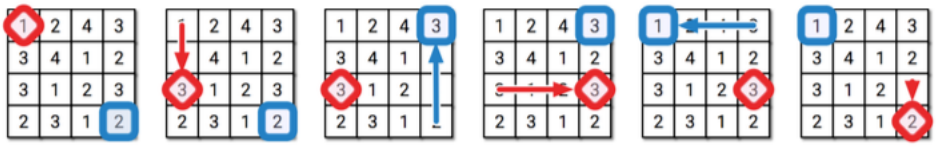
Report

Problem Description

The infamous Mongolian puzzle-warrior Vidrach Itky Leda invented the following puzzle in the year 1437. The puzzle consists of an n × n grid of squares, where each square is labeled with a positive integer, and two tokens, one red and the other blue. The tokens always lie on distinct squares of the grid. The tokens start in the top left and bottom right corners of the grid; the goal of the puzzle is to swap the tokens.

In a single turn, you may move either token up, right, down, or left by a distance determined by the other token. For example, if the red token is on a square labeled 3, then you may move the blue token 3 steps up, 3 steps left, 3 steps right, or 3 steps down. However, you may not move either token off the grid, and at the end of a move the two tokens cannot lie on the same square.

Describe and analyze an efficient algorithm that either returns the minimum number of moves required to solve a given Vidrach Itky Leda puzzle, or correctly reports that the puzzle has no solution. For example, given the puzzle in the following figure, your algorithm would return the number 5.



Algorithm Description

In this problem we can firstly define a graph data structure with directed weighted graph. The weight in graph means convert times. The node in graph means the state in puzzle. Two nodes can be connected when node1 can be moved to node2. And the we can solve the shortest path from the start node to end node to get the final result. Next I will firstly

describe the graph definition. Secondly, I will introduce how to solve the shortest path for definition of graph.

- Graph definition:

In each state of the puzzle including n x n grids of squares. The puzzle contains two tokens, one red and the other blue. Thus, for each state in puzzle we can define a node for graph, the node can be defined by array with 4 integers like [red_i, red_j, blue_i, blue_j]. In this array, “red_i” means the row number in puzzle for red token; “red_j” means the column number in puzzle for red token; “blue_i” means the row number in puzzle for blue token; and “blue_j” means the column number in puzzle for blue token. And then, we can move either token up, right, down, or left by a distance determined by the other token. Therefore, for each moving we can get next state for puzzle with new node [red_i1, red_j1, blue_i1, blue_j1]. For each state, we can get more than one next state. We define the start state is state_0, and define the state_i can be moved to as [state_1, …, state_n]. We can get state_1, …, state_n are neighbor nodes for state_0. For example, the start state in puzzle is like:

The node for state above is: [0, 0, 3, 3], because the red token is in [0 ,0], and the blue token is in [3, 3]. And then neighbor nodes for this node are: [2, 0, 3, 3], [0, 2, 3,3], [0, 0, 2, 3], and [0, 0, 3, 3].

Consequently, we can build graph by recursive method for each node. The recursive termination condition is: the node has been build in graph. We use adjacency matrix to define the graph, and the adjacency matrix defined by dictionary data structure in python. The pseudocode for building adjacency matrix is as follows:

Input: graph store by dictionary data structure; start node for [0, 0, n, n]; puzzle for

2-dimentional array; and the integer n which represents n x n puzzle.

Run recursive to build graph:

```
Create_graph(graph, node, array, n):
    if node in graph: #recursive termination condition
        return
    node_neighbors = dict() #stored nodes which can be connected with input node
    node_index = change_to_index(node)
    red_i, red_j, blue_i, blue_j = get_index(node_index)
    red_step = array[blue_i][blue_j]
    blue_step = array[red_i][red_j]
    # Red go top
    index = red_i - red_step
    if index != red_i and index >= 0:
        next_node = change_to_node(index, red_j, blue_i, blue_j)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Red go bottom
    index = red_i + red_step
    if index != red_i and index <= n:
        next_node = change_to_node(index, red_j, blue_i, blue_j)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Red go left
    index = red_j - red_step
    if index != red_j and index >= 0:
        next_node = change_to_node(red_i, index, blue_i, blue_j)
        node_neighbors[next_node] = 1
```

```
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Red go right
    index = red_j + red_step
    if index != red_j and index <= n:
        next_node = change_to_node(red_i, index, blue_i, blue_j)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Blue go top
    index = blue_i - blue_step
    if index != blue_i and index >= 0:
        next_node = change_to_node(red_i, red_j, index, blue_j)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Blue go bottom
    index = blue_i + blue_step
    if index != blue_i and index <= n:
        next_node = change_to_node(red_i, red_j, index, blue_j)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
        create_graph(graph, next_node, array, n)
    # Blue go left
    index = blue_j - blue_step
    if index != blue_j and index >= 0:
        next_node = change_to_node(red_i, red_j, blue_i, index)
        node_neighbors[next_node] = 1
        graph[node] = node_neighbors
```

```
            create_graph(graph, next_node, array, n)
        # Blue go right
        index = blue_j + blue_step
        if index != blue_j and index <= n:
            next_node = change_to_node(red_i, red_j, blue_i, index)
            node_neighbors[next_node] = 1
            graph[node] = node_neighbors
            create_graph(graph, next_node, array, n)
```

After running recursive method we can build the graph with adjacency matrix. If the graph contains the node [n, n, 0, 0], that means the problem can be solved, because we build graph with start node [0, 0, n, n], and it can be moved to [n, n, 0, 0]. Otherwise, the problem can not be solved. If the problem can be solved, we can use Dijkstra Algorithm to solve the shortest path from start node [0, 0, n, n] to end node [n, n, 0, 0].

- Dijkstra Algorithm:

Suppose G=(V,E) is a weighted directed graph, divide the set V of vertices in the graph into two groups, the first set is the set of vertices with the shortest path found (denoted by S, initially there is only one source point in S. After each shortest path is found, it will be added to the set S until all vertices are added to S, and the algorithm is over. The second group is the set of vertices of the remaining undetermined shortest paths (denoted by U). Add the second set of vertices to S in increasing order of the shortest path length. In the process of joining. Always maintain that the shortest path length of each vertex from v to S is not greater than the shortest path length of any vertex from V to U. In addition, each vertex corresponds to a distance, and the distance of a vertex in S is the shortest path length from V to that vertex, the distance of a vertex in U, is the current shortest path length from v to this vertex including only the vertex in S. The pseudocode for Dijkstra Algorithm is as follows:

```
Step 1: Find unlabeled node which nearest with start node;
Step 2: Take the node in step 1 as center, update the neighbor nodes of this node;
Step 3: Repeat step 1 until find end node.
```

Algorithm Complexity Proof

The algorithm complexity for build graph is: O(n^4);

The algorithm complexity for Dijkstra algorithm is: O(n^2)

Therefore, the algorithm complexity for solving puzzle is: O(n^4) + O(n^2) = O(n^4)

- Proof by mathematical induction:

a. Graph building:

In bad case, the red token can at each grid in puzzle, and blue token can also at each grid in puzzle. Also, the position of red token and blue token can do permutation and combination. And the node is definition with [red_i, red_j, blue_i, blue_j], so the time complexity is: n * n *n *n = n^4

T(n) = O(n^2) * O(n^2) = O(n * n * n * n) = O(n^4)

Firstly, we have T(0) = 0 ^ 4 = 0, and assume T(n-1) = (n-1) ^ 4

We have:

T(n) = 1 + T(0) + T(1) + T(2) + … + T(n-1)

T(n) = 1 + 1 ^ 4 + 2 ^4 + … + (n - 1) ^ 4

T(n) = 1 + [0 ^ 4 * (1 – n ^ 4] / (1 - 2)

T(n) = n ^ 4

Thus, from mathematical induction we get T(n) = O(n^4)

b. Dijkstra algorithm:

T(n) = n * (1 + 2 + 3 + .. + n) = O(n ^ 2)

Firstly, we have T(1) = 0, and assume T(n - 1) = (n - 1) ^ 2

T(n) = 1 + T(0) + T(1) + T(2) + .. + T(n - 1)

T(n) = 1 + 0 * (0) + 1 * (1) + 2 * (1 + 2) + 3 * (1 + 2 + 3) + .. + (n - 1) * (1 + 2 + 3 + … + n -1)

T(n) = 1 + n * (0 + n - 1) / 2

T(n) = 1 + n * (n - 1) / 2

T(n) = O(n^2)

Thus, from mathematical induction we have T(n) = O(n^2)

Consequently, the puzzle problem algorithm complexity is: O(n^4) + O(n^2) = O(n^4).