Artificial Intelligence Problem Solved with Relational Database

MIRELA-CATRINEL VOICU
Faculty of Economics and Business Administration
West University of Timisoara, Romania
ROMANIA
mirelavoicu@feaa.uvt.ro, http://www.feaa.uvt.ro

Abstract: - In this paper we present an implementation method for eight-puzzle game. In the artificial intelligence literature, different algorithms are proposed for implementing this game. These methods concern different heuristic functions.There are being used expert systems, as well as different programming languages or environments (e.g. C, Pascal, Java, Delphi etc.) for implementation, mentioning that users have to exploit a tree data structure. In our work we use databases for model a tree.

Key-Words: - eight-puzzle, relational database, programming.

1 Introduction

The 8-puzzle game it's a 3x3 game, using nine positions, in which we can move in the free space eight pieces. From a start state, we want to obtain a path solution to the goal state. By constructing a search tree, the computer can examine the possible configurations of the puzzle systematically, until it reaches the goal state. Then by following the path from the goal state back to the start state, the computer can determine the correct steps to solve the puzzle. A such example is presented in the *Figure 1*.

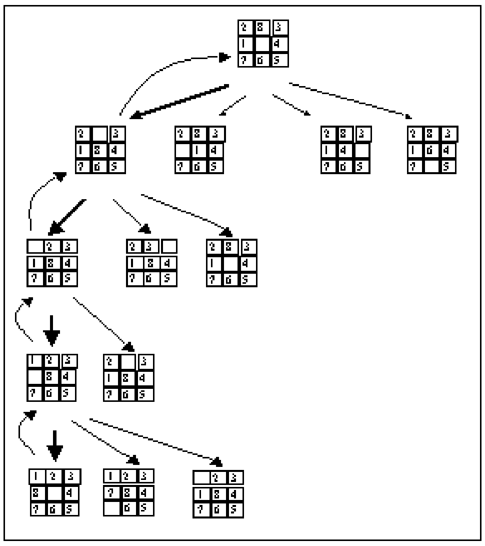


Figure 1: A search tree example for the 8-puzzle problem

The tree comprises an arrangement of *nodes,*which contain information. The nodes are linked by *arcs* (or *edges*). Each node in a tree has two, three or four nodes *descending* from it down to the bottom. The top node (the root) is the start state node.The search tree for a particular problem can grow in size quite rapidly if the goal state is not found quickly. In order to reduce the amount of searching, which the computer must do, the tree can be constructed in a depth-first manner rather than a breadth-first manner. In this way a single branch of the tree is considered first before examining other branches. The advantage of this approach is that more promising branches can be considered first. A* is perhaps the most famous heuristic off-line searching algorithm of all-time. Several real-time algorithms have been based off of A*, including the Learning Real-Time A* and Real-time A* algorithms. The basic concept of the A* algorithm is a best-first search—the most probable paths are explored first, searching outward from the starting node until it reaches the goal node. The best path is determined by choosing the option with the lowest cost, where cost is measured by the function: $f(n) = g(n) + h'(n)$. The function $g(n)$ is the actual cost of the path so far, while $h'(n)$ is a heuristic function of the estimated cost of the path from the current state to the final goal. In our implementation we can use any heuristic function, for this reason we remember not such heuristic functions.

2 Algorithm presentation

In the *Figure 2*, we present the databases used to model the search tree. Each record from this table refers a node from the tree. In each such record we save the values used in a node and additional information that concern the node.

The *level* field is used for the node level from the tree (the root level will be 0). The *code* field is used for the unique identification of the node in tree. This code is given in the order in which the nodes will be created (the root has the code 1). Each node from the tree (excepting the root) has a unique parent node and we save this code in the *parent_code* field (for the root, the *parent_code* will be 0). The *heuristic* field refers the result of the used heuristic function. The *terminal, solution* and *expanded* fields will have the value *n* or *y* depending by situation.

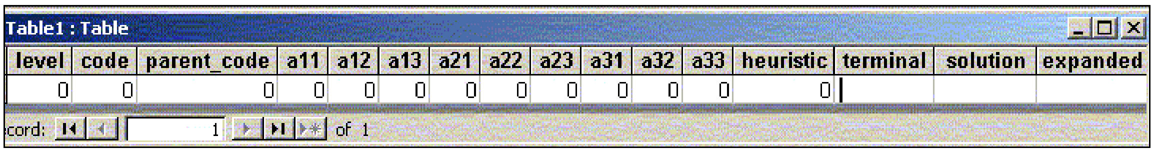


Table1 : Table

level	code	parent_code	a11	a12	a13	a21	a22	a23	a31	a32	a33	heuristic	terminal	solution	expanded
0	0	0	0	0	0	0	0	0	0	0	0	0			

cord: 1 of 1

Figure 2: The database used to model the tree

Now, we present the implementation. We can use any programming environment that allows connection with databases.

The user must introduce the *start state*. The application verifies if the *start state* is the *goal state*. In the affirmative case, the algorithm was finished.
In the negative case, the application inserts the record corresponding to the root:
(*0,1,0,value_of_a11,...,value_of_a33, result_of_heuristic _function,'n', 'n', 'n'*).

Not having a solution while there are still non-expanded nodes, as well as non-terminal ones, makes the algorithm repeats itself.

Step 1. We select a record from the table *Table1*, that corresponds to the nodes with the best heuristic, non-expanded and non-terminals, as follows:

We determine the best heuristic:

```
s:='Select min(heuristic) from Table1 where expandat="n" and terminal="n" ';
adoquery2.SQL.Clear;
adoquery2.SQL.Add(s);
adoquery2.Open;
heumin:=adoquery2.Fields[0].AsInteger;
adoquery2.SQL.Clear;
```

We select all the records which have the best heuristic:

```
s:='insert into heumin select * from table1 where heuristic='+inttostr(heumin)+' and expandat="n" and terminal="n" ';
adoquery1.SQL.Clear;adoquery1.SQL.Add(s);
adoquery1.ExecSQL;
```

We select the first such record:

```
adotable1.TableName:='heumin';
adotable1.Active:=true; adotable1.First;
level:=adotable1.Fields[0].AsInteger; code_p:= adotable1. Fields[1].AsInteger;
The values which form the node which will be expanded:
x[1,1]:=adotable1.Fields[3].AsInteger;
x[1,2]:=adotable1.Fields[4].AsInteger;
x[1,3]:=adotable1.Fields[5].AsInteger;
x[2,1]:=adotable1.Fields[6].AsInteger;
x[2,2]:=adotable1.Fields[7].AsInteger;
x[2,3]:=adotable1.Fields[8].AsInteger;
x[3,1]:=adotable1.Fields[9].AsInteger;
x[3,2]:=adotable1.Fields[10].AsInteger;
x[3,3]:=adotable1.Fields[11].AsInteger;
```

Step 2. For the selected record from the *Step 1*, we create a table with contains all records from the *Table1*, which are the ancestors of this record from *Step 1*, in the following way:

For the selected record, we will insert the data corresponding to its parent node in a table named ***ancestors***:

```
codd:=adotable1.Fields[1].AsInteger;
s:='insert into ancestors select * from eumin where cod='+inttostr(codd);
adoquery1.SQL.Clear;adoquery1.SQL.Add(s);
adoquery1.ExecSQL; adoquery1.SQL.Clear;
```

We will delete the table ***heumin*** (which corresponds to the records with the best heuristic).

```
s:='delete * from heumin'; adoquery1.SQL.Clear;
adoquery1.SQL.Add(s);
adoquery1.ExecSQL; adoquery1.SQL.Clear;
```

For which record from the table ***ancestors***, we will insert its parent in this table:

```
adotable1.Active:=false;
adotable1.TableName:='ancestors';
adotable1.Active:=true;
adotable1.First; codd:=adotable1.Fields[2].AsInteger;
while(codd>0) do
begin
s:='insert into ancestors select * from Table1 where cod='+inttostr(codd);
adoquery1.SQL.Clear;adoquery1.SQL.Add(s);
doquery1.ExecSQL;
adoquery1.SQL.Clear;adotable1.Active:=false;
adotable1.TableName:=' ancestors ';
adotable1.Active:=true;adotable1.Last;
codd:=adotable1.Fields[2].AsInteger;
end;
```

Step 3. For the node corresponding to the selected record from *Step 1*, the application will generate its children nodes, state(x, level+1, code_p,code), in the following way:

Step 3.1. Firstly, the blank position is determined. In function of situation, the application will create 2, 3 or 4 children nodes. We present all the possible situations, in *Table 1.*

```
procedure TForm1.state(a:tablou; n:integer; cp:integer; c:integer);
…
case blank_position of
 11: begin right(); down(); end;
 12: begin left(); down(); right(); end;
 13: begin left(); down(); end;
 21: begin up(); down(); right(); end;
 22: begin left(); up(); down(); right(); end;
 23: begin left(); down(); up(); end;
 31: begin up();right(); end;
 32: begin left(); up();right(); end;
 33: begin left(); up();end;
end;
end;
```

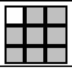
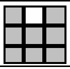
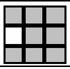
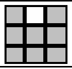
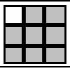
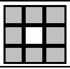
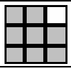
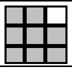
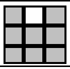
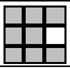
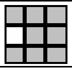
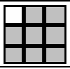
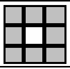
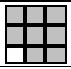
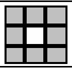
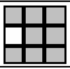
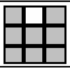
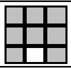
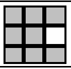
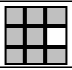
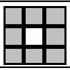
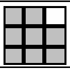
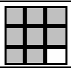
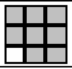
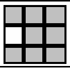
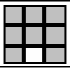
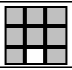
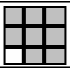
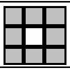
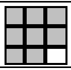
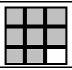
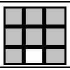
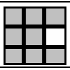
Node	Children nodes			
				
	right	down		
				
	left	down	right	
				
	left	down		
				
	up	right	down	
				
	left	up	down	right
				
	left	up	down	
				
	up	right		
				
	left	up	right	
				
	left	up		

Table 1: Children nodes

Step 3.2. When the application creates a new child node (like in the *Table 1*), it is being calculated the heuristic (corresponding to this new node) and verified if this node:

- is a solution (for the new record, which will be inserted in the table from the *Figure 2*, that means: field *terminal*='y', *solution*='y' and *expanded*='n');
- exists in tree (in the ancestors tables from the *Step 2*) and is not solution (this means: field *terminal*='y', *solution*='n' and *expanded*='n');
- exists not and is not solution (this means: field *terminal*='n', *solution*='n' and *expanded*='n');

Now, the application inserts the corresponding record of the new node into the table from *Figure 2*.

```
procedure TForm1.left(…);
…
sol:='n'; term:='n';exp:='n';
if heuristic=0 then
begin sol:='d'; term:='d'; end;
{b is the array corresponding to the new node}
s:='Select count(*) from ancestors where a11='+inttostr(b[1,1])+' and a12='+inttostr(b[1,2])+' and a13='+inttostr(b[1,3])+' and a21='+inttostr(b[2,1])+' and a22='+inttostr(b[2,2])+' and a23='+inttostr(b[2,3])+' and a31='+inttostr(b[3,1])+' and a32='+inttostr(b[3,2])+' and a33='+inttostr(b[3,3]);
adoquery2.SQL.Clear;
adoquery2.SQL.Add(s);adoquery2.Open;g:=adoquery2.Fields[0].AsInteger; adoquery2.SQL.Clear;
if g>0 then term:='d';
s:='Insert Into Table1(level,code, parent_code,a11,a12,a13,a21,a22,a23,a31,a32,a33,heuristic, terminal, solution, expanded) values( '+inttostr(n)+', '+inttostr(f)+', '+inttostr(cp);
For i:=1 to 3 do
For j:=1 to 3 do
 s:=s +', '+inttostr(b[i,j]);
 s:=s+', '+inttostr(eu1)+', "'+term+'", "'+sol+'", "'+exp+'" )';
adoquery1.SQL.Clear;adoquery1.SQL.Add(s);
adoquery1.ExecSQL;
end;
```

Step 4. The ancestors table from the *Step 2*, will be deleted.

When the application stops to repeat the *Steps 1-4*, one of the following situations might occurs:

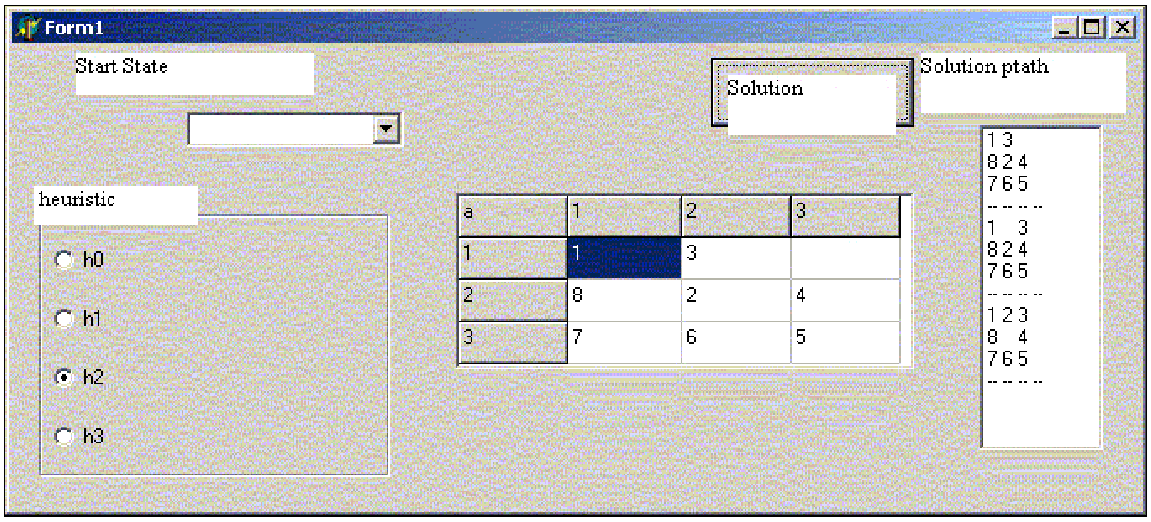
- we have founded a solution (we have the goal state);
- we have not founded a solution, but in the tree all the nodes are expanded or terminals.

In the first case, for viewing the solution path, for the founded goal state, we will the ancestors table.

Using this new table, starting from the start state the goal state, we will obtain the solution path.

```
s:='select count(*) from table1 where solution="y" ';
```

```
adoquery2.SQL.Clear;
adoquery2.SQL.Add(s); adoquery2.Open;
level:=adoquery2.Fields[0].AsInteger;
 if level>0 then
begin
{we insert the goal state in the table solution_path}
 s:='insert into solution_path select * from table1 where
solution="y"' ;
adoquery2.SQL.Clear;adoquery2.SQL.Add(s);
adoquery2.ExecSQL;
 adotable1.Active:=false;
adotable1.TableName:='solution_path';adotable1.Active
:=true;adotable1.First;
codd:=adotable1.Fields[2].AsInteger;
```



heuristic: h0 h1 [✓ h2] h3

a	1	2	3
1	1	3	
2	8	2	4
3	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
-- -- -- --
```

Figure 3: A solution path for a start state

```
{For each record from the table solution_path, we will
also insert, in this table, its parent note}
 while(codd>0) do
 begin
s:='insert into solution_path select * from Table1 where
code='+inttostr(codd);
adoquery1.SQL.Clear;
adoquery1.SQL.Add(s);
adoquery1.ExecSQL;
adoquery1.SQL.Clear;
adotable1.Active:=false;
adotable1.TableName:='solution_path';adotable1.Active
:=true;adotable1.Last;
codd:=adotable1.Fields[2].AsInteger;
 end;

{The states will be ordered from the start state to goal
state}
 s:='Select * from solution_path order by cod';
 adoquery1.SQL.Clear;
adoquery1.SQL.Add(s);            adoquery1.Open;
adoquery1.First;

{The solution path will be displayed for viewing (in this
case-like example, in a ListBox component)}
 while not(adoquery1.Eof) do
 begin
listbox1.Items.Add(adoquery1.Fields[3].AsString+'    '+
adoquery1.Fields[4].AsString+'+adoquery1.Fields[5].A
sString);
listbox1.Items.Add(adoquery1.Fields[6].AsString+'    '+
adoquery1.Fields[7].AsString+'
'+adoquery1.Fields[8].AsString);
listbox1.Items.Add(adoquery1.Fields[9].AsString+'    '+
adoquery1.Fields[10].AsString+'
'+adoquery1.Fields[11].AsString);
listbox1.Items.Add('-- -- -- --');
adoquery1.Next;
end;
 adoquery1.SQL.Clear;
```

In the *Figure 3*, we present a solution path (in *ListBox*) for a start state. We also remember that for certain start states, solutions path may not exist. Also, we can generally find one or more solution paths for each start state.

3 Conclusion

In this paper we have presented a certain case (an application - eight-puzzle) studied in the artificial intelligence domains in which, using database, we can model a tree structure. The using of databases to modes trees, can be applied in more others practical situations. This method conduces to a quick implementation, due to the possibility of using SQL statements when exploiting the tree – and this means: a short program, easily implementation and short time to obtain results. This paper wants to come with the idea that different problems from artificial intelligence domain (which is a very interesting and difficult domain of informatics) can be solved using *SQL* statements. In this way, we want to be focused on the direction that for solve many problems from artificial intelligence it will can be use relational databases. After an implementation using databases, the user will have a nice surprise to observe a facile implementation.

The first goal of *SQL* statement it is, of course, the relational database exploring. Our purpose is to highlight the using of *SQL* statement that can be more and more extended in the informatics problems and domains, with easily implementations and a very good program.

In order to illustrate our observation, we have presented such implementation in *Delphi*, using database from *Access*, but we can use any programming environment which accepts connections with different relational databases types.

References:

[1] http://www.cs.utexas.edu/users/novak/asg-8p.html

[2] http://kantz.com/jason/writing/8-puzzle.htm

[3]http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/5_2.html

[4]http://www.cs.duke.edu/~mlittman/courses/cps271/lect-05/node25.html

[5] http://www.aaai.org/AITopics/html/seachreason.html

[6]http://www.cc.gatech.edu/classes/cs3361_96_spring/lecture-2.html

[7]http://www.informatics.sussex.ac.uk/courses/kr/lec04.html

[8]http://thor.info.uaic.ro/~dcristea/cursuri/IAOnWeb/IA4-SistProd-Control.htm

[9]http://www.zib.de/reinefeld/bib/93ijcai.pdf