Databases used for trees modeling

Associate Professor, Mirela-Catrinel VOICU Ph.D
Faculty of Economic Sciences, West University of Timişoara, România

In this paper we want to present an implementation method for eight-puzzle game. In the artificial intelligence literature, different algorithms are proposed for implement this game. These methods concern different heuristic functions. For implementation, generally, expert systems or different programming languages or environments (like C, Pascal, Java, Delphi etc.) are used, in which the user must exploit a tree data structure. In our work we use databases for to model a tree.

1 Introduction

The 8-puzzle game it's a game 3x3, 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. Such an example is presented in the *Figure 1*.

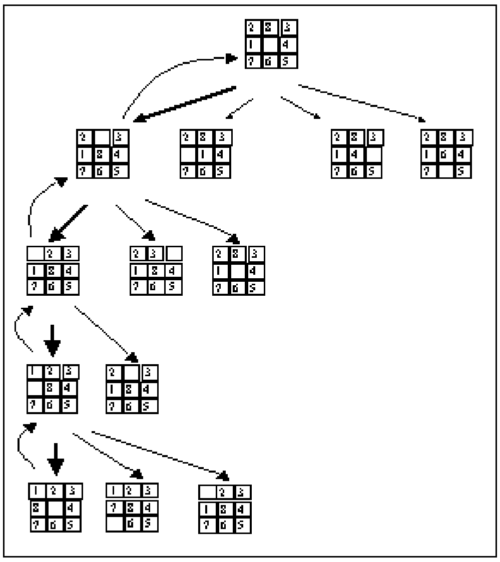


Fig.1. An search tree example for the 8-puzzle problem

The tree comprises an arrangement of *nodes* each of which holds 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 most nodes which have no nodes emanating from them. The top most 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. To reduce the amount of searching 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 state 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 do not recall such heuristics.

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 field *level* is used for the node level from the tree (the root level will be 0). The field *code* 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 field *parent_code* (for the root, the *parent_code* will be 0). The field *heuristic* refers the result of the used heuristic function. The fields *terminal, solution* and *expanded* will have the value *n* or *y* in function of the 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

Fig.2. The database used to model the tree

Now, we present the implementation. We can work with any programming environment, which permits 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 is 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').

Now, while we do not have obtained a solution and there exist not expanded and not terminal nodes, we will repeat the following algorithm:

Step 1. We select a record from the table *Table1*, which corresponds to a node with the best heuristic, non-expanded and non-terminals in the following way:

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 expan-
dat="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, are the followings:
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 forms the ancestors of this record, in the following way:

For the selected record, we will insert the record corresponding to the 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 correspond to the records with the best heuristic).

```
 s:='delete * from heumin'; adoquery1.SQL.Clear; adoquery1.SQL.Add(s); adoquery1.ExecSQL; ado-
query1.SQL.Clear;
```

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

```
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);  adoquery1.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 the *Step 1*, the application will generates its children nodes,

state(x, level+1, code_p,code);

in the following way:

```
procedure TForm1.state(a:tablou; l: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;
```

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 possible situations in the *Table 1*.

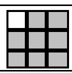
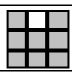
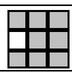
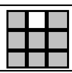
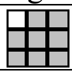
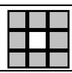
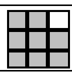
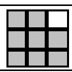
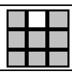
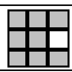
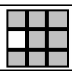
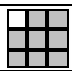
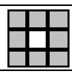
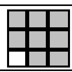
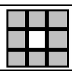
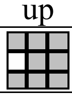
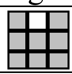
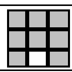
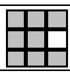
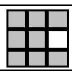
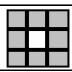
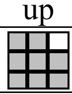
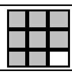
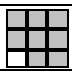
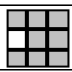
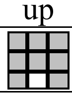
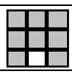
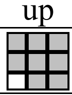
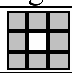
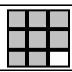
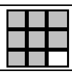
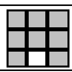
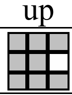
Node	Children nodes			
				
	right	down		
				
	left	down	right	
				
	left	dwn		
				
	up	right	down	
				
	left	up	down	right
				
	left	up	down	
				
	up	right		
				
	left	up	right	
				
	left	up		

Table 1: children nodes

Step 3.2. When the application creates a new children node (like in the *Table 1*), it is calculate the heuristic (corresponding to this new node) and it is verified if this node:
- is solution (for the new record, which will be inserted in the table from the *Figure 2*, this 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');

The application inserts the record corresponding to the new node in the table from the *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(heuristic)+', "'+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.

In the moment in which the application stops to repeat the *Steps 1-4*, we will have one from the following situations:

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

In the first case, for viewing the solution path, for the founded goal state, we will generate the ancestors table. Using this new table, following the path from the start state to the goal state we will obtain a 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; ado-
table1.TableName:='solution_path';adotable1.Active:=true;adotable1.First;
codd:=adotable1.Fields[2].AsInteger;

{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; ado-
table1.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 com-
ponent)}
 while not(adoquery1.Eof) do
 begin
 listbox1.Items.Add(adoquery1.Fields[3].AsString+' '+ adoquery1.Fields[4].AsString+'
'+adoquery1.Fields[5].AsString);
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 recall also that for certain start states there exists zero, one or more solutions paths.

In the moment in which we have displayed the solution path in *ListBox* or we find that there exists not solution (for a certain start state), all records from the databases will be deleted. All construction from the databases are used only for obtain the solution path.

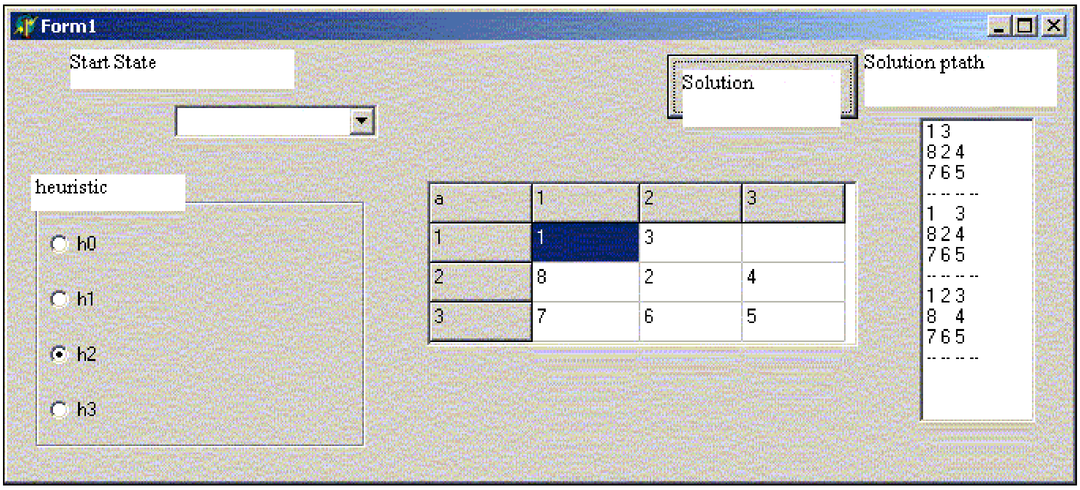


Figure 3: A solution path for a start state

Conclusion

In this paper we have presented a certain case (with applications, and generally studied in the artificial intelligence domains) in which, using database, we can model a tree structure. The using of databases to models tree can be applied in more others practical situations. This method conduces to a quickly implementation, because, in order to exploit the tree, we can use *SQL* statements – and this means: a short program, easy implementation and short time to obtain the results.

We have presented a such implementation in *Delphi*, using database from *Access*, but we can use any programming environment which accept the connection with different database 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. M.C. Voicu – *Algoritmi şi aplicaţii în Delphi vol. I*, Ed. Mirton, Timişoara, 2003
10. M.C. Voicu – *Viewpoints on the multidimensional database theory* – The Central and East European Conference in Business Information Systems, Cluj-Napoca, 20-22 mai 2004, volumul conferinţei, pag. 143-161
11. M.C. Voicu - *On the multidimensional database* - the Seventh International Conference on Informatics in Economy *IE 2005* – Bucureşti, May 19-21, 2005
12. Voicu M., Mircea G. - *Algorithm for obtaining aggregated value sets from multidimensional databases* – Conference Proceedings of 5th WSEAS International Conference on Applied Informatics ad Communications (AIC 05), Malta, September 15-17, 2005
13. Voicu M., Mircea G.- *Algorithms for exploiting multidimensional databases* – WSEAS Transactions on Information Science and Applications, Issue 12, Volume 2, December 2005, ISSN 1790-0832, pag. 2176-2183