CSCI 4511W Final Project

Common Search Algorithms Implemented on the 8-Puzzle Problem

Sid Lin (linx1052) and Noah Park (park1623)

May 7, 2020

1 Abstract

This project is on the analysis of common AI search algorithms, as discussed in the first third of the class: CSCI 4511W, Introduction to Artificial Intelligence. In class, we learned that these algorithms can be used to solve some very common AI problem scenarios. The particular focus of our project is the sliding puzzle problem. The slide puzzle is not a trivial problem to solve. It also has many modern implementations, such as a CAPTCHAs and puzzles nested inside video games. The slide puzzle scenario can be used to research the different methods of search while having the possibility of being published into a mobile game in the future.

Two of the topics that were heavily covered in lectures were informed search and the optimality of a heuristic function. A* search is an extremely efficient algorithm to solve coding problems, and informed search is one of the first topics to understand when entering the world of artificial intelligence. This algorithm makes use of a heuristic function to give it "additional data" about the state space or environment. Using this tactic, we were able to produce an algorithm that produced an optimal solution every time during testing.

2 Problem Description

We focused primarily on the 8-Puzzle sliding tile problem. A sliding puzzle of size 8 is played on a three-by-three grid utilizing eight square tiles labeled one through eight. One of the tiles is removed as a blank tile. Without breaking movement rules, the tiles can only be moved one at a time if there is a space available adjacent to it. Traditionally, the puzzle must be moved into ascending order from top to bottom and left to right. However, a scenario of the 8-puzzle can have any initial and goal state. If the puzzle obtains the desired state just described, the 8-puzzle has been solved. The goal is to solve the 8-puzzle in as few steps as possible, with runtime as an additional but secondary benchmark.

3 Background Information

One of the most common introductory topics to AI are search algorithms. Search algorithms are one of the most basic and familiar topics for computer scientists beginning to expand their learning into the realm of AI. These algorithms can be split into two categories, uninformed search and informed search. Many search algorithms have practical applications with familiar problems, such as the 8 queens, traveling salesperson problem, map coloring, N x N slide puzzle, and much more. For our project, we will be focusing on the 8-puzzle with informed search.

The 8-puzzle scenario is none other than the familiar slide puzzle; each state consists of a N x N grid with all numbered tiles, aside from a single missing one. The goal of the puzzle is to move each tile into a certain order with the fewest moves and time. It may appear as if the moves are made by moving the numbered tiles into the empty space, but a simpler explanation is that the empty tile moves around the board. A board that is 3 x 3 will have 9! possible permutations, but only half of them are solvable. The sheer amount of possible permutations that make the 8-puzzle is an ideal candidate for an informed search analysis [**?**].

Each configuration of the 181,440 possible instances of the 8-puzzle was tested with an optimal iterative deepening algorithm [**?**]. Richard Korf's IDA* algorithm was used in conjunction with the Manhattan distance heuristic to solve each configuration. Each solution had more than one optimal solution, but the average number of steps to solve the puzzle was consistent. The maximum number of steps to solve the 8-puzzle was 31, and the average was about 22 [**?**]. The ideal formulation of a search space for this problem would be a graph [**?**]. However, our instance of the 8-puzzle consisted of nodes that created a search tree. Graphs would be ideal due to the ability to undo a move, but for simplicity we chose to use nodes instead of vertices and edges.

One of the most popular informed search algorithms is A*. It is an extension of Dijkstra's algorithm. The A* algorithm uses a cost function that takes both path cost and heuristic cost into account. Normally, A* search spaces are implemented with nodes, which can have children and a predecessor node. Most importantly, A* uses a priority queue ordered with the lowest function cost to determine each node's order of expansion. The standard function is denoted f(x) = g(x) + h(x), where g and h are the path cost and heuristic cost respectively [**?**].

The optimality of this algorithm depends on the admissibility of the heuristic function as well. Admissible heuristics are ones that will never overestimate the cost of reaching its goal. Furthermore, the other condition for optimality is known as consistency. Consistency, for every node n and every successor n' of n generated by any action a, is described as the estimated cost of reaching a goal from n being no greater than the step cost of getting to n' plus the estimated cost of reaching the goal from n'. This is also known as the triangle inequality [**?**].

With all informed search algorithms, there has to be some way of gathering more information from the search space; this is where the use of a heuristic makes these algorithms distinct. We chose to use a version of the Manhattan distance heuristic. The heuristic is traditionally the straight line distance from each tile to its goal location [**?**]. However, our implementation of the 8-puzzle was done in a one dimensional array, so we decided to use the index distance from a tile to its original place. The chosen heuristic is quite important, as it can change the way the A* algorithm will behave. Daniel R. Kunkle did extensive testing on heuristics with this problem. He noted that the most sophisticated heuristic is the Manhattan Distance, and that a heuristic of 0 will result in a BFS algorithmic behavior. It is important to choose the heuristic wisely when dealing with the A* algorithm [**?**].

Our implementation of the 8-puzzle was done in Swift and published on the iOS App Store. To measure the effectiveness of the A* search algorithm, we had a board randomization function with 4 levels of difficulty, correlated with the number of random moves made. Future advances on our solution could be led to genetic algorithms. An interesting take on this was found by Harsh Bhasin and Neha Singla. They found that by implementing the "natural selection" employment of the nature of genetic algorithms, many optimal solutions were found. They concluded that genetic algorithms were the best type of algorithm to solve these problems [**?**].

4 Problem Solving Approach

As said before, the implementation of the A* search algorithm and 8-puzzle board was done in Swift 5. First, brief overview of the board implementation will be shown below. We created a custom class to store data on the game board. The board was a one-dimensional array, as opposed to a two-dimensional array, of these objects.

```
class MyLabel: UILabel {

   // Value of the tile (1-8)
   var data: Int = 0

   // Current position on the grid
   var currentPosition: Int = 0

   // Goal state position
   var originalPosition: Int = 0

   // Check if the tile is the empty tile
      func isEmptyTile(length: Int) -> Bool {
      return data == length
   }

   // Return if the tile is in the correct plate
   func isInPosition() -> Bool {
      return currentPosition == originalPosition
   }
}
```

Random generation of the board was either done in one pass, or by making a certain amount of random moves starting with the initial state. We will not be covering the initialization of the tiles or randomization, as it does not pertain to the A* algorithm.

We also used a custom Node class with much more complexity to make properties accessible by the A* algorithm. A partial implementaiton will be shown below. In this case, a Node represents the current state of a board, as if the search space was a bidirectional tree or graph.

```
class Node {
// Current state of the node and the solution
var state: [Int]
var solution: [Int]
```

```
// Edges used for the possible move set
var leftEdgeNodes: [Int]
var rightEdgeNodes: [Int]

// Parent node used for backtracking
var parent: Node?

// Heuristic score
var hScore: Int = 0

// How many moves have been taken
var gScore: Int = 0

// F = G + H in A*
var fScore: Int = 0

// Constructor
init(state: [Int], level: Int, parent: Node?) {
   self.state = state
   self.gScore = level
   self.parent = parent

   leftEdgeNodes = [0, 3, 6]
   rightEdgeNodes = [4, 7, 10]
   solution = [1, 2, 3, 4, 5, 6, 7, 8, 9]
   hScore = manhattanDistance()
   fScore = gScore + hScore
}

// Generate child nodes from the current state
func generateChildren() -> [Node] {
   let moves = generatePossibleMoves()
   var nodes: [Node] = []
   // 9 is used as empty space
   let indexOf9 = state.firstIndex(of: 9)!

   // Iterate through the moves and create a node
   for i in 0...moves.count - 1 {
      var tempState = state
      tempState.swapAt(moves[i] - 1, indexOf9)
      nodes.append(Node(state: tempState, level: gScore + 1, parent:
          self))
   }
   return nodes
}

// Generate all valid moves for the empty space as a list of integers
func generatePossibleMoves() -> [Int] {
   // Very long implementation
}

// Distance from actual heuristic with absolute value
func manhattanDistance() -> Int {
   var mhd: Int = 0
   for i in 0...solution.count - 1 {
      let temp = state[i] // Get element at position i
      let actualLocation = i + 1
      mhd += abs(temp - actualLocation)
   }
   return mhd
}
```

Finally, we have the Puzzle class, where the A* algorithm itself is implemented. Using pseudocound found in textbooks and on the internet, we transcribed it into Swift. Only the algorithm will be shown. To reduce runtime, we chose to implement a closed list so the algorithm would not run into infintie loops. To summarize the code below, the algorithm simply sorts the Nodes in the open list and dequeues the Node with the lowest function score. Then, we generate the successor nodes and add them to the open list and run until a solution is found or every state has been used.

```
func AStar() -> [Int]? {
   while open.count > 0 {
   // Sort open list based on lowest f-score
      open.sort { $0.fScore < $1.fScore }
      turns += 1

   // "Dequeue" the node at the start
      let dequeued: Node = open[0]
      currentNode = dequeued
      open.remove(at: 0) // Remove from open list
      openStates.remove(at: 0)

      if dequeued.isSolution() {
      print("Solution found with number of turns: \(turns)")
      return generateSolution()
   }

   let children: [Node] = dequeued.generateChildren()
   for child in children {
      let idxOpen = openStates.firstIndex(of: child.state)
      let idxClosed = closedStates.firstIndex(of: child.state)

      // If the open list contains the child and the existing f-score
          is less
      if idxOpen != nil && open[idxOpen!].fScore < child.fScore {
        continue // Discard the node
      }
      // If the closed list contains the child and the existing
           f-score is less
      if idxClosed != nil && closed[idxClosed!].fScore < child.fScore
           {
        continue // Discard the node
      }

      // Add the child to the open list
      open.append(child)
      openStates.append(child.state)
      }
      // Add the expanded node to the closed list
      closed.append(dequeued)
      closedStates.append(dequeued.state)
    }

    // After the while loop ends
    if !currentNode.isSolution() {
      print("A star failed - no more nodes in the open list!")
      return nil
    }
    else {
      return generateSolution()
    }
  }
```

5 Results and Analysis

To test our algorithm, we ran it 100 times on a randomly generated search space. The search space was created by starting with an initial array of numbers, and randomly selecting one at a time and placing it on the board starting at tile 1. Actual time to solve the problem was not taken into account. However, it should be noted that some trials took minutes for the algorithm to find the solution, even if the amount of steps to solve it were around the average. The A* algorithm is not efficient in terms of runtime, and the statistic we are measuring is the optimality of the solution, not the time it takes to generate it. The results of 100 trials are shown below.

Number of Steps	Occurances
22	1
23	5
24	7
25	15
26	16
27	23
28	15
29	9
30	7
31	2

6 Conclusion and Future Research

The use of the Manhattan Distance heuristic produced results that do not reject the conclusions made by other researchers. It has been proven that the maximum moves to complete the 8-puzzle from any state is 31, and none of our results exceeded this number. Our results are approximately normally distributed, with an average of about 27 moves. This is a few more than the average of 22 discussed earlier [**?**].

The algorithm could be improved in the future by adding new heuristics or adding optimizations to improve runtime. Currently, our algorithm has an exponential runtime with many O(n) helper functions. In addition, the same board could be used with different algorithms. The same one-dimensional array state space can be used with algorithms such as BFS, DFS, iterative deepening DFS, bidirectional search, a genetic algorithm, and much more. The possibility of implementing different algorithms could also be turned into a "difficulty" option for players to eventually compete against the AI, each making one move at a time.

The published application can be found here: https://apps.apple.com/us/app/ai-slider/id1504773369.

7 Contributions

Sid worked primarily on the programming portion and derived the idea for using the 8-Puzzle problem. Noah focused mostly on the research side of the project and assisted in the testing of the testing of the algorithm.