中南大學
CENTRAL SOUTH UNIVERSITY

离散课程报告

DISCRETE MATHEMATICS COURSE REPORT

题　　目：图搜索寻路算法的理论基础与实际应用——基于 BFS、DFS、Dijkstra、A* 算法的深度解析

学生姓名：潘意强

指导老师：沈海澜

学　　院：计算机学院

专业班级：计科 2403 班

2025 年 12 月 26 日

图搜索寻路算法的理论基础与实际应用——基于 BFS、DFS、Dijkstra、A* 算法的深度解析

摘要

图搜索寻路算法是计算机科学与离散数学领域的核心内容，广泛应用于网络路由、地图导航、游戏 AI 等场景。本报告深入探讨了四种经典的图搜索算法：广度优先搜索（BFS）、深度优先搜索（DFS）、Dijkstra 算法以及 A* 算法。

首先，报告从理论层面分析了每种算法的基本原理、数据结构基础（如队列、栈、优先队列）以及时间/空间复杂度。其次，通过构建详细的对比表格，从核心策略、适用场景及优缺点等维度对四种算法进行了横向比较。

随后，基于一个包含 6 个节点和 8 条边的具体无向带权连通图，报告详细演示了四种算法的运行过程。实验结果表明，在带权图中，BFS 和 DFS 无法保证找到最短路径（路径权重为 9），而 Dijkstra 和 A* 算法均能找到最短路径（路径权重为 6）。其中，A* 算法凭借启发式函数的引入，显著减少了节点扩展次数，展现了更高的搜索效率。

最后，报告结合离散数学图论原理，列举了各算法在社交网络分析、拓扑排序、地图导航及机器人路径规划等领域的实际应用，阐明了理论与实践的紧密联系。

关键词：图论；最短路径；BFS；DFS；Dijkstra；A* 算法

目录

第 1 章　四种寻路算法独立分析 …… 1

1.1　广度优先搜索 (BFS) …… 1

1.1.1　算法原理 …… 1

1.1.2　时间/空间复杂度 …… 1

1.1.3　C++ 代码实现 …… 1

1.1.4　优缺点分析 …… 2

1.2　深度优先搜索 (DFS) …… 2

1.2.1　算法原理 …… 2

1.2.2　时间/空间复杂度 …… 2

1.2.3　C++ 代码实现 …… 2

1.2.4　优缺点分析 …… 3

1.3　Dijkstra 算法 …… 3

1.3.1　算法原理 …… 3

1.3.2　时间/空间复杂度 …… 3

1.3.3　C++ 代码实现 …… 4

1.3.4　优缺点分析 …… 5

1.4　A* 算法 …… 5

1.4.1　算法原理 …… 5

1.4.2　时间/空间复杂度 …… 5

1.4.3　C++ 代码实现 …… 5

1.4.4　优缺点分析 …… 6

第 2 章　四种算法综合对比 …… 7

2.1　算法对比表 …… 7

2.2　差异分析 …… 7

2.2.1　无权图与带权图的适用性差异 …… 7

2.2.2　盲目搜索与启发式搜索的效率差异 …… 7

2.2.3　完备性与最优性的权衡 …… 7

第 3 章　指定图的具体运行逻辑解析 …… 9

3.1 图结构基础 …… 9
3.1.1 节点定义 …… 9
3.1.2 边及权重 …… 9
3.2 逐算法运行过程 …… 9
3.2.1 BFS (广度优先搜索) …… 9
3.2.2 DFS (深度优先搜索) …… 10
3.2.3 Dijkstra 算法 …… 11
3.2.4 A* 算法 …… 12
3.3 运行结果验证 …… 13
第 4 章 实际应用场景拓展 …… 14
4.1 BFS 的应用场景 …… 14
4.1.1 社交网络好友推荐 …… 14
4.1.2 网络爬虫页面遍历 …… 14
4.2 DFS 的应用场景 …… 14
4.2.1 拓扑排序与课程规划 …… 14
4.2.2 电路布线与连通性检测 …… 14
4.3 Dijkstra 的应用场景 …… 15
4.3.1 地图导航最短路径 …… 15
4.3.2 网络路由协议 (OSPF) …… 15
4.4 A* 的应用场景 …… 15
4.4.1 游戏 AI 路径规划 …… 15
4.4.2 无人机/机器人自主导航 …… 15
致谢 …… 16

第 1 章　四种寻路算法独立分析

本章将对广度优先搜索（BFS）、深度优先搜索（DFS）、Dijkstra 算法以及 A* 算法进行深入分析。我们将从算法原理、时间/空间复杂度、C++ 代码实现以及优缺点四个维度对每种算法进行详细阐述。

1.1　广度优先搜索 (BFS)

1.1.1　算法原理

广度优先搜索（Breadth-First Search, BFS）是一种用于遍历或搜索树或图的算法。其核心思想是从起始节点开始，首先访问所有邻接节点，然后再依次访问这些邻接节点的邻接节点。这种逐层向外扩展的方式类似于水波纹的扩散。

在数据结构上，BFS 使用**队列（Queue）**来实现。队列遵循先进先出（FIFO, First-In-First-Out）的原则，确保了节点是按照距离起点的层级顺序被访问的。在无权图中，BFS 能够保证找到从起点到终点的最短路径（即经过的边数最少）。

1.1.2　时间/空间复杂度

- **时间复杂度**：$O(V+E)$，其中 V 是顶点数，E 是边数。这是因为在最坏情况下，每个顶点和每条边都会被访问一次。
- **空间复杂度**：$O(V)$。在最坏情况下（例如星型图），队列中可能需要存储所有顶点。

1.1.3　C++ 代码实现

Listing 1.1: BFS 算法核心逻辑

```
// 核心算法：广度优先搜索
void bfs(char start, char goal) {
    queue<char> q; // 队列用于存储待访问节点
    q.push(start); // 起点入队
    visited[start] = true; // 标记起点已访问
    parent[start] = '\0';

    while (!q.empty()) {
        char current = q.front(); // 取出队首元素
        q.pop();

        if (current == goal) return; // 找到目标，结束搜索
```

```
            // 遍历所有邻接节点
            for (char neighbor : graph[current]) {
                if (!visited[neighbor]) { // 若未访问过
                    visited[neighbor] = true; // 标记访问
                    parent[neighbor] = current; // 记录路径
                    q.push(neighbor); // 入队
                }
            }
        }
}
```

1.1.4 优缺点分析

- **优点**：在无权图中能保证找到最短路径；算法逻辑简单，易于实现；完备性好（只要有解，一定能找到）。
- **缺点**：在带权图中无法保证找到最短路径（因为它只考虑边数不考虑权重）；空间复杂度较高，需要存储整层的节点。

1.2 深度优先搜索 (DFS)

1.2.1 算法原理

深度优先搜索（Depth-First Search, DFS）采用“一条路走到黑”的策略。它从起点开始，尽可能深地搜索图的分支。当节点 v 的所在边都已被探寻过，搜索将回溯到发现节点 v 的那条边的起始节点。这一过程一直进行到已发现从源节点可达的所有节点为止。

在数据结构上，DFS 使用**栈（Stack）**来实现（也可以通过递归隐式使用系统栈）。栈遵循后进先出（LIFO, Last-In-First-Out）的原则。

1.2.2 时间/空间复杂度

- **时间复杂度**：$O(V+E)$。与 BFS 类似，每个节点和边最多访问一次。
- **空间复杂度**：$O(V)$。主要取决于递归调用的深度或栈的大小，最坏情况下为 $O(V)$。

1.2.3 C++ 代码实现

Listing 1.2: DFS 算法核心逻辑

```
// 核心算法：深度优先搜索
void dfs(char start, char goal) {
    stack<char> s; // 栈用于存储待访问节点
    s.push(start); // 起点入栈
```

```
    visited[start] = true; // 标记起点已访问

    while (!s.empty()) {
        char current = s.top(); // 取出栈顶元素
        s.pop();

        if (current == goal) return; // 找到目标

        // 遍历邻接节点
        // 注意：为了模拟递归顺序，通常逆序压栈
        for (char neighbor : graph[current]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true; // 标记访问
                parent[neighbor] = current; // 记录路径
                s.push(neighbor); // 入栈
            }
        }
    }
}
```

1.2.4 优缺点分析

- **优点**：空间消耗通常比 BFS 小（只存储当前路径上的节点）；适合目标比较深或需要遍历所有解的情况（如迷宫生成、拓扑排序）。
- **缺点**：不一定能找到最短路径；可能会陷入死循环（如果图有环且未标记访问）；在无限图中可能无法终止。

1.3 Dijkstra 算法

1.3.1 算法原理

Dijkstra 算法是一种用于在加权图中找到最短路径的贪心算法。它维护一个集合，包含已找到最短路径的节点。算法重复从剩余节点中选择距离起点最近的节点加入集合，并更新其邻居的距离。

核心机制是**贪心策略**和**松弛操作**。它保证了每次扩展的节点都是当前未访问节点中距离起点最近的。

1.3.2 时间/空间复杂度

- **时间复杂度**：$O((V+E)\log V)$（使用优先队列优化）。如果使用数组实现优先队列，则为 $O(V^2)$。

- **空间复杂度**：$O(V)$，用于存储距离数组和优先队列。

1.3.3 C++ 代码实现

Listing 1.3: Dijkstra 算法核心逻辑

```
// 核心算法：Dijkstra最短路径
void dijkstra(char start, char goal) {
    // 优先队列，存储{距离，节点}，按距离从小到大排序
    priority_queue<pair<int, char>, vector<pair<int, char
       >>, greater<pair<int, char>>> pq;

    // 初始化距离为无穷大
    for (auto const& [node, edges] : adj) dist[node] =
       INT_MAX;
    dist[start] = 0; // 起点距离为0
    pq.push({0, start});

    while (!pq.empty()) {
        int d = pq.top().first; // 当前最短距离
        char u = pq.top().second; // 当前节点
        pq.pop();

        if (d > dist[u]) continue; // 如果当前距离大于已知
           最短距离，跳过
        if (u == goal) break; // 找到目标

        // 遍历邻居进行松弛操作
        for (auto& edge : adj[u]) {
            if (dist[u] + edge.weight < dist[edge.to]) {
                dist[edge.to] = dist[u] + edge.weight; //
                   更新最短距离
                parent[edge.to] = u; // 记录路径
                pq.push({dist[edge.to], edge.to}); // 入队
            }
        }
    }
}
```

1.3.4 优缺点分析

- **优点**：能保证在非负权图中找到最短路径；适用性广。
- **缺点**：无法处理负权边；效率低于 A*（因为它是盲目地向所有方向扩展，没有方向感）。

1.4 A* 算法

1.4.1 算法原理

A* 算法是 Dijkstra 算法的扩展，它引入了**启发式函数（Heuristic Function）** $h(n)$ 来估计从当前节点到目标节点的代价。总代价函数定义为 $f(n) = g(n) + h(n)$，其中 $g(n)$ 是从起点到当前节点的实际代价。

A* 算法优先扩展 $f(n)$ 最小的节点，从而更有方向性地搜索目标，减少了不必要的探索。

1.4.2 时间/空间复杂度

- **时间复杂度**：取决于启发函数的质量。最坏情况下退化为 Dijkstra 或 BFS。
- **空间复杂度**：$O(V)$，需要存储所有生成的节点。

1.4.3 C++ 代码实现

Listing 1.4: A* 算法核心逻辑

```
// 核心算法：A*启发式搜索
void a_star(char start, char goal) {
    // 优先队列，存储节点，按 f(n) = g(n) + h(n) 排序
    priority_queue<Node, vector<Node>, greater<Node>>
       open_set;

    g_score[start] = 0; // 起点实际代价为0
    // 入队：f = h(start), g = 0
    open_set.push({start, heuristic(start, goal), 0,
       heuristic(start, goal)});

    while (!open_set.empty()) {
        Node current = open_set.top(); // 取出 f 最小的节点
        open_set.pop();

        if (current.id == goal) break; // 找到目标
```

```
            // 遍历邻居
            for (auto& edge : adj[current.id]) {
                double tentative_g = g_score[current.id] + edge
                    .weight; // 计算新的 g 值

                // 如果找到更短路径
                if (g_score.find(edge.to) == g_score.end() ||
                    tentative_g < g_score[edge.to]) {
                    g_score[edge.to] = tentative_g; // 更新 g
                        值
                    double f = tentative_g + heuristic(edge.to,
                        goal); // 计算 f 值
                    // 入队
                    open_set.push({edge.to, f, tentative_g,
                        heuristic(edge.to, goal)});
                    parent[edge.to] = current.id; // 记录路径
                }
            }
        }
}
```

1.4.4 优缺点分析

- **优点**：搜索效率高，只要启发函数是可采纳的（Admissible），就能保证找到最短路径；可以通过调整启发函数在速度和精度之间权衡。
- **缺点**：空间占用大（需要保存所有生成的节点）；启发函数的设计对性能影响巨大。

第 2 章　四种算法综合对比

本章将通过构建详细的对比表格，从核心策略、数据结构、时间复杂度、适用场景、最短路径保证以及优缺点六个维度，对 BFS、DFS、Dijkstra 和 A* 算法进行横向对比。在此基础上，深入分析算法间的核心差异。

2.1　算法对比表

表 2-1　四种寻路算法综合对比

维度	BFS (广度优先)	DFS (深度优先)	Dijkstra	A* (A-Star)
核心策略	逐层扩展，先宽后深	深度优先，回溯	贪心策略，优先扩展距离起点最近的节点	启发式搜索，优先扩展 $f(n) = g(n) + h(n)$ 最小的节点
数据结构	队列 (FIFO)	栈 (LIFO)	优先队列 (Min-Heap)	优先队列 (Min-Heap)
时间复杂度	$O(V + E)$	$O(V + E)$	$O((V + E)\log V)$	取决于启发函数，最坏 $O(b^d)$
适用场景	无权图最短路径，连通性检测	拓扑排序，迷宫生成，全解搜索	带权图最短路径 (无负权)	游戏寻路，地图导航，复杂状态空间搜索
最短路径	仅在无权图中保证	不保证	保证 (无负权边)	保证 (启发函数可采纳)
优点	简单，无权图最优	空间占用少，能深搜	稳定，适用于各种带权图	效率高，方向性强
缺点	带权图失效，空间大	可能陷入死循环，非最短	效率不如 A*，盲目扩展	依赖启发函数设计，空间大

2.2　差异分析

2.2.1　无权图与带权图的适用性差异

BFS 和 DFS 主要设计用于无权图（或权重相同的图）。BFS 通过层级遍历，天然适合寻找无权图中的最短路径（最少边数）。然而，在带权图中，边数最少的路径不一定是总权重最小的路径，因此 BFS 失效。相比之下，Dijkstra 和 A* 算法引入了权重的概念，通过累积路径代价来评估节点，因此适用于带权图。

2.2.2　盲目搜索与启发式搜索的效率差异

BFS、DFS 和 Dijkstra 属于盲目搜索（Uninformed Search），它们在搜索过程中不利用关于目标位置的任何额外信息。Dijkstra 虽然利用了权重，但它向所有方向均匀扩展（类似于等高线），直到碰到目标。A* 算法属于启发式搜索（Informed Search），它利用启发函数 $h(n)$ 预估当前节点到目标的距离。这使得 A* 算法具有“方向感”，能够优先向目标方向探索，从而大大减少了需要访问的节点数量，提高了搜索效率。

2.2.3　完备性与最优性的权衡

- **完备性**：BFS 和 Dijkstra 是完备的，只要路径存在，它们一定能找到。DFS 在无限图中可能不完备。
- **最优性**：BFS 在无权图中是最优的；Dijkstra 在非负权图中是最优的；A* 在启发函数可采纳（$h(n) \le$ 实际代价）时是最优的。DFS 通常不保证最优性。

综上所述，选择哪种算法取决于具体的应用场景、图的性质（是否有权、是否有负权）以及对时间和空间效率的要求。

第 3 章　指定图的具体运行逻辑解析

本章将基于给定的图结构（6 个节点 S, A, B, C, D, T），详细解析 BFS、DFS、Dijkstra 和 A* 四种算法的具体运行过程，并验证其运行结果。

3.1　图结构基础

3.1.1　节点定义

图包含 6 个节点，其坐标如下（用于计算 A* 算法的启发式距离）：

- **S**: 起点 $[-6.0, 0, 0]$
- **A**: $[-3.5, 2.5, 0]$
- **B**: $[-3.5, -2.5, 0]$
- **C**: $[-1.0, 2.5, 0]$
- **D**: $[-1.0, -2.5, 0]$
- **T**: 终点 $[1.5, 0, 0]$

3.1.2　边及权重

这是一个**无向带权连通图**。边及其权重定义如下表所示：

表 3-1　图的边与权重

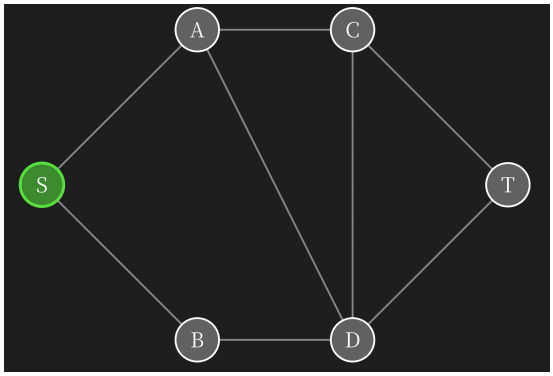
边 (Edge)	权重 (Weight)
(S, A)	3
(S, B)	1
(A, C)	2
(A, D)	5
(B, D)	2
(C, T)	4
(D, T)	3
(C, D)	1

3.2　逐算法运行过程

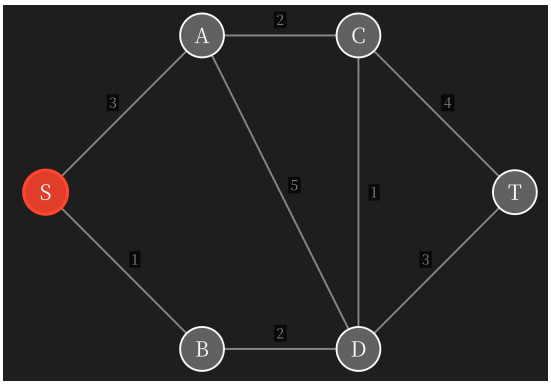
3.2.1　BFS (广度优先搜索)

核心机制：使用队列 (Queue)，先进先出 (FIFO)。邻居按升序排序。

为了更好地理解图的结构，我们展示了无权图（仅关注连接关系）和带权图（关注边的权重）的示意图。



(a) 无权图结构示意图



(b) 带权图结构示意图

图 3-1　实验用图结构示意

1. **初始化**：队列 `['S']`，已访问 `{'S'}`。
2. **S 出队**：邻居 A, B。A 入队，B 入队。队列 `['A', 'B']`。
3. **A 出队**：邻居 S(已访问), C, D。C 入队，D 入队。队列 `['B', 'C', 'D']`。
4. **B 出队**：邻居 S, D 均已访问。无新节点入队。队列 `['C', 'D']`。
5. **C 出队**：邻居 A, D 已访问，T 未访问。T 入队。队列 `['D', 'T']`。
6. **D 出队**：所有邻居均已访问。队列 `['T']`。
7. **T 出队**：找到终点，算法结束。

结果分析：BFS 找到的路径为 **S -> A -> C -> T**。

- **无权图场景**：路径长度为 3 (边数)，是最短路径。
- **有权图场景**：总权重 $3+2+4=9$。实际最短路径为 $S \to B \to D \to T$ (权重 6)。BFS 失败，因为它忽略了权重，仅按层级扩展。

3.2.2　DFS (深度优先搜索)

核心机制：使用栈 (Stack)，后进先出 (LIFO)。邻居按降序排序后入栈（即字母序小的在栈顶）。

1. **初始化**：栈 `['S']`，已访问 `{'S'}`。
2. **S 出栈**：邻居 A, B。排序后 [B, A]。B 入栈，A 入栈。栈 `['B', 'A']`。
3. **A 出栈**：邻居 C, D。排序后 [D, C]。D 入栈，C 入栈。栈 `['B', 'D', 'C']`。
4. **C 出栈**：邻居 T。T 入栈。栈 `['B', 'D', 'T']`。
5. **T 出栈**：找到终点，算法结束。

结果分析：DFS 找到的路径为 **S -> A -> C -> T**。总权重 9。DFS 同样未能找到最短路径，这是由其“不撞南墙不回头”的搜索策略决定的。

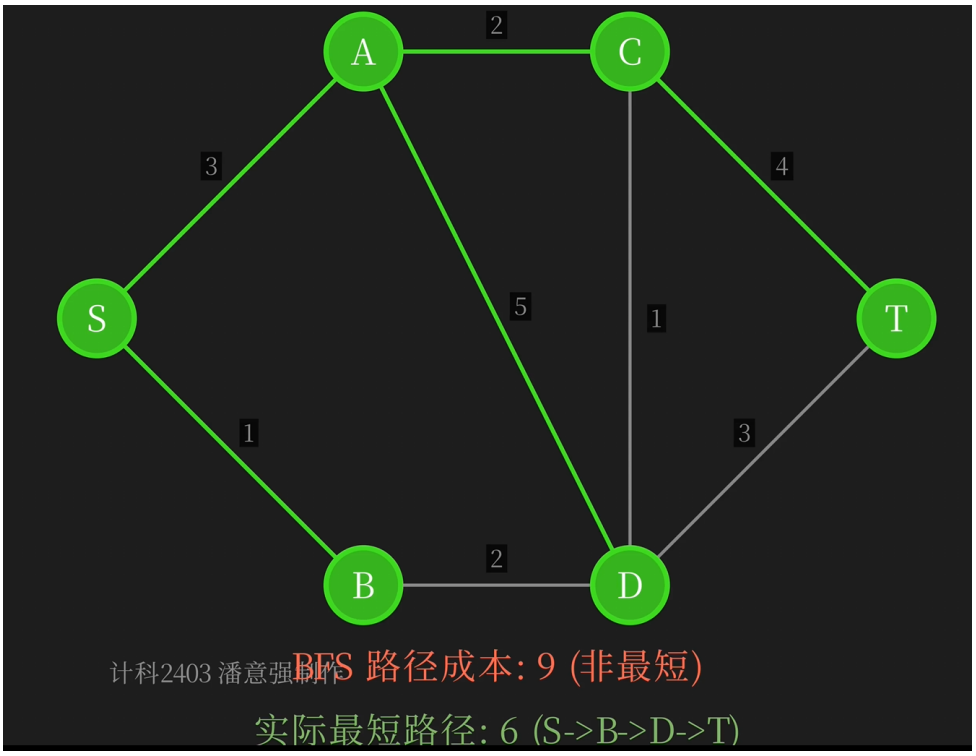


图 3-2　BFS 算法运行示意图（带权图场景）

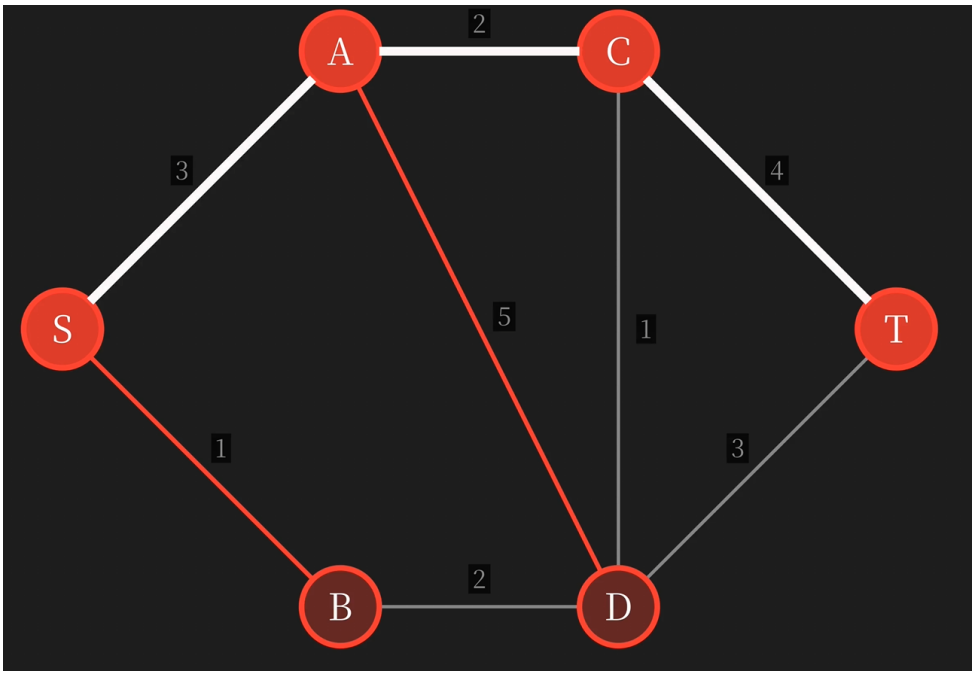


图 3-3　DFS 算法运行示意图

3.2.3　Dijkstra 算法

核心机制：贪心策略，优先队列。

1. **初始化：**$dist[S] = 0$，其余无穷大。PQ: `[(0, S)]`。
2. **S 出队** (0)：更新 A ($0 + 3 = 3$), B ($0 + 1 = 1$)。PQ: `[(1, B), (3, A)]`。

3. **B 出队** (1)：更新 D ($1+2=3$)。PQ: `[(3, A), (3, D)]`。
4. **A 出队** (3)：更新 C ($3+2=5$), D ($3+5=8>3$ 不更新)。PQ: `[(3, D), (5, C)]`。
5. **D 出队** (3)：更新 T ($3+3=6$), C ($3+1=4<5$ 更新)。PQ: `[(4, C), (5, C), (6, T)]`。
6. **C 出队** (4)：更新 T ($4+4=8>6$ 不更新)。PQ: `[(5, C), (6, T)]`。
7. **C 出队** (5)：过期数据，忽略。
8. **T 出队** (6)：找到终点。

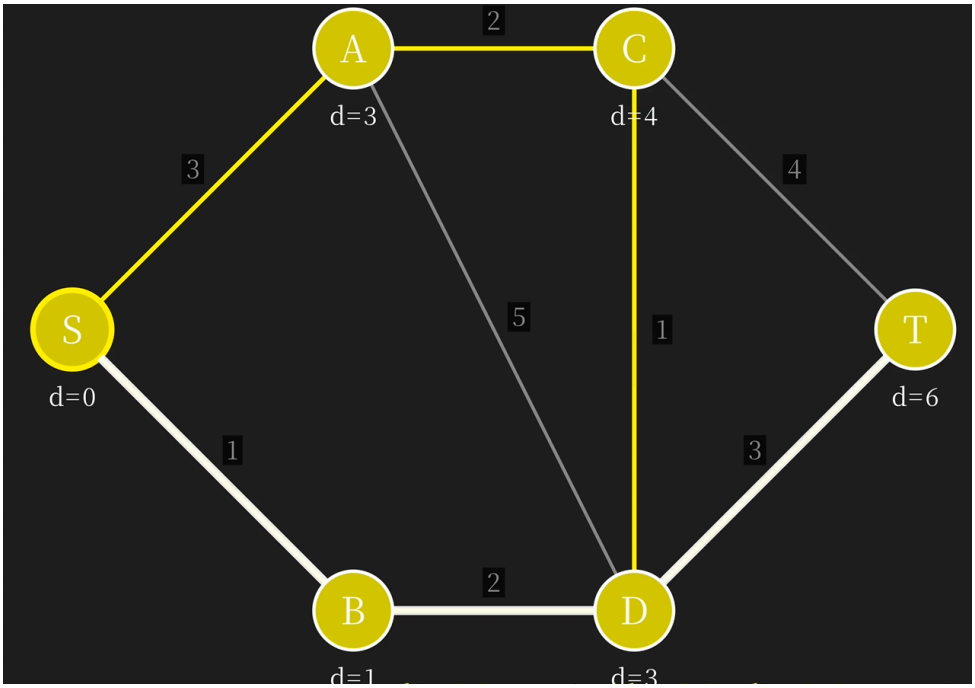


图 3-4　Dijkstra 算法运行示意图

结果分析：Dijkstra 找到的路径为 **S -> B -> D -> T**。总权重 **6**。这是真正的最短路径。

3.2.4　A* 算法

核心机制：$f(n)=g(n)+h(n)$。

1. **初始化**：$g[S]=0, h[S]\approx 7.5, f[S]=7.5$。PQ: `[(7.5, S)]`。
2. **S 出队**：
   - A: $g=3, h\approx 5.0, f=8.0$。
   - B: $g=1, h\approx 5.6, f=6.6$。

   PQ: `[(6.6, B), (8.0, A)]`。
3. **B 出队** (6.6)：
   - D: $g=1+2=3, h\approx 2.5, f=5.5$。

   PQ: `[(5.5, D), (8.0, A)]`。

4. **D 出队** (5.5)：
   - T: $g = 3 + 3 = 6, h = 0, f = 6.0$。
   - C: $g = 3 + 1 = 4, h \approx 2.5, f = 6.5$。

   PQ: `[(6.0, T), (6.5, C), (8.0, A)]`。
5. **T 出队** (6.0)：找到终点。

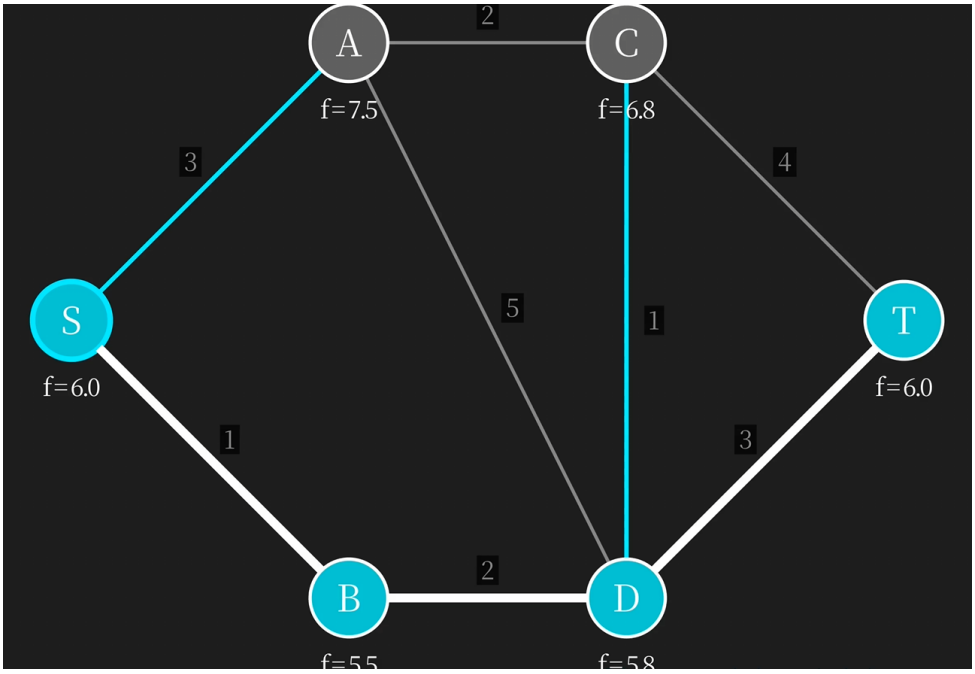


图 3-5　A* 算法运行示意图

结果分析：A* 找到的路径为 **S -> B -> D -> T**。总权重 **6**。A* 访问的节点更少（未扩展 A 和 C 的后续），效率更高。

3.3　运行结果验证

- **BFS**: 路径 S->A->C->T，权重 9 (非最短)。
- **DFS**: 路径 S->A->C->T，权重 9 (非最短)。
- **Dijkstra**: 路径 S->B->D->T，权重 6 (最短)。
- **A***: 路径 S->B->D->T，权重 6 (最短)。

对比可见，在带权图中，Dijkstra 和 A* 能保证找到最短路径，而 A* 通过启发式函数减少了节点扩展次数，效率更优。

第 4 章　实际应用场景拓展

本章将探讨 BFS、DFS、Dijkstra 和 A* 算法在现实世界中的具体应用场景。我们将结合离散数学图论原理，说明这些算法如何将实际问题转化为图论模型并加以解决。

4.1　BFS 的应用场景

4.1.1　社交网络好友推荐

在社交网络（如微信、Facebook）中，用户可以看作图的**节点**，好友关系看作**边**。

- **原理**：BFS 天然适合寻找“几度人脉”。一度人脉是直接好友，二度人脉是好友的好友。
- **应用**：系统可以通过 BFS 搜索用户的二度或三度好友，推荐可能认识的人。例如，如果 A 和 B 是好友，B 和 C 是好友，BFS 可以快速发现 A 和 C 之间的路径长度为 2，从而向 A 推荐 C。

4.1.2　网络爬虫页面遍历

互联网可以看作一个巨大的有向图，网页是**节点**，超链接是**边**。

- **原理**：BFS 可以用于层级抓取网页。从种子 URL 开始，先抓取所有直接链接的页面，再抓取下一层。
- **应用**：这种策略有助于防止爬虫陷入深层死循环（如无限日历页），并能优先抓取离首页较近的重要页面。

4.2　DFS 的应用场景

4.2.1　拓扑排序与课程规划

在大学课程规划中，课程是**节点**，先修关系是**有向边**。

- **原理**：DFS 可以用于检测图中是否有环（循环依赖），并生成拓扑排序。
- **应用**：如果课程 A 是课程 B 的先修课，DFS 可以帮助生成一个合法的选课顺序，确保学生在修读 B 之前已经修完了 A。

4.2.2　电路布线与连通性检测

在电子设计自动化（EDA）中，电路板上的元件引脚是**节点**，导线是**边**。

- **原理**：DFS 可以快速判断两个引脚是否连通，或者寻找所有连通的组件。
- **应用**：在布线阶段，利用 DFS 检测电路网络是否连通，或者在迷宫类游戏中生成复杂的迷宫路径。

4.3 Dijkstra 的应用场景

4.3.1 地图导航最短路径

在地图应用（如高德、百度地图）中，路口是**节点**，道路是**边**，道路长度或通行时间是**权重**。

- **原理**：Dijkstra 算法保证在非负权图中找到总权重最小的路径。
- **应用**：当用户请求从地点 A 到地点 B 的导航时，算法计算距离最短或时间最短的路线。由于道路长度不可能是负数，Dijkstra 非常适用。

4.3.2 网络路由协议 (OSPF)

在计算机网络中，路由器是**节点**，链路是**边**，带宽或延迟是**权重**。

- **原理**：开放最短路径优先（OSPF）协议使用 Dijkstra 算法计算路由表。
- **应用**：每个路由器计算到达网络中其他所有路由器的最短路径，从而决定数据包的最佳转发下一跳。

4.4 A* 的应用场景

4.4.1 游戏 AI 路径规划

在即时战略游戏（如《星际争霸》、《王者荣耀》）中，地图网格是**节点**，移动代价是**权重**。

- **原理**：A* 算法结合了实际距离和预估距离（启发函数），能快速找到路径且避开障碍物。
- **应用**：当玩家点击地图某处让角色移动时，A* 算法能实时计算出绕过障碍物的最优路径。启发函数通常使用曼哈顿距离或欧几里得距离。

4.4.2 无人机/机器人自主导航

在仓储物流机器人（如亚马逊 Kiva）或无人机配送中。

- **原理**：环境被建模为栅格图或路标图。
- **应用**：机器人需要从货架移动到打包台。A* 算法不仅考虑距离，还可以将电量消耗、拥堵程度纳入权重，并利用启发函数快速规划出一条避障且高效的路线。

致谢

感谢我离散老师沈海澜的悉心教导。