ОБХОДЫ И КАРКАСЫ ГРАФОВ

ЛЕКЦИИ 16-17

План лекции

- Обход всех вершин графа
  - Методы поиска в глубину и в ширину
- Построение каркаса графа
  - Алгоритмы Краскала и Прима-Краскала

Поиск в глубину (Depth-First Search, DFS)

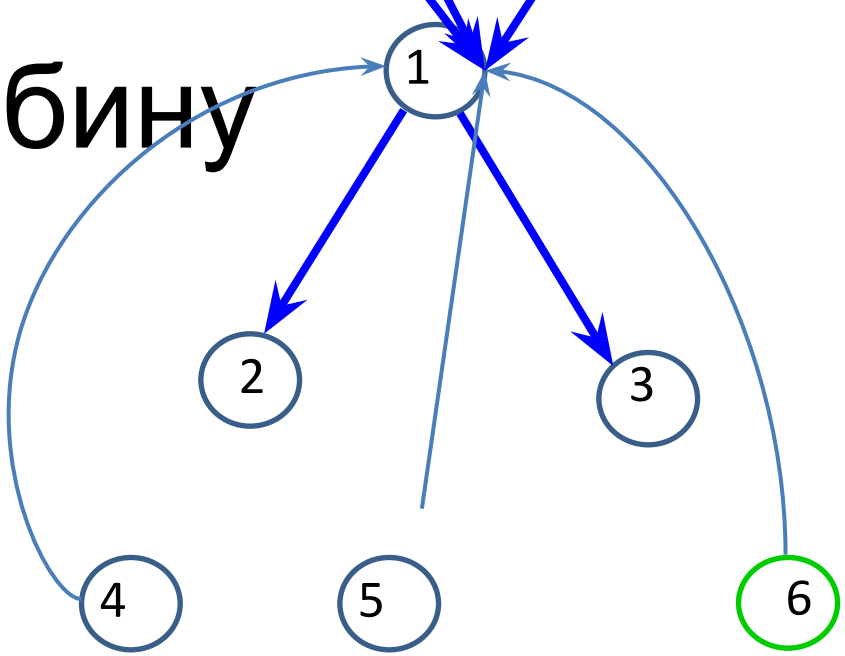
- Один из способов нумерации вершин произвольного графа
- Алгоритмы обработки графов
  - Топологическая сортировка
  - Поиск 1-, 2-, 3-связнных компонент
  - Поиск мостов
  - Поиск сильно связанных компонент
  - Проверка планарности
- Компиляция программ, комбинаторный поиск, компьютерная алгебра

Поиск в глубину (Depth-First Search, DFS)

- Поиск в глубину вычисляет для каждой вершины u три номера
  - П[u] предшественика вершины u при поиске в глубину
  - d[u] время обнаружения вершины u
  - f[u] время окончания обработки вершины u
- Схема алгоритма
  - В начале все вершины непройденные и становятся пройденными в процессе поиска в глубину
  - Для каждой непройденной вершины u
    - Для каждой непройденной вершины v, смежной с вершиной u
      - Поиск в глубину (v)

Поиск в глубину

```
DFS(G) {
    for u ∈ V {
        color[u] = белый;
        П[u] = u;
    }
    time = 0;
    for u ∈ V
        if(color[u] == белый)
            Поиск(u);
}
```



Как соотносятся цвета на рисунке и в описании? Какой цвет лишний?

```
Поиск(u) {
    color[u] = серый;
    d[u]=time++;
    for (u, v) ∈ E
        if(color[v] == белый) {
            П[v] = u;
            Поиск(v);
        }
    color[u] = черный;
    f[u]=time++;
}
```

Подграф предшествования

- Для каждого белого соседа v вершины u
  - П[v] = u
  - Поиск(v)
- Красим u в черный цвет, возвращаемся в П[u] и ищем другого белого соседа П[u], и т.д.
- П – представление списком прямых предков дерева (леса) поиска в глубину графа G

Подграф предшествования $E_П$ графа $G_П = (V, E_П)$:

$$E_П = \{ (П[v], v) \mid П[v] \neq v \}$$

Если G является связным, то $G_П$ называются каркасом (остовным деревом) графа G

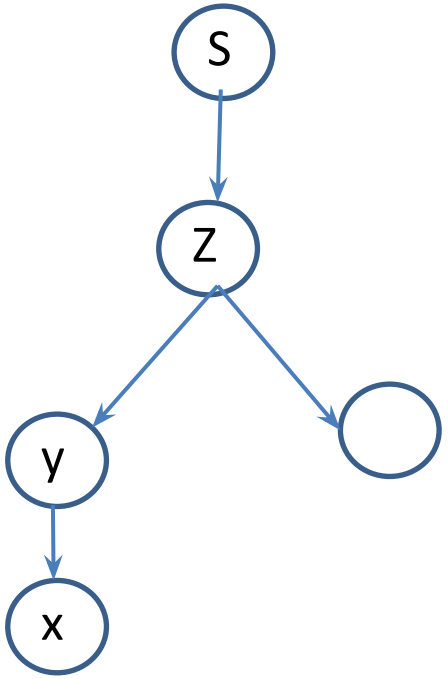
Сложность поиска в глубину по времени

- Для простоты считаем, что время работы пропорционально числу операций
  - присваивания, сравнения, доступ к элементам массивов занимают единицу времени
- Время поиска в глубину = время работы DFS() + ∑ время работы Поиск(u)
  - для каждой вершины Поиск() исполняется 1 раз – почему?
- Время работы DFS() = O(|V|)
- Время работы Поиск(u) = O(|E(u)|)
- Время поиска в глубину = O(|V|) + ∑ O(|E(u)|) = O(|V|+|E|)

Свойства поиска в глубину

Времена обнаружения и окончания обработки вершин d[u] и f[u] образуют правильную скобочную структуру

```
1 2 3 4  5 6 7  8 9 10
(s(z(y(xx)y) (ww)z)s)
```



Теорема о свойствах поиска в глубину

Номера d[u], f[u], d[v], f[v] любых двух вершин *u* и *v* графа G, полученные поиском в глубину, удовлетворяют одному из условий:

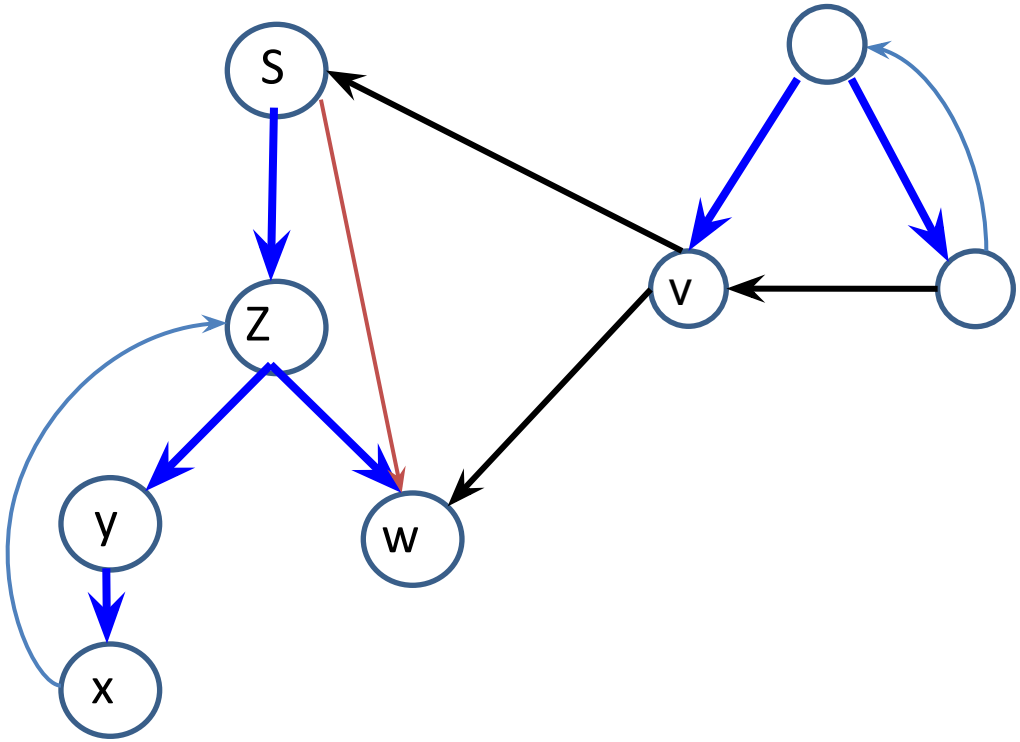
1) Отрезки [d[*u*],f[*u*]] и [d[*v*],f[*v*]] не пересекаются

2) Отрезок [d[*u*],f[*u*]] $\subseteq$ [d[*v*],f[*v*]] и *u* есть потомок *v* в графе поиска в глубину графа G

3) Отрезок [d[*v*],f[*v*]] $\subseteq$ [d[*u*],f[*u*]] и *v* есть потомок *u* в графе поиска в глубину графа G

Классификация рёбер графа при поиске в глубину

- Древесные рёбра
  - входят в граф предшествования
- Прямые рёбра
  - соединяют вершину с её потомком, но не входят в граф предшествования
- Обратные рёбра
  - соединяют вершину с её предком в графе предшествования
- Перекрёстные рёбра
  - все остальные



Метод поиска в ширину (BFS, Breadth-first search)

- Один из способов нумерации вершин произвольного графа
- Алгоритмы обработки графов
  - Поиск кратчайших путей
  - Вычисление максимального потока
  - Проверка связности
- Компьютерное моделирование, графические интерфейсы, анализ транспортных, электрических и т.п. цепей и сетей

Метод поиска в ширину (BFS, Breadth-first search)

- Пусть дан граф G и выбрана некоторая его вершина s
- Поиск в ширину вычисляет для каждой вершины u два номера
  - П[u] предшественика вершины u при поиске в ширину
  - d[u] кратчайшее расстояние от s до u

- Схема алгоритма
  - Шаг 1: d[s] = 0
  - Шаг n: обрабатываем все вершины на расстоянии n-1 от s
    - Каждого соседа v вершины u с пометкой d[u] = *n*-1 нумеруем П[v] = u и d[v] = n

Алгоритм BFS

```
BFS (G, s) {
    1     for u ∈ V && u != s {
    2           Π[u] = u;
    3           d[u] = ∞;
    4     }
    5     d[s] = 0, Π[s]  =  s;
    6     put(s, Q);
    7     while (! empty(Q)) {
    8           u = get(Q);
    9           for v ∈ E(u) {
   10                 if (d[v] > d[u]+1) {
   11                       Π[v]= u;
   12                       d[v]= d[u]+1;
   13                       put(v,Q);
   14           }}
   15     }
}
```

Свойства поиска в ширину -- кратчайшие пути

Пусть $\delta(s,v)$ минимальная длина пути из вершины s в вершину v

Лемма 1. Пусть s – произвольная вершина графа, (u, v) – ребро.

Тогда $\delta(s, v) \leq \delta(s, u) + 1$.

Доказательство. Если u достижима за k шагов, то и v достижима не более чем за $k + 1$ шагов.

Лемма 2. Если s != v, то $\delta(s, v) = \delta(s, u) + 1$ для некоторого соседа u вершины v

Доказательство. Рассмотрим кратчайший путь из s в v. Его длина $\delta(s, v)$. Возьмем вершину u, лежащую на этом пути непосредственно перед v. Убедимся, что до нее расстояние на единицу меньше. У нас есть ведущий в нее путь длины $\delta(s, v) – 1$. Более короткого пути не может быть по лемме 1.

Теорема о поиске в ширину

1. Для любого целого $k >= 0$ найдётся шаг BFS, когда очередь Q состоит из вершин, находящихся на расстоянии k от вершины s
2. Если $d[v] != \infty$, то $\delta(s, \Pi[v]) + 1 = \delta(s, v) = d[v]$, и в графе есть ребро $(\Pi[v], v)$
3. Если $d[v] == \infty$, то $\Pi[v] = v$

Доказательство

Если $k=0$, то все условия выполняются в строке 6

Пусть $k > 0$.

Дождёмся, когда выполнятся условия для $k-1$ – это возможно по предположению индукции.

Очередь Q состоит из вершин, находящихся на расстоянии k-1 от вершины s.

Условия выполнятся для k, когда из очереди будет изъята последняя вершина на расстоянии k-1 от вершины s.

Из п. 1 следуют п. 2 и п. 3, т.к. в очередь добавляются соседи вершин, (k-1)-удалённых от s.

Чем ограничены сверху длины кратчайших путей в G?

Когда в очереди находятся вершины, удалённые от s на |V|?

Печать кратчайших путей

```
void Print_Path(const int parent[], int s, int v)
{
  if (s == v) printf("%d ", s);
      else if (parent[v] == v)
            printf("No Path");
      else {
            Print_Path(parent, s, parent[v]);
            printf("%d", v);
      }
}
```

Каркасы графа

G(V,E) -- связный неориентированный граф
Веса рёбер w : E --> R^+ = [0,∞)

Остовное дерево или каркас графа – это подграф G, который содержит все вершины графа и является деревом

Минимальным каркасом называется такой каркас, сумма весов ребер которого минимальна

Алгоритм Кра́скала

- Josef Bernard Kruskal Jr. 1928-2010
- Алгоритм Краскала 1956 год
  - Joseph. B. Kruskal: On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem. In: Proceedings of the American Mathematical Society, Vol 7, No. 1 (Feb, 1956), pp. 48–50
- Поиск минимального остовного дерева графа

Алгоритм Краскала – схема

- Строим остовный лес K = (V, B) для данного графа G = (V, E) с весами ребер w
- Сортируем ребра E по возрастанию w
- B = ∅; // начинаем с пустого множества ребер
- В порядке возрастания весов w(e) ребер e графа G выполняем
  - если добавление e к K не образует цикл в K, то добавляем e в K
  - если добавление e к K образует цикл в K, то не добавляем e в K

Проверка того, замыкает ли ребро цикл

Делим множество вершин V на компоненты связности Wi -- максимальные по включению попарно непересекающиеся подмножества, состоящие из вершин, связанных лесом K

$V = W0 \cup W1 \cup \ldots \cup Wx$

$Wi \cap Wj = \varnothing$

для любых x и y из Wi x и y связаны путём в K

если x и y связаны путём в K, то x и y принадлежат одному Wi

Алгоритм Краскала – псевдо код

Вход

Неориентированный граф G = (V, E) с весами ребер w

Выход

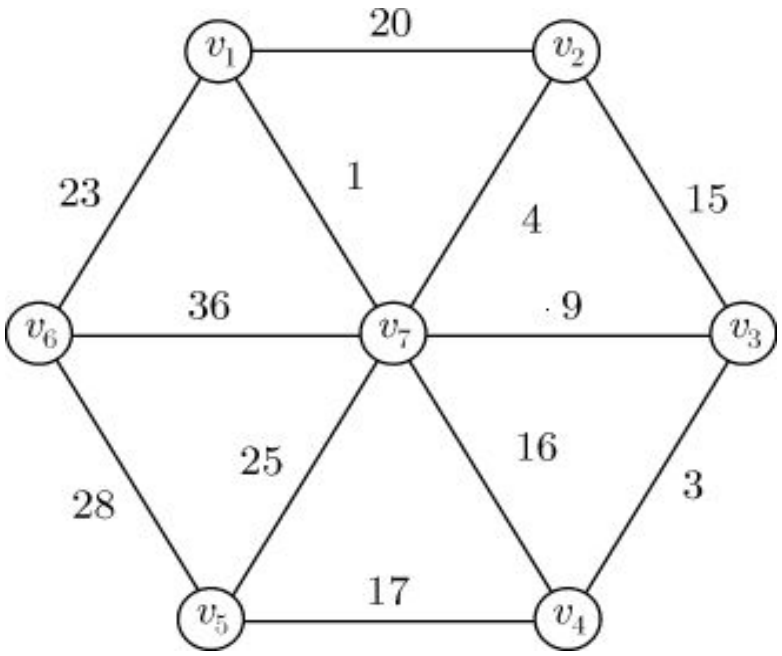
Остовное дерево K = (V, B) наименьшего веса для графа G

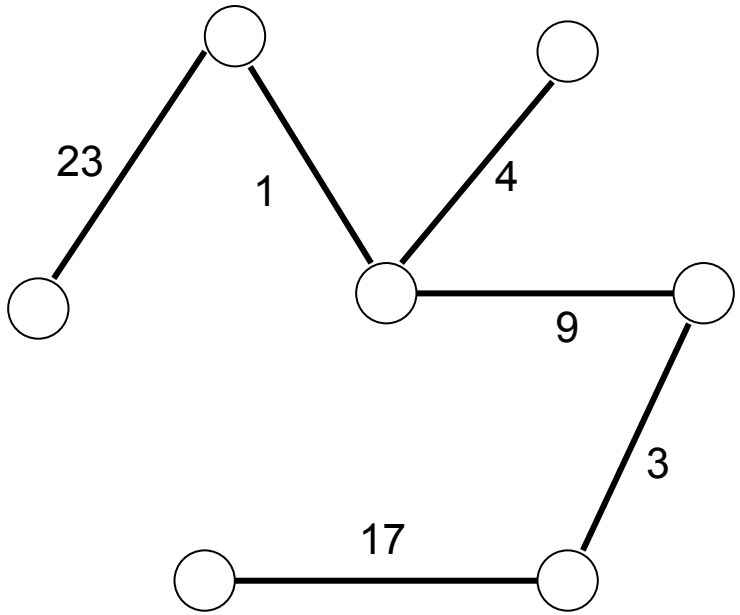
```
B = ∅;      // начинаем с пустого каркаса
W = { {v} | v ∈V } ; // каждая вершина v в своей компоненте связности
Q = очередь(сортировать по возрастанию весов(E));
while (! empty(Q)) {
    (x, y) = get(Q); // ребро наименьшей стоимости
    Wx = find(x, W); // найти компоненты связности содержашие x и y
    Wy = find(y, W);
    if (Wx != Wy) { // (x, y) не замыкает цикл в K
        W = W - Wx - Wy + (Wx ∪ Wy);
        K = K + (x, y);
    }
}
```

Число операций в алгоритме Краскала

- Сортировка рёбер = O(|E|*log|E|)
- Цикл = O(|E| * число операций в find и ∪)
  - Зависит от реализации операций find и ∪
  - Для системы непересекающихся множеств find и ∪ занимают практически O(1) действий

Пример





Система непересекающихся множеств (СНМ)

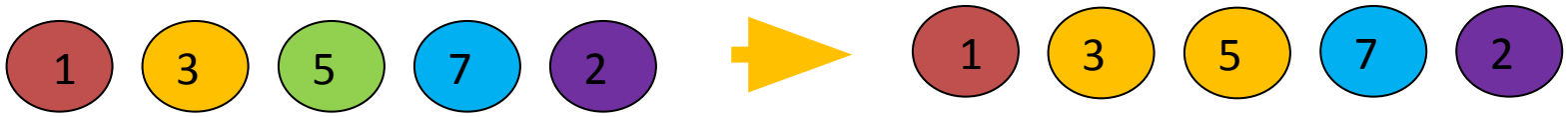
- Разбиение конечного множества носителя на попарно непересекающиеся подмножества
- Операции СНМ (носитель фиксирован)
  - InitSet(S)
    - Создает СНМ, состоящее из одноэлементных подмножеств носителя
  - FindSet(S, X)
    - Возвращает главный элемент множества, которому принадлежит X в СНМ S
  - Join(S, X, Y)
    - Объединяет множества, которым принадлежат элементы X и Y в СНМ S, и возвращает главный элемент нового множества

Реализации СНМ

1) Простая реализация

для каждого элемента храним его "цвет"

FindSet (S, X) – $O(1)$; Join (S, X, Y) – $O(n)$



2) Реализация списком

FindSet (S, X) - $O(n)$; Join (S, X, Y) - $O(1)$

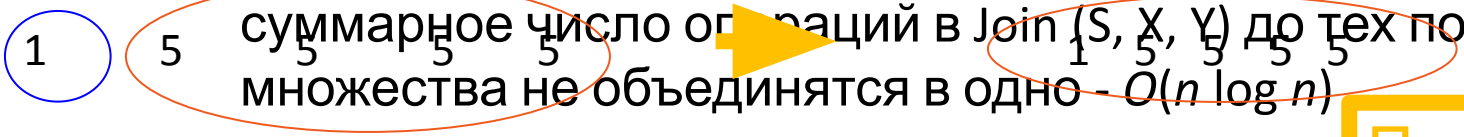
3) Весовая эвристика (улучшение простой реализации)

для каждого элемента храним число элементов этого цвета

перекрашиваем элементы из множества меньшей мощности

FindSet (S, X) - $O(1)$;

суммарное число операций в Join (S, X, Y) до тех пор, пока все множества не объединятся в одно - $O(n \log n)$



Почему n log(n)?

Реализации СНМ

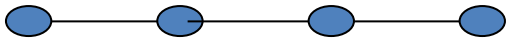
4) Реализация с помощью дерева

для каждого элемента храним его предка

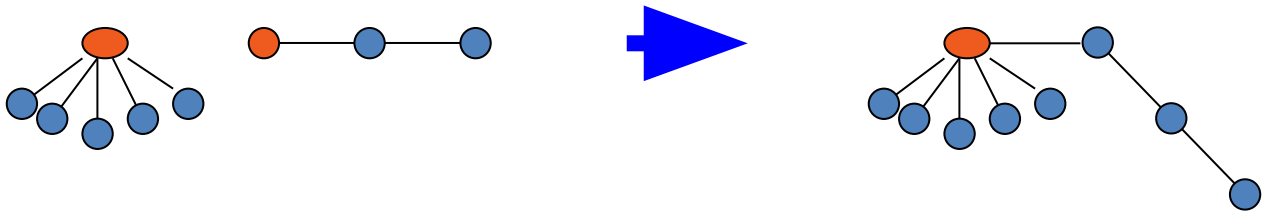
главным элементом множества является корень дерева

FindSet (S, X) - $O(n)$; Join (S, X, Y) - $O(1)$

Худший случай, когда дерево вытягивается в список



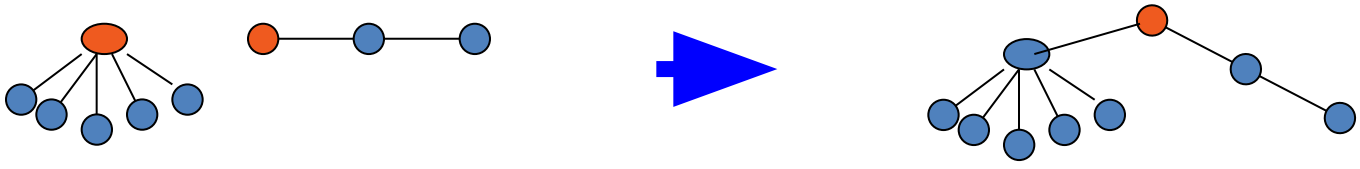
Глубина дерева может расти слишком быстро



Реализации СНМ – объединение по рангу

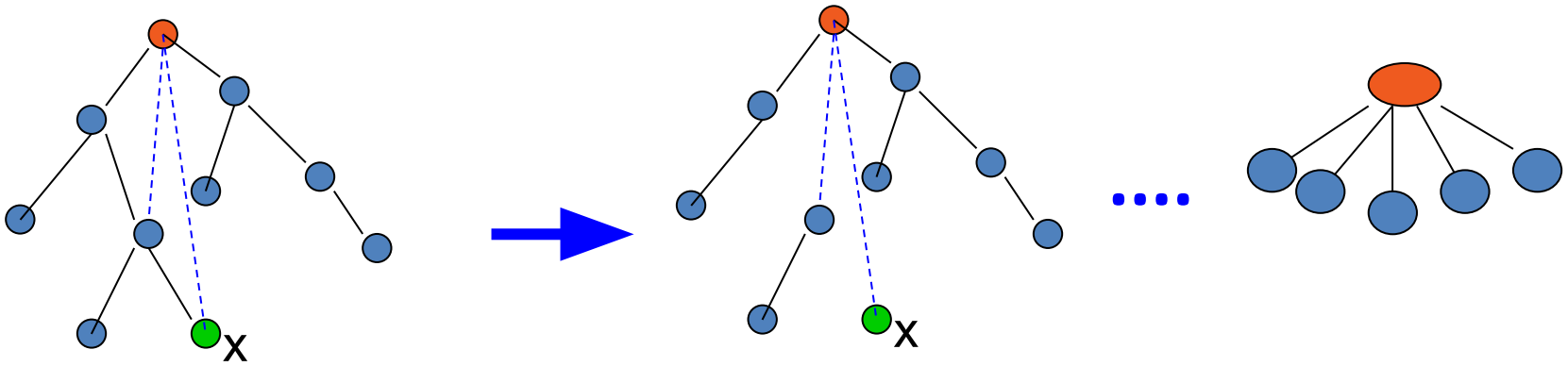
5) Эвристика объединением по рангу

- Храним ранг каждого главного элемента – высоту его поддерева
- Ранг СНМ – max рангов главных элементов
- Объединяем деревья рангов < ранг СНМ -> ранг СНМ не меняется
- Объединяем деревья одинакового ранга == ранг СНМ -> ранг СНМ увеличивается на единицу.
- FindSet (S, X) - $O(\log n)$; Union (S, X, Y) - $O(1)$



Реализации СНМ – объединение по рангу + сжатие путей

- 6) Эвристика сжатия путей:
- FindSet (S, X) делает все элементы на пути от X до главного элемента для X непосредственными потомками этого главного элемента – см. рисунок ниже
- FindSet (S, X) - *O*(обратная функция Аккермана)
  - для 64-битных целых чисел обратная функция Аккермана ≤ 4
  - определение на следующем слайде
- Union (S, X, Y) - *O*(1)



Функция Аккермана

- Определение – один из вариантов
  - A [0, j] = j+1
  - A [i, 0] = A[i-1, 1] , если i > 0
  - A [i, j] = A[i-1, A[i, j -1]], если i, j > 0
- Обратная функция Аккермана
  - $f^{-1}[x] = \min \{ k >= 1 \mid A[k, k] >= x \}$
- Wikipedia
  - сложность A(1,n) O(n)
  - сложность A(2,n) O(n^2)
  - сложность A(3,n) O(4^n)
  - A[4, 4] ~= 2^(2^(10^19729))

Версия 1

```
#define SNM_MAX
static int p[SNM_MAX], rank[SNM_MAX];

void initset()
{	int i;
	for (i = 0; i < SNM_MAX; ++i) p[i] = i, rank[i] = 0;
}
int find_set(int x)
{	if (x == p[x]) return x;
	return p[x] = find_set(p[x]);
}
void join(int x, int y)
{	x = find_set(x);
	y = find_set(y);
	if (rank[x] > rank[y])  p[y] = x;
	else {		p[x] = y;
		if (rank[x] == rank[y])
		++rank[y];
	}
}
```

Версия 2

```
#define SNM_MAX 1000
static int parent[SNM_MAX], rank[SNM_MAX];

void init_set()
{	int i;
	for (i = 0; i < SNM_MAX; ++i) parent[i] = i, rank[i] = 0;
}

int find_set(int x)
{	if (x == parent[x]) return x;
	return parent[x] = find_set(parent[x]);
}

void join(int x, int y)
{	x = find_set(x);
	y = find_set(y);
	if (rank[x] > rank[y])  parent[y] = x;
	else {	parent[x] = y;
		if (rank[x] == rank[y])
		++rank[y];
	}
}
```

Версия 3

```
#define SNM_MAX 1000
static int parent[SNM_MAX], rank[SNM_MAX];
// СНМ, состоящее из одноэлементных подмножеств носителя
void init_set()
{       int i;
        for (i = 0; i < SNM_MAX; ++i) parent[i] = i, rank[i] = 0;
}
// Главный элемент множества, которому принадлежит X
int find_set(int x)
{       if (x == parent[x]) return x;
        return parent[x] = find_set(parent[x]); // сжатие
}
// Объединяет множества, которым принадлежат элементы X и Y
// Возвращает главный элемент нового множества
void join(int x, int y)
{       x = find_set(x);
        y = find_set(y);
        if (rank[x] > rank[y])  parent[y] = x;
        else {parent[x] = y;
                if (rank[x] == rank[y])
                ++rank[y];
        }
}
```

Версия 4

```
#define SNM_MAX 1000
static int parent[SNM_MAX], rank[SNM_MAX];
// СНМ, состоящее из одноэлементных подмножеств носителя
void init_set()
{     int i;
      for (i = 0; i < SNM_MAX; ++i) parent[i] = i, rank[i] = 0;
}
// Главный элемент множества, которому принадлежит X
int find_set(int x)
{     if (x == parent[x]) return x;
      return parent[x] = find_set(parent[x]); // сжатие
}
// Объединяет множества, которым принадлежат элементы X и Y
// Возвращает главный элемент нового множества
int join(int x, int y)
{     x = find_set(x);
      y = find_set(y);
      if (rank[x] > rank[y])  return parent[y] = x;
      else {      parent[x] = y;
            if (rank[x] == rank[y]) ++rank[y];
            return  y;
      }
}
```

Версия 5

```
#define SNM_MAX 1000
struct SNM {int parent[SNM_MAX], rank[SNM_MAX];};
// СНМ, состоящее из одноэлементных подмножеств носителя
void init_set(struct SNM *S)
{       int i;
        for (i = 0; i < SNM_MAX; ++i) S->parent[i] = i, S->rank[i] = 0;
}
// Главный элемент множества, которому принадлежит X
int find_set(struct SNM *S, int x)
{       if (x == S->parent[x]) return x;
        return S->parent[x] = find_set(S->parent[x]); // сжатие
}
// Объединяет множества, которым принадлежат элементы X и Y
// Возвращает главный элемент нового множества
int join(struct SNM *S, int x, int y)
{       x = find_set(x);
        y = find_set(y);
        if (S->rank[x] > S->rank[y]) return S->parent[y] = x;
        else {S->parent[x] = y;
                if (S->rank[x] == S->rank[y]) ++ (S->rank[y]);
                return  y;
        }
}
```

Версия 6

```
#define SNM_MAX 1000
struct SNM {int parent[SNM_MAX], rank[SNM_MAX];};
// СНМ, состоящее из одноэлементных подмножеств носителя
void init_set(struct SNM *S)
{       int i;
        for (i = 0; i < SNM_MAX; ++i) S->parent[i] = i, S->rank[i] = 0;
}
// Главный элемент множества, которому принадлежит X
int find_set(struct SNM *S, int x)
{       if (x == S->parent[x]) return x;
        return S->parent[x] = find_set(S, S->parent[x]); // сжатие
}
// Объединяет множества, которым принадлежат элементы X и Y
// Возвращает главный элемент нового множества
int join(struct SNM *S, int x, int y)
{       x = find_set(S, x);
        y = find_set(S, y);
        if (S->rank[x] > S->rank[y]) return S->parent[y] = x;
        else {  S->parent[x] = y;
                if (S->rank[x] == S->rank[y]) ++ (S->rank[y]);
                return  y;
        }
}
```

Алгоритм Прима-Краскала

- Robert Clay Prim 1921

- Алгоритм Прима (иногда Прима-Краскала)
  - R. C. Prim: Shortest connection networks and some generalizations. In: Bell System Technical Journal, 36 (1957), pp. 1389–1401
  - Похожие алгоритмы предложены Войцехом Ярником (1930) и Дейкстрой (1959)

- Построение минимального каркаса связного взвешенного графа

Алгоритм Прима-Краскала -- схема

1. Выбираем произвольную вершину s -- корень остовного дерева;
2. До тех пор пока в дерево не добавлены все вершины
   /* Находим минимальное расстояние от дерева до вершин, которые не включены в дерево */
   1. найти вершину u, расстояние от дерева до которой минимально
   2. добавить u к дереву (красим в синий цвет)
   3. если расстояние до какой-либо вершины от u меньше текущего расстояния s от дерева, то в s записывается новое расстояние

Алгоритм Прима-Краскала -- схема

Вход
Неориентированный граф G = (V, E) с весами ребер w
Выход
Остовное дерево K = (V, B) наименьшего веса для графа G

красные = {s}; // начинаем с одной вершины s без ребер
d[x] = w[s][x]; // расстояние от x до ближайшей вершины в каркасе
while не все вершины красные {
x* = min { d[x] | x ∉ красные }; // самое короткое ребро
красные = красные U {x*}; // красим...
d[y] = min (d[y], w[x*][y]) для y ∉ красные
}

Число операций в алгоритме Прима

- O(|V|* (число операций для поиска min + число операций для обновления d))

- В худшем случае O(|V|*|V|)
  - Зависит от реализации поиска min и обновления d

Алгоритма Прима-Краскала С

```
void mst(int G[], int N, int parent[]) // N -- число вершин в графе
{
	int *d = calloc(sizeof(*d), 2*N), *red = d+N, i, j;
	for (i = 0; i < N; ++i) parent[i] = i, d[i] = G[0*N+i];
	d[0] = 0; red[0] = 1; // Вершина 0 -- корень каркаса
	for (i = 0; i < N; ++i) {
		int jmin = -1;
		for (j = 0; j < N; ++j)
			if (!red[j] && (jmin == -1 || d[j] <= d[jmin])) jmin = j;
		if (jmin == -1) break; // Нет достижимых вершины вне дерева
		red[jmin] = 1; // Включаем в дерево
		for (j = 0; j < N; ++j) // Обновляем расстояния до соседей jmin
			if (!red[j] && d[jmin]+G[j*N+jmin] < d[j]) {
				d[j] = d[jmin]+G[j*N+jmin];
				parent[j] = jmin;
			}}
}
```

Алгоритма Прима-Краскала С

```
void mst(const int G[], int N, int parent[]) // N -- число вершин в графе
{
	int *d = calloc(sizeof(*d), 2*N), *red = d+N, i;
	if (d == 0) return;
	for (i = 0; i < N; ++i) parent[i] = i, d[i] = G[0*N+i];
	d[0] = 0; red[0] = 1; // Вершина 0 -- корень каркаса
	for (i = 0; i < N; ++i) {
		int jmin = -1, j;
		for (j = 0; j < N; ++j)
			if (!red[j] && (jmin == -1 || d[j] <= d[jmin])) jmin = j;
		if (jmin == -1) break; // Нет достижимых вершин вне дерева
		red[jmin] = 1; // Включаем в дерево
		for (j = 0; j < N; ++j) // Обновляем расстояния до соседей jmin
			if (!red[j] && d[jmin]+G[j*N+jmin] < d[j]) {
				d[j] = d[jmin]+G[j*N+jmin];
				parent[j] = jmin;
			}}
	free(d);
}
```

Как использовать mst для проверки связности графа?

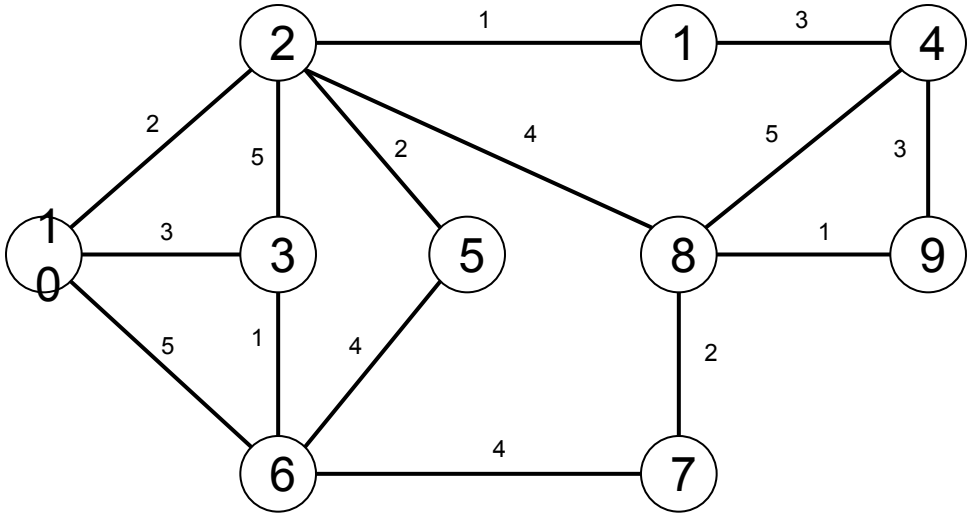
Доказательство корректности алгоритмов построения каркаса

- Вершины в построенной части каркаса красные
- Остальные вершины синие
- Срез – множество ребер, соединяющих красные и синие вершины
- На каждом шаге в каркас обязательно включается одно ребро из текущего среза
  - иначе получится несвязный граф, а не дерево
- Если на одном из шагов включить ребро e != e_{min}, то получится каркас, вес которого можно уменьшить
  - удалим e и добавим e_{min}

Заключение

- Обход всех вершин графа
  - Методы поиска в глубину и в ширину
- Построение каркаса графа
  - Алгоритмы Краскала и Прима-Краскала

Запускаем алгоритм обхода графа, начиная с произвольной вершины.
В качестве контейнера выбираем очередь с приоритетами. Приоритет – текущая величина найденного расстояния до уже построенной части остовного дерева.
Релаксации подвергаются прямые и обратные ребра.



n	1	2	3	4	5	6	7	8	9	10
π		1	10	1	2	3	8	9	4	2
d	0	1	3	3	2	1	2	1	3	2

В результате работы получаем список ребер остовного дерева вместе с весами

Реализация за $O(M \log N + N^2)$

Отсортируем все рёбра в списках смежности каждой вершины по увеличению
веса – $O(M \log N))$.
Для каждой вершины заведем указатель, указывающий на первое доступное
ребро в её списке смежности. Изначально все указатели указывают на начала
списков.
На i-ой итерации перебираем все вершины, и выбираем наименьшее по весу
ребро среди доступных. Поскольку всё рёбра уже отсортированы по весу, а
указатели указывают на первые доступные рёбра, то выбор наименьшего
ребра осуществится за $O(N)$.
После этого обновляем указатели (сдвигаем вправо), т.к. некоторые из них
указывают на ставшие недоступными рёбра (оба конца которых оказались
внутри дерева).
На поддержание работы указателей требуется $O(M)$ действий.

Алгоритм Прима (другой алгоритм)

На каждом шаге вычеркиваем из графа дугу максимальной стоимости с тем условием, что она не разрывает граф на две или более компоненты связности, т.е. после удаления дуги граф должен оставаться связным.

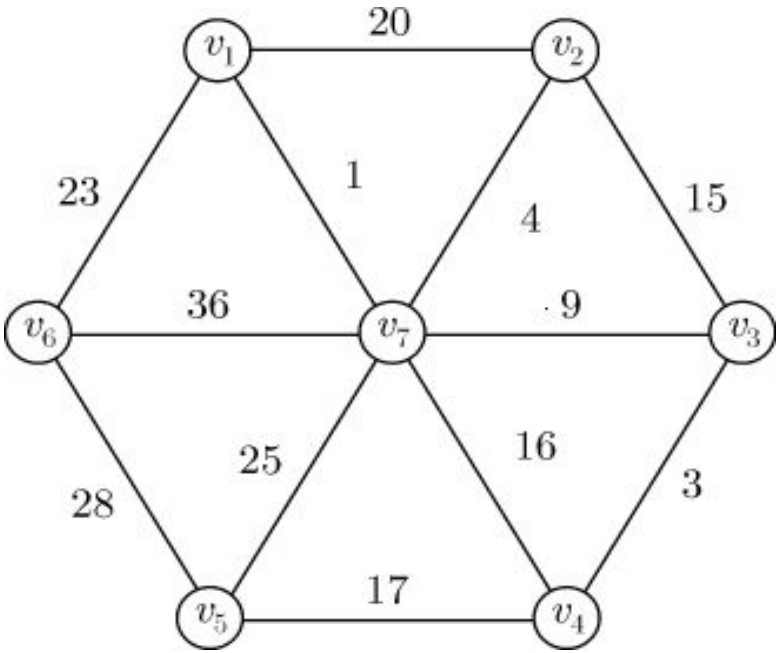
Для того, чтобы определить, остался ли граф связным, достаточно запустить поиск в глубину от одной из вершин, связанных с удаленной дугой.

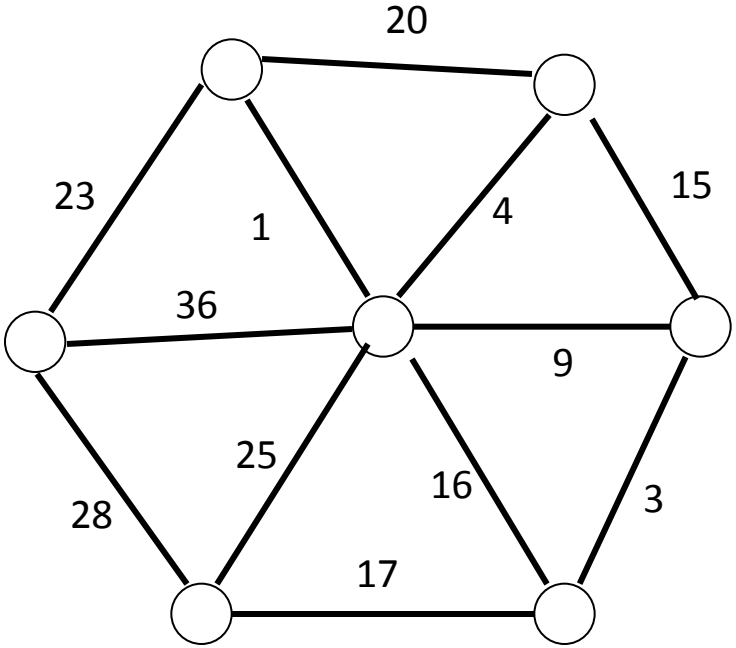
Условие окончания алгоритма?

Например, пока количество ребер больше либо равно количеству вершин, нужно продолжать,

иначе – остановиться.

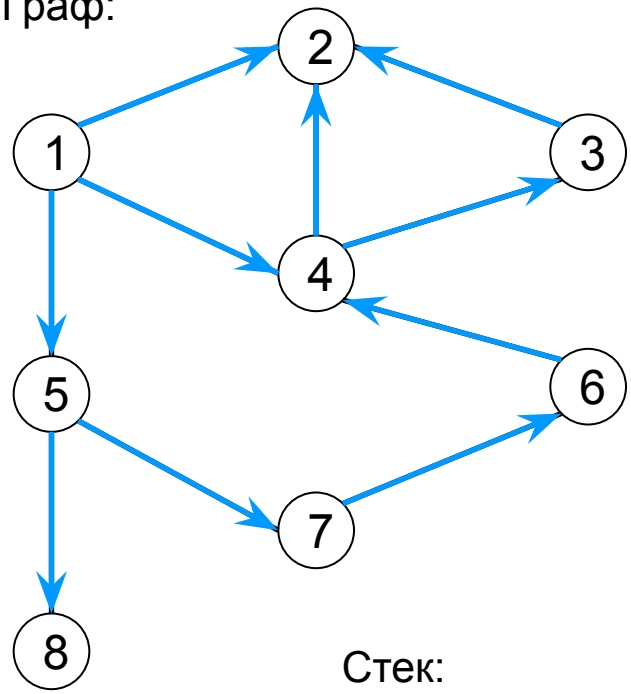
Пример


v_1
v_2
v_3
v_4
v_5
v_6
v_7
20
23
1
4
15
36
9
25
16
28
3
17

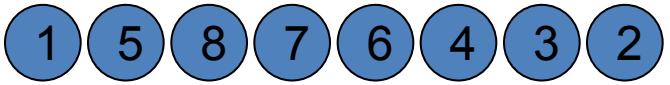

20
23
1
4
15
36
9
25
16
28
3
17

Использование стека для обхода графа

Граф:



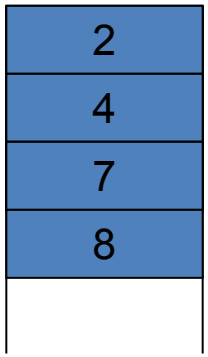
Если в качестве промежуточной структуры хранения при обходе использовать стек, то получим обход в глубину.



Можно также получить дерево обхода в глубину, если отмечать каждую прямую или обратную дугу.

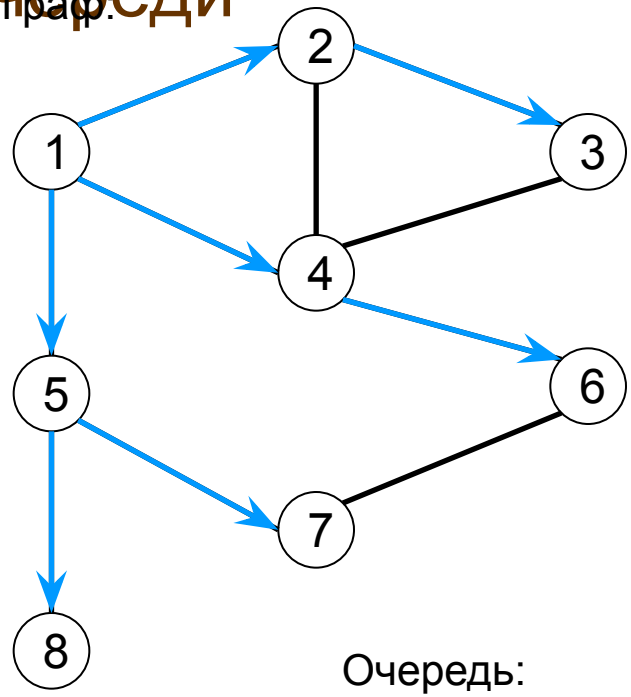
1	2	3	4	5	6	7	8
	3	4	6	1	7	5	5

Стек:



Использование очереди

Граф:

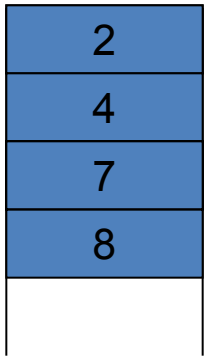


В качестве промежуточной структуры хранения при обходе в ширину будем использовать очередь.

Можно также получить дерево обхода в ширину, если отмечать каждую прямую дугу.

1	2	3	4	5	6	7	8
	1	2	1	1	4	5	5

Очередь:



Нахождение кратчайшего пути в лабиринте

	1	2	3	4	5	6	7	8	9	10
1	15	14		4	3	2	1	2	3	4
2		13		5		2	2	2		5
3		12		6						6
4		11		7						7
5		10	9	8				20		8
6		11						19		9
7	13	12	13	14	15	16	17	18		
8	14					17		19		
9	15	16	17	18	19	18		20		
10				19	20	19				

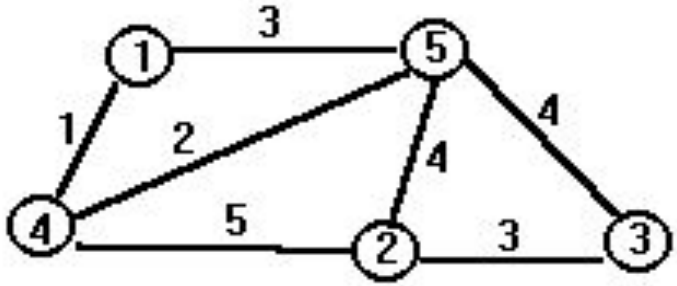
1. Пометить числом 1 и поместить входную клетку в очередь.
2. Взять из очереди клетку.
   Если это выходная клетка, то
   перейти на шаг 4, иначе
   пометить все непомеченные
   соседние клетки числом , на 1 большим, чем данная, и поместить их в очередь.
3. Если очередь пуста, то выдать
   «Выхода нет» и выйти, иначе
   перейти на шаг 2.
4. **Обратный ход**:
   начиная с выходной клетки, каждый раз смещаться на клетку, помеченную на 1 меньше, чем текущая, пока не

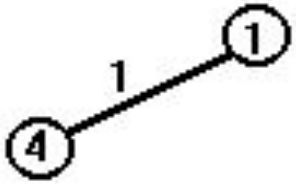
Реализация алгоритма

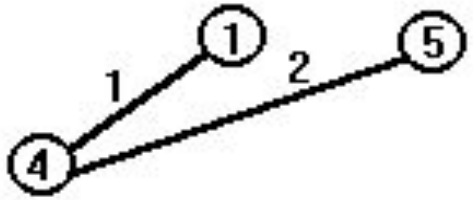
Введем массив меток вершин графа *Mark*.
Начальные значения элементов массива равны номерам
соответствующих вершин (*Mark*[*i*] = *i*; $i \in 1..N$).

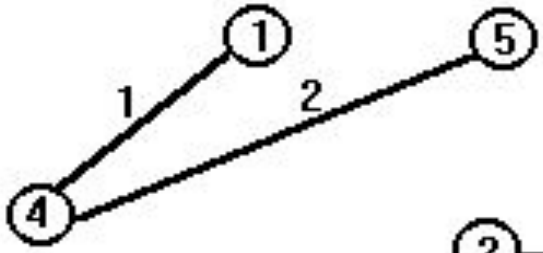
Ребро выбирается в каркас в том случае, если вершины, соединяемые им, имеют разные значения меток.

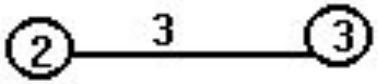
Пример приведен на следующем слайде,
изменения *Mark* показаны в таблице.

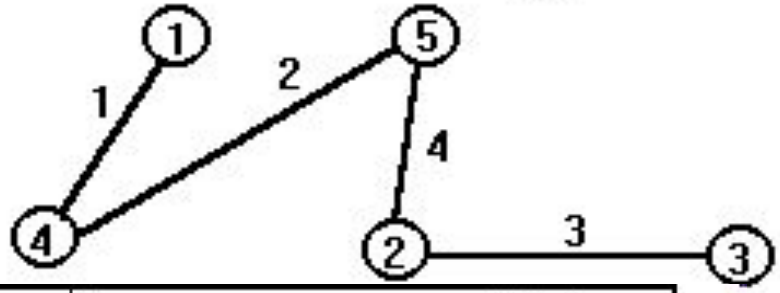












Номер итерации	Ребро	Значения элементов Mark
начальное значение	-	[1,2,3,4,5]
1	<1,4>	[1,2,3,1,5]
2	<4,5>	[1,2,3,1,1]
3	<2,3>	[1,2,2,1,1]
4	<2,5>	[1,1,1,1,1]

Реализации СНМ на Си – объединение по рангу + сжатие путей

Описание алгоритма 6.

Пусть элементы X - это некоторые числа. Вся структура данных хранится в виде двух массивов: ***P*** и ***Rank***.

Массив **P** содержит предков, т.е. *P*[*x*] - это **предок** элемента ***x***.
Т.о., мы имеем древовидную структуру данных: двигаясь по предкам от любого элемента *x*, придём к представителю множества, к которому принадлежит *x*.
Если *P*[*X*] = *X*, то это означает, что *x* является представителем множества, к которому он принадлежит, и корнем дерева.

Массив **Rank** хранит ранги представителей, его значения имеют смысл только для элементов-представителей.
Ранг некоторого элемента-представителя *x* - это верхняя граница его высоты в его дереве. Ранги используются в операции Union.

FindSet (*X*)

Будем двигаться от *X* по предкам, до тех пор пока не найдём представителя. У каждого элемента, который мы проходим, мы также исправляем P, указывая его сразу на найденного представителя.

Т.е. фактически операция **FindSet** двухпроходная: на первом проходе мы ищем представителя, а на втором исправляем значения P.

```
int  find_set (int x) {
   if (x == p[x]) return x;
   return p[x] = find_set (p[x]);
}
```

Union (X, Y)

Сначала заменяем элементы *X* и *Y* на представителей
их множеств, вызывая функцию FindSet.
Объединяем два множества, присваивая *P*[*X*] = *Y* или *P*[*Y*] = *X*:

- если ранги элементов *X* и *Y* отличны, то мы делаем корень с
бо'льшим рангом родительским по отношению к корню с
меньшим рангом.

- если же ранги обоих элементов совпадают, родитель выбирается
произвольным образом, его ранг увеличивается на 1.

```
void unite (int x, int y) {
    x = find_set (x);
    y = find_set (y);
    if (rank[x] > rank[y])  p[y] = x;
   else {
      p[x] = y;
      if (rank[x] == rank[y])
          ++rank[y];
   }
}
```

Реализация со случайным выбором родительского узла

```
void unite (int x, int y) {
   x = find_set (x);
   y = find_set (y);
   if (rand() & 1) p[y] = x;
   else p[x] = y;
}
```

Алгоритм Краскала с системой непересекающихся множеств

Так же, как и в простой версии алгоритма Крускала, отсортируем
все рёбра по неубыванию веса.
Затем поместим каждую вершину в своё дерево (т.е. в своё множество) с помощью вызова функции MakeSet - на это уйдёт в сумме $O(N)$.
Перебираем все рёбра и для каждого ребра за $O(1)$ определяем, принадлежат ли его концы разным деревьям (с помощью двух вызовов *FindSet* за $O(1)$).
Наконец, объединение двух деревьев будет осуществляться
вызовом *Union* - также за $O(1)$.
Итого мы получаем: $\mathbf{O(M \log N + N + M) = O(M \log N)}$.