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

15-Puzzle Problem (IDA*)

17341203 Yixin Zhang

September 7, 2019

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

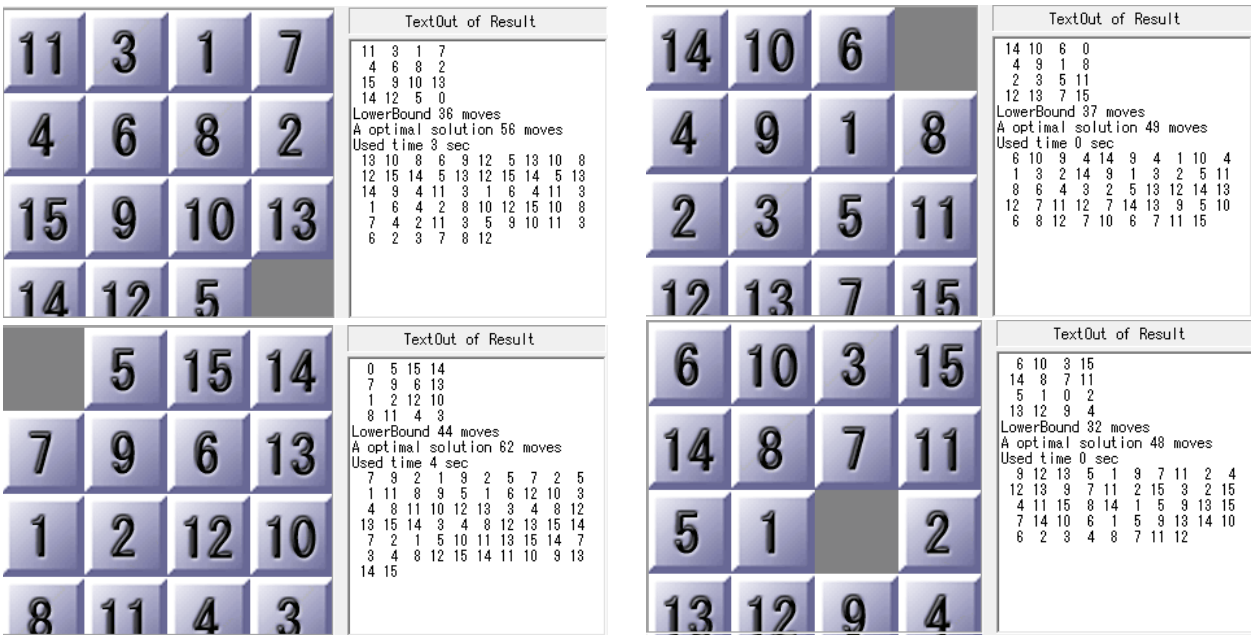
```
path              current search path (acts like a stack)
node              current node (last node in current path)
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)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



- Please send `E02_YourNumber.pdf` to `ai_201901@foxmail.com`, you can certainly use `E02_15puzzle.tex` as the LaTeXtemplate.

3 Implementation

3.1 Overview

Thanks to the succinct syntax of Python, we can easily rewrite the pseudocode in section 1.2 into Python code. The program contains only one file 'main.py'. Each function's doc string explains what it does.

3.2 Code

main.py

```
# -*- coding: utf-8 -*-
import numpy as np
import datetime

def isGoal(node):
    """
    Test if the given node (state) is the goal.
    """
    goal = np.append(range(1, sidelen * sidelen), 0).reshape(sidelen, sidelen)
```

```python
    return (node == goal).all()


def h1(node):
    """
    Heuristic function 1: using the number of misplaced tiles.
    """
    goal = np.append(range(1, sidelen * sidelen), 0).reshape(sidelen, sidelen)
    return sidelen * sidelen - np.count_nonzero(goal == node)


def h2(node):
    """
    Heuristic function 2: using Manhattan distance.
    """
    target = {}
    count = 1
    for i in range(sidelen):
        for j in range(sidelen):
            target[count] = (i, j)
            count += 1
    target[0] = (sidelen-1, sidelen-1)

    total_distance = 0
    for i in range(sidelen):
        for j in range(sidelen):
            val = node[i, j]
            total_distance += abs(i - target[val][0]) + abs(j - target[val][1])
    return total_distance


def ida_star(root):
    """
    Do IDA* algorithm from node ‘root‘.
    """
    bound = h2(root) # initial bound
    path = [root]
    while True:
        ret = search(path, 0, bound)
        if ret == True:
            return path
        if ret == float('inf'):
            return False
        else:
            bound = ret


def search(path, g, bound):
    """
    Do the DFS.
    """
    node = path[-1] # current node is the last node in the path
    f = g + h2(node) # heuristic function
    if f > bound:
        return f
    if isGoal(node):
```

```
            return True

    temp = np.where(node == 0) # find the blank
    blank = (temp[0][0], temp[1][0]) # blank's position

    succs = []
    moves = [(0, -1), (0, 1), (-1, 0), (1, 0)] # up, down, left, right
    for move in moves:
        next_blank = tuple(np.sum([blank, move], axis=0))
        if next_blank[0]>=0 and next_blank[0]<sidelen and next_blank[1]>=0 and
            next_blank[1]<sidelen:
            succ = node.copy()
            succ[blank], succ[next_blank] = succ[next_blank], succ[blank]
            succs.append(succ)

    _min = float('inf')
    succs.sort(key=lambda x: h2(x))
    for succ in succs:
        if not any((succ == x).all() for x in path): # special syntax
            path.append(succ)
            t = search(path, g+1, bound)
            if t == True:
                return True
            if t < _min:
                _min = t
            path.pop()
    return _min


def makeActions(path):
    """
    Constuct a list containing numbers to be moved in each step.
    """
    if path == False:
        raise ValueError('No solution!')

    actions = []
    for i, node in enumerate(path[1:]):
        temp = np.where(node == 0) # find the blank
        blank = (temp[0][0], temp[1][0]) # blank's position
        actions.append(path[i][blank])
    return actions


if __name__ == '__main__':
    print('***STARTING***', datetime.datetime.now().strftime('%Y.%m.%d %H:%M:%S'))

    filename = 'mytest.txt'
    puzzle = np.loadtxt(filename, dtype=np.uint8) # number 0 indicates the blank
    sidelen = len(puzzle) # side length of puzzle
    result = makeActions(ida_star(puzzle))
    print(result)
    print('Length:', len(result))

    print('***Finished***', datetime.datetime.now().strftime('%Y.%m.%d %H:%M:%S'))
```

4 Results

The test cases given above are all too big for my Python code to solve, so I've made my own test cases. Four tests are presented below, each with my solution and a solution from '15puzzle Optimal Solver' [1].

5 Discussion

The IDA* algorithm itself is not hard to understand, but the code implementation progress requires some efforts. I've met several obstacles while writing Python code. The numpy arrays are unhashable, so they cannot be used as keys in Python dictionaries. Due to the special performance on logical operators (such as 'and', 'or', 'not') used on numpy arrays, the syntax must be taken care of – in most cases, keywords like 'all' and 'any' should be used.

At last, I found that my Python code turns out to be very slow when the number of steps exceed 40. I think it is because not only Python always runs more slowly than C++, but the heuristic function using Manhattan distance is not good enough. For further study, a heuristic function using 'walking distance' might be considered.

References

[1] 15puzzle Optimal solver, `http://www.ic-net.or.jp/home/takaken/e/15pz/`