Using Program Synthesis and Inductive Logic Programming to solve Bongard Problems

Sharad Chitlangia[1][⋆], Atharv Sonwane[1][⋆], Tirtharaj Dash[1], Lovekesh Vig[2], Gautam Shroff[1,2], and Ashwin Srinivasan[1]

[1] APPCAIR, BITS Pilani, K K Birla Goa Campus
[2] TCS Research, New Delhi

Abstract. The ability to recognize and make analogies is often used as a measure of or to even test human intelligence [8]. The ability to solve Bongard problems is an example of such a test [8]. It has also been postulated that the ability to rapidly construct novel abstractions is critical to being able to solve analogical problems [3,9]. Given an image, the ability to construct a program that would generate that image is one form of abstraction, as exemplified in the Dreamcoder project [7]. In this paper we present a preliminary examination of whether programs constructed by Dreamcoder can be used for analogical reasoning to solve certain Bongard problems. We use Dreamcoder to discover programs that generate the images in a Bongard problem and represent each of these as a sequence of state transitions. We decorate the states using positional information in an automated manner and then encode the resulting sequence into logical facts in Prolog. We use inductive logic programming (ILP), to learn an (interpretable) theory for the abstract concept involved in an instance of a Bongard problem. Experiments on synthetically created Bongard problems for concepts such as 'above/below' and 'clockwise/counterclockwise' demonstrate that our end-to-end system can solve such problems. We study the importance and completeness of each component of our approach, highlighting its current limitations and pointing to directions for improvement in our formulation as well as in elements of any Dreamcoder-like program synthesis system used for such an approach.

Keywords: Bongard Problems · Program Synthesis · ILP

1 Introduction

It has long been understood that choice of representation can make a significant difference to the efficacy of machine-based induction. "An understanding of the relationship between problem formulation and problem solving efficiency is a prerequisite for the design of procedures that can automatically choose the most 'appropriate' representation of a problem (they can find a 'point of view' of the problem that maximally simplifies the process of finding a solution", Amarel [1]. In fact, in the previous
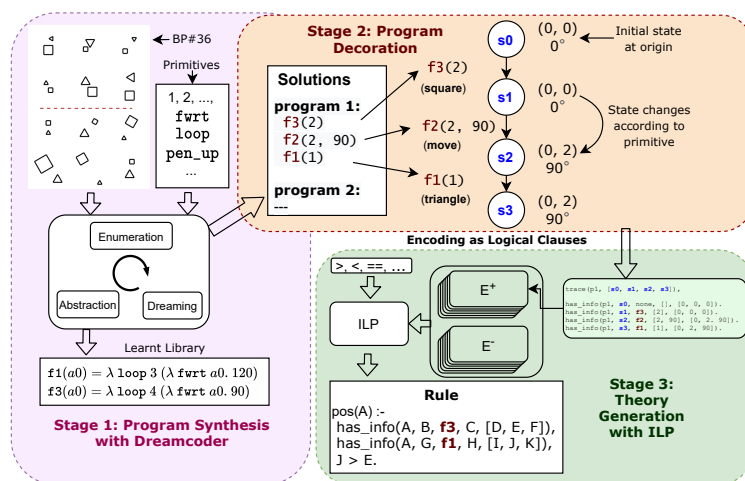

[⋆] equal contribution

Fig. 1. The proposed Inductive Programming System

quote, by 'choose' what is probably meant is 'construct', if we are to avoid kicking Feigenbaum's bottleneck down the road from extracting models to extracting representations. However, quoting Melanie Mitchell in 2021 [9] "...enabling machines to form human-like conceptual abstractions is still an almost completely unsolved problem."

In 1967, Mikhail Bongard proposed [2] a set of 100 puzzles, ordered by increasing complexity, designed to challenge the abstraction and analogy-making abilities of both humans and AI systems [3]. Each Bongard problem captures a unique abstract concept in the space of drawings of shapes and lines with six positive and six negative examples. Since the concepts required for each Bongard problem are different, an AI system for solving them would need to form the required (novel) abstractions from very few examples (a feature of 'analogical' reasoning [9]), using simpler abstractions that are both sufficiently powerful as well as problem-specific. Further, since there are only a hundred or so Bongard problems, a data-intensive approach that assumes access to large volumes of similar problems is ruled out [4].

---

[3] As explained in [9]: (i) constructing abstractions to draw similarities between observations constitutes an analogy (ii) analogies that occur often become categories/classes/concepts (e.g. fruit, 'roundness', 'theft') and are no longer referred to as analogies (iii) thus the term analogy is most often used for situations requiring novel abstractions, e.g. in few-shot scenarios such as Bongard problems.

[4] Quoting [9]: "In short, today's ConvNets, . . . do not have what it takes to do the kinds of abstraction ... required even in Bongard's idealized problems"

Thus the question arises as to what kind of representations might suffice to enable rapid analogy-making, i.e., capturing the 'essential' similarity between one set of drawings vs another.

'Dreamcoder', a recently proposed program induction system [7] synthesizes a *program* to solve input/output problems by creating useful abstractions across problems. From starting with very simple primitives such as 'move forward', 'pen up/down', etc., Dreamcoder can *abduce* new functions such as 'draw a triangle' to draw more complex figures.

There are good reasons to look at this form of program-synthesis as a mechanism for automated representation-learning: (a) Empirical results with programs like Dreamcoder show that it is possible to synthesize programs for diverse tasks from a very small number or even a single example; (b) Symbolic language adopted for primitive functions (λ-expressions) is sufficiently expressive for constructing programs for data of any type; (c) The intermediate representations have clearly defined interpretations, based on functional composition. Further, Inductive Logic Programming (ILP [10]) can use these symbolic abstractions for constructing human-readable discriminatory models.

We postulate that programs synthesized to re-draw the figures in a Bongard problem should serve as good intermediate abstractions from which to learn, 'on the fly', the concept involved. For example, if the concept were 'sameness', programs that generated positive drawings would contain repeated use of the same higher level functions. We also find there are some high-complexity Bongard problems for which program synthesis fails, as well as cases where the program as a representation alone may be insufficient; and we postulate possible mechanisms to alleviate these difficulties in the future.

2   Related Work

Recent critical reviews [3,9] of progress in AI research, posit that few-shot visual reasoning is one of the hallmarks of human intelligence and more than half a century of research in AI, progress on that end is still limited. Bongard Problems [2] are a popular set of visual reasoning puzzles for testing AI systems. To the best of our knowledge, the maximum number of problems solved by an artificial intelligence system is 42 [4]. Similar to our approach, the approach proposed in [4] uses a visual language and a pragmatic approach to solve Bongard Problems. In our approach, the visual language is composed of graphical λ-expression programs that are rich in abstractions. Hence, the novelty in our approach is the amount of background knowledge encoded in the graphical language. For instance, even simple shapes such as triangles, squares, circles are invented before being used for more complicated visual reasoning tasks.

Michie's original definition of Machine Learning was in terms of two orthogonal axes - predictive accuracy and comprehensibility of hypothe-

ses. Where approximate methods such as Neural Networks and Support Vector Machines might excel on predictive accuracy, symbolic methods such as Inductive Programming excel at being comprehensible. In addition to being interpretable, programs, in particular logic programs, have been known to show strong generalization capabilities [12]. Through our 3-staged approach that solves some Bongard problems we show the benefits of using programs as representations.

3   Methodology

Our 3-staged inductive programming approach to solving Bongard Problems is shown in Fig. 1 and described below:

Dreamcoder. The input to Dreamcoder is a set of tasks X, and the goal is to both infer a program ρ_x solving each task $x \in X$ and learn a library L of primitives and program abstractions encoding a prior distribution $P(\rho|L)$ that will enable it to solve tasks in that domain. In our system, Synthesized λ-expressions from Dreamcoder act as a generative representations of images in a Bongard Problem. Given the set of 12 images of a Bongard problem, Dreamcoder outputs 12 programs, one for generating each image.

Transducer. This module converts the synthesised programs into a first order logic representation based the program's decorated state transition diagram.

λ-expression programs that generate images of a Bongard problem in our system are purely functional. Functional Programs are stateless in nature. But as noted originally in Dreamcoder [7], LOGO-style graphics programs are implemented using a state monad and by encoding each primitive action in a continuation passing style. This enables representation of the purely functional program in an imperative-style state transition diagram.

Imperative programs can be seen as sequences of instructions (primitives) each of which when executed cause a *transition* from one *state* to another. We hypothesise that these states and primitive calls in programs representing images from a Bongard problem hold valuable information for identifying its differentiating concept. The state information is extracted from the result after evaluating sub-programs and is used to *decorate* a state-transition diagram of the overall program (see Figure 1). In the current implementation of the system, we limit our program state to the very basic type of information that may be extracted: positional coordinates (x,y) and orientation of the cursor (on the drawing canvas). More complicated state information can also be added using feature extractors (such as the ones used in [4]) working on top of the partial program or the partially generated image. For instance, to extract high level perceptual

information such as if two shapes are touching or not, it might be difficult to extract directly from a sub-program but can be achieved by means of an image processing module working on top of the graphic drawings of the produced programs.

The transducer generates the decorated state transition diagram for a given program as described above and represents this as a set of FOL statements. Here, the sequences of states for some program `p1` are specified by the `trace/2` predicate as: `trace(p1,[s0,s1,s2])`. Then each state in the trace is associated with its decorated information by representing using a `has_info/5` predicate. For example the fact that the state `s0` of program `p1` uses primitive `f0` with arguments `[1,2]` after which the co-ordinates become `(2.0,2.0)` with orientation `1.33` can be written as: `has_info(+Program,-State,#Primitive,-Args,[-X,-Y,-Angle])`.

ILP. ILP has been the focus of much research to solve Bongard problems [11] due to the logical nature of the puzzles and its formulation of learning a theory that differentiates, using some background knowledge, positive from negative examples. We use Aleph [13] to learn a classification theory where the background knowledge for positive and negative examples are the set of FOL facts output from transducer corresponding to the generated programs for the positive and negative images in a Bongard problem respectively. The BK is encoded using the `trace/2` and `has_info/5` predicates as described previously.

4 Empirical Evaluation

We evaluate our system on a set of synthetic Bongard problems with the following aims: (1) Can our end-to-end system solve a reasonable set of Bongard problems? (2) What are some of the breaking points and limitations of our 3-staged inductive programming system in the context of solving Bongard problems and more generally, for visual reasoning?

4.1 Materials

Data. We synthetically create a set of 14 Bongard problems each consisting of 12 images (6 positive, 6 negative)[5] that match original concepts encoded in Bongard problems. We aim for a representative sample of Bongard problems from recent studies. For instance we include problems (e.g. BP #21 and BP #60) for which a convolutional neural network[14,15] fails to learn the concept encoded.

Background Knowledge. The provided primitives to Dreamcoder (see 5) serve as its background knowledge. For the ILP stage, we provide additional comparison style predicates such as `==`, `\==` for comparison of arguments, `>`, `<`, for comparison of `(X,Y)` coordinates, `bw_90_270`, `lt(90)`, etc., for comparison of orientation.

---

[5] These are available at this link.

4.2   Results

The results for various bongard problems, including the libraries learnt with Dreamcoder, the final theories learnt with ILP and their explanations, are shown in Fig. 3. Our system is able to solve 8 of the total 14 problems considered. On the easier end the tasks solved include basic visual reasoning concepts such as the presence of circle (BP #24) and clockwise spirals (BP #16). These are solved by simply learning a theory to check for the relevant invented primitive in the λ-expression program. On the more complicated end, where some of the current systems fail [14] are BP #21, BP #23, BP #36, BP #60, BP #75. Notably, in [14] it was found that even with 20,000 examples, a CNN could not learn the high-level concept encoded in BP #21 and BP #60.

Improving Graphical Program Synthesis. The two major limitations of our current system on the graphical program synthesis module can be seen in Fig. 2: (1) It cannot directly work on top of hand-drawn images, due to lack of certain primitives, such as BP #4, BP #14, BP #94 and (2) It cannot draw figures where there might be a lot to draw. Apart from the inclusion of the required primitives, it may be possible to get around (1) by comparison of the output of a graphics program with hand-drawn images using a learned metric as previously done in [6]. For (2), the enumerative search itself might need to be guided with more than simply a distribution over primitives as previously demonstrated in the sub-area of execution guided synthesis such as [5] which works by evaluating partial programs to direct the search early-on.

Improving State Decorations. The state decorations can be augmented with learned features from Dreamcoder-produced programs or the images. These features could be propositional (e.g. 'Is the first argument of last executed primitive greater than 2?'). It is interesting to note here that a wide variety of feature extractors could be utilised to add to the state decorations., including those described in [2] which was able to solve 42 Bongard Problems when evaluated exhaustively.

Improving the final theory learning step. It is of interest to investigate if multiple theories learned across different Bongard puzzles can be used for inventing new clauses via inverse-resolution techniques (such as inter-construction) that might serve to improve theory learning for subsequent puzzles: For example, the concept of 'smallness' (as seen in the theories for BP #21 and BP #53 where it is expressed as the length of sides being less than number of sides for a shape) over multiple problems can be generalised by a meta-rule allowing learning of simpler discriminative theories. It may also be possible to construct meta-rules that generalise over sub-programs. If we assume (sub-)programs are computational encodings of predicates, then such meta-rules would be statements in at least 2nd order logic, that allow quantification over predicates.

Concept	Concave vs Convex (#4)	Large Total Line Length (#14)	Collinearity (#40)	Number of Lines (#85)	Location of solid shape (#94)	Nuber of sides of hatching (#96)
Failure Stage	Representation	Representation	Search	Search	Representation	Search
Explanation	Solid fills cannot be represented	Arbitrary curves cannot be represented	Two many shapes to be drawn	Unable to draw 5 lirregular ines	Solid fills cannot be represented	Too many lines to be drawn

Fig. 2. The failure modes of the Inductive Programming System

Concept	Invented Primitives	Theory	Explanation
Anti-clockwise vs Clock-wise (BP #16)	f2(a0), f3(a0) both draw anti-clockwise spirals with a0 controlling tightness of the spiral. f2 and f3 use step length 2 and 1.	pos(A):- has_info(A,B,f3,C,[D,E,F]). pos(A):- has_info(A,B,f2,C,[D,E,F]).	Presence of invented primitive for drawing spirals that are anti-clockwise.
Smaller shape present (BP #21)	f1(a0, a1): Draw an a0-sided polygon with sides of length a1	pos(A): - has_info(A,B,rtfwint,C,[D,E,F]), C=[G|H],H=[I|J],G>I, has_info(A,K,f1,L,[D,E,F]). pos(A):- has_info(A,B,f1,C,[D,E,F]), C=[G|H],H=[I|J],G>I.	Program contains a move primitive where the division factor for angle is greater than multiplication factor for distance, *or* there is a polygon with side length less than number of sides. Indicating the shape is small.
Number of Shapes (BP #23)	None	pos(A):- trace(A,B),B=[C|D],D=[E|F],F=[]	Contains only 2 states (in which only one shape can be drawn since first state is always initial state).
Circle present (BP #24)	f0(a0): Draw a circle with radius a0	pos(A):- has_info(A,B,f0,D,[E,F,G]).	Presence of invented primitive for drawing circle.
Triangle above Square (BP #36)	f1(a0): Draws a triangle of side length a0. f3(a0): Draws a square of side length a0	pos(A):- has_info(A,B,f3,C,[D,E,F]), has_info(A,G,f1,H,[I,J,K]), J>E.	Triangle exists with y coordinate greater than that of square
Enclosed shape has fewer sides (BP #53)	f1(a0, a1): Draw an a0-sided polygon with sides of length a1	pos(A):- has_info(A,B,f0,C,[D,E,F]), has_info(A,R,pt,Q,[K,L,M]), has_info(A,I,f0,J,[K,L,M]), C=[G|H],J=[N|O],O=[P|Q], G>N, N>P.	Smaller polygon (having length of side smaller than number of sides) has has fewer sides than larger (enclosing) polygon.
Two Similar Shapes (BP #60)	None	pos(A):- has_info(A,B,C,D,[E,F,G]), has_info(A,H,C,I,[J,K,L]), H\==B.	There are two shapes which are different (i.e. calls on two different invented primitives)
Triangle on concave side of arc (BP #75)	f0(a0): Draws an arc of radius a0 f1(a0): draws a triangle of side length a0	pos(A):- has_info(A,B,f0,C,[D,E,F]), bw_90_270(F), has_info(A,G,f1,H,[I,J,K]), D>I. pos(A):- has_info(A,G,f1,H,[I,J,F]), bw_270_90(K),I>D.	Either the arc is facing right hand side and x coordinates of triangle is greater than those of arc, *or* arc is facing left and x coordinate of triangle is less than that of arc.

Fig. 3. Solutions for Bongard Problems using our approach.

5   Conclusion

Recent progress in program synthesis exemplified by projects such as Dreamcoder [7] led us to examine whether a program generated to reconstruct an image might serve as a good representation for rapidly learning new visual concepts, such as required while solving Bongard problems. We have presented an end-to-end automated system combining inductive functional program synthesis using Dreamcoder and theory induction us-

ing Inductive Logic Programming and used it to conduct experiments on solving Bongard problems.

Our approach is indeed able to solve certain Bongard problems, as well as produce theories that are interpretable; using only few examples unlike recent deep-learning approaches. In contrast, the program synthesis process of Dreamcoder itself involves 'dreaming' examples using a generative mode that aid in discovering higher-level programmatic primitives via refactoring that are crucial to enable the downstream ILP learner to produce a coherent and accurate discriminative theory.

We have also highlighted opportunities for improving both the program synthesis and ILP parts of the process, including possibilities for marrying our approach with deep-learning. We believe our results demonstrate the promise of using automatically synthesised programs as abstract representations to aid in rapidly learning novel concepts and furthering research in analogical reasoning.

References

1. Amarel, S.: On representations of problems of reasoning about actions. In: Readings in artificial intelligence, pp. 2–22. Elsevier (1981)
2. Bongard, M.: Pattern recognition. In: Encyclopedia of Social Network Analysis and Mining. 2nd Ed. (2018)
3. Chollet, F.: On the measure of intelligence. ArXiv **abs/1911.01547** (2019)
4. Depeweg, S., Rothkopf, C., Jäkel, F.: Solving bongard problems with a visual language and pragmatic reasoning. ArXiv **abs/1804.04452** (2018)
5. Ellis, K., Nye, M., Pu, Y., Sosa, F., Tenenbaum, J., Solar-Lezama, A.: Write, execute, assess: Program synthesis with a repl. arXiv preprint arXiv:1906.04604 (2019)
6. Ellis, K., Ritchie, D., Solar-Lezama, A., Tenenbaum, J.B.: Learning to infer graphics programs from hand-drawn images. arXiv preprint arXiv:1707.09627 (2017)
7. Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L.B., Cary, L., Solar-Lezama, A., Tenenbaum, J.: Dreamcoder: bootstrapping inductive program synthesis with wake-sleep library learning. Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (2021)
8. Hofstadter, D.R.: Fluid concepts and creative analogies: Computer models of the fundamental mechanisms of thought. Basic books (1995)
9. Mitchell, M.: Artificial intelligence: A guide for thinking humans. Penguin UK (2019)
10. Muggleton, S., Raedt, L.D.: Inductive logic programming: Theory and methods. J. Log. Program. **19/20**, 629–679 (1994)
11. Saito, K., Nakano, R.: A Concept Learning Algorithm with Adaptive Search, p. 347–363. Oxford University Press, Inc., USA (1996)
12. Silver, T., Allen, K.R., Lew, A.K., Kaelbling, L., Tenenbaum, J.: Few-shot bayesian imitation learning with logic over programs. ArXiv **abs/1904.06317** (2019)
13. Srinivasan, A.: The aleph manual (2001)
14. Stabinger, S., Rodríguez-Sánchez, A., Piater, J.: 25 years of cnns: Can we compare to human abstraction capabilities? arXiv preprint arXiv:1607.08366 (2016)
15. Yun, X., Bohn, T.A., Ling, C.X.: A deeper look at bongard problems. In: Canadian Conference on AI. pp. 528–539 (2020)