# 深度优先搜索和广度优先搜索

## 基本概念

度：某个顶点的度数即为与其相连的边的总数。

入度：存在有向图中, 所有的指向某个顶点的边的总数。

出度：存在有向图中, 某个顶点指向其他顶点的边的总数。

## DFS和BFS

1. **深度优先搜索** (Depth First Search, DFS)基本思路：

深度优先搜索遍历图的方法是,从图中某个顶点出发访问指定的起始顶点。

若当前访问的顶点的邻接顶点有未被访问的,则任选一个顶点访问之；反之, 退回到发现当前访问节点的那条边的起始节点；直到与起始顶点相通的全部顶点都访问完毕。

1. **广度优先搜索** (Breadth First Search, BFS)基本思路：

先将指定的搜索顶点加入到队列中

然后从队列中取出一个顶点,将其标记为已被访问

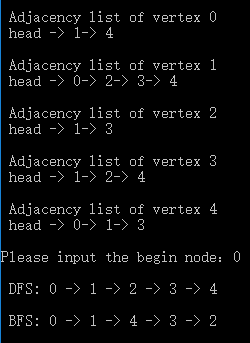
得到该顶点的所有邻接顶点并将其中没有被访问过的顶点加入到队列中去转入步骤2继续从队列取出顶点,直到队列中不存在任何顶点.

## 图的构建

我们还是以图的表示法中讲到的例子为例：

Lightbox

用邻接列表表示上面的图为：



## C语言实现

**深度优先搜索过程：**

1>从节点0开始，输出节点0，进入节点0的邻接表。

2>节点0的邻接表第一个元素是1且没遍历过，所以输出节点1，并进入节点1的邻接表。

3>节点1的邻接表第一个元素是0且被遍历过，继续搜索下一个节点2，节点2没遍历过，输出节点2，进入节点2的邻接表。

4>节点2的邻接表第一个元素是1且被遍历过，继续搜索下一个节点3，节点3没遍历过，输出节点3，进入节点3的邻接表。

5>节点3邻接表中的节点1与节点2都已经被遍历过，所以输出节点4，进入节点4的邻接表。

6>节点4邻接表中所有元素均被遍历过，结束搜索。

7>所以最终结果为：0-1-2-3-4。

**广度优先搜索过程：**

1>从节点0开始遍历，输出节点0，节点0入栈[0]。

2>节点0出栈[空]，输出节点1并入栈[1]，输出节点4并入栈[1,4]。

3>节点4出栈[1]，节点0与节点1被遍历过，输出节点3并入栈[1,3]。

4>节点3出栈[1]，节点1被遍历过，输出节点2并入栈[1,2]，节点4被遍历过。

5>节点2出栈[1]，节点1和节点3都被遍历过。

6>节点1出栈[空]，节点0、2、3以及4都被遍历过。

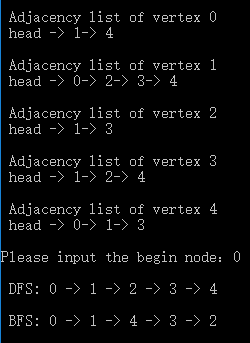
7>堆栈为空，搜索结束。

8>所以最终结果为：0-1-4-3-2。

|  |
| --- |
| // 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  }  }      /\* 拓展->深度优先搜索 \*/  void DFS(struct Graph \*graph,int node,int\* Visited)  {  printf("\n DFS: %d ",node); // 输出开始的节点  Visited[node] = 1; // 这个节点已经看过了  struct AdjListNode \*check\_node = graph->array[node].head; // 创建一个节点用于遍历（起始节点第一个节点）  while(check\_node) // 这个节点非空就遍历  {  if(!Visited[check\_node->dest]) // 如果这个新的节点没被哈看过，就搜索新节点  DFS(graph,check\_node->dest,Visited);  check\_node = check\_node->next; // 跳入下一个节点进行搜索  }  printf("\n");  }      /\* 拓展->总结上述递归似是而非，改为非递归 \*/  void DFS\_Pro(struct Graph \*graph,int node,int\* Visited)  {  printf("\n DFS: %d",node); // 输出开始的节点  Visited[node] = 1; // 这个节点已经看过了  struct AdjListNode \*check\_node = graph->array[node].head; // 创建一个节点用于遍历（起始节点第一个节点）  while(check\_node) // 这个节点非空就遍历  {  if(!Visited[check\_node->dest]) // 如果这个新的节点没被看过  {  printf(" -> %d",check\_node->dest);  Visited[check\_node->dest] = 1;  check\_node = graph->array[check\_node->dest].head;  continue;  }  check\_node = check\_node->next; // 跳入下一个节点进行搜索  }  printf("\n");  }  /\* 拓展->广度优先搜索 \*/  void BFS(struct Graph \*graph,int node,int\* Visited,int v)  {  int node\_buffer[v] = {0};  int buffer\_flag = 0;  printf("\n BFS: %d",node); // 输出开始的节点  Visited[node] = 1; // 开始节点已经看过了  node\_buffer[buffer\_flag++] = node; // 开始节点入栈    struct AdjListNode \*check\_node ; // 创建一个检验节点用于遍历  while(buffer\_flag) // 这个节点缓存栈非空  {  check\_node = graph->array[node\_buffer[--buffer\_flag]].head;  while(check\_node)  {  if(!Visited[check\_node->dest]) // 如果这个新的节点没被看过  {  printf(" -> %d",check\_node->dest); // 输出  Visited[check\_node->dest] = 1; // 看过  node\_buffer[buffer\_flag++] = check\_node->dest; // 入栈  }  check\_node = check\_node->next; // 链接的下一个节点  }  }  printf("\n");  }    // 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);    /\* 图的深度优先搜索DFS \*/  int begin\_node;  int Visited[V]={0}; // 已经看过的节点，初始值都是0代表都没看过  printf("\nPlease input the begin node：");  scanf("%d",&begin\_node);  //DFS(graph,begin\_node,Visited); // 开始遍历——DFS递归实现  DFS\_Pro(graph,begin\_node,Visited); // 开始遍历——DFS非递归实现  for(int i=0;i<V;i++) // 清空观测矩阵  Visited[i]=0;  BFS(graph,begin\_node,Visited,V); // 开始遍历——BFS非递归实现    return 0;  } |

程序运行效果图如下：

输入0按下回车后出现下图



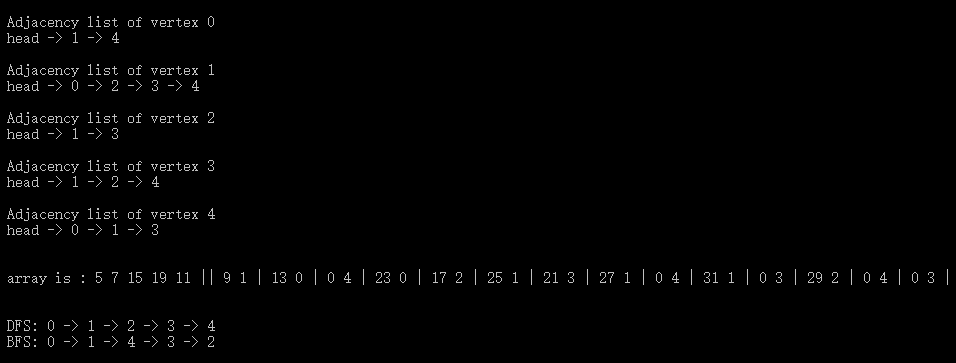
## 转为适合用汇编实现的C语言代码

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。

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 Visited[node\_number]={0}; // 数组->已经观测标志  /\* 添加路劲 \*/  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++;  }  }  /\* 打印图关系 \*/  void printGraph()  {  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");  }  }  /\* 改进->打印数组存储结构 \*/  void printArray()  {  int i,j;  printf("\n\n array is : ");  for(i=0;i<node\_number;i++)  printf("%d ",graph[i]);  printf("|| ");  for(;i<max\_graph\_size;i++)  {  printf("%d ",graph[i]);  j++;  if(j%2==0)  printf("| ");  }  printf("\n");  }  /\* 拓展->深度优先搜索DFS \*/  void printDFS(int node)  {  printf("\n DFS: %d",node); // 输出开始的节点  Visited[node] = 1; // 这个节点已经看过了  int check = graph[node]; // check是地址  while(check) // check非0，有连接着下一个点  {  if(!Visited[graph[check+1]]) // graph[check+1]是节点值  {  printf(" -> %d",graph[check+1]);  Visited[graph[check+1]] = 1;  check = graph[check+1]; // check=头指针  check = graph[check]; // check=头指针指向的节点  continue;  }  check = graph[check];  }  }  /\* 拓展->广度优先搜索BFS \*/  void printBFS(int node,int v)  {  int node\_buffer[v] = {0};  int buffer\_flag = 0;  printf("\n BFS: %d",node); // 输出开始的节点  Visited[node] = 1; // 开始节点已经看过了  node\_buffer[buffer\_flag++] = node; // 开始节点入栈    int check;  while(buffer\_flag) // 这个节点缓存栈非空  {  check=graph[node\_buffer[--buffer\_flag]];  while(check)  {  if(!Visited[graph[check+1]]) // graph[check+1]是节点值  {  printf(" -> %d",graph[check+1]);  node\_buffer[buffer\_flag++] = graph[check+1]; // 入栈  Visited[graph[check+1]] = 1;  }  check = graph[check];  }  }  printf("\n");  }  /\* 主函数 \*/  int main()  {  AddOneNode(0,1);  AddOneNode(0,4);  AddOneNode(1,2);  AddOneNode(1,3);  AddOneNode(1,4);  AddOneNode(2,3);  AddOneNode(3,4);  printGraph(); // 打印图  printArray(); // 打印存储结构  printDFS(0); // 深度优先搜索  for(int i=0;i<node\_number;i++) // 清空观测矩阵  Visited[i]=0;  printBFS(0,5); // 广度优先搜索  } |

程序最终效果如下：



## 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。

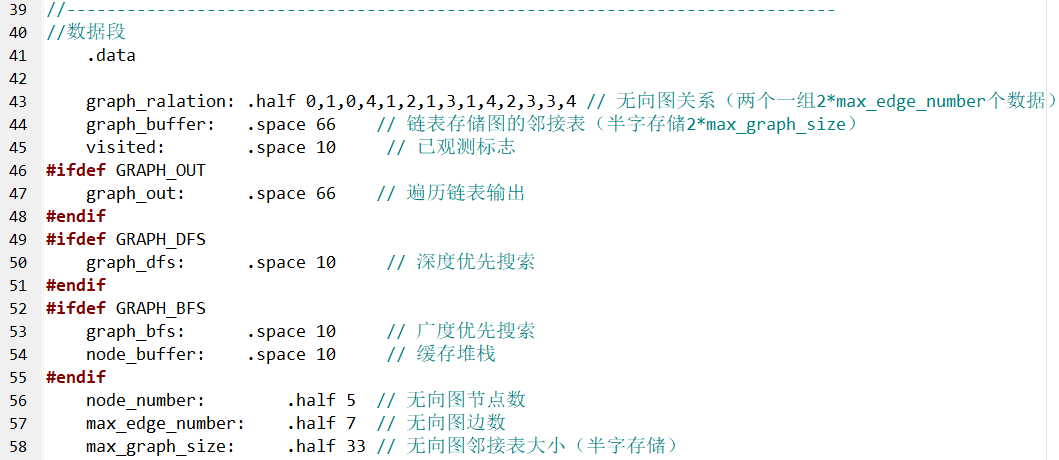
添加bsp\_start.S源文件。

在bsp\_start.S下添加如下代码：

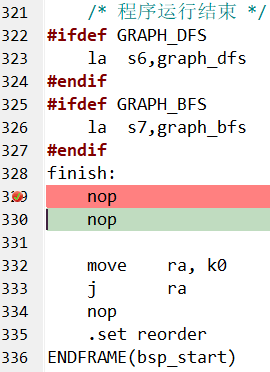
|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/2/23  \* author: Li TianLig  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 存储器寻址辅助寄存器  \* v1 跳转辅助寄存器or观测标志  \*  \* s0 无向图关系graph\_ralation基地址 （常量）  \* s1 无向图邻接表graph\_buffer基地址 （常量）  \* s2 遍历输出邻接表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）（之后用于DFS）  \* t3 参数check  \* t4 哨兵i  \* t5 无向图关系最大坐标  \* t6 DFS and BFS输出位置  \* t7 buffer\_flag  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  #define GRAPH\_OUT  #define GRAPH\_DFS  #define GRAPH\_BFS  //-----------------------------------------------------------------------------  //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）  visited: .space 10 // 已观测标志  #ifdef GRAPH\_OUT  graph\_out: .space 66 // 遍历链表输出  #endif  #ifdef GRAPH\_DFS  graph\_dfs: .space 10 // 深度优先搜索  #endif  #ifdef GRAPH\_BFS  graph\_bfs: .space 10 // 广度优先搜索  node\_buffer: .space 10 // 缓存堆栈  #endif  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）  #ifdef GRAPH\_OUT  la s2,graph\_out // 将graph\_out基地址加载到s2 （常量）  #endif  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:  #ifdef GRAPH\_OUT /\* 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:  #endif /\* PRINT\_GRAPH\_END \*/  #ifdef GRAPH\_DFS /\* PRINT\_DFS\_BEGIN \*/  /\* 默认从0开始搜索,v1=1代表已观测 \*/  la v1,1 // 观测标志初始化  la t6,0 // DFS输出位置初始化  sh zero,graph\_dfs(t6) // 保存第一个节点0  add t6,2  sh v1,visited(zero) // visited[0]=1  lh t3,graph\_buffer(zero)// check=graph[node]    while\_DFS\_begin:  beq t3,zero,while\_DFS\_end// check=0表示无连接点，结束while  nop    if\_DFS\_begin:  add v0,t3,2 // v0=check+2  lh v0,graph\_buffer(v0) // v0=graph[v0]  sll v0,1 // v0\*2坐标  lh v0,visited(v0) // v0=visited[v0]  beq v0,v1,if\_DFS\_end // v0=1表示已观测，结束if  nop    add v0,t3,2 // v0=graph[check+2]  lh v0,graph\_buffer(v0)  sh v0,graph\_dfs(t6) // 保存v0  add t6,2  sll v0,1 // v0\*2坐标  sh v1,visited(v0) // 标记为已观测    move t3,v0 // check=graph[check+2]  lh t3,graph\_buffer(t3) // check=graph[check]    b while\_DFS\_begin // conntinue  nop  if\_DFS\_end:    lh t3,graph\_buffer(t3) // check=graph[check]    b while\_DFS\_begin  nop  while\_DFS\_end:  #endif /\* PRINT\_DFS\_END \*/  #ifdef GRAPH\_BFS  la t4,0 // i=0  for\_clean\_visited\_begin:  bge t4,s3,for\_clean\_visited\_end  nop    sh zero,visited(t4)    add t4,2  b for\_clean\_visited\_begin  nop  for\_clean\_visited\_end:  /\* PRINT\_BFS\_BEGIN \*/  la v1,1 // 观测标志初始化  la t6,0 // BFS输出位置初始化  la t7,0 // buffer\_flag初始化  sh zero,graph\_bfs(t6) // 保存第一个节点0  add t6,2  sh v1,visited(zero) // visited[0]=1  sh zero,node\_buffer(t7)// 开始节点入栈  add t7,2    while\_buffer\_begin:  beqz t7,while\_buffer\_end// buffer\_flag=0堆栈为空，结束while  nop    sub t7,2 // 出栈  lh t3,node\_buffer(t7) // check=node\_buffer[buffer\_flag]  sll t3,1 // check\*2坐标  lh t3,graph\_buffer(t3) // check=graph[check]    while\_BFS\_begin:  beq t3,zero,while\_BFS\_end// check=0表示无连接点，结束while  nop    if\_BFS\_begin:  add v0,t3,2 // v0=check+2  lh v0,graph\_buffer(v0) // v0=graph[v0]  sll v0,1 // v0\*2坐标  lh v0,visited(v0) // v0=visited[v0]  beq v0,v1,if\_BFS\_end // v0=1表示已观测，结束if  nop    add v0,t3,2 // v0=graph[check+2]  lh v0,graph\_buffer(v0)  sh v0,graph\_bfs(t6) // 保存v0  add t6,2  sh v0,node\_buffer(t7) // 入栈v0  add t7,2  sll v0,1 // v0\*2坐标  sh v1,visited(v0) // 标记为已观测  if\_BFS\_end:    lh t3,graph\_buffer(t3) // check=graph[check]    b while\_BFS\_begin  nop  while\_BFS\_end:    b while\_buffer\_begin  nop  while\_buffer\_end:  #endif /\* PRINT\_BFS\_END \*/  /\* 程序运行结束 \*/  #ifdef GRAPH\_DFS  la s6,graph\_dfs  #endif  #ifdef GRAPH\_BFS  la s7,graph\_bfs  #endif  finish:  nop  nop  move ra, k0  j ra  nop  .set reorder  ENDFRAME(bsp\_start) |

## 调试运行

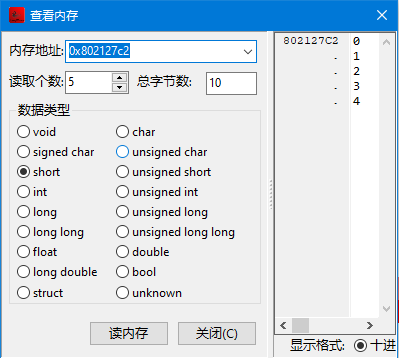
初始数据如下所示：



如下图所示，在finish后打上断点：



联合调试后，双击s6寄存器，即可查看深度优先搜索结果：



双击s7寄存器，查看广度优先搜索结果：

