Тема 9. Графы

Обходы, поиск

Граф — множества вершин и ребер.

Ориентированный и неориентированный графы.

Есть разные другие свойства (планарный, двудольный …) — мы в теорию графов не лезем. Просто знакомство с некоторыми алгоритмами и намеки по реализации.

Представления графов.

1. Матрица инцидентности. $a_{i,j}$ — соответствует ребру из вершины i в вершину j. Для неориентированного графа можно рассматривать треугольную часть.

2. По принципу ссылочной реализации. Вершина — узел со списком (массивом) указателей на соседей.

Достоинства и недостатки каждого способа очевидны.

Базовые алгоритмы.

Поиск конкретной вершины.

Обнаружение циклов.

Поиск одного или всех путей из вершины в вершину.

Поиск пути с минимальным (максимальным) весом.

Есть еще множество задач. Для некоторых алгоритмы нетривиальны.

Здесь рассматриваем два фундаментальных алгоритма. Поиск в ширину и поиск в глубину.

Поиск в ширину

Дана вершина A. Определить на каком расстоянии (по числу ребер) находится от нее вершина B.

Пускаем из вершины A волну, которая за один шаг продвигается на расстояние одного ребра. Каждый шаг распространения волны отмечается номерами 0, 1, 2, на вершинах, находящихся на фронте волны.

Формально. Пусть M — граничное множество (фронт волны). Обозначим `x(k)` — вершине x устанавливается метка k, `x.label` — метка вершины x.

шаг 0. Поставить во всех вершинах метку -1 (не посещалась). $M = \emptyset$.

шаг 1. A(0). Добавить A в M.

шаг 2. Инвариант: все вершины из M имеют одну и ту же метку (будем ее обозначать `M.label`).

```
k = M.label
для каждой вершины x из M {
    для каждого соседа y вершины x {
        если y.label == -1 то {
           y(k+1)
           добавить  y в M
        }
    }
    удалить x из M
}
утв: M.label == k+1
```

Шаг 2 повторяется до тех пор, пока в M не будет добавлена вершина B. Метка вершины B при ее добавлении в M и есть ее искомое расстояние от A.

Если при некотором проходе полного шага 2 не было добавлений новых вершин в множество M и при этом вершина B еще не достигнута, то значит вершины A и B не связаны, и алгоритм зевершается.

Наиболее естественная реализация алгоритма на основе очереди, хранящей вершины из M. Из головы очереди извлекается вершина и все ее непосещенные ранее соседи помещаются в конец очереди с новым значением метки. процесс заканчивается либо добавлением вершины B в очередь, либо опустошением этой очереди (все извлекли, но ничего не добавили)

Обнаружение циклов и поиск путей.

шаг 2.

```
k = M.label
для каждой вершины x из M {
    для каждого соседа y вершины x {
        если y.label == -1 то {
           y(k+1)
           добавить  y в M
        } иначе {
           есть цикл неориентированного графа
           (столкнулись два фронта волны)
        }
    }
    удалить x из M
}
```

Путь из A в B определяется из следующих соображений. У вершины с меткой k должен быть хотя бы один сосед с меткой $k-1$, у этого соседа — сосед с меткой $k-2$ и так до A(0).

Сложность: количество вершин + количество ребер

Поиск в глубину

Знакомый нам рекурсивный обход, стартуя из данной вершины — DepthSearch(A, B).

Чтобы не попасть на цикл, красим вершины по ходу дела в три цвета — белый, серый, черный:

белый — не была посещена;

серый — посещена на проходе вниз;

черный — окончательно обработана.

Предварительно все вершины покрашены в белый: x.color = White.

```
bool DepthSearch(C, B) {
    если C==B  (т.е. попали куда надо),
      то обратная цепочка по рекурсии и есть искомый путь,
      return true
    если C.color != White return false
         // если == Gray, то это цикл ориентированного графа
    C.color = Gray
    для каждого соседа x вершины C {
        DepthSearch(x, B)
    }
    C.color = Black
    обрабатываем C, если надо.
    return false
}
```

Очень естественно обнаруживаются все пути.

Сложность: количество вершин + количество ребер.

Алгоритм Дейкстры

Поиск пути из A в B с наименьшим весом (вес имеют ребра).

Модифицированный поиск в ширину.

Метка вершины — ее расстояние от A в смысле суммы весов ребер вдоль найденного пути.

$w(x, y)$ — вес ребра из x в y.

шаг 0. пометить все вершины меткой “бесконечность”. $M = \emptyset$.

шаг 1. A(0). Добавить A в M.

шаг 2. Есть непустое граничное множество M.

```
// для всех точек M и точек ‘‘внутри’’ M известны длины
// кратчайших путей от A через просмотренные точки
Выбираем из M вершину x с наименьшим весом.
для каждого соседа y вершины x {
        если вес(y) > вес(x) + w(x,y) то {
             вес(y) = вес(x) + w(x,y)
             добавить y в M
        }
}
удалить x из M
повторять шаг 2, пока не будут рассмотрены все ребра, входящие в B.
```

Пример для алгоритма Дейкстры

Максимальное паросочетание

Формальная постановка. Есть двудольный граф, надо оставить в нем мексимальное количество ребер между правой и левой долями так, чтобы эти ребра не “соприкасались”, т.е. любая вершина имеет не более одного ребра.

Смысл задачи — распределение ресурсов, исполнителей и т.п. (танцы :))))

Есть классический алгоритм Куна (Kuhn). Идея — построение чередующейся цепочки (пути), где вершины левой и правой долей чередуются. Далее алгоритим пытается удлинить эту цепочку за счет пока не задействованных вершин. Для таких попыток используется по сути поиск в глубину.

Есть также стандартная реализация такого алгоритма. В ней используются

— описание графа заданием ребер из левой доли в правую,

— описание паросочетания заданием ребер из правой доли в левую,

— массив разметки “посещенных и использованных” вершин для корректного поиска в глубину.

Основная функция алгоритма пытается увеличить паросочетание, добавив к нему новое ребро из числа свободных вершин (левой и правой долей).

```
int n; // размерность левой доли
int k; // размерность правой доли
vector < vector<int> > g;  // массив ребер из левой доли i - g[i][j]
vector<int> mt;            // массив паросочетаний   mt[i] - i  (-1 для пока свободны
vector<char> used;         // использованные вершины левой доли

bool Kuhn (int v) {
    if (used[v])  return false; // нас интересуют свободные v
    used[v] = true;             // пометили при проходе поиска

    // цикл для каждого соседа v
    for (size_t i=0; i<g[v].size(); ++i) {
```

```
        // очередной сосед в правой доле
        int to = g[v][i];
        // попытка удлинить паросочетание
        // либо вершина правой доли свободна (-1)
        // либо нашли новое паросочетание и цепляем к нему v
        if (mt[to] == -1 || Kuhn (mt[to])) {
            mt[to] = v;
            return true;
        }
    }
    return false;
}
```

Таким образом, Kuhn строит паросочетание, которому отвечает цепочка с началом в вершине v. Перебирая все вершины левой доли, мы найдем максимальную цепочку и соответственно максимальное паросочетание.

```
  int main() {
    задание графа - массив g
    определение массивов mt, used
    mt.assign (k, -1);
    // основной цикл
    for (int v=0; v<n; ++v) {
        used.assign (n, false);
        try_kuhn (v);
    }
    // результат
    for (int i=0; i<k; ++i) {
        if (mt[i] != -1) printf ("edge %d %d\n", i, mt[i]);
    }
}
```