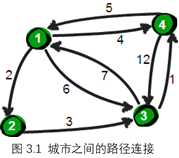
# 弗洛伊德算法

## 弗洛伊德算法原理

为了说明弗洛伊德算法，下面举一个例子。暑假即将来临前，小美准备去一些城市旅游。有些城市之间有公路，有些城市之间没有公路，如图3.1所示。为了节省经费以及方便计划旅程，小美希望在出发之前知道任意两个城市之间的最短路径



从图3.1可知，4个城市之间有8条公路，公路上的数字表示这条公路的长短。图中的公路是单向的。现在的需求是计算两个城市之间的最短路程，也就是求任意两点之间的最短路径。

现在，需要一个数据结构来存储图的信息，可以用一个4×4的二维矩阵e\_(4×4)来存储，即

= （式3.1）

从上面的矩阵可知，从编号为1的城市到编号为2的城市的路程为2，则e[1][2]=2。因为2号城市无法直接到达4号城市，则e[2][4]=∞。此外，约定编号为i的城市达到编号为i的城市路径为0，即e[1][1]=0，e[2][2]=0，e[3][3]=0，e[4][4]=0。

现在遇到的问题是，如何求取两个城市之间的最短路径呢？显然，通过深度优先搜索或者广度优先搜索可以求出两点之间的最短路径。所以，进行n^2遍的深度或广度优先搜索，即对每两个点都进行一次深度或广度优先搜索，就可以求取任意两点之间的最短路径。那还有没有其他方法呢？

换个角度思考，根据一些常识，如果要让任意亮点（比如从a点到b点）之间的路径变短，只能引入第三个点（如k点），即路径从原来的a直接到b（a→b）改成从a经过k到b（a→k→b），这样才可能缩短原来从a点到b点的路径。那么，这个中间的k点是1~n中的哪个点呢？甚至有时候不是只通过一个点，而是经过两个或更多的点才能使路径变的更短，即a→k1→k2→b，或a→k1→k2→…→b。比如，对于图3.1中，从编号4到编号3的城市的路径e[4][3]=12。如果通过编号1的城市进行中转，即4→1→3，则e[4][1]+e[1][3]=5+6=11，路径将由12缩短为11。其实，编号1到编号3的城市也可以通过编号2的城市中转，即1→2→3，则e[1][2]+e[2][3]=5，路径由6缩短为5。所以，从编号4到编号3的城市，经过编号1和编号2的城市进行中转，即4→1→2→3，路径为e[4][1]+e[1][2]+e[2][3]=5+2+3=10，路径进一步缩短为10。

当任意两点之间不允许经过第三点时，城市之间的最短路径就是式3.1给出的初始路径。

假设现在只允许经过1号点，求任意两点之间的最短路径，应该如何处理？只要判断e[i][1]+e[1][j]是否小于e[i][j]即可，其中e[i][1]表示从编号i的城市到编号1的城市的路径长度，e[1][j]表示从编号1的城市到编号j的城市的路径长度。其中，i是1~n的循环，j也是1~n的循环，用C代码描述，如代码清单3-1所示。

代码清单3-1 从i城市经过1城市到j城市的最短路径

for (i=1; i<=n; i++)

{

for (j=1; j<=n; j++)

{

if (e[i][j] >(e[i][1]+e[1][j]))

e[i][j]=e[i][1] + e[1][j];

}

}

执行该代码后，经过编号1城市的情况下，任意两点之间的最短路径更新为：

= （式3.2）

因此，通过（式3.2）可知，在经过编号1城市的情况下，编号3的城市到编号2的城市e[3][2]，编号4的城市到编号2的城市e[4][2]，以及编号4的城市到编号3的城市e[4][3]的路径都变短了，式3.2中带有阴影的数字表示变短的路径。

接下来，继续尝试求解在经过编号1城市和编号2城市的情况下任意两点之间的最短路径。其实很简单，只需要判断e[i][2]+e[2][j]是否比e[i][j]的路径更短。

用C代码描述，如代码清单3-2所示。

代码清单3-2 从i城市经过1城市和2城市到j城市的最短路径

/\* 下面的代码经过编号1的城市，从i城市到j城市的最短路径

for (i=1; i<=n; i++)

{

for (j=1; j<=n; j++)

{

if (e[i][j] >(e[i][1]+e[1][j]))

e[i][j]=e[i][1] + e[1][j];

}

}

/\* 下面的代码经过编号2的城市，从i城市到j城市的最短路径

for (i=1; i<=n; i++)

{

for (j=1; j<=n; j++)

{

if (e[i][j] >(e[i][2]+e[2][j]))

e[i][j]=e[i][2] + e[2][j];

}

}

执行该代码后，经过编号1和编号2城市的情况下，任意两点之间的最短路径更新为：

= （式3.3）

因此，通过（式3.3）可知，在增加经过编号2城市的情况下，编号1的城市到编号3的城市e[1][3]，编号4的城市到编号3的城市e[4][3]的路径都变短了，式3.3中带有阴影的数字表示变短的路径。

同理，继续在只允许经过1、2和3号顶点进行中转的情况下，求任意两点之间的最短路径，更新为

= （式3.4）

式3.4中带有阴影的数字表示变短的路径。

最后，允许通过所有顶点作为中转，任意两点之间最终的最短路径表示为：

= （式3.5）

式3.5中带有阴影的数字表示变短的路径。

总结上面的过程，可以用下面的C代码求从i城市到j城市之间的最短路径，如代码清单3-3所示。

代码3-3 从i城市到j城市之间最短路径的C语言代码

for (k=1; k<=n; k++)

for (i=1; i<=n; i++)

for (j=1; j<=n; j++)

if (e[i][j]>e[i][k]+e[k][j])

e[i][j]=e[i][k]+e[k][j];

需要注意，Floyd-Warshall算法不能解决带有“负权值回路”（或者叫“负权值环”）的图，因为带有“负权值回路”的图没有最短路径。

进一步，将Floyd-Warshall算法描述为：

（1）定义P(i,j,k)表示从v\_i到v\_j，由序号不大于k的顶点为中间点（或直达）可构成的最短路径。

（2）初始化从v\_i到v\_j的目前已知较短路径为从v\_i到v\_j的直达弧（没有经过中间点）；

（3）对每两顶点对(v\_i, v\_j)依次计算P(i,j,k)，k=0…n-1，计算规则为：

P(i,j,k) = min(P(i,k,k-1) + P(k,j,k-1),P(i,j,k-1))

因此，最终可以通过类似代码清单3-3给出的三重for循环完成基于弗洛伊德算法的两点间最短路径的求取。

## C语言实现

本节将使用C语言实现弗洛伊德算法。在使用C语言实现该算法前，构造一个由顶点和边构成的有向图，如图3.2所示。图中，顶点个数为5。

1

1

1

1

1

1

1

1

1

1

1

1

图3.2 五个顶点之间的连接关系

用邻接矩阵表示图3.2中顶点之间的连接关系，即

= （式3.6）

注：矩阵中顶点之间无连接关系，则用∞表示。

在使用C语言实现式3.6的邻接矩阵关系时，将无连接关系用INF表示，且将其定义为一个较大的整数。

使用C语言实现弗洛伊德算法的主要步骤包括：

（1）启动LoongIDE软件工具。

（2）在LoongIDE主界面主菜单中，选择New->New Project Wizard...。

（3）弹出New Project Wizard-C Project对话框界面。在该界面中，按如下设置参数：

Project Type：C Executable（通过复选框设置）。

Project Name：example\_3\_1（通过文本框输入）。

注：工程文件夹设置为：E:\loongson1B\_training\_example\example\_3\_1。

（4）弹出New Project Wizard- MCU, Toolchain & RTOS对话框界面。在该界面中，按如下设置参数：

Mcu Modal：LS1B200（LS232）；

Tool Chain：SDE Lite 4.9.2 for MIPS；

Using RTOS：None（bare programming）；

（5）单击Next按钮。

（6）弹出New Project Wizard-Bare Program Components对话框界面。在该对话框界面中，不勾选任何程序组件（Program Components）前面的复选框。

（7）单击Next按钮。

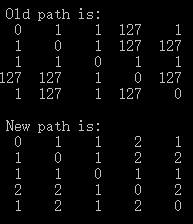
（8）弹出New Project Wizard-New project summary对话框界面。在该对话框界面中，单击OK按钮，完成工程的建立。

（9）在LoongIDE主界面左侧的Project Explorer窗口中，找到并双击main.c文件。

（10）自动打开该文件，删除该文件中main()函数，然后在该文件中添加C语言代码，如代码清单3-4所示。

|  |
| --- |
| #include<stdio.h>  #define INF 127  #define num 5  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* 核心思想：通过中间点迭代出最短路径。  \* 【注意】：遍历每个中间点都会更改原邻接矩阵，  \* 所以不用考虑2个点，3个点……  \* 已经通过迭代的方式处理完成。  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  /\* Floyed算法 \*/  void Floyd(unsigned char e[5][5], unsigned char n) //函数Floyd实现弗洛伊德算法  {  int i,j,k;  for (k=0; k<=n-1; k++) // 最外层for循环(遍历允许通过的点)  for (i=0; i<=n-1; i++) // 中间层for循环(遍历源节点)  for (j=0; j<=n-1; j++) // 最内层for循环(遍历目标节点)  if (e[i][j]>e[i][k]+e[k][j]) // 原始路径和新路径进行比较(进行更新迭代)  e[i][j]=e[i][k]+e[k][j];  }  /\* 输出邻接矩阵 \*/  void output\_graph(unsigned char arr[][num]) // 二维数参数传递  {  int i,j;  for(i=0;i<num;i++)  {  for(j=0;j<num;j++)  printf("%3d ", arr[i][j]);  printf("\n");  }  }  //-----------------------------------------------------------------------  // 主程序  //-----------------------------------------------------------------------  int main()  {  /\* 初始化图的邻接矩阵arr[num][num] \*/  unsigned char arr[num][num]={  {0, 1, 1, INF, 1},  {1, 0, 1, INF, INF},  {1, 1, 0, 1, 1},  {INF, INF, 1, 0, INF},  {1, INF, 1, INF, 0}};  printf("\n Old path is:\n");  output\_graph(arr); // 输出初始邻接矩阵    Floyd(arr, num); // 实现弗洛伊德算法    printf("\n New path is:\n");  output\_graph(arr); // 输出更新之后的邻接矩阵  return 0;  } |

程序运行效果图如下：



## 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。

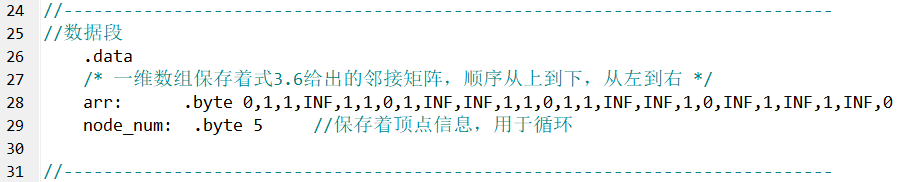
添加bsp\_start.S源文件。

在bsp\_start.S下添加如下代码：

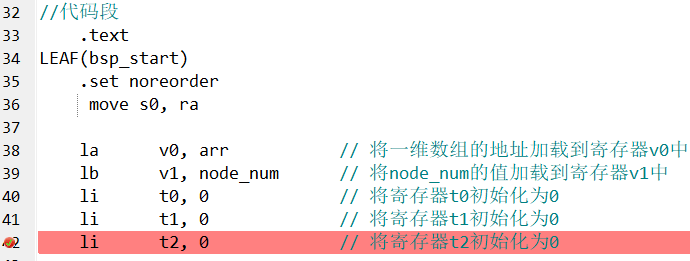
|  |
| --- |
| /\*  \* bsp\_start.S  \*  \* created: 2022/2/21  \* author: Li TianLing  \*/  /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \*  \* v0 邻接矩阵arr的基地址 （常量）  \* v1 参数node\_num （常量）  \*  \* t0 哨兵k（遍历允许通过的点）  \* t1 哨兵i（遍历源节点）  \* t2 哨兵j（遍历目标节点）  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #include "regdef.h"  #include "cpu.h"  #include "asm.h"  #define INF 127  //-----------------------------------------------------------------------------  //数据段  .data  /\* 一维数组保存着式3.6给出的邻接矩阵，顺序从上到下，从左到右 \*/  arr: .byte 0,1,1,INF,1,1,0,1,INF,INF,1,1,0,1,1,INF,INF,1,0,INF,1,INF,1,INF,0  node\_num: .byte 5 //保存着顶点信息，用于循环  //-----------------------------------------------------------------------------  //代码段  .text  LEAF(bsp\_start)  .set noreorder  move s0, ra  la v0, arr // 将一维数组的地址加载到寄存器v0中  lb v1, node\_num // 将node\_num的值加载到寄存器v1中  li t0, 0 // 将寄存器t0初始化为0  li t1, 0 // 将寄存器t1初始化为0  li t2, 0 // 将寄存器t2初始化为0  /\* 下面的代码执行弗洛伊德算法 \*/  /\* 三重循环的索引从1开始 \*/  1: // 对应于最外层的循环  addi t0, t0, 1 // t0对应的k，(t0)+1→(t0)  2: // 对应于中间层的循环  addi t1, t1, 1 // t1对应的i，(t1)+1→(t1)  3: // 对应于最里层的循环  addi t2, t2, 1 // t2对应的j，(t2)+1→(t2)  /\* 循环索引从1开始，数组arr的索引从0开始，循环索引减1匹配数组索引 \*/  addiu t7, t1, -1 // (t1)-1→(t7)  mul t3, t7, v1 // t3 = t7(i)×v1(node\_num)  addiu t7, t2, -1 // (t2)-1→(t7)  addu t4, t3, t7 // t4 = t3 + j(t2)，计算[i][j]对应元素的偏移量  addu t5, t4, v0 // t5 = base(arr)+(t4)，计算[i][j]对应元素存储地址  lb a0, 0(t5) // 将初始路径的值加载到寄存器a0中，arr[i][j]  addiu t7, t0, -1 // (t0)-1→(t7)  addu t4, t3, t7 // t4 = t3 + k(t7)，计算[i][k]对应元素的偏移量  addu t5, t4, v0 // t5 = base(arr)+t4，计算[i][k]对应元素存储地址  lb a1, 0(t5) // 将arr[i][k]路径的值加载到寄存器a1  addiu t7, t0, -1 // (t0)-1→(t7)  mul t3, t7, v1 // t3 = t7(k)×node\_num (v1)  addiu t7, t2, -1 // (t2)-1→(t7)  addu t4, t3, t7 // t4 = t3 + j(t7)，计算[k][j]对应元素的偏移量  addu t5, t4, v0 // t5 = base(arr)+t4，计算[k][j]对应元素的存储地址  lb a2, 0(t5) // 将arr[k][j]路径的值加载到寄存器a2  addu a3, a1, a2 // (a1)+(a2)→(a3)，arr[i][k]+arr[k][j]→(a3)  sltu s7, a3, a0 // 判断(a3)<(a0)?，即[i][k]+[k][j]<[i][j]？  beqz s7, 4f // 不成立，跳转到标号为4的位置  nop // 延迟隙  addiu t7, t1, -1 // (t1)-1→(t7), (t1)=i  mul t3, t7, v1 // t3 = t7(i)×v1(node\_num)  addiu t7, t2, -1 // (t2)-1→(t7), (t2)=j  addu t4, t3, t7 // t4 = t3 + j(t7)，计算[i][j]对应元素的偏移量  addu t5, t4, v0 // t5 = base(arr)+ t4，计算[i][j]对应元素存储地址  sb a3, 0(t5) // arr[i][k]+arr[k][j]保存到arr[i][j]对应存储空间位置  nop // 延迟隙，插入空操作  4:  /\* 下面的代码判断最内侧循环是否结束 \*/  bne t2,v1, 3b // (t2)=(v1)?，不相等则跳转到标号3的位置  nop // 延迟隙，插入空操作  /\* 下面的代码判断中间循环是否结束 \*/  li t2, 0 // 寄存器t2初始化为0  bne t1,v1, 2b // (t1)=(v1)?，不相等则跳转到标号2的位置  nop // 延迟隙，插入空操作  /\* 下面的代码判断哦最外层循环是否结束 \*/  li t1, 0 // 寄存器t1初始化为0  bne t0,v1, 1b // (t0)=(v1)?，不相等则跳转到标号1的位置  nop // 延迟隙，插入空操作  move ra, s0 // 恢复返回地址  j ra // 跳转到寄存器ra所保存的目标地址  nop // 延迟隙，插入空操作  .set reorder  ENDFRAME(bsp\_start) |

## 调试运行

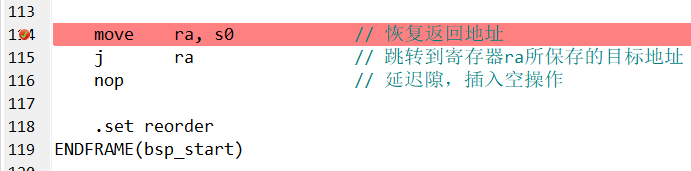
初始数据如下所示：



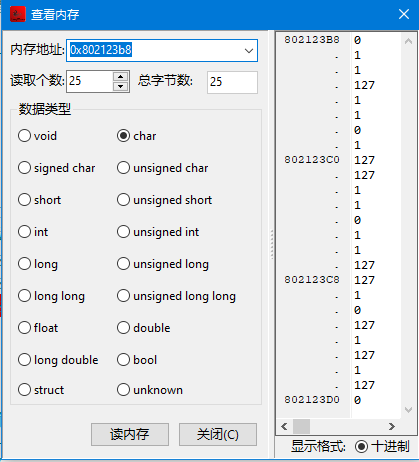
在初始化结束的位置打上断点：



如下图所示，在程序运行结束打上断点：



联合调试后，双击v0寄存器，即可查看初始的邻接矩阵：



再运行至第二个断点，双击v0寄存器，查看Floyed处理后的邻接矩阵：

