Лабораторная работа № 10
Обход ориентированного графа в глубину (4 часа)

Краткий теоретический материал

Обход ориентированного графа в глубину

При обходе неориентированных графов каждое ребро или находится в дереве обхода в глубину, или является обратным ребром к предшествующему ребру в дереве.

Для ориентированных графов диапазон допустимых маркировок обхода в глубину может быть более широким. В самом деле, при обходе ориентированных графов могут использоваться все четыре типа ребер, показанные на рис. 1.

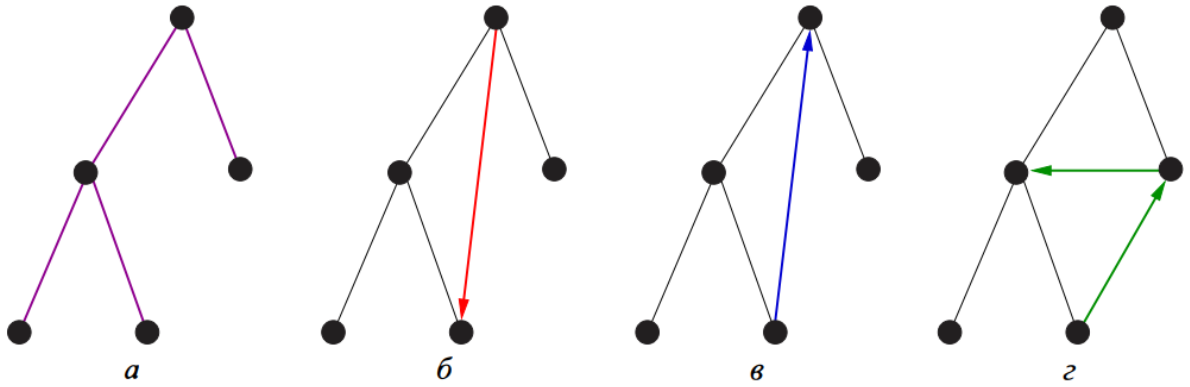

Рис. 1. Возможные типы ребер при обходе графа: *а* — древесные ребра; *б* — прямое ребро; *в* — обратное ребро; *г* — поперечные ребра. Прямые и поперечные ребра могут встречаться при обходе в глубину *только* в ориентированных графах

Но эта классификация оказывается полезной при разработке алгоритмов для работы с ориентированными графами. Для каждого типа ребра обычно предпринимается соответствующее ему действие.

Тип ребра можно без труда определить, исходя из состояния вершины, времени ее открытия и ее родителя. Соответствующая процедура показана в листинге 1.

Листинг 1. Определение типа ребра

```
int edge_classification(int x, int y)
{
    if (parent[y] == x) return(TREE);
    if (discovered[y] && !processed[y]) return(BACK);
    if (processed[y] && (entry_time[y]>entry_time[x])) return(FORWARD);
    if (processed[y] && (entry_time[y]<entry_time[x])) return(CROSS);
    printf("Warning: unclassified edge (%d,%d)\n",x,y);
}
```

Сильно связные компоненты

Классическим применением поиска в глубину является разложение ориентированного графа на *сильно связные компоненты* (strongly connected components). Орграф является *сильно связным* (strongly connected), если для любой пары его вершин (v, u) вершина v достижима из вершины u и наоборот. В качестве практического примера сильно связного графа можно привести граф дорожных сетей с двусторонним движением.

Является ли граф $G = (V, E)$ сильно связным, можно с легкостью проверить посредством обхода графа за линейное время. Начнем обход с произвольной вершины v. Каждая вершина графа должна быть достижимой из этой вершины и, следовательно, открыта обходом в ширину или глубину с начальной точкой в вершине v. В противном случае граф G не может быть сильно связным. Потом создадим граф $G' = (V, E')$ с точно таким же набором вершин и ребер, как и граф G, но с обратной ориентацией ребер, т. е., ориентированное ребро $(x, y) \in E$ тогда и только тогда, когда $(y, x) \in E'$. Таким образом, любой путь от вершины v к вершине z в графе G' соответствует пути от вершины z к вершине v в графе G. Выполнив обход в глубину графа G' от вершины v, мы найдем все вершины с путями к этой вершине в графе G. Граф является сильно связным тогда и только тогда, когда вершина v достижима из любой вершины в графе G и все вершины в графе G достижимы из вершины v.

Графы, которые сами не являются сильно связными, можно разбить на сильно связные компоненты, как показано на рис. 2.

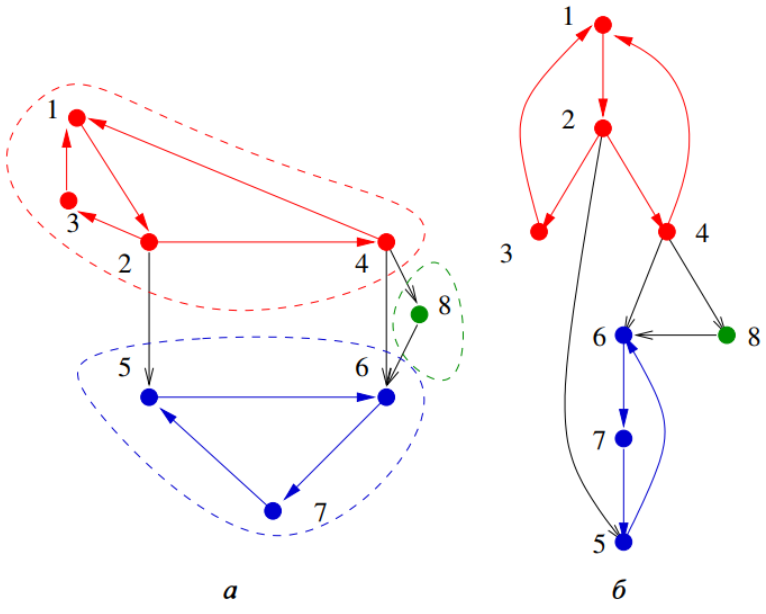

Рис. 2. Сильно связные компоненты графа *(а)* и дерево обхода в глубину *(б)*

Сильно связные компоненты графа и соединяющие их слабо связные ребра можно найти с помощью обхода в глубину. Этот алгоритм основан на том обстоятельстве, что с помощью обхода в глубину легко найти ориентированный цикл, поскольку такой цикл дает любое обратное ребро и путь вниз в дереве обхода в глубину. Все вершины в этом цикле должны быть в одной и той же сильно связной компоненте. Таким образом мы можем сжать вершины в этом цикле в одну вершину, представляющую всю компоненту, после чего повторить обход. Этот процесс прекращается, когда исчерпаны все ориентированные циклы, а каждая вершина представляет отдельную сильно связную компоненту.

Алгоритм, приведённый в листинге 2, отделяет от дерева обхода по одной сильной компоненте за шаг и присваивает всем ее вершинам номер данной компоненты.

Листинг 2. Алгоритм разложения графа на сильно связные компоненты

```
strong_components(graph *g)
{
    int i;      /* Счётчик */
    for (i=l; i<=(g->nvertices); i++) {
        low[i] = i;
        scc[i] = -1;
    }
    components_found = 0;
    init_stack(&active);   /* Инициализация стека */
    initialize_search(&g);
    for (i=l; i<=(g->nvertices); i++)
        if (discovered[i] == FALSE) {
            dfs(g,i);
        }
}
```

Переменная `low[v]` представляет самую старшую известную вершину, находящуюся в той же самой сильно связной компоненте, что и вершина *v*. Эта вершина не обязательно является родителем вершины *v*, но благодаря поперечным ребрам может находиться на одном уровне с ней. Поперечные ребра, которые идут к вершинам из предыдущих сильно связных компонент графа, нам не нужны, т. к. от них нет пути назад к вершине *v*, но в других случаях поперечные ребра подлежат обработке. Прямые ребра не влияют на достижимость по ребрам дерева обхода в глубину и поэтому их можно не принимать во внимание (листинг 3).

Листинг 3. Процедура обработки рёбер

```
int low[MAXV+l];  /* Самая старшая вершина, которая находится в
                                          компоненте вершины v */
int scc[MAXV+l];  /* Номер сильносвязной компоненты для каждой вершины */
process_edge(int x, int y)
{
    int class;  /* Класс ребра */
    class = edge_classification(x,y);
    if (class = BACK) {
        if (entry_time[y] < entry_time[low[x]])
            low[x] = y;
    }
    if (class = CROSS) {
        if (scc[y] == -1)  /* Компонента еще не присвоена */
            if (entry_time[y] < entry_time[low[x]])
                low[x] = y;
    }
}
```

Новая сильно связная компонента считается обнаруженной, когда самой старшей вершиной, достижимой из вершины *v*, является эта же вершина *v*. В таком случае эта компонента снимается со стека (листинг 4).

Листинг 4. Процедуры обработки вершин

```
process_vertex_early(int v)
{
    push(&active,v);
}
process_vertex_late(int v)
{
    if (low[v] == v) { /* Ребро (parent[v],v) отрезает
                          сильно связную компоненту */
        pop_component(v);
    }
    if (parent[v] > 0)
         if (entry_time[low[v]] < entry_time[low[parent[v]]])
             low[parent[v]] = low[v];
}
pop_component(int v)
{
    int t;
    components_found = components_found + 1;
    scc[v] = components_found;
    while ((t = pop(&active)) != v)  {
        scc[t] = components_found;
    }
}
```

Топологическая сортировка

Топологическая сортировка является наиболее важной операцией на бесконтурных ориентированных графах. Этот тип сортировки упорядочивает вершины вдоль линии таким образом, чтобы все ориентированные ребра были направлены слева направо. Такое упорядочивание ребер невозможно в графе, содержащем ориентированный цикл.

Любой бесконтурный ориентированный граф имеет, по крайней мере, одно топологическое упорядочивание (рис. 3).

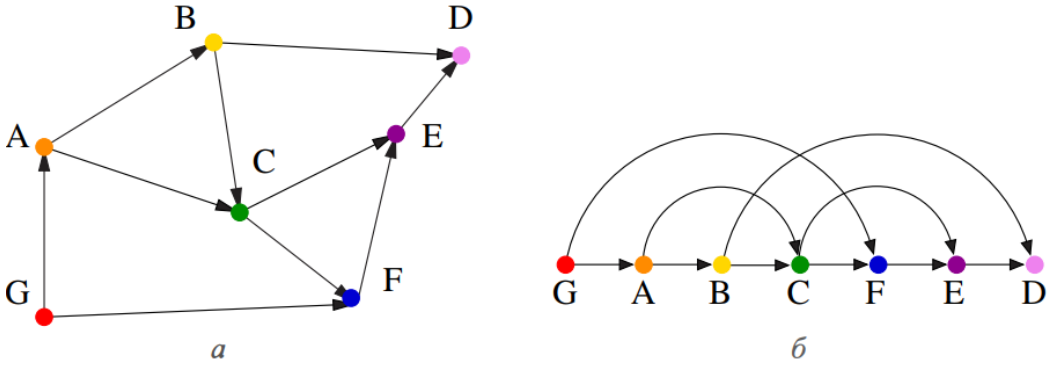

Рис. 3. Бесконтурный ориентированный граф (*а*) с единственным топологическим упорядочиванием: G, A, B, C, F, E, D (*б*)

Важность топологической сортировки состоит в том, что она позволяет упорядочить вершины графа таким образом, что каждую вершину можно обработать перед обработкой ее потомков. Допустим, что ориентированные ребра представляют управление очередностью таким образом, что ребро (x, y) означает, что работу x нужно выполнить раньше, чем работу y. Тогда любое топологическое упорядочивание определяет возможное календарное расписание. Более того, бесконтурный ориентированный граф может содержать несколько таких упорядочиваний.

Обход в глубину является эффективным средством для осуществления топологической сортировки. Ориентированный граф является бесконтурным, если не содержит обратных ребер. Топологическое упорядочивание бесконтурного ориентированного графа осуществляется посредством маркировки вершин в порядке, обратном тому, в котором они отмечаются как обработанные. Почему это возможно? Рассмотрим, что происходит с каждым ориентированным ребром (x, y), обнаруженным при исследовании вершины x:

- если вершина y не открыта, то мы начинаем обход в глубину из вершины y, прежде чем сможем продолжать исследование вершины x. Таким образом,

вершина *y* должна быть помечена как обработанная до того, как такой статус присваивается вершине *x*, и поэтому в топологическом упорядочивании вершина *x* будет находиться перед вершиной *y*, что и требуется;

- если вершина *y* открыта, но не обработана, то ребро (*x*, *y*) является обратным ребром, что невозможно в бесконтурном ориентированном графе, поскольку это создает цикл;
- если вершина *y* обработана, то она помечается соответствующим образом раньше вершины *x*. Следовательно, в топологическом упорядочивании вершина *x* будет находиться перед вершиной *y*, что и требуется.

В листинге 5 приводится реализация топологической сортировки.

Листинг 5. Топологическая сортировка

```
void process_vertex_late(int v) {
    push(&sorted, v);
}

void process_edge(int x, int y) {
    int class;    /* Класс ребра */

    class = edge_classification(x, y);

    if (class == BACK) {
        printf("Warning: directed cycle found, not a DAG\n");
                // Внимание: обнаружен ориентированный цикл,
                // неориентированный граф
    }
}

void topsort(graph *g) {
    int i;    /* Счетчик */

    init_stack(&sorted);

    for (i = 1; i <= g->nvertices; i++) {
        if (!discovered[i]) {
            dfs(g, i);
        }
    }
    print_stack(&sorted);    /* Выводим топологическое упорядочивание */
}
```

Здесь каждая вершина помещается в стек после обработки всех исходящих ребер. Самая верхняя вершина в стеке не имеет входящих ребер, идущих от какой-либо вершины, имеющейся в стеке. Последовательно снимая вершины со стека, получаем их топологическое упорядочивание.

Задание

1. При помощи обхода в глубину определить компоненты сильной связности орграфа. Для каждой компоненты сильной связности вывести входящие в нее вершины.

2. При помощи обхода в глубину выполнить топологическую сортировку вершин ориентированного графа. Если граф содержит ориентированные циклы, вывести сообщение о том, что произвести топологическую сортировку вершин невозможно.

Требования к отчету

Отчет по лабораторной работе должен включать:

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