Topic:

A Study on the Applicability of Explanation-Based Learning for Identifying Functional Structures in a Physical Simulation Game

Bachelor Thesis

in Applied Computerscience, Otto-Friedrich-University Bamberg

Chair:	Smart Environments
Head of Chair:	Professor Dr. Diedrich Wolter
Adviser:	Professor Dr. Diedrich Wolter
Author:	Martin, Jan
Matriculation Number:	1796943
Address:	An der Weberei 7, 96047 Bamberg
E-Mail:	jan.martin@uni-bamberg.de
Field of Study:	Applied Computer Science, Bachelors Degree
Date of Submission:	22/10/2018

Abstract

One challenge of learning systems is how to learn a general concept from a specific example. Explanation-Based methods offer a way to utilize previous knowledge of the domain under study to generalize a concept from just one training example. Their ability to do so is based on their ability to *explain why* the training example is a member of the concept meant to be learned. However, Explanation-Based methods have not seen widespread use in recent years. This thesis explores the applicability of Explanation-Based Learning, implemented in a logic programming language, to learning concepts in a simply physical simulation, exemplified by the video game 'Angry Birds'.

Contents

List of Abbreviations

EBL	Explanation-Based Learning
EBG	Explanation-Based Generalization

List of Figures

List of Tables

1 Introduction

A learning system needs to be able to generate abstract knowledge from information it receives, allowing it to reference that generated knowledge to understand future situations better. As it is to be assumed that such a system will encounter situations similar, but not exactly alike those it previously learned from, abstraction and generalization are important factors in creating a learning system. By abstracting and comparing received information based on previously known or predefined concepts, a learning system can identify similar features in new examples and use them to generate a new generalized concept definition for the object or system under study. At its core, a concept definition is a description of what constitutes a concept and what is not part of it. Out of the pool of all potential definitions and relations, it narrows a concept down to just one specific set of rules. The key challenge of generalization is to limit the search area. Explanation-Based Learning, as described by Mitchell et. al. [1] offers an approach that uses existing 'domain knowledge' to generate a concept definition from just one example. This generalization method doesn't rely on inductive bias, instead limiting the search space through the domain knowledge. Unlike similarity-based approaches [2] , Explanation-Based Learning is capable of explaining why a new example is part of an existing domain theory through a proof tree.

Example:

Let's assume there exists a domain theory describing the rules of chess. If receiving an example of a valid move of a chess piece over the board — our goal concept in this example —, an Explanation-Based Learning approach will not just check if this move is indeed valid, but will provide a justification for why this move is valid.

[1]Explanation Based Generalization: A Unifying View(Mitchell, Keller & Kedar-Cabelli, 1988). The term *Explanation-Based Generalization, or EBG* is used here specifically to denote the domain-independent mechanism proposed by Mitchell et. al.

[2]Mitchell et. al. use the term *similarity-based*, originally suggested by Lebowitz (Lebowitz, 1984), to group generalization methods based on finding similarities among positive examples and differences between positive and negative examples, differentiating them from other approaches like Explanation-Based Learning.

1.1 Thesis Goal

This thesis seeks to answer the question if Explanation-Based Learning offers sufficient means to analyse physical structures and relations in a simple physical simulation, exemplified by the videogame 'Angry Birds'. The goal is to implement a modern adaptation of Explanation-Based Learning in a logic programming language and explore the applicability of the created Algorithm to the detection of interesting features in levels of 'Angry Birds'. The algorithm is implemented in SWI-Prolog and the examples it is meant to work with are generated by a vision module and parser used by the 'BamBirds' team participating in the 'AIBirds' challenge, whose goal is the create AIs that can play 'Angry Birds' as well as a human player. The created Algorithm, given existing domain knowledge and a provided example that we know is part of the given domain, is expected to generate a proof tree showing why that example is part of the given domain. The Algorithm should work with different scenarios consisting of individually defined domain theories and examples, but also work with different examples that are part of the same domain theory. It is, however, not expected to generate proof for multiple examples at once.

1.1.1 Further Contents:

Section 2 illustrates the inner workings, strengths and shortcomings of EBG with an example similar to those used by Mitchell et. al.. Section 3 shows the implementation of the EBG Algorithm in SWI-Prolog, using the previous example as a base. Section 4 explains the level analysis process for 'Angry Birds' Levels and shows the results of running multiple levels through the EBG Algorithm. Section 5 concludes the thesis with an evaluation of results.

2 Explanation-Based Learning

Learning a new concept means learning from examples. As it is usually impractical to feed a machine all possible examples of a concept, doing so requires the capability to abstract and generalize the received information, so as to form a concept definition — which will hopefully cover all possible examples of this concept — from a limited number of examples of that concept. At its heart, generalization involves searching the available examples of what is expected to be a single concept for similarities that are thought to be relevant, then basing the concept on those similarities. What is 'relevant', however, isn't always known, meaning a search can't be easily limited to those features and leading to a large search space. "Because this space of possible concept definitions is vast, the heart of the generalization problem lies in utilizing whatever training data, assumptions and knowledge are available to constrain this search." (Mitchell et al., 1988, p. 1) Mitchell et. al. group the approaches to solve this problem into '*similarity-based*' generalization methods, which rely on inductive bias to constrain the search space, and '*explanation-based*' methods, which use previously available knowledge of the domain and concept, and require just one example to produce both a generalization of the example and a justification for this generalization.

2.1 The Explanation-Based Generalization Method

Explanation-Based Generalization (EBG) generates a proof for a single example based on the domain theory provided. It was intended as a try to unify several approaches to Explanation-Based Learning. The original Algorithm proposed by Mitchel et. al. only confirms a relation if it is asked. Given multiple examples, it can't look for a suitable concept definition by itself, and is thus depending on being asked the correct question. The strength, and purpose, of EBG is to explain why an example provided fits into an existing concept definition, in the same way that a human knowing the rules of chess might be able to explain the eligibility of moving a single chess piece in a specific way based on the current game situation. The aim of this explanation is to generate specific rules applicable to the example from the generalized rules available in the domain knowledge.

The EBG Algorithm takes an existing set of object data and a domain theory, of which it is assumed that the set of object data is a positive example of the domain theory, and

Table 2.1: Explanation-Based Generalization, as described by Mitchell et. al.

Given:
• Goal Concept: A concept definition describing the concept to be learned. (It is assumed that this concept definition fails to satisfy the Operationality Criterion.) • Training Example: An example of the goal concept. Domain Theory: A set of rules and facts to be used in explaining how the training example is an example of the goal concept. • Operationality Criterion: A predicate over concept definitions, specifying the form in which the learned concept definition must be expressed. Determine: • A generalization of the training example that is a sufficient concept definition for the goal concept and that satisfies the operationality criterion.

tries to prove, through back-propagation, why the dataset in the example is a positive example of the theory. This is illustrated here with the 'Cup'-Example(**Fig.**2.1), as seen originally in previous works(Winston, Binford, Katz & & Lowry, 1983), which is used in a simplified form. In this example, all a cup needs be be a cup is to be liftable and to be able to hold liquid. These rules make up the first 'layer' or step of regression. The next lower layer then consists of the requirements for the previous rules and define what is necessary for those to be true. In this example, to be liftable, a cup needs to have a handle and be light. To hold liquid, it requires a concave surface pointing up. EBG will keep regressing a provided example through this explanation structure until all statements in the example can be matched to the same statement from the domain theory. By itself, the Algorithm doesn't actually allow the machine to learn a concept: As noted by Mitchell et.al.[1], the Algorithm assumes a perfect domain theory, meaning the rules are already known. In the Cup-Example, if it wasn't defined what is required for a cup to hold liquid, the Algorithm wouldn't be able to progress . The output of EBG is instead meant to explain to an outside observer the reasoning of why a specific example is part of this perfect domain theory. This observer doesn't need to be a human user, but could also be a compiler using the generated proof to create a specialized program that is more efficient at its task, as it would not need to know the abstracted general rules.

[1]Explanation Based Generalization: A Unifying View(Mitchell et al., 1988, p. 22)

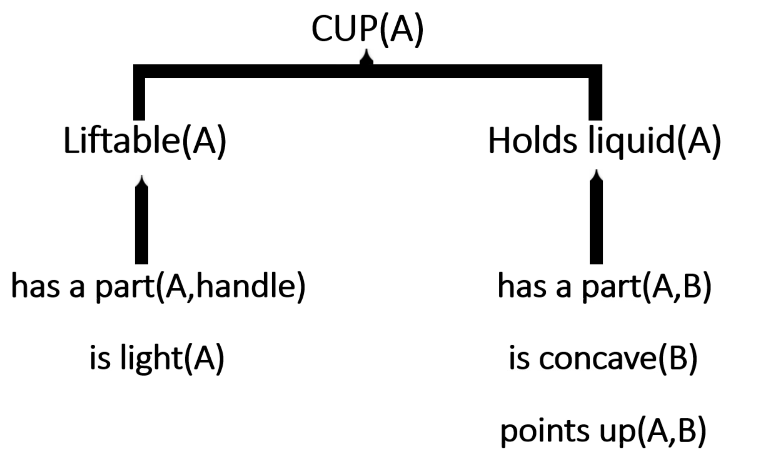

Figure 2.1: Explanation of Cup-Example

3 Recreating the EBG Algorithm in SWI-Prolog

An implemented EBG algorithm is expected to take a domain theory of a concept and a valid example of that same concept, and return a visualization of the proof that the provided example is part of that concept. This is reached by regression through the explanation tree until such a point where the example and domain theory would be equivalent if the objects in the example are instances of the abstract variables used in the domain theory. Prolog was chosen as the language to implement EBG with as logical languages are practical for relations of predicates The Implementation of EBG in this thesis is based on the EBG Meta-Interpreter by Ivan Bratko [1].

The code was adapted to include the layer of a relevant predicate, allowing a proof-tree to be generated from the output. XPCE, as described by Wielemaker et.al.(Wielemaker & Anjewierden, 2002), is used for the visualization of the results.

3.1 Interpreter Prolog Implementation:

To facilitate readability and illustrate the working of the algorithm to the user, the EBG algorithm writes the currently checked predicates to the console. The code lines responsible for these writes have been omitted here for the sake of readability. The original implementation by Bratko used only 3 variable inputs, usually the the goal, generalized goal, and condition. This adaptation requires three more fields, providing in input for the list of predicates generated so far, a possible output for the new list generated after further calculation, and the current layer, or regression step, which will also be added to the list ahead of the next predicate and is required to generate a graphic tree later.

```
ebg( true, true, true, _, _, _)  :-  !.

ebg( Goal, GeneralizedGoal, GeneralizedGoal, Tree,Tree2,Layer)  :-
    isOp( GeneralizedGoal),
    copy_term(Goal,Goal1),
    append(Tree,[], Tree2),
  callN( Goal).
```

[1]META-INTERPRETERS AND META-PROGRAMMING, p. 21-24, found at: https://ailab.si/ivan/datoteke/dokumenti/30/83_79_Meta_programming.ppt , last visited: 02.10.2018

The main inputs of EBG are the goal, a generalizated goal, and a condition which needs to be met for the two to be equal, which is the case when the regression reaches a variable defined as 'operational'. If the EBG Algorithm receives an input consisting of just three true facts, it will simply return whatever variables it was called with. Should it reach a state where the condition *is* the generalized goal, it has reached an operational variable. It will then call the goal to check if this matches the example. If it does, it will return the old list as the new list, and if it does not, it will return false, marking the current list as a dead end. If a predicate from an upper layer had multiple potential conditions, the algorithm will then regress down these other paths.

```
ebg( (Goal1,Goal2), (Gen1,Gen2), Cond,Tree,Tree2,Layer)  :- !,
  add(Layer,1,NewLayer),
  append(H,[T1,T2],Tree),
  append(H,[Layer],TreeL1),
  append(TreeL1,[Gen1],TreeA),
  ebg( Goal1, Gen1, Cond1, TreeA,Tree1,Layer),
  append(Tree1,[Layer],TreeL2),
  append(TreeL2,[Gen2],TreeB),
  ebg( Goal2, Gen2, Cond2, TreeB,Tree2,Layer),
  and( Cond1, Cond2, Cond).
```

Should the EBG Algorithm receive multiple predicates when regressing through the explanation tree, it pick the first 'Branch' and recursively call itself on that branch, then repeat the same step with the other side.

```
ebg( Goal, GenGoal, Cond, Tree,Tree2,Layer)  :-
  \+ isOp( Goal),
  clause( GenGoal, GenBody),
  OldLayer is Layer-1,
  add(Layer,1,NewLayer),
  append(Tree,[NewLayer],TreeL1),
  append(TreeL1,[GenBody],Tree1),
  copy_term( (GenGoal,GenBody), (Goal,Body)), % Fresh copy of(GenGoal,GenBody)
  ebg( Body, GenBody, Cond, Tree1,Tree2,NewLayer).
```

When EBG receives exactly one goal, one generalized goal, and a condition that does not equal the generalized goal, it will add the current layer and the first predicate of the generalized goal to the tree. Following this, it will generate a fresh copy of the remaineder of goal and generalized goal, with which to progress downwards, and then recursively call itself with the next regression of both the goal and generalized goal. If there are multiple of those, it would instead fall to the previously described function to iterate through the different branches until such a time where again an iteration only receives a single goal and generalized goal.

Further, 'Helper Functions' are used for addition and conjunction as well as iterating the layers.

Helper Functions:

```
callN(\+ A):- call(\+ A).
callN(A):- call(A).

add(A,B,C) :-
    C is A+B.

sub(A,B,C) :-
    C is A-B.

 and( A, B, C):
and( true, Cond, Cond)  :-  !.        % (true and Cond) <==> Cond
and( Cond, true, Cond)  :-  !.        % (Cond and true) <==> Cond
and( Cond1, Cond2, ( Cond1, Cond2)).

splitList([],[],[]).
splitList([H,H1|T],[H|T1],[H1s|T2]) :-
    term_string(H1, H1s),
    splitList(T,T1,T2).
```

The main relevant one here is 'splitList', which is used to split the generated 'Tree'

List into two lists, one containing the 'Layer' numbers, the other the actual predicates.

3.1.1 Main Function

The Proof is started by the following 'runEBG', which runs the EBG Algorithm and forwards the two generated lists to the 'treePrinter' function, which generates a graphic output from the results.

```
runEBG(Goal,Gen,Cond,Tree) :-
    nl,
    ebg(Goal,Gen,Cond,[0,Gen],Tree,0),
    ((\+ Cond == true) ->
    write('Is part of Domain because: '),
    nl,print_term(Cond,[]),nl
    ,splitList(Tree,Order,Predicates)
    ,treePrinter(Order,Predicates,0,vertical)
    ;!).
```

EBG usually generates two outputs:

1. The affirmation that the example is indeed a valid example of the concept
2. A list of conditions for the example to be part of the concept

As EBG, as implemented, only outputs the latter with positive examples, the function checks for the value of condition first. If it is the list of conditions, which is essentially a log of the algorithm traversing the explanation tree, it will return it. The Tree list generated in the process of reaching that proof is then split into one list containing the Layer information and one list with the found predicates.

3.1.2 The Cup Example in Prolog

The cup definition serves as our domain theory, describing what a cup is. The definition of operational variables is necessary to generate a usable output, as the algorithm stops iterating through a branch of the proof tree upon stumbling on an operational variable. Without this concept, a valid input could just be accepted as sufficient proof and returned immediately. As such, operational variables are usually at the very bottom of a downwards-growing proof tree. The EBG Algorithm takes an example, in this example 'cup(obj1)', and a generalized concept, in this example 'cup(A)', and returns the reason why 'obj1' can be an instance of 'A'. Valid parts of the proof are all variables that satisfy our operationality criterion and apply to both the example object and the domain theory.

A cup is defined as an entity that is lift-able and can hold liquid. This is expressed in the code as follows:

Domain Theory:

```
cup(X) :- (liftable(X), holds_liquid(X)).
holds_liquid(Z) :-
    part(Z, W), concave(W), points_up(W).
liftable(Y) :-
    light(Y), part(Y, handle).
light(A):- small(A).
light(A):- made_of(A, feathers).
```

Example:

```
cup(obj1).
small(obj1).
owns(bob, obj1).
part(obj1, handle).
part(obj1, bottom).
part(obj1, bowl).
```

```
points_up(bowl).
concave(bowl).
color(obj1, red).
made_of(obj2, feathers).
```

Operationality Criteria:

```
isOp(small(_)).
isOp(part(_, _)).
isOp(owns(_, _)).
isOp(points_up(_)).
isOp(concave(_)).
```

3.1.3 Console Output

This is the unedited console output of running the cup example, including writes illustrating the progress:

```
cup(_1906) <<
liftable(_1906) && holds_liquid(_1906) :
liftable(_1906) <<
light(_1906) && part(_1906,handle) :
light(_1906) <<
Goal: small(_1906) <- isOp.
small(obj1).
Goal: part(_1906,handle) <- isOp.
part(obj1,handle).
holds_liquid(_1906) <<
part(_1906,_3228) && concave(_3228),points_up(_3228) :
Goal: part(_1906,_3228) <- isOp.
part(obj1,_3464).
concave(_3228) && points_up(_3228) :
concave(_3228) && points_up(_3228) :
```

```
concave(_3228) && points_up(_3228) :
Goal: concave(_3228) <- isOp.
concave(bowl).
Goal: points_up(_3228) <- isOp.
points_up(bowl).
Is part of Domain because:
(small(A) ',' part(A,handle)) ',' part(A,B) ',' concave(B) ',' points_up(B)
Starting Tree with the following parameters:
[0,1,2,3,2,1,2,2,2][cup(_6),liftable(_6),light(_6),small(_6),part(_6,handle),holds_liqui
light(_6) <<
made_of(obj2,feathers) <<
cup(obj1) <<
A = obj1,
Condition = true.
```

The output lists the reason why the example is part of the domain, shows the data forwarded to the visual output, and concludes that 'obj1' and 'A' are equivalent.

3.2 Visual Output

The resulting proof, in form of a list, is then used to generate a proof tree with XPCE.

The treePrinter function takes two lists: The Layers of the predicates found, and the actual predicates. Both lists are ordered after when that specific predicate was found. The remaining two inputs are the starting Layer and the direction we want the generated tree to face. In this case, the Layer is always a 0 as we want the entire proof tree.

```
treePrinter([OrH|OrT],[PrH|PrT],Layer,Direction):-
        sformat(A, 'Showing Proof Tree: ~w', [Direction]),
        new(D, window(A)),
        send(D, size, size(640,300)),  % Create Output Window
```

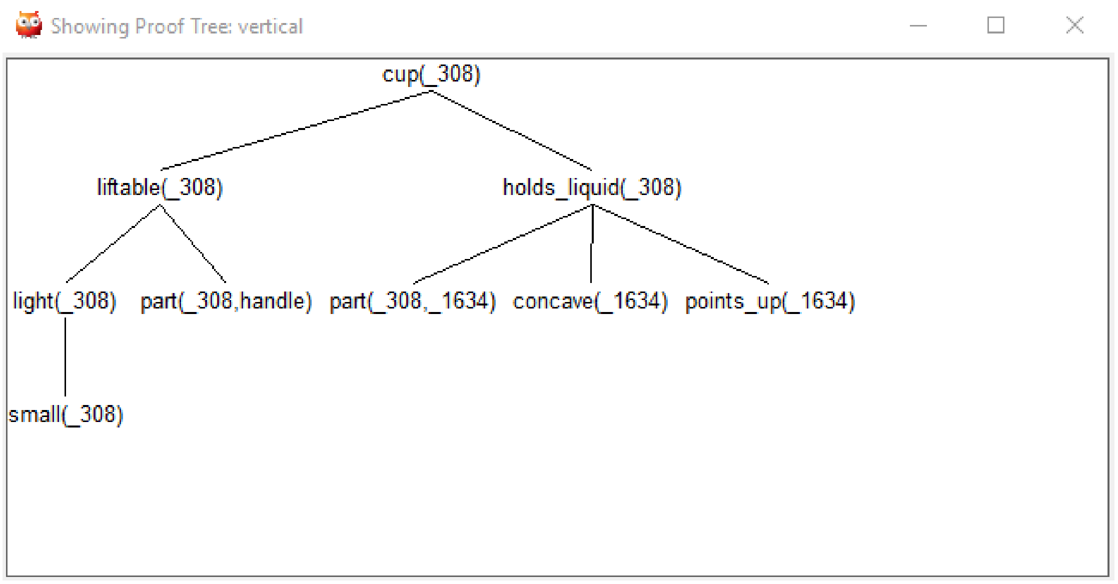

Figure 3.1: Prooftree visualized by XPCE

```
        new(Root, tree(text(PrH))),
        send(Root, neighbour_gap, 10),
        add(Layer,1,NewLayer),
        branchPrinter(OrT,PrT,NewLayer,Root,[Root],1),

        send(Root, direction, Direction),
        send(D, display, Root),
        send(D, open).
```

The function makes the first predicate from the predicate list the root of the new tree, then starts the 'branchPrinter' function one level lower. It also defines the orientation of the tree and the size and name of window. The branchPrinter function is similar to the treePrinter function it was called from, but lacks the capability to initialize the turn. Instead, it is recursive.

```
branchPrinter([],[],_,_,_,Iteration).
branchPrinter([OrH|OrT],[PrH|PrT],Layer,Root,Parents,Iteration):-
    %Call helper functions:
```

```
nl,
add(Layer,1,DeeperLayer),
% Add one Layer in case it's needed. It never goes deeper faster
add(Iteration,1,Iteration2),
append(Grandparents,[Parent],Parents),
% Parent is the last Element of the Parents List

(OrH == 1 ->
new(Node, node(text(PrH))),
send_list(Root, son,[Node]),
branchPrinter(OrT,PrT,1,Root,[Root,Node],Iteration2);
```

If the next predicates layer is 1, start at the root again.

```
OrH <  Layer ->
```

If the next predicates layer is higher than the previous, go up one layer and try again.

```
sub(Layer,1,UpperLayer),
branchPrinter([OrH|OrT],[PrH|PrT],UpperLayer,Root,Grandparents,Iteration2);
% Recursive call

OrH == Layer ->
new(Node, node(text(PrH))),
append(_,[Grandparent],Grandparents),
send_list(Grandparent, son,[Node]),
branchPrinter(OrT,PrT,Layer,Root,Parents,Iteration2);
```

If the layer of the next predicate's layer is equal the layer of the last one, attach it to the previous parent.

```
    OrH == DeeperLayer ->
    new(Node, node(text(PrH))),
    send_list(Parent, son,[Node]),
    append(Parents,[Node],NewParent),
    branchPrinter(OrT,PrT,DeeperLayer,Root,NewParent,Iteration2);!)
    .
```

If the next predicate is a layer below the previous one, attach it as a leaf to the last predicate.

Iteration through the Layers The branch Printer is essentially a single, large 'If-Else' function. It operates a list of Parents, from root down to the current Layer. When the Layer Number of the next predicate is higher than the previous number, which indicates the predicate is a regression step further down, the current predicate will be added as a sub-Branch — or leaf, if it is an end-point — to the current Branch. If it has the same Layer as the previous predicate, it

3.3 Safe to Stack

The original "safe to Stack" example, as used by Mitchell et.al. after Borgidal et. al. (Borgida, Mitchell & Williamson, 1986), can also be tested.

```
notSafeToStack(X,Y) :- \+ safeToStack(X,Y).
isDurable(OBJECT) :- \+ isFragile(OBJECT).

hasWeight(OBJECT,WEIGHT) :- hasVolume(OBJECT,VOLUME), hasDensity(OBJECT,DENSITY),
                            %WEIGHT = OBJECT*DENSITY,!.
                            mult(VOLUME,DENSITY,WEIGHT).
hasWeight(OBJECT,5) :- isA(OBJECT,endtable).
add(X,Y,Z) :- Z is X+Y.
mult(X,Y,Z) :- Z is X*Y.
```

```
isLess(A,B) :- A < B.
isLighter(OBJECT1,OBJECT2) :-
%    (   OBJECT1 \= OBJECT2),
    hasWeight(OBJECT1,WEIGHT1),
    hasWeight(OBJECT2,WEIGHT2),
    isLess(WEIGHT1,WEIGHT2)
    .

safeToStack(OBJECT1,OBJECT2) :- isLighter(OBJECT1,OBJECT2).
safeToStack(OBJECT1,OBJECT2) :- isDurable(OBJECT2).
```

Here, the requirement that the lower box should not be fragile has been replaced with a positive version: The lower box should be 'durable'. The example is defined as follows:

```
isOn(obj1,obj2).
isA(obj1,box).
isA(obj2,endtable).
hasWeight(obj2,5).
hasColour(obj1,red).
hasColour(obj2,blue).
hasVolume(obj1,1.2).
hasDensity(obj1, 0.1).
isFragile(obj2).
```

The EBG algorithm correctly returns a proof for the stackability of 'obj1' on 'obj2', as seen in **Fig.** 3.2.

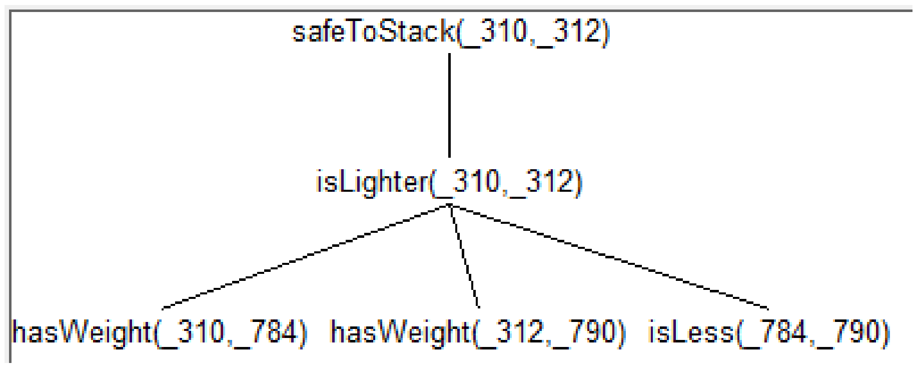

Figure 3.2: The result of running EBG on Safe to Stack

4 Adapting the Algorithm for Angry Birds

'Angry Birds' is a 2-dimensional video game where a player shoots different kinds of projectiles(the birds) at structures comprised of mostly rectangular objects (**Fig.** 4.1). Those objects have different sizes and materials that affect their weight and durability. The goal for the player is to remove all 'pig' objects from the level by application of blunt-force trauma, either by directly hitting them, or by destabilizing structures of objects in such a way that pigs either fall a sufficient height to get destroyed on impact, or will be hit by other objects falling on them. The game uses a rudimentary physics simulation where a projectiles speed and mass affect what effect its impact on a level object will have. The position of the hit-able objects in the level is detected by a vision module used by teams of the AIBirds challenge, a yearly competition where teams from universities aim to create an artificial intelligence that can play 'Angry Birds' as well as, or better than, a human player. The level files used for the example in this paper was generated using the 2018 version of the Bambirds Agent, developed in the university of Bamberg. The vision module detects and identifies all visible objects in a level based on their colour and shape, as seen in Figure4.2. The gathered information is then written into The relevant info from an actual Angry Birds level:

```
hasMaterial(ice11,ice,550,333,5,26).
hasMaterial(ice6,ice,532,333,6,26).
hasMaterial(stone9,stone,539,237,12,13).
hasMaterial(wood0,wood,488,353,6,13).
hasMaterial(wood1,wood,517,314,6,53).
hasMaterial(wood10,wood,542,224,6,13).
hasMaterial(wood12,wood,554,256,6,53).
hasMaterial(wood13,wood,563,314,6,53).
hasMaterial(wood14,wood,592,353,6,13).
hasMaterial(wood2,wood,517,308,53,7).
hasMaterial(wood3,wood,529,256,6,53).
hasMaterial(wood4,wood,530,359,27,6).
hasMaterial(wood5,wood,531,327,26,6).
hasMaterial(wood7,wood,532,250,26,7).
hasMaterial(wood8,wood,538,301,12,6).
pig(pig0,538,289,9,9).
```

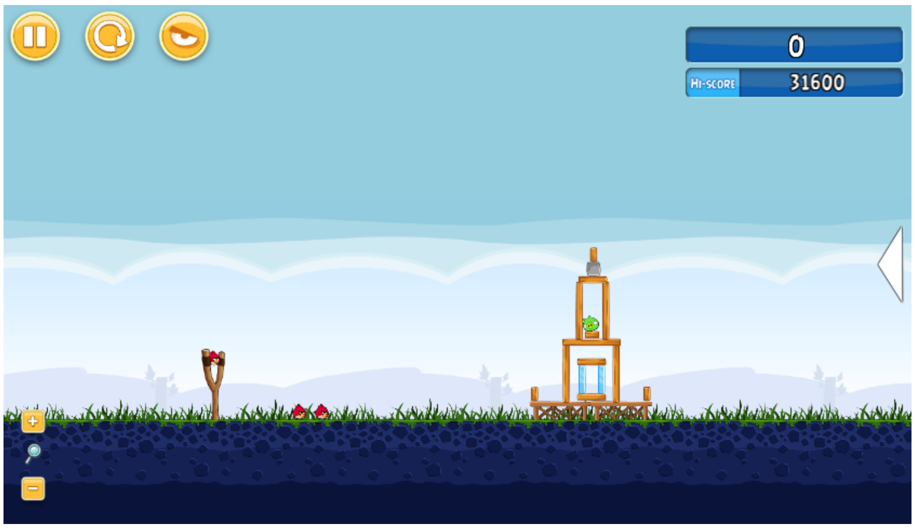

Figure 4.1: A typical Angry Birds level

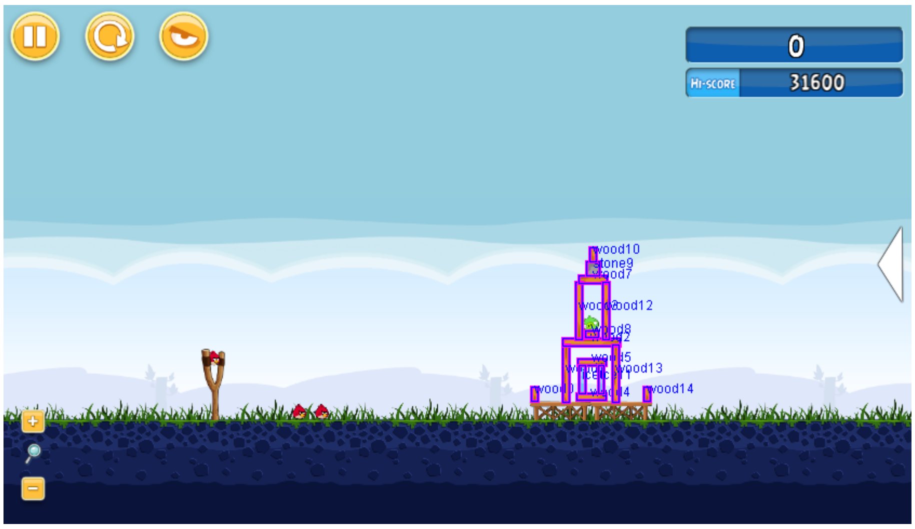

Figure 4.2: Insight into the object detection of the vision module

'hasMaterial' contains the ID, material, and position in the form of X/Y coordinates and the height and width. The Y coordinates show the pixels from the top of the screen, as the vision module begins counting in the top left corner. The level file contains much more derived information pertaining to the objects positioning in relation to each other and the catapult the player can shoot birds from, but this information isn't relevant for EBG. Helper functions with a 2-pixel uncertainty buffer have been added to compensate for possible inaccuracies in the vision module's output:

```
rLess(Var1a,Var1b,Var2a,Var2b) :-
    (Var1a+Var1b) < (Var2a+Var2b+2).
rMore(Var1a,Var1b,Var2a,Var2b) :-
    (Var1a+Var1b+2) > (Var2a+Var2b).
```

These, together with 'hasMaterial/6' and its priority target equivalent, 'pig/5', are used as operational variables. Our aim is to create a domain theory that allows the EBG algorithm to detect a 'Domino' situation, where a chain of objects can be toppled over to eventually hit a priority object. In our examples, all priority objects are 'pig' objects. The eligible objects must be higher than they are wide, and have to be toppled over by hitting the leftmost object with a projectile, causing it to fall into and push over the object to it's right, and so on, until the last object of the chain will fall onto a priority object, crushing it. The following requirements have been put into code for eligible domino objects:

1. The object is at least 20% higher than it is wide

   ```
   isHigherThanWide(OBJECT):-
       hasMaterial(OBJECT,_,_,_,WIDTH,HEIGHT),
       rMore(HEIGHT,-2,WIDTH,(HEIGHT*0.2)).
   ```

2. The object is standing to the left of an eligible target and and either above or on equal height with the target.

   ```
   isChainMember(OBJECT) :-

       isLeftOf(OBJECT,FallTarget),
   ```

```
    isHigherOrEqual(OBJECT,FallTarget),
    hasMaterial(OBJECT,_,XPOS,_,WIDTH,HEIGHT),
    hasMaterial(FallTarget,_,FallXPOS,_,_,_),
    rLess(FallXPOS,0,(XPOS+WIDTH),HEIGHT*0.95),
    isEligibleFallTarget(FallTarget).

isEligibleFallTarget(OBJECT) :- pig(OBJECT,_,_,_,_).
isEligibleFallTarget(OBJECT) :- isChainMember(OBJECT).
```

Eligible targets are either pigs or other eligible domino objects.

```
isHigherOrEqual(OBJECT1,OBJECT2) :-
    hasMaterial(OBJECT1,_,XPOS1,YPOS1,WIDTH1,HEIGHT1),
    hasMaterial(OBJECT2,_,XPOS2,YPOS2,_,HEIGHT2),
    rLess(YPOS1,0,YPOS2,HEIGHT2),
    rMore(HEIGHT1,XPOS1,XPOS2,HEIGHT1*0.1).
```

Close enough, for the purpose of this example, is defined as object 1 being 10% higher than the distance between object 1 and 2.

3. The object does not carry more than twice its own mass in extra weight

```
isEligibleDomino(OBJECT) :-
    notWeightedDown(OBJECT),
    hasForm(OBJECT,bar).

notWeightedDown(OBJECT) :-
    (\+ isOn(OBJECT2,OBJECT)),!.
notWeightedDown(OBJECT) :-
    hasMaterial(OBJECT,pig,_,_,_,_).
notWeightedDown(OBJECT) :-
    agregateWeight(OBJECT,AgWeight),
    hasWeight(OBJECT,Weight),
    rLess(AgWeight,0,Weight*2,Weight).
```

Pigs receive special attention here as they can not hold other objects reliably. In the name of full disclosure, the check for 'agregateWeight', that is, the total weight of the object and all objects above it, has been set to automatically succeed in testing.

Using these requirements, EBG is capable of detecting domino structures, f.Ex. the vertical wood bars right of the first pig in **Fig.**4.3 or the wooden bar on the upper left in **Fig.**4.4.

In **Figure**4.5, the pillars are too far from each other, thus not constituting a domino situation:

```
hasMaterial(stone1,stone,507,329,6,24).
hasMaterial(stone3,stone,546,329,5,24).
hasMaterial(stone5,stone,586,329,6,24).
hasMaterial(stone7,stone,650,306,5,47).
hasMaterial(wood0,wood,498,323,24,6).
hasMaterial(wood2,wood,538,323,24,6).
hasMaterial(wood4,wood,577,323,24,6).
hasMaterial(wood6,wood,641,300,24,6).
pig(pig0,504,311,9,9).
pig(pig1,544,312,8,8).
pig(pig2,584,311,9,9).
pig(pig3,648,289,8,8).
```

If we move them place a pig behind one of the pillars, there will be a domino situation (**Fig.** 4.6. If we also move the pillars closer together, EBG can detect a domino chain as well, as seen in **Fig.** 4.7.

```
hasMaterial(stone1,stone,547,329,6,24).
hasMaterial(stone3,stone,566,329,5,24).
hasMaterial(stone5,stone,586,329,6,24).
hasMaterial(stone7,stone,650,306,5,47).
hasMaterial(wood0,wood,518,323,24,6).
hasMaterial(wood2,wood,558,323,24,6).
```

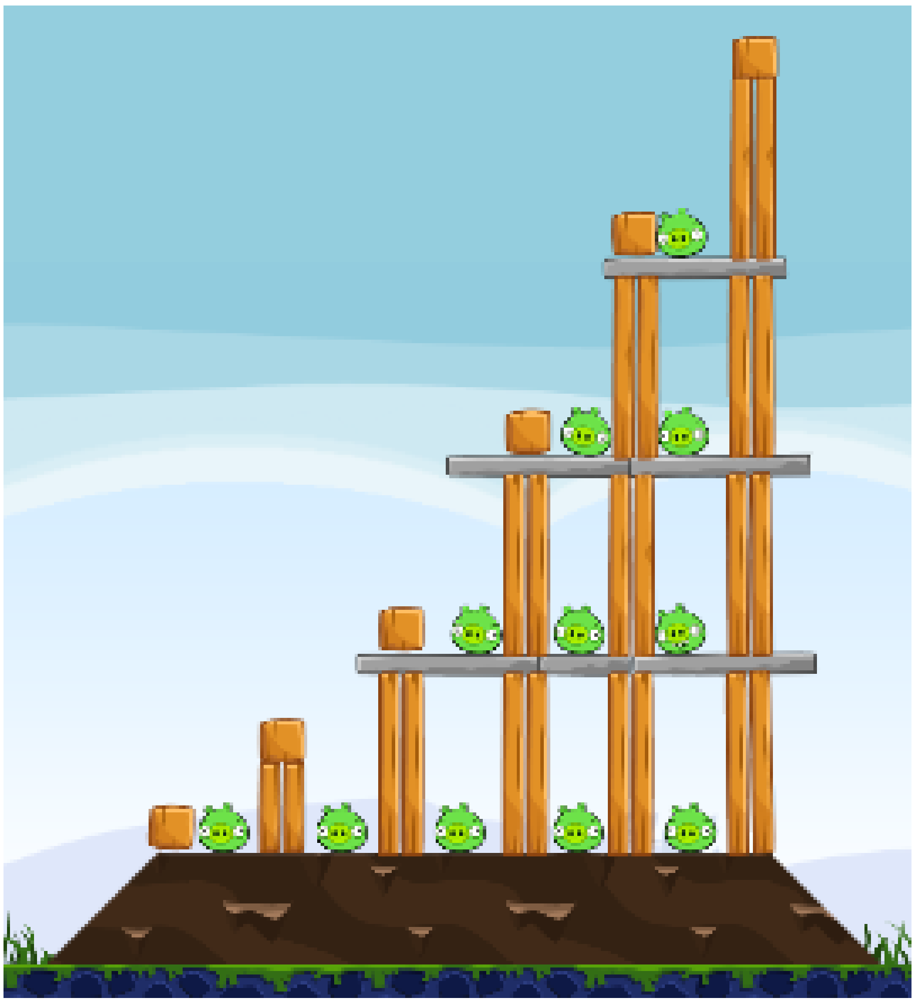

Figure 4.3: An Angry Birds Level where a Domino Situation is detected

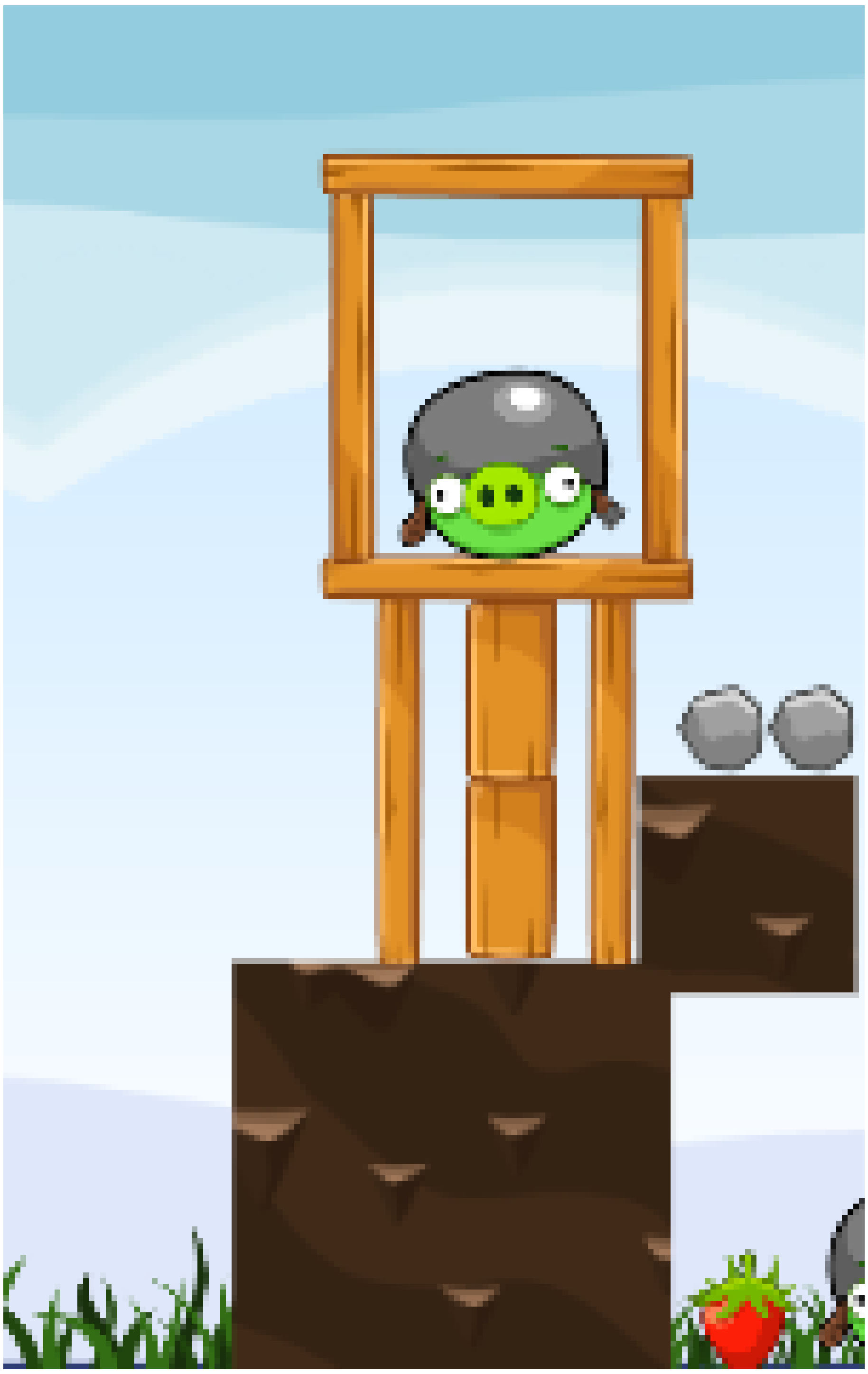

Figure 4.4: A second Angry Birds Level where a Domino Situation is detected

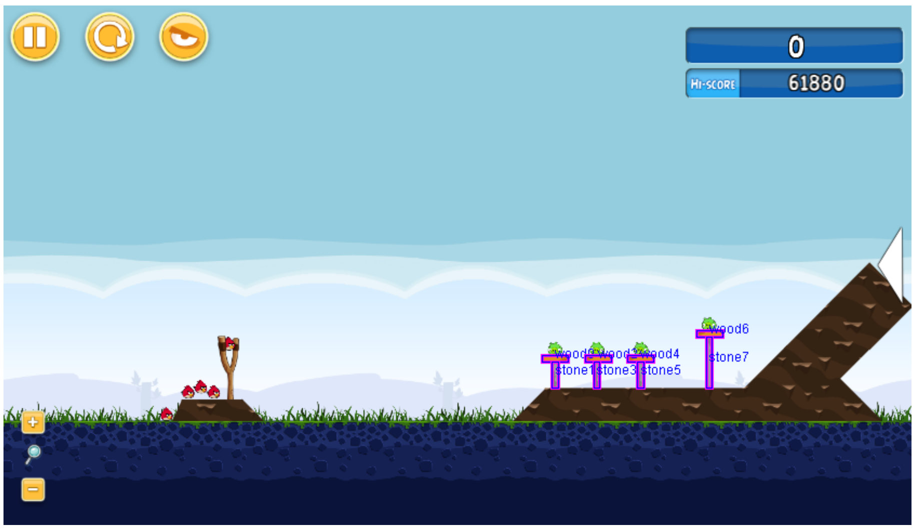

Figure 4.5: An Angry Birds Level where no Domino Situation is detected

```
hasMaterial(wood4,wood,577,323,24,6).
hasMaterial(wood6,wood,641,300,24,6).
pig(pig0,604,344,9,9).
```

4.1 Evaluation

It is quite possible for a Situation to look like a Domino Situation, while it actually isn't, according to the code definition. An example of this is unmodified second level of the Angry Birds campaign 'Poached Eggs', referenced above (**Fig.**4.5). A human player will still be able to hit the stone bars on top of the pillars to topple the structure, and thus topple multiple with a single hit. Whether this constitutes a problem with the definition of a Domino Structure or not is in the eye of the beholder. It is, however, very difficult to reliably predict the behavoir of objects on top of a toppled object. Furthermore, the system does not care for the actual reachability of the objects — a Domino Structure behind a solid wall is still a Domino Structure.

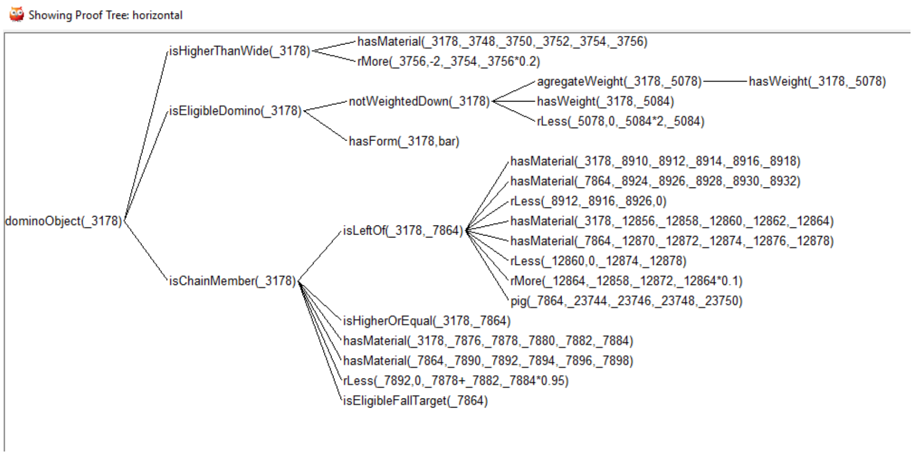
Showing Proof Tree: horizontal
dominoObject(_3178)
isHigherThanWide(_3178)
hasMaterial(_3178,_3748,_3750,_3752,_3754,_3756)
rMore(_3756,-2,_3754,_3756*0.2)
isEligibleDomino(_3178)
notWeightedDown(_3178)
agregateWeight(_3178,_5078)
hasWeight(_3178,_5078)
hasWeight(_3178,_5084)
rLess(_5078,0,_5084*2,_5084)
hasForm(_3178,bar)
isChainMember(_3178)
isLeftOf(_3178,_7864)
hasMaterial(_3178,_8910,_8912,_8914,_8916,_8918)
hasMaterial(_7864,_8924,_8926,_8928,_8930,_8932)
rLess(_8912,_8916,_8926,0)
hasMaterial(_3178,_12856,_12858,_12860,_12862,_12864)
hasMaterial(_7864,_12870,_12872,_12874,_12876,_12878)
rLess(_12860,0,_12874,_12878)
rMore(_12864,_12858,_12872,_12864*0.1)
pig(_7864,_23744,_23746,_23748,_23750)
isHigherOrEqual(_3178,_7864)
hasMaterial(_3178,_7876,_7878,_7880,_7882,_7884)
hasMaterial(_7864,_7890,_7892,_7894,_7896,_7898)
rLess(_7892,0,_7878+_7882,_7884*0.95)
isEligibleFallTarget(_7864)

Figure 4.6: A successful Domino detection.

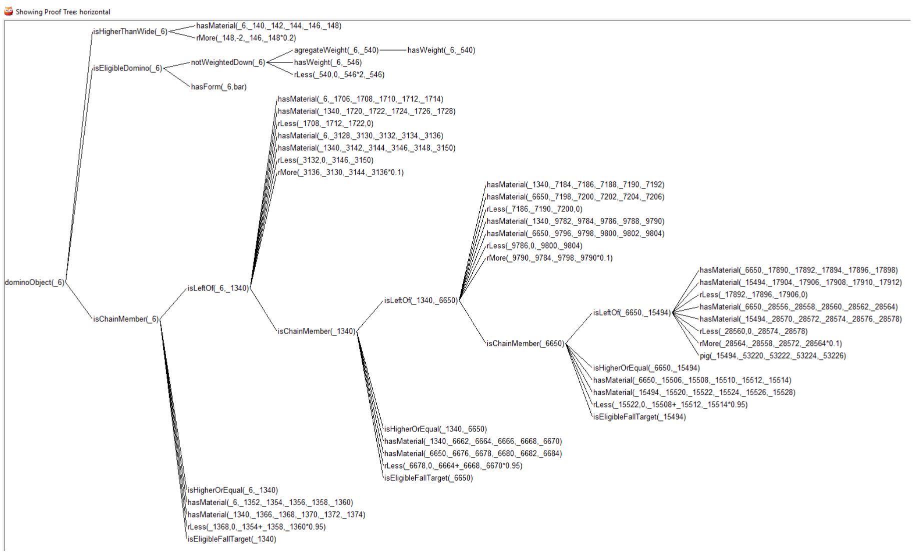
Showing Proof Tree: horizontal
dominoObject(_6)
isHigherThanWide(_6)
hasMaterial(_6,_140,_142,_144,_146,_148)
rMore(_148,-2,_146,_148*0.2)
isEligibleDomino(_6)
notWeightedDown(_6)
agregateWeight(_6,_540)
hasWeight(_6,_540)
hasWeight(_6,_546)
rLess(_540,0,_546*2,_546)
hasForm(_6,bar)
isChainMember(_6)
isLeftOf(_6,_1340)
hasMaterial(_6,_1706,_1708,_1710,_1712,_1714)
hasMaterial(_1340,_1720,_1722,_1724,_1726,_1728)
rLess(_1708,_1712,_1722,0)
hasMaterial(_6,_3128,_3130,_3132,_3134,_3136)
hasMaterial(_1340,_3142,_3144,_3146,_3148,_3150)
rLess(_3132,0,_3146,_3150)
rMore(_3136,_3130,_3144,_3136*0.1)
isChainMember(_1340)
isLeftOf(_1340,_6650)
hasMaterial(_1340,_7184,_7186,_7188,_7190,_7192)
hasMaterial(_6650,_7198,_7200,_7202,_7204,_7206)
rLess(_7186,_7190,_7200,0)
hasMaterial(_1340,_9782,_9784,_9786,_9788,_9790)
hasMaterial(_6650,_9796,_9798,_9800,_9802,_9804)
rLess(_9786,0,_9800,_9804)
rMore(_9790,_9784,_9798,_9790*0.1)
isChainMember(_6650)
isLeftOf(_6650,_15494)
hasMaterial(_6650,_17890,_17892,_17894,_17896,_17898)
hasMaterial(_15494,_17904,_17906,_17908,_17910,_17912)
rLess(_17892,_17896,_17906,0)
hasMaterial(_6650,_28556,_28558,_28560,_28562,_28564)
hasMaterial(_15494,_28570,_28572,_28574,_28576,_28578)
rLess(_28560,0,_28574,_28578)
rMore(_28564,_28558,_28572,_28564*0.1)
pig(_15494,_53220,_53222,_53224,_53226)
isHigherOrEqual(_6650,_15494)
hasMaterial(_6650,_15506,_15508,_15510,_15512,_15514)
hasMaterial(_15494,_15520,_15522,_15524,_15526,_15528)
rLess(_15522,0,_15508+_15512,_15514*0.95)
isEligibleFallTarget(_15494)
isHigherOrEqual(_1340,_6650)
hasMaterial(_1340,_6662,_6664,_6666,_6668,_6670)
hasMaterial(_6650,_6676,_6678,_6680,_6682,_6684)
rLess(_6678,0,_6664+_6668,_6670*0.95)
isEligibleFallTarget(_6650)
isHigherOrEqual(_6,_1340)
hasMaterial(_6,_1352,_1354,_1356,_1358,_1360)
hasMaterial(_1340,_1366,_1368,_1370,_1372,_1374)
rLess(_1368,0,_1354+_1358,_1360*0.95)
isEligibleFallTarget(_1340)

Figure 4.7: A larger Domino tree.

5 Summary and Conclusions

The EBG Algorithm can be implemented in SWI-Prolog, though it had problems with negative statements. This potentially means extra work when adapting input involving negations. However, every description of what something isn't can surely we rewritten as a positive description. The implementation worked according to specification, and was able to function with multiple, differing domain theories, as well as at least one example per such theory, without modification[1]. However, in examples of such a small scale, it requires imagination to see an advantage for a human operator, as it is usually visible at a glance that an example fulfils the defined criteria. In turn, the visual output from generating a proof tree for an Angry Birds level was a challenge to read. Despite this, the output of EBG could still serve as a base for an analysis program meant to evaluate the performance for an agent playing Angry Birds. As the effort of providing sufficient definitions is prohibitively high compared to the result, the author believes that EBG has more potential with more complex systems, were a human operator would not be able to come to the same conclusions at a glance, and an actual explanation is beneficial. A notable shortcoming of EBG is still it's reliance on a domain theory, meaning it can not be used when the program has insufficient knowledge of the domain.

References

Borgida, A., Mitchell, T. & Williamson, K. E. (1986). Learning Improved Integrity Constraints and Schemas From Exceptions in Data and Knowledge Bases. In *Topics in information systems* (S. 259–286). Springer New York. doi: 10.1007/978-1-4612-4980-1_23

Lebowitz, M. (1984). Concept learning in a rich input domain: Generalization-based memory.
(CUCS-111-84)

Mitchell, T. M., Keller, R. M. & Kedar-Cabelli, S. T. (1988). Explanation-based generalization: A unifying view. In *Readings in cognitive science* (S. 382–398). Elsevier. doi: 10.1016/b978-1-4832-1446-7.50034-0

[1]Technically, changing which files are read changes the code, as the implementation offers no way for the user to change the input files from the console or a GUI. This, however, does not actually change the EBG algorithm, and is thus irrelevant.

Wielemaker, J. & Anjewierden, A. (2002). An architecture for making object-oriented systems available from prolog.

Winston, P. H., Binford, T. O., Katz, B. & & Lowry, M. (1983). Aaai-83 proceedings. In (S. 433 - 439). Department of Computer Science, Stanford University.

Statement

I hereby declare in accordance with § 17(2) APO that I wrote the Bachelor Thesis on hand independently and that I didn't use any but the declared sources and aids.

Place, Date Signature

Erklärung

Ich erkläre hiermit gemäß § 17 Abs. 2 APO, dass ich die vorstehende Bachelorarbeit selbst-ständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum Unterschrift