Санкт-Петербургский политехнический университет Петра великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая проект

Дисциплина: Объектно-ориентированное программирование

Тема: Обход графа в глубину и ширину. Топологическая сортировка.

Разработал:
студент гр. 3331506/20102 Васильев Г. В.

Преподаватель Ананьевский М.С.

Санкт-Петербург
2025

Введение

Постановка задачи

Топологическая сортировка — это такое линейное упорядочивание вершин ориентированного графа, при котором для любого ребра *UV* вершина *U* всегда располагается перед *V* в итоговом порядке. Однако выполнить такую сортировку возможно только для ациклических графов (DAG), поскольку наличие циклов делает задачу неразрешимой (невозможно однозначно определить порядок вершин в цикле). [2]

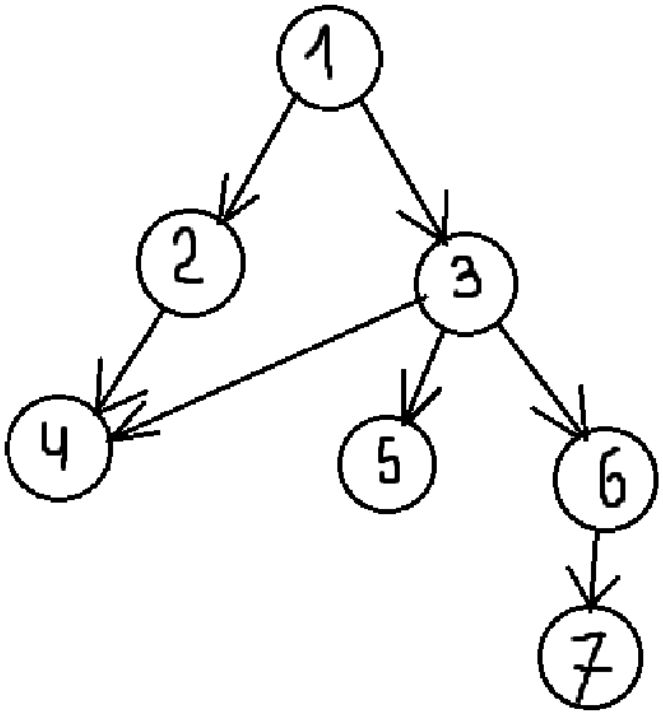


Рисунок 1 – Бесконтурный ориентированный граф (не отсортирован)

Требуется найти нумерацию вершин, при которой все рёбра направлены от вершин с меньшими номерами к вершинам с большими. Если граф содержит циклы, топологическая сортировка для него не существует. [2]

Один из самых простых и быстрых способов выполнить топологическую сортировку — использовать обход в глубину (DFS). [5]

Обход в глубину

DFS можно кратко описать одним предложением: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них. Кроме этого, для топологической сортировки также необходимо определить, есть ли в данном графе циклы. Для этого еще вводится такое понятие как цвет вершины.

Белая вершина – мы еще в ней не были.

Серая вершина – в обработке. То есть идёт процедура обхода в глубину.

Черная вершина – вершина пройдена. [2][5]

Алгоритм программы:

Берем одну вершину

От неё рассматриваем каждое из ребер. Если вершина чёрная – идём к следующему ребру, если серая – обнаружен цикл. Если белая – идём вглубь.

Если нет рёбер из нашей вершины или же они ведут к чёрным, то наша вершина считается конечной и соответственно окрашивается в чёрный.

Обход в ширину

BFS (Breadth-First Search) — это алгоритм обхода графа, который исследует все соседние вершины на текущем уровне, прежде чем переходить к вершинам следующего уровня. [5]

Основной принцип:

Для каждой не посещённой вершины необходимо последовательно обработать всех её соседей, используя очередь.

Для контроля посещения вершин используется цветовая маркировка

- "белый" – не посещена
- "серый" – в обработке
- "чёрный" – завершена

Алгоритм программы:

Берем вершину

От нее рассматриваем каждое ребро. После чего если вершина белая исследуем смежные с ней вершины, если серая – обнаружен цикл.

Если эти вершины не являются конечными или целевыми, движемся по ребрам, которые исходят из вершин, смежных с начальной.

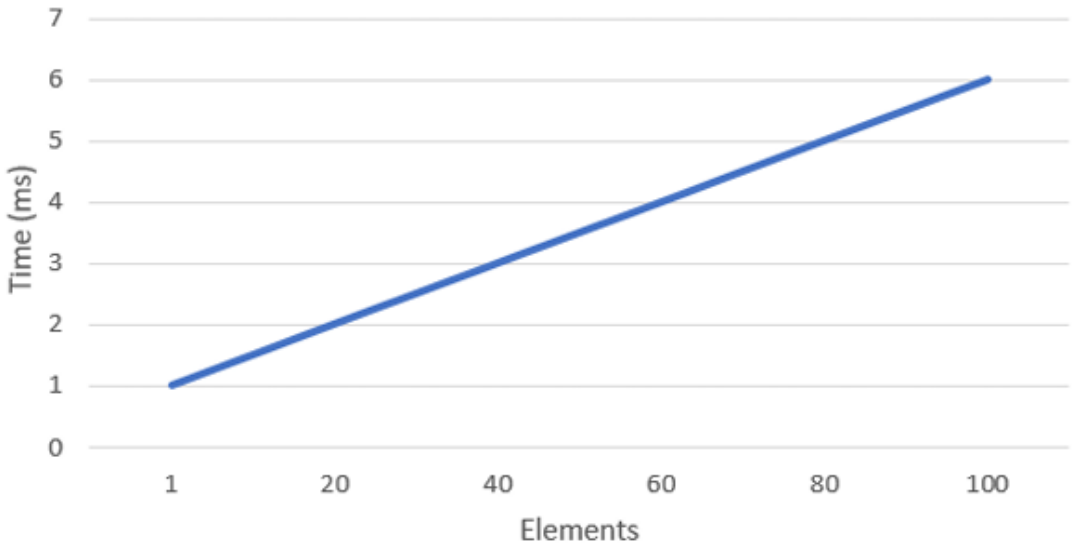
По очереди исследуем вершины этого "уровня". Если эти вершины не являются конечными или целевыми, движемся дальше на следующий уровень по ребрам, которые исходят из этих вершин.

Алгоритм повторяется, пока не будут исследованы все вершины и достигнута конечная целевая вершина.

Алгоритм BFS широко применяется для определения кратчайшего пути между двумя вершинами в графе. Вычислительная сложность данного алгоритма оценивается как O(V + E), где V представляет общее количество вершин, а E — общее число рёбер в графе. Что касается пространственной сложности, она составляет O(V), поскольку в процессе работы алгоритма

используется очередь, которая в худшем случае может содержать все вершины графа. [5]

График зависимости времени выполнения сортировки массива, от его размера, для DFS и BFS будет примерно одинаковым.



Топологическая сортировка с помощью обхода в глубину

Топологическая сортировка реализуется через модифицированный обход в глубину (DFS), где посещённые вершины фиксируются в стеке. Итоговый порядок определяется обратным извлечением вершин из стека. [2][4]

Пошаговый алгоритм:

1. **Рекурсивный обход графа:**
   - Для каждой непосещённой вершины запускаем DFS.
   - При завершении обработки вершины (когда все её рёбра пройдены) помещаем её в стек.
   - Контролируем отсутствие циклов (если при обходе встречается "серая" вершина – граф циклический).
2. **Формирование результата:**
   - После полного обхода всех вершин извлекаем элементы из стека.
   - Порядок извлечения (вершина на вершине стека, след. последняя позиция в результате) даёт топологическую сортировку.

Реализация алгоритма

Реализация алгоритма топологической сортировки выполнена на языке C++, что обусловлено требованиями к производительности и возможностью интеграции с другими алгоритмами работы с графами. Архитектура решения включает два основных класса, обеспечивающих гибкость и расширяемость кода. [3]

Структура реализации:

1. Класс Node:

- Содержит ключевое поле value для хранения значения вершины
- Включает вектор edge для хранения ссылок на смежные узлы
- Имеет публичное свойство color, необходимое для визуализации состояния вершин при топологической сортировке

2. Класс Graph:

- Хранит массив объектов Node и информацию об их количестве (amount)
- Реализует ключевые методы:

    dfs - для обхода графа в глубину

    topological_sort - основной алгоритм сортировки

- Содержит тестовый конструктор, генерирующий ациклический ориентированный граф заданного размера со случайными ребрами (с гарантией отсутствия циклов)

На выходе алгоритм выдает только лишь топологически отсортированную последовательность вершин, так как на практике в большинстве случаев этого достаточно – необходимо лишь знать правильную последовательность действий. [1]

Основная часть кода:

```
#include <vector>
#include <iostream>
#include <cstring>
#include <cstdlib>
#include <ctime>

#define COLOR_OF_NODE_GREY 1
#define COLOR_OF_NODE_BLACK 2

typedef std::vector<unsigned int> uiVector;

class node
{
private:
   int value;
   uiVector edges;
public:
   int color;

   node()
   {
      this->value = 0;
      this->color = 0;
   }

   node(int val)
   {
      this->value = val;
      this->color = 0;
   }

   void setValue(int val)
   {
      this->value = val;
   }

   unsigned int edge(unsigned int num)
   {
      return this->edges[num];
   }

   unsigned int edgeSize()
   {
```

```
        return this->edges.size();
    }

    int getValueOfNode()
    {
        return this->value;
    }

    void addEdge(unsigned int link)
    {
        edges.push_back(link);
    }

    ~node()
    {
        edges.clear();
    }
};

class graph
{
private:
    unsigned int amount;
    node * nodes;
    bool dfs(unsigned int curNode, uiVector &stack);
public:
    graph(unsigned int size);
    void print();
    bool topological_sort(uiVector &result);
    ~graph()
    {
        delete[] nodes;
    }
};

static unsigned int random_Range(unsigned int range)
{
    return rand() % range;
}

graph::graph(unsigned int size)
{
    std::srand(std::time(0));
    this->nodes = new node[size];
```

```
        this->amount = size;

        unsigned int ranks[size][size + 1];
        memset(ranks, 0, size * (size + 1) * sizeof(int));

        for (unsigned int i = 0; i < size; i++)
        {
           nodes[i].setValue(i);
           unsigned int currentRank = random_Range(size);
           ranks[currentRank][0] += 1;
           ranks[currentRank][ranks[currentRank][0]] = i;
        }

        for(unsigned int rank = 0; rank < size - 1; rank++)
        {
           for (unsigned int curNode = 1; curNode < ranks[rank][0] + 1;
curNode++)
           {
              for (unsigned int rankLow = rank + 1; rankLow < size; rankLow++)
              {
                 for (unsigned int link = 1; link < ranks[rankLow][0] + 1; link++)
                 {
                    if (random_Range(100) > 50)
                    {

nodes[ranks[rank][curNode]].addEdge(ranks[rankLow][link]);
                    }
                 }
              }
           }
        }

        void graph::print()
        {
           for (unsigned int i = 0; i < this->amount; i++)
           {
              int value = this->nodes[i].getValueOfNode();
              std::cout << "Node " << i << ":  " << value << ";    edges:";
              unsigned int edgeCount = this->nodes[i].edgeSize();
              for (unsigned int j = 0; j < edgeCount; j++)
              {
                 unsigned int link = this->nodes[i].edge(j);
                 std::cout << "(" << i << "," << link << ")  ";
```

```
        }
        std::cout << std::endl;
    }
}

bool graph::dfs(unsigned int curNode, uiVector &stack)
{
    if (this->nodes[curNode].color == COLOR_OF_NODE_GREY)
    {
        return true;
    }
    if (this->nodes[curNode].color == COLOR_OF_NODE_BLACK)
    {
        return false;
    }
    this->nodes[curNode].color = COLOR_OF_NODE_GREY;

    unsigned int edgeCount = this->nodes[curNode].edgeSize();
    for (unsigned int i = 0; i < edgeCount; i++)
    {
        unsigned int link = this->nodes[curNode].edge(i);
        if (this->dfs(link, stack))
        {
            return true;
        }
    }
    stack.push_back(curNode);
    this->nodes[curNode].color = COLOR_OF_NODE_BLACK;
    return false;
}

bool graph::topological_sort(uiVector &result)
{
    uiVector stack;
    for (unsigned int i = 0; i < this->amount; i++)
    {
        if (dfs(i, stack))
        {
            return false;
        }
    }

    result.resize(this->amount);
    for (unsigned int i = 0; i < this->amount; i++)
```

```
    {
        unsigned int index = stack.back();
        result[index] = i;
        stack.pop_back();
    }
    return true;
}

int main()
{
    graph g(5);
    g.print();

    uiVector sorted;
    if (g.topological_sort(sorted))
    {
        std::cout << "Topological order: ";
        for (unsigned int node : sorted)
        {
            std::cout << node << " ";
        }
        std::cout << std::endl;
    }
    else
    {
        std::cout << "Graph contains a cycle!" << std::endl;
    }

    return 0;
}
```

Сложность алгоритма

Сложность такого алгоритма соответствует сложности алгоритма поиска в глубину, то есть O(m+n), где n – число вершин, m – число ребер. Это доказывается тем, что при проходе в глубину алгоритм лишь единожды проходит через единственную вершину, помечая ее как посещенную. [2][4]

В большинстве случаев количество ребер в графе гораздо больше количества вершин. Поэтому определяющим фактором в сложности алгоритма является количество ребер графа.

Область применения

Топологическая сортировка применяется в самых разных ситуациях, например при создании параллельных алгоритмов, когда по некоторому описанию алгоритма нужно составить граф зависимостей его операций и, отсортировав его топологически, определить, какие из операций являются независимыми и могут выполняться параллельно (одновременно). [1]

Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов.

Заключение

В рамках курсовой работы был изучен алгоритм топологической сортировки, который был успешно реализован на языке C++. Проведен детальный анализ вычислительной сложности алгоритма и выполнены численные эксперименты для проверки его эффективности. Также исследованы основные области применения данного метода и его ключевые особенности.

Список литературы

1. *Рафгарден Тим.* Совершенный алгоритм. Графовые алгоритмы и структуры данных. - СПб.: Питер, 2019. - 256 с.: ил. - (Серия «Библиотека программиста») — ISBN 978-5-4461-1272-2

2. *Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К.* ***Глава 22.4. Топологическая сортировка*** *// Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. И. В. Красикова. — 2-е изд. — М.: Вильямс, 2005. — С. 632-635. — ISBN 5-8459-0857-4.*

3. *Хайнеман, Д. Алгоритмы. Справочник. С примерами на C, C++, Java и Python /Д. Хайнеман, Г. Поллис, С. Селков. – Вильямс, 2017.*

4. *Левитин А. В. Глава 5. Метод уменьшения размера задачи: Топологическая сортировка // Алгоритмы. Введение в разработку и анализ — М.: Вильямс, 2006. — С. 220–224. — 576 с. — ISBN 978-5-8459-0987-9*

5. *Седжвик Р. Алгоритмы на C++. Том 1: Основы. — М.: Вильямс, 2002. — 592 с. (Графы, DFS, топологическая сортировка)*