EDA 软件设计 I

Lecture 12

基于DFS的拓扑排序实现

implementation

拓扑排序（DFS）实现细节

1. 初始化

- 创建一个用于记录节点是否被访问过的**数组或哈希表 visited，初始时每个节点的状态为未访问（False）**
- 创建一个空栈 stack，用于存储排序结果（栈用于记录回溯顺序，即拓扑排序结果）

Python知识点：

- **{}：dictionary**
- **[]：list**
- **dictionary的"组合技"（推导式）**

```python
visited = {node: False for node in graph}  # 初始化为未访问状态
stack = []  # 用来保存拓扑排序结果
```

拓扑排序（DFS）实现

2. DFS 函数定义

- 定义一个**递归的深度优先搜索函数** dfs(node)，用于递归访问每个节点及其相邻节点，在该函数中：
  - 将当前节点 node 标记为已访问
  - 递归访问该节点的所有邻居节点（即它指向的节点），如果某个邻居节点尚未被访问，递归调用 dfs()
  - **当该节点的所有邻居节点都访问完毕后，将该节点压入栈中**

```python
def dfs(node):
    visited[node] = True  # 标记当前节点为已访问

    # 递归访问所有相邻节点
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs(neighbor)

    # 当前节点的所有邻居都访问完毕后，压入栈中
    stack.append(node)
```

Python知识点：

- **List的append方法**
- **访问dictionary value的方法**

```python
# 示例图表示法
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['D'],
    'D': []
}
```

拓扑排序（DFS）实现

3. 遍历所有节点

- 遍历图中所有节点，**如果某个节点尚未被访问**，**则调用 dfs() 函数**开始对该节点及其邻居进行深度优先搜索

```python
for node in graph:
    if not visited[node]:
        dfs(node)
```

Python知识点：

- **函数的调用**
  - 作用域（Scope）：局部、闭包、全局、内置
  - 可变对象与不可变对象的参数传递

拓扑排序（DFS）实现

4. 返回结果

- （节点是通过回溯加入到栈中的，所以栈中的顺序正好是拓扑排序的逆序）在算法结束后，将栈进行反转即可得到最终的拓扑排序结果

Python知识点：

- **List的反转**

```python
topological_order = stack[::-1]  # 反转栈，得到拓扑排序结果
```

基于DFS的拓扑排序复杂度

Complexity

拓扑排序时间复杂度

遍历顶点的时间：对于每个节点，我们最多会访问一次，遍历的时间复杂度是 O(V)

遍历边的时间：在访问每个顶点时，DFS还会遍历该顶点的所有邻接节点，也就是遍历所有边，时间复杂度为 O(E)

若反转节点顺序得出拓扑顺序：O(V)
若通过链表前端记录拓扑顺序：O(1)

① Run DFS (for unvisited nodes)
- ◆ 对于图中还未访问的节点

② 在DFS中，用递归的方式处理节点，**回溯时记录节点顺序**

③ 反转（倒置）节点顺序 → 拓扑排序顺序

O（V+E）

熟悉不同数据结构基本操作的复杂度

拓扑排序空间复杂度

递归栈的使用：DFS是递归实现的，递归调用的深度最多为图中最长路径的长度，因此在最坏情况下，递归栈的深度为 O(V)

其他数据结构：我们还需要使用一个大小为 V 的栈来存储结果，以及一个大小为 V 的已访问数组

① Run DFS (for unvisited nodes)
- ◆ 对于图中还未访问的节点

② 在DFS中，用递归的方式处理节点，**回溯时记录节点顺序**

③ 反转（倒置）节点顺序 → 拓扑排序顺序

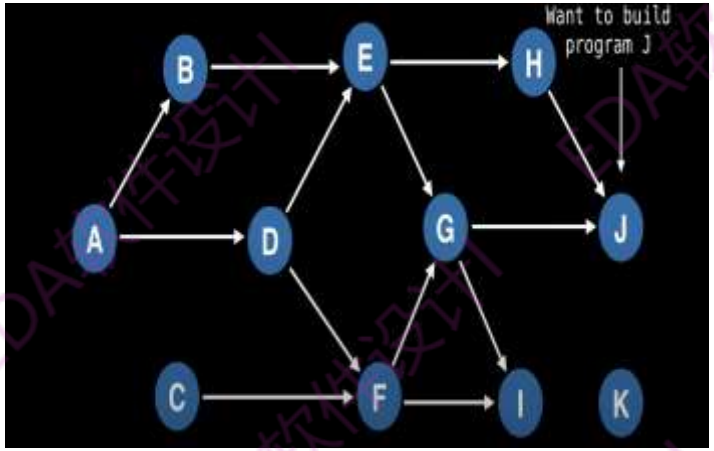
O（V）

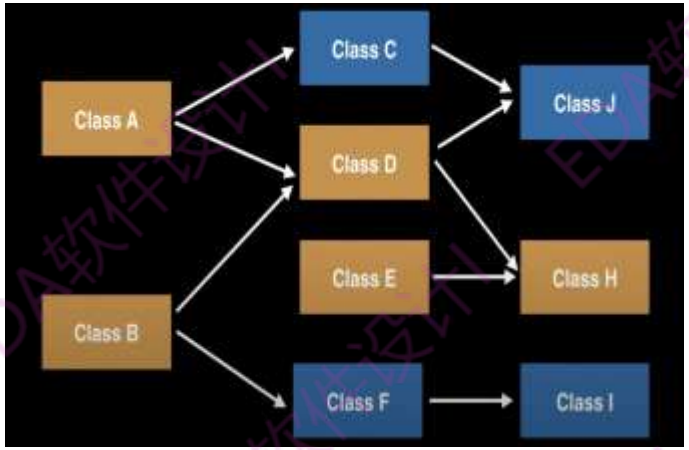
拓扑排序Kahn算法复杂度

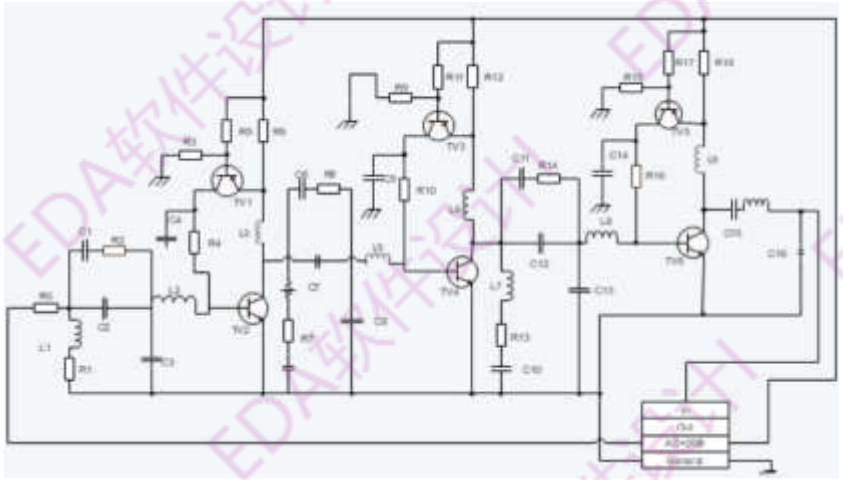
- 基于Kahn的拓扑排序
  - 时间复杂度
    1. 计算入度：首先要遍历所有边来计算每个顶点的入度，时间复杂度是 O(E)
    2. 处理顶点：然后对于每个顶点，它们会被加入队列并从队列中弹出，这个过程的时间复杂度是 O(V)
    3. 更新入度：每次从队列中弹出一个顶点后，需要更新其邻接节点的入度，这一步遍历所有边，时间复杂度为 O(E)
    4. 将这些部分结合，Kahn算法的总时间复杂度也是 O(V + E)
  - 空间复杂度
    1. 队列的使用：使用一个大小为 V 的队列来存储入度为0的节点
    2. 入度数组：需要维护一个大小为 V 的入度数组，这样我们可以方便地进行入度的更新和检查
    3. Kahn算法的空间复杂度为 O(V)，因为我们只需要存储节点和它们的入度，并且没有递归栈开销

拓扑排序的应用

领域或问题
任务调度
编译器
电路设计与分析
版本控制
解析表达式中依赖关系







拓扑排序的应用

领域或问题	典型场景	具体应用
任务调度	工厂生产中的生产步骤排序	产品的某些生产步骤必须按照先后顺序执行
编译器	依赖解析	构建工具（如make或CMake）使用拓扑排序来确定依赖的编译顺序，确保依赖关系的正确性
电路设计与分析	组合逻辑电路	确定电路中的信号传递路径，优化电路的设计和计算流程
版本控制	代码提交之间存在依赖关系，某些功能的合并必须在依赖功能合并之后完成	通过拓扑排序生成变更和合并的顺序，确保依赖关系不会出错，特别是在合并代码分支时
解析表达式中依赖关系	在解释器或编译器中，表达式可能有嵌套依赖，如 f(g(x))，需要先计算 g(x)，再计算 f	通过拓扑排序，确保复杂表达式中的依赖项按正确顺序计算

拓扑排序 Done

Question?

芯片设计全流程（Generally）

EDA参与物理设计阶段

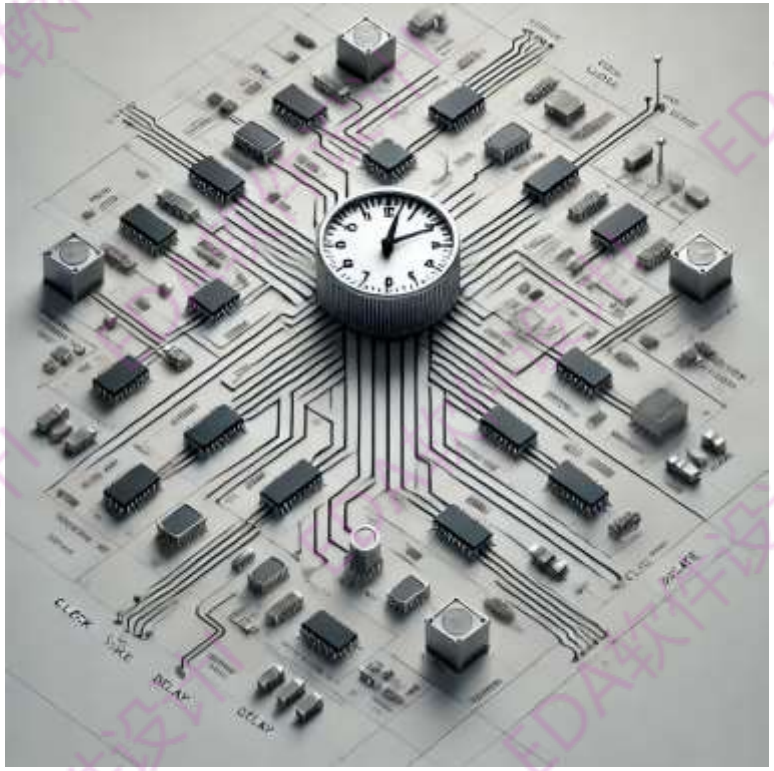
- 物理设计：
  - 布局布线
  - 时钟树综合
  - 信号完整性分析
  - 功耗分析

- **时钟树综合（Clock Tree Synthesis, CTS）**：负责为整个芯片中的各个电路模块同步传输信号
  - 时钟信号需要被精确地分配到芯片中所有需要同步的模块

时钟树入门

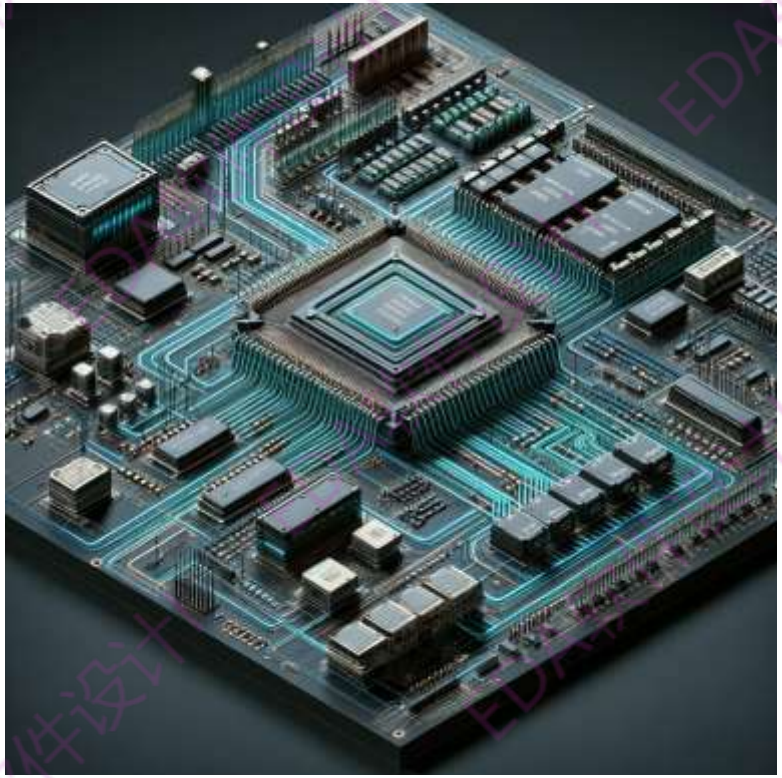
时钟树问题

- 在现代芯片中，时钟信号需要从一个时钟源（如Phase Locked Loop, PLL）传输到多个功能模块（寄存器、触发器等）
  - 这些模块可能分布在芯片的不同位置，因此时钟信号的传输路径需要经过精心设计
  - 时钟信号线组成树结构就是时钟树
- 时钟树需要保证所有模块能够在**同一时刻接收到时钟信号（时钟平衡）**，**时钟树综合就是建立一个时钟网络，使时钟信号能够传递到各个时序器件**
  - 由于布线的电容和延迟，布线的总长度直接影响到时钟延迟和抖动
- **目标: 在保证时钟延迟最小和时钟抖动可控的前提下，最小化时钟布线的总长度，从而减少功耗并提高时钟网络的性能**



布线问题

- 在芯片设计的物理设计阶段，**信号线布线**是非常关键的一步
  - 芯片中的各个电路模块之间需要通过金属连线进行信号传输，如何有效地规划这些连线对于芯片性能、功耗和制造复杂度至关重要
- 在布局布线中，**不仅时钟信号需要布线**，其他的信号线（如数据线、控制线等）也需要通过物理通道在芯片上连接多个电路模块
- 如果直接连接所有的模块，信号线的总长度可能非常长，导致延迟增加、功耗增大，甚至可能因为布线拥塞问题而导致制造难度加大



优化连接成本问题

- 时钟树问题、布局布线问题：**优化连接成本**问题
- 用最低的成本将所有节点连接起来：**最小生成树（Minimum Spanning Tree）**的目标
  - ◆连通
  - ◆最优化（min、max)

优化连接成本问题

1. 连通性问题

- **Connectivity:** 在很多实际问题中，我们需要确保一个系统中的所有节点能够通过某种方式相互连通，无论是构建通信网络、供电系统，还是道路网络，连通性都是最基础的要求
- **Weights**: 而在这些系统中，连接通常有一定的成本（如电缆的长度、铺设道路的成本、通信的延迟等）
- **Goal：** 我们往往需要找到一种最经济的方式来保证连通性，这就是最小生成树问题的根本动机

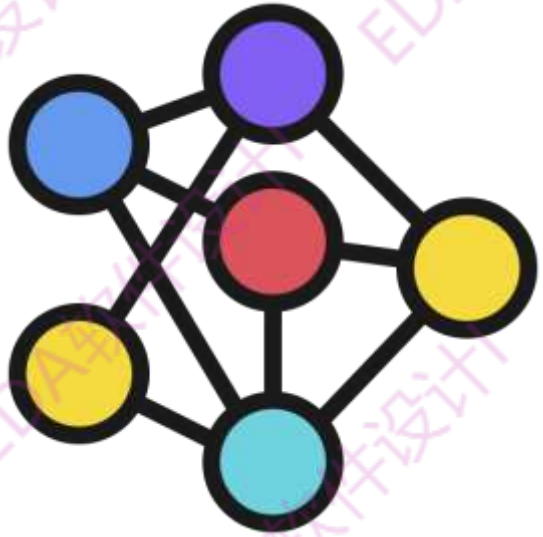
2. 资源优化问题

- 在网络设计或资源分配中，成本和资源（weights）的使用是非常关键的考量
- **Goal：** 在确保功能的前提下，**最小化成本**
  - 最小生成树就是在保证连通性的前提下，找到使用最少资源的网络结构

最小生成树应用场景1

网络设计（通信网络、计算机网络）

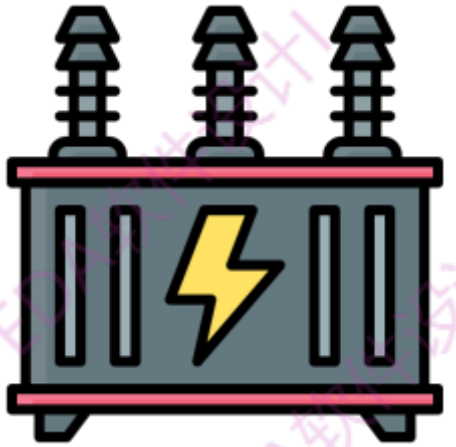
- 在通信网络设计中，最小生成树可以帮助我们**设计最优的网络拓扑结构**。例如：
  - **电话网络/互联网**：在连接不同城市或地区时，我们希望使用最少的光纤或电缆来连接各个地方，并保证所有的城市都能够互相通信
  - **局域网（LAN）设计**：在构建局域网时，最小生成树帮助确定如何连接不同的计算机或路由器，使得网络铺设成本最低且所有设备保持连通



最小生成树应用场景2

电力供应网络

- 在**供电网络**中，电力公司希望用**最少的电缆长度覆盖所有的城市或建筑**
  - 每个建筑物或城市是图中的节点，连接这些城市的电缆是图中的边，边的权重是电缆铺设的成本
  - 通过最小生成树，电力公司可以找到最经济的供电线路，同时保证每个地方都能被供电



最小生成树应用场景3

公路、铁路或交通网络

- 在规划公路或铁路时，我们希望使用**最少的建设成本连接不同的城市或地点**：
  - **公路系统**：在一个国家或地区，政府修建公路系统，使所有城市都互相连通，同时尽量降低建设费用。通过最小生成树，可以找到连接所有城市所需的最少的道路长度
  - **铁路网络**：在规划铁路网络时，最小生成树可以帮助选择哪些城市之间应该修建铁路，以便用最少的铁路长度连接所有城市