6.890 Spring 2019 Project Proposal
LHYRA: Learned HYbrid Recursive Algorithms

Josh Gruenstein Lior Hirschfeld Ben Spector

March 19, 2019

Introduction

We propose a framework for automatically learning recursive algorithms and data structures for arbitrary data distributions and optimization criteria. Our ideas are loosely inspired by Prof. Kraskas work on learned B-Trees and the Recursive Model Index (RMI); we conceive of this project as a sort of generalization of this idea to most recursive data-structures and algorithms.

We observe that there are diverse problems in computer science that can be solved by recursive structures[1], from sorting to hierarchical pathfinding for online applications. Often though, these solutions are operating on a distribution which is very limited compared to the overall space of solutions, or the solutions are heuristics because finding perfect solutions is either intractable or impossible due to imperfect information. With such problems, there is usually some set of possible algorithms which produce solutions, but it is rare that a single solver is optimal for all possible distributions and objective functions: for example, insertion sort is preferable to merge sort in memory, or in time for nearly-sorted lists, but inferior in time for the random case.

Practically, what usually happens is that an expert programmer writes a number of these algorithms, benchmarks them against each other, and chooses the one which appears to perform the best. A clear example of this is in the C++ STL, in which the standard algorithm is Introsort, which calls Quicksort until hitting a certain recursion depth, where it switches to Heapsort. We observe that it would be better that these recursive structures would adapt to their distributions and objective functions, so that not only could they lessen the workload for the expert, but also better explore the potential synergies of the possible algorithms.

We therefore propose the Learned HYbrid Recursive Algorithms framework, or LHYRA. We give a more precise proposal below, but the essential idea is to automate the search of these learned hybrid models given data representing a distribution, candidate solvers, and a cost function. LHYRA will automatically search through the possible candidates (and potentially their hyperparameters, too) in order to determine the optimal algorithm for the task at hand.

Framework

A LHYRA instance is described by the following components:

[1]Not all algorithms used in our framework need be recursive; in some cases it may be best to solve a large base-case directly (for example, using insertion-sort for small nearly-sorted lists), and we recommend providing all the options to LHYRA and letting it sort out everything.

- **Bag of solvers S.** Each solver $s_i \in S$ is equipped to solve an instance of the problem at hand, potentially by calling the LHYRA instance on subproblems. Each solver s_i may also have a set of hyperparameters h_{ij}.
- **Data Generator / Training Data Set D.** In the training phase the problems in this set / generated by this generator are used by the optimizer to learn how to select a solver s_i from S given problem features.
- **Feature extractor $F(x)$.** Our goal is to learn how to select the ideal solver based on a given input, thus we need a nice way of extracting useful features about an input. For example, a feature extractor for a sorting LHYRA instance may return a vector containing the length of the list, number of unique elements, and how sorted it already is. Defaults to a vector of length 1 containing the depth of the recursion.
- **Optimizer O.** Given a feature extractor, a training data set, and a bag of solvers, the optimizer learns how to select and parametrize a solver given a vector of problem features. Optimizers could be implemented at varying degrees of complexity, but we imagine simple solutions such as SVMs to be competitive, and more complex solutions such as neural networks to start to saturate the gains one could achieve from a better optimizer module (at the cost of computation time). Optimizers should default to minimizing computation time and/or memory, but can take an arbitrary cost function $C(x, y)$ that produces some cost to be minimized given a problem and solution.

Once initialized, a call to a LHYRA instance solves a problem x through the following steps:

1. Compute $F(x)$.
2. Pass $F(x)$ into the solver selector learned by the optimizer, which returns a parametrized solver s_{ih}.
3. Compute $s_{ih}(x)$.
   (a) For each subproblem y_i identified by s_{ih}, repeat the above process.
   (b) Combine the subproblems and return a result.

Ideally, the solver selector and feature extractor would be fast enough for the benefits gained by ideal solver selection to outweigh the additional overhead. However, after implementing LHYRA, we may discover a number of possible outcomes:

1. LHYRA instances result in observable performance improvements in a wide array of real-world tasks.
2. LHYRA instances result in observable performance improvements for a few specific tasks.
3. LHYRA instances do not result in any observable performance improvements.

Regardless of the results, our expansive benchmarking will allow us to determine which data structures and applications are most improved by our use of multiple solvers (ignoring the time spend in the decision making process). This information could lead to future advancements both in application specific tasks (where the behavior of our solver selector could be manually replicated) and in generalized cases, as more efficient solver selectors / feature extractors are found.

Optimizer Design

One interesting component of learning hybrid recursive algorithms is developing generic classifiers (and their accompanying optimizers) for recursive algorithms. In LHYRA, this whole problem is encapsulated by the Optimizer abstraction.

If one were to think of the most naive possible way to do this, you could imagine randomly sampling the bag of solvers (including their hyperparameters) and the training data to build a table of (feature set, solver) $\rightarrow$ cost. One could then perform a variant of nearest neighbor on that table given an input x to find the solver with the lowest cost.

However, this approach is rife with complication. First, it is difficult using this framework to identify relationships between solvers (for example, if one solver is very good at solving its own subproblems). It would even be difficult to individually measure solver performance, as a fast solver that calls big subproblems might actually be less efficient than a slow solver that calls small subproblems. Even if one tries to get around this issue by averaging subproblem approaches, one still runs into the above issue of failing to model more complex relationships between solvers and the problems they best perform on.

Instead, we plan to model our optimization as a sort of reinforcement learning problem. We start with a randomly initialized solver picker $P(x) \rightarrow s_i$, and use it to evaluate LHYRA on an example from the training set. We record the time to perform this entire process, and treat that as a reward signal. We can then either use fancy methods from reinforcement learning (such as Q-Learning) to update our classifier that have the potential to understand these more complex solver relationships, or even just perform simulated annealing on P's parameters. Regardless, this provides a far more powerful framework for designing optimizers.

However, we do note a specific subcase in which the above is not ideal: that of intentional overfitting. For applications like RMI where the training data is the evaluation data, we should instead sample the entire solution space, and build a hashmap that provides $O(1)$ evaluation by looking up subproblems (this is possible, as weve already seen them before). Thus we expect to build at least two Optimizers: at least one using the RL method above, and one using this $O(1)$ method.

Potential Applications

Many problems and data structures in computer science are naturally recursive, and thus we expect no shortage of potential applications to benchmark LHYRA. Below are some of the most obvious examples we can think of where we might expect performance gains over a non-learned solution.

- **B-Trees.** If possible, we would like to re-implement Kraskas B-Trees in LHYRA to demonstrate how LHYRA is a true generalization. As LHYRA is partially modelled after RMI, we expect this to be a fairly straightforward adaptation.
- **Sorting.** Wed like to show how LHYRA can adapt to different distributions in choosing the optimal sorting method. Wed also like to show that LHYRA can also adapt to quickly sort approximately, and utilize processor parallelism where available.
- **Hierarchical Pathfinding.** Traditional pathfinding algorithms waste substantial amounts of computation and have difficulty adjusting to dynamic, online environments. The method of choice for hierarchical pathfinding is likely highly dependent on the exact optimization criteria, graph connectivity, and dynamism of the environment. This is a place where LHYRA may be invaluable, and also has clear real-world applications.
- **Matrix Multiplication.** LHYRA should be able to adaptively select naive matrix multiplication, naive divide and conquer, Strassens algorithm, or CoppersmithWinograd. This would be particularly

useful for machine learning. Nearest Neighbor: Given high-dimensional data, find the nearest neighbor. Examples of methods for this could be Locality sensitive hashing, Greedy search in proximity neighborhoods, and naive search.

Practical Notes

This is a project with a lot of breadth, flexibility, and engineering effort, which we believe warrants a three person group. We intend to develop LHYRA in Python for the sake of implementation speed, as we can still demonstrate performance improvements over naive Python implementations.

We expect that in addition to building implementations for the above applications (each with its own bag of solvers, etc), well experiment with different optimizers, classifiers, and additional applications. Wed also like to open source our codebase, and write sufficient documentation that future programmers may be able to use LHYRA for their own applications. We've setup a publically accessible repository here:

`https://github.com/joshuagruenstein/lhyra`