An Application Using Artificial Intelligence

This page was written by Josh Richard

What is an 8 Puzzle?

1	2	3
4	5	6
7	8	

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty.
The object is to move to squares around into different positions and having the numbers displayed in the "goal state".
The image to the left can be thought of as an unsolved initial state of the "3 x 3" 8 puzzle.

What is an 8 Puzzle Program?

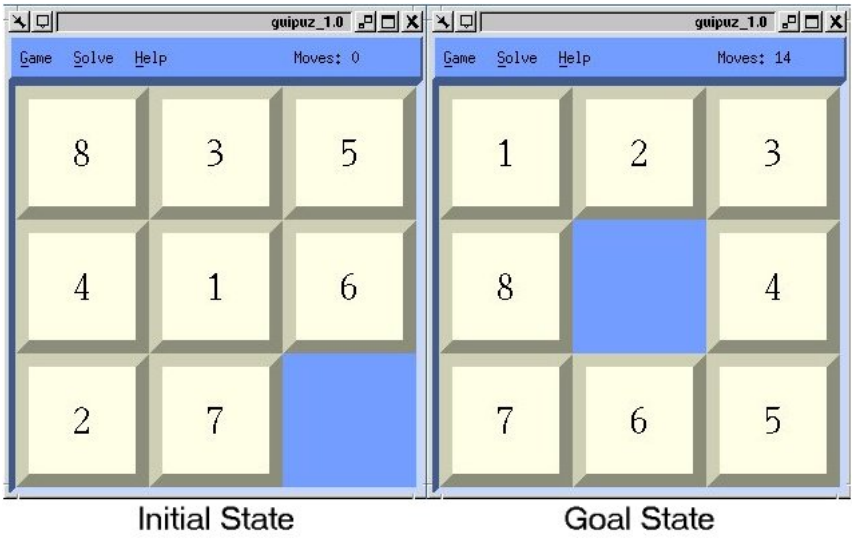
The 8 puzzle program was written as a 2-person project for Dr. Tim Colburn's Software Development course (CS2511) by Brian Spranger and Josh Richard. The assignment was to write a program that is intelligent enough to solve the 8-puzzle game **in any configuration, in the least number of moves.** We were also to provide a Graphical User Interface (GUI) to the user to make the program easier to use. The Intelligent engine was written in C++ and the GUI was written in Motif/X11 converted from standard C to use encapsulated Object Orientated Design Constructs.

Possible "Game States":

```
Goal:       Easy:       Medium:       Hard:       Worst:

1 2 3       1 3 4       2 8 1         2 8 1       5 6 7
8   4       8 6 2         4 3         4 6 3       4   8
7 6 5       7   5       7 6 5           7 5       3 2 1
```

The image below shows the finished program with a random start state and correct goal state displayed.



Here is the shortest solution generated by the 8 puzzle program for the initial state given above. (The program animates through each game state)

```
+-----+
|8 3 5|
|4 1 6|
|2 7  |
+-----+

+-----+
|8 3 5|
|4 1  |
|2 7 6|
+-----+

+-----+
|8 3  |
|4 1 5|
|2 7 6|
+-----+

+-----+
|8   3|
|4 1 5|
|2 7 6|
+-----+

+-----+
|8 1 3|
|4   5|
|2 7 6|
+-----+

+-----+
|8 1 3|
|  4 5|
|2 7 6|
+-----+

+-----+
|8 1 3|
|2 4 5|
|  7 6|
+-----+

+-----+
```

```
|8 1 3|
|2 4  |
|7 6 5|
+-----+

+-----+
|8 1 3|
|2   4|
|7 6 5|
+-----+

+-----+
|8 1 3|
|  2 4|
|7 6 5|
+-----+

+-----+
|  1 3|
|8 2 4|
|7 6 5|
+-----+

+-----+
|1   3|
|8 2 4|
|7 6 5|
+-----+

+-----+
|1 2 3|
|8   4|
|7 6 5|
+-----+
```

```
Total Number of moves: 14
```

Problem Approach

The complexity of possible moves toward the final solution in a game like this is great. It can take an average computer great lengths of time to find the correct sequences for a particular configuration of the 8 puzzle game if the search method is "blind". The problem is how do you make the computer order the moves intelligently to find the shortest path to the winning game state? Solving a problem such as this can be done two ways:

1. Guess through every possible state by doing a blind search of the state space.
2. Implement a search operation that is guaranteed to find the shortest solution using "informed methods" (How?).

Informed Search Strategies

Informed methods help us gain information about solving a problem through its current state space. This keeps a program from blundering around blindly guessing. Informed search strategies make use of information about the path of moves we took to get where we are currently by using an "evaluation function".

Game State Expansion:

In our solution design, we coded an expansion routine that takes the current state of the game and expands it to the next possible moves for any given state. If you use your imagination, the shape of the expanded nodes looks like an upside down tree. Each level of the tree is called "depth"

(fig.a) Click for a Larger View

(fig a): The numbers next to the game states represent the number of tiles out of place for the given game state. The lines represent the search depth of the tree. The solving algorithm (heuristic) to set up move ordering intelligence uses these values to become more"informed".

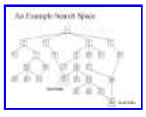
An uninformed search would expand about:

$$3^{20} = 3.4 \times 10^{9} = 3.4 \text{ Billion game states.}$$

Thus a good heuristic is necessary in order to reduce the number of game states that are evaulated.

Heuristic Value Calculations:

In order for the game to be able to solve itself, we need to approach the solution with an idea of how to differentiate between a good move and a bad one. This is called a heuristic. Our heuristic was to order the moves based on the number of tiles out of place. We also took into consideration how far away the out of place tiles was from their correct position (called Manhattan distances). This entire process determines the heuristic value (H). We can then take H at any given time and successively find the path to a solution. But this will not provide us with the shortest solution. See figure:



(fig.b) Click for a Larger View

Notice that the goal state can be achieved in 3-6 in this example. If you recall we want to solve this in the least number of moves so we can achieve a higher sense of accomplishment. This will also allow us to solve more complicated puzzles. How do we make this more efficient?

A* Value and Priority Queues:

The answer to the question posed is to refine the approach used to determine the next move. If we think about fig. b again, we can see that winning state occurs in two (or more) places in the game tree (states 22, and 28). We can order these moves with the help of a special queue that orders based on priority. The priority is determined by taking the current heuristic value and adding it to the depth. This gives us our A* (a star) value for a particular state. If the queue is ordered properly (lowest first), the next best move is always "next in line" for expansion. This process will always yield the correct solution in the least number of moves, as long as the puzzle is initially a solvable state.

A* Searching Algorithm:

The A* searching algorithm is a recursive algorithm that continuously calls itself until a winning state is found. When the game is designed, in order to randomly generate different games, start with the goal state and make many random moves on the state.

```
Where:
       n is the number of move directions (4) and the moves
       made on a particular state are valid.

This Yields a simple formula:

       move = (rand() Modulus n)
           i
or in C++:
       srand(time(NULL)); // Or use a large prime to see rand()

       for(int i= 0;i<100;i++){
         move = (rand() % 4);
         if (is_valid(move,state))
           make_move(move, state);
       }// end loop
Where:
       move is an enumerated type and is_valid is a boolean
       function that determines the validity of a move based on
       the current state.  If the move is valid, just perform
       the move on the state and continue looping.
```

This way, you can be sure to have a game state that can be solved This will keep you from having an algorithm that gets lost in its own thoughts and never evaluates to true (infinite recursion).

Algorithm Pseudocode

```
boolean
Solution::search(PriorityQueue pq)
{ if pq.isEmpty() then
   return false
puz = pq.extract()
if puz.isGoal()
   return true
successors = puz.expand()
           // all possible successors to puz
for each suc in successors do
   pq.insert(suc)
if search(pq)
   return true
else
   return false
}
```

Slide Source :Dr. T. Colburn

Visit my homepage