UDC 004.02:004.832.2

O.I. Molchanovskiy, S. Nakhostin

USING ARTIFICIAL INTELLIGENCE METHODS TO SOLVE BOSS PUZZLE (USING A SINGLE AGENT)

Introduction

In this report we tried to give a comprehensive, practical method for solving the $(n^2 - 1)$ puzzle. It is assumed in advance that this algorithm is going to be realized as a computer program in a Rule Based System.

The $(n^2 - 1)$ is the generalized form of the famous 15 puzzle and is the subject of Ian Parberri's paper under the title of: "A real time algorithm for $(n^2 - 1)$ puzzle" [2] in which he uses a divide and conquer (D&C) approach to rearrange a solvable grid position. A solvable grid position is the one with an even permutation of moves from the beginning (initial) state to the goal state.

In 15-puzzle we have a 4x4 grid with 15 tiles numerated from 1 to 15 and a hollow position to which we refer as space. The agent's mission is to scramble the grid and then start to rearrange it by moving the space in 4 directions (moving the space is analogous to pushing the neighbor tiles into the space so whenever we say move the space for example to the right it means that we push the right neighbor of the space into the space's position). As the result we reach to the final arrangement of the grid which is shown in the Fig. 1.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	S

Рис. 1 – Goal position of 15-puzzles

It is proven that half the number of all possible initial states, are led to the final (goal) state. If we generate the initial state randomly we can then test the evenness of the initial state (even permutation of moves from the initial state to the goal state) to find out that the given grid configuration is solvable or not. This test is represented in the Sam Loyd 15-puzzle paper written by Richard Hayes [1].

The $(n^2 - 1)$ algorithm is based on two standard algorithm design techniques: divide and conquer, and the greedy algorithm has the following features:

- it is a real-time algorithm, that is, it generates a series of moves with $O(1)$ computation time required before the first move, and between each successive move;

- it makes no more than 5 times more moves than necessary on the worst-case configuration;
- it makes no more than 7.5 times more moves than necessary on average configurations;
- it makes no more than 19 times more moves than necessary on random configurations, with probability one (we follow the popular shorthand of writing with probability *one* for a probability that approaches 1 as n increases. In our case, the probability approaches 1 exponentially fast) [2].

Here we are not going to peruse the Ian Parberry's paper and eschew getting involved with proving theorems which are used in his work. What we are interested to represent is a complete algorithm (with all underlying nuances) in a more practical approach to make us enable to build an intelligent agent to scramble and solve all possible configurations of a grid with any given dimension.

It is assumed that this algorithm is going to be realized in a rule-based architecture, so being acquainted with the basic aspects of these architectures is required.

The algorithm's outline

We can divide the algorithm into 4 phases; each phase can be represented by a separated function:

1) Getting the dimension of the grid.
2) Scrambling the grid and setting the initial state.
3) Test for even permutation.
4) Solving the puzzle for the initial state.

First phase

For this phase the dimension (size) of the grid is being defined using a simple input function. This we can draw the grid with the given number of rows and columns.

Second phase

Using a randomization function, the initial state of the grid from 1 to n (grid dimension) is being generated.

Third Phase

Using a separated function, we can check the evenness of the initial state. This function must have two sub functions.

1) For the state in which space is in its eventual (goal) position.
2) For the state in which space is not in its eventual (goal) position.

Forth Phase

This phase is composed of sub phases.

The fundamental approach to solve this part is based on Divide and Conquer (D&C) method which is represented in Sam Loyd's paper and we

 ISSN 1562-9945

implement it as the outermost function which is being used to sort the whole grid (by frequently reducing the size of the grid).

Important point: in practice we can skip second and third phases by scrambling the sorted grid manually.

This, we can be assure that our initial state is always solvable (even number of permutations).

This process is being described further in this paper.

Attributes and features: there are some basic features we need to introduce first, in order to describe each configuration of the grid during the process. As we intend to use a rule based system in order to implement our algorithm, these features are widely being used to define templates as well as defining easy-to-understand facts which are the main units of data and help us to fire rules during the execution of program.

Basic attributes which all tiles in grid posses:

1) Tile position – number of positions are between 1 and n^2.

2) Tile caption – number of captions are between 1 and $n^2 - 1$.

3) Tile column – varies between 1 and n.

4) Tile row – varies between 1 and n.

5) Tile neighbors – depending on the position of tile, varies between 3 (on corners) and 8.

6) Marginality – with the value of 1 (true) for marginal tile and 0 (False) for none marginal.

7) Main diagonal – with the value of 1 to diagonal tile and 0 to none diagonal.

8) Fixed/Unfixed – with the value of 1 for fixed tile, row or column and 0 for unfixed ones.

Here "n" is the dimension of the board. Attributes 1 to 4 are shown in the Fig. 2.

	COL 1	COL 2
ROW 1	Caption = x Position = 1	Caption = y Position = 2
ROW 2	Caption = z Position = 3	Caption = w Position = 4

Рис. 2 – Attributes of a tile

Neighbors

We have two types of neighborhood forward neighbors and diagonal neighbors.

Forward neighbors

Each tile has up to 4 forward neighbors. Here we have the tile "T" which has four forward neighbors (n_1 ,n_2 ,n_3 ,n_4):

$$Column(n_1) = Column(T),\ Row(n_1) = Row(T) - 1$$
$$Column(n_2) = Column(T),\ Row(n_2) = Row(T) - 1$$
$$Column(n_3) = Column(T) - 1,\ Row(n_3) = Row(T)$$
$$Column(n_4) = Column(T) + 1,\ Row(n_4) = Row(T)$$

We can see the forward neighbors of “T” in Fig. 3.

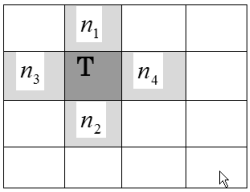

Рис. 3 – Forward Neighbors

Diagonal neighbors

Each tile has up to 4 diagonal neighbors. Here we have the tile “T” which has four diagonal neighbors (n_1, n_2, n_3, n_4):

$$Column(n_1) = Column(T) - 1,\ Row(n_1) = Row(T) - 1$$
$$Column(n_2) = Column(T) - 1,\ Row(n_2) = Row(T) + 1$$
$$Column(n_3) = Column(T) + 1,\ Row(n_3) = Row(T) + 1$$
$$Column(n_4) = Column(T) + 1,\ Row(n_4) = Row(T) - 1$$

We can see diagonal neighbors of “T” in Fig. 4.

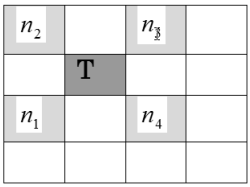

Рис. 4 – Diagonal Neighbors

Marginality

Marginal tiles are those who have placed on the outermost columns (or rows). We can see this in Fig. 5.

Important point

Maximum number of tiles neighbors is 8, and different tiles depending on their locations could have fewer neighbors. Marginal tiles always have less than 8 neighbors and those which are placed in the corners have only 3 neighbors.

Main Diagonal

Set of all tiles with the same number of column and row is called main diagonal.

Each tile on the main diagonal has its corresponding attribute (main diagonal) set to 1 (true) and the rest of tiles have 0 (falls).

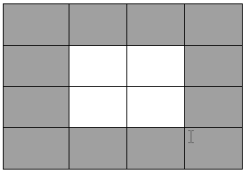

Рис. 5 – Marginal tiles

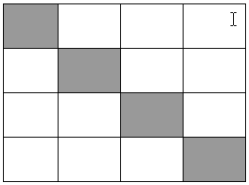

Рис. 6 – Main diagonal

Columns or rows which have all their tiles sorted as the goal state arrangement are called fixed columns (rows). We can not change the arrangement of these columns (rows) during solving the rest of the grid (this is the main rule of divide and conquer) Fig. 7.

Fixed Tile

Tiles which are placed on a sorted column (row) are called fixed tiles.

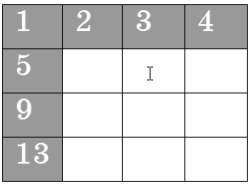

Рис. 7 – Fixed Column/Row

Important point

During the sorting process there may be some tiles which indiscriminately placed on their home position. In these cases the agent is allowed to move them from their position, as they are not located in the current column (row) which is under the sorting process.

Mirror Effect

In order to sort each column of the grid the agent needs to take similar steps as it took in order to sort the rows. In another word, the agent uses the same algorithm to sort columns as well as rows. Mirror effect is an inline function which can be used to refer to each column as a row during solving the puzzle by replacing the grids counterpart tiles according the main diagonal.

This is to consider each column as a row during sorting columns.

Each time we start to sort a new column we can exchange their counterpart row in order to simplify the process of sorting tiles. This enables us to refer each column as a row and use the same rules for columns and rows.

Divide and Conquer Algorithm

We can observe the general approach of the divide and conquer which frequently reduces the grid into a smaller one in the Fig. 8**.**

Here we represent the algorithm of D&C in pseudo code:

```
i:=1;
//Sort i-th ROW
2 Adjust Row (i)
Mirror Effect
//Sort i-th Column
Adjust Column (i)
Mirror Effect
i:=i+1;
if i=n then Finish
else goto 2
```

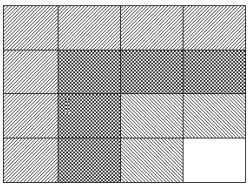

Рис. 8 – The process of D&C

Let's Start

Assume that we already have the shuffled grid and our agent is ready to solve the puzzle. First of all we need an inspector to locate space, target and home.

Space

Space is the tile which we move through the grid in order to change the configuration of it.

Target

Target is the tile which its caption and position are not the same numbers. The agent has to place target into the position that is equal to the caption of the target.

Home

This is the final position of each target.

Universal Rule

We are not allowed to move sorted tiles. There are several different configurations which can occur, after locating space, target and home and each can dramatically change the strategy of the agent.

 ISSN 1562-9945

We magnify all these different configurations and use all of them in order to give a complete routing strategy.

By identifying the locations of space, target and home we have column, row and position of each of them.

Routing strategy

Very important issue, we must consider, each time before moving the space is to answer these questions:

1. Where is the destination of space?
2. Which way to chose?

Destination of space

1. To get nearer to target (if it is not currently located in the targets neighborhood).
2. To move around the target in order to find the best position.
3. The best position is the one which allows the target to get nearer to home.

"Diagonal neighbors" rule

Space can not exchange its location with the diagonal neighbors.

"Home and space in different columns" rule

We have to first move the space into the neighborhood of the target and locate it in the position between the target and home. If the circumstance allows, it would be preferred to move the space directly to this position (by this we can reduce the amount of moves).

Choosing direction

There are some fixed actions which space takes in order to move its target in four directions.

"Up" rule

Not depending on the location of the space, "up" means to exchange the caption of space and the tile above it.

"Down" rule

To exchange the caption of space and the tile bellow it.

"Left" rule

To exchange the caption of space and the tile locating on the left side of space.

"Right" rule

To exchange the caption of space and the tile locating on the right side of space.

"Swap" rule

After locating the space, up, down or next to the target we have to push the target into its new place which is space's current position; in another word we swap the caption of space and target.

"Semi circle" rule

After exchanging the captions of space and target, and getting nearer to the home position, as we can see in the figures below the same order of actions

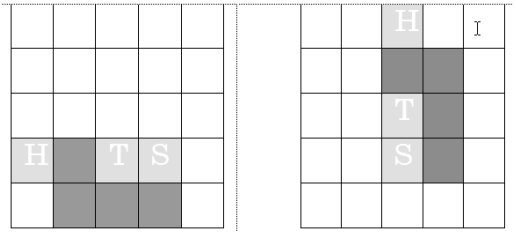

Рис. 9 – Semi circle rule

must be repeated until the target can be placed next to the home position .in the case shown in Fig. 9 we have “down”, “left”, “left”, “up”.

“Last tile” rule

There is a fixed strategy for sorting the last two tiles of each row or column:

1. put the $(n-1)$ in the position of (n);
2. put the (n) below the $(n-1)$;
3. push $(n-1)$ left in its original position;
4. push (n) up in its original position.

Special row configuration

There is one configuration in which the above algorithm is unable to work properly; we can see this configuration in the Fig. 10.

1	2	3	5	4
				S

Рис. 10 – Special configuration

To fix this configuration we can use a predefined set of moves in order to get last two tiles out of their position and rearrange them properly. We assume that space is situated under the last tile as it depicted in Fig. 10. Obviously for other positions space have to take additional steps: “left”, “up”, “right”, “down”, “down”, “left”, “up”, “right”, “up”, “left”.

Conclusion

This paper concisely represents the essential knowledge which is needed for a single agent in order to solve the (n^2-1) puzzle. As it mentioned at the beginning of the paper, solving this problem (which is somehow stands under

the combinatorial problems) we intended to use an AI approach to develop the agent and design a knowledge base for it in order to study how Artificial Intelligence techniques can help us to cope with computer science problems in a more efficient way.

Implementing this algorithm in a cognitive architecture like CLIPS or SOAR can be the subject of a separated paper.

Bibliography

1. 1) *Hayes Richard*. The Sam Loyd 15-Puzzle // http://citeseer.comp.nus.edu.sg/487673.html
2. 2) *Parberry Ian*. A real-time algorithm for the (n^2-1)-puzzle // Information Processing Letters. Vol. 56, 1995. Pp. 23-28.

Получено 06.11.2008