ОСНОВЫ ТЕОРИИ И ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

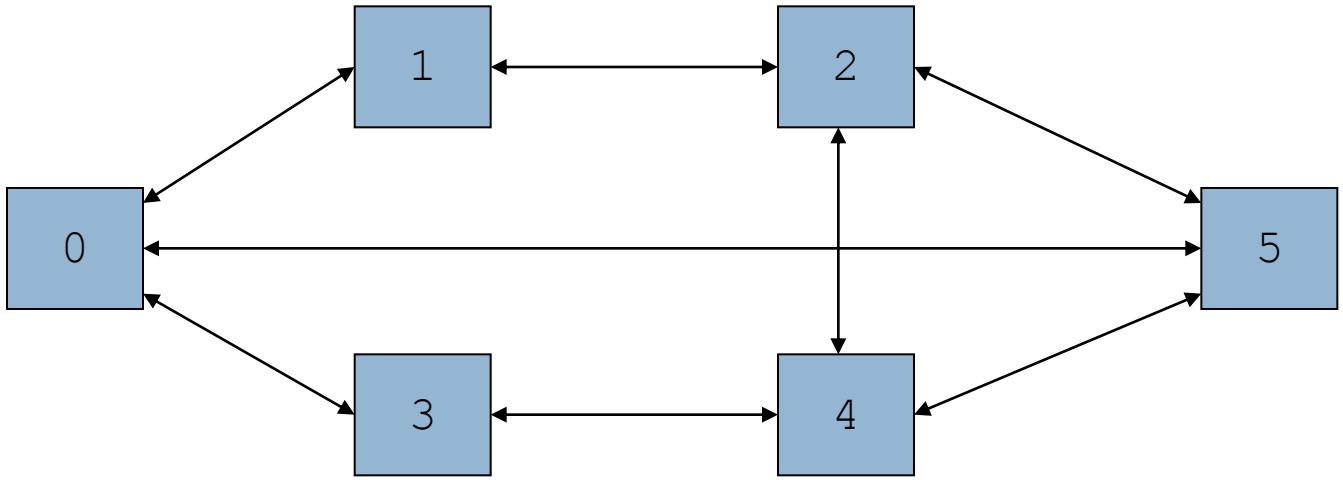
Пышкин Евгений Валерьевич
к.т.н., доцент

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

Блок 9. Графы и поиск на графе

Граф

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

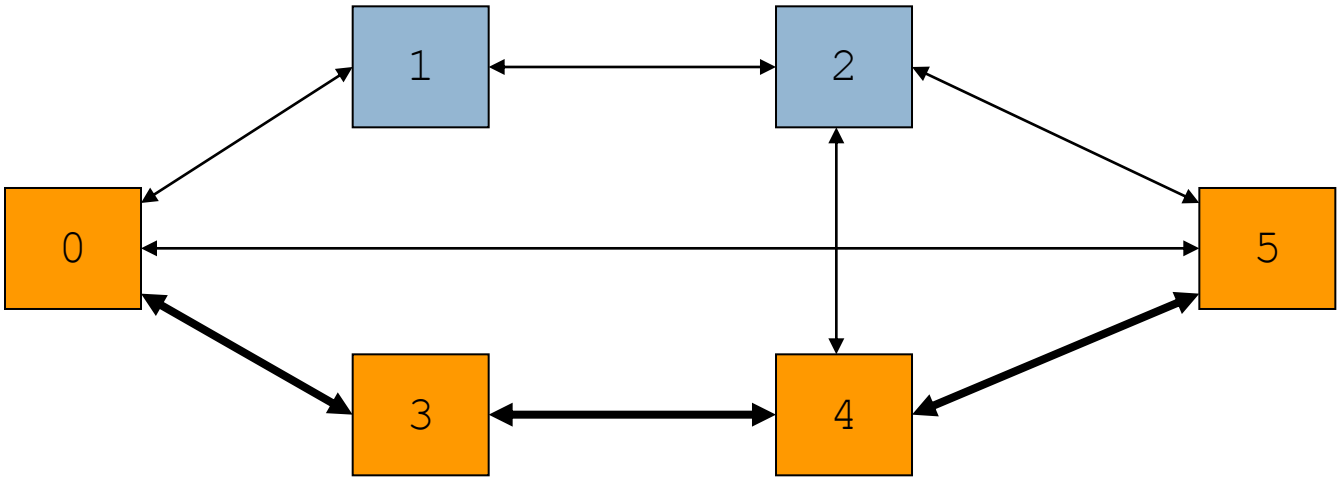


Использование графов

- Графы чаще всего используются для описания системы связей между какими-либо объектами, например
  - сети автомобильных дорог
  - компьютерных сетей
  - …

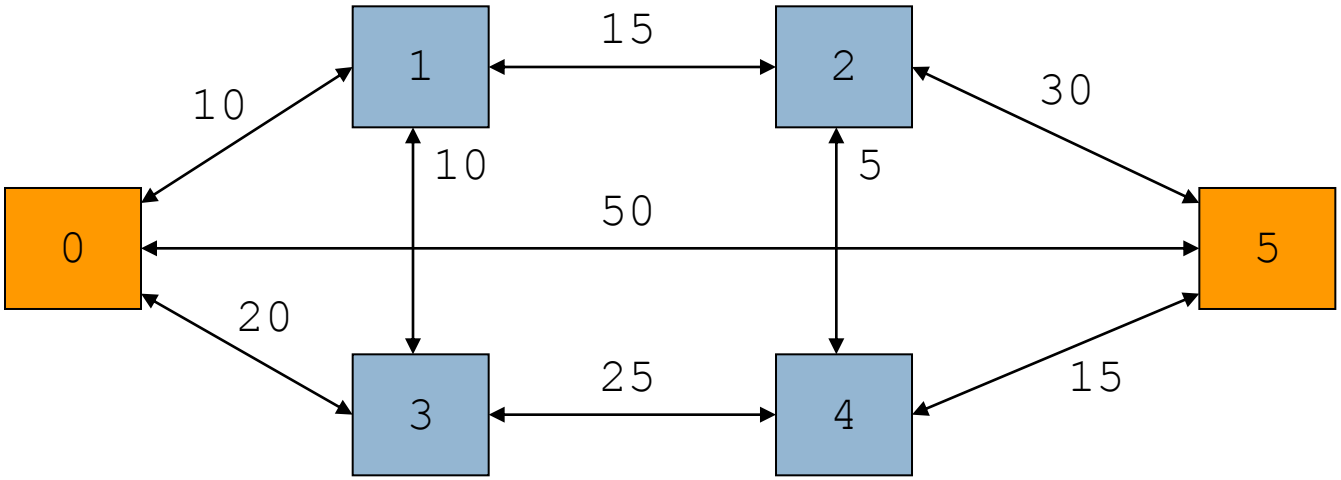
Путь в графе

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

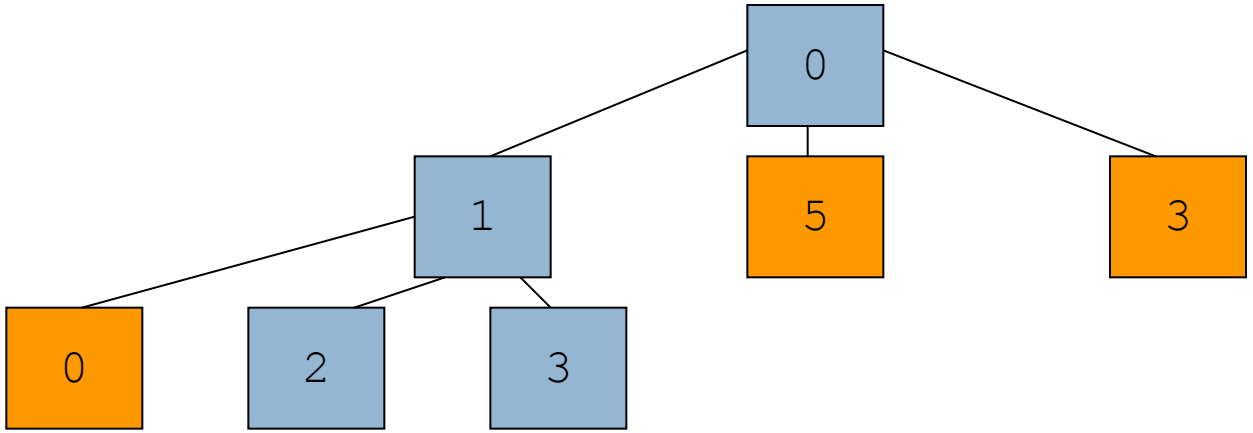


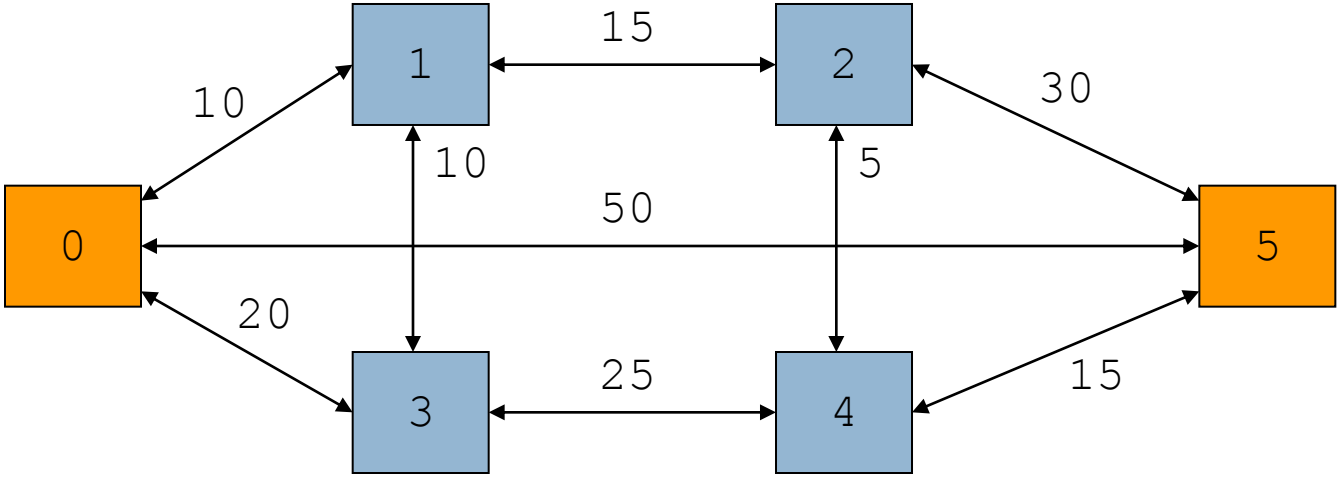
Поиск кратчайшего пути

- Пусть дан граф, причем каждому его ребру сопоставлен **вес** (взвешенный граф). Требуется найти путь между двумя заданными вершинами с наименьшим весом

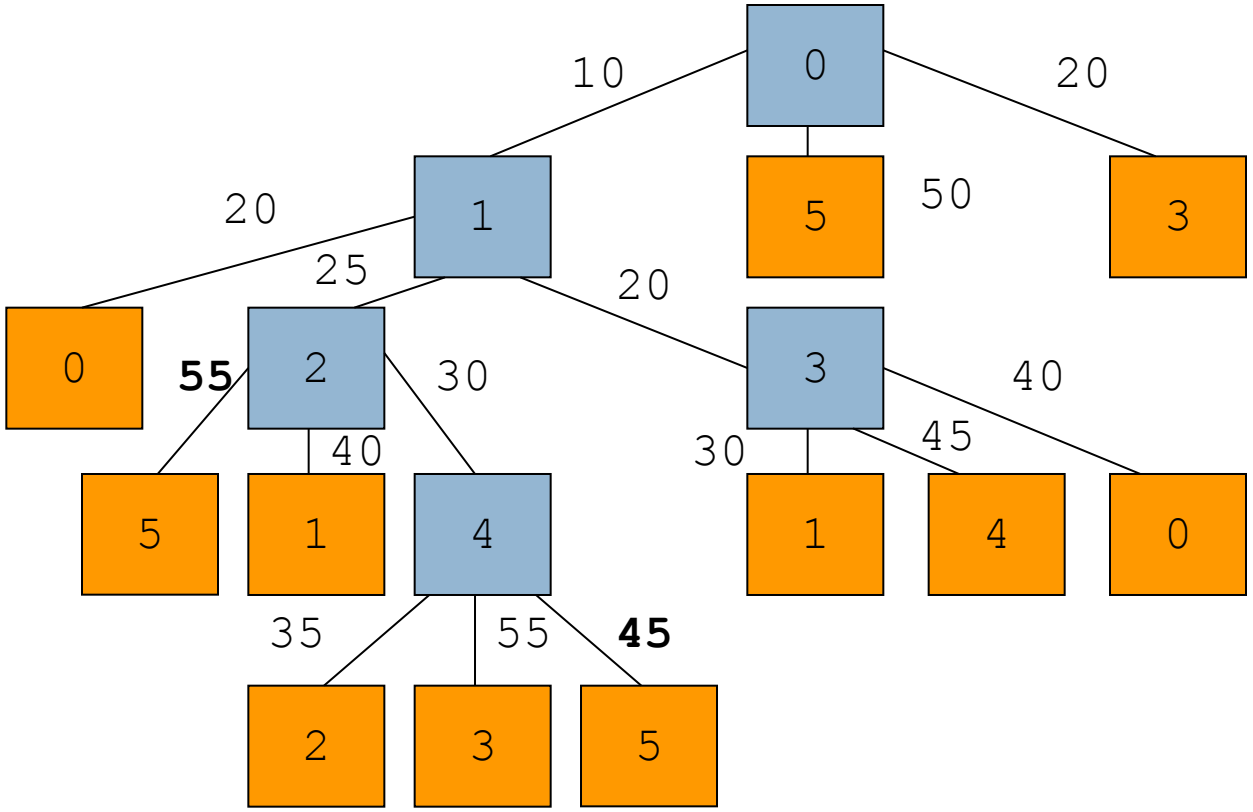


Алгоритм рекурсивного перебора





Алгоритм рекурсивного перебора



Способы описания графа

- 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

- (Расширенная) 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

- Число вершин + список ребер
  - 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

Описание графа с помощью списка ребер


```
struct Edge {
  int begin, end;
  int cost;
  Edge(int b, int e, int c);
};
class Graph {
  int vertexNum;
  vector<Edge> edges;
public:
  Graph(int vnum);
  void connect(int begin, int end, int cost);
  vector<Edge> getNearEdges(int vertexIndex) const;
  int getVertexNum() const;
};
```

Функция соединения вершин


```
void Graph::connect(int begin, int end, int cost) {
  if (begin < 0    || begin >= vertexNum ||
      end < 0      || end >= vertexNum   ||
      begin==end || cost <= 0)
      return;
  Edge edge(begin, end, cost);
  edges.push_back(edge);
}
```

Функция поиска инцидентных ребер


```
vector<Edge> Graph::getNearEdges(int vertexIndex) const {
  vector<Edge> nearEdges;
  for (vector<Edge>::const_iterator it=edges.begin();
       it!=edges.end(); it++) {
    if (it->begin==vertexIndex ||
        it->end==vertexIndex)
      nearEdges.push_back(*it);
  }
  return nearEdges;
}
```

Описание структур данных

- Необходимо хранить информацию о том, куда мы идем
- Для каждой вершины необходимо помнить лучшую стоимость пути и предыдущую вершину на этом пути

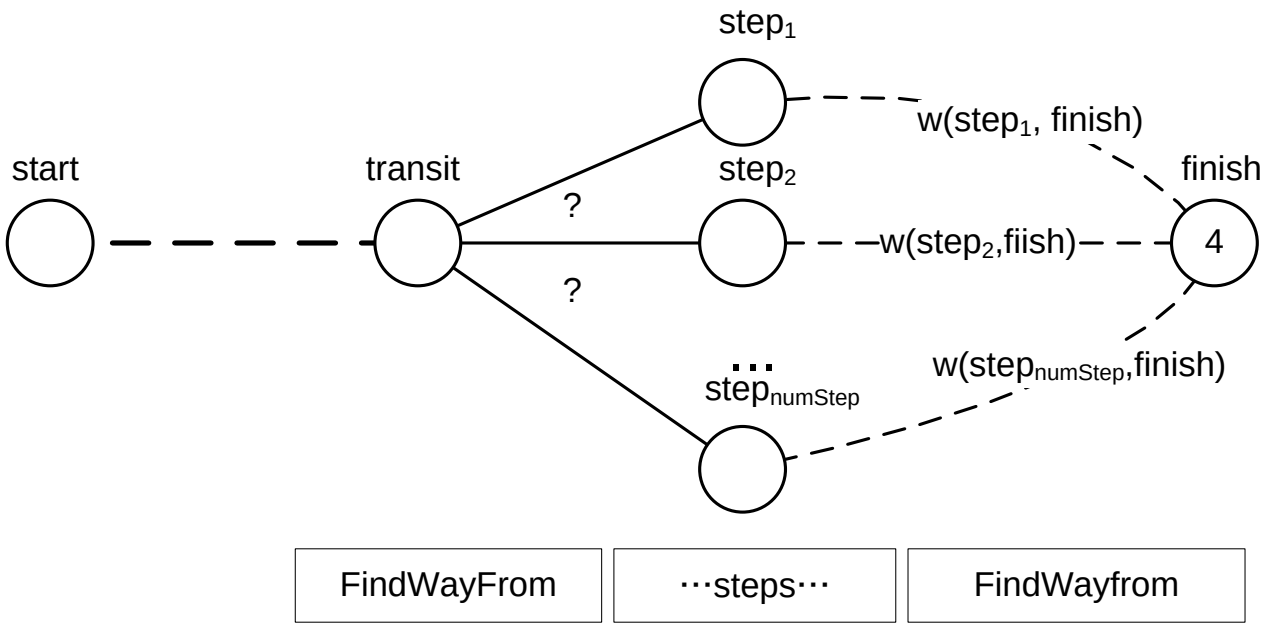
Описание путей


```
struct WayInfo {
  bool exists;
  int prev;
  int cost;
  WayInfo();
};

struct Way {
  vector<int> vertexes;
  bool exists;
  int cost;
  Way();
};
```

Общая идея алгоритма поиска

- Рекурсивный процесс



$$\min_{i \in \{1, numStep\}} (w(transit, step_i) + sumW(step_i, finish))$$

Класс для решения задачи поиска


```
class Voyager
{
  const Graph& graph;
  int target;
  int optimalCost;
  WayInfo* ways;
  void findWayFrom(int vertexIndex);
public:
  Voyager(const Graph& g);
  Way findWay(int begin, int end);
};
```

Конструкторы

- *// По умолчанию создаются*
  // несуществующие пути
- WayInfo::WayInfo():
  exists(false), prev(-1), cost(-1)
- {}
- Way::Way():
  vertexes(), exists(false), cost(0) {}
- Voyager::Voyager(const Graph& g):
  graph(g), ways(0)
- {}

Поиск пути между двумя вершинами

- Необходимо:
  - создать список путей (WayInfo)
  - заполнить его элемент для начальной вершины
  - вызвать рекурсивную функцию
  - заполнить структуру Way

Поиск пути между двумя вершинами


```
Way Voyager::findWay(int begin, int end) {
  Way way;
  if (begin < 0 || begin >= graph.getVertexNum() ||
     end < 0 || end >= graph.getVertexNum())
     return way;
  delete[] ways;
  ways = new WayInfo[graph.getVertexNum()];
  optimalCost = -1;
  ways[begin].exists = true;
  ways[begin].prev = -1;
  ways[begin].cost = 0;
  target = end;
  // …
}
```

Поиск пути между двумя вершинами


```
Way Voyager::findWay(int begin, int end) {
  // …
  findWayFrom(begin);
  int currVertex = end;
  if (!ways[end].exists)
    return way;
  way.exists = true;
  way.cost = optimalCost;
  while (currVertex != -1)  {
    way.vertexes.push_back(currVertex);
    currVertex = ways[currVertex].prev;
  }
  return way;
}
```

Рекурсивная функция обхода


```
void Voyager::findWayFrom(int vertexIndex) {
  if (vertexIndex==target) {
    optimalCost = ways[target].cost;
    return;
  }
  vector<Edge> nearEdges =
    graph.getNearEdges(vertexIndex);
  for (vector<Edge>::iterator it=nearEdges.begin();
      it!=nearEdges.end(); it++) {
    // …
  }
}
```

Рекурсивная функция обхода


```
void Voyager::findWayFrom(int vertexIndex) {
  // …
  for (vector<Edge>::iterator it=nearEdges.begin();
      it!=nearEdges.end(); it++) {
    int next = it->end;
    if (next==vertexIndex) next = it->begin;
    int cost = ways[vertexIndex].cost + it->cost;
    if (optimalCost != -1 && cost > optimalCost) continue;
    if (!ways[next].exists || ways[next].cost > cost){
      ways[next].exists = true;
      ways[next].prev = vertexIndex;
      ways[next].cost = cost;
      findWayFrom(next);
    }
  }
}
```

Тестирование

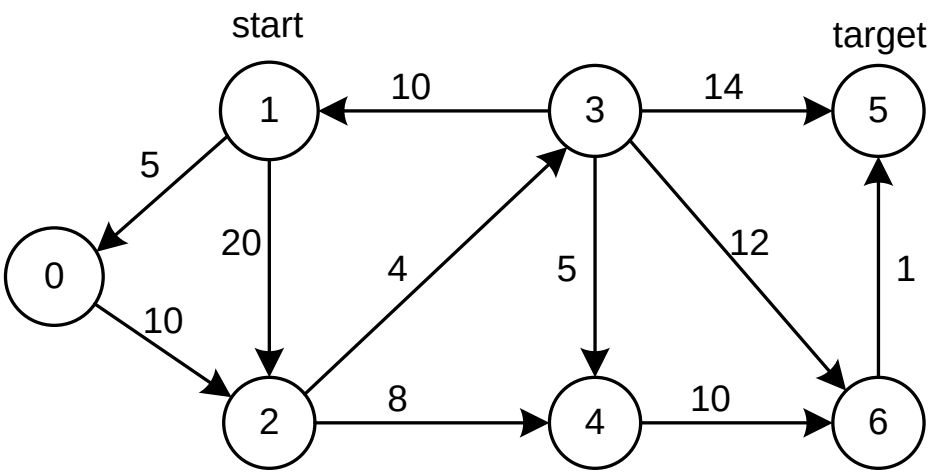

```
int main(void) {
  Graph g(6);
  g.connect(0, 1, 10);
  g.connect(1, 2, 15);
  g.connect(2, 5, 30);
  g.connect(0, 3, 20);
  g.connect(3, 4, 25);
  g.connect(4, 5, 15);
  g.connect(0, 5, 50);
  g.connect(2, 4, 5);
  Voyager v(g);
  Way way = v.findWay(0, 5);
  //…
}
```

Тестирование


```
int main(void) {
  // …
  Voyager v(g);
  Way way = v.findWay(0, 5);
  if (!way.exists) {
    cout<<"Way does not exists"<<endl;
  } else {
    cout<<"Optimal way has cost "<<way.cost<<endl;
    cout<<"Optimal way goes through: ";
    for (vector<int>::iterator it=way.vertexes.begin();
      it!=way.vertexes.end(); it++)
      cout<<*it<<' ';
    cout<<endl;
  }
  return 0;
}
```

Иллюстрация процесса (1)

Диаграмма взвешенного орграфа



start – начальная вершина для поиска

target – конечная вершина

Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

Иллюстрация процесса (2)

Начальное состояние массива ways

индекс узла	0	1	2	3	4	5	6
exist	false	true	false	false	false	false	false
cost		0					
prev		-1					

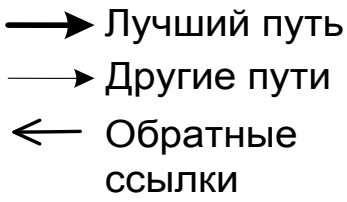
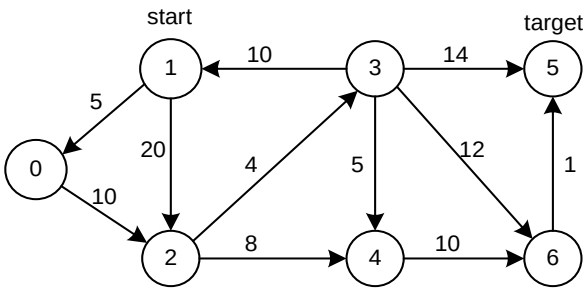


Иллюстрация процесса (3)

Диаграмма взвешенного орграфа



Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

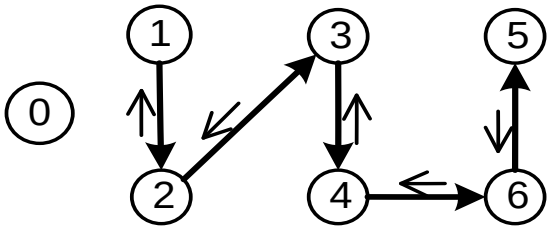
start – начальная вершина для поиска

target – конечная вершина

Состояния массива ways

0	1	2	3	4	5	6
false	true	true	true	true	true	true
	0	20	24	29	40	39
	-1	1	2	3	6	4

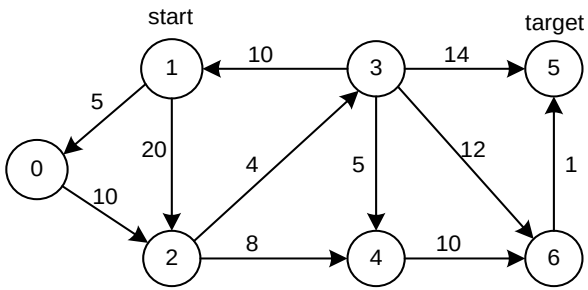
Открытые пути



Открыт путь 1-2-3-4-6-5

Иллюстрация процесса (4)

Диаграмма взвешенного орграфа



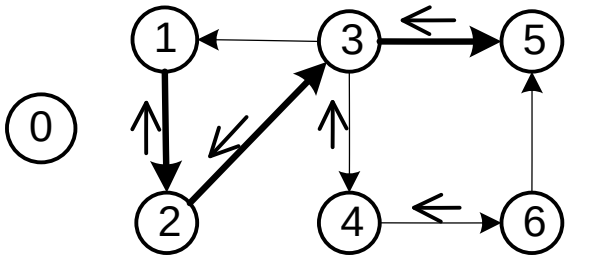
start – начальная вершина для поиска

target – конечная вершина

Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

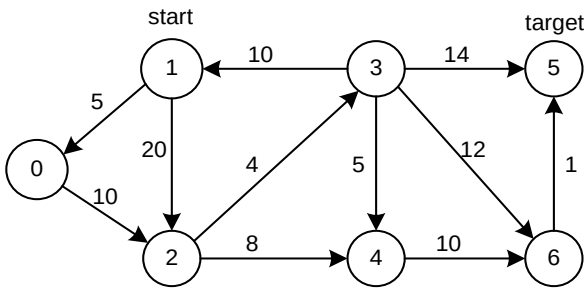
0	1	2	3	4	5	6
false	true	true	true	true	true	true
	0	20	24	29	**38**	39
	-1	1	2	3	**3**	4



Путь 3-5 лучше пути 3-4-6-5

Иллюстрация процесса (5)

Диаграмма взвешенного орграфа



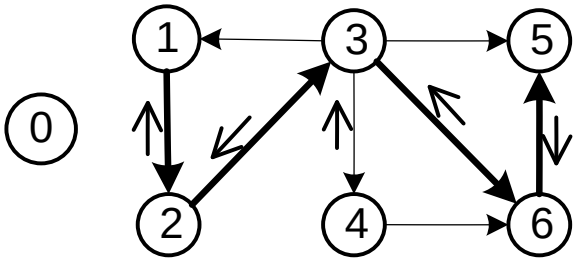
start – начальная вершина для поиска

target – конечная вершина

Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

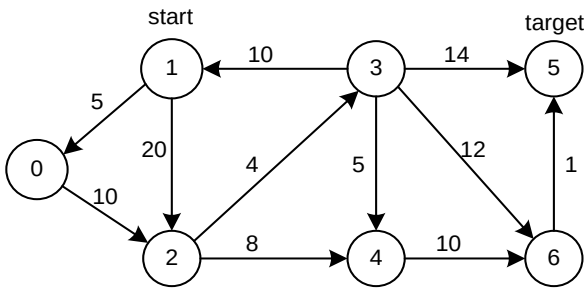
0	1	2	3	4	5	6
false	true	true	true	true	true	true
	0	20	24	29	**37**	**36**
	-1	1	2	3	**6**	**3**



Путь 3-6-5 лучше пути 3-5

Иллюстрация процесса (6)

Диаграмма взвешенного орграфа



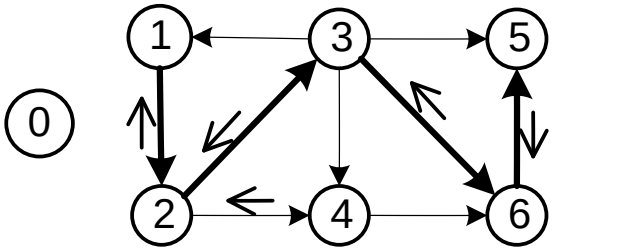
start – начальная вершина для поиска

target – конечная вершина

Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

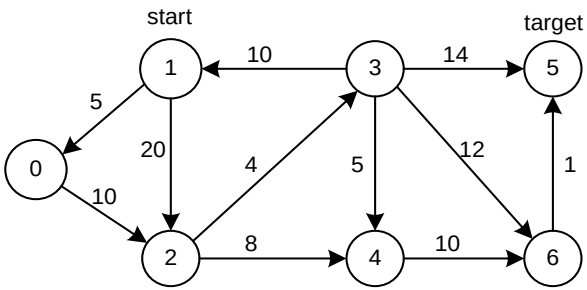
0	1	2	3	4	5	6
false	true	true	true	true	true	true
	0	20	24	**28**	37	36
	-1	1	2	**2**	6	3



Путь 2-4 лучше пути 2-3-4, но путь 2-4-6-5 не лучше ранее открытого 2-3-6-5

Иллюстрация процесса (7)

Диаграмма взвешенного орграфа



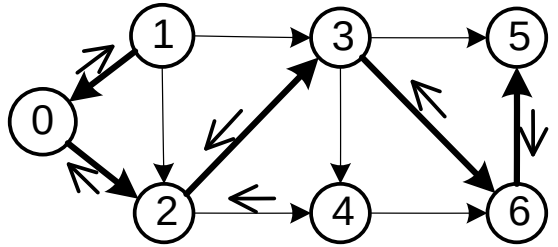
start – начальная вершина для поиска

target – конечная вершина

Массив дуг

	begin	end	cost
0	1	2	20
1	1	0	5
2	0	2	10
3	2	3	4
4	2	4	8
5	3	1	10
6	3	4	5
7	3	5	14
8	4	6	10
9	3	6	12
10	6	5	1

0	1	2	3	4	5	6
true	true	true	true	true	true	true
5	0	**15**	**19**	28	**32**	**31**
1	-1	**0**	2	2	6	3



Путь 1-0-2 лучше пути 1-2,
искомый путь: 1-0-2-3-6-5

Подведем итоги

- Рассмотрено понятие графа
- Рассмотрена задача поиска кратчайшего пути и возможный алгоритм ее решения
  - Существует множество других алгоритмов
    - Дейкстры
    - Беллман-Форда
    - Флойда
    - ….
- Рассмотрена реализация алгоритма на языке C++