ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

Кафедра математического и программного обеспечения информационных систем

Программная реализация алгоритма решения задачи почтальона

Выпускная квалификационная работа

обучающегосяпо направлению подготовки
02.03.03 Математическое обеспечение и администрирование информационных систем
очной формы обучения,
группы 07001402
ТойчыеваШатлыкаМухаммедовича

Научный руководитель:
доцент, к.т.н., Муромцев В. В.

БЕЛГОРОД 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Задачи маршрутизации значимы для сфер логистики и управлениятранспортом. Проблемы маршрутизации в основном связаны с определением оптимального набора линий в мультиграфе. Цель почтальона считается особенным случаем проблемы маршрутизации, обладающим большое количество возможных дополнений. Почтальон обязан разнести почту согласно вверенному ему району, с целью чего он проходит по абсолютно всем без исключения улицам района и возвращается в начальную точку (в почтовое отделение). Необходимо отыскать короткий путь почтальона. Эта задача достаточно современна: оптимальные маршруты необходимо проводить для различных автомобилей, размечающих, посыпающих и поливающих дороги населенных пунктов. В представлении проблемы почтальона графом перекрестки соответствуют вершинам, а отрезки улиц между перекрестками — ребрам. В общем случае данной модели мало, так как отрезки улиц обладают различную длину, а в задаче надо уменьшить непосредственно длину маршрута. Точной моделью станет взвешенный граф, в котором ребрам приписаны положительные числа.

Целью, этой выпускной квалификационной работы является описание и реализация различных алгоритмов решения задачи почтальона, а кроме того дальнейшее сравнение эффективности и методы оптимизации данных алгоритмов в графах с различными характеристиками.

Данная работа состоит из введения, 3 глав, заключения, списка литературы и приложения. Содержит 10 рисунков, 14 использованных источников. В введении коротко сформулирована предметная область и цели исследования.

Первая глава содержит в себе описание главных понятий теории графов, описывается критерий существования эйлерова цикла. Эта глава включает теорему существования эйлерова цикла и доказательство

достаточности и необходимости. Описывается неформальная задача почтальона и рассматривается алгоритмы решения задачи почтальона в терминах теорииграфов. Также описывается алгоритмы Дейкстры, алгоритм Флёри и алгоритм Маркеса-Куна.

Во второй главеописаны формальная и неформальная постановки задачи почтальона. Показаны блок-схемы алгоритмов решения задачи почтальона.

В третьей главе описывается практическая реализация алгоритма. Рассмотрена работа алгоритма в примере тестовых путей. Приведено сравнение эффективности работы в таблицах и в диаграмме для графов, содержащих различное число вершин и ребер.
В заключении приведены основные выводы, в приложениях содержится полный код разработанных программ.

ТЕОРИТИЧЕСКИЙ РАЗДЕЛ

1.1Основные определения и понятия в теории графов

В математике теория графов - это исследование графов, которые являются математическими структурами, используемыми для моделирования парных отношений между объектами. Граф в этом контексте состоит из вершин, узлов или точек, соединенные ребрами, дугами или линиями. Граф может быть неориентированным, т. е. нет различия между двумя вершинами, связанные с ребрами, или эти ребра могут быть направлены от одной вершины к другой для более подробного описания. Графы являются одним из основных объектов исследования в дискретной математике.

В наиболее широком понимании этого термина граф - это упорядоченная пара G = (V, E) состоящая из множества V вершин и множества E ребер, которые представляют собой 2-элементных подмножеств V. Чтобы избежать двусмысленности, этот тип графа может быть описан как неориентированный и простой.

Другие значения графа вытекают из различных концепций множества ребер. В еще одном обобщенном понятии V является множеством вместе с отношением инцидентности, которое связывает с каждым ребром две вершины. В другом обобщенном понятии E является мультимножеством неупорядоченных пар (не обязательно различных) вершин. Многие авторы называют этот тип объекта мультиграфом или псевдографом.

Вершины, принадлежащие ребру, называются концами или конечными вершинами ребра. Вершина может существовать в графе и не принадлежать ребру.

V и E обычно считаются конечными множествами, и многие из известных результатов не верны для бесконечных графов, потому что многие из аргументов терпят неудачу в бесконечном случае. Степень графа равен |V|,

его количество вершин. Размер графа равен |E|, его количество ребер. Степень или валентность вершины - это число ребер, которые соединяются с ней, где ребро, которое соединяет вершину с собой (петля), подсчитывается дважды.

Для ребра {x, y} теоретики графов обычно используют несколько более короткую запись xy.

Область применения

Графы могут быть использованы для моделирования многих типов отношений и процессов в физических, биологических, социальных и информационных системах. Многие практические задачи могут быть представлены графами. Подчеркивая их применение к реальным системам, термин сеть иногда определяется как граф, в котором атрибуты (например, имена) связаны с узлами и/или ребрами.

В информатике графы используются для представления сетей связи, организации передачи данных, вычислительных устройств, потока вычислений и т. д. Например, структура ссылок веб-сайта может быть представлена ориентированным графом, в котором вершины представляют веб-страниц, а направленные ребра представляют ссылки с одной страницы на другую. Аналогичный подход может быть применен к проблемам в социальных сетях, путешествиях, биологии, разработке компьютерных чипов, картированию прогрессирования нейродегенеративных заболеваний и многих других областях. Поэтому разработка алгоритмов обработки графов представляет большой интерес для информатики. Преобразование графов часто формализовано и представлено системами перезаписи графов. В дополнение к системам преобразования графов, ориентированным на обработку графов в памяти на основе правил, графические базы данных ориентированы на безопасное для транзакций, постоянное хранение и запрос структурированных графов данных.

1.2Эйлеровы графы

Замкнутая цепь в графе G, содержащая все ребра G, называется Эйлеровым циклом в G. А граф, содержащий цикл Эйлера, называется графом Эйлера.

Мы знаем, что цепь всегда связана. Поскольку цикл Эйлера содержит все ребра графа, граф Эйлера связан, за исключением любых изолированных вершин, которые могут быть в графе. Поскольку изолированные вершины ничего не способствуют пониманию графа Эйлера, полагается, что графы Эйлера не имеют изолированных вершин, таким образом, связаны.

Теорема (Эйлер). Связный граф G является графом Эйлера тогда и только тогда, когда все вершины G имеют четную степень.

Доказательство

Необходимость. Пусть G (V, E) - Эйлеров граф. Таким образом, G содержит Эйлеров цикл Z, который является закрытой цепью. Пусть эта цепь начинается и заканчивается в вершине u∈V. Поскольку каждое посещение Z до промежуточной вершины v из Z вносит два значения в степень v, а так как Z проходит через каждое ребро ровно один раз, тогда d (v) четно для каждой такой вершины. Каждое промежуточное посещение u дает два значения степени u, а также начальное и конечное ребра Z вносят каждый из них в степень u. Таким образом, степень d (u) и также четна.

Достаточность. Пусть G-связный граф, а степень каждой вершины G-четная.

Предположим, что G не Эйлерова и пусть G содержит наименьшее число ребер. Так как $\delta \geq 2$, G имеет цикл. Пусть Z-замкнутая цепь в G максимальной длины. Ясно, что G-E (Z) – граф с четной степенью. Пусть C1 является одной из составляющих G-E (Z). Поскольку C1 имеет меньшее количество ребер, чем G,тогда она Эйлерова и имеет вершину v, общую с Z. Пусть Z0-Эйлерова цепь в C1. Так какZ0 [Z замкнутая цепь в G,

начинающаяся и заканчивающаяся в v] длиннее Z, выбор Z противоречившая. Следовательно, G является Эйлеровым.

Задача о Кенигсбергских мостах.

Значительный вклад в развитие комбинаторных методов внёс замечательный швейцарский математик Л. Эйлер (1707-1783), большую часть жизни работавший в Санкт-Петербурге и Берлине.

Именно Эйлеру принадлежит постановка и решение первой комбинаторной задачи на графах. В 1736 г. он опубликовал свой знаменитый трактат, в котором решил задачу о кёнигсбергских мостах. В то время это была популярная головоломка, привлекавшая внимание учёных разных стран. В задаче предлагалось найти замкнутый маршрут, проходящий ровно один раз по каждому из семи мостов в Кёнигсберге, соединявших берега реки Прегель с двумя островами (см. рис. 1.1).

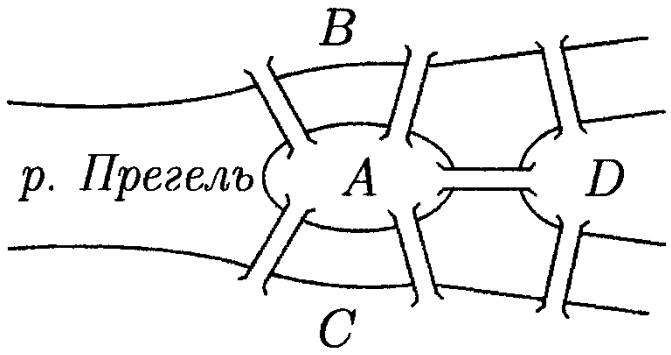

Рис. 1.1. Упрощенная схема мостов в городе Кенигсберг

Чтобы прояснить математическую сущность задачи, Эйлер изобразил участки суши точками, а мосты — линиями, соединяющими эти точки. Получился мультиграф, изображённый на рисунке1.2.

Эйлер решил задачу о кёнигсбергских мостах, хотя никогда не бывал в Кенигсберге (ныне российский город Калининград). Мало того, он указал общие условия разрешимости или неразрешимости задач подобного рода, т.е., как мы говорим сейчас, доказал критерий существования эйлерова цикла в связном графе.

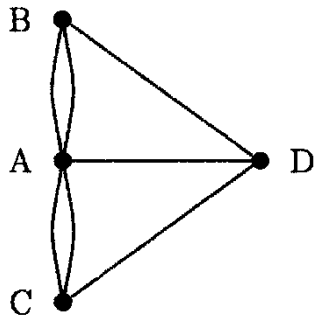

Рис.1.2. Представление мостов в виде графа

Теорема. Связный граф G является эйлеровым тогда и только тогда, когда его множество ребер может быть разложено на циклы.

Доказательство пусть G (V, E) - связный граф и пусть G разлагается на циклы. Если кэти циклы падают в определенной вершине v, затем d (v) = 2k. Поэтому степень каждой вершиныG четна и, следовательно, Gэйлеров.

И наоборот, пусть G-эйлеров граф. Покажем, что G можно разложить на циклы. Подтверждатьэто, мы используем индукцию по числу ребер.

Поскольку d (v) 2 для каждого v 2 V, G имеет цикл C. тогда G-E(C), возможно, отключен граф, каждая из компонент которого C1, C2, ..., Ck - график четной степени и, следовательно, Эйлерова. По индукционной гипотезе каждый Ci является несвязным объединением циклов. Вместе с C обеспечивают разбиение E (G) на циклы.

Доказательство.пусть G (V, E) - связный граф и пусть G разлагается на циклы. Если k из этих циклов падают в определенной вершине v, то d (v) = 2k. Поэтому степень каждой вершины G четна и, следовательно, GЭйлерова.

И наоборот, пусть G-Эйлерова. Покажем, что G можно разложить на циклы. Чтобы доказать это, мы используем индукцией по числу ребер.

Поскольку d(v) 2 для каждого v 2 V, G имеет цикл C. тогда G−E (C), возможно, является несвязным графом, каждая из компонент которого C1, C2, ..., Ck-граф четной степени и, следовательно, Эйлеров. По индукционной

гипотезе каждый Ci является несвязным объединением циклов. Эти вместе с C обеспечивают перегородку E(G) в циклы.

1.3 Задача почтальона

В 1962 году китайский математик МейгуГуань (часто известный как Мэй-Ко Кван) представил проблему, с которой может столкнуться почтальон.

Предположим, что почтальон начинает с почтового отделения и имеет почту для доставки в дома вдоль каждой улицы по его почтовому маршруту. Когда он завершает доставку почты, он возвращается на отделение. Проблема заключается в том, нужно найти минимальную длину пути туда и обратно.

Задача почтальона, описанная выше, может быть представлена связным графом G, вершины которого являются пересечениями улиц и ребра являются улицами на маршруте доставки почты.

Одной из возможных интерпретаций задачи почтальона является определение минимального количества пройденных дорог, проходя через каждую дорогу хотя бы один раз.Это эквивалентно нахождению минимальной длины замкнутой цепи в графе G, который использует каждое ребро G хотя бы один раз.Разумеется, такая замкнутая цепь существует в G, если для каждого ребра e в G добавляется ребро e, соединяющее ту же пару вершин, что и e, то получается мультиграф H, в котором каждое ребро G дублируется и каждая вершина в H имеет четную степень. Фактически, degH v = 2degG v для каждой вершины v графа G.Поскольку H эйлеров граф, в G существует замкнутая цепь, которая проходит через каждое ребро G дважды.

Естественно, если G - эйлеров граф размера m, то проблема китайского почтальона легко решить, поскольку любой эйлеров цепь в G является

эйлеровым циклом, а его длина равна m. Если G не является эйлеровым, то трюк для решения проблемы состоит в том, чтобы дублировать как можно меньше ребер G так, чтобы полученный мультиграф стал эйлеровым. В частности, если нужно дублировать минимальное количество ребер G, чтобы каждая вершина имела четную степень в полученноммультиграфе, равную на k, то длина эйлерового цикла в G будет равна m + k.

В простейшем случае предположим, что G содержит ровно две нечетные вершины, например u и v. Тогда нам нужно определить длину кратчайшего взвешенного пути от u до v, которую мы обозначим через dG (u, v).

1.4. Основные структуры данных для машинного представления графов

Выбор структуры данных для представления графов имеет значительное влияние на эффективность алгоритмов. Рассмотрим несколько различных способов представления графов и кратко разберём их основные достоинства и недостатки.

Будем рассматривать как неориентированные, так и ориентированные графы. Граф, как обычно, будем обозначать $G = (V, E)$,где V — множество вершин, а E — множество рёбер или дуг. Будем считать, что $|V| = n$, $|E| = m$.

При решении комбинаторных задач на графах с помощью алгоритмов наиболее часто приходится отвечать на следующие два основных вопроса:

1) смежны ли некоторые вершины u, v?

2) существует ли вершина, смежная с данной вершиной v?

Кроме того, нас всегда будет интересовать насколько просто, используя ту или иную структуру данных, удалять и добавлять рёбра или дуги.

Матрица инцидентности

Это матрица с n строками, соответствующими вершинам, и mстолбцами, соответствующими рёбрам или дугам. Для неориентированного графа столбец, соответствующий ребру uv, содержит единицы в строках, соответствующих вершинам u,v, и нули в остальных строках. Для орграфа столбец, соответствующий дуге uv, содержит —1 в строкеu, 1 в строке v и нули во всех остальных строках. Петлю, т. е. дугу вида vv удобно представлять значением 2 в строке v.

Матрица инцидентности — классический способ представления графа в теории. С алгоритмической точки зрения эта структура является самым худшим способом представления графа. Во-первых, она требует порядка nm(т. е. $\Theta(nm)$) ячеек памяти, большинство из которых занято нулями. Во-вторых, неудобен доступ к информации. Ответ на элементарный вопрос типа «смежны ли некоторые вершины u, v?» или «существует ли вершина, смежная с данной вершиной v?» требует в худшем случае просмотра всей строки, т. е. $O(m)$ шагов.

Матрица смежности

Это квадратная матрица $A = (a_{ij})$размера $n \times n$, где

$$a_{ij} = \begin{cases} 1, & \text{если } v_i v_j \in E, \\ 0, & \text{в противном случае.} \end{cases} \qquad (1.1)$$

В неориентированном графе $v_i v_j \in E \Leftrightarrow v_j v_i \in E$, так, что матрица смежности неориентированного графа симметрична, адля ориентированного графа — не обязательно.

Основное достоинство матрицы смежности — прямой доступ к информации, т. е. возможность за один шаг получить ответ на вопрос «смежны ли некоторые вершины u, v?», а также удалить или добавить ребро uv. Однако ответ на вопрос «существует ли вершина, смежная с данной

вершиной *v*?», как и в случае матрицы инцидентности, требует в худшем случае просмотра всей строки, т. е. *O(n)* операций.

Недостатком является также тот факт, что независимо от числа рёбер или дуг графа объём занятой памяти составляет $\Theta(n^2)$. Кроме того, уже начальное заполнение матрицы смежности путем «естественной» процедуры имеет трудоёмкость $\Theta(n^2)$, что сразу сводит на нет алгоритмы линейной трудоёмкости *O(n)* при работе с графами, содержащими *O(n)* рёбер.

Массив рёбер или дуг

Эта структура данных более предпочтительна по сравнению с двумя предыдущими в смысле экономии памяти, если $m<<n^2$ (т. е. m много меньше, чем n^2). При использовании такого массива для хранения всего графа требуется порядка *m* ячеек памяти.

Недостатком является то, что ответ на каждый из двух наших основных вопросов: «смежны ли некоторые вершины *u*, v?» или «существует ли вершина, смежная с данной вершиной v?» - требует в худшем случае просмотра всего массива, т. е. *O(n)*шагов.

Списки соседних вершин (списки инцидентности)

Это динамическая структура данных, основанная на аппарате ссылочных переменных. Для неориентированного графа она содержит для каждой вершины v∈V список вершин, смежных с *v*. Каждый элемент такого списка является записью, содержащей информационное поле с меткой вершины *u*, смежной с *v*, и поле с указателем на следующий элемент списка. Начало каждого списка хранится в массиве A ссылочных переменных, каждый элементA[*v*]которого является указателем на начало списка, содержащего вершины, смежные с вершиной *v*. Весь такой список вместе с указателем будем обозначать A[*v*].

Заметим, что для неориентированного графа каждое ребро *uv* представлено в списках дважды: элементом v в списке A[*u*] и элементом *u* в спискеA[*v*].

Преимущество такого способа организации данных состоит в том, что для отыскания вершины, смежной с данной вершиной v, не нужно просматривать строку, как в матрице смежности, а достаточно лишь перейти по ссылке A[*v*].

Число ячеек памяти, необходимое для представления графа посредством списков соседних вершин, имеет порядок $\Theta(n+m)$. Кроме того, это динамическая структура: при удалении ребра *uv* список A[*u*] автоматически «сжимается», чем достигается экономия памяти.

Недостатком таких однонаправленных списков является то, что для удаления ребра *uv* требуется *O(n)*операций: удалив элемент *v* из списка A[*v*], необходимо отыскать элемент *u* в списке A[v],затратив в худшем случае *n* переходов по ссылке.

Поэтому предпочтительнее использовать следующую модифицированную структуру данных.

Списки соседних вершин с перекрёстными ссылками

В этой структуре элемент *v* списка A [*u*] содержит ссылку на элемент *u* спискаA[*v*], и наоборот. При использовании этой структуры данных удаление ребра *uv* может быть выполнено за *O(1)*операций (т. е. за число операций, ограниченное константой независимо от *n*). Для этого, удалив элемент v из списка A[*u*], мы просто переходим по ссылке на элемент *u* списка A[v] и удаляем его (фактически удаляется элемент, следующий за текущим элементом *u*, но его содержимое предварительно переписывается в текущий элемент).

Списки соседних вершин для орграфов

В этой структуре A[v] является указателем на начало списка, содержащего вершины, в которые ведут дуги из *v*. Заметим, что для орграфа каждая дуга *uv* представлена лишь один раз —элементом *v* в списке A[*u*]. Поэтому проблем с удалением дуг не возникает, удаление каждой дуги требует O(1) операций.

Однако, решая комбинаторные задачи на ориентированных графах, часто бывает нужно знать, не только какие дуги выходят из данной вершины, но и какие дуги входят в неё. Для этого приходится дополнительно использовать списки B[*v*], содержащие вершины, из которых идут дуги в вершину *v*.

В ряде случаев вместо пары списков A, B для представления ориентированного графа предпочтительнее использовать двумерный список, в котором каждый элемент соответствует дуге *uv* и является как бы элементом сразу двух списков — «горизонтального» A[*u*] и «вертикального» B[*v*].

1.5. Влияние структур данных на трудоёмкость алгоритмов

Специалистам хорошо известен один из основополагающих принципов программирования о взаимосвязи и взаимном влиянии алгоритмов и структур данных, используемых этими алгоритмами (достаточно вспомнить название книги создателя языка Паскаль Н. Вирта «Алгоритмы + структуры данных = программы»). В соответствии с этим принципом программу, записанную на каком-либо формальном языке программирования, можно понимать как конкретную, основанную на определённом строении данных реализацию абстрактного алгоритма.

Хорошо известно, что одна и та же задача может быть решена различными алгоритмами, имеющими неодинаковую вычислительную сложность. Гораздо меньше внимания уделяется тому факту, что разные структуры данных, используемые одним и тем же алгоритмом, способны заметно изменить трудоёмкость получаемых в результате программ.

В качестве примера рассмотрим алгоритм отыскания эйлеровацикла в неориентированном графе.

Напомним, что эйлеровым циклом в неориентированном графе называется цикл, проходящий по каждому ребру ровно один раз. Граф, содержащий такой цикл, называется эйлеровым графом.

Рассмотрим следующий абстрактный алгоритм отыскания эйлерова цикла.

Алгоритм EULER

Вход: связный граф $G = (V, E)$ без вершин нечётной степени.

Выход: последовательность вершин Эйлерова цикла в стеке EC.

1.{ EC = Ø;

2. ST = Ø; // ST — вспомогательный стек

3. ν = произвольная вершина графа;

4. ν → ST; // вершина ν помещается в стек

5. while (ST != Ø) // главный цикл

6. { v = top(ST)] // ν — верхний элемент стека

7. if (∃ u∈V :vu∈E)

8. { u = одна из вершин, смежных с ν,

9. u→ST;

10. удалить ребро vu из графа;

11. ν = u;

12. }

13. else // в графе нет вершин, смежных с υ

14. { ν ← ST; // вершина ν переносится из стека ST

15. ν → EC; // в стек EC

16. }

17. }

18.}

Теорема. Число итераций алоритма**EULER** равно 2m.

Доказательство. Итерация алгоритма **EULER** — это одно исполнение главного цикла в строках с 5 по 17. На каждой итерации алгоритм:

1) либо помещает в стек ST вершину u, смежную с текущей вершиной v, и удаляет ребро vu (строки 9-10), что соответствует переходу из вершины v в вершину u;

2) либо (если в оставшемся графе нет вершин, смежных с текущей вершиной v) переносит вершину v из стека ST в стек EC (строки 14-15), что соответствует возврату из вершины v в предыдущую вершину стека ST по ранее пройденному ребру.

Таким образом, каждая итерация соответствует либо переходу по новому ребру, либо возврату по ранее пройденному ребру,причём и тот, и другой переход осуществляется ровно один раз. Следовательно, с каждым ребром связано ровно две итерации, и общее число итераций равно $2m$.

Теорема доказана.

Вычислительная сложность каждой итерации (а значит, и всего алгоритма EULER) существенно зависит от используемой структуры данных. Рассмотрим три реализации алгоритма с разными структурами данных и оценим трудоёмкость каждой реализации.

Первая реализация: структура данных — матрица смежности. На каждой итерации для проверки условия «существует ли вершинаu, смежная с текущей вершиной v?» (строка 7 алгоритма) необходимо просматривать строку матрицы смежности A, соответствующую вершине v (до первой единицы A[v][u] = 1 либо чтобы убедиться, что все A[v][u] = 0). Это требует O(n) операций. Все остальные действия на каждой итерации, включая и удаление ребра vu в строке 10, требуют O(1) операций, т. е. числа операций, ограниченного константой. Итак, каждая итерация требует O(n) операций, а число итераций, как уже установлено в теореме, равно $2m$. Значит, общая трудоёмкость первой реализации алгоритма EULER — O(mn). Заметим, что в худшем случае это $O(n^3)$, так как m — $O(n^2)$.

Вторая реализация: структура данных — списки соседних вершин. В этой реализации проверка условия «существует ли вершина u, смежная с

текущей вершиной v?» в строке 7 осуществляется за один шаг - нужно лишь проверить, пуст или нет список A[v] вершин, смежных с *v*. На это требуется O(1) операций. Однако при удалении ребра *vu* в строке 10, кроме удаления элемента *u* из списка A[*v*], мы будем вынуждены просмотреть список A[u] с начала до тех пор, пока не доберёмся до элемента *v,* чтобы удалить и этот элемент. В худшем случае для этого придётся просмотреть весь список A[u], что потребует *O(n)* операций. Таким образом, вычислительная сложность каждой итерации есть O(n), и общая трудоёмкость второй реализации алгоритма EULER будет такой же, как и первой — *O (mn)*.

Третья реализация: структура данных — списки соседних вершин с перекрёстными ссылками. При такой структуре данных удаление ребра *vu* в строке 10 алгоритма EULER можно произвести за O(1) операций. Действительно, как уже указывалось ранее, удалив элемент *u* из списка A[*v*], мы переходим по ссылке на элемент v списка A[u] и удаляем его. Таким образом, каждая итерация требует O(1) операций, а всего в алгоритме 2m итераций. Следовательно, трудоёмкость третьей реализации алгоритма EULER на порядок меньше, чем в первых двух реализациях, — всего O(m) (или $O(n^2)$,учитывая, что $m = O(n^2)$).

1.6. Алгоритм Дейкстры

Вход: неориентированный граф G = (V, E), представленный весовой матрицей c[u][v], u, v ∈ V; множество вершин орграфа G отдельно представлено в виде списка V, выделена вершина s ∈ V.

Выход: дерево кратчайших путей от вершины *s*, представленное«стекающим» к s списком T. в котором элементы соответствуют вершинам дерева и каждая вершина *v* имеет указатель ρ[v] на предпоследнюю вершину кратчайшего (s, *v*)-пути.

```
1. { T = Ø;                    //в начале дерево пусто
2. V = V\{s}; TU{s};
3. d[s] = 0;
4. for (v ∈ V)                 // нулевая итерация
5. { d[v] = c[s][v];           // начальные метки вершин
6. ρ [v] = s;                  // в начале все указатели указывают на s
7.}
8. while (V!=Ø)                // главный цикл
9. { u = произвольная вершина множества V такая, что
d[u] == min{d[v],v ∈V};        // метка становится постоянной
10. if (d[u] ==∞) return;      // не все вершины достижимы из s
11. V = V \ {u}; T = T U {u};  // наращивание дерева
12. for (v∈ V)
13. if (d[u] + c[u][v] <d[v])
14. {d[v] = d[u] + c[u][v];    // преобразование временных меток
15. ρ[v] = u;                  //и указателей
16. }
17. }
18.}
```

Комментарий. Как уже отмечалось, метка d[*v*] равна верхней оценке длины кратчайшего пути из s в v, а для вершин, включённых в дерево T кратчайших путей, d[v] равна длине кратчайшего (s,v)-пути; ρ[v] — указатель на предпоследнюю вершину кратчайшего (s, v)-пути.

Обоснование и трудоёмкость алгоритма Дейкстры

Лемма. В любой момент выполнения главного цикла имеют место следующие утверждения:

(1) ∀v∈ T d[v].= d(s,v) (длине кратчайшего (s,v)-пути);

(2) ∀v∉ T d[v] = длине кратчайшего (s,v)-пути из тех путей, в которых все вершины, кроме v, принадлежат T.

Доказательство. Индукция по числу k исполненных итераций.

Очевидно, утверждения (1),(2) имеют место при k = 0, когда T = {s}.

Предположим, что они верны после исполненияk-1итераций.

Рассмотрим подробно k-ю итерацию и докажем, что по её завершении утверждения (1),(2) верны.

В строке 9 мы находим такую вершину u∉ T, что d[u] является минимальным значением из всех d[v] для v ∉ T. Покажем, что d[u] = d(s,u) — длине кратчайшего (s, u)-пути.

Предположим противное, т. е. d(s, u) < d[u]. По предположению индукции для *u* выполнено утверждение (2). Значит, в абсолютно кратчайшем (s,u)-пути есть отличные от *u* вершины, не принадлежащие T. Пусть w — первая вершина кратчайшего (s, u)- пути, не принадлежащая T. Очевидно, что (s, w)-фрагмент кратчайшего (s, u)-пути является кратчайшим (s, w)-путём, причём в нём все вершины, кроме w, принадлежат T. Значит, по предположению индукции в силу условия (2) d[w] = d(s,w). Но

d[w] = d(s,w) ≤ d(s,u) <d[u],

что противоречит выбору вершины u в строке 9 алгоритма.

Итак, d[u] = d(s, u), и мы можем включить вершину u в дерево T, не нарушая условия (1).что и происходит в строке 11 на k-й итерации главного цикла.

Выполнение условия (2) к концу k-й итерации достигается с помощью цикла 12-16, в котором проверяются пути из s во все вершины v ∉T, предпоследняя вершина которых есть *u*.

Лемма доказана.

По окончании главного цикла дерево T содержит все вершины орграфа G, достижимые из s, и в силу условия (1) леммы d[v] = d(s, v) для любой вершины v ∈T, откуда вытекает следующая теорема

Теорема. Алгоритм Дейкстры правильно строит дерево кратчайших путей от вершины s.

Теорема. Трудоёмкость алгоритма Дейкстры — $O(n^2)$.

Доказательство. Главный цикл 8-17 выполняется не болеen- 1 раз, причём каждое его исполнение требует O(n) операций (O(n)операций для отыскания вершины u в строке 9 и O(n)операций для выполнения цикла (12-16). Таким образом, общая трудоёмкость алгоритма Дейкстры — $O(n^2)$.

1.7Алгоритм построения эйлерова цикла

АлгоритмФлёри построения эйлерова цикла начинает работу с некоторой вершины x и каждый раз вычеркивает пройденное ребро. Не проходить по ребру, если удаление этого ребра приводит к разбиению графа на две связные компоненты (не считая изолированных вершин).

Опишем алгоритм построенияэйлерова цикла в связном неорграфе с четными степенями вершин, а также в орграфе с совпадающими полустепенями захода и исхода.

Замечание. Граф $G=(X,V)$ задается множеством X своих вершин и набором реберных окрестностей всех его вершин $S(x)=\{v$ |ребро v инцидентно вершине $x\}$.

Шаг 1. Выбрать произвольную вершину $x_0 \hat{I} X$ и положить $x=x_0$, $z=x_0$, $C=x_0$, $D=x_0$, гдеC - чередующаяся последовательность вершин и ребер, представляющая строящийся эйлеров цикл; D - чередующаяся последовательность, представляющая начальный отрезок цикла, который будет присоединен к текущему циклу C; x - конечная вершина в последовательности D; z - вершина на цикле C, которая служит началом D.

Шаг 2. В множестве $S(x)\backslash[V(C)\dot{E}V(D)]$ выбрать произвольное ребро 1. Если $S(x)=Æ$, то прейти к шагу 5. Здесь $V(C)$ и $V(D)$ обозначают множество ребер изC и D.

Шаг 3. К D дописать ребро 1 и его конец y, т.е. положить $D=\{D, 1, y\}$.

Шаг 4. Положить $x=y$ и перейти к шагу 2.

Шаг 5. Присоединить кC в вершине z цикл D, т.е. положить $C=\{C[x_0,z),D,C(z,x_0]\}$, где $C[x_0,z)$ - начальный отрезок последовательности C до вершиныz, не включая z; $C(z,x_0]$ – отрезок последовательности C от z до x_0, не включая z.

Шаг 6. Просматривая последовательностьC слева направо, ищем первую такую вершину v, что $S(v)\backslash V(C)=Æ$. Если такой вершины нет, то перейти к шагу 8, иначе перейти к шагу 7.

Шаг 7. Положить $x=v$, $z=v$, $D=v$ и перейти к шагу 2.

Шаг 8. ВыдатьпоследовательностьC, которая является эйлеровым циклом.

1.8Венгерский алгоритм

Вход: взвешенный граф $K_{n,n}[W]$ с долями X и Y .

Выход: множество ребер оптимальногопаросочетания P в данном графе.

1. Задать в $K_{n,n}[W]$ произвольную допустимую разметку f и найти,подграф равенств $G_{W,f}$.

2. Венгерским алгоритмом найти максимальное паросочетание P в графе$G_{W,f}$ и множество F свободных относительно P вершин доли X .

3. Если $F = \emptyset$, закончить работу.

4. Найти все чередующиеся цепи в графе $G_{W,f}$, начинающиеся в F,положить S и T равными множеству всех вершин доли X (соответственно,доли Y), встретившихся в этих цепях.

5. Если в T нет свободных вершин, положить

$$\Delta = \min_{x_i \in S, y_j \in Y \setminus T} \{ f(x_i) + f(y_j) - w_{ij} \}, \quad (1.2)$$

f (x) = f (x)−Δ для всех x ∈ S , f (y) = f (y) + Δ для всех y ∈ T , найтиновый граф $G_{W,f}$ и вернуться на шаг 4.

6. Увеличить P, перекрасив найденную увеличивающую цепь, и вернутьсяна шаг 3.

В соответствии с леммой о подграфе равенств, алгоритм Куна–Манкресаперестраивает исходную разметку так, чтобы найти совершенноепаросочетание в подграфе равенств. Величина Δ всегда строго положительна (по построению S и T,вершины из S смежны в $G_{W,f}$только вершинам из T; значит, в (1.2),для любого ребра $(x_i , y_j)f(x_i) + f(y_j) - w_{ij} >$ 0);измененная на шаге 5 функция f остается допустимой разметкой(сумма меток вершин, инцидентных ребру, уменьшилась только дляребер, соединяющих вершины из S и Y\T, причем величина Δопределена так, чтобы эта сумма осталась не меньше веса ребра);ребра, вошедшие в максимальноепаросочетание в текущем граферавенств, остаются в графе равенств после изменения разметки(метки увеличились только для вершин из T, но по построению онисвязаны темными ребрами только с вершинами из S, у которых меткисоответственно уменьшились);

после изменения графа $G_{W,f}$ на шаге 5 множество S не меняется, аво множество T добавляется хотя бы одна вершина (та, на которойдостигается минимум в (2); это обеспечивает конечность числаитераций шагов 4–5 между увеличениями паросочетания, а значит,тот факт, что алгоритм остановится).

2. ПРОЕКТНЫЙ РАЗДЕЛ

2.1 Неформальная постановка задачи

В неформальной постановке задачи главным действующим лицом является почтальон, которому необходимо забрать корреспонденцию в почтовом отделении, разнести почту по улицам города, и затем обратно вернуться в почтовое отделение.Задача заключается в том, чтобы найтикратчайшиймаршрут пути для почтальона. Сформулированная выше задача имеет много потенциальных приложений. Например, оптимальный обход улиц снегоуборочной техникой, а также решение других проблемы, связанных с маршрутизацией.

Стоит заметить, что только в случае ориентированного и неориентированного графа задача почтальона имеет полиномиальное решение. В случае смешанного графа задача является NP-полной.

2.2 Задача, сформулированная в терминах теории графов

Сформулируем данную задачу в терминах теории графов. Из неформальной постановки задачи следует, что существует сильно связный неориентированный граф. Предполагается, что в нем не существует параллельных ребер иребер отрицательного веса.

По условию, ребрам графа сопоставлены положительные веса. Требуется найти цикл минимального веса, проходящий черезкаждое ребро графа, по крайней мере, один раз. Очевидно, что если содержитэйлеров цикл, то любой такой цикл будет оптимальным, так как каждое ребропроходится по один раз. В этомслучае достаточно найти эйлеров цикл, используя алгоритм эйлерого цикла,который будет описан далее. В случае неэйлерова графа в первую очередьнужно решить следующую подзадачу. В неэйлеровом

графе необходимо найти минимальное дополнение графа до эйлерова, где минимальное дополнениеграфа до эйлерова — это рёбра и числаkповторений каждого ребра, такие, чтопри дублировании каждого ребра из этого подмножестваkраз будет полученграф, в котором все вершины будут сбалансированы, т. е. эйлеров граф. Послетого, как будет найдено минимально дополнение графа до эйлерова, следуетприменить алгоритм поиска эйлерова цикла. Таким образом, будет полученорешение задачи почтальона.

Входные данные:n-количество вершин графа

Distance[50][50]-символьная матрица смежности

Выходные данные:Путь обхода графа. Суммарный вес пройденных ребер.

Метод решения:Программа состоит из 2 основных подпрограмм

1)Программа реализующая алгоритм Дейкстры (описанный в разделе 1)

2)Алгоритм Флёри (раздел 1) содержащий в себе часть венгерского алгоритма.

2.3. Проектирование алгоритмов решения задачи почтальона

Нарис. 2.1приведенаблок-схемафункцииmain(). В функции main() происходит ввод данных с клавиатуры.

В main() подсчитываются степени каждой вершины и вершины с нечетной степенью передаются в функцию Dijkstra(). И здесь вызываются функции Dijkstra() и Euler().

На рисунке 2.2 показана блок-схема работы алгоритма Дейкстры. Функция находит кратчайшие пути между вершинами нечетной степени.

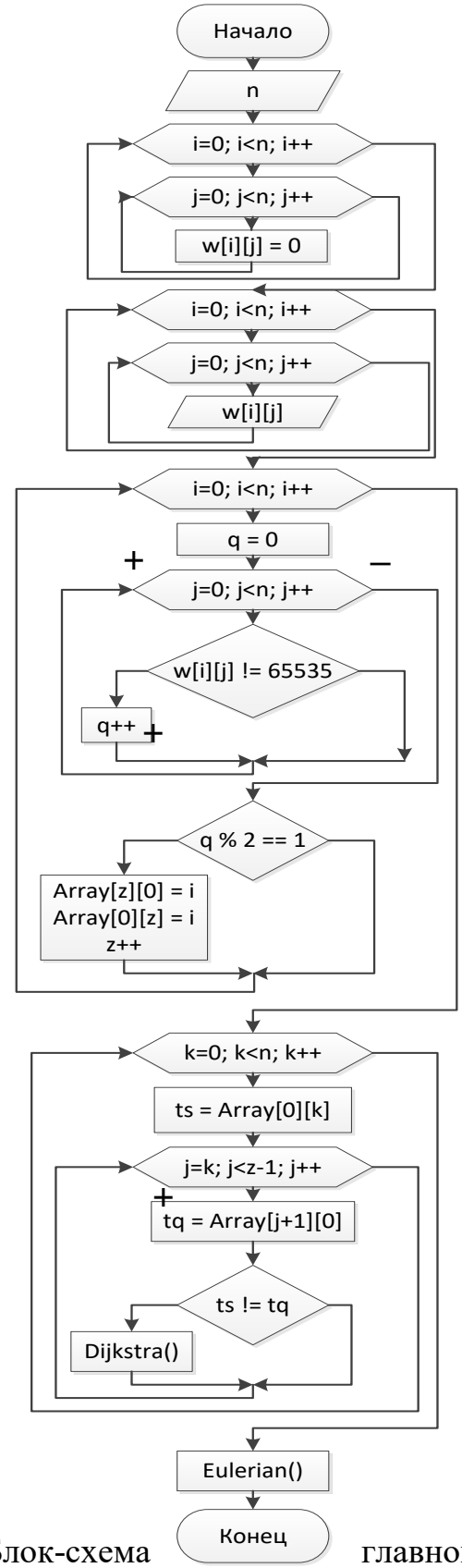

Рис. 2.1. Блок-схема главной функции main()

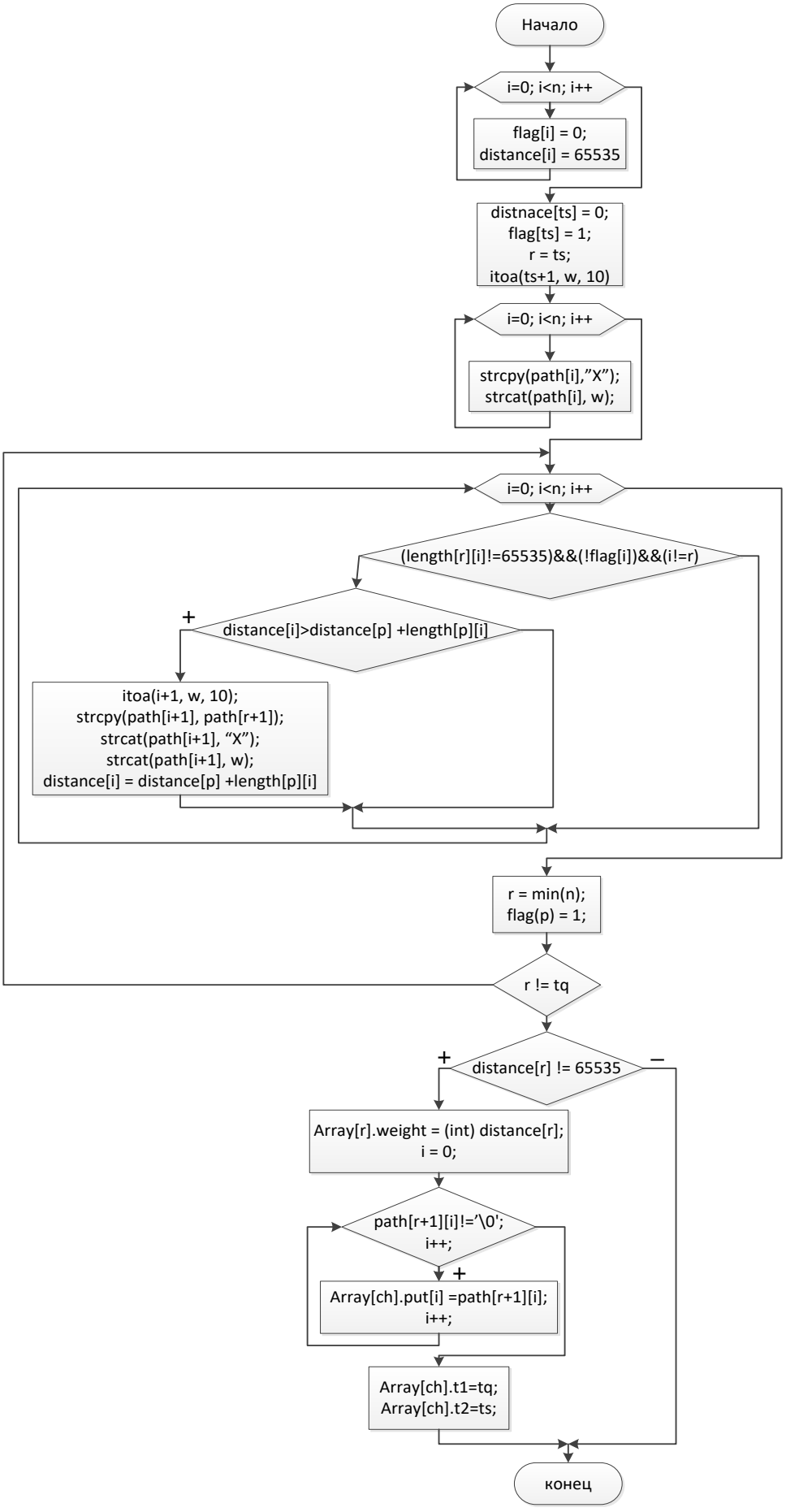

Рис.2.2. Блок-схема функции Dijkstra()

ФункцияSearch() реализуеталгоритм Флери, который находит эйлеров цикл в графе. Эта функция вызывается внутри функции Euler().(см. рис. 2.3)

На рисунках 2.4 и 2.5 приведена работа функции Euler(), которая находит эйлеров цикл в полученномультиграфе. Если эйлерова цикла не существует в графе, функция вызывает функцию not_connected().

В функции Euler() маршрут найденного эйлерова цикла выводится на экран. Этот маршрут и есть наш искомый оптимальный путь почтальона. Длина кратчайшей пути почтальона соответствует суммарному весу всех пройденных ребер.

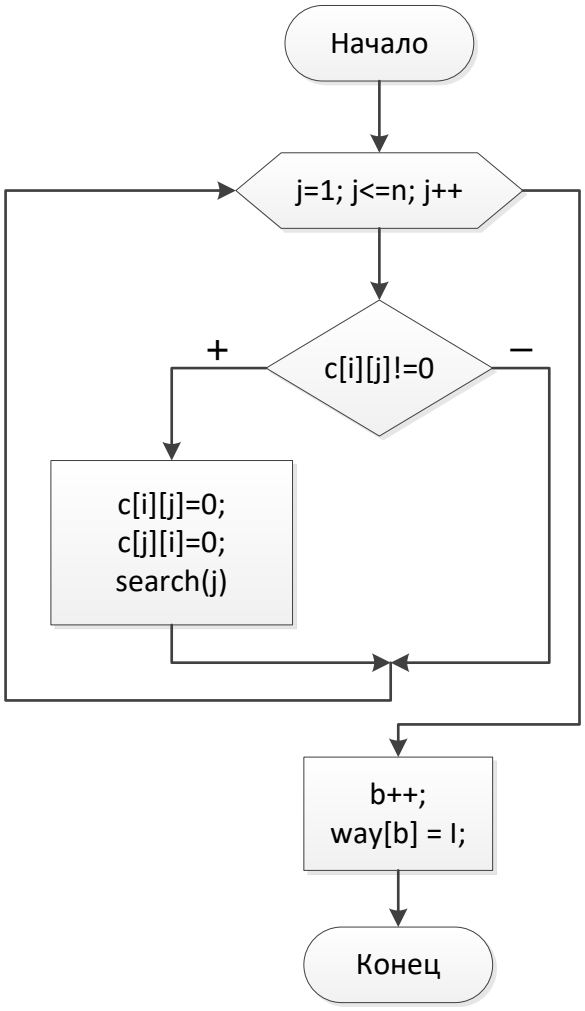

Рис. 2.3. Блок-схемафункцииSearch().

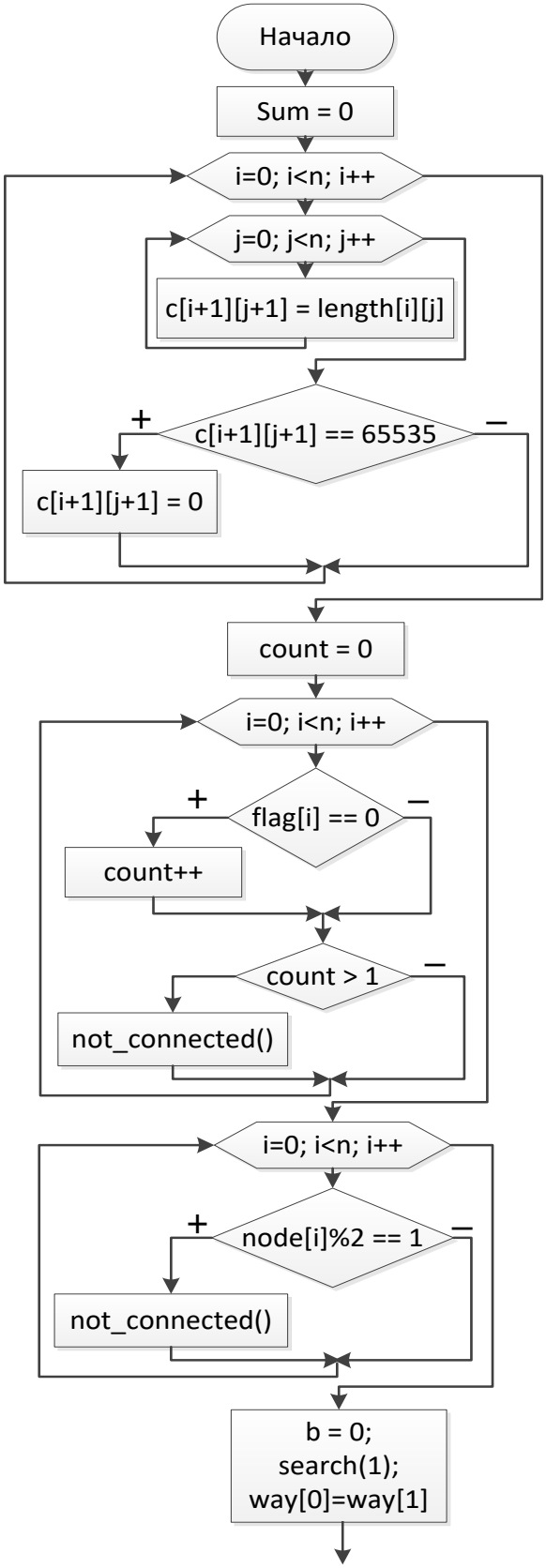

Рис. 2.4. Блок-схема выполнения функции Euler()

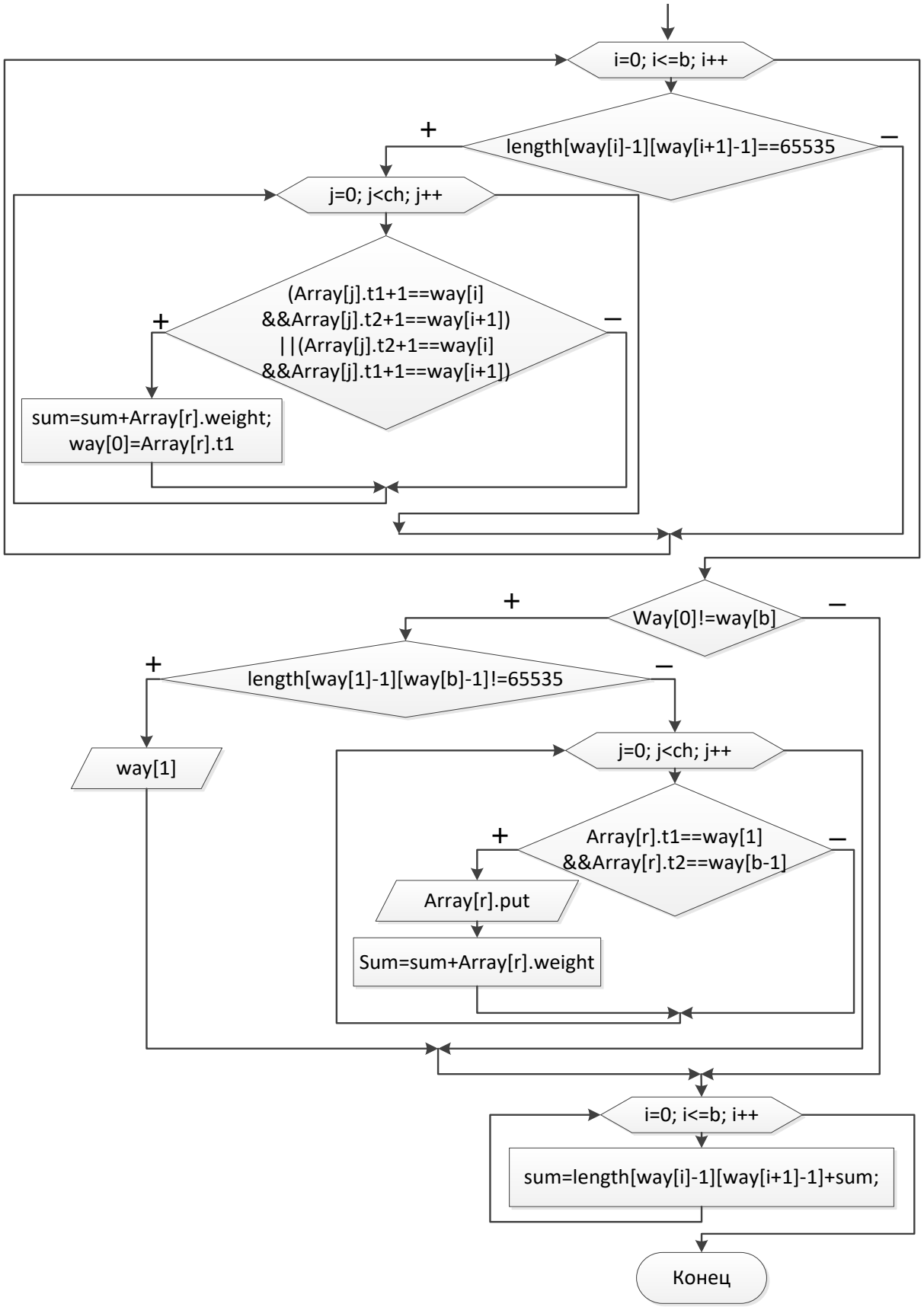

Рис. 2.5. Блок-схема выполнения функции Euler()

3. РАЗРАБОТКА И ТЕСТИРОВАНИЕ ПРОГРАММЫ

3.1. Реализация алгоритма решения задачи почтальона

voidmain(intargc, char* argv[])

Главная функция программы. В ней происходит ввод данных с клавиатуры:

n-числа вершин

length[i][j]-расстояние между конкретными вершинами i и j

В результате чего формируется матрица смежности wordC[n][n]. В этой функции происходит вызов функции Dijkstra() и Euler();В функцию Dijkstra() передаются номера вершин с нечетной степенью в всевозможных попарных сочетаниях.

void Dijkstra()

Функция нахождения кратчайшего расстояния между переданными точками.

Функция ничего не возвращает, но при каждом вызове выводит на экран путь между заданными вершинами и длину этого пути.

Для работы данной функции необходимо подключить функцию int min(int n).

- int min(int n)- возвращает номер ближайшей не пройденной вершины

void Euler()

Функция нахождения эйлерова цикла в заданом графе. В случае, если данный цикл не может быть найден,цикл ищеться с добавлением необходимых ребер, после чего необходимое ребро заменяеться самой короткой цепью из возвращаемых функцией Dijkstra(). Так же в этой функции просчитывется суммарный вес обхода. Для работы данной функции необходимо подключение 2 функций:

- void search(int i)- динамическая функция непосредственного поиска пути. В ней же происходит обнуления рабочей матрицы смежности A[50][50]
- void not_connected()-функция выводящая сообщение о невозможности нахождения эйлерова цикла в данном графе(если граф не связан).

3.2.Тестирование разработанного приложения

Это приложение работает только лишь с графами число вершин менее 50 (данное количество было установлено при разработке, и по этой причине в случае нужды может варьироваться). Однако приложение основана не на динамическом распределении памяти, а на хранении элементов (матрицы смежности) в виде массива, но верхний предел значений все время будет иметь место.

Также эта программа не работает с отрицательными значениями в матрице смежности. Такое связанно с основой поставленной задачи.

Данная задача может иметь большое число различных решений, таким образом как единственный способ получения достоверного решения сводиться к полному перебору вариантов. Этот способ в данном труде не рассматривается.

Пример работы приложения в обычных условиях показан на рис 3.1.

Привведенных значений (которые указаны на рисунке выше) сформировалась матрица смежности. Как видно, единственные вершины с нечетными степенями – это вершины x1 (3 ребра) и x4(3 ребра). Они были самый короткий путь между ними x1-x4,вес этого пути равен 3.

После был составлен путь по ребрам:

(4 – 3)(3 – 5)(5 – 2) (2 – 4)(4 – 1)(1 – 3)(3 – 2)(2 – 1)(1 – 4)

Всего девять переходов. Дважды встречается переход по ребру(X4-X1), так как степени этих вершин нечетные.

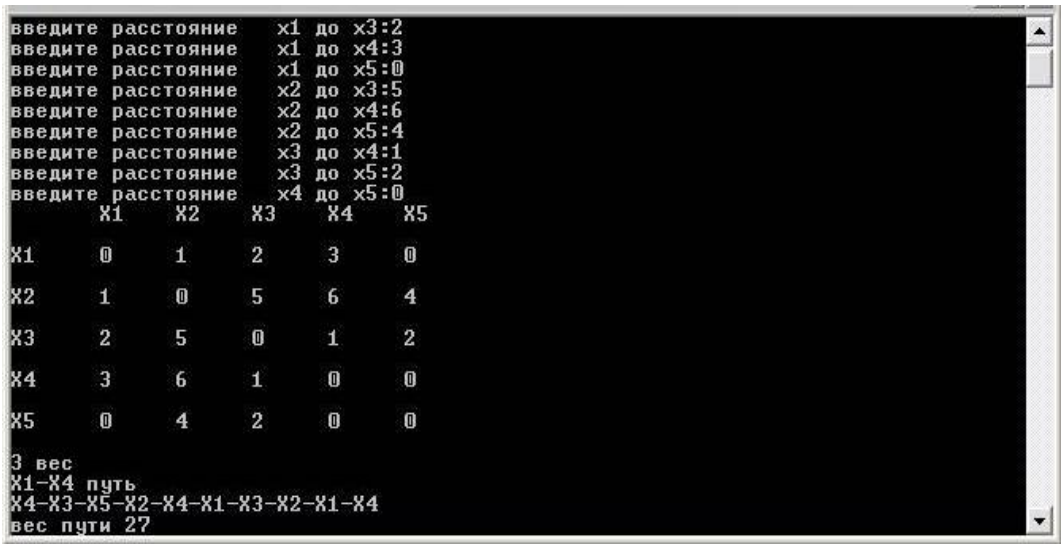

Рис 3.1. Результат работы программы

Минимальный путь обхода складывается из:

1+2+3+5+6+4+1+2=24 – длина эйлерова цикла (его можно просчитать сложив все числа над главной диагональю).

24+3=27(3-путь (4-1),который повторяется).

3.3. Эффективность работы алгоритма решения задачи почтальона

На рис. 3.2 приведены графики зависимости времени работы алгоритма в миллисекундах от количества ребер и вершин. График построен в логарифмической шкале.

Как можно заметить при достижении 500 вершин и 1000 ребер заметно сильное увеличение времени работы алгоритма. Так же стоит заметить,что время работы алгоритма на ориентированном графе незначительно выше. Наглядно можно увидеть на рис. 3.2.

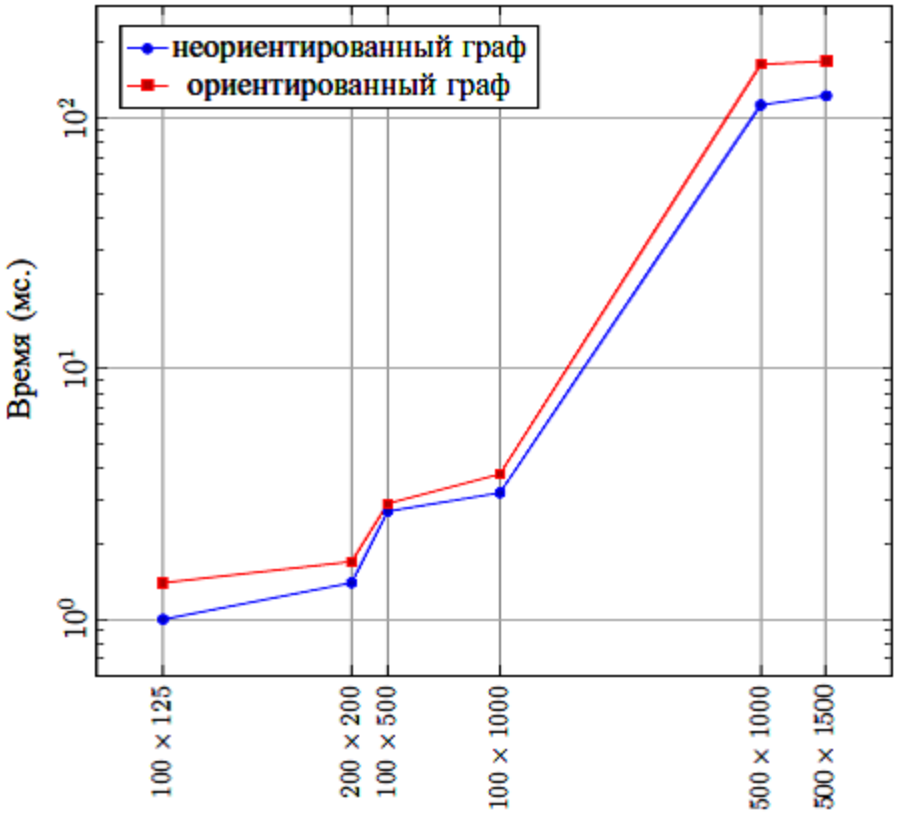

Рис. 3.2. Диаграмма зависимости времени работы алгоритма решения задачи почтальона отколичества вершин и ребер графа.

ЗАКЛЮЧЕНИЕ

В данной выпускной квалификационной работе были выполнены все поставленные задачи. Были реализованы алгоритмы решения задачи почтальона, и проведено их тестирование.

В рамках данной работы был реализован алгоритм решения задачи почтальона на языке C++ в среде MicrosoftVisualStudio 2015.

Следует заметить, что алгоритм решения задачи почтальона может применяться для различных задач маршрутизации, как например для разработки кратчайшего маршрута снегоуборочной машины.

Стоит заметить, что только в случае ориентированного и неориентированного графа задача почтальона имеет полиномиальное решение.

На диаграмме видно, что поиск кратчайшего пути на ориентированном графе занимает больше времени. Так же стоит отметить: при достижении 500 вершин и 1000 ребер заметносильное увеличение времени работы алгоритма.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

[1] Кармен, Т. Алгоритмы. Построение и анализ / Т. Кармен. — Издательство «Вильямс», 2010.

[2] Munchers, J.Algorithms for the Assignment and Transportation Problems / J. Munchers.— 1957.— Pp. 32–38.

[3] Kuhn, H.W. The Hungarian Method for the assignment problem / H. W. Kuhn. — 1955.— Pp. 83–97.

[4] Асанов, М. Дискретная математика: Графы, матрицы, алгоритмы / М. Асанов, В. Баранский, В. Расин. — СПБ: Издательство «Лань», 2010.

[5] Jensen, J. Digraphs.Theory, Algorithms and Applications / J. Jensen.— 2007.

[6] Берж, К. Теория графов и ее приложения / К. Берж. — Москва: издательство иностранной литературы, 1962.

[7] Новиков, Ф. Дискретная математика для программистов. / Ф. Новиков. — Издательство «Питер», 2000.

[8] Ловас, Л. Н. Прикладные задачи теории графов / Л. Н. Ловас. — Москва: Мир, 1998.

[9] Кристофидес, Н. Теория графов. Алгоритмический подход. / Н. Кристофидес. — Издательство «Мир», 1978г.

[10] Басакер, Р. Конечные графы и сети / Р. Басакер, Т. Сали. — Москва: Наука,1974.

[11] В.А. Носов. Комбинаторика и теория графов, МГТУ, 1999 г.

[12] Ф.А. Новиков. Дискретная математика для программистов, Питер, 2001г.

[13] В.М. Бондарев, В.И. Рублинецкий, Е.Г. Качко. Основы программирования, 1998 г.

[14] Ильев, В.П. Комбинаторные задачи на графах: учебное пособие / В. П. Ильев. - Омск: Изд-во Ом.гос. ун-та, 2013. - 80 с.

Приложение

```
#include"stdafx.h"
#include<iostream>
#include<cstring>
#include<cstdio>
#include<cstdlib>
#include<conio.h>
#include<algorithm>
#include"locale.h"
#defineuse_CRT_SECURE_NO_WARNINGS


usingnamespacestd;

voiddijkstra();
voidnot_connected();
voidEuler();
void search(inti);
int a[50][50];
inti, j, p, xn, xk, z, k = 1, Mas[100][100], ch = 0;
int vertex[10000];                          //степеньвершин
int way[10000];                             //Эйлеровцикл
intflag[10000];                        //компоненты связности
int x, y, w;
int n, m;                          // m - число дуг, n - число вершин
intcountName;                              // числокомпонентсвязности
unsignedint c[50][50], distance[50];
char s[80], path[80][50];
structst {
      char put[50];
      int x1;
      int x2;
      int weight;
};
stArray[100];
int min(int n)
{
      inti, result;
      for (i = 0; i<n; i++)
      if (!(flag[i])) result = i;
      for (i = 0; i<n; i++)
      if ((distance[result]>distance[i]) && (!flag[i])) result = i;
```

```
	return result;
}

void main(intargc, char* argv[])
{
	setlocale(LC_ALL, "Russian");
	cout<<"введитечисловершин: ";
	cin>> n;
	for (i = 0; i<n; i++)
	for (j = 0; j<n; j++) c[i][j] = 0;
	for (i = 0; i<n; i++)
	for (j = i + 1; j<n; j++)
	{
		cout<<"введите расстояние x"<< i + 1 <<" до x"<< j + 1
<<":";
		cin>> c[i][j];
	//записываем расстояния в матрицу c
	}
	cout<<" ";
	for (i = 0; i<n; i++) cout<<"\t"<<"X"<<i + 1 ;
	cout<<endl<<endl;
	for (i = 0; i<n; i++)
	{
		cout<<"X"<< i+1;
		for (j = 0; j<n; j++)
		{
			cout<<"\t"<< c[i][j];
			c[j][i] = c[i][j];
		}
		cout<<"\n\n";
	}
	for (i = 0; i<n; i++) {
		for (j = 0; j<n; j++)
		if (c[i][j] == 0)
			c[i][j] = 65535;
	}
	for (i = 0; i<n; i++)
	{
		z = 0;
		for (j = 0; j<n; j++) {
			if (c[i][j] != 65535)
				z++;
		}
		if (z % 2 == 1)		//поиск вершин с нечетными степенями
```

```
            {
                  Mas[k][0] = i;  //сохраняем номера этих вершин
                  Mas[0][k] = i;
                  k++;
            }
      }
      for (int m = 1; m<k; m++) {
            xn = Mas[0][m];
            for (j = m; j<k - 1; j++)
            {
                  xk = Mas[j + 1][0];
                  if (xn != xk)
                        Dijkstra();
            }
      }
      Euler();
      system("pause");
}
//---------------------------------------------------------------------
void Dijkstra()
{
      for (i = 0; i<n; i++)
      {
            flag[i] = 0;
            distance[i] = 65535;
      }
      distance[xn] = 0;
      flag[xn] = 1;
      p = xn;
      _itoa_s(xn + 1, s, 10);          //преобразование числа в символы
      for (i = 1; i<= n; i++)
      {
            strcpy_s(path[i], "X");
            strcat_s(path[i], s);
      }
      do
      {
            for (i = 0; i<n; i++)
            if ((length[p][i] != 65535) && (!flag[i]) && (i != p))
            {
                  if (distance[i]>distance[p] + length[p][i])
                  {
                        _itoa_s(i + 1, s, 10);
                        strcpy_s(path[i + 1], path[p + 1]);
```

```
                    strcat_s(path[i + 1], "-X");
                    strcat_s(path[i + 1], s);
                    distance[i] = distance[p] + c[p][i];
               }
               else
               distance[i] =distance[i];
          }
          p = min(n);
          flag[p] = 1;
     } while (p != xk);
     if (distance[p] != 65535)
     {
          Array[p].weight = (int)distance[p];
          i = 0;
          while (path[p + 1][i] != '\0') {
               Array[ch].put[i] = path[p + 1][i];
               i++;
          }
          Array[ch].x1 = xk;
          Array[ch].x2 = xn;
          cout<<"вес\n"<<Array[p].weight;
          cout<<"путь \n"<<Array[ch].put;
     }

     ch++;
}
//---------------------------------------------------------------------
voidEuler()
{
     intpath_weight = 0;
     for (i = 0; i<n; i++)
     for (j = 0; j<n; j++) {
          a[i + 1][j + 1] = (int)c[i][j];
          if (a[i + 1][j + 1] == 65535)
               a[i + 1][j + 1] = 0;
     }

     countName = 0;
     for (inti = 1; i<n; i++)
     {
          if (flag[i] == 0)
               countName++;
          if (countName>1)
               not_connected();           // графнесвязен
```

```
	}
	for (inti = 1; i<n; i++)
	if (vertex[i] % 2 == 1)
		not_connected();		// естьвершинынечётнойстепени
	w = 0;
	search(1);
	way[0] = way[1];
	for (inti = 1; i<= w; i++) {
		if (lengh[way[i] - 1][way[i + 1] - 1] == 65535) {
			for (j = 0; j<ch; j++) {
				if ((Array[j].x1 + 1 == way[i] &&Array[j].x2 + 1
== way[i + 1]) || (Array[j].x2 + 1 == way[i] &&Array[j].x1 + 1 ==
way[i + 1])) {
					cout<< mas[p].put;
					path_weight = path_weight + Array[p].weight;
					way[0] = Array[p].x1;
				}
			}
		}
		else
			cout<<"X-"<< way[i];
	}
	if (way[0] != way[w]) {
		if (length[way[1] - 1][way[w] - 1] != 65535) {
			cout<<"X"<< way[1];
			path_weight = path_weight + length[way[i] - 1][way[w]
- 1];
		}
		else {
			for (j = 0; j<ch; j++) {
				if (Array[p].x1 == way[1] &&Array[p].x2 == way[w
- 1]) {
					cout<<Array[p].put;
					path_weight = path_weight + Array[p].weight;
				}
			}
		}
	}
	for (inti = 0; i<= w; i++) {
		if (length[way[i] - 1][way[i + 1] - 1] != 65535)
			path_weight = length[way[i] - 1][way[i + 1] - 1] +
path_weight;
	}
	cout<<"\n";
```

```
      cout<<"веспути "<<path_weight;
}
//-------------------------------------------------
voidnot_connected() {

      cout<<"Графнесвязен! \n";
      system("pause");
      exit(0);
}

//-------------------------------------------------
void search(inti) {
      int j;
      for (int j = 1; j <= n; j++)
      if (a[i][j] != 0) {
            a[i][j] = 0;
            a[j][i] = 0;
            search(j);
      }
      w++;
      way[w] = i;
}
```