МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования
"Южно-Уральский государственный университет"
(национальный исследовательский университет)
Факультет Вычислительной математики и информатики
Кафедра экономико-математических методов статистики

ОТЧЕТпо дисциплине «Теория конечных графов и ее приложения»

Выполнил:
студент группы ВМИ-302 Е.К. Редькин

Проверил:
к.ф.-м.н., доцент
В.А. Голодов

Челябинск-2016

Оглавление

1. Практическая работа 1: Разработка класса Graph

Требуется реализовать класс, моделирующий граф, заданный матрицей смежности, списком смежности, или списком ребер. Класс должен иметь следующую функциональность:

— Ввод/вывод из файла

— Добавление/удаление ребра графа

— Изменение веса ребра графа

— Преобразование одного представления в другое

Все вершины графа пронумерованы натуральными числами от 1 до N.

1.1. Представление графа матрицей смежности

1.1.1. Описание входного файла

В первой строке тестового файла представлены: символ «*C*» — индикатор представления матрицей смежности и число «*N*»— количество вершин графа. Во второй строке файла записаны числа «*D, W*» — индикаторы ориентированности и взвешенности графа. Далее следует N строк по N чисел каждая, числа в которой разделены пробелом, где число на пересечении строки i и столбца j описывает ребро $i \to j$.

1.1.2.Описание структуры данных

Для хранения матрицы смежности было решено использовать следующую структуру: *std::vector<std::vector<int>>* — вектор векторов для хранения ребер и весов будущего представления графа. Этот способ позволяет выделить необходимое количество памяти, которая будет использоваться для представления графа.

1.2. Представление графа списком смежности

1.2.1. Описание входного файла

В первой строке файла записаны символ *«L»* — индикатор представления списком смежности и число *«N»* — количество вершин графа. Во второй строке записаны числа *«D, W»* — индикаторы ориентированности и взвешенности графа. Далее следует N — строк чисел *bi* (для невзвешенного графа) или пар *bi, wi* (для взвешенного графа), разделенных пробелом. Каждая строка описывает соседей очередной вершины, строка файла с номером *i + 2* описывает связи вершины *i*.

1.2.2. Описание структуры данных

Для хранения матрицы смежности используется тип данных *std::vector<std::map<int, int>>*, что позволяет осуществлять вставку, поиск, изменение и удаление за время, пропорциональное логарифму от числа элементов списка смежных вершин.

1.3. Представление графа списком ребер

1.3.1.Описание входного файла

В первой строке файла записаны символ «E» — индикатор представления списком смежности, число «N» — количество вершин графа и число «M» — количество ребер графа. Во второй строке записаны числа «D, W» — индикаторы ориентированности и взвешенности графа. Далее следует «M» — строк пар *ai, bi* (для невзвешенного графа) или троек *ai, bi, wi* (для взвешенного графа), разделенных пробелом.

1.3.2. Описание структуры данных

Для хранения матрицы смежности используется тип данных *std::vector<std::tuple<int, int, int>>*, что позволяет выделять ровно столько памяти. сколько нужно для представления графа.

1.4. Интерфейс класса Graph

Публичные методы:

— *void GraphFromFile(char*);* — инициализация графа из файла;

— *void GraphToFile(string);* — сохранить граф в файл;

— *void AdjMatrix(char*);*— сохранить матрицу смежности в указанный файл;

—*void ListOfNighbors(char*);*— сохранить список смежных вершин в указанный файл;

—*void ListOfEdges(char*);*— сохранить список ребер в указанный файл;

—*void AddEdge(vector<int>);*— добавить ребро (если уже есть, то обновить вес ребра);

—*void RemEdge(int x, int y);*— удалить ребро между вершинами x и y;

—*void EditEdge(vector<int>);*— редактировать ребро (если ребро отсутствует, то оно будет создано);

2. Практическая работа №2

Требуется обеспечить нахождение остовного леса оптимального веса для взвешенного неориентированного графа с помощью алгоритмов Прима, Краскала и Борувки.

2.1. Алгоритм Прима

На вход алгоритма подается связный неориентированный граф. Для каждого ребра задается его стоимость.

```
PRIM_MST:
     result = Graph(graph.V.size)
     fill(keys, INF) fill(parents, -
     1)
     keys[0] = 0
     queue = min_priority_queue(graph.V, keys)
     while not queue.empty
          v = queue.pop()
          if parent[v] != -1
               result.add(v, parents[v], keys[v])
          for vu in graph.E
               if u in queue and vu.weight< keys[u]
                    parents[u] = v
                    keys[u] = vu.weight
                    queue.push(u, keys[u])
     return result
```

Рис. 1. ЛистингпсевдокодаалгоритмаПрима

2.2. Алгоритм Краскала

На вход алгоритма подается связный неориентированный граф. Для каждого ребра задается его стоимость.

```
KRUSCAL_MST:
     result = Graph(graph.V.size)
     dsu = DSU(graph.V.size)
     graph.E.sort()
     for vu in graph.E
          if dsu.find(v) != dsu.find(u)
               dsu.join(v, u) result.add(v,
               u, vu.weight)
     return result
```

Рис. 2. Листинг псевдокода алгоритма Краскала

2.3. Алгоритм Борувки

На вход алгоритма подается связный неориентированный граф. Для каждого ребра задается его стоимость.

```
BORUVKA_MST()
     result = Graph(graph.V.size)
     dsu = DSU(graph.V.size)
     fill(min, INF)
     k = graph.V.size
     while k > 1
          is_forest = true for
          vu in graph.E
               if dsu.find(v) != dsu.find(u)
                    if vu.weight < graph.E[min[v]].weight
                         min[v] = vu
                         is_forest = false
                    if vu.weight < graph.E[min[u]].weight
                         min[u] = vu
                         is_forest = false
          if is_forest
               break
          for i = 1..graph.V.size
               if min[i] < INF
                    vu = graph.E[min[i]]
                    if dsu.find(v) != dsu.find(u)
                         result.add(v, u, vu.weight)
                         dsu.join(v, u)
                         dec(k)
     return result
```

Рис. 3. Листинг псевдокода алгоритма Борувки

2.4. Интерфейс класса Graph

Конструкторы:

— *Graph(intn);* — граф*«N»* изолированными вершинами.

Публичные методы:

— *GraphgetLinkTreePrim();* — методкласса, реализующийалгоритмПрима;

— *GraphgetLinkTreeKruscal();* — методкласса, реализующийалгоритмКраскала;

— *GraphgetLinkTreeBoruvka();* — методкласса, реализующийалгоритмБорувки;

2.5.Интерфейс класса DSU

Для реализации проверки принадлежности вершин к различным компонентам связности была реализована структура данных «система непересекающихся множеств».

Конструкторы:

— *DSU(intn)* — система из N изолированных элементов;

Публичные методы:

— intfindingSets(intx) — в каком множестве находится элемент x;

— voidunitSets(intx, inty) — объединение множеств, которым принадлежат x и y.

3. Практическая работа №3

Требуется реализовать проверку существования Эйлерова пути или цикла в графе, алгоритма Флери —поиска Эйлерова пути или цикла в графе, и эффективный алгоритм построения Эйлерова пути.

3.1. Проверка существования Эйлерова пути

Для того, чтобы граф содержал Эйлеров цикл, необходимо и достаточно, чтобы: все вершины имели четную степень и все компоненты связности кроме, может быть одной, не содержали ребер.

Для того, чтобы граф содержал Эйлеров цикл, необходимо и достаточно, чтобы количество вершин с нечетной степенью было меньше или равно двум и все компоненты связности кроме одной не содержали бы ребер.

```
CHECK_EULER(circle_exist)
     circle_exist   =   false
     odd_degree = 0
     start = 0
     for v in graph.V
          if odd(v.degree)
               inc(odd_degree)
               start = v
     if odd_degree > 2
          return 0

     dsu = DSU(graph.V.size)
     for vu in graph.E
          dsu.join(v, u)

     fill(distinct, false)
     for i = 1 .. graph.V.size - 1 v
          = dsu.find(i)
          u = dsu.find(i + 1)
          if v != u
               distinct[v] = true
               distinct[u] = true

     count = 0
     for v in graph.V
          if distinct[v] and v.degree > 0
               inc(count)
          if count > 1
               return 0

     if start != 0 circle_exist
          = true
returnstart
```

Рис. 4. Листингпсевдокодапроверки Эйлерова Пути

3.2. Алгоритм Флери

Алгоритм находит Эйлеров цикл как в ориентированном, так и в неориентированном графе.

Перед запуском алгоритма необходимо проверить граф на эйлеровость. Чтобы построить Эйлеров путь, нужно запустить алгоритм из вершины с нечетной степенью, если таковая имеется.

```
EULERAN_TOUR_FLERI()
     result = []
     start = CHECK_EULER(circle_exist) if
     start == 0
          return result v
     = start
     result.add(v) while
     true
          if v.degree == 0
               break
          flag = true
          for u in v.neighbours
               if v.deg == 1 or not IS_BRIDGE(v, u)
                    flag = false
                    graph.remove(v, u) v
                    = u result.add(v)
                    break
          if flag
               u = v.neighbours[0]
               graph.remove(v, u) v =
               u
               result.add(v)
     return result
```

Рис. 4. Листингпсевдокодаалгоритма Флери

```
IS_BRIDGE(from, to)
     fill(used, false)
     queue.push(from)
     used[from] = true while
     not queue.empty
          v = queue.pop()
          for u in v.neighbours
               if v == from and u == to
                    continue
               if not used[u] if
                    u == to
                         return false
                    used[u] = true
                    queue.push(u)
     returntrue
```

Рис. 5. Проверка Ребро — Мост

3.3. Эффективный алгоритм построения Эйлерова пути

Алгоритм находит Эйлеров цикл как в ориентированном, так и в неориентированном графе. Перед запуском алгоритма необходимо проверить граф на эйлеровость. Чтобы построить Эйлеров путь, нужно запустить алгоритм из вершины с нечетной степенью, если таковая имеется.

```
EULERAN_TOUR_EFFECTIVE()
     result = []
     start = CHECK_EULER(circle_exist) if
     start == 0
          return result
     stack.push(start) while
     not stack.empty
          v = stack.top()
          for u in v.neighbours
               stack.push(u)
               graph.remove(v, u)
               break
          if v == stack.top()
               stack.pop()
               result.add(v)
     returnresult
```

Рис. 6. Листинг псевдокода алгоритма оптимального построения Эйлерова пути

3.4. Интерфейс класса Graph

Публичные методы класса:

— *intreviewEulerWay();*— метод класса, позволяющий произвести проверку существования Эйлерова пути

— *std::vector<int>buildFleriPath();*— методкласса, находящий Эйлеров цикл как в ориентированном, так и в неориентированном графе.

4. Лабораторная работа №4

Требуется реализовать построение максимального числа паросочетаний в двудольном графе с помощью алгоритма Куна. Помимо этого, необходимо реализовать проверку графа на двудольность.

4.1. Проверка графа на двудольность

Граф является двудольным тогда и только тогда, когда он содержит более одной вершины и все его циклы имеют четную длину.

```
CHECK_BIPART(marks)
     if graph.V.size < 2
          return false
     for v in graph.V
          if marks[v] != 0 and marks[v] != 1
               marks[v] = 0
               queue.push(v)
               while not queue.empty u
                    = queue.pop()
                    for w in u.neighbours
                         if marks[w] != 0 and marks[w] != 1
                              marks[w] = marks[u] == 0 ? 1 : 0
                              queue.push(w)
                    else if marks[u] == marks[w]
                         return false
     returntrue
```

Рис. 7. Листинг псевдокода проверки графа на двудольность

4.2. Алгоритм Куна

Алгоритм основан на следующих принципах:если из всех элементов произвольной строки или столбца вычесть одно и то же число y, общая стоимость уменьшится на y, а оптимальное решение не изменится;если есть решение нулевой стоимости, оно оптимально.

Алгоритм ищет значения, которые надо вычесть из всех элементов каждой строки и каждого столбца (разные для разных строк и столбцов), такие, что все элементы матрицы останутся неотрицательными, но появится нулевое решение.

```
KUHN()
     result = [] fill(bipart, -1)
     fill(marks, -1) fill(used,
     false) for v in graph.V
          if marks[v] == 1 continue
          for u in v.neighbours if bipart[u] !=
               -1
                    bipart[u] = v used[v] =
                    true break

     if CHECK_BIPART(marks) for v in
          graph.V
               if used[v] or marks[v] == 1 continue
               fill(used_dfs, false) DFS(v, used_dfs,
               bipart)
          for v in graph.V
               if marks[v] == 1 and bipart[v] != -1
                    result.add((bipart[v], v))
     returnresult
```

Рис. 8. Листинг псевдокода алгоритма Куна

4.3. Интерфейс класса Graph

Публичныеметоды:

— *intreviewPart(std::vector<char>);*—методкласса, реализующий построение максимального числа паросочетаний в двудольном графе;

— *std::vector<std::pair<int, int>>landEverPart();* — метод, собственно, получения максимального паросочетаний в двудольном графе

5. Лабораторная работа №5

Требуется реализовать следующие алгоритмы вычисления максимального потока — алгоритм Форда—Фалкерсона и алгоритм Диница.

5.1. Алгоритм Форда-Фалкерсона

Алгоритм начинает свою работу с нулевого потока и на каждой своей итерации увеличивает поток в сети. На каждом шаге находится увеличивающая величину потока цепь. Поток увеличивается вдоль дуг этой цепи, пока она не станет насыщенной.

Псевдокод вспомогательного поиска в глубину:

```
DFS(source, sink, flow, used, edges, head)
     if source == sink
          return flow
     used[source] = true
     for i = head[source]; i != -1; i = edges[i].next
          if not used[edges[i].to] and edges[i].flow < edges[i].capacity
               new_flow = min(flow, edges[i].capacity – edges[i].flow) pushed
               = DFS(edges[i].to, sink, new_flow, used, edges, head) if
               pushed != 0
                    edges[i].flow += pushed
                    edges[i ^ 1].flow -= pushed
                    return pushed
     return 0
```

Рис. 9. Листинг псевдокода вспомогательного поиска в глубину

```
FORD_FULKERSON(source, sink) result
     = Graph(graph.V.size)
     fill(head, -1)
     edges = []
     for vu in graph.E
          edges.add(v, u, head[v], vu.weight, 0)
          edges.add(u, v, head[u], 0, 0)
          head[v] = graph.E.size – 2
          head[u] = graph.E.size – 1
     while true
          fill(used, false)
          if DFS(source, sink, INF, used, edges, head) == 0
               break
     for i = 0; i < graph.E.size; i += 2 result.add(edges[i].from,
          edges[i].to, edges[i].flow)
     returnresult
```

Рис. 10. Листинг псевдокода алгоритма Форда—Фалкерсона

5.2. Алгоритм Диница

Основная идея метода реализации:алгоритм состоит из фаз, на которых поток увеличивается сразу вдоль всех кратчайших цепей определенной длины. Для этого на *i-ой* фазе строится вспомогательная бесконтурная сеть. Эта сеть содержит все увеличивающие цепи, длина которых не превышает k_i, где k_i – длина кратчайшего пути из s в t. Величину k_i называют длиной вспомогательной сети.

Псевдокод вспомогательного поиска в глубину:

```
DFS(source, sink, flow, layer, edge_capacity, edge_flow)
     if source == sink or flow == 0
          return flow
     for v in source.neighbours
          if layer[v] != layer[source] + 1
               continue
          new_flow = min(flow, edge_capacity[(source, v)] –
edge_flow[(source, v)]
          pushed = DFS(v, sink, new_flow, layer, edge_capacity, edge_flow)
          if pushed != 0
               graph.E[(source, to)].weight -= pushed
               graph.E[(to, source)].weight += pushed
               edge_flow[(source, to)] += pushed
               edge_flow[(to, source)] -= pushed return
               pushed
     return 0
```

Рис. 11. Листинг псевдокода вспомогательного алгоритма поиска в глубину

```
DINITZ(source, sink)
     result = Graph(graph.V.size) g
     = Graph(graph.V.size)
     edge_capacity = []
     edge_flow = [] for
     vu in graph.E
          g.add(v, u, vu.weight) g.add(u,
          v, 0) edge_capacity[vu] =
          vu.weight edge_capacity[uv] =
          vu.weight edge_flow[vu] = 0
          edge_flow[uv] = vu.weight
     while true fill(layer, -
          1) layer[source] = 0
          queue.push(source)
          while not queue.empty and layer[sink] == -1 v
               = queue.pop()
               for u in v.neighbours
                    if layer[u] == -1 and edge_flow[vu] < edge_capacity[vu]
                         queue.push(u)
                         layer[u] = layer[v] + 1
          if g.DFS(source, sink, INF, layer, edge_capacity, edge_flow) == 0
               break
     for v in g.V
          for u in v.neighbours
               if v >= u
                    result.add(u, v + 1, vu.weight)
     returnresult
```

5.3. Интерфейс класса Graph

Публичныеметоды:

— *GraphspringFordFulkerson(intsourse, intsink);*—методкласса, позволяющийпроизвести вспомогательный поиск в глубину;

— *GraphspringDinitz(intsourse, intsink).*— методкласса, позволяющий увеличивать поток вдоль всех кратчайших цепей;

6. Заключение

В ходе выполнения лабораторных работ были изучены вышеперечисленные алгоритмы, а именно: алгоритмы Прима, Краскала, Борувки, Флери, Куна, Диница и Форда-Фалкерсона. Все алгоритмы были написаны на языке программирования C++.Помимо этого, были реализованы вспомогательные методы и структуры данных для этих алгоритмов. Например, различные модификации поисков в глубину и в ширину и система непересекающихся множеств.