Adaptive Compressed Search

Robert Wünsche(✉)

Technische Universität Dresden, Dresden, Germany
robertw89@gmail.com

Abstract. Program-search as induction and abduction is one of the key pillars of any sufficiently advanced AGI. In this paper, we present a mechanism to search for programs given a specific bias. This bias is flexible to some degree. Another novel attribute of the mechanism is the use of compression that selects simple programs over complex ones. The complexity of the program is changing all the time over the lifetime of the agent.

Keywords: Compression · Universal-search · Induction

1 Introduction

The problem of searching for the (optimal) solution to an inversion problem is an old problem. It was formalized first by Leonid Anatolievich Levin in his Levin Search (LS) [2] algorithm. LS searches for the shortest program which solves a problem in the shortest time. Longer programs get exponentially longer time allocated. Ideally, the interpretation of all programs is done in parallel - this is not feasible even on modern hardware for moderate programs. Ray Solomonoff introduced the idea to search for the solutions of optimization problems with Induction [14]. Biasing the search process of LS with the use of a probabilistic selection of the candidate instruction was introduced by Jürgen Schmidhuber in his Adaptive Levin search (ALS) [12] algorithm.

Life long incremental learning is necessary for some AGI subsystems. Life long learning is realized in this algorithm with a compressed storage of programs or fragments. The storage is used for reading (by composing new programs from existing/known parts) and writing (the algorithm stores the solution and parts thereof). Thus it fulfills the criteria of a long-term storage proposed by Solomonoff [14].

2 Algorithm

This algorithm[1] reuses the idea of biasing the search, similar to ALS. One difference is that it applies the probabilistic selection not just to instruction, but

[1] The sourcecode can be found at https://github.com/PtrMan/AGIconf2018CompressedSearch.

M. Iklé et al. (Eds.): AGI 2018, LNAI 10999, pp. 303–310, 2018.
https://doi.org/10.1007/978-3-319-97676-1_29

to parts of, already known and/or learned, programs. These parts are called fragments in this publication. The advantage of this is that it biases the search towards programs which already contain known parts for reuse.

Another idea from Schmidhuber is used to bias the search so that simpler programs are preferred. This is done by exploiting compression progress [11] to guide the search towards smaller compressed descriptions of the candidate programs first. This is done because a reuse of already known parts of a program results in a smaller compressed size. Solomonoff called this idea "Conceptual jump-size" [13], thus this algorithm is preferring to test solutions with a smaller conceptual jump-size first over ones with a larger one.

The time for the compression, generation of candidates, testing and adaptation after finding a solution all have upper bounds. This is because the number of elements in the primary storage must be limited on a real machine. Indeed, this causes no issue in the algorithm, because not all parts of previously found solutions are important for the compression or the sampling of candidate programs.

This algorithm can be described as a non-optimal strategy of Optimal Ordered Problem Solver(OOPS) [10] for program induction/abduction.

Learning Algorithm

(Compress1) - compress dictionary programs into primary storage

```
until(foundAllSolutionForLength || globalTimeboundReached)
   call "enumerator algorithm"
      < to generate secondary candidate programs >
   (process A) compress secondary candidate programs one by one and
               store tuple (program, required#Bits)
   (process B) sort secondary candidate programs tuples by
               required#Bits
   (process C) test secondary candidate programs ordered by
               required#Bits, break if solution(s) found for
               the smallest possible required#Bits
   if solution(s) found
      store found program(s) into primary storage
      change probability masses of the relevant fragments
      break (only if the algorithm has to calculate one
             solution to the problem)

put solution(s) into primary storage by adding it
and compressing it to the primary storage, adapt probabilities
after the strategy of ALS or any other strategy
```

Compress1 is just necessary for the first run of the algorithm when the primary storage is empty and thus has to be initialized.

We note the following:

Parallelism:

The processes A, B and C are applied in sequence in the implementation of this algorithm. It is possible to compute the processes concurrently, whereby processes A, B and C work in parallel.

Optimization for Simple Problems:

Processes A and B might get disabled for problems where the time of the process C is significantly larger than the time required for A and B.

Enumerator algorithm - generates candidate programs:

```
parameters:
   < number of instructions and fragments >
   nInstructionsOrFragments
   probabilityOfInstruction < range [0; 1) >
   < maximal number of fragments which are reused in program >
   nMaxFragments

candidateProgram = []
nFragments = 0

for i from 1 to nInstructionsOrFragments
   selectInstruction = rand(0.0, 1.0) < probabilityOfInstruction
   if nFragments < nMaxFragments || selectInstruction
      append to candidateProgram random Instruction sampled by
      probability of instruction as in ALS
   else
      append to candidateProgram random Fragment sampled by
      probability of instructions as in ALS
      nFragments++

return candidateProgram
```

Fast Hash-Based Compressor

For this work, it was necessary to develop a fast compression algorithm. The fact that the primary storage has to be compressed just once can be exploited for the design of the compression algorithm. One possibility for engineering a fast compression algorithm is to exploit the two phases - compression of the training set and compression of the program candidates - into two phases which we call primary and secondary phrases, respectively.

Additionally, the compression with GZip as a compression algorithm is too slow.

Compression primary algorithm

```
parameters: prgrms : input programs

for iPrgrm in prgrms
   for iSlice in allPossibleSlices(iPrgrm)
      < increment a counter for the slice >
      storage.primaryIncForSlice(iSlice)

      < store >
      storage.primaryPut(iSlice)
```

storage.primaryPut(Slice) stores the slice in a hash-table based storage. The slice is just an array of integers. It can check for the existence of the slice, but it doesn't have to. The algorithm can afford a check because the primary fragments are stored once for every run, thus the additional time is not critical.

storage.primaryIncForSlice(Slice) increments a counter of the slice, it is (re)using the existing hashing functionality of the storage.

Compression secondary algorithm

```
parameters: prgrm : program to be compressed

i = 0
while i < len(prgrm)
   longestSliceInfo = storage.searchForLongestPossibleSlice(progrm[i:])

   if longestSliceInfo.longestSubsequentWasFound:
      if longestSliceInfo.foundInPrimary:
         out.appendRefPrimary(prgrm[i], longestSliceInfo)
      elif longestSliceInfo.foundInSecondary:
         out.appendRefSecondary(prgrm[i], longestSliceInfo)

   if not longestSliceInfo.longestSubsequentWasFound
      out.appendUncompressed(prgrm[i])

   < store all old subsequences which we don't yet know >
   sliceEndInFrontIdx = longestSliceInfo.sliceStartIdx
   for iSlice in allPossibleSlices(prgrm, 0, sliceEndInFrontIdx)
      if storage.hasPrimary(iSlice)
         continue
      if storage.hasSecondary(iSlice)
         continue

   i+=longestSliceInfo.len
```

The secondary algorithm attempts to find the longest possible remaining sequence. The longest (sub)sequence can be found by searching for the subsequence in the primary storage by hash. If this fails then the same test is done for the secondary storage. If this

test fails then the longest possible subsequence is added to the secondary storage. This treatment ensures that the program can be compressed - without the need to write to the primary storage. Thus a speedup of several orders of magnitude can be achieved for the compression of programs.

Discussion of Time-Bound for Generated Programs

To determine the maximal number of steps executed by the program one can use the following strategies or a linear combination of these

- calculate the max # of steps by the formula as used in ALS depending on the probabilities of the used instruction and fragments derived by Schmidhuber [12].
- calculate the max # of steps by the required number of bits in the corresponding compressed description of the generated program [9].

3 Experiment

Pong was chosen as a simple Test environment. The task was to generate a program to control the bat. The only input to the program was the signed horizontal distance from the bat to the ball. Every program was run for a fixed maximal amount of steps. A program failed the test if it didn't hit the bat in the given time.

The mean and standard deviation of the number of generated programs for 100 runs is shown in the Table 1. The compression bias was disabled for all runs. The samples were split at 250000 attempts and assumed a central distribution for both sets.

Table 1. Number of generated programs

	Mean	Standard deviation
Samples >= 250000 tries	646657	311304
Samples < 250000 tries	101545	68270

4 Comparison to Other Methods

Comparison with Evolutionary Algorithms

Genetic Programming [1] can search for Turing complete programs. However, it is unable to store and reuse fragments over multi runs for different problems as it is possible for the presented algorithm. The presented algorithm doesn't maintain a population of candidates for each problem. It does, however, store fragments between runs for problems. Genetic Programming uses "crossover" for transferring parts of programs between solutions. The presented algorithm does something similar - but it is more controlled because the fragment selection can be biased.

It doesn't favor simple programs over more complex ones (where the complexity is measured by the compression ratio). Genetic Programming requires supervision in the form of a scoring function. The presented algorithm is an unsupervised learning algorithm.

Cartesian Genetic Programming [4] can also search for Turing complete programs. It can learn to reuse parts of the program for the same solution. It inherits the other Problems from Genetic Programming.

Comparison with Meta-Optimizing Semantic Evolutionary Search
MOSES [3] is a supervised learning algorithm. It uses a population of programs which are derived from a single program with different parameter tunings. These groups are called demes. The presented algorithm doesn't have the concept of tuning parameters, because it is an unsupervised algorithm. Thus it can't know how to tune the parameters.

MOSES doesn't reuse parts of solutions of solved problems for future problems. There are some hints from the authors that this functionality is added later.

Comparison with Probabilistic Incremental Program Evolution (PIPE)
PIPE [6,7] reuses parts of found solutions with a memory [6], this is very similar to the fragments in this paper. PIPE can only work with tree-structured programs. The presented algorithm is described just for non-tree-structured programs. PIPE is a supervised algorithm contrary to the presented algorithm.

PIPE biases the search to smaller solutions - thus it uses compression. It doesn't, however, compress each candidate program like it is done by the algorithm presented in this paper.

Variants of PIPE can employ filtering [6,8] to automatically decompose a task into simpler subtasks which can be solved independently. The presented algorithm does something similar over multiple problems - it learns to reuse parts of solutions.

Comparison with OOPS
OOPS reuses parts of solved problems for current problems. The main difference is that it biases its search to programs with a higher probability, which is computed from the product of the probabilities of each instruction. OOPS tends to search shorter programs rather than longer programs which can be found quicker. The tradeoff is that the found longer programs might not generalize as well as the shorter ones.

Comparison with ALS
ALS biases the probabilities of the instructions to adapt found solutions to past problems. It doesn't have any way to represent fragments or adapt the probability distribution of them. ALS and this presented algorithm are both unsupervised (learning) algorithms.

5 Conclusion and Further Work

In this section, we present a few ideas for future work.

One avenue for future research is unifying the compressor and program generator by using a common data-structure for both.

A downside of the currently used compression scheme is that it doesn't have a way to encode n repetitions of the same word.

The used instruction-set of the programs is rather minimalistic and could be enhanced with functional prefixes and instructions to manipulate the virtual machine(s) [5]. These instructions could increase the generalization capabilities and shorten the programs for some problems [5].

Behavior-trees could be used as a flexible mechanism for the enumerator to generate candidates for an a Priori given problem-class. A problem-class could be, for example, data-structure manipulation algorithms.

The size of found programs can be reduced by trail and error by removing instructions. It is a valid solution to the problem(s) if it still solves them. The author refers to this process as the "shorting principle". The process is valid because shorter programs generalize better (occam's razor).

The selection and manipulation of fragments can be done by adaptive mechanisms. Techniques from Machine Learning, Evolutionary Algorithms and AGI can be employed here.

Another way to supplement the used principles is the use of emotions to control the search process. Emotions which interact could be aggression and depression.

- Aggression in the range [0; 1] is how often the AI tries unlikely or longer variations of programs
- Depression in the range [0; 1] could control after how many failed attempts it gives up looking for solutions to a problem

Aggression and depression could interact to control the direction of the search. It could, for example, try unlikely enumerator algorithms to generate candidate programs depending on a certain range of aggression and depression. For example, it could try an enumerator algorithm to generate algorithms which call into (random) addresses of the program if aggression is in the range [0.8;1.0] and depression in [0.9;1.0]. The bounds of the ranges to enable a certain strategy could be modified if they prove successful.

The novel improvement over previous approaches is the sorting of the checked programs by the compressed size. None of the compared algorithms has a functionality to check the solutions in a order determined by the compression ratio of all potential solutions. Almost all compared algorithms reuse solution parts for each solution. A few reuse parts of solutions between problems.

References

1. Koza, J.R.: Genetic Programming: A Paradigm for Genetically Breeding Populations of Computer Programs to Solve Problems, vol. 34. Department of Computer Science Stanford, Stanford University, CA (1990)
2. Levin, L.A.: Universal sequential search problems. Problemy Peredachi Informatsii **9**(3), 115–116 (1973)
3. Looks, M.: Competent program evolution. Ph.D. thesis, Washington University (2007)
4. Miller, J.F., Thomson, P.: Cartesian genetic programming. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 121–132. Springer, Heidelberg (2000). https://doi.org/10.1007/978-3-540-46239-2_9
5. Nowostawski, M., Purvis, M., Cranefield, S.: An architecture for self-organising evolvable virtual machines. In: Brueckner, S.A., Di Marzo Serugendo, G., Karageorgos, A., Nagpal, R. (eds.) ESOA 2004. LNCS (LNAI), vol. 3464, pp. 100–122. Springer, Heidelberg (2005). https://doi.org/10.1007/11494676_7
6. Salustowicz, R.: Probabilistic incremental program evolution (2003)

7. Salustowicz, R., Schmidhuber, J.: Probabilistic incremental program evolution. Evol. Comput. **5**(2), 123–141 (1997)
8. Sałustowicz, R.P., Schmidhuber, J.: Sequence learning through pipe and automatic task decomposition. In: Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2, pp. 1184–1191. Morgan Kaufmann Publishers Inc. (1999)
9. Schmidhuber, J.: The speed prior: a new simplicity measure yielding near-optimal computable predictions. In: Kivinen, J., Sloan, R.H. (eds.) COLT 2002. LNCS (LNAI), vol. 2375, pp. 216–228. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45435-7_15
10. Schmidhuber, J.: Optimal ordered problem solver. Mach. Learn. **54**(3), 211–254 (2004)
11. Schmidhuber, J.: Driven by compression progress: a simple principle explains essential aspects of subjective beauty, novelty, surprise, interestingness, attention, curiosity, creativity, art, science, music, jokes. In: Pezzulo, G., Butz, M.V., Sigaud, O., Baldassarre, G. (eds.) ABiALS 2008. LNCS (LNAI), vol. 5499, pp. 48–76. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02565-5_4
12. Schmidhuber, J., Zhao, J., Wiering, M.: Shifting inductive bias with success-story algorithm, adaptive levin search, and incremental self-improvement. Mach. Learn. **28**(1), 105–130 (1997)
13. Solomonoff, R.J.: A system for incremental learning based on algorithmic probability. In: Proceedings of the Sixth Israeli Conference on Artificial Intelligence, Computer Vision and Pattern Recognition, pp. 515–527 (1989)
14. Solomonoff, R.J.: Progress in incremental machine learning. In: NIPS Workshop on Universal Learning Algorithms and Optimal Search, Whistler, BC (2002)