必经点最短路径算法的设计与实现

谭钧升

第二章 背景介绍（问题与基本蚁群算法，蚁群系统）

第三章 混合蚁群算法的设计与实现

由于必经点最短路径问题与经典的 TSP 问题不同， TSP 问题求解的是经过所有点的哈密顿回路，而必经点最短路径问题求解的是从起点到终点而且经过必经点的最短路径。另外，华为比赛给出的题目的边是有向边，两点直接甚至可能有多达 8 条的同向边。所以，对于华为的这个题目，不能简单的直接应用蚁群算法来求解。因此，我们需要对基本的蚁群算法进行大量的改进以求出题目的解。而第一章里面提到，蚁群优化算法(Ant Colony optimization，下文称 ACO)的种类很多，包括精英蚂蚁系统（Elitist Strategy for Ant system, EAS），最大最小蚂蚁系统（MAX-MIN Ant System, 下文称 MMAS）和蚁群系统（Ant Colony System, 下文称 ACS）等。而根据[1]的实验结果，ACS 在求解 TSP 问题时，解的质量和求解时间都比其他的 ACO 算法好，因此，我们决定选用 ACS 作为我们整个算法的基本框架。

基本的 ACS 算法

蚁群系统（ACS）是由 Dorigo 教授和 Gambardella 教授于 1997 年提出的蚁群优化算法。它的优化主要体现在：第一，它采用了一种更积极的行为选择规则，从而更好地利用了蚁群的积累的信息素；第二，信息素挥发和释放只在至今最优路径的边上进行；第三，蚂蚁每一次使用边（i，j）从点 i 移动到点 j 后，都会对该边上的信息素进行一定量的挥发，以增加其他蚂蚁探索其它路径的可能性。

参照[1][2]，我们用 C++实现了 ACS 算法的基本框架。以下是 ACS 的主要数据结构和变量：

```
struct Ant{
    std::vector<unsigned short>    tour;   // the path
    unsigned short curNode;              // which node this ant is currently at
    unsigned short requireCnt;           // how many require nodes has this ant passed
    int    tour_length;
    bool visited[MAX_NODES];
    bool stop;                    // whether this ant stops
};
struct Edge{
     unsigned short linkid;              // the max. of linkid is 4800
     unsigned short cost;
     Edge() { cost = 0;}
};
/* matrix[i][j] contains edge for node i->j */
Edge       matrix[MAX_NODES][MAX_NODES];
/* adjList[i]: contains the adjacent list of node i */
std::vector<unsigned short>         adjList[MAX_NODES];
/* pheromone matrix, one entry for each arc */
double     pheromone[MAX_NODES][MAX_NODES];
/* combination of pheromone times heuristic information */
double     total[MAX_NODES][MAX_NODES];
double     rho;                              /* parameter for evaporation */
double     alpha;                            /* importance of trail */
double     beta;                            /* importance of heuristic evaluate */
Ant        *best_so_far_ant;                /* struct that contains the best-so-far ant */
```

其中，Ant 表示每只蚂蚁，其中的 requireCnt 用来保存该蚂蚁已经经过了多少个必经点。Edge 则代表图的边。由于题目给出最多只有 600 个点，因此我们使用一个二维数组 matrix 来保存所有点两两之间的边。而用一个邻接表来保存每个节点的出度点的列表。而二维数组 pheromone 保存的则是点 i 到点 j 的边上的信息素，为了方便计算，我们使用 total 数组保存信息素与启发式信息相乘之后的结果。Best_so_far_ant 保存的则是算法每轮迭代之后找到的最优结果，我们会使用这个蚂蚁来进行全局信息素的更新动作。

ACS 算法的流程可以总结如下：

1， 初始化所有路径的信息素浓度；
2， 让每只蚂蚁都构造出一个解（可行解或无效解）；
3， 找出所有蚂蚁构造的解中，路径最优的一个，并进行相应的信息素浓度更新；
4， 重复 2，直到满足算法终止条件。

以下是 ACS 算法的主要实现：

```
for ( n_try = 0 ; n_try < max_tries ; n_try++ ) {
        init_try(n_try);
        while ( !termination_condition() ) {
                construct_solutions();
                update_statistics();
                pheromone_trail_update();
                search_control_and_statistics();
                n_tours++;
        }
}
bool termination_condition( )
{
   return ( ((n_tours >= max_tours) && (elapsed_time( VIRTUAL ) >= max_time)));
}
double heuristic(const unsigned short &from, const unsigned short &to)
{
        return 1.0 / ((double) matrix[from][to].cost);
}
```

其中 for 循环表示会进行 *max_tries* 次尝试，而每次尝试之间是相互独立的，也就是通过 init_try 函数来将保存的路径的信息素等数据清空。而每次尝试的终止条件是：（1），总共进行了 n_tours 次迭代；（2），超时。而 heuristic 函数则表示从点 i 到 j 的启发式因子，函数使用的是点 i 到点 j 的边的长度的倒数作为启发因子，比如点 i 能够到达点 j 和 h，但是到达点 j 的边权重是 5，到达点 h 的边权重是 3，由于该启发函数的作用，蚂蚁在点 i 时选择点 j 的概率会更大。

每次迭代都会循环的执行：（1），让每只蚂蚁都构造一个解（construct_solutions）；（2），更新相关统计信息，如统计本轮迭代的最优结果（update_statistics）；（3），进行路径信息素的更新，包括用最优的解来进行全局信息素的释放等（pheromone_trail_update）；（4），其他的搜索控制和数据统计工作（search_control_and_statistics）。

而整个迭代过程中，又以 constrcut_solutions()函数最为核心。该函数的主要实现如下所示：

```
void construct_solutions(){
/* Place the ants on start node*/
      for ( k = 0 ; k < n_ants ; k++ )
            re_place_ant(ant[k]);
      int stopAnts = 0;
      while ( stopAnts < n_ants ) {
            for ( k = 0 ; k < n_ants ; k++ ) {
                  if(ant[k].stop) continue;
                  bool stop = choose_and_move_to_next(ant[k]);
                  local_acs_pheromone_update( &ant[k]);
                  if(stop) {
                        stopAnts++;
                  }
            }
      }
}
```

函数首先将所有蚂蚁重新置于起点，接下来的 while 循环的终止条件是每只蚂蚁都停止了运动，也就是每只蚂蚁都构造了一个解，但是这个解可能是可行解也可能是非可行解，非可行解的情况包括：(1), 蚂蚁没有经过所有必经点就到达了终点; (2),蚂蚁进入了死胡同。接着每次 while 循环都让每只蚂蚁前进一步，也就是 choose_and_move_to_next 函数的作用，在该函数里，通过 pheromone 信息素和 heuristic 启发因子来共同决定下一步要选择的点。选择某个点的概率计算公式为: p = pow(pheromone[from][to], alpha) * pow(heuristic(from, to), beta)，某个点的 p 值越大，蚂蚁选择该点的概率也越高。在蚂蚁移动了一步以后，local_acs_pheromone_update 函数对这只蚂蚁刚走过的路径的信息素浓度(pheromone)进行衰减，以让其它蚂蚁有更大的概率选择其它路径。

必经点最短路径问题的 ACS 算法

前面小节介绍的只是 ACS 算法的基本框架，对于解决必经点最短路径问题来说远远不够。而 Dorigo 教授最初提出 ACS 算法是为了解决 TSP 问题，而且在其论文[2]中针对的还是 TSP 问题中较简单的一种：对称 TSP 问题。因此，为了解决必经点最短路径问题，我们需要首先分析该问题与 TSP 问题的区别，或者说，华为提出的这个必经点最短路径问题比 TSP 问题复杂在哪些地方：

1. **可行解的个数不同**：对于 TSP 问题，要求的是一条经过所有点的哈密顿回路，起点和终点是不固定的。因此，对于最简单的对称 TSP 问题，我们随便选择一个点的排列，比如 abcdefg…z 或者 bcdeqz…a，都可以是问题的一个可行解（只是路径的长度肯定不是最优的）。但对于必经点最短路径问题，起点和终点是固定的，而且要求路径必须经过所有的必经点才能到达终点，因此，可行解的个数大大减少了。
2. **图是否可能存在“死胡同”**：对于 TSP 问题，整个图是强连通的，也就是在任意一个点，都会有路径到达下一个未访问过的节点（或者回到起点形成一条哈密顿回

路）。但对于华为的这个问题来说，图却存在着“死胡同”，也就是蚂蚁在到达某个点之后，发现这个点的出度表中，所有的点都已经被访问过了，或者这个点的出度为 0。这时，这个蚂蚁没有构造出一个可行解便停止了运动。

而正是这两点区别，导致了求出必经点最短路径问题的一个可行解的困难。而后来我们在实际比赛中也发现，对于复杂的图，比如结点数在 200 以上，或者环数、死胡同非常多的图，蚁群系统在 10 秒内甚至不能求出一个可行解，更别提求出最短路径了。

除此之外，我们发现这个问题还有一个陷阱会导致使用基本的 ACS 算法很难求出甚至求不一个可行解：如图 1 所示，点 0 和 3 分别是起点和终点，1 是必经点，除了边(0,1)的权重是 20 以外，其他的边的权重都是 1。在这种情况下，由于 ACS 使用了边的权重作为启发因子，当蚂蚁在起点 0 选择下一个点进行移动时，边（0,1）的权重比边（0，2）的权重大太多，因此在计算概率时，概率 p(0,1)会比 p(0,2)小很多，也就是蚂蚁在起点 0 选择点 2 作为下一个点的概率会大很多，导致构造的路径缺少必经点 1。

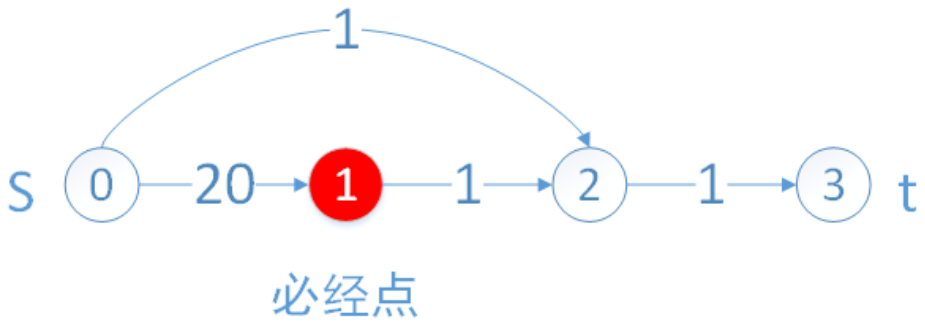

图 1

基于这些发现，我们将 ACS 算法划分了两种模式，一种是忽略边的权重的模式(ignoreWeightMode)， 一种是权重模式。算法一开始时进入的是忽略权重的模式，直到找到一个可行解之后，再进入权重模式对求出的该路径的权重进行优化。 相对应的，我们需要对启发信息函数做如下修改：

```
double heuristic(const unsigned short &from, const unsigned short &to){
    if(ignoreWeightMode) {
        return 1;
    }else{
        return 1.0 / ((double) matrix[from][to].cost);
    }
```

而我们的 ACS 算法也由此分成了两部分，首先是忽略边的权重求出一个可行解，然后是基于这个求出的解来求更优的解。接下来，我们也从这两个部分来进行算法的设计和分析。

1. 求出可行解

传统的 ACS 算法在所有蚂蚁构造出一条路径之后，会根据相应的评判标准来选择一条所有路径中最优的一条路径，然后使用这条路径进行相应的信息素释放的动作，以增加该路径上的信息素浓度，这个步骤称为蚁群算法的“正反馈”机制。而为了便于理解和描述，我们将该机制称为对蚂蚁的“奖励”。由上文提到的题目存在的陷阱可以知道，我们在刚开始求一个有效解的时候，不能将边的权重考虑进来。那么，算法对蚂蚁“奖励”的评判标准是

什么呢？答案就是蚂蚁经过的必经点的个数。如果一个蚂蚁经过的必经点个数越多，那么可以说该蚂蚁越可能满足所有条件（从起点出发，经过所有必经点，到达终点）并构造出一条可行路径。如果在该蚂蚁经过的路径上释放信息素，就可以吸引更多的蚂蚁过来并探索路径周围的点（在算法中表现为蚂蚁选择该路径上的点的概率更高），从而能在该蚂蚁经过的路径的基础上构造更好的路径。

但是，有了“奖励”还不够。在对算法测试中我们发现，在算法初期，许多蚂蚁会重复的走进同一条“死胡同”，只有在算法后期通过“奖励”将好的路径的信息素浓度提高之后，才能减少蚂蚁走进“死胡同”的概率。而这个问题会导致算法的收敛时间变长，如果只是求出一条可行路径都花费了很长时间，就谈不上对可行路径的权重进行优化了。为了解决这个问题，我们尝试增加对最优蚂蚁的“奖励”，也就是增大对该蚂蚁经过的路径的信息素释放的浓度，但发现这样会导致所有蚂蚁选择路径以外的点的概率大大减少了，导致最后所有蚂蚁都只走这一条路径，算法进入停滞状态。既然加大“奖励”好的路径不行，我们可以引入“惩罚”机制，对坏的路径（死胡同）进行“惩罚”。与“奖励”机制相反，“惩罚”机制是对某个十分差的解（比如“死胡同”）进行信息素浓度的衰减，这样蚂蚁选择该条路径的概率就会降低。通过“惩罚”机制，我们减少了蚂蚁进入已知的“死胡同”的概率，却不会影响蚂蚁对其他路径的探索。

我们的 ACS 算法的“奖励”和“惩罚”机制可以总结如下：

1， 每只蚂蚁在到达终点（此时蚂蚁可能还没经过所有的必经点）或者进入“死胡同”时都会停止运动；
2， 将停止运动的蚂蚁视为构造出了一个解；
3， 进行奖励：找出所有蚂蚁中经过必经点个数最多的蚂蚁，对该蚂蚁经过的路径进行信息素的强化；
4， 进行惩罚：（1），如果蚂蚁经过了 0 个必经点，对该蚂蚁经过的路径进行信息素的衰减；（2），如果蚂蚁没有到达终点就停止了运动（也就是进入了“死胡同”），对该蚂蚁经过的路径上所有不包含必经点的边进行信息素的衰减；

相应的，我们的 ACS 算法的主要流程也需要增加奖励和惩罚的步骤，如下所示：

```
while ( !termination_condition() ) {
        construct_solutions();
        update_statistics();
        pheromone_trail_update();
        punish_and_reward();          // add
        search_control_and_statistics();
        n_tours++;
}
```

通过增加奖励和惩罚机制，我们的算法的求解能力得到了增强，对于规模不大的图，算法找到一条可行路径的时间也减少了。但是，对于规模很大并且很复杂的图，比如点的个数超过 400 个点的图，我们的算法仍然需要很长的时间才能求出一条可行路径。我们经过不断的测试观察发现，这是由于算法开始时，我们将图的所有路径上的信息素浓度都设为了同一个初始值，因此蚂蚁选择每一个点的概率都是一样的，这就导致了在算法初期，所有的蚂蚁基本都在“碰运气”：运气好的可能会走到必经点，然后通过释放信息素提高其他蚂蚁经过这个必经点的概率。运气差的可能一直都走不到必经点，导致大量的时间浪费在走其他无用

路径上。因此，我们需要一个机制将必经点与非必经点区别开来，让蚂蚁即使是在算法的初期也会更大概率的选择必经点，或者说，这个机制可以让必经点“吸引”蚂蚁。于是，我们设计出了“气味”机制：让必经点“散发”气味，蚂蚁在进入距离该必经点一定范围后会被该“气味”吸引过来。如图 2 所示，必经点（用红色标注的点）散发的“气味”（阴影部分）的范围包括了点 0 和点 1。这样，当蚂蚁在起点 S 选择下一个点时，由于点 0 上有气味影响，所以蚂蚁选择点 0 的概率会更高。因此，我们的“气味”机制相当于以必经点为圆心，一定的半径画圆，如果某个点在圆的范围内，则蚂蚁选择这个点的概率就会更高。

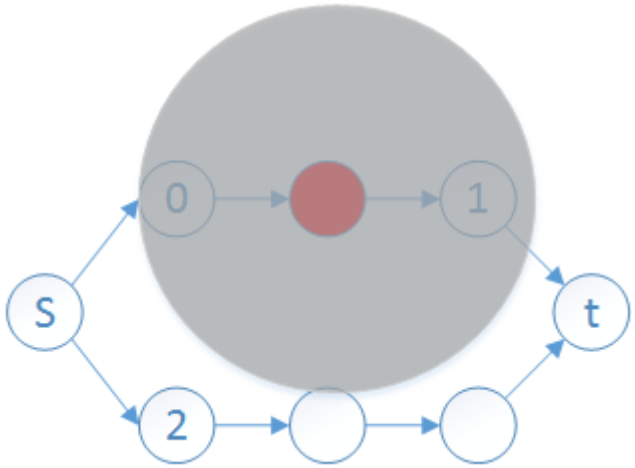

图 2

引入“气味”机制后，蚂蚁在点 i 选择点 j 的概率的计算公式则变成了：

```
p[i][j] = pow(pheromone[i][j], alpha) * pow(heuristic(i,j),beta) * pow(smell[j], gama);
```

其中，smell[j]表示必经点在该点的气味浓度，gama 表示气味的权重。而关于“气味”机制的具体实现，读者可以前往[3]了解。

经过测试，在引入“气味”机制后，对于华为的 15 个测试用例，我们的 ACS 算法能够很快就求出一个可行解（第 14,15 个测试用例耗时接近 1 秒，其他 13 个测试用例只需几毫秒或者几十毫秒）。这为我们进一步求最短路径打下基础。

2. 优化可行解

在求出一条可行路径后，我们的 ACS 算法便切换为权重模式，即使用边的权重作为启发因子。而且，我们会使用算法第一步求出的可行路径作为当前最优路径来指导蚁群的探索，也即在每轮迭代里，使用该路径作为全局最优路径来进行全局信息素浓度的更新。在蚁群找到一条更短的可行路径之后，便会将该路径替换为全局最优路径。

在测试中我们发现，虽然将边的权重考虑进来以后，找到的可行路径的权重被优化了不少，但是却出现了算法过早停滞问题，也就是算法收敛到了一个局部最优解。这是由于随着算法的运行，某条可行路径上的信息素浓度越积越多，导致蚂蚁基本只会走这条路径。所以，为了解决这个问题，我们又引入了最大最小蚂蚁系统（MMAS）里面限制路径信息素浓度范围的机制：1，把信息素大小的取值范围限制在一个区间[T_{min}, T_{max}]；2，每当算法进入停滞

```
double p_x = exp(log(0.05)/nodes.size());
// 信息素的最小值
trail_min = 1. * (1. - p_x) / (p_x * (double)((8) / 2));
// 信息素的最大值
trail_max = 1. / ( best_so_far_ant->tour_length );
```

状态，或者在一定数量的迭代过程中不再有更优的路径出现时，重新初始化所有的信息素等数据。而这个信息素的最大最小值也是会在算法运行过程中根据当前最优路径的权重动态调节的，参考[1]，我们可以得出如下的计算公式：

通过将 MMAS 与 ACS 结合，我们的混合蚁群算法求出来的权重得到了一定的优化，如图 3 所示是将 MMAS 引入 ACS 之前的华为测试的结果，图 4 所示是使用 MMAS 优化之后的结果。其中测试用例 13 的权重甚至提升了 2 倍多。

用例名称	Cost	Time	用例得分
1	4	1	100.0
2	16	1	100.0
3	31	1	100.0
4	62	1	100.0
5	56	1	100.0
6	149	360	76.55
7	256	9900	74.34
8	335	1800	76.55
9	281	3190	82.74
10	511	3460	67.7
11	1191	9910	76.11
12	352	9910	68.58
13	1131	9740	64.6
14	489	9910	95.13
15	1139	9920	93.81

图 3

用例名称	Cost	Time	用例得分
1	4	1	100.0
2	16	1	100.0
3	31	1	100.0
4	62	1	100.0
5	56	1	100.0
6	143	9930	88.31
7	214	9930	87.76
8	267	9930	87.76
9	302	9930	82.56
10	319	9930	84.97
11	1250	9950	82.0
12	286	9930	80.52
13	358	9930	78.11
14	652	9930	88.13
15	1061	9920	86.46

图 4

虽然解的权重已经得到了不小的提升，但是离最优值还是有不小的差距。表格 1 列的是我们算法求出的解与华为提供的每个测试用例的最优解的对比：

测试用例	求出的权重	最优解	偏差
1	4	4	0%
2	16	16	0%
3	31	31	0%
4	62	62	0%
5	56	56	0%
6	143	143	0%
7	214	192	11.4%
8	267	254	5%
9	302	228	32%
10	319	264	20.8%
11	1250	444	181%
12	286	221	29%
13	368	235	56%
14	652	292	123%
15	1061	587	80%

通过表格可以看出，求出的权重与最优解甚至能达到 181%的差距。而这也是蚁群算法的缺陷：容易陷入局部最优解。为了解决这个问题，我们不能再局限于对蚁群算法本身的优化，而是应该引入一个新的算法来改进蚁群算法，下一章介绍的便是如何改进蚁群算法本身。

第四章 混合蚁群算法的改进

通过第二章我们知道，蚁群算法存在易于陷入局部最优解的问题。通过阅读资料[1]和[2]我们发现可以采用“局部搜索”算法来改进蚁群算法。局部搜索可以简单理解为：从一个初始解出发，然后搜索解的邻域，如有更优的解则移动至该解并继续执行搜索，否则返回当前解。更具体地，我们使用的是局部搜索算法里面的 3-opt 算法[4]：3-opt 算法是局部搜索算法中效率最高的 k-opt 邻域算法，其基本过程是：设 T 是一路，产生 3 随机位置 t1、t3、t5，它们的下一个节点分别记为 t2、t4、t6，对应的 3 条边的集合记为{ (t1，t2) ，(t3，t4) ，(t5，t6) } ，该算法断开这三条边试图找到另一个边的集合{ a，b，c} ，使得新产生的回路长度变小，这里是 3 条边则称为 3-opt，如图 5 所示。

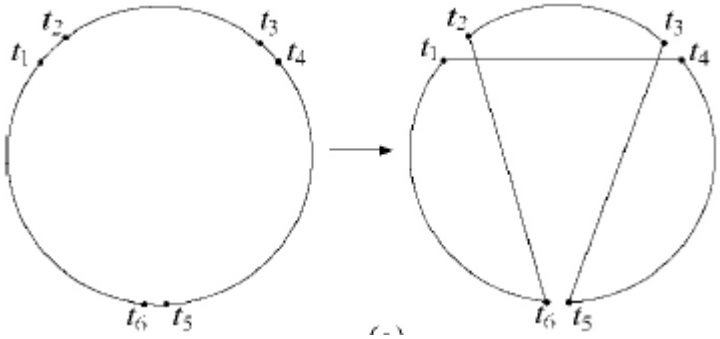

图 5

简单的说，3-opt 算法的思想是：如果某条路径是最优路径，那么该路径里的任意 3 条子路径也应该是最优的。3-opt 算法的时间复杂度是 $O(n^3)$，对于规模较大的图耗时可能会较多。限于篇幅原因，我们不再细讲 3-opt 的具体实现细节，感兴趣的读者可以前往[3]查看。

但是，3-opt 算法是专门为 TSP 问题设计的，并不适用于本题。因此，我们需要将本题转换成 TSP 问题。转换算法如下图所示：

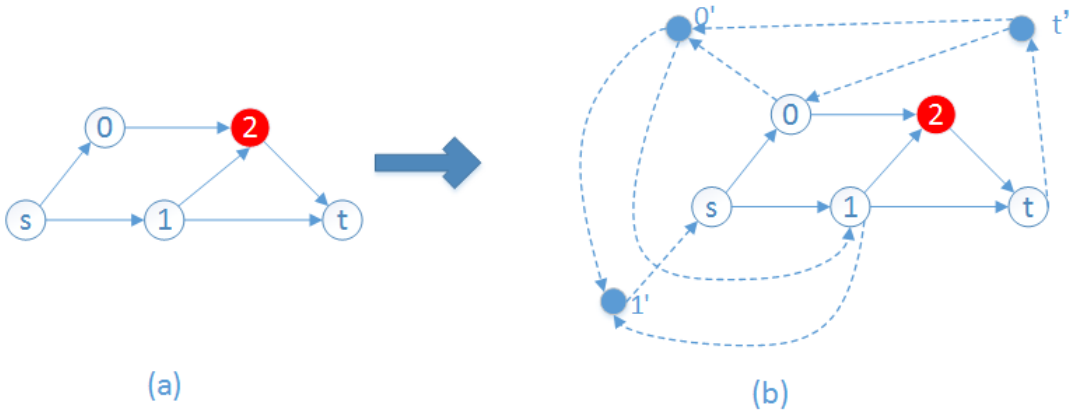

图 6

图 6(a)表示的是必经点最短路径中的一个示例图，其中点 2 是必经点。图 6(b)表示的是(a)经过转换之后对应的 TSP 的图，在图 6(b)中可以构造一条哈密顿回路。图 6(b)中，蓝色的点表示新增的“虚拟节点”。具体转换规则如下所示：

1， 假设有 n 个非必经点，则需要新增 n+1 个虚拟节点，如图 6(b)所示，虚拟节点分别是对应终点的虚拟节点 t’，对应点 0 的虚拟节点 0’，对应点 1 的虚拟节点 1’；
2， 将非必经点和终点连接到与其相对应的虚拟节点上，如图 6(b)中 t 有一条有向边连接到 t’；另外，将最后一个虚拟节点连接到起点，如图 6(b)中，虚拟节点 1’有一条边连接到起点 s。
3， 将每个虚拟节点相连，如图 6(b)中，t’连接到 0’，0’连接到 1’。
4， 每条与虚拟节点相连的边的权重都是-∞（负无穷）。

经过如上转换，我们便将原必经点最短路径问题的图转换成了 TSP 的图。例如，在图 6(a)中，如果有一条路径是: S->1->2->t, 那么经过转换成图 6(b)所示的图之后，原路径就可以是：S->1->2->t->t’->0->0’->1’->S，这样就形成了一条经过所有点的哈密顿回路。这样，我们就可以对转换后的路径应用 3-opt 算法来进行优化。在优化之后，我们将路径转换回本题的路径的方法也很简单：只要忽略路径中的虚拟节点和终点 t 以后的点即可。

相对应的，我们的算法的主要流程也需要增加 3-opt 局部搜索部分，如下所示：

```
while ( !termination_condition() ) {
    construct_solutions();
    local_search();              // add
    update_statistics();
    pheromone_trail_update();
    punish_and_reward();         // add
    search_control_and_statistics();
    n_tours++;
}
```

在增加了 3-opt 局部搜索优化之后，算法的测试结果如下图所示：

用例名称	Cost	Time	用例得分
1	4	1	100.0
2	16	1	100.0
3	31	1	100.0
4	62	1	100.0
5	56	1	100.0
6	143	8000	89.4
7	192	510	96.72
8	256	1240	92.87
9	231	1760	97.11
10	264	2750	98.46
11	446	6900	97.88
12	223	7000	97.5
13	235	6060	97.69
14	567	9920	89.79
15	952	9910	88.44

图 7

其中，算法增加 3-opt 优化前后的对比如下表所示：

表格 1 增加 3-opt 算法前后对比

测试用例	增加 3-opt 前	增加 3-opt 后	权重优化
1	4	4	0%
2	16	16	0%
3	31	31	0%
4	62	62	0%
5	56	56	0%
6	143	143	0%
7	214	192	10%
8	267	256	4%
9	302	231	23%
10	319	264	17%
11	1250	446	64%
12	286	223	22%
13	368	235	36%
14	652	567	13%
15	1061	952	10%

而增加 3-opt 局部搜索优化后，算法所求出的解与最优解的对比如下表所示：

表格 2

测试用例	增加 3-opt 后	最优解	偏差
1	4	4	0%
2	16	16	0%
3	31	31	0%
4	62	62	0%
5	56	56	0%
6	143	143	0%
7	192	192	0%
8	256	254	0.78%
9	231	228	1.3%
10	264	264	0%
11	446	444	0.4%
12	223	221	0.9%
13	235	235	0%
14	567	292	94%
15	952	587	62%

由表格 1，2 可以看出，除了第 14,15 个测试用例由于规模较大（都是 595 个点）而导致求解时间较长，所以优化幅度不大以外，前 13 个测试用例求出的解的质量都得到了大幅的提升。前 13 题求出的解与最优解的偏差从之前的最高 181%下降到了最高 1.3%，不得不说是一个巨大的优化。

最后，我们的算法的总成绩也从增加优化前的 80 多分增加到了 95.6 分（见图 8 所示），成功进入赛区前 32 名。

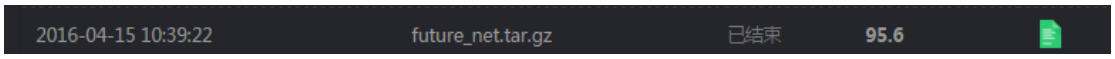

图 8

参考

[1]，多里戈 (Dorigo, Marco); 施蒂茨勒 (Stützle, Thomas) 《蚁群优化（Ant colony optimization）》，清华出版社，2007；

[2]，M. Dorigo , IRIDIA, Vrije Univ., Brussels, Belgium , L. M. Gambardella， “Ant colony system: a cooperative learning approach to the traveling salesman problem”， IEEE Transactions on Evolutionary Computation，1997

[3], https://github.com/whutjs/AntColonySystem

[4], S. Lin., “Computer solutions of the traveling salesman problem,” Bell Syst. J., vol. 44, pp. 2245–2269, 1965.