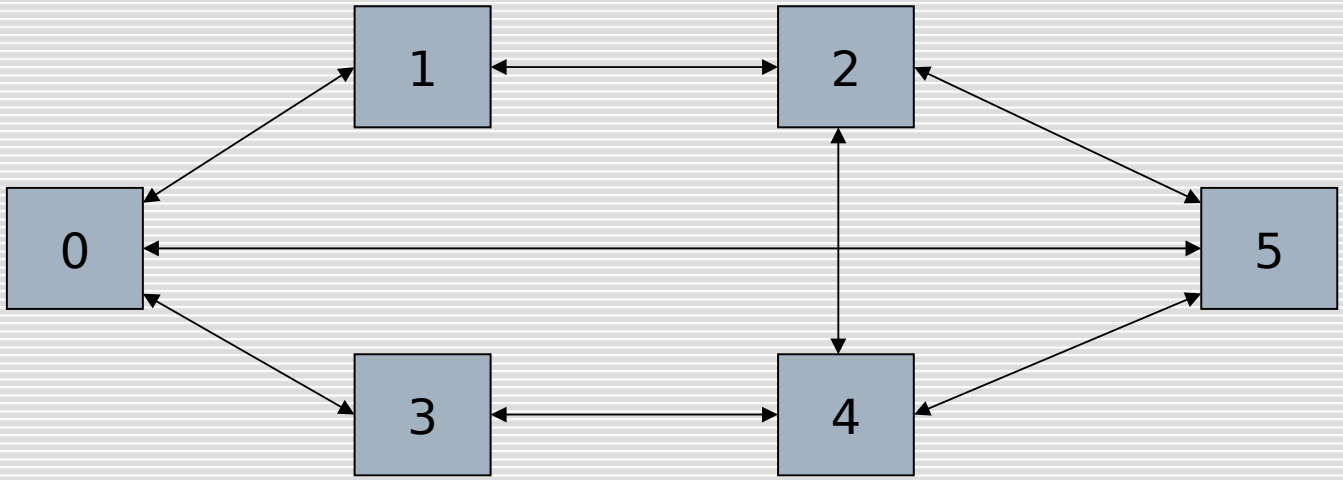
Основы Объектно-ориентированного программирования

Лекция Алгоритмы на графах

Граф

- Граф представляет собой множество **вершин**, соединенных **ребрами**. Каждое **ребро** соединяет ровно две **вершины**

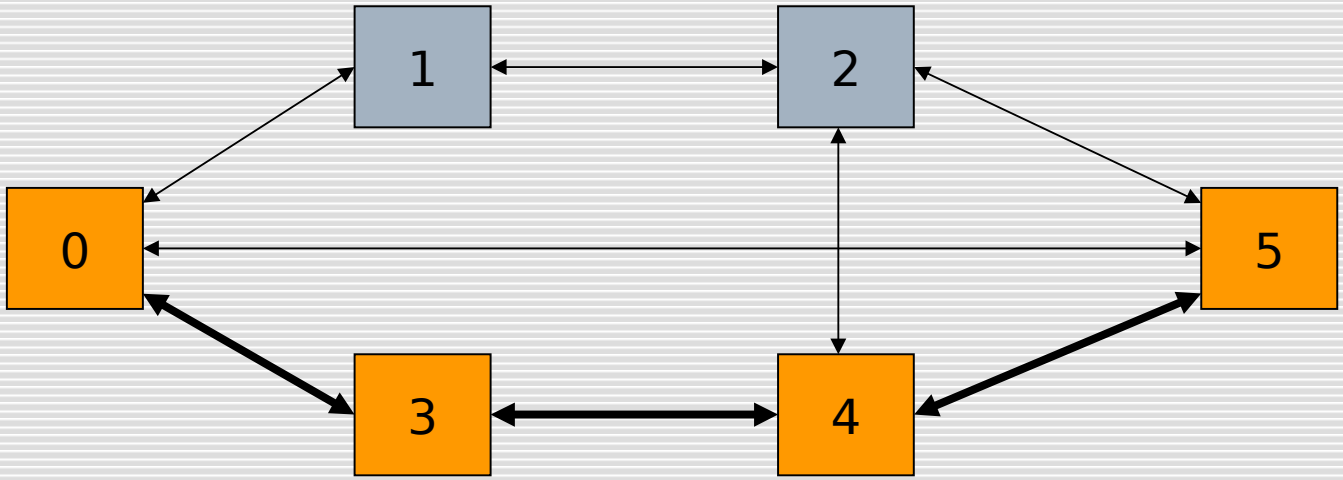


Использование графов

- Графы чаще всего используются для описания системы связей между какими-либо объектами, например
  - сети автомобильных дорог
  - схем метро
  - компьютерных сетей
  - логических схем
  - схем лабиринтов
  - ...

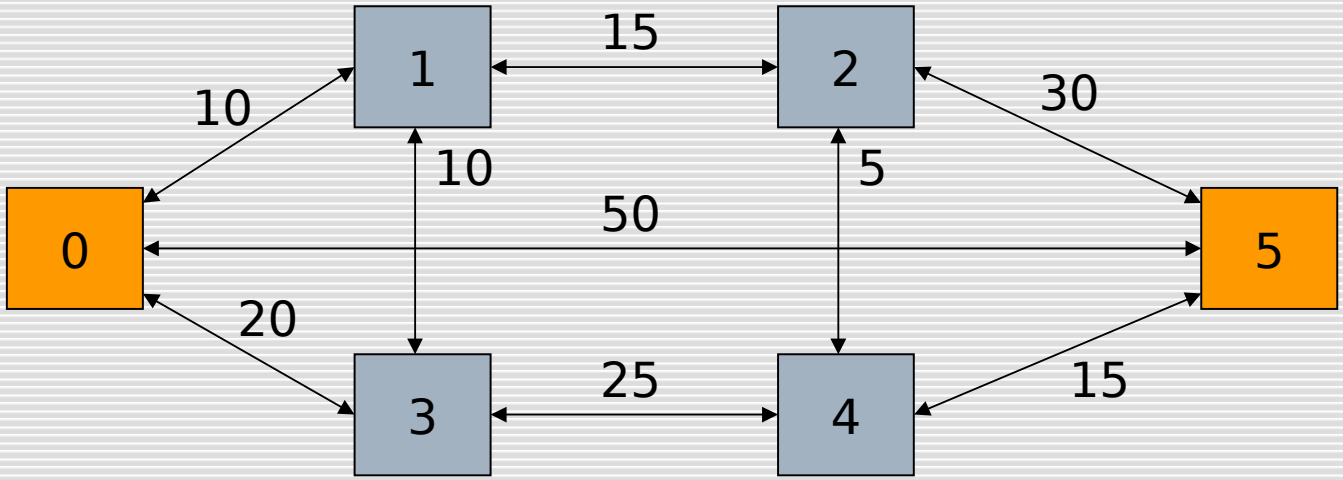
Путь в графе

- Последовательность вершин графа, такая, что любые две соседние вершины соединены ребром, например, (0, 3, 4, 5)



Взвешенный граф, поиск кратчайшего пути

- Каждому его ребру сопоставлен **вес**
- Одна из типовых задач: найти кратчайший путь между двумя вершинами в графе



Граф как структура данных

1. Непосредственная информация о системе связей (какие рёбра соединяют какие вершины).
2. Дополнительная информация о вершинах (например, имена соответствующих городов).
3. Дополнительная информация о рёбрах (например, длины соответствующих путей).

Способы описания графа #1

- Матрица смежности

	0	1	2	3	4	5
0	0	10	0	20	0	50
1	10	0	15	10	0	0
2	0	15	0	0	5	30
3	20	10	0	0	25	0
4	0	0	5	25	0	15
5	50	0	30	0	15	0

Способы описания графа #2

- Матрица инцидентности

	0	1	2	3	4	5	
0	1	1	0	0	0	0	10
1	0	1	1	0	0	0	15
2	0	0	1	0	0	1	30
3	1	0	0	0	0	1	50
4	1	0	0	1	0	0	20
5	0	0	0	1	1	0	25
6	0	0	0	0	1	1	15
7	0	1	0	1	0	0	10
8	0	0	1	0	1	0	5

Способы описания графа #3

- Число вершин + список ребер с весами
  - 6 вершин
  - 0, 1 – 10
  - 1, 2 – 15
  - 2, 5 – 30
  - 0, 5 – 50
  - 0, 3 – 20
  - 3, 4 – 25
  - 4, 5 – 15
  - 1, 3 – 10
  - 2, 4 – 5

Способы описания графа #4

- Число вершин + списки смежности для каждой вершины (м.б. с весами)
  - 6 вершин
  - 0: (1, 10), (3, 20), (5, 50)
  - 1: (1, 10), (3, 10), (2, 15)
  - 2: (1, 15), (4, 5), (5, 30)
  - 3: (0, 20), (1, 10), (4, 25)
  - 4: (2, 5), (3, 25), (5, 15)
  - 5: (0, 50), (2, 30), (4, 15)

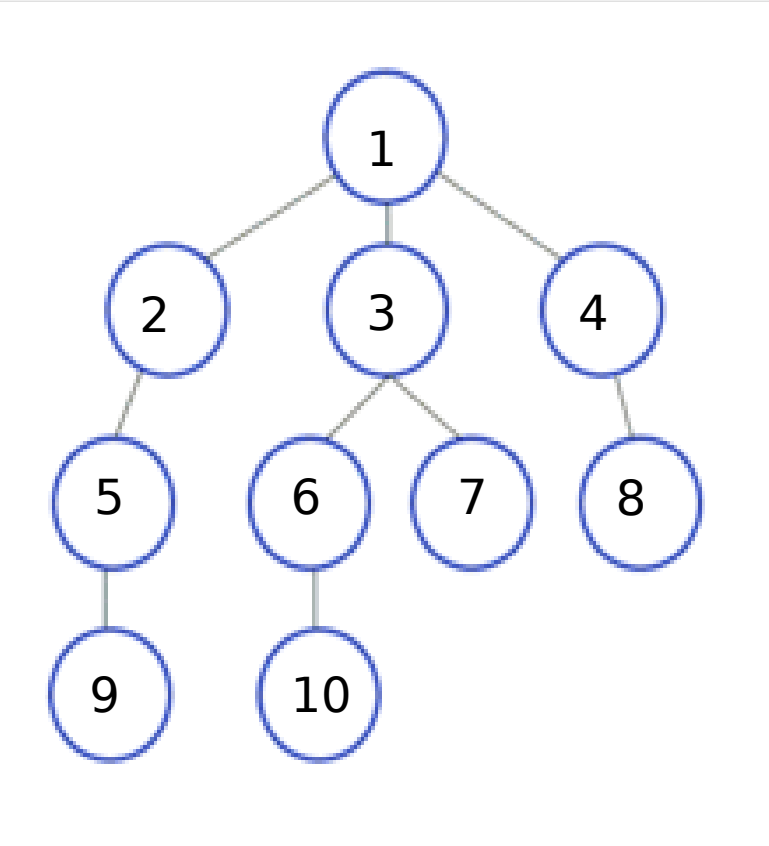
Типовые операции с графом

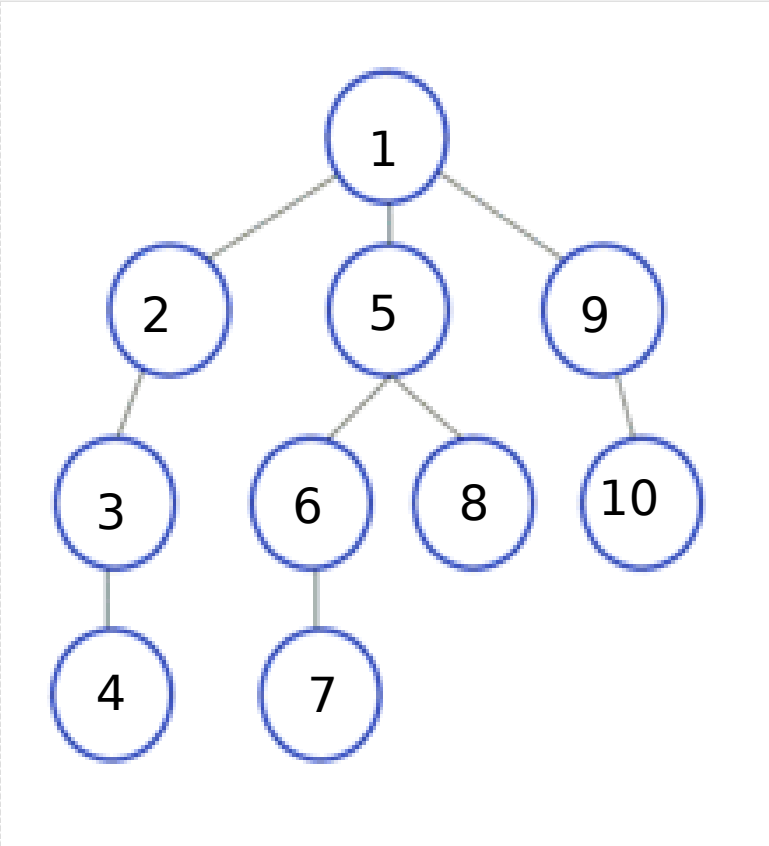
- Добавить вершину / ребро
- Удалить вершину / ребро
- Определить смежность вершин, инцидентность вершины и ребра
- Найти список смежных вершин для заданной
- ...

Методы обхода графа

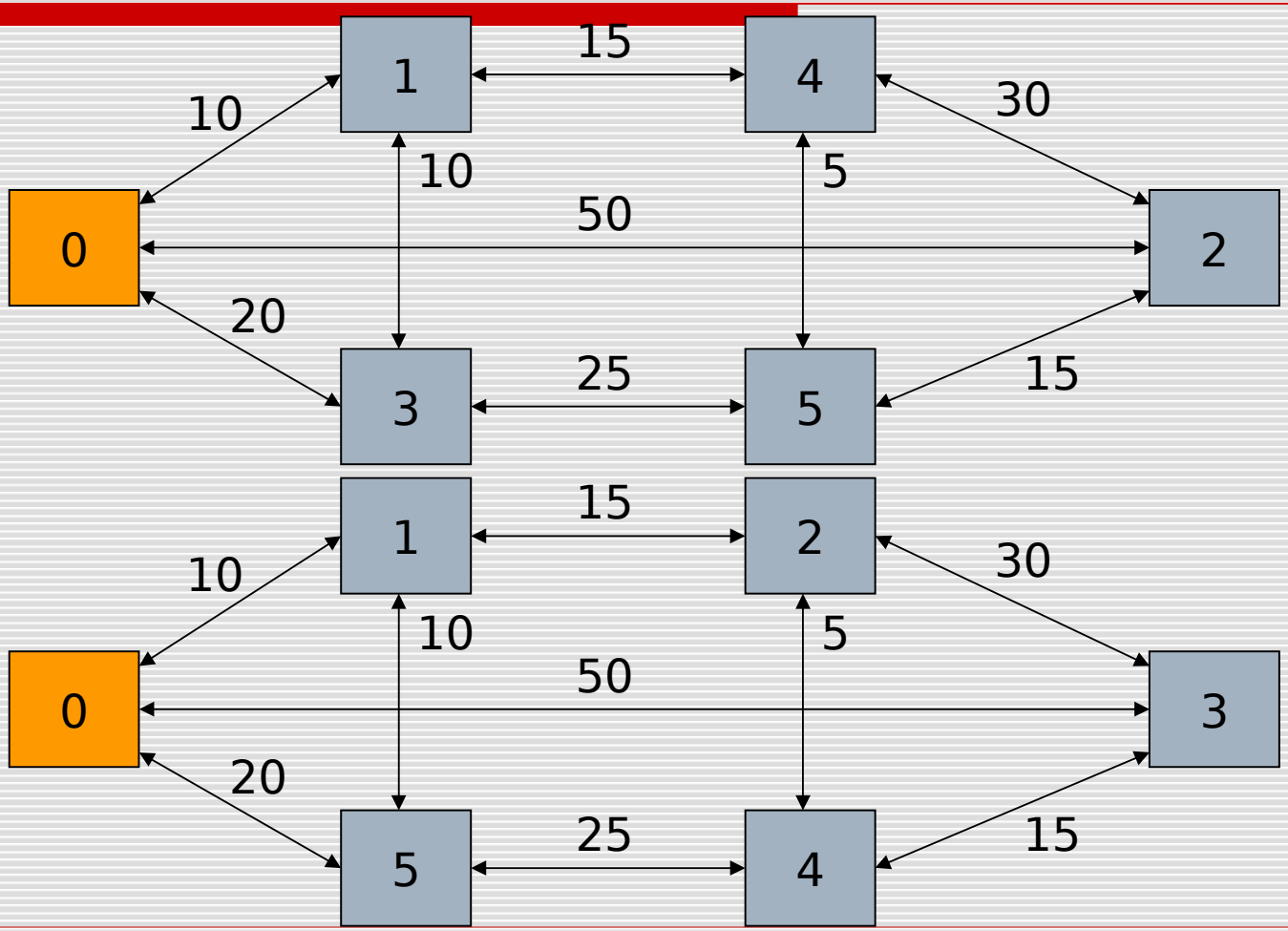
- Поиск в ширину -- начинаем с первой вершины
  - Последовательно обходим её соседей, запоминая по ходу дела их соседей
  - Затем обходим соседей 2-го уровня, запоминая соседей 3-го уровня
  - ...
- Поиск в глубину -- начинаем с первой вершины
  - Обходим её 1-го соседа, затем его 1-го соседа, затем его 1-го соседа, ..., пока новых соседей не будет
  - Возвращаемся на уровень вверх и обходим следующих соседей
  - ...

Поиск в ширину / глубину – порядок перебора





Поиск в ширину / глубину – порядок перебора



Пример: представление графа

- Один из многих возможных вариантов
  - Граф хранит список узлов
  - Каждый узел хранит своё имя и список своих соседей, если это требуется – также с информацией о весах дуг

Граф

```
typedef std::set<Node*>::const_iterator
   node_iterator;
class Graph {
   std::set<Node*> nodes;
public:
   void addNode(Node* node);
   void removeNode(Node* node);
   void addEdge(Node* begin, Node* end);
   void removeEdge(Node* begin, Node* end);
   node_iterator begin() const {
      return nodes.begin(); }
   node_iterator end() const { return nodes.end(); }
};
```

Узел

```
class Node {
   std::string name;
   std::set<Node*> neighbours;
   void addNeighbour(Node* neighbour);
   void removeNeighbour(Node* neighbour);
public:
   Node(const std::string& aname) : name(aname) {}
   const std::string& getName() const { return name; }
   node_iterator nb_begin() const {
      return neighbours.begin(); }
   node_iterator nb_end() const { return neighbours.end(); }
   friend class Graph;
};
```

Некоторые методы – Graph::removeNode()

```
void Graph::removeNode(Node* node) {
    nodes.erase(node);
    // Remove also from all neighbours list
    for (std::set<Node*>::iterator it = nodes.begin();
         it != nodes.end(); it++) {
        (*it)->removeNeighbour(node);
    }
}
```

Некоторые методы – Graph::addEdge

```
void Graph::addEdge(Node* begin, Node* end) {
   if (nodes.find(begin) == nodes.end())
      return;
   if (nodes.find(end) == nodes.end())
      return;
   begin->addNeighbour(end);
   end->addNeighbour(begin);
}
```

Вопрос на подумать

- Как можно «сломать» пару классов Node / Graph, используя их открытые методы?
- И в дополнение – как защититься от найденных вами поломок?

Реализация поиска в ширину

- Цель – определить, существует ли путь между двумя заданными вершинами

```
class BFS {
    const Graph& graph;
public:
    BFS(const Graph& agraph) : graph(agraph) {}
    bool connected(Node* begin, Node* end);
};
```

Реализация поиска в ширину – идея

- Имеем множество уже посещённых узлов (visited)
- Имеем очередь узлов, которые надо посетить (nodes)
- Посещённый узел складывается в visited, а его ещё не посещённые соседи – в nodes
- Затем из nodes берётся следующий узел

Реализация поиска в ширину – метод

```
bool BFS::connected(Node* begin, Node* end) {
   std::queue<Node*> nodes; nodes.push(begin);
   std::set<Node*> visited;
   while (!nodes.empty()) {
      Node* next = nodes.front(); nodes.pop();
      if (end == next) return true;
      visited.insert(next);
      for (node_iterator it = next->nb_begin();
           it != next->nb_end(); it++)
         if (visited.find(*it) == visited.end())
           nodes.push(*it);
   }
   return false;
}
```

Реализация поиска в глубину

- Цель та же – определить, существует ли путь между двумя заданными вершинами

```
class DFS {
   const Graph& graph;
   std::set<Node*> visited;
   bool connected(Node* begin, Node* end, int depth);
public:
   DFS(const Graph& agraph) : graph(agraph) {}
   bool connected(Node* begin, Node* end);
};
```

Реализация поиска в глубину – идея

- Также используем visited для хранения множества посещённых узлов
- Используем **рекурсию** – приём, когда функция для выполнения своей задачи вызывает сама себя, но в более простой ситуации
- Узел A соединён с B, если A==B или если любой из его соседей соединён с B

Реализация поиска в глубину – метод

```
bool DFS::connected(Node* begin, Node* end) {
   visited.clear(); return connected(begin, end, 0);
}
bool DFS::connected(Node* begin, Node* end, int depth) {
   if (begin == end) return true;
   visited.insert(begin);
   for (node_iterator it = begin->nb_begin();
        it != begin->nb_end(); it++) {
      if (visited.find(*it) == visited.end()) {
         if (connected(*it, end, depth + 1)) return true;
      }
   }
   return false;
}
```

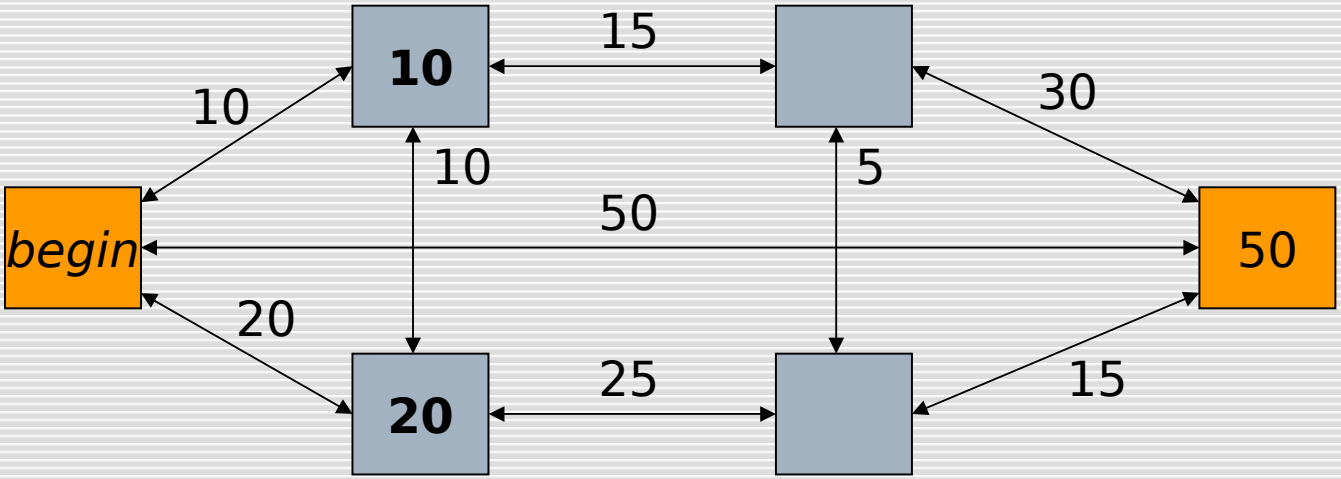
Разновидности поиска в глубину

- Поиск с ограничением глубины
  - Проверяем, что depth не достигла некоторого предела
- Поиск с заглублением
  - Вначале ищем на глубину 1
  - Потом последовательно увеличиваем глубину на 1

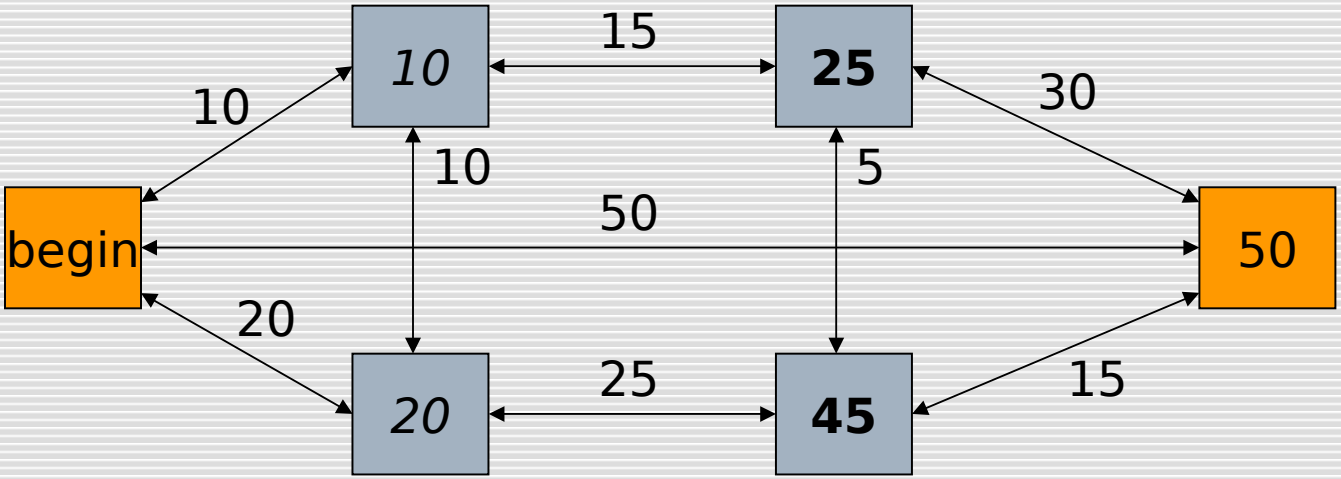
Разновидности поиска в ширину

- Поиск в ширину с учётом весов = алгоритм Дейкстры
  - Вместо обычной очереди используется очередь с приоритетами, приоритетом служит длина пути от начальной вершины, чем он короче, тем раньше рассматривается вершина
- Волновой алгоритм = Алгоритм Ли = поиск в ширину на прямоугольной сетке
- A* = алгоритм Дейкстры + эвристика для оценки пути от текущей вершины до конечной

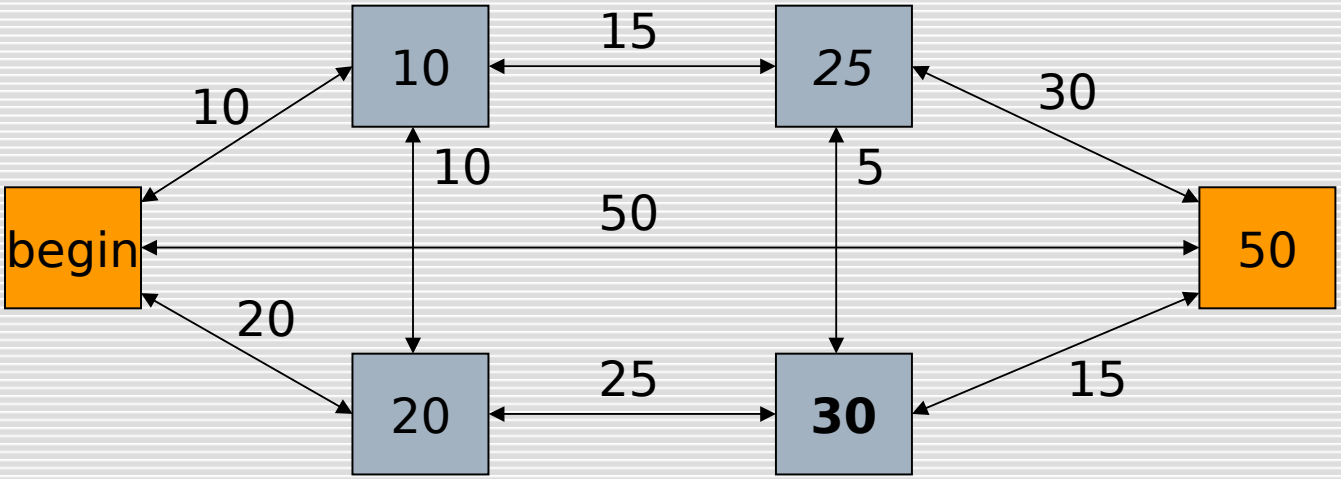
Алгоритм Дейкстры -- иллюстрация



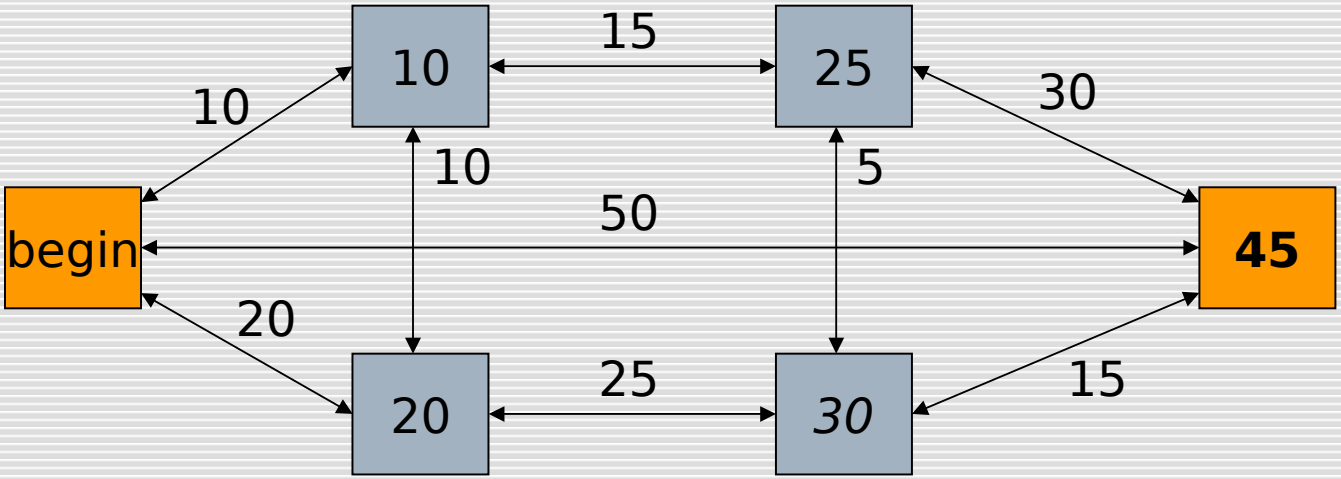
Алгоритм Дейкстры – иллюстрация #2



Алгоритм Дейкстры – иллюстрация #3



Алгоритм Дейкстры – иллюстрация #4



Реализация алгоритма Дейкстры – PriorityQueue

```
struct MarkedNode {
   Node* node; int mark;
   Node* prev;
   MarkedNode(Node* anode=0, int amark=0, Node* aprev=0):
      node(anode), mark(amark), prev(aprev) {}
};
class PriorityQueue {
   std::vector<MarkedNode> nodes;
public:
   MarkedNode pop();
   void push(Node* node, int mark, Node* prev);
   bool empty() const { return nodes.empty(); }
};
```

Очередь с приоритетами -- методы

```
MarkedNode PriorityQueue::pop() {
   MarkedNode mn = nodes.back();
   nodes.pop_back();
   return mn;
}
void PriorityQueue::push(Node* node, int mark, Node* prev) {
   std::vector<MarkedNode>::iterator it = nodes.begin();
   MarkedNode mn(node, mark, prev);
   // From higher to lower
   while (it != nodes.end() && mark < it->mark) it++;
   if (it == nodes.end()) nodes.push_back(mn);
   else nodes.insert(it, mn);
}
```

Алгоритм Дейкстры -- класс

```
struct Way {
   std::vector<Node*> nodes;
   int length;
   Way() : length(-1) {}
};
class Dijkstra {
   const Graph& graph;
public:
   Dijkstra(const Graph& agraph) : graph(agraph) {}
   Way shortestWay(Node* begin, Node* end);
};
```

Алгоритм Дейкстры – основная функция

```
Way Dijkstra::shortestWay(Node* begin, Node* end) {
   PriorityQueue nodes; nodes.push(begin, 0, 0);
   std::map<Node*, MarkedNode> visited;
   while (!nodes.empty()) {
      MarkedNode next = nodes.pop();
      visited[next.node] = next;
      if (end==next.node) return unroll(visited,begin,end);
      for (node_iterator it = next.node->nb_begin();
           it != next.node->nb_end(); it++) {
         int weight = (*it)->getWeight(next.node)+next.mark;
         if (visited.find(*it)==visited.end())
            nodes.push(*it, weight, next.node);
   }  }
   return Way();
}
```

Алгоритм Дейкстры – раскрутка пути

```
static Way unroll(std::map<Node*, MarkedNode> visited,
                  Node* begin, Node* curr) {
   Way way;
   way.length = visited[curr].mark;
   while (curr != begin) {
      way.nodes.push_back(curr);
      curr = visited[curr].prev;
   }
   way.nodes.push_back(begin);
   return way;
}
```

Маленькое отступление – цикл for-each, C++11

```
// For-each loop
// NB: nodes must have begin() and end(),
// iterator must have ++, ->, *
for (Node* other : nodes) {
   other->removeNeighbour(node);
}
// Is equivalent to
for (std::set<Node*>::iterator it = nodes.begin();
     it != nodes.end(); it++) {
   (*it)->removeNeighbour(node);
}
```