ИКН НИТУ МИСИС

Комбинаторика и теория графов

Представления графов в компьютере

Исполнитель:

Лосев В.Л. БИВТ-23-18

(https://github.com/SomethingWF/Grafs)

Москва 2024 год

Определения

С точки зрения дискретной математики, а именно в теории графов, граф – это структура, состоящая из набора объектов, некоторые пары которых в некотором смысле «связаны». Объекты представлены абстракциями, называемыми вершинами (также называемыми узлами или точками), а каждая связанная пара вершин называется ребром (также называемым связью или линией). Как правило, граф изображается в диаграммной форме в виде набора точек или кругов для вершин, соединённых линиями или кривыми для рёбер.

Если же описывать графы в терминах множеств, то графом G называется любая пара (V, U), где $V = \{v_1, v_2, \ldots\}$ – множество элементов любой природы, а $E = \{e_1, e_2, \ldots\}$ – семейство пар из элементов V, причем допускаются пары вида (v_i, v_j) и одинаковые пары вида (v_i, v_i).

Структуры данных, хранящие графы, поддерживает следующие операции:

1. Удаление вершины – преобразование графа G в граф $G \setminus v$ содержащий все вершины графа G, за исключением v, и все ребра графа G, не инцидентные v.
2. Удаление ребра – преобразование графа G в граф $G \setminus e$, содержащий все вершины и все ребра графа G за исключением e.
3. Добавление (новой) вершины – преобразование исходного графа $G = \langle V, E \rangle$ к виду $G = \langle V \cup \{v\}, E \rangle$.
4. Добавление ребра – преобразование исходного графа $G = \langle V, E \rangle$ к виду $G = \langle V, E \cup \{e\} \rangle$.
5. Стягивание графа $G = \langle V, E \rangle$ по множеству вершин $J \subseteq V$ подразумевает удаление вершин J, добавление новой вершины w и соединение с ней вершин из G, смежных вершинам в J, с помощью ребер.

Конкретные структуры данных, реализуемые мною в данной работе:

1. Матрица смежности – это квадратная матрица, используемая для представления графа. Элементы матрицы указывают, являются ли пары вершин смежными или нет в графе.
2. Список смежности – это набор неупорядоченных списков, используемых для представления графа. Каждый список соответствует вершине графа и содержит вершины, с которыми она связана.

Перечень инструментов

Языком программирования, использовавшимся для реализации структур данных, хранящих графы, является C++. В частности, стандартная библиотека шаблонов(STL) из которой были взяты уже реализованные контейнеры vector и forward_list.

Описание реализации

Для реализации матрицы смежности было решено использовать массив типа «vector», а именно – матрица является вектором, содержащим вектора одинаковой длины, и индексы ячеек векторов отождествляются с номером вершины. Сами же ячейки векторов хранят целочисленный тип данных «int» и отождествляют наличие связи между вершинами, пересечением столбцов и строк которых является данная ячейка. Соответственно, значение ячейки «1», трактуется как наличие ребра между вершинами, «0» – как отсутствие. Именно матрица является первым полем реализуемой структуры данных, а вторым – поле типа «int», указывающее количество вершин графа.

```
class Graph {
private:
  int v_ = 0;
  std::vector<std::vector<int>> adjMatrix;
```

Рис 1 – Поля матрицы смежности

Операция добавления вершины заключается в добавлении новых строки и столбца в матрицу и последующим увеличением значения количества вершин на единицу.

```
void addVertice() {
  if (v_ >= adjMatrix.size()) {
    for (int i = 0; i < v_; ++i) {
      adjMatrix[i].push_back(0);
    }
    adjMatrix.emplace_back(v_ + 1, 0);
  }
  else {
    for (int i = 0; i < v_ + 1; ++i) {
      adjMatrix[i][v_] = 0;
      adjMatrix[v_][i] = 0;
    }
  }
  ++v_;
}
```

Рис 2 – Реализация операции добавления вершины

Операция добавления ребра представляет собой редактирование ячейки матрицы для обозначения связи между вершинами.

```
void addEdge(int u, int v) {
  if (u != v) {
    adjMatrix[u][v] = 1;
    adjMatrix[v][u] = 1;
  }
  else std::cout << "Trying to create a loop";
}
```

Рис 3 – Реализация операции добавления ребра

Операция удаления вершины требует смещения всех строк и столбцов на место удаляемых, отождествленных с указанной вершиной.

```
void removeVertice(int x) {
  if (x < v_)
  {
    while (x < v_ - 1) {
      for (int i = 0; i < v_; ++i) adjMatrix[i][x] = adjMatrix[i][x + 1];
      for (int i = 0; i < v_; ++i) adjMatrix[x][i] = adjMatrix[x + 1][i];
      ++x;
    }
    --v_;
  }
}
```

Рис 4 – Реализация операции удаления вершины

Операция удаления ребра идентична операции добавления ребра за тем лишь исключением, что в данном случае в ячейку матрицы заносится значение «0».

```
void removeEdge(int u, int v) {
  adjMatrix[u][v] = 0;
  adjMatrix[v][u] = 0;
}
```

Рис 5 – Реализация операции удаления ребра

Операция стягивания графа проводится следующим образом: одной из вершин присваивают ребра, смежные исключительно с другой вершиной, таким образом избегая создания кратных ребер. Вершина же, у которой были взяты ребра, удаляется по вышеописанному алгоритму.

```
void merge(int x, int y) {
  if (x != y)
  {
    adjMatrix[x][y] = 0;
    adjMatrix[y][x] = 0;
    for (int i = 0; i < v_; ++i) {
      int a = (adjMatrix[x][i] != adjMatrix[y][i]);
      int b = (adjMatrix[x][i] + adjMatrix[y][i]) / 2;
      adjMatrix[x][i] = a + b;
      adjMatrix[i][x] = adjMatrix[x][i];
    }
    removeVertice(y);
  }
  else std::cout << "Trying to merge itself";
}
```

Рис 6 – Реализация операции стягивания графа

Для реализации листа смежности также было решено использовать массив типа «vector», который представляет собой вектор, содержащий списки смежных вершин для каждой вершины графа. Каждый элемент вектора соответствует вершине, а сам список хранит целочисленные значения типа «int», которые представляют собой номера смежных вершин. Таким образом, наличие ребра между вершинами обозначается присутствием соответствующего номера одной вершины в списке другой. Первым полем реализуемой структуры данных является вектор, содержащий списки смежных вершин, а вторым – поле типа «int», указывающее количество вершин графа.

```
private:
    int v_;
    std::vector<std::forward_list<int>> adjList;
};
```

Рис 7 – Поля списка смежности

Операция добавления вершины заключается в добавлении нового списка в вектор и последующем увеличении значения количества вершин на единицу. При этом новый список изначально будет пустым, так как новая вершина не имеет смежных ребер.

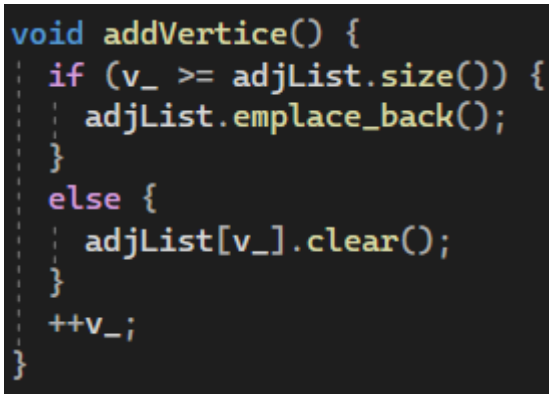

Рис 8 - Реализация операции добавления вершины

Операция добавления ребра представляет собой добавление номера смежной вершины в список соответствующей вершины, а также добавление номера текущей вершины в список смежной вершины, если граф неориентированный.

```
void addEdge(int x, int y) {
  if (x < v_ && y < v_) {
    if (x != y)
    {
      for (auto bgn = adjList[x].begin(); bgn != adjList[x].end(); ++bgn) {
        if (*bgn == y) {
          std::cout << "Trying to create already existing edge" << std::endl;
          return;
        }
      }
      adjList[x].push_front(y);
      adjList[y].push_front(x);
    }
    else std::cout << "Trying to create a loop" << std::endl;
  }
  else std::cout << "Some of vertices does not exist" << std::endl;
}
```

Рис 9 – Реализация добавления ребра

Операция удаления вершины требует удаления списка смежных вершин из вектора и удаления всех упоминаний о данной вершине из списков смежных вершин остальных вершин. Также требуется смещение ячеек вектора, для удаления пробелов с последующим обновлением всех списков, так как при смещении вершины с порядковым номером большим, чем у удаленной, уменьшаются на единицу.

```
void removeVertice(int x) {
  if (x < v_) {
    for (int i = 0; i < v_; ++i) {
      auto bgn_curr = adjList[i].begin();
      auto bgn_prev = adjList[i].begin();
      if ((*bgn_curr) == x) {
        adjList[i].pop_front();
      }
      else {
        ++bgn_curr;
        for (bgn_curr; bgn_curr != adjList[i].end(); ++bgn_curr) {
          if ((*bgn_curr) == x) {
            adjList[i].erase_after(bgn_prev);
            break;
          }
          ++bgn_prev;
        }
      }
    }
```

```
    for (int i = 0; i < v_; ++i) {
      auto bgn = adjList[i].begin();
      for (bgn; bgn != adjList[i].end(); ++bgn) {
        *bgn -= (*bgn > x);
      }
    }
    for (int i = x; i + 1 < v_; ++i) {
      adjList[i] = adjList[i + 1];
    }
    --v_;
  }
  else std::cout << "Trying to remove not existing vertice" << std::endl;
}
```

Рис 10 – Реализация операции удаления вершины

Операция удаления ребра заключается в удалении номера смежной вершины из списка соответствующей вершины, а также удаления номера текущей вершины из списка смежной вершины, если граф неориентированный.

```
void removeEdge(int x, int y) {
    if (x < v_ && y < v_) {
        auto bgn_prev = adjList[x].begin();
        auto bgn_curr = adjList[x].begin();
        if ((*bgn_curr) == y) {
            adjList[x].pop_front();
        }
        else {
            ++bgn_curr;
            for (bgn_curr; bgn_curr != adjList[x].end(); ++bgn_curr) {
                if ((*bgn_curr) == y) {
                    adjList[x].erase_after(bgn_prev);
                    break;
                }
                ++bgn_prev;
            }
        }
        bgn_prev = adjList[y].begin();
        bgn_curr = adjList[y].begin();
        if ((*bgn_curr) == x) {
            adjList[y].pop_front();
        }
        else {
            ++bgn_curr;
            for (bgn_curr; bgn_curr != adjList[y].end(); ++bgn_curr) {
                if ((*bgn_curr) == x) {
                    adjList[y].erase_after(bgn_prev);
                    break;
                }
                ++bgn_prev;
            }
        }
    }
    else std::cout << "Trying to erase not existing edge" << std::endl;
}
```

Рис 11 – Реализация операции ударения ребра

Операция стягивания графа осуществляется путем переноса всех смежных ребер одной вершины к другой, избегая создания кратных ребер. Вершина, у которой были взяты ребра, удаляется по вышеописанному алгоритму.

```
void merge(int x, int y) {
  for (int i : adjList[x]) {
    bool isExist = false;
    for (int j : adjList[y]) {
      if (i == j) {
        isExist = true;
        break;
      }
    }
    if (isExist == false && i != y) {
      adjList[y].push_front(i);
    }
    isExist = false;
    for (int j : adjList[i]) {
      if (j == y) isExist = true;
    }
    if (isExist == false && i != y) adjList[i].push_front(y);
  }
  removeVertice(x);
}
```

Рис 12 – Реализация операции стягивания графа

Анализ временной сложности реализованных операций

Рассмотрим асимптотическую сложность операций, реализованных в матрице смежности:

1. Добавление вершины. Так как для добавления вершины в конец каждого вектора необходимо добавить новый элемент, и сложность данной операции составляет O(1), то в худшем случае общая сложность данной операции достигает O(v) (Здесь и далее, v – все вершины, содержащиеся в графе).
2. Добавление ребра. Известно, что доступ к элементу массива осуществляется за константное время O(1). В рамках данной операции обращение к элементу массива осуществляется дважды, но так как константы при подсчете сложности не учитываются, то итоговая сложность операции O(1).
3. Удаление вершины. В данной операции применяются вложенные циклы, причем два цикла вложены в один. Как упоминалось ранее, константы при подсчете временной сложности не учитываются, поэтому итоговая сложность операции составляет $O(v^2)$.

4. Удаление ребра. Реализация данной операции схожа с добавлением ребра и по тем же причинам обладает итоговой асимптотической сложностью O(1).
5. Стягивание графа. При стягивании помимо использования цикла, составляющего сложность O(v), используется вызов операции удаления вершины, чья сложность составляет $O(v^2)$. При оценке асимптотической сложности составляющие меньшей степени «поглощаются» составляющими большей степени, оттого итоговая сложность операции $O(v^2)$.

Сложность данной структуры по занимаемой памяти составляет $O(v^2)$.

Рассмотрим асимптотическую сложность операций, реализованных в списке смежности:

1. Добавление вершины. В зависимости от обстоятельств операция просто добавляет новый элемент в вектор, либо очищает следующий за крайним элементом вектора список. Заявлено, что функция clear() обладает линейной сложностью, значит в худшем случае сложность операции O(v).
2. Удаление вершины. Данная операция содержит три цикла, два из которых также содержат по вложенному циклу. Как говорилось ранее более сложные алгоритмы «поглощают» простые, оттого сложность данной операции составляет $O(v^2)$.
3. Добавление ребра. Реализация этой операции включает в себя один цикл, на каждой итерации которого дважды происходит обращение к элементу вектора и добавление нового элемента в начало списка. Итоговая сложность операции O(v).
4. Удаление ребра. В этой операции содержатся два цикла, следовательно, итоговая сложность производимой операции O(v).
5. Стягивание графа. Данная операция имеет три цикла, двое из которых вложены в другой. Также содержится вызов операции удаления вершины со сложностью $O(v^2)$. Итоговая сложность операции составляет $O(v^2)$.

Сложность данной структуры по памяти O(v + e) (где e – сумма степеней всех вершин).

Ссылка на реализацию

https://github.com/SomethingWF/Grafs

Список источников

https://en.cppreference.com/w/

https://habr.com/ru/companies/otus/articles/675730/

http://blog.kislenko.net/show.php?id=3034