МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра дискретной математики и информационных технологий

АЛГОРИТМЫ НА ГРАФАХ И ИХ ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

КУРСОВАЯ РАБОТА

студента 2 курса 221 группы
направления 09.03.01 — Информатика и вычислительная техника
факультета КНиИТ
Устюшина Богдана Антоновича

Научный руководитель
к. ф.-м. н., доцент _____ Л. Б. Тяпаев

Заведующий кафедрой
к. ф.-м. н., доцент _____ Л. Б. Тяпаев

Саратов 2023

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Теория графов – активно развивающаяся отрасль дискретной математики, которая имеет большое количество приложений. Так, графы помогают визуализировать отношения – также один из фундаментальных инструментов дискретной математики. Как следствие, с помощью графов также описываются конечные автоматы.

Дадим простое объяснение центральному понятию курсовой работы – графу. По своей сути, он является множеством вершин, которые как-либо связаны между собой. Из такого определения можно понять, что сфера применения графов, особенно в прикладных задачах, по-настоящему велика.

Теория графов как раздел прикладной математики «открывалась» несколько раз. Ключ к пониманию теории графов и её комбинаторной сущности отражены в словах Джеймса Сильвестра: «Теория отростков» (англ. ramification) – одна из теорий чистого обобщения, для неё не существенны ни размеры, ни положение объекта; в ней используются геометрические линии, но они относятся к делу не больше, чем такие же линии в генеалогических таблицах помогают объяснять законы воспроизведения».

Диаграмма одной из разновидностей графа – дерева – использовалась издавна (конечно, без понимания, что это «граф»). Генеалогическое древо применялось для наглядного представления родственных связей.

Однако первым открытием теории графов является задача о кёнигберских мостах, датированной 1736 годом. Это старинная математическая задача, в которой спрашивалось, как можно пройти по всем семи мостам центра старого Кёнигсберга, не проходя ни по одному из них дважды. Впервые была решена в статье, датированной 1736 годом, математиком Леонардом Эйлером, который доказал, что это невозможно, и по ходу доказательства изобрёл эйлеровы циклы. Решение Эйлером задачи о кёнигсбергских мостах явилось первым в истории применением теории графов, но без использования термина «граф» и без рисования диаграмм графов.

Видно, что модель, заложенная Эйлером (даже без использования строгих определений), имела огромную описательную способность: с помощью графов возможно описание сложных объектов в большом количестве областей. Так, структуры графов и деревьев позже «переоткрывались» учёными в самых разных областях:

— В 1847 году немецкий физик Густав Кирхгоф фактически разработал теорию деревьев при решении системы уравнений для нахождения величины силы тока в каждом контуре электрической цепи. Кирхгоф фактически изучал вместо электрической цепи соответствующий ей граф и показал, что для решения системы уравнений нет необходимости анализировать каждый цикл графа, достаточно рассмотреть только независимые циклы, определяемые любым остовным деревом графа.
— В 1857 году английский математик Артур Кэли, занимаясь практическими задачами органической химии, открыл важный класс графов – деревья.
— В 1859 году ирландский математик сэр Уильям Гамильтон придумал игру «Вокруг света». В этой игре использовался додекаэдр, каждая из 20 вершин которого соответствовала известному городу. От играющего требовалось обойти «вокруг света», то есть пройти по рёбрам додекаэдра так, чтобы пройти через каждую вершину ровно один раз (что впоследствии получило название «гамильтонов цикл») [1].

Очевидно, что изучение теории графов является актуальным для любого IT-специалиста. Сегодня графы используются в изучении работы распределённых систем, примером которой является Интернет, при моделировании дорог и путей транспортного соединения, а также при построении моделей в других науках (как в вышеописанных примерах). Из этого следует личная актуальность: изучений некоторых определений и алгоритмов на графах, часть которых выходят за рамки программы курса дискретной математики.

Также в дальнейшем реализованный на языке C# класс может быть преобразован в библиотеку для выполнения задач по компьютерной обработке графов, что является актуальным в силу вышеописанной важности теории графов для решения современных прикладных задач.

В ходе данной курсовой работы поставлены следующие задачи:

1. Раскрыть теоретическую основу наиболее важных алгоритмов на графах
2. Описать сами алгоритмы с приведением доказательства корректности алгоритмов и вычислительной сложности
3. Реализовать данные алгоритмы на языке программирования C# в виде отдельного класса

При теоретическом описании алгоритмов сразу будет указываться идея алгоритма, а также их псевдокод, который далее будет реализован на языке программирования. После чего будет описана теоретическое обоснование его правильности.

1 Теоретические сведения из теории графов

Поскольку теория графов оперирует большим количеством достаточно небольших понятий, целесообразно отдельно вынести лишь их часть – самую значимую.

Определение 1. ***Простой граф*** $G(V, E)$ *есть совокупность двух множеств – непустого множества* V *и множества* E *неупорядоченных пар различных элементов множества* V *. Множество* V *называется множеством вершин, множество* E *называется множеством рёбер:*

$$G(V, E) = \langle V, E \rangle, \quad V \neq \varnothing, \quad E \subseteq V \times V, \quad \{v, v\} \notin E, \quad v \in V$$

Определение 2. ***Ориентированный граф*** $G(V, E)$ *есть совокупность двух множеств – непустого множества* V *и множества* E *дуг или упорядоченных пар различных элементов множества* V*.*

Определение 3. ***Связный граф*** *– это граф, у которого две любые вершины соединены путём.* ***Компонента связности****, или компонента, графа — это максимальный связный подграф графа.*

Далее рассмотрим наиболее применимые способы хранения графов:

Определение 4. ***Матрица смежности*** *– таблица, где как столбцы, так и строки соответствуют вершинам графа. В каждой ячейке этой матрицы записывается число, определяющее наличие связи от вершины-строки к вершине-столбцу (либо наоборот).*

Определение 5. ***Список смежности*** *– список, где каждой вершине графа соответствует строка, в которой хранится список смежных вершин. Такая структура данных не является таблицей в обычном понимании, а представляет собой «список списков».*

Определение 6. ***Маршрутом*** *в графе называют конечную последовательность вершин, в которой каждая вершина (кроме последней) соединена со следующей в последовательности вершиной ребром.* ***Цепью*** *называется маршрут без повторяющихся рёбер.* ***Простой цепью*** *называется маршрут без повторяющихся вершин (откуда следует, что в простой цепи нет повторяющихся рёбер).*

Определение 7. ***Цикл*** *в графе – это подграф, представляющий собой замкнутую последовательность различных вершин, в которой каждая вершина соединена со следующей ребром [2, 3].*

2 Теоретическое обоснование алгоритмов

2.1 Обход (поиск) в глубину

Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них. При отсутствии таковых алгоритм для данной вершины завершается и рекурсия сворачивается.

Вход:

Граф $G(V, E)$, внутри которого следует совершить обход

Выход:

Граф, в каждой вершине которого мы побывали один раз

Алгоритм:

procedure DFS(u)
    visited[u] = true
    for $\forall v : (u, v) \in G$ **do**
        if visited[v] = false **then**
            DFS(v)
        end if
    end for
end procedure

visited[n, false] – массив булевых переменных, указывающих, посетили ли мы данную вершину

for i = 1 to n **do**
    if !visited[u] **then**
        DFS(u)
    end if
end for

Утверждение 1. *Процедура поиска в глубину из алгоритма обхода в глубину помечает все достижимые вершины из вершины, из которой он запускался, и только их.*

Доказательство. Необходимость. В каждый момент времени все вершины достижимы из предыдущих, поскольку в течение всего алгоритма сохраняется его инвариант: множество помеченных вершин достижимо из начальной вершины. Этот инвариант поддерживается, поскольку при переходе из вер-

шины множества достижимых в какую-либо другую мы используем ребро → новая вершина обладает путём из стартовой → она достижима.

Достаточность. Пусть из начальной вершины s достижима вершина v. Пусть алгоритм DFS не вошёл в вершину v. Значит, по пути, по которому v достижима из s алгоритм DFS остановился и не пошёл в следующую вершину пути: пускай это будет вершина w. Но DFS запускается рекурсивно от всех смежных с ним не отмеченных вершин. Если w уже отмечена, то он в действительности обошёл эту вершину. Если же не отмечена, то он обязан был в неё зайти и запустить алгоритм DFS → противоречие. □

Данный алгоритм имеет цикл по всем вершинам графа для того, чтобы при наличии нескольких компонент связности обойти все вершины. Поэтому его можно модифицировать, добавив составление компонент связности неориентированного графа (для поиска компонент связности ориентированного графа существуют отдельные алгоритмы). Так, в реализации лишь потребуется добавить в последний цикл лист, который после каждого обхода сообщал бы о добавленных вершинах. По утверждению 1, после каждого вызова процедуры мы будем получать вершины из ровно одной компоненты связности, так как он помечает лишь достижимые (так и звучит определение компоненты связности неориентированного графа).

Оценим время работы обхода в глубину. Процедура DFS вызывается от каждой вершины не более одного раза, а внутри процедуры рассматриваются все такие ребра $\{e \mid begin(e) = u\}$. Всего таких ребер для всех вершин в графе $O(E)$, следовательно, время работы алгоритма оценивается как $O(V+E)$ [4].

2.2 Поиск в ширину

Идея алгоритма схожа с поиском в глубину: отличие лишь в порядке вызова метода. Поиск в ширину осуществляется с помощью очереди, в которую добавляются все смежные с данной вершины, после чего для каждого элемента из очереди рекурсивно вызывается обход в ширину.

Вход:

Граф $G(V, E)$, внутри которого следует совершить обход

Вершина u – стартовая вершина

Выход:

Граф, в каждой вершине которого мы побывали один раз

Алгоритм:

$visited[n, false]$ – массив булевых переменных, указывающих, посетили ли мы данную вершину

Добавим в очередь q вершину u

while q не пуста **do**

$u =$ первый элемент в очереди

$visited[u] = true$

for $\forall v : (u, v) \in G$ **do**

Добавим в q вершину v

end for

end while

Утверждение 2. *Алгоритм обхода в ширину помечает все достижимые вершины из вершины, из которой он запускался, и только их.*

Доказательство аналогично утверждению 1.

Операции внесения в очередь и удаления из неё требует $O(1)$ времени, поэтому общее время операций с ней составляет $O(V)$. Поскольку каждый список смежности сканируется только при удалении соответствующей вершины из очереди, каждый список сканируется не более одного раза. Так как сумма длин всех списков смежности равна E, то общее время работы процедуры составляет $O(V + E)$ [5].

2.3 Алгоритм Дейкстры

Алгоритм Дейкстры – алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году. Находит кратчайшие пути от одной из вершин графа до всех остальных. Алгоритм работает только для графов без рёбер отрицательного веса.

В алгоритме поддерживается множество вершин U, для которых уже вычислены длины кратчайших путей до них из s. На каждой итерации основного цикла выбирается вершина $u \notin U$, которой на текущий момент соответствует минимальная оценка кратчайшего пути. Вершина u добавляется в множество U и производится релаксация всех исходящих из неё рёбер.

Разумеется, обычно нужно знать не только длины кратчайших путей, но и получить сами пути. Покажем, как сохранить информацию, достаточную для последующего восстановления кратчайшего пути из s до любой вершины.

Для этого достаточно так называемого массива предков: массива $p[]$, в котором для каждой вершины $v \neq s$ хранится номер вершины $p[v]$, являющейся предпоследней в кратчайшем пути до вершины v. Здесь используется тот факт, что если мы возьмём кратчайший путь до какой-то вершины v, а затем удалим из этого пути последнюю вершину, то получится путь, оканчивающийся некоторой вершиной $p[v]$, и этот путь будет кратчайшим для вершины $p[v]$. Итак, если мы будем обладать этим массивом предков, то кратчайший путь можно будет восстановить по нему, просто каждый раз беря предка от текущей вершины, пока мы не придём в стартовую вершину s – так мы получим искомый кратчайший путь, но записанный в обратном порядке. Итак, кратчайший путь P до вершины v равен:

$$P = (s, \ldots, p[p[p[v]]], p[p[v]], p[v], v)$$

Вход:

Граф $G(V, E)$, внутри которого следует совершить обход

Вершина u – стартовая вершина

Выход:

Одномерный массив d – массив кратчайших расстояний от заданной вершины u до всех остальных

Алгоритм:

$d[n]$ – массив кратчайших расстояний от заданной вершины до v.

$visited[n]$ – массив посещённых вершин алгоритмом Дейкстры (за основу фактически берётся обход в глубину)

$prev[n]$ – массив вершин, сохраняющий обратные пути

for $v \in V$ **do**
    $d[v] = \inf$
    $visited[v] = false$
end for
$d[u] = 0$
for $i \in V$ **do**
    $v = null$
    for $j \in V$ **do**
        if !visited[j] **and** (v == null **or** $d[j] < d[v]$) **then**
            $v = j$

```
        end if
    end for
    if d[v] == inf then
        break
    end if
    for (e : (v, e) ∈ G) do          ▷ произведём релаксацию по всем рёбрам,
        if d[v] + len(e) < d[e] then          ▷ исходящим из v
            d[e] = d[v] + e.len
            prev[e] = v
        end if
    end for
end for
```

Теорема 1. *Пусть $G = (V, E)$ – ориентированный взвешенный граф, вес рёбер которого неотрицателен, s – стартовая вершина. Тогда после выполнения алгоритма Дейкстры $d(u) = \rho(s, u)$ для всех u, где $\rho(s, u)$ – длина кратчайшего пути из вершины s в вершину u.*

Доказательство. Докажем по индукции, что в момент посещения любой вершины u, $d(u) = \rho(s, u)$.

— На первом шаге выбирается s для неё выполнено: $d(s) = \rho(s, s) = 0$.

— Пусть для n первых шагов алгоритм сработал верно и на $n + 1$ шагу выбрана вершина u. Докажем, что в этот момент $d(u) = \rho(s, u)$. Для начала отметим, что для любой вершины v, всегда выполняется $d(v) \geqslant \rho(s, v)$ (алгоритм не может найти путь короче, чем кратчайший из всех существующих). Пусть P — кратчайший путь из s в u, v — первая непосещённая вершина на P, z — предшествующая ей (следовательно, посещённая). Поскольку путь P кратчайший, его часть, ведущая из s через z в v, тоже кратчайшая, следовательно $\rho(s, v) = \rho(s, z) + w(z, v)$. По предположению индукции, в момент посещения вершины z выполнялось $d(z) = \rho(s, z)$, следовательно, вершина v тогда получила метку не больше чем $d(z) + w(z, v) = \rho(s, z) + w(z, v) = \rho(s, v)$, следовательно, $d(v) = \rho(s, v)$. С другой стороны, поскольку сейчас мы выбрали вершину u, её метка минимальна среди непосещённых, то есть $d(u) \leqslant d(v) = \rho(s, v) \leqslant \rho(s, u)$, где второе неравенсто верно из-за ранее упомянутого определения вершины v в качестве первой непосещённой

вершины на P, то есть вес пути до промежуточной вершины не превосходит веса пути до конечной вершины вследствие неотрицательности весовой функции. Комбинируя это с $d(u) \geqslant \rho(s,u)$, имеем $d(u) = \rho(s,u)$, что и требовалось доказать.

— Поскольку алгоритм заканчивает работу, когда все вершины посещены, в этот момент $d(u) = \rho(s,u)$ для всех u.

□

В реализации алгоритма присутствует функция выбора вершины с минимальным значением d и релаксация по всем рёбрам для данной вершины. Асимптотика работы зависит от реализации. Рассмотрим реализацию с хранением d в массиве (также возможно хранение в двоичной куче или фиббоначчиевой куче). n раз осуществляем поиск вершины с минимальной величиной d среди $O(n)$ непомеченных вершин и m раз проводим релаксацию за $O(1)$. Для плотных графов ($m \approx n^2$) данная асимптотика является оптимальной [6, 7].

2.4 Алгоритм Флойда-Уоршелла

Обобщим задачу, выполняемую алгоритмом Дейкстры: теперь попытаемся найти кратчайшее расстояние из всех вершин графа. Для этого применим метод динамического программирования.

На каждом шаге алгоритма, мы будем брать очередную вершину (пусть её номер — i) и для всех пар вершин u и v вычислять $d_{uv}^{(i)} = \min(d_{uv}^{(i-1)}, d_{ui}^{(i-1)} + d_{iv}^{(i-1)})$. То есть, если кратчайший путь из u в v, содержащий только вершины из множества $1 \dots i$, проходит через вершину i, то кратчайшим путем из u в v является кратчайший путь из u в i, объединенный с кратчайшим путем из i в v. В противном случае, когда этот путь не содержит вершины i, кратчайший путь из u в v, содержащий только вершины из множества $1 \dots i$ является кратчайшим путем из u в v, содержащим только вершины из множества $1 \dots i-1$.

Однако необязательно хранить двумерный массив для каждой итерации алгоритма: достаточно использовать один двумерный массив, поскольку $(u,v) \leqslant d_{uv} \leqslant d_{uv}^{(i)}$.

Утверждение 3. *Первое неравенство из двойного неравенства выполняется.*

Доказательство. Докажем первое неравенство от противного.

Пусть неравенство было нарушено, рассмотрим момент, когда оно было нарушено впервые. Пусть это была i-ая итерация и в этот момент изменилось значение d_{uv} и выполнилось $\rho(u,v) > d_{uv}$. Так как d_{uv} изменилось, то $d_{uv} = d_{ui} + d_{iv} \geqslant$ (так как ранее$\forall u, v \in V : \rho(u,v) \leqslant d_{uv}) \geqslant \rho(u,i) + \rho(i,v) \geqslant$ (по неравенству треугольника) $\geqslant \rho(u,v)$.

Итак, $d_{uv} \geqslant \rho(u,v)$ — противоречие. □

Утверждение 4. *Второе неравенство из двойного неравенства выполняется.*

Доказательство. Докажем второе неравенство индукцией по итерациям алгоритма. Пусть также d'_{uv} – значение d_{uv} сразу после $i-1$ итерации. Покажем, что $d_{uv} \leqslant d^{(i)}_{uv}$, зная, что $d'_{uv} \leqslant d^{(i?1)}_{uv}$.

Рассмотрим два случая:

1. Значение $d^{(i)}_{uv}$ стало меньше, чем $d^{(i-1)}_{uv}$. Тогда $d^{(i)}_{uv} = d^{(i-1)}_{ui} + d^{(i-1)}_{iv} \geqslant$ (выполняется на шаге $i-1$, по индукционному предположению) $\geqslant d'_{ui} + d'_{iv} \geqslant$ (в силу выполнения 7-ой строчки алгоритма на i-ой итерации и невозрастания элементов массива d) $\geqslant d_{uv}$.
2. В ином случае всё очевидно: $d^{(i)}_{uv} = d^{(i-1)}_{uv} \geqslant d'_{uv} \geqslant d_{uv}$, и неравенство тривиально.

□

Таким образом, в алгоритме возможно использование двумерного, а не трёхмерного массива. [8]

Также алгоритм Флойда легко модифицировать таким образом, чтобы он возвращал не только длину кратчайшего пути, но и сам путь. Для этого достаточно завести дополнительный массив next, в котором будет храниться номер вершины, в которую надо пойти следующей, чтобы дойти из u в v по кратчайшему пути. Если находиться в j, а требуется попасть в i, то нужно пойти из j в prev[i.j]. По своей сути схоже с n массивами из алгоритма Дейкстры.

Вход:

Граф $G(V,E)$, внутри которого следует совершить обход

Выход:

Массив d – массив кратчайших расстояний от вершины i до j

Массив путей $prev[n, n]$ – двумерный массив, показывающий, в какую вершину нужно вернуться

Алгоритм:

Массив $d[n, n]$ – массив кратчайших расстояний от вершины i до j. Изначально массив заполняется

for $i \in \{1 \dots n\}$ **do**
  for $u \in V$ **do**
    for $v \in V$ **do**
      Присваиваем переменной newDistance расстояние через текущую вершину i $newDistance = d[u, i] + d[i, v]$
      if newDistance $<$ d[u,v] **then**
        $d[u, v] = newDistance)$
        $prev[u, v] = prev[i, v]$
      end if
    end for
  end for
end for

Нетрудно понять, что сложность алгоритма составляет $O(n^3)$. Алгоритм Дейкстры, несмотря на то что при выполнении от каждой вершины также занимает временную сложность $O(n^3)$ является менее оптимальным, поскольку в алгоритме Флойда память выделяется только один раз.

2.5 Алгоритмы поиска сильной связности

Рассмотрим два алгоритма поиска компонент сильной связности в ориентированном графе. Первый из них работает за $O(n^3)$ и достаточно тривиален, опираясь на понятие транзитивного замыкания.

1. При помощи транзитивного замыкания проверяем, достижима ли t из, s, и s из t $\forall(s, t)$.
2. Затем определяем такой неориентированный граф, в котором для каждой такой пары содержится ребро, то есть по строим неориентированный граф транзитивного замыкания.
3. Поиск компонент связности такого неориентированного графа даёт компоненты сильной связности орграфа.

Транзитивное замыкание графа получаем благодаря алгоритму Флойда-

Уоршелла, затем, если для вершин u, v $M[u][v] = M[v][u] = 1$, то в новом неориентированном графе получаем ребро между этими двумя вершинами. После этого с помощью алгоритма поиска в глубину получаем компоненты связности неориентированного графа, эквивалентные изначальному [9, 10].

Также существует три алгоритма, решающих данную задачу за линейное время. Это алгоритмы Косарайю, поиска компонент сильной связности с двумя стеками и Тарьяна. Рассмотрим алгоритм Косарайю.

Для описания данного алгоритма введём несколько понятий:

Определение 8. ***Такт DFS*** *из вершины v – обход (в глубину) всех вершин графа, достижимых из v. Такт можно интерпретировать как рекурсивный вызов функции. Такт обработки вершины, у которой нет соседей, будет равняться 1.*

Время выхода вершины *– число, соответствующее времени выхода рекурсии алгоритма DFS из вершины. Притом, изначально счётчик времени 0, увеличивается он лишь в двух случаях:*

- *Начало нового такта DFS*
- *Прохождение по ребру (при том, не важно, рекурсивный проход или нет)*

Алгоритм Косарайю состоит из трёх шагов:

1. Выполнить поиск в глубину (DFS), пока не будут «помечены» все вершины. Вершина считается «помеченной», когда ей присвоено время выхода из рекурсии. На программном уровне возможно просто добавление в момент выхода из рекурсии
2. Инвертировать исходный граф
3. Выполнить DFS в порядке убывания пометок вершин

Полученные деревья каждого такта DFS последнего шага являются компонентами сильной связности. Докажем корректность алгоритма.

Теорема 2. *Вершины u и v сильно связаны тогда и только тогда, когда после выполнения алгоритма они принадлежат одному дереву третьего шага алгоритма.*

Доказательство. Необходимость. Если вершины u и v были сильно связаны в графе G, на третьем этапе будет найден путь из одной вершины в другую,

так как на первом шаге был найден путь $u \to v$, а на третьем – путь $v \to u$. Это означает, что по окончании алгоритма обе вершины лежат в одном дереве. Достаточность Вершины u и v лежат в одном и том же дереве поиска в глубину на третьем шаге алгоритма. Значит, они обе достижимы из корня r этого дерева.

Вершина r была рассмотрена на третьем шаге раньше всех, значит время выхода из неё на первом шаге больше, чем время выхода из вершин u и v. Из этого мы получаем 2 случая:

— Обе эти вершины были достижимы из r в исходном графе. Это означает сильную связность вершин u и r и сильную связность вершин v и r. Склеивая пути мы получаем связность вершин u и v.

— Хотя бы одна вершина не достижима из r в исходном графе, например v. Значит и r была не достижима из v в исходном графе, так как время выхода из r — больше (если бы она была достижима, то время выхода из v было бы больше, чем из r). Значит между этими вершинами нет пути (ни в исходном, ни в инвертированном графах), но последнего быть не может, потому что по условию v достижима из r на 3 шаге (в инвертированном графе).

Значит, из случая 1 и не существования случая 2 получаем, что вершины u и v сильно связаны в обоих графах. □

Оценим сложность алгоритма. Поиск в глубину в исходном графе выполняется за $O(V + E)$. Для того, чтобы инвертировать все ребра в графе, представленном в виде списка смежности потребуется $O(V + E)$ действий. Для матричного представления графа не нужно выполнять никакие действия для его инвертирования (индексы столбцов будут использоваться в качестве индексов строк и наоборот. Количество рёбер в инвертированном равно количеству рёбер в изначальном графе, поэтому поиск в глубину будет работать за $O(V + E)$. В сумме получаем, что сложность алгоритма $O(V + E)$ [11].

Приведём псевдокод: [12]

Вход:

Граф $G(V, E)$, внутри которого следует совершить обход

Выход:

Лес деревьев, каждое дерево представляет собой компоненту сильной связности изначального орграфа

Вспомогательные функции:

$order[]$ – массив с вершинами в порядке временных меток (от меньшего времени к большему) в DFS1.

$component[]$ – массив, в который сохраняются компоненты сильной связности орграфа в DFS2.

$visited[n, false]$ – массив, содержащий уже просмотренные элементы (как в DFS).

function DFS1(v)
    $visited[v] = true$;
    for $\forall(v, u) \in V$ **do**
        if $!visited[u]$ **then**
            DFS1(u)
        end if
    end for
    Добавить в $order$ текущую вершину v
end function
function DFS2(v)
    $visited[v] = true$;
    Добавить в component текущую вершину v
    for $\forall(v, u) \in V$ **do**
        if !visited[u] **then**
            DFS2(u)
        end if
    end for
end function
Алгоритм:
for $\forall i \in \{1 \dots n\}$ **do**
    if $!visited[v]$ **then**
        DFS1(v)
    end if
end for
Обнулить массив $visited$
Инвертировать массив:
for $\forall(i, j) \in V$ **do**

if $(i, j) \in V$ **then**
удалить из графа ребро (i, j), добавить (j, i).
end if
end for
for $\forall i \in \{1 \dots n\}$ **do**
Считать следующий с конца элемент в массиве – v = order[n-i]
if $!visited[v]$ **then**
DFS2(v)
Вывести полученную компоненту (*component*),
обнулить массив *component*.
end if
end for

2.6 Поиск Эйлерова пути

Определение 9. ***Степень, или валентность, вершины графа*** *– это число рёбер, инцидентных вершине. Обозначение степени вершины* $\deg v \equiv d(v)$.

Определение 10. ***Эйлеров путь*** *– это маршрут в графе, который проходит по всем рёбрам ровно один раз.* ***Эйлеров цикл*** *графа — это замкнутый эйлеров путь.*

Алгоритм поиска Эйлерова пути опирается на критерий эйлеровости графа:

Теорема 3. *Связный граф является эйлеровым тогда и только тогда, когда все вершины чётные.*

Доказательство. Необходимость. Дан связный эйлеров граф, покажем, что все вершины чётные. Граф является эйлеровым $\rightarrow$ в нём существует циклический эйлеров цикл. Допустим в графе существует вершина с нечетной степенью. Рассмотрим эйлеров обход графа. Заметим, что при попадании в вершину и при выходе из нее мы уменьшаем ее степень на два (помечаем уже пройденные ребра), если эта вершина не является стартовой (она же конечная для цикла). Для стартовой (конечной) вершины мы уменьшаем её степень на один в начале обхода эйлерова цикла, и на один при завершении. Следовательно вершин с нечетной степенью быть не может, то есть все вершины обязательно чётные.

Достаточность. Пусть все вершины чётные, покажем, что граф является эйлеровым. Выберем произвольную вершину v_0 , двигаясь по рёбрам, окрашиваем их. Продолжим движение по неокрашенным рёбрам и в итоге вернёмся в v_0. Мы получили циклический путь, состоящий из окрашенных рёбер. Здесь возможны два случая:

1. Все рёбра окрашены $\rightarrow$ пройден эйлеров цикл $\rightarrow$ граф является эйлеровым.
2. Окрашены не все рёбра. Поскольку граф связный, существует ребро, один конец которого принадлежит окрашенному ребру. Продолжим движение по этому ребру пока не вернёмся в начало (конец). Если новый построенный путь охватил все рёбра, то заканчиваем процесс, иначе продолжаем до тех пор, пока не получим эйлеров путь.

□

Опишем псевдокод алгоритма, определяющий Эйлеровость графа:

Вход:
Граф $G(V, E)$, внутри которого следует найти минимальное покрывающее дерево
Выход:
True, если граф является Эйлеровым, False – если нет
Алгоритм:
counter– счётчик числа вершин с нечётной степенью
$visited[n, false]$ – массив посещённых (пройденных) рёбер
for $\forall v \in V$ **do**
  if $deg(v) \mod 2 == 1$ **then**
    Увеличить counter на единицу
  end if
  if $counter > 2$ **then return** False
  end if
end for
for $\forall v \in V$ **do**
  if $deg(v) > 0$ **then**
    DFS(v, visited) ▷ В DFS будем помечать рёбра,
  end if ▷ по которым производим обход
end for

for $\forall v \in V$ **do**
  if $!visited[v]$ **then return** False ▷ Если компонент связности $\geqslant 1$,
  end if ▷ то граф не Эйлеров
end for
return True

Вход:

Граф $G(V, E)$, внутри которого следует найти минимальное покрывающее дерево

Вершина v, из которой мы будем искать Эйлеров путь

Выход:

Стек, в который сложены вершины, по которым надо пройти для получения Эйлерова пути

Алгоритм:

a – матрица смежности графа

S – стек вершин

function SearchEuler(v, a, S)
  for $\forall i \in \{1 \dots n\}$ **do**
    if $a[v, i] \neq 0$ **then**
      $a[v, i] = a[i, v] = 0$
      SearchEuler(i, a, S)
    end if
  end for
  Добавить в стек вершину v
end function
for $\forall u \in V$ **do** ▷ Если граф является полуэйлеровым,
  if $deg(u) \mod 2 == 1$ **then** ▷ то алгоритм следует запускать из
    $v = u$ ▷ вершины нечетной степени
  end if
end for
SearchEuler(v, a, S)
return S

Алгоритм поиска эйлерова графа корректен:

Утверждение 5. *Данный алгоритм проходит по каждому ребру, причем ровно один раз.*

Доказательство. Допустим, что в момент окончания работы алгоритма имеются еще не пройденные ребра. Поскольку граф связен, должно существовать хотя бы одно не пройденное ребро, инцидентное посещенной вершине. Но тогда эта вершина не могла быть удалена из стека S, и он не мог стать пустым. Значит алгоритм пройдёт по всем рёбрам хотя бы один раз. Но так как после прохода по ребру оно удаляется, то пройти по нему дважды алгоритм не может [13]. □

Алгоритмическая сложность полученного алгоритма – $O(E)$, то есть линейная относительно количества рёбер в данном графе.

2.7 Алгоритм построения минимального покрывающего дерева

Определение 11. ***Дерево*** *– это связный граф без циклов.*

Покрывающее (или остовное) *дерево – это дерево, подграф данного графа, с тем же числом вершин, что и у исходного графа.*

Минимальное покрывающее дерево *– это остовное дерево этого графа, имеющее минимальный возможный вес, где под весом дерева понимается сумма весов входящих в него рёбер.*

Алгоритм Борувки позволяет получить минимальное покрывающее дерево (при этом веса рёбер должны быть неотрицательны).

1. Составить список ребер сети в порядке возрастания их весов
2. Первое из списка ребро перекрашивается, оба его конца образуют компоненту
3. Рассматриваем следующее ребро
   - *а*) Если оба конца ребра принадлежат одной и той же компоненте, ребро не окрашивается и не добавляется в компоненту
   - *б*) Если один конец лежит в некоторой компоненте, а второй не принадлежит ни одной из компонент, это ребро окрашивается, 2-й конец ребра помещается в компоненту, содержащую 1-й конец ребра
   - *в*) Если один конец ребра принадлежит одной компоненте, а второй конец ребра принадлежит другой другой компоненте, то ребро окрашивается, компоненты, содержащие концы этого ребра, сливаются в одну

г) Если оба конца не принадлежат никаким компонентам, ребро окрашивается и его концы образуют новую компоненту

4. Если рассмотрены все рёбра, работа алгоритма завершена, дерево, составленное из окрашенных ребер и есть минимальное остовное дерево. Иначе, возвращаемся к шагу (3)

Алгоритм Борувки действительно находит остовное дерево минимального веса, поскольку он является частным случаем алгоритма Радо — Эдмондса для графического матроида, где независимые множества — ациклические множества рёбер [14].

Псевдокод алгоритма:

Вход:

Граф $G(V, E)$, внутри которого следует найти минимальное покрывающее дерево.

Выход:

Массив edgesH – массив, хранящий все рёбра нового покрывающего дерева.

Алгоритм:

$edgesG[]$ – массив рёбер старого графа.

$edgeID[n]$ – массив, хранящий номер компоненты, в которую сейчас входит вершина i. Если все они будут равны одному значению, то мининимальное покрывающее дерево построено.

Отсортировать $edgesG$.

for $\forall(i, j) \in edgesG$ **do**
  if $i \neq j$ **then**
    Добавить (i, j) в $edgesH$
    $newID = edgeID[i], oldID = edgeID[j]$
    for $k \in \{1 \dots n\}$ **do**
      if $edgeID[k] = oldID$ **then** $edgeID[k] = newID$
      end if
    end for
  end if
end for

Этот код самым непосредственным образом реализует описанный выше алгоритм, и выполняется за $O(M \log N + N^2)$. Сортировка рёбер потребует $O(M \log N)$ операций. Принадлежность вершины тому или иному поддереву

хранится просто с помощью массива treeID - в нём для каждой вершины хранится номер дерева, которому она принадлежит. Для каждого ребра мы за $O(1)$ определяем, принадлежат ли его концы разным деревьям. Наконец, объединение двух деревьев осуществляется за $O(N)$ простым проходом по массиву treeID. Учитывая, что всего операций объединения будет $N - 1$, мы и получаем асимптотику $O(M \log N + N^2)$ [15].

3 Реализация

Для графов используется представление с помощью как матриц смежности, так и списков смежности. Первый вариант предпочтительнее для плотных графов, где $|V| \sim |E|^2$ или когда требуется быстро определить, существует ли ребро между двумя произвольными вершинами. Из недостатков матрицы смежности следует отметить большое количество памяти, занимаемое ею (в любом случае $|E|^2$). Недостатком же списка смежности является асимптотически менее выгодное обращение к списку с целью получения информации о наличии ребра между двумя данными вершинами, которое в худшем случае может достигать $O(n)$.

Списки и матрицы смежности легко адаптируются для представления взвешенных графов, т.е. графов, с каждым ребром которого связан определённый вес [16].

Также для того, чтобы согласовать восстановления путей алгоритмов Дейкстры и Флойда был адаптирован массив сохранения путей. Результат работы приведён в приложении А.

ЗАКЛЮЧЕНИЕ

В заключение работы следует отметить, что вся теоретическая база алгоритмов, описанная выше, является очень важным элементом в обучении IT-специалиста, поскольку является основой развития алгоритмического мышления и способности писать эффективный и красивый код. Изучение компьютерных наук, в том числе и теории графов, несомненно упростит создание продуктивных решений.

В ходе работы были рассмотрены наиболее популярные и важные алгоритмы на графах, а также рассмотрено их обоснование корректности. Практическая часть состоит в реализации этих алгоритмов на языке C#, которая также была выполнена. В дальнейшем данный класс должен стать основой для библиотеки обработки информации в формате графов, что поможет упростить работу с моделированием большого количества систем, описанных во введении.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Харари Фрэнк. Теория графов / Пер. с англ. В. П. Козырева. Под ред. Г. П. Гаврилова. Изд-е 2-е. М.: Едиториал УРСС, 2003. 296 с. – с.13-17

2 Салий В. Н., Богомолов А. М. Алгебраические основы теории дискретных систем. — М.: Физико-математическая литература, 1997.

3 Фрич Р., Перегуд Е. Е., Мациевский С. В. Избранные главы теории графов: Учебное пособие / Пер. с нем. Е. Е. Перегуда; Пол ред. С. В. Мациевского. Калининград: Изд-во РГУ им. И. Канта, 2008. 204 с.

4 Статья про поиск в глубину. [Электронный ресурс] URL: http://neerc.ifmo.ru/wiki/index.php?title=%D0%9E%D0%B1%D1%85%D0%BE%D0%B4_%D0%B2_%D0%B3%D0%BB%D1%83%D0%B1%D0%B8%D0%BD%D1%83%2C_%D1%86%D0%B2%D0%B5%D1%82%D0%B0_%D0%B2%D0%B5%D1%80%D1%88%D0%B8%D0%BD (Дата обращения: 3 июня 2023)

5 Статья про поиск в ширину. [Электронный ресурс] URL: http://e-maxx.ru/algo/bfs (Дата обращения: 3 июня 2023)

6 Статья про алгоритм Дейкстры. [Электронный ресурс] URL: http://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B (Дата обращения: 5 июня 2023)

7 Реализация алгоритма Дейкстры на языке C++. [Электронный ресурс] URL: http://e-maxx.ru/algo/dijkstra (Дата обращения: 5 июня 2023)

8 Статья про алгоритм Флойда-Уоршелла. [Электронный ресурс] URL: http://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BB%D0%BE%D0%B9%D0%B4%D0%B0 (Дата обращения: 6 июня 2023)

9 Описание транзитивного замыкания. [Электронный ресурс] URL: https://www.techiedelight.com/ru/transitive-closure-graph/ (Дата обращения: 6 июня 2023)

10 Роберт Седжвик. Алгоритмы на графах = Graph algorithms. — 3-е изд. — Россия, Санкт-Петербург: «ДиаСофтЮП», 2002. — С. 496.

11 Статья про алгоритм Косарайю. [Электронный ресурс] https://habr.com/ru/articles/537290/ (Дата обращения: 7 июня 2023)

12 Реализация алгоритма Косарайю на языке C++. [Электронный ресурс] URL: https://studfile.net/preview/16688709/page:2/ (Дата обращения: 7 июня 2023)

13 Статья об алгоритме построения Эйлерова цикла. [Электронный ресурс] URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%BF%D0%BE%D1%81%D1%82%D1%80%D0%BE%D0%B5%D0%BD%D0%B8%D1%8F_%D0%AD%D0%B9%D0%BB%D0%B5%D1%80%D0%BE%D0%B2%D0%B0_%D1%86%D0%B8%D0%BA%D0%BB%D0%B0
(Дата обращения: 7 июня 2023)

14 В. Е. Алексеев, В. А. Таланов, Графы и алгоритмы, Интуит.ру, 2006.

15 Статья о реализации алгоритма Краскала на языке C++. [Электронный ресурс]
URL: http://e-maxx.ru/algo/mst_kruskal (Дата обращения: 8 июня 2023)

16 Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ. — 3-е изд. — Издательский дом «Вильямс», 2013. — С. 626-700. — 1328 с.

ПРИЛОЖЕНИЕ А
Код класса графов

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.ComponentModel.Design;
using System.Globalization;
using System.IO;
using System.Security;


namespace graph_1_1
{
    class Graph
    {
        private GraphData graph;
        class GraphData
        {
            public bool[,] adjacencyMatrix;
            public int[,] weightMatrix;
            public bool[] isChecked;
            public int[,] pathVertices;
            public GraphData(bool[,] adjacencyMatrix, int[,]
             ↪ weightMatrix)
            {
                adjacencyMatrix = new
                 ↪ bool[adjacencyMatrix.GetLength(0),
                 ↪ adjacencyMatrix.GetLength(0)];
                isChecked = new bool[VerticesCount];
                pathVertices = new int[VerticesCount,
                 ↪ VerticesCount];

                this.weightMatrix = new int[VerticesCount,
                 ↪ VerticesCount];
```

```
        for (int i = 0; i < VerticesCount; i++)
        {
            for (int j = 0; j < VerticesCount; j++)
            {
                this.adjacencyMatrix[i, j] =
                ↪ adjacencyMatrix[i, j];
                pathVertices[i, j] = -1;
                if (!adjacencyMatrix[i, j])
                {
                    this.weightMatrix[i, j] = i == j ? 0 :
                    ↪ int.MaxValue;
                }
                else
                {
                    this.weightMatrix[i, j] =
                    ↪ weightMatrix[i, j];
                }
            }
        }
    }
    public int VerticesCount
    {
        get
        {
            return adjacencyMatrix.GetLength(0);
        }
    }
    public void ResetChecking()
    {
        for (int i = 0; i < VerticesCount; i++)
        {
            isChecked[i] = false;
        }
    }
```

```
//DFS
public void DepthSearch(in int currentIndex)
{
    isChecked[currentIndex] = true;
    Console.WriteLine(currentIndex);
    for (int i = 0; i < VerticesCount; i++)
    {
        if (isChecked[i] == false &&
        ↪ adjacencyMatrix[i, currentIndex])
        {
            DepthSearch(i);
        }
    }
}
//BFS
public void BreadthSearch(in int currentIndex,
↪ Queue<int> queue)
{
    queue.Enqueue(currentIndex);
    while (queue.Count != 0)
    {
        isChecked[currentIndex] = true;
        for (int i = 0; i < VerticesCount; i++)
        {
            if (isChecked[i] == false &&
            ↪ adjacencyMatrix[i, currentIndex])
            {
                queue.Enqueue(currentIndex);
            }
        }
        Console.WriteLine(queue.Dequeue());
    }
}
//Dijkstra
```

```
public long[] DijkstraSearch(int startSource)
{
    long[] minimalLength = new long[VerticesCount];
     ↪ //d
    for (int i = 0; i < VerticesCount; i++)
    {
        minimalLength[i] = int.MaxValue;
    }
    minimalLength[startSource] = 0;
    long newDistance;
    for (int i = 0; i < VerticesCount; i++)
    {
        int currentClosest = -1;
        for (int j = 0; j < VerticesCount; j++)
        {
            if (isChecked[j] == false &&
             ↪ (currentClosest == -1 ||
             ↪ minimalLength[j] <
             ↪ minimalLength[currentClosest]))
            {
                currentClosest = j;
            }
        }
        if (currentClosest == -1)
        {
            break;
        }
        isChecked[currentClosest] = true;
        for (int j = 0; j < VerticesCount; j++)
        {
            if (adjacencyMatrix[currentClosest, j])
            {
```

```
                    newDistance =
                     ↪ minimalLength[currentClosest] +
                     ↪ weightMatrix[currentClosest, j];
                    if (newDistance < minimalLength[j])
                    {
                        minimalLength[j] = newDistance;
                        pathVertices[startSource, j] =
                         ↪ currentClosest;
                    }
                }
            }
        }
        return minimalLength;
    }
    //Floyd
    public long[,] FloydSearch()
    {
        long[,] minimalDistance =
         ↪ (long[,])weightMatrix.Clone();
        long newDistance;
        for (int k = 0; k < VerticesCount; k++)
        {
            for (int i = 0; i < VerticesCount; i++)
            {
                for (int j = 0; j < VerticesCount; j++)
                {
                    if (minimalDistance[i, k] !=
                     ↪ int.MaxValue && minimalDistance[k,
                     ↪ j] != int.MaxValue)
                    {
                        newDistance = minimalDistance[i,
                         ↪ k] + minimalDistance[k, j];
                        if (newDistance <
                         ↪ minimalDistance[i, j])
```

```
                    {
                        minimalDistance[i, j] =
                        ↪ newDistance;
                        pathVertices[i, j] =
                        ↪ pathVertices[k, j];
                    }
                }
            }
        }
    }
    return minimalDistance;
}
//WayBack
public void WayBack(int startIndex, int targetIndex,
↪ ref Stack<int> path)
{
    if (targetIndex == startIndex)
    {
        path.Push(targetIndex);
        return;
    }
    path.Push(targetIndex);
    WayBack(startIndex, pathVertices[startIndex,
    ↪ targetIndex], ref path);
}
// Function that returns reverse (or transpose) of
↪ this graph
private GraphData Transpose(GraphData graphData)
{
    GraphData temp = new
    ↪ GraphData(graphData.adjacencyMatrix,
    ↪ graphData.weightMatrix);
    for (int i = 0; i < temp.VerticesCount; i++)
    {
```

```
            for (int j = 0; j < temp.VerticesCount; j++)
            {
                if (temp.adjacencyMatrix[i, j] == true)
                {
                    temp.adjacencyMatrix[j, i] =
                    ↪ temp.adjacencyMatrix[i, j];
                    temp.weightMatrix[j, i] =
                    ↪ temp.weightMatrix[i, j];
                    temp.adjacencyMatrix[i, j] = false;
                    temp.weightMatrix[i, j] = i == j ? 0 :
                    ↪ int.MaxValue;
                }
            }
        }
        return temp;
    }
    private void DFS1(int v, bool[] visited, Stack<int>
    ↪ stack)
    {
        visited[v] = true;
        for (int i = 0; i < VerticesCount; i++)
        {
            if (!visited[i] && adjacencyMatrix[v, i])
                DFS1(i, visited, stack);
        }
        stack.Push(v);
    }
    private void DFS2(int v, bool[] visited, List<int>
    ↪ components)
    {
        visited[v] = true;
        components.Add(v);
        for (int i = 0; i < VerticesCount; i++)
        {
```

```
            if (!visited[i] && adjacencyMatrix[v, i])
            {
                DFS2(i, visited, components);
            }
        }
    }
    //KosarajuSearch
    public List<List<int>> KosarajuSearch()
    {
        Stack<int> stack = new Stack<int>();
        bool[] visited = new bool[VerticesCount];
        for (int i = 0; i < VerticesCount; i++)
            visited[i] = false;
        for (int i = 0; i < VerticesCount; i++)
        {
            if (!visited[i])
                DFS1(i, visited, stack);
        }
        GraphData transposedGraph = Transpose(this);
        ResetChecking();
        List<List<int>> components = new
        ↪  List<List<int>>();
        while (stack.Count != 0)
        {
            int v = stack.Pop();
            int j = 0;
            if (!visited[v])
            {
                List<int> component = new List<int>();
                transposedGraph.DFS2(v, visited,
                ↪  components[j]);
                Console.WriteLine(string.Join(", ",
                ↪  component));
            }
```

```
    }
    return components;
}
//BoruvkiSearch
public GraphData BoruvkiSearch()
{
    List<(int, int)> edgesH = new List<(int, int)>();
    List<(int, (int, int))> edgesG = new List<(int,
     ↪  (int, int))>();
    int[] edgeID = new int[VerticesCount];
    for (int i = 0; i < VerticesCount; i++)
    {
        for (int j = i; j < VerticesCount; j++)
        {
            if (adjacencyMatrix[i, j])
            {
                edgesG.Add((weightMatrix[i, j], (i,
                 ↪  j)));
            }
        }
    }
    edgesG.Sort();
    GraphData result = new GraphData(adjacencyMatrix,
     ↪  weightMatrix);
    for (int i = 0; i < VerticesCount; i++)
    {
        for (int j = 0; j < VerticesCount; j++)
        {
            result.adjacencyMatrix[i, j] = false;
            weightMatrix[i, j] = i == j ? 0 :
             ↪  int.MaxValue;
        }
    }
    foreach (var edge in edgesG)
```

```
        {
            if (edgeID[edge.Item2.Item1] !=
             ↪ edgeID[edge.Item2.Item2])
            {
                result.adjacencyMatrix[edge.Item2.Item1,
                 ↪ edge.Item2.Item1] = true;
                result.weightMatrix[edge.Item2.Item1,
                 ↪ edge.Item2.Item1] = edge.Item1;
                int newID = edgeID[edge.Item2.Item1];
                int oldID = edgeID[edge.Item2.Item2];
                for (int i = 0; i < VerticesCount; i++)
                {
                    if (edgeID[i] == oldID)
                        edgeID[i] = newID;
                }
            }
        }
        return result;
    }
    //EulerSearch
    public void SearchEuler(int currentVertice, ref
     ↪ bool[,] adjacencyMatrix, ref Stack<int>
     ↪ eulerVertices)
    {
        for (int i = 0; i < adjacencyMatrix.GetLength(0);
         ↪ i++)
        {
            if (adjacencyMatrix[currentVertice, i] ==
             ↪ true)
            {
                adjacencyMatrix[currentVertice, i] =
                 ↪ false;
                adjacencyMatrix[i, currentVertice] =
                 ↪ false;
```

```
                SearchEuler(i, ref adjacencyMatrix, ref
                ↪ eulerVertices);
            }
        }
        eulerVertices.Push(currentVertice);
    }
}
public Graph(in bool[,] adjacencyMatrix)
{
    int[,] weightMatrix = new int[adjacencyMatrix.Length,
    ↪ adjacencyMatrix.Length];
    for (int i = 0; i < weightMatrix.Length; i++)
    {
        for(int j = 0; j < weightMatrix.Length; j++)
        {
            if (adjacencyMatrix[i, j])
            {
                weightMatrix[i, j] = 1;
            }
            else
            {
                weightMatrix[i, j] = int.MaxValue;
            }
        }
    }
    graph = new GraphData(adjacencyMatrix, weightMatrix);
}
public Graph(bool[,] adjacencyMatrix, int[,] weightMatrix)
{
    graph = new GraphData(adjacencyMatrix, weightMatrix);
}
public int Size
{
    get { return graph.VerticesCount; }
```

```
}
public void DepthSearch(in int startIndex)
{
    graph.DepthSearch(startIndex);
    graph.ResetChecking();
}
public void BreadthSearch(in int startIndex)
{
    Queue<int> queue = new Queue<int>();
    graph.BreadthSearch(startIndex, queue);
    graph.ResetChecking();
}
public long[] DijkstraSearch(in int startIndex)
{
    long[] result = graph.DijkstraSearch(startIndex);
    graph.ResetChecking();
    return result;
}
public long[,] FloydSearch()
{
    long[,] result = graph.FloydSearch();
    graph.ResetChecking();
    return result;
}
public List<List<int>> KosarajuSearch()
{
    return graph.KosarajuSearch();
}
public Graph BoruvkiSearch()
{
    GraphData temp = graph.BoruvkiSearch();
    return new Graph(temp.adjacencyMatrix,
     ↪  temp.weightMatrix);
}
```

```
        public Stack<int> EulerSearch()
        {
            bool[,] a = new bool[graph.VerticesCount,
             ↪ graph.VerticesCount];
            for (int i = 0; i < graph.VerticesCount; i++)
            {
                for (int j = 0; j < graph.VerticesCount; j++)
                {
                    a[i, j] = graph.adjacencyMatrix[i, j];
                }
            }
            Stack<int> path = new Stack<int>();
            graph.SearchEuler(0, ref a, ref path);
            return path;
        }
        public Stack<int> WayBack(int startIndex, int targetIndex)
        {
            Stack<int> path = new Stack<int>();
            if (startIndex == targetIndex)
            {
                path.Push(startIndex);
                return path;
            }
            if (graph.pathVertices[startIndex, targetIndex] == -1)
            {
                DijkstraSearch(startIndex);
            }
            graph.WayBack(startIndex, targetIndex, ref path);
            return path;
        }
    }
}
```