетание в направлении вершины 6. На данном этапе паросочетание составляет следующие не инцидентные ребра: 7-2, 9-0, 8-6

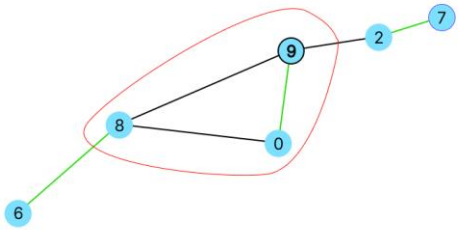

Рисунок 7-Восстановление третьего цветка

Запоминаем проставленное паросочетание и продолжаем восстановление графа. Ребро 7-2 уже инвертированно, поэтому переходим сразу к базе: в цветке определяем паросочетание "в обратной последовательности". База уже относится к ребру паросочетания, поэтому мы не можем пометить ребро 9-1. Значит помечаем следующее ребро 1-3. Аналогично с ребром 3-9. На данном этапе паросочетание составляет следующие не инцидентные ребра: 7-2, 9-0, 8-6, 3-1

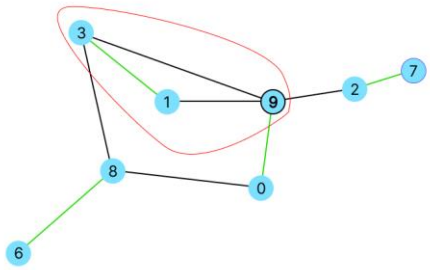

Рисунок 8-Восстановление второго цветка

Восстанавливаем последний цветок: аналогично помечаем ребра, начиная с базы. Ребро 9-4 — не помечаем, ребро 4-5 — помечаем, ребро 5-9 — не помечаем.

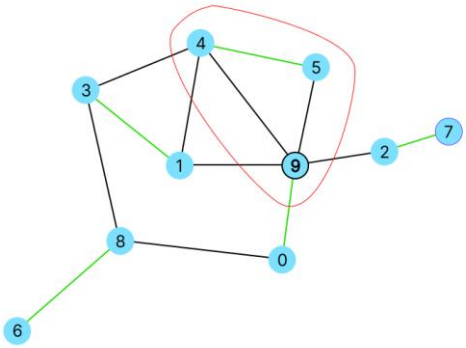

Рисунок 9- Восстановление первого цветка

Мы закончили восстановление графа и нашли максимальное паросочетание.

Реализация

Для реализации алгоритма был выбран язык программирования C++, среда разработки — Visual Studio 2022, является одним из самых популярных средств написания кода. Написана библиотека blossom.h для нахождения максимального паросочетания, в которой реализованы следующие функции:

```
#pragma once
#include <vector>
#include <iostream>

int get_match(std::vector<std::vector<int>>&, std::vector<int>&);

void print_match(std::vector<int>&);
```

Поиск дополняющего пути

Функция get_match() принимает список инцидентности и вектор, куда будет записано паросочетание. В результате работы помещает в переменную a паросочетание.

```
int get_match(std::vector<std::vector<int>>& g, std::vector<int>& match) {

	match.clear();
	match.resize(g.size(), -1);
	std::vector<int> p;
	p.resize(g.size(), -1);

	//запуск поиска пути из каждой вертев графа, проставление паросочетания
	for (int i = 0; i < g.size(); i++) {
		if (match[i] == -1) {
			int v = find_path(i, g, match, p);
			while (v != -1) {
				int pv = p[v], ppv = match[pv];
				match[v] = pv, match[pv] = v;
				v = ppv;
			}
		}
	}
	return 0;
}
```

Для реализации функции get_match() были написаны вспомогательные функции:

Нахождение ближайшего предка

Функция lca() находит общего ближайшего предка для вершин цветка — базу.

```
//нахождение ближайшего предка
int lca(int a, int b, std::vector<int>& base, std::vector<int>& match, std::vector<int>& p) {
	std::vector<bool> used;
	used.resize(base.size(), false);
	// поднимаемся от вершины a до корня, помечая все чётные вершины
	for (;;) {
		a = base[a];
		used[a] = true;
		if (match[a] == -1)  break; // дошли до корня
		a = p[match[a]];
	}
	// поднимаемся от вершины b, пока не найдём помеченную вершину
	for (;;) {
		b = base[b];
		if (used[b])  return b;
```

```
                b = p[match[b]];
            }
        }
```

Обозначение дополняющего пути

Функция mark_path() помечает чередующийся путь.

```
        //обозначение доп. пути
        void mark_path(int v, int b, int children, std::vector<int>& base, std::vector<int>& match, std::vector<int>& p,
std::vector<bool>& blossom) {
            while (base[v] != b) {
                blossom[base[v]] = blossom[base[match[v]]] = true;
                p[v] = children;
                children = match[v];
                v = p[match[v]];
            }
        }
```

Функция find_path() ищет дополняющий путь из каждой вершины. Результатом работы функции является последняя вершина дополняющего пути

```
        //поиск доп. пути из свободной вершины root и возвращает последнюю вершину этого пути, либо -1, если
ув. путь не найден
        int find_path(int root, std::vector<std::vector<int>> &g, std::vector<int> &match, std::vector<int>& p) {
            p.clear();
            p.resize(g.size(), -1);

            std::vector<bool> used;
            used.resize(g.size(), false);
            std::vector<bool> blossom;
            std::vector<int> q;
            q.resize(g.size(), 0);
            std::vector<int> base;

            for (int i = 0; i < g.size(); i++)
                base.push_back(i);

            //обход в ширину
            used[root] = true;
            int qh = 0, qt = 0;
            q[qt++] = root;
            while (qh < qt) {
                int v = q[qh++];
                for (size_t i = 0; i < g[v].size(); i++) {
                    int to = g[v][i];
                    if (base[v] == base[to] || match[v] == to)  continue;
                    if (to == root || match[to] != -1 && p[match[to]] != -1) {
                        int curbase = lca(v, to, base, match, p);

                        blossom.clear();
                        blossom.resize(g.size(), false);

                        mark_path(v, curbase, to, base, match, p, blossom);
                        mark_path(to, curbase, v, base, match, p, blossom);
                        for (int i = 0; i < g.size(); i++)
                            if (blossom[base[i]]) {
                                base[i] = curbase;
                                if (!used[i]) {
                                    used[i] = true;
                                    q[qt++] = i;
                                }
                            }
                    }
                    else if (p[to] == -1) {
```

```
                                        p[to] = v;
                                        if (match[to] == -1)
                                                return to;
                                        to = match[to];
                                        used[to] = true;
                                        q[qt++] = to;
                                }
                        }
                }
                return -1;
}
```

Также была реализована функция print_match() — принимает паросочетание и выводит его в консоль.

```
void print_match(std::vector<int>& a) {
        for (int i = 0; i < a.size(); i++) {
                if (i < a[i]) {
                        std::cout << i << '-' << a[i] << "\n";
                }
        }
}
```

Тестирование

Для проведения тестирования был создан набор из 37 файлов.

Часть тестов были созданы вручную, остальные автомтатически сгенерированы на языке программирования P ython.

Реализация генератора

1 import networkx as nx

2 import matplotlib.pyplot as plt

3 import scipy

4

5 n = vertices_count

6 constructor = [(n, n ∗ n, 0.5)]

7 g = nx.random_shell_graph(constructor)

8 nx.draw(g, with_labels=True)

9 graph = [set() for _ in range(n)]

10 for i, j in g.edges():

11 graph[j].add(i)

12 graph[i].add(j)

Запись в файл в виде списка инцидентности:

1 with open(r'C:\Users\user\source\repos\Blossom\graph.txt', 'w') as fout:

2 for vertex in graph:

3 print(∗vertex, file=fout)

Тесты покрывают различные ситуации:

Набор одиночных вершин; цепочек из ребер и вершин; некрупных

(5-19 вершин), средних (20-100 вершин) и крупных (100+ вершин) произвольных графов с четным, нечетным стеблем и без него; некрупных (5-19

вершин), средних (20-100 вершин) и крупных (100+ вершин) полносвязных графов; лесов из разного количества деревьев разной сложности;

произвольных графов, состоящих из более чем 100, 200 и 500.

18

Каждый тест включает два файла: input.txt и output.txt. Файл

input.txt сожержит список инцидентности: номер строки — вершина;

значения, записанные через пробел — вершины, сопряженные с данной.

Пример входного файла:

1 3 4

2 8 5

3 3 6

4 0 2 5 6 7 8

5 0 6

6 1 3 7

7 2 3 4

8 3 5

9 1 3

Файл output.txt содержит только одно число — количество несопряженных ребер в итоговом парасочетании.

Проверка результатов работы программного кода осуществляется

по критериям:

1. Проверяется уникальность вершин в итоговом просочетании.

2. Проверяется количество ребер в итоговом парасочетании относительно готового решения.

3. Проверяется подлинность (существование) ребер в итоговом парасочетании.

Данные критерии позволят исключить возможность упущения решений из возможного множества паросочетаний графа. Уникальность

вершин позволит нам убедиться, что среди ребер нет смежных (в случае

обнаручения неуникальной вершины, т.е. пренадрежащей двум ребрам,

решение можно считать неверным). В случае, если граф имеет набор парасочетаний, то количество ребер в наборах будет всегда одинаковым.

Ложное решение может соответствовать первым двум критериям, но содержать не подлинные ребра — для этого необходимо проверить существование ребер в графе.

Анализ производительности

Корректность алгоритма

Проверка алгоритма на корректную работу проводилась следующим образом:

Первый этап — решение тестов вручную.

Второй этап — атоматическая проверка с помощью готового решения. Для реализации второго этапа использовалась функция networkx.maxweightmatching()

из библиотеки networkx языка программирования Python.

Реализованный алгоритм и функция networkx.maxweightmatching()

возвращали одинаковый набор ребер на всех тестах.

Производительность

Тестирование производительности проводилось на трех наборах графов: полносвязные графы, произвольные графи и лес. Тест производительности показал, что полносвязные графы больше всего нагружают алгоритм как по памяти, так и по времени работы. Вне зависимости от структуры графа, время работы алгоритма не превышала 6 мс для графов до 100 вершин. При увеличении вершин наблюдаются различия в работе алгоритма: Работа алгоритма на полносвязных графах от 500 вершин больше по памяти (до 2 МБ) и дольше по времени (до 6.2 с).

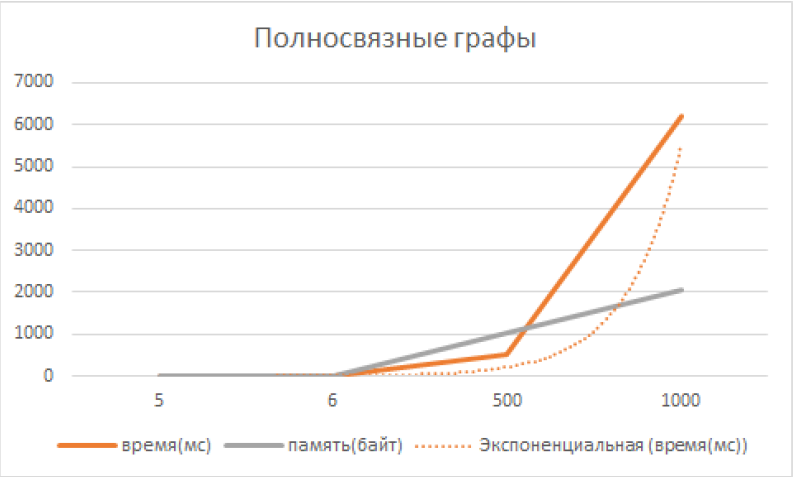

Рисунок 10- График зависимости памяти и времени от количества вершин

Обработка произвольного графа от 100 вершин и более не занимает

порядка 20 мс и не нагружает алгоритм по памяти.

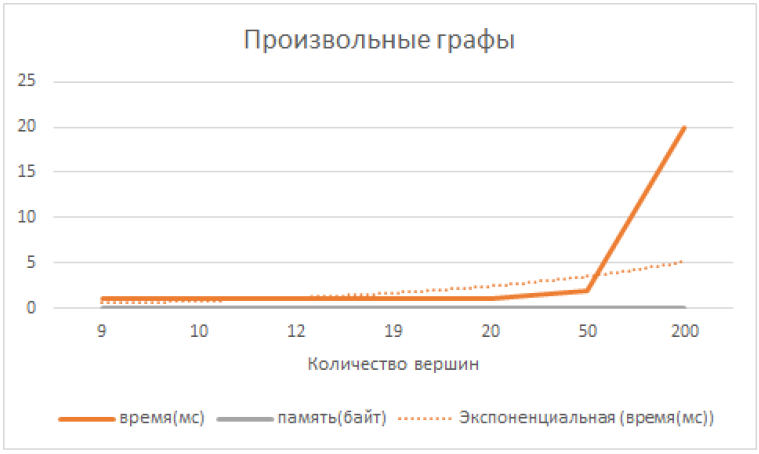

Рисунок 11- График зависимости памяти и времени от количества вершин

Результат отработки лесов свыше 100 вершин не нагружает память, но выходит дольше - от 6 мс и порядка 109 мс для 1000 вершин.

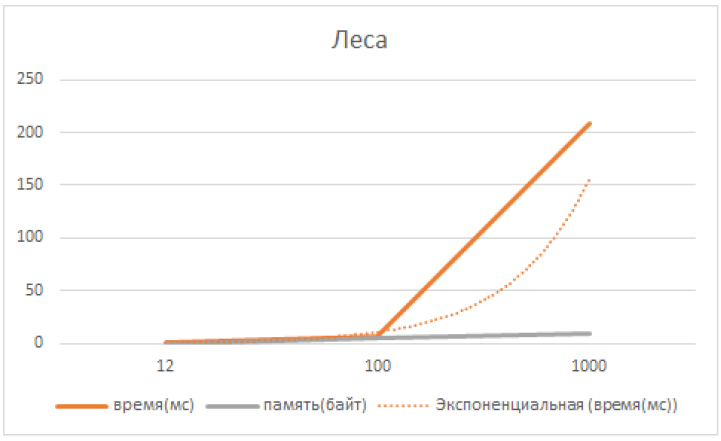

Рисунок 12- График зависимости памяти и времени от количества вершин

Результат тестирования производительности

По результатам тестирования было выявлено, что алгоритм работает корректно, было достигнуто 100% покрытие кода.

rarchy	Covered (Blocks)	Not Covered (Blocks)	Covered (Lines)	Partially Covered (Lines)	Not Covered (Lines)
find_path	104	0	44	0	0
get_match	27	0	16	0	0
lca	28	0	13	0	0
mark_path	18	0	8	0	0

Рисунок 13-Покрытие кода

Сравнение

Проведена сравнительная оценка с работой аналогичной функции для поиска максимального парасочетания — networkx.maxweightmatching() из библиотеки networkx языка программирования Python. Время работы алгоритма и функции примерно одинаково на графах до 100, но алгоритм выигрывает по времени выполнения. Однако при большем количестве вершин на полносвязных графах функция работает значительно быстрее, чем реализованный алгоритм.

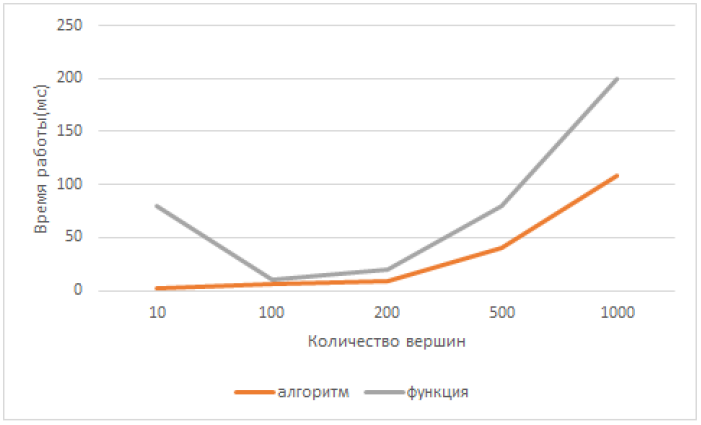

Рисунок 14-График зависимости памяти и времени от количества вершин

Заключение

В ходе изучения и реализации алгоритма были сделаны следующие выводы:

1. Появление алгоритма «Сжатие цветков» позволило решать новые задачи на графах с нечетными циклами.

2. Реализация библиотеки позволяет использовать алгоритм в других проектах.

3. Тестирование показало, что алгоритм работает корректно и эффективно.

4. Сравнительный анализ показал, что алгоритм работает эффективнее с малым количеством данных, но не сохраняет преимущество при больших объемах данных.

Список литературы

1. https://ru.wikipedia.org/wiki/Алгоритм_сжатия_цветков#Ссылки
2. https://algorithmica.org/ru/matching
3. https://depth-first.com/articles/2020/09/28/edmonds-blossom-algorithm-part-1-cast-of-characters/
4. http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/notes/Edmonds/index.html
5. https://intellect.icu/algoritm-vyrezaniya-sotsvetij-i-szhatiya-tsvetkov-8626
6. http://e-maxx.ru/algo/matching_edmonds
7. https://руни.рф/index.php/Алгоритм_сжатия_цветков#Цветки_и_стягивание
8. https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_вырезания_соцветий
9. https://algorithms.discrete.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html
10. https://infogalactic.com/info/Blossom_algorithm