PsodaScript: Applying Advanced Language Constructs to Open-source Phylogenetic Search

Hyrum Carroll
hyrumcarroll@gmail.com

Mark J. Clement
clement@cs.byu.edu

See next page for additional authors

Follow this and additional works at: https://scholarsarchive.byu.edu/facpub

Part of the Computer Sciences Commons

Original Publication Citation

BYU ScholarsArchive Citation
Carroll, Hyrum; Clement, Mark J.; Krein, Jonathan; Snell, Quinn O.; and Teichert, Adam R., "PsodaScript: Applying Advanced Language Constructs to Open-source Phylogenetic Search" (2007). All Faculty Publications. 229.
https://scholarsarchive.byu.edu/facpub/229

This Peer-Reviewed Article is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.
PsodaScript: Applying Advanced Language Constructs to Open-source Phylogenetic Search

Jonathan L. Krein¹, Adam R. Teichert¹, Hyrum D. Carroll, Mark J. Clement and Quinn O. Snell
Computer Science Department
Brigham Young University
Provo, UT 84602

Abstract—Due to the immensity of phylogenetic tree space for large data sets, researches must rely on heuristic searches to infer reasonable phylogenies. By designing meta-searches which appropriately combine a variety of heuristics and parameter settings, researchers can significantly improve the performance of heuristic searches. Advanced language constructs in the open-source PSODA project—including variables, mathematical and logical expressions, conditional statements, and user-defined commands—give researchers a better framework for the exploration and exploitation of phylogenetic meta-search algorithms. PSODA’s approach to scripting meta-search algorithms is unique among open-source packages and addresses several limitations of other phylogenetic applications.

Index Terms—meta-searching, phylogenetic analysis, scripting languages

I. INTRODUCTION

Phylogenetic trees model the evolutionary relationships among species, enabling researchers to both analyze species differences [1] and to better understand the processes by which those differences arise [2]. Phylogenetic trees represent an important research tool in many scientific areas. In AIDS research, for example, scientists are using phylogenies to better understand how the Human Immunodeficiency Virus (HIV) mutates in response to the human immune system. Because HIV evolves faster than most known organisms, understanding its evolution will hopefully lead to improved vaccines [3]. Phylogenetic research is also contributing to many other areas of study, including nucleic acids research [4] and endangered species conservation [5].

A major task related to phylogenetic research is that of efficiently inferring, from molecular data, the correct phylogenetic tree for a set of organisms. One approach to the problem is to arrange a data set into all possible phylogenies, scoring each tree to find the one that most likely represents the actual evolutionary relationships among the organisms. Unfortunately, because phylogenetic inference is an NP-complete problem [6], such an exhaustive approach is typically unreasonable. Therefore, to infer trees from data sets with more than 15-25 taxa, phylogenetic inference software must rely heavily on heuristic algorithms [7]. This is significant because most realistic biological studies require data sets with at least 20 taxa [6], if not hundreds or thousands.

¹Equally contributing authors

Researchers have developed a number of heuristic algorithms that perform reasonably well on large data sets; nevertheless, local optima in a search space often prevent these algorithms from finding the globally optimal solution. This problem is exacerbated by the inability of researchers—due to the vastness of the search space—to confidently distinguish between local and global optima.

One answer to the local optimum problem is meta-searching, which combines various heuristic searches together into a single search algorithm [8]. In phylogenetic inference software packages, meta-searches may be constructed from existing algorithms either via an internal scripting language or an external wrapper. Unfortunately, wrapper applications tend to be cumbersome, often requiring considerable effort to orchestrate the interaction among heuristic searches. Furthermore, having to code meta-search algorithms in a wrapper increases complexity and reduces clarity. In an effort to build upon predecessor applications and provide a solid, usable framework for developing meta-search algorithms, Phylogenetic Search Open-source Data Analysis (PSODA) [9] implements a scripting language—PsodaScript—with syntax and constructs for clearly and concisely defining phylogenetic searches.

PSODA is an open-source phylogeny reconstruction application. It provides basic and advanced phylogeny searching capabilities and a progressive multiple sequence alignment method. It can perform phylogenetic analysis with Maximum Parsimony and Maximum Likelihood. Additionally, it comes with a graphical user interface (GUI) that allows for visualizing individual phylogenies as well as the progress of a search. PSODA’s performance is comparable to existing phylogenetic programs (see Figure 1). PSODA can be obtained for free from http://csl.cs.byu.edu/psoda/

This paper seeks to familiarize researchers with PsodaScript as a powerful tool in the utilization and exploration of phylogenetic meta search algorithms. We first provide a context for PsodaScript by reviewing related work in phylogenetic inference software and meta-searching. Next, we present a brief overview of the PsodaScript language. Finally, we discuss two specific meta search applications that can be implemented in PsodaScript.
II. RELATED WORK

Over the past few decades, developers have produced a number of phylogenetic inference software packages. Even a superficial summary of the many packages is beyond the scope of this paper. We discuss only a few of the more influential packages—PHYLIP [11], POY [12], PAUP* [13], BioPerl [14], and TNT [15]—and their relevance to PSODA’s internal scripting language, PsodaScript (see Tables I and II).

Existing packages already offer varying degrees of meta-searching capabilities. To perform advanced meta-searches, however, most of these packages (including PHYLIP, POY, and PAUP*) must be wrapped by an external program or language such as DCM [16], Perl, or shell scripting.

PHYLIP, an open-source package, is one of the longest maintained phylogenetic analysis applications, offering a collection of separate programs that can be used as a toolkit in phylogenetic research. PHYLIP’s component programs are written in C and are relatively easy to install and use. However, PHYLIP also runs much more slowly on large data sets than other phylogenetic applications [7]. Furthermore, because it has no internal scripting language, PHYLIP cannot perform meta-search algorithms without the help of external programs, which introduces inordinate complexity—a common difficulty with the wrapper approach to meta-searching.

One reason why using wrapper applications to run meta-search algorithms causes unnecessary complexity is that these programs must be written in languages whose syntax and terms are typically unfamiliar to phylogenetic researchers. POY and PAUP* provide command sets that draw from familiar terminology and facilitate some basic meta-searches, but both applications still require external programming to run more advanced meta-search algorithms. One of the goals of PsodaScript is to provide a language with a syntax and keywords that make describing phylogenetic search algorithms intuitive.

Another reason why using wrapper programs to run meta-searches introduces additional complexity is the difficulty of coordinating and converting inputs and outputs between heuristic searches. BioPerl addresses this problem, and many researchers use it to make needed conversions. However, as a framework for writing phylogenetic search algorithms it still suffers from over-complexity. BioPerl brings the power of Perl to phylogenetics, but installing Perl and the necessary modules can be frustrating, particularly for users with limited experience in computer systems administration. Moreover, Perl syntax can be an obstacle for researchers who wish to write complex algorithms, but who do not have the time to learn the finer points of Perl programming.

TNT implements its own internal scripting language with functionality similar to what PSODA provides. Although TNT has become a pacesetter in phylogenetic software, offering excellent search speeds [17], it ultimately limits users because it is closed source. For instance, TNT scores all trees with parsimony; users cannot score trees with likelihood, and they cannot extend TNT if they need it. In addition to being closed-source, TNT’s syntax is similar to Perl’s in that it provides great power, but at the expense of simplicity. PsodaScript takes the middle ground between the command languages of POY and PAUP* and the heavy scripting languages of Perl and TNT.

III. PSODA SCRIPT: METHODS & PROCEDURES

To date, researchers have developed several successful meta-searching techniques [18], [19]. However, without internal scripting capabilities and more advanced language constructs, using and experimenting with these techniques and other meta-search algorithms is awkward and limited. Our challenge was to design and implement an internal scripting language for PSODA in a way that would enable (even encourage)
meta-search exploration, without compromising simplicity or usability.

A. The “Available, but Optional” Principle

To develop an internal scripting language for PSODA powerful enough to perform advanced meta-searches without compromising simplicity, we decided to follow the paradigm that the language’s features—particularly the more advanced ones—must be available, but optional. In other words, not knowing about an advanced feature should not inhibit the user’s ability to use features that are already familiar. In fact, an empty PSODA program block executes without errors or warnings; thus, the user begins with a working program.

Another application of the “available, but optional” principle is found in the “execute” command, which allows users to execute external programs from within a PsodaScript. PSODA and PsodaScript are also open-source, giving interested developers access to modify the underlying C++ source code as needed. Thus, users have multiple ways to step into a more complex programming environment. PsodaScript makes these alternatives available without increasing the complexity of the core PSODA language or affecting users who want nothing to do with them; they are available, but optional.

B. Compatibility with the NEXUS File Format and PAUP*’s Command Set

Despite its meta-searching limitations, PAUP* has become a popular proprietary package, offering a rich variety of commands and performing heuristic searches with competitive speed. The success of PAUP* has helped inspire the creation of the open source project PSODA.

As is the case with PAUP*, input files for PSODA generally follow the NEXUS file format. This format was proposed in 1997 to be an extensible format for describing systematic information [20]. PSODA is also designed to execute PAUP* blocks, although it’s language does not yet implement every PAUP* command. Before execution, the PSODA interpreter warns the user about any unimplemented commands, references the commands by line number, and then skips over them during execution. As PSODA continues to be extended, more features and algorithms will be incorporated into the application.

There are two important reasons why PsodaScript was developed on the foundation of the NEXUS file format and PAUP*’s command set (see Sample 1). First, many people are already familiar with this format and command style, making it easier for users to begin using PsodaScript. The other motivation lies in the simplicity of PAUP*’s syntax and commands, which seem to be well suited to phylogenetic researchers. Building on PAUP*’s syntax, PsodaScript is intended to be understandable by someone with little programming experience. Where appropriate, PSODA uses words rather than obtuse symbols, and the more advanced language constructs are intended to read like English statements. A person who is familiar with PAUP* commands should be able to understand the general idea of what a PSODA program does, even if they have had no prior exposure to it.

C. A Quick Introduction to PsodaScript Syntax

In keeping with PAUP*’s style, PSODA instructions are each listed as distinct statements terminated by a semicolon. The instruction in Sample 1 tells PSODA to run a heuristic search beginning with the current trees in the repository and using the TBR method to explore the search space. Even with very little exposure, the syntax and terminology make sense.

Sample 1 PsodaScript is based on PAUP*’s command syntax and the NEXUS file format

```plaintext
hsearch start = current
swap = TBR;
```

In addition to running PAUP* commands, PsodaScript also allows users to store values in user variables. Assignment to a variable is performed via the = operator. For instance, the statement `height = 5;` assigns the value of 5 to the variable `height`. If the variable does not yet exist, it is created and initialize to the value 5. To avoid unintended side effect of assigning variables, PsodaScript uses the notion of variable scoping, which means that a variable exists within a well defined region in the program. If a variable is created within a loop, for instance, then it will not be visible, or accessible, outside of that loop; it is considered, therefore, local to that loop. If a variable is created outside of all constructs, then it will be visible throughout the program (except for in user defined command bodies).

Another point to note about variables in PsodaScript is that they are loosely typed, as in Perl or JavaScript. This means that users do not have to explicitly declare the type of data that a variable will hold when they create it. One variable in PsodaScript can be assigned data of any type, and when necessary, the PSODA interpreter will attempt to perform conversions. PsodaScript variables can also be combined with operators to form arithmetic and logical expressions.

Sample 2 Syntax for a Conditional Construct

```plaintext
if (<condition-1>)
    ...
elsif (<condition-2>)
    ...
elsif (<condition-3>)
    ...
else
    ...
endif;
```

Conditional constructs allow users to determine how a program (or meta-search) should proceed based on the current state of the search. Sample 2 illustrates the syntax for a
conditional construct. In a conditional construct, the instructions following the first true condition are executed once, after which the interpreter skips to the end of the conditional (endif;) to continue on with the program. If none of the conditions evaluate to true, then the else component is executed. A conditional statement may or may not have an else component, and it may have zero or more elsif components.

The loop construct is written in a similar format (see Sample 3). In a while loop construct, the body is repeatedly executed as long as the condition expression evaluates to true.

D. Error Checking and Handling

Given its critical impact on usability, there is an ongoing effort to develop PsodaScript’s error checking and handling. Because phylogenetic searches frequently run for several days or weeks at a time, error handling should include some degree of data recovery following a fatal scripting error. Good error checking and handling is even more important when running meta-search algorithms, which are more likely (due to code branching) to run longer before encountering faulty logic. Before execution, the PSODA interpreter informs the user of fatally incorrect syntax and gives warnings for several non-fatal errors. For fatal logic errors, which may surface during execution, the interpreter informs the user of the error and the need to quit. It then gives the user an opportunity to save the trees in the repository before exiting.

E. User-defined Commands: Facilitating Algorithm Readability and Design

PsodaScript also supports user-defined commands for grouping program statements together under a single command name. The ability to group statements is a simple but powerful concept. Consider a large meta-search algorithm that has the ability to perform various heuristic searches, each of which was derived from a base search and modified via a specific configuration of parameters, weights, and so forth. Duplicating the code for each heuristic search every time it is used leads to a messy program block, which translates into more mistakes and decreased algorithm readability. In short, grouping and labeling program segments is a simple technique that makes an algorithm much easier to understand as a whole; it helps current and future researchers answer the question, “What is this algorithm really doing?”

Sample 4 illustrates a user-defined command that randomly skews a given percentage of column weights—part of a technique used by the ratchet [18] to escape a local optima. It is called as if it were a built-in PsodaScript command (i.e. randomReweight). The first several lines of Sample 4 are a comment, and are not executed. Notice that the first statement in the randomReweight command creates two variables, range and percent, with the default values of 3 and 25 respectively. Just like any other PsodaScript commands, this command could be called with a parameter: randomReweight percent = 20; Listing percent = 20 as a parameter, overrides the default value of 25.

The real benefit of user-defined commands, therefore, is their ability to abstract away substantial segments of code, which can then be executed as often as needed via a single, parameterized command call.

IV. APPLICATIONS

To demonstrate the use of PsodaScript for meta-search construction, we present here the implementation of two specific meta-search algorithms: Parsimony Ratchet and Alignment with Feedback.

A. Parsimony Ratchet

As discussed above, heuristic searches for optimal phylogenetic trees are often caught in local optima and thus fail to find the global optimum. The Parsimony Ratchet attempts to avoid this problem by executing a series of heuristic searches alternating between skewed and normal weightings for tree scoring [18]. For the sake of comparison, we present a ratchet that could be run by PAUP* together with a possible implementation of a ratchet in PsodaScript (see Sample 5). Developers have created tools that facilitate using the ratchet in PAUP* (or PSODA) by generating long sequences of commands that simulate looping and random number generation [21]. The PAUP* ratchet dramatically improves search performance and can be run in PSODA; however, using the programming
The constructs of PsodaScript to implement the ratchet offers a number of advantages in addition to the performance boost. For instance, during the ratchet, better scoring trees are replaced by other (perhaps worse) trees on subsequent iterations. With variables and conditional constructs, however, PsodaScript can also track the best score found so far, and save only those trees that achieve a new best score. Additionally, PAUP* merely simulates a loop by running a long sequence of heuristic searches, the length of which is ultimately limited by memory and/or disk space. On the other hand, with a simple loop, PsodaScript can run indefinitely, generating new random numbers on each iteration.

Another disadvantage to the PAUP* ratchet is the added dependence on external software, which further obfuscates the algorithm. On the other hand, consider the while loop in the main body of the PSODA program (see Sample 5). It makes the algorithm of the ratchet search clear. First, twenty percent of the weights are randomly re-weighted with skewed values, and the search continues with the skewed weights. Afterward, the weights are reset to their initial values, and the search continues under normal weights. This process of alternating the search between skewed and normal weights is repeated until the user tells PSODA to stop.

A further benefit of the PsodaScript framework is the way it facilitates experimentation with parameters, such as the range of random numbers used in the weighting or the percentage of skewed weights. Additionally, researchers can use print statements to periodically output information about
the state of the search. Performing these simple tasks without a programming language is impractical, and as noted above, external languages introduce unnecessary complexity.

B. Feedback Alignment

Phylogenetic inference frequently begins by producing a multiple sequence alignment (MSA) using software such as ClustalW [22] and an initial phylogenetic guide tree. In preparation for a phylogenetic search, MSA inserts gaps into the sequence data to make all sequences the same length. Research has shown that the quality of the alignment significantly impacts the success of the phylogenetic search [23]. The software package POY performs a MSA for every tree searched—thus making expensive MSA calculations on suboptimal trees [24]. A possibly more efficient alternative to POY’s approach is feedback alignment [25], which performs occasional realignment of the taxa based on the best tree found so far. The sample code below shows the simplicity of implementing feedback alignment in PsodaScript.

Sample 6 A Possible Implementation of Feedback Alignment

```
hs (start=stepwise, nreps=5);
while (true)
  hsearch (start=current);
  align (guidetree=best);
endwhile;
```

V. CONCLUSION

The immensity of phylogenetic tree space for interesting data sets demands that researchers rely on heuristic searches to infer phylogenies. By designing meta-searches which appropriately combine various heuristics and parameter settings, phylogenists greatly improve the practicability of using inferred phylogenetic trees to solve problems. The internal scripting abilities of the open-source PSODA project give researchers the flexibility to better explore and exploit the realm of phylogenetic meta-searching while addressing several limitations of previous phylogenetic applications.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 0120718.

REFERENCES


