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1实线，而 $h(x)$ 是从节点 x 到目标节点的估计代价，如图1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。

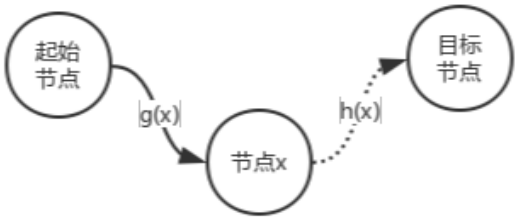


图 1. g(x) 与 h(x) 示意图

3、 A* 算法

（1） 算法描述

A* 算法是升级版的 BFS 算法，它每次边界节点中选出 $f(x)$ 最小的节点扩展，计算邻节点的 $g(x)$ 和 $h(x)$ 值存入边界中。它维护两个列表，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。

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。

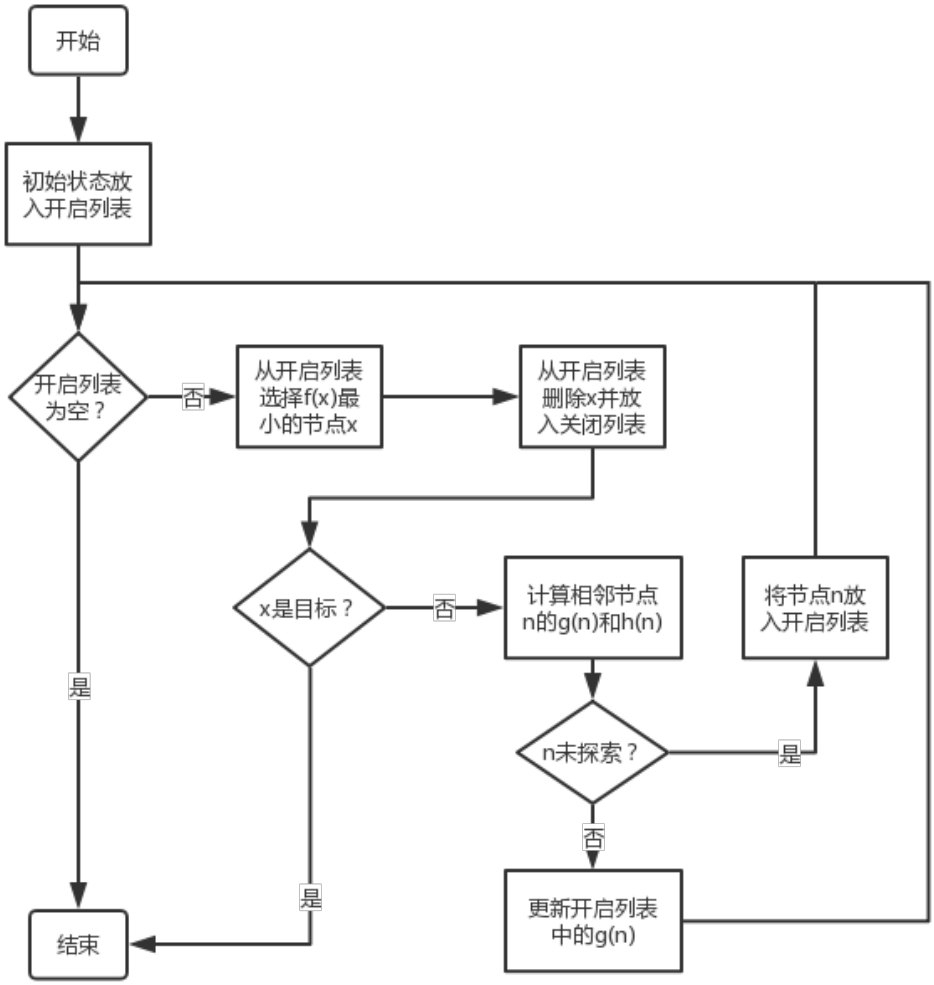


图 2. A* 算法总体流程

（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：

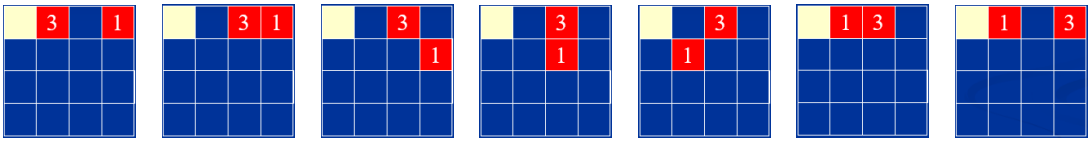
$$Manhattan(x, y) = \sum_{i=1}^{d} abs(x_i - y_i) \tag{2}$$

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

要计算线性冲突距离，就需要统计拼图中有几对线性冲突的数码。最终的距离是对数的 2 倍，因为每对线性冲突的数码，想要移动到目标位置，就需要其中一个先移动到其他行，等另一个移动完成后，它再移动回原来的行，这个过程至少需要额外的 2 步。

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（BITWISE）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 位无符号整形（uint_fast64_t）来表示拼图，位运算实现取数和存数操作。

```
// 1111, 用于取低4位
#define LOWERBIT 15
class Puzzle {
  uint_fast64_t _puzzle;          // 16个数刚好用去64位
public:
  int space_i, space_j;     // 空白块的位置
  static const Puzzle goal(0x0FEDCBA987654321ULL);

  explicit Puzzle(uint_fast64_t puzzle = 0) : _puzzle(puzzle) { }
  // 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 * 4 + 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) {
      uint_fast64_t x64 = LOWERBIT;;
      int offset = (i * 4 + 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
  void swap(int i, int j) {
      set(space_i, space_j, get(i, j));
      set(i, j, 0);
  }
}
```

代码清单 1. 拼图压缩

2、 A* 算法

代码清单2维护了两个列表，一个是 *open_set* 优先队列，相当于开启列表，它存储所有未找到最优路径的节点；另一个是*close_set* 哈希表，由于拼图是一个 64 位整数，哈希的键就是这个整数的哈希值，比康托展开快数倍。在探索一个节点时，先用 *can_visit* 函数检测节点是否在关闭列表或找到一个更短的最优路径，并将节点和当前的实际代价存入关闭列表 *close_set*；扩展节点时也用 *can_visit* 检测从而把具有更小实际代价的邻居节点放入开启列表 *open_set*。

理论上，第一次探索一个节点时，就找到了从起始节点到它的最优路径，因此探索节点时 *can_visit* 检测似乎没必要。但是在实际运行程序时发现如果不检测就找不到最优路径。经过思考，认为是当状态空间存在环路，即一种拼图 X 可以从多种不同的拼图得来，这样当 A* 算法在非最优路径上探索到 X 时，会认为找到了最优路径，而实际上回溯再次探索到 X 时的花费可能更少。

```
void astar_search(const Puzzle& start, Heuristic* 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(close\_set, node->puzzle, node->cost)) {
          if (node->puzzle == node->puzzle.goal)
              break;
          close\_set[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(close\_set, n->puzzle, n->cost))
                      open_set.push(n);
              }
          }
      }
  }
  ...
}
```

代码清单 2. A*

3、 IDA* 算法

迭代加深的 A* 算法最关键的部分就是 DFS。在每次递归，需要保证算法不会走回头路。也就是说，如果上一步交换空白块右边的块 B，这样 B 就在空白块左边了，那么下一步就不应该交换左边的 B。

代码清单3使用了一个小技巧——如果当前方向 i 等于前一个方向 $prev\_dir \oplus 1$，那么它们就是相反的方向。这是因为 dy 和 dx 方向数组被设计成上、下、左、右的顺序，容易验证 $0 == 1 \oplus 1$、$1 == 0 \oplus 1$，2、3 同理。

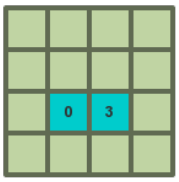
DFS 的重中之重是回溯，交换了拼图 $node$ 后，往下递归如果没有找到解，就需要换回来恢复原本的拼图，以供下一个邻居交换。

```
// prev_dir记录上一次移动的方向
bool dfs(Puzzle& node, int cost, int estimate,
         int limit, int prev_dir, Heuristic* 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;
}
```

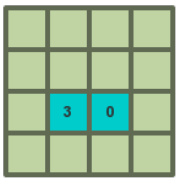
代码清单 3. IDA*-DFS

4、 线性冲突相对距离

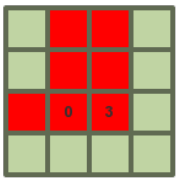
要计算相对距离，一个很自然的问题是，交换之后哪些数码被影响了？考虑图3(h)到图3(i)的情况，现在想要交换数码 3 得到新的邻居节点。显然，0 和 3 数码所在的位置必受影响。此外，由于计算一个数码的线性冲突个数需要遍历同一行、列后面的元素，因此数码 0 和 3同一行、列前面的块也受影响。如图3(j)红色的块。



(h) node



(i) 相邻节点



(j) 受影响的块

```
int lc_relative_distance(Puzzle& b, int i, int j) {
    int d = 0;
    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;
}
```

代码清单 4. 线性冲突相对距离

三、创新点 & 优化

1、 **优化拼图矩阵的拷贝**

对拼图矩阵进行压缩，用一个 64 位整数来表示整个拼图而不是 4×4 整数矩阵；并且直接对这个整数哈希存入关闭列表，而且比康托展开快。

2、 **优化距离计算过程**

使用相对距离而不是绝对距离，即计算一次绝对距离，之后从一种状态到另一种状态的转换只需重新计算那些受影响的块，其他块的启发距离不变；

3、 **使用线性冲突（linear conflict）启发函数**

Manhattan、Minkowski、Dijkstra 三种启发函数都过小地估计了节点的距离（依次减小），通过在 Manhattan 距离上加上线性冲突，更接近真实距离。

四、实验结果及分析

实验中使用了两种算法（A* 和 IDA*）、两种关闭列表（字典和哈希表）、两种启发函数（Manhattan 和 Manhattan+Linear Conflict）、两种距离计算方式（相对和绝对）。表1、2分别列出了使用相对和绝对距离时算法的运行时间。

首先，四个样例中样例 2、4、1、3 的求解时间依次递增，也就是说算法的运行时间和最优解的步数是正相关的。假设启发函数能使节点的估计代价接近真实代价，在 A* 算法中，这是因为 BFS 的探索过程会一层一层探索完那些估计代价小的节点，直至估计代价达到最优路径真实代价；在 IDA* 算法中，阈值的迭代也会达到最优路径的真实代价。

其次，Manhattan+Linear Conflict 启发函数比 Manhattan 启发函数更优。因为前者在后者的基础上增加了，而且是可采纳的，这样它的估计值就比后者更接近真实距离，进而减少了需要探索的状态空间的大小，即探索的过程更偏向于最优路径。

再次，IDA* 算法比 A* 算法更优。在时间复杂度上，IDA* 与 A* 相当，这类似于迭代加深搜索和 BFS。但是，IDA* 最显著的优点是它不需要对状态进行拷贝，而 A* 算法每形成一个新状态都要拷贝整个拼图，这大大减少了 IDA* 算法的求解时间。

另外，A* 算法的关闭列表使用哈希表比使用字典更快（将近一半）。主要数据结构本身复杂度的问题，因为拼图已经做了压缩，字典是按拼图的字典序（实际上是 64 位整数的大小）存储，哈希表存储的是这个 64 位整数的哈希值，两者在数据拷贝方面相差无几。但是，使用拼图的哈希值存入关闭列表这种思想是很宝贵的，例如未对拼图压缩时，康托展开的哈希方法使关闭列表从矩阵的数组或字典变成哈希表，不仅省去拼图的拷贝过程，还大大减少拼图检索时间。

最后，相对距离计算方法比绝对距离更优。这毋庸置疑，绝对距离每次都遍历整个拼图重新计算每个位置的距离，而相对距离只重新计算那些交换后受影响的位置，至少可以保证比遍历所有位置要少计算。

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)	650.09	15.95	969.81	46.43
A*(Manhattan+hashmap)	304.53	8.47	411.39	22.04
A*(Manhattan+linear conflict)	160.85	2.29	207.85	14.4
A*(Manhattan+linear conflict+hashmap)	84.52	1.72	99.96	8.21
IDA*(Manhattan)	449.57	2.75	778.88	10.49
IDA*(Manhattan+linear conflict)	81.75	3.04	304.83	9.55

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)	1371.57	8.42	2362.81	31.91
IDA*(Manhattan+linear conflict)	137.1	5.27	509.78	15.9

五、思考题

（一）估价值和真实值的差距会造成算法性能差距的原因？

不妨假设两种算法需要探索的状态空间大小都是 b^d，其中 d 是最优路径的深度。一个好的启发函数能对这 b^d 个节点起到很好的剪枝作用。对于 A* 算法来说，它贪婪地选择探索估计代价最小的节点，如果估计值接近真实值，那么算法每次都能选择最优路径上的节点来探索，直到找到目标节点后返回，这样就只探索了 d 个节点；对于 IDA* 算法来说，如果估计值接近真实值，阈值的迭代次数就会少很多（极端情况下估计值等于真实值时，只需迭代一次就能找到解）。