School of Data and Computer Science, Sun Yat-sen University

E2 15-Puzzle Problem (IDA*)

18364066 Yanzuo Lu

September 11, 2020

Contents

1 IDA* Algorithm

1.1 Description

Iterative deepening A* (IDA*) was first described by Richard Korf in 1985, which is a graph traversal and path search algorithm that can find the shortest path between a designated start node and any member of a set of goal nodes in a weighted graph.

It is a variant of **iterative deepening depth-first search** that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the **A* search algorithm**.

Since it is a depth-first search algorithm, its memory usage is lower than in A*, but unlike ordinary iterative deepening search, it concentrates on exploring the most promising nodes and thus does not go to the same depth everywhere in the search tree.

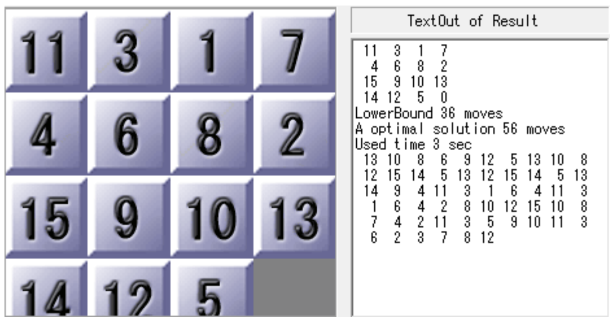
Iterative-deepening-A* works as follows: at each iteration, perform a depth-first search, cutting off a branch when its total cost $f(n) = g(n) + h(n)$ exceeds a given threshold. This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm. At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

1.2 Pseudocode

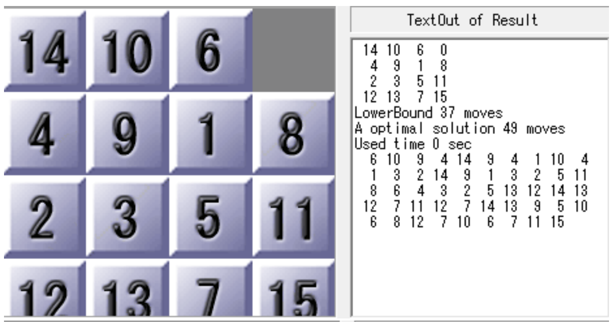
```
path              current search path (acts like a stack)
node              current node (last node in current path)
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function, expand nodes ordered by g + h(node)
ida_star(root)    return either NOT_FOUND or a pair with the best path and its cost

procedure ida_star(root)
  bound := h(root)
  path := [root]
  loop
    t := search(path, 0, bound)
    if t = FOUND then return (path, bound)
    if t = ∞ then return NOT_FOUND
    bound := t
  end loop
end procedure

function search(path, g, bound)
  node := path.last
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min := ∞
  for succ in successors(node) do
    if succ not in path then
      path.push(succ)
      t := search(path, g + cost(node, succ), bound)
      if t = FOUND then return FOUND
      if t < min then min := t
      path.pop()
    end if
  end for
  return min
end function
```

2 Tasks

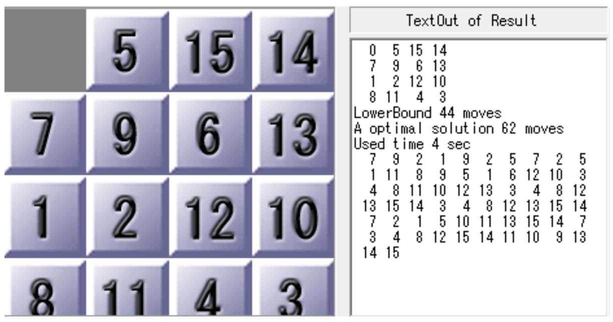
- Please solve 15-Puzzle problem by using IDA* (Python or C++). You can use one of the two commonly used heuristic functions: h1 = the number of misplaced tiles. h2 = the sum of the distances of the tiles from their goal positions.
- Here are 4 test cases for you to verify your algorithm correctness. You can also play this game (`15puzzle.exe`) for more information.



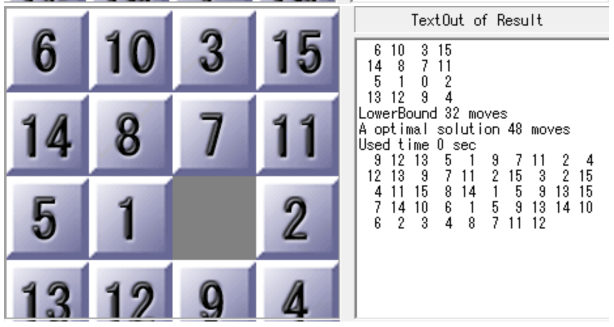
```
TextOut of Result
 11  3  1  7
  4  6  8  2
 15  9 10 13
 14 12  5  0
LowerBound 36 moves
A optimal solution 56 moves
Used time 3 sec
 13 10  8  6  9 12  5 13 10  8
 12 15 14  5 13 12 15 14  5 13
 14  9  4 11  3  1  6  4 11  3
  1  6  4  2  8 10 12 15 10  8
  7  4  2 11  3  5  9 10 11  3
  6  2  3  7  8 12
```



```
TextOut of Result
 14 10  6  0
  4  9  1  8
  2  3  5 11
 12 13  7 15
LowerBound 37 moves
A optimal solution 49 moves
Used time 0 sec
  6 10  9  4 14  9  4  1 10  4
  1  3  2 14  9  1  3  2  5 11
  8  6  4  3  2  5 13 12 14 13
 12  7 11 12  7 14 13  9  5 10
  6  8 12  7 10  6  7 11 15
```



```
TextOut of Result
  0  5 15 14
  7  9  6 13
  1  2 12 10
  8 11  4  3
LowerBound 44 moves
A optimal solution 62 moves
Used time 4 sec
  7  9  2  1  9  2  5  7  2  5
  1 11  8  9  5  1  6 12 10  3
  4  8 11 10 12 13  3  4  8 12
 13 15 14  3  4  8 12 13 15 14
  7  2  1  5 10 11 13 15 14  7
  3  4  8 12 15 14 11 10  9 13
 14 15
```



```
TextOut of Result
  6 10  3 15
 14  8  7 11
  5  1  0  2
 13 12  9  4
LowerBound 32 moves
A optimal solution 48 moves
Used time 0 sec
  9 12 13  5  1  9  7 11  2  4
 12 13  9  7 11  2 15  3  2 15
  4 11 15  8 14  1  5  9 13 15
  7 14 10  6  1  5  9 13 14 10
  6  2  3  4  8  7 11 12
```

- Please send `E02_YourNumber.pdf` to `ai_2020@foxmail.com`, you can certainly use `E02_15puzzle.tex` as the LaTeXtemplate.

3 Codes

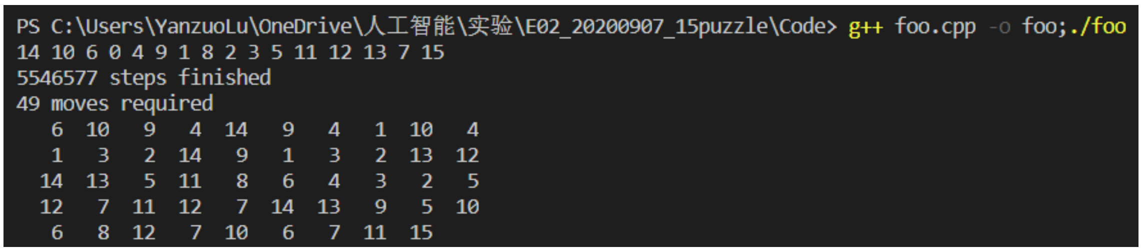
```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <stack>
#include <vector>
using namespace std;

constexpr int N = 4;
constexpr int dir[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
int puzzle[N][N];
int next_limit = 0x7f7f7f7f;
int cnt = 0;

typedef struct Node
{
    int p[N][N];          // puzzle当前状态
    int x, y;             // 数码“0”的横纵坐标
    vector<int> path;    // 记录初始状态到当前状态的数码移动顺序
}Node;

bool valid()
{
    // 计算初始状态逆序数
    int reverse = 0;
    int v[N * N - 1];
    int pos = 0;
    int x;
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            if (puzzle[i][j] == 0)
            {
                x = i;
                continue;
            }
            else v[pos++] = puzzle[i][j];
        }
    for (int i = 0; i < N * N - 1; i++)
        for (int j = 0; j < i; j++)
            if (v[j] > v[i]) reverse++;

    // N为偶数时，先计算出从初始状态到指定状态，空位要移动的行数m,
    // 如果初始状态的逆序数加上m与指定状态的逆序数奇偶性相同，则有解;
    // N为奇数时，初始状态与指定状态逆序数奇偶性相同即有解
    if (N % 2 == 0)
    {
        int res = reverse + N - 1 - x;
        if (res % 2 == 0) return true;
        else return false;
    }
    else
    {
        if (reverse % 2 == 0) return true;
        else return false;
    }
}

int h(int p[N][N])
{
    int dism = 0;

    // 计算曼哈顿距离
    for (int i = 0; i < N; i++)
        for (int j = 1; j <= N; j++)
        {
            if (p[i][j - 1] == 0) continue;
            else if (p[i][j - 1] != i * N + j)
            {
                int x = (p[i][j - 1] - 1) / N;
                int y = p[i][j - 1] - x * N;
                dism += abs(i - x) + abs(j - y);
            }
        }
    return dism;
}

bool is_goal(int p[N][N])
{
    // 判定当前是否已经到达目标状态
    for (int i = 0; i < N; i++)
        for (int j = 1; j <= N; j++)
        {
            if (p[i][j - 1] == 0)
            {
                if (i != N - 1 && j != N) return false;
            }
            else
            {
                if (p[i][j - 1] != i * N + j) return false;
            }
        }
    return true;
}

void print(Node n)
{
    // 打印结果
    int len = n.path.size();
    printf("%d␣steps␣finished\n", cnt);
    printf("%d␣moves␣required\n", len);
    for (int i = 1; i <= len; i++)
    {
        printf("%4d", n.path[i - 1]);
        if (i % 10 == 0) printf("\n");
    }
}

bool dfs(int limit, Node tmpn)
{
    // 函数调用计数
    cnt++;

    // 判定是否到达最终状态
    if (is_goal(tmpn.p))
    {
        print(tmpn);
        return true;
    }

    // 四个方向依次循环
    for (int i = 0; i < 4; i++)
    {
        int dx = dir[i][0];
        int dy = dir[i][1];
        int x = tmpn.x + dx;
        int y = tmpn.y + dy;

        // 判定目标位置是否合法
        if (x >= 0 && x < N && y >= 0 && y < N)
        {
            // 不能往回走
            if (tmpn.path.size() != 0 && \
                tmpn.path[tmpn.path.size() - 1] == tmpn.p[x][y])
                continue;

            // 下一步
            int tmpx = tmpn.x;
            int tmpy = tmpn.y;
            tmpn.p[tmpn.x][tmpn.y] = tmpn.p[x][y];
            tmpn.p[x][y] = 0;
            tmpn.path.push_back(tmpn.p[tmpn.x][tmpn.y]);
            tmpn.x = x;
            tmpn.y = y;

            // 判定边界
            int f = h(tmpn.p) + tmpn.path.size();
            if (f <= limit)
            {
                if (dfs(limit, tmpn)) return true;
            }
            else
            {
                if (f < next_limit) next_limit = f;
            }

            // 回溯
            tmpn.x = tmpx;
            tmpn.y = tmpy;
            tmpn.path.pop_back();
            tmpn.p[x][y] = tmpn.p[tmpn.x][tmpn.y];
            tmpn.p[tmpn.x][tmpn.y] = 0;
        }
    }
    // 搜索失败
    return false;
}

int main()
{
    int zx, zy;
    // 输入puzzle
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
        {
            cin >> puzzle[i][j];
            if (puzzle[i][j] == 0)
            {
                zx = i;
                zy = j;
            }
        }

    // 判定是否有解
    bool v = valid();
    if (!v)
    {
        cout << "Impossible!" << endl;
        return 0;
    }

    // 定义起始节点和深度
    Node start;
    int limit = h(puzzle);
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            start.p[i][j] = puzzle[i][j];
    start.path = {};
    start.x = zx;
    start.y = zy;

    while (true)
    {
        if (dfs(limit, start)) break;
        limit = next_limit;
        next_limit = 0x7f7f7f7f;
    }

    return 0;
}
```

4 Results

本次实验主要采用的算法是 IDA* 即迭代加深 A* 算法，而该算法最典型的应用就是八数码问题和十五数码问题，我们需要解决的十五数码问题相较于八数码问题，最大的区别在于从最初状态到最终状态所需要搜索的步数以指数级上升，最终结果的序列也更长，若要取得较好的算法效果需要对代码本身进行一定的优化。

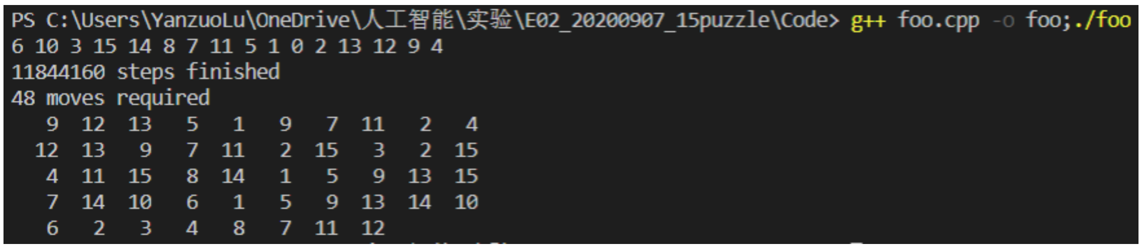
根据实验要求所提供的伪代码，我的代码实现大致分为以下几个部分：第一个是对 DFS 搜索节点的抽象，需要记录当前节点 puzzle 数组存储的数字、数码“0”的 x,y 位置和初始状态到当前状态移动数码的顺序，分别以 int 数组、int 变量和 vector 数组存储，第二个是对初始状态的判断即 valid 函数，根据 N 的奇偶性分为两种不同的情况进行判断，具体方法在代码注释中写明，第三个是对 puzzle 数组的曼哈顿距离的计算即 h 函数，需要注意的一点是不能对数码“0”的距离算入总距离，否则不能保证启发式函数 h 的单调性，第四个是 DFS 搜索的递归调用函数，参数是 limit 即每一次迭代的深度（函数 f 值）限制和当前节点 tmpn，注意对四个方向的后继节点进行判定是否合法，避免出现数组越界等代码错误。

最后我采用了实验方案中的第二组数据和第四组数据，原因是这两组数据的结果较为简单，在我的程序中运行时间也较短，大致在 5s 以内，显然程序运行结果的移动序列和实验方案提供的是完全一致的。



```
PS C:\Users\YanzuoLu\OneDrive\人工智能\实验\E02_20200907_15puzzle\Code> g++ foo.cpp -o foo;./foo
14 10 6 0 4 9 1 8 2 3 5 11 12 13 7 15
5546577 steps finished
49 moves required
   6  10   9   4  14   9   4   1  10   4
   1   3   2  14   9   1   3   2  13  12
  14  13   5  11   8   6   4   3   2   5
  12   7  11  12   7  14  13   9   5  10
   6   8  12   7  10   6   7  11  15
```

Figure 1: result of data2



```
PS C:\Users\YanzuoLu\OneDrive\人工智能\实验\E02_20200907_15puzzle\Code> g++ foo.cpp -o foo;./foo
6 10 3 15 14 8 7 11 5 1 0 2 13 12 9 4
11844160 steps finished
48 moves required
   9  12  13   5   1   9   7  11   2   4
  12  13   9   7  11   2  15   3   2  15
   4  11  15   8  14   1   5   9  13  15
   7  14  10   6   1   5   9  13  14  10
   6   2   3   4   8   7  11  12
```

Figure 2: result of data4