P155-P157

# 第11章 图和图算法

网络研究已经成为本世纪最棒的科学温床之一，虽然数学家和其他人已经研究了几百年网络。科学技术（例如，互联网）以及社会理论的近期发展（社交网络，随着“六度分隔理论”概念而普及），更不用提社交媒体，都将焦点放在了网络研究上。

本章我们将看到网络是怎样使用图来建模的。我们将定义什么是图，怎样用JavaScript来表示图，以及如何实现重要的图算法。我们还将讨论使用图时选择正确数据展现的重要性，因为图算法的效率大部分依赖用于表示图的数据结构。

## 图的定义

***图***（graph）由顶点的集合和边的集合组成。想像一下美国的一个州的地图。每个镇都是通过某种类型的道路与其他镇相连。地图就是一种图，其中每个镇都是一个***顶点***（vertex），连接两个镇之间的每条道路都是一条***边***（edge）。边使用(v1,v2)顶点对定义，其中v1和v2是图的两个顶点。顶点还可以有权重（weight），有时候也被称为开销（cost）。顶点对有序的图被称为***有向图***（directed graph），或者digraph。有向图中的顶点对有顺序时，会从一个顶点到另一个顶点画一个箭头。有向图指出了从顶点到顶点之间的流动方向。计算机程序中指示计算方向的流程图就是一个有向图的实例。图11-1展示了一个有向图：

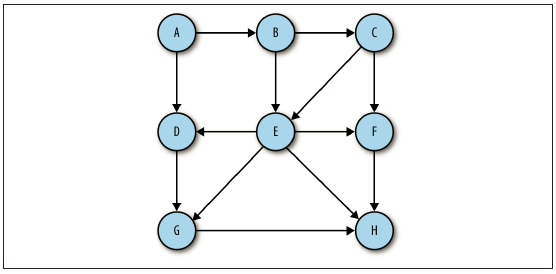


图11-1.有向图（digraph）

如果一个图的顶点对没有顺序，就被称为无向图，或者图。图11-2展示了一个无向图的实例：

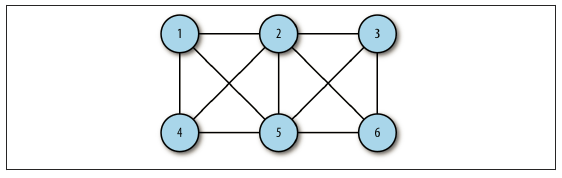


图11-2.无向图

***路径***（path）是图中顶点的序列，路径中的所有顶点都通过边连接。路径的长度是从路径中第一个顶点到最后一个顶点所经过的边的数量。路径也可以是一个指向自己的顶点，称为***环***（loop），环的长度为0。

***圈***（cycle）是至少有一条边且起点和终点相同的路径。对于有向图和无向图，没有重复边和重复顶点的圈叫***简单圈***（simple cycle）。除了第一个顶点和最后一个顶点有其他顶点重复的路径被称为***平凡圈***（general cycle）。

如果从一个点到另一个点有路径，则认为这两个点是***强连通***（strongly connected）的，反之亦然。如果一个图是有向图，而且图的所有顶点都是强连通的，那么这个有向图也被认为是强连通的。

## 用图建模的真实系统

现实世界中很多不同类型的系统都使用图来建模。交通流就是一个例子。顶点代表了街道的交叉口，边代表街道。边的权值可用于表示速度限制或者车道数量。建模者可以用该系统确定最佳路径以及最有可能陷入交通阻塞的街道。

任何类型的交通系统都可以使用图来建模。例如，航空公司可以用图来为其飞行系统建模。每个机场是一个点，从一个点到另一个点的每条航线是一条边。边的权值可以表示从一个机场到另一个机场的飞行成本，或者从一个机场到另一个机场的距离，这取决于建模对象。

计算机网络，包括局域网和更加广域的网络，例如互联网，也常使用图来建模。另一个可以使用图来建模的真实系统实例是消费者市场，其中点代表机构（供应商）和消费者。

## 图类

初看起来，图很像一棵树或者二叉树，你很可能会像创建树一样来创建图类，使用节点来代表顶点。然而，像这样使用基于对象的方法是有问题的，因为图可能增长得非常大。仅使用对象来表示图很快会变得低效，因此我们使用一种不同的模式来表示顶点和边。

## 顶点的表示

创建图类的第一步是创建一个保存图顶点的Vertex类。这个类和链表以及二叉搜索树的Node类有相同的职责。Vertex类需要两个数据成员：一个用来确定点，另一个用布尔类型来标记这个点是否被访问过。这两个数据成员分别命名为label和wasVisited。Vertex类只需要一个构造函数，来允许我们为点的数据成员设值。这是Vetex类的代码：

Function Vertext(lable) {

this.lable = label;

}

我们把点的列表存到数组中，并在Graph类中通过它们在数组中的位置来引用。

P158-

## 边的表示

图的真实信息存储在边里，因为边描述了图的结构。正如我们前面提到的，将图用二叉树表示充满诱惑，但这样做是错误的。二叉树的表示特别固定，因为一个父亲节点只能有两个子节点，而图的结构提供了更大的灵活性。例如，可以有多条边，或者仅一条边连接到一个单独的顶点。

我们把表示图的边的方法称为***邻接表***（adjacency list）或者邻接表数组。这种方法将边存储为以顶点做索引，由顶点的相邻顶点列表（数组）组成的数组。使用这种方案，在程序中引用一个顶点时，我们可以有效地访问这个顶点连通的所有顶点。例如，如果顶点2连通了顶点0、1、3和4，并且存储在数组的位置2，那么访问这个元素就可以访问在数组位置2存储的由顶点0、1、3、4组成的顶点数组。这是我们在本章选择使用的表示方法，如图11-3所示：

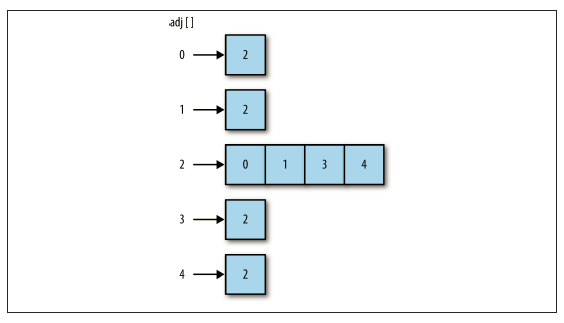


图11-3：邻接表

另一种表示图边的方法被称为邻接矩阵（adjacency matrix）。这是一个二维数组，数组中的元素表示两个顶点之间是否有一条边。

## 构建图

决定图的代码表示后，构建一个表示图的类就直观了。这是Graph类的第一次定义：

function Graph(v) {

this.vertices = v;

this.edges = 0;

this.adj = [];

for (var i = 0; I < this.vertices; ++i) {

this.adj[i] = [];

this.adj[i].push(“”);

}

this.addEdge = addEdge;

this.toString = toString;

}

这个类会记录图中表示了多少条边，并使用一个长度与图中顶点数相同的数组来记录顶点的数量。for循环会给数组中的每个元素添加一个子数组来存储所有的相邻点，并将所有元素初始化为空字符串。

addEdge()函数定义如下：

function addEdge(v, w) {

this.ajd[v].push(w);

this.adj[w].push(v);

this.edges++;

}

使用两个顶点A和B调用该函数时，函数会查找顶点A的邻接表，将顶点B添加到列表中，接着该函数会查找顶点B的邻接表，将顶点A加入列表。最后，这个函数会将边的数量加1。

showGraph()函数会显示所有顶点，以及他们的相邻顶点列表来显示图：

function showGraph() {

for (var I = 0; i < this.vertices; ++i) {

putstr(i + ‘>’);

for (var j = 0; j < this.vertices; ++j) {

if (this.adj[i][j] != undefined)

putstr(this.adj[i][j] + ‘ ’);

}

print();

}

}

示例11-1显示了Graph类的完整定义。

示例11-1：图类

function Graph(v) {

this.vertices = v;

this.edges = 0;

this.adj = [];

for (var i = 0; i < this.vertices; ++i) {

this.adj[i] = [];

this.adj[i].push(“”);

}

this.addEdge = addEdge;

this.showGraph = showGraph;

}

function addEdge(v, w) {

this.adj[v].push(w);

this.adj[w].push(v);

this.edges++;

}

function showGraph() {

for (var i = 0; i < this.vertices; ++i) {

putstr(i + “ -> “);

for (var j = 0; j < this.vertices; ++j ) {

if (this.adj[i][j] != undefined) {

putstr(this.adj[i][j] + ‘ ’);

}

}

print();

}

}

下面是一个演示如何使用图类的测试程序：

load(“Graph.js”);

g = new Graph(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.showGraph();

这段程序的输出为：

0 -> 1 2

1 -> 0 3

2 -> 0 4

3 -> 1

4 -> 2

输出显示从顶点0有到顶点1和顶点2的边；顶点1有到顶点0和顶点3的边；顶点2有到顶点0和4的边；顶点3有到顶点1的边；顶点4有到顶点2的边。当然，这种显示存在冗余，例如，0和1之间的边和1到0之间的边相同。如果只是为了显示，这样是不错的，但是在开始探索图的路径之前，我们需要调整一下输出。

## 搜索图

确定从一个指定顶点可以到达哪些顶点是在图上执行的常见操作。我们可能想了解在地图上从一个镇有到另一个镇有哪些道路，或者哪些航线可以让我们从一个机场到另一个机场。

在图上的这些操作是用搜索算法执行的。在图上可以执行两种基础搜索：深度优先搜索和广度优先搜索。本节我们将仔细推究这两种算法。

## 深度优先搜索

深度优先搜索包括追溯从起始顶点到达最后顶点经过的路径，再回溯和追踪下一条路径直到到达最后的顶点，如此继续直到没有路径剩余。这不是搜索特定项目，而是通过搜索来查看在图中有哪些路径可以追循。图11-4阐述了深度优先的运行方式。

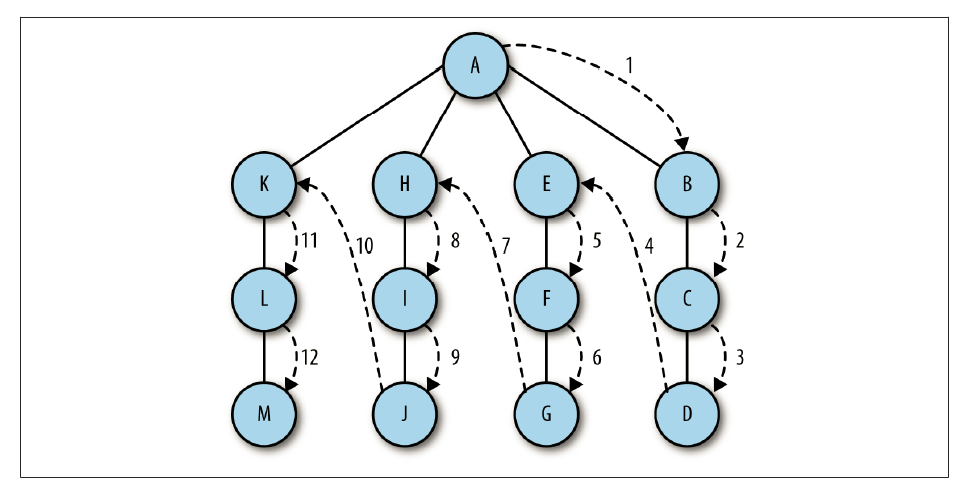


图11-4：深度优先搜索

执行深度优先搜索的算法相对简单——访问一个没有访问过的顶点，标记为已访问，再递归访问在初始顶点的邻接表中没有访问的其他顶点。

要让该算法运行，我们需要添加一个存储已访问顶点的数组到图类，并将它的值全部初始化为false。这里有一段来自图类，用于显示这个新数组及其初始化过程的代码片段：

this.marked = [];

for (var i = 0; i < this.vertices; ++i ) {

this.marked[i] = false;

}

现在我们可以开始编写深度优先函数：

function dfs(v) {

this.marked[v] = true;

// if statement for print is not required

if (this.adj[v] != undefined)

print(“Visited vertex: “ + v);

for each(var w in this.adj[v]) {

if (!this.marked[w]) {

this.dfs(w);

}

}

}

注意我包含了一个print()函数，这样我们就可以查看正在访问的顶点。当然，dfs()不需要print()函数也能正常运行。

演示depthFirst()函数的程序，以及完整的图类定义，如示例11-2所示：

function Graph(v) {

this.vertices = v;

this.edges = 0;

this.adj = [];

for (var i = 0; i < this.vertices; ++i) {

this.adj[i] = [];

this.adj[i].push(“”);

}

this.addEdge = addEdge;

this.showGraph = showGraph;

this.dfs = dfs;

this.marked = [];

for (var i = 0; i < this.vertices; ++i) {

this.marked[i] = false;

}

}

function addEdge(v, w) {

this.adj[v].push(w);

this.adj[w].push(v);

this.edges++;

}

function showGraph() {

for (var i = 0; i < this.vertices; ++i) {

putstr(i + “ -> “);

for (var j = 0; j < this.vertices; ++j) {

if (this.adj[i][j] != undefined)

putstr(this.add[i][j] + ‘ ‘);

}

print();

}

}

function dfs(v) {

this.marked[v] = true;

if (this.adj[v] != undefined) {

print(“Visited vertex: “ + v);

}

for each(var w in this.adj[v]) {

if (!this.marked[w]) {

this.dfs(w);

}

}

}

//测试dfs()函数的程序

load(“Graph.js”);

g = new Graph(5);

g.addEdge(0, 1);

g.addEdge(0,2);

g.addEdge(1,3);

g.addEdge(2,4);

g.showGraph();

g.dfs(0);

这段程序的输出为：

0 -> 1 2

1 -> 0 3

2 -> 0 4

3 -> 1

4 -> 2

Visited vertex: 0

Visited vertex: 1

Visited vertex: 3

Visited vertex: 2

Visited vertex: 4

## 广度优先搜索

广度优先搜索从第一个顶点开始，尝试访问尽可能靠近的顶点。本质上，这种搜索在图上按照层次来移动，首先检查最靠近第一个顶点的层，再向下移动到里开始起点最远的层。图11-5演示了深度优先搜索的运行方式。

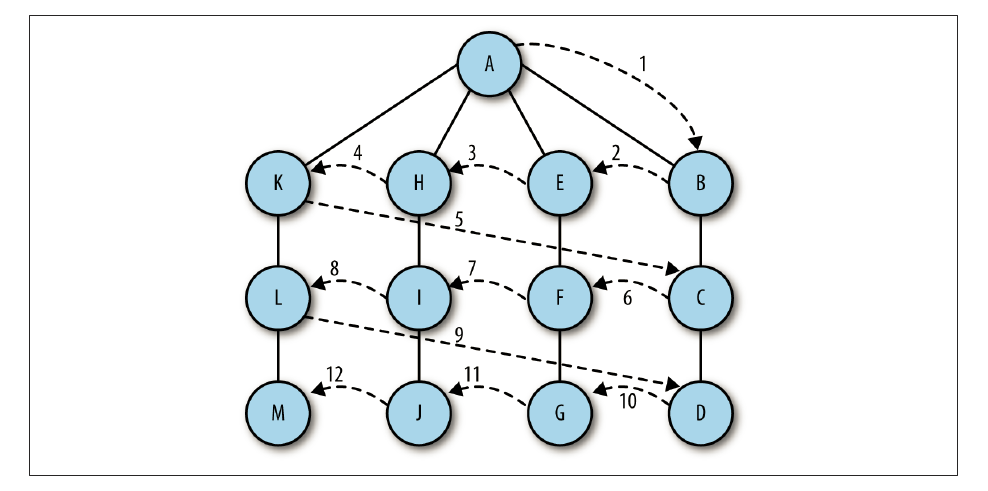


图11-5.深度优先搜索

搜索优先搜索算法使用了抽象的队列而不是数组来排序已访问过的顶点。算法运行如下：

1. 查找与当前顶点相邻的未访问顶点，将其加入已访问顶点列表，并加入队列。
2. 从图中取下一个顶点v加入已访问顶点列表。
3. 将所有与v相邻的未访问顶点添加到队列。

这是广度优先搜索函数的定义：

function bfs(s) {

var queue = [];

this.marked[s] = true;

queue.push(s); //添加到队列末尾

while (queue.length > 0) {

var v = queue.shift(); //从队列头部移除

if (v == undefined) {

print(‘Visisted vertex: ‘ + v);

}

for each(var w in this.adj[v]) {

if (!this.marked[w]) {

this.edgeTo[w] = v;

this.marked[w] = true;

queue.push(w);

}

}

}

}

广度优先搜索函数的测试程序如示例11-3所示。

示例11-3.执行广度优先搜索

load(“Graph.js”);

g = new Graph(5);

g.addEdge(0, 1);

g.addEdge(0, 2);

g.addEdge(1, 3);

g.addEdge(2, 4);

g.showGraph();

g.bfs(0);

这段程序的输出为：

0 -> 1 2

1 -> 0 3

2 -> 0 4

3 -> 1

4 -> 2

Visited vertex: 0

Visited vertex: 1

Visited vertex: 2

Visited vertex: 3

Visited vertex: 4

## 查找最短路径

图最常见的操作之一就是寻找从一个顶点到另一个顶点的最短路径。考虑下面的例子：因为假期，你将在两个星期的时间里游历10个大联盟城市来观看棒球比赛。你想要使用最短路径算法来将开车访问所有10个城市需要的里程数最小化。另一个最短路径问题是创建一个计算机网络，其中的开销是两台电脑之间传递数据的时间，或者两台电脑建立和维护连接的成本。最短路径算法可以确定构建网络的最有效方法。

## 广度优先搜索到最短路径

在执行广度优先搜索时，我们会自动查找从一个顶点到另一个顶点的最短路径。例如，在我们想要查找从顶点A到顶点D的最短路径时，我们会首先查找从A到D是否有任何一条边的路径，接着查找两条边的路径，等等。这正是广度优先搜索的运行方式，因此我们可以轻松地修改广度优先搜索算法来查找最短路径。

## 确定路径

要查找最短路径，我们需要修改广度优先搜索算法来记录从一个顶点到另一个顶点的路径。这需要对图类（Graph）做一些修改。

首先，我们需要一个数组来保存从一个顶点到下一个顶点的记录。我们将这个数组命名为edgeTo。随着我们对广度优先搜索函数的使用，每次我们遇到一个没有标记的顶点时，除了做标记，还会从邻接列表中我们正在探索的点添加一条边到这个点。这是新的bfs()函数，以及需要添加到图类（Graph）的代码：

// 将这行添加到Graph类

this.edgeTo = [];

// bfs 函数

function bfs(s) {

var queue = [];

this.marked[s] = true;

queue.push(s); //添加到队列末尾

while (queue.length > 0) {

var v = queue.shift(); //从队列头部移除

if (v == undefined) {

print(‘Visisted vertex: ‘ + v);

}

for each(var w in this.adj[v]) {

if (!this.marked[w]) {

this.edgeTo[w] = v;

this.marked[w] = true;

queue.push(w);

}

}

}

}

现在我们需要一个函数，来向我们展示连接到一个图的不同顶点的路径。这个函数，pathTo()，创建了一个栈来存储与特定顶点有共同边的所有顶点。这是pathTo()函数的代码，以及一个简单的帮助函数：

function pathTo(v) {

var source = 0;

if (!this.hasPathTo(v)) {

return undefined;

}

var path = [];

for (var i = v; i != source; i = this.edgeTo[i]) {

path.push(i);

}

path.push(s);

return path;

}

function hashPathTo(v) {

return this.marked[v];

}

确保将核实的声明添加到了Graph()函数：

this.pathTo = pathTo;

this.hasPathTo = hashPathTo;

有了这个函数，我们要做的就是编写一些客户端代码来显示从源顶点到某个特定顶点的最短路径。示例11-4显示了创建一个图的程序，以及对于一个特定顶点的最短路径。

示例11-4.查找一个顶点的最短路径

load(“Graph.js”);

g = new Graph(5);

g.addEdge(0,1);

g.addEdge(0,2);

g.addEdge(1,3);

g.addEdge(2,4);

var vertex = 4;

var paths = g.pathTo(vertex);

while (paths.length > 0) {

if (paths.length > 1) {

putstr(paths.pop() + '-');

} else {

putstr(paths.pop());

}

}

这段程序的输出为：

0-2-4

即从源顶点0到顶点4的最短路径。

## 拓扑排序

拓扑排序会将一个有向图的所有顶点排序，让所有的有向边按照顶点顺序从一个顶点指向另一个顶点。例如：图11-6展示了典型计算机科学课程的有向图模型。

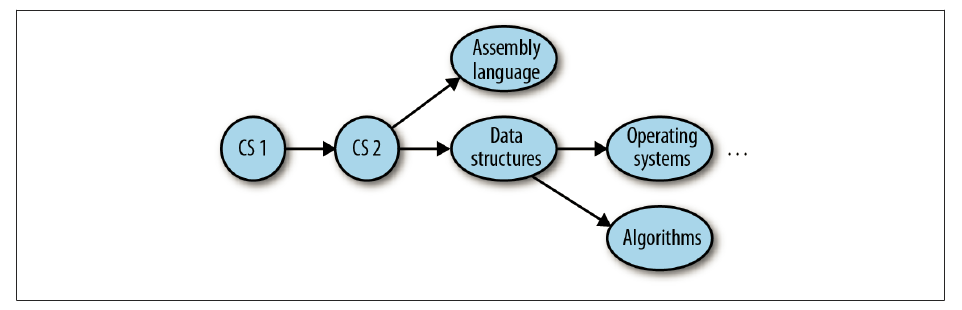


图11-6.计算机科学课程的有向图模型

这个图的拓扑排序会是以下序列：

1. CS 1
2. CS 2
3. Assembly language（汇编语言）
4. Data Structures（数据结构）
5. Operating Systems（操作系统）
6. Algorithms（算法）

课程3和4可以同时上，课程5和6也可以同时。

这类问题被称为优先级约束的调度算法（precedence-constrained scheduling），每个大学生都熟悉。只有先上英语写作1才能上英语写作2。

## 拓扑排序算法

拓扑排序算法类似于深度优先搜索。然而，算法不会立即打印已经访问的顶点，而是访问与当前顶点相邻的所有顶点，并且在列表穷尽时，将当前顶点压入栈中。

## 实现拓扑排序算法

拓扑排序算法被拆分为两个函数。第一个函数topSort()，会设置排序进程并调用一个帮助函数topSortHelper()，然后显示排序好的顶点列表。

主要的工作在递归函数topSortHelper()中完成。这个函数会标记当前顶点为已访问，然后递归访问当前顶点的邻接列表中的每个邻接顶点，标记这些顶点为已访问。最后，将当前顶点推入栈中。

示例11-5显示了这两个函数的代码。

示例11-5.topSort()和topSortHelper()

function topSort() {

var stack = [];

var visited = [];

for (var i = 0; i < this.vertices; i++) {

visited[i] = false;

}

for (var i = 0; i < this.vertices; i++) {

if (visited[i] == false) {

this.topSortHelper(i, visited, stack);

}

}

for (var i = 0; i < stack.length; i++) {

if (stack[i] != undefined && stack[i] != false) {

print(this.vertexList[stack[i]]);

}

}

}

function topSortHelper(v, visited, stack) {

visited[v] = true;

for each(var w in this.adj[v]) {

if (!visited[w]) {

this.topSortHelper(visited[w], visited, stack);

}

}

stack.push(v);

}

图类（Graph）也被修改，这样就可以应用于符号顶点（symbolic vertices）而不只是数字。在代码中，每个顶点都只是标注了数字，但是我们添加了一个数组vertexList来将各个顶点关联到一个符号（在我们的示例中为课程名称）。

要确保类的新定义清晰，我们在下面展示了完整的定义，包括用于拓扑排序的函数。showGraph()函数的定义也被修改，这样可以显示符号名称而不只是顶点数字。示例11-6展示了代码。

示例11-6.图类

function Graph(v) {

this.vertices = v;

this.vertexList = [];

this.edges = 0;

this.adj = [];

for (var i = 0; i < this.vertices; ++i) {

this.adj[i] = [];

this.ajd[i].push("");

}

this.addEdge = addEdge;

this.showGraph = showGraph;

this.dfs = dfs;

this.marked = [];

for (var i = 0; i < this.vertices; ++i) {

this.marked[i] = false;

}

this.bfs = bfs;

this.edgeTo = [];

this.hasPathTo = hashPathTo;

this.topSortHelper = topSortHelper;

this.topSort = topSort;

}

function topSort() {

var statck = [];

var visited = [];

for ( var i = 0; i < this.vertices; i++ ) {

visited[i] = false;

}

for ( var i = 0; i < stack.length; i++ ) {

if ( visited[i] == false ) {

this.topSortHelper(i, visited, stack);

}

}

for ( var i = 0; i < stack.length; i++ ) {

if (stack[i] != undefined && stack[i] != false){

print(this.vertexList[stack[i]]);

}

}

}

function topSortHelper(v, visited, stack) {

visited[v] = true;

for each(var w in this.adj[v]) {

if (!visited[w]) {

this.topSortHelper(visited[w], visited, stack);

}

}

stack.push(v);

}

function addEdge(v, w) {

this.adj[v].push(w);

this.adj[w].push(v);

this.edges++;

}

/\*

function showGraph() {

for (var i = 0; i < this.vertices; ++i) {

putstr(i + "->");

for (var j = 0; j < this.vertices; ++j) {

if (this.adj[i][j] != undefined) {

putstr(this.adj[i][j] + ' ');

}

}

print();

}

}

\*/

// a new function to display symbolic names instead of numbers

function showGraph() {

var visited = [];

for ( var i = 0; i < this.vertices; ++i) {

putstr(this.vertexList[i] + " -> ");

visited.push(this.vertexList[i]);

for ( var j = 0; j < this.vertices; ++j ) {

if (this.adj[i][j] != undefined) {

if (visited.indexOf(this.vertexList[j]) < 0) {

putstr(this.vertexList[j] + ' ');

}

}

}

print();

visited.pop();

}

}

function dfs(v) {

this.marked[v] = true;

if (this.adj[v] != undefined) {

print('Visited vertex: ' + v);

}

for each(var w in this.adj[v]) {

if (this.marked[w]) {

this.dfs(w);

}

}

}

function bfs(s) {

var queue = [];

this.marked[s] = true;

queue.unshift(s);

while (queue.length > 0) {

var v = queue.shift();

if (typeof(v) != 'string') {

print('Visited vertex: ' + v);

}

for each(var w in this.adj[v]) {

if (!this.marked[w]) {

this.edgeTo[w] = v;

this.marked[w] = true;

queue.unshift(w);

}

}

}

}

function hasPathTo(v) {

return this.marked[v];

}

function pathTo(v) {

var source = 0;

if (!this.hasPathTo(v)) {

return undefined;

}

var path = [];

for (var i = v; i != source; i = this.edgeTo[i]) {

path.push(i);

}

path.push(s);

return path;

}

拓扑排序实现的测试程序如示例11-7所示：

示例11-7.拓扑排序

load(‘Graph.js’);

g = new Graph(6);

g.addEdge(1, 2);

g.addEdge(2, 5);

g.addEdge(1, 3);

g.addEdge(1, 4);

g.addEdge(0, 1);

g.vertexList = [“CS1”, “CS2”, “Data Structures”, “Assembly Language”, “Operating Systems”, “Algorithms”];

g.showGraph();

g.topSort();

上面代码的输出为：

cs1

cs2

Data Structure

Assembly Language

Operating Systems

Algorithms

## 练习

1. 编写一个程序，确定宽度优先和深度优先搜索哪种图搜索速度更快。使用不同大小的图来测试你的程序。
2. 编写一个用文件来存储图的程序。
3. 编写一个从文件读取图的程序。
4. 构建一个图，为您生活区域的地图建模。确定从一个开始顶点到最后顶点的最短路径。
5. 在示例4创建的图中执行深度优先和宽度优先搜索。

# 第12章 排序算法

在计算机存储数据上执行的两种最常见操作为搜索和排序。这是从计算机工业开始便有的事实，这也意味着搜索和排序是计算机科学中研究最多的操作。本书中讨论的许多数据结构，其主要设计目标就是让搜索和排序数据结构中存储的数据更加容易、更加高效。

本章将向您介绍一些数据排序的基本和高级算法。这些算法都只依赖数组来存储数据。本章我们还将研究为了确定哪种算法最高效而使用的程序计时方法。

## 基于数组的测试

要开启本章，我们首先会开发一个用于支持基本排序算法研究的测试数组。我们会构建一个数组数据类以及封装了一些常用操作的函数：插入新数据、显示数组数据以及调用不同排序算法。这个类中还包含了一个swap()函数用于数组中的元素交换。

示例12-1展示了该类的代码

示例12-1.数组测试类

function CArray(numElements) {

this.dataStore = [];

this.pos = 0;

this.numElements = numElements;

this.insert = insert;

this.toString = toString;

this.clear = clear;

this.setData = setData;

this.swap = swap;

for ( var i = 0; i < numElements; ++i ) {

this.dataStore[i] = i;

}

}

function setData() {

for ( var i = 0; i < this.numElements; ++i ) {

this.dataStore[i] = Math.floor(Math.random() \* (this.numElements + 1));

}

}

function clear() {

for ( var i = 0; i < this.dataStore.length; ++i ) {

this.dataStore[i] = 0;

}

}

function insert(element) {

this.dataStore[this.pos++] = element;

}

function toString() {

var restr = "";

for ( var i = 0; i < this.dataStore.length; ++i ) {

retstr += this.dataStore[i] + " ";

if (i > 0 & i % 10 == 0) {

retstr += "\n";

}

}

return retstr;

}

function swap(arr, index1, index2) {

var temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

一个使用CArray类的简单程序（这个类被命名为CArray是因为JavaScript已经有了一个Array类）：

示例12-2.使用测试类

var numElements = 100;

var myNums = new CArray(numElements);

myNums.setData();

print(myNums.toString());

这段程序的输出为：

76 69 64 4 64 73 47 34 65 93 32

59 4 92 84 55 30 52 64 38 74

40 68 71 25 84 5 57 7 6 40

45 69 34 73 87 63 15 96 91 96

88 24 58 78 18 97 22 48 6 45

68 65 40 50 31 80 7 39 72 84

72 22 66 84 14 58 11 42 7 72

87 39 79 18 18 9 84 18 45 50

43 90 87 62 65 97 97 21 96 39

7 79 68 35 39 89 43 86 5

## 生成随机数据

你会注意到setData()函数生成了存储在数组中的随机数字。Math类的random()函数生成从0到1的随机数字（不包含0和1）。换句话说，random()函数生成的随机数字不会等于0，也不会等于1。这些随机数字并不是非常有用，因此我们通过将随机数字乘以我们想要的元素数目加一来扩展，并使用Math类的floor()函数来对最终的数字进行处理。正如上面的输出所见，这个公式可以成功的生成1到100之间的随机数字集。

更多关于JavaScript生成随机数字的信息，参考Mozilla关于随机数生成的页面***使用Math.random()函数***。

## 基本排序算法

接下来涉及的基本排序算法的根本理念在于有一组需要重新排序的数据。这些算法用于重新排列数据的技巧在于一组嵌套的for循环。外层循环会遍历数组项，而内层循环用于元素比较。这些算法非常确切地模拟了人类在真实生活中对数据排序的方法，类似于一个玩牌的人对纸牌进行排序的方法，或者一位教师按照字母顺序或者年纪顺序对试卷排序的方法。

## 冒泡排序

我们将了解的第一个排序算法是冒泡排序，冒泡排序是最慢的排序算法之一，但是也是最容易实现的排序算法之一。

冒泡排序的起名是因为使用这种排序算法存储数据时，数据值会像泡泡一样从数组的一端漂浮到另一端。假设您正在将一组数字按照升序排列，较大的值会浮动到数组的右侧，而较低的值会浮动到数组的左侧。这种行为是因为算法会将多次移动数组、比较邻近数组值，并在左侧值大于右侧值时将数据进行交换。

这里有一个冒泡排序的简单例子。我们从下面的列表开始：

E A D B H

排序的第一轮生成如下列表：

A E D B H

第一个元素和第二个元素被交换。排序下一轮的结果为：

A D E B H

第二个元素和第三个元素进行了交换。下一轮会变成如下顺序：

A D B E H

因为第三个和第四个元素进行了交换。最后，第二个和第三个元素再次发生了交换，形成了最终顺序：

A B D E H

图12-1展示了冒泡排序在更大的数据集上的运行方式。在该图中，我们检查插入数组中的两个特定值：2和72。每个数字都被画了一个圈。可以观察72是怎样从数组开图移动到数组中部的，也可以观察2是怎样从数组中间移动到数组开头的。



图12-1.冒泡排序

示例12-3是冒泡排序的代码：

示例12-3.bubbleSort()函数

function bubbleSort() {

var numElements = this.dataStore.length;

var temp;

for ( var outer = numElements; outer >= 2; --outer) {

for ( var inner = 0; inner <= outer - 1; ++inner ) {

if (this.dataStore[inner] > this.dataStore[inner + 1]) {

swap(this.dataStore, inner, inner + 1);

}

}

}

}

确保将这个函数添加到了CArray的构造函数中。示例12-4是一个使用bubbleSort()函数来对10个数字进行排序的简单程序：

示例12-4.使用bubbleSort()对10个数字排序

var numElements = 10;

var mynums = new CArray(numElements);

mynums .setData();

print(mynums.toString());

mynums.bubbleSort();

print();

print(mynums.toString());

这段程序的输出为：

10 8 3 2 2 4 9 5 4 3

2 2 3 3 4 4 5 8 9 10

我们可以看到冒泡排序算法的运行方式，但是最好能够看到算法运行的中间结果，因为排序过程记录对于帮助我们理解算法的运行方式非常有用。