Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт

по рубежному контролю №1

Название «Разработка параллельного алгоритма поиска в глубину в неориентированном графе из заданной вершины»

Дисциплина «Анализ алгоритмов»

Студент ИУ7-55Б _____ (подпись, дата) Бугаенко А.П. (Фамилия И.О.)

Преподователь _____ (подпись, дата) Волкова Л.Л. (Фамилия И.О.)

Москва, 2022

Содержание

Введение

Графом $G(V, E)$ называется совокупность двух множеств - непустого множества V и множества E неупорядоченных пар различных элементов множества V. Множество V называется множеством вершин, множество E называется множеством ребер [1]. Граф G называется полным, если в его состав входит хотя бы одна вершина, и граф G называется связным, если любые две его вершины соединены путём в G [2].

Целью данного рубежного контроля является разработка параллельного алгоритма поиска в глубину в неориентированном графе из заданной вершины. Для того, чтобы достичь поставленной цели необходимо выполнить следующие задачи:

1) провести анализ алгоритма поиска в глубину;
2) провести анализ параллельного алгоритма поиска в глубину;
3) описать используемые в алгоритмах структуры данных;
4) привести схемы рассматриваемых алгоритмов;
5) программно реализовать описанные выше алгоритмы;
6) провести сравнительный анализ скорости работы алгоритмов по времени относительно количества вершин в графе.

1 Аналитический раздел

В данном разделе рассматриваются теоретические основы работы алгоритма поиска в глубину и рассматриваются возможности его параллелизации.

1.1 Алгоритм поиска в глубину в неориентированном графе из заданной вершины

Поиск в глубину (Depth-first search, DFS) — один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

1.2 Возможности параллелизации алгоритма поиска в глубину в неориентированном графе из заданной вершины

При поиске в глубину невозможен параллельный проход по вершинам подобно алгоритму поиска в ширину, поскольку путь строится как одна цепочка узлов, причём положение каждого следующего зависит от того, какой узел был добавлен в конец ветки пути на прошлой итерации. Причём разбиение графа на подмножества не даст нужный результат, поскольку пути, найденные в подмножествах, не будут составлять корректный путь (такой, что образуется при непараллельном поиске в глубину). Ниже на изображении 1.1 изображён граф G.

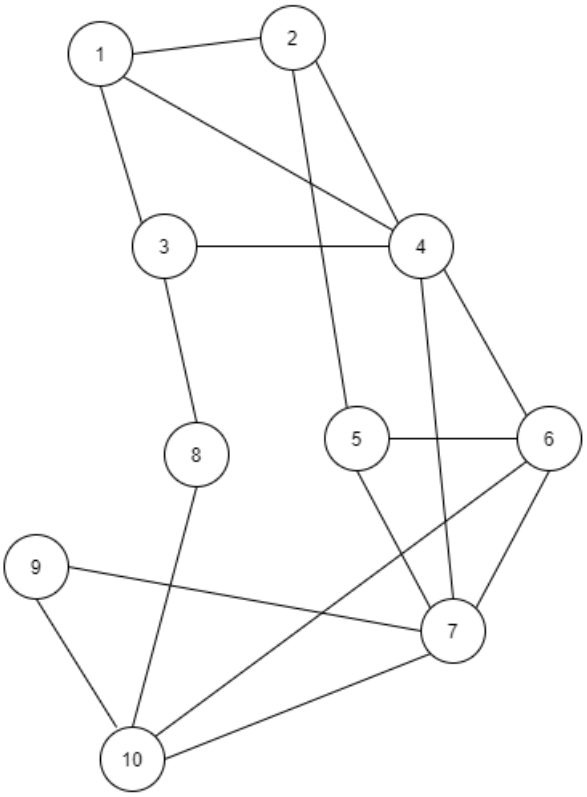

Рисунок 1.1 — Граф G

В качестве примера мы попытаемся разбить его на множества. Разобъем его на два непересекающихся множества вершин. Если мы попытаемся использовать алгоритм поиска в глубину для каждого из них, то в итоге получатся конфликтующие пути, как показано ниже на изображении 1.2.

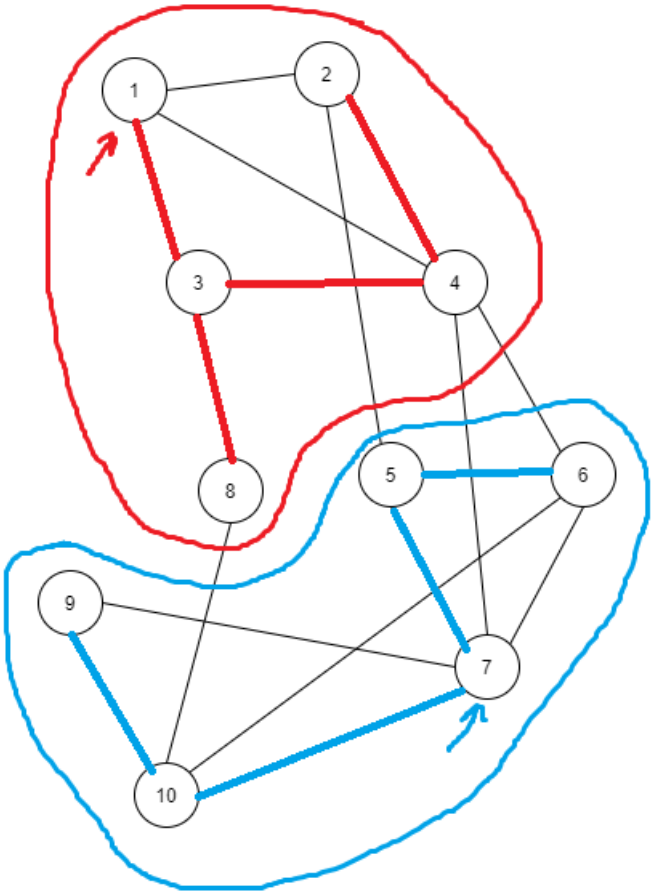

Рисунок 1.2 — Граф G, разбитый на два пересекающихся множества

В данном случае поиск в множестве, содержащем вершины 1, 2, 3, 4 и 8 вёлся начиная с вершины 1, во втором множестве начиная с вершины 7. В результате возникает вопрос, как корректно соединить эти множества? Возможным решением может быть использование частично пересекающихся множеств, которые имеют выраженное начало и конец, и конец предыдущего множества является начало следующего. Данная ситуация рассмотрена ниже на рисунке 1.3.

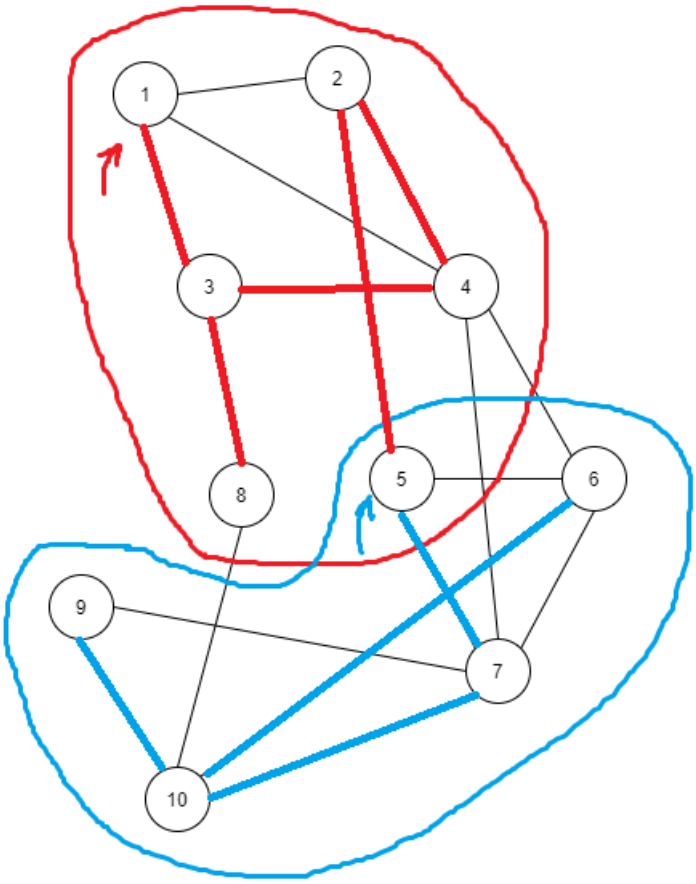

Рисунок 1.3 — Граф G, разбитый на два частично пересекающихся множества

В этом случае получившиеся пути не являются корректным результатом поиска в глубину, поскольку в этом случае вершина 8 была бы соединена с вершиной 10, так как дополнительные пути добавляются с конца. Для сравнения ниже на изображении 1.4 приводится результат, полученный путем применения стандартного алгоритма поиска в глубину. Стрелками помечен порядок обхода графа.

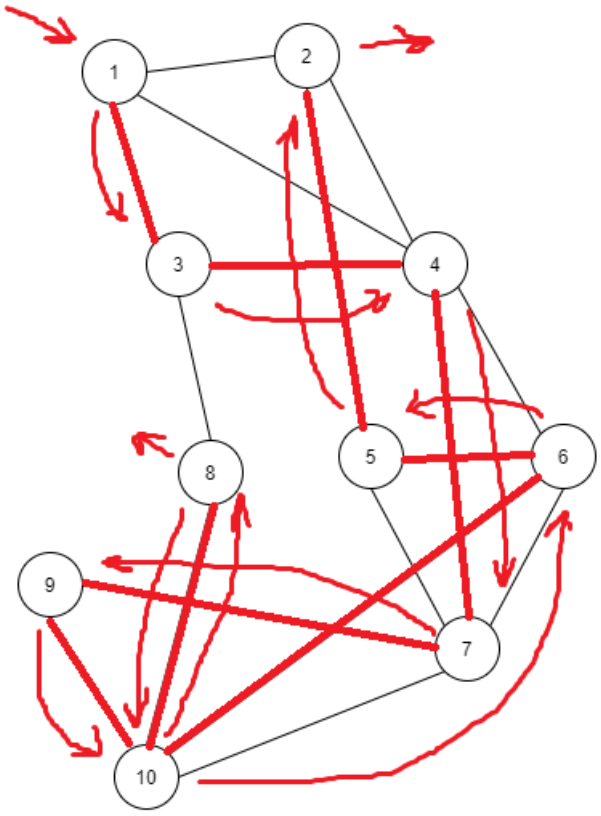

Рисунок 1.4 — Граф G с результатом поиска в глубину

Не смотря на то, что получаемые с помощью способа разбиения графа на подмножества пути могут быть достаточно близки к пути, получаемому при обходе в глубину, смысла рассматривать этот способ дальше не имеет, поскольку время, потраченное на составление и проверку пути из подмножеств полностью сведет на нет полученную в результате разбиения разницу в скорости выполнения со стандартным алгоритмом.

Если подробно рассмотреть работу непараллельного алгоритма поиска в глубину, то можно заметить, что множество времени тратится на обратный проход, когда необходимо найти непосещённые вершины и построить путь до них. Этот этап можно параметризировать, выполнив проверку параллельно и использовав результаты после этого.

1.3 Анализ входных и выходных данных

В качестве входных данных программа принимает непустой связный граф, и вершину, с которой следует начать поиск пути в глубину. На выходе программа возвращает множество найденных путей.

В случае параллельной реализации алгоритма к входным данным необходимо добавить количество используемых потоков.

1.4 Анализ ограничений, в рамках которых разрабатываются алгоритмы

На вход программе подаётся непустой связный граф, имеющий минимум одну вершину. Верхняя граница количества вершин ограничивается возможностями устройства, на котором выполняется программа.

1.5 Вывод

В данном разделе были рассмотрены основные теоретические сведения об алгоритмах поиска в глубину в графе и параллельного поиска в глубину в графе. На вход разрабатываемому алгоритму поиска в глубину на вход подаётся непустой связный граф и вершина, с которой начинается поиск. На вход разрабатываемому алгоритму параллельного поиска в глубину на вход подаётся непустой связный граф, вершина, с которой начинается поиск и количество потоков. Граф, подающийся на вход должен иметь минимум одну вершину, верхняя граница количества вершин ограничивается возможностями устройства, на котором выполняется программа.

2 Конструкторский раздел

В данном разделе будут рассмотрены схемы, структуры данных, способы тестирования для следующих алгоритмов:

— алгоритм поиска в глубину;
— параллельный алгоритм поиска в глубину.

2.1 Алгоритм поиска в глубину

Используемые типы и структуры данных включают в себя:

1) целое число - используется для хранения и работы с целочисленными переменными;
2) список - двунаправленный список для работы с массивами данных;
3) узел (Node) - пользовательский класс, включающий в себя поля id (идентификатор узла) и connections (список узлов, связанных с этим узлом ребрами) и метод print_node, выводящий информацию об узле;

4) граф (Graph) - пользовательский класс, включающий в себя поле nodes (список узлов, которые составляют граф) и методы draw_graph (вывести визуальную интерпретацию графа на экран), print_graph (вывести все узлы), get_node_with_id (возвращает объект класса Node по id).

Схема алгоритма приведена на рисунках 2.1, 2.2 ниже.

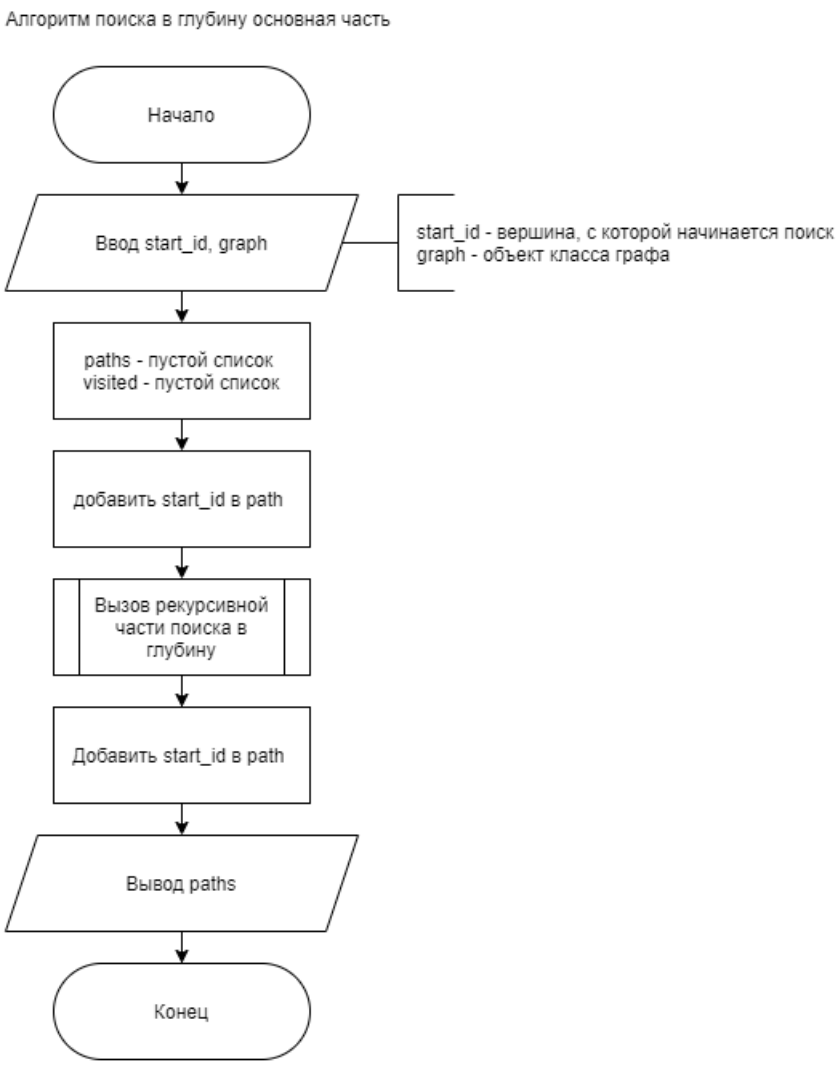

Рисунок 2.1 — Основная часть алгоритма

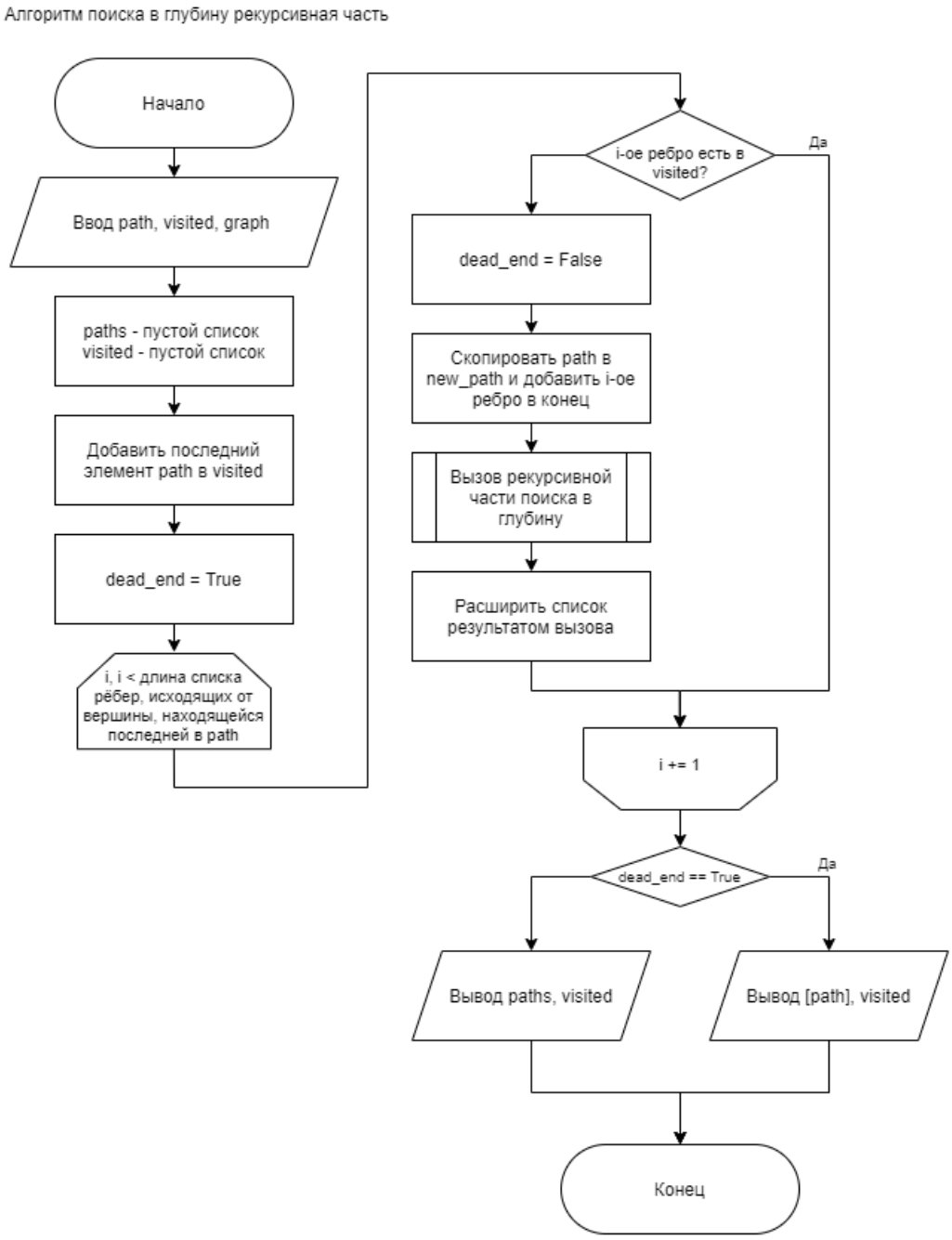

Рисунок 2.2 — Рекурсивная часть алгоритма

2.2 Алгоритм параллельного поиска в глубину

1) целое число - используется для хранения и работы с целочисленными переменными;

2) список - двунаправленный список для работы с массивами данных;

3) поток (Thread) - объект класса потока, позволяющий запустить выполнение функции в отдельном потоке;

4) узел (Node) - пользовательский класс, включающий в себя поля id (идентификатор узла) и connections (список узлов, связанных с этим узлом ребрами) и метод print_node, выво-

дящий информацию об узле;

5) граф (Graph) - пользовательский класс, включающий в себя поле nodes (список узлов, которые составляют граф) и методы draw_graph (вывести визуальную интерпретацию графа на экран), print_graph (вывести все узлы), get_node_with_id (возвращает объект класса Node по id).

Схема алгоритма приведена на рисунках 2.3 - 2.6 ниже.

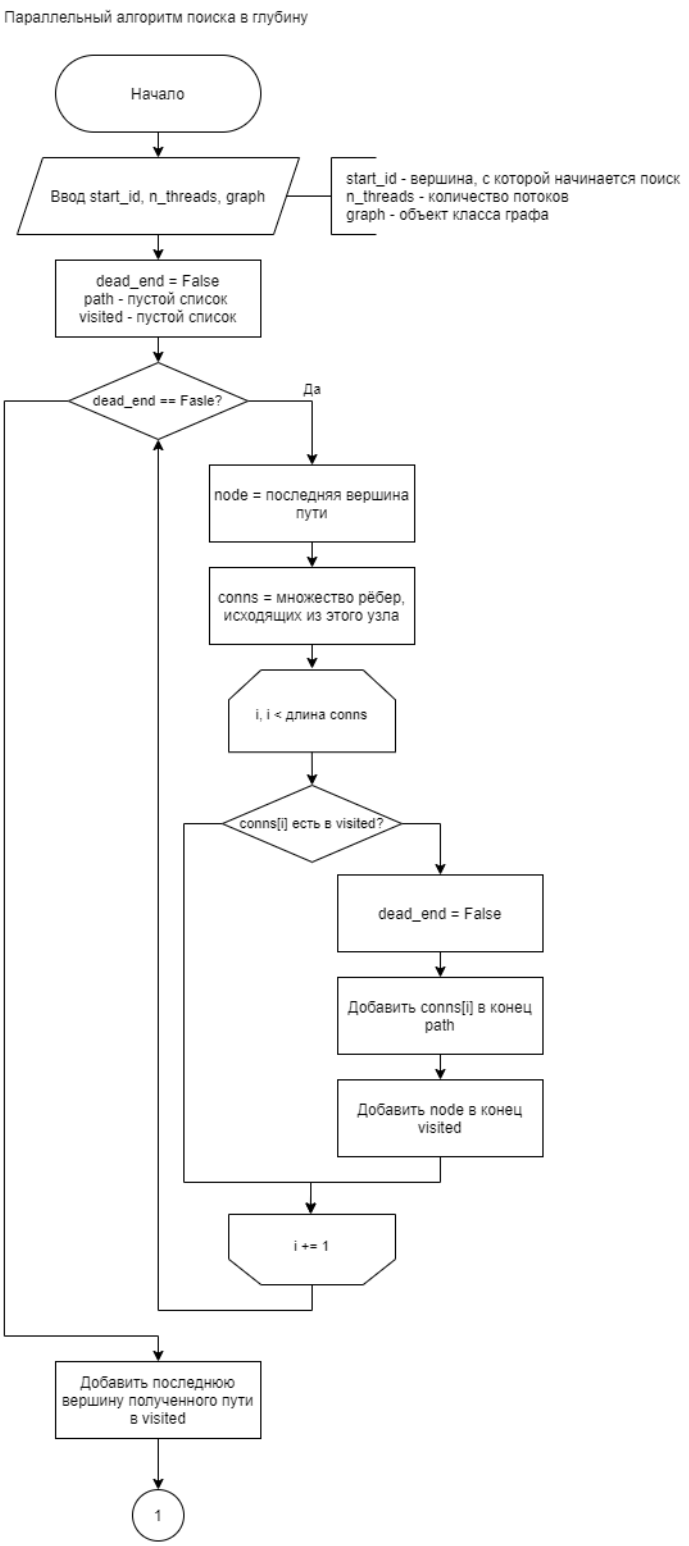

Рисунок 2.3 — Схема параллельного алгоритма, часть 1

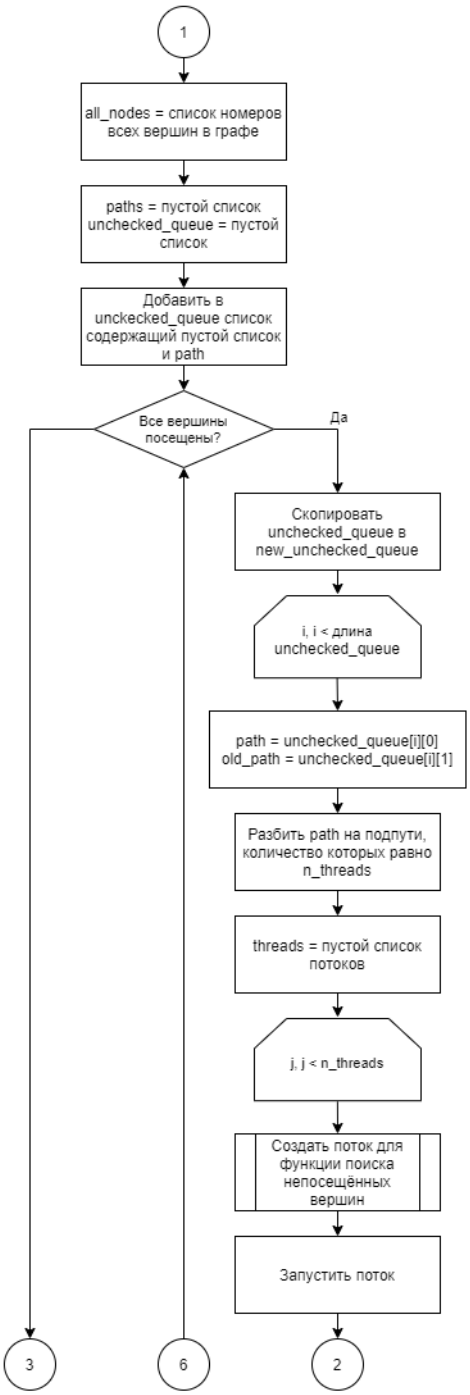

Рисунок 2.4 — Схема параллельного алгоритма, часть 2

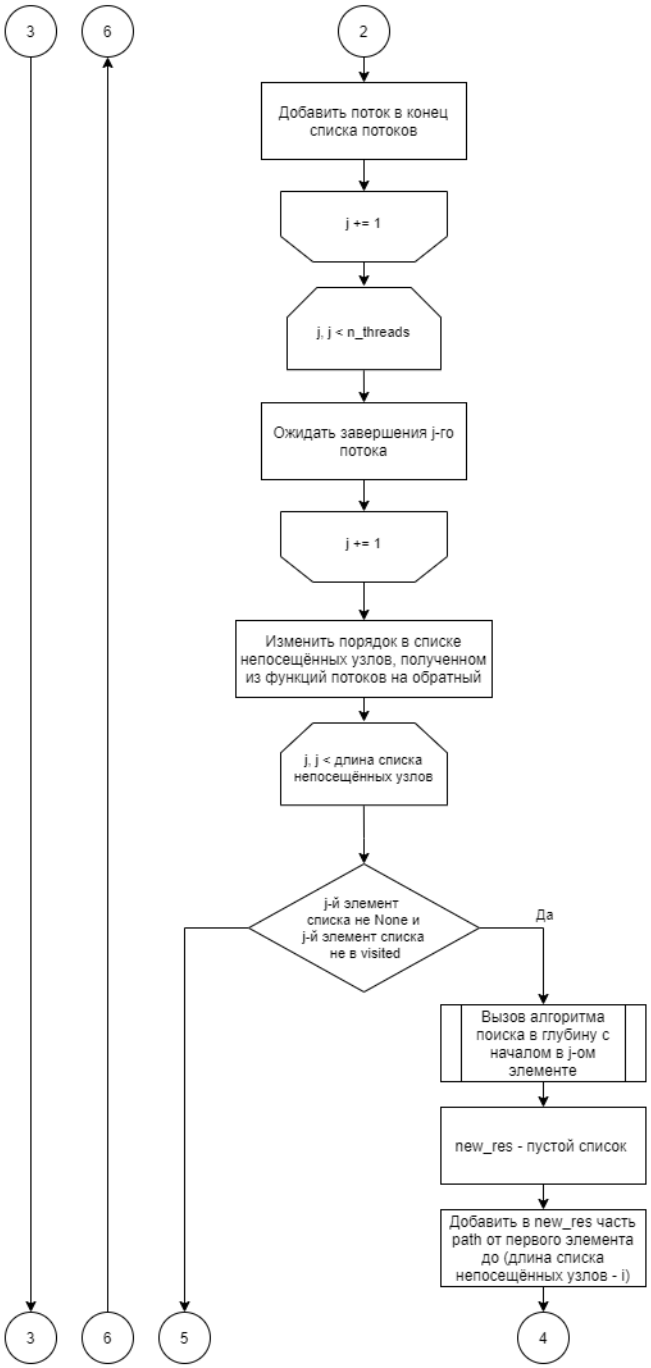

Рисунок 2.5 — Схема параллельного алгоритма, часть 3

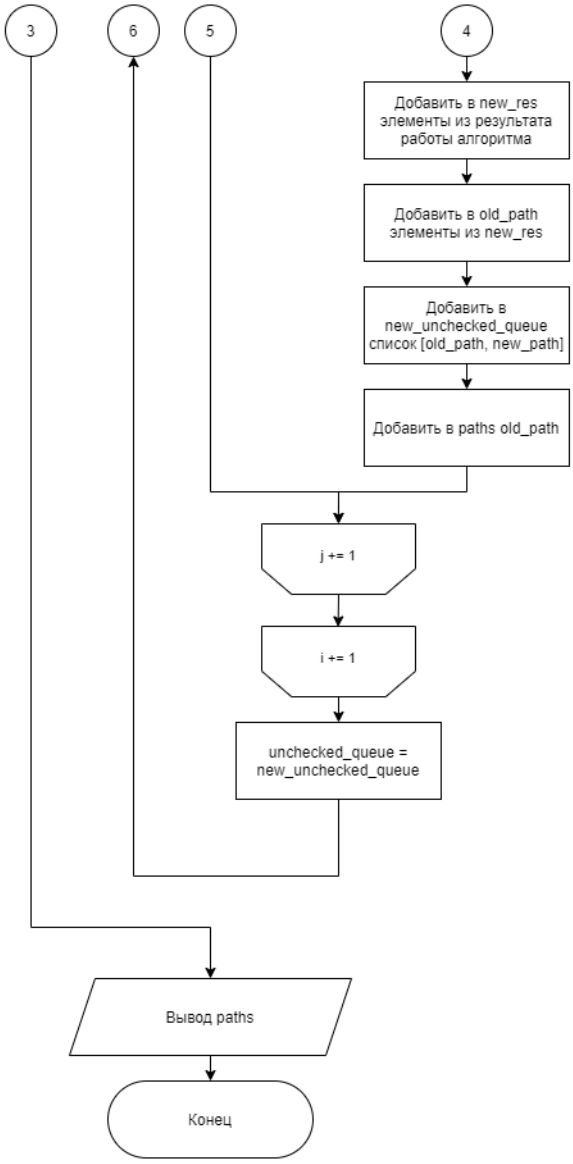

Рисунок 2.6 — Схема параллельного алгоритма, часть 4

2.3 Тестирование алгоритма поиска в глубину

Описание тестов для алгоритам поиска в глубину:

1) проверка работы на графе из одной вершины;
2) проверка работы на графе из чётного количества вершин;
3) проверка работы на графе из нечётного количества вершин;

2.4 Функциональная схема ПО

На изображении ниже представлена функиональная схема разрабатываемого ПО. На вход подаётся граф, идентификатор вершины, с которой начинается поиск и количество по-

токов, при помощи алгоритмов, реализованных на языке Python мы получаем в результате работы множество найденных путей.

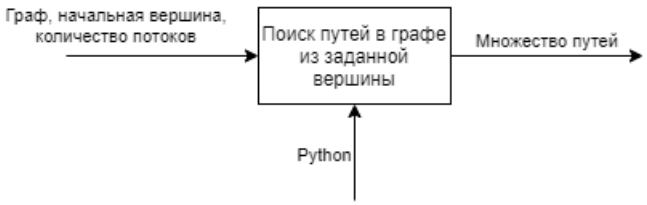

Рисунок 2.7 — IDEF0 диаграмма разрабатываемой программы

2.5 Вывод

В данном разделе были рассмотрены схемы алгоритма поиска в глубину, параллельного алгоритма поиска в глубину. Были определены тесты для каждого алгоритма и описаны типы и структуры данных, использующихся в алгоритмах. Также была приведена функциональная схема разрабатываемого ПО.

3 Технологический раздел

В данном разделе будут рассмотрены подробности реализации описаных выше алгоритмов. Также будут обоснованы выбор языка программирования для реализации, выбор библиотек для проведения экспериментов и представлены важные фрагменты кода написанной в рамках работы программы.

3.1 Выбор языка программирования

В качестве языка программирования для реализации данной лабораторной работы использовался язык программирования Python поскольку он предоставляет широкие возможности для эффективной реализации алгоритмов. В качестве среды разработки использовалась Visual Studio Code по причине того, что данная среда имеет встроенные средства отладки и анализа работы программы, позволяющие быстро и эффективно писать код.

3.2 Сведения о модулях программы

Реализованное ПО состоит из трёх модулей:

1) main - основной файл программы, где находится точка входа;
2) simple_dfs - реализация рекурсивного алгоритма поиска в глубину;
3) paralle_dfs - реализация параллельного алгоритма поиска в глубину;
4) test - реализация тестирования;
5) time - реализация тестирования временной производительности;
6) graph - реализация классов графа и вершины.

3.3 Реализация алгоритмов

Листинг 3.1 — Реализация алгоритма поиска в глубину

```
1  def simple_dfs(self, start_id: int, graph: Graph) -> list[list[int]]:
2      path = [start_id]
3      visited = []
4      paths, _ = self.simple_dfs_rec(path, visited, graph)
5      return paths
6
7  def simple_dfs_rec(self, path: list[int], visited: list[int], graph: Graph) ->
       list[list[int]]:
8      paths = []
9      visited.append(path[-1])
10     dead_end = True
11     for conn in graph.get_node_with_id(path[-1]).connections:
12         if conn not in visited:
13             dead_end = False
14             new_path = path.copy()
15             new_path.append(conn)
```

```
            res_path, visited = self.simple_dfs_rec(new_path, visited, graph)
            paths.extend(res_path)

    if dead_end:
        return [path], visited
    else:
        return paths, visited
```

Листинг 3.2 — Реализация параллельного алгоритма поиска в глубину

```

def parallel_search_unvisited(self, node_ids: list[int], return_ids:
    list[list[int]], index: int, visited: set[int], graph: Graph):
    return_id = [None] * len(node_ids)

    for i in range(0, len(node_ids)):
        conns = graph.get_node_with_id(node_ids[i]).connections
        for j in range(0, len(conns)):
            if conns[j] not in visited:
                return_id[i] = conns[j]
                break
    return_ids[index] = return_id

def parallel_dfs_subgraph(self, start_id: int, visited: set[int], graph: Graph):
    dead_end = False
    path = [start_id]
    visited.add(start_id)
    node = path[-1]

    while not dead_end:
        conns = graph.get_node_with_id(node).connections
        dead_end = True
        for conn in conns:
            if conn not in visited:
                dead_end = False
                path.append(conn)
                node = conn
                visited.add(conn)
                break

    return path, visited


def parallel_dfs(self, start_id: int, n_threads: int, graph: Graph):
    dead_end = False
    path = [start_id]
    visited = set()
```

```
37
38      while not dead_end:
39          node = path[-1]
40          conns = graph.get_node_with_id(node).connections
41
42          dead_end = True
43          for conn in conns:
44              if conn not in visited and dead_end:
45                  dead_end = False
46                  path.append(conn)
47                  visited.add(node)
48
49      visited.add(path[-1])
50
51      all_nodes = set()
52      for node in graph.nodes:
53          all_nodes.add(node.id)
54
55      paths = [path]
56      unchecked_queue = [[], path]]
57
58      while len(all_nodes) != len(visited):
59          new_unchecked_queue = unchecked_queue.copy()
60          for pair in unchecked_queue:
61              path = pair[1]
62              old_path = pair[0]
63
64              # find unchecked nodes
65              node_index = 0
66              batch_size = ceil(len(path) / n_threads)
67              batches = []
68              batch = []
69
70              path_len = len(path)
71              while node_index < path_len:
72                  batch.append(path[node_index])
73                  if len(batch) == batch_size:
74                      batches.append(batch)
75                      batch = []
76                  node_index += 1
77
78              if len(batch) > 0:
79                  if len(batches) == 0:
80                      batches.append(batch)
81                  else:
82                      batches[-1].extend(batch)
83
```

```
84              sum_size = 0
85              for batch in batches:
86                  sum_size += len(batch)
87
88              result_ids = [[]] * len(batches)
89              index = 0
90              threads = []
91
92              for batch in batches:
93                  thread = Thread(target=self.parallel_search_unvisited, args=(batch,
                        result_ids, index, visited, graph))
94                  thread.start()
95                  threads.append(thread)
96                  index += 1
97
98              for thread in threads:
99                  thread.join()
100
101             result_pass = []
102             for result_id in result_ids:
103                 result_pass.extend(result_id)
104             result_pass.reverse() # this list contains inverted next node to check
                    in path
105
106             # search for unvisited vertex
107             res_pass_len = len(result_pass)
108             for i in range(0, res_pass_len):
109                 if result_pass[i] != None and result_pass[i] not in visited:
110                     new_path, visited = self.parallel_dfs_subgraph(result_pass[i],
                            visited, graph)
111                     new_res = path[:(res_pass_len - i)]
112                     new_res.extend(new_path)
113                     old_path.extend(new_res)
114                     new_unchecked_queue.append([old_path, new_path]) # adding path
                            and last added node to unchecked queue
115                     paths.append(old_path)
116         unchecked_queue = new_unchecked_queue
117
118     return paths
```

3.4 Результаты тестирования алгоритмов

Для тестирования алгоритмов было реализованы следующие тесты:

1) проверка работы на графе из одного элемента;
2) проверка работы на ключе из середины словаря;

Результаты тестов:

3.5 Вывод

В данной разделе были представлены реализации алгоритма полного перебора, бинарного поиска и частичного анализа, а также представлены результаты работы модуля тестирования реализованных алгоритмов.

4 Экспериментальный раздел

В данном разделе описываются измерения временных характеристики алгоритмов полного перебра, бинарного поиска и частичного анализа для различных положений ключа в словаре, а также делается вывод об эффективности алгоритмов для соответствующих случаев.

4.1 Технические характеристики

— Операционная система - Windows 10, 64-bit;
— Оперативная память - 16 GiB;
— Процессор - Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz 2.59 GHz, 6 ядер, 12 потоков.

4.2 Результаты экспериментов

В данном разделе рассматриваются результаты экспериментов над разработанными алгоритмами. Результаты приведены ниже в таблицах 4.1 и на графиках 4.1 - .

Таблица 4.1 — Время работы алгоритмов для одного потока

количество узлов	t не параллельного (нс)	t не параллельного (нс)
100	1562500.0	1562500.0
150	6250000.0	9375000.0
200	4687500.0	10937500.0
250	1562500.0	20312500.0
300	10937500.0	21875000.0
350	12500000.0	25000000.0
400	15625000.0	29687500.0
450	18750000.0	37500000.0
500	28125000.0	40625000.0
550	34375000.0	40625000.0
600	37500000.0	42187500.0
650	43750000.0	62500000.0
700	48437500.0	42187500.0
750	59375000.0	59375000.0
800	70312500.0	56250000.0
850	78125000.0	84375000.0
900	84375000.0	98437500.0
950	103125000.0	125000000.0
1000	123437500.0	107812500.0
1050	134375000.0	117187500.0
1100	148437500.0	143750000.0
1150	153125000.0	132812500.0
1200	165625000.0	162500000.0
1250	196875000.0	162500000.0
1300	192187500.0	195312500.0
1350	207812500.0	198437500.0
1400	218750000.0	268750000.0
1450	229687500.0	276562500.0
1500	273437500.0	234375000.0

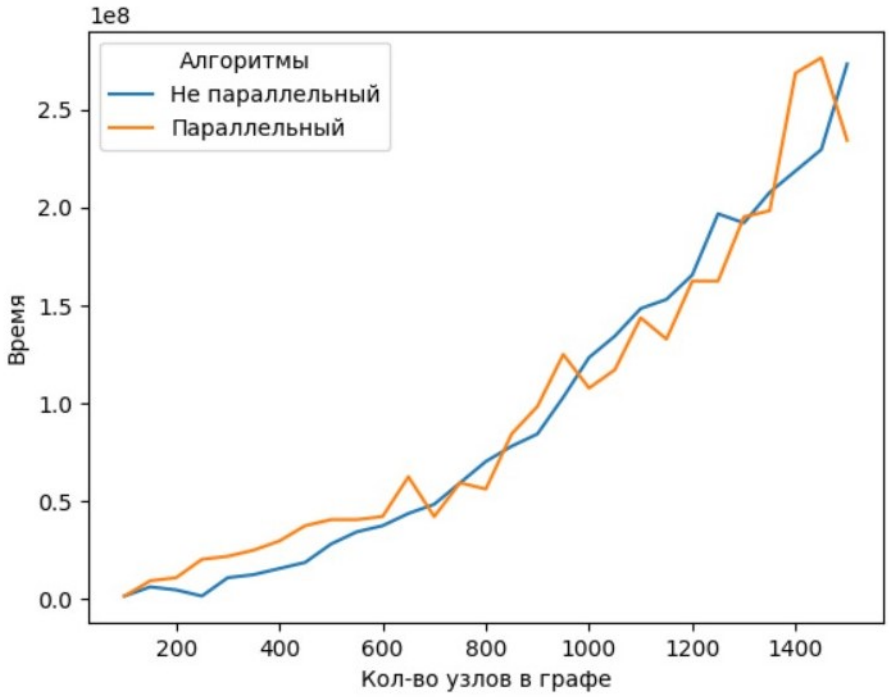

Рисунок 4.1 — График зависимости времени работы параллельного и непараллельного алгоритмов в зависимости от количества узлов для одного потока

Таблица 4.2 — Время работы алгоритмов для двух потоков

количество узлов	время не параллельного (нс)	время не параллельного (нс)
100	1562500.0	6250000.0
150	4687500.0	10937500.0
200	4687500.0	7812500.0
250	4687500.0	14062500.0
300	7812500.0	12500000.0
350	15625000.0	18750000.0
400	15625000.0	29687500.0
450	20312500.0	26562500.0
500	25000000.0	34375000.0
550	31250000.0	40625000.0
600	32812500.0	34375000.0
650	45312500.0	45312500.0
700	57812500.0	51562500.0
750	59375000.0	56250000.0
800	71875000.0	60937500.0
850	79687500.0	78125000.0
900	84375000.0	79687500.0
950	93750000.0	81250000.0
1000	101562500.0	89062500.0
1050	115625000.0	90625000.0
1100	126562500.0	146875000.0
1150	145312500.0	128125000.0
1200	154687500.0	126562500.0
1250	173437500.0	154687500.0
1300	178125000.0	139062500.0
1350	200000000.0	164062500.0
1400	209375000.0	206250000.0
1450	223437500.0	242187500.0
1500	239062500.0	231250000.0

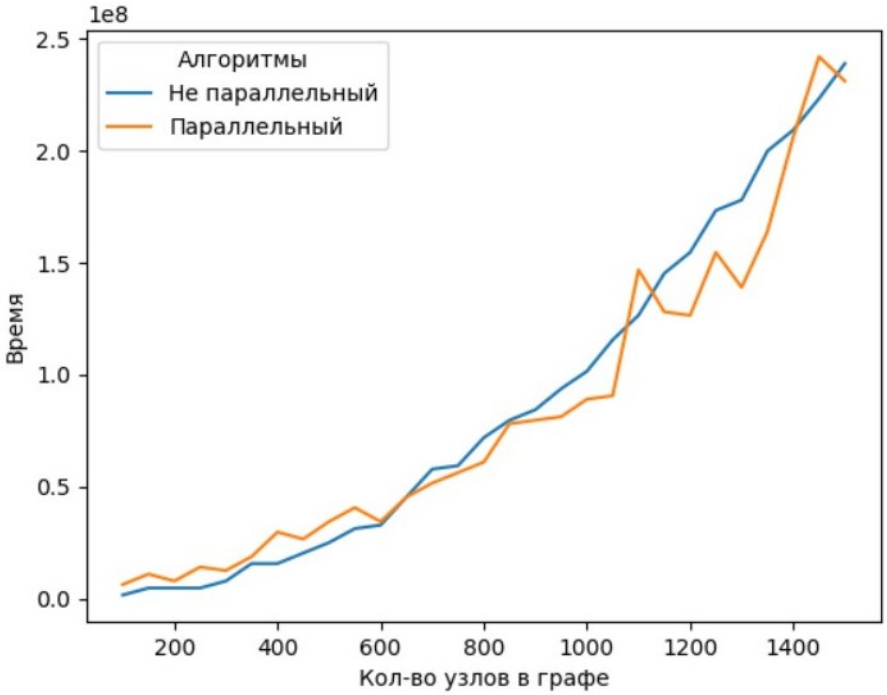

Рисунок 4.2 — График зависимости времени работы параллельного и непараллельного алгоритмов в зависимости от количества узлов для двух потоков

Таблица 4.3 — Время работы алгоритмов для четырех потоков

количество узлов	время не параллельного (нс)	время не параллельного (нс)
100	0.0	1562500.0
150	4687500.0	3125000.0
200	4687500.0	7812500.0
250	7812500.0	9375000.0
300	10937500.0	14062500.0
350	15625000.0	14062500.0
400	17187500.0	25000000.0
450	25000000.0	18750000.0
500	29687500.0	26562500.0
550	34375000.0	32812500.0
600	39062500.0	35937500.0
650	46875000.0	46875000.0
700	54687500.0	50000000.0
750	68750000.0	56250000.0
800	85937500.0	64062500.0
850	85937500.0	64062500.0
900	93750000.0	78125000.0
950	104687500.0	79687500.0
1000	117187500.0	89062500.0
1050	135937500.0	98437500.0
1100	148437500.0	139062500.0
1150	162500000.0	117187500.0
1200	170312500.0	137500000.0
1250	181250000.0	140625000.0
1300	193750000.0	157812500.0
1350	206250000.0	145312500.0
1400	206250000.0	176562500.0
1450	229687500.0	160937500.0
1500	243750000.0	200000000.0

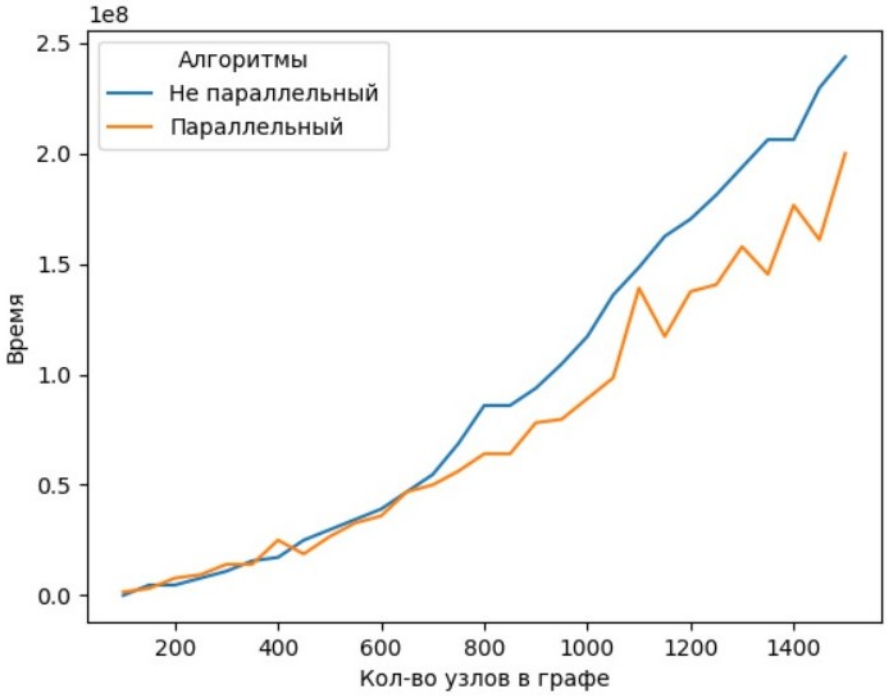

Рисунок 4.3 — График зависимости времени работы параллельного и непараллельного алгоритмов в зависимости от количества узлов для четырёх потоков

Таблица 4.4 — Время работы алгоритмов для восьми потока

количество узлов	время не параллельного (нс)	время не параллельного (нс)
100	0.0	1562500.0
150	3125000.0	3125000.0
200	14062500.0	1562500.0
250	6250000.0	9375000.0
300	12500000.0	10937500.0
350	14062500.0	14062500.0
400	17187500.0	23437500.0
450	26562500.0	20312500.0
500	29687500.0	20312500.0
550	34375000.0	25000000.0
600	40625000.0	37500000.0
650	46875000.0	40625000.0
700	54687500.0	42187500.0
750	67187500.0	48437500.0
800	76562500.0	50000000.0
850	90625000.0	67187500.0
900	93750000.0	78125000.0
950	123437500.0	75000000.0
1000	115625000.0	93750000.0
1050	135937500.0	90625000.0
1100	145312500.0	100000000.0
1150	160937500.0	114062500.0
1200	168750000.0	129687500.0
1250	185937500.0	132812500.0
1300	210937500.0	139062500.0
1350	229687500.0	160937500.0
1400	239062500.0	151562500.0
1450	259375000.0	190625000.0
1500	275000000.0	179687500.0

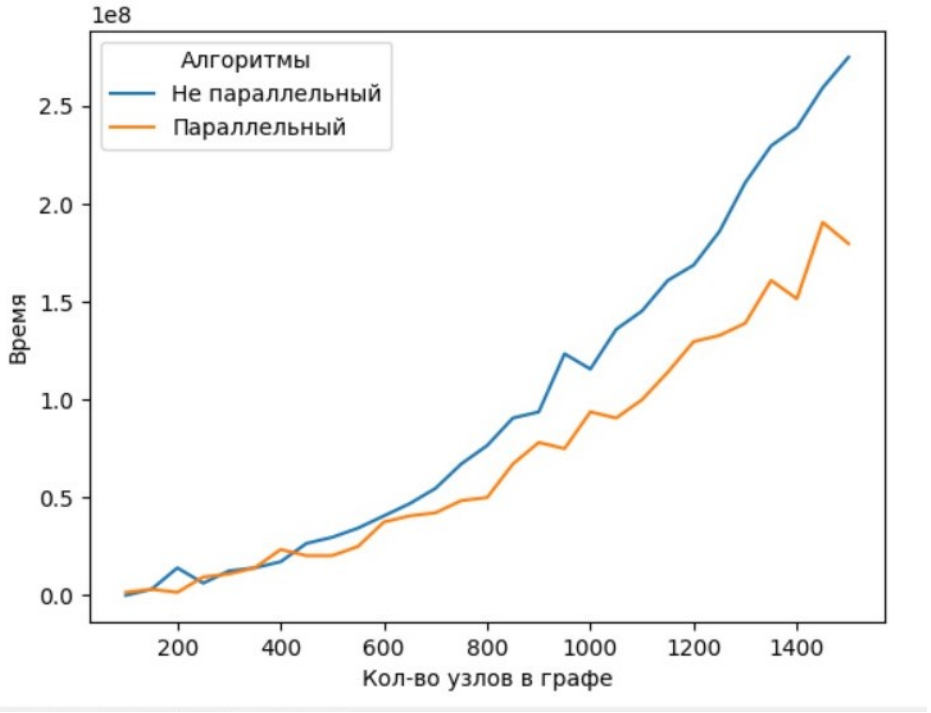

Рисунок 4.4 — График зависимости времени работы параллельного и непараллельного алгоритмов в зависимости от количества узлов для пяти потоков

4.3 Вывод

В результате экспериментов было получено, что при использовании параллельного метода при небольших размерах графа разница не является значительной, однако при использовании 8 потоков при размере 1500 узлов достигается ускорение в 1.5 раза.

Заключение

В процессе выполнения рубежного контроля был изучен алгоритм поиска в глубину. Был выполнен анализ алгоритма поиска в глубину и параллельного алгоитма поиска в глубину, представлены схемы алгоритмов и функциональная схема ПО. Алгоритмы были реализованы на языке Python в IDE Visual Studio Code. Помимо этого были проведены эксперименты с целью получить информацию о временной производительности алгоритмов. В результате эксперимента было получено, что при использовании параллельного метода при небольших размерах графа разница не является значительной, однако при использовании 8 потоков при размере 1500 узлов достигается ускорение в 1.5 раза. Целью данной лабораторной работы являлось изучение алгоритма поиска в глубину и разработка его параллельного варианта, что было успешно достигнуто.

Список использованных источников

1. Буркатовския Ю. Б. Теория графов. - Томск: Издательство Томского политехнического университета, 2014. - Т. 1. - С. 200.

2. Дистель Р. Теория графов. - Новосибирск: Издательство Института математики им. С. Л. Соболева СО РАН, 2002. - С. 336.

3. Кормен Т., Лейзерсон Ч., Ривест Р. Глава 22. Элементарные алгоритмы для работы с графами // Алгоритмы: построение и анализ(второе издание). - «Вильямс», 2005. - С. 622-632.

4. Документация к среде разработки Visual Studio Code [Электронный доступ], режим доступа: https://code.visualstudio.com/docs (дата обращения: 10.01.2022)

5. Документация языка Python [Электронный доступ], режим доступа: https://docs.python.org/3/index.html (дата обращения: 5.01.2022)