Лабораторная работа № 8
Обход графа в ширину (4 часа)

Краткий теоретический материал

Обход графа

Самой фундаментальной задачей на графах, возможно, является систематизированное посещение каждой вершины и каждого ребра графа. В действительности, все основные служебные операции по работе с графами (такие как распечатка или копирование графов или преобразования графа из одного представления в другое) являются приложениями обхода графа (graph traversal).

Ключевая идея обхода графа – пометить каждую вершину при первом ее посещении и помнить о том, что не было исследовано полностью. В обходах графов мы будем пользоваться булевыми флагами или перечислимыми типами.

Каждая вершина будет находиться в одном из следующих трех состояний:

- неоткрытая (undiscovered) – первоначальное, нетронутое состояние вершины;
- открытая (discovered) – вершина обнаружена, но мы еще не проверили все инцидентные ей ребра;
- обработанная (processed) – все инцидентные данной вершине ребра были посещены.

Очевидно, что вершину нельзя обработать до того, как она открыта, поэтому в процессе обхода графа состояние каждой вершины начинается с неоткрытого, переходит в открытое и заканчивается обработанным.

Обход графа в ширину

Обход в ширину (breadth-first traversal) является основой для многих важных алгоритмов работы с графами. Далее приводится базовый алгоритм обхода в ширину. На определённом этапе каждая вершина графа переходит из состояния *неоткрытая* в состояние *открытая*. При обходе в ширину неориентированного графа каждому ребру присваивается направление, от «открывающей» к открываемой вершине. В этом контексте вершина u называется родителем (parent) или предшественником (predecessor) вершины v, а вершина v – потомком вершины u. Так как каждый узел, за исключением корня, имеет только одного родителя, получится дерево вершин графа. Это дерево определяет кратчайший путь от корня ко всем другим узлам графа.

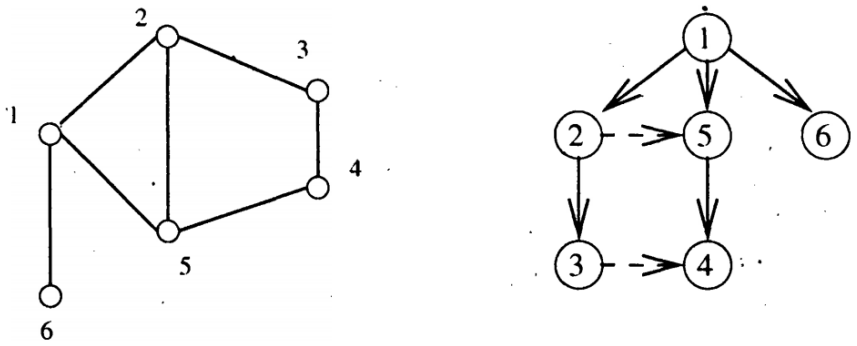

Рисунок 1 – Неориентированный граф и его дерево обхода в ширину

Это свойство делает обход в ширину очень полезным в решении задач поиска кратчайшего пути. В листинге 1 приводится псевдокод алгоритма обхода графа в ширину.

В этом алгоритме приняты следующие обозначения:

`G` – граф, обход которого необходимо выполнить;

`s` – вершина графа, с которой начинается обход;

`state` – массив состояний вершин;

`p` – массив вершин-родителей;

`Q` – очередь вершин для обработки.

`dequeue` – метод, который возвращает очередной элемент из очереди, а затем удаляет его из очереди;

`enqueue` – метод, который добавляет элемент в очередь;

`Adj[u]` – множество вершин, смежных с вершиной `u`.

Листинг 1. Обход графа в ширину

```
BFS (G, s)
     for each vertex u ∈ V(G) - {s} do
          state[u] = "undiscovored"
          p[u] = nil,    // в начале вершины-родители отсутствуют
     state[s] = "discovored"
     p[s] = nil
     Q = {s}             // добавляем в очередь начальную вершину
     while Q ≠ ∅ do
          u = dequeue[Q]
               обрабатываем вершину u требуемым образом
```

```
for each v ∈ Adj[u] do
     обрабатываем ребро (u, v) требуемым образом
     if state[v] = "undiscovored" then
          state[v] = "discovored"
          p[v] = u
          enqueue[Q,v]
state[u] = "processed"
```

Замечание. При практической реализации обхода в ширину на языке C++ можно использовать готовый STL контейнер – queue.

Способы хранения графа в памяти компьютера

Способ 1 – Хранение графа в виде матрицы смежности

```
#define MAXV 1000                    /* Максимальное количество вершин */

/* Описание графа */
bool adjacency[MAXV + 1][MAXV + 1];
int nvertices;                       /* Количество вершин в графе */
int nedges;                          /* Количество ребер в графе */

/* Чтение неориентированного графа */
void read_graph()
{
      int v, u;                      /* Вершины ребра (v, u) */
      cin >> nvertices >> nedges;
      for (int i = 0; i < nedges; i++)
      {
            cin >> v >> u;
            adjacency[v][u] = true;
            adjacency[u][v] = true;
      }
}

/* Вывод списка смежных вершин */
void print_graph()
{
      for (int i = 1; i <= nvertices; i++)
      {
            cout << i << ": ";
            for (int j = 1; j <= nvertices; j++)
                  if (adjacency[i][j] == true)
                        cout << " " << j;
            cout << endl;
      }
}
```

Способ 2 – Хранение графа в виде списка смежности, реализованном с помощью компонентов стандартной библиотеки шаблонов C++

```
#define MAXV 1000                    /* Максимальное количество вершин */

/* Описание графа */
vector<int> edges[MAXV + 1];
int nvertices;                       /* Количество вершин в графе */
int nedges;                          /* Количество ребер в графе */

/* Чтение неориентированного графа */
void read_graph()
{
      int v, u;                      /* Вершины ребра (v, u) */
      cin >> nvertices >> nedges;
      for (int i = 0; i < nedges; i++)
      {
            cin >> v >> u;
            edges[v].push_back(u);
            edges[u].push_back(v);
      }
}

/* Вывод списка смежных вершин */
void print_graph()
{
      for (int i = 1; i <= nvertices; i++)
      {
            cout << i << ": ";
            for (int j = 0; j < edges[i].size(); j++)
                  cout << " " << edges[i][j];
            cout << endl;
      }
}
```

Способ 3 – Хранение графа в виде списка смежности, реализованном с помощью односвязных списков

```
#define MAXV 1000                    /* Максимальное количество вершин */

/* Описание графа */
struct Edgenode
{
      int u;                         /* Информация о смежности */
      Edgenode *next;                /* Следующее ребро в списке */
};
struct Graph
{
      Edgenode *edges[MAXV + 1];     /* Информация о смежности */
      int nvertices;                 /* Количество вершин в графе */
      int nedges;                    /* Количество ребер в графе */
};
```

```
/* Добавление ребра v -> u */
void insert_edge(Graph *g, int v, int u)
{
      Edgenode *t = new Edgenode(); /* Временный указатель */
      t->u = u;
      t->next = g->edges[v];
      g->edges[v] = t;              /* Вставка в начало */
}

/* Чтение неориентированного графа */
void read_graph(Graph *g)
{
      int v, u;                     /* Вершины ребра (v, u) */
      cin >> g->nvertices >> g->nedges;
      for(int i = 0; i < g->nedges; i++)
      {
            cin >> v >> u;
            insert_edge(g, v, u);
            insert_edge(g, u, v);
      }
}

/* Удаление графа */
void delete_edge(Edgenode *t)
{
      if(t != NULL)
      {
            delete_edge(t->next);
            delete t;
      }
}

void delete_graph(Graph *g)
{
       for(int i = 1; i <= g->nvertices; i++)
             delete_edge(g->edges[i]);
}

/* Вывод списка смежных вершин */
void print_graph(Graph *g)
{
       Edgenode *t;
       for(int i = 1; i <= g->nvertices; i++)
       {
             cout << i << ":";
             t = g->edges[i];
             while(t != NULL)
             {
                   cout << " " << t->y;
                   t = t->next;
             }
             cout << endl;
       }
}
```

Задание

1. Выбрать способ хранения графа в памяти компьютера. Реализовать считывание информации о графе из файла.

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

3. При помощи обхода в ширину для выбранной вершины найти кратчайшие пути до всех остальных достижимых вершин графа.

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

5. При помощи обхода в ширину покрасить вершины графа в два цвета так, чтобы вершины одного цвета не были соединены ребром. Выдать соответствующее сообщение, если это сделать невозможно.

Требования к отчету

Отчет по лабораторной работе должен включать:

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