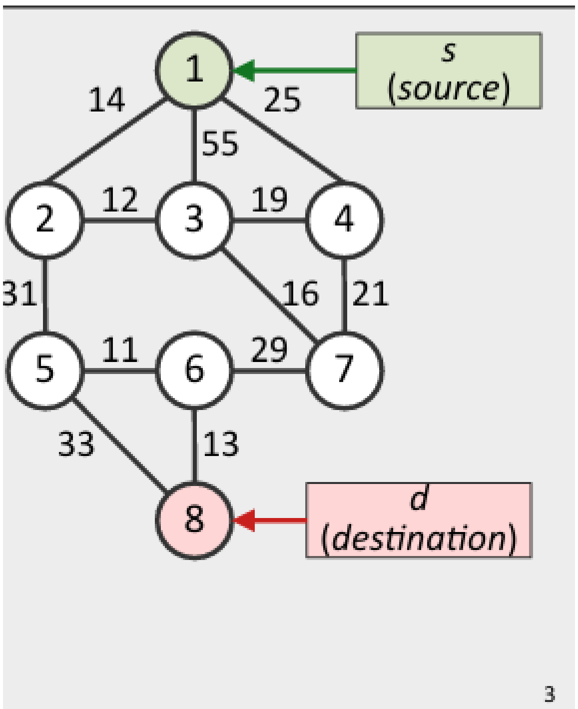
Задача поиска кратчайшего пути в графе. Постановки задачи о кратчайшем пути. Алгоритмы поиска кратчайшего пути в графе. Алгоритм Дейкстры. Вычислительная сложность алгоритма Дейкстры.

Задача поиска кратчайшего пути в графе.

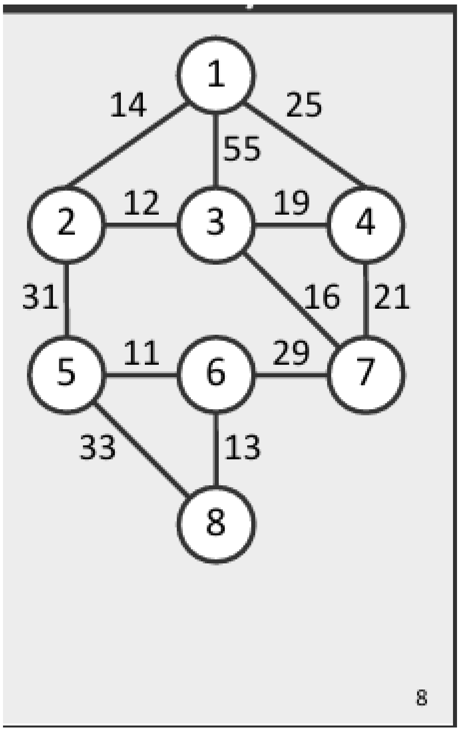
- Имеется взвешенный граф G = (V, E)
- Каждому ребру (i, j) ∈ **E** назначен вес w[i , j]
- Заданы начальная вершина **s** ∈ **V** и конечная **d** ∈ **V**
- Требуется найти **кратчайший путь** из вершины **s** в вершину **d** *(shortest path problem)*
- Длина пути *(path length, path cost, path weight)* – это сумма весов ребер, входящих в него



``

Постановки задачи о кратчайшем пути

- **Задача о кратчайшем пути между парой вершин** *(single-pair shortest path problem)*
  Требуется найти кратчайший путь из заданной вершины s в заданную вершину d
- **Задача о кратчайших путях из заданной вершины во все** *(single-source shortest path problem)*
  Требуется найти кратчайшие пути из заданной вершины s во все
- **Задача о кратчайшем пути в заданный пункт назначения** *(single-destination shortest path problem)*
  Требуется найти кратчайшие пути в заданную вершину v из всех вершин графа
- **Задача о кратчайшем пути между всеми парами вершин** *(all-pairs shortest path problem)*
  Требуется найти кратчайший путь из каждой вершины u в каждую вершину v



Алгоритмы поиска кратчайшего пути в графе

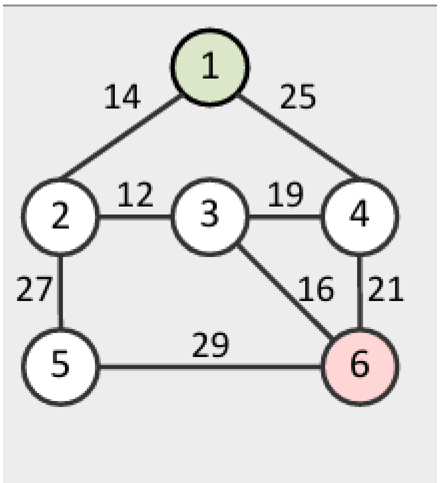
Алгоритм	Применение
Алгоритм Дейкстры	Находит кратчайший путь от одной из вершин графа до всех остальных. Алгоритм работает только для графов без ребер отрицательного веса ($w_{ij} \geq 0$)
Алгоритм Беллмана-Форда	Находит кратчайшие пути от одной вершины графа до всех остальных во взвешенном графе. Вес ребер может быть отрицательным
Алгоритм поиска A* (A star)	Находит путь с наименьшей стоимостью от одной вершины к другой, используя алгоритм поиска по первому наилучшему совпадению на графе
Алгоритм Флойда-Уоршелла	Находит кратчайшие пути между всеми вершинами взвешенного ориентированного графа
Алгоритм Джонсона	Находит кратчайшие пути между всеми парами вершин взвешенного ориентированного графа (должны отсутствовать циклы с отрицательным весом)
Алгоритм Ли (волновой алгоритм)	Находит путь между вершинами *s* и *t* графа, содержащий минимальное количество промежуточных вершин (трассировки электрических соединений на кристаллах микросхем и на печатных платах)
Алгоритмы Viterbi, Cherkassky, …	

13

Алгоритм Дейкстры

- **Алгоритм Дейкстры** *(Dijkstra's algorithm, 1959)* – алгоритм поиска кратчайшего пути в графе из заданной вершины во все остальные (single-source shortest path problem)
- Находит кратчайшее расстояние от одной из вершин графа до всех остальных
- Применим только для графов **без ребер отрицательного веса и петель** (w[i , j] ≥ 0)

Принцип работы :



1. Устанавливаем расстояние D[i] от начальной вершины s до всех остальных в ∞
2. Полагаем D[s] = 0

3. Помещаем все вершины в очередь с приоритетом Q (min-heap): приоритет вершины i – это значение D[i]
4. Запускаем цикл из n итераций (по числу вершин):
   - Извлекаем из очереди Q вершину v с минимальным приоритетом – ближайшую к s вершину
   - Отмечаем вершину v как посещенную (помещаем во множество H)
   - Возможно, пути из s через вершину v стали короче, выполняем проверку: для каждой вершины u, смежной с v и не включённой в H, проверяем и корректируем расстояние D[u]
   - Повторяем для всех вершин.

```
/**
 * Реализация алгоритма Дейкстры для поиска кратчайших путей из одной
вершины
 *
 * @param graph - указатель на структуру графа
 * @param src   - индекс начальной вершины
 * @param dist  - массив для хранения расстояний от src до каждой вершины
 * @param prev  - массив для хранения предшественников вершин в кратчайших
путях
 */
void dijkstra(Graph* graph, int src, int dist[], int prev[]) {
    int V = graph->V; // Количество вершин в графе
    MinHeap* minHeap = createMinHeap(V); // Создаем приоритетную очередь
(мин-кучу)

    // Инициализация кучи и массивов расстояний/предшественников
    for (int v = 0; v < V; ++v) {
        dist[v] = INF;       // Изначально все расстояния бесконечны
        prev[v] = -1;        // Предшественники не определены
        minHeap->array[v] = newMinHeapNode(v, dist[v]); // Создаем узлы кучи
        minHeap->pos[v] = v; // Запоминаем позиции вершин в куче
    }

    // Устанавливаем расстояние до начальной вершины в 0
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]); // Обновляем позицию в куче

    minHeap->size = V; // Устанавливаем размер кучи равным количеству вершин

    // Основной цикл алгоритма
    while (!isEmpty(minHeap)) {
        // Извлекаем вершину с минимальным расстоянием
        MinHeapNode* minHeapNode = extractMin(minHeap);
        int u = minHeapNode->v; // Текущая вершина

        // Перебираем всех соседей текущей вершины
```

```
        AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL) {
            int v = pCrawl->dest; // Соседняя вершина

            // Проверяем:
            // 1. Что вершина еще в куче
            // 2. Что расстояние до u не бесконечно
            // 3. Что новый путь короче текущего
            if (isInMinHeap(minHeap, v) && dist[u] != INF &&
                pCrawl->weight + dist[u] < dist[v]) {
                dist[v] = dist[u] + pCrawl->weight; // Обновляем расстояние
                prev[v] = u; // Запоминаем предшественника
                decreaseKey(minHeap, v, dist[v]); // Обновляем позицию в
куче
            }
            pCrawl = pCrawl->next; // Переходим к следующему соседу
        }

        free(minHeapNode); // Освобождаем память извлеченного узла
    }

    // Освобождаем память, выделенную под кучу
    free(minHeap->pos);
    free(minHeap->array);
    free(minHeap);
}
```

Восстановление кратчайшего пути:

```
int* search_shortest_path(GRAPH *g, int src, int dst, int *pathlen) {

    int n = g->nvertices;

    int *path = NULL;

    *pathlen = 0;

    // Проверка корректности вершин

    if (src < 1 || src > n || dst < 1 || dst > n) {

        return NULL;

    }

    int *dist = malloc(n * sizeof(int));

    int *prev = malloc(n * sizeof(int));
```

```c
    if (!dist || !prev) {
        free(dist);
        free(prev);
        return NULL;
    }
    dijkstra(g, src, dist, prev);
    // Проверяем, существует ли путь
    if (dist[dst-1] == INT_MAX) {
        free(dist);
        free(prev);
        return NULL;
    }
    // Вычисляем длину пути
    int current = dst;
    while (current != -1 && *pathlen <= n) {
        (*pathlen)++;
        current = prev[current-1];
    }
    if (*pathlen > n) { // Обнаружен цикл
        free(dist);
        free(prev);
        *pathlen = 0;
        return NULL;
    }
    // Формируем путь
```

```
    path = malloc(*pathlen * sizeof(int));

    if (!path) {

        free(dist);

        free(prev);

        *pathlen = 0;

        return NULL;

    }

    current = dst;

    for (int i = *pathlen-1; i >= 0; i--) {

        path[i] = current;

        current = prev[current-1];

    }

    free(dist);

    free(prev);

    return path;

}
```

Вычислительная сложность алгоритма Дейкстры

- Вычислительная сложность алгоритма Дейкстры определяется следующими факторами:
  1. Выбором структуры данных для хранения графа (матрица смежности, ◦ списки смежных вершин)
  2. Способом поиска вершины с минимальным расстоянием D[i]:
     - Очередь с приоритетом: Binary heap – O(logn), Fibonacci heap, …
     - Сбалансированное дерево поиска: Red-black tree – O(logn), AVL-tree, …
     - Линейный поиск – O(n)

- Вариант 1. D – это массив или список: поиск за время O(n)

$$T_{Dijkstra} = O(n^2 + m) = O(|V|^2 + |E|)$$

- Вариант 2. D – это бинарная куча

$$T_{Dijkstra} = O(n\log n + m\log n) = O(m\log n)$$

- Вариант 3. D – это фибоначчиева куча (*Fibonacci heap*)

$$T_{Dijkstra} = O(m + n\log n)$$

- Вариант 4. …

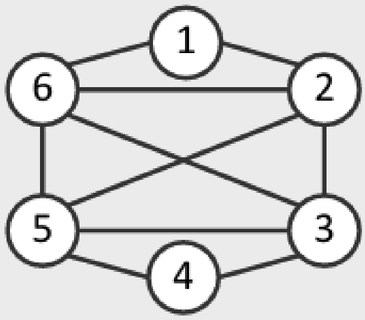
В ориентированных ациклических графах (*directed acyclic graph*) кратчайший путь можно найти за время O(n)

Вариант реализации алгоритма Дейкстры	Насыщенный граф $m = O(n^2)$	Разреженный граф $m = O(n)$
D – массив	$T = O(n^2 + m) =$ $\mathbf{O(n^2)}$	$T = O(n^2 + m) =$ $\mathbf{O(n^2)}$
D – бинарная куча	$T = O(n\log n + m\log n) =$ $\mathbf{O(n^2\log n)}$	$T = O(n\log n + m\log n) =$ $\mathbf{O(n\log n)}$
D – фибоначчиева куча	$T = O(m + n\log n) =$ $\mathbf{O(n^2)}$	$T = O(m + n\log n) =$ $\mathbf{O(n\log n)}$

- **Насыщенный граф** (*dense graph*) – это граф, в котором количество ребер близко к максимально возможному

$$|E| = O(|V|^2)$$

$$D = \frac{2 \cdot 10}{6 \cdot (6-1)} = \frac{20}{30} = 0.67, \quad D > 0.5$$



- **Разреженный граф** (*sparse graph*) – граф, в котором количество ребер близко к количеству вершин в графе

$$|E| = O(|V|)$$

$$D = \frac{2 \cdot 6}{6 \cdot (6-1)} = \frac{12}{30} = 0.4, \quad D < 0.5$$