Solving the 8-Puzzle via Optimizing Applications of the A* Algorithm

Project 8: Puzzle

Elven Shum, Kevin Gu, Alex Yuk

Jan 26 2020

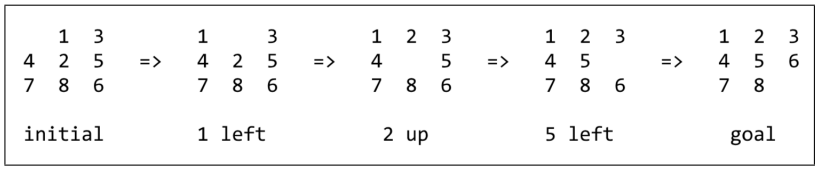
1 Introduction and Background

In this project, we endeavor to solve the 8-Puzzle by utilizing the A* Algorithm. Throughout this process, not only do we implement a successful A* Algorithm for solving the puzzle, but we optimize the algorithm by applying several clever tricks.

8-Puzzle: Briefly Explained

The puzzle is played on a 3x3 grid with 8 tiles, and a blank tile. To solve, we must rearrange the tiles in-order–using only moves that slide into the blank tile. An example path can be found in Fig 1.

Figure 1: Example Solution Path of 8-Puzzle



An ideal 8-Puzzle solver would find the optimal–least number of steps–solution, but also be able to solve any (solveable) 3x3 board presented to it. Ultimately, we extrapolate this to an NxN board.

A* Algorithm: What and How

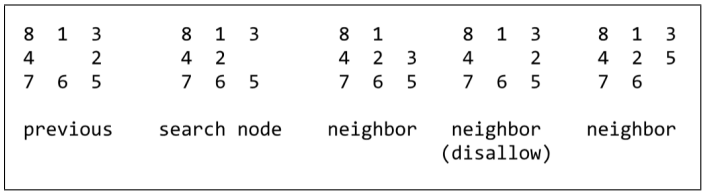
While we understood the 8-Puzzle solving process from the 8-Puzzle Programming Assignment, we kept asking ourselves: *So, what actually **is** the A* Algorithm?* Yes, we knew it was a "a graph traversal and path search algorithm," but we were confused with what differentiates and qualiies an A* Algorithm? In this subsection, we explain both the A* Algorithm fundamentals, but also our process/application.

The A* Algorithm is comprised of three fundamental components:

Closed List One of the A* Algorithm's key features: it remembers the visited nodes. For this problem, think about each "board" as a node. This way, it saves massive time by removing redundant searches. All already-searched nodes (think: test isSolution) comprise the Closed List.

Open List The algorithm also holds a changing list of unexplored nodes. In our example, the unexplored nodes are derived from the Closed List.The derived-open list is simply neighbors of the most closed-list. See Fig 2. Then, the most optimal node from the Open List is chosen to be evaluated. That optimal-ness is based upon the following:

Figure 2: Deriving the Open List from the Closed List, Using Neighbors



f-score Heuristic Ultimately, the A* algorithm's speed is primarily due to priority queues. Using the f-score, the program can determine which node is the most promising, and thus evaluate it first–rather than blindly choosing nodes. A* selects the node which minimizes their f(n), which is defined as

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

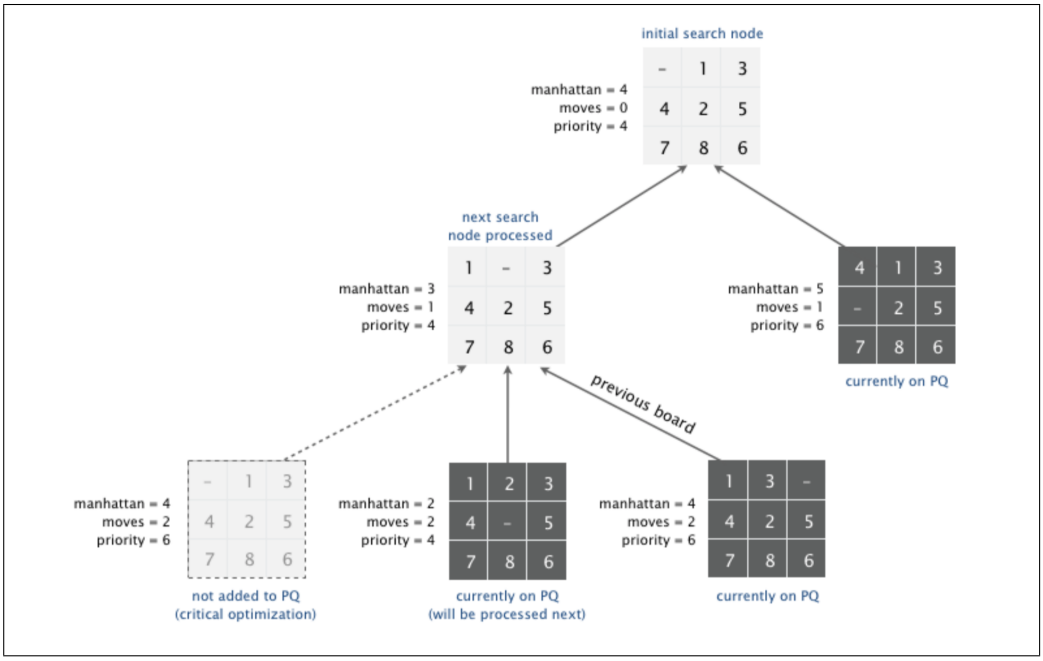
where n is the next node on the path, g(n) is the number of nodes since the start (distance from eldest parent), and h(n) is a heuristic function for "distance to the

goal". This composite f-score is fundamental to A*, requiring both a h(n) "success metric" and also the current g(n) cost of the current node.

For our current problem, our required user-defined h-heuristic was the sum of a node's Manhattan distance and Hamming distance.

- Manhattan distance: the sum of all tile's taxi-cab distance (horizontal + vertical distance) to their solution-position.
- Hamming Distance: the number tiles in the wrong position–equivalently, current board's difference from the solution.

Figure 3: Example Solving Tree Path



To solve, we continually expand the Open List, pushing non-solutions to the Closed List, and check the current state. The state with the least f-score is processed every time, as this will reduce redundant computations. Fig 3 is an example ofa possible solved "Tree Path" Like in Fig 2, we determined our Closed-List neighbors based off the potential moves around the blank tile.

2 Further Optimization Extensions

2.1 h-score validation

In addition to simply creating a solver, we sought to determine possible optimizations– with our first and simplist test being checking different h-score heuristics.

Table 1: Contents of Each Solution for Examining Photosynthesis's Kinetics

h-score heuristic	**Ave. Number of Queues**
Manhattan Only	1852.4417
Hamming Only	1455.3191
Both	1454.2034

With our current system, it's clear that both the Manhattan and Hamming Distance applied together results in the lowest average number of queues; however, the Hamming distance alone results in a marginally worse addition, while Manhattan alone results in a nearly 30% increase. This might indicate that the Hamming distance is doing most of the efficiency heuristic work. These were the best two distance metrics we could find. Through this test, we determine that none of our h-score components are doing negative work, ie hindering the system.

2.2 Disallow Redundant Neighbors

An optimization that significantly reduces speed include what was eluded to in Fig 2. Namely, the exclusion of the "previous node state." A simple A* algorithm would simply calculate all the neighbors of the current search node; however that results in redundancies because it inherently checks a node that's already been placed into the "Closed List". Thus, by disallowing that search, we reduce computations significantly–loosely estimated to a 30% decrease.

2.3 Caching the Manhattan Distance

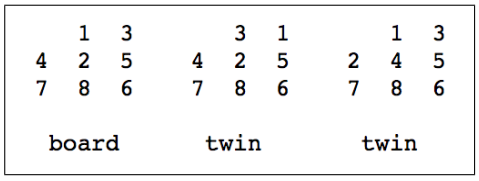
Calculating the Manhattan distance from scratch each neighbor search consumes much time. To reduce this, we cache the Manhattan Distance from the previous board state as an instance variable and simply edit it for our new boards.

3 Further Conceptual Extensions

Solvability & the Twin Method: an intuitive proof

Interestingly, not all boards are solvable. The way our Problem Sheet solves this is using the Twin() method, where it produces a "Twin" of the input board. See Fig 3 for an example. Any of those permutations around the current board are "Twins".

Figure 4: Possible Twin Examples



Our Sheet asserts: Of the Board and it's Twin, expactly one is solvable
We asked: *how come? That's not intuitive* We developed a proof for why.

Theorem: Exactly one of a "Board" or it's "Twin" are solvable.

Proof: Without loss of generality, let any board be represented equivalntly as a list:

$B_{1,1}$	$B_{1,2}$	$B_{1,3}$
$B_{2,1}$	$B_{2,2}$	$B_{2,3}$
$B_{3,1}$		$B_{3,3}$

$$\Vert$$

$$\{B_{1,1}, B_{1,2}, B_{1,3}, B_{2,1}, B_{2,2}, B_{2,3}, B_{3,1}, B_{3,3}\}$$

Note: the omission of the blank tile.

Let an "Inversion" be defined as a a switch of any 2 elements. Ex:

$$\{1,2,3,4\} \xrightarrow{\text{1 inversion}} \{3,2,1,4\} \xrightarrow{\text{1 inversion}} \{3,1,2,4\}$$

We can define board's in relation to each other by distance, based on number of non-redundant inversions. Let's define the "Origin Board" to be the solved

$$\{1,2,3,4,5,6,7,8\} \equiv 0\, inversions\, from\, Origin$$

By observation, all board states are either and Odd or Even number of inversions away from the Origin Board. Boards are either even number of inversions away, called **Even Boards**; or an odd number of inversions away, called **Odd Boards**.

To "Solve" a board, we have a total of 2 moves:

Horizontal Moves

1	2	3
4	5	6
	7	8

$\xrightarrow{\text{Horizontal Move}}$

1	2	3
4	5	6
7		8

$$\{1,2,3,4,5,6,7,8\} \xrightarrow{\text{Horizontal Move}} \{1,2,3,4,5,6,7,8\}$$

Notice that for all Horizontal moves, the List Representation doesn't change. Thus All Horizontal Moves don't change a board's Parity (even-ness or odd-ness).

Vertical Moves

$$\{1,2,3,4,5,6,7,8\} \xrightarrow{\text{Vertical Move}} \{1,2,3,5,6,4,7,8\}$$

Notice that for all Horizontal moves, the list representation changes as such:

$$\{...x,y,z...\} \xrightarrow{\text{Horizontal Move}} \{...y,z,x...\}$$

This move is actually equivalent to two single inversions, namely:

$$\{...x,y,z...\} \xrightarrow{\text{inversion}} \{...y,x,z...\} \xrightarrow{\text{inversion}} \{...y,z,x...\}$$

Thus All Vertical Moves don't change a board's Parity.

We have shown that given our two moves(Vertical and Horizontal), we're unable to change a board's Parity. Thus, when provided with a Even Board, it's solvable. When provided an Odd Board, it's Twin will be solvable. Since a Board, Twin pair must contain both even and odd board, either a board's Twin or itself is solvable. Q.E.D

Pre-calculating the Parity: Abusing Twin

Currently, as prescribed by the Problem Sheet, we alternate between searching between a Current Graph and it's Twin. This results in a solution search that requires 2x as much computation than necessary. To reduce the computations by half, we will determine whether a board is Solvable or Not-Solvable, Even or Odd–it's Parity. Then solve the Original Board, or the Twin accordingly.
This property is in our isSolvable() Method, but doesn't yet determine which board we computer–ie, we still compute both. For Future Otimization, utilize this.

3.1 Bidirectional Search

Alternatively, a bidirectional search could reduce our times significantly. This means, instead of only searching from our Origin Board, we also simultaneously search from the solution. Once we find matching boards, we've discovered the solution