数据结构与算法

第 12 讲：最短路径

韩文弢

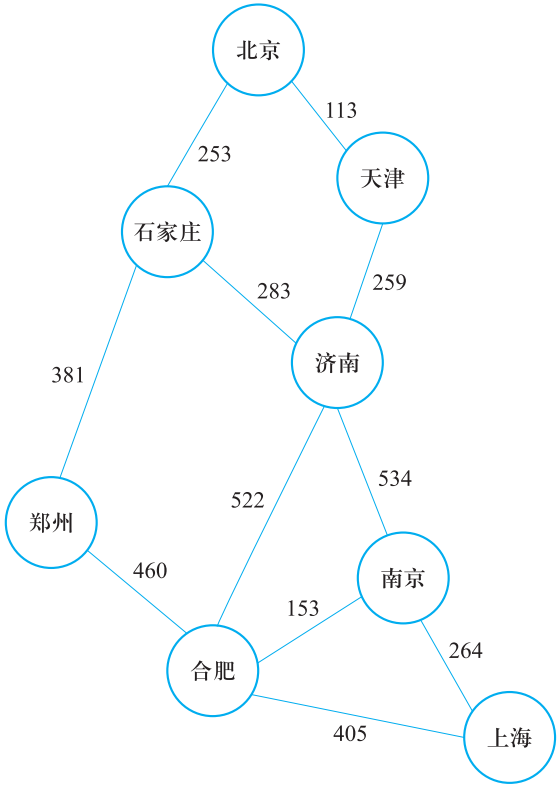
清华大学

2024—2025 学年度春季学期

本讲内容

12.1 问题引入

12.1.1 最短路径问题



左图是一张交通网络图，图中顶点表示城市，边表示城市之间的道路，边上的权值表示道路的长度（单位：千米）。从中找一条从北京到上海的最短路径，如何计算?

最朴素的方法是枚举从北京到上海的所有路径（如何枚举？考虑深度优先遍历的扩展），计算每条路径上边的权值之和，从中选取和最小的路径作为最短路径。

问题：路径数量太多，如何设计高效的算法?

12.1.2 问题定义

定义 1 给定一个加权有向图 $G = (V, E)$，令 $w(e)$ 表示边 e 的权值，路径 $P = (v_0, v_1, v_2, ..., v_k)$ 满足 $(v_i, v_{i+1}) \in E$，其中 $0 \leq i < k$，路径 P 的权值

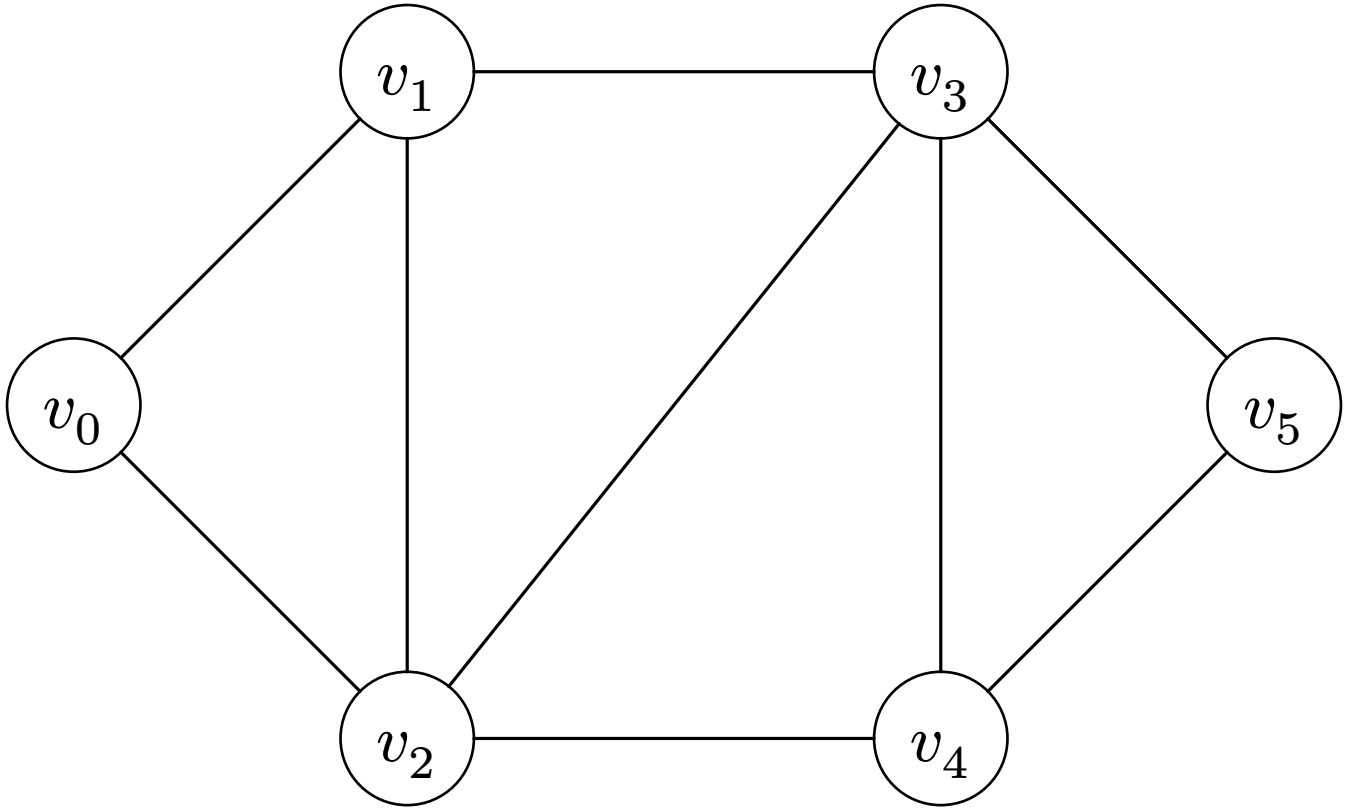
$$w(P) = \sum_{i=0}^{k-1} w((v_i, v_{i+1}))$$

图中两个顶点 $s, t \in V$ 之间权值最小的路径称为从 s 到 t 的**最短路径**。求解图中从某个顶点 s 到所有顶点最短路径的问题称为**单源最短路径**问题。无向图中也有类似定义。

12.2 无权图的最短路径

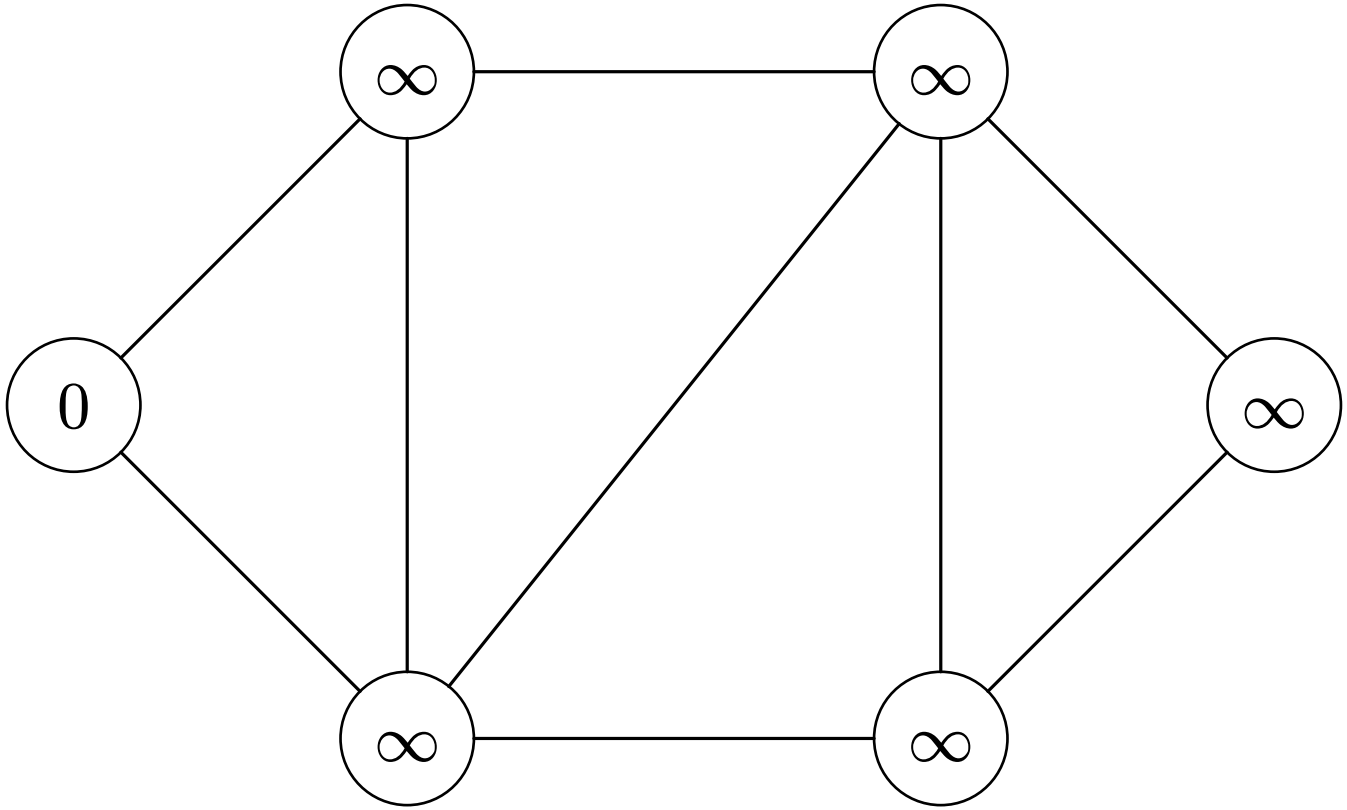
12.2.1 权值为 1 的情况

考虑边权均为 1 的情况（也就是无权图），求 v_0 和 v_5 之间的距离。



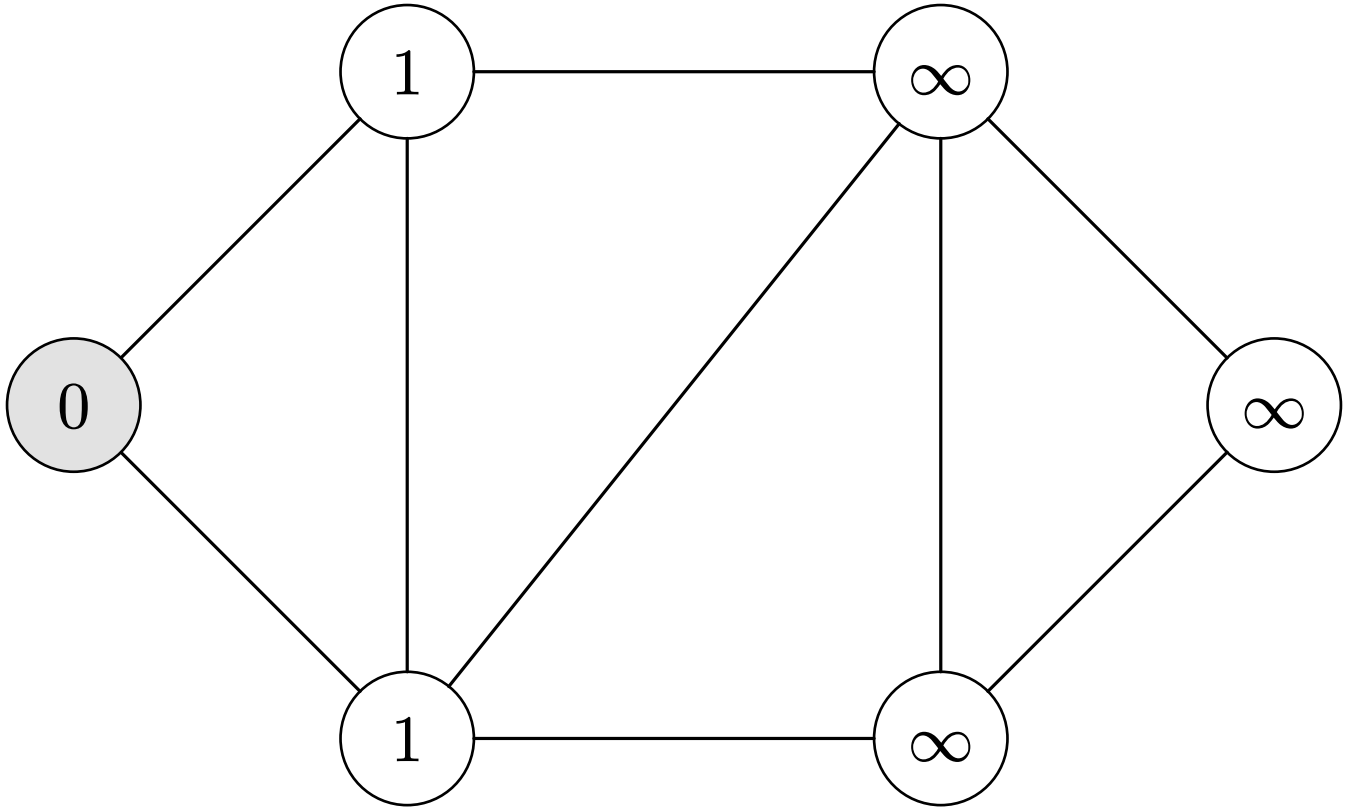
12.2.2 求解过程⓪

首先，v_0 的距离是 0，其余顶点的距离未知，用 ∞ 表示。



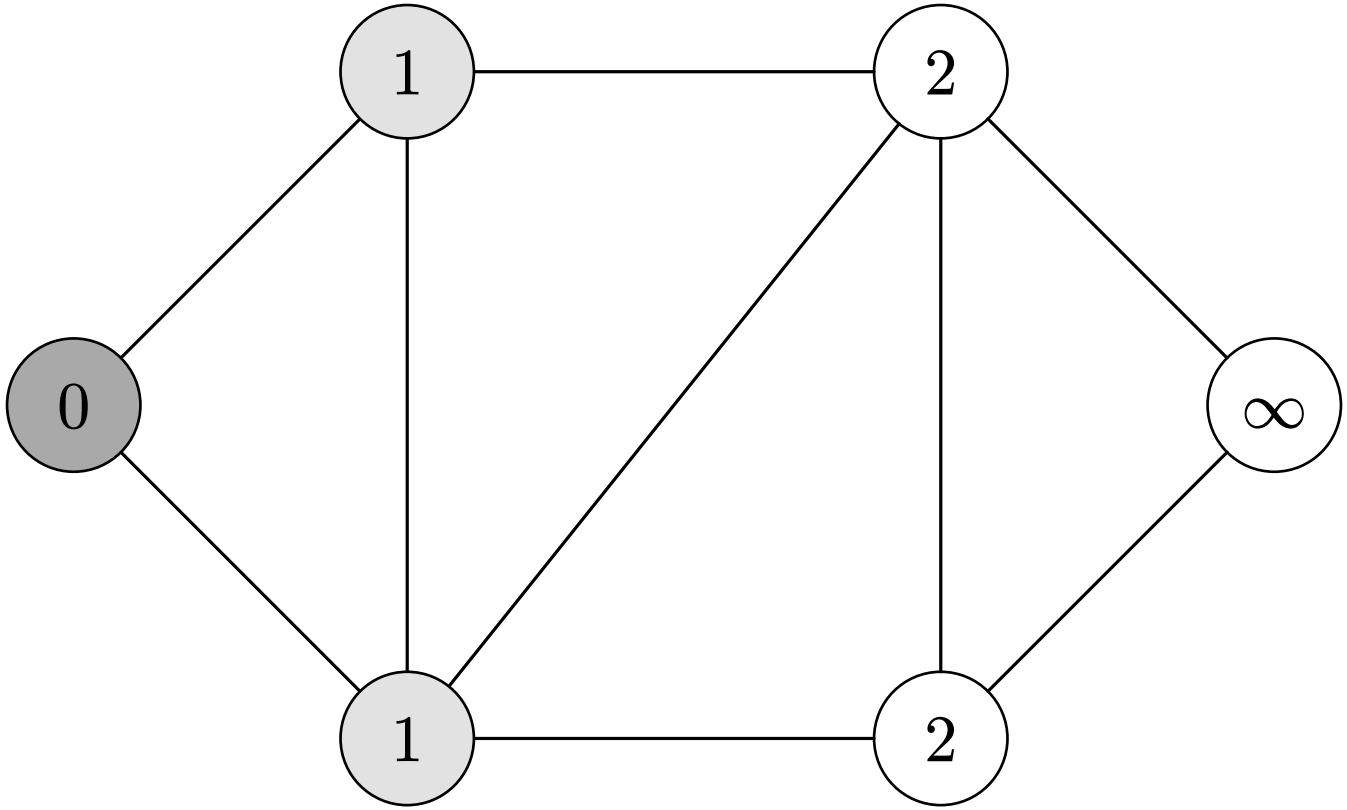
12.2.3 求解过程①

处理 v_0，从 v_0 可达 v_1 和 v_2，因此它们的距离都是 1。



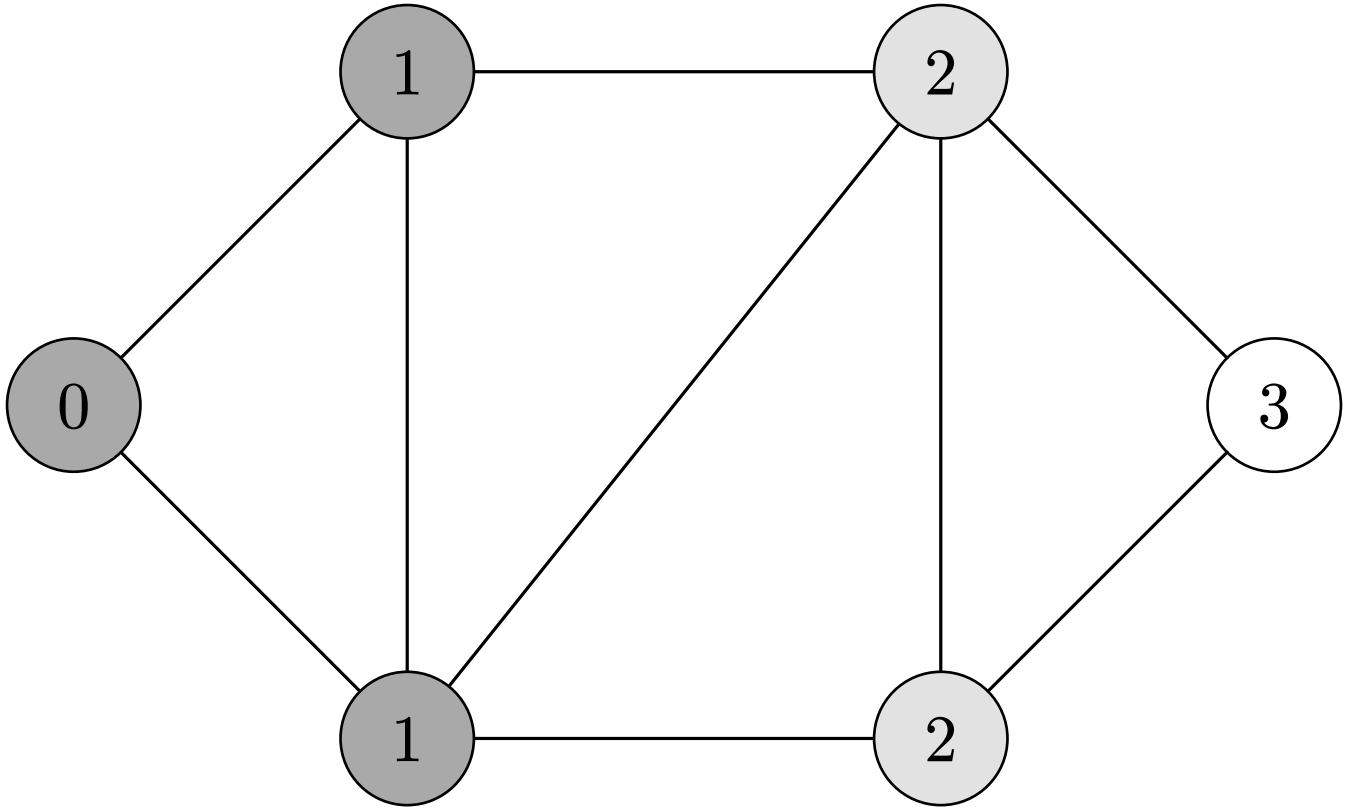
12.2.4 求解过程②

处理 v_1 和 v_2，可达 v_3 和 v_4，因此它们的距离都是 2。



12.2.5 求解过程③

处理 v_3 和 v_4，可达 v_5，因此 v_5 的距离是 3。



12.2.6 广度优先遍历

总结上述的访问过程：

1. 从某个未访问过的顶点 u 出发，访问该顶点，并标记该顶点已访问。
2. 对于顶点 u 的所有邻接顶点 $v_0, v_1, v_2, \ldots, v_{d-1}$，如果有顶点未访问过，则将其记录下来。
3. 取出目前记录且未访问的最早的顶点，重复 1。如果没有这样的顶点，过程结束。

这样的过程也构成遍历，称为**广度优先遍历**（breadth-first search, BFS）。广度优先遍历需要使用队列来辅助。

12.2.7 广度优先遍历算法

伪码 12.1: 图的广度优先遍历

```
BFS(G, u):
1   Initialize(Q)
2   level[u] ← 0
3   Enqueue(Q, u)
4   while ¬ IsEmpty(Q)
5       u ← Dequeue(Q)
6       输出访问顶点 u 的信息
7       for v ∈ N(u)   # N(u) 是与 u 邻接的顶点集合
8           if level[v] = ∞   # 顶点 v 尚未发现
9               level[v] ← level[u] + 1
10              Enqueue(Q, v)
```

12.2.8 无权图的最短路径算法

可以使用 BFS 来求解无权图的最短路径。

伪码 12.2: 无权图最短路径算法

```
BFSShortestPath(G, s, t):
1   for v ∈ V
2       level[v] ← ∞
3   BFS(G, s)   # 以顶点 s 为起始顶点进行遍历
4   return level[t]
```

12.2.9 广度优先遍历的时间复杂度分析

伪码 12.3: 图的广度优先遍历

```
BFS(G, u):
1   Initialize(Q)
2   level[u] ← 0
3   Enqueue(Q, u)
4   while ¬ IsEmpty(Q)   # 每个顶点一次，共循环 n 次
5       u ← Dequeue(Q)
6       输出访问顶点 u 的信息
7       for v ∈ N(u)   # 每条边一次，共循环 m 次
8           if level[v] = ∞
9               level[v] ← level[u] + 1
10              Enqueue(Q, v)
```

12.2.10 广度优先遍历的时间复杂度和应用

在邻接表表示的图上进行广度优先遍历的时间复杂度为 $O(n+m)$。

如果使用邻接矩阵，则时间复杂度为 $O(n^2)$。

广度优先遍历可用于判定两个顶点之间是否连通，还可以用来求解无权图的距离，以及无权无向图的直径等。

12.2.11 广度优先遍历求无权图距离

C++

```cpp
#include <iostream>
#include <queue>
#include <vector>

class Graph {
 public:
  // 输入一个图
  Graph() {
    int n, m;
    std::cin >> n >> m;
    adj_.resize(n);
    for (int i = 0; i < m; i++) {
      int u, v;
      std::cin >> u >> v;
```

12.2.11 广度优先遍历求无权图距离

```
15        // 无向图要插入两条反向的有向边
16        adj_[u].push_back(v);
17        adj_[v].push_back(u);
18      }
19    }
20    // 从顶点 u 出发进行一趟广度优先遍历
21    void BFS(int u) {
22      std::queue<int> q;
23      q.push(u);
24      level_[u] = 0;
25      while (!q.empty()) {
26        u = q.front();
27        q.pop();
28        for (int v : adj_[u]) {
```

12.2.11 广度优先遍历求无权图距离

```
29          if (level_[v] == -1) {
30            level_[v] = level_[u] + 1;
31            q.push(v);  // 扩展下一层顶点
32          }
33        }
34      }
35    }
36    // 求从 s 到 t 的距离
37    int Distance(int s, int t) {
38      int n = adj_.size();
39      level_.assign(n, -1);
40      BFS(s);
41      return level_[t];
42    }
```

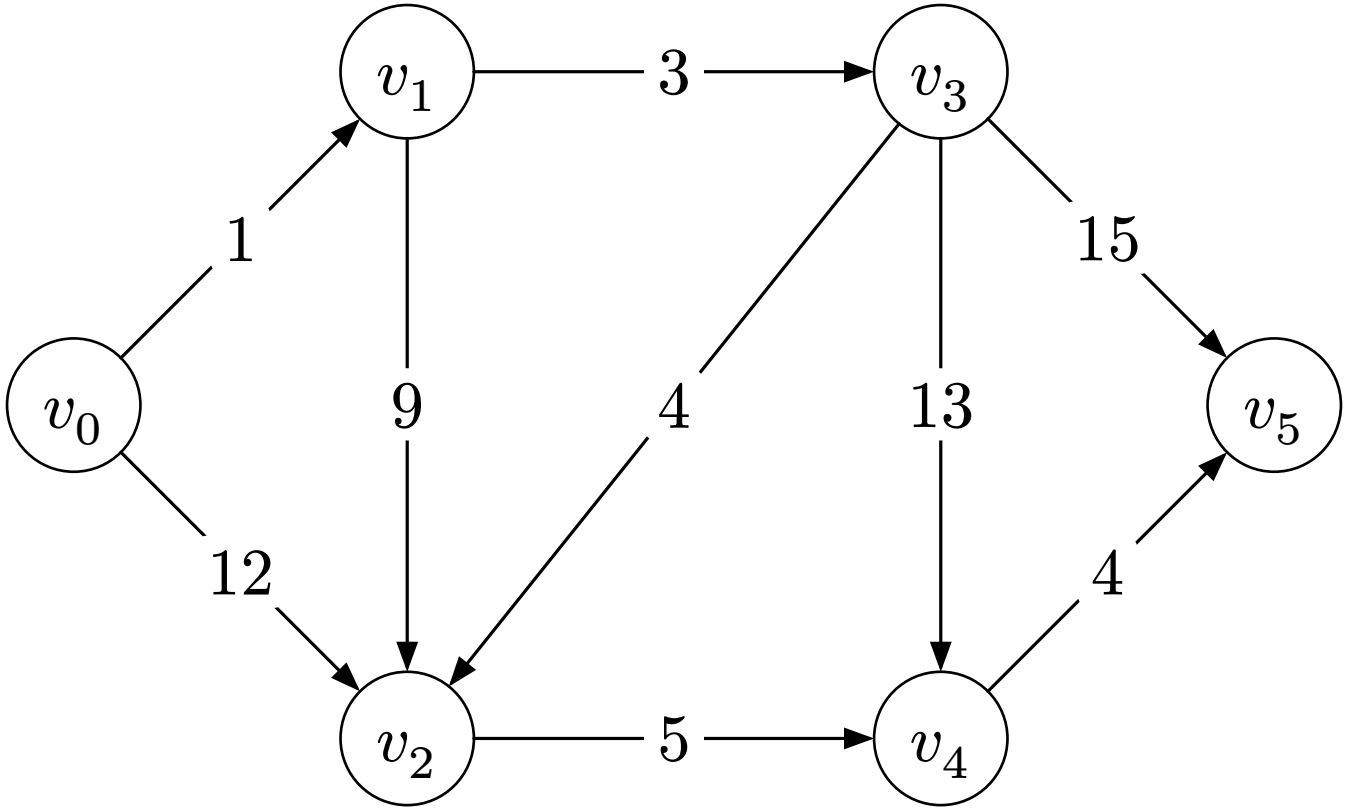
12.2.11 广度优先遍历求无权图距离

```
43
44  private:
45    std::vector<std::vector<int>> adj_;  // 邻接表
46    std::vector<int> level_;  // 从起点开始的距离，-1 表示未访问
47  };
48
49  int main() {
50    Graph g;
51    int s, t;
52    std::cin >> s >> t;
53    int d = g.Distance(s, t);
54    std::cout << d << std::endl;
55  }
```

12.3 加权图的最短路径

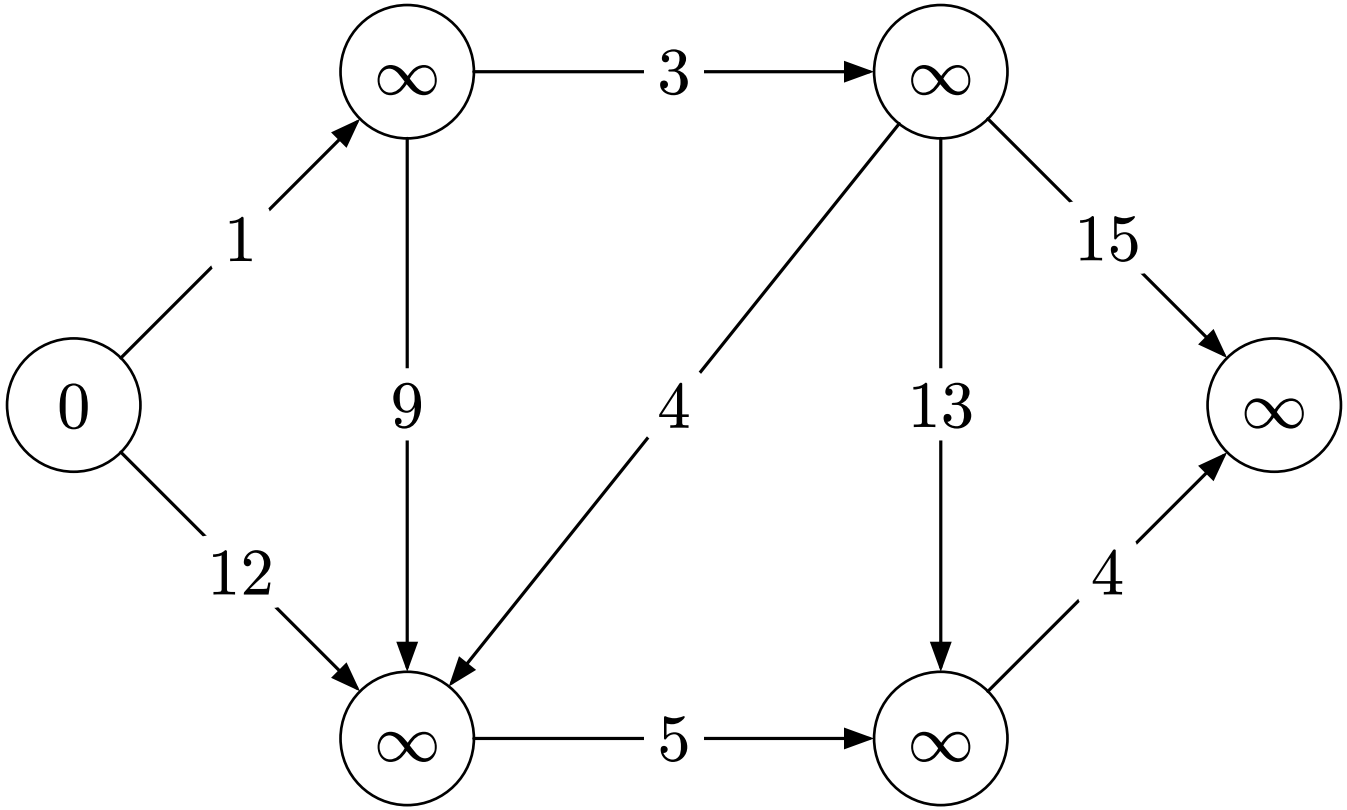
12.3.1 加权图最短路径示例

考虑加权图的情况，求从 v_0 到 v_5 的最短路径。



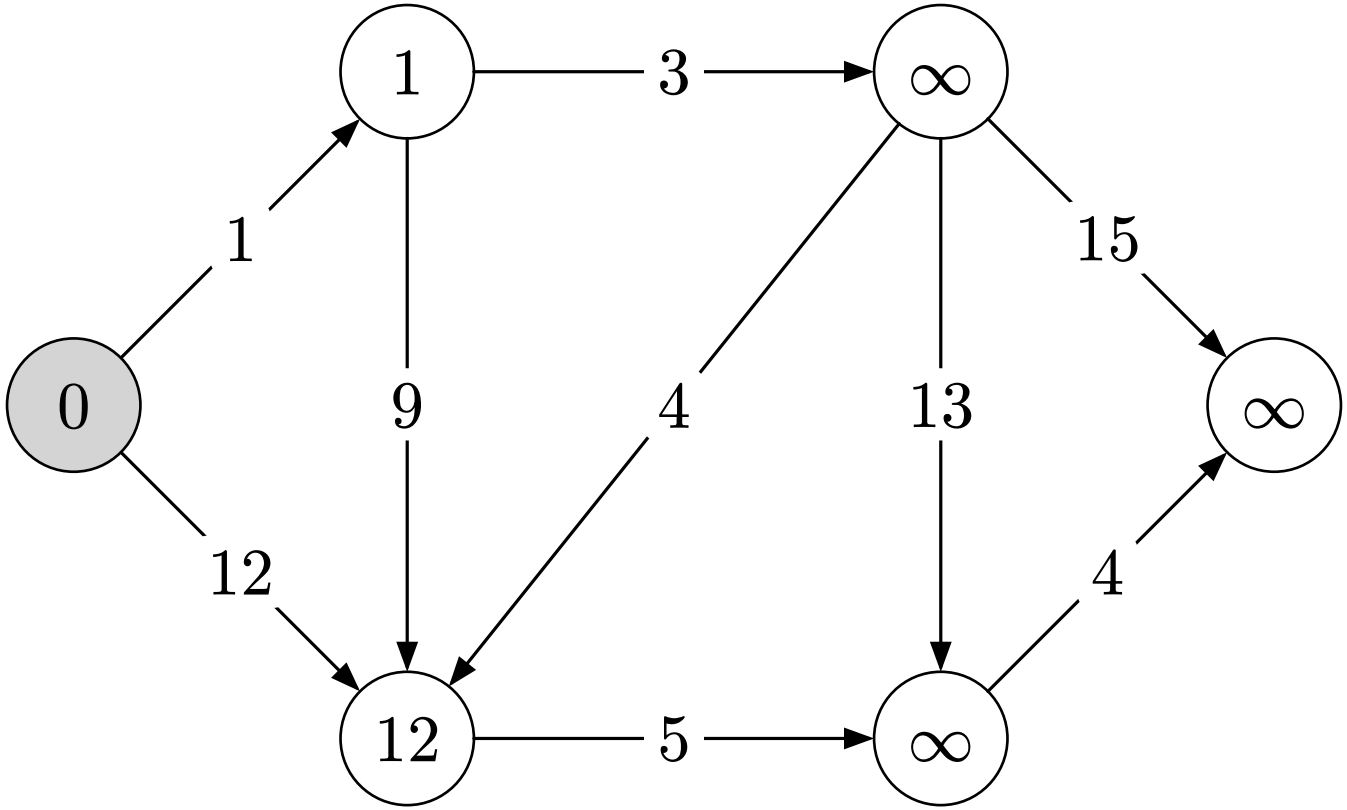
12.3.2 加权图最短路径示例⓪

一开始 v_0 的最短距离为 0，其余未知。



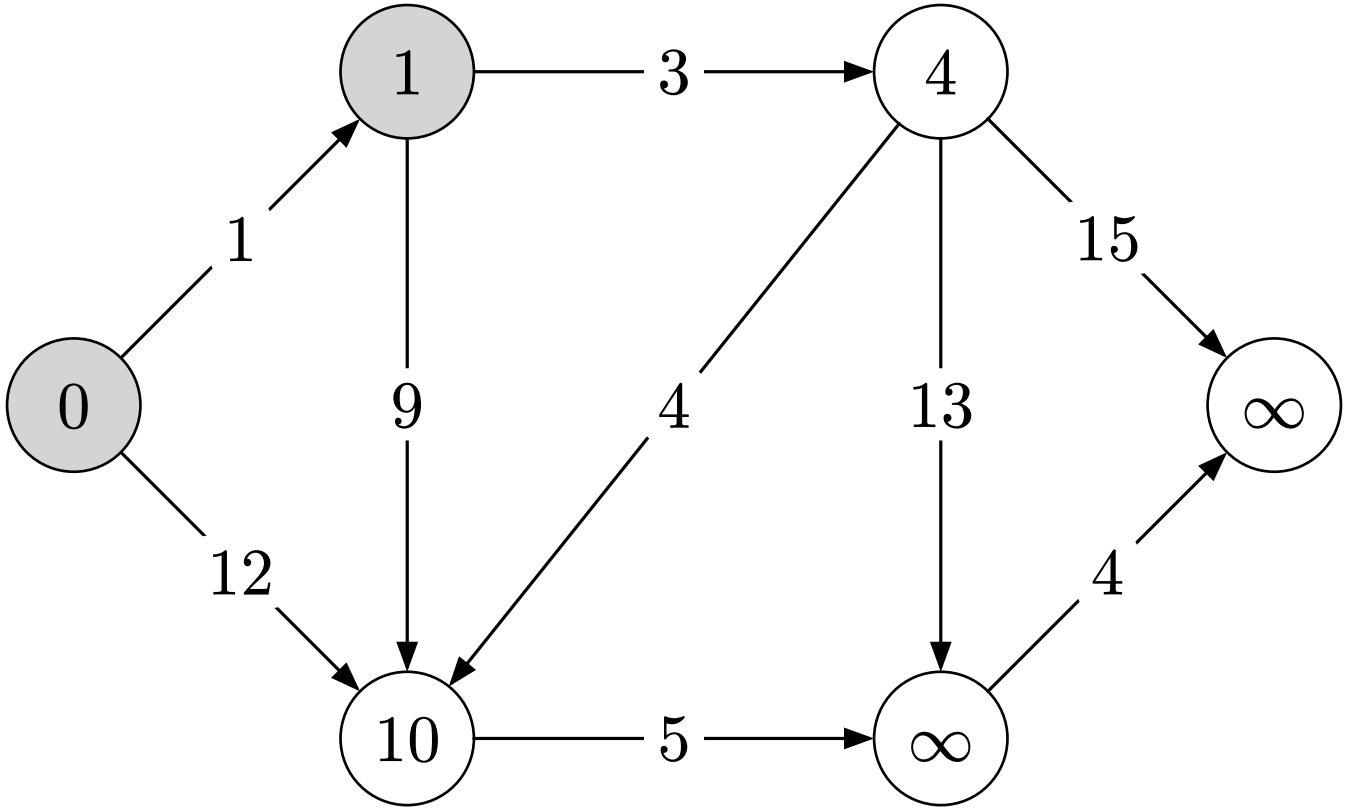
12.3.3 加权图最短路径示例①

v_0 的最短距离确定，更新 v_1 和 v_2 的最短距离。



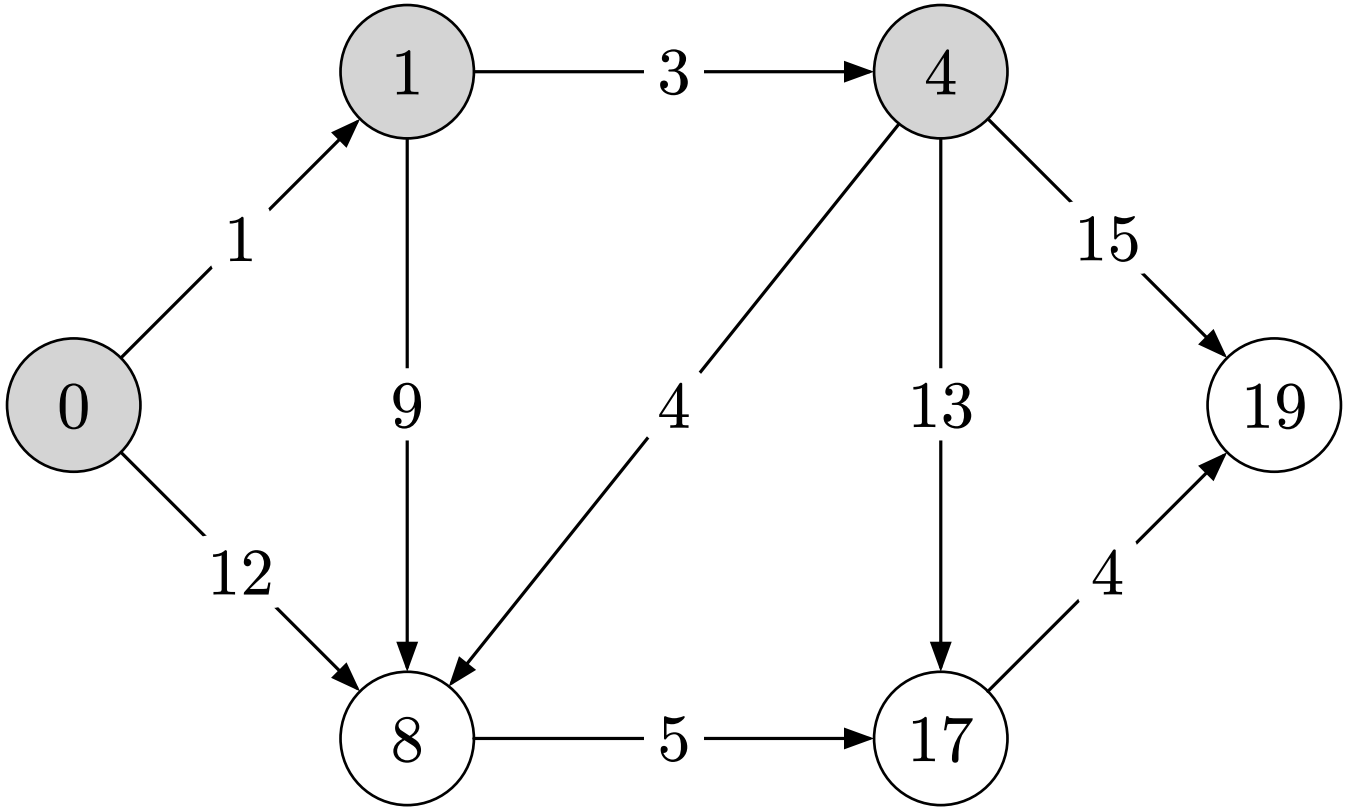
12.3.4 加权图最短路径示例②

v_1 的最短距离确定，更新 v_2 和 v_3 的最短距离。



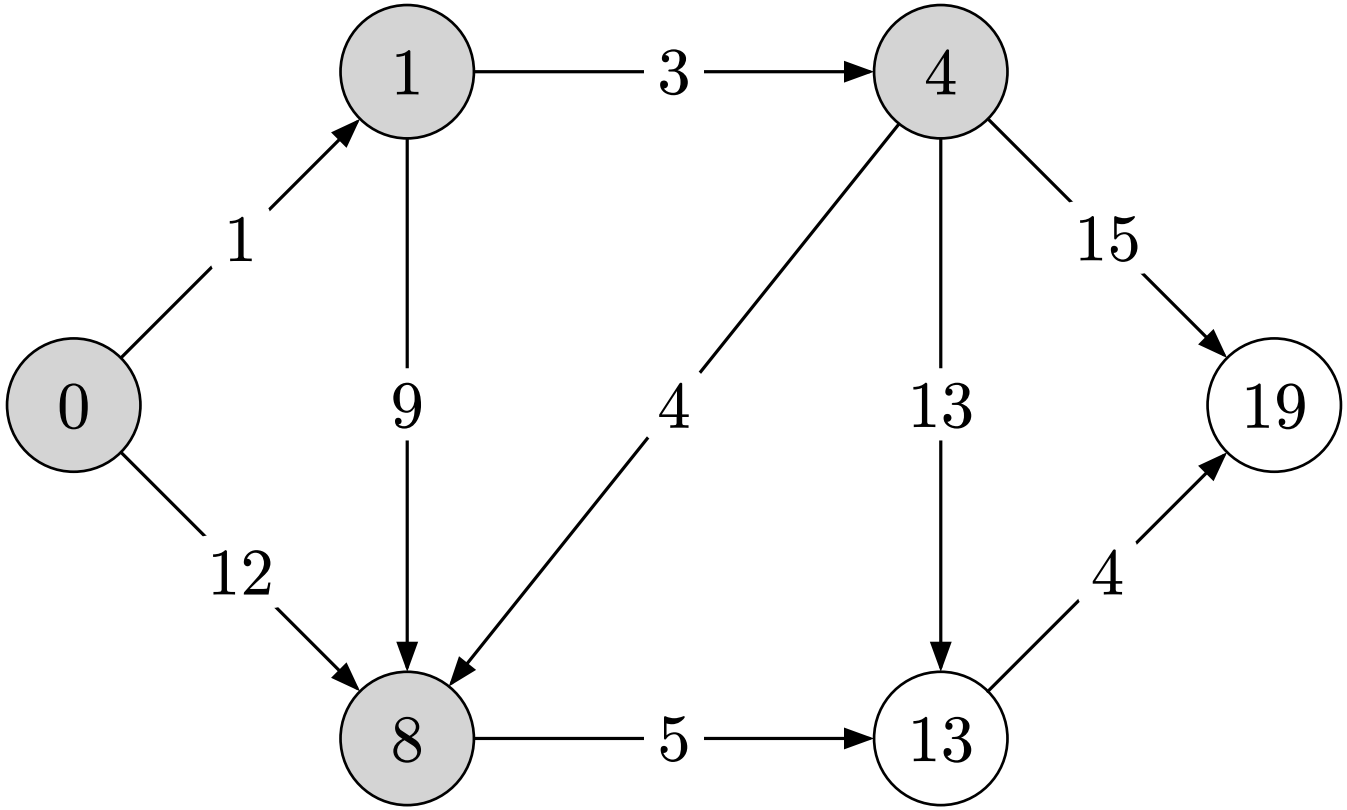
12.3.5 加权图最短路径示例③

v_3 的最短距离确定，更新 v_2、v_4 和 v_5 的最短距离。



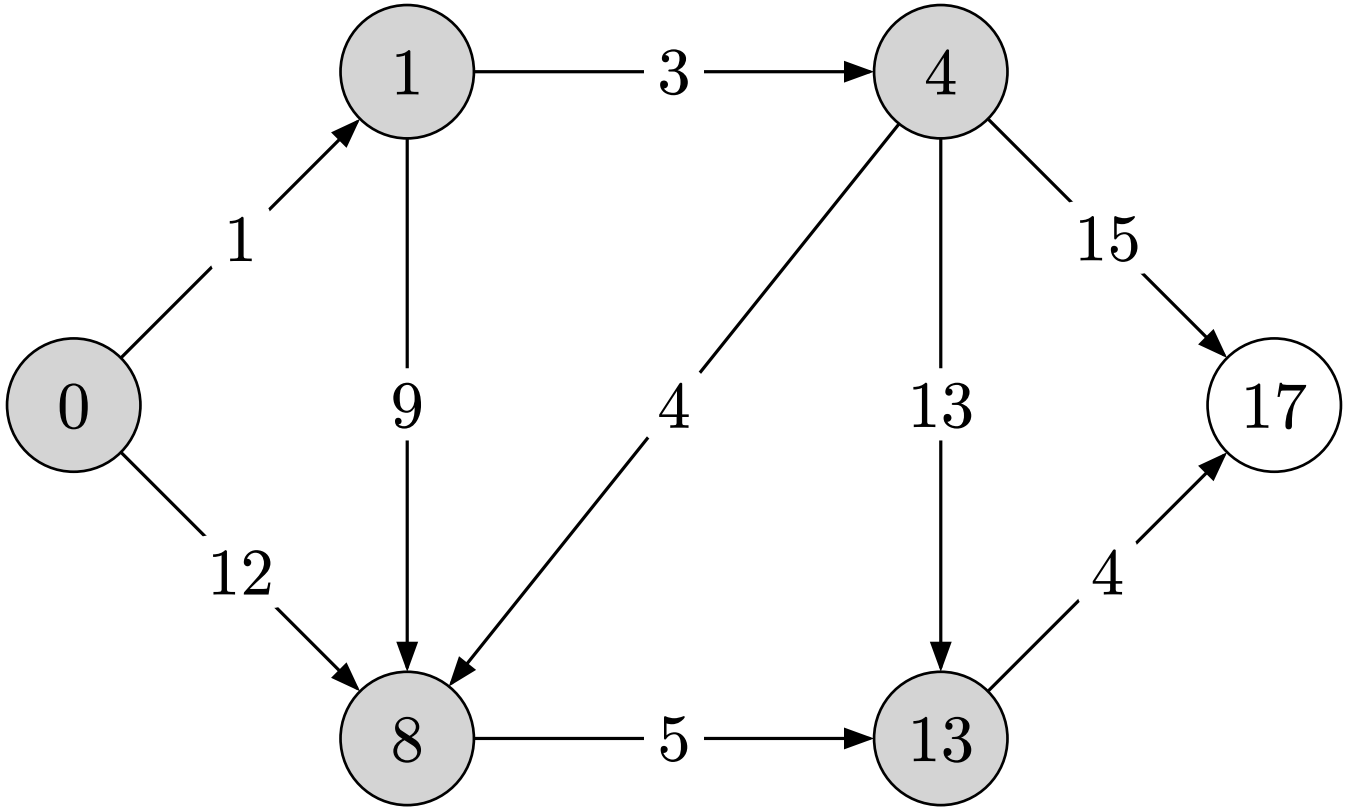
12.3.6 加权图最短路径示例④

v_2 的最短距离确定，更新 v_4 的最短距离。



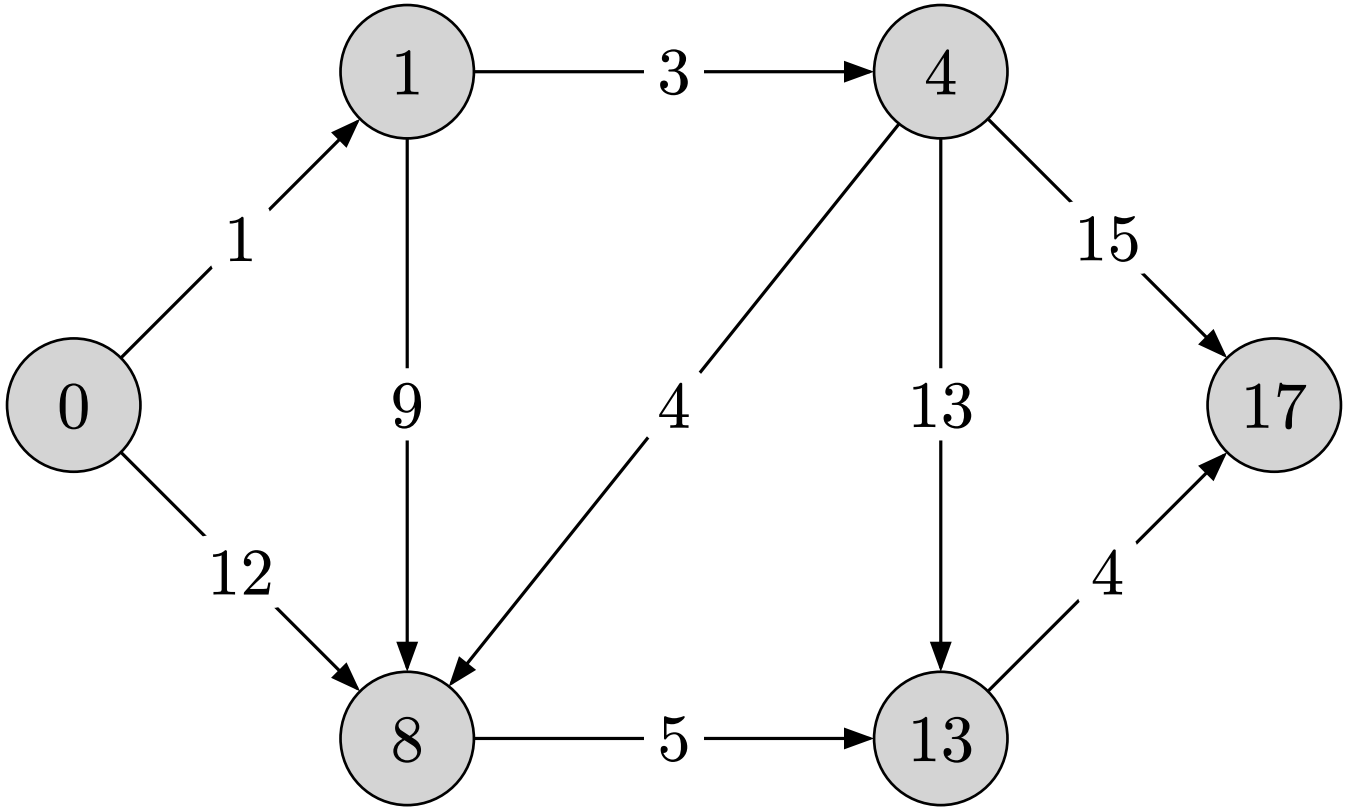
12.3.7 加权图最短路径示例⑤

v_4 的最短距离确定，更新 v_5 的最短距离。



12.3.8 加权图最短路径示例⑥

v_5 的最短距离确定，求解 v_0 到 v_5 的最短路径完成。



12.3.9 加权图最短路径的求解思路

荷兰计算机科学家 Edsger W. Dijkstra 在 1956 年提出了求解**边权非负**的单源最短路径问题的 Dijkstra（戴克斯图拉[1]）算法。

该算法的核心思想是：每次取出未确定的顶点中从起点开始距离最小的那个，再用该顶点的距离更新其他顶点的距离。

由于问题规定了边权非负的条件，因此每次取距离最小的顶点一定能保证该顶点的距离不会在之后的过程中变得更小。

[1]很多中文文献翻译成迪杰斯特拉算法，是错误的。荷兰语中 ij 合在一起是一个字母，发音类似于英语中的 y。

12.3.10 Dijkstra 算法

伪码 12.4: 边权非负单源最短路径 Dijkstra 算法

```
Dijkstra(G, s):
1   for v ∈ V
2       d[v] ← ∞
3   d[s] ← 0
4   F ← ∅
5   while F ≠ V
6       从 V \ F 中取出 d 最小的顶点 u，并加入 F
7       for v ∈ N(u)   # 更新 u 的邻接顶点
8           if d[u] + w((u, v)) < d[v]
9               d[v] ← d[u] + w((u, v))
```

12.3.11 Dijkstra 算法分析

正确性：可通过归纳法来证明，归纳假设为每次从 $V \setminus F$ 中取出 d 最小的顶点 u 时，u 的最短路径已经确定。通过 u 更新其邻接顶点 v 的距离，确保始终维护当前已知的最短路径估计。

获得具体的路径：在更新最短距离时维护 v 的最短路径前一个顶点是 u，则后续可以反向从终点 t 倒推到起点 s。

时间复杂度：取决于图的表示方式和 $V \setminus F$ 这个集合的实现方式。如果图用邻接矩阵实现，则时间复杂度为 $O(n^2)$；如果图用邻接表实现，且集合用堆来维护，则时间复杂度为 $O((n+m)\log n)$。注意用堆维护时，需要支持堆中元素变小的动态调整操作。

12.3.12 Dijkstra 算法的实现

C++

```cpp
#include <iostream>
#include <tuple>
#include <vector>

class Graph {
 public:
  // 读入加权图
  Graph() {
    int n, m;
    std::cin >> n >> m;
    adj_.resize(n);
    for (int i = 0; i < m; i++) {
      int u, v, w;
      std::cin >> u >> v >> w;
```

12.3.12 Dijkstra 算法的实现

```
15        adj_[u].push_back(std::make_tuple(v, w));
16      }
17    }
18    // 用 Dijkstra 算法求解最短路径
19    int ShortestPath(int s, int t) {
20      int n = adj_.size();
21      fixed_.assign(n, 0);
22      dist_.assign(n, -1);
23      prev_.assign(n, -1);
24      dist_[s] = 0;
25      while (true) {
26        // 寻找当前尚未确定的距离最小的顶点
27        int u = -1;
28        for (int v = 0; v < n; v++) {
```

12.3.12 Dijkstra 算法的实现

```
29          if (!fixed_[v] && dist_[v] != -1 &&
30              (u == -1 || dist_[v] < dist_[u])) {
31            u = v;
32          }
33        }
34        // 如果未找到，或找到了终点，则算法结束
35        if (u == -1 || u == t) break;
36        fixed_[u] = 1;
37        // 更新这个顶点的邻接顶点
38        for (const auto& [v, w] : adj_[u]) {
39          if (!fixed_[v] &&
40              (dist_[v] == -1 || dist_[u] + w < dist_[v])) {
41            dist_[v] = dist_[u] + w;
42            prev_[v] = u;
```

12.3.12 Dijkstra 算法的实现

```
43          }
44        }
45      }
46      return dist_[t];
47    }
48    // 打印路径上的顶点
49    void PrintPath(int s, int v) {
50      if (s == v) {
51        std::cout << s;
52      } else {
53        PrintPath(s, prev_[v]);
54        std::cout << ' ' << v;
55      }
56    }
```

12.3.12 Dijkstra 算法的实现

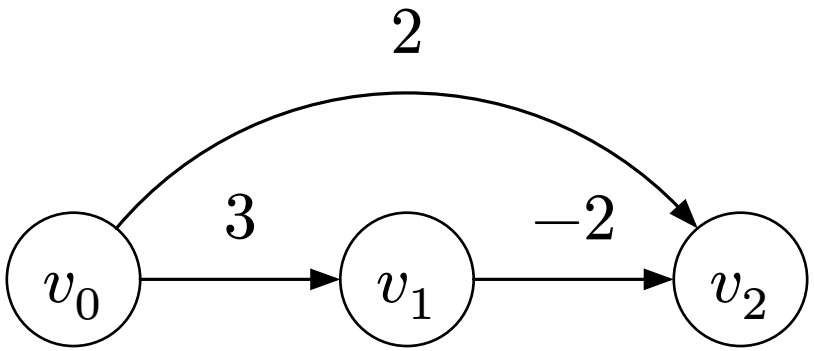
```
57
58  private:
59    std::vector<std::vector<std::tuple<int, int>>> adj_;  // 邻接表
60    std::vector<int> dist_;   // 从起点开始的最短距离
61    std::vector<int> prev_;   // 最短路径的上一个顶点
62    std::vector<int> fixed_;  // 最短距离是否已经确定
63  };
64
65  int main() {
66    Graph g;
67    int s, t;
68    std::cin >> s >> t;
69    int d = g.ShortestPath(s, t);
70    if (d != -1) {
```

12.3.12 Dijkstra 算法的实现

```
71      g.PrintPath(s, t);
72      std::cout << std::endl;
73    }
74  }
```

12.3.13 边权允许为负的最短路径问题

当边权允许为负时，Dijkstra 算法无法保证正确性。



此外，还有可能会出现权值为负的环，此时最短路径没有定义。

可以使用 Bellman-Ford 算法来求解边权允许为负的单源最短路径问题，能够检测是否存在负环。

12.4 最短路径问题的应用

12.4.1 无权图最短路径的应用

很多求解最佳步数的问题都可以通过建模转化为无权图的最短路径问题。

例. **过河问题**：一个人带了一条狗、一只羊和一筐菜要过河，河上只有一条小船，除人外，每次只能装一样东西。当人不在时，狗要咬羊，羊要吃菜。问如何带这三样东西安全过河，最少需要几个来回?

12.4.2 过河问题的思路

尝试用无权图来建模。

将出发一侧的对象集合视作状态，初始为 $\{人, 狗, 羊, 菜\}$。

集合本身和它的补集都必须是合法的。例如，$\{狗, 菜\}$ 是合法的，而 $\{人, 狗\}$ 不合法。

对于两个集合，如果通过一次过河能达到，则在两者之间连边。例如，$\{人, 狗, 羊, 菜\}$ 和 $\{狗, 菜\}$ 之间有边。

于是，问题就转化成无权图最短路径，目标是 $\emptyset$，可以用广度优先遍历来解决。

12.4.3 加权图最短路径的应用

无权图最短路径还可以用来求解华容道、魔方等各种步数相关的可行性和最优问题。

加权图的最短路径可以用来建模各种最优距离问题，例如地图导航（根据不同的需求，如距离最短、用时最少、换乘最少，可以设置不同的权重）。现实中大多数此类问题的权重都是非负的，因此可以用 Dijkstra 算法来高效求解。

12.5 小结

12.5.1 本讲小结

- 最短路径问题的定义
- 单源最短路径问题
- 广度优先遍历
- Dijkstra 算法