中山大学数据科学与计算机学院

计算机科学与技术专业-人工智能

本科生实验报告

（2018-2019 学年秋季学期）

学　　号：16337113

姓　　名：劳马东

教学班级：教务 2 班

专　　业：超算

一、实验题目

1、 使用 A* 与 IDA* 算法解决 15-Puzzle 问题，启发式函数可以自己选取，最好多尝试不同的启发式函数；

2、 代码要求使用 python 或者 C++；

3、 报告要求

(a) 报告中需要包含对两个算法的原理解释

(b) 需要包含性能和结果的对比和分析

(c) 如果使用了多种启发式函数，最好进行对比和分析

(d) 需要在报告中分情况分析估价值和真实值的差距会造成算法性能差距的原因

二、实验内容

（一）算法原理

1、 拼图问题的形式化

（1） 问题定义

- 初始状态：将整个拼图建模为一个状态，初始的整个拼图为初始状态
- 行动：交换空白块及其相邻块的位置
- 状态空间：所有可能的拼图状态
- 目标测试：拼图从上到下、从左到右依次为 1、2、3...、14、15、0
- 路径耗散：交换的次数

（2） 问题的解：从最初拼图到目标，交换的非空白块的序列（用空白块上的数字表示）

2、 启发式搜索

无信息搜索最大的问题是将边界节点一视同仁，不管该节点到达目标节点的代价，因此常常无法找到最优解，需要搜索的状态空间也很大。

启发式搜索利用问题所拥有的启发信息，设计估价函数来引导搜索的方向，从而减少搜索范围。也就是说，每个边界节点到目标节点的代价是可以预计的，这样就能优先探索那些最有可能到达目标状态的节点。

通常估价函数 $f(x)$ 由两部分组成：

$$f(x) = g(x) + h(x) \tag{1}$$

其中 $g(x)$ 是从初始节点到节点 x 付出的实际代价，如图**??**实线，而 $h(x)$ 是从节点 x 到目标节点的估计代价，如图**??**虚线。$g(x)$ 在探索到 x 时就已经确定了，而 $h(x)$ 则是人为设计的一个可采纳启发函数，它的好坏直接影响到问题求解时间。一般希望 $h(x)$ 与 x 到目标节点的实际代价越接近越好，这样每次探索基本上都是沿着最优路径。

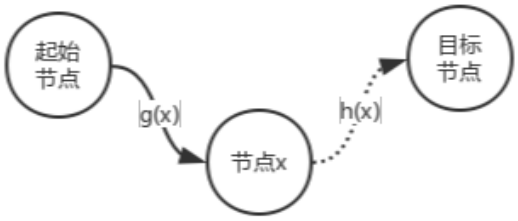


图 1. g(x) 与 h(x) 示意图

3、 A* 算法

（1） 算法描述

A* 算法维护两个列表，分别记为开启列表和关闭列表。开启列表存放边界节点，关闭列表存放所有已求出最优路径的节点。值得一提的是，A* 算法探索到一个节点 x 时，就找到了从起始节点到 x 的最优路径。

一个节点 x 的相邻节点是那些可以通过交换空白块一步到达的状态。扩展 x 的相邻节点 n 时，需要将 $g(n)$ 最小的 n 放入开启列表，因为 n 可以由几个节点扩展。一种做法是直接更新开启列表中 n 的 $g(n)$ 值，另一种做法是在开启列表中保留 n 的多个副本，只访问第一次遇到的。

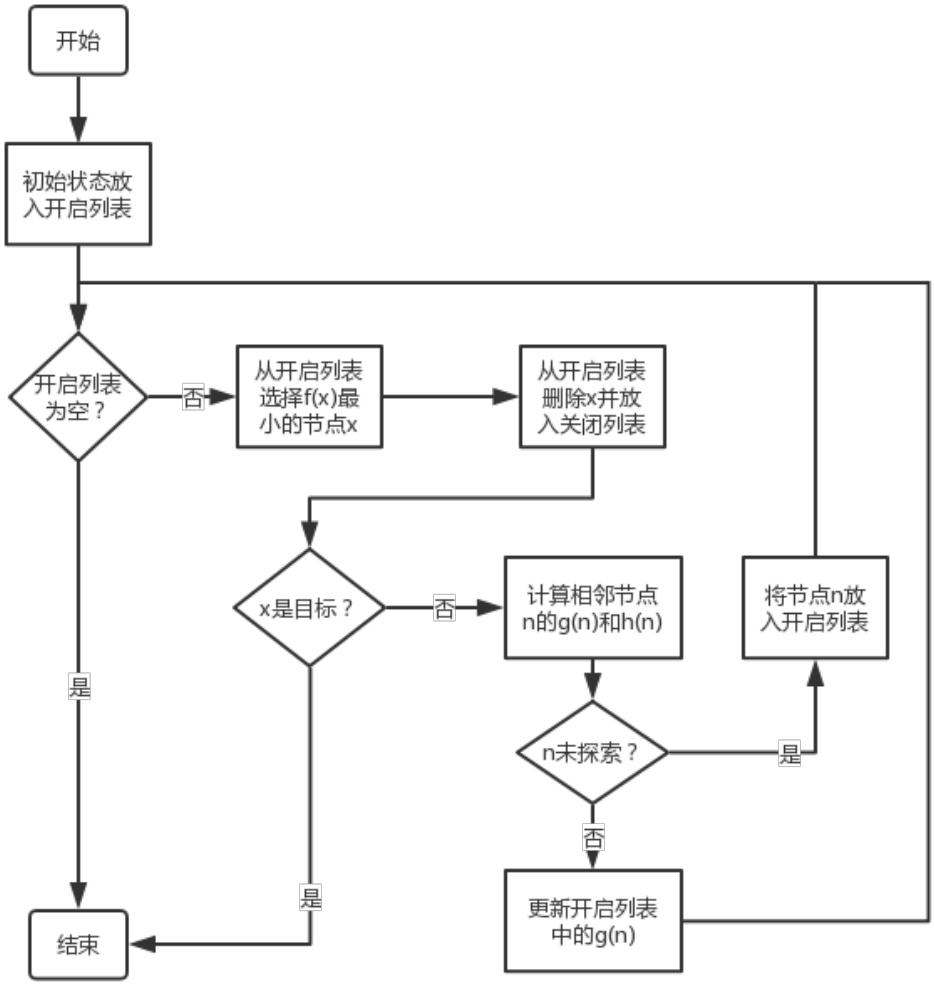


图 2. A* 算法总体流程

（2） 优缺点

优点是时间复杂度与 BFS 相当；缺点是空间复杂度是指数级的，最坏情况下，在无界搜索空间中，扩展的节点数与解的深度 d 呈指数关系，为 $O(b^d)$，其中 b 是后继节点的数目。

4、 IDA* 算法

许多 15 puzzle 问题都无法通过 A* 算法来求解，因为它需要生成太多的新状态。假设用二维数组来表示拼图，那么每个新状态都是整个二维数组的拷贝，这无疑需要消耗很多内存来保存开启列表和关闭列表。

IDA*(Iterative-Deepening-A*) 克服了 A* 算法需要很大内存的缺点。它使用的是受限的 DFS 而不是 BFS，在探索完一个节点后回溯，就能恢复拼图的状态，供下一个邻居使用，因此不需要拷贝二维数组。IDA* 描述如下：

（1） 设置初始阈值为初始状态到目标状态的估计值；

（2） 执行一次 DFS，对那些 $f(x)$ 超过阈值的节点进行剪纸，即不再探索；

（3） 如果未找到解，增大阈值，回到步骤（2）。

算法 1 IDA* 的 DFS 算法

输入: *cur_state* 当前拼图状态，*cur_cost* 状态的实际代价，*threshold* 迭代阈值，*Goal*? 目标测试

输出: 是否找到解，找到返回 success, 找不到返回 failure

```
function DFS(cur_state, cur_cost, threshold, Goal?)
    if cur_cost+h(cur_state) > threshold then
        return failure
    end if
    if Goal?(cur_state) then
        return success
    end if
    space_i, space_j ← cur_state.space_position()
    for i, j ∈ successors(cur_state) do
        cur_state.swap(i, j)
        if DFS(cur_state, cur_cost + 1, threshold, Goal?) == success then
            return success
        end if
        // 回溯
        cur_state.swap(space_i, space_j)
    end for
    return failure
end function
```

5、 启发函数的设计

启发函数最大的作用在于剪枝，它直接关系到需要探索的状态的多少。一个好的启发函数应该使所有节点的预计距离与真实距离接近，这样每次探索时基本都在选择最优路径上的节点。在实际应用中，15 puzzle 问题主要有以下几种启发函数：

（1） 曼哈顿距离：计算每个数码与目标位置的坐标各个维度的差的绝对值，即：

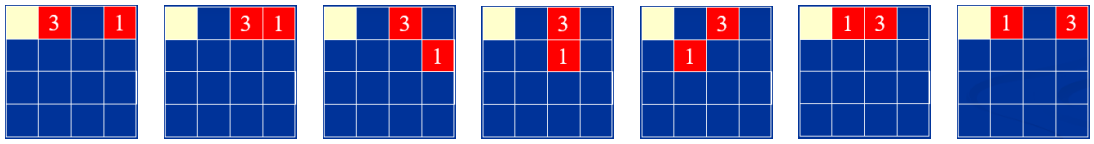
$$Manhattan(x, y) = \sum_{i=1}^{d} abs(x_i - y_i) \tag{2}$$

其中 d 是坐标维度。

（2） 曼哈顿距离 + 线性冲突（linear conflict）

Definition 1 线性冲突 假设数码 t_j 和 $t_k(t_j < t_k)$ 在同一行或同一列，并且它们的目标位置也在这一行或这一列。如果现在 t_j 在 t_k 的右边或下面，则它们是线性冲突的。

曼哈顿距离忽略了一个事实——数码 A 要移动到数码 B 的位置，数码 B 就需要先移开。例如图3(a)到图3(g)的曼哈顿距离是 4，而实际上从图3(a)到图3(g)需要经过至少 6 步，因为数码 1 要先往下移动，给数码 3 “腾出”位置。



(a) 起始 (b) 第一步 (c) 第二步 (d) 第三步 (e) 第四步 (f) 第五步 (g) 目标

啦啦啦

啦啦啦

算法 2 计算数码 x 线性冲突距离

输入: 拼图 $puzzle$，x 的当前坐标 (i, j)，拼图大小 n

```
function LINEAR_CONFLICT(puzzle, i, j, n)
    x ← puzzle[i][j]
    goal_i, goal_j ← goal_pos(x)
    count ← 0
    // 每个元素只检查它后面的元素
    for k ∈ [j + 1, n) do
        y ← puzzle[i][k]
        // x > y 是产生冲突的前提，x-y<n 表明可能在目标状态的同一行
        if y != 0 and x > y and x − y < n then
            y_goal_i, y_goal_j ← goal_pos(y)
            if goal_i == y_goal_i then
                count ← count + 1
            end if
        end if
    end for
    同理，在列上重复以上循环
    return 2 * count
end function
```

（二）关键代码

1、 拼图压缩

正常情况下，如果使用整形二维数组存储拼图，假设每个整数的大小是 4 个字节，那么表示 15 puzzle 就需要 $16 \times 4 = 64$ 个字节。然而，15 puzzle 中的数字是 0-15，使用 4 位来表示一个数码就已经足够，这样总

共需要 8 个字节，只有原来的 $\frac{1}{8}$。如代码清单1，使用 64 位整形（___int64）来表示拼图，位运算实现取数和存数操作。

Code Listing 1. 拼图压缩

```
// 用于表示数码的位数，0-15一个数字只需4位
#define BITWISE 4
// 1111，用于取低4位
#define LOWERBIT 15
class Puzzle {
  __int64 _puzzle;          // 16个数刚好用去64位
public:
  int space_i, space_j;     // 空白块的位置
  const int dim_size;       // 拼图的大小n

  Puzzle(int numbers) : dim_size(sqrt(numbers + 1) + 0.5), _puzzle(0) { }
  // 实现类似于二维数组下标索引的get、set接口
  int get(int i, int j) const {
      // 计算偏移量，与位数相乘，得到（i,j）元素最低位的地址
      i = (i * dim_size + j) * BITWISE;
      // 右移，取低4位就是（i,j）元素
      return (_puzzle >> i) & LOWERBIT;
  }
  void set(int i, int j, int x) {
      __int64 x64 = LOWERBIT;;
      int offset = (i * dim_size + j) * BITWISE;
      _puzzle &= ~(x64 << offset);     // 先将（i,j）元素置为0
      x64 = x;
      _puzzle |= (x64 << offset);      // 左移取或就能把（i,j）置为x
      if (x == 0) {
          space_i = i;
          space_j = j;
      }
  }
  // 移动（i,j）位
  int swap(int i, int j) {
      int exchange = get(i, j);
      set(space_i, space_j, exchange);
      set(i, j, 0);
      return exchange;
  }
}
```

2、 A* 算法

Code Listing 2. A*

```
template <int numbers>
void astar_search(const Puzzle<numbers>& start, Heuristic<numbers>* h,
                  vector<int>& path) {
  ...
  auto* start_node = new node_type(start, h);
  open_set.push(start_node);       // open_set是一个极小堆，按f(x)排序
  while (!open_set.empty()) {
      node_type* node = open_set.top();    // 取出f(x)最小的一个节点
      open_set.pop();
      // 开启集合中可能有node的多个副本，访问最小的一个
      if (can_visit(visited, node->puzzle, node->cost)) {
          if (node->puzzle == node->puzzle.goal)
              break;
          visited[node->puzzle] = node->cost;
          // dy、dx组合成四个方向，返回空白块四个方向邻居的坐标
          node->puzzle.neighbors(dy, dx, neighbors);
          for (auto &neighbor : neighbors) {
              if (neighbor.first > -1 && neighbor.second > -1) {
                  // 生成一个新的相邻节点，并移动位数码
                  auto *n = new node_type(node,neighbor.first,neighbor.second);
                  if (can_visit(visited, n->puzzle, n->cost))
                      open_set.push(n);
              }
          }
      }
  }
  ...
}
```

3、 IDA* 算法

Code Listing 3. IDA*-DFS

```
// prev_dir记录上一次移动的方向
template <int numbers>
bool dfs(Puzzle<numbers>& node, int cost, int estimate,
        int limit, int prev_dir, Heuristic<numbers>* h, vector<int>& path) {
    if (node == node.goal) return true;
    if (cost + estimate > limit) return false;

    int space_i = node.space_i, space_j = node.space_j;
    int exchange;
    pair<int, int> neighbors[4];

    node.neighbors(dy, dx, neighbors);    // 获取空白块四个方向邻居的坐标
    for (int i = 0; i < 4; i++) {
        if (i == (prev_dir ^ 1))    // 0上1下2左3右，不走回头
            continue;

        int ni = neighbors[i].first, nj = neighbors[i].second;
        if (ni > -1 && nj > -1) {
            exchange = node.get(ni, nj);
            // 从node移动（ni,nj）块变成相邻节点的相对距离
            int relative = h->relative_distance(node, ni, nj);
            path.push_back(exchange);
            // 更新g(x)和h(x)继续dfs，找到解则返回
            if (dfs(node, cost + 1, estimate + relative, limit, i, h, path))
                return true;
            // 找不到解需要回溯，恢复拼图，去掉白探索的路径
            path.pop_back();
            node.swap(space_i, space_j);
        }
    }
    return false;
}
```

4、 线性冲突相对距离

Code Listing 4. 线性冲突相对距离

```
int relative_distance(Puzzle<numbers>& b, int i, int j) {
    if (i == space_i) {
        d -= linear_conflict(b, i, j);
        // 第j列、第space_j列在（i,j）块和空白块前的元素
        for (int k = 0; k < i; ++k)
            d -= linear_conflict(b, k, j) + linear_conflict(b, k, space_j);
        // 第i行在（i,j）块和空白块前的元素
        for (int k = 0; k < j && k < space_j; ++k)
            d -= linear_conflict(b, i, k);
        b.swap(i, j);
        d += linear_conflict(b, space_i, space_j);
        for (int k = 0; k < i; ++k)
            d += linear_conflict(b, k, j) + linear_conflict(b, k, space_j);
        for (int k = 0; k < j && k < space_j; ++k)
            d += linear_conflict(b, i, k);
    else {    // 列相同同理
      ...
    }
    return d;
}
```

三、实验结果及分析

（一）实验结果展示

用小样例说明算法正确性。

（二）评测指标展示及分析

分析并说出自己的思考。

表 1. 运行时间（/秒）——相对距离

搜索策略	样例 1	样例 2	样例 3	样例 4
A*(Manhattan)	634	15	918	46
A*(Manhattan+linear conflict)	128	2	357	10
IDA*(Manhattan)	452	2	784	11
IDA*(Manhattan+linear conflict)	52	1	200	6

表 2. 运行时间（/秒）——绝对距离

搜索策略	样例 1	样例 2	样例 3	样例 4
IDA*(Manhattan)	1374	8	2364	32
IDA*(Manhattan+linear conflict)	135	5	505	15

四、思考题