Artificial Intelligence and Neural Networks

Project Documentation

Sliding Puzzle Solver using A*

Submitted By

Syed Danish Hasan Shah

0107-BSCS-2017

Section: B

Submitted To

Ms. Asma Kanwal

Government College University, Lahore

Table of Contents

1 Introduction:

This section is intended to provide a brief description of my project, so that the reader is familiarized with some aspects of it. We will dive deeper into the inner workings of my project after this section.

1.1 What is a Sliding Puzzle?

In a sliding puzzle game, players are given an n*n grid, in which one cell is blank while the others contain digits from 1 — (n^2-1).The objective of the game is to configure the position of each cell according to a final state in which all the digits appear sequentially (row-wise), with the blank cell either being at the start or at the end.

The rules of the game state that in order to move a cell, only the cells adjacent to the blank cell can be swapped with the blank-cell position. The movement can only be made vertically or horizontally, not diagonally. No other cells can be moved legally. For instance, in a 3x3 puzzle, the objective would be to position the cells in an ascending order (rows first), placing the blank symbol either at the top-left cell or the bottom-right cell, all the while following the above-mentioned rules. [1]

Sliding puzzle is also referred to as n-puzzle, where n is the highest digit involved in the puzzle. For example, in a 3x3 sliding puzzle, the highest number would be 8, and as such, a 3x3 puzzle can also be referred to as 8-puzzle. The most common variants of the Sliding Puzzle are the 8-puzzle and the 15-puzzle.

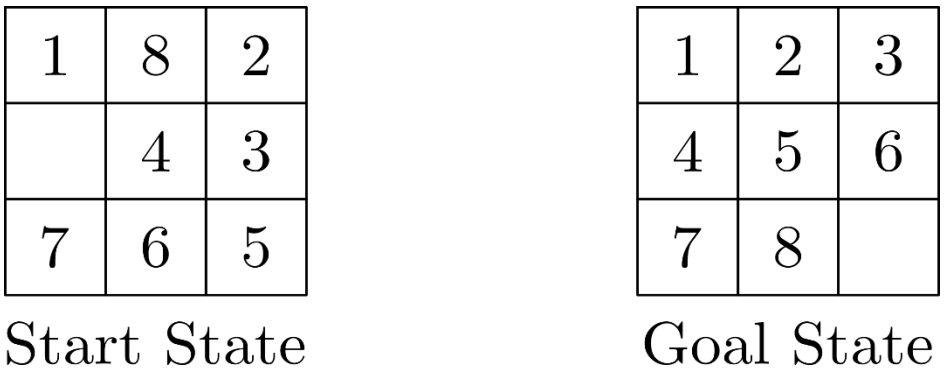

Figure 1.1-1: Example of a Sliding Puzzle.

1.2 What is the A* Algorithm?

A* algorithm is a variation of or an extension to Dijkstra's Shortest Path Algorithm. While Dijkstra's algorithm is uninformed, i.e. it does not have any information about whether it is moving towards the correct node or not, the A* algorithm is a bit 'smarter'. Instead of just considering the cost of the path, the A* algorithm also considers some heuristic function in its calculation for the path to explore. In this way, as the heuristic value gets lower and lower, the algorithm 'knows' that it is moving in the correct direction. [1]

The heuristic function, h(x), can be a straight-line distance from the current node to the goal node in case of a conventional graph. However, in the case of our puzzle, the heuristic function is the number of 'tiles' that do not match the goal state's tiles. [1]

A* algorithm is both complete and optimal. If a solution exists, it will find it, and that solution will always be the most optimal solution. [1]

1.3 What Does my Project Accomplish?

The objective of my project is to accept an initial puzzle state by the user, check if it is solvable, and then return the puzzle states that lead to the goal state if it is indeed solvable. If not, then return a message to the user communicating that the puzzle is insolvable.

This project has been written in JavaScript. HTML and CSS have been used to develop the front-end or the GUI.

To start the application, simply open the *index.html* file. The code related to the implementation of the A* algorithm is found in *index.js*. While, the code for providing structure and stylization is found in *index.html* and *style.css*

With the introduction of my project out of the way, let us consider the project in detail.

NOTE: The main purpose of the pseudo-codes provided below is to explain succinctly how the algorithms work. To that end, I have taken some liberties and modified them from my original source code in such a way that they can be understood easily. If I were to include my original source code as it is in this report, it would also include some implementation details that are specific to my source language (JavaScript) and would divert attention away from what the algorithm is really supposed to do. So, the pseudo-codes following this section are intended to be language-agnostic. Again, their main purpose is not fidelity to my original source code, but rather their understanding in an easy way.

2 A* Algorithm:

Before considering the A* algorithm's pseudocode, we need to define some terms/variables that are used:

openList: This list is used to keep track of nodes that have been visited (perhaps as the child of some other node) but not expanded yet. openList is implemented as a priority queue and always returns the node with the lowest f(n) value.

closeList: This list is used to keep track of nodes that have been both visited AND expanded). closeList is implemented as a map in the source code, to provide O(1) look-ups instead of the linear O(N) we get when implemented as a list.

f(n): This is the function on the basis of which nodes are selected for expansion. In the code, it is given by the function *getPriority()*. It is defined as:

$$f(n) = g(n) + h(n)$$

g(n): Cost of the path from the sourceNode to node n. In the code, g(n) is represented by the property *moves*.

h(n): heuristic function that predicts the cost to reach goalNode from node n. In the code, h(n) is given by the function *getNumberOfMisplacedTiles()*.

Our heuristic function should always be admissible in order for A* to be optimal. A heuristic function h(n) is said to be admissible if for each node n, h(n) <= h*(n), where h*(n) is the actual cost of reaching the goal node from n. An admissible heuristic is always 'optimistic' i.e. it never over-estimates the actual cost of reaching the goal node from the current node n.

2.1 Pseudo-code:

Pseudo-code for the A* used in my project is given below:

```
aStar(sourceNode S) {
  openList.insert(sourceNode)
  while (!openList.isEmpty()) {
    currentNode = openList.remove()
    if (currentNode.isGoal()) {
      break
    }
    if (currentNode in closeList) {
      continue
    }
    children = currentNode.getChildren()
    for (i = 0; i < children.length; i++) {
      if (children[i] == currentNode.previous) {
        continue
      }
      children[i].moves = currentNode.moves + 1
      // moves property is the g(n) function, and it stores the
depth of this node i.e. distance from
root.
      openList.insert(children[i])
    }
    closeList.insert(currentNode)
  }
}
```

2.2 Finding the Children of a Particular Node:

In addition to the openLists' basic priority queue functions like insert() and delete(), we have also used the getChildren() function in **2.1**'s pseudo-code. Intuitively, finding the children of a particular node is quite easy. We just swap the blank tile with one of its adjacent tiles horizontally or vertically. But, how do we translate this into code?

First of all, let us consider how a puzzle board is represented in my code. For this, we shall only be considering the board's variables, not its functions:

```
class Board {

variable board

variable moves

variable previous

}
```

Writing the above class as a set:

$$B = \{b, m, p\}$$

b is 2-D array that stores the configuration of the digits and the blank tile of the board.

m is the number of moves it takes to reach this board. Essentially, this is the depth of the board, as explained in **2.1**. It is originally set to 0.

Finally, p is the board from which this board came into being. It can be thought of as the parent of this board. It is originally set to NULL.

To get the children of a particular board, first we have to locate the indices of the blank tile. Once the location has been located, we have to add and subtract 1 from its row and column index one by one (not simultaneously). This subtraction essentially amounts to moving the blank tile up and down, left and right. After this subtraction, we have to check if the new index is within the range of our grid i.e. if newIndex >= 0 AND newIndex < grid.length. If the newIndex is out of bounds, then we simply ignore it. If, however, the newIndex is legal, we swap the value that is currently present at newIndex with the blank tile. So, now, the blank tile is present at the newIndex while the digit that was previously at new tile, is now at the previous position of the blank tile. This new configuration of the board is a child of the original board. Now, we set the previous property of this board equal to the currentNode.

2.2.1 Pseudo-Code for Finding the Children of a Node:

NOTE: Here, '0' represents the blank tile.

```
 getChildren(parent) {

    children = [] // initialized as empty array

    zeroRow, zeroColumn = this.getZeroIndices(); // returns the
indices of the blank tile

    moves = [-1, 1] // Vertically, -1 means up, 1 means down.
Horizontally, -1 means left. 1 means right.

    for (i = 0; i < moves.length; i++) { // loop for horizontal
moves
```

```
      potentialChild = parent.board // initialize a potential
child's board with the same board as the parent.
      newZeroRow = zeroRow + moves[i]
      newZeroColumn = zeroColumn
      if (newZeroRow >= 0 && newZeroRow < this.board.length) {
        temp = potentialChild[newZeroRow][newZeroColumn]
        potentialChild[newZeroRow][newZeroColumn] = 0
        potentialChild[zeroRow][zeroColumn] = temp
        child = new Board(potentialChild) // create a new child
with the board stored in potentialChild
        child.previous = parent
        children.insert(child)
      }
    }
    for (i = 0; i < moves.length; i++) { // loop for vertical
moves
      potentialChild = parent.board
      newZeroRow = zeroRow
      newZeroColumn = zeroColumn + moves[i]
      if (newZeroColumn >= 0 && newZeroColumn <
this.board.length) {
        temp = potentialChild[newZeroRow][newZeroColumn]
        potentialChild[newZeroRow][newZeroColumn] = 0
        potentialChild[zeroRow][zeroColumn] = temp
        let child = new Board(potentialChild)
        child.previous = parent
        children.insert(child)
      }
    }
```

```
    return children // returns an array consisting of elements
of the type Board.

  }
```

2.3 Determining If a Node is the Goal Node:

The condition that stops our A* algorithm is if we have reached the goal node. To check that, we have used the function isGoal(). It works the following way:

```
isGoal() {

  goalBoard = this.getGoalBoard()

  return this.getNumberOfMisplacedTiles(goalBoard) == 0

}
```

This function is quite self-explanatory. It checks if the number of misplaced tiles, when compared to a goal node is equal to zero or not. If it is, then that means that our current node is equal to the goal node, and we return true. In any other case, we return false. To check the number of misplaced tiles, we use the following function:

NOTE: Although this function calculates the goalBoard by itself in the original code. But, in order to keep the focus on its main function, I have decided to pass the goalBoard as an argument:

```
getNumberOfMisplacedTiles(goalBoard) {

    count = 0;

    for (i = 0; i < this.board.length; i++) {

      for (j = 0; j < this.board.length; j++) {

        if (this.board[i][j] != goalBoard[i][j]) {

          if (this.board[i][j] == 0) {

            continue // continue if we encounter a blank tile

          }

          count++ // increase count if the tile does not match
the goal board tile

        }

      }

    }

    return count // return the total number of mismatched tiles
(not counting the blank tile)
```

```
    }
}
```

2.4 A* Simulation on a 3x3 Grid:

Let us apply the A* algorithm given in **2.1** to the start node given below:

Figure 2.4-1: Start and Goal nodes of our simulation.

The A* simulation is given on the next page as it is unable to fit in the current page:

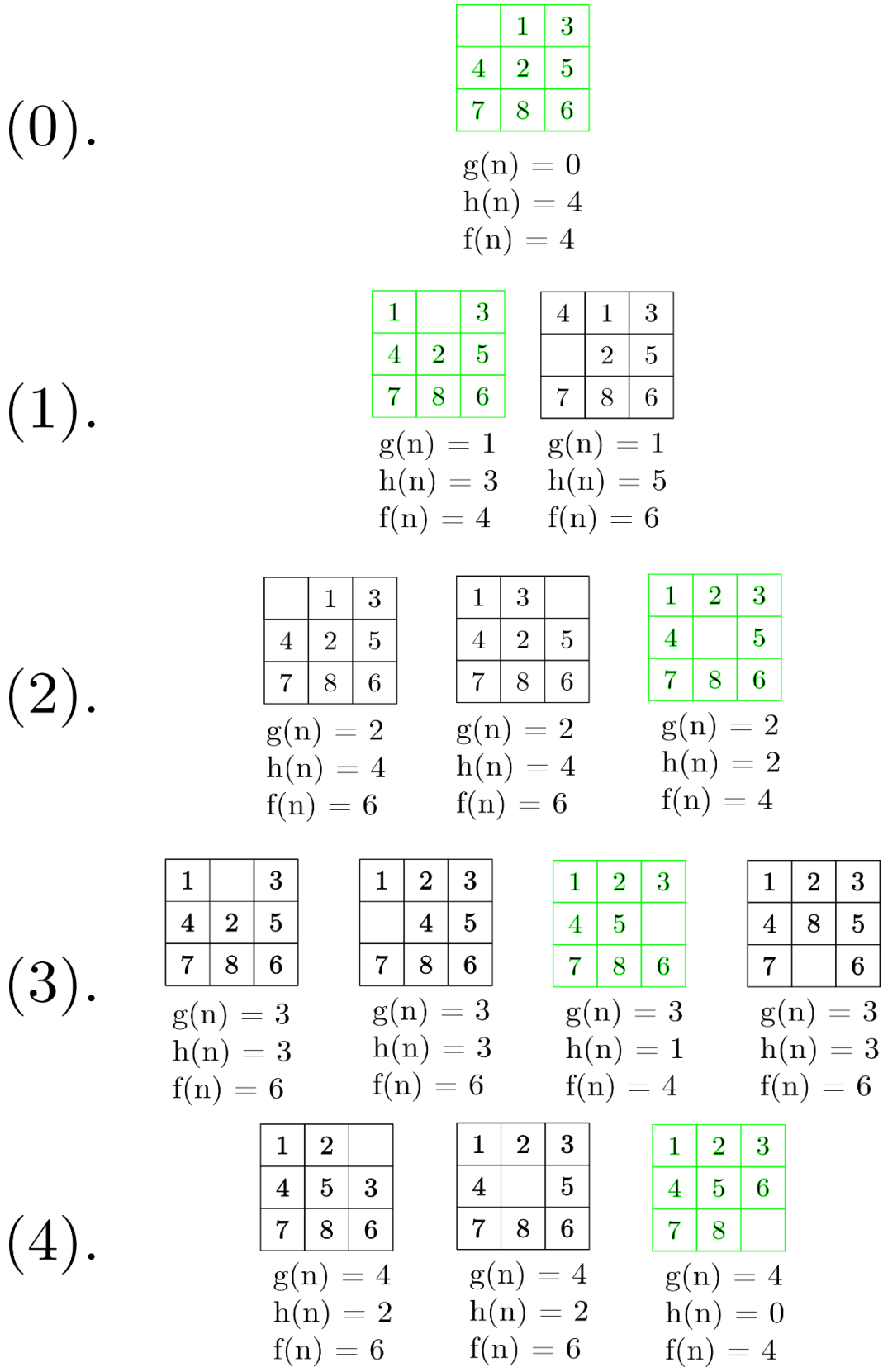

Figure 2.4-2: A* simulation on the start node given in Figure 2.3-1.

Let us look at the simulation in detail. At depth d = 0, we are starting with the start state. As it is the only state at d = 0, we choose it and expand it.

At d = 1, we have the child nodes of the starting node. We calculate the f(n) of each node and choose the node with the lowest f(n). Let us see how f(n) is calculated:

$$f(n) = g(n) + h(n)$$

where g(n) = the number of moves made so far to reach node n. Essentially, this is the distance of the node n from root i.e. the depth.

and h(n) = the number of misplaced digits when compared with the goal node. Note that while calculating the number of misplaced tiles, we do not consider the blank tile. One way to calculate it would be:

$$h(n) = (n^2 - 1) - (\text{number of tiles that match the goal node})$$

Or we can simply just count the number of tiles that do not match the goal node by using a loop.

At each level, the node that gets selected to be expanded is represented with green outlines. At each level, we choose the node with the lowest f(n) value and expand it. Suppose at d = k, we choose node n. Then, at d = k + 1, all the nodes are children of node n.

When we encounter a node with h(n) = 0, meaning it has no such tiles that do not match the goal tile i.e. it is the goal tile. That is when we stop our simulation. The path within the tree represented by green-outlined notes is the sequence of moves that lead from the initial node to the goal node.

Although all possible nodes have been represented in the above simulation and their f(n) is calculated for the sake of completeness, but, practically, if a node n is such that the parent of n is equal to the parent of the parent of n, then it is excluded since it has already been expanded. The piece of pseudo-code in **2.1** that performs that function is the following:

```
if (children[i] == currentNode.previous) {

  continue

}
```

3 Determining the Solvability of a Puzzle:

The simulation performed in **2.4** led us from the initial state to the required goal state. But, the unfortunate fact of the matter is, not all initial states can be transformed into the goal state through a sequence of legal moves. What is fortunate, though, is that we can detect whether a puzzle is solvable without attempting to solve it. Let us introduce the concept of *inversions*.

3.1 Inversions:

Given a grid of numbers for a node, an *inversion* is any pair of tiles i and j where i < j but i appears after j when considering the node in a row-first order.

Let us take the example of our starting node in **2.4** and calculate its inversions. As we have to consider the grid row-first, for simplicity, we shall lay out our grid in vector (1-D) form:

A = [1 3 4 2 5 7 8 6]

Now, we have to calculate the inversions for each A_i where (0 <= i < n). To calculate the inversions, for each A_i, if there exists an A_j, where (i + 1 <= j <= n), such that $A_j < A_i$, then we increase the inversions for that A_i by 1.

For each A_i, let us calculate the inversions:

1: 0 inversions (no A_j found such that $A_j < 1$).

3: 1 inversion (2 < 3)

4: 1 inversion (2 < 4)

2: 0 inversions (no A_j found such that $A_j < 2$).

5: 0 inversions (no A_j found such that $A_j < 5$).

7: 1 inversion (6 < 7)

8: 1 inversion (6 < 8)

6: 0 inversions (no A_j found such that $A_j < 1$).

NOTE: The last element will always have 0 inversions. It is trivial to see that there exist no elements after the last element, and as such, for $A_i = A_{n-1}$, the first value of j will be j = i + 1 = (n – 1) + 1 = n. As, for an array of length n, the elements are in the range 0 — (n – 1). Therefore, $A_j = A_n$ will be out of bounds and the loop will be immediately terminated. However, for the sake of completeness, the inversions for the last element 6 are listed here too.

The total amount of inversions our puzzle in **2.4** has is 4. But, how does inversions help us determine whether a puzzle is solvable?

3.2 Determining Solvability through Inversions:

Now that we have calculated the inversions, we are in the position to determine the solvability of our puzzle.

For a node of odd-size, where n mod 2 != 0, if the number of inversions is even, then the puzzle is said to be solvable. If the number of inversions is odd, then the puzzle is solvable. For our puzzle in **2.4**, n = 3, while the number of inversions = 4 (even). So, our puzzle is solvable.

What is the logic behind this? Well, for an odd-sized node, each move changes the number of inversions by an even number. Since the number of inversions in the goal node is even (0), if the number of inversions in our starting node is odd, it cannot possibly lead to the goal node. Since, again, the goal node has an even number of inversions, equal to 0. However, if the board has an even number of inversions, then it is possible that a sequence of legal moves shall lead from the initial node to the goal node.

For an even-sized node, the above logic requires some slight modification. This is because the *parity* of inversions will not always remain the same, as is the case with an odd-sized node. Each move may change the number of inversions by both an odd or an even amount. However, the sum of the number of inversions and row-number of the blank tile will always remain the same. Using this, we can detect if an instance of even-sized node is solvable or not. If the sum is odd, then the puzzle is solvable. Otherwise, if the sum is even, then the puzzle is insolvable.

Let us check that for the following even-sized node, n = 2

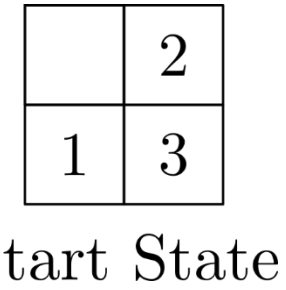

Figure 3.2-1: Starting even-sized node.

Laying it out vertically:

A = [2, 1, 3]

Calculating the inversions:
2: 1 inversion (1 < 2)
1: 0 inversions (No Aj found such that Aj < 1)
3: 0 inversions (No Aj found such that Aj < 3)

The total number of inversions is 1. The row-number of the blank tile is 0. Adding them up, we get 1. As 1 is odd, hence, the puzzle is solvable.

3.3 Pseudo-Code for Calculating Inversions:

Following is the pseudo-code of the actual code used in my project to calculate inversions:

NOTE: For the sake of simplicity, our n*n grid (2-D) is converted into a (1-D) array to make the calculation easier. Moreover, the blank tile is represented with ‘0’ in my code. As we do not consider the blank tile while calculating the inversions, there are some checks in the following pseudo-code to account for that.

```
getInversions(arr) {
inversions = 0
  for (i = 0; i < arr.length - 1; i++) {
    if (arr[i] == 0) {
      continue
    }
    for (j = i + 1; j < arr.length; j++) {
      if (arr[j] == 0) {
        continue
      }
      if (arr[j] < arr[i]) {
        inversions++
      }
    }
  }
  return inversions;
}
```

3.4 Pseudo-Code for Calculating the Solvability:

Using the above pseudo-code for getInversions() and another trivial function getZeroRow(), that, as the name implies, returns the row of the blank tile, we can calculate whether a puzzle is solvable before attempting to solve it:

```
isSolvable(arr) {
inversions = getInversions(arr)
n = sizeOfGrid;
if (n % 2 != 0) {
  return inversions % 2 == 0
}
else {
```

```
  return (inversions + getZeroRow()) % 2 != 0
}
}
```

4 Conclusion:

The problem of solving a sliding-puzzle belongs to the complexity class of *PSPACE*, meaning that a computer can solve this problem while utilizing a polynomial amount of space. A* also has a worst-case performance complexity of $O(b^d)$, where *b* is the branching factor and *d* is the depth. It is exponentially dependent on the depth of the solution. [2] So, while the implementation of the algorithm works for any n-sized board, but as n increases, the algorithm requires huge amounts of space and CPU cycles to find the solution, which a conventional computer might not be capable of providing. But, for a 3x3 puzzle requiring 30 moves in the optimal solution, my implementation of A* is able to solve it in ~6s. Obviously, solving the puzzles requiring less moves is even faster.

References

[1] Russell, Stuart; Peter, Norvig; Artificial Intelligence – A Modern Approach; 3rd Edition

[2] http://sumitg.com/assets/n-puzzle.pdf