Санкт-Петербургский политехнический университет Петра Великого
Институт металлургии, машиностроения и транспорта
Кафедра мехатроники и роботостроения при ЦНИИ РТК

Курсовая работа

Дисциплина: Программирование на языках высшего уровня

Тема: Топологическая сортировка

Выполнил
студент гр. 33335/2 Сидоренко В. А.
Преподаватель Ананьевский М. С.

«____» _____ 2018 г.

Санкт-Петербург
2018 г.

Введение

В программировании часто возникает задача упорядочивания вершин бесконтурного орграфа – направленного графа, в котором отсутствуют направленные циклы, но могут быть параллельные пути, выходящие из одного узла и разными путями приходящие в конечный узел. Такие графы широко используются в компиляторах, машинном обучении, статистике и в других задачах, где необходимо корректно определить последовательность зависимых друг от друга действий. Пример такого графа представлен на рисунке 1.

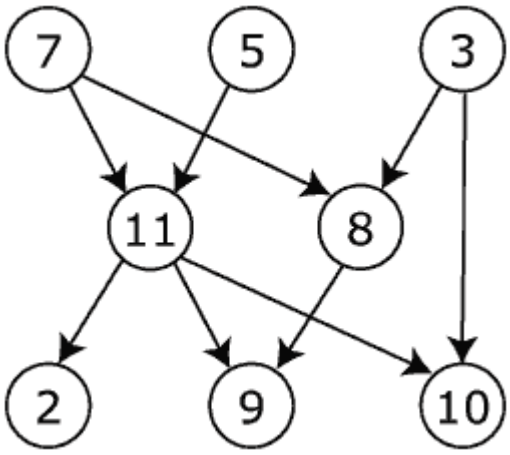

Рисунок 1 – Бесконтурный ориентированный граф (не отсортирован)

Топологическая сортировка – это один из основных алгоритмов на графах, который применяется для решения множества более сложных задач. Топологическая сортировка применяется в самых разных ситуациях, например при распараллеливании 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). Примером использования топологической сортировки может служить создание карты сайта, где имеет место древовидная система разделов.

Задача топологической сортировки графа состоит в следующем: указать такой линейный порядок на его вершинах, чтобы любое ребро вело от вершины с меньшим номером к вершине с большим номером. Очевидно, что если в графе есть циклы, то такого порядка не существует. Наиболее простая и быстрая реализация этого алгоритма — с помощью обхода в глубину (англ. *Depth-first search*, сокращенно *DFS*).

Обход в глубину

В программировании очень часто возникает задача обхода графа, то есть просмотреть все ребра и вершины с целью отыскания некоторых удовлетворяющих определенным условиям. Одним из таких методов является обход в глубину. Его 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:

- Если вершина белая – мы еще в ней не были.
- Вершина серая – вершина проходится в текущей процедуре поиска в глубину (если мы упираемся в серую вершину, то это значит, что граф циклический).
- Черная вершина – вершина пройдена.

Словесное описание такого алгоритма выглядит следующим образом:

1. Окрашиваем текущую вершину в серый цвет
2. Рассматриваем каждое ребро, ведущее из текущей вершины. Если следующая вершина белая, вызываем `dfs()` для этой вершины. Если вершина черная – переходим к следующему ребру. Если вершина серая – обнаружен цикл.
3. Если ребра, ведущие из текущей вершины, отсутствуют или ведут только в черные вершины, то текущая вершина считается конечной вершиной и окрашивается в черный цвет.

В случае с топологической сортировкой на 3 шаге мы также заносим «конечную» вершину в стек (на самом деле они не являются конечными, так как для каждой следующей вершины также вызывается dfs() и таким образом все вершины в какой-то момент будут «конечными»).

Топологическая сортировка с помощью обхода в глубину

Вся топологическая сортировка сводится к последовательному обходу графа в глубину и занесением «черных» вершин в стек. После того, как все вершины окрашены в черный цвет, поочередно вынимаем новый порядок вершин из стека и именно такая последовательность и будет топологически отсортированной

Словесно описать такой алгоритм можно следующим образом:

1. Вызываем обход в глубину для каждой вершины, при этом проверяем, чтобы в графе не было циклов.
2. Когда все вершины пройдены, поочередно вынимаем из стека новые номера следующим образом: достаём верхний номер `k`, тогда `result[k] = i`, где `i` – номер элемента в стеке сверху (начиная с 0).

Псевдокод

Пусть у нас есть объект `graph`, содержащий в себе `n` объектов типа `node`. При этом каждый объект `node` может иметь `m` целых чисел `edge`, которые есть номер вершины, на которую этот объект указывает. Функция `dfs` возвращает `true`, если натыкается на цикл. Функция сортировки возвращает `false` в том случае, если был обнаружен цикл и топологическая сортировка невозможна

```
bool dfs(node)
{
    if (node.color == grey) return true;
    if (node.color == black) return false;
    node.color = grey;

    for i in node.edges
    {
       if (dfs(node.edge[i])) return true;
    }

    stack.push(node);
    node.color = black;
    return false;
}

bool topologicalSort()
{
    for i in graph.nodes
    {
        if (dfs(graph.node[i])) return false;
    }

    for i in graph.nodes
    {
        result[stack.pop] = i;
    }

    return true;
}
```

Практическая реализация

Алгоритм реализован на C++. Так как известно, что топологическая сортировка используется для более сложных алгоритмах на графах, то было принято решение реализовать алгоритм на C++, который позволяет создать такой интерфейс, который было бы удобно использовать не только для топологической сортировки. Созданы 2 класса:

- Класс `node`, который имеет ключ `value`, а также вектор `edge` с набором ссылок на другие узлы. Также в целях лучшей читаемости кода в этот класс также добавлено публичное свойство color, которое является необходимым для топологической сортировки (но не обязательным для дальнейшего использования графа).
- Класс graph, который содержит массив объектов node, а также их количество amount. В этом классе реализованы методы dfs и topological_sort, которые и являются ключевыми функциями алгоритма топологической сортировки. Также, для тестирования, в этом классе реализован конструктор, который создает ациклический орграф заданного размера со случайными ребрами (но не допуская 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.

Основная часть кода приведена ниже:

```
bool graph::dfs(unsigned int curNode, uiVector &stack)
{
    //проверяю цвета текущей вершины
    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;
    //запускаю поиск в глубину для всех вершин, на которые ссылается
текущая
    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;
        }
    }
    //если больше идти некуда - заношу в стек и крашу в черный
    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;
        }
    }
    //отсортированная последовательность будет лежать в этом векторе
    result.reserve(this->amount);
    for (unsigned int i = 0; i < this->amount; i++)
    {
        unsigned int index = stack.back();
        result[index] = i;
        stack.pop_back();
    }
}
```

Полный код можно посмотреть по ссылке:

https://github.com/soomrack/MR2018/tree/master/Vlbager/Topological_Sorting

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, окрашивая ее в черный цвет. Таким образом, от каждой серой вершины, возможно только некоторое количество проверок на то, является ли смежная ей белой, а общее количество таких проверок от всех вершин равно количеству ребер в графе. Непосредственно в функции топологической сортировки производится еще некоторое количество сравнений, не больше, чем n. В большинстве случаев количество ребер в графе гораздо больше количества

вершин. Поэтому определяющим фактором в сложности алгоритма является количество ребер графа.