# 图的邻接表表示

## 基本概念

图是一种数据结构，由节点和边两部分组成。

在有向图中，边是有方向的，因为(u,v)和(v,u)不同。(u,v)表示节点u到节点v的边。这个边可能包含权重、值以及距离。

图在现实生活中有很多代表性的应用：图经常被用来表示网络（例如：城市之间的道路交通网、电话网或电网）。图也经常用于微信和QQ等社交网络（例如：在微信中用一个节点代表一个人，每个节点都是包含身份证号、名字、性别以及家庭住址的结构体）。

下图是以5个节点为例的无向图，

Lightbox

图结构一般有两种表示方式：邻接矩阵和邻接列表。当然也有其他的表示方式，例如入射矩阵和入射列表。选择何种表示方式要视情况而定，这完全取决于图的用途和要执行的操作类型。

## 邻接矩阵

假设途中共有V个节点，那么邻接矩阵就是一个大小为VxV的二维数组。如果该二维数组用adj[ ][ ]表示，那么adj[i][j]=1就代表存在一条从节点i到节点j的边。不难发现，用邻接矩阵来表示无向图始终是对称的。如果adj[i][j]=w,那么就表示从节点i到节点j的路径权重为w。

用邻接矩阵表示上面的图为：

Adjacency Matrix Representation

优点：邻接矩阵这种表示方式非常容易实现和遍历。删除一条边只需要花费一个单位的时间，时间复杂度0(1)。查询是否存在节点u到节点v的路径效率也非常高，也只需要花费一个单位的时间，时间复杂度0(1)。

缺点：邻接矩阵要花费更多的存储空间，V个节点就要V^2个存储空，空间复杂度0(V^2)。即使这个图很分散（节点之间的边很少），他也还是要花费V^2个存储空间。并且构建这个邻接矩阵（添加节点）要花费V^2单位的时间，时间复杂度0(V^2)。

## 邻接列表

邻接列表是一个列表数组，该数组是一个大小等于节点数的一维数组。数组array[i]中存放的是一个链表，该链表中的n个数据表示的就是连接到第i个节点的n个节点。临界列表也可以用来表示加权图，边的权重可以用一对列表来表示。

用邻接列表表示上面的图为：

Adjacency List Representation of Graph

总结：如何适当的选用邻接矩阵和邻接列表？

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 适用特性 | 图中的节点数 | 图的结构 | 节点访问次数 | 路径更改次数 |
| 邻接矩阵 | 少 | 密集 | 频繁 | 频繁 |
| 邻接列表 | 多 | 稀疏 | 偶尔 | 偶尔 |

## C语言实现

**（一）如何实现邻接表呢？首先要构建邻接表的存储类型。**

1>我们先要有一个结构体graph来表示图，graph结构体中有两部分信息，一是此图的节点数，二是链表数组array[i]，而这个array[i]对应的就是第i个节点的邻接链表，array[0]到array[V-1]组成的整体就是整个图的邻接表。

2>接下来看表示邻接表的结构体graph->array[i]，这个结构体中只包含一个指向节点的指针称为“头指针”（本质上是链表中第一个节点的存储地址）。

3>最后，观察表示邻接表节点的结构体。由于邻接表是以链表形式存储的，所以每个节点结构体都包含两部分内容，即当前节点的值dest与下一个节点的存储位置next。

**（二）以上就是图的存储结构，接下来讨论如何建立一个图。**

1>首先声明图中有V个节点，然后根据节点数V用createGraph(V)函数构建一个空的图。具体操作如下：①先声明graph是指向图结构体的指针，再对该指针指向地址分配空间，大小是(struct Graph)的一个单元，并强制类型转换，graph->V = V对节点数赋值；②在图结构体中已经声明过array是指向邻接表结构体的指针，故不用再次声明，对该指针指向地址分配空间，大小是(struct AdjList)的V个单元，并强制类型转换；③将每个graph->array[i].head赋值为空，表示各个链表中均无结点，当前邻接表为空。

2>通过addEdge()函数向上一步创建的图中添加边（即向邻接表中添加节点），该函数传入参数为两个整型数据“源节点src”与“目标节点dest”。具体操作如下：①先创建一个空节点check用于检验链表末尾来找到合适的节点插入位置；②因为传入的dest并不是节点而是整型数据，所以要先用dest的值构建一个新的节点newNode来插入链表；③如果源节点src对应的链表头为空，那么就将目标节点直接放在该链表的第一个（头指针指向该新节点，新节点的下一个节点为空），否则用check来查找该链表最后一个节点，并将新节点挂在之后（细节如上不再赘述）；④最后因为该图是无向图，节点之间的连接是双向的，所以把原来的源节点src当作目标节点，把原来的目标节点dest当作源节点，继续一次第三步的操作。

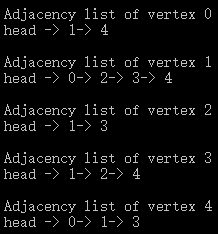
（三）至此，我们知道了无向图的存储结构与建立方式，最后来看一下无向图的遍历。

1>无向图遍历较为简单，先输出某源节点再访问该源节点对应的链表，依次打印链表之中的节点直至结束即可。

2>换下一个源节点继续1中的步骤，待所有源节点遍历完成即可。

|  |
| --- |
| // 一个C程序演示邻接表（图的表示法）  #include <stdio.h>  #include <stdlib.h>    // 表示邻接表节点的结构体  struct AdjListNode {  int dest;  struct AdjListNode\* next; // 指向节点的指针  };    // 表示邻接表的结构体  struct AdjList {  struct AdjListNode\* head; // 也是指向节点的指针  };    // 表示图的结构体. 图是邻接表的数组  // 数组的大小为V(图中的顶点数)  struct Graph {  int V;  struct AdjList\* array;  };    // 创建一个新的邻接表节点的函数  struct AdjListNode\* newAdjListNode(int dest) // 返回值类型是指向 邻接表节点结构体 的指针  {  struct AdjListNode\* newNode; // 先声明newNode是指向邻接表节点结构体的指针  newNode= (struct AdjListNode\*)malloc(sizeof(struct AdjListNode)); // 其大小是一个(struct AdjListNode)  newNode->dest = dest; // 当前节点是dest  newNode->next = NULL; // 下一个节点为空  return newNode;  }    // 创建V顶点图的效用函数  struct Graph\* createGraph(int V) // 返回值类型是指向 图结构体 的指针  {  // struct Graph\* graph= (struct Graph\*)malloc(sizeof(struct Graph));  struct Graph\* graph; // 先声明graph是指向图结构体的指针  graph= (struct Graph\*)malloc(sizeof(struct Graph)); // 再对该指针指向地址分配空间， 大小是(struct Graph)的一个单元，并强制类型转换  graph->V = V; // 图结构体下有五个节点    // 创建邻接表数组。（数组大小为V） // array是指向 邻接表结构体 的指针，在图结构体中已经声明过  graph->array = (struct AdjList\*)malloc( // 再对该指针指向地址分配空间， 大小是(struct AdjList)的V个单元，并强制类型转换  V \* sizeof(struct AdjList));    // 通过将head设置为NULL，将每个邻接表初始化为空  int i;  for (i = 0; i < V; ++i)  graph->array[i].head = NULL;    return graph;  }    // 向无向图添加一条边  void addEdge(struct Graph\* graph, int src, int dest)  {  // 添加一条从src到dest的路劲  // 向节点src的邻接表中添加新的节点  // 节点是在开头添加的  struct AdjListNode\* check = NULL; // check是邻接表节点结构体，用于检查邻接表末尾  struct AdjListNode\* newNode = newAdjListNode(dest); // 将目标节点创建为新节点结构体    if (graph->array[src].head == NULL) { // 如果图中的src邻接表表头为空（原来没节点=>需要放的是第一个节点）  newNode->next = graph->array[src].head; // 那么目标节点是放在邻接表的第一个节点，即目标节点之后的节点为空  graph->array[src].head = newNode; // 邻接表里面有了一个节点，该节点是newNode  }  else{ // 如果图中的src邻接表表头不为空（邻接表已经有节点了）  check = graph->array[src].head; // 从邻接表中第一个节点开始检查  while (check->next != NULL) // 一直检测直到邻接表末尾  {  check = check->next;  }  check->next = newNode; // 将新的待添加节点放在邻接表末尾  }    // 由于图形是无向的，添加一条边从dest到src也  newNode = newAdjListNode(src); // 将原来的源节点构建为待添加的新节点  if (graph->array[dest].head == NULL) { // 将原来的目标节点作为新的源节点进行相同操作  newNode->next = graph->array[dest].head;  graph->array[dest].head = newNode;  }  else {  check = graph->array[dest].head;  while (check->next != NULL) {  check = check->next;  }  check->next = newNode;  }  }    // 打印图的邻接表表示的实用函数  void printGraph(struct Graph\* graph)  {  int v;  for (v = 0; v < graph->V; ++v) { // 遍历源节点  struct AdjListNode\* pCrawl; // 声明 pCrawl是指向目标节点的指针  pCrawl = graph->array[v].head; // 这个目标节点刚开始的时候是源节点邻接表中的第一个节点（头节点）  printf("\n Adjacency list of vertex %d\n head ", v); // 先输出源节点  while (pCrawl) { // 如果目标节点存在就输出目标节点  printf("-> %d", pCrawl->dest);  pCrawl = pCrawl->next; // 目标节点指向下一个  }  printf("\n"); // 当前源节点对应的所有目标节点输出完毕就换行  }  }    // 驱动程序来测试以上功能  int main()  {  // 创建上面给出的图表  int V = 5;  struct Graph\* graph = createGraph(V);  addEdge(graph, 0, 1);  addEdge(graph, 0, 4);  addEdge(graph, 1, 2);  addEdge(graph, 1, 3);  addEdge(graph, 1, 4);  addEdge(graph, 2, 3);  addEdge(graph, 3, 4);  // 打印上面图的邻接表表示  printGraph(graph);    return 0;  } |

程序运行效果图如下：



## 转为适合用汇编实现的C语言代码

由上述程序可知，该代码中包含malloc动态分配内存空间的命令。此类命令非常不利于用汇编语言实现，故分析上述代码链表的存储结构，然后用数组代替链表实现相同操作。

要求一：首先要去除malloc与结构体，改用数组实现，这样做就丧失了随机存储的优点。

要求二：其次是不能改变链表最重要的特点，每个节点都包含本节点的值与指向下一节点的地址，并且还要保持链表存储的无序性。

综上所述：①用一个一维数组代替一个graph结构体；②该数组中前V个存储单元即是V个节点的头指针，初始值为0表示空；③后面每两个存储单元表示一个节点，其中第一个存储单元是下一个节点在数组中的存储位置（即指向下一个单元的指针，也是存储地址），第二个存储单元是当前节点的值；④不难算出，如果一个无向图有V个节点和i条边，总共需要V个头指针与2\*i个节点，一个节点需要两个存储单元，所以共需要（V+4\*i）个存储单元。本例中共5+4\*7=33个存储单元。

更改存储结构之后，构建图的邻接表与其遍历都和之前大同小异，在此不再过多赘述。

|  |
| --- |
| #include <stdio.h>  #define node\_number 5 // 图节点数  #define max\_edge\_number 7 // 无向图最多有几条边边  #define max\_graph\_size 33 // 4\*7+5  int graph[max\_graph\_size]={0}; // 前五个数字单个一组，存储链表中下个节点位置。后面的两个一组，（值，下个节点位置）  int graph\_label=node\_number; // 起始存储位置  int AddOneNode(int src,int dest) // 添加路径（源节点，目标节点）  {  if(graph[src]==0) // 如果图中的src邻接表表头为空（原来没节点=>需要放的是第一个节点）  {  graph[src]=graph\_label;  graph\_label++;  graph[graph\_label]=dest;  graph\_label++;  }  else  {  int check=src;  while(graph[check]!=0)  {  check = graph[check];  }  graph[check]=graph\_label;  graph\_label++;  graph[graph\_label]=dest;  graph\_label++;  }  if(graph[dest]==0) // 如果图中的src邻接表表头为空（原来没节点=>需要放的是第一个节点）  {  graph[dest]=graph\_label;  graph\_label++;  graph[graph\_label]=src;  graph\_label++;  }  else  {  int check=dest;  while(graph[check]!=0)  {  check = graph[check];  }  graph[check]=graph\_label;  graph\_label++;  graph[graph\_label]=src;  graph\_label++;  }  }  void printGraph(int node)  {  for(int i=0;i<node\_number;i++)  {  int check;  printf(" \n Adjacency list of vertex %d \n head",i);  for(check=graph[i];check!=0;check=graph[check])  {  printf(" -> %d",graph[check+1]);  }  printf("\n");  }  }  void printArray()  {  printf("\n\n array is : ");  for(int i=0;i<max\_graph\_size;i++)  {  printf("%d ",graph[i]);  }  printf("\n");  }  int main()  {  AddOneNode(0,1);  AddOneNode(0,4);  AddOneNode(1,2);  AddOneNode(1,3);  AddOneNode(1,4);  AddOneNode(2,3);  AddOneNode(3,4);  printGraph(node\_number);  printArray();  } |

程序最终效果如下：



由上图可知，图的邻接表完全正确，并且数组形式存储的链表满足预定要求。

## 汇编语言实现

用LoongIDE新建一个工程。

移除core文件夹下的bsp\_start.c源文件。

移除main.c源文件。

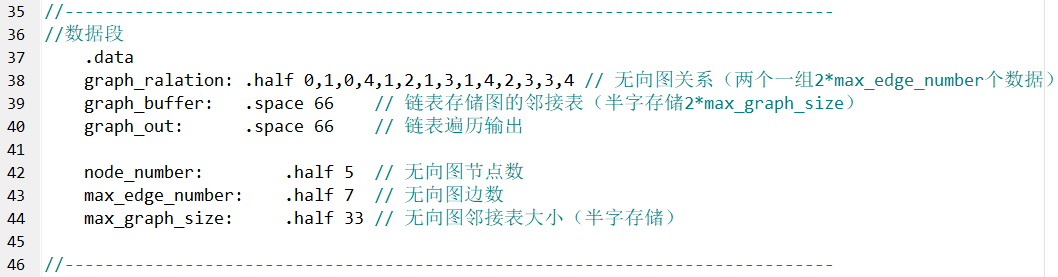
添加bsp\_start.S源文件。

在bsp\_start.S下添加如下代码：

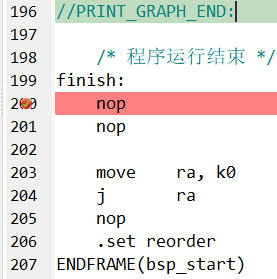
|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/2/19  \* author: Li TianLing  \* 链表表示无向图的邻接表  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 存储器寻址辅助寄存器  \* v1 跳转辅助寄存器  \*  \* s0 无向图关系graph\_ralation基地址 （常量）  \* s1 无向图邻接表graph\_buffer基地址 （常量）  \* s2 遍历输出邻接表graph\_out基地址 （常量）  \* s3 无向图节点数node\_number （常量）（坐标.半字翻倍）  \* s4 无向图边数max\_edge\_number （常量）  \* s5 无向图邻接表大小max\_graph\_size （常量）（坐标.半字翻倍）  \*  \* t0 参数src （源节点）  \* t1 参数dest （目标节点）  \* t2 参数graph\_label （节点存储位置）  \* t3 参数check  \* t4 哨兵i  \* t5 无向图关系最大坐标  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  //-----------------------------------------------------------------------------  //数据段  .data  graph\_ralation: .half 0,1,0,4,1,2,1,3,1,4,2,3,3,4 // 无向图关系（两个一组2\*max\_edge\_number个数据）  graph\_buffer: .space 66 // 链表存储图的邻接表（半字存储2\*max\_graph\_size）  graph\_out: .space 66 // 链表遍历输出    node\_number: .half 5 // 无向图节点数  max\_edge\_number: .half 7 // 无向图边数  max\_graph\_size: .half 33 // 无向图邻接表大小（半字存储）  //-----------------------------------------------------------------------------  //代码段  .text  FRAME(bsp\_start,sp,0,ra)  .set noreorder  move k0, ra /\* 返回地址 \*/  la s0,graph\_ralation // 将graph\_ralation基地址加载到s0（常量）  la s1,graph\_buffer // 将graph\_buffer基地址加载到s1 （常量）  la s2,graph\_out // 将graph\_out基地址加载到s2 （常量）    la v0,node\_number // 将node\_number加载到s3 （常量）  lh s3,(v0)  sll s3,1  la v0,max\_edge\_number // 将max\_edge\_number加载到s4（常量）  lh s4,(v0)  la v0,max\_graph\_size // 将max\_graph\_size加载到s5 （常量）  lh s5,(v0)  sll s5,1  move t2,s3 // 初始化graph\_label=node\_number  la t4,0 // 哨兵i，遍历关系数组  move t5,s4 // 关系数组最大坐标=边长\*2\*2  sll t5,2  for\_add\_all\_node\_begin:  bge t4,t5,for\_add\_all\_node\_end // t4>=t5,遍历完关系矩阵，节点添加完成  nop    lh t0,graph\_ralation(t4) // 加载源节点src  sll t0,1 // src\*2j用于坐标计算  add t4,2  lh t1,graph\_ralation(t4) // 加载目标节点dest  sll t1,1 // dest\*2j用于坐标计算  add t4,2    //ADD\_ONE\_NODE\_BEGIN:  if\_src\_begin:  lh s6,graph\_buffer(t0) // 加载graph[src]  bne s6,zero,else\_sec // 非空跳转到else  nop    sh t2,graph\_buffer(t0) // graph[src]=graph\_label  add t2,2 // graph\_label=graph\_label+2  srl t1,1 // dest/2用于存储  sh t1,graph\_buffer(t2) // graph[graph\_label]=dest  sll t1,1 // dest\*2用于坐标计算  add t2,2 // graph\_label=graph\_label+2    b if\_sec\_end  nop  else\_sec:  move t3,t0 // 初始化check=src  while\_src\_begin:  lh s6,graph\_buffer(t3) // 将graph[check]加载到s6  beq s6,zero,while\_src\_end // 为0跳转，结束while  nop  move t3,s6 // check=graph[check]  b while\_src\_begin  nop  while\_src\_end:  sh t2,graph\_buffer(t3) // graph[check]=graph\_label  add t2,2 // graph\_label=graph\_label+2  srl t1,1 // dest/2用于存储  sh t1,graph\_buffer(t2) // graph[graph\_label]=dest  sll t1,1 // dest\*2用于坐标计算  add t2,2 // graph\_label=graph\_label+2    if\_sec\_end:  //add\_src\_node\_end\_&\_add\_dest\_node\_begin:  if\_dest\_begin:  lh s6,graph\_buffer(t1) // 加载graph[dest]  bne s6,zero,else\_dest // 非空跳转到else  nop  sh t2,graph\_buffer(t1) // graph[dest]=graph\_label  add t2,2 // graph\_label=graph\_label+2  srl t0,1 // src/2用于存储  sh t0,graph\_buffer(t2) // graph[graph\_label]=src  sll t0,1 // src\*2用于坐标计算  add t2,2 // graph\_label=graph\_label+2  b if\_dest\_end  nop  else\_dest:  move t3,t1 // 初始化check=dest  while\_dest\_begin:  lh s6,graph\_buffer(t3) // 将graph[check]加载到s6  beq s6,zero,while\_dest\_end // 为0跳转，结束while  nop  move t3,s6 // check=graph[check]  b while\_dest\_begin  nop  while\_dest\_end:  sh t2,graph\_buffer(t3) // graph[check]=graph\_label  add t2,2 // graph\_label=graph\_label+2  srl t0,1 // src/2用于存储  sh t0,graph\_buffer(t2) // graph[graph\_label]=src  sll t0,1 // src\*2用于坐标计算  add t2,2 // graph\_label=graph\_label+2  if\_dest\_end:  //ADD\_ONE\_NODE\_END:  b for\_add\_all\_node\_begin  nop  for\_add\_all\_node\_end:  //PRINT\_GRAPH\_BEGIN:  la t2,0 // 初始化graph\_label=0，辅助输出  la t4,0 // i=0  for\_i\_begin:  bge t4,s3,for\_i\_end // i>=node\_number,遍历完所有节点，结束  nop    srl t4,1 // i/2，数据输出  sh t4,graph\_out(t2) // graph\_out[graph\_label]=i  add t2,2 // graph\_label=graph\_label+2  sll t4,1 // i\*2，坐标计算    lh t3,graph\_buffer(t4) // check=graph[i]  for\_check\_begin:  beq t3,zero,for\_check\_end // check=0结束  nop    add t3,2 // check=check+2,用于输出数据  lh s6,graph\_buffer(t3) // 将graph[check]加载到s6  sub t3,2 // check=check-2,恢复check  sh s6,graph\_out(t2) // 将t6保存到graph\_out[graph\_label]  add t2,2 // graph\_label=graph\_label+2  lh t3,graph\_buffer(t3) // check=graph[check]  b for\_check\_begin  nop  for\_check\_end:    add t2,4 // graph\_label=graph\_label+6 【注】：两个0作为结束标志    add t4,2 // i=i+2  b for\_i\_begin  nop  for\_i\_end:  //PRINT\_GRAPH\_END:  /\* 程序运行结束 \*/  finish:  nop  nop  move ra, k0  j ra  nop  .set reorder  ENDFRAME(bsp\_start) |

## 调试运行

初始数据如下所示：



如下图所示，在finish后打上断点：



联合调试后，双击s1寄存器，即可查看邻接表的存储（乱序的链表形式），由于数据是半字类型存储的，一个半字占用两个字节，所以该地址与C语言中的地址相比会翻倍：



双击s2寄存器，查看邻接表的输出，以两个连续的0作为间隔符：



由上图可以看出，邻接表为：

|  |
| --- |
| 0 -> 1 -> 4 |
| 1 -> 0 -> 2 -> 3 -> 4 |
| 2 -> 1 -> 3 |
| 3 -> 1 -> 2 -> 4 |
| 4 -> 0 -> 1 -> 3 |

经验证，结果正确。