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

# E2 15-Puzzle Problem (IDA*)

18340149 Xinmeng Sun

September 12, 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.

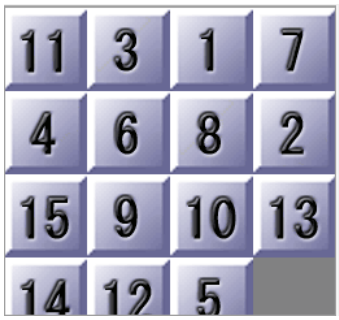
**Iterative-deepening-A* works as follows:** at each iteration, perform a depth-first search, cutting off a branch when its total cost $f(n) = g(n) + h(n)$ exceeds a given threshold. This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm.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

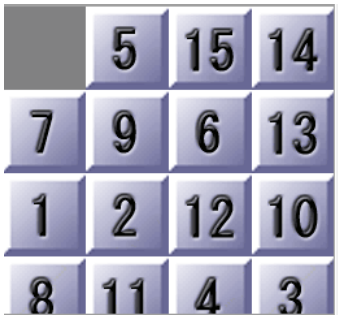
```
path              current search path (acts like a stack)
node              current node (last node in current path)
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
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

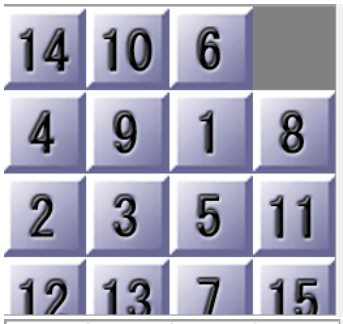
- Please solve 15-Puzzle problem by using IDA* (Python or C++). You can use one of the two commonly used heuristic functions: h1 = the number of misplaced tiles. h2 = the sum of the distances of the tiles from their goal positions.
- Here are 4 test cases for you to verify your algorithm correctness.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



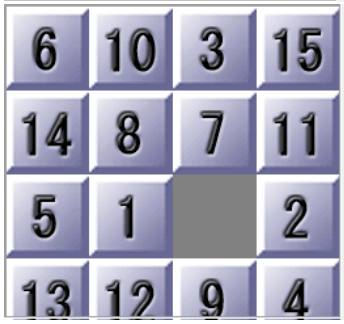

```
TextOut of Result
 11  3  1  7
  4  6  8  2
 15  9 10 13
 14 12  5  0
LowerBound 36 moves
A optimal solution 56 moves
Used time 3 sec
 13 10  8  6  9 12  5 13 10  8
 12 15 14  5 13 12 15 14  5 13
 14  9  4 11  3  1  6  4 11  3
  1  6  4  2  8 10 12 15 10  8
  7  4  2 11  3  5  9 10 11  3
  6  2 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




```
TextOut of Result
  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
 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
  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
```

## 3 My Idea to solve the problem

**How can we abstact the problem as an IDA*?**

- Here we define state as the method tiles be put in the board,and we start from an order.
- H(n), the heuristic function is the estimated value of the cost of the cost from a state to the goal state. Here we use the second one, Manhatton distance. Each time when we calculate thhe h(n), we add the least steps to reach the goal state. Don't forget g(n), because each tile we can only move one tile, the g is the cost to get to n, each time we add 1 to the original g value to get the g value of successors.
- Each time we order the frontier using f=g+h, and firstly expand the best node, and add the successors(if not out of bound) into frontier.
- If we reach goal, then return -1.

## 4 Codes

```
"""
Name:SunXinmeng
ID:18340149
Program:15puzzle.py
```

```
Compile and Run:python15puzzle.py
"""

target = {}
num = 1
#initialization target matrix
for i in range(4):
    for j in range(4):
        target[num] = (i, j)
        num += 1

    target[0] = (3, 3)

#define heuristic function h(n),node is the matrix to store state's picture
def h(node):
    cost = 0
    for i in range(4):
        for j in range(4):
            num = node[i][j]
            x, y = target[num]
            cost += abs(x - i) + abs(y - j)

    return cost


#judge if the node reaches final state
def is_goal(node):
    index = 1
    for row in node:
        for col in row:
            if (index != col):
                break
            index += 1
    return index == 16

#find successors of a node after ordering
def successors(node):
    x, y = 0, 0
    #find 0
    for i in range(4):
        for j in range(4):
            if (node[i][j] == 0):
                x, y = i, j
    success = []
    moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]  #direction
    for i, j in moves:
        a, b = x + i, y + j
        if (a < 4 and a > -1 and b < 4 and b > -1):
            temp = [[num for num in col] for col in node]
            temp[x][y] = temp[a][b]
            temp[a][b] = 0
            success.append(temp)

    return sorted(success, key=lambda x: h(x))

#recursive function
def search(path, g, bound):
    node = path[-1]
    f = g + h(node) #f value of the node
    if (f > bound):
        return f
```

```
    if (is_goal(node)):
        return -1

    Min = 9999
    for succ in successors(node):
        if succ not in path:    #cycle detect
            path.append(succ)
            t = search(path, g + 1, bound)
            if (t == -1):
                return -1
            if (t < Min):
                Min = t;
            path.pop()

    return Min


def ida_star(root):
    bound = h(root)#use the h(root) to be bound
    path = [root]

    while (True):
        t = search(path, 0, bound)
        if (t == -1):   #is path == -1, then return the path and bound
            return (path, bound)
        if (t > 70):    #no path
            return ([], bound)

        bound = t

#choose a begin state to get started
def load():
    # root = [2,  7,  5,  3], [11, 10,  9, 14], [4,  0,  1,  6], [4,  0,  1,  6]
    # root = [[11, 3, 1, 7], [4, 6, 8, 2], [15, 9, 10, 13], [14, 12, 5, 0]]
    root = [[5, 1, 3, 4], [2, 7, 8, 12], [9, 6, 11, 15], [0, 13, 10, 14]]
    # root = [[6, 10, 3, 15], [14, 8, 7, 11], [5, 1, 0, 2], [13, 12, 9, 4]]
    return root


root = load()
(path, bound) = ida_star(root)
step = 0
print('********Solve 15 Puzzle********')
for p in path:
    print('---------------------')
    print('step', step,'>>')
    step += 1
    for row in p:
        print(row)
```

## 5 Results

Here we use the second puzzle to run on our program.As the picture followed shows, we list the step from the start state to the goal state, and the bound is 21.

```
********Solve 15 Puzzle********
-------------------
step 0 >>
[5, 1, 3, 4]
[2, 7, 8, 12]
[9, 6, 11, 15]
[0, 13, 10, 14]
-------------------
step 1 >>
[5, 1, 3, 4]
[2, 7, 8, 12]
[9, 6, 11, 15]
[13, 0, 10, 14]
-------------------
step 2 >>
[5, 1, 3, 4]
[2, 7, 8, 12]
[9, 6, 11, 15]
[13, 10, 0, 14]
-------------------
```

```
step 3 >>
[5, 1, 3, 4]
[2, 7, 8, 12]
[9, 6, 11, 15]
[13, 10, 14, 0]
--------------------
step 4 >>
[5, 1, 3, 4]
[2, 7, 8, 12]
[9, 6, 11, 0]
[13, 10, 14, 15]
--------------------
step 5 >>
[5, 1, 3, 4]
[2, 7, 8, 0]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
step 6 >>
[5, 1, 3, 4]
[2, 7, 0, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
```

```
step 7 >>
[5, 1, 3, 4]
[2, 0, 7, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
step 8 >>
[5, 1, 3, 4]
[0, 2, 7, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
step 9 >>
[0, 1, 3, 4]
[5, 2, 7, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
step 10 >>
[1, 0, 3, 4]
[5, 2, 7, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
```

next steps:

```
step 11 >>
[1, 2, 3, 4]
[5, 0, 7, 8]
[9, 6, 11, 12]
[13, 10, 14, 15]
--------------------
step 12 >>
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 0, 11, 12]
[13, 10, 14, 15]
--------------------
step 13 >>
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 0, 14, 15]
--------------------
step 14 >>
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 0, 15]
--------------------
```

```
step 15 >>
[1, 2, 3, 4]
[5, 6, 7, 8]
[9, 10, 11, 12]
[13, 14, 15, 0]
bound =  21

Process finished with exit code 0
```

# 6 What I leart from the experient?

- Firstly, IDA* search , iteratively deepening A* search .It is a serch algorithm combining A* and IDDFS.
  Iterative deepening algorithm is based on THE DFS search algorithm to gradually deepen the search depth, which avoids the shortcoming of breadth first search occupying too much search space, and also reduces the blindness of depth first search. It basically determines at the beginning of the recursive search function whether the current search depth is greater than the predefined maximum search depth, if greater than, exit this level of search, if not greater than, continue the search. So the solution you end up with is going to be the optimal solution. In the A* algorithm, we use A reasonable evaluation function, and then select the node with the lowest fCost among the obtained child nodes for extension, so as to complete the search for the optimal solution. But A * algorithm, the need to maintain close list and the open list, need to test the extension node, and ignore has entered to the closed list node (that is, the so-called "have already tested the node"), in addition to also want to check with extended node repeat, if repeated corresponding updates. Therefore, the main cost of A* algorithm is spent on state detection and sorting of nodes with the minimum cost. This process takes up A large amount of memory, and hash is generally used for repeated state detection in order to improve efficiency.

- IDA*'s properties:Combined with the artificial intelligence of A* algorithm and the advantages of backtracking that consumes less space, it can have unexpected effects in some large scale search problems.

- This time I try to write in python. As for me, it is not as familiar as C++, but I am really absorbed with its convenience and simple code. I will try my best to get experieced with Python.