Inducing Game Rules from Varying Quality Game Play

Alastair Flynn

February 5, 2020

Chapter 1

Abstract

When learning programs in Inductive Logic Programming (ILP) optimisations to the dataset used to train the systems can often be as effective as improvements to the systems themselves. General Game Playing (GGP) is a framework in which an artificial intelligence program is required to play a variety of games successfully. The framework includes repositories of game descriptions. The Inductive General Game Playing (IGGP) problem challenges ILP systems to learn the rules of a game from observation of gameplay. Existing work on IGGP has always assumed that the game player being observed makes random moves. To address this limitation, in this paper we analyse the effect of using optimal verses random gameplay traces as well as the effect of varying the number of traces in the training set. The General Game Playing competition winner in 2014, Sancho, is used to generate optimal game traces for a large number of games and the systems, Metagol, Aleph and ILASP are trained and tested with combinations of optimal and random data. Our results show...

Chapter 2

Introduction

General Game Playing (GGP) is a framework in which artificial intelligence programs are required to play a large number of games successfully.[6] Games in the framework range in both number of players and complexity; from the single player Eight Puzzle to the six player Chinese Checkers, and from Rock Paper Scissors to Chess[1]. Every year the GGP competition takes place, aiming to find the best GGP program. A consequence of this framework is large repositories of game descriptions written in the Game Description Language (GDL), a logic programming language built for describing games as state machines[11]. GIVE EXAMPLE OF GDL GAME These game descriptions provide a perfect setting for an instance of the ILP problem. In Inductive Logic Programming machine learning systems are tasked with learning logic programs given some background knowledge and a set of values for which the programs are true of false[10]. In his 2019 paper Andrew Croppers paper [4] defines the Inductive General Game Playing (IGGP) problem; given a set of gameplay observations the goal is to induce the rules of the game. GIVE EXAMPLE OF IGGP PROBLEM In this paper we use the IGGP framework to evaluate the ability of ILP agents to correctly induce the rules of a game given different sets of gameplay observations - optimal gameplay verses random gameplay. So far no work has been done on how the observations given as training data to the systems affect their ability to learn games. In machine learning the probably approximately correct learning framework (PAC) is a mathematical model that can be applied to machine learning systems for analysis. It gives an upper bound on the number of training examples needed for a learner to probably (with probability at least $[1 - \delta]$) learn a hypothesis that is approximately (within error ϵ)correct[9]. In section ? we test to find the number of examples beyond which the reduction in ϵ and δ becomes negligible. We would expect models trained on the same distribution as they are tested on the perform the best since it is generally accepted

that the accuracy of a model increases the closer the test data distribution is to the training data distribution.[9]. However, Gonzales [7] suggests that a system trained on a different domain to the one it is tested can outperform a system trained and tested on the same domain. In his 2010 paper Ben-David shows that training data taken from multiple different domains can in fact give lower error on testing data that traning data taken from any single domain, including the testing domain [2]. It is not clear in our case what selection of training data will result in the most effect learning. We will train the ILP systems on different combinations of optimal and random data as well as testing them on each individually. In section ? we test to find the optimal selection of distributions to take our training data from.

Contributions

- We evaluate ILP systems ability to synthesise correct programs using the IGGP framework with both optimal gameplay data and random gameplay data

- We evaluate ILP systems ability to synthesise correct programs with different numbers of gameplay observations

Chapter 3

Related work

3.1 Machine learning

?

(in PAC) The cardinality of this example set must be at most a polynomial function of the size of the vocabulary used in constructing hypotheses.

More examples allows for longer programs therefore better programs?

3.2 ILP

Inductive logic programming is a form of machine learning that uses logic programming to represent examples, background knowledge, and learned programs[5]. To learn the program is supplied with positive examples, negative examples and the background knowledge. In the general inductive setting we are provided with three languages.

- $\mathcal{L}_O$: the language of observations (positive and negative examples)

- $\mathcal{L}_B$: the language of background knowledge

- $\mathcal{L}_H$: the language of hypotheses

The general inductive problem is as follows: given a consistent set of examples or observations $O \subseteq \mathcal{L}_O$ and consistent background knowledge $B \subseteq \mathcal{L}_B$ find an hypothesis $H \in \mathcal{L}_H$ such that

$$B \wedge H \vDash O$$

[10] That is that the generated hypothesis and the background knowledge imply the positive examples and not imply the negative examples.

Example task (grandparent relation)

ILP systems generally regard ILP as a search problem. The search space is the set of well formed hypothesis. Often a set of inference rules are applied to the starting hypothesis, the new hypothesise are then pruned and expanded according to how often $B \wedge H \vDash O$ in the observations O. When all the observations are implied then a correct program has been found.

3.2.1 Metagol?

?Should I add a section on how metagol works here as an example?

To evaluate ILP systems the learning tasks they are given need to be small enough that it is feasible for them to learn the programs, but also diverse enough that we test the full capability of the system. The Inductive general game playing framework provides this. To understand IGGP we first need to explain the General game playing framework.

3.3 GGP

General game playing is a framework for evaluating an agenet's general intelligence accross a wide range of tasks [4]. The idea is that agents are able to accept declarative descriptions of arbitrary games at run time and are able to use such descriptions to play those games effectively. All the games are finite, discrete, deterministic multi-player games of complete information. The games can vary in number of players, dimensions and complexy. For example games such as rock paper scissons have 0 dimensions and only 10 rules in the given GGP ruleset, more complex games such as checkers has 52 rules and is 2 dimensional. There are also single player games such as eight-puzzle or Fizz Buzz. The agents play games selected from an exsisting database which lists their descriptions described in the Game Description Language (GDL). GDL is a language based on first order logic described in section 3.4.1. The list of games along with their descriptions is available online[1]. Matches in the GGP framework take place through an online framework called the Gamemaster. The agents connect to an online Game Manager (part of the Gamemaster) service which arbitrates indavidual matches. The connecting agents send several pieces of information to the Game Manger inclding a game that is already know by the manager and the required number of players. The match can then be started by pressing a start button, when pressed the agents will recive a *Start* message including the role of the player (e.g. black or white in chess) and a description on the game in GDL. The Game Manger communicates all instructions to the players through HTTP[6]. In

2005 an annual International General Game Playing Competition (IGGPC) was set up which still runs to this day. Each year hopeful participants pit GGP agents against one another to find the best one. The competitors take part in a series of rounds of increasing complexity. The agent that wins the most games in these rounds is the winner. The 2014 winner Sancho is used in this paper to generate optimal game traces for the IGGP task. The Game description language is used in the IGGP as an example of the rules to learn and the rules to use when generating examples. It is a type of logical programming language.

3.4 Logical Programming

Logic programming is a programming paradigm based on formal logic. Programs are made up of facts and rules. Rules are made up of two parts: the head and the body. They can be read as logical implications where the conjunction of all the elements in the body imply the head. The syntax is different for different logical programming languages but the head is usually written before the body. A fact is simply a rule without a body, that is, a statement that is taken as true. The compiler takes queries and returns weather they are true of false. If there are free variables in the query the compiler assigns them values for which the query is true. Logical programming is good for symbolic non-numeric computation. It is well suited to solving problems that involve well defined objects and relations between them, such as a GGP game.

3.4.1 GDL

Game definition language is the formal language used in the GGP competition to specify the rules of the games.[11] It is suited to specifying game rules because:

- It is a purely declarative language

- It has restrictions to ensure that all questions of logical entailment are decidable

- There are some reserved words, which tailor the language to the task of defining games

These descriptions define the games in terms of a set of true facts capturing the information needed to give the following predicates:

- The initial game state

- The goal state

- The terminal state

In addition, logical rules are used to describe the following:

- The legal moves for a given player and state

- The next state for a given player state and move

- The termination and goal conditions

In the IGGP problem the tasks are the rules for the goal, the next state, the legal states and the terminal state.

3.5   IGGP

The IGGP problem uses the GDL game descriptions from the GGP framework to construct the IGGP dataset containing 50 games. The IGGP problem itself is described in section **??**. A machanism is also provided by Andrew Cropper et al. [4] to turn the GGP games into new IGGP tasks. This mechanism has been modified and added to to generate optimal traces.

3.6   Prolog

Prolog is one of the most popular and well used logical programming languages. The syntax in Prolog for rules and facts is fairly intuitive. A simple fact might be `son(bob,alice)`.. This tells us that the atom `alice` relates the atom `bob` in the `son` relation. A rule is written `parent(X,Y) :- son(Y,X)` which means if we have the relation son of Y and X then X relates Y in parent. The symbol :- is the same as reverse implication ($\Leftarrow$). We can query a program in the Prolog environment. If we typed in "`parent(alice,bob)`." then it would return true because we have `son(bob,alice)` and the rule which tells us that if X is a son of Y else then Y is a parent of X so `alice` is a parent of `bob`. Predicates can be conjoined with the comma "," (e.g. `a(X,Y,Z) :- b(X,Y), c(Z,Y)`). Disjunction is expressed with the semicolon ";" (e.g. `a(X,Y,Z) :- b(X,Y) ; c(Z,Y)`.

Prolog answers queries using a process know as proof seach and unification. The unification process is similar to logical unification, two terms unify if they are the same term or if they contain variables that can be instantiated

with terms in such a way that the new terms are equal. For example the terms `name(bob).` and `name(X).` will unify with `X = bob`. The Prolog ISO defines the Herbrand Algorithm for unification [8]. A Prolog query is a set of goals. The Prolog System decides weather they are satisfiable or not. When a query is asked the of the Prolog system it executes something similar to the following algorithm. The exact implementations vary among Prolog systems however they are all roughly this algorithm.

Algorithm 1: Execute Prolog Goals

Input: A list of goals $GoalList = G_1, G_2, ..., G_M$

Function ***execute*** *(Program, GoalList, Success):*

    if $empty(GoalList)$ **then**

        | $Success \leftarrow true$

    end

    while ***not*** *empty(GoalList)* **do**

        $Goal \leftarrow head(GoalList)$;

        $OtherGoals \leftarrow tail(GoalList)$;

        $Satisfied \leftarrow false$;

        while ***not*** *Satisfied and there are more clauses in the program* **do**

            Let next clause in the program be $H \vdash B_1, ..., B_n$;

            Construct a variant of this clause $H' \vdash B'_1, ..., B'_n$;

            $match(Goal, H', MatchOK, Instant)$;

            if $MatchOK$ **then**

                $NewGoals \leftarrow append([B'_1, ..., B'_n], OtherGoals)$;

                $NewGoals \leftarrow substitue(Instant, NewGoals)$;

                $execute(Program, NewGoals, Satisfied)$;

            end

        end

        $Success \leftarrow Satisfied$;;

    end

---

the program listing is wearched for a term to unify with. The listing is searched in the order it is written in. When Prolog finds a matching rule it then attempts to sequentially unify the terms of the body using the same method. If the rule has no body then the variables are assigned and the terms unify. If Prolog fails to unify two terms then it backtracks, assignes differnt values to [3]

Chapter 4

Logical setting

- What precisely is the problem? - You can paraphrase a lot from the IGGP paper

The IGGP problem is defined in Andrew Croppers paper Inductive general game playing [4]. Much like the problem of ILP 3.2 the problem setting consists of examples about the truth or falsity of a formula F and a hypothesis H which covers F if H entails F. We assume the languages of:

- $\mathscr{E}$ the language of examples (observations)

- $\mathscr{B}$ the language of background knowledge

- $\mathscr{H}$ the language of hypotheses

Each of these languages can be see as a subset of those described in 3.2. All the predicates involved in the task are taken from the GDL descriptions of games in the Stanford GGP* library. A lot of the atoms in the descriptions are not function-free, that is, they are nested predicates. For example `true(count(9))`. We flatten all of these to single, non nested predicates, i.e. `true_count(9)`. This is done because some ILP systems do not support function symbols. We can therefore assume that both $\mathscr{E}$ and $\mathscr{B}$ are function-free. The language of hypotheses $\mathscr{H}$ can be assumed to consist of datalog programs with stratified negation as described here[12]. Stratified negation is not necessary but in practice allows significantly more concise programs, and thus often makes the learning task computationally easier. We first define an IGGP input the use it to define the IGGP problem. An IGGP input needs to capture the idea of an observation about a game. The input is based on the general input for the Logical induction problem of section 3.2 since this is a sub problem of it.

The IGGP Input: An input Δ is a set of triples $\{(B_i, E_i^+, E_i^-)\}_m^{i=1}$ where

- $B_i \subset \mathcal{B}$ represents background knowledge

- $E_i^+ \subseteq \mathscr{E}$ and $E_i^- \subseteq \mathscr{E}$ represent positive and negative examples respectively

An IGGP input forms the IGGP problem:

The IGGP Problem: Given an IGGP input Δ, the IGGP problem is to return a hypothesis $H \in \mathscr{H}$ such that for all $(B_i, E_i^+, E_i^-) \in \Delta$ it holds that $H \cup B_i \vDash E_i^+$ and $H \cup B_i \nvDash E_i^-$

In this paper the aim is to experiment how different ILP systems preform at this problem given optimal traces verses how they preform with random traces. That is varying the contents of $B \subseteq \mathscr{B}$. Experiments will also be preformed to determine the effect of varying the size of the input set $|\Delta|$.

Chapter 5

Implementations

5.1 Eight Puzzle

To write an optimal player for Eight Puzzle I decide to use an approach based on best first search.

5.1.1 Best first search

Best first search generates a graph of the game states, expanding the most promising nodes first according to a heuristic for how close to the goal state the current state is. When the goal node is reached the search gives the path from the root node to it, i.e. the list of moves made to get to the goal. Since Best first search is a well know and often used algorithm in Prolog we decided to use a standard implmentation from Bratko[3]. This decision was made to avoid needless mistakes and inefficiency that would have come with a new independent implementation of this algorithm.

Best First search is a generic algorithm that needs certain problem specific predicates to be implemented for it to function. These predicates are:

- **The successor predicate -** `s(Node,Node1,Cost)`: This predicate is true if there is an arc costing `Cost` between state `Node` and state `Node1`. In Eight Puzzle we set the cost of all arcs to be 1.

- **The Goal predicate -** `goal(Node)`: This is true if the state `Node` is a goal state.

- **The heuristic -** `h(Node,H)`: This is a relation that relates a state `Node` to a value `H` that is a heuristic for how close to the state is to the goal state.

To represent the board I decided to use a 9 element list with `b` representing the blank tile INSERT IMAGE OF GOAL BOARD

```
[1,2,3,4,5,6,7,8,b]
```

. I defined some useful helper predicates to start with. The Manhatten distance between two tiles between two tiles is the difference in X coordinates plus the difference in Y coordinates, I wrote a predicate to calculate this relation:

```
mandist(X1-Y1,X2-Y2,D) :-
    diff(X1,X2,Dx),
    diff(Y1,Y2,Dy),
    D is Dx+Dy.
```

I also wrote predicates to calculate the coordinates of a tile from its position in the list, one that relates two tiles if they are next to each other on the grid and one that gives the position in the list of the blank tile.

Successor

To define the successor predicate I decided the best way to do it was to relate two boards by swapping the blank tile with its neighbours. I first wrote a function swap:

```
swap(I,J,L1,L3) :-
    same_length(L1,L3),
    append(BeforeI,[AtI|PastI],L1),
    append(BeforeI,[AtJ|PastI],L2),
    append(BeforeJ,[AtJ|PastJ],L2),
    append(BeforeJ,[AtI|PastJ],L3),
    length(BeforeI,I),
    length(BeforeJ,J).
```

This function only uses built in predicates that all do the obvious thing. In the first two appends this rule takes L1 and replaces the element I with the element J to make list L2 then in the second two takes list L2 and replaces J with I to make list L3. The variable AtI and AtJ will be instantiated with the elements at index I and J because of the restriction on the length of the sublist before them at the end of the rule. The successor function is defined as:

```
s(B1,B2,1) :-
    member(N,B1),
```

```
    nth0(NI,B1,N),
    blank_pos(B1,BI),
    neighbor(BI,NI),
    swap(BI,NI,B1,B2).
```

Since all arc (move) costs in my version of Eight Puzzle are 1 the predicate is only true when Cost is 1. `member(N,B1)` is true when N is a member of B1. Here it restricts N to values on the board. `nth0(NI,B1,N)` sets NI to the index of the value N in the list B1, i.e. it tells us the position on the board of the value we are considering. `blank_pos(B1,BI)` sets BI to the position on the board of the blank. `neighbor(BI,NI)` is true if the positions BI and NI are next to each other on the board. If all of these are true we then instantiate B2 with the new board by swapping the blank and its neighbour.

Goal

Defining the goal was actually very simple, I added the predicate:

```
goal([1,2,3,4,5,6,7,8,b])
```

This is all that is needed to define the goal state.

Heuristic

A good heuristic for Eight Puzzle needs to capture an estimate of the distance of the current state from the goal. I initially decided to use the sum of the Manhattan distances of each tile from its position in the goal state. To calculate this I wrote a predicate `totdist(L,Goal,N)` which relates a board L, a goal Goal and the sum of the Manhattan distances N.

```
totdist(L,Goal,N) :-
    totdist1(L,Goal,N,0).

totdist1([],_Goal,0,_Pos).

totdist1([b|T],Goal,N,Pos) :-
    !,Pos1 is Pos + 1,
    totdist1(T,Goal,N,Pos1).

totdist1([V|T],Goal,N,Pos) :-
    nth0(I,Goal,V),
    coord(Pos,X1-Y1),
    coord(I,X2-Y2),
```

```
    mandist(X1-Y1,X2-Y2,D),
    Pos1 is Pos + 1,
    totdist1(T,Goal,N1,Pos1),
    N is N1 + D.
```

My implementation of this predicate iterates through the list working out the Manhattan distance for each tile. When iterating through the list the way to keep track of how far through it you are (the position on the board you are currently looking at) is use a separate variable as a counter. Here I used Pos. After testing my program with this heuristic I found that it took about 6 seconds to come up with the first solution. I decided that I could probably do better if I had another heuristic that I combined with the Manhattan distance. I decided to use the number of tiles that were out of the row and/or column they were in. I wrote a predicate `outofpos(B,Goal,N,Pos)` that relates the board and the goal to the number of tiles out column plus the number of tiles out of row.

```
outofpos([],_Goal,0,_Pos).

outofpos([b|T],Goal,N,Pos) :- !,
    Pos1 is Pos + 1,
    outofpos(T,Goal,N,Pos1).

outofpos([V|T],Goal,N,Pos) :-
    coord(Pos,X1-Y1),
    nth0(PosG,Goal,V),
    coord(PosG,X2-Y2),
    Pos1 is Pos + 1,
    outofpos(T,Goal,N1,Pos1),
    (   xandy(X1 = X2, Y1 = Y2) ->
        N is N1
    ;   xory(X1 = X2, Y1 = Y2) ->
        N is N1 + 1
    ;   N is N1 + 2
    ).

xandy(X,Y) :- X,Y.
xory(X,Y) :- X;Y.
```

This predicate iterates through the board and for each tile looks at the place it is on the goal board and checks which coordinates match. If both match then it is the right column and row so the total remains the same, otherwise

14

it is incremented. I eventually found that the heuristic the worked the fastest was Manhattan distance total + 2 * number of tiles out of position. I wrote this predicate for the heuristic:

```prolog
h(B,H) :-
    goal(Goal),
    totdist(B,Goal,D),
    outofpos(B,Goal,N,0),
    H is D + (2*N).
```

5.2   Noughts and crosses

Chapter 6

Experiments

6.1 Optimal Play vs Random with fixed sample size

6.2 Mixed datasets (50/50 random and optimal)

6.2.1 Results

Training Data	Testing Data	Predicate	Metagol	Aleph

6.3 Optimal Play vs Random with varying sample size

Bibliography

[1] Gdl website.

[2] Shai Ben-David, John Blitzer, Koby Crammer, Alex Kulesza, Fernando Pereira, and Jennifer Wortman Vaughan. A theory of learning from different domains. *Machine Learning*, 79(1-2):151–175, 2010.

[3] Ivan Bratko. *Prolog Programming for Artificial Intelligence, 4th Edition*. Addison-Wesley, 2012.

[4] Andrew Cropper, Richard Evans, and Mark Law. Inductive general game playing. *CoRR*, abs/1906.09627, 2019.

[5] Andrew Cropper and Stephen H. Muggleton. Learning efficient logic programs. *Machine Learning*, 108(7):1063–1083, 2019.

[6] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[7] Carlos R. González and Yaser S. Abu-Mostafa. Mismatched training and test distributions can outperform matched ones. *Neural Computation*, 27(2):365–387, 2015.

[8] ISO/IEC. Information technology programming languages prolog part 1: General core. ISO 13211-1:1995, International Organization for Standardization, Geneva, Switzerland, 1995.

[9] Tom M. Mitchell. *Machine learning, International Edition*. McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[10] Stephen Muggleton. Inductive logic programming. *New Generation Comput.*, 8(4):295–318, 1991.

[11] David Haley Eric Schkufza Michael Genesereth Nathaniel Love, Timothy Hinrichs. General game playing: Game description language specification, 2006.

[12] Kenneth A. Ross. Modular stratification and magic sets for datalog programs with negation. *J. ACM*, 41(6):1216–1266, 1994.