Synthesis of Sorting Kernels

Marcel Ullrich
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Sebastian Hack
Saarland University, Saarland Informatics Campus
Saarbrücken, Germany

Abstract

Recently, AlphaDev has shown significant advances in the synthesis of branchless sorting kernels for arrays of lengths 3 to 5. In this paper, we propose an enumerative search technique based on A* search and present novel optimality-preserving heuristics and non-optimality-preserving cuts for sorting kernel synthesis. Our algorithm outperforms AlphaDev in synthesis time by two orders of magnitude ran on a standard notebook instead of a TPU cluster. Because our algorithm can explore the solution space, we are able to enumerate *all* correct sorting kernels for length 3 and simply select the best-performing one. For larger array lengths, we intelligently sample the solution space and find a sorting kernel that outperforms the state-of-the-art. Furthermore, we establish a new tight lower bound for the shortest sorting kernel for length 4. Finally, we provide a comprehensive comparison against several other existing synthesis techniques and show that none of them is able to synthesize sorting kernels for arrays longer than 3.

ACM Reference Format:
Marcel Ullrich and Sebastian Hack. 2024. Synthesis of Sorting Kernels. In . ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Sorting algorithms run billions of times every day on machines around the globe. Every small improvement in their runtime directly translates into energy savings and performance gains. Therefore, many performance-critical algorithms are still optimized manually by experts on the assembly-code level. However, creating and optimizing such algorithms for a specific hardware architecture is complex and error-prone. In recent years, program synthesis techniques have received increasing attention to automate this process. Last year, AlphaDev [13] showed that reinforcement learning (RL) can be used to synthesize assembly code for kernels that sort a fixed amount of elements. Such kernels typically sort arrays of lengths 3–10 and are invoked as the base case of divide-and-conquer sorting algorithms. AlphaDev synthesized kernels that sort arrays of lengths 3–5 and showed that they outperform the state-of-the-art sorting algorithms for these lengths. However, their technique only discovers a single sorting kernel for a given array size and does not provide any guarantees on the length of the synthesized kernel. Furthermore, their technique requires significant compute resources (several TPU cores) and runs for hours.

In this paper we provide a comprehensive evaluation of sorting kernel synthesis using a variety of existing synthesis techniques. Among them, we investigate constraint programming, stochastic search, mixed-integer programming, counter-example guided inductive synthesis using SMT solvers, and approaches from the planning community. We show that classical synthesis approaches fail to scale beyond arrays of length 3 if they are at all able to synthesize a kernel for that length.

We propose an enumerative search approach for assembly-level synthesis of sorting kernels. To that end, we present several optimality-preserving heuristics and non-optimality-preserving cuts to prune the search space that lead to a synthesis algorithm that outperforms the synthesis time of AlphaDev by two orders of magnitude on a standard notebook. Additionally, in contrast to AlphaDev, our synthesis algorithm is in principle able to enumerate *all* optimal sorting kernels which allows us to determine the fastest sorting kernel for a given microarchitecture by simply benchmarking *all* possible sorting kernels at least for small input sizes. Using our algorithm we find new sorting kernels that outperform the state-of-the-art in terms of runtime. For array length 3, we are able to enumerate all optimal sorting kernels and hence can chose the best performing one. For array length 4, the solution space is too large to enumerate all correct sorting kernels. We devise an intelligent sampling strategy that finds a sorting kernel that outperforms the state-of-the-art. Furthermore, we prove for array length 4 that the shortest sorting kernel has length 20 by exhaustively enumerating the search space of all length 19 kernels and proving that there is no correct kernel in that space. This bound is new and no existing technique was yet able to establish that bound.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In summary, we make the following contributions:

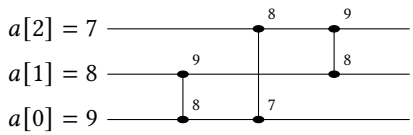
- We model the synthesis problem for sorting kernels in a variety of existing program synthesis techniques (Section 4). We show that none of the existing techniques is able to synthesize sorting kernels for arrays of length ≥ 4.
- We present a novel enumerative search approach and synthesize sorting kernels for arrays of length 3 to 5 (Section 3). Our synthesis algorithm is two orders of magnitude faster than the only other existing approach that is able to scale beyond length 3.
- Our experimental evaluation (Section 5) shows that for $n = 3$, we can enumerate *all* correct kernels and choose the best one simply by running them all. For $n = 4$ we obtain a sorting kernel that outperforms the state-of-the-art by cleverly sampling the solution space. Furthermore, we establish a new tight lower-bound for the shortest sorting kernel for $n = 4$.

2 Sorting Kernels

2.1 Background

The goal of this work is to synthesize sorting kernels (small branchless programs) that sort arrays of *fixed* length. Many algorithms of standard libraries like quicksort or mergesort resort to a highly-optimized sorting kernel for small arrays that they invoke as a base case. Such kernels are *oblivious:* The order of comparison of the individual elements of the array does not depend on the values of the array elements.

A common way to create a sorting kernel of a given length is by implementing a *sorting network*. A sorting network is an arrangement of compare-and-swap operations, often visualized by a graph in which horizontal lines represent values and vertical lines indicate the compare-and-swap operations. The following graph shows a sorting network for an array of length 3 and the flow of values (from left to right) when sorting the array $a = \{9, 8, 7\}$:



To implement a sorting network in software, one uses a specific code pattern for a compare-and-swap and instantiates this code pattern for each compare-and-swap in the network. Two common techniques to implement compare-and-swaps are 1) conditional move or 2) min/max instructions. For example, on x86, a compare-and-swap operation between registers `rax` and `rbx` (`xmm0` and `xmm1`) can be encoded by the following snippets. The left one uses conditional moves and operates on the general purpose register file and the second one uses min/max instrutions and operates on the vector register file.

```
mov   rdi, rax
cmp   rbx, rax
cmovl rax, rbx
cmovl rbx, rdi
```

```
movdqa xmm7, xmm0
pminsd xmm0, xmm1
pmaxsd xmm1, xmm7
```

`rdi` and `xmm7` are temporary registers that are needed to compensate for the fact that x86 is a two-address code machine which means that the instructions destructively update their first operand with the result. The move instructions in both snippets save a register that is overwritten by the first cmovl/pminsd instruction. On an out-of-order machine, these move instructions do not cause computational load in a functional unit because they only influence register renaming. Nevertheless, they consume other resources such as instruction cache footprint and decoding bandwidth.

As it turns out, on two-address code machines like x86 there are sorting kernels that are shorter (in the number of instructions) than the implementation of the *smallest* (in terms of number of compare-and-swaps) sorting network with the above snippets. It is the goal of previous [13] and this work to find such kernels. As an example, consider the above sorting network and its implementation (we assume that the array elements have been loaded into `rax`, `rbx`, `ecx`) shown in the left column:

```
# sorting network
mov   rdi, rax
cmp   rbx, rax
cmovl rax, rbx
cmovl rbx, rdi

mov   rdi, rax
cmp   rcx, rax
cmovl rax, rcx
cmovl rcx, rdi

mov   rdi, rbx
cmp   rcx, rbx
cmovl rbx, rcx
cmovl rcx, rdi
```

```
# synth cmov
mov   rdi, rax
cmp   rcx, rdi
cmovl rdi, rcx
cmovl rcx, rax

cmp   rbx, rcx
mov   rax, rbx
cmovg rbx, rcx
cmovg rcx, rax

cmp   rax, rdi
cmovl rbx, rdi
cmovg rax, rdi
```

```
# synth min/max

movdqa xmm7, xmm1
pminud xmm7, xmm2
pmaxud xmm2, xmm1

movdqa xmm1, xmm2
pminud xmm1, xmm0
pmaxud xmm2, xmm0

pmaxud xmm1, xmm7
pminud xmm0, xmm7
```

The kernel in the middle column has been found by our synthesizer and is one instruction shorter than the sorting network kernel. The first two groups of four instructions are standard compare-and-swap snippets. The last block however is not a compare-and-swap but computes the following functions:

$$\texttt{rbx} = \text{ite}(b > \min(a, c), \min(b, \max(a, c)), \min(a, c))$$
$$\texttt{rax} = \min(b, \min(a, c))$$

where a, b, c are the initial values for `rax`, `rbx`, `rcx` and ite means if-then-else.

The rightmost code is a min/max kernel synthesized with our approach. Similarly, the first two blocks are compare-and-swaps but the last block is not. It implements the functions:

$$\texttt{xmm1} = \max(\min(\max(c, b), a), \min(b, c))$$
$$\texttt{xmm0} = \min(a, \min(b, c))$$

where a, b, c are the initial values for `xmm0`, `xmm1`, `xmm2`. The kernel is one `movdqa` instruction shorter than the sorting net implementation.

Note that removing the final move instruction cannot be achieved by classical compiler optimizations like copy coalescing. It requires semantical reasoning on min/max/ite expressions, for example showing that

$$\min(a, \min(b, c)) = \min(\min(\max(c, b), a), \min(b, c))$$

for the min/max kernel shown above.

2.2 Model

In this paper, we focus on the x86 architecture and conditional move instructions because that is the setting of the related work [13] against which we compare. Nevertheless, the approach we present in this paper is also suitable for synthesizing min/max sorting kernels and we discuss experimental results of min/max kernel synthesis in Section 5.4.

We use the following model of assembly instructions for sorting kernel synthesis: We have registers $r_1, \ldots, r_n$ holding the numbers to be sorted, additional scratch registers $s_1, \ldots, s_m$ for swapping, and flags lt and gt for comparison results. The flag registers are written by the comparison operation and read by the conditional operations. We call the initial assignment of $r_1, \ldots, r_n$ the permutation of our test case. We call the complete assignment of $r_1, \ldots, r_n, s_1, \ldots, s_m$, and the flags the register assignment.

Our sorting kernels consist of the following commands and we call a list of commands a program:

- `mov` r_i, r_j: Move the value of register r_j to register r_i.
- `cmp` r_i, r_j: Compare the values of registers r_i and r_j and set the flags accordingly (e.g. flag lt if $r_i < r_j$).
- `cmovl` r_i, r_j: If the lt flag is set, move the value of register r_j to register r_i.
- `cmovg` r_i, r_j: If the gt flag is set, move the value of register r_j to register r_i.

These commands are consistent with the commands used by related work Mankowitz et al. [13]. The following example shows the execution of a sorting algorithm for $n = 2$.

r_1	r_2	s_1	lt	gt	instruction
2	1	0	—	—	`mov s1 r2`
2	1	1	—	—	`cmp r1 r2`
2	1	1	—	>	`cmovg r2 r1`
2	2	1	—	>	`cmovg r1 s1`
1	2	1	—	>	

Note that we assume that the values to sort are present in registers before the kernel runs and that the sorted values are put into registers $r_1, \ldots, r_n$ by the kernel.

2.3 Correctness

A sorting program is correct if the resulting state o for an initial state r fulfills

$$\underbrace{(\forall 1 \le i < n : o_i \le o_{i+1})}_{\text{ascending}} \wedge \underbrace{\forall x : |\{i \mid o_i = x\}| = |\{i \mid r_i = x\}|}_{\text{same elements}} \quad (1)$$

which means that the output is sorted and a permutation of the input. For the case that the input only consists of a permutation of the numbers $1, \ldots, n$, the output is the sorted list $1, \ldots, n$: $\forall 1 \le i < n : o_i = i$.

Our sorting kernels do not contain constants. Therefore, they cannot discriminate different sets of inputs and only compare the values of a given input array among themselves. This is an important property as it allows us to verify the correctness of the sorting kernel by executing it on all permutations of an arbitrarily chosen input array of n distinct numbers. The concrete numbers do not matter. We choose the numbers $1, \ldots n$ for simplicity. Hence, any algorithm that behaves correctly on the $n!$ permutations of $1, \ldots, n$ is correct for all inputs. Between equal elements, the order does not matter. We can choose any arbitrary order, i.e. we are not restricted to stable sorting kernels. Therefore, we restrict our commands to `cmovl` and `cmovg` and do not consider `cmovle` and `cmovge`.

It is known that under certain conditions, the correctness of sorting algorithms can be verified by only checking the correctness on a subset of the input space of size 2^n which is less than $n!$ for $n > 3$. This principle is the case for sorting networks where the 0–1 sorting lemma* applies [5]. This lemma only applies to programs with compare-and-swap operations and does not apply to instruction sets like ours in which compare and conditional move operations are not a single instruction. We do not consider single-instruction compare-and-swaps because common instruction sets do not provide them. Hence, we cannot use the 0–1 sorting lemma and have to test all $n!$ permutations of the input space.

Complexity. The sorting kernel synthesis problem is hard due to exponential growth in multiple dimensions. The number of possible programs grows exponentially with the program length. The optimal program lengths according to our tests and Mankowitz et al. [13] are 11, 20, 33, and 45 for $n = 3$ to $n = 6$. Simultanously, we have more inputs to check for correctness 6, 24, 120, 720 ($n!$) for $n = 3$ to $n = 6$. Especially the number of different intputs to check grows substantially so that checking all inputs becomes practically infeasible quickly, in our experience for $n \ge 6$.

*The sorting lemma states that the correctness of sorting networks can sufficiently be checked using all combinations of zeros and ones as input.

3 Enumerative Synthesis

In this section, we present a novel enumerative approach to synthesize sorting algorithms. Our approach outperforms existing approaches in synthesis time as well as in the performance of synthesized kernels. As the search space is too large to enumerate all programs (see Section 5.1 for more details), we present several novel heuristics specific to sorting kernel synthesis that improve the performance of our enumerative search substantially. In contrast to related work, we are also able to find multiple if not all solutions and are able to verify the optimality (in terms of program length) of the found solutions.

Our approach is based on Dijkstra/A* search and uses the following **steps** to explore the search space:

1. Select an open state (representing a partial program).
2. Select and add an instruction to the partial program.
3. Check if the partial program is viable to be completed.
4. Check the correctness of the program on the permutation test suite.
5. Prune non-promising states.
6. Check the partial program for equivalence with an already found programs.

The initial state consists of register assignments for each possible permutation of the input $1, \ldots, n$. We then proceed by executing instructions on the state resulting in a successor state of concrete register assignments. A state is final if all register assignments are sorted.

We optimize each of the six steps above and discuss these optimizations in detail in the following subsections. Our motivation behind each optimization is to either rule out parts of the search space that never lead to an optimal solution or to combine equivalent solution classes to reduce the search space. In each case, we preserve at least one representative solution.

3.1 Step 1: Select an Open State

For Dijkstra, we enumerate the states by their program length. The first solution we find is guaranteed to be of minimal length. Furthermore, this approach is parallelizable as we can process all programs of a certain length in parallel to obtain the next length.

For A*, we augment the search using heuristics that guide the search towards the correct solution:

- The number of distinct permutations in the state. Note that a state contains a register assignment for each permutation of $1, \ldots, n$. Hence, the number of distinct register assignments in a state indicates "how much" the array has been sorted already.
- The number of distinct register assignments remaining in the state. This is similar to the point above, but includes the scratch registers.
- The maximum of the instructions needed for any of the register assignment in the state. Before starting the search, we precompute the shortest sorting program for *each* individual permutation of $1, \ldots, n$ *individually*. The length of each such program is a lower bound for the length of the sorting program that would sort any permutation. Hence, if we consider the length of the shortest program for any of the register assignments in the state, we have a lower bound for the remainder of the program we want to synthesize.

3.2 Step 2: Selecting an Instruction

To select an instruction we iterate over all possible instructions. However, we only permit comparisons where the index of the second register is strictly greater than that of the first register. With this constraint, we prevent nonsensical comparisons (where register numbers are equal) as well as utilizing the symmetry resulting from swapping the greater-than and less-than flags.

In our practical evaluation (Section 5), we also evaluated the following non-optimality-preserving heuristic: Alternatively to iterating over all possible instructions, we use our precomputed register assignment search (see Section 3.1) as a guide. We only consider instructions that are part of the precomputed (see Step 1) optimal instruction sequence for any of the register assignments we computed before:

$$\text{Actions} := \bigcup_{\text{assignment } A} \{instr \mid instr \in \text{start of opt. solution}(A)\}$$

3.3 Step 3: Check for Viability

After applying the instruction(s) to the state, we check if the partial program is still viable for a correct solution. A program is not viable for a correct solution if it eliminates at least one of the numbers to be sorted in some register assignment. For instance, `mov r1 r2` in `1 2 3 0` results in `2 2 3 0` erasing the number 1. Hence, this program cannot be completed to a correct one anymore and is therefore not viable.

As we are only searching for solutions of minimal length, we reject a new program if it is longer than a found solution or an initially given length bound. Furthermore, we reject a program if not all individual register assignments can be completed to a sorted state within the remaining budget of instructions. If one assignment cannot be sorted individually in the remaining time, the program cannot be completed to a correct one.

3.4 Step 4: Check for Correctness

We check if all register assignments are sorted.

3.5 Step 5. Cutting Non-Promising States

In this step, we prune the search space by discarding states that are not promising. It is based on the following observation:

The semantical progress of the sort algorithm is the number of distinct remaining permutations (see Step 1). For instance, starting with [1,2,3], [1,3,2], [2,1,3], [2,3,1], the sequence `cmp r2 r3; cmovg r4 r2; cmovg r2 r3; cmovg r3 r4` swaps the content of r_2 and r_3 if $r_2 > r_3$. The result is [1,2,3], [1,2,3], [2,1,3], [2,1,3], reducing the number of distinct permutations from 4 to 2. In that, the program removes one misalignment in the register assignments ensuring $r_2 \leq r_3$. The intuition is that programs that significantly reduce the number of distinct permutations are more likely to be better.

We use the number of distinct permutations as a score to reduce the size of the search space. For each state s we check how the number of distinct permutations relates to the best state s' that we have found for a program length that is one smaller than the one of s. The state s will be discarded if its number of distinct permutations is beyond the number of distinct permutations of s' times a factor k, where k has to be chosen empirically. For example, consider some program length ℓ and assume s' is a state that represents a program of length $\ell - 1$ that has the fewest distinct permutations. We discard each state s of length ℓ if the number of distinct permutations in s is at least k times the number of distinct permutations in s'. This is expressed by the following inequality:

$$k \cdot \min_{s' \in \text{State}_{\ell-1}} \text{perm\_count}\, s' < \text{perm\_count}\, s$$

where perm_count is the number of different permutations in the state and k is a parameter that has to be chosen empirically.

Note that this pruning criterion is not guaranteed to preserve optimality. However, for larger program sizes, too many states are enumerated to be kept in memory and it is necessary to discard states to make the search feasible.

3.6 Step 6: Deduplication of Equivalent Programs

The most important pruning technique is the deduplication of equivalent programs. Two programs are equivalent if they behave the same on all permutations. For instance, `cmp r1 r2; mov r3 r2` is equivalent to `mov r3 r2; cmp r1 r2` and `cmp r1 r2; cmp r2 r3` is equivalent to `cmp r2 r3` as the flags are overwritten. To check for equivalence, we hash the register assignments of a state. To further reduce symmetries, we sort the register assignments of each state in lexicographic order. Additionally, this processing allows us to deduplicate equal assignments.

4 Solver-based Synthesis Techniques

In this section, we discuss several existing synthesis techniques based on constraint solvers (satisfiability modulo theories (SMT), constraint programming (CP), integer linear programming (ILP)) and apply them to the sorting kernel synthesis problem. They serve as a basis for comparison in our evaluation in Section 5. For each technique we focus on the guarantees (minimality and correctness) about the resulting algorithms, formulations of the synthesis problem, and properties of the technique.

Constraint solvers are typically very sensitive to details in the formulation of the problem. Therefore, we investigate multiple goal formulations of the sorting correctness criterion for each technique:

- All resulting permutations need to be equal and each number from 1 to n needs to be in the registers.
- The resulting state needs to be ascending and the registers need to be a permutation of the initial state.
- Similarly, we can specialize to $1, \ldots, n$ and check only that the amount of 0s, 1s, 2s, etc. is the same in the input and output. Even more specialized, we can assert that the amount is exactly 1 for each number.
- We can combine both principles and assert that the output is $1, \ldots, n$ in that order.
- Alternatively to the positive constraints, we can check that each output is between 1 and n and all are different and in ascending order.

We also devised heuristics to rule out non-sensical programs and instructions that will never be part of an optimal solution. Additionally, we employ heuristics to utilize symmetries shrink down the search space:

- Do not perform two consecutive compare operations
- Do not compare a register with itself
- The arguments to a comparison need to be in lexicographic order (otherwise, we could swap the arguments)
- Read from registers in incremental order (allowed due to symmetry between permutations)
- Do not read uninitialized registers
- Do not ultimately erase a value from all registers
- Provide a skeleton to complete a partly program

As we will show in Section 5, some of the heuristics improve the synthesis time while others have negligible or even negative effects.

4.1 Satisfiability Modulo Theories

SMT solvers can be used as a verification oracle to check the correctness of the program or provide counterexamples where the program behaves incorrectly. Hence, an SMT solver can be used in conjunction with a counter example guided synthesis (CEGIS) [11] loop to find correct programs with any other synthesis technique that suggests candidate programs.

Given some input/output examples, the synthesizer generates a candidate program that behaves correctly on the examples. If such a program is found, the SMT solver checks the correctness of the program on all possible inputs. Either the solver confirms the correctness of the program or provides a counterexample for further synthesis iterations.

Our first formulation SMT-Perm directly gives all necessary input/output samples (all permutations). The CEGIS formulation with interactive counterexample generation is called SMT-CEGIS. The technique is similar to the one presented by Gulwani et al. [7].

The advantage of SMT-Perm over SMT-CEGIS is that only a single query is necessary for synthesis. The resulting candidate program is guaranteed to be correct. On the other hand, SMT-CEGIS can be more efficient as the initial queries are much smaller and can find a correct program even without the complete set of input/output examples.

For the SMT approaches, we encode the state as a tuple of an int array with length $n + m$ for the registers and swap registers and two boolean values for the flags. We also tested formulations with enumeration types instead of ints and encoding the flags along in the array.

For each input sample, we have a separate state tuple that is transformed in each step of the program. This transformation resulting from execution of the program instructions is encoded as a relation of a state with its successor state. The program consists of free variables for instruction and its argument for each step that are synthesized. All our instructions have exactly two register arguments in the range $1, \ldots, n + m$.

In addition to the direct SMT encoding akin to CEGIS, we also investigate syntax-guided synthesis (SyGuS) [2] formulations for the synthesis of sorting algorithms in SMT-SyGuS and SMT-Metalift using MetaLift [3]. We encode a single abstract state with arbitrary initial register states and assert the correctness as synthesis constraint.

4.2 Constraint Programming

Similarly to SMT, we test CP for synthesis. Analogously to the SMT approaches, create variables for each permutation and timestep as well as for the instructions.

We first investigate ILP. In general CP, we have no concept of arrays and indexing. Therefore, we encode each register using a separate variable.

We use binary variables for every possible command to indicate whether it is selected in the current time stamp. To only select one command, the sum of all binary variables for a time stamp has to be one. Depending on which command is selected, we restrict which variables in the successor state are equal to the current state.

The most complex transition is the conditional move like `cmovl a b`: The register a in timestep $t + 1$ is set to the value of register b in timestep t if the flag $f_<$ is set.

$$a_{t+1} = \underbrace{b_t \cdot \text{instr}_{t,\text{cmovl},a,b} \cdot f_<}_{\text{change}} + \underbrace{a_t \cdot \text{instr}_{t,\text{cmovl},a,b} \cdot (1 - f_<)}_{\text{flag not set}} + \underbrace{a_t \cdot (1 - \text{instr}_{t,\text{cmovl},a,b})}_{\text{no instruction}}$$

Conceptually, we use multiplications as logical conjunctions and additions as logical disjunctions. By multiplying, we can set the correct value in the appropriate cases. If the flag $f_<$ is set, the second term is zero and the first term is the value of b. If the flag is not set, the first term is zero and the second term is the value of a which remains unchanged. If another instruction is selected, the third term is the value of a and the other terms are zero.

In our formula above, we have cubic constraints as we multiply three variables. For conditional moves, we introduce activated commands as binary variables that are true if the flag is set and the command is selected: $\text{active\_cmovl}_{t,a,b} = \text{instr}_{t,\text{cmovl},a,b} \cdot f_<$. Using this indirection, we only have quadratic constraints where at least one variable is binary. We identify this approach using Gurobi [8] as CP-Gurobi.

In ILP solvers that do not support these special quadratic constraints, we introduce additional variables for the result of such a multiplication and couple the result via a big-M[†] constraint. We identify this approach as CP-ILP.

In addition to dedicated ILP solvers, we also test general **CP** solvers in the CP-Minizinc approach. We use the MiniZinc [16] language to formulate the problem. In that, we can use the same constraints as in the SMT formulation.

We test the use of a partial test suite as input to the synthesis process. However, prohibitively many wrong programs are generated. We test generating all programs and filtering them using the full permutation test suite in CP-Minizinc-Filter.

5 Evaluation

In this section, we experimentally evaluate our enumerative approach presented in Section 3. First, we investigate the structure of the search space to the solver-based techniques presented in Section 4, and two more techniques from the related work: AlphaDev [13] and Stoke [19] with respect to synthesis time and the performance of the generated sorting algorithms.

5.1 Search Space Structure

In the following paragraphs, we discuss the structure of the search and solution space of our enumerative approach. When sorting n elements and using m scratch registers, there are

$$\Big(\underbrace{4}_{\text{cmds}} \cdot \underbrace{(n+m)^2}_{\text{regs}} \Big)^{\ell}$$

possible programs of length ℓ that have to be considered in the search. The following table shows that the search space is prohibitively large for exhaustive enumeration.

[†]See https://en.wikipedia.org/wiki/Big_M_method.

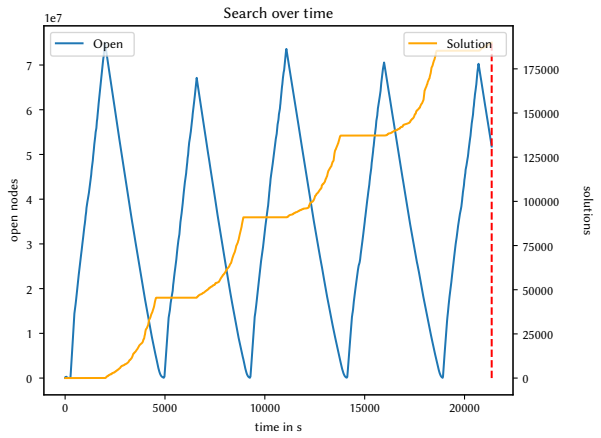


Figure 1. Solutions and open states for $n = 4$ over time with a cut of ·1. The first 10% of the search is shown.

n	$n!$	Optimal Size	Program Space
3	6	11	$\approx 10^{19.9}$
4	24	20	$\approx 10^{40.0}$
5	120	≈ 33	$\approx 10^{71.2}$
6‡	720	≈ 45	$\approx 10^{108.4}$

In practice, we enumerate a much smaller portion of $7 \cdot 10^3$, $7 \cdot 10^4$, and $6 \cdot 10^6$ states for $n = 3$, 4, and 5 respectively using our enumerative approach. For comparison, AlphaDev enumerates $4 \cdot 10^5$, $1 \cdot 10^6$, and $6 \cdot 10^6$ states.

Figure 1 shows how our enumerative algorithm explores the search space for $n = 4$. We set $k = 1$ for the cut heuristic presented in Section 3.5 which is the most aggressive setting in that it also might prune optimal solutions. Note that $k = 1$ means that we do not allow for a program of length $\ell + 1$ that has more distinct permutations than then program of length ℓ that has the least number of distinct permutations. The x-axis shows time and the blue line shows the number of open states in our search and the orange line the number of found optimal solutions over time. This figure shows that our heuristics are able to guide the search quickly towards optimal solutions. Once a given direction is taken in the search, the algorithm quickly finds all optimal solutions in that region, closes non-optimal states, and then explores a new region.

We further investigate different choices of k in the cut heuristic of Section 3.5 and visualize the solutions for $n = 3$ in Figure 2 using t-distributed stochastic neighbor embedding (tSNE). tSNE is a technique to reduce the dimensionality of the data while preserving the structure and is frequently used for visualizing high-dimensional data.

Cuts with $k = 2$ (marked in orange) preserve all 5602 solutions while speeding up the synthesis significantly as discussed in Section 5.2. Lower values of $k = 1.5$ (green) and $k = 1$ (red), cut down the search space further (potentially

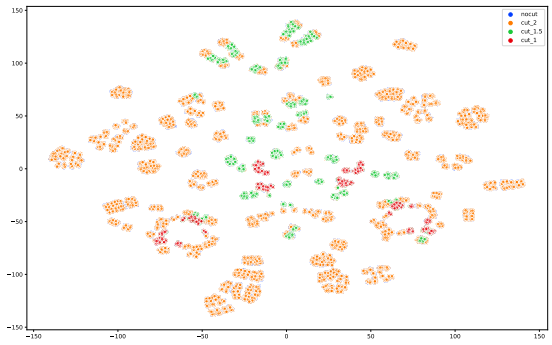


Figure 2. tSNE visualization of the solutions for $n = 3$. Blue are all solution without applying any cut (5602 solutions; hidden by orange because $k = 2$ preserves all optimal solutions). Orange are the solution applying our cut heuristic with a constant of 2 (5602 solutions). Green are the solutions applying our cut heuristic with a constant of 1.5 (838 solutions). Red are the solutions applying our cut heuristic with a constant of 1 (222 solutions). For better visual impression, nodes are slightly perturbed to avoid overlap.

pruning optimal solutions) resulting in 838 and 222 solutions respectively.

Despite the amount of solutions, we observe little variety between different solutions. For $n = 3$, out of 5602 solutions only 23 distinct command combinations are used, i.e. many programs are actually equivalent modulo the order of the instructions and renaming of registers.

For $n = 4$, we explore the *entire* solution space with $k = 1$ which took more than a week on a standard notebook. We enumerated $4.9 \cdot 10^9$ programs out of which 2233360 are optimal solutions, we skip $16.8 \cdot 10^9$ semantically identical programs. Out of these solutions, only 63 are distinct regarding their command combination. We verified that the solutions are indeed optimal (their length is 20) by validating that there is no solution of length 19 by exhaustive search ($k = \infty$) which took two weeks.

5.2 Synthesis Time

In this section, we compare our enumerative approach to the solver-based techniques presented in Section 4, and two more techniques from the related work: AlphaDev [13] and Stoke [19] with respect to synthesis time.

If not otherwise stated, all our experiments ran on a standard notebook with 16 AMD Ryzen 7 5800X 8-core processors with 4850MHz and 32GB of RAM. We conduct our GPU experiments using a NVIDIA GeForce RTX 3070 Ti (8192MB VRAM). For each test, we set the time out to 5 hours (600min or 18000s). For tests under 15 minutes, we took the average

over 5 runs, for longer tests, we only sampled one execution time.

Because the code of AlphaDev is not publicly available, we can not benchmark reproduce their benchmarks and have to refer to the results they report [13]. They used a tensor processing unit (TPU) v.3 with 16 TPU cores with a batch size of 1024 per core for training. For the actor, they used a TPU v.4 with 512 actors.

For discovering the branchless synthesis algorithms, they use two techniques: AlphaDev-RL uses Monte Carlo tree search (MCTS) with an RL agent to find the best program. AlphaDev-S is a stochastic superoptimization approach similar to Stoke [19] run in a warm start mode meaning that the search starts with a given program and tries to optimize it.

In the best configuration (see below for a more in-depth discussion), our enumerative approach is able to synthesize the $n = 5$ case in a reasonable time and outperforms AlphaDev-RL by approximately two orders of magnitude:

Time	$n = 3$	$n = 4$	$n = 5$
Enum, best	97 ms	2443 ms	11 min
AlphaDev-RL	6 min	30 min	≈1050 min
AlphaDev-S	0.4 s	0.6 s	≈345 min

In the remainder of this section, we state the times for $n = 3$. We tried each approach that found a solution for $n = 3$ (for instance CP) again for $n = 4$. For $n = 4$ we set the time out to a week (10080min or 604800s) and moved to a larger cluster with 1 TB of RAM. However, none of the other contenders was able to synthesize a program for $n = 4$.

SMT-based Techniques. The best-performing approach is the CEGIS formulation in which we restrict the counterexamples to be permutations of $1, \ldots, n$. The second best is the approach without counterexample loop in which we synthesize based on all permutations. We also tested the different goal formulations and heuristics mentioned in the beginning of Section 4 but none of them had a significant influence. Neither SyGuS nor Metalift are able to synthesize a program for $n = 3$. We provide the runtime for $n = 3$:

Approach	Time	Note
SMT-Perm	44 min	z3
SMT-CEGIS	97 min	z3, arbitrary inputs
SMT-CEGIS	25 min	z3, inputs in range $1, \ldots, n$
SMT-SyGuS	—	cvc5
SMT-Metalift	—	

Contraint Programming. None of the linear programming formulations was able to synthesize a program for $n = 3$. Only MiniZinc with the Chuffed solver [4] was able to solve the $n = 3$ case.

Approach	Time	Note
CP-Gurobi	—	with quadratic constraints
CP-ILP mip	—	gurobi
CP-ILP mip	—	cbc
CP-Minizinc	—	gurobi/cplex/couenne/ coin-bc/gecode/or tools
CP-Minizinc	874 ms	chuffed solver
CP-Minizinc-Filter	—[§]	partial input test suite

For the successful MiniZinc/Chuffed combination, we investigate the impact of different heuristics and goal formulations that we describe in Section 4 below. $= 123$ means that the goal requires numbers in that order in the output. $\leq$ means a goal that enforces an ascending order of the output. $\#a_1a_2\ldots$ means that the goal enforces that the value a_i occurs as often in the input as in the output. Note that $\leq$ and #123 are equivalent to $= 123$.

Goal formulation	Heuristic	Time
$= 123$	—	247 s
$\leq$, #0123	—	232 s
$\leq$, #0123	(I) := no consecutive compares	10 s
$\leq$, #0123	(II) := symmetry for compares	68 s
$\leq$, #0123	(I) + (II)	874 ms
$= 123$	(I) + (II)	70 s
$\leq$, #0123, $= 123$	(I) + (II)	11 9s
$\leq$, #123	(I) + (II)	30 s
$\leq$, #0123	(I) + (II), cmd[1]=Cmp	64 s
$\leq$, #0123	(I) + (II), #123 ≥ 1[¶]	52 s
$\leq$, #0123	(I) + (II), only read initialized	54 s

The goal formulation $\leq$, #0123 gives the fastest synthesis time by far. Surprisingly, adding 0 to the goal constraint is more efficient than #123 even though 0 does never appear as an input (note that the constraint forces the same amount of occurrences in input and output). We found that providing "too much" information slows down the synthesis process as can be seen in the $\leq$, #0123, $= 123$ case.

Using heuristics that eliminate consecutive compare instructions and removing symmetries for compare instructions speeds up the synthesis process but surprisingly, additional heuristics do not improve the synthesis time. Even a partially given program (e.g., fixing the first instruction as a comparison) does not improve the synthesis time.

While finding one solution with our CP-Minizinc approach takes roughly a second, we are able to enumerate all possible solutions (33612) in 154 minutes. Using symmetries and only enumerating programs producing the sorted output in ascending order, we reduce the solution space to 5602 programs and the synthesis time to 13 minutes.

We note that the solver makes a significant difference. While the underlying techniques are similar between solvers, only the Chuffed solver is able to synthesize the program for $n = 3$. All other solvers do not terminate even if given ample time (a week) for the computation. As with the SMT approaches, none of the CP approaches produces a solution for $n = 4$.

When we relax the goal formulation to only a subset of all permutations we are able to synthesize a program for $n = 4$ in 23 minutes. However, this algorithm is not guaranteed to be correct for all inputs and all solutions that were provided until the timeout were incorrect. Hence, synthesizing with a relaxed goal formulation is not practical.

Stochastic Search. We use Stoke [19] for stochastic search. Stoke has two modes, cold and warm start. Cold start synthesizes a program from scratch where as warm start takes an existing program and tries to optimize it. Stoke uses test cases as correctness oracle. We tested all permutations and 1000 randomly drawn subsets of all permutations. For warm start, we tried different initial programs implementing sorting networks and different hand-written implementations of a sorting algorithm that sorts three elements.

Stoke is unable to synthesize a correct program for $n = 3$. Also the warm-start optimization mode fails to find a program of optimal length for the $n = 3$ case.

Approach	Time	Note
Stoke-Cold	—	permutation test suite
Stoke-Cold	—	random test suite
Stoke-Warm	—	manual sorting network start
Stoke-Warm	—	indexed sorting start
Stoke-Warm	—	branching sorting start

Planning. To widen the scope of investigated synthesis techniques, we also formulate our synthesis problem as a planning problem in the planning domain definition language (PDDL). In this setting, a plan is a sequence of instructions that sorts the input. As with the SMT and CP formulations, we encode each possible permutation and transform them in tandem with the program execution.

Each instruction is an action in the planning domain. We use a formulation Plan-Parallel with conditional effects as well a linearized formulation Plan-Seq that handles each possible permutation one after another. We test the planners fast-downward [9], Scorpion [21], Lama [18], and CPDDL [6].

fast-downward allows us to test multiple variations of standard heuristics from the planning community including pattern databases, landmarks, landmark cuts, and the use of the FF heuristic. However, we are unable to synthesize a program for $n = 3$ with fast-downward.

As with the other approaches, we are unable to extend the approach to $n = 4$.

Approach	Time	Note
Plan-Parallel	—	
Plan-Seq	679 s	Scorpion
Plan-Seq	398 s	CPDDL (symbolic search)
Plan-Seq	3.54 s	Lama
Plan-Seq	3.86 s	Lama, no negative precond.
Plan-Seq	216 s	Lama, grounded actions

Enumerative Approach. Finally, we evaluate our enumerative approach in a variety of settings. We present the effects of individual optimizations presented in Section 3 in isolation and then look at the combination of the best optimizations.

Approach	Time	Note
Enum	56 s	dijkstra, single core
Enum	17 s	dijkstra, parallel
Enum	46 s	dijkstra, gpu
Enum	219 s	(I) := A*, deduplication, no heuristic
Enum	1713 ms	(I) + permutation count
Enum	2582 ms	(I) + register assignment count
Enum	7176 ms	(I) + assignment instructions needed
Enum	37 s	(I) + cut with 2
Enum	3221 ms	(I) + cut with 1.5
Enum	325 ms	(I) + cut with 1
Enum	16 s	(I) + cut with +2
Enum	90 s	(I) + assignment optimal instructions
Enum	8646 ms	(I) + assignment viability check
Enum	690 ms	(II) := (I) + permutation count, optimal instructions, assignment viability check
Enum	97 ms	(III) := (II) + cut 1

Using combinations of optimizations, we observe that each optimization is mostly independent of the others, which means that adding an optimization to another one has a positive impact on the synthesis time.

We observe that the permutation count is the best search heuristic of the ones described in Section 3.1. The best result is using this search heuristic in combination with checking for viable assignments (Section 3.3) and the non-optimality-preserving cut described in Section 3.2. The largest improvement comes from the cut heuristic (Section 3.5). Therefore, we investigate the cut heuristic for different values of k in the context of the set of best optimizations we have identified above. To estimate how much of the solution space is cut, we give the count of solutions remaining for $n = 3$:

k	Time for $n = 3$	Time for $n = 4$	Sol. rem. $n = 3$
1	97 ms	2443 ms	222
1.5	215 ms	82 s	838
2	629 ms	763 s	5602
3	631 ms	—	5602
4	623 ms	—	5602

For $k \geq 2$ we see that all solutions are preserved. However, we want to choose k as small as possible to cut down the search space maximally.

We apply our best enumerative approach (III) to the $n = 4$ and $n = 5$ case:

Time	$n = 3$	$n = 4$	$n = 5$
Enum, best	97 ms	2443 ms	11 min

Given our results, we conclude that with domain knowledge, even the complicated $n = 5$ case becomes feasible to synthesize on a standard laptop in a reasonable time. Although the search space possesses a structure that allows for heuristics and cuts, classical techniques are unable to utilize this structure efficiently and are not able to find solutions for $n > 3$.

5.3 Evaluation of Sorting Algorithms

Using our enumerative approach, we are able to synthesize multiple, given sufficient time, all optimal sorting algorithms. We compare our results to the algorithms by Mankowitz et al. [13], Neri [15], Mimicry [14], and additional ones, that we wrote ourselves in Assembly, C, C++, and Rust. We embed the assembly algorithms using inline assembly and link the other algorithms via C's application binary interface (ABI).

As mentioned, we do not synthesize the load and store instructions from the memory to the registers and back. These instructions are always necessary and only their placement in the algorithm is up to preference. Therefore, our assembly algorithms are correspondingly shorter than the ones discussed by Mankowitz et al. [13].

To evaluate the performance of the sorting algorithms, we use random test cases with length n as well embedding the algorithms into quicksort and mergesort. We use the Google benchmark library[‖] for the benchmarking. In the following tests, we use `clang` version 17.0.6 in the release configuration and compare sorting algorithms for $n = 3$. For each contestant (cassioneri, alphadev, mimicry, enumeration), we take the best algorithm.

Additionally, we benchmark handwritten C++ and Rust implementations. The `default` algorithm uses three conditionals and a temporary variable to swap contents of the memory buffer. The `branchless` algorithm uses index arithmetic with comparisons to write the smallest, middle, and largest value to the memory buffer. The `swap` algorithm operates the same as `default` but uses local variables and `std::swap` to swap the values. The `std` algorithm uses the standard library's `std::sort` function. For each algorithm, we give the time taken by the benchmark and its rank among all tested algorithms.

We first compare the sorting algorithms in a standalone setting. Each algorithm is tested with random test cases of length 3 with values between −10000 and 10000. We benchmark over multiple iterations over the full test suite and average the resulting times. We count how often each instruction is used for each program. This count includes the move instructions between the memory and registers.

Algorithm	Time	Rank	Cmp	Mov	CMov	Other
enum	5.8 ms	1	3	8	6	-
$\text{enum}_{\text{worst}}$	10.9 ms	5614	3	6	8	-
cassioneri	7.1 ms	4483	3	9	6	-
mimicry	8.0 ms	5569	3	6	-	8
alphadev	6.7 ms	1494	3	8	6	-
branchless	7.1 ms	3861	3	7	-	12
default	27.7 ms	5618	4	13	-	6
swap	6.6 ms	928	3	9	6	-
std	29.0 ms	5621				

[‖] https://github.com/google/benchmark

The `swap` approach is one of the best approaches only beaten by the `enum` approach. Additionally, most handwritten approaches are slower than the slowest `enum` approach.

To compare the algorithm in a natural way, we embed them into a quicksort and mergesort algorithm indicated by subscript Q and M respectively. The input is recursively sorted until three elements remain onto which the sorting algorithm is applied. We use lists of random length up to 20000 elements. The instruction counts differ between the standalone run and the embedded run as we always select the best algorithm for each category.

Algorithm	Time_Q	Rank_Q	Cmp	Mov	CMov	Other
enum	759 ms	1	3	8	6	-
$\text{enum}_{\text{worst}}$	982 ms	5623	3	7	7	-
cassioneri	776 ms	546	3	6	-	6
mimicry	782 ms	845	-	-	-	9
alphadev	787 ms	1129	3	8	6	-
branchless	786 ms	1090	3	7	-	12
default	829 ms	4833	4	13	-	6
swap	778 ms	633	3	9	6	-
std	810 ms	3810				

Algorithm	Time_M	Rank_M	Cmp	Mov	CMov	Other
enum	1223 ms	2	3	8	6	-
$\text{enum}_{\text{worst}}$	1595 ms	5623	3	7	7	-
cassioneri	1220 ms	1	3	6	-	6
mimicry	1327 ms	4202	3	6	-	8
alphadev	1297 ms	2113	3	8	6	-
branchless	1301 ms	2394	3	7	-	12
default	1362 ms	5212	4	13	-	6
swap	1318 ms	3667	5	14	-	6
std	1359 ms	5160				

In the quicksort test, other approaches like `cassioneri` and `mimicry` perform significantly better than in the standalone test. Many of our manual sorting approaches are competitive with the synthesized approaches.

For our mergesort benchmark, the `cassioneri` approach is slightly faster than our best enumerative approach. However, the difference is not significant. Most other manual and synthesized approaches are measurably slower than the `enum` approach.

Our enumerative approach is competitive with the best synthesized and handwritten algorithms and in most benchmarks beats state-of-the-art approaches.

The `alphadev` approach is competitive and is only slightly slower than our enumerative approach. Our handwritten `swap` approach is also well-optimized by the compiler resulting in competitive performance.

For $n = 4$, even our cut solution set is too large to measure the runtime of all 2233362 solutions. Therefore, we sample a subset of 4000 solutions and evaluate them. To make sure we get good candidates, we assign a score to each program in the solution space in the following way: We compute the critical path length, and a score based on the used instructions. We weigh `mov` with 1, `cmp` with 2, and conditional moves with

4. The scores of our solutions are $\{55, 58, 61, 64, 67, 70\}$. We sample 2000 solutions with score 55 and 2000 with score 58. We validated for programs with $n = 3$ that this scoring scheme picks good programs.

We evaluate the sorting algorithms for $n = 4$ in a standalone setting (subscript S) and embedded in a quicksort algorithm (subscript Q). Note that Neri [15] does not provide a `cassioneri` algorithm for $n = 4$.

Algorithm	Time_S	Rank_S	Time_Q	Rank_Q
enum	9.4 ms	2	800 ms	1
$\text{enum}_{\text{worst}}$	12.1 ms	4004	1033 ms	4009
mimicry	8.8 ms	1	826 ms	1877
alphadev	10.4 ms	2549	822 ms	1282
branchless	14.9 ms	4007	824 ms	1609
default	14.8 ms	4006	861 ms	3822
swap	10.2 ms	1977	827 ms	2084
std	16.7 ms	4009	854 ms	3705

Similar to $n = 3$, the swap approach is the best manual approach for the standalone benchmark. Our enumerative approach leads the benchmark only beaten by the vectorized `mimicry` approach which is slightly faster in the standalone benchmark.

For $n = 5$, we evaluate the program presented by `alphadev` and sample kernels generated by our approach:

Algorithm	Time
enum	14.84 ms
$\text{enum}_{\text{worst}}$	17.77 ms
alphadev	16.20 ms

For `alphadev`, we test the version they provide as well as a version where we reorder all memory move instructions to the beginning and end. In the standalone tests, our reordered version is faster than the original version.

As we enumerate the search space completely, we are able to confirm the optimality of our solution. For $n = 3$, we enumerate all 5602 solutions of length 11. For $n = 4$, we enumerate the search space for length 19 without finding a solution. Therefore, we confirm that our solutions of length 20 are optimal for $n = 4$. This bound is new and has not been reported in the literature before. Note that the length-20 solution for $n = 4$ is identical to using 5 swaps which is the standard implementation of an optimal sorting network for $n = 4$.

5.4 Min/Max Kernels

In addition to the conditional move setting presented by [13], we also investigate min/max kernels as mentioned in Section 2.1. We have also investigated "hybrid" kernels that use cmovs and min/max instructions at the same time. However, since min/max instructions work on the vector register file and cmovs on the general purpose registers, such kernels require additional instructions that transfer the values between both register files which makes them not competitive.

In a min/max kernel, a compare-and-swap can be implemented using three instructions compared to four for the cmov case. This results in 9, 15, 27 for a straight-forward implementation of a minimal-size sorting network for sizes $n = 3, 4, 5$. Our approach synthesizes min/max kernels of the following sizes, synthesis time, and speed:

			Runtime		
n	# Instr	Synthesis	min/max	cmov	network
3	8	3.8 ms	4.57 ms	5.80 ms	5.29 ms
4	15	70.5 ms	7.00 ms	9.48 ms	8.12 ms
5	26	32.5 s	10.66 ms	14.84 ms	12.23 ms

The last column shows the runtime of the synthesized min/-max kernel, compared to the runtime of the best cmov kernel, and to the runtime of the min/max implementation of an optimal sorting network. A detailed analysis using uiCA [1] of the best synthesized kernel also showed that the synthesized code has a better dependence structure that allows for higher instruction-level parallelism (and therefore throughput) than the sorting network implementation.

For $n = 3$ and $n = 4$, we verified the minimality of the solutions with respect to the program length. For $n = 3$, our CP approach generates a solution in 15.8s while our SMT approach takes 10s. Neither can generate a solution for $n = 4$.

6 Related Work

Gulwani et al. [7] use SMT-based synthesis to generate programs in a CEGIS loop. Similar to our restricted set of instructions, they use library of components that are employed for the synthesis. The SMT solver enable the synthesis of programs involving constants that can not be exhaustively enumerated. However, this capability is not necessary for our problem as sorting is a purely structural problem.

Alur et al. [2] introduce the SyGuS framework for syntax-guided synthesis. In their work, they evaluate multiple classical synthesis techniques including SMT-based CEGIS synthesis, enumerative learning, constraint-based learning, and stochastic search. Their focus are functional programs according to a fixed grammar. Their techniques are similar to ours.

In their enumerative approach, they deduplicate expressions that behave the same on the current input-output examples. This deduplication corresponds to our handling of states. Each state represents the execution of the partial program under consideration. If two programs behave the same, we only keep the better one. In their experiments, Alur et al. [2] conclude that the enumerative solver is the often most effective. Furthermore, they also note that the formulation of the synthesis problem can have a significant impact on the performance of the synthesis algorithm.

Mankowitz et al. [13] introduce AlphaDev, a system for synthesizing programs using reinforcement learning via MCTS. Their approach is based on AlphaZero [22]. The RL approach is able to solve much larger problems than the classical synthesis techniques. However, the approach is not

guaranteed to find an optimal solution. The reinforcement-learning approach of Mankowitz et al. [13] is also not fully autonomous and applies the following human-provided heuristics. In comparison to their approach, we directly start with the values inside the registers and end our algorithm with the sorted registers avoiding the need for heuristics about memory access. The other heuristics Mankowitz et al. [13] uses are already covered by our equivalence checks and symmetries.

Mimicry [14] employ shuffle vectors to sort the data. They use single instruction, multiple data (SIMD) instructions to transform the data reducing the overall needed computation cycles. In their tests, they improve upon the speedup of AlphaDev.

Mankowitz et al. [13] claim that they exhaustively checked for 3 days that their solution is minimal for $n = 3$. We validate their claim and extended the claim to $n = 4$ by more efficiently enumerating the search space pruning programs that are guaranteed to be suboptimal or incorrect as discussed in Section 5.3. Our search is restricted to `mov`, `cmp`, and conditional move instructions. We presume Mankowitz et al. [13] used the same set of instructions that are also explained in the appendix of their paper.

Neri [15] challenges the claim of Mankowitz et al. [13] that their solution is minimal. They present shorter solutions for $n = 3$. One of their algorithms uses loops. The other uses more complex instructions like `movsb`, `adc`, and `sbb`. Furthermore, they propose an algorithm that is faster in their tests than the one by Mankowitz et al. [13].

Recent research on synthesis techniques focuses on the combination of traditional search techniques like MCTS and A* with neural networks to guide the search via heuristics.

Neural networks are able to learn from the encountered states and discover heuristics and properties on their own. Especially large language models (LLMs) have shown to be able to learn complex patterns and properties showing logic reasoning capabilities.

Parsert and Polgreen [17] present a reinforcement learning approach for SyGuS synthesis using MCTS and gradient-boosted trees for the machine learning part.

Li et al. [12] present a LLM based approach for SyGuS synthesis. They investigate the use of probability grammars and interactive queries to the LLM to guide the synthesis process.

Lehnert et al. [10] present a search approach for planning problems using LLMs and A* search. The LLM is used to learn the pattern of the search trace and find shortcuts to speed up the search.

Planning is most commonly applied to concrete instances of a problem of the form ∃path. The synthesis problem is more abstract and can be formulated as ∃ program : ∀ inputs : correct. The synthesis problem in planning is called generalized planning. Segovia-Aguas et al. [20] investigates the heuristics for generalized planning.

7 Conclusions

In this paper, we showed an enumerative search technique based on A* search and presented novel optimality-preserving heuristics and non-optimality-preserving cuts for sorting kernel synthesis. We showed that our heuristics and cuts significantly reduce the search space and allow us to synthesize sorting kernels for arrays of lengths 3 to 5. In our evaluation we compared against a wide range of existing program synthesis approaches using different techniques and showed that none of them was able to synthesize a sorting kernel of length greater than 3. The only other approach that scales to lengths beyond 3 is AlphaDev [13] which uses reinforcement learning and Monte-Carlo tree search. We outperform AlphaDev in synthesis time by two orders of magnitude running on a standard notebook.

Approaches like AlphaDev have demonstrated the potential of RL for program synthesis by scaling to larger instances. However, they still require human knowledge to guide the search, lack guarantees about the quality of the found solutions, and require significant compute resources. We have shown that simple classical search techniques can be enhanced with domain-knowledge to close the gap to reinforcement learning techniques in terms of scaling while still providing guarantees and resource efficiency.

References

[1] Andreas Abel and Jan Reineke. 2022. uiCA: Accurate throughput prediction of basic blocks on recent Intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–14.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. *Syntax-guided synthesis*. IEEE.

[3] Shadaj Laddad Alvin Cheung, Sahil Bhatia. 2023. MetaLift – A program synthesis framework for verified lifting applications. https://github.com/metalift/metalift.

[4] Geoffrey Chu, Andreas Schutt Peter Stuckey, Gaeme Gange Thorsten Ehlers, and Kathryn Francis. 2023. Chuffed, a lazy clause generation solver. https://github.com/chuffed/chuffed.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 1990. 33.3: Finding the convex hull. *Introduction to Algorithms* (1990), 955–956.

[6] Daniel Fiser. 2024. CPDDL. https://gitlab.com/danfis/cpddl.

[7] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of loop-free programs. *ACM SIGPLAN Notices* 46, 6 (2011), 62–73.

[8] Gurobi Optimization, LLC. 2023. Gurobi Optimizer Reference Manual. https://www.gurobi.com

[9] Malte Helmert. 2006. The fast downward planning system. *Journal of Artificial Intelligence Research* 26 (2006), 191–246.

[10] Lucas Lehnert, Sainbayar Sukhbaatar, Paul Mcvay, Michael Rabbat, and Yuandong Tian. 2024. Beyond A*: Better Planning with Transformers via Search Dynamics Bootstrapping. *arXiv preprint arXiv:2402.14083* (2024).

[11] A Solar Lezama. 2008. *Program synthesis by sketching*. Ph. D. Dissertation. Citeseer.

[12] Yixuan Li, Julian Parsert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. *arXiv*

preprint arXiv:2403.03997 (2024).

[13] Daniel J Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, et al. 2023. Faster sorting algorithms discovered using deep reinforcement learning. *Nature* 618, 7964 (2023), 257–263.

[14] Mimicry. 2023. Faster Sorting Beyond DeepMind's AlphaDev. https://www.mimicry.ai/faster-sorting-beyond-deepminds-alphadev Accessed: 2023-09-20.

[15] Cassio Neri. 2023. Shorter and faster than Sort3AlphaDev. *arXiv preprint arXiv:2307.14503* (2023).

[16] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. 2007. MiniZinc: towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming* (Providence, RI, USA) *(CP'07)*. Springer-Verlag, Berlin, Heidelberg, 529–543.

[17] Julian Parsert and Elizabeth Polgreen. 2024. Reinforcement Learning and Data-Generation for Syntax-Guided Synthesis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 10670–10678.

[18] Silvia Richter and Matthias Westphal. 2010. The LAMA planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research* 39 (2010), 127–177.

[19] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* 41, 1 (2013), 305–316.

[20] Javier Segovia-Aguas, Sergio Jiménez, and Anders Jonsson. 2021. Generalized planning as heuristic search. In *Proceedings of the International Conference on Automated Planning and Scheduling*, Vol. 31. 569–577.

[21] Jendrik Seipp. 2023. Scorpion 2023. *Tenth International Planning Competition (IPC-10): Planner Abstracts* (2023).

[22] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.