

Министерство образования и науки Российской Федерации
Пензенский государственный университет
Кафедра «Вычислительная техника»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К курсовому проектированию
по курсу «Логика и основы алгоритмизации
в инженерных задачах»
на тему «Реализация алгоритма Прима»

ОТМ
18.12.24


Выполнил:

студент группы

23BVB3

Точнов Е. Е.

Принял:

Митрохин М. А.

Пенза 2024

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет Вычислительной техники

Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ Михайлов М.А.

 « » 20

ЗАДАНИЕ

на курсовое проектирование по курсу

Логика и основы алгоритмизации в инженерных задачах
Студенту Тючкову Евгению Евгеньевичу Группа 23ВВВ3
Тема проекта Реализация алгоритма Фриша

Исходные данные (технические требования) на проектирование

Разработка алгоритмов и программного обеспечения
в соответствии с данным заданием курсового проекта.
Техническая записка должна содержать:

1. Постановку задачи;
2. Теоретическую часть задания;
3. Описание алгоритма поставленной задачи;
4. Пример ручного решения задачи и вычисления (на
небольшом участке работы алгоритма);
5. Описание самого программного;
6. Тесты;
7. Список литературы;
8. Лицензия программного;
9. Результаты работы программы.

Объем работы по курсу

1. Расчетная часть

Ручной расчет работы алгоритма

2. Графическая часть

Схема алгоритма в формате блок-схем.

3. Экспериментальная часть

Тестирование программы;
Результаты работы программы на тестовых данных

Срок выполнения проекта по разделам

- 1 Исследование теоретической части курсового
- 2 Разработка алгоритмов программы
- 3 Разработка программы
- 4 Тестирование и завершение разработки программы
- 5 Оформление пояснительной записки
- 6
- 7
- 8

Дата выдачи задания "03" 09.2024

Дата защиты проекта " " "

Руководитель _____

Задание получил " " 20__ г.

Студент _____

Содержание

Реферат.....	5
Введение.....	6
1. Постановка задачи.....	7
2. Описание алгоритма программы.....	8
3. Описание модулей программы.....	11
4. Описание работы программы.....	13
5. Тестирование.....	17
6. Ручной расчет задачи.....	20
Заключение.....	21
Список литературы.....	22
Приложение А.Листинг программы.....	23

Реферат

Отчёт 27 стр., 9 рисунков.

ГРАФ, ТЕОРИЯ ГРАФОВ, ПРИМ, ДЕРЕВО, МИНИМАЛЬНОЕ ОСТОВНОЕ ДЕРЕВО.

Цель исследования — разработка программы, способной находить в графе минимальное остовное дерево, используя алгоритм Прима.

В работе рассмотрен алгоритм Прима для поиска минимального остовного дерева в неориентированном графе. Алгоритм строит дерево, начиная с произвольной вершины и постепенно добавляет ребро с наименьшим весом, которое соединяет вершину в дереве с вершиной вне дерева, при условии, что добавление ребра не создает цикла. Этот алгоритм гарантированно находит минимальное остовное дерево для любого связного неориентированного графа с весами рёбер.

Введение

Алгоритм Прима представляет собой эффективный метод построения минимального остовного дерева в неориентированном связном графе. Данный алгоритм был впервые предложен математиком Робертом Примом в 1959 году и с тех пор получил широкое применение в различных областях, таких как проектирование сетей, организация логистики, а также при решении задач оптимизации.

Отличительной особенностью алгоритма Прима является его жадный характер, заключающийся в последовательном добавлении к растущему остовному дереву ребер с минимальными весами. Это позволяет достичь высокой вычислительной эффективности, что делает данный метод особенно привлекательным для работы с графами большого размера. Однако стоит отметить, что в некоторых случаях алгоритм Прима может не находить глобальный оптимум, а лишь локальный, что может рассматриваться как его недостаток.

Несмотря на свой недостаток, благодаря своей простоте реализации и высокой производительности, алгоритм Прима занимает важное место в арсенале методов, применяемых в современной теории графов и оптимизации.

В качестве среды разработки был выбран текстовый редактор Microsoft Visual Code, а язык программирования — Golang.

Целью данной курсовой работы является разработка программы на языке Golang, который постепенно завоёвывает популярность за счёт простоты синтаксиса, богатой стандартной библиотекой и скорости компиляции. С его помощью в данном курсовом проекте реализуется алгоритм Прима, осуществляющий построение минимального остовного дерева.

1. Постановка задачи

Требуется разработать программу, которая будет способна построить минимальное остовное дерево из исходного графа, используя алгоритм Прима.

Исходный граф в программе должен задаваться списком смежности, причём при вводе данных должны быть предусмотрены граничные условия.

Программа должна работать так, чтобы пользователь вводил параметры ребер для создания списка смежности. После обработки этих данных на экран должен выводиться список смежности графа. Необходимо предусмотреть различные исходы поиска, чтобы программа не выдавала ошибок и работала правильно. Помимо этого требуется сделать вывод графического представления полученных графов и минимального остовного дерева полученного после выполнения алгоритма Прима.

Устройство ввода – клавиатура.

2. Описание алгоритма программы

Теория графов представляет собой раздел дискретной математики, изучающий структуры, состоящие из множества вершин и ребер, которые соединяют эти вершины. В контексте алгоритмов построения минимального остовного дерева алгоритм Прима занимает особое место.

Алгоритм Прима — это эффективный метод построения минимального остовного дерева во взвешенном неориентированном графе. Его основная цель заключается в нахождении подмножества ребер, которые соединяют все вершины графа с минимальной суммарной стоимостью.

Принцип работы алгоритма заключается в последовательном построении минимального остовного дерева. Алгоритм начинает с произвольной вершины графа и последовательно добавляет ребра с наименьшим весом, которые соединяют уже выбранные вершины с новыми вершинами, гарантируя формирование остовного дерева с минимальной суммарной стоимостью.

С математической точки зрения, алгоритм принимает на вход взвешенный неориентированный связный граф $G(V, E)$ и produce остовное дерево T , являющееся подмножеством ребер E с минимальной суммой весов. Алгоритмическая сложность варьируется от квадратичной при naïve реализации до логарифмической при использовании бинарной кучи.

Для реализации алгоритма требуются специфические структуры данных, включающие приоритетную очередь, массив посещенных вершин и массив минимальных весов ребер. Математическая модель алгоритма предполагает последовательную инициализацию начальной вершины, ее маркировку как посещенной, добавление инцидентных ребер в приоритетную очередь, выбор ребра с минимальным весом и добавление нового ребра в остовное дерево с повторением процесса до охвата всех вершин графа.

Алгоритм Прима обладает рядом существенных преимуществ, среди которых простота реализации, эффективность для разреженных графов и гарантия получения оптимального решения. Однако следует учитывать и его недостатки, такие как квадратичная сложность при наивной реализации и неэффективность для густонаселенных графов.

Практическое применение алгоритма охватывает широкий спектр областей, включая проектирование транспортных сетей, прокладку коммуникационных линий, оптимизацию логистических маршрутов и моделирование инженерных и экономических систем.

Математическая строгость и эффективность алгоритма Прима делают его фундаментальным инструментом в теории графов и дискретной оптимизации, позволяющим решать сложные задачи минимизации в различных научных и прикладных контекстах.

Для программной реализации алгоритма потребуется два массива: `pq` — приоритетная очередь для каждого ребра графа; `visited` — для хранения всех посещенных вершин. Соответственно имеется граф $G = (V, E)$. Каждая из вершин которого изначально помечена как не посещенная в массиве `visited`.

В качестве исходной выбирается вершина `graphVertexStart` и ребро с такой конечной вершиной помещается в приоритетную очередь. После этого, пока приоритетная очередь не пуста, выполняется цикл в котором, из очереди достаётся ребро, а его конечная вершина становится текущей вершиной. Данная вершина помечается как посещённая и добавляется в граф представляющий собой минимальное остовное дерево. К общей стоимости дерева прибавляется стоимость текущей вершины.

После этого для каждого ребра связанного с текущей вершиной, конечная вершина которого не посещена, добавляется в приоритетную очередь. Цикл возвращается в своё начальное положение и повторяется до того, пока вся очередь не опустеет.

Ниже представлен псевдокод функции Prim().

Prim()

1. Создать пустую очередь pq;
2. Создать массив посещенных вершин visited;
3. Создать пустое минимальное остовное дерево mst;
4. Создать переменную общей стоимости дерева totCost;
5. pq.Push(E(V₂));
6. Пока pq не пуста делать
 7. curEdge = pq.Pop();
 8. curNode = curEdge.DestNode;
 9. Если visited[curNode] == true;
 10. Вернуться к началу цикла;
 11. Конец условия;
 12. visited[curNode] = true;
 13. Добавить в mst curEdge(V₁, V₂);
 14. totCost = totCost + curEdge.cost;
 15. Для каждого E в G(V, E) делать;
 16. Если visited[E(V₂)] == false делать;
 17. pq.Push(E(V₂));
 18. Конец условия;
 19. Конец цикла;

Полный код программы представлен в Приложении А.

3. Описание модулей программы

Программа состоит из двух основных модулей и четырех файлов, составляющих данные модули.

Модуль `main` является главным модулем программы, имеющим единственную функцию `main()` и состоящим из одного файла `kursach.go`, являющейся входной точкой в программу. Данная функция создаёт пустой исходный граф, который реализован во втором модуле, и выводит псевдографическое меню.

Само меню позволяет пользователю добавлять ребра в граф, отобразить список смежности и выполнить алгоритм Прима.

Модуль `graph` в свою очередь состоит из нескольких исходных файлов: `edge.go`, `graph.go` и `queue.go`. В файле `edge.go` находится реализация структуры ребра графа и имеет один метод — `GetCost()`, который возвращает стоимость самого ребра.

В файле `graph.go` реализована структура описывающая сам граф, как хэш таблицу ребер, реализованных в файле `edge.go`. Также здесь присутствует функция `New()`, возвращающая указатель на структуру графа. Помимо данной функций, в данном файле реализованы все основные методы для графа:

1. `AddEdge()` - добавляет ребро в граф и принимает номера исходной и конечной вершин, а также стоимость ребра.
2. `PrintAdjList()` - выводит список смежности графа.
3. `Prim()` - выполняет алгоритм Прима и принимает на вход стартовую вершину для выполнения обхода алгоритма и возвращает общую стоимость дерева, минимальное остовное дерево в виде графа и ошибку, если она была обработана.

4. `ConvertToGraphics()` - преобразует структуру графа для его представления в графическом виде и сохраняет файл данного представления.

Файл `queue.go` содержит в себе реализацию приоритетной очереди для ребер графа, которая необходима для эффективной реализации алгоритма Прима, описанной в файле `graph.go`. Приоритетная очередь состоит из пяти методов:

1. `Len()` - возвращает длину очереди.
2. `Less()` - принимает индексы элементов очереди и возвращает наименьший из них.
3. `Swap()` - принимает индексы двух элементов и меняет их местами.
4. `Push()` - добавляет элемент в очередь.
5. `Pop()` - достаёт последний элемент из очереди.

На рисунке 1 показана схема взаимодействия модулей программы.

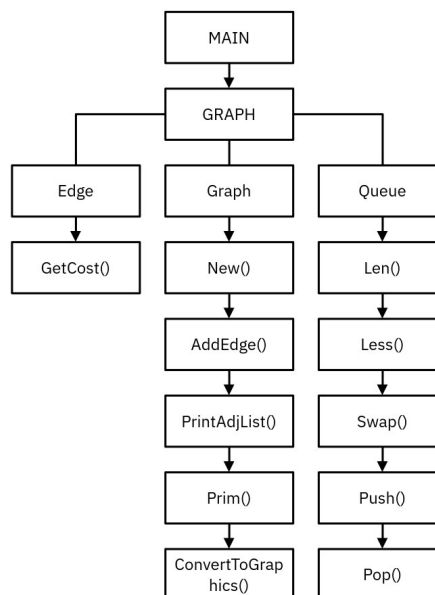


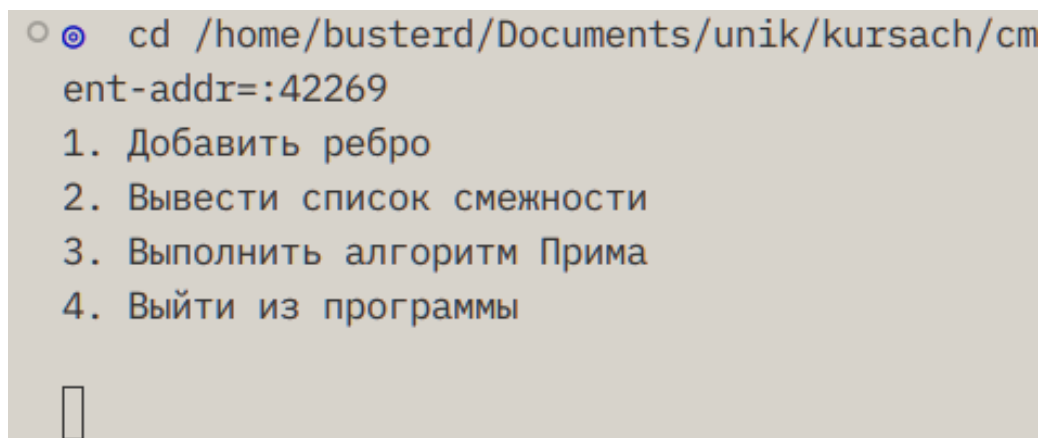
Рисунок 1 — Схема взаимодействия модулей программы

4. Описание работы программы

Для написания данной программы были использованы язык программирования Golang и среда разработки Microsoft Visual Code. Данный язык был разработан в корпорации Google и получил своё признание за счёт универсальности применения, простого, для понимания, синтаксиса, стандартной библиотеки охватывающей широкий спектр задач, высокой скоростью компиляции программ, а также их высокой производительностью.

Данная программа является многомодульной, так как состоит из нескольких так называемых «пакетов» (packages англ.), которые в свою очередь состоят из нескольких исходных файлов. Сама программа включает в себя следующие «пакеты»: main, graph.

Работа программы начинается с создания пустого графа и вывода псевдографического интерфейса меню, как показано на рисунке 1, где пользователю предлагается сделать выбор действий: добавить новое ребро к графу; вывести список смежности; выполнить алгоритм Прима; выйти из программы.



```
cd /home/busterd/Documents/unik/kursach/cm
ent-addr=:42269
1. Добавить ребро
2. Вывести список смежности
3. Выполнить алгоритм Прима
4. Выйти из программы
```

Рисунок 2 — Главное меню программы

В случае выбора добавления нового ребра пользователь должен ввести исходную и конечную вершины ребра, а также его вес, как показано на рисунке 2. После чего происходит возвращение в главное меню.

```
1. Добавить ребро
2. Вывести список смежности
3. Выполнить алгоритм Прима
4. Выйти из программы

1
Вершина источник: 1
Вершина назначение: 4
Стоимость ребра: 32
```

Рисунок 3 - Ввод нового ребра

При выборе вывода списка смежности графа происходит, собственно, вывод списка смежности графа, что показано на рисунке 3. После этого программа сделает запрос у пользователя на сохранение текущего графа в файле представления графов.

```
Список смежности:
6: 7 (14), 3 (493), 4 (212), 2 (241), 1 (50)
5: 3 (187), 8 (493), 1 (188), 7 (13), 2 (410)
2: 6 (241), 3 (74), 8 (183), 7 (494), 4 (135), 5 (410), 1 (477)
4: 3 (456), 6 (212), 7 (193), 1 (244), 2 (135)
3: 4 (456), 1 (112), 6 (493), 8 (113), 7 (392), 5 (187), 2 (74)
1: 3 (112), 5 (188), 8 (107), 4 (244), 6 (50), 7 (127), 2 (477)
7: 8 (310), 6 (14), 3 (392), 4 (193), 5 (13), 2 (494), 1 (127)
8: 7 (310), 3 (113), 2 (183), 5 (493), 1 (107)
Вы желаете экспортировать текущий граф в DOT файл?
1. Да
2. Нет

```

Рисунок 4 — Вывод списка смежности

Однако если пользователь выберет запуск алгоритма Прима, то программа выполнит своё основное предназначение и выведет список смежно-

```
Общая стоимость дерева: 505
Список смежности:
7: 6 (14), 5 (13)
5: 7 (13)
8: 1 (107)
3: 1 (112), 2 (74)
2: 3 (74), 4 (135)
4: 2 (135)
1: 6 (50), 8 (107), 3 (112)
6: 1 (50), 7 (14)
Вы желаете экспортировать МОД в DOT файл?
1. Да
2. Нет
☐
```

Рисунок 5 — Вывод результата выполнения алгоритма Прима

В случае если пользователь решит сохранить текущее представление графа в графическом формате, ему будет предложено выбрать имя файла. После этого текущее представление графа будет сохранено в формате DOT, языка выполняющего представление любых графов, который затем можно преобразовать в привычное растровое или векторное изображение, с использованием набора инструментов Graphviz, ориентированных для работы

с файлами представления графов. На рисунках 5-6 показаны растровые изображения графа и минимального остовного дерева, который был задан и найдено, соответственно, ранее на рисунках 3-4, сохраненные с помощью

данной программы в формат DOT и впоследствии преобразованы сторонними средствами для графического представления графов.

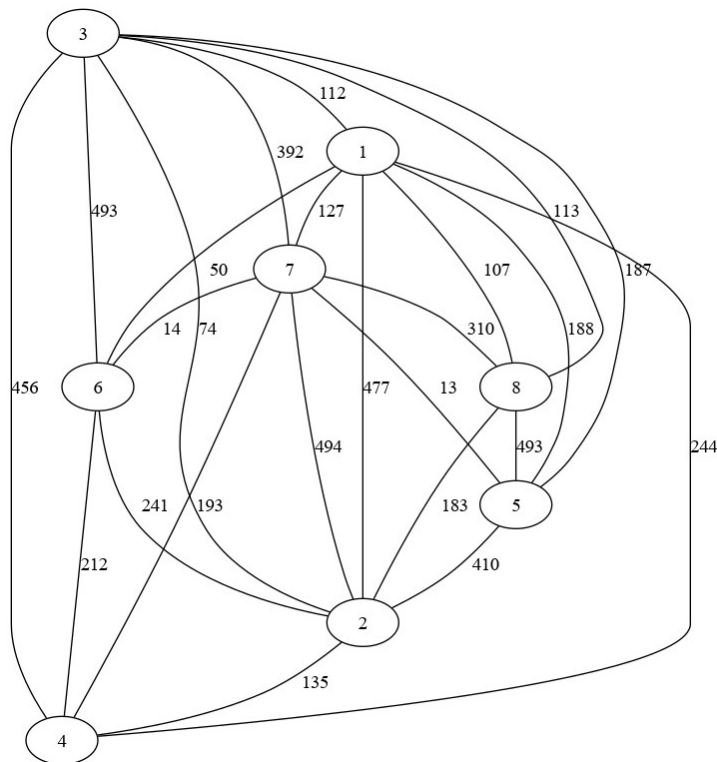


Рисунок 6 — Графическое представление, ранее введенного пользователем, графа

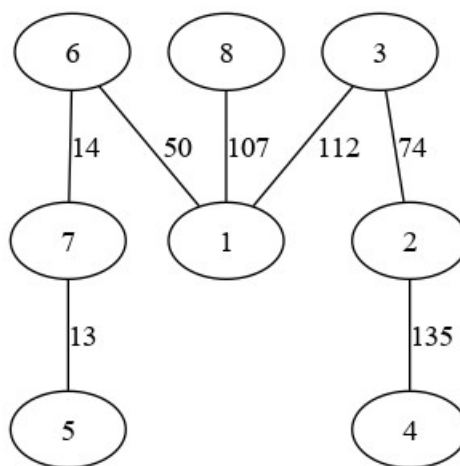


Рисунок 7 — Найденное минимальное остовное дерево с использованием алгоритма Прима

5. Тестирование

Для тестирования и отладки программы использовались встроенные средства среды разработки Microsoft Visual Code, так как она предоставляет все необходимые инструменты для отладки многомодульных программ, написанных на языке Golang.

В процессе разработки, параллельно с написанием кода, проводилось непрерывное тестирование. Это позволило выявить и исправить большое число ошибок, затрагивающих ввод данных, отображение результатов, алгоритмическую логику и взаимодействие отдельных функций.

Ниже представлен результат тестирования ввода пользователем различных стартовых вершин обхода графа посредством алгоритма Прима.

```
Введите стартовую вершину: 4
Общая стоимость дерева: 505
Список смежности:
1: 3 (112), 6 (50), 8 (107)
6: 1 (50), 7 (14)
7: 6 (14), 5 (13)
5: 7 (13)
8: 1 (107)
4: 2 (135)
2: 4 (135), 3 (74)
3: 2 (74), 1 (112)
```

Рисунок 8 — Результат тестирования при вводе пользователем стартовой вершины 4

Введите стартовую вершину: 789
данной вершины не существует

1. Добавить ребро
2. Вывести список смежности
3. Выполнить алгоритм Прима
4. Выйти из программы

Рисунок 9 — Результат тестирования при вводе пользователем не существующей вершины

Таблица 1 — Описание поведения программы при тестировании

Описание теста	Ожидаемый результат	Полученный результат
Запуск программы	Вывод псевдографического меню предоставляющего выбор необходимого действия	Верно
Выбор пункта для добавления ребра графа	Запрос пользователя на ввод параметров ребра: начальной и конечной точек, его стоимости	Верно
Выбор пункта для отображения списка смежности	Вывод списка смежности графа и запроса на сохранение представления графа	Верно
Согласие сохранить файл представления графа	Сохранение файла представления	Верно

Продолжение таблицы 1

Описание теста	Ожидаемый результат	Полученный результат
Выбор пункта для выполнения алгоритма Прима	Запрос на выбор стартовой вершины, последующее выполнение алгоритма Прима и вывод списка смежности найденного минимального остовного дерева, а также запрос на сохранение его представления в виде графа	Верно
Согласие сохранить файл представления минимального остовного дерева в виде графа	Сохранение файла представления	Верно
Ввод не существующей вершины	Вывод соответствующего сообщения и возвращение в главное меню	Верно
Выбор пункта выхода из программы	Завершение программы	Верно

В результате тестирования было выявлено, что программа успешно выполняет поставленные перед ней задачи и корректно проверяет пользовательский ввод.

6. Ручной расчет задачи

Проведём проверку работы программы посредством ручного расчёта, на примере графа представленного на рисунке 3.

Ручная проверка метода построения минимального остовного дерева начинается с произвольной вершины, в данном случае — с вершины 1. На первом шаге осуществляется поиск ребра с наименьшим весом, инцидентного вершине 1. Таким ребром является ребро, соединяющее вершину 1 с вершиной 6, имеющее вес 50. Это ребро включается в остовное дерево.

Процесс продолжается. Теперь рассматривается вершина 2 (не включенная в остовное дерево) и осуществляется поиск ребра с минимальным весом, соединяющего её с уже существующей частью дерева. Это ребро соединяет вершину 2 с вершиной 3 и имеет вес 75. Ребро включается в остовное дерево.

Аналогичным образом, для вершины 3 находится минимальное ребро, ведущее к вершине 1 (вес 112), для вершины 4 — к вершине 2 (вес 135), для вершины 5 — к вершине 7 (вес 13), и для вершины 6 — к вершине 7 (вес 14). Все найденные ребра включаются в остовное дерево.

Наконец, для вершины 8 выполняется поиск минимального ребра, соединяющего её с уже построенной частью дерева. Это ребро ведет к вершине 1 и имеет вес 107. Ребро включается в остовное дерево. Проверка завершается, так как все вершины уже объединены в одно связное множество, и дальнейшее добавление ребер не требуется. Вершина 7, имеющая минимальные веса ребер, уже включена в дерево и не приводит к добавлению новых ребер минимального веса. В результате проверки получено минимальное остовное дерево.

Таким образом, можно сделать вывод, что программа работает правильно.

Заключение

В процессе создания данного проекта была разработана программа реализующая алгоритм Прима для поиска минимального остовного дерева в среде разработки Microsoft Visual Code.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания списков смежностей, основанных на теории графов. Приобретены навыки по построению алгоритма поиска минимального остовного дерева. Углублены знания языка программирования Golang.

Недостатком данной программы является примитивный псевдографический интерфейс, ввиду отсутствия в стандартной библиотеке языка методов для создания оконных и графических приложений.

Программа обладает небольшим, но достаточным набором возможностей для поставленных перед ней задач.

Список литературы

1. Саммерфильд, М. Программирование на Go. Разработка приложений XXI века : практическое руководство / М. Саммерфильд ; пер. с англ. А. Н. Киселёва. — 2-е изд. - Москва : ДМК Пресс, 2023. - 581 с. - ISBN 978-5-89818-611-1. - Текст : электронный. - URL: <https://znanium.com/catalog/product/2108489> (дата обращения: 09.12.2024). — Режим доступа: по подписке.
2. Батчер, М. Go на практике / Мэтт Батчер, Мэтт Фарина ; пер. с англ. Р.Н. Рагимова ; под науч. ред. А.Н. Киселева. - Москва : ДМК Пресс, 2017. - 374 с. - ISBN 978-5-97060-477-9. - Текст : электронный. - URL: <https://znanium.com/catalog/product/1028131> (дата обращения: 09.12.2024). — Режим доступа: по подписке.
3. Калитин, Д. В. Основы дискретной математики : теория графов : практикум / Д. В. Калитин, О. С. Калитина. - Москва : Изд. Дом НИТУ «МИСиС», 2017. - 67 с. - ISBN 978-5-906846-68-6. - Текст : электронный. - URL: <https://znanium.com/catalog/product/1230538> (дата обращения: 09.12.2024). — Режим доступа: по подписке.
4. Алексеев, В. Е. Графы и алгоритмы. Структуры данных. Модели вычислений : краткий учебный курс / В. Е. Алексеев, В. А. Таланов. - Москва : ИНТУИТ, 2016. - 104 с. - ISBN 5-9556-0066-3. - Текст : электронный. - URL: <https://znanium.ru/catalog/product/2139047> (дата обращения: 09.12.2024). — Режим доступа: по подписке.

Приложение А.

Листинг программы

Файл prim.go

```
package main

import (
    "fmt"
    "os"
    "primalgorithm/internal/graph"
    "strings"
)

func main() {
    var g *graph.Graph = graph.New()
    for {
        var (
            option    uint
            src, dest uint
            cost      int
        )
        fmt.Println("1. Добавить ребро")
        fmt.Println("2. Вывести список смежности")
        fmt.Println("3. Выполнить алгоритм Прима")
        fmt.Println("4. Выйти из программы")
        fmt.Printf("\n")
        fmt.Scan(&option)
        if option == 4 {
            os.Exit(0)
        }
        switch option {
        case 1:
            fmt.Printf("Вершина источник: ")
            fmt.Scanf("%d", &src)
            fmt.Printf("Вершина назначение: ")
            fmt.Scanf("%d", &dest)
            fmt.Printf("Стоимость ребра: ")
            fmt.Scanf("%d", &cost)
            g.AddEdge(src, dest, cost)
        case 2:
            g.PrintAdjList()
            for {
                DOT файл?"
                fmt.Println("Вы желаете экспортировать текущий граф в")
                fmt.Println("1. Да")
                fmt.Println("2. Нет")
                fmt.Scan(&option)

                if option == 2 {
                    break
                } else {
                    var (
                        fileIn    string
                        filePath = "./graph.dot"
                    )
                    fmt.Println("Куда вы хотите сохранить файл?")
                    fmt.Scan(&fileIn)
```

```

        fileIn = strings.TrimSpace(fileIn)
        if fileIn != "" {
            filePath = fileIn
        }
        err := g.ConvertToGraphics(filePath)
        if err != nil {
            fmt.Println("Ошибка сохранения файла!")
            fmt.Println(err)
        }
        break
    }
}
case 3:
    fmt.Printf("Введите стартовую вершину: ")
    fmt.Scan(&option)
    cost, mst, err := g.Prim(option)
    if err != nil {
        fmt.Println(err)
        continue
    }
    fmt.Printf("Общая стоимость дерева: %d\n", cost)
    mst.PrintAdjList()
    for {
        fmt.Println("Вы желаете экспортировать МОД в DOT
файл?")

        fmt.Println("1. Да")
        fmt.Println("2. Нет")
        fmt.Scan(&option)
        if option == 2 {
            break
        } else {
            var (
                fileIn string
                filePath = "./graph.dot"
            )
            fmt.Println("Куда вы хотите сохранить файл?")
            fmt.Scan(&fileIn)
            fileIn = strings.TrimSpace(fileIn)
            if fileIn != "" {
                filePath = fileIn
            }
            err := mst.ConvertToGraphics(filePath)
            if err != nil {
                fmt.Println("Ошибка сохранения файла!")
                fmt.Println(err)
            }
            break
        }
    }
}
default:
    fmt.Println("Неизвестная опция. Попробуйте снова...")
}
}
}

```

Файл graph.go

```

package graph
import (
    "container/heap"
    "errors"
    "fmt"

```

```

        "os"
        "github.com/dominikbraun/graph"
        "github.com/dominikbraun/graph/draw"
    )
    type Graph struct {
        adjList map[uint][]Edge
    }
    func New() *Graph {
        return &Graph{adjList: make(map[uint][]Edge)}
    }
    func (g *Graph) AddEdge(source uint, dest uint, cost int) {
        g.adjList[source] = append(g.adjList[source], Edge{Source: source, Dest:
        dest, cost: cost, index: 0})
        g.adjList[dest] = append(g.adjList[dest], Edge{Source: dest, Dest:
        source, cost: cost, index: 0})
    }
    func (g Graph) PrintAdjList() {
        fmt.Println("Список смежности:")
        for i, n := range g.adjList {
            fmt.Printf("%d: ", i)
            for j, v := range n {
                fmt.Printf("%d (%d)", v.Dest, v.cost)
                if j < len(n)-1 {
                    fmt.Print(", ")
                }
            }
            fmt.Print("\n")
        }
    }
    func (g Graph) Prim(graphStartVertex uint) (uint, Graph, error) {
        pq := make(priorQueue, 0)
        visited := make(map[uint]bool)
        mst := New()
        totCost := 0
        if _, ok := g.adjList[graphStartVertex]; !ok {
            return 0, Graph{}, errors.New("данной вершины не существует")
        }
        heap.Push(&pq, Edge{0, graphStartVertex, 0, 0})
        for len(pq) > 0 {
            curEdge := heap.Pop(&pq).(Edge)
            curNode := curEdge.Dest
            if visited[curNode] {
                continue
            }
            visited[curNode] = true
            if curEdge.Source != 0 {
                mst.AddEdge(curEdge.Source, curEdge.Dest, curEdge.cost)
            }
            totCost += curEdge.cost
            for _, edge := range g.adjList[curNode] {
                if !visited[edge.Dest] {
                    heap.Push(&pq, edge)
                }
            }
        }
        return uint(totCost), *mst, nil
    }
    func (g Graph) ConvertToGraphics(output string) error {
        gf := graph.New(graph.IntHash)
        for k := range g.adjList {
            gf.AddVertex(int(k))
        }
    }

```

```

        for _, v := range g.adjList {
            for _, e := range v {
                gf.AddEdge(int(e.Source), int(e.Dest),
                    graph.EdgeWeight(e.GetCost()),
                    graph.EdgeAttribute("label",          fmt.Sprintf("%d",
e.GetCost()))))
            }
        }
        f, err := os.Create(output)
        if err != nil {
            return err
        }
        defer f.Close()
        if err = draw.DOT(gf, f); err != nil {
            return err
        }
        return nil
    }
}
func (g Graph) findStartPoint() uint {
    const maxStartVertex = 1000
    for i := uint(0); i < maxStartVertex; i++ {
        if _, ok := g.adjList[i]; ok {
            return i
        }
    }
    return 1
}
}

```

Файл edge.go

```

package graph
type Edge struct {
    Source uint
    Dest   uint
    cost   int
    index  int
}
func (e Edge) GetCost() int {
    return e.cost
}

```

Файл queue.go

```

package graph
type priorQueue []Edge
func (pq priorQueue) Len() int { return len(pq) }
func (pq priorQueue) Less(i int, j int) bool {
    return pq[i].cost < pq[j].cost
}
func (pq priorQueue) Swap(i int, j int) {
    pq[i], pq[j] = pq[j], pq[i]
    pq[i].index = i
    pq[j].index = j
}
func (pq *priorQueue) Push(x interface{}) {
    n := len(*pq)
    item := x.(Edge)
    item.index = n
    *pq = append(*pq, item)
}

```

```
func (pq *priorQueue) Pop() interface{} {  
    old := *pq  
    n := len(old)  
    item := old[n-1]  
    old[n-1] = Edge{}  
    item.index = -1  
    *pq = old[0 : n-1]  
    return item  
}
```