SAND2015-2327R

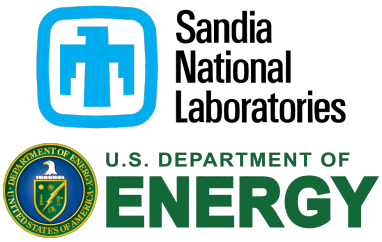
Visualization for Hyper-Heuristics: Back-End Processing

Luke Simon

Faculty Advisor
Dr. Daniel Tauritz, Department of Computer Science

Natural Computation Laboratory

Sandia National Laboratories[1]



March 28, 2015

[1]*Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.*

Abstract

Modern society is faced with increasingly complex problems, many of which can be formulated as generate-and-test optimization problems. Yet, general-purpose optimization algorithms may sometimes require too much computational time. In these instances, hyper-heuristics may be used. Hyper-heuristics automate the design of algorithms to create a custom algorithm for a particular scenario, finding the solution significantly faster than its predecessor. However, it may be difficult to understand exactly how a design was derived and why it should be trusted. This project aims to address these issues by creating an easy-to-use graphical user interface (GUI) for hyper-heuristics and an easy-to-understand scientific visualization for the produced solutions. To support the development of this GUI, my portion of the research involved developing algorithms that would allow for parsing of the data produced by the hyper-heuristics. This data would then be sent to the front-end, where it would be displayed to the end user.

1 Introduction

There are many problems within the realm of computer science for which exhaustive search methods cannot find optimal solutions. In these instances, using exhaustive search methods to find the optimal solution can cost too much time. Instead, practitioners may use what are called heuristics to find a solution. Though heuristics are incomplete search methods that find sub-optimal solutions to problems, they find these solutions significantly faster than exhaustive search methods[1]. Hyper-heuristics, then, automate the design of algorithms by combining multiple heuristics in order to create a custom algorithm for a particular scenario.

While hyper-heuristics are efficient at building supposed good solutions to generic problem sets, they are not good at telling exactly why a particular solution is favorable over other solutions. Not knowing where a solution comes from could potentially lead the practitioner to abandon it altogether in favor of a lesser solution of known origin. Though the optimal solution is found, it remains unused and its validity unproven.

The main purpose of this research was to provide a Graphical User Interface (GUI) whereby practitioners could quickly and easily validate a particular solution generated by a hyper-heuristic. By providing tools which delineate the origin of a particular solution, the legitimacy can be trusted and the solution used. In order to provide this GUI, our research integrated with an already created framework. Rather than generate the solution itself, our research focused instead on validating a solution produced by a standalone framework.

The VASUKI system was integrated with the Hyper-Heuristic Developed Robust Algorithms (HYDRA) framework. HYDRA focuses on using Hyper-Heuristics to produce Black Box Search Algorithms (BBSAs)[2]. In order to validate these solutions, we sought methods to visualize both the individual BBSAs and their traits and the phylogenetic trees displaying the genetic ancestry of these BBSAs. By providing these two visualization tools, the practitioner could not only determine which attributes a supposed optimal solution contained but also where these attributes came from. Doing so would hopefully allow the practitioner to

validate the produced solutions.

2 Design Decisions

A GUI's usefulness lies not only in the scope of its functionality, but also in its usability[3]. An interface containing vast amounts of functionality not easily accessible is perhaps less likely to be used than an interface containing limited though easily accessible functionality. Furthermore, a GUI whose functionality is limited to a single machine or framework may also prove to be inadequate. This being considered, usability largely motivated the design of the VASUKI interface.

2.1 The Need for a Cross-Platform System

Limiting the capability of VASUKI to a single operating system would greatly hinder the system's usage. Thus, choosing a development language that supported this cross-platform goal became crucial. Initially, the entirety of the VASUKI system was to be written and developed using Python's Tkinter[1]. Being that HYDRA was developed using Python, we thought it relatively easily to integrate this framework into the VASUKI system if VASUKI was also developed in Python. However, the limitations of Tkinter quickly became manifest. Though Tkinter could potentially allow for easy integration of Python based hyper-heuristic frameworks, it did not support cross-platform development as well as other potential alternatives. Although Tkinter was easily compatible with desktop based operating systems, it was not compatible with mobile operating systems. Thus, we explored other options.

Immediately after abandoning Tkinter we were drawn to the idea of developing VASUKI as a web-based application. The power of web-based applications lies in their ability to be run from any device that has an Internet browser. Developing VASUKI in this manner opened up the possibility of utilizing the system on a mobile device. The system could easily be custom built for any screen size, and run on virtually any platform. In addition to these benefits, numerous JavaScript libraries exist that render useful data visualization. Rather than develop our own methods to visualize data, we could simply use existing visualization techniques and spend more time developing algorithms that suited other needs such as parsing through the data.

Though it did not meet the need for a cross-platform system, Tkinter did provide a method for easy integration of other frameworks because of its native Python language. Thus, we did not completely abandon Python. Rather than use Tkinter, we decided to use Django[2] to create our web application. With this approach, our Django app could sit on the same server containing some of the frameworks VASUKI supports, such as HYDRA. The Django/Python code would then be able to act as an interface between the client side JavaScript and the HYDRA code, also potentially making it possible to do remote runs of HYDRA in the future.

[1] A Graphical User Interface (GUI) that comes standard with Python

[2] Django is a high-level Python web framework that allows for rapid development.

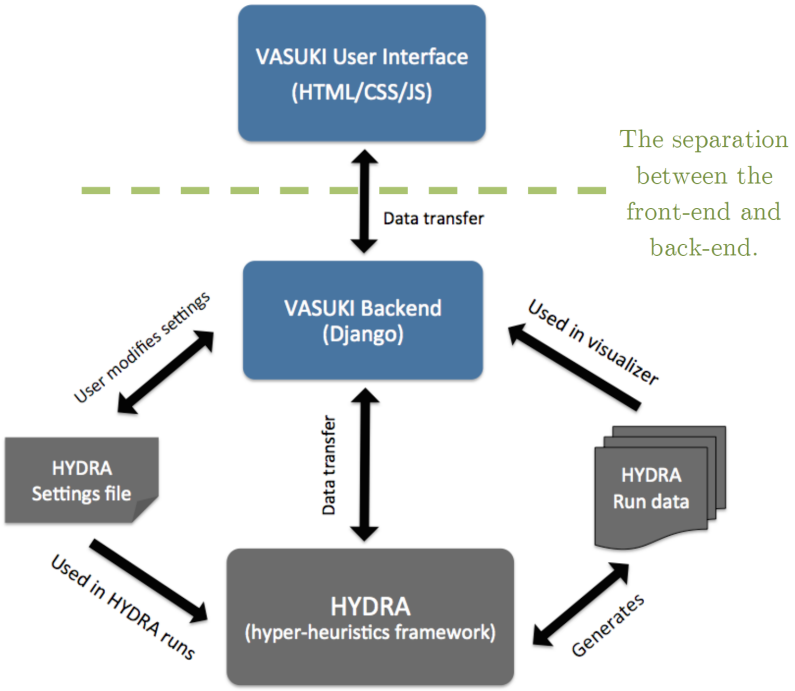


Figure 1: The workflow of the web interface

2.2 Integration of Multiple Frameworks

VASUKI was also designed in such a way that would allow the integration of multiple frameworks. VASUKI does not exist to produce hyper-heuristic solutions to problems. Rather, it exists to validate and explore these solutions. This being the case, the system needed to be designed so that any solutions produced by any frameworks could be explored and validated. In order to allow for future additions to the system, this design choice became a requirement.

3 Implementation Details

As it is a web interface, the VASUKI system consists of both a server side (the back-end) and a client side (the front-end). The majority of the data parsing occurs in the back-end, whereas the front-end contains what is visible to the user. Aforementioned requirements and design choices led the back-end to be developed using Django and Python and the front-end to be developed using JavaScript, CSS, jQuery, and HTML5. While my collaborator focused on developing the front-end, I focused on developing the back-end. My branch of research involved taking all of the data associated with each hyper-heuristic produced solution and parsing it so that the front-end could visualize it.

Figure 1 demonstrates the important fact that the back-end has frequent communication with the front-end. The higher level work-flow can be described as follows:

1. Based off of the provided settings file, HYDRA uses hyper-heuristics to generate supposed optimal BBSA solutions to a problem.

2. The generated run data is sent to the VASUKI back-end, where it will later be analyzed and parsed.

3. Once the run data has been parsed and stored in a useful format, it is sent to the front-end.

4. The front-end then provides tools to interact with and visualize the data, thus allowing the practitioner to hopefully validate an optimal solution.

5. Data may then be sent again to the back-end, where it is parsed and updated again to be resent to the front-end.

3.1 Developing Generic Solutions

To develop the back-end meant to develop very generic solutions to particular parsing problems. Mainly because there was only a single integrated framework instead of multiple, it would have been easiest to explicitly hard-code all of the data parsing. HYDRA has a standard format that all produced data files are in, and a specific solution to parse these data files would have been straightforward to implement. However, this would have been contrary to the entire vision of the research. We wished to allow for additional framework support if desired, and the system had to be designed in such a way that allowed for this. Parsing using a method specific to the data from an individual framework would have greatly discouraged additional frameworks from being integrated. Thus, extra time was spent to ensure that any developed code was generic and robust enough to parse data from multiple frameworks. This would prove to be very difficult. However, it also ensured that VASUKI remained robust to changing requirements and desires of the practitioner.

3.2 Dynamically Parsing the Settings File

The HYDRA framework relies on a settings file to define specifics for each run. Specifics such as but not limited to problem type and representation, convergence, penalties, and others are defined by this file. One of the first objectives within this research was to develop a methodology that would enable a user to easily view and edit the settings files used by HYDRA. By displaying the settings file in an easily recognizable and editable form, we enabled the practitioner to have a further understanding of where a particular hyper-heuristic solution comes from. In order to display the settings file in an easily recognizable format, it had to first be parsed by the back-end. To allow for the best user experience, the front-end required information about the data type, tool-tip, and requirement of each field. All of this information was parsed and stored within Python. The process was greatly expedited by the developer of the HYDRA system, who provided a file containing all information related

Figure 2: The provided file with information about each field of the settings file

to data types and requirements of each item in the settings file. A second file containing tool-tip information was produced based off of this first file.

As shown by Figure 2, the provided file took on the form of a Python dictionary[3], where keys represented the field name and values represented the type of field (that is, a number, string, etc.). The file containing information about tool-tips had the same format, with the distinction being that values were instead the tool-tip message associated with each field.

Due to the varying depths at which information could be found, parsing the given file for the required information proved to be very difficult. Given a dictionary of a single depth, it would have been straightforward to iterate through each and every field and record the associated field type. However, this file contained information at depths of up to five layers deep. Furthermore, it did not contain information about how deep particular information could be found. Even if the file did contain this information, it was highly probable that settings files from different frameworks could not have provided the same data. Thus, the generic nature of VASUKI required a generic parsing solution.

To parse the settings file, a programming method known as recursion[4] was used. Simplified, a function was written which continued moving deeper through each layer until it encountered necessary data. The required data could then be stored, and other layers of varying depths explored. This function explored files generically enough so that it could be applied to frameworks of varying types.

[3]https://docs.python.org/2/tutorial/datastructures.html
[4]http://pages.cs.wisc.edu/~vernon/cs367/notes/6.RECURSION.html

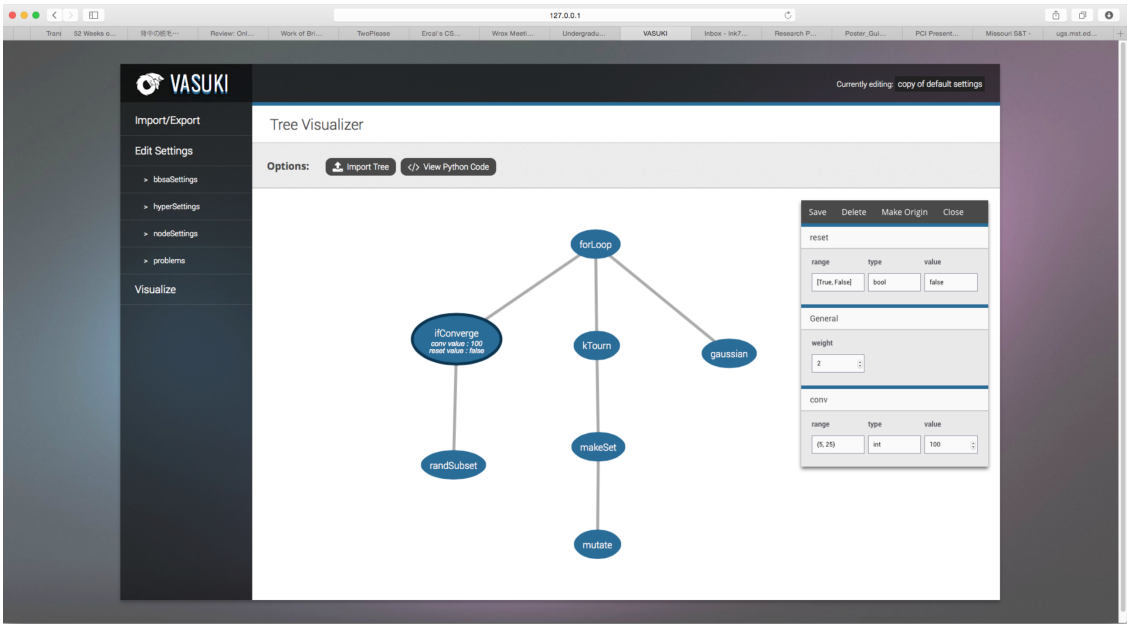


Figure 3: The BBSA visualizer

3.3 Displaying BBSA trees

HYDRA uses an initial population to evolve BBSAs fitted to particular problems. One of the tools (see Figure 3) provided by VASUKI is used to edit and create the BBSAs used for this initial population. The user is able to upload a BBSA tree to edit and visualize its traits, and then output the new BBSA in the same format used for uploading. This capability is greatly advantageous. Giving practitioners the ability to view and edit the initial population used by HYDRA provides yet another method of tracing the origin of a solution. Being able to visualize the BBSAs found within this initial population allows for more insight into what initial conditions may produce superior solutions, and being able to edit the BBSA trees allows the practitioner to investigate through experimentation the causation of these superior solutions.

Each BBSA tree is made up of multiple nodes of varying types (forLoop, ifConverge, kTourn, etc.). The available types can all be found within the settings file provided by the HYDRA developer. Hence, the data associated with BBSA trees had to be parsed in a way similar to how the settings file was parsed. Thankfully, this functionality was already provided from previous implementation. The provided implementation was used to parse the given data, and the result sent to the front-end. The benefit of compartmentalization is shown quite obviously here. The back-end simply parses the data and makes it usable by the front-end. What the front-end does with this data is not known by the back-end. Thus, changing either component does not require updating the other component. This allows developers to work independently on either part. In this instance, the back-end simply makes available the potential nodes and their associated data. How these individual nodes are rendered and the BBSA tree created is unimportant from the point of view of the

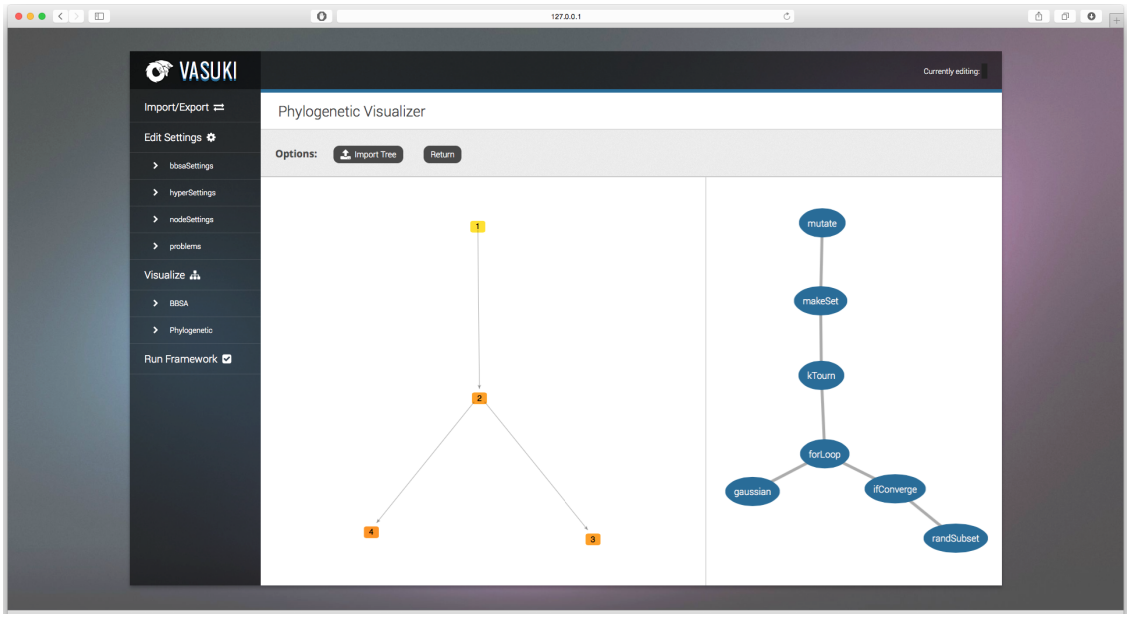


Figure 4: The phylogenetic tree visualizer

back-end.

3.4 Displaying Phylogenetic Trees

One method to determine the validity of a solution produced by hyper-heuristics is to trace its origin. By investigating the origin of particular gene in a tree, the practitioner is also able to determine why a particular trait was passed on. If a trait survives several generations, this implies that this particular trait is likely to be advantageous to the solution. If a trait dies off after one generation, the opposite conclusion could be reached. To quickly spot the traits persistent or temporary throughout each generation of the BBSA trees, we developed a way to view an interactive phylogenetic tree[5] representing the ancestry of each and every BBSA solution (see Figure 4)

Every run of Hydra produces a file used by GraphViz[6] to visualize a static image of the phylogenetic tree. This file is human-readable and contains valuable information regarding all ancestry found within the tree. To create an interactive phylogenetic tree, the GraphViz file produced by HYDRA was used. Information regarding nodes, edges, and colors was all provided, so I simply had to parse the GraphViz data and convert it into a format usable by the front-end of VASUKI. In this instance, the created parser was generic in the sense that it allowed for the parsing of any GraphViz file. Admittedly, it could be possible for future frameworks to not provide this GraphViz file. However, given that a GraphViz file was provided by HYDRA and given that it contained all of the information necessary to create

[5]http://evolution.berkeley.edu/evolibrary/article/phylogenetics_02
[6]www.graphviz.org

an interactive phylogenetic tree, this detail was ignored and the functionality implemented. Furthermore, one could easily generate the GraphViz file of a future framework using an intermediate step. The produced file could then be used to create the interactive phylogenetic tree.

4 Conclusion

Runs performed in hyper-heuristic frameworks produce complex data that can be difficult to interpret. The visualizations produced by VASUKI facilitate understanding evolutionary paths. The strength of Vasuki lies in its interactive creation of visualizations, rather than just displaying static diagrams. Examining a solution more closely or tracing its origin can grant a practitioner further confidence that the solution produced by a hyper-heuristic is valid and trustworthy. By providing tools that visualize many stages of hyper-heuristic frameworks, VASUKI aids the practitioner in understanding hyper-heuristics and their resulting solutions.

5 Acknowledgments

Acknowledgments go to my faculty advisor, Dr. Daniel Tauritz, and mentor, Dr. Samuel Mulder. Furthermore, I would like to thank my collaborator Lauren Kroenung for her continued support throughout this project.

References

[1] Burke, Edmund. "Hyper-Heuristics." *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Second ed. Springer, 2013. Print.

[2] Matthew A. Martin and Daniel R. Tauritz. 2013. Evolving black-box search algorithms employing genetic programming. In *Proceedings of the 15th annual conference companion on Genetic and evolutionary computation* (GECCO '13 Companion), Christian Blum (Ed.). ACM, New York, NY, USA, 1497-1504.

[3] Nielsen, Jakob. "Nielsen Norman Group." *Usability 101: Introduction to Usability*. Nielsen Norman Group, 4 Jan. 2012. Web. 20 Mar. 2015. http://www.nngroup.com/articles/usability-101-introduction-to-usability/ .