Основы программирования

Поиск на графах

Обход графа: поиск в глубину

Поиск в глубину позволяет обойти все вершины графа по одному разу, переходя от вершины к вершине по ребрам.

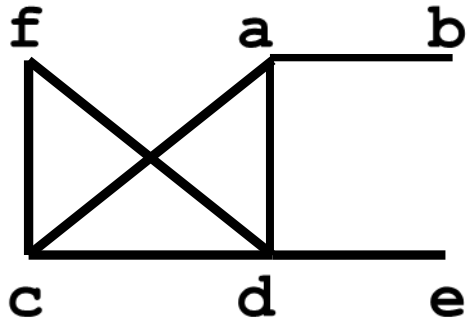
Поиск в глубину реализуется рекурсивным алгоритмом:

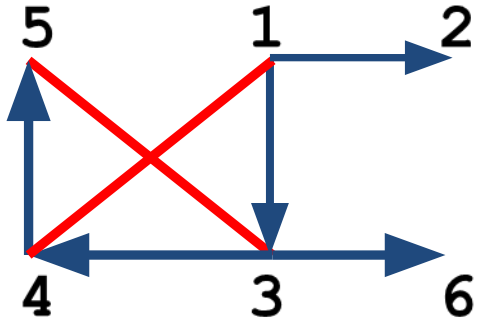
- Пусть выбрана некоторая не рассмотренная ранее вершина **A**
- Перебираем все исходящие из **A** ребра, при этом:
  1) если ребро ведет в не рассмотренную ранее вершину **B**, то продолжаем поиск рекурсивно от **B**
  2) после обработки **B** возвращаемся в **A** и продолжаем перебирать исходящие из **A** ребра
  3) если все ребра из **A** проверены, то либо возвращаемся к вершине, из которой мы пришли в **A** (если такая есть), либо выбираем любую ранее не проверенную вершину и выполняем алгоритм от нее.

Обход графа: поиск в глубину

Граф $G = \{V, E\}$ связный, если для любой пары вершин $v, w \in V$ существует соединяющий их маршрут, проходящий по ребрам.

Поиск в глубину разбивает множество ребер E на 2 подмножества: древесные (от текущей вершины к непроверенной) и обратные (к уже проверенной вершине). Древесные ребра образуют дерево (связный граф без циклов), для которого определен порядок обхода вершин при поиске в глубину.





Классы MGraph и LGraph

Будем добавлять методы к классам MGraph (матрица смежности) и LGraph (списки смежных вершин):

```
class MGraph
{
    bool **mat;    // матрица смежности
    int vernum;    // число вершин
    …
};

class LGraph
{
    List *lst;     // списки смежных вершин
    int vernum;    // число вершин
    …
};
```

Методы MGraph для поиска в глубину

Формируется массив **R[0…n-1]**, где **R[i]** – номер вершины **i** в порядке обхода в глубину (от 1).

```
void MGraph::deep(int cver, int *R, int &cnum)
{
  R[cver] = ++cnum;
  for (int i = 0; i < vernum; i++)
    if (mat[cver][i] && !R[i]) deep(i, R, cnum);
}
int* MGraph::DFS()
{
  int i, cnum, *R = new int[vernum];
  for (i = 0; i < vernum; i++) R[i] = 0;
  for (cnum = i = 0; i < vernum; i++)
    if (!R[i]) deep(i, R, cnum);
  return R;
}
```

Метод deep для LGraph

```
void LGraph::deep(int cver, int *R, int &cnum)
{
  int *pv;
  R[cver] = ++cnum;
  for (pv = lst[cver].get_first();
       pv != NULL;
       pv = lst[cver].get_next())
    if (!R[*pv]) deep(*pv, R, cnum);
}
```

Метод **DFS** точно такой же, как в классе **MGraph**.

Трудоемкость алгоритма **DFS** на списках смежных вершин составляет $O(|E|)$.

Компоненты связности

Компоненты связности неориентированного графа это максимальные (по включению вершин) связные подграфы.

Для выделения компонент связности используем поиск в глубину, внеся следующие изменения:

- добавим в класс MGraph целую переменную **`comptotal`**, в которой будет вычисляться число компонент
- изменим процесс формирования массива **`R: R[i]`** будет хранить номер компоненты (от 1), включающую вершину **`i`** (**`R[i]=0`**, если вершина **`i`** еще не просмотрена).

Приводимые далее методы **`cdeep`** и **`get_comp`** – это модификации методов **`deep`** и **`DFS`**.

Методы MGraph для выделения компонент

```
void MGraph::cdeep(int cver, int *R)
{
  R[cver] = comptotal;
  for (int i = 0; i < vernum; i++)
    if (mat[cver][i] && !R[i]) cdeep(i, R);
}

int* MGraph::get_comp()
{
  int i, *R = new int[vernum];
  comptotal = 0;
  for (i = 0; i < vernum; i++) R[i] = 0;
  for (i = 0; i < vernum; i++)
    if (!R[i])
    { comptotal++; cdeep(i, R); }
  return R;
}
```

Обход графа: поиск в ширину

Поиск в ширину обычно используется для проверки, существует ли маршрут из некоторой вершины-источника **v** в целевую вершину **w**, проходящий по ребрам графа.

В процессе поиска вершины помещаются в очередь для просмотра. Начальная очередь содержит только **v**.

Пусть **u** – текущая извлекаемая из очереди вершина.

Рассмотрим все ребра (**u,x**), при этом возможны варианты:

- **x** просмотрена или находится в очереди – изменений нет,
- **x=w** – маршрут найден, алгоритм завершается
- **x** добавляется в очередь.

Если **w** не найдена, то после обработки всех ребер (**u,x**) из очереди выбирается следующая текущая вершина **u**, и поиск продолжается.

Если очередь закончилась, то маршрута из **v** в **w** нет.

Обход графа: поиск в ширину

При поиске в ширину можно не только определить, существует ли маршрут из **v** в **w**, но и вычислить его минимальную длину (минимальное число пройденных ребер). Для этого нужно вычислять уровни просмотренных вершин (массив **Lev[0…n-1]**):

- вершина-источник **v** имеет уровень 1, начальные значения для остальных вершин **Lev[i]=0** (это означает, что вершина **i** еще не рассмотрена),
- если **Lev[u]=a**, существует ребро **(u,x)** и **Lev[x]=0**, то для **x** устанавливается значение уровня **Lev[x]=a+1**,
- если **Lev[w]>0**, то существует, по крайней мере, один маршрут, и кратчайший маршрут содержит **Lev[w]-1** ребро.

Обход графа: поиск в ширину

Функции для поиска в ширину имеют 1 параметр – номер **v** (от 0) вершины-источника. Поиск закончится, когда будут просмотрены все вершины, достижимые из **v**.

Для всех вершин графа будем формировать и возвращать массив уровней вершин **Lev**.

Если в результате поиска **Lev[w]=0** для некоторой вершины **w**, то **w** не достижима из **v**.

Для организации очереди используем класс **IQueue** (очередь целых чисел) из раздела «Структуры и классы».

Метод MGraph для поиска в ширину

```
int* MGraph::BFS(int v)
{
  int u, x, *Lev = new int[vernum];
  IQueue Que(vernum);
  for (u = 0; u < vernum; u++) Lev[u] = 0;
  Lev[v] = 1; Que.push(v);
  for (u = Que.pop(); u >= 0; u = Que.pop())
    for (x = 0; x < vernum; x++)
      if (mat[u][x] && !Lev[x])
      {
        Lev[x] = Lev[u] + 1;
        Que.push(x);
      }
  return Lev;
}
```

Метод LGraph для поиска в ширину

```
int* LGraph::BFS(int v)
{
  int u, *px, *Lev = new int[vernum];
  IQueue Que(vernum);
  for (u = 0; u < vernum; u++) Lev[u] = 0;
  Lev[v] = 1; Que.push(v);
  for (u = Que.pop(); u >= 0; u = Que.pop())
    for (px = lst[u].get_first();
         px != NULL; px = lst[u].get_next())
      if (!Lev[*px])
      {
        Lev[*px] = Lev[u] + 1;
        Que.push(*px);
      }
  return Lev;
}
```

Выделение минимального остова

Пусть $G = (V, E)$ – связный неориентированный граф, содержащий $n = |V|$ вершин и $e = |E|$ ребер.

Остов (каркас) G – это некоторый связный суграф $G' = (V, E')$, $E' \subset E$, не содержащий циклов (дерево).

В общем случае можно выделить множество каркасов (например, если G – полный граф, то для него можно определить n^{n-2} различных каркасов).

Для любой пары вершин $i, j \in V$ в G' существует единственный соединяющий их путь.

Добавление к остову любого ребра всегда приводит к образованию цикла.

Выделение минимального остова

Любой остов связного графа G содержит ровно $n-1$ ребро.

Доказательство (по мат. индукции):

1. Остов включает 0 ребер при $n=1$ и 1 ребро при $n=2$.
2. Пусть остов связного графа с $k \geq 2$ вершинами содержит $k-1$ ребро. Если к графу будет добавлена новая вершина $\mathrm{k}+1$, то достаточно включить только одно ребро вида $(\mathrm{i}, k+1)$, $i \epsilon \{1, \ldots, k\}$, чтобы полученный граф стал связным. Добавление к графу еще одного ребра (любого) приведет к образованию цикла.

В общем случае, если граф G содержит $m \geq 1$ компонент связности, то для него можно выделить m каркасов (остовный лес), которые в сумме будут содержать $n-m$ ребер.

Выделение минимального остова

Пусть G – взвешенный граф (задана матрица весов ребер $c_{ij},\ i,j = \overline{0 \ldots n-1},\ c_{ij} = \infty,$ если $(i,j) \notin E$). В этом случае можно поставить задачу выделения минимального по весу остова W.

Правила выделения ребер минимального остова определяет

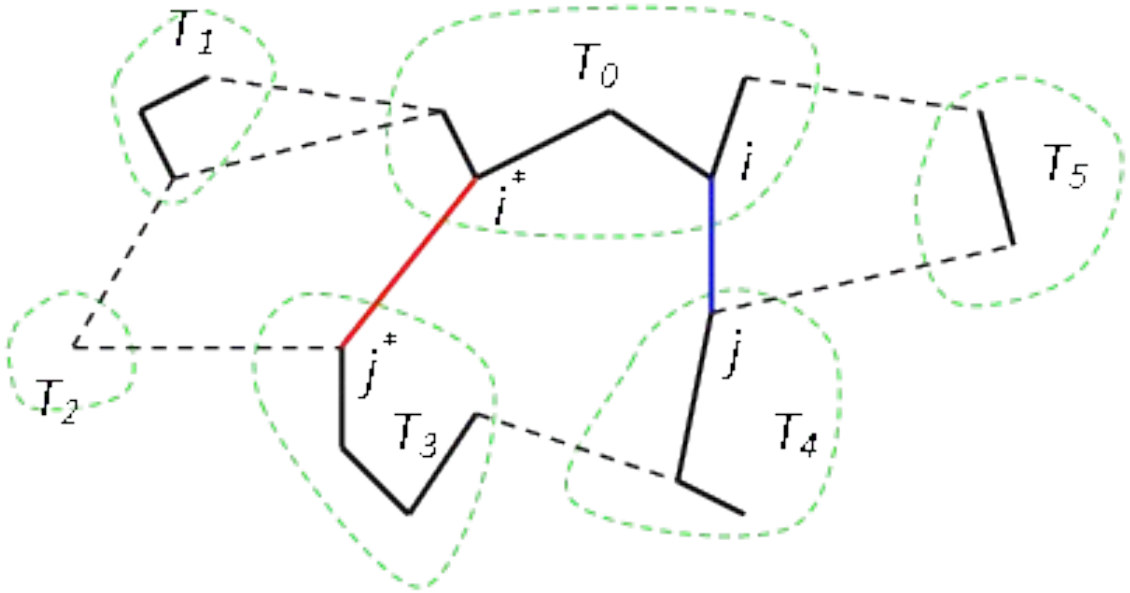
Лемма:

Пусть $T_0, T_1, \ldots, T_k$ – некоторые подграфы (поддеревья) W с попарно непересекающимися множествами вершин $V_0, V_1, \ldots, V_k,\ \cup V_i = V$.

Тогда ребро (i,j) с весом $c_{ij} = \min(c_{st}) : s \in T_0, t \notin T_0$ принадлежит минимальному остову.

Выделение минимального остова

Доказательство (от противного): пусть $(i, j) \notin W$ и есть такое ребро $(i^*, j^*) \in W$, что $i^* \in T_0, j^* \notin T_0$ и $c_{i^* j^*} > c_{ij}$.



В минимальном остове W есть пути из i^* в i и из j^* в j. Подграф T_0 связан с остальной частью графа ребром (i^*, j^*). Поэтому при добавлении (i, j) образуется цикл. Если из этого цикла удалить ребро (i^*, j^*), то будет получен новый остов, вес которого меньше, чем у W - **противоречие**.

Выделение минимального остова

Пусть в остов добавлено ребро (i, j), причем $i \in V_p, j \in V_q$. Тогда деревья T_p и T_q объединятся в одно, т.е. общее число построенных поддеревьев W уменьшится на 1.

Процесс построения W закончится, когда останется одно дерево, содержащее $n - 1$ ребро (или m отдельных деревьев, если исходный граф содержит m компонент).

В алгоритмах выделения минимального остова в качестве начальных используются n поддеревьев, содержащих по 1 вершине (и 0 ребер). Затем производится последовательный выбор минимальных по весу ребер, соединяющих текущие поддеревья, и объединение пар поддеревьев.

Алгоритм Прима

Данный алгоритм выгодно использовать, если граф задан матрицей весов и содержит много ребер. Будем считать, что если в графе отсутствует ребро (i, j), то соответствующий элемент матрицы весов $c_{ij} = MAX$.

В алгоритме производится постоянное расширение только одного поддерева (T_0). Вначале T_0 содержит только одну вершину, затем к T_0 последовательно добавляется по одному ребру и одной вершине, «ближайшей» к текущему поддереву.

Пусть на текущем шаге T_0 содержит k вершин. Поиск очередного ребра остова с прямым перебором всех ребер $(i, j), i \in T_0, j \notin T_0$ потребует $k(n-k)$ сравнений, и общая трудоемкость алгоритма составит $\sum_{k=1}^{n-1} k(n-k) = O(n^3)$.

Алгоритм Прима

Для снижения трудоемкости формируется дополнительный массив B длины n:

$B[j] = -1$, если вершина j уже включена в остов,

$B[j] = i \geq 0$, если i – ближайшая к j уже включенная в остов вершина.

За один просмотр массива B (n элементарных шагов) можно выделить очередное ребро минимального остова – это минимальное по весу ребро вида $(j, B[j])$, где $B[j] \geq 0$.

Пусть это ребро $(vm, B[vm])$ и его вес равен $wmin$. Тогда если $wmin < MAX$, то ребро действительно содержится в графе и его нужно добавить в остов (вместе с вершиной vm).

Алгоритм Прима

Если $wmin < MAX$, то ребро действительно содержится в графе и его нужно добавить в остов (вместе с вершиной vm).

После этого необходимо перестроить массив B (за один проход): для каждой пока не включенной в остов вершины j проверить, не будет ли ребро (j, vm) легче, чем $(j, B[j])$.

Если $wmin = MAX$, то это означает, что исходный граф несвязный, и было выделено одно из поддеревьев остова (для одной компоненты связности). Построение следующего поддерева можно начинать с любой вершины, входящей в следующую компоненту.

Метод WGraph для алгоритма Прима

```
void WGraph::get_span_tree() {
  double wmin;
  int i, j, vm, *B = new int[vernum];
  B[0] = -1;
  for (i = 1; i < vernum; i++) B[i] = 0;
  for (i = 1; i < vernum; i++) {
    wmin = MAX; vm = 0;
    for (j = 1; j < vernum; j++)
      if (B[j] != -1 && wmin > mat[j][B[j]])
      { vm = j; wmin = mat[j][B[j]]; }
    if (!vm) return;
    add_edge(vm, B[vm]); B[vm] = -1;
    for (j = 1; j < vernum; j++)
      if (B[j]!=-1 && mat[j][B[j]]>mat[j][vm])
        B[j] = vm;
  }
}
```

Алгоритм Крускала

Данный алгоритм выгодно использовать, если исходный граф содержит относительно немного ребер, которые лучше задавать массивом троек (i, j, c_{ij}).

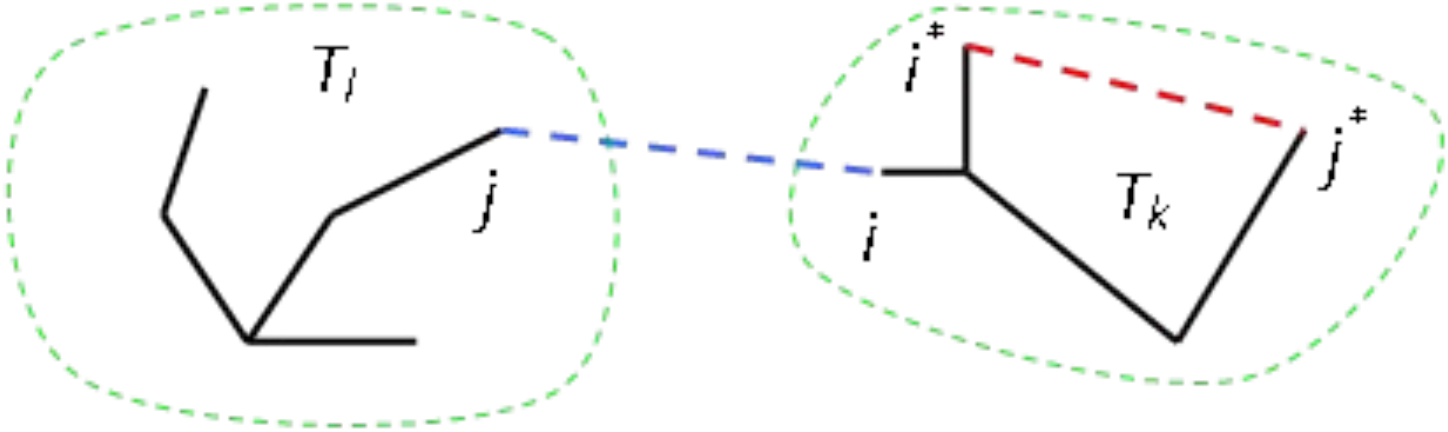
Все ребра сортируются по весу и последовательно проверяются, начиная от самого легкого: если очередное ребро не образует цикла в уже построенной части остова, то оно добавляется в остов, иначе отбрасывается.

Процесс продолжается, пока не будет получено $n - 1$ ребро (для связного графа), либо пока не будут просмотрены все ребра (для несвязного).

Сортировка требует $O(e \log e)$ элементарных шагов, поэтому если $e = O(n^2)$, то для выделения минимальных ребер выгоднее построить бинарную кучу.

Алгоритм Крускала

Для проверки, приводит ли добавление ребра к образованию цикла, формируются (и последовательно объединяются) множества связных вершин: вершины a и b принадлежат одному множеству, если существует путь из a в b, состоящий из уже выделенных ребер остова.



$i^*, j^* \in T_k$: ребро (i^*, j^*) образует цикл в T_k – недопустимое,

$i \in T_k, j \in T_l$: ребро (i, j) соединяет точки из разных множеств. (i, j) можно добавить к остову, при этом T_k и T_l объединятся в одно множество.

Быстрое объединение множеств

Множества вершин удобно представлять в виде деревьев принадлежности вершин со ссылкой от «сына» к «отцу» и хранить ссылки в целочисленном массиве A:

- начальные значения $A_i = i$ (отдельные корни деревьев),
- далее $A_i = i$ только для корней, иначе A_i – это «отец» i.

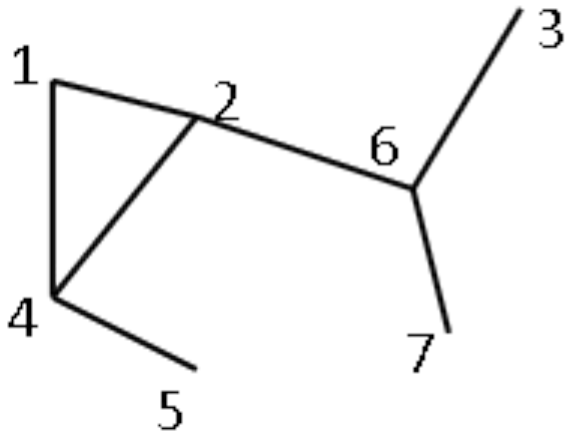
По ссылкам A_i можно пройти от вершины до корня, а корень однозначно определяет множество вершин.

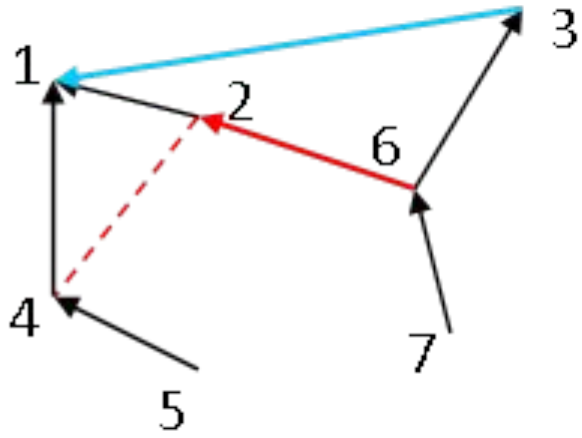
При проверке ребра (i, j) необходимо:
найти корни множеств, содержащих вершины i и j,

- если корень(i) = корень(j), то обе вершины принадлежат одному множеству, т.е. ребро (i, j) образует цикл,
- если корень(i) ≠ корень(j), то ребро добавляется в остов, а 2 множества объединяются путем формирования в A ссылки с одного корня на другой.

Быстрое объединение множеств

Пример построения деревьев принадлежности (порядок выбора ребер (1,2), (6,7), (4,5), (3,6), (1,4), (2,4), (2,6)):





	1	2	3	4	5	6	7
	1	2	3	4	5	6	7
	1	1	3	1	4	3	6
	1	1	1	1	4	3	6

Быстрое объединение множеств

Чтобы деревья при объединении не вырождались в линейный список, нужно меньшее дерево делать поддеревом большего.

Для этого нужен дополнительный массив весов B, в котором каждому корню приписывается либо число вершин, либо высота дерева (эти значения модифицируются при объединении множеств).

При указанных выше условиях дерево высоты h будет содержать не менее 2^{h-1} вершин (по матиндукции):

1. Для $h = 1$ выполняется.
2. Пусть объединяются деревья высоты h_1 и h_2, с числом вершин n_1 и n_2, $n_1 \geq 2^{h_1-1}$, $n_2 \geq 2^{h_2-1}$ и $n_1 \geq n_2$.

Если $h_1 > h_2$, то объединенное дерево будет также иметь высоту h_1. Если $h_1 = h_2 = h$, то высота нового дерева будет $h + 1$, а число вершин $n_1 + n_2 \geq 2^{h-1} + 2^{h-1} = 2^h$.

Быстрое объединение множеств

Если множество содержит m вершин, то его корень можно найти не более, чем за $\log m$ шагов.

Трудоемкость выделения всех ребер остова (после сортировки ребер графа) не превышает $O(e \log n)$.

Еще более эффективной будет проверка со сжатием путей, когда при поиске корня ссылки на «отца» заменяются на ссылки прямо на корень множества для всех пройденных вершин. Тогда трудоемкость выделения остова $O(eG(n))$, где $G(n)$ – «обратная» к частному случаю функции Аккермана $F(n)$: $F(0) = 1,\ F(n) = 2^{F(n-1)}, n > 0$.

$G(n) = \min(k) :\ F(k) \geq n$ (это фактически $const < 5$).

	0	1	2	3	4	5
	1	2	4	16	65536	
	0	0	1	2	2	3

Алгоритм Крускала для массива ребер

Алгоритм Крускала приводится в виде отдельной функции **span_tree**, которая выделяет минимальный остов графа, заданного массивом длины **e** взвешенных ребер **R** (число вершин равно **n**). Остов сохраняется в массиве **W** длины **n-1**, который также содержит взвешенные ребра и должен быть выделен заранее. **span_tree** выделяет остов и возвращает число его ребер (<= n-1).

Взвешенное ребро представляется структурой

```
struct Edge
{
   int a, b;          // номера двух вершин ребра
   double weight;     // вес ребра (для сортировки)
}
```

Функция **sort_edges** производит сортировку массива **R** по возрастанию весов ребер.

Алгоритм Крускала для массива ребер

```
int span_tree(Edge *R, int e, int n, Edge *W)
{
  int k = 0, ra, rb, i, *A, *B;
  A = new int[n]; B = new int [n];
  sort_edges(R, e);
  for (i=0; i < n; i++) { A[i] = i; B[i] = 1; }
  for (i = 0; k < n-1 && i < e; i++) {
    for (ra = R[i].a; ra != A[ra]; ra = A[ra]);
    for (rb = R[i].b; rb != A[rb]; rb = A[rb]);
    if (ra == rb) continue;
    W[k++] = R[i];
    if (B[ra] >= B[rb])
    { A[rb] = ra; B[ra] += B[rb]; }
    else { A[ra] = rb; B[rb] += B[ra]; }
  } return k;
}
```

Жадные алгоритмы

Алгоритмы Прима и Крускала – **жадные**: на каждом их шаге делается локально оптимальный (жадный) выбор, который никогда не отменяется на последующих шагах.

Жадный алгоритм можно использовать, если для задачи выполняются 2 условия:

- оптимальное решение задачи содержит в себе оптимальные решения подзадач (свойство оптимальности подзадач);
- последовательность локально оптимальных выборов дает глобально оптимальное решение (т.е. жадный выбор на каждом шаге не закрывает путь к оптимальному решению).