谈一谈MIPS汇编Challenge题：找哈密尔顿回路

> 阅读本文时，我们假定读者对MIPS汇编指令、伪指令有一定了解，知道内存的寻址方式，并且能够编写50行以内的汇编程序

题目大意

输入一个具有n个顶点的无向图G，判断G是否有哈密尔顿回路。

所谓**哈密尔顿回路**指的就是：由指定的起点前往指定的终点，途中经过所有其他节点且只经过一次的路径。在图论中含有哈密顿回路的图中闭合的哈密顿路径称作哈密顿回路

这里要求使用汇编语言解决

条件中有$0 < n < 8$，$0 < m < 100$，~~然后我就想，如果一个图只有八个顶点，就算全联通哪里用的到100条边呢...~~

但是顶点数很小，很自然的我们不会去用什么邻接表、链式前向星等高级的存图方法，最简单粗暴的**邻接矩阵**就够了

很容易就能写出C++/C语言的代码，~~不会吧，连这都写不出来，那就~~`\remake`~~去吧~~

```
#include <iostream>
using namespace std;

const int MAXN = 10;

int G[MAXN][MAXN];
bool vis[MAXN];
int n, m;
int ans;

int dfs(int i, int start) {
    bool flag = false;
    vis[i] = true;
    cout << i;
    for(int j = 1; j <= n; j++) if(!vis[j] && G[i][j]) {
        dfs(j, start);
    }
    for(int j = 1; j <= n; j++) if(!vis[j]) {
        flag = true;
        break;
    }
    if(G[i][start] && !flag) ans = 1;
    vis[i] = false;
    return 0;
}

int main() {
    cin >> n >> m;
    for(int i = 0; i < m; i++) {
        int u, v;
        scanf("%d%d", &u, &v);
```

```
        G[u][v] = G[v][u] = 1;
    }

    for(int i = 1; i <= n; i++) {
        dfs(i, i);
        if(ans == 1) {
            cout << 1 << endl;
            return 0;
        }
    }
    cout << 0 << endl;
    return 0;
}
```

我们来分析一下C++/C的思路，首先主体肯定是用深度优先遍历去，其实是一个很容易的dfs，由于欧拉路径需要经过所有点，所以从任意一个点开始遍历即可，当发现当前点无路可走时，就判断是否已经遍历完所有点，当前点是否与起点相连，如果满足上面两个条件，那么我们就发现了一条哈密顿回路，这里我并没有考虑所谓的自环等特判情况，但是一样可以AC

如何从C/C++翻译到MIPS汇编

这里我想说一下编程范式，对于C/C++的特定组成模块，都是有对应的可以直接翻译的编程范式直接使用

使用的时候，**汇编语言也可以使用合理的缩进来增加可读性**

`if/else`语句

```
if(a >= b) //Do something...
else if(a < c+b) //Do something...
else //Do something...
```

可以被翻译为

```
# $s0 = a, $s1 = b, $s2 = c
if_begin:
bgt $s1, $s0, if_else1
    # Do something
if_else1:
add $t0, $s1, $s2
bngt $t0, $s0, if_else2
    # Do something
if_else2:
    # Do something
if_end:
```

`switch()`语句

```
switch(a) {
    case 2:
        // Do something
        break;
    case 4:
        // Do something
        break;
    default:
        // Do something
}
```

可以被翻译为

```
# $s0 = a
switch_begin:
    case_2:
    bne $s0, 2, case_4
        # Do something
    j switch_end

    case_4:
    bne $s0, 4, default
        # Do something
    j switch_end

    default:
        # Do something
switch_end:
```

`for`循环语句

```
for(int i = 0; i < n; i++) // Do something
```

可以被翻译为

```
# $s0 = n
li $t0, 0
for_begin:
bne $t0, $s0, end_for
    # Do something
addi $t0, $t0, 1
j for_begin
end_for:
```

> ### 关于循环指令中的大小比较
>
> 在循环条件判断时，我们时常会遇到大小比较问题，这时需要我们了解常用的大小比较跳转指令
>
> 通常我们遇到的都是有符号整数，所以我们不必使用无符号判断指令，后面我会专门写一下有符号和无符号操作的比较
>
> 以下我们约定 `$t0 = a, $t1 = b`

> $a < b$
>
> 转换成$b \leq a$判断退出循环
>
> ```
> blt $t1, $t0, end_loop
> beq $t1, $t0, end_loop
> ```
>
> 当然，如果a是单调增加的，那么也可以转换成$a \neq b$的情况判断（这样使用时请保证a的初始值不大于b，否则会出大问题）
>
> $a \leq b$
>
> 转换成$b < a$判断退出循环
>
> ```
> blt $t1, $t0, end_loop
> ```
>
> $a \neq b$
>
> 转换成判断$a = b$判断退出循环
>
> ```
> bne $t0, $t1, end_loop
> ```

`while`循环语句

```
int i = a;
while(i < n) {
    // Do something...
    i++;
}
```

其实这玩意好像和`for`循环是一样的

```
# $s0 = n, $s1 = a, $t0 = i
move $t0, $s1
while_begin:
bne $t0, $s0, end_while
    # Do something
    addi $t0, $t0, 1
j while_begin
end_while:
```

`do...while`循环语句

```
do {
    i++;
    //Do something
} while(i < n);
```

这个应该这样写

```
# $t0 = i, $s0 = n
li $t0, 1
dowhile:
    addi $t0, $t0, 1
    # Do something
beq $s0, $t0, end_dowhile
j dowhile
end_dowhile:
```

一维数组的定义与使用

在C/C++中非常简单的几行代码

```
int arr[100];
for(int i = 0; i < 100; i++) arr[i] = i;
```

在汇编语言中挺麻烦的

```
.data
    arr: .space 400             #长度100的int型数组，总共使用400字节
.text
    # $t0 = i
    li $t0, 0;
    for_begin:
    beq $t0, 100, end_for
        move $t1, $t0
        sll $t1, $t1, 2         # i*4得到偏移的字节数，MIPS按照字节寻址，地址从
0x00000000, 0x00000004...以此类推
                                # 此外MIPS还是小端地址，如果输入0x12345678，那么
0x00000002存的是0x56
        sw $t0, arr($t1)        # 这跟直接访问还挺像的
                                # 实际上，arr是指一系列空间的首地址，加上偏移量$t1，得到
arr[i]的地址
    addiu $t0, $t0, 1
    j for_begin
    end_for:

li $v0, 10                      # 类似于C/C++中的return 0
syscall
```

二维数组的定义与使用

在汇编语言中更加麻烦，比Python还麻烦，~~Python的三维数组已经够麻烦的了~~
C/C++版本（这里我们来写一个完整的汇编程序）

```cpp
#include <iostream>
using namespace std;
int arr[64][64];

int main() {
    int m, n;
    cin >> n >> m;
    for(int i = 0; i < n; i++)
        for(int j = 0; j < m; j++) cin >> arr[i][j];
    for(int i = n-1; i >= 0; i--)
        for(int j = m-1; j >= 0; j--) cout << i << ' ' << j << ' ' << arr[i][j]
<< endl;
    return 0;
}
```

很简单是不是，~~我们来看一看汇编是多么的笨拙~~

```asm
.data
    arr: .space 16384

# 下面两个宏定义与数组大小密切相关，64*64大小的数组是这么做的
# 我们约定$t7, $t8, $t9只在宏定义中使用
.macro setarr(%d, %i, %j)       # 把arr[i][j]设置为d
    sll $t8, $t8, 6
    add $t9, $t8, %j
    sll $t9, $t9, 2
    sw %d, arr($t9)
.end_macro
.macro getarr(%d, %i, %j)       # 把d赋值为arr[i][j]
    sll $t8, $t8, 6
    add $t9, $t8, %j
    sll $t9, $t9, 2
    lw %d, arr($t9)
.end_macro

.text
    # $s0 = n, $s1 = m
    li $v0, 1
    syscall
    move $s0, $v0
    li $v0, 1
    syscall
    move $s0, $v0

    # $t0 = i, $t1 = j
    li $t0, 0
    li $t1, 0
    for_in_i:
    beq $t0, $s0, end_for_in_i
        for_in_j:
        beq $t1, $s1, end_for_in_j
            li $v0, 1
            syscall
            setarr($v0, $t0, $t1)
        addi $t1, $t1, 1
        j for_in_j
```

```
        end_for_in_j:
    addi $t0, $t0, 1
    j for_in_i
    end_for_in_i:

    # $t0 = i, $t1 = j
    subi $t0, $s0, 1
    subi $t1, $s1, 1
    for_out_i:
    blt $t0, 0, end_for_out_i
        for_out_j:
        blt $t1, 0, end_for_out_j
            getarr($t3, $t0, $t1)
            # 输出$t0, $t1, $t2这里省略了
        subi $t1, $t1, 1
        j for_out_j
        end_for_out_j:
    subi $t0, $t0, 1
    j for_out_i
    end_for_out_j:

li $v0, 10                          # 类似于C/C++中的return 0
syscall
```

这道题实际上是`MIPS`练习题，原题如下

> ## 矩阵转化
>
> 题目编号 *712-3*
>
> **实现满足下面功能的汇编程序：**
>
> 输入一个n乘m的稀疏矩阵$\mathbf{A}$（矩阵每个元素为占一个字的整数），将$\mathbf{A}$转化为三元组列表(该列表的排列顺序为：行号小的在前，如果行号相同则列号小的在前)，并将三元组列表逆序输出。

函数声明

首先有必要来复习一下关于寄存器的约定

序号	名称	MIPS汇编约定的用途	分类	保护的前提条件	何时保护与恢复
16~23	$s0~$s7	程序员变量	强保护	如果需要使用	进入函数时保存 退出函数前恢复
31	$ra	函数返回地址		如果调用子函数	
2~3	$v0~$v1	函数返回值	弱保护	如果要求其值在调用子函数前后必须保持一致	调用子函数前保存 子函数返回后恢复
4~7	$a0~$a3	函数参数			
8~15, 24~25	$t0~$t9	临时变量			

此外还有`$sp`寄存器，用来存放栈指针，`$fp`寄存器用来存放帧指针

强保护（`$s`寄存器，`$ra`）

我们处于一个函数中，对于调用的多个子函数都要可能使用的共同变量,相比临时变量，他们有着更长的生命周期， 我们保存在$s0$ s7，他们的生命周期 = 函数的生命周期

- **我们对`$s`寄存器们采取的操作是**

1. 进入每个函数时， 这个函数用哪些个`$s`寄存器， 就保存哪个`$s`
2. 在函数结束时， 恢复`$s`到最开始的值

这样可以保证，函数的子函数们不会改变这些s寄存器的内容，（即便子函数使用了他们，改变了他们，但最后还是复原了他们），保证了他们的生命周期

- **对于`$ra`，我们对他采取的操作是**

1. 通过jal进入一个函数时， 这个`$ra`（指向该函数返回时跳回的指令地址呢）得保存好了
2. 中间可以再次调用自己或其他函数，或者做一些操作， 使得`$ra`发生变化，但根本不用怕
3. 在函数结束时，要`jr $ra`，因此要先恢复`$ra`到最开始的值

对于叶子函数(leaf function)，由于其不调用其他函数，因此`$ra`可以不保存，~~但是避免出错，最好习惯保存，平时做题时栈空间因该不会不够用~~

对于强保护， 我们可以看到一些共同点：**即刚进入函数时保存；退出函数前恢复**

一句话总结，进入函数后，压栈保存所有必要的`$s`和`$ra`

```
int calc(int a, int b) {
    int t = a + b/a + a*b;
    return t;
}
```

写成汇编程序

```
calc:   # $a0 = a, $a1 = b, $s0 = t, $s1 = b/a, $s2 = a*b 其实可以用$t，但是这里为了展示压栈
addiu $sp, $sp, -16     # 四个需要保存的变量，$sp增长 4*4 = 16
sw $s0, 0($sp)
sw $s1, 4($sp)
sw $s2, 8($sp)
sw $ra, 12($sp)

    mult $a0, $a1
    mflo $s2
    div $a1, $a0
    mflo $s1
    add $v0, $s0, $s1
    add $v0, $v0, $s2

lw $s0, 0($sp)
lw $s1, 4($sp)
lw $s2, 8($sp)
lw $ra, 12($sp)
addiu $sp, $sp, 16
jr $ra
```

调用时使用

```
jal calc
```

弱保护（`$t`寄存器，`$v`，`$a`）

仅仅保存那些生命周期长的寄存器是不够的， 在函数一次次调用中， 我们很可能还要记录下当前一些临时值的状态， 以便在调用子函数（子函数很可能更改了`$t`）后`jr $ra`完了，能再恢复他们，

比如循环变量`i`，无论是存放在`$t`， 还是存放在`$a`中， 都需要对i做保护， 要不然返回后怎么继续我们的循环？

一句话总结，进入函数之前，压栈保存所有必要的`$t`，`$a`，`$v`

汇编语言示例

```
    # 假如我们需要调用calc之前保护$a0, $a1, $t0, $t1, $v0
    addiu $sp, $sp, -20
    sw $a0, 0($sp)
    sw $a1, 4($sp)
    sw $t0, 8($sp)
    sw $t1, 12($sp)
    sw $v0, 16($sp)

    jal calc

    lw $a0, 0($sp)
    lw $a1, 4($sp)
    lw $t0, 8($sp)
    lw $t1, 12($sp)
    lw $v0, 16($sp)
    addiu $sp, $sp, 20
    # 调用过程结束，寄存器基本完全恢复到原来的状态
```

有了上述知识，我们就可以根据上述范式，一点一点把C/C++的程序翻译成MIPS汇编程序了

最后得到的找哈密尔顿回路的MIPS汇编程序是

```
.data
    graph: .space 1024
    vis: .space 512
    endl: .word '\n'

.macro setGraph(%data, %i, %j)
    sll $t8, %i, 4
    add $t8, $t8, %j
    sll $t8, $t8, 2
    sw %data, graph($t8)
.end_macro
.macro getGraph(%ans, %i, %j)
    sll $t8, %i, 4
    add $t8, $t8, %j
    sll $t8, $t8, 2
    lw %ans, graph($t8)
.end_macro

.text
main:
    # $s0 = n
```

```
    li $v0, 5
    syscall
    move $s0, $v0
    # $s1 = m
    li $v0, 5
    syscall
    move $s1, $v0

    move $t0, $zero
    for_input:
    # for(i = 0; i != m; i++)
    beq $t0, $s1, end_for_input
        li $v0, 5
        syscall
        move $t1, $v0
        li $v0, 5
        syscall
        move $t2, $v0
        li $t3, 1
        setGraph($t3, $t1, $t2)
        setGraph($t3, $t2, $t1)
    addi $t0, $t0, 1
    j for_input
    end_for_input:

    li $t0, 1
    for_iter:
    # for(i = 1; i <= n; i++)
    bgt $t0, $s0, end_for_iter
        move $a0, $t0   # $a0 = p
        move $a1, $t0   # $a1 = start
        # 这里是我之前写的程序，没有按照规范做保护，但是在编写dfs函数时避免寄存器使用冲突的情
况，调用的时候没出问题真是万幸
        jal dfs
        beq $s3, 1, print_ans
    addi $t0, $t0, 1
    j for_iter
    end_for_iter:

    print_ans:
    li $v0, 1
    move $a0, $s3
    syscall
li $v0, 10
syscall

dfs:    # $a0 = p, $a1 = start
addi $sp, $sp, -24
sw $ra, 20($sp)
sw $t0, 16($sp)
sw $t1, 12($sp)
sw $t2, 8($sp)
sw $t3, 4($sp)
sw $t4, 0($sp)
    sll $t0, $a0, 2
    li $t1, 1
    sw $t1, vis($t0)
    li $t0, 1
```

```
    for_nextpoint:
    # for(i = 1; i <= n; i++) if(graph[p][i] == 1 && !vis[i])
    bgt $t0, $s0, end_for_nextpoint
        getGraph($t1, $a0, $t0)
        bne $t1, 1, nextpoint_continue
        sll $t2, $t0, 2
        lw $t3, vis($t2)
        bne $t3, $zero, nextpoint_continue
            move $t4, $a0
            move $a0, $t0
            jal dfs
            move $a0, $t4
    nextpoint_continue:
    addi $t0, $t0, 1
    j for_nextpoint
    end_for_nextpoint:

    li $t0, 1
    for_check:
    bgt $t0, $s0, end_for_check
        sll $t1, $t0, 2
        lw $t2, vis($t1)
        beq $t2, $zero, end_function
    addi $t0, $t0, 1
    j for_check
    end_for_check:

    getGraph($t0, $a0, $a1)
    bne $t0, 1, end_function
    li $s3, 1

    end_function:
    sll $t0, $a0, 2
    sw $zero, vis($t0)
lw $ra, 20($sp)
lw $t0, 16($sp)
lw $t1, 12($sp)
lw $t2, 8($sp)
lw $t3, 4($sp)
lw $t4, 0($sp)
addi $sp, $sp, 24
jr $ra
```

关于Mars软件的命令行调试与对拍器编写

```
java -jar mars.jar db mc CompactDataAtZero nc std.asm <in.txt >std_out.txt
```

使用上述代码可以命令行运行`std.asm`并从`in.txt`重定向输入，重定向输出到`std_out.txt`

但是不知道因为什么原因，这一段代码不能在C/C++里面使用

```
void running() {
    system("java -jar mars.jar db mc CompactDataAtZero nc std.asm <in.txt
>std_out.txt");
    system("java -jar mars.jar db mc CompactDataAtZero nc src.asm <in.txt
>src_out.txt");
    system("fc std_out.txt src_out.txt");
}
```

这样一段代码在运行时会出现`Java`抛出的异常

```
Exception in thread "MIPS" java.lang.NullPointerException: Cannot invoke
"String.trim()" because "input" is null
        at mars.util.SystemIO.readInteger(SystemIO.java:102)
        at
mars.mips.instructions.syscalls.SyscallReadInt.simulate(SyscallReadInt.java:57)
        at
mars.mips.instructions.InstructionSet.findAndSimulateSyscall(InstructionSet.java
:3241)
        at
mars.mips.instructions.InstructionSet.access$200(InstructionSet.java:47)
        at
mars.mips.instructions.InstructionSet$63.simulate(InstructionSet.java:1196)
        at mars.simulator.Simulator$SimThread.construct(Simulator.java:346)
        at mars.simulator.SwingWorker$2.run(SwingWorker.java:115)
        at java.base/java.lang.Thread.run(Thread.java:831)
```

貌似时`Java`的`String`类中的`trim()`函数挂了，它好像想要我输入一个整数，但是实际上读入了空字符串`null`，但我也不知道为啥

所以暂时还不知道怎么进行命令行对拍……等待进一步研究吧