4.2

状态：状态由轨道块的组合方式决定

初始状态：任选一个工件均可作为初始状态

行动：当前组合工件的扇出与一个的扇入相连，弧形工件需要与弧形工件或者分支工件相连。工件相连时拐角需要对应，连接时可以有[-10,10]度角度误差。

转移模型：行动会产生期待的后果

目标测试：所有零件拼接成铁路，无重叠的轨道

路径耗散：无

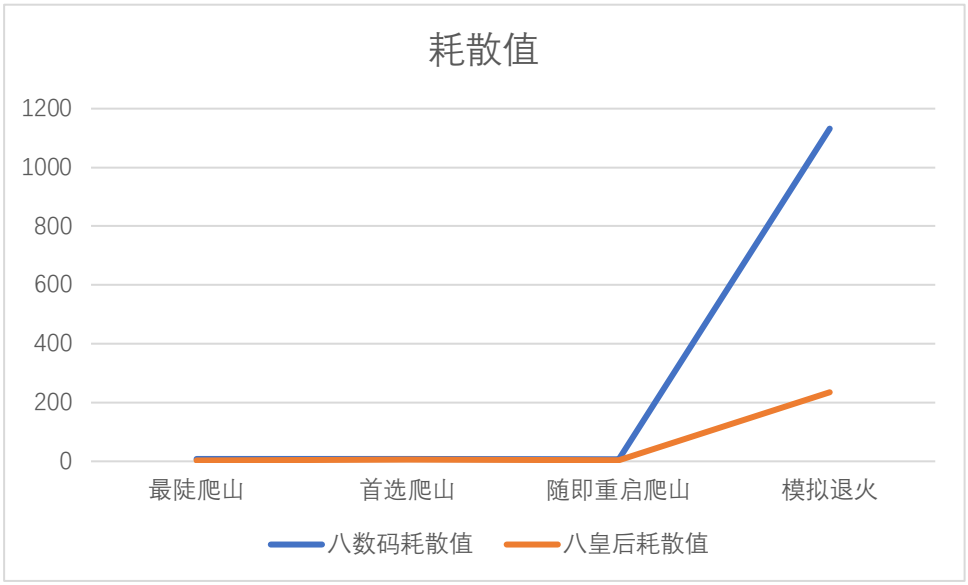
4.4

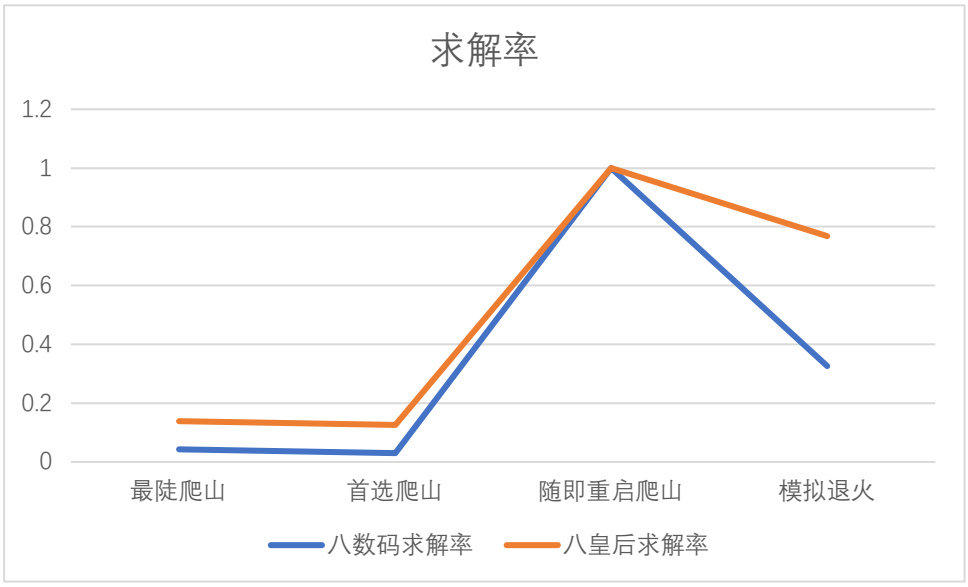
生成八数码问题和八皇后问题各 1000 个，编码求解如下。

八数码问题：

八皇后问题：

四种方法求解八数码问题和八皇后问题，求解率及耗散值如下图





观察上图发现最陡爬山法以及首选爬山法问题求解能力很相似，性能比较低的原因是陷入了局部极值。

模拟退火法相比最陡爬山法以及首选爬山法性能得到了提升，这是因为模拟退火法能够以一定概率跳出局部极值，获得问题的全局极值。

随机重启爬山法的性能最好，这是因为对一个有解问题，随机重启总能够寻找到问题的解。

4.6

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ··· if s_{n-1} then plan_{n-1} else plan_n]
```

图 4.11　不确定性环境生成的与或图的搜索算法。它会返回一个有条件的规划，在所有情况下都可以到达目标状态（[x|l]表示将对象 x 加进表 l 的头）

对书上图 4.11 不确定性环境的与或图搜索算法进行改进。伪代码如下所示：

```
Function and-or-graph-search(problem) returns a conditional plan, or failure
   Or-search(problem.initial-state, problem,[])

Function or search(state, problem, path) returns a conditional plan, or failure
   If problem.goal-test(state) then return the empty plan
   If state is on path then return loop
   Plan-a = none
   For each action in problem.actions(state) do
      Plan = and-search(results(state, action), problem, [state | path])
      If plan != failure then
         If plan is 无环 then return [action | plan]
         Else plan-a = [action | plan]
   If plan-a != none then return plan-a
   Else return failure

Function and-search(state, problem, path) returns a conditional plan ,or failure
   flag = none
   For each si in states do
      Plani = or-search(si, problem, path)
      If plani = failure then return failure
      If plan != loop then flag = false
      Else flag= true
   If not flag then
      Return[if s1 then plan1 else if s2 then plan2 ……..else palnn]
   Return failure

   注：and-search 算法里的 flag 表示是否循环
```

在或搜索中，如果循环到路径上的状态时，就返回一个 loop 信号；在搜索过程中，如果有环，则 plan-a 存储当前 plan 条件下的 action；通过这两种做法，既使得有环规划可以指向规划的早期部分，还能够在找到有环规划后继续寻找无环规划。

附录

Code1.cpp

//八数码问题

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>
#include <vector>
using namespace std;

int direction[4][2] = { {0, 1}, {1, 0}, {0, -1}, {-1, 0} };  // 右下左上
int current[3][3];  // 当前状态
int row_0, col_0;  // 记录0的坐标
int totalTrial;    // 统计移动步数

int Manhattan() {  // 计算曼哈顿距离
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (current[i][j] == 0) continue;
            int row = current[i][j] / 3;
            int col = current[i][j] % 3;
            int distance = abs(row - i) + abs(col - j);
            sum += distance;
        }
    }
    return sum;
}

void print() {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j)
            cout << current[i][j] << " ";
        cout << endl;
    }
    cout << endl;
}

void initial() {
    for (int i = 0; i < 3; ++i) {  // 初始状态为目标状态
        for (int j = 0; j < 3; ++j) {
            current[i][j] = i * 3 + j;
```

```
        }
    }
    row_0 = 0, col_0 = 0;
    int last = -1;   // 上一次移动方向
    for (int i = 0; i < 20; i++) {  // 随机打乱
        bool upset = false;
        while (!upset) {  // 打乱成功才跳出循环
            int dir = rand() % 4;  // 随机选取一个方向
            if (last != -1 && last != dir && abs(last - dir) == 2)  continue;  // 避免
反向走
            int x = row_0 + direction[dir][0];
            int y = col_0 + direction[dir][1];
            if (x >= 0 && x < 3 && y >= 0 && y < 3) {  // 方向可行
                swap(current[row_0][col_0], current[x][y]);  // 交换0和相邻数字的位置
                row_0 = x, col_0 = y;     // 更新0的坐标
                last = dir;               // 更新此次移动方向
                upset = true;             // 标记打乱成功
            }
        }
    }
}

// 判定是否有解
bool check() {
    for (int i = 0; i < 3; ++i) {
        for (int j = 0; j < 3; ++j)
            if (current[i][j] != i * 3 + j)
                return false;
    }
    return true;
}

// 爬山法
bool hillClimbing() {
    for (int trial = 0; trial < 200; trial++) {
        int curManha = Manhattan();  // 当前状态
        int minMan = 99999, minX = 0, minY = 0;
        for (int i = 0; i < 4; i++) {  // 在后继状态中找最小值
            int x = row_0 + direction[i][0];
            int y = col_0 + direction[i][1];
            if (x >= 0 && x < 3 && y >= 0 && y < 3) {  // 方向可行
                swap(current[row_0][col_0], current[x][y]);  // 交换0和相邻位置
                int nextManha = Manhattan();
                if (nextManha < minMan) {  // 获取下一状态的最小值
```

```
                    minMan = nextManha;
                    minX = x, minY = y;
                }
                swap(current[x][y], current[row_0][col_0]);  // 复原0和相邻位置
            }
        }
        if (curManha > minMan) {  // 最小值优于当前状态
            swap(current[row_0][col_0], current[minX][minY]);
            row_0 = minX, col_0 = minY;
        }
        if (check()) {   // 成功找到解
            totalTrial += trial;
            return true;
        }
    }
    return false;
}

// 首选爬山法
bool firstchose() {
    for (int trial = 0; trial < 500; trial++) {
        // 随机选取第一个优于当前状态的下一步
        bool next = false;
        int times = 0;
        while (!next) {
            int dir = rand() % 4;
            int curManha = Manhattan();
            int x = row_0 + direction[dir][0];
            int y = col_0 + direction[dir][1];
            if (x >= 0 && x < 3 && y >= 0 && y < 3) {  // 方向可行
                swap(current[row_0][col_0], current[x][y]);
                int nextManha = Manhattan();
                if (nextManha < curManha) {
                    row_0 = x, col_0 = y;
                    next = true;
                }
                else {
                    swap(current[x][y], current[row_0][col_0]);
                }
            }
            if (++times > 20) break;
        }
        if (check()) {   // 成功找到解
            totalTrial += trial;
```

```
            return true;
        }
    }
    return false;
}

// 模拟退火算法
bool simulated() {
    double temperature = 5;  // 初始温度
    int trial = 0;
    while (temperature > 0.00001) {
        vector<int> v;    // 选出可行的方向
        for (int i = 0; i < 4; i++) {
            int x = row_0 + direction[i][0];
            int y = col_0 + direction[i][1];
            if (x >= 0 && x < 3 && y >= 0 && y < 3) {  // 方向可行
                v.push_back(i);
            }
        }
        int curManha = Manhattan();      // 当前状态的曼哈顿距离之和
        int dir = v[rand() % v.size()];  // 随机选取一个可行方向
        int x = row_0 + direction[dir][0];
        int y = col_0 + direction[dir][1];
        swap(current[row_0][col_0], current[x][y]);  // 交换0和相邻节点的位置
        int nextManha = Manhattan();    // 交换之后的曼哈顿距离之和
        int E = nextManha - curManha;
        if (E < 0) {                    // 下一状态优于当前状态
            row_0 = x, col_0 = y;       // 更新0的位置
            trial++;
        }
        else if (exp((-1) * E / temperature) > ((double)(rand() % 1000) / 1000)) { //
以一定的概率选取
            row_0 = x, col_0 = y;
            trial++;
        }
        else {  // 不成功的话，复原0和相邻节点的位置
            swap(current[x][y], current[row_0][col_0]);
        }
        temperature *= 0.999;        // 温度下降
        if (check()) {   // 成功找到解
            totalTrial += trial;
            return true;
        }
    }
```

```
    return false;
}

// 最陡上升爬山法
int steepestAscent() {
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        initial();
        if (hillClimbing())
            count++;
    }
    return count;
}

// 首选爬山法
int firstChose() {
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        initial();
        if (firstchose())
            count++;
    }
    return count;
}

// 随机重新开始爬山法
int randomRestart() {
    bool find = false;
    while (!find) {
        initial();
        find = hillClimbing();
    }
    return find;
}

// 模拟退火搜索
int simulatedAnnealing() {
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        initial();
        if (simulated())
            count++;
    }
    return count;
```

```
}

int main(int argc, char const* argv[]) {
    srand((int)time(0));

        totalTrial = 0;
        cout << "最陡爬山" << endl;
        int count = steepestAscent();
        cout << "ACC:" << count << "/1000" << endl;
        cout << "耗散:" << totalTrial / count << endl;

        totalTrial = 0;
        cout << "首选爬山" << endl;
        int count3 = firstChose();
        cout << "ACC:" << count3 << "/1000" << endl;
        cout << "耗散:" << totalTrial / count3 << endl;

        totalTrial = 0;
        cout << "随即重启爬山" << endl;
        int count2 = randomRestart();
        cout << "ACC:" << count2 << endl;
        cout << "耗散:" << totalTrial / count2 << endl;


        totalTrial = 0;
        cout << "模拟退火" << endl;
        int count4 = simulatedAnnealing();
        cout << "ACC:" << count4 << "/1000" << endl;
        cout << "耗散:" << totalTrial / count4 << endl;


    return 0;
}
```

Code2.cpp

//八皇后问题

```
#include <iostream>
#include <time.h>
#include <stdlib.h>
#include <algorithm>
#include <cmath>
using namespace std;

int queens[8][8];  // 8*8棋盘
int temp[8][8];
int totalTrial;   // 统计移动步数

// 随机生成初始状态
void initial() {
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; j++) {
            queens[i][j] = 0;
        }
    }
    for (int i = 0; i < 8; i++) {
        int num = rand() % 8;
        queens[i][num] = 1;
    }
}

void print() {
    for (int i = 0; i < 8; ++i) {
        for (int j = 0; j < 8; j++)
            cout << queens[i][j] << " ";
        cout << endl;
    }
}

// 统计在该位置下所有皇后的冲突个数
int findCollision(int row, int col) {
    int count = 0;
    // 该位置为1
    temp[row][col] = 1;
    for (int k = 0; k < 64; k++) {
        if (temp[k / 8][k % 8] == 1) {
            for (int i = 0; i < 8; i++)                              // 同一列
                if (i != k / 8 && temp[i][k % 8] == 1)
```

```
                    count++;
            for (int i = k / 8, j = k % 8; i < 8 && j < 8; i++, j++)    // 右下方
                if (i != k / 8 && temp[i][j] == 1)
                    count++;
            for (int i = k / 8, j = k % 8; i >= 0 && j >= 0; i--, j--)  // 左上方
                if (i != k / 8 && temp[i][j] == 1)
                    count++;
            for (int i = k / 8, j = k % 8; i < 8 && j >= 0; i++, j--)   // 左下方
                if (i != k / 8 && temp[i][j] == 1)
                    count++;
            for (int i = k / 8, j = k % 8; i >= 0 && j < 8; i--, j++)   // 右上方
                if (i != k / 8 && temp[i][j] == 1)
                    count++;
        }
    }
    temp[row][col] = 0;  // 复原位置
    return count / 2;
}

bool check(int h[8][8]) {
    for (int i = 0; i < 8; i++) {
        bool flag = false;
        for (int j = 0; j < 8; j++) {
            if (queens[i][j] == 1 && h[i][j] == 0) { //皇后所在位置没有冲突
                flag = true;
                break;
            }
        }
        if (!flag) { // 皇后所在位置仍有冲突，还需要继续查找
            return false;
        }
    }
    return true;
}

int ct = 0;

// 爬山法
bool hillClimbing() {
    // 尝试次数大于100则判定为无解
    for (int trial = 0; trial <= 100; trial++) {
        // 拷贝原始棋盘数据到temp
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
```

```
                temp[i][j] = queens[i][j];
            }
        }
        int h[8][8];
        int minH = 9999, minX = 0, minY = 0, curState;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                // 在计算h(i, j)之前，对i行所有位置赋值为0
                for (int k = 0; k < 8; k++)
                    temp[i][k] = 0;
                // 查找h(i, j)
                h[i][j] = findCollision(i, j);
                // 当前状态的h值
                if (queens[i][j] == 1) {
                    curState = h[i][j];
                }
                // 先找出冲突个数最小的位置
                if (h[i][j] < minH) {
                    minH = h[i][j];
                    minX = i;
                    minY = j;
                }
                // 计算h(i,j)之后要复原数据，避免计算错误
                for (int k = 0; k < 8; k++)
                    temp[i][k] = queens[i][k];
            }
        }

        // 将皇后放在该行冲突最少的位置处
        if (curState > minH) {
            for (int i = 0; i < 8; i++)
                queens[minX][i] = 0;
            queens[minX][minY] = 1;
        }

        // 判断是否找到解，有解则返回值为真
        if (check(h)) {
            totalTrial += trial;
            return true;
        }
    }
    return false;
}
```

```
// 首选爬山法
bool firstchose() {
    // 尝试次数大于100则判定为无解
    for (int trial = 0; trial <= 100; trial++) {
        // 拷贝原始棋盘数据到temp
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                temp[i][j] = queens[i][j];
            }
        }
        int h[8][8], curState;

        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                // 在计算h(i, j)之前，对i行所有位置赋值为0
                for (int k = 0; k < 8; k++)
                    temp[i][k] = 0;
                // 查找h(i, j)
                h[i][j] = findCollision(i, j);
                // 当前状态的h值
                if (queens[i][j] == 1) {
                    curState = h[i][j];
                }
                // 计算h(i,j)之后要复原数据，避免计算错误
                for (int k = 0; k < 8; k++)
                    temp[i][k] = queens[i][k];
            }
        }

        // 随机选取第一个优于当前状态的下一状态
        bool better = false;
        int next, nextState, times = 0;
        while (!better) {
            next = rand() % 64;
            nextState = h[next / 8][next % 8];
            if (nextState < curState) {
                better = true;
            }
            if (++times > 100) break;
        }

        if (better) {
            for (int i = 0; i < 8; i++)
                queens[next / 8][i] = 0;
```

```
            queens[next / 8][next % 8] = 1;  // 放置皇后
        }

        // 判断是否找到解，有解则返回值为真
        if (check(h)) {
            totalTrial += trial;
            return true;
        }
    }
    return false;
}

// 模拟退火搜索
bool simulated() {
    double temperature = 5;
    int trial = 0;
    while (temperature > 0.00001) {
        // 拷贝原始棋盘数据到temp
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                temp[i][j] = queens[i][j];
            }
        }
        int h[8][8], curState;
        for (int i = 0; i < 8; i++) {
            for (int j = 0; j < 8; j++) {
                // 在计算h(i, j)之前，对i行所有位置赋值为0
                for (int k = 0; k < 8; k++)
                    temp[i][k] = 0;
                // 查找h(i, j)
                h[i][j] = findCollision(i, j);
                // 当前状态的h值
                if (queens[i][j] == 1) {
                    curState = h[i][j];
                }
                // 计算h(i,j)之后要复原数据，避免计算错误
                for (int k = 0; k < 8; k++)
                    temp[i][k] = queens[i][k];
            }
        }

        // 随机选取一个下一状态
        bool better = false;
        int next, nextState, times = 0;
```

```
            next = rand() % 64;
            nextState = h[next / 8][next % 8];
            int E = nextState - curState;
            if (E < 0) {
                better = true;
            }
            else if (exp((-1) * E / temperature) > ((double)(rand() % 1000) / 1000)) {
                better = true;
            }

            if (better) {
                for (int i = 0; i < 8; i++)
                    queens[next / 8][i] = 0;
                queens[next / 8][next % 8] = 1;  // 放置皇后
                trial++;
            }


            // 判断是否找到解，有解则返回值为真
            if (check(h)) {
                totalTrial += trial;
                return true;
            }

            temperature *= 0.99;
        }
        return false;
    }

    // 最陡上升爬山法
    int steepestAscent() {
        int count = 0;
        for (int i = 0; i < 1000; i++) {
            initial();
            if (hillClimbing())
                count++;
        }
        return count;
    }

    // 首选爬山法
    int firstChose() {
        int count = 0;
```

```
    for (int i = 0; i < 1000; i++) {
        initial();
        if (firstchose())
            count++;
    }
    return count;
}

// 随机重新开始爬山法
int randomRestart() {
    bool find = false;
    while (!find) {
        initial();
        find = hillClimbing();
    }
    return find;
}

// 模拟退火搜索
int simulatedAnnealing() {
    int count = 0;
    for (int i = 0; i < 1000; i++) {
        initial();
        if (simulated())
            count++;
    }
    return count;
}

int main(int argc, char const* argv[]) {
    srand((int)time(0));

    totalTrial = 0;
    cout << "最陡爬山" << endl;
    int count = steepestAscent();
    cout << "ACC:" << count << "/1000" << endl;
    cout << "耗散:" << totalTrial / count << endl;

    totalTrial = 0;
    cout << "首选爬山" << endl;
    int count3 = firstChose();
    cout << "ACC:" << count3 << "/1000" << endl;
    cout << "耗散:" << totalTrial / count3 << endl;
```

```
    totalTrial = 0;
    cout << "随即重启爬山" << endl;
    int count2 = randomRestart();
    cout << "ACC:" << count2 << endl;
    cout << "耗散:" << totalTrial / count2 << endl;


    totalTrial = 0;
    cout << "模拟退火" << endl;
    int count4 = simulatedAnnealing();
    cout << "ACC:" << count4 << "/1000" << endl;
    cout << "耗散:" << totalTrial / count4 << endl;

    return 0;
}
```