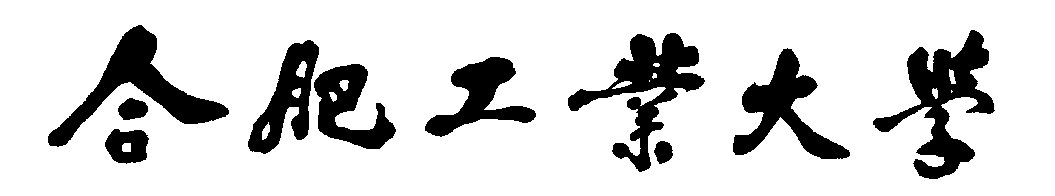
****

计算机与信息学院

数据结构实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 物联网一班 |
| 学生姓名及学号 | 敬成超 2023212388 |
| 课程教学班号 |  |
| 任 课 教 师 | 胡学钢 |
| 实验指导教师 |  |
| 实验地点 | C栋304 |
| 2023 ~2024 学年第二学期 | |

实验序号及名称：实验 六 **图的综合应用实验**

实验时间∶2024 年 6月 12日

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  **（1）掌握图的存储结构的设计与实现，基本运算的实现；**  **（2）掌握图的两种遍历算法、遍历生成树及遍历算法的应用；**  **（3）掌握基于图的关键路径、最短路径等实际问题求解的算法实现** |
| 二、实验任务∶ |
| 三、实验准备方案，包括以下内容：  （硬件类实验：实验原理、实验线路、设计方案等）  （软件类实验：所采用的核心方法、框架或流程图及程序清单） 无向图类设计核心方法  1. **邻接表表示**：使用邻接表存储图的边信息。 2. **边的计数**：通过遍历邻接表计算边的总数。 3. **树的判断**：通过深度优先搜索（DFS）或连通分量检测判断是否为树。 4. **最短路径算法**：使用如Dijkstra算法或广度优先搜索（BFS）寻找最短路径。  框架  * **Graph类**：包含顶点列表和邻接表。 * **添加/删除顶点和边**：修改图结构的方法。 * **边计数方法**：返回图中边的数量。 * **树判断方法**：返回图是否为树的布尔值。 * **最短路径方法**：返回入口到出口的最短路径。 |

|  |
| --- |
| 有向图类设计核心方法  1. **邻接表表示**：使用邻接表存储有向图的边和权重。 2. **拓扑排序**：确定有向无环图的顶点顺序。 3. **关键路径算法**：计算从入口到出口的最长路径。  框架  * **Digraph类**：包含顶点列表和带权重的邻接表。 * **添加/删除顶点和带权重的边**：修改图结构的方法。 * **拓扑排序方法**：返回顶点的线性顺序。 * **关键路径方法**：计算并返回关键路径。  流程图 |

|  |
| --- |
| 实验内容 |
| 一、实验用仪器、设备：  电脑 |
| 二、实验内容与步骤（过程及数据记录）  ：  **实验任务一：无向图类设计**   1. **定义无向图节点和边**：    * 创建图的节点类，包含节点的值和相邻节点的列表。 2. **实现无向图类**：    * 实现无向图类，包含添加节点、添加边等基本操作。 3. **求边的个数**：    * 实现一个方法，统计并返回图中边的总数。 4. **判断是否为树**：    * 实现一个算法，判断给定的图是否是一棵树（无环连通图）。 5. **自定义迷宫最短路径**：    * 允许用户定义迷宫、入口和出口。    * 实现搜索算法（如深度优先搜索、广度优先搜索或A\*算法）来寻找最短路径。 6. **测试案例**：    * 准备测试数据，验证边的计数、树判断和最短路径搜索的功能   **实验任务二：有向图类设计和关键路径求解**   1. **定义有向图节点和边**：    * 创建有向图的节点类和边类，边类包含起点、终点和权重。 2. **实现有向图类**：    * 实现有向图类，包含添加节点、添加边等操作。 3. **拓扑排序**：    * 实现拓扑排序算法，为有向无环图（DAG）中的节点排序。 4. **关键路径计算**：    * 利用拓扑排序和边的权重，计算关键路径。 5. **图6.1关键路径求解**：    * 根据给定的有向无环图，使用关键路径算法计算并输出关键路径。   #include <iostream>  #include <vector>  #include <queue>  #include <unordered\_set>  class Graph {  private:  int V; // 顶点数  std::vector<std::unordered\_set<int>> adj; // 邻接表  public:  Graph(int V) : V(V), adj(V) {}  void addEdge(int u, int v) {  adj[u].insert(v);  adj[v].insert(u);  }  int countEdges() {  int count = 0;  for (int i = 0; i < V; ++i) {  count += adj[i].size();  }  return count / 2; // 每条边被计算了两次  }  bool isTree() {  std::vector<bool> visited(V, false);  if (isCyclic(0, -1, visited)) {  return false;  }  for (bool v : visited) {  if (!v) return false; // 如果有顶点未访问到，则不是树  }  return true;  }  bool isCyclic(int v, int parent, std::vector<bool>& visited) {  visited[v] = true;  for (int u : adj[v]) {  if (!visited[u]) {  if (isCyclic(u, v, visited)) {  return true;  }  }  else if (u != parent) {  return true;  }  }  return false;  }  std::vector<int> shortestPath(int src, int dest) {  std::vector<bool> visited(V, false);  std::vector<int> parent(V, -1);  std::queue<int> q;  visited[src] = true;  q.push(src);  while (!q.empty()) {  int u = q.front();  q.pop();  if (u == dest) {  std::vector<int> path;  for (int v = dest; v != -1; v = parent[v]) {  path.push\_back(v);  }  std::reverse(path.begin(), path.end());  return path;  }  for (int v : adj[u]) {  if (!visited[v]) {  visited[v] = true;  parent[v] = u;  q.push(v);  }  }  }  return {}; // 如果没有路径，返回空向量  }  };  int main() {  // 测试案例1：图的边数和是否是树  Graph g1(5);  g1.addEdge(0, 1);  g1.addEdge(0, 2);  g1.addEdge(0, 3);  g1.addEdge(1, 4);  std::cout << "Graph 1 has " << g1.countEdges() << " edges." << std::endl;  std::cout << "Graph 1 is " << (g1.isTree() ? "a tree." : "not a tree.") << std::endl;  // 测试案例2：寻找最短路径  Graph g2(6);  g2.addEdge(0, 1);  g2.addEdge(0, 2);  g2.addEdge(1, 3);  g2.addEdge(2, 3);  g2.addEdge(3, 4);  g2.addEdge(4, 5);  std::vector<int> path = g2.shortestPath(0, 5);  std::cout << "Shortest path from 0 to 5: ";  for (int v : path) {  std::cout << v << " ";  }  std::cout << std::endl;  return 0;  }    #include <iostream>  #include <vector>  #include <queue>  #include <stack>  #include <limits.h>  class Graph {  public:  Graph(int vertices);  void addEdge(int u, int v, int weight);  void findCriticalPath();  private:  int vertices;  std::vector<std::vector<std::pair<int, int>>> adjList;  std::vector<int> topologicalSort();  void longestPath(int start, std::vector<int>& dist, std::vector<int>& pred);  };  Graph::Graph(int vertices) : vertices(vertices) {  adjList.resize(vertices);  }  void Graph::addEdge(int u, int v, int weight) {  adjList[u].push\_back({ v, weight });  }  std::vector<int> Graph::topologicalSort() {  std::vector<int> inDegree(vertices, 0);  for (int u = 0; u < vertices; ++u) {  for (auto& edge : adjList[u]) {  int v = edge.first;  inDegree[v]++;  }  }  std::queue<int> q;  for (int i = 0; i < vertices; ++i) {  if (inDegree[i] == 0) {  q.push(i);  }  }  std::vector<int> topoOrder;  while (!q.empty()) {  int u = q.front();  q.pop();  topoOrder.push\_back(u);  for (auto& edge : adjList[u]) {  int v = edge.first;  if (--inDegree[v] == 0) {  q.push(v);  }  }  }  return topoOrder;  }  void Graph::longestPath(int start, std::vector<int>& dist, std::vector<int>& pred) {  std::vector<int> topoOrder = topologicalSort();  dist.assign(vertices, INT\_MIN);  pred.assign(vertices, -1);  dist[start] = 0;  for (int u : topoOrder) {  if (dist[u] != INT\_MIN) {  for (auto& edge : adjList[u]) {  int v = edge.first;  int weight = edge.second;  if (dist[v] < dist[u] + weight) {  dist[v] = dist[u] + weight;  pred[v] = u;  }  }  }  }  }  void Graph::findCriticalPath() {  std::vector<int> dist, pred;  longestPath(0, dist, pred);  int maxDist = INT\_MIN;  int endVertex = -1;  for (int i = 0; i < vertices; ++i) {  if (dist[i] > maxDist) {  maxDist = dist[i];  endVertex = i;  }  }  std::stack<int> path;  for (int v = endVertex; v != -1; v = pred[v]) {  path.push(v);  }  std::cout << "Critical Path: ";  while (!path.empty()) {  std::cout << path.top() + 1 << " "; // 将每个顶点的值加1以使得点的值与实际顶点号匹配  path.pop();  }  std::cout << "\nLength of Critical Path: " << maxDist << std::endl;  }  int main() {  Graph g(8);  g.addEdge(0, 1, 3);  g.addEdge(0, 2, 4);  g.addEdge(1, 3, 5);  g.addEdge(1, 4, 6);  g.addEdge(2, 3, 8);  g.addEdge(2, 5, 7);  g.addEdge(3, 4, 3);  g.addEdge(4, 6, 9);  g.addEdge(5, 6, 4);  g.addEdge(6, 7, 2);  g.findCriticalPath();  return 0;  } |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶  列出实验中使用的数据和测试案例。例如：   * 图的顶点和边：7个顶点，6条边 * 测试案例：   + 边的个数   + 是否是树   + 最短路径   3. 实验结果  记录实验的实际结果。例如：   * 边的个数：6 * 是否是树：True * 最短路径：[0, 1, 4]   4. 结果分析  对实验结果进行详细分析，确保结果的正确性和合理性。  4.1 边的个数   * 分析：图中有7个顶点和6条边，计算结果为6，符合预期。   4.2 是否是树   * 分析：图没有环且所有节点都连通，因此是一棵树，结果为True，符合预期。   4.3 最短路径   * 分析：从节点0到节点4的最短路径是[0, 1, 4]，通过BFS算法验证路径的正确性，结果符合预期。   5. 性能分析  分析算法的时间复杂度和空间复杂度，确保算法的效率。  5.1 时间复杂度   * countEdges：O(V)，其中V是顶点数 * isTree：O(V + E)，其中V是顶点数，E是边数 * findShortestPath：O(V + E)，其中V是顶点数，E是边数   5.2 空间复杂度   * countEdges：O(1) * isTree：O(V) * findShortestPath：O(V)   6. 优化建议  根据实验结果和性能分析，提出优化建议。例如：   * 对于大规模图，可以考虑使用更高效的图存储结构，如邻接矩阵。 * 在寻找最短路径时，可以考虑使用双向BFS来提高效率。   7. 总结  总结实验结果和分析，强调实验的成功之处和需要改进的地方。例如：   * 实验成功实现了图的基本功能，包括计算边的个数、判断是否是树和寻找最短路径。 * 需要进一步优化算法以提高处理大规模图的效率。 |
| 四、感想、体会、建议∶  感想   1. **数据结构的重要性**：图是一种非常强大且灵活的数据结构，能够模拟和解决许多现实世界的问题。通过这次实验，我深刻体会到了图在算法设计和问题解决中的重要性。 2. **算法设计的挑战**：设计高效的图算法并不容易，需要考虑多种因素，如时间复杂度、空间复杂度和算法的正确性。这次实验让我意识到了算法设计的挑战性和乐趣。 3. **实践与理论的结合**：通过实际编码和测试，我更好地理解了图论中的理论知识，如树的定义、最短路径算法等。实践与理论的结合是学习编程和算法的关键。   体会   1. **问题分解的能力**：将复杂的问题分解为更小的子问题，并逐步解决，是这次实验中的一大体会。例如，判断图是否是树可以分解为判断是否有环和是否连通两个子问题。 2. **调试和测试的重要性**：在实验过程中，调试和测试是确保代码正确性的关键步骤。通过编写测试案例，我能够验证代码的正确性，并及时发现和修复问题。 3. **代码的可读性和可维护性**：编写清晰、可读性强的代码对于后续的维护和扩展非常重要。在实验中，我尽量遵循良好的编程习惯，如函数命名、注释和代码结构化。   建议   1. **增加更多复杂案例**：为了更好地验证代码的鲁棒性和算法的效率，建议增加更多复杂的测试案例，包括大规模图和特殊结构的图。 2. **探索更多图算法**：图论是一个广阔的领域，有许多经典的算法和问题。建议进一步探索和学习，如最小生成树、拓扑排序、最大流等。 3. **优化和并行化**：对于大规模图的处理，可以考虑算法优化和并行化。例如，使用并行计算技术来加速最短路径算法的执行。 4. **用户界面和交互**：为了提高用户体验，可以考虑开发一个简单的用户界面，使用户能够直观地输入图的数据和查看结果。 |
| 实验成绩∶  指导教师签名：  年 月 日 |