

The N-Puzzle Problem

Zdravko Markov¹, Ingrid Russell, Todd Neller
June 20, 2005

1. Introduction

The N-puzzle game provides a good framework for illustrating conceptual AI search in an interesting and motivating way. Various uninformed and informed search algorithms are usually applied in this setting and their performance is evaluated. In the 8-puzzle version of the game, a 3x3 board consists of 8 tiles numbered 1 through 8 and an empty tile (marked as 0). One may move any tile into an orthogonally adjacent empty square, but may not move outside the board or diagonally. The problem is to find a sequence of moves that transforms an initial board configuration into a specified goal configuration. The following are examples of an initial and goal configurations:

Initial configuration Goal configuration

1 6 2	1 2 3
5 7 3	4 5 6
0 4 8	7 8 0

The domain theory of the N-puzzle problem can be expressed by a set of facts describing state transitions, and a search engine that can be used to find paths between initial and goal states. Given a pair of an initial and a goal state (a training example), the search algorithm finds the shortest path between them (explanation or proof). Then applying the Explanation-Based Learning (EBL) techniques, the path is generalized so that it can be used later to match other initial states and bring the search algorithm directly to the goal state, without the resource-consuming exploration of the huge state space of the game. With carefully chosen training examples, useful rules for typical moves can be learnt and then integrated into the search algorithm to achieve better performance.

¹ Corresponding author: MarkovZ@mail.ccsu.edu, Department of Computer Science, Central Connecticut State University, 1615 Stanley Street, New Britain, CT 06050.

2. Project Overview

In this project students are asked to incorporate Explanation-Based Learning (EBL) into the N-puzzle problem. This allows them to better understand the concepts of analytical learning, and to see how learning improves the performance of search algorithms. The goal is to introduce the student to Analytical (Explanation-Based) Learning using the classical AI framework of search. Hands-on experiments with search algorithms combined with an EBL component give the student a deep, experiential understanding of the basics of EBL and the role it plays in improving the performance of search algorithms in particular and problem solving approaches in general. The algorithms may be implemented in Lisp or Prolog. The examples in the project descriptions are done in Lisp. Examples of Prolog implementations are provided in Appendix A.

3. Basics of EBL

Many learning algorithms (usually considered as inductive learning) generalize on the basis of regularities in training data. These algorithms are often referred to as similarity based, i.e. generalization is primarily determined by the syntactical structure of the training examples and the use of domain knowledge is limited to specifying the hypothesis language and exploring the hierarchy of the attribute values. Typically a learning system uses domain knowledge and is expected to have some ability to solve problems. Then the objective of learning is to improve the system's knowledge or system's performance using that knowledge. This task could be seen as knowledge reformulation or theory revision.

EBL uses a domain theory to construct an explanation of the training example, usually a proof that the example logically follows from the theory. Using this proof the system filters the noise, selects only the aspects of the domain theory relevant to the proof, and organizes the training data into a systematic structure. This makes the system more efficient in later attempts to deal with the same or similar examples. The basic components in EBL are the following:

- *Target concept.* The task of the learning system is to find an effective definition of this concept. Depending on the specific application the target concept could be a classification rule, theorem to be proven, a plan for achieving goal, or heuristic to make a problem solver more efficient (e.g. a state space search heuristic).
- *Training example.* This is an instance of the target concept. For example, this may be a good (efficient) solution in a state space search.
- *Domain theory.* Usually this is a set of rules and facts representing domain knowledge. They are used to explain how the training example is an instance of the target concept.
- *Operationality criteria.* Some means to specify the form of the concept definition. In other words this is the language of expressing the target concept definition, which is usually a part of the language used in the domain theory.

In the form outlined above, EBL can be seen as partial evaluation. In terms of theorem-proving, this technique is also called unfolding, i.e. replacing body goals with the bodies of the rules they match, following the order in which goals are reduced (depth-first). Hence in its pure form EBL doesn't learn anything new, i.e. all the rules inferred belong to the deductive closure of the domain theory. This means that these rules can be inferred from the theory without using the training example at all. The role of the training example is only to focus the theorem prover on relevant aspects of the problem domain. Therefore EBL is often viewed as a form of speed-up learning or knowledge reformulation. Consequently EBL can be viewed not as a form of generalization, but rather as specialization, because the rule produced is more specific than a theory itself (the EBL rule is applicable to only one example). All this however does not undermine EBL as a Machine Learning approach.

There are small and well defined theories, however practically inapplicable. For example, consider the game of chess. The rules of chess combined with an ability to perform unlimited look-ahead on the board states will allow a system to play well. Unfortunately this approach is impractical. An EBL system, given well chosen training examples, will not add anything new to the rules of chess playing, but will actually learn some heuristics to apply these rules, which might be practically useful. The N-puzzle

domain is another typical example of this approach. As the search space is huge, any practical solution requires heuristics. And the role of EBL is to learn such heuristics from examples of successful searches.

4. Game representation

To simplify the discussion hereafter we discuss the 5-puzzle problem. In this representation tiles are numbered 1, 2, 3, 4, 5. The empty square (no tile) is represented by 0. The state of the game is represented by a list of tiles (including 0), where their position in the list corresponds to their board position. For example (1,2,3,4,5,0) correspond to the following board:

```
-----  
| 1 | 2 | 3 |  
-----  
| 4 | 5 | 0 |  
-----
```

5. State transitions

The state transitions are represented by reordering tiles in the list. For this purpose we use variables, so that the number of transitions that have to be described is minimized.

Positions are mapped to variables, which hold the actual tile numbers as follows:

```
-----  
| first | second | third |  
-----  
| fourth| fifth  | sixth |  
-----
```

For example, moving the empty tile from position 1 to position 2 is represented as transforming state

`(first, second, third, fourth, fifth, sixth)`

into state

`(second, first, third, fourth, fifth, sixth),`

where all list elements except for first (which holds the 0) can take actual tile numbers from 1 to 5 (all different). Thus, this generalized transition represents 5! actual transitions

between game states.

Depending on the position of the empty tile (0), we may have two or three possible transitions of the type discussed above. In LISP, this is implemented as a function, which adds the new states in the beginning of the list of current states. For example, the function below extends the current list of states new-list with two new states, which are the two possible transitions from state.

```
(defun move-1 (state)
  (setf new-states (cons(append
    (list (second state))(list (first state))
    (nthcdr 2 state)) new-states))
  (setf new-states (cons(append
    (list (fourth state))(list (second state))
    (list (third state)) (list (first state))
    (nthcdr 4 state)) new-states)))
```

Here is an example of extending state (0 1 2 3 4 5) by using the move-1 function:

```
> (setf new-states (list (list 0 1 2 3 4 5)))
((0 1 2 3 4 5))
> (move-1 (list 0 1 2 3 4 5))
((3 1 2 0 4 5) (1 0 2 3 4 5) (0 1 2 3 4 5))
```

5.1 Deliverable 1: State Space Representation

1. Write a complete function to implement all possible transitions from all possible current states (positions of the empty tile).
2. Change the representation accordingly and implement the state transition function for the 8-puzzle problem.
3. Implement 5 and/or 8-puzzle state transitions in Prolog (see Appendix A for details).

6. Search algorithm

Any uninformed search algorithm that is able to find the shortest path in a graph can be used here. As the goal of EBL is to improve the efficiency, the algorithm can be a simple one. Iterative deepening and breadth-first search are good choices, because they have

high computational complexity. Thus after the EBL step the speed-up would be easily measured by the reduction of the size of the path between initial and goal states and run time and memory usage. Here is an example of solving the 5-puzzle with breadth-first search (the print shows the path between start and finish found by the algorithm):

```
> (setf start '(4 5 3 0 1 2))
> (setf finish '(1 2 3 4 5 0))
> (breadth-first start finish)
((4 5 3 0 1 2) (0 5 3 4 1 2) (5 0 3 4 1 2)
 (5 1 3 4 0 2) (5 1 3 4 2 0) (5 1 0 4 2 3)
 (5 0 1 4 2 3) (0 5 1 4 2 3) (4 5 1 0 2 3)
 (4 5 1 2 0 3) (4 0 1 2 5 3) (4 1 0 2 5 3)
 (4 1 3 2 5 0) (4 1 3 2 0 5) (4 1 3 0 2 5)
 (0 1 3 4 2 5) (1 0 3 4 2 5) (1 2 3 4 0 5)
 (1 2 3 4 5 0))
```

6.1: Deliverable 2: Uninformed and Informed Search

1. Find/Implement uninformed and informed search algorithms (e.g. breadth-first, breadth-first, iterative deepening, best-first, a-star, beam search) in Lisp or Prolog using a standard graph representation. See Russell and Norvig's book and the book web site to download implementations of the algorithms.
2. Update the algorithms so that they use the state transition functions implementing the moves for the 5 or 8-puzzle (the student projects for Section 5).
3. Experiment with solving the 5-puzzle transition (4 5 3 0 1 2) => (1 2 3 4 5 0) with uninformed search.
4. Solve the 8-puzzle transition: (2 3 5 0 1 4 6 7 8) => (0 1 2 3 4 5 6 7 8) with uninformed search:
 - a. Compare breadth-first, iterative deepening and depth-first.
 - b. Explain why depth-first fails.
 - c. Figure out an approach to find board states that can be solved.
5. Implement a heuristic function for the informed search algorithms (see Russell and Norvig) and solve the 8-puzzle transition (2 3 5 0 1 4 6 7 8) => (0 1 2 3 4 5 6 7 8)
6. Use best-first, a-star and beam search. For beam search try n=100,10,1. What changes? Explain why it fails with n=1?
7. Compare results with depth-first, breadth-first and iterative deepening.
8. Collect data about the time and space complexity of solving the above problems with uninformed and informed search algorithms. Analyze the results.

7. Training example and target concept

First we specify a training example, as a pair of start and finish states. Let us consider the transition from state (4 5 3 0 1 2) to (5 1 3 4 2 0). According to the EBL principles, the training example is an instance of the target concept. So, we have to run the algorithm in order to verify that this is a correct training example, i.e. it is an instance of a correct target concept:

```
> (setf start '(4 5 3 0 1 2))
> (setf finish '(5 1 3 4 2 0))
> (breadth-first start finish)
((4 5 3 0 1 2) (0 5 3 4 1 2) (5 0 3 4 1 2)
 (5 1 3 4 0 2) (5 1 3 4 2 0))
```

8. EBL generalization

In our setting, EBL generalization is simply substituting constants for variables.

Following the representation adopted here, we get a new generalized transition from state

```
(first, second, third, fourth, fifth, sixth)
```

to state

```
(second, fifth, third, first, sixth, fourth),
```

where the following substitutions apply:

```
first = 4 second = 5 third = 3
fourth = 0 fifth = 1 sixth = 2
```

9. Improving the search (in EBL terms: improving the domain theory)

The last step in EBL is to add to new target concept definition to the domain theory. In the particular example, this means defining a new function that will allow the search algorithm to use the new state transition. In our LISP implementation we have to modify the state transition function in order to add the new state to the resulting list.

```
(defun move-4 (state)
  (setf new-states (cons (append
    (list (second state))(list (fifth state))
    (list (third state))(list (first state))
```

```

      (list (sixth state))(list (fourth state)))
      new-states))
(setf new-states (cons (append
  (list (fourth state))(list (second state))
  (list (third state))(list (first state))
  (nthcdr 4 state)) new-states))
(setf new-states (cons (append
  (butlast state 3)(list (fifth state))
  (list (fourth state))(last state))
  new-states)))

```

It is important to note that the new state transition generated by EBL should be used first by the search algorithm. We achieve this by adding the new state as a first element of the path extension (the new state is added by the first setf).

To preserve the completeness of the algorithm (in EBL terms: completeness of the theory), the new transitions should not replace the original basic ones (one-tile move). Rather, it should be just added, thus expanding the search space with new transitions. This is implemented in our move-4 function too, as it returns also the state transitions based on one-tile moves.

The newly learned EBL state transition may represent useful search heuristics. To achieve this, however, the training examples have to be carefully chosen. They should represent expert strategies to solve the game or at least pieces of such strategies. In fact, our training example was chosen with this idea in mind. Thus, the newly learnt concept (incorporated in move-4) improves the efficiency of the algorithm. This can be shown with the same pair of start and finish states that produced a path of 19 states with the standard breadth-first search. Now the path has only 13 states, which means that the new transition is used twice during the search.

```

> (setf start '(4 5 3 0 1 2))
> (setf finish '(1 2 3 4 5 0))
> (breadth-first start finish)
((4 5 3 0 1 2) (5 1 3 4 2 0) (5 1 0 4 2 3)
 (5 0 1 4 2 3) (0 5 1 4 2 3) (4 5 1 0 2 3)
 (4 5 1 2 0 3) (4 0 1 2 5 3) (4 1 0 2 5 3)

```

(4 1 3 2 5 0) (4 1 3 2 0 5) (4 1 3 0 2 5)
(1 2 3 4 5 0))

9.1 Deliverable 3: EBL phase

- Identify useful search heuristics and generate and verify the corresponding EBL training examples.
- Perform experiments with training examples and update the state transition function manually. Then measure the improvement in terms of time and space complexity after the EBL step.
- Implementing automatic update of the theory given a training example. This includes the EBL generalization step and incorporating new transition rules into the search algorithm.
- Evaluating the effect of learning if too many or bad examples are supplied.

Reading

The project requires good understanding of uninformed and informed search algorithms and their implementations either in Lisp or in Prolog. For the principles of search and implementations of search algorithms we recommend reading chapters 3 and 4 of

Stuart Russell and Peter Norvig. [*Artificial Intelligence: a modern approach, 2-nd ed.*](#) Prentice Hall, Upper Saddle River, NJ, USA, 2003.

Lisp implementations of search algorithms are available from the AIMA site at

<http://aima.cs.berkeley.edu/lisp/doc/overview.html>

More about Lisp and search algorithms in Lisp can be found in the classical text:

Patrick Winston and Berthold Horn, *Lisp* (3rd edition), Addison-Wesley, 1993.

An excellent book on Prolog and Prolog implementations of AI algorithms (including search) is:

Ivan Bratko, *PROLOG Programming for Artificial Intelligence* (3rd edition), Addison-Wesley, 2000.

An overview of the EBL is provided in Section 19.3. of of the Russell and Norvig's text.

The basics of EBL are covered in Chapter 11 of the Mitchell's classical ML text

Tom Mitchell, Machine Learning, McGraw Hill, 1997.

Appendix A: The Prolog approach

As in the Lisp approach the tiles are numbered 1, 2, 3, 4, 5, 6, 7, 8 and the empty square (no tile) is represented by 0. The state of the game is represented by the Prolog term `board(A,B,C,D,E,F,G,H,I)`, where each variable takes one of the values of 1, 2, 3, 4, 5, 6, 7, 8 and 0.

```
-----  
| A | B | C |  
-----  
| D | E | F |  
-----  
| G | H | I |  
-----
```

For example, the goal state

```
-----  
| 1 | 2 | 3 |  
-----  
| 8 | 0 | 4 |  
-----  
| 7 | 6 | 5 |  
-----
```

is represented as `board(1,2,3,8,0,4,7,6,5)`.

The transitions in the state space are represented as the following Prolog facts, where the third argument represents the cost of the transition.

```
% empty tile placed on position 1  
arc(board(0,B,C,D,E,F,G,H,I),board(B,0,C,D,E,F,G,H,I),2).  
arc(board(0,B,C,D,E,F,G,H,I),board(D,B,C,0,E,F,G,H,I),2).  
  
% empty tile placed on position 2  
arc(board(A,0,C,D,E,F,G,H,I),board(0,A,C,D,E,F,G,H,I),2).  
arc(board(A,0,C,D,E,F,G,H,I),board(A,C,0,D,E,F,G,H,I),2).  
arc(board(A,0,C,D,E,F,G,H,I),board(A,E,C,D,0,F,G,H,I),2).  
  
% empty tile placed on position 3  
arc(board(A,B,0,D,E,F,G,H,I),board(A,0,B,D,E,F,G,H,I),2).  
arc(board(A,B,0,D,E,F,G,H,I),board(A,B,F,D,E,0,G,H,I),2).  
  
% empty tile placed on position 4  
arc(board(A,B,C,0,E,F,G,H,I),board(0,B,C,A,E,F,G,H,I),2).  
arc(board(A,B,C,0,E,F,G,H,I),board(A,B,C,E,0,F,G,H,I),2).  
arc(board(A,B,C,0,E,F,G,H,I),board(A,B,C,G,E,F,0,H,I),2).  
  
% empty tile placed on position 5  
arc(board(A,B,C,D,0,F,G,H,I),board(A,0,C,D,B,F,G,H,I),2).  
arc(board(A,B,C,D,0,F,G,H,I),board(A,B,C,0,D,F,G,H,I),2).  
arc(board(A,B,C,D,0,F,G,H,I),board(A,B,C,D,F,0,G,H,I),2).  
arc(board(A,B,C,D,0,F,G,H,I),board(A,B,C,D,H,F,G,0,I),2).
```

```

% empty tile placed on position 6
arc(board(A,B,C,D,E,0,G,H,I),board(A,B,0,D,E,C,G,H,I),2).
arc(board(A,B,C,D,E,0,G,H,I),board(A,B,C,D,0,E,G,H,I),2).
arc(board(A,B,C,D,E,0,G,H,I),board(A,B,C,D,E,I,G,H,0),2).

% empty tile placed on position 7
arc(board(A,B,C,D,E,F,0,H,I),board(A,B,C,0,E,F,D,H,I),2).
arc(board(A,B,C,D,E,F,0,H,I),board(A,B,C,D,E,F,H,0,I),2).

% empty tile placed on position 8
arc(board(A,B,C,D,E,F,G,0,I),board(A,B,C,D,0,F,G,E,I),2).
arc(board(A,B,C,D,E,F,G,0,I),board(A,B,C,D,E,F,0,G,I),2).
arc(board(A,B,C,D,E,F,G,0,I),board(A,B,C,D,E,F,G,I,0),2).

% empty tile placed on position 9
arc(board(A,B,C,D,E,F,G,H,0),board(A,B,C,D,E,0,G,H,F),2).
arc(board(A,B,C,D,E,F,G,H,0),board(A,B,C,D,E,F,G,0,H),2).

```

A simple heuristics function is implemented as follows:

```

h(board(A,B,C,D,E,F,G,H,I),W) :-
    distance([A,B,C,D,E,F,G,H,I],[1,2,3,8,0,4,7,6,5],W).

distance([],[],0).
distance([X|T],[X|V],W) :- !,
    distance(T,V,W).
distance([_|T],[_|V],W) :-
    distance(T,V,W1),
    W is W1+1.

```

Below we provide Prolog implementations of three basic search algorithms (“+” means input and “-“ is the output; see the example query at the end).

```

%-----%
% Depth-first search by using a stack %
% call: depth_first(+[[Start]],+Goal,-Path,-ExploredNodes). %
%-----%
depth_first([[Goal|Path]|_],Goal,[Goal|Path],0).
depth_first([Path|Queue],Goal,FinalPath,N) :-
    extend(Path,NewPaths),
    append(NewPaths,Queue,NewQueue),
    depth_first(NewQueue,Goal,FinalPath,M),
    N is M+1.

%-----%
% Breadth-first search %
% call: breadth_first(+[[Start]],+Goal,-Path,-ExploredNodes). %
%-----%
breadth_first([[Goal|Path]|_],Goal,[Goal|Path],0).
breadth_first([Path|Queue],Goal,FinalPath,N) :-
    extend(Path,NewPaths),
    append(Queue,NewPaths,NewQueue),
    length(NewQueue,ZZZ),writeln(ZZZ),
    breadth_first(NewQueue,Goal,FinalPath,M),

```

```

N is M+1.

extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode,_),
             \+ member(NewNode,Path)), % for avoiding loops
            NewPaths).

%-----%
% Best-First Search %
% call: best_first(+[[Start]],+Goal,-Path,-ExploredNodes). %
%-----%
best_first([Goal|Path|_],Goal,[Goal|Path],0).
best_first([Path|Queue],Goal,FinalPath,N) :-
    extend(Path,NewPaths),
    append(Queue,NewPaths,Queue1),
    sort_queue1(Queue1,NewQueue), wrq(NewQueue),
    best_first(NewQueue,Goal,FinalPath,M),
    N is M+1.

extend([Node|Path],NewPaths) :-
    findall([NewNode,Node|Path],
            (arc(Node,NewNode,_),
             \+ member(NewNode,Path)), % for avoiding loops
            NewPaths).

sort_queue1(L,L2) :-
    swap1(L,L1), !,
    sort_queue1(L1,L2).
sort_queue1(L,L).

swap1([A1|B1],[A2|B2]|T],[A2|B2],[A1|B1]|T) :-
    hh(A1,W1),
    hh(A2,W2),
    W1>W2.
swap1([X|T],[X|V]) :-
    swap1(T,V).

```

More implementations are available from:

http://www.cs.ccsu.edu/~markov/ccsu_courses/aiprograms/search1.pl
http://www.cs.ccsu.edu/~markov/ccsu_courses/aiprograms/search2.pl

Here is an example of running the breadth-first algorithm. It also shows the number of transitions (the length of the path to the goal state) and the number of explored nodes (time complexity). Note that the path is shown in reverse order.

```
?- breadth_first([[board(2,3,5,0,1,4,6,7,8)]], board(0,1,2,3,4,5,6,7,8), Path, N),
length(Path, Len).
```

```
Path = [board(0, 1, 2, 3, 4, 5, 6, 7, 8), board(1, 0, 2, 3, 4, 5, 6, 7, 8), board(1, 2, 0, 3, 4, 5,
6, 7, 8), board(1, 2, 5, 3, 4, 0, 6, 7, 8), board(1, 2, 5, 3, 0, 4, 6, 7, 8), board(1, 2, 5, 0, 3, 4,
6, 7, 8), board(0, 2, 5, 1, 3, 4, 6, 7, 8), board(2, 0, 5, 1, 3, 4, 6, 7, 8), board(..., ..., ..., ..., ...,
..., ..., ..., ...)]...
```

N = 393
Len = 10