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

Кафедра информатики и программирования

РЕФЕРАТ

Алгоритмы поиска мостов в графе

студента 3 курса 342 группы
направления 02.03.02 Математическое обеспечение и администрирование информационных систем
факультета компьютерных наук и информационных технологий
Гаджимурадова Руслана Магомедовича

Зав. кафедрой
к.ф.-м.н., доцент М.В. Огнева

Саратов 2020

СОДЕРЖАНИЕ

ОПРЕДЕЛЕНИЯ

Связный граф — граф, в котором все вершины связаны.

Компонента связности графа — некоторое подмножество вершин графа, такое, что для любых двух вершин из этого подмножество существует путь из одной вершины в другую, и не существует пути из вершины этого подмножество в вершину не из этого подмножество.

Мост — ребро, удаление которого увеличивает количество компонент связности в графе.

Точка сочленения — вершина графа, в результате удаления которой вместе со всеми инцидентными ей ребрами количество компонент связности в графе возрастает.

ВВЕДЕНИЕ

Графы используются для решения большого количества задач в самых разных сферах жизни [1]. Во многих задачах важным условием является количество компонент связанности рассматриваемого графа, что непосредственно связано с понятием точки сочленения и моста. Например, при попытке оптимизировать некий процесс, представленный в виде графа, необходимо сократить число ребер, но не нарушить связанность графа, то есть, не повлиять на логику процесса и его результат. В таком случае на помощь могут прийти алгоритмы поиска моста в графе. На данный момент, применяются два алгоритма для решения поставленной задачи – алгоритм Тарьяна и алгоритм поиска мостов с помощью цепочной декомпозиции.

Цель реферата – исследование существующих алгоритмов поиска мостов в графе.

Поставленная цель определила следующие задачи:

1. Рассмотреть теоретические основы алгоритмов поиска мостов.
2. Реализовать данные алгоритмы и применить их на практике.

1. ТЕОРИЯ И ПОСТАНОВКА ЗАДАЧИ

Определение моста уже было дано выше, но, для удобства, повторим его еще раз. Мост — ребро, удаление которого увеличивает количество компонент связности в графе. Из определения очевидно, что количество мостов в графе непосредственно зависит от его структуры. В связи с этим стоит отдельно рассмотреть некоторые подклассы графов, такие как: деревья и леса.

Дерево — это связный ациклический граф [2]. Из определения следует, что число рёбер в дереве на единицу меньше числа вершин, а между любыми парами вершин имеется один и только один путь. Последнее, в рамках рассматриваемой работы, означает, что дерево – это граф, в котором каждое ребро является мостом, а точнее, графы, имеющие мостов в точности на единицу меньше числа вершин, называются деревьями, а графы, в которых любое ребро является мостом — лесом [3].

Важной открытой проблемой, связанной с мостами, является гипотеза о двойном покрытии циклами, высказанная Сеймуром и Секерешем, которая утверждает, что любой граф без мостов можно покрыть простыми циклами, содержащими каждое ребро дважды [4].

Формулировка задачи, рассматриваемой в данной работе, крайне проста: найти все мосты в заданном графе.

2. АЛГОРИТМ ТАРЬЯНА ПОИСКА МОСТОВ

Данный алгоритм для нахождения мостов в неориентированном графе был описан Робертом Тарьяном в 1974 году. Время работы алгоритма оценивается в $O(n+m)$, где n – количество вершин в графе, а m – количество ребер. Математическое описание алгоритма [5]:

1. Для каждой компоненты связности графа найти какое-либо остовное дерево T.
2. Перенумеровать вершины T в порядке обратного обхода.
3. В порядке возрастания номера вершины выполнить следующие действия:
   a. $D(v) := 1 + \sum_{(v,w)} D(w)$
   b. $L(v) := min\{N(v) - D(v) + 1\} \cup \{L(w) | (v,w)\} \cup \{N(w) | v \cdots w\}$
   c. $H(v) := max\{N(v)\} \cup \{H(w) | (v,w)\} \cup \{N(w) | v \cdots w\}$
   d. Пометить ребро (v,w) мостом, если $H(w) \leq N(w)$ и $L(w) > N(w) - D(w)$.

Где $D(v)$ – количество потомков вершины v в ориентированном дереве T.

На практике для обхода графа $G(V,E)$ необходимо будет использовать обход в глубину. Поэтому перейдем к более подробному описаю алгоритма Тарьяна.

Запустим обход в глубину из произвольной вершины графа. Находясь во время обхода в некой вершине $v \in V$, рассмотрим все инцидентные ребра и смежные вершины. Зафиксируем некоторую вершину to. Тогда, если текущее ребро (v, to) таково, что из вершины to и из любого её потомка в дереве обхода в глубину нет обратного ребра в вершину v или какого-либо её предка, то это ребро является мостом. В противном случае оно мостом не является. Таким образом проверяется наличие какого-ибо иного пути из v в to, кроме как спуск по ребру (v, to).

Для эффективной проверки данного условия необходимо воспользоваться «временем входа в вершину», вычисляемым алгоритмом поиска в глубину.

Пусть $tin[v]$ — это время захода поиска в глубину в вершину v. Дополнительный массив $fup[v]$ позволит ответить на вышеописанные запросы.

Время $fup[v]$ равно минимуму из времени захода в саму вершину v, времён захода в каждую вершину u, являющуюся концом некоторого обратного ребра (v, u), а также из всех значений $fup[to]$ для каждой вершины to, являющейся непосредственным сыном v в дереве поиска. Формально:

$$fup[v] = \min(tin[v], tin[u], fup[to])$$

$$\forall(v, u) - \text{обратных ребер и } \forall(v, to) - \text{ребер дерева обхода в глубину}$$

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что $fup[to] \leq tin[v]$. Если $fup[to] = tin[v]$, то это означает, что найдётся обратное ребро, приходящее точно в v, если же $fup[to] < tin[v]$, то это означает наличие обратного ребра в какого-либо предка вершины v.

Таким образом, если для текущего ребра (v, to), принадлежащего дереву обхода, выполняется $fup[to] > tin[v]$, то это ребро является мостом [6].

Алгоритм был реализован на языке C# как метод в обобщенном классе Graph<T>. Данный класс хранит граф в виде списка смежности в коллекции, представленной в листинге 1. Коллекция хранит вершины графа и списки смежных им ребер в виде пары – номер/название смежной вершины и вес ребра до нее.

Листинг 1 – коллекция, хранящая список смежности графа

```
private Dictionary<T, Dictionary<T, double>> _graph =
    new Dictionary<T, Dictionary<T, double>>();
```

Реализация описанного выше обхода в глубину представлена в качестве приватного метода класса Graph<T> и показана в листинге 2.

Листинг 2 – обход в глубину алгоритма Тарьяна

```
private void TarjanDFS(T v, T u, int timer,ref Dictionary<T, bool> used,
      ref Dictionary<T, int> tin, ref Dictionary<T, int> fup,
      ref List<KeyValuePair<T, T>> bridges)
{
      used[v] = true;

      // timer определяет «время захода» в вершину
      timer++;
      tin[v] = timer;
      fup[v] = timer;

      foreach (var adj in _graph[v])
      {
            var to = adj.Key;
            if (to.Equals(u))
            {
                  continue;
            }

            if (used[to])
            {
                  fup[v] = Math.Min(fup[v], tin[to]);
            }
            else
            {
                  TarjanDFS(to, v, timer, ref used, ref tin, ref fup, ref bridges);

                  fup[v] = Math.Min(fup[v], fup[to]);
                  if (fup[to] > tin[v])
                  {
                        bridges.Add(new KeyValuePair<T, T>(v, to));
                  }
            }
      }
}
```

Реализация инициализирующего метода вызова алгоритма Тарьяна представлена в листинге 3. Результатом работы данного метода будет список пар вершин, инцидентных ребру, являющегося мостом.

Листинг 3 – метод, инициализирующий вызов алгоритма Тарьяна

```
public List<KeyValuePair<T, T>> Tarjan()
{
	List<KeyValuePair<T, T>> bridges = new List<KeyValuePair<T, T>>();

	Dictionary<T, bool> used = new Dictionary<T, bool>();
	Dictionary<T, int> tin = new Dictionary<T, int>();
	Dictionary<T, int> fup = new Dictionary<T, int>();

	int timer = 0;
	foreach (var v in _graph)
	{
		used[v.Key] = false;
	}

	foreach (var v in _graph)
	{
		if (!used[v.Key])
		TarjanDFS(v.Key, v.Key, timer, ref used, ref tin, ref fup, ref bridges);
	}

	return bridges;
}
```

3. АЛГОРИТМ ПОИСКА МОСТОВ С ПОМОЩЬЮ ЦЕПОЧНОЙ ДЕКОМПОЗИЦИИ

Данный алгоритм поиска мостов основан на разложение графа на цепи. Цепочная декомпозиция позволяет получить не только все мосты, но и все точки сочленения графа, давая тем самым базу для проверки рёберной и вершинной 2-связности [3].

Цепочная декомпозиция — это специальный способ представления графа, позволяющий ответить на вопросы, описанные выше. Для осуществления декомпозиции необходимо дерево поиска в глубину, получаемое после обхода графа. Реализация нерекурсивного алгоритма обхода представлена в листинге 4. Стоит отметить, что дерево обхода представлено структурой *Dictionary<T, List<T>> treeDFS* и является выходным параметром данного метода. Также, в реализации обхода присутствует дополнительный цикл *while(d.Count < _graph.Count)*, который необходим в случае, если граф не является связанным, и требуется перейти к обходу следующей компоненты связанности графа.

Листинг 4 – реализация нерекурсивного поиска в глубину

```
public Dictionary<T, int> DFS(T v, out Dictionary<T, T> parents, out Dictionary<T,
List<T>> treeDFS)
{
	Dictionary<T, int> d = new Dictionary<T, int>(_graph.Count);
	parents = new Dictionary<T, T>();
	treeDFS = new Dictionary<T, List<T>>();

	Stack<T> stack = new Stack<T>();
	int step = 0;

	T tempV = v;
	stack.Push(v);
	parents[v] = v;

	// проверка, что просмотрены все вершины, следовательно, и все компоненты
	// смежности графа
	while (d.Count < _graph.Count)
	{
		while (stack.Count != 0)
		{
			// проверка, что текущая вершина не была уже пройдена
			if (!d.ContainsKey(stack.Peek()))
			{
				if (_graph[tempV].ContainsKey(stack.Peek()))
```

```
                    {
                        parents[stack.Peek()] = tempV;
                    }

                    // добавление текущей вершины в список смежности ее предка
                    // в дереве обхода в глубину
                    if (!stack.Peek().Equals(tempV))
                    {
                        treeDFS[parents[stack.Peek()]].Add(stack.Peek());
                    }

                    tempV = stack.Pop();
                    d.Add(tempV, step);
                    treeDFS.Add(tempV, new List<T>());

                    // обход вершин в порядке следования их ключа
                    foreach (var u in _graph[tempV]
                        .OrderByDescending(pair => pair.Key))
                    {
                        if (!d.ContainsKey(u.Key))
                        {
                            stack.Push(u.Key);
                        }

                        if (!parents.ContainsKey(u.Key))
                        {
                            parents.Add(u.Key, tempV);
                        }
                    }

                    step++;
                }
                else
                {
                    // принудительное удалений вершины из стека, так как
                    // она была повторно добавлена из некой другой вершины по
                    // пути обхода и уже пройдена
                    stack.Pop();
                }
            }

            // если существуют не просмотренные вершины, то переходим к первой из них
            if (d.Count < _graph.Count)
            {
                tempV = _graph.Where(pair => !d.ContainsKey(pair.Key)).First().Key;
                stack.Push(tempV);
                parents[tempV] = tempV;
                step = 0;

            }
        }
```

```
        return d;
}
```

Зная дерево обхода, опишем алгоритм цепочной декомпозиции:

1. Все вершины помечаются как не посещенные.
2. Просматриваются все вершины $v \in V$ в восходящем порядке номеров обхода. Для каждой вершины v рассматривается каждое инцидентное ей обратное ребро – ребро, не принадлежащее дереву обхода.
    a. Вдоль каждого выбранного ребра следуем к корню дерева обхода.
    b. Пройденные вершины добавляются в цепочку и помечаются как посещённые.
    c. При попадании в посещенную вершину, алгоритм прерывается, полученная цепочка добавляется в список цепочек и происходит возврат к пункту 2.

Результирующий список цепочек будет содержать как пути, начинающиеся в v, так и циклы. Полученный список цепочек и является цепочной декомпозицией графа.

Реализация метода цепочной декомпозиции представлена в листинге 5. Одним из ключевых моментов в реализации алгоритма является осуществление проверки направления обхода. Из описания алгоритма, необходимо при переходе от выбранной вершины вдоль обратного ребра осуществлять обход в сторону корня дерева поиска. Для данной проверки используется проверка условия *dfs[parents[temp]] < dfs[temp]*, где *dfs[]* – массив расстояний до вершин графа от корня дерева обхода, а *temp* – просматриваемая вершина.

Листинг 5 – реализация метода цепочной декомпозиции на основе дерева обхода в глубину

```
public List<Queue<T>> ChainDecomposition()
{
        List<Queue<T>> chains = new List<Queue<T>>();

        Dictionary<T, T> parents;
        Dictionary<T, List<T>> treeDFS;
        Dictionary<T, int> dfs = DFS(_graph.First().Key, out parents, out treeDFS);
```

```
        // просмотр всех вершин в графе
        foreach (var v in _graph)
        {
                // g - список смежных вершин, не принадлежащих дереву обхода
                var g = v.Value.Where(pair => !treeDFS[v.Key].Contains(pair.Key)
                    && !treeDFS[pair.Key].Contains(v.Key));
                foreach (var adj in g)
                {
                        var chain = new Queue<T>();
                        chain.Enqueue(v.Key);

                        T temp = adj.Key;
                        // просмотр вершин в сторону корня пока не встретится пройденная
                        // условие dfs[parents[temp]] < dfs[temp] позволяет осуществлять
                        // ход в сторону корня
                        while (!temp.Equals(v.Key) && dfs[parents[temp]] < dfs[temp])
                        {
                                chain.Enqueue(temp);
                                temp = parents[temp];
                        }
                        chain.Enqueue(temp);

                        if (chain.Count > 1)
                        {
                                chains.Add(chain);
                        }
                }
        }

        return chains;
}
```

Разложив граф на цепочки, можно проанализировать его на наличие мостов, воспользовавшись следующими свойствами цепочной декомпозиции [3]:

1) Ребро графа является мостом в том и только в том случае, когда данное ребро не содержится ни в одной цепочке из декомпозиции.
2) Граф является рёберно 2-связным в том и только в том случае, когда цепочки содержат все рёбра графа.
3) Если граф является рёберно 2-связным, цепочная декомпозиция является ушной декомпозицией [7].
4) Граф является вершинно-2-связным в том и только в том случае, когда граф имеет минимальную степень 2 и первая цепочка в декомпозиции

является единственным циклом в разложении.

Реализация метода поиска мостов на основе цепочной декомпозиции и свойства 1) представлена в листинге 6.

Листинг 6 – реализация метода поиска мостов с помощью цепочной декомпозиции

```
public List<KeyValuePair<T, T>> FindBridgesByChains()
{
	List<KeyValuePair<T, T>> bridges = new List<KeyValuePair<T, T>>();

	var decompositions = ChainDecomposition();

	foreach (var v in _graph)
	{
		foreach (var adj in v.Value)
		{
			// если граф не направленный, то необходимо провести дополнительную
			// проверку, чтобы избежать повторения ребер в ответе
			if (_directed)
			{
				// проверка, что ни одна цепочка не содержит данной вершины,
				// а следовательно, и инцидентного ей моста
				if (!decompositions.Any(l => l.Contains(adj.Key)))
				{
					bridges.Add(new KeyValuePair<T, T>(v.Key, adj.Key));
				}
			}
			else
			{
				// аналогичная проверка, но с учетом требования избежать
				// повторений в ответе
				if (!decompositions.Any(l => l.Contains(adj.Key))
				   && !bridges.Contains(new
					KeyValuePair<T, T>(adj.Key, v.Key)))
				{
					bridges.Add(new KeyValuePair<T, T>(v.Key, adj.Key));
				}
			}
		}
	}

	return bridges;
}
```

4. ПРИМЕРЫ

Пример №1. В качестве входного графа дан неориентированный граф, представленный на рисунке 1. Очевидно, что мостами в данном графе являются ребра (4,5) и (5,6).

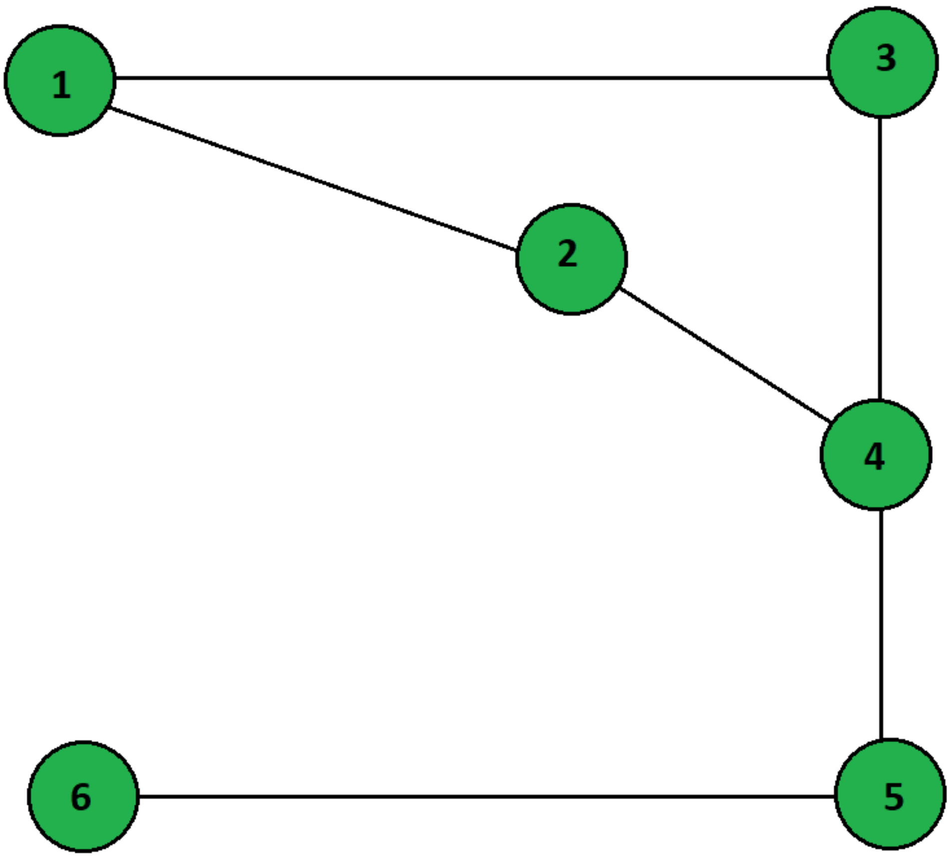

Рисунок 1 – пример не ориентированного графа

Программное представление графа и результаты работы двух алгоритмов показаны на рисунке 2. Стоит отметить, что в отличии от алгоритма Тарьяна, алгоритм, основанный на цепочной декомпозиции, выдает ребра как бы в направлении следования из вершины 1 в вершину 6. Однако в данном случае, этим отличием можно пренебречь, так как граф является неориентированным.

```
Входной граф
не ориентированный | не взвешенный

1 | {2} {3}
2 | {4} {1}
3 | {1} {4}
4 | {3} {5} {2}
5 | {6} {4}
6 | {5}

Алгоритм Тарьяна:
5 -> 6
4 -> 5

Алгоритм цепочной декомпозиции:
4 -> 5
5 -> 6
```

Рисунок 2 – результат работы алгоритмов на неориентированном графе

Пример №2. В качестве входного графа дан ориентированный граф, представленный на рисунке 3.

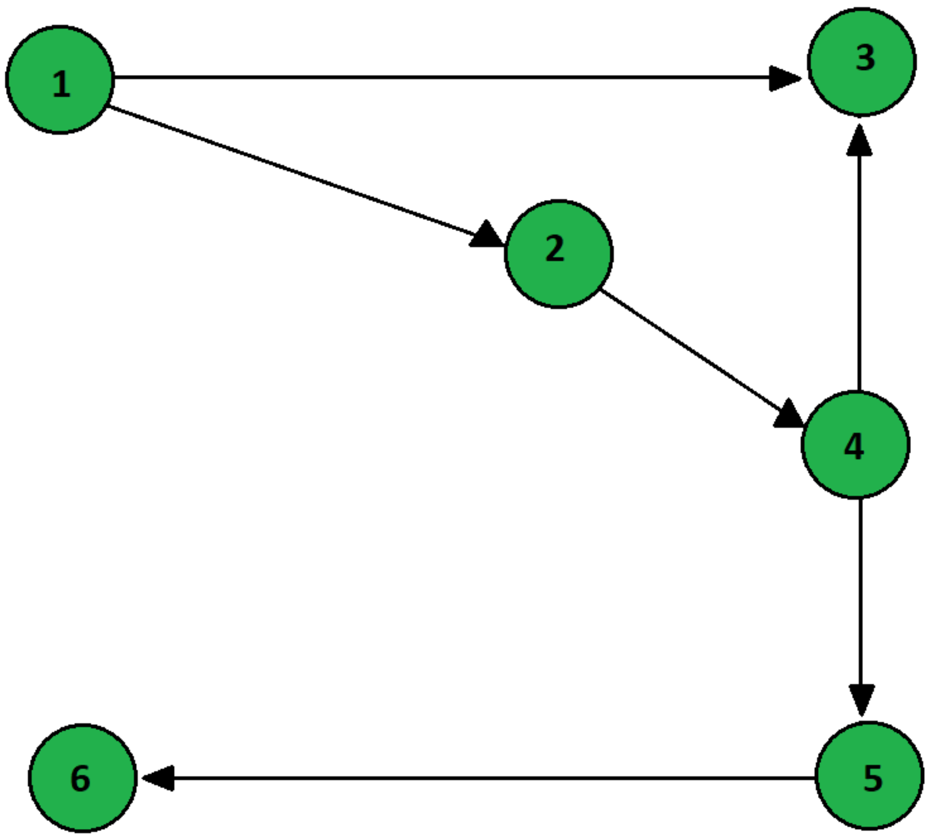

Рисунок 3 – пример ориентированного графа

Как и в предыдущем примере, мостами в данном графе являются ребра (4,5) и (5,6). Однако алгоритм Тарьяна не сможет корректно отработать на данном графе. Результат работы алгоритмов и программное представление графа показаны на рисунке 4.

```
Входной граф
ориентированный | не взвешенный

1 | {2} {3}
2 | {4}
3 |
4 | {3} {5}
5 | {6}
6 |

Алгоритм Тарьяна:
4 -> 3
5 -> 6
4 -> 5
2 -> 4
1 -> 2

Алгоритм цепочной декомпозиции:
4 -> 5
5 -> 6
```

Рисунок 4 – результат работы алгоритмов на ориентированном графе

Как видно на рисунке 4, алгоритм Тарьяна некорректно работает с орграфами, считая все ребра графа мостами. В отличии от него, алгоритм на основе цепочной декомпозиции может найти мосты даже в орграфе. Данное преимущество объясняется тем, что декомпозиция происходит после построения дерева обхода в глубину, что позволяет корректно распознавать пути в орграфе.

ЗАКЛЮЧЕНИЕ

Поставленные в данной работе цели и задачи были успешно выполнены, что позволило изучить алгоритмы поиска мостов в графах и разобрать их реализацию. Можно сделать вывод о том, что метод поиска мостов с помощью цепочной декомпозиции является более универсальным, однако, несмотря на простоту его формального описания, он сложнее в реализации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Графы и области их применения [Электронный ресурс]: статья – URL: https://school-science.ru/4/7/33615 (дата обращения 20.11.2020).
2. Дерево (теория графов) [Электронный ресурс]: свободная энциклопедия / текст доступен по лицензии Creative Commons Attribution-ShareAlike; Wikimedia Foundation, Inc, некоммерческой организации. – URL: https://ru.wikipedia.org/wiki/Дерево_(теория_графов) (дата обращения 02.12.2020). – Последнее изменение страницы: 06:02, 11 Октября 2020 года.
3. Мост (теория графов) [Электронный ресурс]: свободная энциклопедия / текст доступен по лицензии Creative Commons Attribution-ShareAlike; Wikimedia Foundation, Inc, некоммерческой организации. – URL: https://ru.wikipedia.org/wiki/Мост_(теория_графов) (дата обращения 25.11.2020). – Последнее изменение страницы: 14:01, 27 Июля 2019 года.
4. Двойное покрытие циклами [Электронный ресурс]: свободная энциклопедия / текст доступен по лицензии Creative Commons Attribution-ShareAlike; Wikimedia Foundation, Inc, некоммерческой организации. – URL: https://ru.wikipedia.org/wiki/Двойное_покрытие_циклами (дата обращения 01.12.2020). – Последнее изменение страницы: 15:38, 02 Января 2020 года.
5. Алгоритм Тарьяна поиска «мостов» в графе [Электронный ресурс]: статья – URL: https://algowiki-project.org/ru/Алгоритм_Тарьяна_поиска_«мостов»_в_графе (дата обращения 02.12.2020). – Последнее изменение страницы: 16:58, 14 Марта 2018 года.
6. Поиск мостов [Электронный ресурс]: статья – URL: https://e-maxx.ru/algo/bridge_searching (дата обращения 02.12.2020). – Последнее изменение страницы: 11:23, 23 Августа 2011 года.
7. Ушная декомпозиция [Электронный ресурс]: свободная энциклопедия / текст доступен по лицензии Creative Commons Attribution-ShareAlike; Wikimedia Foundation, Inc, некоммерческой организации. – URL: https://ru.wikipedia.org/wiki/Ушная_декомпозиция (дата обращения 03.12.2020). – Последнее изменение страницы: 16:25, 21 Февраля 2020 года.