EDA 软件设计 I

Lecture 15

Check Course Site Frequently

最短路径问题定义

- **最短路径**：图中从一个起始节点（源节点）到另一个目标节点（汇节点）的**具有最小总权重**的路径
- **问题目标**：找到一条路径，使得沿路径经过的所有边的权重之和最小，从而实现最优的路径选择

"在图中寻找从一个节点到另一个节点的总权重最小的路径"

最短路径问题分类

① **单源**最短路径问题（Single-Source Shortest Path）
- 从**一个特定的源节点（source）**到图中**所有其他节点**的最短路径问题

② **单对**最短路径问题（Single-Pair Shortest Path）
- 从一个**特定的源节点**到一个**特定的目标节点**的最短路径问题

③ **全源**最短路径问题（All-Pair Shortest Path）
- 在图中**每对节点**之间的最短路径问题

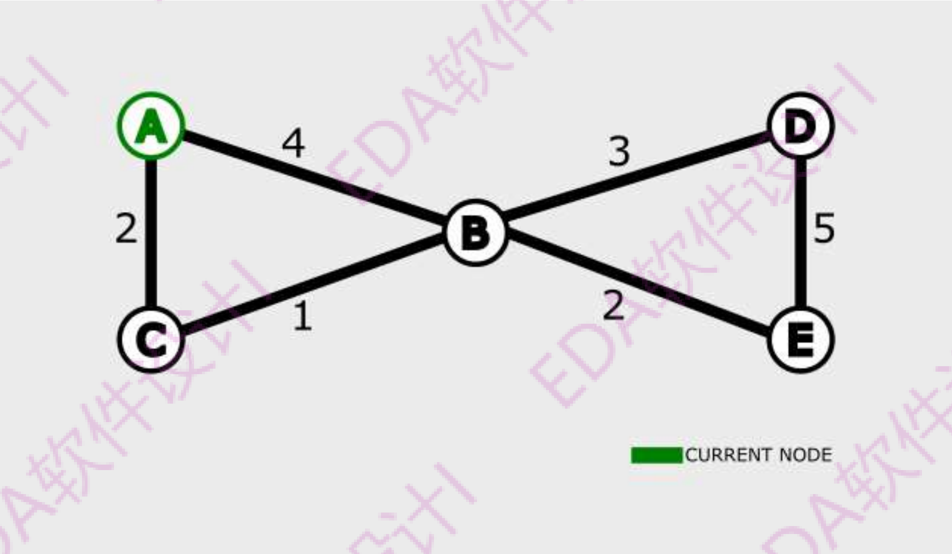
④ 限制条件最短路径问题（Constrained Shortest Path）

⑤ K最短路径问题（K-Shortest Paths）

Review: Dijkstra Algorithm

- Dijkstra算法是一种<u>单源最短路径算法</u>（**底层机制：Greedy**）
  - 更新：持续记录当前已知的最短路径，并在发现更短路径时进行更新——**dist[v] = min (dist[v], dist[u] + weight(u,v))**
  - Greedy地选下一节点: 距离源节点最近的未访问节点
  - 维护已访问：一旦确定了源节点与某个节点之间的最短路径，该节点将被标记为“已访问”，并被加入到路径中
  - 这个过程会不断重复，直到所有节点都被添加到路径中，从而获得从源节点出发访问所有其他节点的最短路径方案

Review: Run Dijkstra Manually

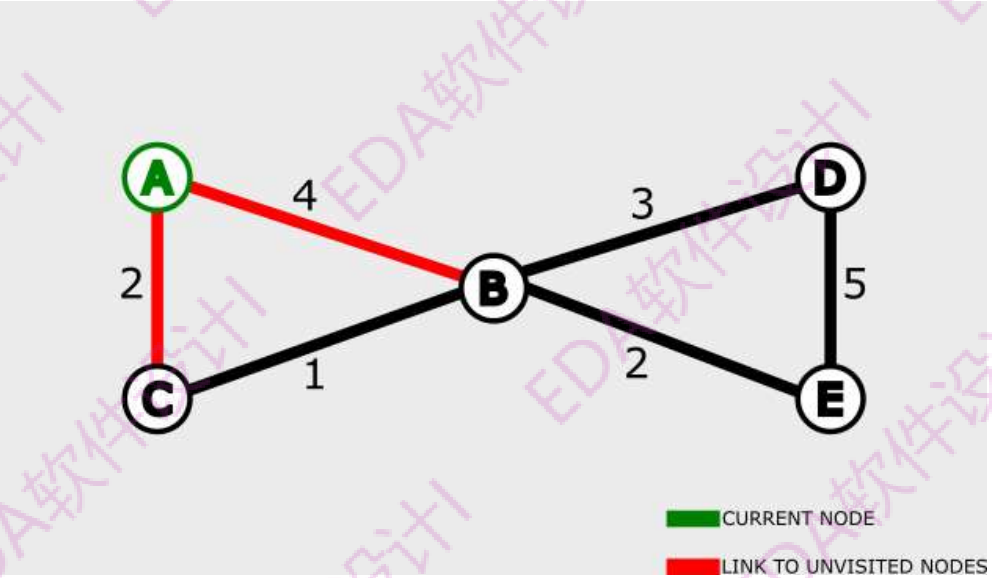


Node	Shortest distance from fixed node
A	0
B	∞
C	∞
D	∞
E	∞

① 对于当前节点u的所有未访问邻居v，更新其最短距离：**dist[v] = min (dist[v], dist[u] + weight(u,v))**

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

Review: Run Dijkstra Manually

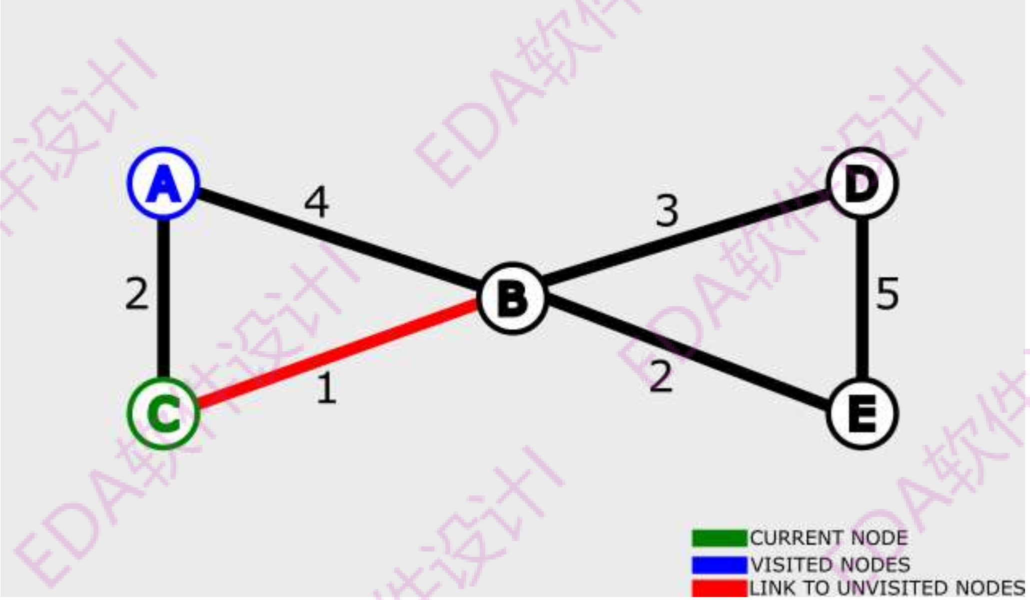


Node	Shortest distance from fixed node
A	0
B	4
C	2
D	∞
E	∞

① 对于当前节点u的所有未访问邻居v，更新其最短距离：
dist[v] = min (dist[v], dist[u] + weight(u,v))

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

Review: Run Dijkstra Manually



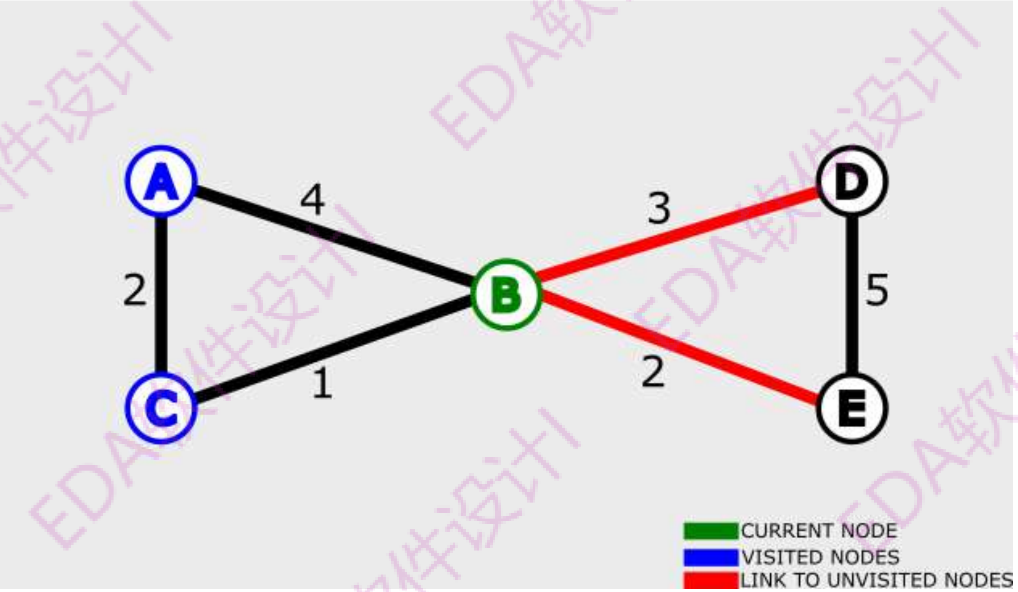
Node	Shortest distance from fixed node
A	0
B	3
C	2
D	∞
E	∞

① 对于当前节点u的所有未访问邻居v，更新其最短距离：
dist[v] = min (dist[v], dist[u] + weight(u,v))

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

A --> **B** = 4 (First iteration).
A --> **C** --> **B** = 3 (Second iteration).

Review: Run Dijkstra Manually



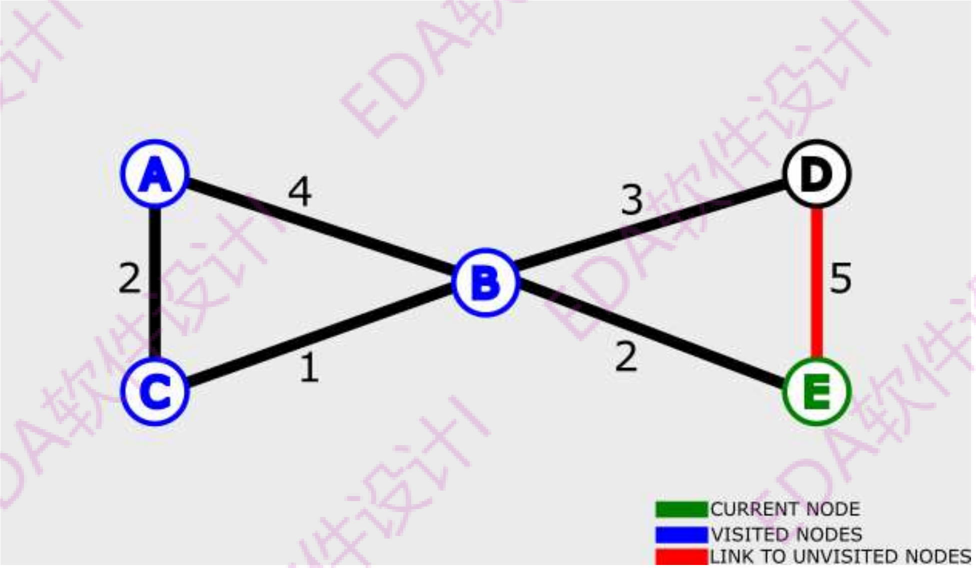
Node	Shortest distance from fixed node
A	0
B	3
C	2
D	6
E	5

For node **D**, 3 + 3 = 6.
For node **E**, 3 + 2 = 5.

① 对于当前节点u的所有未访问邻居v，更新其最短距离：
dist[v] = min (dist[v], dist[u] + weight(u,v))

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

Review: Run Dijkstra Manually



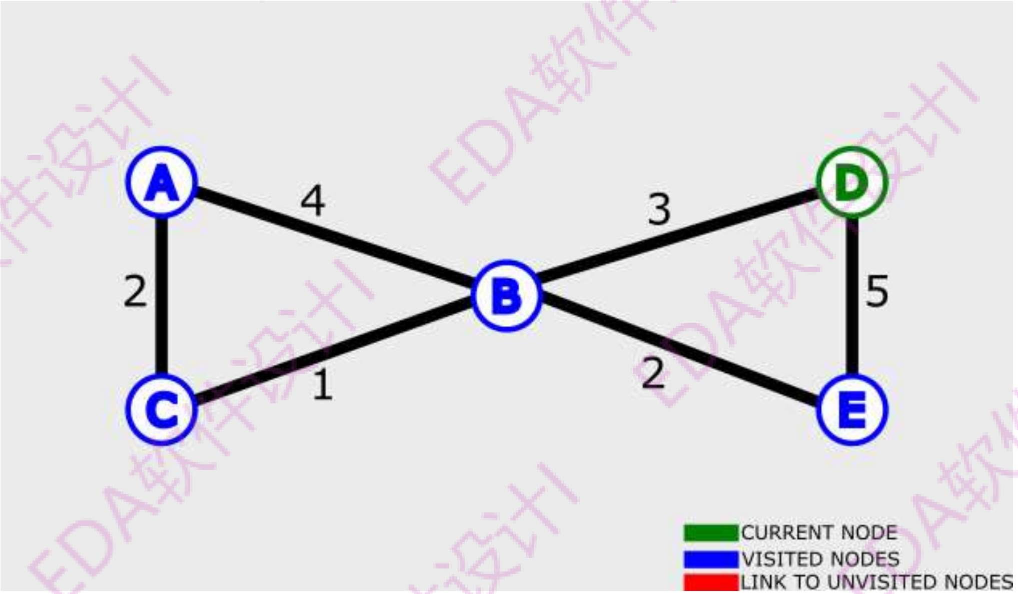
Node	Shortest distance from fixed node
A	0
B	3
C	2
D	6
E	5

For **D** in the current iteration,
5 + 5 = 10.

① 对于当前节点u的所有未访问邻居v，更新其最短距离：
dist[v] = min (dist[v], dist[u] + weight(u,v))

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

Review: Run Dijkstra Manually



Node	Shortest distance from fixed node
A	0
B	3
C	2
D	6
E	5

① 对于当前节点u的所有未访问邻居v，更新其最短距离：
dist[v] = min (dist[v], dist[u] + weight(u,v))

② 如何选取下一点：Greedy on 优先访问距离源节点最近的未访问节点

Dijkstra算法实现

Pseudocode：**一种不依赖具体编程语言的简洁描述，用于清晰地表达算法的核心逻辑和步骤**

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
        u ← vertex in Q with min dist[u]
        remove u from Q

        for each neighbor v of u still in Q:
            alt ← dist[u] + Graph.Edges(u, v)
            if alt < dist[v]:
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

初始化

主循环：

- 从**优先队列(Priority Queue)**中弹出距离最小的节点
- 对当前节点的每个邻居进行检查，比较通过当前节点到达邻居节点的路径是否比已有的路径更短
- 如果更短，则更新邻居的最短路径距离，并将邻居加入优先队列

优先队列（Priority Queue）：一种特殊的队列数据结构，其中每个元素都有一个优先级，取出元素时总是取优先级最高（或最低）的元素，而不是按插入顺序

Priority Queue的实现方式

① 数组

- 方法：将所有元素插入一个数组或列表中，然后每次取出时扫描整个数组以找到最小（或最大）元素
- 优点：实现简单，插入和取出逻辑直观
- 缺点：每次取出都需要遍历数组，时间复杂度为 $O(n)$ ，不适合大规模数据

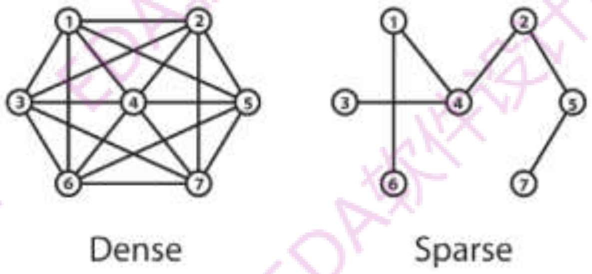
② 链表

- 方法：将所有元素插入到一个链表中，可以按照优先级排序插入，或在无序链表中按优先级查找
- 优点：有序链表可以加快取出速度
- 缺点：插入需要遍历链表或插入排序，效率不高，特别是在无序链表中插入和取出都需要 $O(n)$

③ 堆（Heap）：一种特殊的二叉树数据结构，通常用于实现优先队列。最小堆（Min-Heap）是优先队列的理想实现，因为它能够快速找到最小元素

- 方法：用最小堆实现优先队列，每次插入或删除时保持堆的结构，使堆顶元素为最小元素
- 插入元素复杂度和取出最小元素复杂度：$O(logn)$

Dijkstra局限性



1. **无法处理负权边**：图中如果出现负权边，算法失效
2. **对稠密图的效率较低**：算法的时间复杂度取决于图的边和节点数量，尤其在稠密图中，边的数量接近于节点数量的平方，使得运行时间较长
3. **对动态图的支持不足**：算法不适用于动态变化的图，尤其当图的边权或结构发生变化时，必须重新运行算法
4. **实现和优化上对数据结构的选择依赖较高**：算法通常**通过优先队列（如最小堆）来实现**，对于大型稀疏图，如果没有使用合适的数据结构（如斐波那契堆），算法的性能会大幅下降
5. **空间复杂度较高**：在大规模图上运行时，Dijkstra算法需要存储较多的节点和边的状态信息，可能会占用大量内存，不适合空间资源受限的系统

- 稠密图(dense graph)：E接近V^2
- 稀疏图(sparse graph)：E接近 V 或远小于V^2

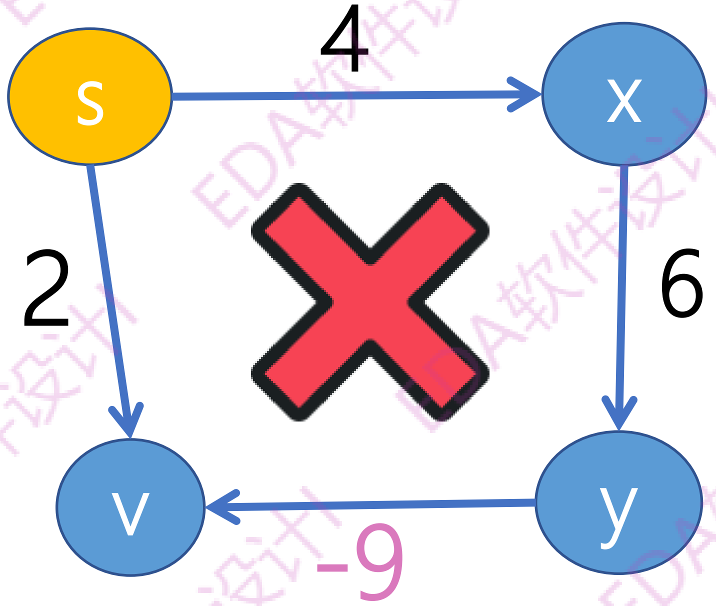
稠密图常用于节点间需要频繁直接交互的场景，比如：
社交网络
计算机网络
电路设计

负权边破坏Dijkstra正确性

- Dijkstra's Algorithm fails when there is negative edge

Ex: Dijkstra algorithm **selects vertex v immediately after s**

But **shorter path from s to v is $s \to x \to y \to v$**



负权边可出现在不同模型里面

- **交通网络**：某些路段因为交通管理措施（如限速或交通灯优化）导致通行时间减少。这些路段可以被建模为负权重
- **经济模型**：市场竞争模型中，企业之间的价格关系表示相对成本
- **电路设计**：在电路布局中，某些路径的延迟可能由于特定设计决策而降低
- **网络流问题**：流网络中，某些流量因为优惠政策而享受折扣（负权边）
- **社交网络**：权重表示关系的质量，负权重表示不良关系或损害关系的影响

正权边：延迟
负权边：由于设计优化导致的延迟减少

边 A → B（权重为 4）中，边的权重为4，表示企业1（A）相比于企业2（B）在价格上有一定劣势，或需要更高的成本
在边 B → C（权重为 -1）中，权重为负数，表示企业2（B）与企业3（C）之间存在激烈竞争，导致价格下降。

正权边：正常行驶时间
负权边：因优化措施而减少的时间

Bellman-Ford Algorithm
贝尔曼-福特算法

一种**可处理负权边**的单源最短路径算法

History of Bellman-Ford Algorithm

- Bellman-Ford算法的历史可以追溯到20世纪50年代，它由Richard Bellman和Lester R. Ford Jr.分别独立提出，因此被称为Bellman-Ford算法

Richard E. Bellman

Lester R.Ford ,Jr.

History of Bellman-Ford

- Richard Bellman的贡献:
  - 1956年，美国数学家Richard Bellman在研究动态规划时，提出了基于**"松弛"**操作的算法来求解最短路径问题。这一算法通过逐步更新路径长度的方式，能够处理负权重边，因此成为当时计算图最短路径的重要方法
  - Bellman的研究聚焦在**动态规划**的思想上，这种思想也成为Bellman-Ford算法的核心，使得算法能够多次迭代图中的边来更新最短路径

- Lester R. Ford Jr.的贡献:
  - 1958年，美国数学家Lester R. Ford Jr.在独立的研究中也提出了类似的算法。**Ford主要研究图论中的路径问题**，他的方法同样利用了**"松弛"**操作，并用于处理带有负权边的图，后来成为了Bellman-Ford算法的一部分
  - Ford的研究奠定了图论中的最短路径求解理论，为Bellman-Ford算法提供了理论基础

Bellman-Ford Algorithm

- 一种能够处理**含负权边**的图的最短路径算法，通过逐步松弛所有边（算法核心原理）来更新最短路径，同时还能检测图中是否存在负循环（若存在负循环，则无最短路径）

Negative edge weights

Negative cycles

什么是负循环（negative cycle）

- 负循环、负权环、负权回路
- 通常是针对有向图而言
- 定义：它是指在图中**一个回路（环）**的**边权总和为负值**的情况