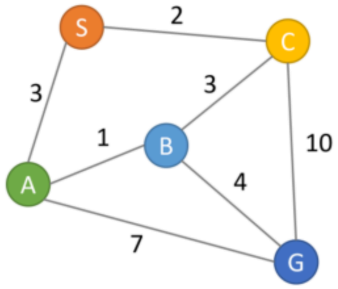
# 课上ppt上的伪代码切片

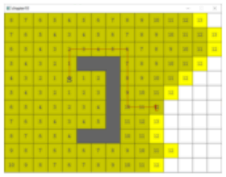
## 求s点到所有点的最短距离

- 对于任意两个有边相连的节点m、n，d(m,n)为边对应的权值
- 对于任意节点n，设定g(n)储存n到起点s的距离代价




迪杰斯特拉算法伪代码

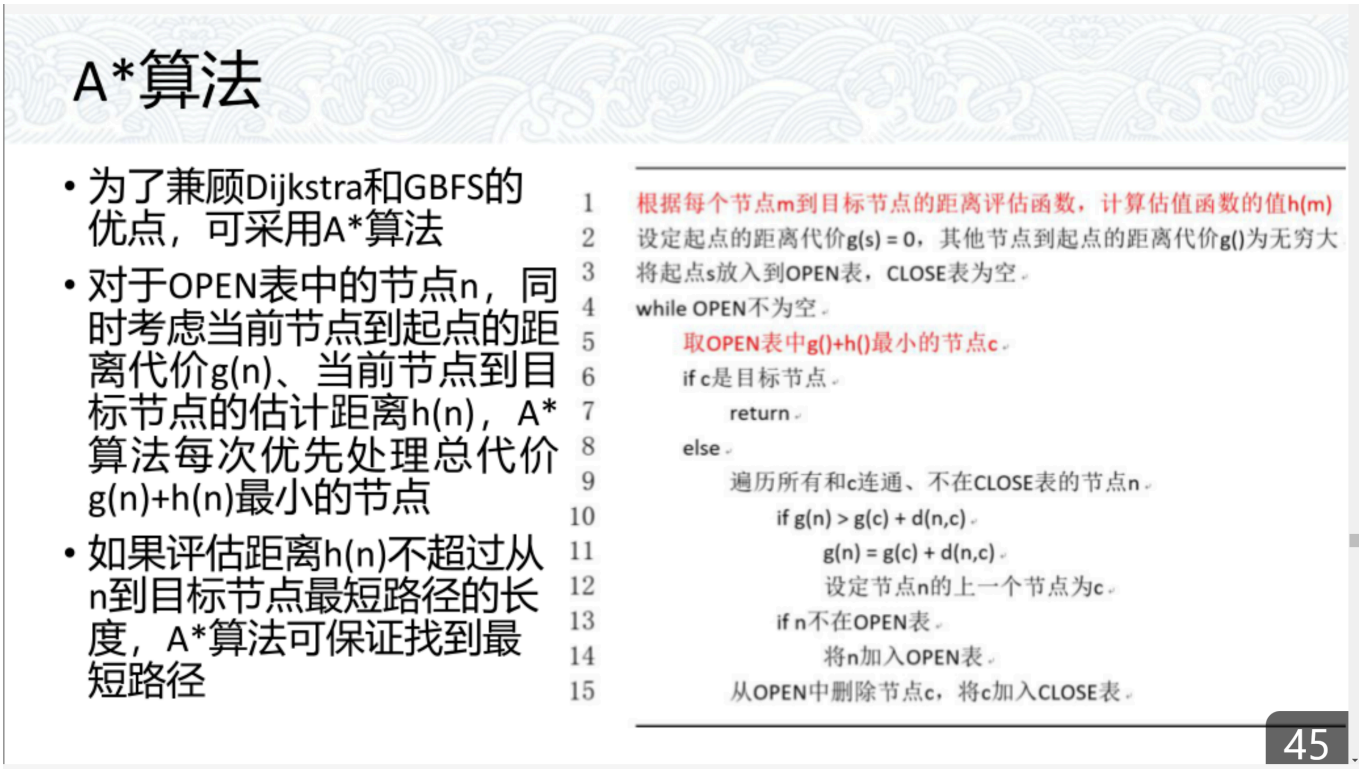
```
1  设定起点的距离代价g(s) = 0，其他节点到起点的距离代价g()为无穷大
2  将起点s放入到OPEN表，CLOSE表为空
3  while OPEN不为空
4      取OPEN表中距离代价g()最小的节点c
5
6
7      else
8          遍历所有和c连通、不在CLOSE表的节点n
9              if g(n) > g(c) + d(n,c)
10                 g(n) = g(c) + d(n,c)
11                 设定节点n的上一个节点为c
12             if n不在OPEN表
13                 将n加入OPEN表
14         从OPEN中删除节点c，将c加入CLOSE表
```



## 贪婪最佳优先算法

- 为了提高搜索效率，可以采用启发式的贪婪最佳优先算法（Greedy Best First Search，GBFS）
- 对于OPEN表中的节点，每次优先处理距离目标节点较近的节点

```
1  根据每个节点m到目标节点的距离评估函数，计算估值函数的值h(m)
2  设定起点的距离代价g(s) = 0，其他节点到起点的距离代价g()为无穷大
3  将起点s放入到OPEN表，CLOSE表为空
4  while OPEN不为空
5      取OPEN表中估值函数h()最小的节点c
6      if c是目标节点
7          return
8      else
9          遍历所有和c连通、不在CLOSE表的节点n
10             if g(n) > g(c) + d(n,c)
11                 g(n) = g(c) + d(n,c)
12                 设定节点n的上一个节点为c
13             if n不在OPEN表
14                 将n加入OPEN表
15         从OPEN中删除节点c，将c加入CLOSE表
```




# 总结

## 一、 核心概念：代价函数 (Cost Functions)

在理解这三个算法之前，必须先明确两个用于评估节点价值的基础变量：

- $g(n)$ **（实际代价）**：从起点 $s$ 沿着已知的路径走到当前节点 $n$ 的实际距离代价。
- $h(n)$ **（启发估值）**：从当前节点 $n$ 走到目标节点 $t$ 的预计距离（通常根据曼哈顿距离或欧几里得距离计算）。

---

## 二、 三大算法特性拆解

根据对 $g(n)$ 和 $h(n)$ 的不同使用策略，衍生出了幻灯片中的三种算法：

### 1. 迪杰斯特拉算法 (Dijkstra)

- **设计目的**：求起点 $s$ 到所有（或目标）点的最短距离。
- **优先级评估**：只考虑已走过的代价。每次优先处理 OPEN 表中 $g()$ **最小**的节点。
- **算法特性**：
  - 绝对公平的“广度优先”盲目搜索。
  - 只要图中的边权值为正，就**一定能保证**找到最短路径。
  - 搜索范围大，效率相对较低。

### 2. 贪婪最佳优先算法 (GBFS - Greedy Best First Search)

- **设计目的**：为了提高搜索效率，引入启发式策略。
- **优先级评估**：只考虑离终点有多近。每次优先处理 OPEN 表中 $h()$ **最小**（即估值函数最小）的节点。
- **算法特性**：
  - 极具方向性的“深度优先”贪婪搜索。
  - 在没有障碍物的情况下速度极快。
  - 容易陷入局部最优（比如死胡同），**不能保证**找到的是最短路径。

### 3. A* 算法 (A-Star)

- **设计目的**：兼顾 Dijkstra 的“最优性”和 GBFS 的“高效性”。
- **优先级评估**：综合考量实际代价与未来估值。每次优先处理 OPEN 表中总代价 $g(n)+h(n)$ **最小**的节点。
- **核心定理（幻灯片重点）**：如果评估距离 $h(n)$ **不超过**从 $n$ 到目标节点实际的最短路径长度（即 $h(n)$ 是乐观估算，绝不高估），A* 算法可**保证**找到最短路径。

---

## 三、 统一的算法伪代码框架

这三个算法在代码实现上几乎一模一样。你可以把它们看作同一个模板，只需要替换掉“取出节点”的那一行代码即可：

Plaintext

```
// 1. 初始化
计算所有节点的估值函数 h(m) （仅 GBFS 和 A* 需要）
设定起点距离代价 g(s) = 0，其他节点 g() = 无穷大
将起点 s 放入 OPEN 表，CLOSE 表为空

// 2. 主循环
while OPEN 表不为空:

    // 【核心区别点】决定下一步探索哪个节点 c ----------------------
    // Dijkstra: 取 OPEN 表中 g() 最小的节点 c
    // GBFS:     取 OPEN 表中 h() 最小的节点 c
    // A*:       取 OPEN 表中 g() + h() 最小的节点 c
    // ------------------------------------------------------------

    // 3. 终止条件
    if c 是目标节点:
```

```
        return 寻路成功

    // 4. 扩展节点
    else:
        遍历所有和 c 连通、且不在 CLOSE 表的相邻节点 n:

            // 发现更短的路径到达 n
            if g(n) > g(c) + d(n,c):
                g(n) = g(c) + d(n,c)  // 更新 n 的实际代价
                设定节点 n 的上一个节点为 c // 记录路径

                if n 不在 OPEN 表:
                    将 n 加入 OPEN 表

        // 5. 节点 c 处理完毕
        从 OPEN 中删除节点 c，将 c 加入 CLOSE 表
```

# 注水问题


## 方法一：先装满 3 加仑水壶

这种方法通过不断用 3 加仑的水壶往 4 加仑的水壶里“凑”水来求解。

1. **装满 3 加仑水壶**。（状态：3加仑壶里有 **3**，4加仑壶里有 **0**）
2. **将 3 加仑水壶里的水全部倒入 4 加仑水壶**。（状态：3加仑壶里有 **0**，4加仑壶里有 **3**）
3. **再次装满 3 加仑水壶**。（状态：3加仑壶里有 **3**，4加仑壶里有 **3**）
4. **用 3 加仑水壶里的水去倒满 4 加仑水壶**。因为4加仑壶里已经有3加仑了，所以只能倒进去1加仑。此时，3加仑水壶里剩下的水刚好是 2 加仑。（状态：3加仑壶里有 **2**，4加仑壶里有

**4**）
5. **清空 4 加仑水壶**。（状态：3加仑壶里有 **2**，4加仑壶里有 **0**）
6. **将 3 加仑水壶里剩下的 2 加仑水倒入 4 加仑水壶**。（状态：3加仑壶里有 **0**，4加仑壶里有 **2**）

**成功！**

---

## 方法二：先装满 4 加仑水壶

这种方法通过不断用 4 加仑的水壶往 3 加仑的水壶里倒水来求解。

1. **装满 4 加仑水壶**。（状态：3加仑壶里有 **0**，4加仑壶里有 **4**）
2. **用 4 加仑水壶去倒满 3 加仑水壶**。4加仑壶里会剩下1加仑。（状态：3加仑壶里有 **3**，4加仑壶里有 **1**）
3. **清空 3 加仑水壶**。（状态：3加仑壶里有 **0**，4加仑壶里有 **1**）
4. **将 4 加仑水壶里剩下的 1 加仑水倒入 3 加仑水壶**。（状态：3加仑壶里有 **1**，4加仑壶里有 **0**）
5. **再次装满 4 加仑水壶**。（状态：3加仑壶里有 **1**，4加仑壶里有 **4**）
6. **用 4 加仑水壶去倒满 3 加仑水壶**。因为3加仑壶里已经有1加仑了，所以只需要倒进去2加仑就能装满。此时，4加仑水壶里剩下的水刚好是 2 加仑。（状态：3加仑壶里有 **3**，4加仑壶里有 **2**）

**成功！**
如果你平时在洛谷等平台上刷过算法题，对这类过程应该很熟悉。我们可以把当前两个水壶的水量抽象为一个二维状态 `(x, y)`。系统允许进行 6 种合法的状态转移操作：

- 装满 x 或 y
- 清空 x 或 y
- 把 x 倒入 y（直到 y 满或 x 空）
- 把 y 倒入 x（直到 x 满或 y 空）

将这个经典的逻辑问题转化为代码，最核心的思想就是构建一个**状态空间（State Space）**，然后在这个空间中寻找从初始状态到目标状态的**最短路径**。

对于求解最少操作步数的问题，**广度优先搜索（BFS）** 是最经典且最稳妥的算法选择。它能保证找到的第一个解就是步骤最少的解，这在处理图论或隐式图搜索时非常有效。

下面是求解这个问题的伪代码。我们把两个水壶当前的水量定义为一个状态 `(current_A, current_B)`。

Plaintext

```
// 定义主搜索函数
// capacity_A = 3, capacity_B = 4, target = 2
function BFS_WaterJug(capacity_A, capacity_B, target):

    // 初始化队列用于 BFS，存储：[当前状态, 记录的历史操作路径]
    queue = new Queue()

    // 使用哈希集合记录已经访问过的状态，防止死循环（例如倒过去又倒回来）
    visited = new Set()

    // 初始状态：两个水壶都是空的
    start_state = (0, 0)
    queue.push( (start_state, []) )
    visited.add(start_state)

    // 开始广度优先搜索
    while queue is not empty:
        current_state, path = queue.pop()
        current_A, current_B = current_state

        // 检查是否满足终止条件（任何一个水壶达到目标水量）
        if current_A == target or current_B == target:
            return path  // 找到解，返回操作路径

        // 获取当前状态下，所有合法的下一步状态及对应的操作
        next_states = get_next_states(current_A, current_B, capacity_A,
capacity_B)

        for next_state, action in next_states:
            if next_state not in visited:
                visited.add(next_state)  // 标记为已访问

                // 将新动作加入路径，并将新状态压入队列
                new_path = path.copy()
                new_path.append(action)
                queue.push( (next_state, new_path) )

    return "无解"

// ------------------------------------------------------------
// 定义状态转移函数：穷举 6 种可能的操作
function get_next_states(A, B, max_A, max_B):
    states = new List()

    // 操作 1：装满水壶 A
    if A < max_A:
```

```
        states.append( ( (max_A, B), "装满 3 加仑水壶" ) )

    // 操作 2：装满水壶 B
    if B < max_B:
        states.append( ( (A, max_B), "装满 4 加仑水壶" ) )

    // 操作 3：清空水壶 A
    if A > 0:
        states.append( ( (0, B), "清空 3 加仑水壶" ) )

    // 操作 4：清空水壶 B
    if B > 0:
        states.append( ( (A, 0), "清空 4 加仑水壶" ) )

    // 操作 5：把 A 中的水倒入 B
    if A > 0 and B < max_B:
        // 核心逻辑：倒入量要么是 A 剩下的全部，要么是 B 还能装的量，取两者的最小值
        pour_to_B = min(A, max_B - B)
        states.append( ( (A - pour_to_B, B + pour_to_B), "3 加仑倒入 4 加仑" ) )

    // 操作 6：把 B 中的水倒入 A
    if B > 0 and A < max_A:
        pour_to_A = min(B, max_A - A)
        states.append( ( (A + pour_to_A, B - pour_to_A), "4 加仑倒入 3 加仑" ) )

    return states
```

## 算法实现的关键点：

1. **防重机制（Visited Set）**：这是搜索算法中必不可少的。如果没有 `visited` 集合，程序会在“装满 A -> 清空 A -> 装满 A”这种无意义的操作中陷入死循环。
2. **倒水逻辑（Min 函数）**：在实现“互相倒水”时，`min(源水壶当前水量, 目标水壶剩余容量)` 这个边界判断是最容易写错的地方，它完美地用数学方式表达了“直到一个倒空或者另一个倒满”的物理过程。