Лабораторная работа № 9
Обход графа в глубину (4 часа)

Краткий теоретический материал

Обход неориентированного графа в глубину

Существует два основных алгоритма обхода графов: *обход в ширину* (breadth-first search, BFS) и *обход в глубину* (depth-first search, DFS). Для некоторых задач нет абсолютно никакой разницы, какой тип обхода (поиска) использовать, но для других эта разница является критической.

Разница между поиском в ширину и поиском в глубину заключается в порядке исследования вершин. Этот порядок зависит полностью от структуры-контейнера, используемой для хранения *открытых*, но не *обработанных* вершин.

- *Очередь*. Помещая вершины в очередь типа FIFO, мы исследуем самые старые неисследованные вершины первыми. Таким образом, наше исследование медленно распространяется вширь, начиная от стартовой вершины. В этом суть обхода в ширину.
- *Стек*. Помещая вершины в стек с порядком извлечения LIFO, мы исследуем их, отклоняясь от пути для посещения очередного соседа, если таковой имеется, и возвращаясь назад, только если оказываемся в окружении ранее открытых вершин. Таким образом, мы в своем исследовании быстро удаляемся от стартовой вершины, и в этом заключается суть обхода в глубину.

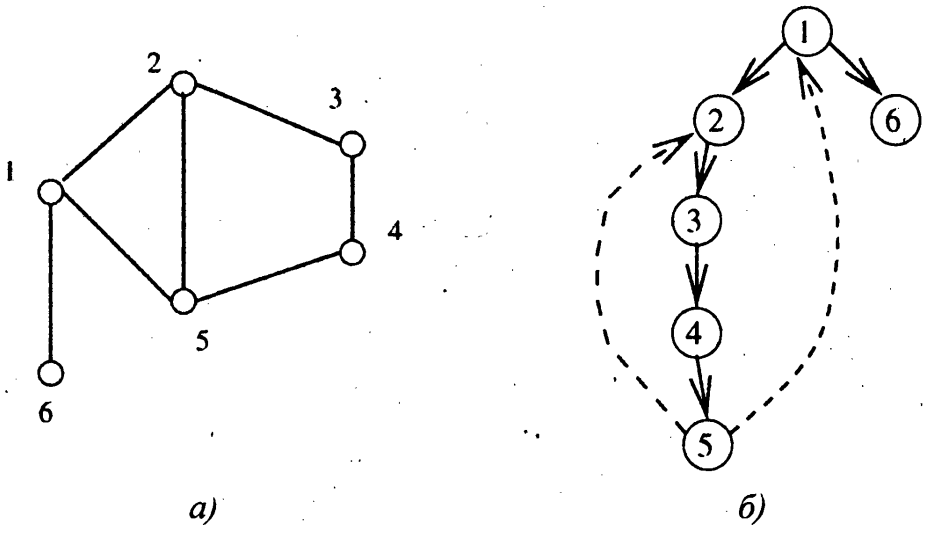

Рис. 1. Неориентированный граф и дерево его обхода в глубину

Наша реализация процедуры обхода в глубину отслеживает время обхода для каждой вершины. Каждый вход в любую вершину и выход из нее считаются затратой времени. Для каждой вершины ведется учет затрат времени на вход и выход.

Процедуру обхода в глубину можно реализовать рекурсивным методом, что позволяет избежать явного использования стека. Псевдокод алгоритма обхода в глубину показан в листинге 1.

Листинг 1. Обход графа в глубину

```
DFS(G, u)
    state[u] = “discovered”
    обрабатываем вершину u, если необходимо
    entry[u] = time
    time = time + 1
    for each v ∈ Adj[u] do
        обрабатываем ребро (u, v), если необходимо
        if state[v] = “undiscovered” then
            p[v] = u
            DFS(G, v)
    state[u] = “processed”
    exit[u] = time
    time = time + 1
```

При обходе в глубину интервалы времени, потраченного на посещение вершин, обладают интересными свойствами. В частности:

- *Посещение предшественника*. Допустим, что вершина *x* является предшественником вершины *y* в дереве обхода в глубину. Это подразумевает, что вершина *x* должна быть посещена раньше, чем вершина *y*, т. к. невозможно быть рожденным раньше своего отца или деда. Кроме этого, выйти из вершины *x* можно только после выхода из вершины *y*, т. к. механизм поиска в глубину предотвращает выход из вершины *x* до тех пор, пока мы не вышли из всех ее потомков. Таким образом, временной интервал посещения *y* должен вкладываться в интервал посещения вершины *x*.
- *Количество потомков*. Разница во времени выхода и входа для вершины *v* свидетельствует о количестве потомков этой вершины в дереве обхода в глубину. Показания часов увеличиваются на единицу при каждом входе и каждом выходе из вершины, поэтому количество потомков данной вершины будет равно половине разности между моментом выхода и моментом входа.

Мы будем использовать время входа и выхода в разных приложениях обхода в глубину. Нам нужно будет выполнять разные действия при каждом входе и выходе из вершины, для чего из процедуры `dfs` будут вызываться функции `process_vertex_early()` и `process_vertex_late()` соответственно.

Другим важным свойством обхода в глубину является то, что он разбивает ребра неориентированного графа на два класса: *древесные* (tree edges) и *обратные* (back edges). Древесные ребра используются при открытии новых вершин и закодированы в родительском отношении. Обратные ребра – это те ребра, у которых второй конец является предшественником расширяемой вершины, и поэтому они направлены обратно к дереву.

Удивительным свойством обхода в глубину является то, что все ребра попадают в одну из этих двух категорий. Почему ребро не может соединять одноуровневые узлы, а только родителя с потомком? Потому, что все вершины, достижимые из данной вершины *v*, уже исследованы ко времени окончания обхода, начатого из вершины *v*, и такая топология невозможна для неориентированных графов. Данная классификация ребер является принципиальной для алгоритмов, основанных на обходе в глубину.

На рис. 1 изображен неориентированный граф и дерево его обхода в глубину.

Реализация

Обход в глубину можно рассматривать как обход в ширину с использованием стека вместо очереди. Достоинством реализации обхода в глубину посредством рекурсии (листинг 2) является то, что она позволяет обойтись без явного использования стека.

Комментарии к листингу 2:

- `finished` – флаг, позволяющий принудительно завершить обход графа;
- `parent[v]` – родитель вершины v в дереве обхода в глубину;
- `discovered[v] == TRUE`, если вершина *v* открыта;
- `processed[v] == TRUE`, если вершина *v* обработана;
- `process_vertex_early(v)` – функция, вызываемая в начале обработки вершины;
- `process_vertex_late(v)` – функция, вызываемая в конце обработки вершины;
- `entry_time[v], exit_time[v]` – время входа и время выхода из вершины *v*;
- `g->directed == TRUE`, если граф является ориентированным.

Листинг 2. Обход графа в глубину

```
dfs(graph *g, int v)
{
    edgenode *p;            /* Временный указатель */
    int y;                  /* Следующая вершина */

    if (finished) return;   /* Завершение поиска */

    discovered[v] = TRUE;
    time = time + 1;
    entry_time[v] = time;

    process_vertex_early(v);

    p = g->edges[v];

    while (p != NULL) {
        y = p->y;
        if (discovered[y] == FALSE) {
            parent[y] = v;
            process_edge(v, y);
            dfs(g, y);
        }
        else if ((!processed[y]) || (g->directed))
            process_edge(v,y);

        if (finished) return;

        p = p->next;
    }
    process_vertex_late(v);

    time = time + 1;
    exit_time[v] = time;

    processed[v] = TRUE;
}
```

Поиск циклов

Наличие обратных ребер является ключевым фактором при поиске циклов в неориентированных графах. Если в графе нет обратных ребер, то все ребра являются древесными и дерево не содержит циклов. Но любое обратное ребро, идущее от вершины *x* к предшественнику *y*, создает цикл, или замкнутый маршрут между вершинами *y* и *x*, цикл легко найти посредством обхода в глубину, как показано в листинге 3.

Листинг 3. Поиск цикла

```
process_edge(int x, int y)
{
    if (discovered[y] && parent[x] != y) {   /* Найдено обратное ребро */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        printf("\n\n");
        finished = TRUE;
    }
}
```

Вызов функции `find_path(y, x, parent)` находит путь от вершины *x* до вершины *y* при движении по родителям вершин.

Поиск точек сочленения

Запустим обход в глубину из произвольной вершины графа; обозначим её через `root`. Заметим следующий факт (который несложно доказать):

- Пусть мы находимся в обходе в глубину, просматривая сейчас все рёбра из вершины `v≠root`. Тогда, если текущее ребро `(v, to)` таково, что из вершины `to` и из любого её потомка в дереве обхода в глубину нет обратного ребра в какого-либо предка вершины `v`, то вершина `v` является точкой сочленения. В противном случае, т.е. если обход в глубину просмотрел все рёбра из вершины `v`, и не нашёл удовлетворяющего вышеописанным условиям ребра, то вершина `v` не является точкой сочленения. (В самом деле, мы этим условием проверяем, нет ли другого пути из `v` в `to`).
- Рассмотрим теперь оставшийся случай: `v=root`. Тогда эта вершина является точкой сочленения тогда и только тогда, когда эта вершина имеет более одного сына в дереве обхода в глубину. (В самом деле, это означает, что, пройдя из `root` по произвольному ребру, мы не смогли обойти весь граф, откуда сразу следует, что `root` – точка сочленения).

Теперь осталось научиться проверять этот факт для каждой вершины эффективно. Для этого воспользуемся «временами входа в вершину», вычисляемыми алгоритмом поиска в глубину.

Итак, пусть `entry_time[v]` – это время захода поиска в глубину в вершину `v`. Теперь введём массив `f[v]`, который и позволит нам отвечать на вышеописанные запросы. Время `f[v]` равно минимуму из времени захода в саму вершину

entry_time[v], времён захода в каждую вершину p, являющуюся концом некоторого обратного ребра (v, p), а также из всех значений f[to] для каждой вершины to, являющейся непосредственным сыном v в дереве поиска:

$$f[v] = min \begin{cases} entry\_time[v], \\ entry\_time[p], & \text{для всех обратных рёбер } (v,p) \\ f[to], & \text{для всех древесных рёбер } (v,to) \end{cases}$$

Тогда, из вершины v или её потомка есть обратное ребро в её предка тогда и только тогда, когда найдётся такой сын to, что f[to]<entry_time[v].

Таким образом, если для текущего ребра (v, to) (принадлежащего дереву поиска) выполняется f[to]≥entry_time[v], то вершина v является точкой сочленения. Для начальной вершины v=root критерий другой: для этой вершины надо посчитать число непосредственных сыновей в дереве обхода в глубину.

Задание

1. При помощи обхода в глубину определить количество компонент связности графа и вывести для каждой компоненты входящие в нее вершины.

2. При помощи обхода в глубину определить, содержит ли заданный неориентированный граф хотя бы один цикл. В случае существования цикла вывести его длину и список вершин в порядке обхода.

3. При помощи обхода в глубину найти все точки сочленения графа.

Требования к отчету

Отчет по лабораторной работе должен включать:

1. Титульный лист; задание; исходный код.
2. Примеры работы программы (скриншоты).
3. Выводы.