Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №8 по курсу «Дискретный анализ»

Студент: М. А. Инютин
Преподаватель: А. А. Кухтичев
Группа: М8О-207Б-19
Дата:
Оценка:
Подпись:

Москва, 2021

Лабораторная работа №8

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++, соответсвующую построенному алгоритму. Формат входных и выходных данных описан в варианте задания.

Дан набор из N объектов. Каждому из объектов присвоен свой номер: 1, 2, 3 и т.д. Кроме того, заданы M условий с ограничиениями на расположение вида «A должен находиться перед B». Необходимо найти такой минимальный набор правил, что все остальные ограничения будут выполняться.

1 Описание

Требуется реализовать топологическую сортировку. Согласно [1], топологическая сортировка перенумерует вершины графа таким образом, что каждое рёбро будет вести из вершины с меньшим номером в вершину с большим.

Применимо к моей задаче, можно представить ограничиение вида «A должен находиться перед B» как дугу графа из вершины A в вершину B. Тогда применение топологической сортировки переставит элементы в требуемом порядке, удовлетворяющим ограничениям.

Граф правил может быть несвязным, поэтому добавим правило, что некоторый элемент x должен быть перед всеми другими элементами. Иными словами, добавим вершину, которая смежна со всеми остальными.

Чтобы топологическая сортировка существовала, нужно проверить граф на наличие циклов. Это можно сделать с помощью обхода в глубину [2]. Для этого непосещённые вершины красим в белый цвет, посещённые в чёрный, а вершины, из которых ещё может быть продолжен обход в глубину, серым. Если при обходе мы встретим серую вершину, то в графе присутствует цикл — топологическая сортировка невозможна.

После проверки графа на циклы снова выполним обход в глубину и сохраним вершины в порядке выхода из них. Так самыми первыми в полученном векторе будут листы и более глубокие вершины (те элементы, которые должны быть расположены правее остальных).

Такой подход использует идею жадного алгоритма: мы не перебираем все возможные варианты перестановок даже между элементами на одинаковой глубине. Оптимальный выбор запоминать самые глубоки вершины, а затем менее глубокие даёт оптимальное решение в целом. Пусть для какой-то вершины u известны оптимальные решения для всех вершин, смежных с ней. Тогда в указанных ограничениях элемент в этой вершине должен стоять впереди всех элементов, находящихся в соседях. Как только обход в глубину пройдёт по всем соседям, они будут добавлены в ответ, а уже после них будет добавлена вершина u, что удовлетворяет всем ограничениям. В конце останется только сделать реверс ответа.

Требуется вывести минимальное количество правил, что все остальные ограничения будут выполняться. После сортировки мы получим граф, в котором из каждой вершины, кроме последней, будет вести только одна дуга. Так количество правил в наборе будет равно $N - 1$, так как полученный граф будет связным и ацикличным.

Обход в глубину имеет сложность $O(|V| + |E|)$, где V — множество вершин, а E — множества ребёр графа. Два обхода в глубину будут иметь временную сложность $O(N + M)$. Объём дополнительной затраченной памяти $O(N)$.

2 Исходный код

Для эффективного хранения графа правил я использую списки смежности $TGraph$. Константы $COLOUR\_WHITE$, $COLOUR\_GRAY$ и $COLOUR\_BLACK$ нужны для поиска циклов в графе и покраске вершин в соответсвующие цвета. $DFSCycle$ возвращает истину, если находит цикл в графе, ложь в противном случае.

Функция DFS выполняет обход в глубину и добавляет в ответ вершину тогда, когда завершён обход во всех её соседях.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4
5  using TGraph = std::vector< std::vector<int> >;
6
7  const int COLOUR_WHITE = 0;
8  const int COLOUR_GRAY = 1;
9  const int COLOUR_BLACK = 2;
10
11 bool DFSCycle(int u, const TGraph & g, std::vector<int> & colour) {
12     if (colour[u] == COLOUR_GRAY) {
13         return true;
14     } else if (colour[u] == COLOUR_BLACK) {
15         return false;
16     }
17     colour[u] = COLOUR_GRAY;
18     for (size_t i = 0; i < g[u].size(); ++i) {
19         int v = g[u][i];
20         if (DFSCycle(v, g, colour)) {
21             return true;
22         }
23     }
24     colour[u] = COLOUR_BLACK;
25     return false;
26 }
27
28 void DFS(int u, const TGraph & g, std::vector<int> & visited, std::vector<int> & res)
       {
29     if (visited[u]) {
30         return;
31     }
32     visited[u] = true;
33     for (size_t i = 0; i < g[u].size(); ++i) {
34         int v = g[u][i];
35         DFS(v, g, visited, res);
36     }
37     res.push_back(u);
38 }
```

```
39
40 int main() {
41     int n, m;
42     std::cin >> n >> m;
43     TGraph g(n + 1);
44     for (int i = 0; i < n; ++i) {
45         g.back().push_back(i);
46     }
47     for (int i = 0; i < m; ++i) {
48         int u, v;
49         std::cin >> u >> v;
50         g[u - 1].push_back(v - 1);
51     }
52     std::vector<int> colour(n + 1, COLOUR_WHITE);
53     if (DFSCycle(n, g, colour)) {
54         std::cout << "-1\n";
55         return 0;
56     }
57     std::vector<int> res;
58     std::vector<int> visited(n + 1);
59     DFS(n, g, visited, res);
60     std::reverse(res.begin(), res.end());
61     for (size_t i = 1; i < res.size() - 1; ++i) {
62         std::cout << res[i] + 1 << ' ' << res[i + 1] + 1 << '\n';
63     }
64 }
```

3 Консоль

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ make
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror main.cpp -o solution
.engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ cat tests/1.in
3 2
1 2
2 3
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./solution <tests/1.in
1 2
2 3
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ cat tests/2.in
8 8
6 4
8 5
2 5
1 7
3 1
7 2
2 8
3 6
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./solution <tests/2.in
3 6
6 4
4 1
1 7
7 2
2 8
8 5
```

4 Тест производительности

В тесте производительности сравним жадный алгоритм с наивным, который перебирает все перестановки элементов.

Тесты состоят из 8, 10 и 12 чисел с соответсвующим числом ограничений.

Наивный алгоритм:

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ make brute
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror brute.cpp -o brute
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./brute <tests/2.in
Brute time is 1.028 ms
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./brute <tests/3.in
Brute time is 21.427 ms
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./brute <tests/4.in
Brute time is 5598.469 ms
```

Жадный алгоритм:

```
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ make greedy
g++ -g -O2 -pedantic -std=c++17 -Wall -Wextra -Werror greedy.cpp -o greedy
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./greedy <tests/2.in
Greedy time is 0.007 ms
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./greedy <tests/3.in
Greedy time is 0.009 ms
engineerxl@engineerxl-GF63-Thin-9RCX:~/Study/DA/lab8$ ./greedy <tests/4.in
Greedy time is 0.009 ms
```

Перебор всех перестановок элементов имеет сложность $O(n!)$, что уже для 12 элементов работает больше секунды. Видно, что жадный алгоритм выигрывает даже на достаточно маленьких ограничениях.

5 Выводы

В результате выполнения лабораторной работы я изучил основные алгортмы, использующие идею жадных алгоритмов, составил и отладил программу для своего варианта задания.

В отличие от динамического программирования жадные алгоритмы предполагают, что задача имеет оптимальное решение, которое строится из оптимальных решений для подзадач с заранее определённым выбором, а не перебором всех вариантов перехода. Такой подход уменьшает временные и пространственные ресурсы, нужные для решения задачи.

К сожалению, жадные алгоритмы применимы к не очень большому кругу задач из-за структуры этих задач. Например, известная NP-полная задача коммивояжёра может быть решена методом динамического программирования по подмножествам, а жадный алгоритм достаточно быстро может найти приближённое решение.

На практике почти все наши решения подчиняются логике жадных алгоритмов: пообедать вкуснее, купить телефон получше, заплатить меньше. Полезно понимать, что такой подход не всегда работает и что в половине случаев нужно продумывать свои действия наперёд, перебирая возможные варианты.

Список литературы

[1] *MAXimal :: algo :: Топологическая сортировка — e-maxx.ru*
URL: `https://e-maxx.ru/algo/topological_sort` (дата обращения: 01.05.2021).

[2] *MAXimal :: algo :: Поиск в глубину — e-maxx.ru*
URL: `https://e-maxx.ru/algo/dfs` (дата обращения: 01.05.2021).