程序思路总结-六院八队

1 概述......3
1.1 写在前面的话......3
1.2 目录结构......4
1.3 程序结构......6
1.3.1 coding 分工......6
1.3.2 search_route 函数......6
1.3.3 search_double_route 函数......6
1.3.4 search_single_route 函数......7
1.3.5 LKH 函数......7
1.3.6 findTour 函数......8
1.3.7 其它重要函数......8
1.4 复赛和决赛使用用例......8
1.5 符号描述......9
1.6 调试环境......9
2 求单条路思路......10
2.1 算法描述......10
2.2 规模压缩阶段......11
2.3 解决 atsp 问题......11
2.3.1 ascent 阶段-求候选集......11
2.3.2 LKH report 几个非常重要的理论......12
2.3.3 无向 K-opt......13
2.3.4 关于有向 K-opt 和无向 K-opt......14
2.3.5 关于 Non-sequential K-opt 和 Sequential K-opt......18
2.3.6 LKH 短板：initial tour......18
2.3.7 指派算法 KM 粗略克服 LKH 短板......19
2.3.8 小结......19
2.4 “升级”操作的两种方案......19
2.5 规模压缩阶段-续......20
2.6 规模压缩-优点与缺点......20
3 迭代求两条路思路......21
3.1 算法描述......21
3.2 “增量式惩罚”方案......21
3.3 “小无穷惩罚”方案......22
3.4 启发式规则“必经点集合互不侵占”......22

3.5 启发式规则“规模压缩阶段不走老路”......22
3.5 使用 hash 保存结果避免重算......23
3.6 两点间有多条边的处理......23
3.6.1 读入阶段......24
3.6.2 两条路迭代阶段......24
3.6.3 关于 3.6.2 的进一步思考......24
参 考 文 献......25

1 概述

1.1 写在前面的话

首先，感谢比赛的工作人员们，比赛的成功举办离不开你们在比赛背后的努力与付出，为我们选手保驾护航。这次深圳之行也见到了出题人，出 case 的博士，回答问题的版主大人。作为选手，见到大牛们和 HR 在为我们不辞辛苦地奔波真的感到非常感动。让我们再看他们英俊的身影看 ，快戳我。

我们作为参加比赛的老司机，本科时候也是参加过各种比赛，拿过各种奖，在我们看来，比赛有三点好处：

1、收获一群志同道合的小伙伴。

2、获得知识、技能，锻炼自己。人的潜力在比赛中、在竞争中才会更加凸显。所谓 deadline 是第一生产力。

3、拿奖、拿钱、拿 offer。

至于公平性、开源代码、跨赛区等撕逼话题，请转 http：//www.znczz.com/thread-105261-1-1.html，其中的“传承”可以替换为“开源代码”，意思差不多。这是本人（王）本科做了三年的某比赛的论坛某贴，一直珍藏。虽然比赛不一样，可是里面写的话我感觉不仅适用于比赛，也适用于工作生活，拿来与大家分享。附上当年参加比赛的身影：http://v.youku.com/v_show/id_XNzQ3MDUzODQ0.html，百度云链接：http://pan.baidu.com/s/1o7SK8yQ 密码：ihbn。 这个比赛是偏硬件、控制，为期更长，需要付出更多，会有更大的不确定性，由于学校传承、裁判的偏心或失误会造成更大的不公平性。

回到华为比赛，我感觉才举办第二届就办成这么有模有样已经很成功了。现实中没有十全十美的事情，组委会既然制定了规则，只有没有 bug，大家在同一规则下竞技就好了，失败了的原因只有一个：技不如人。少找客观原因，多找找代码的 bug 和新的思路才是正道。经常看到论坛的楼盖着盖着就歪了，也是醉了。。。

写着写着又写多了，总之一句话：少些撕逼多些技术交流，付出多少和获得多少成正相关。大家通常看到的只是结果，但看不到比赛背后选手没日没夜的付出。从赛题公布到比赛结束，我们队三个人休息的日子一只手可以数的过来，一起往返于寝室、食堂、实验室，感谢两位妹子队友，一起走过比赛的日子。Ps，不要看低妹子哦，写代码也是不比男生差的。

题外话 1：我们认为 8 强中的 Freedom 队应该是前三的水准，谁知被我们队坑死在 8 强中，尤其是官方 case4 求解出权值在 2000 内，佩服佩服。可惜啦，orz...

题外话 2：武长赛区三只队伍分到一个小组，如果没分到一个小组的话很可能武长拿到 4 个 16 强，可惜啦，orz...

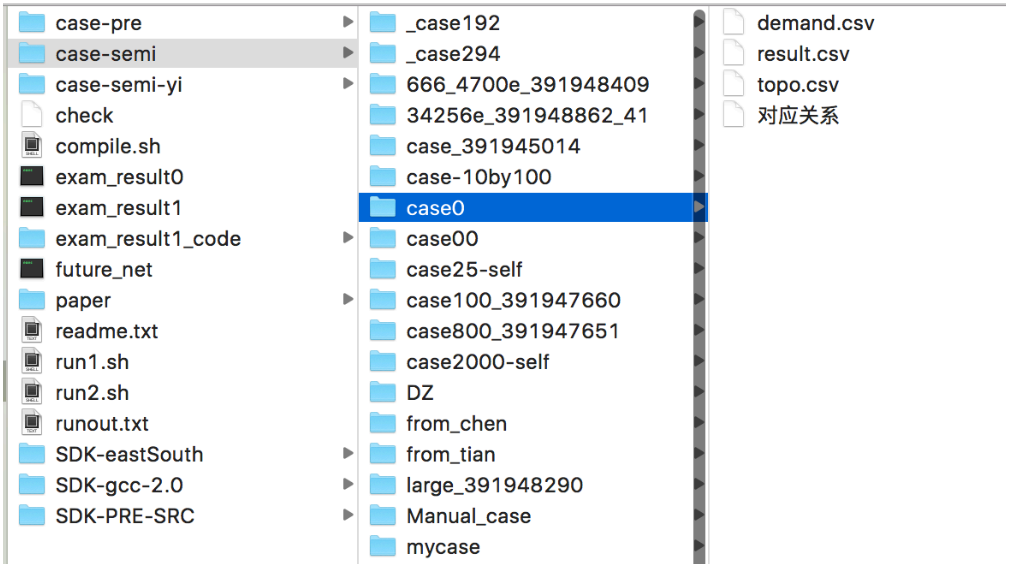
顺带推荐一个 download 论文的利器，传送门：http://www.sci-hub.cc/。具体是什么大家自行百度，一般人我不告诉他。

关于本项目的所有编码上的、算法上的、case 上的问题，欢迎邮件或 QQ 等方式同我们交流。QQ：452570607(易) 912385457(王) 799279601(韩)。

1.2 目录结构

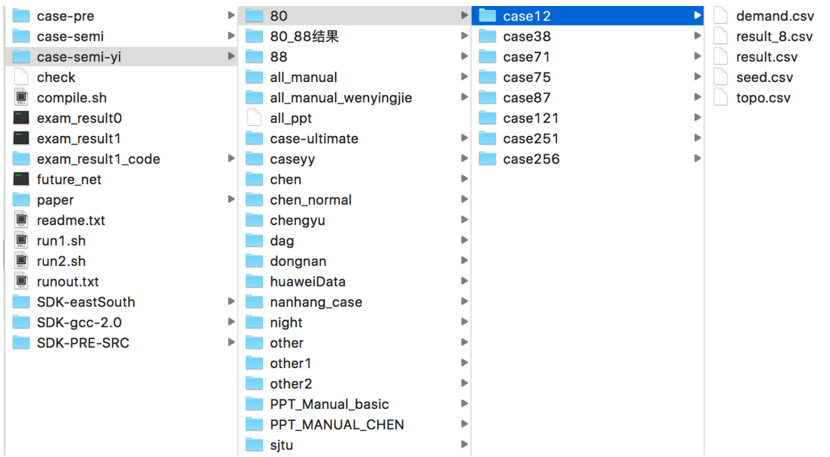
run1.sh： 编译、执行、检测三合一脚本（测试零散的 case，方便精细测试），对应的 case 位于 case-semi 文件夹下，如果想增加例子可以依照脚本写的那样增加 N[${#N[@]}]="yourcase"，在 case-semi 下增加 yourcase 文件夹，放入 topo.csv 和 demand.csv，目录结构如下图。并且 run1.txt 中有很多 case 的最优解记录，供大家参考。

case-semi：复（决）赛 case，里面的每个子文件夹就是一个 case，所有 case 起名必须是 topo.csv 和 demand.csv。类似 case800_391947651 或 case100_391947660 这样的 case，后面的那个数字是论坛网址，例如第一个是 http://bbs.csdn.net/topics/391947651。



run2.sh：编译、执行、检测三合一脚本（测试整个文件夹 case，方便整个文件夹例子的测试，其实主要用于决赛前和很多队伍交流 case 的测试），对应的 case 位于 case-semi 文件夹的子文件夹下，如果想增加例子可以依照脚本写的那样增加 NAME[0]="case-semi-yi/yourcaseFolder/"，在 case-semi-yi 下增加 yourcaseFolder 文件夹，将含有 topo.csv 和 demand.csv 的多个文件夹放进去，如下图所示。

case-semi-yi：复（决）赛 case，里面的每个子文件夹包含 n 个 case，所有 case 起名必须是 topo.csv 和 demand.csv。



compile.sh：用于编译，输入参入为 0、1、2，默认为 1。代表的含义是 0：开源代码 SDK-PRE-SRC/，1：自己的代码 2：南京苏州赛区“司机小胖”队代码。

runout.txt：是每次运行时候程序 stdout（即 printf）的输出结果（执行 future_net 时做了输出重定向）

future_net：程序的可执行文件

check：来自 dz 队的 result.csv 检查程序

exam_result0：自己的 result.csv 检查程序的可执行文件，用于初赛检测单条路的合法性

exam_result1：自己的 result.csv 检查程序的可执行文件，用于复赛和决赛检测两条路的合法性，无 argv 参数则输出简略结果，随便增加一个 argv 参数就可以输出详细结果

exam_result1_code/：自己的 result.csv 检查程序源码

case-pre：初赛 case，里面的每个子文件夹就是一个 case，所有 case 起名必须是 topo.csv 和 demand.csv

SDK-PRE-SRC/：使用 LKH 开源代码实现的复（决）赛程序

SDK-gcc-2.0/：自己完全重写的 LKH 复（决）赛程序（和原版 lkh 最大区别在于 K-opt 是有向的）

SDK-eastSouth/：南京苏州赛区“司机小胖”队代码，同样用的 LKH 算法。在求一些 case 时结果非常棒，但是代码不是很健壮，某些例子会 segmentation fault。个人感觉程序再做的健壮和完善一些，是很有实力进四强的，惋惜~

paper/：几篇很有用的文献

paper/opt-pictures/：有向 K-opt 的图解

1.3 程序结构

声明：由于源码过多，以下每个图片中的代码仅仅是源代码中的一部分，部分代码用文字替代,可以看成是个框架，这样做只是方便大家阅读和顺下思路，具体细节请看源文件。

1.3.1 coding 分工

整个工程会出现两种风格的代码。如遇到相应代码读不懂的情况请对号找人。

易:MinimunSpanningTree.cpp、Minimun1Tree.cpp 、KM.cpp、candidate.cpp、Heap.cpp，手造特例说明.pptx。

王、韩：其它 cpp 文件和 shell 脚本。

1.3.2 search_route 函数

```
void search_route(char *topo[MAX_EDGE_NUM], int edge_num, char *demand[MAX_DEMAND_NUM], int demand_num)
{
    G.initial(topo,edge_num);
    Road0.initial(demand[0], &G, 0);
    Road1.initial(demand[1], &G, 1);
    Road::search_double_route();
}
```

1.3.3 search_double_route 函数

```
void Road::search_double_route()
{
    setPunishMethod();//设置惩罚方式和迭代轮数
    for(routeCn = 0; routeCn < routeCnMax; routeCn++)
    {
        初始化两个road;
        Rp和Rpo赋值;
        for(iteration = 0; iteration < iter[routeCn] ;iteration++)
        {
            //寻找单条路
            Rp->search_single_route(Rpo);
            if(寻找单条路失败)
                break;

            //求出两条路重复边的数量和id号
            setReIdList(&Road0,&Road1);

            //如果找到更好的解就重写结果字符串
            if(iteration>=1 && findBetterResult)
                reWriteResult(Road0.VtourCn,Road0.VtourId,Road1.VtourCn,Road1.VtourId);

            //满足(超时、无重边、和上次迭代路权值一样、和上上次迭代路权值一样)则输出、跳出,否则继续迭代
            if(breakAvailable)
                break;
            SWAP(Rp,Rpo);
        }
        if(超时) break;
    }
}
```

1.3.4 search_single_route 函数

```
void Road::search_single_route(Road *Rpo)
{
    给当前要求的这个road设置惩罚;
    if(之前带形同的惩罚求过这条路)
    {
        从hash列表里面取出结果;
        return ;
    }

    初始化initial isMust、num、list、roadCost、solveTspCn、isUseCopy、cost;

    //转化为tsp问题,每次都是新的tsp
    while(1)
    {
        solveTspCn++;          //tsp求解次数+1
        setVcost_SPFA();       //find shortest path and record path in V
        setCost();             //Vcost到cost值的转换
        roadCost[0] = LKH();
        if(road有解)       setTour();

        if(road无解)
        {
            if(solveTspCn == 1) break;          //case A:第一次就没求出路径直接退出
            if(不使用double重复non-must点方案) //case B:上次没使用复制点导致没有解,恢复有复制点的状态
            {
                设置:使用double重复non-must点方案;
                恢复使用前的状态;
                continue;
            }
            else break;                     //case C:上次使用复制点了仍然没有求出解(其实不太可能)就跳出吧
        }
        else
        {
            设置:不使用double重复non-must点方案;
            //有路径条件下: 记录最终路径, 包含所有must-node和部分non-must-node
            setVtour();
            //有路径条件下: 没有重复点则跳出, 正常输出路径; 有重复点则把重复点加入vv, 继续找
            if(examRepeatNode() == 0)    break;
            //还有重复点,但是time out,then break,output NA
            if(超时) {break;}
        }
    }
    if(road有解)
        setVtourId();          //设置解得路径上的id
    swapGraphCostBack(Rpo); //将Graph的权值矩阵恢复设置惩罚前的状态
    if(road有解)
        setVtourCost();        //设置解得路径的cost
}
```

1.3.5 LKH 函数

```
long long Road::LKH()
{
    //使用次梯度优化来得到候选集,如果直接得到tour就跳到"存储返回"
    if(creatCandidates(bestCostMin))
        return bestCostMin;

    //使用Trials次findTour(主要是opt边交换操作),记录最好的路
    resetBestSucc();
    for(trial = 0,Trials = num;trial < Trials;trial++)
    {
        bestCost = findTour(bestCostMin<UnReachCost);
        if(bestCost < bestCostMin)
        {
            bestCostMin = bestCost;
            storeBestSucc();
        }
    }
    return bestCostMin;
}
```

1.3.6 findTour 函数

声明：在 findTour 和 3、4、5-opt 等函数中，会使用点的权值（程序变量名为 pi），增加点的权值相关理论请见 2.3.2。

在 LKH 源码中，有一个表示精度“Precision”的概念，在源代码中默认是 100。通常点权 pi 是不太大的，可以把新边权值设为：

Dij= Precision *Cij + Pi(i) + Pi(j);

所以在最后求到路得解得总权值时候，要减去 Pi 的和，再除以 Precision。

我们队的做法是 Dij= Precision <<7u + Pi(i) + Pi(j)；相当于 Precision=128，因为移位操作比乘法省时间。所以 findTour 最后有：减去 sumPi 和>>7u 。

```
long long Road::findTour(unsigned char hasFindTour)
{
    unsigned char KMFindTour = false;
    long long bestCost = initialTour(hasFindTour,KMFindTour);
    if(!KMFindTour)
    {
        while(1)
        {
            _5_OPT(bestCost);
            if(_4_OPT(bestCost))     continue;
            break;
        }
    }
    return (bestCost - sumPi)>>7u;
}
```

1.3.7 其它重要函数

setPunishMethod()：见代码注释。

_3_OPT()、_4_OPT()、_5_OPT()：见/paper/opt-pictures 的图解和 LKH 源代码 LKH-2.0.7 和 LKH report。这里读懂论文理解起来就非常简单了。

initialTour：情况 LKH 源代码的相应文件，注释写的比我清楚。如果在极端情况，最后一次 findTour 还求不出可能解，可以认为出现了 LKH 的短板，使用 KM 算法初始化一个可行解。

setReIdList：这个函数用与求解出两条路得重复边，及其数量。使用了一个 label 变量，使时间复杂度为 O（m+n），m 和 n 分别是两条路的节点数。

1.4 复赛和决赛使用用例

我们复赛使用的 case 是：case-semi/_case192 和 case-semi/_case294。

其中 192 这个 case 是 http：//bbs.csdn.net/topics/391943287 这四个例子的简化版本（去掉最后那个大尾巴）的组合。

其中 294 这个 case 也是小 case 的组合，素材来自于 case-anti/目录下的 4 个 case，这几个是队友手工制造的，详情可参考同一目录的“手造特例说明.pptx”。

决赛使用的两个 case 都是 case-semi/topo1800（同 mycase/case2），来自论坛，网址是：http：//bbs.csdn.net/topics/391939567。感谢分享这个 case 的小伙伴 。

亚军 Spirits 队使用的 case 是 Spirits/case4 和 DZ/up2(同 mycase/case5)。后面这个 case 来源比较曲折，本来是 DZ 队复赛时候的 case，互相交流时候发给我们队进行测试，后来我发给我们武长赛区测试 7 个 case 中就包含这个 case，结果就这一个 case 不如 Spirits 队，结果 Spirits 队就用上了，orz...

1.5 符号描述

全文使用以下符号：

必须点	must-node
非必须点	non-must-node
可达边最大值 MaxReachCost	100
所有点个数上限 MaxV	2000
小无穷不可达 SubUnReachCost	（MaxReachCost*MaxV + 1）
大无穷不可达 UnReachCost	（SubUnReachCost*（MaxV + 1））

1.6 调试环境

主机系统为 mac，虚拟机为官方指定 ubuntu 系统。两个系统使用文件共享，可以在 mac 下高效编程，ubuntu 下真实运行。mac 下 ssh 远程 ubuntu，很方便在 mac 中用命令行（实际使用的是 iTerm+zsh）调试 ubuntu 下的程序。不光是 mac，也推荐 windows 下的小伙伴使用这样的方式，高效编程与调试。

主机 CPU 是 2.7 GHz Intel Core i5，如果机器较慢，如 i3，可以把 future_net.h 中的时限宏定义 TIME_OUT 调大一点，否则将影响运行结果。

推荐一个软件 Understand，查看代码的利器，可以把程序的各种结构用图表示出来，还有代码分析，感觉非常棒。

2 求单条路思路

LKH 算法是 Keld Helsgaun 对于 Lin-Kernighan 算法的改进。

求单条路的精髓在于 LKH 算法中的 K-opt（边交换）操作，初赛时我们在源码（8000 行代码）的基础上进行修改，复赛后我们将 LKH 的核心 K-opt 重写了一遍（核心不到 500 行代码，主要是_3_opt.cpp，_4_opt.cpp，_5_opt.cpp）。

由于原版代码是无向 K-opt（即使是 atsp 问题也是转化为 tsp 来求的），而华为比赛的题目是有向图，所以我们自己实现了有向 3、4、5-opt（不存在有向 2-opt）。6-opt 虽然也搞出来了，但是 5-opt 已经很好了，而且没时间调 6-opt 了，所以最后没有用 6-opt。

原码 LKH-2.0.7 下载地址是 www.akira.ruc.dk/~keld/research/LKH/，理论依据参考这个下载包中的 DOC/LKH_REPORT.pdf，也可以参考 Keld Helsgaun 的几篇论文。另外理论依据还有 DOC/LKH_REPORT.pdf 中参考文献的几篇文献，例如 karp 的最小生成树的两篇篇，atsp 到 tsp 转化的文章，等等，均列在了本文最后的参考文献部分，另外还有《迷茫的旅行商_一个无处不在的计算机算法问题》这本书也不错。

2.1 算法描述

search_single_route 算法描述
1、规模压缩：使用 SPFA 算法求所有 must-node 间最短路，并记录路径（见 setVcost_SPFA()函数）；
2、转化为 atsp 问题：起点 s 和终点 t 合并（新点的出度为 s 的出度，入度为 t 的入度），两个 must-node 间如果没有路则设为 UnReachCost(见 Road::setCost()函数)；
3、使用 LKH 算法求解 atsp 问题：一次 Ascent 操作求候选集（见 creatCandidates 函数），n（必须点个数）次 findTour()操作。每次 findTour 操作包括一次 initialTour()和多次“有向 4-opt”和“有向 5-opt”操作；
4、根据 3 中得到的 atsp 的解和 1 中记录的路径，检查有无“重复使用的 non-must-node”。如果有，“升级”这些重复使用的 non-must-node 为 must-node，转到 1 重新求解；如果没有则求解成功（见 examRepeatNode 函数）。

2.2 规模压缩阶段

通过测试，dijkstra 的 heap 版本和 spfa 性能差不多，都优于 floyd。最后我们使用了 n 次（所有点个数）单点 spfa。具体代码详见 setVcost_SPFA（）函数和 setVcost_DIJ（）函数。

我们组的理解：non-must 点用做松弛操作的中间点，must 点不用做松弛操作的中间点。

2.3 解决 atsp 问题

解决 tsp 的算法中，LKH 很厉害，求解速度和权值都很棒，而且还能解决 atsp 问题（atsp 转化为 tsp）。

atsp 到 tsp 转化的理论依据为：TRANSFORMING ASYMMETRIC INTO SYMMETRIC TRAVELING SALESMAN PROBLEMS.pdf。这个论文可以把 tsp 问题方便地转化为 atsp 问题，顶点只需要 double 下。理解起来很简单：每个点都有个复制点，原点与复制点间距离设置为-∞（可以在程序中设置为一个绝对值很大的负数），原点只保留入度，舍弃的出度给复制点。（吐槽下，LKH report 里写的是原点保留入度，但是代码里原点保留出度）。

2.3.1 ascent 阶段-求候选集

LKH 对于 LK 的最大改进有两点：每个点求 x 个候选集，opt 操作从 2、3opt 变成 5-opt。

候选集个数 x 默认为 5，LKH 通过大量的实验测试过 5 个应该足够，有时候甚至 4 个就足够。论文原话为(LKH report Page27 最下)：Thus, in all test problems the algorithm was able to find optimal tours using as candidate edges only edges the 5 α-nearest edges incident to each node. Most of the problems could even be solved when search was restricted to only the 4 α-nearest edges。

Ascent 阶段就是求候选集的阶段，在这一阶段我们使用的是求 2*n 个节点的候选集(我们只在求候选集的阶段转化为 2*n 个节点了，opt 操作时是 n 个)。求出 2*n 个节点的候选集后：

原点（包含出度的点）的候选集（实际是出度候选集）作为真实的候选集

复制点（包含入度的点）的候选集（实际是入度候选集）放入相应入度节点的真实候选集。

如果没有读过 LKH 的 report 和上面提到的 atsp 到 tsp 转化的文章，上面写的比较难理解。

举个简单的例子：如果有三个点 A，B，C，经过 atsp 到 tsp 的转化，有 A，A’，B，B’，C，C’（带’的是复制点，包含入度的点）。

假设经过 ascent 求得候选集如下（由于 X 到 X’的距离为-∞，所以 X 到 X’互相出现在对方的候选集里面）：

A：A’

A’：A，B

B：B’，C’

B’：B，A

C：C’，A’

C’：C

使用①，得到“出度候选集”：

A：A’

B：B’，C’

C：C’，A’

使用②，以 A’：A，B 为例，由于 A’是包含入度的节点，所以 A 和 B 是 A’的“入度候选集”，可以把 A’放到 A 和 B 的“出度候选集”。所以扩充的“出度候选集”如下：

A：A’，A’

B：B’，C’，A’

C：C’，A’

然后把 B’：B，A 和 C’：C 也扩充进去，最后得：

A：A’，A’，B’

B：B’，C’，A’，B’

C：C’，A’，C’

去掉重复的，去掉自己到自己的，得到

A：B’

B：C’，A’

C：A’

去掉’，得到

A：B

B：C，A

C：A

至此，候选集已经求出，为下面的 opt 操作做好了准备。上面的例子只是随便举得，很多不得当的地方，勿较真，理解就好，如不理解请去看论文。

2.3.2 LKH report 几个非常重要的理论

1-tree（report P18 页）：在最小生成树上加一条边，会出现一个环。

α-nearness（report P20 页）：可以简单理解为某一条边成为最优解的一条边的可能性，值越小，可能性越大。

增加点权，最优路不变原理（report P24 页上半页）：dij=cij+πi+πj，就是说新的边权=原来的边权+两端点的点权，在 report 中点权表示为 π。

次梯度优化（report P25 页）：不断求 1-tree，优化节点的“点权”。简单地理解就是：不断用力挤压 1-tree，目的是形成一棵 thin tree，最好的情况下是形成只有一个环的 1-tree。

当然还有好多理论也必将重要，如果想弄懂 LKH 的原理，精读 LKH 的 report 是最快最有用的。

2.3.3 无向 K-opt

原版 LKH 使用的是无向 K-opt，具体可以参考 report 的 P8-P16 和 P28-P30，或者通读 LK 的文章 An Effective Heuristic Algorithm for the Travelling-Salesman Problem。

简单理解 K-opt：可以打个简单的比喻，在有一条初始化路的前提下，可以把这个环路看成一条绳子，K-opt 操作就是把绳子剪 K 刀，然后把这 K 小段重新组合在一起。可以看下图，分别是无向 2、3、4-opt：

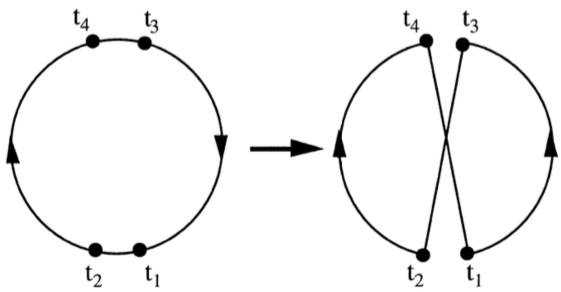


Fig. 1. A 2-opt move.

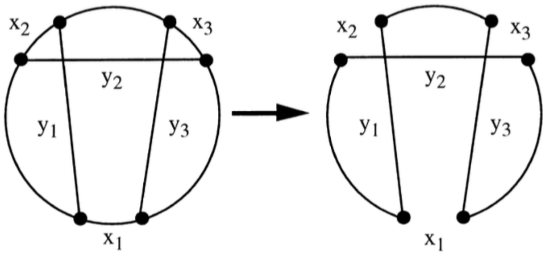


Fig. 2. A 3-opt move.

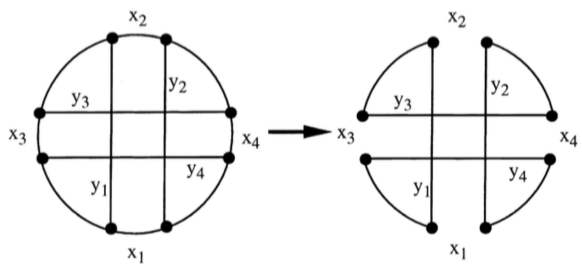


Fig. 4. Non-sequential exchange ($r = 4$).

2.3.4 关于有向 K-opt 和无向 K-opt

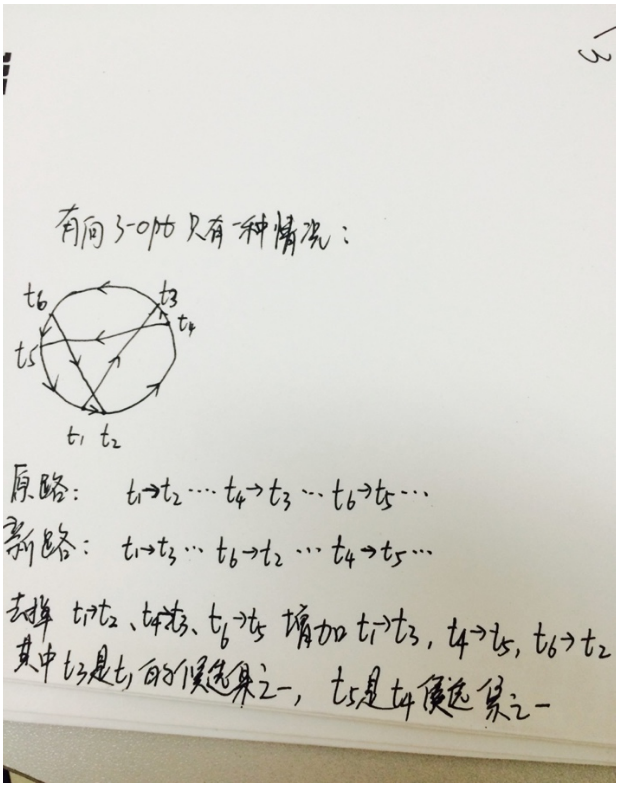
有向 K-opt 就是环路是有方向的，每个小段也是有方向的，不能反过来。而无向 K-opt 则可以随便将每个小段反过来。

现实中的 tsp 问题大多是无向的，也就对称的 tsp 问题（stsp），比如说欧氏距离上的 tsp 问题，a->b->c->d 这样的小段可以反过来走，d->c->b->a。

而华为比赛的图是有向图，可以转化为非对称的 tsp（atsp）解决，然后作为输入给 LKH。实际上 LKH 内部也是将 atsp 转化为 tsp 问题求的，所以我们组就想到可以直接用有向 K-opt 操作，就省去了 LKH 内部的 atsp 到 tsp 的转化(但是 ascent 阶段转化了，因为未来得及改成不转化版本的）。

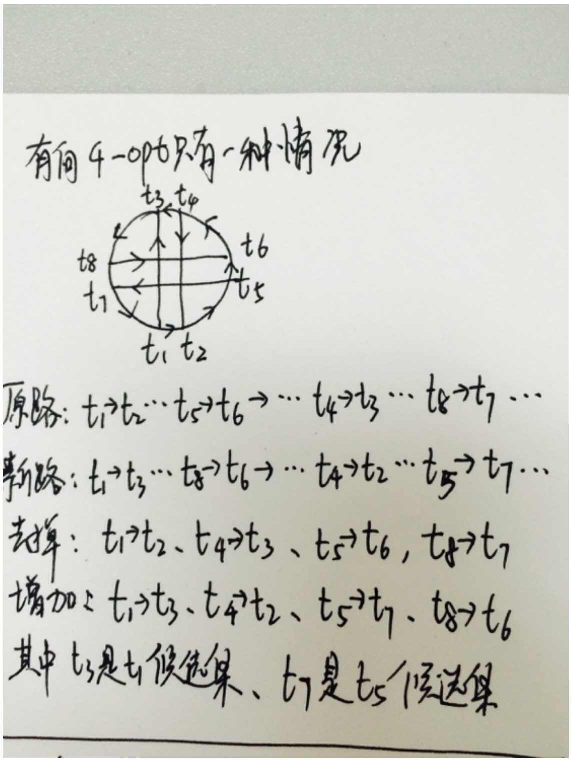
（1）有向 3-opt

如果原来的环路是 1->2>3->，其中 1，2，3 是三小段路，切三刀后重组，只能是 1->3>2->，具体看下图（高清原图在/paper/opt-pictures 中）。相关代码请看_3_OPT.cpp。



（2）有向 4-opt

如果原来的环路是 1->2>3->4->，其中 1，2，3，4 是四小段路，切四刀后重组，只能是 1->4>3->2，具体看下图（高清原图在/paper/opt-pictures 中）。相关代码请看_4_OPT.cpp。



（3）有向 5-opt

如果原来的环路是 1->2>3->4->5->，其中 1，2，3，4，5 是五小段路，切五刀后重组，有八种情况，具体看下图（高清原图在/paper/opt-pictures 中），相关代码请看_5_OPT.cpp。

1->3>2->5->4->

1->3>5->2->4->

1->3>5->4->2->

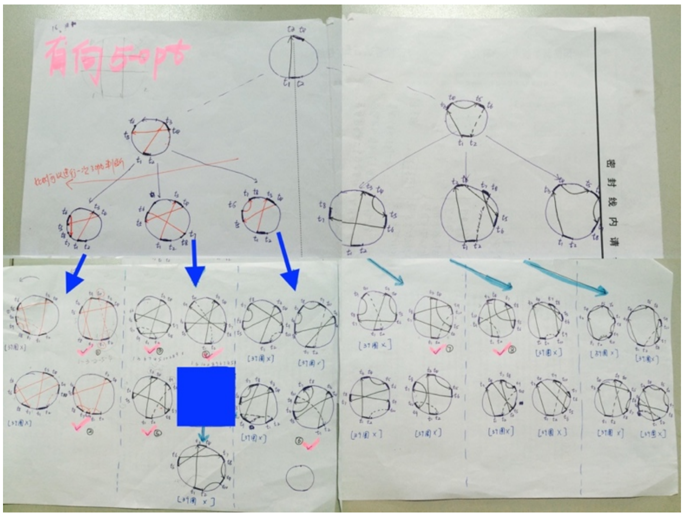
1->4>2->5->3->

1->4>3->5->2->

1->5>2->4->3->

1->5>3->2->4->

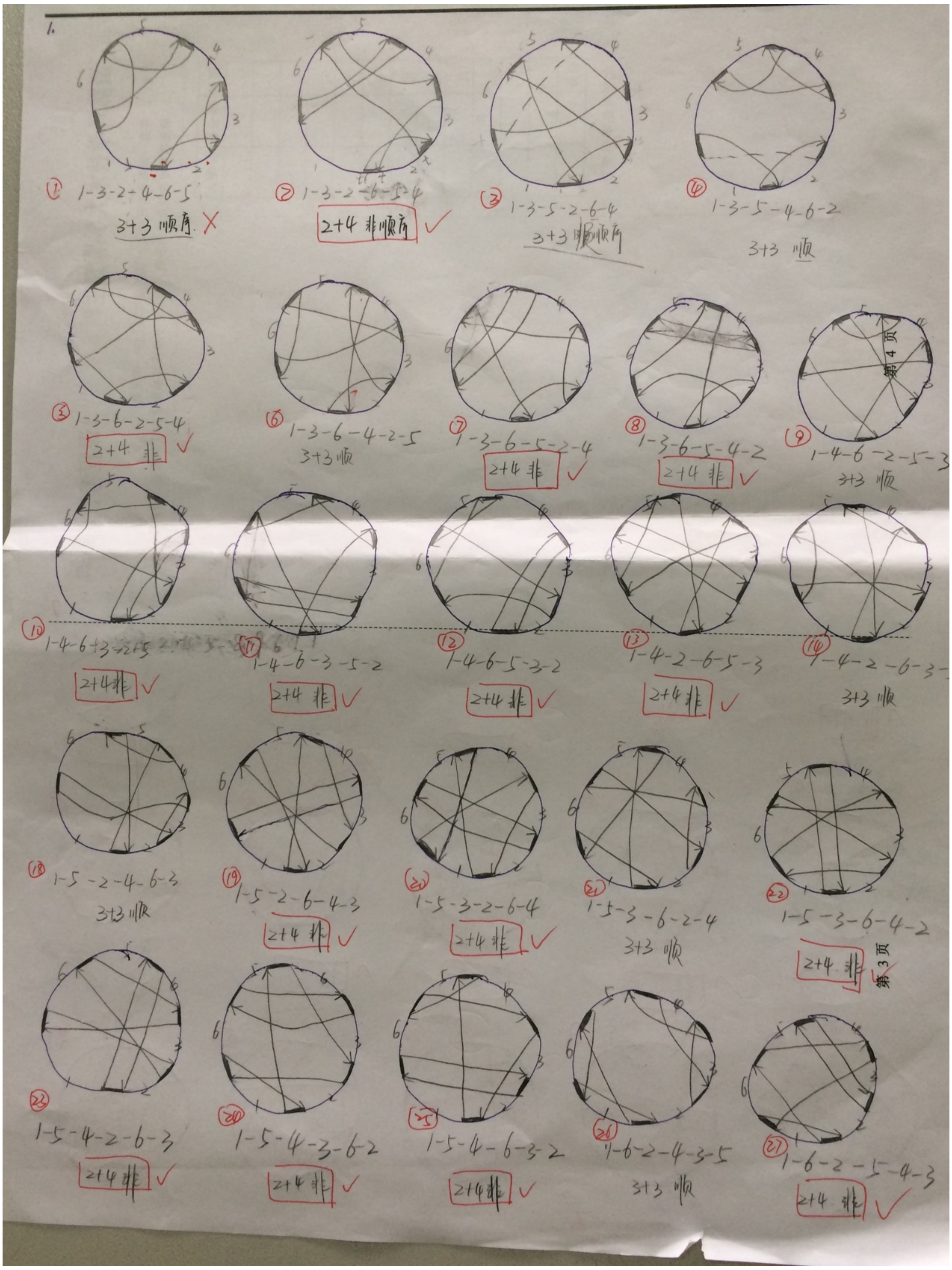
1->5>4->3->2->

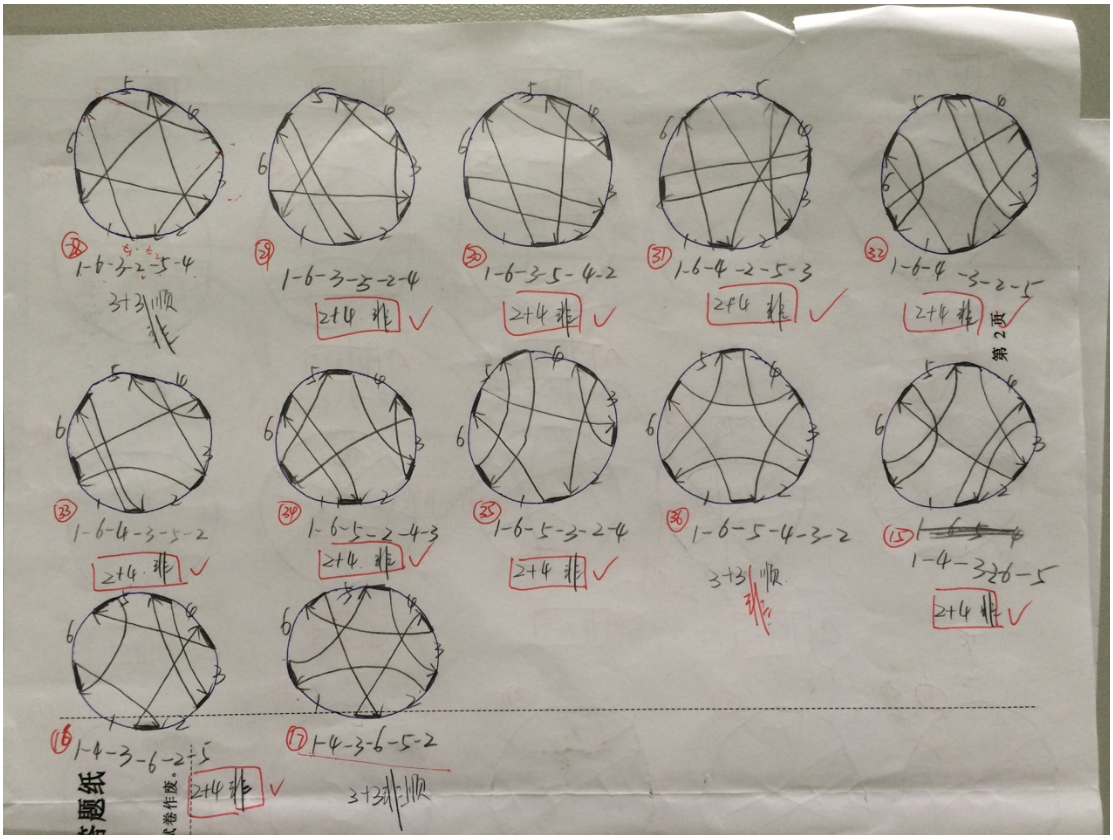


（4）有向 6-opt

虽然程序没有用到，但是也画出来了，主要是因为 LKH 说 5-opt 基本上足够了，而且 6-opt 毕竟复杂，没时间编码调试。

一共有 36 种情况，并且全部是 Non-sequential 的。其中 11 种是 3-sequential-opt 和 3-sequential-opt 的组合，这种情况可以看成是两次有向 3-opt 操作，所以是无效的。剩下的其中 22 种是（2+4）-opt，3 种是（3+3）-opt，是有效的，具体看下图（高清原图在/paper/opt-pictures 中）。


1-3-2-4-6-5
3+3 顺序 ×
1-3-2-6-5-4
2+4 非顺序 ✓
1-3-5-2-6-4
1-3-5-4-6-2
3+3 顺
1-3-6-2-5-4
2+4 非 ✓
1-3-6-4-2-5
3+3 顺
1-3-6-5-2-4
2+4 非 ✓
1-3-6-5-4-2
2+4 非 ✓
第 4 页
1-4-6-2-5-3
3+3 顺
2+4非 ✓
1-4-6-3-5-2
2+4 非 ✓
1-4-6-5-3-2
2+4 非 ✓
1-4-2-6-5-3
2+4 非 ✓
3+3 顺
1-5-2-4-6-3
3+3 顺
1-5-2-6-4-3
2+4 非 ✓
1-5-3-2-6-4
2+4 非 ✓
1-5-3-6-2-4
3+3 顺
1-5-3-6-4-2
2+4 非
第 3 页
1-5-4-2-6-3
2+4 非 ✓
1-5-4-3-6-2
2+4 非 ✓
1-5-4-6-3-2
2+4 非 ✓
1-6-2-4-3-5
3+3 顺
1-6-2-5-4-3
2+4 非 ✓



2.3.5 关于 Non-sequential K-opt 和 Sequential K-opt

翻译成中文就是“非顺序 K-opt”和“顺序 K-opt”，具体请看 report Page10。

简单理解：Non-sequential 就是，不能顺序地找到要断开的边和新加的边。可以简单对比上面的有向 3-opt 和有向 4-opt，分别是 Sequential 和 Non-Sequential 的，一看便懂得怎么区分。再具体请看相关论文。

2.3.6 LKH 短板：initial tour

initial tour 是 LKH 短板，在 opt（边交换）之前，需要有一个切实可行解或非常接近可行解的解，否则怎么 opt 操作也没用。打个比喻，如果初始化路是 8 个连通的小段组成，且这 8 段之间是不连通的，那么怎么使用 5-opt 也是没用的，即使是 7-opt 也没有，因为 K-opt 就是切 k 刀，如果 k<8，怎么也不会将初始化的 8 段重组为一个完全连通的可行路。

结论就是：LKH 更加适用于稠密图，不适用于非常稀疏的图，或者可以说成是可行解非常非常少的图。

LKH 算法不能解决 http://bbs.csdn.net/topics/391941776 这个 101 条重复边的 case。据我们研究，这个 case 的可行解貌似只有一个，然后 LKH 就呵呵了。。。估计 8-opt 也解决不了这个 case。

2.3.7 指派算法 KM 粗略克服 LKH 短板

在上一小节提到了 LKH 的短板，可以这么说，只要找到一条可行解作为初始解，LKH 基本上是无敌的。

所以我们增加了 KM 指派算法，在 findTour 的最后一轮还无法形成一个可行解，就使用 KM 指派算法强行初始化一个可行解，在这个解得基础上进行 opt 操作。见 initialTour.cpp 的 20-25 行。

如果不使用 KM 算法，有少部分 case 是求解不出的，使用后，除了上节提到的 http：//bbs.csdn.net/topics/391941776 这个 case 解不出，还未遇到其它解不出的 case。

2.3.8 小结

有向 3-opt：只有一种，而且是 Sequential 的。

有向 4-opt：只有一种，而且是 Non-Sequential 的。

有向 5-opt：有 8 种，而且是 Sequential 的。

有向 6-opt：有 36 种，而且是 Non-Sequential 的，11 种是无效的，22 种是（2+4）-opt，3 种是（3+3）-opt，

程序只使用了 4-opt 和 5-opt，不用 3-opt 是因为它被 5-opt 包含进去了（因为 5-opt 在运行中会出现 3-opt 的情况）。不用 6-opt 是因为比较复杂，而且 5-opt 已经足够了（但 6-opt 毕竟没有实际测试过，小伙伴们可以测试下把使用的效果反馈给我）。

2.4 “升级”操作的两种方案

第一种方案：直接将所有的重复 non-must 点升级为 must 点，并配套使用“double 所以重复 non-must 点模型”。

第二种方案：和上一个方案比，使用某一规则有选择性地升级，即只升级一部分点。

复（决）赛使用的是方案一，初赛使用的是方案二。

先说说方案二，有选择性地进行升级有两点缺点：

1、使用什么规则？

2、每次只升级一部分节点，在极端情况下会运行非常多轮 atsp 求解。

针对缺点 1、使用什么规则，经过研究测试，初赛最后使用的规则是：只加入两个 must 点间最边上的 non-must 点。这个规则通用性比较好，但是有一些极端 case 直接造成求不出解，详见 HuaWei_Code/case-pre/case-anti/手造特例说明.pptx 和相同目录下的几个 case。

为了克服方案二的两个缺点，复（决）赛改为使用方案一，升级所有点可以减少求解 atsp 次数，因为一次性把 non-must 点都升级了。但是也会造成

HuaWei_Code/case-pre/case-anti/手造特例说明.pptx 这几个 case 求不出解，此时就配套使用“double 所有重复 non-must 点模型”，相信好多队伍使用了类似的模型“double 所有 non-must 点模型”。所以，我们没有 double 所有的 non-must 点，只是 double 所有的重复 non-must 点。因为 non-must 点最多有 1900 个，double 下变成 3800 个，规模太大了，而只 double 重复的 non-must 点，在大多数情况下，规模很小，因为本身重复的 non-must 点就很少（虽然极端情况下也会出现 3800 的最坏情况）。

为此，我们的程序是默认每次只是升级重复 non-must 点，如果求不出解，还原求解前的状态，再使用“double 所有重复 non-must 点模型”。

2.5 规模压缩阶段-续

整个程序的主要耗时都花在 spfa 上，大约占 70%的时间，在重复"升级重复结点，跑 spfa，跑 tsp"这一步骤中，重复结点只增不减，而且通过测试大量 case 可以发现，每次升级的点（重复 non-must 点）很少。故可以利用上次 spfa 的结果来减少当前 spfa 的计算次数（实现的函数是 setVcost_reduced_SPFA（）），这一点在决赛中并没有好的效果， 原因在于决赛时官方 case 的规模并不算大。如果对于大规模 case，通常是可以省一些时间的。

代码使用方法为:把 search_single_route.cpp 中第 37 行 setVcost_SPFA();改成如下：

```
if(solveTspCn==1)
    setVcost_SPFA();
else
    setVcost_reduced_SPFA();
```

2.6 规模压缩-优点与缺点

优点:非常省时间。规模可以从 2000 降到 100，通常寻路算法至少是 O(n^2)的，所以压缩后的寻路程序执行时间非常短 。

缺点：要重复使用求解 atsp 几次才能求得一条正确的路，而且规模压缩的 spfa 很费时。还有就是，处理小规模的 case，压缩会增加很多静态开销，比如说 300 个点的 case，更适合一次性求解，不适合压缩（为了比赛公平性，我们没有做 case 规模判断，所有 case 都压缩求解）。

对比优点缺点，还是优点更加突出，因为不压缩的话，最大规模的 case 执行时间很可能非常长（尤其是搜索类算法）。

3 迭代求两条路思路

3.1 算法描述

search_double_route 算法描述
1：根据问题难易程度设置惩罚方式（Road：：setPunishMethod（）函数）： ①裸求 Road0，把 Road0 走过的路设为：原来的权值+ SubUnReachCost，然后求 Road1，如果 Road0 或 Road1 无解则转②，否则转③ ②裸求 Road1，把 Road1 走过的路设为：原来的权值+ SubUnReachCost，然后求 Road0 ③三种情况： 重边数为 0：使用“增量式惩罚”方案（setPunishMethod.cpp 的 38 行） Road0 或 Road1 无解：使用“增量式惩罚”方案（setPunishMethod.cpp54 行） 重边数不为 0：使用“小无穷惩罚”方案（setPunishMethod.cpp 的 60 行） 2：最大 4*10 次（四轮的迭代次数在 1 中设置了）循环迭代（Road：：search_double_route（）函数） 第一轮：1->2->1->2->1… 第二轮：2->1->2->1->2… 第三轮：使用启发式规则“必经点集合互不侵占” 1->2->1->2->1… 第四轮：使用启发式规则“必经点集合互不侵占” 2->1->2->1->2…

3.2 “增量式惩罚”方案

在某一轮迭代中，1->2->1->2->1…，1 和 2 交替惩罚，惩罚值每次加 10，即惩罚值从 10 增长到 100。

“增量式惩罚”方案更加侧重：两条路权值和更小。因此在 3.1 算法描述的步骤 1 中，先预估 case 的重复边数，如果重复边为 0，则使用此惩罚方案，会对权值和有好处（因为此时问题比较简单，预估阶段就出现了重复边个数为 0）。

经验来看，如果 case 规模较小（依据运行时间判断），则只需执行第 1、3 轮，这两轮只迭代 7 次即可。

这个方案在我们测试来看，在无重复边的前提下，是最最好的一个的方案，能够使权值降的很低。

3.3 “小无穷惩罚”方案

在某一轮迭代中，1->2->1->2->1…，1 和 2 交替惩罚，惩罚值=原来边的权值+小无穷 SubUnReachCost 。

“小无穷惩罚”方案更加侧重：两条路重复边个数最优。因此在 3.1 算法描述的步骤 1 中，先预估 case 的重复边数，如果重复边不为 0，则使用此惩罚方案，会对重复边个数有好处。

经验来看，预估阶段（setPunishMethod()）重复边不为 0，如果在 4 轮迭代运行中，出现了重复边个数为 0 的情况，惩罚方案立马转变成 3.2 的 “增量式惩罚”方案会让结果更好。（setPunishMethod.cpp 的 82-86 行）

3.4 启发式规则“必经点集合互不侵占”

按照经验来看，在 1->2->1->2->1…的迭代中，其实其中一条路求出解后，不必所有走过的路都设置惩罚，可以只设置一部分。此思路来自于东南

问题中一共有三个集合，必须点集合 M1、M2 和非必须点集合 N。如果把这三个集合比喻成领土的话，M1、M2 是主权国家，N 是没有主权的第三方土地，M1 和 M2 都想要 N 中的点是情有可原的，因为 N 毕竟是没有主权的（非必须点）。

求解过程中，M1 和 M2 是互为非毕竟点集合的，不可避免造成 M1 和 M2 使用对方的节点，按照国家领土竞争的思维，这就相当于对方国家已经用到我的国家的领土，这是不能忍的，所以启发式规则如下：求出路径 1 后，在给路径 2 设置惩罚时，1 经过了 2 的必经点的边不加惩罚。

这条启发式规则已经被我们测试过，并使用在程序中，效果很棒。在复赛前，官网的两个 case 能得到 756 和 1449。

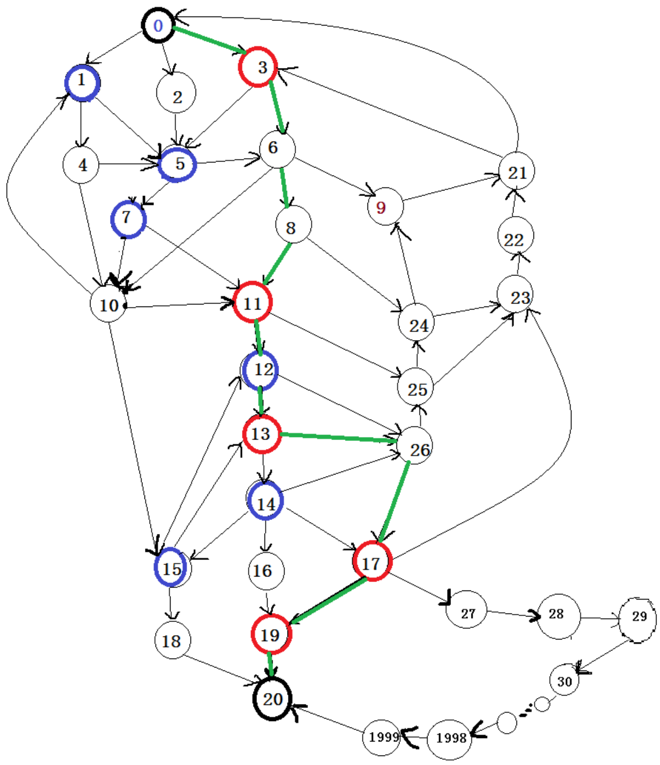
这条启发式规则用于 4 轮迭代中的后两轮（swapGraphCost.cpp 的 36-40 行），前两轮不用这个启发式规则，因为题目毕竟是 np 难的问题，更好的解不一定会出现在哪一轮，所以 4 轮都测试。

3.5 启发式规则“规模压缩阶段不走老路”

在 spfa（规模压缩）过程中，随机选择“顺序”枚举邻接表还是“逆序”枚举邻接表，这样做的效果是“不走老路”，对于减少两条路得重复边很有好处。

这条启发式规则已经被我们测试过，效果很棒，尤其对于处理这四个例子（如下图）效果非常棒 http：//bbs.csdn.net/topics/391943287，均能达到已知的的最优解。

相关代码出现在，setVcost_SPFA.cpp 的 28 和 44 行。



3.5 使用 hash 保存结果避免重算

在迭代过程中，解会重复出现， 使用 hash 保存结果避免重算。

相关代码出现在 search_single_route.cpp 的 16-22，swapGraphCost.cpp 的 55 行和 85-98 行。

使用的 hash 算法参考自 https://www.byvoid.com/blog/string-hash-compare，最后使用的是 SDBMHash 算法。

3.6 两点间有多条边的处理

我们组数据结构如下，含义见注释，id 和 cost 存两点间最短边，subId 和 subCost 存次短边。

```
class Graph
{
private:
    void setCost();              //used in initial()
public:
    int cost[MaxV][MaxV];        //两点之间最短路径
    int id[MaxV][MaxV];          //两点之间最短路径ID
    int subCost[MaxV][MaxV];     //两点之间第二短路径
    int subId[MaxV][MaxV];       //两点之间第二短路径ID
    VNode Node[MaxV];            //所有点的结构体,used in SPFA
    int num;                     //number of nodes in V

    Graph(){};                                               //used in route()
    void initial(char *topo[MAX_EDGE_NUM], int edge_num);    //used in route()
};
```

3.6.1 读入阶段

由于只求两个路径，所以只需要保存从 v1 到 v2 最小的两条边即可（Graph.cpp 的 36-53 行）。

3.6.2 两条路迭代阶段

以 1->2->1->2->1…为例，迭代的思路是：首先让 1 占用图中两点间最小的边，1 走过的路设置阻碍求 2。

上面思路的前提是 1 走过的边无“次短边”。特殊地，如果 1 经过[v1,v2]，且[v1，v2]有“次短边”，在给 2 设置阻碍阶段，[v1，v2]这条边权值换成“次短边”的权值。

3.6.3 关于 3.6.2 的进一步思考

以 1->2->1->2->1…为例，如果 1 不完全用最短边，用了一些次短边，会对迭代结果造成影响，有小的几率结果会好，应该是带了一些随机性的原因。

通常按照 2.5.2 那样做就已经足够了，如果有更好的思路处理次短边可联系我。

参 考 文 献

[1] LKH-2.0.7/DOC/LKH_REPORT.pdf
[2] S. Lin & B. W. Kernighan,
“An Effective Heuristic Algorithm for the Traveling-Salesman Problem”, Oper. Res. 21, 498-516 (1973).
[3] R. Jonker & T. Volgenant,
“Transforming asymmetric into symmetric traveling salesman problems”, Oper. Res. Let., 2, 161-163 (1983).
[4] M. Held & R. M. Karp,
“The Traveling-Salesman Problem and Minimum Spanning Trees”, Oper. Res., 18, 1138-1162 (1970).
[5] M. Held & R. M. Karp,
“The Traveling-Salesman Problem and Minimum Spanning Trees: Part II”, Math. Programming, 1, 16-25 (1971).
[6] William_J_迷茫的旅行商_一个无处不在的计算机算法问题