

**HUBEIUNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**算法设计与分析**

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目 | 实验五 | 实验类别 | | | 验证性 |
| 学生姓名 | 章崇文 | 学生学号 | | | 202202296 |
| 学生班级 | 计算机222 | | | | |
| 完成日期 | 2024-06-06 | | | | |
| 实验成绩 |  | | 评阅日期 |  | |
| 评阅教师 |  | | | | |

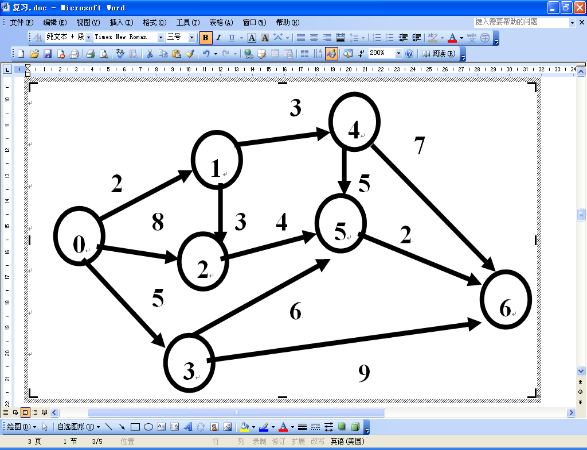
**实验五：分枝限界法**

**【实验目的】**

应用分枝限界法的算法设计思想求解单源最短路径问题。

**【实验内容与要求】**

采用分支限界法编程求源点0到终点6的最短路径及其路径长度。



要求完成：⑴算法描述⑵写出程序代码⑶完成调试⑷进行过程与结果分析。

**【实验性质】**

在完成的过程中注意与回溯算法思想的比较，重点注意两种算法思想各自的特点以及实现方式比较。此实验的性质为综合性实验。

**【分枝限界法与回溯算法思想】**

分枝限界法（Branch and Bound）和回溯算法（Backtracking）是两种用于解决组合优化问题和决策问题的重要算法思想。它们都以树形结构进行问题求解，但在具体实现和应用场景上存在一些区别。以下是对这两种算法思想的详细比较。

**相同点**

1. **树形结构**：两者都使用树形结构来表示问题的解空间。每个节点代表一个部分解或决策步骤，根节点代表初始状态，叶节点代表可能的完整解。
2. **递归求解**：两者都采用递归方式来探索解空间，通过递归调用逐步生成并测试部分解。
3. **剪枝策略**：为了减少搜索空间，两者都会使用剪枝策略。回溯算法通过约束条件进行剪枝，而分枝限界法则通过上界和下界进行剪枝。
4. **最优解**：两者都可以用于寻找问题的最优解。回溯算法可以用于求解所有解并从中选择最优解，分枝限界法则直接在搜索过程中保持和更新当前最优解。

**不同点**

1. **搜索策略**：
   * **回溯算法**：采用深度优先搜索（DFS）策略，沿着树的一个分支深入搜索，直到找到一个解或确定当前分支不可能产生解，然后回溯到上一个决策点继续搜索。
   * **分枝限界法**：通常采用广度优先搜索（BFS）或优先队列（Priority Queue）来管理部分解，选择当前最有希望产生最优解的分支进行扩展。
2. **剪枝依据**：
   * **回溯算法**：基于问题的约束条件进行剪枝，判断某个部分解是否可能扩展为一个可行解。
   * **分枝限界法**：基于上下界进行剪枝，通过计算当前部分解的上下界来判断是否继续扩展。如果当前部分解的上界小于已知解的下界，则剪去该分支。
3. **适用范围**：
   * **回溯算法**：适用于解空间较小、所有解都需要生成或验证的问题，如八皇后问题、数独等。
   * **分枝限界法**：适用于解空间较大、需要找到最优解的问题，如旅行商问题、0/1背包问题等。

**优缺点**

**回溯算法**：

* **优点**：
  + 实现简单，适合处理约束条件明确的问题。
  + 能够找到所有可行解，适合用于需要验证所有解的场景。
* **缺点**：
  + 对于解空间较大的问题，效率低下，容易导致指数级时间复杂度。
  + 需要大量的回溯操作，可能导致栈空间的快速耗尽。

**分枝限界法**：

* **优点**：
  + 能够有效减少搜索空间，通过上下界的估计进行剪枝，提高求解效率。
  + 适用于大规模优化问题，能够在较短时间内找到最优解或近似最优解。
* **缺点**：
  + 实现复杂度较高，需要设计和计算上下界的策略。
  + 在某些情况下，剪枝效果不明显，可能仍然需要遍历大量部分解。

**结论**

分枝限界法和回溯算法各有其适用场景和优缺点。回溯算法适合用于解空间较小且需要生成所有解的问题，而分枝限界法则适用于需要找到最优解的复杂优化问题。在实际应用中，选择哪种算法取决于具体问题的特点以及对解的要求。

**【算法思想及处理过程】**

**分治法解决算法问题的基本思想**

分治法（Divide and Conquer）是一种算法设计思想，其核心在于将一个复杂问题分解为多个相对简单的子问题，分别解决这些子问题后，再将子问题的解合并以得到原问题的解。分治法通常包括以下三个步骤：

分解（Divide）：将问题分解为若干个规模较小的子问题。

解决（Conquer）：递归地解决这些子问题。如果子问题足够小，则直接解决。

合并（Combine）：将子问题的解合并，得到原问题的解。

**处理该问题的详细算法**

问题描述

给定一个有向图，图的边权重非负，求从起点（节点0）到终点（节点6）的最短路径。

算法分析

该问题可以通过Dijkstra算法（基于贪心思想）解决，这里不完全符合分治法的标准定义，但可以借助分治思想进行理解。Dijkstra算法通过逐步扩展已知最短路径的节点集合，逐层分解并解决每一层的最短路径问题。因此，我们可以将其视为一种特殊的分治策略。

处理步骤

初始化：

使用一个优先队列（最小堆）来存储当前已知距离最短的节点。

初始化每个节点的最短距离为无穷大（INF），起点的最短距离为0。

分解与解决：

从优先队列中取出距离最小的节点u，并更新其邻接节点的最短距离。

对于每个邻接节点v，如果通过u到达v的距离小于当前已知的v的最短距离，则更新v的最短距离并将其加入优先队列。

重复上述过程，直到到达终点或优先队列为空。

合并：

通过记录的父节点数组，回溯构建最短路径。

**数据结构**

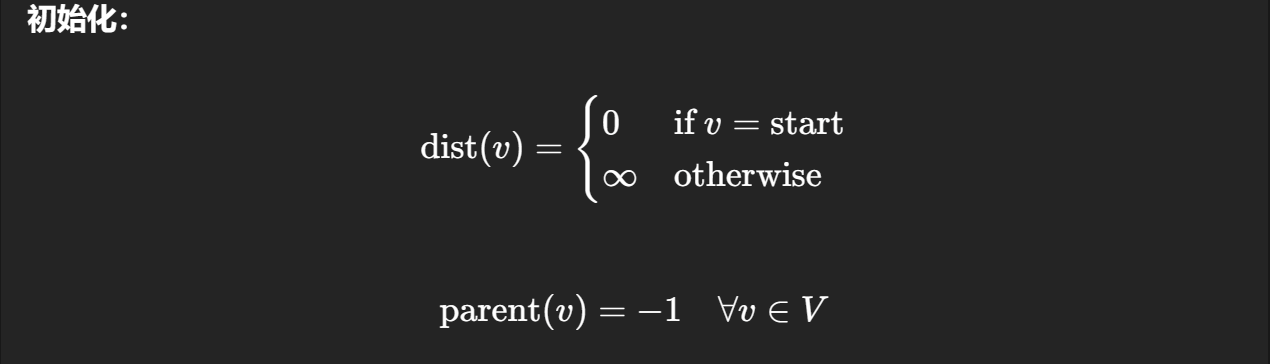
优先队列：用于存储和快速获取当前距离最小的节点。

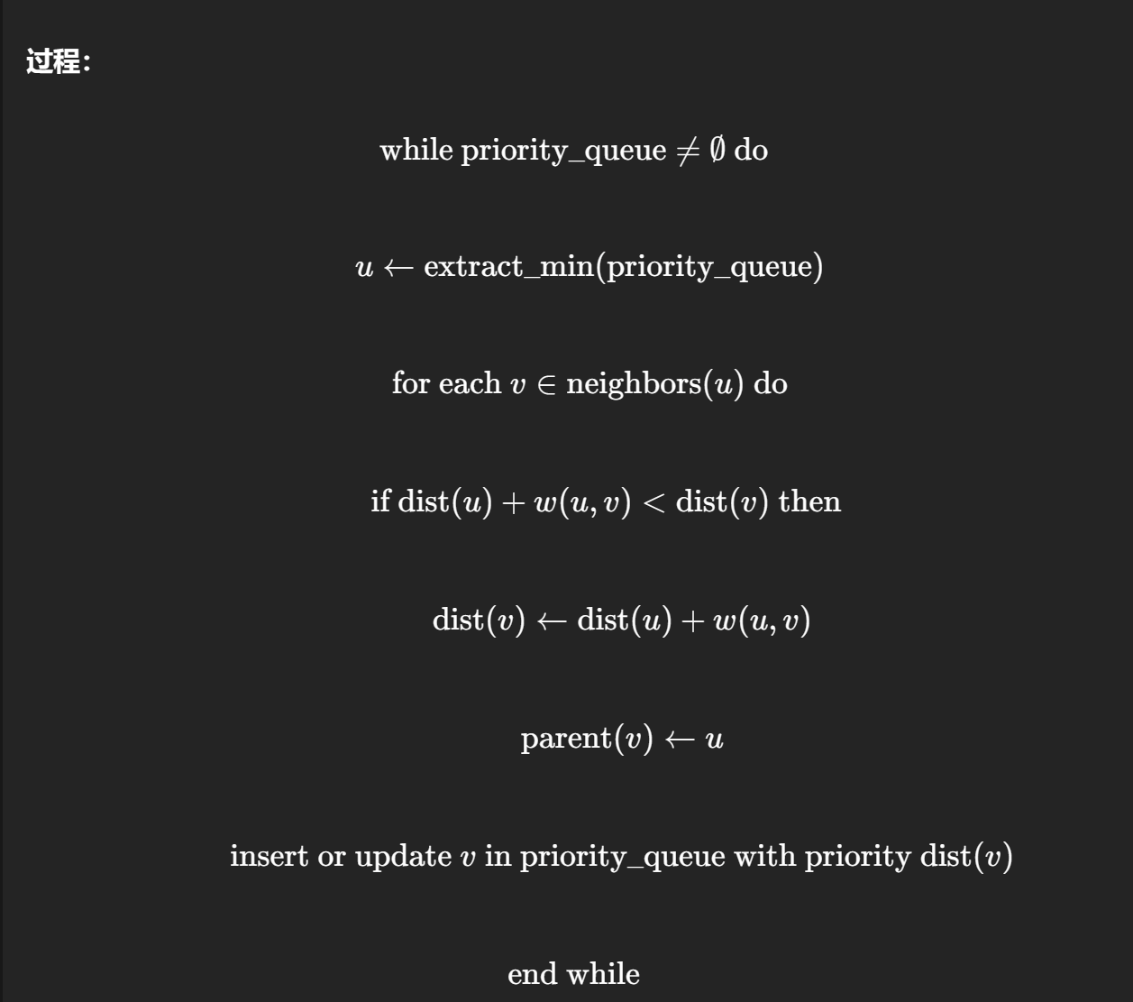
距离数组：存储从起点到每个节点的最短距离。

父节点数组：记录每个节点在最短路径上的前驱节点，用于路径回溯。

Dijkstra 算法的数学表达

假设有一个图 G = (V, E)，其中 V 是节点集合，E 是边集合，w(u, v) 表示边 (u, v) 的权重。Dijkstra 算法的数学过程可以描述如下：





**【程序代码】**

#include <iostream>

#include <queue>

#include <vector>

#include <limits>

using namespace std;

const int INF = numeric\_limits<int>::max();

vector<vector<int>> graph = {

{INF, 2, 8, 5, INF, INF, INF},

{INF, INF, 3, INF, 3, INF, INF},

{INF, INF, INF, INF, INF, 4, INF},

{INF, INF, INF, INF, INF, 6, 9},

{INF, INF, INF, INF, INF, 5, 7},

{INF, INF, INF, INF, INF, INF, 2},

};

vector<int> distances(7, INF);

vector<int> parent(7, -1);

struct Compare {

bool operator()(const pair<int, int>& a, const pair<int, int>& b) {

return a.second > b.second;

}

};

bool dijkstra(int start, int end) {

priority\_queue<pair<int, int>, vector<pair<int, int>>, Compare> pq;

distances[start] = 0;

pq.push({start, 0});

while (!pq.empty()) {

int u = pq.top().first;

pq.pop();

if (u == end) return true;

for (int v = 0; v < 7; ++v) {

if (graph[u][v] != INF && distances[u] + graph[u][v] < distances[v]) {

distances[v] = distances[u] + graph[u][v];

parent[v] = u;

pq.push({v, distances[v]});

}

}

}

return false; // 无法到达

}

void printPath(int end) {

if (distances[end] == INF) {

cout << "没有找到路径\n";

return;

}

int current = end;

while (current != -1) {

cout << current << " <- ";

current = parent[current];

}

cout << "0\n";

}

int main() {

if (dijkstra(0, 6)) {

cout << "最短路径长度" << distances[6] << "\n";

printPath(6);

} else {

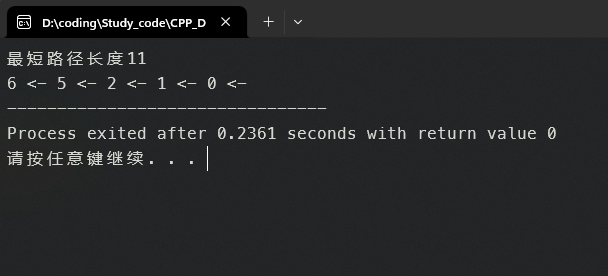
cout << "没有到6的路径" ;

}

return 0;

}

**【运行结果】**



**【算法分析】**

时间复杂度分析

Dijkstra算法的时间复杂度主要取决于以下几个方面：

优先队列的操作：

插入操作 (push)：每次插入一个元素到优先队列中，时间复杂度为 O(\log V)。

提取最小值 (pop)：每次从优先队列中提取最小值，时间复杂度为 O(\log V)。

邻接节点的遍历：

每个节点的所有邻接节点都需要被遍历一次，总的时间复杂度为 O(E)，其中 E 是边的数量。

综合考虑，Dijkstra算法的总体时间复杂度为 O((V + E) \log V)，其中 V 是节点数量，E 是边数量。

具体代码：

优先队列操作：

每次插入和提取操作的时间复杂度是 O(\log V)，最坏情况下，每个节点都会被插入和提取一次，因此是 O(V \log V)。

遍历所有边：

每个节点的所有邻接节点都会被遍历一次，总的时间复杂度是 O(E)。

因此，综合起来，时间复杂度是 O((V + E) \log V)。

在你的图中，节点数量 V = 7，边数量 E = 10（显式表示的边，包括权重为 \infty 的边），适用于上述时间复杂度分析。

空间复杂度分析

Dijkstra算法的空间复杂度主要由以下几个方面组成：

优先队列：

优先队列在最坏情况下会存储所有节点，空间复杂度为 O(V)。

距离数组 (distances)：

用于存储每个节点的最短距离值，空间复杂度为 O(V)。

父节点数组 (parent)：

用于存储每个节点的父节点，空间复杂度为 O(V)。

图的表示：

图用一个二维向量表示，空间复杂度为 O(V^2)，因为每个节点对其他所有节点都有一个权重值。

具体到代码：

优先队列：最坏情况下会包含所有节点，空间复杂度为 O(V)。

距离数组和父节点数组：每个数组的空间复杂度为 O(V)。

图的表示：使用的是一个 V \times V 的二维向量，空间复杂度为 O(V^2)。

因此，综合起来，空间复杂度为 O(V^2)。

**总结**

时间复杂度：O((V + E) \log V)，具体为 O((7 + 10) \log 7)。

空间复杂度：O(V^2)，具体为 O(7^2)。

【**实验总结**】