Learning goal-oriented strategies in problem solving

Martin Možina, Timotej Lazar, Ivan Bratko

Faculty of Computer and Information Science
University of Ljubljana, Ljubljana, Slovenia

Abstract

The need for conceptualizing problem solving knowledge has been presented in several areas, such as in behavior cloning, acquiring teaching knowledge for intelligent tutoring systems or simply to be used as heuristics in game playing. In this paper, we present an algorithm for learning hierarchical strategies, where a strategy is composed of a sequence of subgoals. Each subgoal is a prerequisite for the next goal in the sequence, such that achieving one goal enables us to achieve the following goal. As the sequence of subgoals concludes with the main goal, such strategy facilitates problem solving in a step-by-step manner. The algorithm learns from a state-space representation of the domain. Each state needs to be described with a set of attributes that are used to define subgoals, hence a subgoal can be seen as a subset of states having the same attribute-values. We demonstrate the algorithm on three domains. In the simplest one, learning strategies for mathematical equation solving, we learn strategies from a complete state-space representation, where each state corresponds to a learning example. In other two examples, the 8-puzzle game and prolog programming, the complete state-space is too extensive to be used in learning, and we introduce an extension of the algorithm that can learn from particular solution traces, can exploit implicit conditions and uses active learning to select states that are expected to have the most influence on the learned strategies.

Keywords:
problem solving, procedural knowledge, strategy learning

---

1. Introduction

Problem solving in artificial intelligence is often defined as a systematic search through possible actions in order to reach a predefined goal. With this technique, computers are capable of solving relatively difficult problems, yet their solutions are often hard to understand. In computer chess playing, for example, the moves drawn by a computer are often incomprehensible to a human expert, because a computer can conduct a far more extensive search than a human player can. The main question explored in this paper is: can computers' problem solving be characterized by some intermediate goals that would help humans understand (and maybe learn from) computers' plans and intentions?

On the other hand, human problem solving does not involve exhaustive search, yet we are often able to solve some of the difficult problems - with far less searching and memorizing. Our problem solving relies on understanding intrinsic properties of the problem, such as important concepts, subgoals, and invariant features. However, this knowledge is often stored on the sub-cognitive layer, which makes our solutions hard to explain. Consider a simple problem of finding some function of an electronic device, say flash on a camera. An experienced user will be able to quickly find a way to disable flash on any device, however fail to instruct another user on how to do it without actually trying it first. This problem has been demonstrated in several domains. For example, it is difficult to reconstruct the skill of controlling a dynamic system, such as riding a bike or controlling a crane, mainly due to tacit properties of the expert's knowledge (Šuc, 2003). The question is therefore similar: can human problem solving be characterized by some intermediate goals?

The goal of our paper is not only to explain a single solution to a problem, but to reconstruct a strategy (or several strategies) from a set of problem-solving traces. A strategy represents a conceptualization of a domain that enables humans to understand and solve these problems comfortably. While the basic domain theory (such as rules in chess or laws of physics) is, in principle, sufficient for solving problems (e.g. knowing only the rules of chess could in theory enable optimal play), finding a solution using only the basic theory requires far too extensive searching. Using a strategy, however, a problem is represented with some higher-level concepts (subgoals) that can guide this search. Ideally, the number of these concepts should be low enough that they can be memorized by a human and enable fast derivation of solutions. Such conceptualization enables effective reasoning about problems and solutions in a domain (Možina et al., 2012; Tadepalli, 2008).

Our proposed approach learns a sequence of intermediate subgoals (called a strategy) that leads you from the initial problem to the final solution. Since there are usually several possible strategies leading to a solution, we decided to organize these strategies in an abstract structure called *goal-oriented* tree. To illustrate the idea, consider solving an equation with one variable x (letters

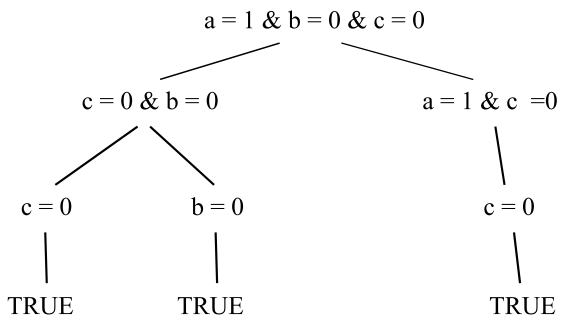


Figure 1: A goal-oriented tree for solving linear equations with one variable.

a, b, c, d represent constants):

$$ax + b = cx + d$$

We can solve this equation in three steps: one step to put both terms with variable x to one side of the equation; the next step to bring constants to the other side; and the last step to divide the equation by the constant at variable x. The resulting equation will have the term $1x$ on one side and the solution on the other side.

Figure 1 shows a goal-oriented tree learned with our method that embodies all possible strategies implementing the idea presented above. The root node represents a description of the goal states, which, in our case, are those with $a = 1$ (coefficient by the variable x) and $b = c = 0$. Other nodes in the tree correspond to intermediate subgoals that lead to the final goal.

Each path from a leaf node to the root represents one possible strategy. For example, the leftmost strategy starts by achieving goal $c = 0$. However, the strategy does not suggest how to achieve this goal, only what to achieve. A problem solver therefore still needs to figure out a way to achieve this goal; in our case by subtracting the term cx on both sides. The following goal $c = 0 \wedge b = 0$ is shared by two strategies that differ only in the order of steps. The rightmost strategy, however, shares only the main goal. It starts by achieving $c = 0$ first, as in the first strategy, but continues by achieving $a = 1$. Note that the goal $a = 1 \wedge c = 0$ can not be achieved in a different order: if we divided first to get $a = 1$, subtracting cx would then alter the value a and therefore not preserve $a = 1$.

In the following section we proceed by listing related work and comparing it to our approach. Afterwards we introduce the basic algorithm for learning goal-oriented strategies from a fully specified state-space. In the fourth section we demonstrate the algorithm on a larger domain: eight-puzzle problem-solving domain, where learning from a full state-space would be impractical, and computer-generated solution traces were used as learning examples. In the same section we introduce several extensions to the algorithm, such as active learning, that improve the quality of the learned goal-oriented trees. In the last technical section we present a different approach, where solutions of several human problem solvers (students solving programming exercises in Prolog) were taken as learning examples. The resulting strategies are common conceptualizations of Prolog programming of a number of students, both correct or incorrect. Finally we present our conclusions and pointers for further work.

2. Related work

The idea of representing problem solving knowledge in terms of goals was first introduced by several implementations of Advice Languages (Bratko, 2012; Michie, 1976). In Advice Languages an expert provides a sorted list of "pieces-of-advice", where each advice is composed of a goal and means to achieve this goal. A goal is specified in terms of descriptive predicates (similar to our approach), whereas the means is a subset of interesting moves that should be considered when solving the goal. Therefore, in Advice Languages each goal has a priority and goals are tested in the descending order of priority, while in our approach goals are organized in a hierarchy, achieving one goal is a prerequisite for another goal. Such hierarchical representation can be seen as a combination of Advice Languages and the means-ends approach to planning (Bratko, 2012).

An alternative model for representing problem solving knowledge is the ACT-R (adaptive character of thought – rational) (Anderson, 1993). The main goal of ACT-R is to model how humans recall "chunks" of information from memory and how they solve problems by breaking them down into subgoals using production rules (called the procedural knowledge). A production rule specifies how a particular goal can be achieved when a specified pre-condition is met. The pre-condition part also includes the goal to be achieved and the consequent contains the steps to achieve the goal. Although the model from this paper completely ignores the actions, these two representations are still interchangeable: to translate our representation to ACT-R the problem solver needs to automatically derive the actions achieving the selected goal.

The ACT-R theory is the formal background theory of the model-tracing intelligent tutoring systems (Woolf, 2008). One of the main challenges within tutoring systems is the difficulty of acquiring problem-solving knowledge from experts. Thus, while ITS are proving to be useful they are also difficult and expensive to build (Murray, 1999). They require complete domain knowledge, which requires a lot of knowledge engineering. And although several authoring tools (tools for building an ITS without the need of a programmer) were proposed, most of them still require manual elicitation of domain expertise (Murray, 1999). Furthermore, developers are faced with a similar problem when they try to update or improve knowledge models of existing systems. For example, maintaining the Andes Physics Tutor requires a full-time knowledge engineer (VanLehn et al., 2005). The alternative approach to author an ITS is to use machine learning to learn domain knowledge automatically. While it was shown that machine learning can be successful in building knowledge bases for expert systems (Langley and Simon, 1995) in terms of performance, such models usually do not mimic the expert's or student's cognitive processes when solving problems. We see our proposal as an alternative way to learn procedural knowledge, since we abstract from the specific actions and focus on more general subgoals that are arguably easier to learn. We have already demonstrated that goal-oriented learning can be used for learning procedural knowledge in the domain of KBNK chess ending (Možina et al., 2012).

There have been several attempts to learn strategies in game playing, especially in chess. The initial attempts focused on the illegality task, where the problem is to learn how to distinguish legal from illegal positions. The next research problem was to predict the number of moves needed to win (Bain and Srinivasan, 1995), which already implicitly defines a short-term strategy: during playing, one should search for moves that decrease this value. An algorithm for learning long term strategies from computer-generated tablebases was first presented by (Sadikov and Bratko, 2006). In their approach, a chess ending was first split into phases and then a classification tree was induced to classify a sample position into a mating phase. A problem solver is then supposed to find a move (or a series of moves) that transit a position to a phase closer to a solution. A survey of machine learning in games is given in (Fürnkranz, 2000).

Planning is another area where machine learning has been used to learn strategies. Interesting, if only partially related work is learning STRIPS operators from observed traces (Wang, 1994; Košmerlj et al., 2011). The idea is to learn the definitions of actions (and not goals) from provided solution traces. Their main goal is to obtain search-control knowledge to enable more efficient future planning. However, experiments with hierarchical task networks (HTN) suggest that planning could also benefit by having a hierarchical goal decomposition. In HTN, tasks are split into primitive and non-primitives and are hierarchically organized (Kutluhan, 1996). In other words, such organization defines which goals need to be achieved and in what order, which is similar to the output of our method. However, the hierarchy in HTN is provided by the knowledge engineer, while in our case it is learned automatically.

Learning to solve problem from exercises (Tadepalli, 2008) is another work researching related ideas. They explored how learning simpler skills first could help in learning more complex skills and applied it to learning of problem-solving skills. Their approach was demonstrated in several domains. For instance, in the 8-puzzle domain they first manually split the domain into

8 subproblems (phases) and sorted these subproblems by difficulty. They then learned a set of macro-operators that can bring you from one phase to the next phase. A similar approach was also applied to learning control rules and decision-list policies, in all cases showing improved computational efficiency of learning. It appears that out work is complementary to the work of Tadepalli, as our goal hierarchies can be used to decompose domains into subproblems.

In behaviour cloning, the goal is to develop an automatic controller mimicking the problem-solving skills of an experienced operator. The reconstruction process is similar to our learning approach: gather execution traces and reconstruct an automatic controller that can perform tasks in the same way as the operator. An automatic controller is usually a mapping from states to actions. An extensive review of approaches in behaviour cloning and an implementation using qualitative modeling is given in Šuc (2003). Note that our approach can also be used for behaviour cloning, however the actual actions would then need to be derived by the controller.

3. Learning goal-oriented strategies

3.1. Problem definition

We first define the problem of learning goal-oriented strategies. Items from the definition are explained in the following text.

Given:

- A state-space representation, where states are the learning examples.
- An attribute-based description of each example.

- A goal condition specifying the goal states in terms of attributes.
- A parameter "search depth" defining the maximal search depth.

Learn:

- A goal-oriented tree that *covers* all learning examples (states). A learning example is covered if there exists a node in the tree that covers this learning example. A node covers an example if the goal in the node is *achievable* in this example. We say that a goal is achievable if there exists a path (of length "search depth" or less) from this example (state) to another state, where conditions of the goal are met.

The algorithm described in this paper assumes that a problem is defined using a state-space representation. A state-space is a directed graph, where states (nodes) correspond to different problem situations and arcs between states correspond to legal transitions between problem situations. States where the main problem is solved are called goal states. Figure 2 shows a part of the state-space for the equation problem from the introduction:

$$ax + b = cx + d. \tag{1}$$

The initial state contains the original Equation 1. Each following state contains a new equation obtained by applying a chain of actions on the path from the initial node. In our case, possible actions include division and subtraction. For example, after applying the action $-cx$ that subtracts the term cx from both sides we reach a new state $(a - c)x + b = 0x + d$, effectively removing the term with the variable x on the right-hand side of

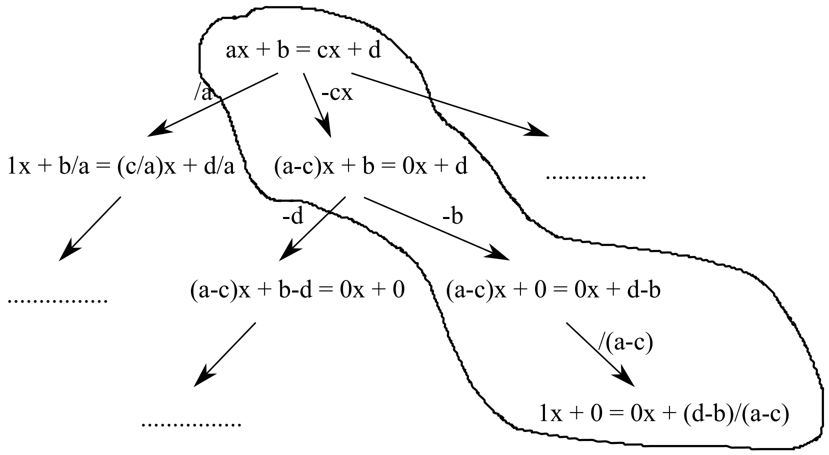


Figure 2: A part of the state-space for the equation problem.

the equation. After a series of such actions, a final state is achieved with the variable x on the left- and its value on the right-hand side. The final state is the right-bottom state in Figure 2.

Each example should be described with a set of relevant attribute values. These attributes are used for specifying the goal conditions (when the problem is solved) and by our algorithm to define subgoals and conditions that lead to those subgoals. For our example we used four discrete attributes: a (left coefficient of x), b (left free coefficient), c (right coefficient of x), and d (right free coefficient). To make the final goal-oriented tree easier to understand, we named the attributes the same as the initial values in the original Equation 1. Attributes have three possible values: 0, 1, and *other*. First two values correspond to actual values of the coefficient (0 and 1), whereas the last represents any other value. The goal condition of this domain is: $(a = 1 \wedge b = 0 \wedge c = 0) \vee (c = 1 \wedge d = 0 \wedge a = 0)$.

Parametrizing the maximal search depth for solving subgoals allows us to tailor the learned strategies to the expected skill level of the problem solver.

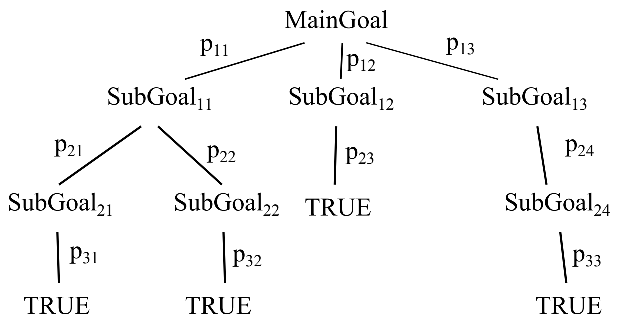


Figure 3: The generalized format of a goal-oriented tree.

If the problem is to be solved by a computer, the depth can be set to a higher value, while for a novice human problem solver (*e.g.* a math student), the search depth should be set to a relatively low value.

A goal-oriented tree represents a hierarchy of goals. A generalized goal-oriented tree is given in Figure 3. A parent goal can be achieved (with some probability and within the provided maximum depth) if any of the child goals are achieved. A parent-child pair $ParentGoal \leftarrow p \leftarrow ChildGoal$ should therefore be interpreted as: the probability of achieving the $ParentGoal$ goal from the $ChildGoal$ within the provided number of steps is p. $MainGoal$ is the predefined final goal to be achieved (e.g. solving an equation, or mating in chess). Each subgoal is represented as a conjunction of attribute-value pairs, similar to a complex in rule learning (Clark and Niblett, 1989). A complete branch in such a tree represents a single problem-solving strategy. Therefore, if any of the subgoals is achievable in a particular case, then the corresponding branch leading to the root is the solution for this particular problem instance. If there are several such subgoals, the corresponding branches represent alternative solution strategies. And finally, if a subgoal has no conditions (such as leaves in our example tree), then that particular branch solves all instances of the problem.

The task of goal-oriented learning is to learn a goal-oriented tree that covers every learning example.

3.2. The algorithm

The skeleton of the algorithm for learning goal-oriented trees is given in Alg. 1. Description of the algorithm follows.

The inputs of the algorithm are: a) the main goal that defines the solution of the problem, b) a set of learning examples that are possible states described with attributes, and c) a parameter *depth* specifying the maximal search depth. For now we will assume that the learning examples contain all possible states. The result of learning is a tree structure where each node is represented by three values: the subgoal, the probability of achieving the goal in the parent node, and the set of examples solved by this branch.

Algorithm 1: The basic goal-oriented learning algorithm. Methods `selectGoal` and `ruleLearner` are components of the algorithm that can be changed according to the domain.

```
input : finalGoal (main goal), learningData(described states from a
        state-space), depth(maximal search depth)
output: goalTree(a goal-tree learned from examples)

  /* A node in the tree is represented by a triple: (goal
     to be solved, solve probability, solved examples)     */
1 let goalTree contain only root node (finalGoal, 1.0, ∅).
2 mark finalGoal as unexpanded.
3 while goalTree contains unexpanded goals do
      /* Select an unexpanded goal in goalTree               */
4     let selectedGoal, prob, prevSolved = selectGoal(goalTree)
5     mark selectedGoal as expanded.
      /* For every example determine if the selected goal can
         be achieved (within depth steps) or not.            */
6     let alreadySolved = positiveExamples = negativeExamples = ∅
7     for each example in learningData do
8         if example in prevSolved then
9             continue        /* skip previously solved examples */
10        end
11        if selectedGoal already achieved in example then
12            add example to alreadySolved
13        else if selectedGoal is achievable in depth moves then
14            add example to positiveExamples
15        else
16            add example to negativeExamples
17        end
18    end
      /* Learn rules that separate positive from negative
         examples: ``IF condition THEN class=positive''.
         */
19    rules = ruleLearner(positiveExamples, negativeExamples)
20    let newSolved = prevSolved ∪ positiveExamples ∪ alreadySolved
21    for each rule in rules do
22        add (conditions of rule, accuracy of rule, newSolved) as a child
          node to selectedGoal node.
23    end
24 end
```

Initially the tree contains a single unexpanded node representing the final goal. The following main loop continuously processes one unexpanded node at a time until there are no unexpanded nodes left in the tree.

The loop starts by selecting an unexpanded goal. Our implementation uses breadth-first search and first expands the goals that are closer to the root node. A more sophisticated approach would be to expand the most promising node. However, to make such an approach efficient, we would also need to implement pruning of unnecessary branches. We have not yet investigated this option and leave it for future work.

After a goal was selected, the learning data is partitioned in four disjoint subsets:

previously solved are examples that were already solved by previous goals in this branch,

already solved are examples where the selected goal is already achieved,

positive examples are examples where the selected goal is achievable within *depth* moves, and

negative examples are examples where the selected goal is not achievable.

Given these sets of examples, we need to learn a pattern or a rule that describes the states where the goal can be achieved. We accomplish that by learning classification rules to separate positive examples (where the goal can be achieved) from negative examples (goal can not achieved). The conditions of the learned rules therefore define subspaces from where the above selected goal is achievable, hence these conditions become the new subgoals - the descendants of the selected node.

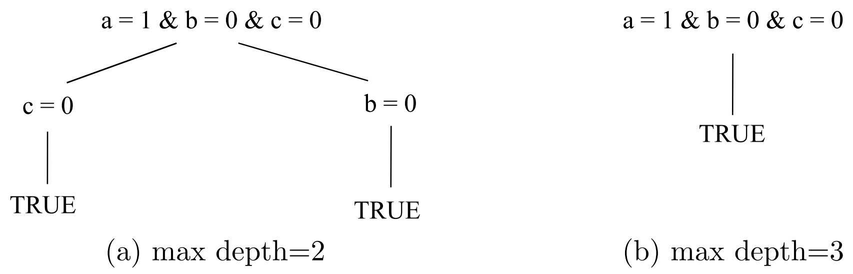


Figure 4: Goal-oriented trees for solving one equation with one variable with different maximal depths.

It is the rule-learning algorithm that will mainly determine the properties of our learned strategy. Do we want specific goals and high probabilities of achieving one goal from another, or are we looking for less accurate transitions and more general goals? We decided to use the CN2 algorithm (Clark and Boswell, 1991) with m-estimate of probability as the measure for evaluating rules (Džeroski et al., 1993). With the m-parameter we can guide the algorithm to learn different kinds of hypotheses: higher values of m will lead to more general strategies, while lower values will result in more specific strategies. The default value of m-parameter is set to 20 as suggested by Janssen and Fürnkranz (2010).

The final step of the algorithm adds the new subgoals to the goal-oriented tree. The conditions of the learned rules become new subgoals, while new solved examples are all previously solved examples together with all examples where the selected goal is achievable. The probability of achieving the parent goal is the class accuracy of the rule generating the new subgoal.

When the Algorithm 1 was used on our equation problem from the introduction, it produced exactly the tree described in introduction. When search depth was increased to 2 and 3, we obtained the trees from Figure 4.

3.3. Another example: two equations with two variables

We will now consider a more complicated example of two equations with two variables x and y:

$$ax + by = c \tag{2}$$

$$dx + ey = f \tag{3}$$

Let each state be described with six discrete attributes: a, b, c, d, e and f that correspond to parameter values from above equations. The possible values for each attribute are 0, 1, and *other*, as in the previous example. The set of learning examples included all possible states of these four attribute that can be achieved by the two basic operations: dividing an equation by the coefficient of x or y, and adding (or subtracting) one equation from another, together consisting of 80 learning examples.

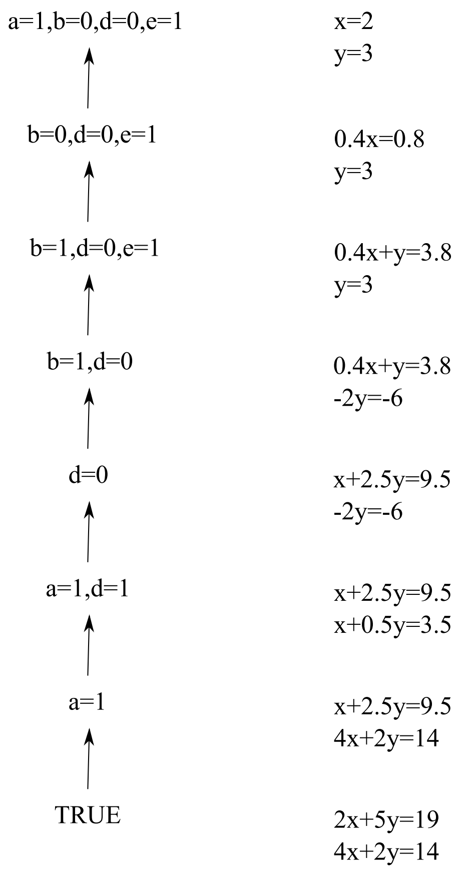
Figure 5 shows two strategies, learned with parameters *depth* = 1 and 2, respectively. Since the complete tree is too complex to visualize and the learned strategies from different branches are similar, we selected only one strategy from the tree. The estimated probabilities on edges are all equal to 1.0 (all rules had pure class distributions without any negative examples).

To better understand the learned strategy, we also prepared a sample two-equations problem:

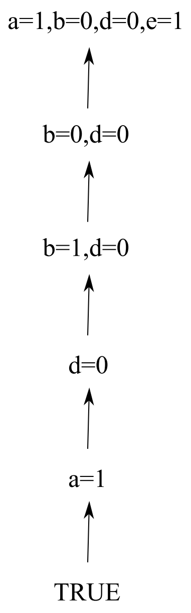
$$2x + 5y = 19 \tag{4}$$

$$4x + 2y = 14 \tag{5}$$

The step-by-step solution of this problem is given in Figure 5 next to the



(a) Depth = 1



(b) Depth = 2

Figure 5: Strategies for solving two equations with two variables. Different depths were used while learning these two strategies. In the left strategy, each goal can be achieved directly from the goal below with one step only, while in the right strategy the depth was set to 2, hence we need 1-2 steps to achieve the next goal.

strategy computed with parameter *depth* $= 1$. The solution starts at the bottom with the original problem and finishes at the top with the result $x = 2, y = 3$. Each step of the solution can be interpreted by looking at the corresponding subgoal in the strategy.

4. Learning goal-oriented strategies from large state spaces

4.1. Learning from traces

The algorithm in the previous section assumes that a complete problem state space is given as a set of learning examples. However, a large number of states could render our algorithm unacceptably slow, and a subset of examples (states) needs to be selected instead. We decided to sample with traces, where a set of solutions traces is first constructed, and then each step from a trace becomes a learning example. Sampling with traces is preferred over random sampling because it covers all problem-solving phases, whereas random sampling might neglect some (critical) parts of the state-space.

The learning algorithm from the previous section can already learn from such sample data. However, as we will show in the remainder of this section, it performs poorly if not properly adjusted. The necessary improvements to the algorithm will be described in the following sections.

4.2. Trace coverage and the one strategy – one trace principle

A solution trace is a sequence of states $t = s_1, s_2, \ldots, s_n$, where s_1 is the starting problem, s_n is the final state where solution is achieved, and the two consecutive states s_i, s_{i+1} are directly connected in the state-space. A strategy S in a goal-tree *covers* a trace t if there exists such $i < n$ that the partial trace $s_i, \ldots, s_n$ can be derived from this strategy. In other words, with the selected strategy S we should be able to recreate this trace from the i-th state onward. We will denote this covering relation as $covers(S, t, i)$.

Using the covers relation we can define the *one strategy - one trace principle*, which requires each learned strategy in the goal-tree to uniquely cover at least one learning trace. In other words, for each learned strategy there must exist a trace where this strategy performs best. Or formally, for each strategy S in a goal-oriented tree, a covered learning trace t_k should exist so that $covers(S, t_k, i)$ holds, and no other strategy covers equal (or larger) part of this trace: $\nexists S', j : b \neq a \wedge j \leq i \wedge covers(S', t_k, j)$.

The above requirement filters out unnecessary strategies. An interesting consequence of this requirement is that every strategy will follow at least one particular learning trace. We believe that such strategies are more likely to be correct, as they characterize a particular solution.

We implemented the requirement as an additional pre-pruning condition in the rule-learning method. Every node in the goal-tree now also contains a list of all uniquely covered traces, and the rule-learning algorithm needs to learn such rules that cover at least one of these traces. If we need to achieve greater generalization (over several traces), we can easily extend this requirement by enforcing that each strategy uniquely covers several traces.

4.2.1. Implicit conditions

A fundamental problem of traces is that they provide mostly only positive examples for the rule learning. The lack of negative examples leads to overgeneralized rules, since learning is not considering unseen examples. Adding implicit conditions to the learned rules turned out to be beneficial.

After a rule is learned, there usually exist several conditions that are not included in the condition part of the this rule, yet they do cover all positive examples covered by the original rule. Adding such conditions will not decrease the number of covered positive examples, however it might decrease the number of negative examples – both covered and those yet unseen. We call such conditions *implicit conditions*. The distinction between the originally-learned rules and rules with added implicit conditions is analogous to most general and most specific hypotheses from the version-space theory (Mitchell, 1997).

4.2.2. Active learning

An alternative way of dealing with the lack of negative examples is to introduce new examples through active learning. In problems where new examples can be easily generated, we can use active learning to generate only those additional examples needed to learn more accurate rules.

Algorithm 2 shows our implementation of single rule active learning. In the first step we use a standard method for learning a single rule from positive and negative examples, such as described in CN2 algorithm (Clark and Boswell, 1991).

If the learned rule already covers enough negative examples (in our case this threshold was set to 200), adding new negative examples is unlikely to help and the rule can be returned by the method. Otherwise, we need to generate new examples covered by the learned rule. Our generation technique executes a breadth-first search in the problem state-space, starting with the examples already covered by the rule (positive and negative). When a certain number of new covered examples are found (in our case 20) or when a large number of examples haven been explored (in our case 1000), the search procedure terminates.

The newly found examples are split into positive, from where the active goal can be achieved, and negative examples. If new examples contain only positive examples, then we have failed to generate new negative examples, and choose to select the rule as best rule and return it. Otherwise new examples are added to positive and negative learning examples. This loop is repeated until one of the stopping criteria is fulfilled.

Algorithm 2: An algorithm for active learning of a single rule. The method `findBestRule` is in our case the same as the method `findBestCondition` from the CN2 algorithm (Clark and Boswell, 1991). The threshold for covered negative examples T was set to 200, the number of generated examples n was set to 20.

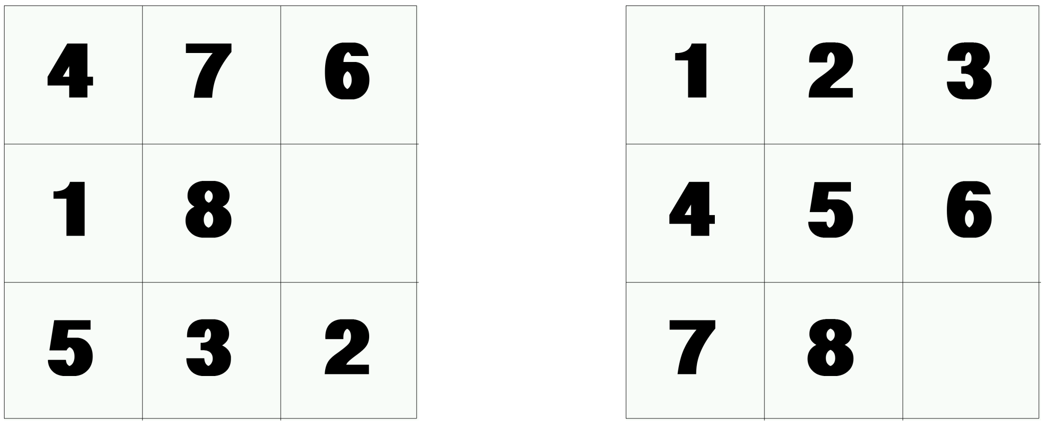
```
input  : positiveExamples, negativeExamples
output: rule(a single best rule learned from examples)
1  while True do
2      let rule = findBestRule(positiveExamples, negativeExamples).
3      let neg = number of negative examples covered by rule.
4      if neg > T then
5          return rule.
6      end
       /* Generate n new examples from state-space that are
          covered by rule.  Examples are generated using
          breadth-first search starting at currently covered
          examples.  Search is continued until n new covered
          examples are found.  If a large number of examples
          are explored without finding enough examples, the
          search is stopped and less than n examples are
          generated.                                          */
7      generate newExamples covered by rule.
8      split new examples to newPositiveExamples and
       newNegativeExamples.
9      if len(newNegativeExamples) == 0 then
10         return rule.
11     end
12     add newPositiveExamples to positiveExamples.
13     add newNegativeExamples to negativeExamples.
14 end
```

4.3. Evaluation

4.3.1. Domain: the 8-puzzle game

We chose the 8-puzzle game to demonstrate goal-oriented learning from traces, as this domain is understandable and large enough (362880 states) that we need to resort to sampling. In this game, one has to find a series of moves that lead from an arbitrarily-shuffled position to the goal position, where tiles are ordered. Figure 6 shows a random starting position and the final position with ordered values from 1 to 8. In some cases , where it will be necessary, value 0 will be used to represent the empty tile. In each step a tile next to the empty tile can be moved to the empty place, so that the tile and the empty place swap positions. The goal is to find a series of moves that will lead us from the starting position to the goal position.

Using the 8-puzzle domain, we shall illustrate the problems of the basic algorithm when learning from a sample of traces, and evaluate the benefits of suggested improvements in terms of problem solving efficiency. To estimate the efficiency of learned strategies, we preselected 1000 random starting positions and used learned strategies to solve these positions. In cases where several strategies were applicable, we compared their lowest transition probabilities (the weakest link in the strategy) and selected the strategy with highest lowest probability (the maxi-min principle). We used the chosen strategy to select moves until the final position was achieved or the threshold of 1000 moves was reached. Whenever the successive subgoal in the strategy was not immediately reachable within the number of allowed moves, a random move was executed. The success rate of a goal-oriented tree was measured with the following statistics:



(a) A random starting position (b) The final position

Figure 6: Two different positions from the eight puzzle game

# solved The number of successfully solved positions (out of 1000). This measure estimates the overall success of the learned strategies.

avg. start dtg Average starting distance-to-goal of solved positions (the number of steps required to solve the problem if optimal moves are played). We used this measure to test whether the learned strategies tend to solve problems that are closer to the solution (low starting dtg).

avg. num of steps Average number of steps needed to solve the problem (only solved positions were considered).

For the sake of clarity, we decided to use a relatively simple attribute description of the domain. Each state was described with 81 binary attributes, where each attribute is an item from a cartesian product between tiles (9 values) and squares (9 values). For example, the attribute “tile 3 at square 2” equals true if tile 3 is at the square 2, and false otherwise. This representation enables easy interpretation of the learned strategies. However,

Table 1: The tables compare different strategies for solving 8-puzzle with different search depths, with or without implicit conditions, and with our without active learning. We used 50 learning traces to train the goal-oriented trees, the m-parameter set to 20. The learned strategies were tested on 1000 random starting positions. The first row shows the number of solved problems (out of 1000) within 1000 moves. The last two rows give the starting average distance-to-goal of solved positions and the average number of steps used to solve these problems.

(a) Without active learning.

Search depth	3		5		7	
Implicit conditions	no	yes	no	yes	no	yes
# solved	27	357	139	562	960	993
avg. start dtg	17.63	21.59	21.07	21.43	21.78	21.79
avg. num of steps	230.44	164.80	307.19	171.52	287.10	147.77

(b) With active learning.

Search depth	3		5		7	
Implicit conditions	no	yes	no	yes	no	yes
# solved	232	538	238	893	572	910
avg. start dtg	21.18	21.66	20.34	21.78	21.25	21.84
avg. num of steps	240.41	434.72	35.56	89.38	57.95	43.20

better attribute description would lead to better strategies. Initial experiments suggest that simply changing these 81 binary attributes to continuous, where each value is the manhattan distance between the corresponding tile and a square, results in more accurate strategies. More sophisticated attributes that are usually used in the 8-puzzle problem, such as the overall manhattan distance or the sequence score (Bratko, 2012), should therefore result in even more accurate and succinct strategies.

4.3.2. Results and Discussion

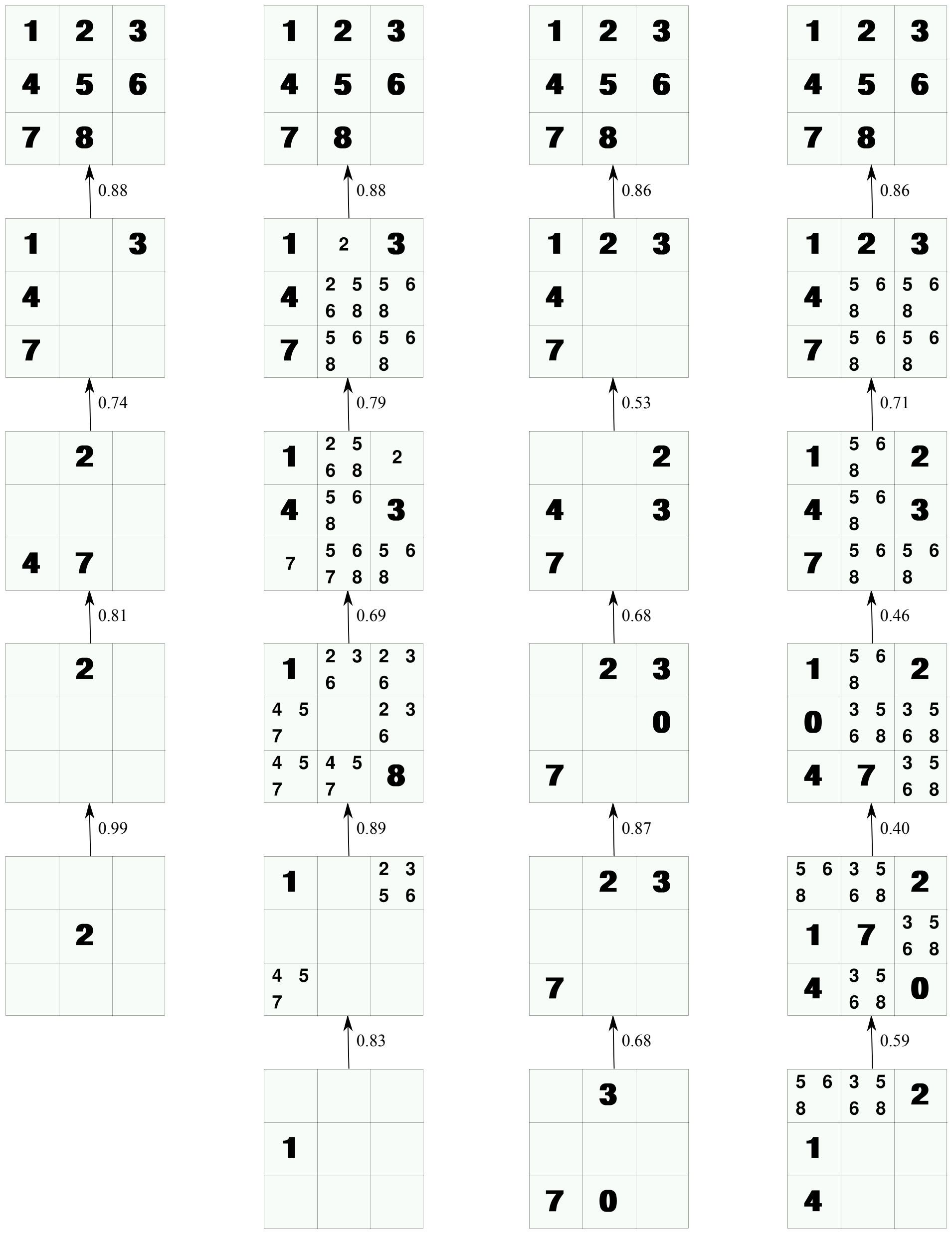
Tables 1a and 1b contain results of performance of learned strategies using implicit conditions, active learning, and different search depths (3, 5, 7). We can notice that increasing the search depth consistently leads to higher problem-solving success rate, independent of other parameters. This result was expected, since higher search depths usually mean fewer subgoals in a goal-oriented tree and are, therefore, easier to correctly conceptualize. However, these better results come at a higher solving cost: it puts a greater burden on the problem solver using this strategy, as transition between subgoals requires more moves, which is usually more difficult to find.

The results also imply that strategies learned with implicit conditions will solve problems better than those without implicit conditions. In experiments, with or without active learning, implicit conditions always increased the number of solved problems. The main reason can recognized by analyzing strategies from Figure 7. Each sub-figure in Figure 7 visualizes the most commonly used strategy from tested goal-trees; without and with implicit conditions and active learned strategy with and without implicit conditions. Search depth in all cases was 5.

Figures 7a and 7b contain strategies learned without and with using implicit conditions, respectively. In both figures, the topmost diagram is the final goal. The diagrams immediately after the top diagrams are the prerequisite subgoals for solving the main goal. The diagrams following are then their prerequisites, and so on. For example, in the second diagram of the first figure, we have tiles with numbers 1-4-7 vertically aligned in the first column and tile 3 in the top-right square. This strategy, therefore, suggests achieving this pattern first and afterwards we should be able to reach the final goal in 88% of cases. This estimation is, however, highly optimistic. The majority of positions with this 1-4-7-3 pattern are still far from the final goal, yet our algorithm did not encounter such examples while learning the goal-tree.

This problem is partially solved by adding implicit conditions. In the second diagram in Figure 7b the small values represent implicit conditions, namely all positions of other tiles (not 1,4,7,3) that were encountered in the learning examples. Notice that the tile with value 2 occurred only in the upper-middle square, which is what makes the probability estimate 88% realistic. Apparently our problem solver (computer) never came across this 1-4-7-3 pattern, while having tile 2 on bottom side. The results from experiments show that such additional conditions significantly increased the probability of achieving the following goal and therefore increase the success of the strategy as a whole.

The results for active learning are a bit less favorable. While in cases with search depth 3 and 5 the success percentage indeed increased, there is a case (search depth 7 without implicit conditions) where success significantly decreased with active learning. This result was especially surprising after we investigated the learned strategies. The strategies learned with active learning are considerably more specific than those without active learning. For example, when comparing Figure 7c with Figure 7a, we noticed that active learning correctly included the tile 2 into the subgoal from the second diagram. With tiles 1,2,3,4,7 on their final position, it is almost always possible to solve the puzzle within 5 moves. Furthermore, the subgoals from the third diagram in Figure 7c, with tiles 2 and 3 in the last column, contains a position that is typical to a human problem solver. The only flaw of this diagram is the missing tile 1 in the top left corner, which can be solved with implicit conditions, as seen in the third diagram of Figure 7d. After a careful



(a) Without implicit conditions (b) With implicit conditions (c) Active without implicit conditions (d) Active with implicit conditions

Figure 7: The figures show the most common strategies of the respective goal-oriented trees. Search depth was 5 in all cases, the m-parameter was set to 20. Due to space restrictions, only last 5 steps of strategies are visualized. When a square has no labels, it can either mean that it is empty or that there could 5 or more different tiles.

interpretation of all learned strategies, we conclude that the combination of active learning and implicit conditions is arguably the best. However, the question remains: why doesn't this combination always bear best result in terms of problem solving efficiency? It does with search depth 3 and 5, but not with depth 7. This issue requires more experimentation.

Given the results, the most accurate strategy from Figure 7 is the one with active learning and with implicit conditions. We can translate the diagrams from Figure 7d into a human-understandable strategy, where we also included steps from the part of the strategy that was visualized in the diagrams. Therefore, starting from a random position, do:

1. Move tile 4 to the bottom-left square.
2. Move tile 1 to the square above tile 4.
3. Move tile 2 to the top-right square. At the same time take care that tile 7 does not occur at the top row and tile 3 is not at top-left square. Note that we reached the last subgoal from Figure 7d.
4. Move tile 7 to the middle square and empty tile to the bottom-right square. Tile 3 should not be at the top-left square.
5. Move the empty tile in the following order: left, up, up, left, down. This exact order is the only sequence of five moves that will reach our next goal.
6. Move the empty tile down and right to achieve the 1-4-7 column. Then, move tile 3 on the square below tile 2.
7. Reach pattern 1-2-3 in the top row. This should be quite easy, since tiles 2 and 3 are already in the right order.
8. Rearrange tiles 5, 6, and 8 to achieve the final goal.

We encourage the reader to try the above strategy.

Another relevant question is how the m-parameter impacts learned strategies. To this end, we tested various m-parameters in rule evaluation measure using active learning, implicit conditions, and both combined. Tables 2 show results of this experiment. The strategy from the last Table 2c, with combined active learning and implicit conditions, achieves almost perfect efficiency when m-parameter is set to a low value, in our case 2. Remember that low m values in m-estimate of probability give higher weight to actual covered data and lower weight to prior probability. Such learning of rules is less biased, which can potentially result in more accurate rules. However, less bias leads to higher variance and to higher probability of overfitting. Yet apparently, the combination of active learning and implicit conditions seems to mitigate overfitting.

Having low values of m-parameter introduces another problem: the number of learned strategies increases. In all three cases, the number of learned strategies decreased if m-parameter was increased. In cases with low m value, the number of strategies was around 50. Considering that we only have 50 learning traces, it seems that the majority of learned strategies cover only a single learning trace. To understand the problem better, we visualized the last five steps of the most commonly used strategy in Figure 8. The first diagram is, as always, the final goal. The following two subgoals are similar to subgoals in the strategy described above. However, these two diagrams are similar to the previous two diagrams: the squares with fixed tiles are in symmetrical positions. Therefore, achieving the subgoal from the fourth diagrams is as difficult as achieving the subgoal from the second diagram,

Table 2: Comparison of strategies for solving 8-puzzle. Each column shows results for a different value of m-parameter in the rule evaluation measure. Learned strategies were tested on 1000 starting positions, search depth set to 5, number of learning traces is 50.

(a) Only implicit conditions

Parameter m	2	10	20	100	1000
# solved	561	651	562	639	706
avg. start dtg	21.66	21.70	21.43	21.20	21.55
avg. num of steps	111.94	196.87	171.52	215.42	345.78
# of strategies	49	46	43	40	30

(b) Only active learning

Parameter m	2	10	20	100	1000
# solved	294	259	238	198	208
avg. start dtg	20.49	20.59	20.34	20.35	20.98
avg. num of steps	37.85	40.42	35.56	73.93	238.26
# of strategies	50	42	42	27	10

(c) Active learning and implicit conditions

Parameter m	2	10	20	100	1000
# solved	999	993	893	724	756
avg. start dtg	21.80	21.81	21.78	21.69	21.80
avg. num of steps	116.65	111.10	89.38	126.71	113.82
# of strategies	49	43	43	29	8

which effectively makes the last two subgoals irrelevant. Why achieve these subgoals if it is equally hard to achieve the first subgoal that leads us directly to the final goal? The subgoals in Figure 8 are due to low m values evidently over-specified, namely they are fitted to a specific trace.

Results indicate that using active learning and implicit conditions with low values of m yields the most efficient strategies. This efficiency, however, comes at the cost of generality. This is acceptable when dealing with a computer problem solver that can remember many subgoals, however such strategies do not fit a human problem solver. The parameter m should be

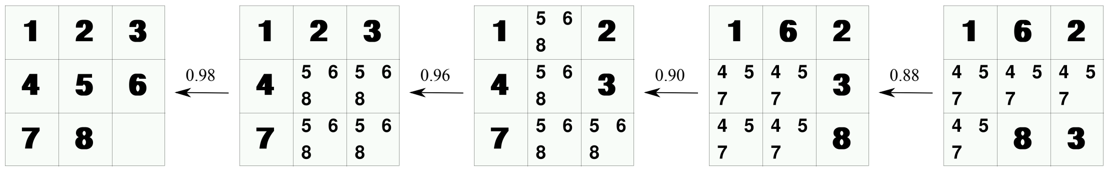


Figure 8: The most often used strategy learned with active learning, with implicit conditions, and m-parameter set to 2.

adjusted to learn strategies that are appropriate to a human problem solver.

5. Learning goal-oriented strategies from partially defined state spaces

In large or complex domains it is impractical or even impossible to fully specify the state space. Consider, for example, solving programming exercises. The number of possible actions available to a programmer at each step is enormous, resulting in a large state space with a high branching factor. Even basic operations, such as finding possible successors to a given state, can be quite challenging in such domains.

Nevertheless, automatic analysis of programming domains is a rewarding topic. Acquiring problem-solving knowledge in a form usable for intelligent tutoring systems is a long-standing issue. Most published approaches use handcrafted procedural rules; see for example (Corbett and Anderson, 1995).

Recently, Mostafavi and Barnes (2010) suggested a data-driven approach to constructing intelligent tutoring systems: gather the students' problem-solving traces and use them to derive a partial state space. The state space is indeed only partial, as students' solutions will not cover the entire space. However, it does cover the relevant subset of states - those that were actually reached by students. Initial published results seem to be promising.

Our educational data was gathered during Prolog courses (spring semesters in 2013 and 2014) at University of Ljubljana (Lazar and Bratko, 2014). The students had an option of solving lab exercises in a specialized editor where every keystroke was logged. For each attempt we thus obtained a sequence of program versions from the empty program to a correct solution, with every version inserting or deleting one character.

A trivial way of building a state space from such traces would be to consider each version a separate state, with transitions between states corresponding to character insertions and deletions. To reduce the state-space size, we extend transitions to include continuous sequences of character edits. So, for example, typing the sequence of characters `member` would result in just one new state. In other words, a new state is created only when the student moves the editing cursor to a new location.

We lexed each program version into tokens to remove variability due to whitespace and comments, and normalized variable and predicate names. A state space was then formed out of these solutions traces. For each state we also note whether the program is correct or not according to preset test cases. A more detailed description of the data and preprocessing steps can be found in the original paper (Lazar and Bratko, 2014). We used n-grams of program tokens (with n ranging from 4 to 10) as attributes for describing states, required for goal-oriented learning.

Given that the problem solvers in this case are the students, the results of a goal-oriented conceptualization should in fact represent the thought process of these students. The remainder of this section describes and discusses the results of goal-oriented learning on two basic Prolog predicates: `member` (tests whether an item is in a list) and `del` (deletes an item from a list).

5.1. The `member` predicate

In one of the exercises of the Prolog course, the students are asked to write a program to check whether a list contains a certain element. The following three lines show the most common program implementing the `member` predicate:

```
% member(X, L): test whether element X is in list L.
```

```
member(X,[X|_]).
member(X,[_|T]):-
  member(X,T).
```

The first line is a comment describing the program's function. The first clause (the second line) covers the base case where the selected element is at the front (head) of the list. If `X` is not the first element in the list, the second (recursive) clause will be used to reduce the problem of finding `X` in the list `L` to a simpler problem of finding `X` in the tail (all elements but the head) of the list `L`.

We created a state space with 1170 different states from 149 solution traces (after applying normalization). We used our algorithm for learning goal-oriented strategies with maximal search depth set to 1, m-parameter to 20, and using implicit conditions. Active learning was not used since we can not generate new states.

Figure 9 shows a part of the learned tree. It shows the main branch and the main alternative at each step. We manually translated sets of n-grams representing goals to partial programs to increase readability. The root node represents the main goal, namely the set of programs that pass the test set. However, the figure shows only the most common program, where the actual programs may be different from this one. The arrow labels show class distributions of the learned rules: the number of observed programs matching the current subgoal from which the next goal was or was not achievable.

We first take a look at the most common strategy, represented by the left-most, vertical branch. The main goal (solution) is at the top, followed by a partial solution that leads to that goal. The two lines in the first subgoal are

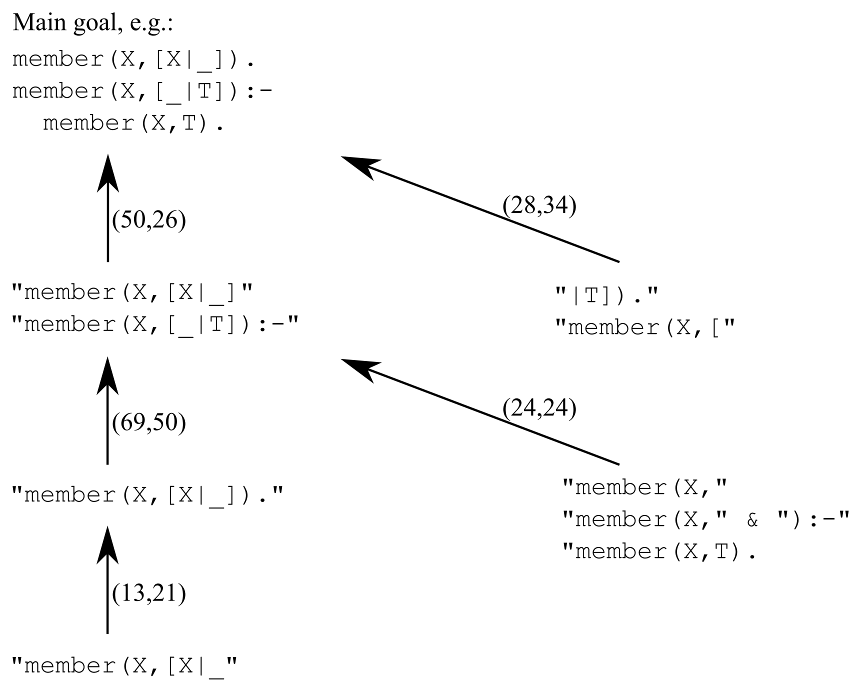


Figure 9: Main branch and alternatives from the goal-tree for `member`. Search depth was set to 1. Values in parentheses represent success rate of goal transition (positive, negative).

almost identical to the first two lines of the solution, and in 50 out of 76 cases the students were able to achieve the main goal with a single continuous edit. Similarly, the next subgoal in the main branch represents programs with only the base case, and, again, the students were able to correctly write the head of the recursive rule (to achieve the next subgoal) with a single transition in 69 out of 119 states.

This result is surprising because it implies that the majority of students solved the problem by simply typing out the correct program without editing. The course teachers explained that this is the introductory exercise for explaining lists in Prolog, and is indeed usually solved on the whiteboard by the teacher. Our algorithm has thus correctly identified the most commonly used "strategy" for this problem – copying the solution from the whiteboard.

Some students did solve the exercise on their own. The alternative branches in Figure 9 shows two strategies potentially describing these students. The first such strategy (alternative subgoal in the second row) uses a named variable `T` in place where the canonical solution uses an anonymous variable _. This is likely due to novices' unfamiliarity with anonymous variables in Prolog. The other alternative branch (in the third row) demonstrates a different behavior. Here, the students correctly wrote the body of the recursive clause first, and only then corrected the remaining lines. The branch represents solutions that either made a syntax error, for example having a list without the closing bracket `member(X,[_|T):-` , or the generic `member(X,L)` line was used in the initial rules, and was only later refined into the correct versions.

5.2. The `del` predicate

In this exercise students implement a predicate to delete an item from a list. The typical solution of the `del` predicate consists of two clauses:

```
% del(X, L, L2): delete element X from L, result is list L2.
del(X,[X|T], T).
del(X,[H|T], [H|T2]):-
  del(X,T,T2).
```

The solution is similar to that of the `member` predicate. The first clause deletes the item if it is in the head of the list `L`, while the second clause deletes the item from the tail of `L`.

We collected 142 solution traces, resulting in a state space with 1794 states. Unlike in the `member` exercise, all students implement the `del` predicate on their own. For that reason, goal-oriented learning did not provide meaningful results with maximal search depth set to 1; we had to increase it to 3. All other settings were the same as in the previous section. Figure 10 shows the most common strategy and the main alternatives in the goal-oriented tree constructed by our method.

Again we first look at the main strategy, represented by the left-most branch. The description of the first subgoal on this branch is relatively short with only five characters: "`T],[H`". This attribute relates to the head of the recursive clause, where the last argument must be a compound list with `H` as the first element. Once a student figures out this pattern, the probability of completing the exercise within the next three steps is very high: in 81 out of 101 cases the students were able to finish the program. This result aligns with the teachers' experience – forgetting to add `H` at the beginning of the

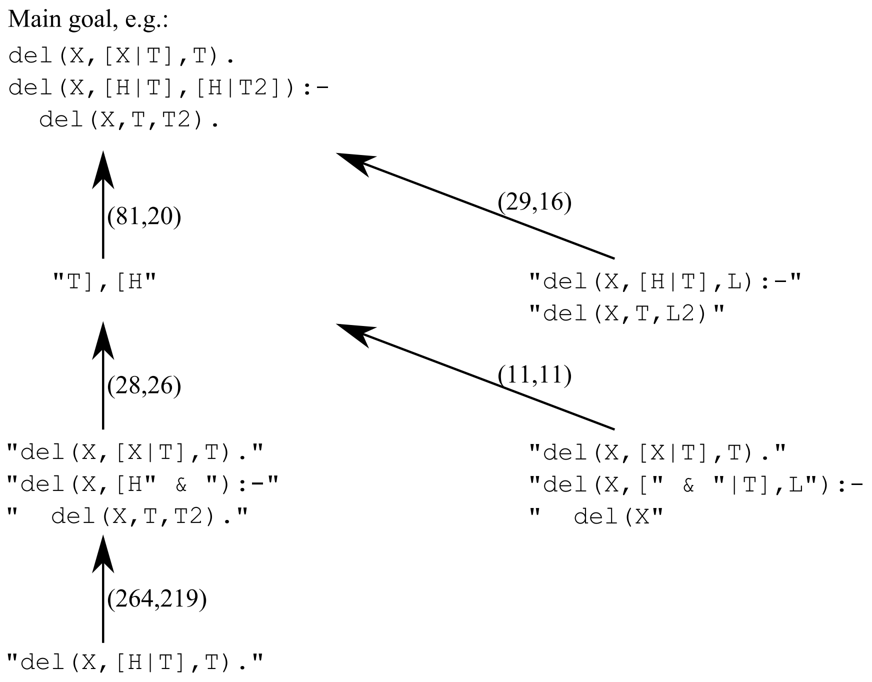


Figure 10: Main branch and alternatives from the goal-tree for `del`. Search depth was set to 3. Values in parentheses represent success rate of goal transition (positive, negative).

recursively processed list is one of the most common and persistent mistakes for this problem.

The subgoals leading to this goal (third row in Figure 10) are fairly similar, with a correct base clause and a partially finished recursive clause. However, both cover only a small number of cases – 54 and 22, respectively – and have an approximately 50:50 class distribution. We hypothesize that this part of the state space is difficult to conceptualize. This is also supported by the teachers' observation that many students struggle with the definition of the recursive clause; they often resort to "randomly" tweaking the program, which can confound the state space.

Going one step further in the learned strategy we found that it starts with the correct base clause: "`del(X,[X|T],T).`". This node is well represented in our strategy with at least 483 (264+219) states. In 264 of those states, students were able to reach the next goal in three or fewer steps, but only some of those solutions reached the next subgoal. Other students simply skipped this intermediate subgoal and moved directly to the subgoal with the "`T],[H`" pattern. This result also confirms our previous hypothesis that it is difficult to formulate intermediate concepts that would help students getting the last argument right in the recursive clause.

Figure 10 also shows an alternative way of achieving the main goal, where the recursive rule is completed first. The same issue as before can be found in this branch: instead of returning a new list `L`, the recursive clause should return `[H|L2]`. This strategy therefore suggests that writing down the recursive call first will aid students getting the "output" in the recursive rule right. Once a student has figured it out, they are usually able to finish the program quickly. It should be noted, however, that the distribution of this strategy is notably worse with 29 positive and 16 negative cases.

We demonstrated that learning strategies from solution traces can find some interesting programming patterns in Prolog. The comments given by teachers confirm this, as the learned patterns often agree with what they observed during teaching. The learned strategies are however not perfect, since the used attribute space (n-grams) is agnostic - the models were learned without any expert knowledge. A better solution would be to acquire and consider meaningful concepts provided by teachers. Strategies learned from such concepts would be, for example, less prone to syntactical errors, which did cause some misconceptions in the above-described strategies. Moreover, such strategies would also conform to how student think, being an important step before these strategies could be used within a programming tutor to suggest possible ways to proceed in certain situations.

6. Conclusions and future work

We described an algorithm for learning goal-oriented strategies in problem solving domains. The learned strategies are organized in a tree-like structure, with the main goal itself at the root of the tree, and the intermediate goals at other nodes. To achieve the main goal one needs to achieve the intermediate goals first, starting with a goal from a leaf.

We demonstrated our algorithm on three domains: solving equations, 8-puzzle game, and Prolog programming. The domain of equation solving is an example of a small domain, where the complete state-space representation can be used in learning. This enabled us to learn full strategy trees covering every possible strategy that leads to the solution of a set of equations.

In the case of the larger 8-puzzle game domain, learning from the full state-space becomes infeasible. Instead, it is necessary to learn from a sample of data. We described an extension of the algorithm that learns only from solution traces. Furthermore, we also studied if implicit conditions and active learning could help to deal with larger domains. Our results suggest that implicit conditions should always be used, while active learning should be used in most of the cases. Additionally, we noticed the need to correctly choose the m-parameter used in rule learning. Having low values will produce goal-oriented trees with many subgoals with high probabilities of achieving them, while having high values of m-parameter will product trees with less goals, however with lower probabilities. We found similar results when testing with the *searchdepth* parameter. Therefore, m-parameter and *searchdepth* should be tailored to suit the capabilities of an actual problem solver.

In the last part we experimented with the Prolog programming domain. Due to the large number of options a programmer has at each step it is virtually impossible to generate the complete state-space. In this paper, we proposed to use a recently developed method that creates a state-space from student solutions traces and showed some promising results of applying our method to a such domain. We showed that the concepts automatically discovered by our algorithm do correspond to patterns that teachers are observing during the course.

Each subgoal appearing within the goal-oriented trees is a conjunction of attribute-values. These attributes and their corresponding values are defined by the knowledge engineer. In our experiments, we used relatively simple attributes that were a) easy to implement and b) easy to interpret, which often resulted in learned goals that do not correspond to the same concepts that humans use in solving these problems. Nevertheless, the emphasis of this paper is on conveying the idea of goal-oriented learning rather than on feature construction. A possible technique to alleviate this problem would be argument-based machine learning (Možina et al., 2007) that facilitates feature construction through a dialog involving a computer and a domain expert.

Another potential weakness of the current algorithm is the requirement to chose the maximal search depth to achieve a goal. In problem solving, different phases of solving vary by difficulty. Therefore, it would reasonable to use smaller search depths to solve easier parts and larger search depths for the more difficult. We could use iteratively increasing search depth as an alternative to deal with this problem. Furthermore, as a part of the future work, we also plan to extend our algorithm to domains that involve two-players, such as the game of chess, by using an adversarial-search mechanism, e.g. and-or search.

Anderson, J. R., 1993. Rules of the Mind. Lawrence Erlbaum Associates.

Bain, M., Srinivasan, A., 1995. Inductive logic programming with large-scale unstructured data. In: MACHINE INTELLIGENCE. Oxford University Press, pp. 233–267.

Bratko, I., 2012. Prolog Programming for Artificial Intelligence. Fourth edition. Pearson Addison-Wesley, Harlow, England.

Clark, P., Boswell, R., 1991. Rule induction with cn2: Some recent improvements. In: Kodratoff, Y. (Ed.), Machine Learning - EWSL-91. Vol. 482 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 151–163.

Clark, P., Niblett, T., 1989. The CN2 induction algorithm. Machine Learning Journal 4 (3), 261–283.

Corbett, A., Anderson, J., 1995. Knowledge tracing: Modeling the acquisition of procedural knowledge. User Modeling and User-Adapted Interaction 4 (4), 253–278.

Džeroski, S., Cestnik, B., Petrovski, I., 1993. Using the m-estimate in rule induction. Journal of Computing and Information Technology 1 (1), 37–46.

Fürnkranz, J., 2000. Machine learning in games: A survey. In: MACHINES THAT LEARN TO PLAY GAMES, CHAPTER 2. Nova Science Publishers, pp. 11–59.

Janssen, F., Fürnkranz, J., 2010. On the quest for optimal rule learning heuristics. Machine Learning 78 (3), 343–379.

Košmerlj, A., Bratko, I., Žabkar, J., 2011. Embodied concept discovery through qualitative action models. International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 19 (3), 453–475.

Kutluhan, E., 1996. Hierarchical task network planning: Formalization, analysis, and implementation. Ph.D. thesis, University of MaryLand, College Park.

Langley, P., Simon, H. A., 1995. Applications of machine learning and rule induction. Communications of the ACM 38 (11), 54–64.

Lazar, T., Bratko, I., 2014. Data-driven program synthesis for hint generation in programming tutors. In: Intelligent Tutoring Systems - 12th International Conference. pp. 306–311.

Michie, D., 1976. An advice-taking system for computer chess. Computer Bulletin 2 (10), 12–14.

Mitchell, T., 1997. Machine Learning. McGraw Hill.

Mostafavi, B., Barnes, T., 2010. Towards the creation of a data-driven programming tutor. In: Aleven, V., Kay, J., Mostow, J. (Eds.), Intelligent Tutoring Systems. Springer Berlin Heidelberg, pp. 239–241.

Možina, M., Guid, M., Sadikov, A., Groznik, V., Bratko, I., 2012. Goal-oriented conceptualization of procedural knowledge. In: Springer (Ed.), Intelligent Tutoring Systems. Vol. 7315. Chania, Greece, pp. 286–291.

Možina, M., Žabkar, J., Bratko, I., 2007. Argument based machine learning. Artificial Intelligence 171 (10/15), 922–937.

Murray, T., 1999. Authoring intelligent tutoring systems: An analysis of the state of the art. International Journal of Artificial Intelligence in Education (IJAIED) 10, 98–129.

Sadikov, A., Bratko, I., 2006. Learning long-term chess strategies from databases. Machine Learning 63 (3), 329–340.
URL `http://dx.doi.org/10.1007/s10994-006-6747-7`

Tadepalli, P., 2008. Learning to solve problems from exercises. Computational Intelligence 24 (4), 257–291.

VanLehn, K., Lynch, C., Schulze, K., Shapiro, J. A., Shelby, R., Taylor, L., Treacy, D., Weinstein, A., Wintersgill, M., 2005. The andes physics tutoring system: Lessons learned. International Journal of Artificial Intelligence in Education 15 (3), 147–204.

Šuc, D., 2003. Machine Reconstruction of Human Control Strategies. IOS Press.

Wang, X., 1994. Learning planning operators by observation and practice. In: Proceedings of the 2nd International Conference on AI Planning Systems. pp. 335–340.

Woolf, B. P., 2008. Building Intelligent Interactive Tutors: Student-centered strategies for revolutionizing e-learning. Elsevier & Morgan Kaufmann, Burlington, MA.