Stepping Stones to Inductive Synthesis of Low-Level Looping Programs

Christopher D. Rosin
Parity Computing, Inc.
San Diego, California, USA
christopher.rosin@gmail.com

Abstract

Inductive program synthesis, from input/output examples, can provide an opportunity to automatically create programs from scratch without presupposing the algorithmic form of the solution. For induction of general programs with loops (as opposed to loop-free programs, or synthesis for domain-specific languages), the state of the art is at the level of introductory programming assignments. Most problems that require algorithmic subtlety, such as fast sorting, have remained out of reach without the benefit of significant problem-specific background knowledge. A key challenge is to identify cues that are available to guide search towards correct looping programs. We present MAKESPEARE, a simple delayed-acceptance hillclimbing method that synthesizes low-level looping programs from input/output examples. During search, delayed acceptance bypasses small gains to identify significantly-improved stepping stone programs that tend to generalize and enable further progress. The method performs well on a set of established benchmarks, and succeeds on the previously unsolved "Collatz Numbers" program synthesis problem. Additional benchmarks include the problem of rapidly sorting integer arrays, in which we observe the emergence of comb sort (a Shell sort variant that is empirically fast). MAKESPEARE has also synthesized a record-setting program on one of the puzzles from the TIS-100 assembly language programming game.

1 Introduction

Automated synthesis of programs from user requirements has a long history as an AI research goal (Waldinger and Lee 1969; Gulwani, Polozov, and Singh 2017). Recent interest in the problem has led to synthesis success for non-looping programs (e.g. clever bit-twiddling (Gulwani et al. 2011)), partial program "sketches" with holes to be synthesized (Solar-Lezama 2008; So and Oh 2017), domain-specific languages (e.g. Flash Fill in Microsoft Excel (Gulwani 2011)), and other areas. But for synthesis of general looping programs from scratch, the state of the art is at the level of introductory programming exercises (Helmuth and Spector 2015b; Helmuth, McPhee, and Spector 2018; Feser, Chaudhuri, and Dillig 2015). Most problems requiring algorithmic subtlety, such as fast sorting, have remained out of reach without using significant problem-specific background knowledge (Cai, Shin, and Song 2017; Agapitos and Lucas 2006).

We seek a simple method that can make progress and provide insight on synthesis problems requiring algorithmic subtlety. We target low-level programming languages, close to assembly language; this provides some grounding (avoiding open-ended exploration of high-level language design), and has the potential to yield a large speed advantage. We focus on inductive program synthesis[1] from input/output examples, which provides an opportunity to automatically create programs without presupposing the algorithmic form of the solution (other approaches such as synthesis from natural language description (Yin and Neubig 2017) or deduction from formal specifications (Manna and Waldinger 1980) often constrain the form of the solution (Gulwani, Polozov, and Singh 2017)). Our goal is *pure inductive synthesis*: without problem-specific primitives or background knowledge.

The search space of programs is vast, and a key challenge is to identify cues that are available to guide search towards correct programs. We focus on stochastic search in the hopes of tackling programs that are prohibitively complex for exhaustive methods (we note though that work on exhaustive methods is progressing (Balog et al. 2017; So and Oh 2017)). While problems requiring simple loops (e.g. same operation on each array element) can provide a clear path for stochastic search to hillclimb, problems requiring more complex loops could lack partially-correct *stepping stone* programs that enable progress to complete solutions.

We find a simple *Delayed Acceptance* extension to hillclimbing can identify stepping stones that lead to successful synthesis of looping programs with challenging requirements. Our method, *MAKESPEARE*, performs well on established benchmarks and on previously unsolved problems.

The main contributions of this paper are:

[1] "Program synthesis" here refers generally to the automated creation of programs that meet user-supplied specifications, and *inductive* program synthesis indicates that the specification takes the form of input/output examples – see (Pantridge et al. 2017) for similar usage. Others have used the term program *induction* to indicate that the program is implicit, e.g. in a neural network (Devlin et al. 2017), instead of being explicitly generated. In this paper though, we seek to output an explicit program.

```
delayed_acc_hillclimbing(I):  // I is period length.
  T:=0, B:=0, J:=0            // T is threshold, B is
  Repeat:                     //   best, J is #evals.
    If B==0 Then Z:=rand_pt() // Random init points.
            Else Z:=loc_op(Y) // Local change to Y.
    E:=eval(Z)                // Score (note E>=0).
    If E>=B Then B:=E , N:=Z  // Save N&accept later.
    If E>=T Then Y:=Z         // If E<T undo change.
    J:=J+1
    If J==I Then:             // End of period:
      If B==T Then Return Y   // No progress; exit.
      Else Y:=N , T:=B , J:=0 // Accept N, update T.
```

Figure 1: Delayed Acceptance Hillclimbing. When applied to program synthesis, N is the period's best program, Y is search's current program, and Z is the new variant to eval.

- We show Delayed Acceptance synthesizes low-level looping programs near the native assembly language level.
- Using this method, we identify properties of stepping stone programs that enable stochastic search to succeed on challenging program synthesis problems.
- We show how to solve problems that were previously open challenges, including the "Collatz Numbers" benchmark, the problem of synthesizing a fast sorting program without problem-specific background knowledge, and a record-setting result on a puzzle from the TIS-100 assembly language programming game.

2 Problem Statement

The program synthesizer receives a set of training examples with input and required output, along with the execution time bound for each. The synthesizer must return a program in the target language that yields correct output for all the training examples, within the execution time bound. The returned program is then tested on a set of test examples, and programs that are fully correct on these test examples are reported to *generalize perfectly*. In the problems here, there are several times more test examples than training examples in each problem, and some problems test *extrapolation* in which test examples are much larger than training examples. All problems here require synthesis of looping programs.

3 Method

3.1 Required Format of Target Languages

Our synthesizer, MAKESPEARE, is intended to target low-level languages, near the native assembly language level. A program is a sequence of S *instructions*. Each instruction consists of an *opcode* and a single *operand*. For example, an instruction might have an INC opcode with an operand specifying which variable register is to have its value incremented. Any of O opcodes can be paired with any of P operands (though some pairings may be nonfunctional). Specific languages may be instantiated in this format in various ways; we present two instantiations below.

```
swapP:=0.1 , doubleP:=0.9 , copyP:=0.5  // parameters
rand_pt(): Return Y with all Y[i] random instructions

loc_op(Y):     // given program Y, make local changes
  If rand(1.0)<swapP Then Z:=swap(Y) // swap 2 instr.
  Else:
    Z:=replacement(Y)
    With probability doubleP, Z:=replacement(Z)
  Return Z

replacement(Y):      // 1 or 2 point replacement in Y
  W := random instruction (random opcode and operand)
  With prob copyP, Set W’s opcode:=Y[i]’s for rand i
  With prob copyP, Set W’s operand:=Y[j]’s for rand j
  Return Y with Y[k]:=W for random k
```

Figure 2: Local Search Operators

```
eval(Y):       // Evaluate program Y on training set
  For each training example X:
    Run Y on X.  Initialize E:=0.
    E:=E+1 if X needs scalar result & Y’s is correct
    E:=E+1 per correct output array element
  If fully correct Then: // apply simplicity bonus:
    E:=E+1 per opcode N occurrence in Y
  Return E
```

Figure 3: Eval Scoring Function (see Sec 3.3 for TIS-100 detail)

Note that native[2] assembly language instructions often require more than one operand. The two instantiations below handle this differently: one adds a pseudo-opcode to specify the destination operand, and the other folds one operand into the opcode. Limiting our form to one operand may reduce the size of the search space, and increase the probability of search randomly generating a specific required instruction.

In any instantiation, one distinguished opcode N identifies instructions that do not execute any function (e.g. NOP no-operation) – search uses this opcode to simplify solution programs. In addition, search can generate variant programs by copying operands from one instruction to another, which benefits from a language in which operands generally have the same interpretation for different opcodes (although this isn't universally true for either instantiation below). Beyond these constraints, search does not require any further knowledge of the semantics of the language.

3.2 Search Method

A goal for our work is to use simple stochastic search methods with readily interpreted results, and to explore whether such methods can be effective in searching for looping programs. To this end, we consider Basic Hillclimbing: local search operators generate small changes to a current-best candidate solution, and if the changes result in a program at least as good then it is accepted as the current-best candidate. An immediate problem though is that Basic Hillclimb-

[2] We sometimes refer to *native* assembly language, to distinguish from our language which may differ in some respects.

Program: Sequence of S=32 instructions.
Data: R=6 reg (min R supporting **Input**), memory block
P Operands: Operand x is reg r, memory $[r]$ if used, immediate constant (0-3), or jump target $x\lfloor S/P\rfloor$. Note P=16 if memory is used, P=10 if not (scalar input).
O=14 Opcodes: x86-64 opcodes MOV, arithmetic ADD/SUB/IMUL/INC, comparison CMP/TEST, bitwise SHR/SHL, jumps JMP/JZ/JNZ/JG. Pseudo-opcode ARG sets current destination; scope continues to next ARG (unaffected by jumps). See Fig. 4 for example.
Distinguished Opcode N for Simplification: ARG.
Input: Memory block of n locations with arrays. Initial val for R reg: any scalar input, index of last elem of each input array (-1 if none), first elem of next (if any), and n. With 2d array of row size m, 2 reg get values m-1 and m.
Output: Scalar return val is reg R0. Overwrite input to return array; any separate output array must be in input.
Time and Memory Bounds: Time bound specified as *loopcount*: each backjump taken incurs loopcount 1. Memory access is limited to memory provided at input. Time/memory bound violations terminate execution.

Table 1: Language Instantiation: Simplified x86-64 Subset

ing can fairly readily progress by accumulating special-case code along the lines of "if input=X then output Y". This can quickly run into a local optimum where there's no further room in the S instructions to improve, and in any case such programs will not usually generalize. In Section 5.6 we observe such behavior for Basic Hillclimbing.

We therefore turn to *Delayed Acceptance hillclimbing* (Figure 1). Rather than immediately accepting an update to the current-best candidate solution, we continue to gather additional candidates for a *period* of I steps, and then accept the best found during that period. At that point, the score of the current-best sets a threshold T. During the next period of I steps we take a sequential random walk through candidates having score at least T; with large I this random walk can do some global exploration. We terminate when a period of I steps passes with no improvement. The resulting search trajectory can be summarized by a small number of milestone programs, one per period. The method has a single parameter I; larger I give longer runs that explore more before accepting improvements.

We use the name *Delayed Acceptance* to place the method in the same family as previously established *step counting hill climbing* (Bykov and Petrovic 2016) and *late acceptance hill climbing* (Burke and Bykov 2008). These are related modifications to hillclimbing, that also use a single parameter like I, but differ in details such as resource management for which our approach is more appropriate to our experiments. While these prior methods were initially developed for domains outside of program synthesis, late acceptance hillclimbing is competitive with genetic programming on common loop-free benchmarks (McDermott and Nicolau 2017).

Program: S=15 instr.; jump to first after last finishes.
Data: Reg ACC&BAK hold integers in $[-999, +999]$.
P Operands: Operand x is jump target $\lfloor xS/P\rfloor$ or immediate const $\in [-\lfloor P/2\rfloor, +\lfloor P/2\rfloor]$. Full range: P=1999. Also tested P=401, P=101, and P=21.
O=16 Opcodes: SAV; SWP; MOV ACC/op,DOWN (sends op to imager); MOV op,ACC; NEG; ADD/SUB op/ACC; NOP; jumps JMP/JEZ/JNZ/JLZ/JGZ.
Distinguished Opcode N for Simplification: NOP
Input: None needed here. ACC and BAK are initially 0.
Output: 30x18 pixel image. Imager receives X; Y; seq of colors at X, X+1, etc.; negative val ends seq. Program terminates if it exactly gets target image. Otherwise, score is max (at any point in execution) # of matching pixels.
Time Bounds: 1 cycle/instr., 2 for MOV op,DOWN.

Table 2: Language Instantiation: Subset of TIS-100

Local Search Operators The basic search operation is single-point replacement of one instruction with a random opcode and random operand. It may be limiting though to restrict to trajectories of single-point changes, each of which must leave a functioning program, so we also include two-point replacement and swap. Similar operators have previously been used in stochastic search of programs represented as sequences of instructions (Schkufza, Sharma, and Aiken 2016).

Finally, we use copy operations which modify replacement by copying operand or opcode from another instruction. Operand copying can help when several instructions need the same register operand. We also note research on machine learning to modify the distribution of instructions selected by an exhaustive search method (Balog et al. 2017); we aren't exploring such methods here, but copying provides a basic way to adapt replacement's instruction distribution.

The local search operators are detailed in Figure 2. Sec. 3.4 describes a grid search over parameter settings, and then the parameters in Fig. 2 are fixed for all remaining experiments presented here.

Evaluating Candidate Programs Stochastic search depends on an objective function to be optimized, and the choice of objective function affects the existence of partially-correct stepping stones that enable success. We evaluate candidate programs by running on the training examples and scoring the output. Each training example receives 1 point per fully-correct integer in the output (Figure 3). This is similar to prior approaches (Helmuth and Spector 2015b) but coarser-grained in that it doesn't assign partial credit for a partially-correct output integer.

Fully correct programs get a simplicity bonus (#occurrences of opcode N, since it doesn't produce executable code); this can aid generalization (Helmuth et al. 2017).

Programs run on each training example for its time bound. "Time" is instantiation dependent, but must be deterministic.

A run's final simplest program achieving training set success is evaluated on the test set (test set results do not affect scores used for search). In experiments here, the primary measure of a run's success is whether the final program generalizes perfectly, following precedent used for established benchmarks (Helmuth and Spector 2015b).

3.3 Language Instantiations

We use two instantiations of Sec. 3.1's language format. Both include typical assembly language opcodes like MOV for copying data (e.g. from register to memory), SUB for subtraction, JMP for jump (goto), and conditional jumps (e.g. JNZ jumps if previous result was nonzero).

Simplified x86-64 Subset Most of the benchmarks use a language based on a simplified subset of the x86-64 assembly language used in Intel and AMD CPUs (Table 1). This is just a tiny subset of native x86-64, but it does include relevant frequently used native opcodes (Lawlor 2012).

Our focus is limited to programs operating on integers and arrays; extensions such as floating point and string manipulation would typically be addressed with additional primitives (Helmuth and Spector 2015c; Forstenlechner et al. 2017).

Time is measured using *loopcount*: each backwards jump taken incurs loopcount 1, and total loopcount cannot exceed the bound initially specified. This is a simple deterministic scheme that usually has an intuitive interpretation; e.g. if a program requires a single pass through an array of n elements performing constant work on each element, it will use a loopcount of n. Coupled with MAKESPEARE's bonus for simplification, search favors small loops requiring few iterations. We note though that this is a coarse-grained performance measure, and puts no pressure on programs for fine-grained optimizations (e.g. based on instruction choice and sequencing). Replacing loopcount with actual execution time measurements could favor fine-grained optimizations, but would be nondeterministic which we do not support.

TIS-100 We explore a small portion of the TIS-100 assembly language programming game (Zachtronics 2015), that fits easily within our approach (see Table 2). We use just one TIS-100 "node," which leaves aside the game's unusual multi-node parallel programming. This still allows us to use the game's "Image Test Pattern" puzzles, where a program must generate a 30x18 pixel image to match a target.

We vary the number of allowed operands P. The full P=1999 constants [-999,+999] have a low probability of a randomly-chosen operand hitting a specific needed value (e.g. "3" which is a pixel color needed for benchmarks here). We therefore also consider restricted ranges with smaller P.

The TIS-100 game scores successful programs according to program size and cycle count. MAKESPEARE directly optimizes program size, but not cycle count. We have MAKESPEARE record the lowest cycle count achieved in each run, at the minimum size correct program that is found.

Implementation Code and data are available for download.[3] Both languages are compiled to native x86-64 using DynASM (Pall 2017). For our simplified x86-64 subset, this gives hardware semantics for native opcodes. Code is instrumented with time/memory bounds checks. The full set of Delayed Acceptance experiments below finish in 10 days, using 6 machines, each with a 4-core Intel i7-6700 CPU.

3.4 Search Parameters

With swapP$\in\{0, 0.1\}$, doubleP$\in\{0, 0.1, 0.5, 0.9\}$, copyP $\in\{0, 0.1, 0.5\}$, and I$\in\{3k, 10k, 25k, 75k, 150k\}$, a grid search was run on the *preliminary benchmarks* in Table 3. For each combination, Delayed Acceptance runs 100 times with max 300k evaluated programs per run. We find the parameters in Fig. 2 with I=75k have the most total runs that generalize perfectly, solving all 5 preliminary benchmarks.

Running the grid search with Basic Hillclimbing yields similar results. In experiments below, we compare Delayed Acceptance and Basic Hillclimbing using the same parameters for both, with Basic Hillclimbing using the same resource bound as Delayed Acceptance and the same stopping criterion (stop after period of length I with no progress).

The choice of I=75k permits only 4 periods with our max 300k evaluated programs per run. Note Delayed Acceptance terminates naturally when progress stops (Fig. 1), and it usually does so within 10-20 periods. With unrestricted tests until natural termination at I=75k, on each of the 5 preliminary benchmarks over 98% of eventual test set successes come within 10 periods – so we target about 10 periods when using larger numbers of evaluated programs.

Our protocol on a problem starts with 100 runs, each with I=75k and max 4 periods (300k programs), enabling quantitative comparison with other published work on established benchmarks (Helmuth and Spector 2015b). If there's no training set success, the next level is 100 runs with I=2M and max 9 periods (18M programs) each. If still no training set success, the final level is 30 runs with I=100M and max 10 periods (1 billion programs) each – this is a reasonable maximum given typical runtimes. At the first level with training set success, take the simplest training set success program from each run and evaluate on the test set, reporting the total percentage of runs that generalize perfectly. This protocol works well across a range of difficulty, and is a recommended starting point on new problems.

4 Benchmark Descriptions

Table 3 describes the benchmarks. The first 5, used in Sec. 3.4, were the hardest array problems for a constrained exhaustive search (So and Oh 2017) using input/output examples plus problem-specific program templates (e.g. prespecifying some loops). We use only input/output examples, and generate our own larger training and test sets.

The second set are previously established benchmarks (Helmuth and Spector 2015b), created partly in response to a community call for stronger benchmarks (McDermott et al. 2012). These enable comparison with published results (Helmuth and Spector 2015b; Forstenlechner et al. 2017; Pantridge et al. 2017; Helmuth, McPhee, and Spector 2018; Forstenlechner et al. 2018). Appropriate to MAKESPEARE's scope, we use all integer and array benchmarks from this suite which require a loop for MAKESPEARE (excluding string and floating point benchmarks, which need additional language capabilities (Helmuth and Spector

[3] https://github.com/ChristopherRosin/MAKESPEARE

Benchmark	**Time**
Prelim. Benchmarks adapted from *(So and Oh 2017)*	
Cube Elements: Given array a of n elements, cube each element (in place)	$2n$
4th Power: Raise each elem. of a to 4th power	$2n$
Sum Sq of Elem: Given a, return $\sum_{i=1}^{n} a[i]^2$	$2n$
Prod Sq of Elem: Given a, return $\prod_{i=1}^{n} a[i]^2$	$2n$
Sum Abs: Given a, return $\sum_{i=1}^{n} \lvert a[i] \rvert$	$2n$
Established Benchmarks *(Helmuth and Spector 2015b)*	
Negative To Zero: Given a, $b[i]$=$\max(a[i], 0)$	300
Vectors Summed: Given a&b, $c[i]$=$a[i]$+$b[i]$	300
Last Index of Zero: Return max i with $a[i]$=0	300
Count Odds: Given a, return count of odd $a[i]$	300
Mirror Image: Return 1 iff a is reverse of b	300
Sum of Squares: Given x, return $\sum_{i=1}^{x} i^2$	300
Collatz Numbers: Return #steps to reach 1 in Collatz sequence starting from input x	300
Additional/adapted benchmarks	
Binary Search: Given x&sorted a,find $a[i]$=x	$2\lg n$
Integer Sqrt: Given $x \geq 0$, return $\lfloor \sqrt{x} \rfloor$	$2\lg x$
Merge: Given sorted a,b, merge into sorted c.	$2n$
Slow Sort: Sort a in increasing order (in place)	$2n^2$
Fast Sort: Sort a in increasing order (in place)	$2n\lg n$†
Topological Sort: Given $v \times v$ edge array a, set $b[i]$ to min $L \geq 0$ so that $b[j] < L$ if $a[j][i] = 1$.	$2n$
DAG Sources: Given a as above, binary $b[i]$=1 iff for all j,$a[j][i]$=0 (L=0 in Topological Sort).	$2n$
TIS-100 benchmarks *(Zachtronics 2015)*	
Image Test Patt. 1: Set 30x18 img to color 3	10000
Image Test Patt. 2: Checkerboard: (X,Y) is color 3 if X+Y is even, else 0	10000

Table 3: Benchmarks. In Time, n is total input size (may have multiple arrays & preallocated output space); add 1 to n before taking lg, add 1 to bound before truncating to integer. † $2n\lg n$ Train, and $n^{5/3}$ Test; see Sec. 5.4.

2015b; Forstenlechner et al. 2017)). We use the explicit instances made available for download (Helmuth and Spector 2015a) rather than the protocol for generating new instances (Helmuth and Spector 2015c).

The third set includes difficult benchmarks. Our input/output sets require extrapolation (unlike the established benchmarks), with test set array sizes much larger than training. Time bounds scale with input size, and extrapolation tests that solution validity and speed scale to large instances.

This third set of benchmarks were adapted from prior work. Integer Sqrt is a classic benchmark for deductive program synthesis (Manna and Waldinger 1986), but hasn't been solved by pure inductive synthesis. A version of Merge was unsolved in the TerpreT framework for pure inductive synthesis (Gaunt et al. 2016b; 2016a) by several methods including gradient descent, SMT, and *SKETCH* (Solar-Lezama 2008). A targeted genetic programming effort previously synthesized binary search (Wolfson and Sipper 2009).

ADATE uses evolutionary methods and synthesized a slow sort without problem-specific knowledge (Hofmann et al. 2007). But fast sorts have not been produced by pure inductive synthesis. Inductive logic programming (Muggleton and De Raedt 1994) and genetic programming (Agapitos and Lucas 2006) have synthesized quicksort when given high-level problem-specific primitives like "partition". The neural programming framework has learned quicksort using problem-specific program execution traces which constrain the learned program (Cai, Shin, and Song 2017); an explicit long-term goal of the work is to remove this constraint. This same work solved a version of Topological Sort with execution traces.

DAG Sources is a simpler problem related to Topological Sort. These graph problems may be hard: our language lacks 2d array indexing; any needed indexing must be synthesized.

We refer to Merge, Integer Sqrt, Collatz Numbers, Fast Sort, and Topological Sort as **Challenge Problems** with no published solution using pure inductive synthesis, despite attention to versions of these in program synthesis literature.

The established benchmarks use up to 200 train and 2000 test examples, and array length up to 50 (Helmuth and Spector 2015c). Other benchmarks use 200 train and 2000 test examples, randomly generated. These training sets are large compared to programming-by-example work that seeks to minimize numbers of user-supplied examples in domain-specific applications (Raza, Gulwani, and Milic-Frayling 2014), but we need more examples for our less-constrained language and algorithmic problems.

Most array problems train up to length 21 (length 6 for the first 5 problems) and test extrapolation up to length 2001, with random lengths chosen uniformly. Exceptions are the graph problems which train up to 9×9 edge arrays and test up to 201×201, and Binary Search and Fast Sort which train up to 1001 and 201 (resp.) and test up to 100,001. Each instance picks random k in [1,63] and then random integers with k bits, except the first 5 benchmarks pick elements with up to 31 bits and then give them random sign. Topological Sort instances select random output, then a minimal set of edges that yield that output, and then additional edges with random density. DAG Sources uses the same graphs as Topological Sort.

The last 2 benchmarks are puzzles from the TIS-100 game. A program must output an image matching a target. This has similarity to other program synthesis benchmarks requiring output of pictures or patterns (So and Oh 2018).

5 Results

5.1 Benchmark Results

Table 4 shows results (see Sec. 5.7 for TIS-100). All benchmarks including the Challenge Problems are solved by Delayed Acceptance, except Topological Sort. Inspection of final programs for the easier problems shows they implement an expected algorithm, sometimes in condensed form. See Sec. 5.4&5.6 for example solutions from harder problems.

Table 4 compares Basic Hillclimbing at the same resource level: it performs worse in nearly all problems, and solves only one Challenge Problem.

% runs fully successful:	Basic Hillcl.	**Delayed Acceptance**	Smallest Program
First training success in 100 runs of 300k programs			
Cube Elements	44	**67**	5
4th Power	99	**100**	4
Sum Sq of Elem	1	**2**	4
Prod Sq of Elem	**65**	36	4
Sum Abs	29	**36**	6
Negative To Zero	42	**50**	6
Vectors Summed	17	**26**	5
Last Index of Zero	88	**97**	4
Count Odds	42	**80**	5
Mirror Image	0	**1**	6
Sum of Squares	**1**	**1**	5
First training success in 100 runs of 18M programs			
Slow Sort	5	**60**	10
Binary Search	0	**17**	7
DAG Sources	13	**59**	7
Merge	6	**13**	12
Integer Sqrt	0	**3**	11
First success (or failure) in 30 runs of 1G programs			
Collatz Numbers	0	**60**	8
Fast Sort	0	**3.33**	14
Topological Sort	0	0	-

Table 4: MAKESPEARE Delayed Acceptance results, and Basic Hillclimbing comparison: % of runs with full test set success, at first resource level with any Delayed Acceptance training set success. **Challenge Problems are shown in bold**; no prior published solution using pure inductive synthesis, despite prior attention in the literature. *Preliminary benchmarks are shown in italics*. “Smallest program” is size (# non-ARG instructions) of smallest training set success in any run.[4]

5.2 Comparison on Established Benchmarks

The best published results on the established benchmarks are from the well-developed genetic programming systems PushGP (Helmuth, McPhee, and Spector 2018) and G3P (Forstenlechner et al. 2017), which use differing program representations and search operators. We compare using the train/test protocol and resource bound (100 runs of 300k evaluated programs each) originally established for these benchmarks (Helmuth and Spector 2015b). MAKESPEARE's results are compared in Table 5 to the best results from PushGP and G3P across several published configurations of their methods. MAKESPEARE has the best result for 2 of the 6 benchmarks, as do each of the other systems.

Beyond genetic programming, other synthesis methods solved less than half of the established benchmarks in a comparative benchmarking effort (Pantridge et al. 2017).

MAKESPEARE has the first reported success on the Collatz Numbers problem (Helmuth and Spector 2015b; Pantridge et al. 2017; Perelman et al. 2014).

	Push	G3P	**MAKESPEARE**
Negative To Zero	82	**98**	50
Vectors Summed	11	**85**	26
Last Index of Zero	72	24	**97**
Count Odds	12	10	**80**
Mirror Image	**100**	1	1
Sum of Squares	**26**	13	1

Table 5: Comparison: %runs that generalize perfectly on established benchmarks (Helmuth and Spector 2015b), in 100 runs of max 300k programs each. PushGP (Helmuth, McPhee, and Spector 2018) and G3P (Forstenlechner et al. 2017; 2018) are genetic programming systems; results shown are best across several published configurations.

5.3 Comparison to Exhaustive Search

“Smallest program” in Table 4 is small enough that one may wonder if exhaustive search could succeed. But, even with the benefit of constraint to the minimum number of required instructions and registers, we find experimentally that exhaustive search in randomized order on the smallest example (Last Index of Zero: 4 non-ARG instructions plus one ARG) needs an expected 400M programs until first success: over 1000x as many as MAKESPEARE. Other problems need larger programs (non-ARG + ARGs). Larger exhaustive experiments become prohibitive, and based on search space size the gap (between MAKESPEARE and exhaustive search) could grow rapidly as program sizes increase.

5.4 Fast Sort Solution

```
   ARG R8
A: MOV 0    // Initialize R8 to 0.
   ARG R2   // R2 is initially n; used as gap size.
   IMUL 3   // Multiply gap size by 3/4.
   SHR 2
   ARG R0   // Initialize R0 to R2; gapped bubble
   MOV R2   //      sort pass compares [R8],[R0].
B: ADD 1    // Ensure final passes’ gap is 1 (not 0).
   CMP R6   // R6 is init. n-1 (and doesn’t change).
   JG A     // Pass is finished; start another.
   ARG R9
   MOV [R0] // Compare [R8],[R0]...
   SUB [R8]
   JG C
   ARG [R0] // ...exchange unless [R8]<[R0] already.
   MOV [R8]
   ARG [R8]
   ADD R9
C: INC R8   // Proceed to pass’s next pair.
   JNZ B
```

Figure 4: MAKESPEARE's Fast Sort (Comb Sort). Program terminates via time bound. Note ARG sets destination, with lexical scope down to next ARG (regardless of jumps).

[4] For benchmarks that succeed within 4 periods at I=75k, runs with training set success are extended until natural termination, solely to measure “smallest program” (these extensions do not affect other results). 4 periods is rarely enough to simplify programs.

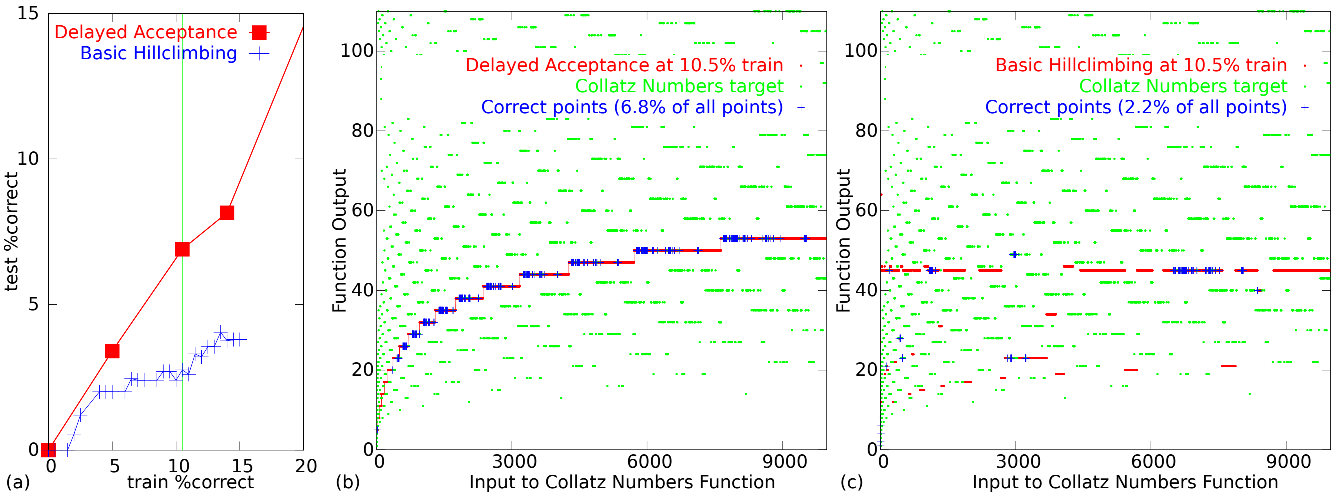

Figure 5: (a) Test %correct as function of train %correct, for sample Collatz Numbers trajectories. Search time goes from left to right too, since train %correct increases monotonically. Points show programs accepted by search. Next Delayed Acceptance point, off the graph, is 100% train&test. The programs' functions from 10.5% train (green vertical) are plotted in (b)&(c).

While we obtain multiple Fast Sort training set successes with time bound $2n \lg n$, none of them generalize perfectly to large test examples (up to length 100,001) at the $2n \lg n$ loopcount bound. We therefore select a relaxed subquadratic bound $n^{5/3}$ that arises in the analysis of particularly simple sorts (Sedgewick 1996). The simplest training set success program (Figure 4), with 14 non-ARG instructions, generalizes perfectly at this level.

This program turns out to be *comb sort*, which was originally suggested by Knuth (Knuth 1973) and later rediscovered independently by others (Dobosiewicz 1980; Lacey and Box 1991). Comb sort performs linear-time bubble sort passes on gapped sequences, multiplying gap size by a constant factor each pass; MAKESPEARE's factor of $3/4$ matches an original choice (Dobosiewicz 1980). When gap size reaches 1, further ungapped passes can continue to complete the sort (MAKESPEARE's version terminates at the time bound). Comb sort is a variant of Shell sort, which does a complete gapped insertion sort each pass.

While one can craft quadratic-time worst-case inputs for comb sort (Drozdek 2005), we are unlikely to encounter these in our random inputs. Comb sort is empirically fast, and even demonstrated faster average times than quicksort on random sequences up to length 1000 (well beyond our training set lengths) (Dobosiewicz 1980; Incerpi and Sedgewick 1987).

5.5 Program Simplification for Generalization

A motivation for program simplification is to improve generalization. This has been previously observed in genetic programming on benchmarks used here (Helmuth et al. 2017). We also see a benefit from simplification in our experiments.

Table 6 shows that the "Final" simplified program from each training set success run has improved generalization compared to an average over "All" accepted training success programs within the run (this trend is only reversed for one of the 18 benchmarks). The last row shows consistent perfect generalization if we select only the runs achieving training set success with "Minimal" program size (achieved within the time bound on max evaluated programs).

All training success programs within runs	79.9%
Final program from each training success run	84.3%
Runs yielding Minimal program size	100.0%

Table 6: Percentage of training successes that generalize perfectly. Averaged over the 18 solved benchmarks in Table 4.

"Final program" is a natural measure of generalization rate, and it varies from 100% (e.g. Collatz Numbers) down to much lower rates (e.g. 72% for Merge). "Minimal program" shows one approach to encourage reliable generalization is to do a large number of runs and take the simplest resulting program. Another approach (not explicitly tested here) would be to use an independent verification set, to check generalization of programs with training set success.

5.6 Stepping Stones

A *stepping stone* here is a partially-correct program that enables search to progress to a complete solution, as opposed to a partially-correct program that leads search to a dead end.

Figure 5(a) shows typical sample trajectories for Collatz Numbers. To help illustrate Delayed Acceptance's behavior, we compare it to Basic Hillclimbing. Delayed Acceptance and Basic Hillclimbing both make initial progress on training and test. But Basic Hillclimbing then accumulates small special-case training improvements, leading to a complex program representing the piecewise-constant function in Fig. 5(c). Such programs do not generalize well, and Basic Hillclimbing's test %correct stagnates while Delayed Acceptance finds larger training improvements that generalize well and enable progress to a full solution.

The Collatz Numbers target function repeatedly maps x to $\frac{x}{2}$ if even and $3x+1$ if odd, counting steps until x reaches 1. This target looks complicated (green in Fig. 5(b)&(c)); it isn't obvious that partially-correct stepping stones would exist.

Delayed Acceptance finds a simple partially-correct program that repeatedly multiplies by 3 and divides by 4 while counting steps (Fig. 5(b)); this generalizes relatively well and the loop's elements provide a stepping stone towards eventual full solution. This stepping stone is typical of successful Delayed Acceptance runs. Final Collatz Numbers solutions resemble the above description of the target function, though the simplest programs don't use the constant 3 but instead for odd n compute $n + \lceil \frac{n}{2} \rceil$ and count 2 steps.

Other challenging problems, including Integer Sqrt and Binary Search, show typical trajectories similar to the pattern in Fig. 5(a). While challenging problems may provide many opportunities for small special-case training improvements, accumulating these can lead to dead ends in terms of generalization and further progress. Delayed Acceptance bypasses these for more significantly-improved stepping stones that generalize well and enable further progress.

5.7 Results for TIS-100

TIS-100 (Zachtronics 2015) is an assembly language programming game that attracted a dedicated community of players who deeply explored the game and tracked the best programs they could find for each puzzle (Leaderboard 2018). This enables a concrete comparison of synthesis with strong human efforts, but we are not aware of previous strong/published program synthesis results on TIS-100.

The TIS-100 game evaluates solution programs by three measures: number of instructions, number of cycles, and number of "nodes" (which is always 1 in our implementation). For each benchmark, the community leaderboard tracks the best known programs by these measures (Leaderboard 2018). MAKESPEARE directly minimizes the number of instructions, so we focus on that category.

For Image Test Pattern 1 with 100 runs at I=2M, MAKESPEARE matches the best reported program at 7 instructions and 2282 cycles, with the same approach of a nested loop using just a single index (GltyBystndr 2016). Synthesis is successful despite a wide range of P=1999 operands, with the final program utilizing a large constant.

For Image Test Pattern 2, the community had no prior report of a single-node solution with under 11 instructions. To explore a variety of solutions, and to see whether restricted ranges of operands could help, we performed extended sets of 5000 runs at I=2M with each of the 4 operand ranges in Table 2. The best program was found with P=401 and has 9 instructions and 3596 cycles, which sufficed to set a record and claim a spot on the leaderboard (Leaderboard 2018).

6 Conclusion

We have shown that a simple Delayed Acceptance hill-climbing method successfully synthesizes low-level looping programs, near the assembly language level. Delayed Acceptance bypasses small gains to identify significantly-improved stepping stone programs that tend to generalize and enable further progress. Novel results include (a) the first reported solution of the "Collatz Numbers" benchmark, (b) in the problem of fast sorting, the emergence of comb sort, an empirically fast sort; this is a significant milestone in the long-term goal of synthesizing efficient sorting algorithms from low-level primitives, and (c) algorithmic novelty in the form of a record-setting program in the TIS-100 assembly language programming game.

7 Acknowledgments

Thanks to Mark Land and anonymous reviewers for helpful comments.

References

Agapitos, A., and Lucas, S. 2006. Evolving efficient recursive sorting algorithms. In *IEEE CEC*, 2677–2684.

Balog, M.; Gaunt, A.; Brockschmidt, M.; Nowozin, S.; and Tarlow, D. 2017. DeepCoder. In *ICLR*.

Burke, E., and Bykov, Y. 2008. A late acceptance strategy in hill-climbing for exam timetabling problems. In *PATAT*.

Bykov, Y., and Petrovic, S. 2016. A step counting hill climbing algorithm applied to university examination timetabling. *J. Sched.* 19:479–492.

Cai, J.; Shin, R.; and Song, D. 2017. Making neural programming architectures generalize via recursion. In *ICLR*.

Devlin, J.; Uesato, J.; Bhupatiraju, S.; Singh, R.; Mohamed, A.; and Kohli, P. 2017. RobustFill: neural program learning under noise I/O. In *ICML*.

Dobosiewicz, W. 1980. An efficient variation of bubble sort. *Inf Proc Lett* 11:5–6.

Drozdek, A. 2005. Worst case for comb sort. *Informatyka Teoretyczna i Stosowana* 5:23–27.

Feser, J.; Chaudhuri, S.; and Dillig, I. 2015. Synthesizing data structure transformations from input-output examples. In *PLDI*.

Forstenlechner, S.; Fagan, D.; Nicolau, M.; and O'Neill, M. 2017. A grammar design pattern for arbitrary program synthesis problems in GP. In *Evostar*.

Forstenlechner, S.; Fagan, D.; Nicolau, M.; and O'Neill, M. 2018. Towards effective semantic operators for program synthesis in genetic programming. In *GECCO*.

Gaunt, A.; Brockschmidt, M.; Singh, R.; Kushman, N.; Kohli, P.; Taylor, J.; and Tarlow, D. 2016a. Summary – Terpret. *NIPS NAMPI Workshop, arXiv:1612.00817*.

Gaunt, A.; Brockschmidt, M.; Singh, R.; Kushman, N.; Kohli, P.; Taylor, J.; and Tarlow, D. 2016b. Terpret. *arXiv:1608.04428*.

GltyBystndr. 2016. Image test pattern 1:2282/1/7. www.reddit.com/r/tis100/comments/3ab1t7/.

Gulwani, S.; Jha, S.; Tiwari, A.; and Venkatesan, R. 2011. Synthesis of loop-free programs. In *PLDI*.

Gulwani, S.; Polozov, O.; and Singh, R. 2017. Program synthesis. *Foundations & Trends in Programming Languages* 4:1–119.

Gulwani, S. 2011. Automating string processing in spreadsheets using I/O examples. In *PoPL*.

Helmuth, T., and Spector, L. 2015a. General program synthesis training and test examples CSVs. github.com/thelmuth/Program-Synthesis-Benchmark-Data.

Helmuth, T., and Spector, L. 2015b. General program synthesis benchmark suite. In *GECCO*, 1039–1046.

Helmuth, T., and Spector, L. 2015c. Detailed problem descriptions for general program synthesis benchmark suite. web.cs.umass.edu/publication/details.php?id=2387.

Helmuth, T.; McPhee, N.; Pantridge, E.; and Spector, L. 2017. Improving generalization of evolved programs through automatic simplification. In *GECCO*.

Helmuth, T.; McPhee, N.; and Spector, L. 2018. Program synthesis using uniform mutation by addition and deletion. In *GECCO*.

Hofmann, M.; Hirschberger, A.; Kitzelmann, E.; and Schmid, U. 2007. Inductive synthesis of recursive functional programs. In *KI*, 468–472.

Incerpi, J., and Sedgewick, R. 1987. Practical variations of shellsort. *Inf Proc Lett* 26:37–43.

Knuth, D. 1973. *The Art of Computer Programming, vol. 3*. Reading, MA, Addison-Wesley.

Lacey, S., and Box, R. 1991. A fast, easy sort. *BYTE* 4:315.

Lawlor, O. 2012. Instruction encoding & frequency. https://bit.ly/2PPpLFq.

Leaderboard. 2018. www.reddit.com/r/tis100/wiki/index.

Manna, Z., and Waldinger, R. 1980. A deductive approach to program synthesis. In *ACM TOPLAS*.

Manna, Z., and Waldinger, R. 1986. The origin of a binary-search paradigm. SRI Technical Note 351R.

McDermott, J., and Nicolau, M. 2017. Late-acceptance hill-climbing with a grammatical program representation. In *GECCO*.

McDermott, J.; White, D.; Luke, S.; Manzoni, L.; Castelli, M.; Vanneschi, L.; Jaskowski, W.; Krawiec, K.; Harper, R.; De Jong, K.; and O'Reilly, U.-M. 2012. Genetic programming needs better benchmarks. In *GECCO*, 791–798.

Muggleton, S., and De Raedt, L. 1994. Inductive logic programming. *J Logic Prog* 19:629–679.

Pall, M. 2017. Dynasm. luajit.org/dynasm.html.

Pantridge, E.; Helmuth, T.; McPhee, N.; and Spector, L. 2017. On the difficulty of benchmarking inductive program synthesis methods. In *GECCO*.

Perelman, D.; Gulwani, S.; Grossman, D.; and Provost, P. 2014. Test-driven synthesis. In *PLDI*.

Raza, M.; Gulwani, S.; and Milic-Frayling, N. 2014. Programming by example using least general generalizations. In *AAAI*.

Schkufza, E.; Sharma, R.; and Aiken, A. 2016. Stochastic program optimization. *Comm. ACM* 59:114–122.

Sedgewick, R. 1996. Analysis of Shellsort and related algorithms. In *ESA*.

So, S., and Oh, H. 2017. Synthesizing imperative programs for introductory programming assignments. In *Static Analysis Symposium*.

So, S., and Oh, H. 2018. Synthesizing pattern programs from examples. In *IJCAI*.

Solar-Lezama, A. 2008. *Program synthesis by sketching*. Ph.D. Dissertation, UC Berkeley.

Waldinger, R., and Lee, R. 1969. PROW: A step toward automatic program writing. In *IJCAI 1*.

Wolfson, K., and Sipper, M. 2009. Evolving efficient list search algorithms. In *Evolution Artificielle (EA)*.

Yin, P., and Neubig, G. 2017. A syntactic neural model for general-purpose code generation. In *ACL*.

Zachtronics. 2015. TIS-100.