Amir Aslamov,
Uliana Ozerova
COT4400
April 20, 2022

Report. 8-Puzzle.

Introduction

An 8 puzzle is a 3x3 board with 8 tiles and one empty space. The objective is to reach the goal state by sliding adjacent tiles one by one(to the left, up, right, down) using the empty space(marked by 0), which, in turn, is changing the state of the 8-puzzle.

1	2	3
8	0	4
7	6	5

Problem

The main goal of the assignment is to practice and demonstrate the understanding of various algorithms by implementing the 8-puzzle. We need to find the shortest path(the lowest cost) to the goal state by applying rules and restrictions set by the instruction.

We are to implement the game using Breadth-First Search, Depth-First Search, and Dijkstra algorithms.

Set of Restrictions

1. Tiles move in this order(for BFS and DFS): left, up, right, down.
2. The goal state is known.

3. The cost of the swap for BFS and DFS is 1.
4. The cost of the swap for Dijkstra is the sum of the number of a tile moved AND the number of tiles displaced in a certain state(in a place different from their position in the goal state).

Solution

1. **BFS**

Theory:

BFS is a graph-traversing algorithm that visits all adjacent vertices - neighbors - of a vertex before moving to the next one.

Explanation/Pseudocode:

```
Algorithm:
    BFS(Node c):
        create a queue Q
        create list visited
        Put v into visited into Q
        while Q is non-empty:
            current = head u of Q
            remove the head u of Q
            expandChildren //get the children of u
            for each child k
                if k is not in the visited list and not in queue
                    Enqueue(k)
        return path and cost if goal state is found
```

The running time of the algorithm if we do not consider the time spent on setting up the puzzle will be O(V + E), as we are visiting every vertex and edge at least once.
To implement a breadth-first search, we declared a LinkedList, which we used as a queue that stored all adjacent neighbors. After storing the starting node in the queue, we deque it to explore. Then, we add all adjacent neighbors/children of the node to the

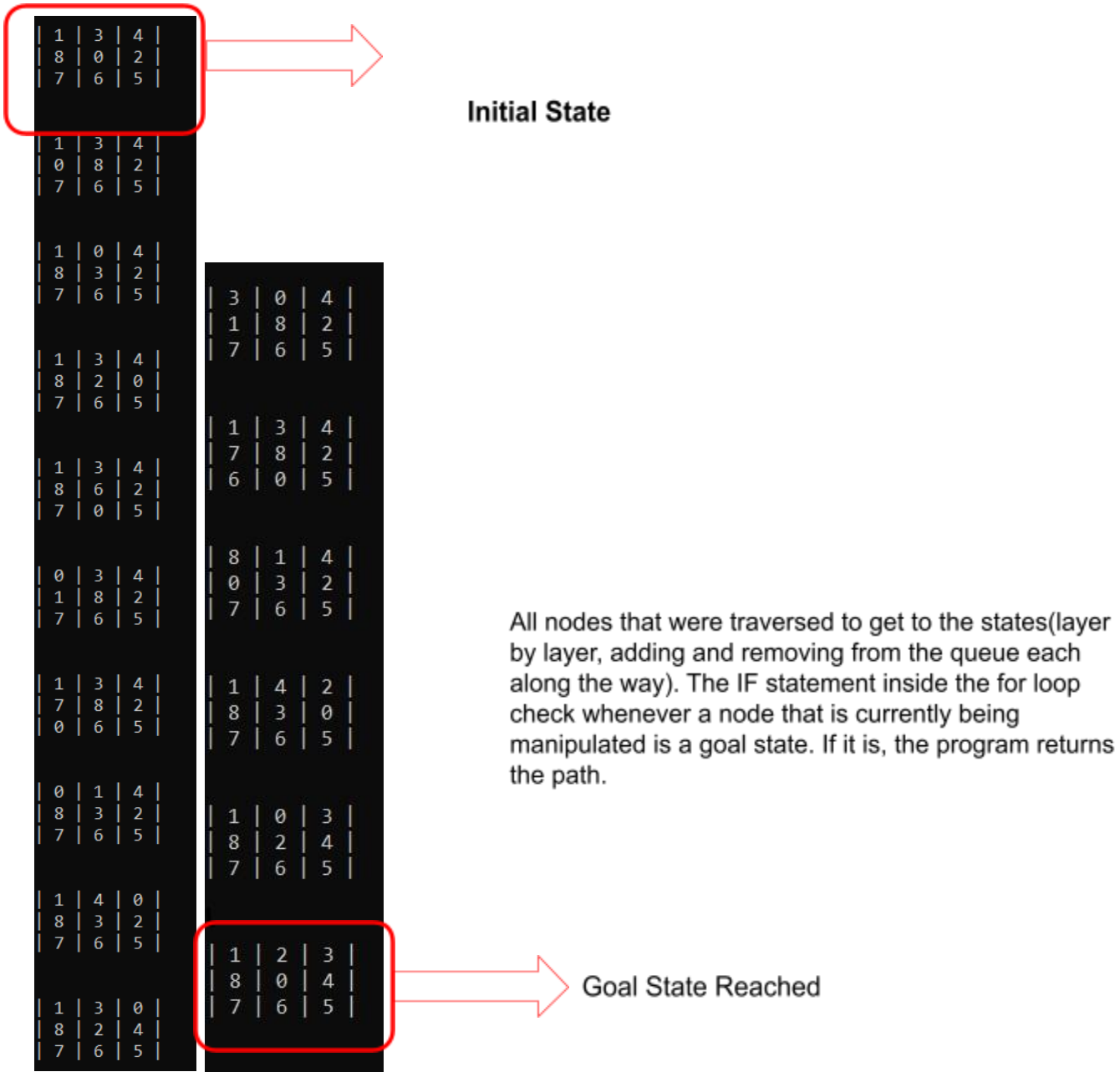
queue and explore them one by one, adding the children to the queue. This approach makes sure that until we check all the neighbors of the vertex U at the same level, we won't go to the current node's grandchildren(in other words, will not go to the deeper level before we explore all neighbors).

Our Implementation:

```
0 references
public List<Node> BFS(Node root)
{
    List<Node> path = new List<Node>();
    List <Node> explored = new List<Node>();
    LinkedList <Node> qu = new LinkedList<Node>();

    if(root.checkIfGoal())
    {
        tracePath(path, root);
        return path;
    }

    qu.AddLast(root);

    bool goalFound = false;

    //if openlist = 0, we explored all nodes and did not find a solution
    while(qu.Any()  && !goalFound)
    {
        Node current = qu.First();
        explored.Add(current);
        qu.RemoveFirst();

        //expands in 4 ways getting.
        current.ExpandMove();

        //explore each child from left to right
        for(int i = 0; i < current.children.Count; i++)

        {
            Node current_child = current.children[i];
            if(current_child.checkIfGoal())
            {
                Console.WriteLine("Goal puzzle found");
                goalFound = true;
                //trace path back
                tracePath(path, current_child);
            }

            if(!checkIfContainsLL(qu, current_child) && !checkIfContains(explored, current_child))
            {
                //openList.Add(current_child);
                qu.AddLast(current_child);
            }

        }
    }
    return path;
}
```

Example/Result:

If the order of the direction is left, up, right, down, the following displays simulates the path of the work of BFS to reach the goal state on example 1:



2. DFS

Theory:

DFS is a graph-traversing algorithm that starts from the starting vertex and continues to explore as far as it can before backtracking.

Explanation/Pseudocode:

```
Algorithm:
   bool goalFound = false            // a global boolean to help with DFS
   DFS(Node initial):
        Create a list "visited"      // to keep the list of visited nodes
        Create a stack "stack"       // to help keep the children of a visited node for later
exploration
        Create a list "path"         // to keep track of the list of nodes leading up to the goal state

        visited.Add(initial)         // add the initial node to the list of visited
        stack.Push(initial)          // push it onto the stack

        while(stack.Count != 0 AND !goalFound):      // while the stack not empty and goal not
found
              initial = stack.Pop()                  // we pop the last node visited from stack
              initial.ExpandMove()                   // it is a helper method to explore all the
possible
                                                     // moves we can take with the current node
              for  (child in initial.Children):      // because ExpandNode makes all the possible
                                                     // moves into children-nodes, we explore them
                    if (child.CheckGoal() == true):
                          goalFound = true
                          tracePath(path, child)
                    If (!visited.Contains(child)):   // if any child is not yet visited, we push it onto
the
                                                     // the stack and the list of visited
                         visited.Add(child)
                         stack.Push(child)

         return path
```

The running time of the algorithm if we do not consider the time spent on setting up the puzzle will be O(V), as we are visiting every vertex at least once.

The algorithm above implements the DFS with the help of the stack data structure, because in DFS traversal we keep traversing down the children of a node before exploring the parents, so the first node visited gets explored the last.

We also used some helper variables and helper functions to help with the algorithm.

First of all, a boolean variable “goalFound” is used to help with determining if the goal state is detected inside a loop, so that the loop does not keep iterating.

Secondly, the helper method “ExpandMove” helps us get the list of all possible ways we can move the tile, thus producing respective nodes with states represented by the location of tiles. When those possible nodes are created, they are stored as children nodes of this current node, and the parent property of all these children nodes becomes this current node.

Additionally, the helper function “checkIfGoal” helps with determining if the node is the goal node, in which all the tiles in the array board are all in the correct place.

Finally, another helper function “tracePath” is called to help us track the path up to the root of all the nodes that lead up to the goal node. This is done with the help of the parent property of all the nodes, so until the parent property is not equal to Null, in which case the node is the root with no parent assigned, we keep tracing up the graph.

Overall, the idea of the algorithm is to keep pulling the nodes from the stack so as to explore the nodes visited the latest, while keeping track of all the nodes visited; as each node is visited all the possible children of the node are pushed onto the stack, to then

be explored on the next cycle of the loop. Once the goal node is found, the list path is assigned the output of the helper function “tracePath” and is then returned by this function DFS.

Our Implementation:

This is the code snippet of our DFS method:

```
4 references
bool goalFound = false;
0 references
public List <Node> DFS (Node initial)
{
    List <Node> visited = new List<Node>();
    Stack <Node> stack = new Stack<Node>();
    List <Node> path = new List<Node>();

    visited.Add(initial);
    stack.Push(initial);

    while (stack.Count != 0 && !goalFound)
    {
        initial = stack.Pop();
        initial.ExpandMove();
        for (int i = 0; i < initial.children.Count; i++)
        {
            if (initial.children[i].checkIfGoal())
            {
                goalFound = true;
                tracePath(path, initial.children[i]);
                return path;
            }
             if (!visited.Contains(initial.children[i]))
            {
                visited.Add(initial.children[i]);
                stack.Push(initial.children[i]);
            }
        }
    }
    return path;
}
```

3. Dijkstra

Theory:

Dijkstra's algorithm is a greedy method of finding the shortest distance between any two nodes in a graph data structure. When applied to our 8 Puzzle project, Dijkstra's algorithm finds the goal node that can be reached with the least cost of displacement of the tiles in the board.

Explanation/Pseudocode:

```
Algorithm:
    Dijkstra(Node current):
        bool goalFound = false          // a helper boolean to determine if the goal is found
        create a priority queue "pq"    // a priority queue which will help in sorting all the nodes based
                                        // on visited nodes' cost
        Create a list "visited"         // to store a list of visited nodes
        Create a list "path"            // to store the nodes that make up the nodes up to the initial

        pq.Enqueue(current, current.costSum.Cost)
        visited.Add(current)

        int cost = current.getCost()       // call a helper method to calculate the cost of the node
        int displd_tiles = current.getNumOfDisplacedTiles()  // call a helper method to calculate the
                                                             // the amount of displaced tiles
        Int sum = cost + displd_tiles       // append the amount above to the cost
        current.setCostSumPair(cost, sum)       // call a helper method to set the tuple that stores
                                                // a pair of cost and sum

         while(pq.Count > 0):
            Node visit = pq.Dequeue()           // we dequeue from the priority queue, visit node
                                                // has the lowest cost among the nodes in the queue
            visit.ExpandMove()                  // helper method to explore possible moves

            for(child in visit.Children):       // visit all children
                int cost_child = child.getCost()    // calculate the cost and amount of displaced tiles
                int dsplcd_tiles_child = child.getNumOfDisplacedTiles()
                int sum_child = cost_child + dsplcd_tiles_child
                child.setCostSumPair(cost_child, sum_child)
```

```
            if (child.checkGoal()):              // if the goal is found, we break out the loops
                goalIsFound = true
                tracePath(path, child)
                break
            if (!checkIfContains(visited, child)):      // if the child is in visited list, we enqueue it
                pq.Enqueue(child, child.costSumPair.Cost)
        if goalFound:
            break

    return path
```

The algorithm above implements the Dijskstra's algorithm on the problem with the help of the priority queue data structure, which keeps the nodes sorted according to the cost associated with them, and each time, we select the node with the lowest cost.

Again, we also used helper variables and helper methods to help us with the algorithm. For example, the boolean variable "goalFound" was used to navigate the flow of the program inside the loops in case of the goal being found. The helper methods "getCost" and "getNumOfDisplacedTiles" help with the calculation of the respective sum and the amount of displaced tiles of the node. The "getCost" works by calculating the tile that gets moved due to the expansion of move of the node, while the "getNumOfDisplacedTiles" works by going through each tile on the board and comparing to each tile in the goal board, and each time the tile is misplaced, the misplaced tile cost is added to the sum amount and returned at the end.

The helper method "ExpandMove" helps with the exploration of all possible moves of a node and making these moves into the node's children nodes. The helper method "setCostSumPair" helps with setting the tuple property of the node, the first of which is the cost and the second the amount of displaced tiles.

Overall, the algorithm explores the nodes and calculates the appended cost of each node, while also storing them into the property cost of each node. The Priority queue helps with picking the nodes for exploration, but the main search method used is DFS on the nodes.

Our Implementation:

Here is the code snippet of the code:

```
0 references
public List <Node> dijkstra(Node c)
{
    bool goalIsFound = false;
    PriorityQueue <Node, int> pq = new PriorityQueue<Node, int>();

    List<Node> visited = new List<Node>();
    List <Node> path = new List<Node> ();
    pq.Enqueue(c, c.costSumPair.Item1); //cost from c to c is 0
    visited.Add(c);

    int costi = c.getCost();
    int dti = c.getNumOfDisplacedTiles();
    int sumi = costi + dti;
    c.setCostSumPair(costi, sumi);

    while (pq.Count > 0)
    {
       Node current = pq.Dequeue();

        //look at all children
        current.ExpandMove();

        //for each child
        for(int i = 0; i < current.children.Count; i++)
        {
            Node current_child = current.children[i];
            int cost = current_child.getCost();
            int dt = current_child.getNumOfDisplacedTiles();
            int sum = cost + dt;
            current_child.setCostSumPair(cost, sum);

            if (current_child.checkIfGoal())
            {
                goalIsFound = true;
                tracePath(path, current_child);

            }
            if (!checkIfContains(visited, current_child))
            {
                pq.Enqueue(current_child, current_child.costSumPair.Item2);
            }
            if(i == current.children.Count -1)
            {

            }
        }
    }
    return path;
}
```

```
}
0 references
public int getCost(Node p)
{
    //compare the parent and the child
    //record what was moved
    int cost = -1;
    if (p.parent != null)
    {
        for (int i = 0; i < p.puzzle.Length && i < p.parent.puzzle.Length; i++)
        {
            if (p.parent.puzzle[i] != p.puzzle[i])
            {
                if (p.parent.puzzle[i] >= p.puzzle[i])
                    cost = p.parent.puzzle[i];
                else
                    cost = p.puzzle[i];
            }

        }
    }

    return cost;
}

0 references
public int getNumOfDisplacedTiles(Node p)
{
    int numDisplacedTiles = 0;

    int[] goal =
   {
        1, 2, 3,
        8, 0, 4,
        7, 6, 5,
    };

    for (int i = 0; i < goal.Length && i < p.puzzle.Length; i++)
    {
        if (p.puzzle[i] != goal[i])
        {
            numDisplacedTiles++;
        }

    }
    return numDisplacedTiles;
}
```

Example/Result: