Лабораторная работа № 8 по курсу: Дискретный анализ

Выполнил студент группы М8О-208Б-17 МАИ *Милько Павел.*

Задача

Разработать жадный алгоритм решения задачи, определяемой своим вариантом. Доказать его корректность, оценить скорость и объём затрачиваемой оперативной памяти. Реализовать программу на языке C или C++, соответствующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

Вариант 6: Топологический поиск.
Заданы N объектов с ограничениями на расположение вида "A должен находиться перед B". Необходимо найти такой порядок расположения объектов, что все ограничения будут выполняться.

Входные данные: на первой строке два числа, N и M, за которыми следует M строк с ограничениями вида "A B" ($1 \leq A, B \leq N$) определяющими относительную последовательность объектов с номерами A и B.

Выходные данные: -1 если расположить объекты в соответствии с требованиями невозможно, последовательность номеров объектов в противном случае.

Информация

Суть жадных алгоритмов состоит в принятии локально-оптимальных решений в надежде, что конечное решение так же окажется оптимальным. Но не всегда применим жадный алгоритм, чтобы доказать корректность его применения, нужно доказать что исследуемый объект является матроидом.

Матроидом называется пара множеств E, I, состоящая из конечного множества E, называемого базовым множеством матроида, и множества его подмножеств I, называемого множеством независимых множеств матроида, причем I удовлетворяет свойствам:

- Множество I не пусто. Даже если исходное множество E было пусто – $E = 0$, то I будет состоять из одного элемента - множества, содержащего пустое $I = \{\{0\}\}$.
- Любое подмножество любого элемента множества I также будет элементом этого множества.
- Если $X, Y \in I$, причем $|X| = |Y| + 1$, тогда существует элемент $x \in X \backslash Y$, такой что $Y \cup \{x\} \in I$.

Этапы построения алгоритма решения подзадач:

- Описать структуру оптимального решения.

- Составить рекурсивное решение для нахождения оптимального решения.
- Вычисление значения, соответствующего оптимальному решению, методом восходящего анализа.
- Непосредственное нахождение оптимального решения из полученной на предыдущих этапах информации.

Метод решения

Для решения моей задачи использовался немного упрощенный (казалось бы куда ещё проще) dfs для поиска циклов в графе. Оригинальный алгоритм раскрашивал вершины в три цвета, в моей же задаче циклов нет по условию, мне лишь нужно не заходить в уже посещённые вершины. Сам обход тривиален и жаден до невозможности.

Оценочная сложность: по сути вызывается обход в глубину для каждой компоненты связности графа. Есть 2 крайних случая: когда все вершины принадлежат разным компонентам связности (граф без рёбер), и случай когда компонента связности всего одна, зато путей в графе максимально возможное количество.

В первом случае сложность будет $O(n)$. Во втором — $O(m)$ или $O(n^2)$ в худшем случае. Итого асимптотика алгоритма такая же как у dfs.

Исходный код

main.cpp

```
#include <iostream>
#include <vector>

class TopologicalSort {
    std::vector<bool> used;
    std::vector<size_t> ans;

    void dfs(const std::vector<std::vector<bool>>& matrix, size_t v)
    {
        used[v] = true;
        for (size_t i = 0; i < matrix.size(); ++i)
            if (not used[i] and matrix[v][i])
                dfs(matrix, i);

        ans.push_back(v);
    }

public:
    TopologicalSort(const std::vector<std::vector<bool>>& matrix)
        : used(matrix.size(), false)
        , ans({})
    {
        for (size_t i = 0; i < matrix.size(); ++i)
```

```
            if (not used[i])
                dfs(matrix, i);

        for (auto i = ans.rbegin(); i != ans.rend(); ++i)
            std::cout << *i + 1 << " ";
        std::cout << std::endl;
    }
};

int main()
{
    size_t n, m;
    std::cin >> n >> m;
    std::vector<std::vector<bool>> matrix(n);
    for (size_t i = 0; i < n; ++i)
        matrix[i].assign(n, false);

    for (size_t i = 0; i < m; ++i) {
        size_t a, b;
        std::cin >> a >> b;
        matrix[a - 1][b - 1] = true;
    }

    TopologicalSort result(matrix);
    return 0;
}
```

Генератор тестов

```
#!/usr/bin/python

import random
import sys

if len(sys.argv) == 1:
    print("Usage:\n{} <pair count>(int) <output file>(default stdout)".format(
        sys.argv[0]))
    exit(1)

print("set output array>>> ", end='')
line = [int(i) for i in input().split()]

with (sys.stdout if len(sys.argv) < 3 else open(sys.argv[2], 'w')) as out:
    print(len(line), int(sys.argv[1]), file=out)
    for i in range(int(sys.argv[1])):
        rnd = random.randint(1, len(line)-1)
        print(line[rnd-1], line[random.randint(rnd+1, len(line))-1], file=out)
```

Выводы

Чем жаднее тем проще! Жадные алгоритмы мне понравились больше чем динамическое программирование, хотя теория и кажется намного сложнее, она интуитивно понятна и её довольно просто применять на практике. Они встречаются сплошь и рядом. Те же алгоритмы обхода в глубину и в ширину. Наверное все алгоритмы на графах (и на деревьях, как частных случаях графов, соответственно) являются жадными. Тут простор для примеров открывается огромный: от Укконена до Хаффмана.

Хотя отчасти жадные алгоритмы являются частным случаем динамического программирования. Не так сложно сделать жадный алгоритм, как доказать что его можно использовать.