哈尔滨工业大学计算学部

实验报告

课程名称：数据结构与算法

课程类型：专业核心基础课（必修）

实验项目：图型结构及其应用

实验题目： 图的存储结构的建立与搜索

实验日期：2023.11.18

班级：2203202

学号：2022112266

姓名：魏圣卓

|  |  |  |
| --- | --- | --- |
| 设计成绩 | 报告成绩 | 指导老师 |
|  |  | 张岩 |

1. **实验目的**

图的搜索是图型结构最重要的算法，本作业要求掌握图结构的典型存储结构的建立方法和搜索算法的实现方法。

**二、实验要求及实验环境**

1. 分别实现无向图（或有向图）的邻接矩阵和邻接表存储结构的建立算法，分析和比较各建立算法的时间复杂度以及存储结构的空间占用情况。

2. 实现无向图（或有向图）的邻接矩阵和邻接表两种存储结构的相互转换算法。

3. 在上述两种存储结构上，分别实现无向图（或有向图）的深度优先搜索(递归和非递归)和广度优先搜索算法。并以适当的方式存储和展示相应的搜索结果，包括：深度优先或广度优先生成森林（或生成树）、深度优先或广度优先序列和深度优先或广度优先编号。并分析搜索算法的时间复杂度和空间复杂度。

4. （1）对于无向图，采用“邻接表”存储结构，设计和实现计算每个顶点度的算法，并分析其时间复杂度。 或者，

（2）对于有向图，采用“邻接表”存储结构，设计和实现计算每个顶点入度、出度和度的算法，并分析其时间复杂度。

5. 作业发送要求：（1）每次使用同一个邮箱交作业；（2）每次作业发送一封且仅一封邮件；（3）每次实验发送一封且仅一封邮件

5. 以适当的方式输入图的顶点和边，并显示相应的结果。要求顶点不少于 10

个，边不少于 13 个。

**三、设计思想**（本程序中的用到的所有数据类型的定义，主程序的流程图及各程序模块之间的调用关系、核心算法的主要步骤）

**存储结构：**

    vector<vector<int>> g;

    vector<vnode\*> list;

    vector<int> visited;

    vector<int> BH;

    vector<vector<int>> dfs\_forest; // Store DFS forest information

    vector<vector<int>> bfs\_forest; // Store BFS forest information

    int v;

    int edges;

typedef struct tnode

{

    int data;

    struct tnode\* lchild;

    struct tnode\* brother;

}TREE;

class vnode {

public:

    int v;

    vnode\* next;

    vnode(int x)

    {

        v = x;

        next = nullptr;

    };

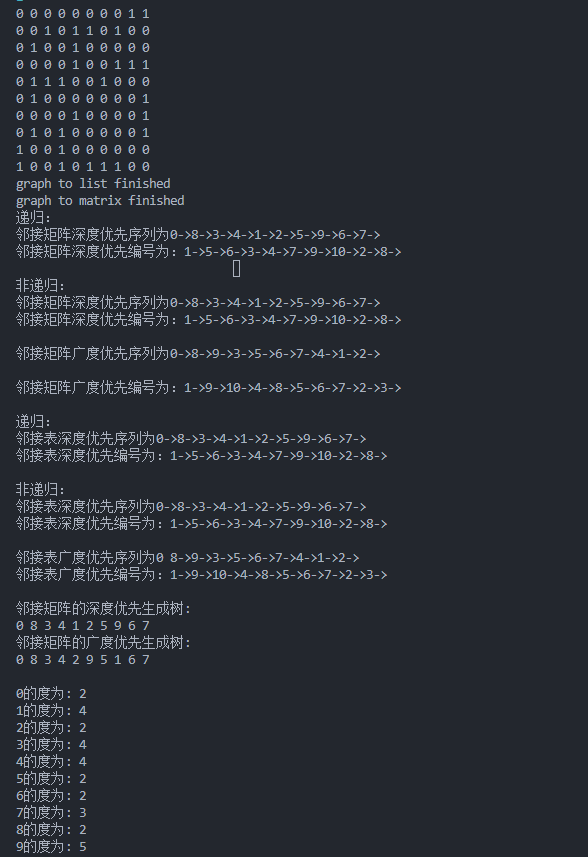
    ~vnode() {};

};

**四、测试结果**（包括**测试数据、结果数据及结果的简单分析和结论，可以用截图得形式贴入此报告**）

输入的图：

10 15 1 2 1 4 1 5 1 7 2 4 4 3 4 6 6 9 3 9 3 8 3 7 7 9 5 9 8 0 0 9



**五、经验体会与不足**

在此次学习中，我了解而到了遍历图的几种方法，包括深度优先搜索和广度优先搜索。同时掌握的图的典型存储结构，包括邻接矩阵结构和邻接表结构。后续还可以继续把关于有向图的算法进一步完善。

**六、附录：源代码（带注释）**

1. #include <bits/stdc++.h>

2. #include <vector>

3.

4. using namespace std;

5.

6. const int INF = numeric\_limits<int>::max(); // 代表无穷大

7. // const int INF = INFINITY;

8.

9. // 树结点的结构体

10. typedef struct tnode

11. {

12. int data; // 结点数据

13. struct tnode\* lchild; // 左孩子指针

14. struct tnode\* brother; // 兄弟结点指针

15. } TREE;

16.

17. // 邻接表结点的结构体

18. class vnode

19. {

20. public:

21. int v; // 结点数据

22. vnode\* next; // 指向下一个邻接结点的指针

23. vnode(int x) // 构造函数

24. {

25. v = x;

26. next = nullptr;

27. };

28. ~vnode() {}; // 析构函数

29. };

30.

31. // 图的类定义

32. class graph

33. {

34. public:

35. vector<vector<int>> g; // 邻接矩阵

36. vector<vnode\*> list; // 邻接表

37. vector<int> visited; // 标记结点是否被访问过

38. vector<int> BH; // 结点编号

39. vector<vector<int>> dfs\_forest; // DFS生成森林

40. vector<vector<int>> bfs\_forest; // BFS生成森林

41.

42. int v; // 结点数

43. int edges; // 边数

44.

45. // 构造函数

46. graph(int n, int m)

47. : v(n)

48. , edges(m)

49. , visited(n, 0)

50. , BH(n, 0)

51. , g(n, vector<int>(n, 0))

52. , list(n, nullptr)

53. {

54. for (int i = 0; i < n; i++) {

55. list[i] = new vnode(i);

56. }

57. }

58.

59. // 析构函数

60. ~graph()

61. {

62. }

63.

64. // 添加边

65. void addEdge(int x, int y)

66. {

67. g[x][y] = 1;

68. g[y][x] = 1;

69. }

70.

71. // 将邻接矩阵转换为邻接表

72. void trans(int print)

73. {

74. // 构建邻接表

75. for (int i = 0; i < v; i++) {

76. vnode\* curr = list[i];

77. for (int j = 0; j < v; j++) {

78. if (g[i][j] == 1) {

79. vnode\* p = new vnode(j);

80. curr->next = p;

81. curr = curr->next;

82. }

83. }

84. }

85.

86. if (print) {

87. // 输出邻接表

88. for (int i = 0; i < v; i++) {

89. vnode\* curr = list[i];

90.

91. for (int j = 0; j < v; j++) {

92. cout << curr->v;

93. if (curr->next == nullptr) {

94. break;

95. }

96. curr = curr->next;

97. cout << "->";

98. }

99. cout << endl;

100. }

101. }

102. cout << "graph to list finished" << endl;

103.

104. // 将邻接表还原为邻接矩阵

105. for (int i = 0; i < v; i++) {

106. for (int j = 0; j < v; j++) {

107. g[i][j] = 0;

108. }

109. }

110.

111. for (int i = 0; i < v; i++) {

112. vnode\* p = list[i]->next;

113. while (p != NULL) {

114. g[i][p->v] = 1;

115. p = p->next;

116. }

117. }

118. if (print) {

119. display();

120. }

121. cout << "graph to matrix finished" << endl;

122. }

123.

124. // 邻接矩阵深度优先搜索（递归版）

125. void g\_dfs(int curr, int& count)

126. {

127. visited[curr] = 1;

128. cout << curr << "->";

129. BH[curr] = count;

130. count++;

131. for (int i = 0; i < v; i++) {

132. if (g[curr][i] && !visited[i]) {

133. g\_dfs(i, count);

134. }

135. }

136. }

137.

138. // 邻接矩阵深度优先搜索

139. void g\_dfs(int curr)

140. {

141. for (int i = 0; i < v; i++) {

142. visited[i] = 0;

143. BH[i] = 0;

144. }

145. cout << "邻接矩阵深度优先序列为";

146. int count = 1;

147. g\_dfs(curr, count);

148. cout << endl;

149. cout << "邻接矩阵深度优先编号为：";

150. for (int i = 0; i < v; i++) {

151. cout << BH[i] << "->";

152. }

153. cout << endl;

154. }

155.

156. // 邻接矩阵深度优先搜索（非递归版）

157. stack<int> s;

158. void g\_dfs\_no\_rec(int curr, int& count)

159. {

160. s.push(curr);

161. cout << curr << "->";

162. BH[curr] = count;

163. count++;

164. while (!s.empty()) {

165. curr = s.top();

166. visited[curr] = 1;

167. for (int i = 0; i < v; i++) {

168. if (g[curr][i] && !visited[i]) {

169. cout << i << "->";

170. BH[i] = count;

171. count++;

172. s.push(i);

173. break;

174. }

175.

176. if (i == v - 1) {

177. s.pop();

178. }

179. }

180. }

181. }

182.

183. // 邻接矩阵深度优先搜索（非递归版）

184. void g\_dfs\_no\_rec(int curr)

185. {

186. for (int i = 0; i < v; i++) {

187. visited[i] = 0;

188. BH[i] = 0;

189. }

190. cout << "邻接矩阵深度优先序列为";

191. int count = 1;

192. g\_dfs\_no\_rec(curr, count);

193. cout << endl;

194. cout << "邻接矩阵深度优先编号为：";

195. for (int i = 0; i < v; i++) {

196. cout << BH[i] << "->";

197. }

198. cout << endl;

199. }

200.

201. // 邻接矩阵广度优先搜索

202. queue<int> q;

203. void g\_bfs(int curr,int count)

204. {

205. for (int i = 0; i < visited.size(); i++) {

206. visited[i] = 0;

207. }

208. q.push(curr);

209. cout << curr << "->";

210. BH[curr] = count;

211. count++;

212. visited[curr] = 1;

213. while (!q.empty()) {

214. curr = q.front();

215. q.pop();

216. for (int i = 0; i < v; i++) {

217. if (!visited[i] && g[curr][i]) {

218. q.push(i);

219. visited[i] = 1;

220. cout << i << "->";

221. BH[i] = count;

222. count++;

223. }

224. }

225. }

226. cout << endl;

227. for (int i = 0; i < visited.size(); i++) {

228. visited[i] = 0;

229. }

230. }

231.

232. // 邻接矩阵广度优先搜索

233. void g\_bfs(int curr)

234. {

235. for (int i = 0; i < v; i++) {

236. visited[i] = 0;

237. BH[i] = 0;

238. }

239. cout << "邻接矩阵广度优先序列为";

240. int count = 1;

241. g\_bfs(curr, count);

242. cout << endl;

243. cout << "邻接矩阵广度优先编号为：";

244. for (int i = 0; i < v; i++) {

245. cout << BH[i] << "->";

246. }

247. cout << endl;

248. }

249.

250. // 邻接表深度优先搜索

251. void list\_dfs(int curr,int &count)

252. {

253. visited[curr] = 1;

254. cout << curr << "->";

255. BH[curr] = count;

256. count++;

257. vnode\* p = list[curr]->next;

258. while (1) {

259. if (p == nullptr) {

260. break;

261. }

262. if (!visited[p->v]) {

263. list\_dfs(p->v,count);

264. }

265. p = p->next;

266. }

267. }

268.

269. // 邻接表深度优先搜索

270. void list\_dfs(int curr)

271. {

272. for (int i = 0; i < v; i++) {

273. visited[i] = 0;

274. BH[i] = 0;

275. }

276. cout << "邻接表深度优先序列为";

277. int count = 1;

278. list\_dfs(curr, count);

279. cout << endl;

280. cout << "邻接表深度优先编号为：";

281. for (int i = 0; i < v; i++) {

282. cout << BH[i] << "->";

283. }

284. cout << endl;

285. }

286.

287. // 邻接表深度优先搜索（非递归版）

288. void list\_dfs\_no\_rec(int curr ,int count)

289. {

290. cout << curr << "->";

291. BH[curr] = count;

292. count++;

293. s.push(curr);

294. visited[curr] = 1;

295. while (!s.empty()) {

296. curr = s.top();

297. vnode\* p = list[curr]->next;

298. while (p != NULL) {

299. if (!visited[p->v]) {

300. s.push(p->v);

301. visited[p->v] = 1;

302. cout << p->v << "->";

303. BH[p->v] = count;

304. count++;

305. break;

306. }

307. p = p->next;

308. if (p == NULL) {

309. s.pop();

310. }

311. }

312. }

313. }

314.

315. // 邻接表深度优先搜索（非递归版）

316. void list\_dfs\_no\_rec(int curr)

317. {

318. for (int i = 0; i < v; i++) {

319. visited[i] = 0;

320. BH[i] = 0;

321. }

322. cout << "邻接表深度优先序列为";

323. int count = 1;

324. list\_dfs\_no\_rec(curr, count);

325. cout << endl;

326. cout << "邻接表深度优先编号为：";

327. for (int i = 0; i < v; i++) {

328. cout << BH[i] << "->";

329. }

330. cout << endl;

331. }

332.

333. // 邻接表广度优先搜索

334. void list\_bfs(int curr,int count)

335. {

336. q.push(curr);

337. visited[curr] = 1;

338. cout << curr << " ";

339. BH[curr] = count;

340. count++;

341. while (!q.empty()) {

342. curr = q.front();

343. q.pop();

344. vnode\* p = list[curr]->next;

345. while (p != NULL) {

346. if (!visited[p->v]) {

347. q.push(p->v);

348. visited[p->v] = 1;

349. cout << p->v << "->";

350. BH[p->v] = count;

351. count++;

352. }

353. p = p->next;

354. }

355. }

356. }

357.

358. // 邻接表广度优先搜索

359. void list\_bfs(int curr)

360. {

361. for (int i = 0; i < v; i++) {

362. visited[i] = 0;

363. BH[i] = 0;

364. }

365. cout << "邻接表广度优先序列为";

366. int count = 1;

367. list\_bfs(curr, count);

368. cout << endl;

369. cout << "邻接表广度优先编号为：";

370. for (int i = 0; i < v; i++) {

371. cout << BH[i] << "->";

372. }

373. cout << endl;

374. // 深度优先生成树（森林）

375. void DFSTree(int i, TREE\*\* T)

376. {

377. visited[i] = 1; // 标记已访问

378. int flag = 1;

379. TREE\* p = NULL;

380. int j;

381. for (j = 0; j < v; j++)

382. {

383. if (g[i][j] && !visited[j])

384. {

385. TREE\* temp = new tnode; // 创建新的树结点

386. temp->data = j;

387. temp->lchild = NULL;

388. temp->brother = NULL;

389. if (flag)

390. {

391. (\*T)->lchild = temp; // 若是第一个孩子，作为左孩子

392. flag = 0;

393. }

394. else

395. {

396. p->brother = temp; // 否则作为兄弟结点

397. }

398. p = temp;

399. DFSTree(j, &p); // 递归构建子树

400. }

401. }

402. }

403.

404. // 深度优先生成森林并转化为二叉树

405. void DFSForest(TREE\*\* T)

406. {

407. (\*T) = NULL;

408. for (int i = 0; i < v; i++)

409. {

410. visited[i] = 0; // 重置访问标记

411. }

412. TREE\* p;

413. for (int i = 0; i < v; i++)

414. {

415. if (!visited[i])

416. {

417. TREE\* temp = new tnode; // 创建新的树结点

418. temp->data = i;

419. temp->lchild = NULL;

420. temp->brother = NULL;

421. if (!(\*T))

422. {

423. (\*T) = temp; // 若是第一个树，直接赋值给根结点

424. }

425. else

426. {

427. p->brother = temp; // 否则作为兄弟结点

428. }

429. p = temp;

430. DFSTree(i, &temp); // 递归构建子树

431. }

432. }

433. }

434.

435. // 先序遍历树结点

436. void PreOrder(TREE\* T)

437. {

438. if (T)

439. {

440. printf("%d ", T->data); // 打印结点数据

441. PreOrder(T->lchild); // 递归遍历左孩子

442. PreOrder(T->brother); // 递归遍历兄弟结点

443. }

444. return;

445. }

446.

447. TREE\* queue2[50];

448. int front;

449. int rear;

450.

451. // 广度优先生成树

452. void BFSTree(int i, TREE\*\* T)

453. {

454. TREE\* p = NULL;

455. front = rear = -1; // 初始化队列

456. queue2[++rear] = (\*T); // 根结点入队

457. while (front != rear)

458. {

459. int flag = 1;

460. p = queue2[++front]; // 出队

461. i = p->data;

462. visited[i] = 1; // 标记已访问

463. for (int j = 0; j < v; j++)

464. {

465. if ((!visited[j]) && g[i][j])

466. {

467. TREE\* temp = new tnode; // 创建新的树结点

468. temp->data = j;

469. temp->lchild = NULL;

470. temp->brother = NULL;

471. queue2[++rear] = temp; // 新结点入队

472. visited[j] = 1;

473. if (flag)

474. {

475. p->lchild = temp; // 若是第一个孩子，作为左孩子

476. flag = 0;

477. }

478. else

479. {

480. p->brother = temp; // 否则作为兄弟结点

481. }

482. p = temp;

483. }

484. }

485. }

486. }

487.

488. // 广度优先生成森林

489. void BFSForest(TREE\*\* T)

490. {

491. (\*T) = NULL;

492. for (int i = 0; i < v; i++)

493. {

494. visited[i] = 0; // 重置访问标记

495. }

496. TREE\* p = NULL;

497. for (int i = 0; i < v; i++)

498. {

499. if (!visited[i])

500. {

501. TREE\* temp = new tnode; // 创建新的树结点

502. temp->data = i;

503. temp->lchild = NULL;

504. temp->brother = NULL;

505. if (!(\*T))

506. {

507. (\*T) = temp; // 若是第一个树，直接赋值给根结点

508. }

509. else

510. {

511. p->brother = temp; // 否则作为兄弟结点

512. }

513. p = temp;

514. BFSTree(i, &temp); // 广度优先生成子树

515. }

516. }

517. }

518.

519. // 计算顶点的度

520. void cal()

521. {

522. for (int i = 0; i < v; i++)

523. {

524. int cnt = 0;

525. vnode\* p = list[i]->next;

526. while (p != NULL)

527. {

528. cnt++;

529. p = p->next;

530. }

531. printf("\n%d的度为: %d", i, cnt);

532. }

533. }

534.

535. // 显示邻接矩阵

536. void display()

537. {

538. for (int i = 0; i < v; ++i)

539. {

540. for (int j = 0; j < v; ++j)

541. {

542. cout << g[i][j] << " ";

543. }

544. cout << endl;

545. }

546. }

547. int main()

548. {

549. TREE\* T1; // 存储深度优先生成树的根结点

550. TREE\* T2; // 存储广度优先生成树的根结点

551.

552. int n, m; // 图的顶点数和边数

553.

554. FILE\* f = fopen("graph.txt", "r"); // 以只读方式打开文件

555.

556. if (f == nullptr)

557. {

558. perror("Error opening file");

559. return 1; // 文件打开错误，返回错误代码

560. }

561.

562. // 从文件中读取图的顶点数和边数

563. fscanf(f, "%d %d", &n, &m);

564.

565. // 创建图对象，传入顶点数和边数

566. graph g(n, m);

567.

568. int source, dest;

569. // 从文件中读取每条边的起点和终点，并添加到图中

570. for (int i = 1; i <= m; i++)

571. {

572. fscanf(f, "%d %d", &source, &dest);

573. g.addEdge(source, dest);

574. }

575.

576. g.display(); // 显示邻接矩阵

577. g.trans(0); // 将邻接矩阵转换为邻接表

578. cout << "递归深度优先遍历：" << endl;

579. g.g\_dfs(0); // 递归深度优先遍历

580. cout << endl;

581. cout << "非递归深度优先遍历：" << endl;

582. g.g\_dfs\_no\_rec(0); // 非递归深度优先遍历

583. cout << endl;

584.

585. cout << "广度优先遍历：" << endl;

586. g.g\_bfs(0); // 广度优先遍历

587. cout << endl;

588. cout << "邻接表递归深度优先遍历：" << endl;

589. g.list\_dfs(0); // 邻接表递归深度优先遍历

590. cout << endl;

591. cout << "邻接表非递归深度优先遍历：" << endl;

592. g.list\_dfs\_no\_rec(0); // 邻接表非递归深度优先遍历

593. cout << endl;

594.

595. cout << "邻接表广度优先遍历：" << endl;

596. g.list\_bfs(0); // 邻接表广度优先遍历

597. cout << endl;

598.

599. // 以下两行注释掉的代码涉及生成树，根据需要选择解注释

600. // g.g\_dfs\_forest(); // 生成深度优先生成森林

601. // g.g\_bfs\_forest(); // 生成广度优先生成森林

602.

603. cout << "邻接矩阵的深度优先生成树：" << endl;

604. g.DFSForest(&T1); // 生成深度优先生成树

605. g.PreOrder(T1); // 先序遍历并输出树结点

606. cout << endl;

607. cout << "邻接矩阵的广度优先生成树：" << endl;

608. g.BFSForest(&T2); // 生成广度优先生成树

609. g.PreOrder(T2); // 先序遍历并输出树结点

610. cout << endl;

611.

612. g.cal(); // 计算顶点的度

613. return 0;

614. }

615.