Missouri S&T Library and Learning Resources

Scholars' Mine

Masters Theses

Student Theses and Dissertations

Summer 2019

Advanced techniques for improving canonical genetic programming

Adam Tyler Harter

Follow this and additional works at: https://scholarsmine.mst.edu/masters_theses

Part of the Computer Sciences Commons

Department:

Recommended Citation

Harter, Adam Tyler, "Advanced techniques for improving canonical genetic programming" (2019). *Masters Theses*. 7905.
https://scholarsmine.mst.edu/masters_theses/7905

This thesis is brought to you by Scholars' Mine, a service of the Missouri S&T Library and Learning Resources. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

ADVANCED TECHNIQUES FOR IMPROVING CANONICAL GENETIC PROGRAMMING

by

ADAM TYLER HARTER

A THESIS

Presented to the Graduate Faculty of the

MISSOURI UNIVERSITY OF SCIENCE AND TECHNOLOGY

In Partial Fulfillment of the Requirements for the Degree

MASTER OF SCIENCE

in

COMPUTER SCIENCE

2019

Approved by:

Daniel Tauritz, Advisor
Steven Corns
Patrick Taylor

Copyright 2019

ADAM TYLER HARTER

All Rights Reserved

PUBLICATION THESIS OPTION

This thesis consists of the following two papers, formatted in the style used by the Missouri University of Science and Technology:

Paper I: Pages 4-15 have been published in the proceedings of the Parallel and Distributed Evolutionary Inspired Methods workshop at the Genetic and Evolutionary Computation Conference of 2017.

Paper II: Pages 16-35 have been accepted by the Evolutionary Computation for the Automated Design of Algorithms workshop for publication in the proceedings of the Genetic and Evolutionary Computation Conference of 2019.

ABSTRACT

Genetic Programming (GP) is a type of Evolutionary Algorithm (EA) commonly employed for automated program generation and model identification. Despite this, GP, as most forms of EA's, is plagued by long evaluation times, and is thus generally reserved for highly complex problems. Two major impacting factors for the runtime are the heterogeneous evaluation time for the individuals and the choice of algorithmic primitives. The first paper in this thesis utilizes Asynchronous Parallel Evolutionary Algorithms (APEA) for reducing the runtime by eliminating the need to wait for an entire generation to be evaluated before continuing the search. APEA is applied to Cartesian Genetic Programming and is successful in reducing the runtime with sufficiently complex problems. The second paper in this thesis introduces Primitive Granularity Control (PGC), a method for reducing the impact and importance of the choice of primitives by allowing the primitive set to change throughout the course of evolution. Evidence is presented that demonstrates the potential for PGC to improve the quality of solutions, reduce the runtime of the algorithm, or both. However, the evidence was obtained via an exhaustive search, and how to effectively utilize PGC still requires research.

ACKNOWLEDGMENTS

Without the support of others, completing this degree would have been impossible, and for this I am indebted to them. I would like to thank Dr. Daniel Tauritz for all of his support, guidance, and teaching and for continuing to believe in my ability to finish this degree when I had nearly given up. I would like to thank all of my co-authors, Dr. William Siever, Aaron Pope, and Chris Rawlings, for pushing my publications beyond where I could have brought them by myself. I would like to thank Dr. Steven Corns and Dr. Patrick Taylor for being a part of my thesis committee. I would like to thank Los Alamos National Laboratory for supplying funding for the second publication via the Cyber Security Sciences Institute under subcontract 259565 and the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project number 20170683ER. I would like to thank my sisters and my parents for all of the support they've given me throughout both my undergraduate and my graduate years.

TABLE OF CONTENTS

Page

PUBLICATION THESIS OPTION . . . iii

ABSTRACT . . . iv

ACKNOWLEDGMENTS . . . v

LIST OF ILLUSTRATIONS . . . viii

LIST OF TABLES . . . ix

SECTION

1. INTRODUCTION . . . 1

PAPER

I. ASYNCHRONOUS PARALLEL CARTESIAN GENETIC PROGRAMMING . . . 4

ABSTRACT . . . 4

1. INTRODUCTION . . . 5

2. RELATED WORK . . . 6

3. ASYNCHRONOUS PARALLEL CGP . . . 7

4. EXPERIMENTATION . . . 8

4.1. PROBLEM . . . 8

4.2. EXPERIMENT DESIGN . . . 8

5. RESULTS . . . 10

6. CONCLUSION . . . 14

7. FUTURE WORK . . . 14

REFERENCES 15

II. EMPIRICAL EVIDENCE OF THE EFFECTIVENESS OF PRIMITIVE GRANULARITY CONTROL FOR HYPER-HEURISTICS 16

ABSTRACT 16

1. INTRODUCTION 17

2. RELATED WORK 18

3. PRIMITIVE GRANULARITY CONTROL 19

4. TRAVELING THIEF PROBLEM 21

5. METHODOLOGY 22

5.1. META-SEARCH 23

5.2. TTP HEURISTIC EVOLUTION 23

6. EXPERIMENTATION 25

7. RESULTS 28

8. CONCLUSION 29

9. FUTURE WORK 32

ACKNOWLEDGMENTS 33

REFERENCES 33

SECTION

2. SUMMARY AND CONCLUSIONS 36

VITA 38

LIST OF ILLUSTRATIONS

Figure Page

SECTION

1.1. Canonical Evolutionary Algorithm Process 2

PAPER I

1. Exploration of Search Space in Synchronous CGP 9

2. Exploration of Search Space in Asynchronous Parallel CGP. 9

3. Results for 3-parity with an Overhead of 200 10

4. Results for 3-parity with an Overhead of 150 11

5. Results for 3-parity with an Overhead of 100 11

6. Evaluations per Second of Synchronous Serial with a Variety of Overheads for 2-bit Parity 12

7. Evaluations per Second of Asynchronous Parallel and Synchronous Parallel with a Variety of Overheads for 2-bit Parity 13

PAPER II

1. Example TTP Heuristic at Three Primitive Coarseness Levels. 21

2. TTP Example Instance 22

3. High Level Overview of Experiment 29

4. Max Fitness Versus Runtime for Each Run of the Best Dynamic and Static Plans for Maximum Fitness 30

5. Comparison of Each Best Configuration 31

LIST OF TABLES

Table Page

PAPER I

1. Parameters Used for Experimentation 9
2. Statistical Analysis of 3-parity Results with an Overhead of 200 10
3. Statistical Analysis of 3-parity Results with an Overhead of 150 11
4. Statistical Analysis of 3-parity Results with an Overhead of 100 12
5. Statistical Analysis of 2-parity Results with an Overhead of 400 13
6. Statistical Analysis of 2-parity Results with an Overhead of 175 14

PAPER II

1. Terminal Primitive Effective Coarseness 24
2. Non-terminal Primitive Effective Coarseness 24
3. List of Basic Primitives 26
4. List of Macro Primitives 27
5. TTP Solver Specific Parameters 28
6. Comparison of Static and Dynamic Configurations 31

SECTION

1. INTRODUCTION

Evolutionary Algorithms (EA's) are a class of randomized population based black box search algorithms inspired by biological evolution. The canonical form of EA is shown in Figure 1.1, a description of each major component follows. Initialization is the creation of the starting population; there are many methods for performing this, but random initialization is common. Parent Selection & Child Generation is furthering the search by using the members of the previous generation to generate new solutions. The individuals selected as the basis for the new children are stochastically chosen based on how well they solve the problem; the better the solution, the more likely they are to be chosen. Evaluation is the determination of how well each individual solves the problem. Survival Selection is the process of selecting which individuals will be kept for the next generation and which will be discarded; the better the individual, the higher chance it will survive. The Termination Check is simply the determination if the algorithm should continue or be stopped. Common stopping criteria are stagnation of solution quality and reaching a pre-chosen limit on the number of generations.

Evolutionary algorithms are generally utilized when more traditional search algorithms are unsuitable, either due to problem complexity or non-traditional search spaces. One such common case is automated program generation — the creation of programs with minimal or no human intervention. For this, a type of EA known as Genetic Programming (GP) is often used, in which the members of the population can represent executable programs. While a powerful search method, GP is not without limitations. Search times quickly grow prohibitively large and the performance of the search is heavily dependent on the choice of primitives, the building blocks of code that an individual can utilize. While

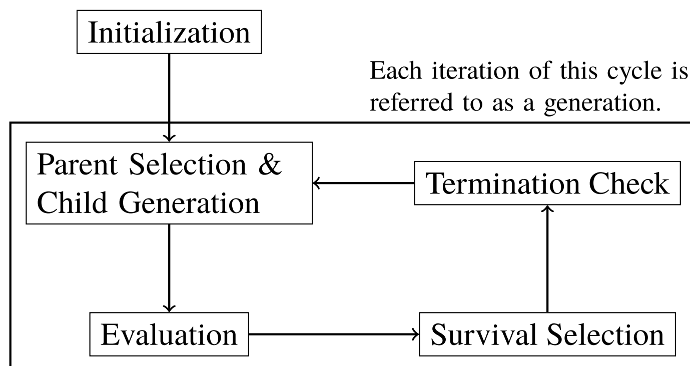


Figure 1.1. Canonical Evolutionary Algorithm Process

parallelization helps to combat the long runtime, canonical GP algorithms fail to take full advantage of resources due the heterogeneous evaluation times, caused by individuals in GP taking greatly different amounts of time to be evaluated. This can create a bottleneck where individuals that take a disproportionally long time to evaluate. To combat this, Asynchronous Parallel Evolutionary Algorithms (APEA) changes the traditional generational model to one that generates a new individual in response to the completion of the evaluation of an individual. Such a change also requires changing how parent selection and survivor selection function, as both methods traditionally operate on the entire population at the same time. Applying APEA to Cartesian Genetic Programming (CGP) requires a change in methodology due to CGP's unique population size of one. The first paper presents a method for combining CGP and APEA.

Primitive selection is one of the most vital components for the performance of a GP, as it can radically change the search space. Primitives that are too low-level are likely to be able to find an optimal solution, but the search space becomes so large that doing so is improbable. High-level primitives will converge quickly, but are unlikely to be able to find an optimal solution. By allowing the primitive set to change between different levels

of complexity during evolution, both high level and low level primitives can be used while maximizing the benefits and minimizing the deficits of both. The second paper presents a method, Primitive Granularity Control, to do this as well as evidence of its effectiveness.

PAPER

I. ASYNCHRONOUS PARALLEL CARTESIAN GENETIC PROGRAMMING

Adam Harter and Daniel R. Tauritz
Natural Computation Laboratory, Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, 65409

William M. Siever
Department of Computer Science and Engineering, Washington University, St. Louis, MO, 63130

ABSTRACT

The run-time of evolutionary algorithms (EAs) is typically dominated by fitness evaluation. This is particularly the case when the genotypes are complex, such as in genetic programming (GP). Evaluating multiple offspring in parallel is appropriate in most types of EAs and can reduce the time incurred by fitness evaluation proportional to the number of parallel processing units. The most naive approach maintains the synchrony of evolution as employed by the vast majority of EAs, requiring an entire generation to be evaluated before progressing to the next generation. Heterogeneity in the evaluation times will degrade the performance, as parallel processing units will idle until the longest evaluation has completed. Asynchronous parallel evolution mitigates this bottleneck and techniques which experience high heterogeneity in evaluation times, such as Cartesian GP (CGP), are prime candidates for asynchrony. However, due to CGP's small population size, asynchrony has a significant impact on selection pressure and biases evolution towards genotypes with shorter execution times, resulting in poorer results compared to their synchronous counterparts. This paper:

1) provides a quick introduction to CGP and asynchronous parallel evolution, 2) introduces asynchronous parallel CGP, and 3) shows empirical results demonstrating the potential for asynchronous parallel CGP to outperform synchronous parallel CGP.

Keywords: Genetic Programming, Asynchronous Parallel Evolution, Cartesian Genetic Programming, Evolutionary Computing

1. INTRODUCTION

Cartesian Genetic Programming (CGP) arranges problem-specific operations as function nodes on a two-dimensional grid. Unlike the genotypes in most forms of GP, these grids remain a static size and may need to be quite large to encapsulate complex solutions. Evaluating the fitness of this structure requires that input be passed to a set of initial nodes that then produces output for other nodes. Inputs are propagated from one function node to the next through the grid; however, not all nodes will necessarily be evaluated. The number of evaluated nodes in the genotype heavily influences the fitness evaluation time, therefore the variation in these times can become significant with large grid sizes. Much like most traditional evolutionary algorithms (EAs), evaluations of individuals are independent of each other in CGP and can be performed in parallel. Classic CGP employs the synchronous model common to the vast majority of EAs, in which all offspring in a generation are evaluated before survival selection is executed. Upon parallelization, the variation of evaluation times can cause classic CGP to excessively idle while waiting for individuals to be evaluated. To combat this problem, we are proposing an asynchronous model, in which survival selection is performed for each offspring individually immediately after evaluation is finished.

The contributions of this paper are as follows:

- Demonstrate statistical evidence that our proposed asynchronous parallel CGP (APCGP) may converge faster than synchronous parallel CGP (SPCGP) in regards to wall-time
- Provide analysis of scalability of APCGP with regards to problem complexity with comparison to SPCGP

2. RELATED WORK

Durillo et al. have shown empirical evidence supporting the significant improvement in terms of various quality metrics when employing asynchronous parallel EA's (APEAs) rather than synchronous parallel EAs for NSGA-II. The APEA master process creates and sends individuals to be evaluated as the slave processors become idle. In the generational version, the population is replaced when enough offspring have been generated. With the steady-state alternative, the offspring are considered as each is received. The researchers employed homogeneous populations as the test cases during experimentation. Bertels and Tauritz performed similar experiments, evolving SAT solvers asynchronously and synchronously, with the asynchronous models outperforming the synchronous ones.

APEAs with heterogeneous populations have been found to be biased toward individuals with shorter evaluation times. This is a result of the master process receiving those individuals sooner and more often, flooding the population. This potentially reduces the search space that can be reached within a given runtime. Yagoubi and Schoenauer attempt to circumvent this with a duration-based selection on the received offspring. This supposed defect can also be taken advantage of in various situations, one of which is evolving genetic programs, which must use a mechanism such as parsimony pressure or must minimize a size-related objective value to prevent any individual from becoming too large. The bias

provided by heterogeneous evaluation times can be used to produce an implicit time pressure; however, in cases with flat fitness landscapes, individuals tend to converge to both long and short evaluation times.

3. ASYNCHRONOUS PARALLEL CGP

Synchronous CGP, both serial and parallel, were implemented using the Standard CGP model, as defined by Miller, the only difference is that SPCGP evaluates all individuals of a generation simultaneously, while synchronous serial CGP evaluates only one individual at a time. SPCGP and APCGP both have a master node that generates new individuals that are later evaluated by slave nodes. SPCGP waits for all individuals in a generation to be returned, while APCGP acts on each individual as it is returned. In the case of APCGP, using the $(1 + 4)$ survival strategy advocated by Miller, the returned individual is compared to the existing best. If the new individual is better than or equal to the current best, it becomes the current best. Following this, a new individual is generated from the current best via mutation and the process continues until termination criteria are met. In this particular implementation, the evolutionary cycle terminates when the best individual has a fitness that exceeds a user-defined threshold. Although APCGP intuitively seems faster than SPCGP, the method by which APCGP explores the search space may lead to more evaluations until convergence. As seen in Figure 1, four individuals from the local search space of the current best individual are evaluated at each generation in SPCGP. In contrast to this, APCGP performs survival selection from only two individuals, and if a high-fitness solution has a long evaluation time, sub-optimal individuals will produce offspring to be evaluated while the high-fitness solution is being evaluated. An example of such an exploration in illustrated by Figure 2.

4. EXPERIMENTATION

4.1. PROBLEM

The problem chosen was n-bit parity, a classical digital circuit problem that CGP has been used to solve in the past. This was chosen as it has a known solution, allowing termination once correct. Although more computationally complex problems would benefit more from parallelization, CGP suffers from high variation, which becomes more pronounced as the problem complexity increases. Thus, to simulate more computationally complex problems and to reduce the effects of overhead due to parallelization, the fitness evaluation is configured to repeat any number of times.

4.2. EXPERIMENT DESIGN

The experiment was run with the parameters shown in Table 1, as recommended by Miller. n_i, the number of inputs, was equivalent to n for the n-bit parity problem trying to be solved (2 or 3). The function set was the bitwise functions {`nand`, `and`, `nor`, `or`} and thus the maximum parity of the functions, a, was two. The overhead, or the number of times the fitness evaluation was repeated, was varied between 1 and 400 to investigate performance based on problem complexity. 2-bit and 3-bit parity problems were run using a serial synchronous model, a parallel synchronous model, and an asynchronous parallel model. Each of these experiments was run thirty times. The parallel synchronous and parallel asynchronous models used a master/slave model, with one master thread and four slave threads. The implementation was done in Python, while parallel code was achieved using the multiprocess module.

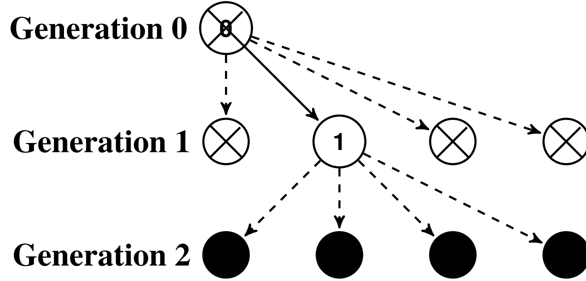


Figure 1. Exploration of Search Space in Synchronous CGP. The Best Individual of the Parent and its Four Children is Used for Producing the Next Generation.

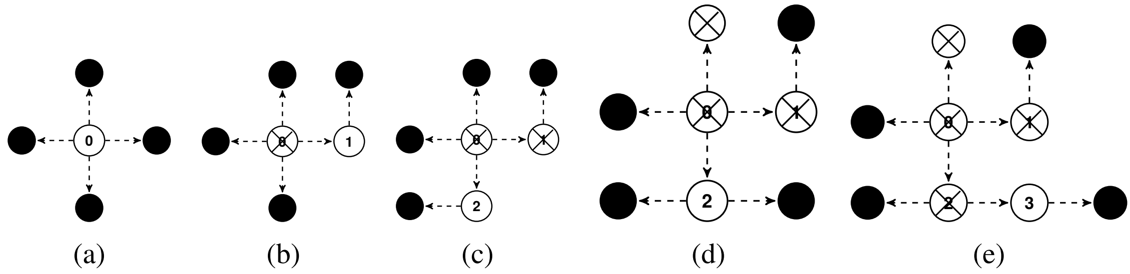


Figure 2. Exploration of Search Space in Asynchronous Parallel CGP. a) The initial state – Node 0 produces four children. b) Node 1 returns and is better than or equal to Node 0, Node 1 replaces Node 0 and produces a child. c) Node 2 returns and is better than or equal to Node 1, Node 2 replaces Node 1 and produces a child. d) One of the children from Node 0 finishes evaluating. It is worse than Node 2, so it is discarded and Node 2 produces a child. e) Node 3 returns and is better than or equal to Node 2. Node 3 replaces Node 2 and Node 3 produces a child. Note that one of the children from Node 0 is still being evaluated.

Table 1. Parameters Used for Experimentation

Parameter	Description	Value
n_c	Number of columns	4000
n_r	Number of rows	1
n_0	Number of outputs	1
l	Look back level	4000
μ	Population size	1
λ	Offspring size	4
μ_r	Mutation rate	0.01

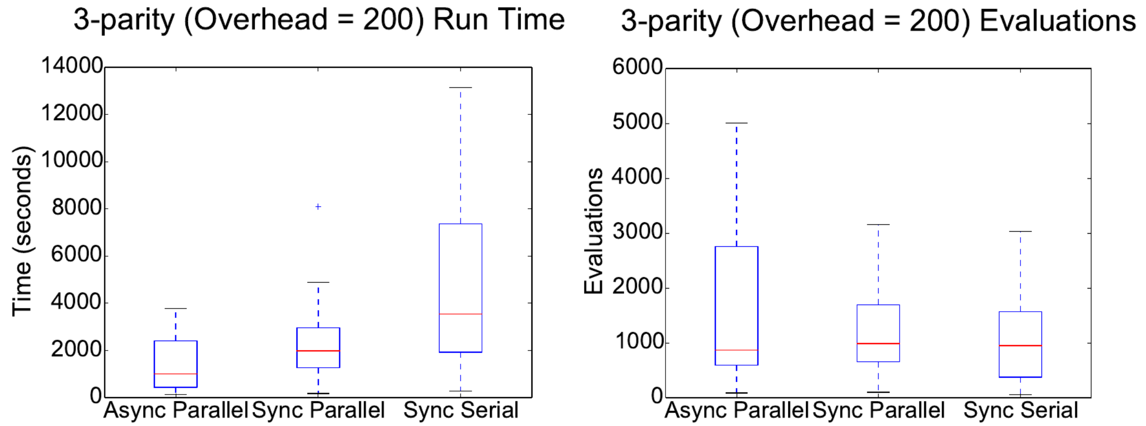


Figure 3. Results for 3-parity with an Overhead of 200 (Lower is Better)

5. RESULTS

As can be seen in Figure 3, asynchronous parallel and synchronous parallel models clearly have better run time averages than synchronous serial equivalent, while being close to each other in performance. The figure also indicates that asynchronous parallel takes more evaluations than synchronous parallel and synchronous serial, which are nearly identical in the regard. The statistical analysis of the results is shown in Table 2, indicating that there is statistical evidence that asynchronous parallel runs faster than synchronous parallel, while there does not seem to be strong statistical evidence that the number of evaluations differ.

Table 2. Statistical Analysis of 3-parity Results with an Overhead of 200

	Time (seconds)		**Evaluations**	
	Async Parallel	Sync Parallel	Async Parallel	Sync Parallel
Mean	1387	2272	1598	1197
Standard Deviation	1140	1551	1394	740
Equal Variance Assumed?	**No**		**No**	
t Stat		-2.5182		1.3915
Two-tailed p-value		0.0148		0.1711

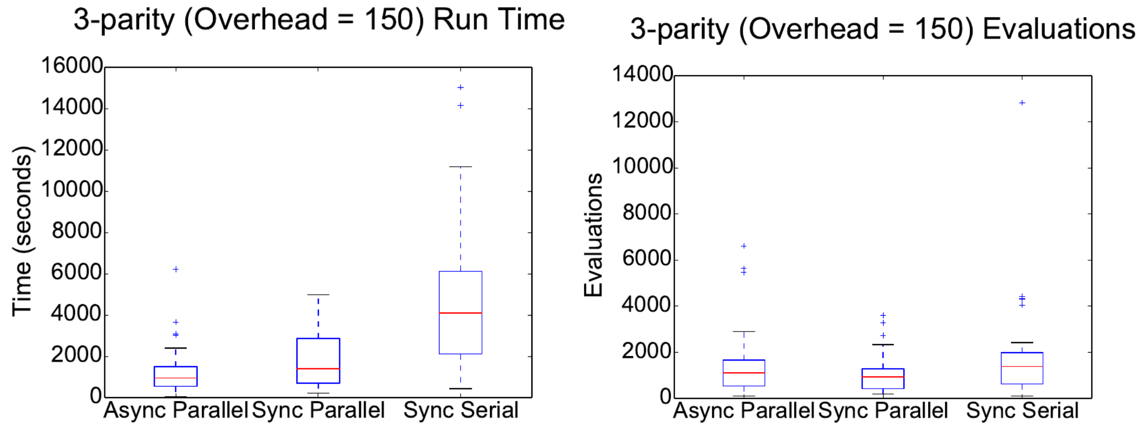


Figure 4. Results for 3-parity with an Overhead of 150 (Lower is Better)

Table 3. Statistical Analysis of 3-parity Results with an Overhead of 150

	Time (seconds)		**Evaluations**	
	Async Parallel	Sync Parallel	Async Parallel	Sync Parallel
Mean	1291	1843	1567	1107
Standard Deviation	1299	1404	1646	893
Equal Variance Assumed?	**No**		**No**	
t Stat		-1.5810		1.3445
Two-tailed p-value		0.1193		0.1856

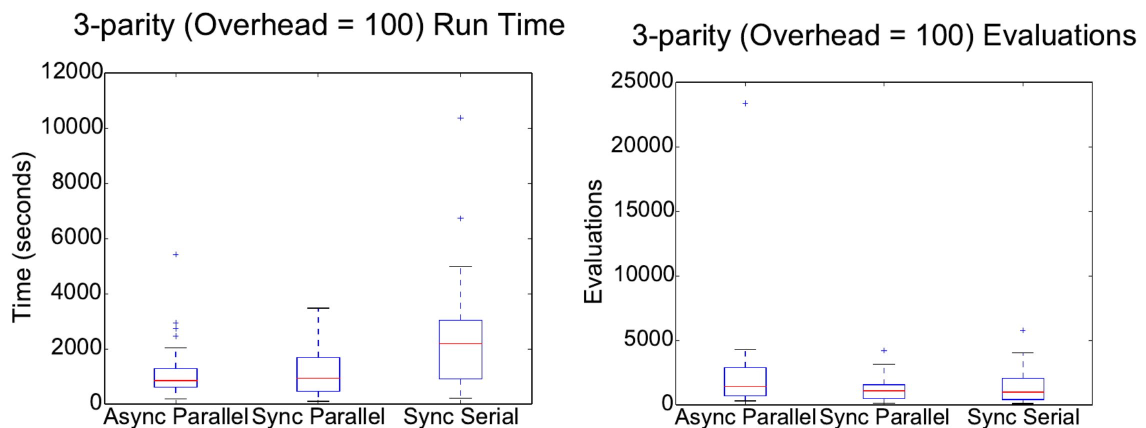


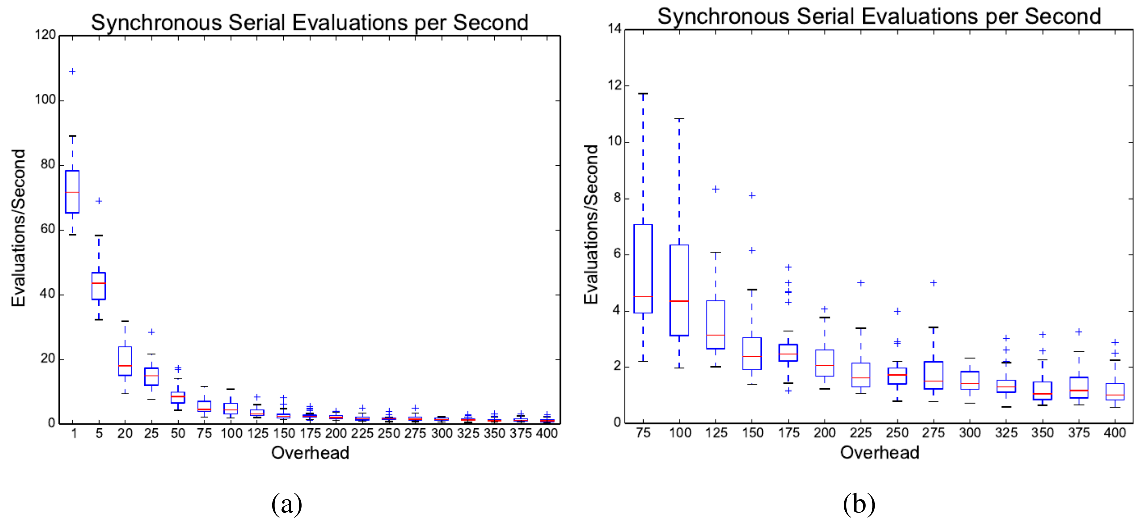
Figure 5. Results for 3-parity with an Overhead of 100 (Lower is Better)

Using an overhead of 150, shown in Figure 4 with statistical analysis shown in Table 3, there is not strong statistical evidence that the runtime or the number of evaluations differ. When the overhead is lowered to 100, shown in Figure 5 with statistical analysis

Table 4. Statistical Analysis of 3-parity Results with an Overhead of 100

	Time (seconds)		**Evaluations**	
	Async Parallel	Sync Parallel	Async Parallel	Sync Parallel
Mean	1180	1217	2493	1224
Standard Deviation	1060	899	4113	959
Equal Variance Assumed?	**No**		**No**	
t Stat		-0.1439		1.6453
Two-tailed p-value		0.8861		0.1097

shown in Table 4, there is no statistical evidence that there is a difference between the convergence time of APCGP and SPCGP, while there is still not strong statistical evidence that the number of evaluations differ.



(a) (b)

Figure 6. Evaluations per Second of Synchronous Serial with a Variety of Overheads for 2-bit Parity (Higher is Better). a) Overhead ranging from 1 to 400. b) Overhead ranging from 75 to 400.

As demonstrated in Figure 6, the synchronous serial model begins with a high evaluations/second rating, which quickly drops as the overhead increases. These results can be compared to those in Figure 7, asynchronous parallel and synchronous parallel both begin with lower evaluations/second, but the rate of decrease is substantially smaller in asynchronous parallel and synchronous parallel than in synchronous serial. Furthermore,

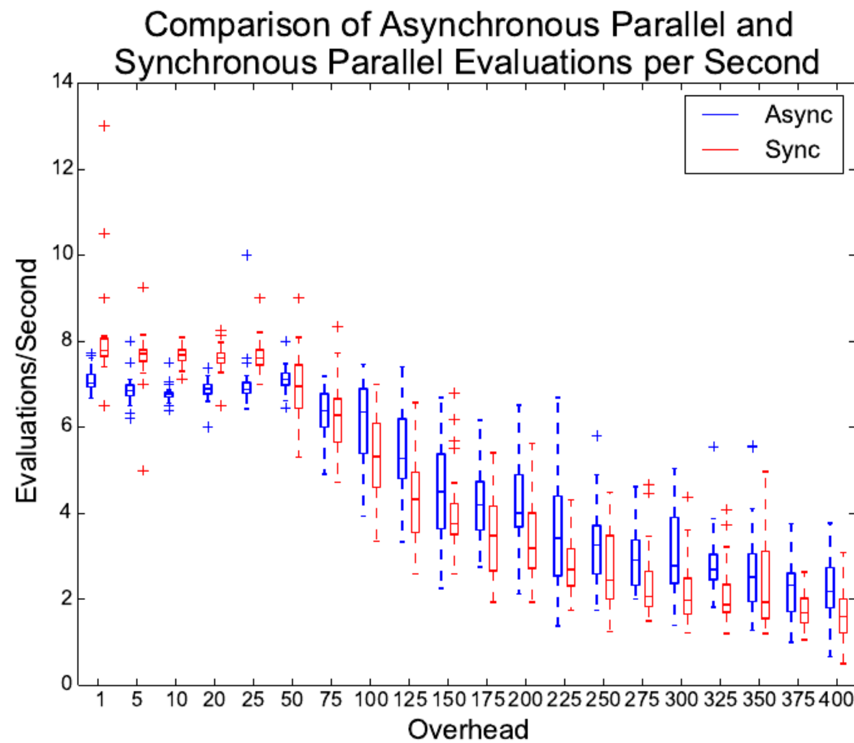


Figure 7. Evaluations per Second of Asynchronous Parallel and Synchronous Parallel with a Variety of Overheads for 2-bit Parity (Higher is Better)

Table 5. Statistical Analysis of 2-parity Results with an Overhead of 400

	Time (seconds)		**Evaluations**	
	Async Parallel	Sync Parallel	Async Parallel	Sync Parallel
Mean	79	228	187	261
Standard Deviation	62	172	167	238
Equal Variance Assumed?	**No**		**No**	
t Stat		-4.4609		-1.4025
Two-tailed p-value		0.0001		0.1667

as demonstrated by the statistical analysis with an overhead of 175, shown in Table 6, there is statistical evidence that APCGP is faster than SPCGP. This evidence is only strengthened as the overhead increases, demonstrated by the statistical analysis with an overhead of 400, showing strong statistic evidence that ASCGP is faster than SPCGP.

Table 6. Statistical Analysis of 2-parity Results with an Overhead of 175

	Time (seconds)		Evaluations	
	Async Parallel	Sync Parallel	Async Parallel	Sync Parallel
Mean	57	156	241	393
Standard Deviation	46	176	193	440
Equal Variance Assumed?	**No**		**No**	
t Stat		-2.9725		-1.7320
Two-tailed p-value		0.0055		0.0910

6. CONCLUSION

This paper has presented statistical evidence showing that APCGP outperforms SPCGP for computationally expensive tasks, while both outperform synchronous serial CGP; we hypothesize that the former is caused by greater heterogeneity in evaluation times. If the task is computationally inexpensive, then APCGP and SPCGP perform similarly, but both are inferior to serial CGP. This provides evidence that parallelization should only be performed if the task is computationally expensive, and when performed, an asynchronous model should be preferred.

7. FUTURE WORK

More advanced versions of CGP exist which exhibit superior performance to standard CGP on various important problems; applying the asynchronous model to them may further increase their performance. Although CGP showed improved performance, there are many forms of GP; these forms may not show the same increase in performance when using the asynchronous model. Additionally, the asynchronous model could be applied to different types of EAs, such as co-evolutionary EAs or multi-objective EAs. Although this study used CGP's traditional (1 + 4) population model for parallel synchronous, changing the number of offspring could potentially result in further improvements over synchronous

serial. In order to validate the hypothesis stated in the conclusion, that more computationally expensive tasks cause greater heterogeneity in evaluation times, the range of evaluation times should be diligently recorded and closely analyzed.

REFERENCES

Bertels, A. R. and Tauritz, D. R., 'Why Asynchronous Parallel Evolution is the Future of Hyper-heuristics: A CDCL SAT Solver Case Study,' in 'Proceedings of the 2016 on Genetic and Evolutionary Computation Conference Companion,' GECCO '16 Companion, ACM, New York, NY, USA, ISBN 978-1-4503-4323-7, 2016 pp. 1359–1365, doi:10.1145/2908961.2931729.

Churchill, A. W., Husbands, P., and Philippides, A., 'Tool Sequence Optimization using Synchronous and Asynchronous Parallel Multi-Objective Evolutionary Algorithms with Heterogeneous Evaluations,' in '2013 IEEE Congress on Evolutionary Computation (CEC),' IEEE, 2013 pp. 2924–2931.

Durillo, J. J., Nebro, A. J., Luna, F., and Alba, E., 'A Study of Master-Slave Approaches to Parallelize NSGA-II,' in 'IEEE International Symposium on Parallel and Distributed Processing,' IEEE, 2008 pp. 1–8.

Goldman, B. W. and Punch, W. F., 'Analysis of Cartesian Genetic Programming's Evolutionary Mechanisms,' IEEE Transactions on Evolutionary Computation, 2015, **19**(3), pp. 359–373, ISSN 1089-778X, doi:10.1109/TEVC.2014.2324539.

Harding, S. L., Miller, J. F., and Banzhaf, W., 'Self-modifying Cartesian Genetic Programming,' in 'Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation,' GECCO '07, ACM, New York, NY, USA, ISBN 978-1-59593-697-4, 2007 pp. 1021–1028, doi:10.1145/1276958.1277161.

Martin, M. A., Bertels, A. R., and Tauritz, D. R., 'Asynchronous Parallel Evolutionary Algorithms: Leveraging Heterogeneous Fitness Evaluation Times for Scalability and Elitist Parsimony Pressure,' in 'Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation,' GECCO Companion '15, ACM, New York, NY, USA, ISBN 978-1-4503-3488-4, 2015 pp. 1429–1430, doi:10.1145/2739482.2764718.

Miller, J., *Cartesian Genetic Programming*, Natural Computing Series, Springer-Verlag, Heidelberg, Berlin, 2000.

Scott, E. O. and De Jong, K. A., 'Evaluation-Time Bias in Asynchronous Evolutionary Algorithms,' in 'Proceedings of the Companion Publication of the 2015 on Genetic and Evolutionary Computation Conference,' ACM, New York, NY, USA, 2015 pp. 1209–1212.

Scott, E. O. and De Jong, K. A., ‘Evaluation-Time Bias in Quasi-Generational and Steady-State Asynchronous Evolutionary Algorithms,’ in ‘Proceedings of the Genetic and Evolutionary Computation Conference 2016,’ GECCO ’16, ACM, New York, NY, USA, ISBN 978-1-4503-4206-3, 2016 pp. 845–852, doi:10.1145/2908812.2908934.

Yagoubi, M. and Schoenauer, M., ‘Asynchronous Master/Slave MOEAs and Heterogeneous Evaluation Costs,’ in ‘Proceedings of the Fourteenth International Conference on Genetic and Evolutionary Computation Conference,’ ACM, 2012 pp. 1007–1014.

Yagoubi, M., Thobois, L., and Schoenauer, M., ‘Asynchronous Evolutionary Multi-Objective Algorithms with Heterogeneous Evaluation Costs,’ in ‘2011 IEEE Congress on Evolutionary Computation (CEC),’ IEEE, 2011 pp. 21–28.

II. EMPIRICAL EVIDENCE OF THE EFFECTIVENESS OF PRIMITIVE GRANULARITY CONTROL FOR HYPER-HEURISTICS

Adam Harter and Daniel R. Tauritz
Natural Computation Laboratory, Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, 65409

Aaron Scott Pope
Natural Computation Laboratory, Department of Computer Science, Missouri University of Science and Technology, Rolla, MO, 65409
Los Alamos National Laboratory, Los Alamos, NM, 87544

Chris Rawlings
Los Alamos National Laboratory, Los Alamos, NM, 87544

ABSTRACT

The set of primitive operations available to a generative hyper-heuristic can have a dramatic impact on the overall performance of the heuristic search in terms of efficiency and final solution quality. When constructing a primitive set, users are faced with a trade-off between generality and time spent searching. A set consisting of low-level primitives provides the flexibility to find most or all potential solutions, but the resulting heuristic search space might be too large to find adequate solutions in a reasonable time frame. Conversely, a set of high-level primitives can enable faster discovery of mediocre solutions, but prevent the fine-tuning necessary to find the optimal heuristics. By varying the set of primitives throughout evolution, the heuristic search can utilize the advantages of both high-level and low-level primitive sets. This permits the heuristic search to either quickly traverse parts of the search space as needed or modify the minutiae of the search to find optimal solutions in reasonable amounts of time not feasible with implicit levels of primitive granularity. This paper demonstrates this potential by presenting empirical evidence of improvements to

solvers for the Traveling Thief Problem, a combination of the Traveling Salesman Problem and the Knapsack Problem, a recent and difficult problem designed to more closely emulate real world complexity.

Keywords: Genetic Programming, Hyper-heuristics, Evolutionary Computing

1. INTRODUCTION

Unlike a traditional search, which aims to find a high-quality solution for the particular instance of a problem, a hyper-heuristic search instead seeks to find an algorithm that produces high-quality solutions to a specific problem class. This can be accomplished through one of two means, heuristic selection or heuristic generation. Heuristic selection, as its name implies, selects a solution heuristic from a pool of potential candidate solutions that best fits the application. This approach can be powerful, but it relies on having a high-quality set of available candidate heuristics a priori.

Generative hyper-heuristics instead aim to construct novel heuristics that are tailored to the specific target application. Genetic programming (GP) is a common generative hyper-heuristic technique that relies on an evolutionary search to generate and optimize executable program solutions. The evolutionary search has more effective genes and operations propagate from generation to generation while less effective genes tend to be removed, allowing quality heuristics to be generated over time. Conventionally, the fundamental set of operations used to construct heuristic solutions is generated by extracting a set of basic functions from existing techniques related to the application. For instance, in symbolic regression applications, the primitive set typically consists of arithmetic operations (e.g., addition).

The proper construction of the set of primitive operations is critical to the success of a hyper-heuristic application. If crucial operations are not present, the approach will not be expressive enough to produce high-quality solutions. Alternatively, if the primitive set

is bloated with irrelevant operations, a substantial amount of search time will be wasted on useless solutions.

Even if the crucial operational elements can be identified, the level of primitive granularity can still have a dramatic effect on the search efficiency. A set of high-level primitives may lead to faster convergence, but be incapable of the fine-tuning needed to find optimal solutions. Conversely, a set of low-level primitives may be able to find optimal solutions, but take an unacceptably long time to converge. Carefully selecting the proper level of primitives requires a great deal of time, specific domain knowledge, and human expertise. But even with those prerequisites met, the optimal set of primitives is likely to change as the search advances, making human intervention infeasible and ineffective.

This work investigates the impact of dynamically changing the level of primitive granularity during the hyper-heuristic search. A meta-level search is used to find schedules for controlling the level of primitive granularity that improve over static configurations. To demonstrate potential improvements, solvers for the Traveling Thief Problem (TTP) were evolved using both static and dynamic primitive sets and the best configurations were compared in runtime, average fitness, and maximum fitness.

2. RELATED WORK

Hyper-heuristics and evolutionary algorithms have both been successfully applied to the traveling salesman problem and the knapsack problem in the past. Additionally, most methods for solving TTP are partially or entirely based on evolutionary methods . Previously, a GP approach utilizing a higher level primitive set was used to create TTP solvers that sometimes outperformed current state-of-the-art solvers. Martin and Tauritz and Pope et al. previously demonstrated that adding lower level primitives to a primitive set can increase the fitness at the cost of increasing the runtime. A similar approach was previously used by Goldman and Tauritz to demonstrate the effectiveness of other dynamic parameters. In that work, different parameters, such as the population size, number of children, etc.

were changed throughout evolution by using a vector of values for specific generations and interpolating for values between the generations. The dynamic configurations found showed improvements in fitness when given an equivalent amount of time to run.

This work can be viewed as somewhat oppositionary to previous methods of finding reusable blocks of code as primitives during evolution, such as Evolutionary Module Acquisition, Hierarchy Locally Defined Modules, and Adaptive Representation. Each of these examines the population, searching for reoccurring blocks of code that can be used as primitives while generating new individuals in later generations. A related, more recent approach, Emergent Tangled Graph Representations, introduced by Kelly and Heywood, approaches that problem differently. This method utilizes small programs grouped together as teams and uses the output of these teams as a part of other teams. Evolution develops not only the higher-level teams and the programs within them, but also the links between different teams.

3. PRIMITIVE GRANULARITY CONTROL

In a conventional GP application, the set of primitive operations available to the search is decided a priori and does not change over the course of evolution. The construction of the primitive set has the potential to bias the search and have a significant impact on the performance of the GP. Practitioners can include complex primitives that have some key functionality that is targeted at the application in question. A set of such high-level operations can allow a GP to quickly find complex solutions that perform well. Unfortunately, these complex operations typically come in an “all or nothing” form. If an optimal solution requires a small modification to the provided functionality, the high-level primitive set might prevent the necessary fine-tuning.

Alternatively, a set of primitives with more basic functionality can result in a GP with a far greater range of algorithmic expression. However, this improved flexibility can come at the cost of a dramatically increased search complexity as the GP must “reinvent

the wheel" to achieve more complex functionality. Primitive granularity control (PGC), a technique proposed in this work, aims to leverage the benefits of both the high-level and low-level approaches.

A set of low-level primitives is extracted from previous methods that target the TTP. More complex operations, referred to as macro primitives, are then constructed manually from the basic primitives. This process can be repeated, incorporating macro primitives within other macro primitives to achieve even more complex functionality. All operations in the primitive set are assigned a numerical "coarseness level" that indicates their relative complexity. Basic primitives are assigned a coarseness level of one, and macro primitives are assigned a level of one greater than the highest operation they contain. For instance, a macro primitive that contains only basic primitives will have a coarseness level of two; any macro primitive that contains this level two primitive will have a coarseness level of at least three.

To leverage these coarseness indicators, the GP is provided a schedule that controls the level of coarseness available in the primitive set at any given point during evolution. This schedule restricts the primitive set used during population initialization (i.e., parse tree generation) and within the variation operators (i.e., mutation and recombination). If the schedule lowers the coarseness level below that of any primitives present within the solutions in a population, these macro operations are replaced with the lower-level subtrees that provide the same functionality. See Section 3 for an example parse tree presented at three coarseness levels.

The goal of this preliminary work is to investigate the potential for GP performance improvements when a dynamic schedule is used to control the level of primitive coarseness. A subset of all possible coarseness schedules was considered in an exhaustive meta-level search. The best performing dynamic schedules (i.e., schedules with at least one change in coarseness levels) were compared to the best static schedules found.

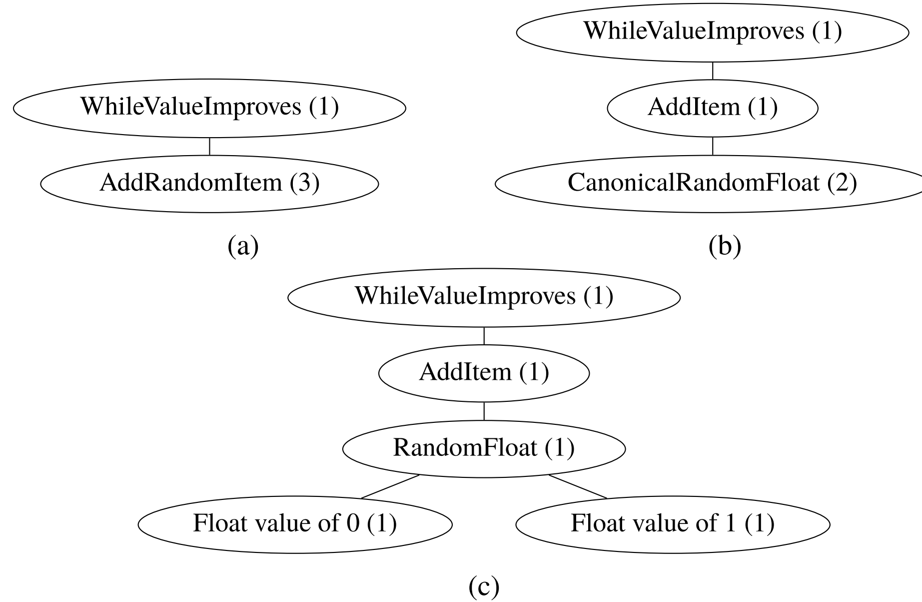
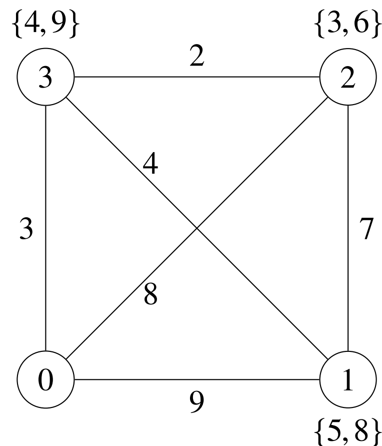


Figure 1. Example TTP Heuristic at Three Primitive Coarseness Levels. The coarseness of the primitives is indicated by the number in parentheses after the primitive name. a) Coarseness level 3 b) Coarseness level 2 c) Coarseness level 1

4. TRAVELING THIEF PROBLEM

The Traveling Thief Problem (TTP) is a combination of two NP-hard problems, the Traveling Salesman Problem and the Knapsack Problem, designed to more closely emulate real-world problems by having the two sub-problems interact in complex and non-trivial ways. A TPP instance consists of a list of cities and a list of items. Each item has a weight, a value, and a location, while each pair of cities has a distance between them. A solution consists of a path that visits each city exactly once, ending with returning to the starting city, and a picking plan of which items to take. There is a maximum weight of items that can be taken, and the time taken to travel between cities scales linearly with the ratio of the sum of the weight of the items carried to the maximum weight. The solution value is the total worth of the picking plan subtracted by the travel time multiplied by a constant specified by the instance known as the renting ratio. An extremely simple example TTP instance can be seen in Figure 2.



The value pair at each city other than 0 is the item present at that city, denoted as {value, weight}. The value for each edge is the distance between the cities. The tour must begin at city 0.

Figure 2. TTP Example Instance

TTP was chosen as a test ground for PGC as simpler problems, such as those in the general program synthesis benchmark proposed by Helmuth and Lee, typically do not require primitives that are complex enough to be implemented at multiple levels of coarseness. TTP is a modern, difficult to solve problem, with even small instances not having known optimal solutions. It has also enjoyed a great amount of attention from the field of evolutionary computation in general. These approaches generally start by finding a good starting TSP solution, usually using the Lin-Kernighan heuristic, and then modifying either only the picking plan or both the picking plan and the path. To minimize the risk of starting in local optima, the GP solvers in this work begin with random initial paths and an empty picking plan.

5. METHODOLOGY

In PGC, the coarseness level is varied throughout evolution; these configurations are referred to as dynamic plans. To test the effectiveness of different dynamic plans, an

exhaustive search was performed over a subset of all possible configurations. Each dynamic plan was evaluated with a GP search for effective heuristics for a TTP instance. A high level overview of the process can be seen in Figure 3.

5.1. META-SEARCH

Each dynamic and static configuration consisted of a tuple of a user-defined length s, each value in the tuple representing an overall coarseness level. All experiments in this paper use a tuple of length 5, chosen as a balance between limiting the search time while still allowing room for improvement. The static configurations are represented as all members of the tuple being the same value. Given N generations, the target coarseness would change every $\lfloor N/s \rfloor$ generations until the last segment was reached. Due to the strong typing of the tree, primitives for the coarseness level were not always available; in this situation, the coarseness level was temporarily and repeatedly lowered by one until a primitive matching that coarseness level was available, this procedure is shown in Algorithm 1. In essence, this means that for each type and coarseness level there is an effective coarseness level for terminals and non-terminals, which are shown in Table 1 and Table 2, respectively. The tables show a mapping from the overall coarseness level to a type specific coarseness level; for example, when generating an *int* terminal, any overall coarseness level of two or higher results in a primitive of coarseness two being generated. Each of these dynamic plans were evaluated 30 times for statistical purposes.

5.2. TTP HEURISTIC EVOLUTION

The heuristics for the TTP problems were represented as strongly typed Koza-style GP Trees. Population initialization was performed using a ramped half-and-half approach. The list of basic primitives and macro primitives can be seen in Table 3 and Table 4,

Algorithm 1 Terminal and Non-terminal Filter Process

```
procedure FilterPrimitives(coarseness, target)
    primitiveSet ← AddPrimitives(coarseness, target)
    if primitiveSet is empty then
        if target = Terminal then
            primitiveSet ← AddPrimitives(coarseness, Non-terminal)
        else
            primitiveSet ← AddPrimitives(coarseness, Terminal)
    return primitiveSet

procedure AddPrimitives(coarseness, target)
    toAdd ← ∅
    while toAdd is empty and coarseness > 0 do
        Add primitives of type target with
                coarseness level of coarseness to toAdd
        coarseness ← coarseness − 1
    return toAdd
```

Table 1. Terminal Primitive Effective Coarseness

Generated Type	Coarseness 1	2	3	4	5
int	1	2	2	2	2
float	1	2	2	2	2
worker	1	1	3	3	3
float_list	-	-	-	-	-
bool	1	1	1	1	1

No terminal primitives are of type *float_list*.

Table 2. Non-terminal Primitive Effective Coarseness

Generated Type	Coarseness 1	2	3	4	5
int	1	1	1	1	1
float	1	1	1	1	1
worker	1	2	3	4	5
float_list	1	1	1	1	1
bool	1	1	1	1	1

respectively. The solvers start with a random initial path and an empty picking plan, and can manipulate both. The strongly-typed parse tree implementation requires all primitives have an associated type; the list of available types is as follows:

float Floating point number

int Integer number

bool Boolean value

float_list Finite list of floating point numbers

worker Program control operators and operations that manipulate the path and picking plan

Parse trees must have a *worker* type primitive as their root. Evaluation is performed against a single TTP instance at a time, and the algorithm for evaluation of an individual can be seen in Algorithm 2. Crossover was a standard sub-tree crossover, while mutation could either replace a subtree with a randomly generated one or with one of its children. Survival selection and parent selection were both k-tournament, with the fitness of the individuals being penalized by the number of nodes in its representation to encourage efficient solutions. Evolution was performed for a set number of generations with a set population size and number of children generated. A set number of generations was utilized instead of running to convergence to reduce the runtime of the system.

6. EXPERIMENTATION

Even the simplest instances in the current standard benchmark suite for TTP were computationally infeasible due to the exhaustive nature of the search. Therefore, three new, smaller, individual problems were created: a 10 city problem, a 12 city problem and a 26 city problem, these instances are available at https://github.com/dtauritz/NC-LAB-Public.

Table 3. List of Basic Primitives

Primitive	Signature	Description
CurrentSolutionValue	float()	The current solution's value
ValueChange	float()	Change in value from the last path/item change
Negate	int(int) float(float)	Negates a value
Velocity	float()	Final velocity of the thief with the current picking plan
RandomBool	bool(float)	Random bool with specified probability of being true
RandomInt	int(int, int)	Random integer within the given range
RandomFloat	float(float, float)	Random float within the given range
LoopVariableInt	int()	Variable used for looping
MapValueIndex	int(float_list)	Index of maximum value of a list
MaxValue	float(float_list)	Maximum value of a list
MapNodes	float_list(float)	Evaluate a subtree for each node in the path with the loop variable set to the index of the city
DoNothing	worker()	Does nothing
ChainWork	worker(worker, worker) worker(worker, worker, worker)	Chains work to be performed one after the other
IfStatement	int(bool, int, int) float(bool, float, float) worker(bool, worker, worker)	Evaluates a boolean expression and evaluate and return the first argument if true, or the second argument if false
LKGain	float(int)	Returns the gain of performing an LKSwap at the specified location
LKTransform	worker(int)	Performs an LKSwap at the specified location
TwoOptTransform	worker(int, int)	Performs a two-opt transform at the specified location
SwapCities	worker(int, int)	Swaps two cities in the path
Distance	float(int, int)	Returns the distance between two cities
AddItem	worker(int) worker(float)	Sets the loop variable equal to each item outside the bag that can fit in the bag and evaluates the child tree, placing the item the produces the largest value in the bag
RemoveItem	worker(int) worker(float)	Similar to AddItem, but removes an item instead
ItemWeight	int(int)	Item weight at the specified index
ItemValue	int(int)	Item value at the specified index
ItemRatio	float(int)	Cost/weight ratio of the item at the specified index
ItemLocation	int(int)	City index where the specified item is found
EffectiveItemValue	float(int)	Solution's value changed by adding the specified item
WhileValueImproves	worker(worker)	Evaluate the child tree until it does not improve the solution's value
SavePath	worker()	Append the current path to the saved paths
SaveItems	worker()	Append the current picking plan to the saved picking plans
RestorePath	worker()	Restore the latest saved path
RestoreItems	worker()	Restore the latest saved picking plan
GetFirstImprovementForPath	worker(worker)	For each city in the path in order, evaluates the child tree, setting the loop variable to the city, and exiting the loop early if an improvement is made to the solution
IntToFloat	float(int)	Converts an integer to a float
FloatToInt	int(float)	Converts a float to an integer
$+, -, *$	int(int) float(float)	Standard mathematical arithmetic.
SafeDivide	int(int) float(float)	If the divisor is zero, return zero, otherwise standard division.
$>, =$	bool(int, int) bool(float, float)	Standard comparison operators.
And, Or	bool(bool, bool)	Standard boolean operators.
Not	bool(bool)	Standard boolean operator.

Table 4. List of Macro Primitives

Primitive	Coarseness	Signature	Description
MaxLKGain	2	float()	The max LKGain for the current path
MaxLKGainIndex	2	int()	Index for an LKSwap for maximum LKGain
LinKernighan0	3	worker()	Performs Lin-Kernighan with no look-back
LinKernighan1	4	worker()	Performs Lin-Kernighan with one depth look-back
LinKernighan2	5	worker()	Performs Lin-Kernighan with two depth look-back
SavePathAndItems	2	worker()	Saves the current path and picking plan
RestorePathAndItems	2	worker()	Restores the most recently saved path and picking plan
KeepIfImproves	3	worker(worker)	Evaluates a child tree, discarding the changes it did not improve the solution
GreedyKPSearch	2	worker()	Adds items that increase the solution value the most until no items fit or the value fails to improve
RemoveHeaviest	2	worker()	Removes the heaviest item from the bag
CanonicalRandomFloat	2	float()	Generates a float in the range [0, 1)
AddRandomItem	3	worker()	Adds a random item to the bag
RemoveRandomItem	3	worker()	Removes a random item from the bag

Algorithm 2 TTP Individual Evaluation

```
value ← 0
outerStagnantCount ← 0
outerBestValue ← −∞
bestPath ← default path
bestPickingPlan ← default picking plan
while time remains and
        outerStagnantCount < Outer Stagnant Limit do
    path ← Random Initial Path
    pickingPlan ← Empty Plan
    innerStagnantCount ← 0
    innerBestValue ← −∞
    while innerStagnantCount < Inner Stagnant Limit do
        if no time remains then
            return outerBestValue, bestPath, bestPickingPlan
        Evaluate individual, changing path and pickingPlan
        value ← TTP value using path and pickingPlan
        if value > innerBestValue then
            innerBestValue ← value
            innerStagnantCount ← 0
            bestPath ← path
            bestPickingPlan ← pickingPlan
        else
            innerStagnantCount ← innerStagnantCount + 1
    if value > outerBestValue then
        outerBestValue ← value
        outerStagnantCount ← 0
    else
        outerStagnantCount ← outerStagnantCount + 1
return outerBestValue, bestPath, bestPickingPlan
```

Table 5. TTP Solver Specific Parameters

Parameter	Value
Population Size	24
Number of Children	48
Number of Generations	60
Survival Strategy	$(\mu + \lambda)$
Minimum Initial Depth	3
Maximum Depth	6
Mutation Minimum Depth	1
Mutation Maximum Depth	2
Inner Stagnant Limit	5
Outer Stagnant Limit	5
Evaluation Time Limit	*Varied with Problem*
Tournament Size	3

Each of these problems has a single item at each location, excluding the starting city. Parameters specific to the TTP solvers can be found in Table 5, which were manually fine tuned. The evaluation time limit was set to 0.5ms for the 10 city problem, 1.5ms for the 12 city problem, and 5ms for the 26 city problem; these time limits were hand tuned to balance reducing the runtime of the meta-search and providing sufficient time for finding quality TTP solutions.

7. RESULTS

All statistical data can be found in Table 6 and Figure 5. A graphical comparison of the runtime versus the maximum fitness can be seen in Figure 4. For each problem instance and each optimization target, PGC produced improved results when compared against static primitive sets. For the 10 city problem, PGC completed in less time when optimizing for mean fitness and runtime, always improved on the mean fitness, and improved on maximum fitness when prioritizing runtime. For the 12 city problem, PGC produced

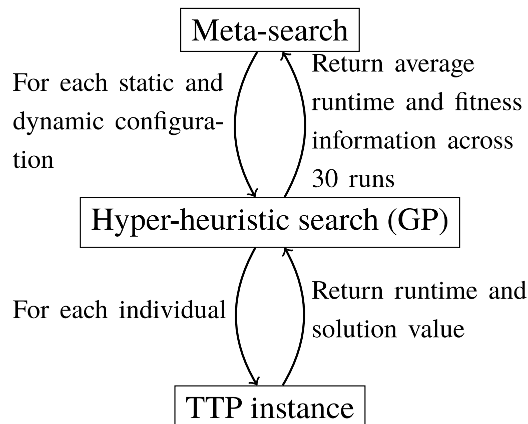


Figure 3. High Level Overview of Experiment

worse mean fitnesses for the mean and maximum fitness configurations, but the maximum fitness was unaffected, and PGC always produced shorter runtimes. For the 26 city problem, PGC completed in less time when optimizing for fitness, and outperformed for mean and maximum fitness when optimizing for time.

Even with a small number of coarseness levels and a small number of points where the coarseness level was changed, PGC still demonstrated improvements. This is despite the coarseness levels being heavily distributed towards *worker* primitives, as the majority of macro primitives were *worker* primitives. However, this is likely not as big of a problem as it initially seems. Primitives of type *worker* are the most important primitives as they actually operate on the solution; all the other primitives types are simply inputs, parameters, etc. The primary improvement demonstrated by PGC compared to static granularity is in reaching the same fitness in less time or reaching greater fitness in the same amount of time.

8. CONCLUSION

This paper presented empirical evidence that dynamic primitive granularity (referred to as coarseness levels in this work) has the potential to outperform static primitive granularity (standard GP). Using an exhaustive search, dynamic sets of primitives were found

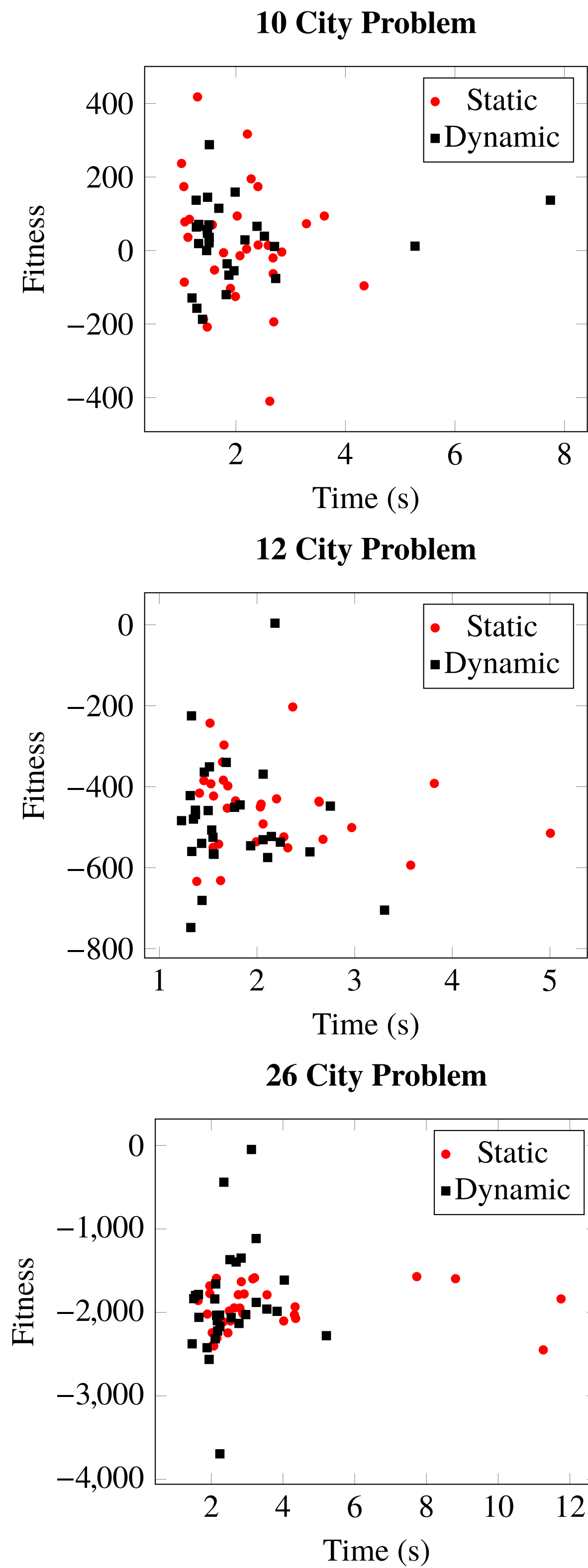


Figure 4. Max Fitness Versus Runtime for Each Run of the Best Dynamic and Static Plans for Maximum Fitness

Table 6. Comparison of Static and Dynamic Configurations

Best Configuration For	Mean Value Across 30 Evaluations							
	Mean Fitness		Max Fitness		Runtime		Coarseness Level	
	Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic
10 City Problem								
Mean Fitness	-502.369	**-430.871**	16.976	-10.537	2.081	**1.646**	1	[1, 3, 1, 1, 3]
Max Fitness	-502.369	**-432.732**	16.976	26.578	2.081	2.019	2	[2, 1, 2, 5, 3]
Runtime	-714.276	**-556.406**	-181.861	**-67.854**	1.762	**1.440**	5	[5, 5, 1, 2, 3]
12 City Problem								
Mean Fitness	**-826.416**	-877.186	-452.449	-503.021	2.146	**1.793**	2	[5, 2, 2, 2, 1]
Max Fitness	**-826.416**	-987.628	-452.449	-481.590	2.146	**1.759**	2	[5, 1, 1, 5, 5]
Runtime	-873.578	-912.443	-516.237	-522.040	2.042	**1.588**	1	[4, 1, 2, 1, 3]
26 City Problem								
Mean Fitness	-2660.408	-2679.844	-1946.679	-1885.881	3.701	**2.538**	2	[2, 4, 2, 1, 1]
Max Fitness	-2660.408	-2679.844	-1946.679	-1885.881	3.701	**2.538**	2	[2, 4, 2, 1, 1]
Runtime	-3497.207	**-3098.771**	-2808.712	**-2351.676**	2.516	2.240	5	[1, 1, 2, 3, 4]

Better values that are statistically significant using the Student's T-test with $\alpha = 0.05$ are in **bold**.

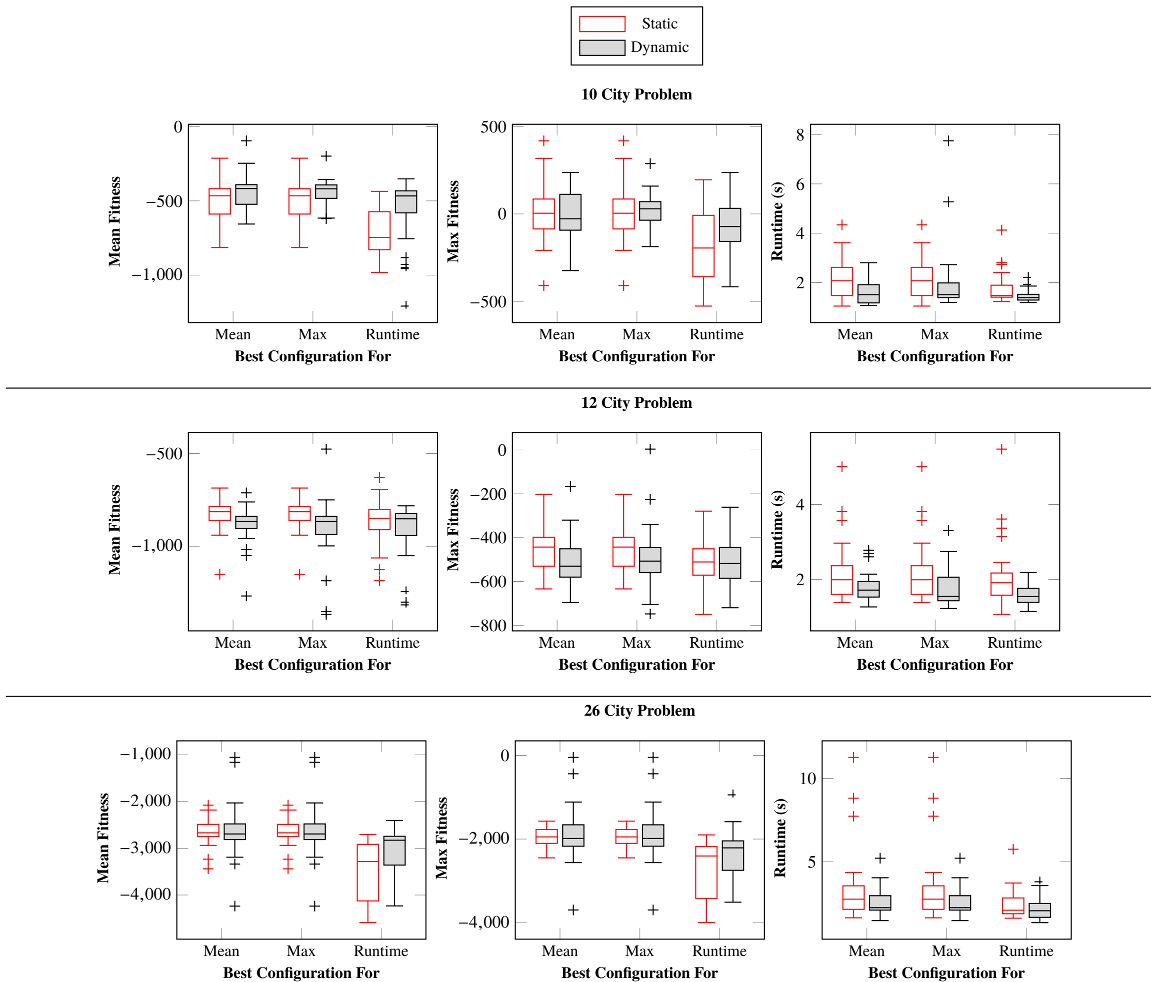


Figure 5. Comparison of Each Best Configuration

that in the majority of cases exceeded or met the performance of static ones in measures of runtime, average fitness, and maximum fitness. For easier problems, improvements were mainly found in the fitness measures, while for more complex problems, improvements were found in runtime. PGC may be expected to have the capability to improve runtime and solution quality in other complex problems as well. While the results presented here prove our hypothesis of dynamic primitive granularity outperforming static primitive granularity, the exhaustive search employed is not practical for real world use, thus motivating future research to create an efficient control method for PGC.

9. FUTURE WORK

A method to create dynamic primitive granularity plans without significant runtime overhead may be expected to result in GP having reduced runtime, improved solution quality, or both. If a method is found, automated generation of higher level primitives (composition) would further reduce the need for domain-specific expertise. The reverse, automated decomposition of higher level primitives into simpler primitives, would also be beneficial, because it would allow fine tuning of individuals without requiring a priori human specification of coarseness levels. Closer examination of the convergence and other factors of the dynamic and static configurations may also provide additional insight on PGC.

An extension to PGC could be changing the coarseness level to a range or set of allowed coarseness levels, allowing more fine-tuned control. The use of primitives with higher coarseness levels that can not be decomposed may also have use in PGC; such primitives may exist due to infeasibility of representation with simpler primitives, but the work performed is non-trivial. More sophisticated methods of assigning coarseness levels may also be worth examining, such as changing the level based on stagnation, rate of fitness change, or other population measures. Additionally, each individual generated type could have its own coarseness level, which may result in further benefits. The fact that the small

number of effective coarseness levels for each primitive type already resulted in noticeable improvements indicates the high likelihood that providing a richer set of macro primitives would yield further improvements.

ACKNOWLEDGMENTS

This work was supported by Los Alamos National Laboratory via the Cyber Security Sciences Institute under subcontract 259565 and the Laboratory Directed Research and Development program of Los Alamos National Laboratory under project number 20170683ER.

REFERENCES

Angeline, P. J. and Pollack, J., 'Evolutionary Module Acquisition,' in 'Proceedings of the second annual conference on evolutionary programming,' Citeseer, 1993 pp. 154–163.

Aziz, Z. A., 'Ant Colony Hyper-heuristics for Travelling Salesman Problem,' Procedia Computer Science, 2015, **76**, pp. 534–538.

Banzhaf, W., Banscherus, D., and Dittrich, P., *Hierarchical Genetic Programming Using Local Modules*, Secretary of the SFB 531, 1999.

Blank, J., Deb, K., and Mostaghim, S., 'Solving the Bi-objective Traveling Thief Problem with Multi-objective Evolutionary Algorithms,' in 'International Conference on Evolutionary Multi-Criterion Optimization,' Springer, 2017 pp. 46–60.

Bonyadi, M. R., Michalewicz, Z., and Barone, L., 'The Travelling Thief Problem: The First Step in the Transition from Theoretical Problems to Realistic Problems,' in 'Evolutionary Computation (CEC), 2013 IEEE Congress on,' IEEE, 2013 pp. 1037–1044.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., and Qu, R., 'Hyper-heuristics: A survey of the state of the art,' Journal of the Operational Research Society, 2013, **64**(12), pp. 1695–1724.

Burke, E. K., Hyde, M. R., Kendall, G., and Woodward, J., 'Automating the Packing Heuristic Design Process with Genetic Programming,' Evolutionary computation, 2012, **20**(1), pp. 63–89.

Drake, J. H., Hyde, M., Ibrahim, K., and Ozcan, E., 'A Genetic Programming Hyper-heuristic for the Multidimensional Knapsack Problem,' Kybernetes, 2014, **43**(9/10), pp. 1500–1511.

El Yafrani, M. and Ahiod, B., 'Population-based vs. Single-solution Heuristics for the Travelling Thief Problem,' in 'Proceedings of the Genetic and Evolutionary Computation Conference 2016,' ACM, 2016 pp. 317–324.

El Yafrani, M., Martins, M., Wagner, M., Ahiod, B., Delgado, M., and Lüders, R., 'A Hyperheuristic Approach Based on Low-level Heuristics for the Travelling Thief Problem,' Genetic Programming and Evolvable Machines, 2018, **19**(1-2), pp. 121–150.

Fogel, D. B., 'Applying Evolutionary Programming to Selected Traveling Salesman Problems,' Cybernetics and systems, 1993, **24**(1), pp. 27–36.

Goldman, B. W. and Tauritz, D. R., 'Meta-evolved Empirical Evidence of the Effectiveness of Dynamic Parameters,' in 'Proceedings of the 13th annual conference companion on Genetic and evolutionary computation,' ACM, 2011 pp. 155–156.

Helmuth, T. and Spector, L., 'Detailed Problem Descriptions for General Program Synthesis Benchmark Suite,' Technical report, Technical Report UM-CS-2015-006, School of Computer Science, University of Massachusetts Amherst, 2015.

Hough, P. D. and Williams, P. J., 'Modern Machine Learning for Automatic Optimization Algorithm Selection,' in 'Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop,' 2006 pp. 1–6.

Kelly, S. and Heywood, M. I., 'Emergent Tangled Graph Representations for Atari Game Playing Agents,' in 'European Conference on Genetic Programming,' Springer, 2017 pp. 64–79.

Kendall, G. and Li, J., 'Competitive Travelling Salesmen Problem: A Hyper-heuristic Approach,' Journal of the Operational Research Society, 2013, **64**(2), pp. 208–216.

Koza, J. R., *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, MIT Press, Cambridge, MA, USA, 1992, ISBN 0-262-11170-5.

Kumar, R., Joshi, A. H., Banka, K. K., and Rockett, P. I., 'Evolution of Hyperheuristics for the Biobjective 0/1 Knapsack Problem by Multiobjective Genetic Programming,' in 'Proceedings of the 10th annual conference on Genetic and evolutionary computation,' ACM, 2008 pp. 1227–1234.

Lin, S. and Kernighan, B. W., 'An Effective Heuristic Algorithm for the Traveling-salesman Problem,' Operations research, 1973, **21**(2), pp. 498–516.

Martin, M. A. and Tauritz, D. R., 'Hyper-heuristics: A Study on Increasing Primitive-space,' in 'Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation,' ACM, 2015 pp. 1051–1058.

Mei, Y., Li, X., Salim, F., and Yao, X., ‘Heuristic Evolution with Genetic Programming for Traveling Thief Problem,’ in ‘Evolutionary Computation (CEC), 2015 IEEE Congress on,’ IEEE, 2015 pp. 2753–2760.

Parada, L., Herrera, C., Sepúlveda, M., and Parada, V., ‘Evolution of New Algorithms for the Binary Knapsack Problem,’ Natural Computing, 2016, **15**(1), pp. 181–193.

Polyakovskiy, S., Bonyadi, M. R., Wagner, M., Michalewicz, Z., and Neumann, F., ‘A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem,’ in ‘Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation,’ ACM, 2014 pp. 477–484.

Pope, A. S., Tauritz, D. R., and Kent, A. D., ‘Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity,’ in ‘2016 IEEE Symposium Series on Computational Intelligence (SSCI),’ IEEE, 2016 pp. 1–8, doi: 10.1109/SSCI.2016.7849929.

Rosca, J. P. and Ballard, D. H., ‘Hierarchical Self-organization in Genetic Programming,’ in ‘Machine Learning Proceedings 1994,’ pp. 251–258, Elsevier, 1994.

Ryser-Welch, P., Miller, J. F., and Asta, S., ‘Generating Human-readable Algorithms for the Travelling Salesman Problem using Hyper-heuristics,’ in ‘Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation,’ ACM, 2015 pp. 1067–1074.

Wu, J., Polyakovskiy, S., Wagner, M., and Neumann, F., ‘Evolutionary Computation plus Dynamic Programming for the Bi-Objective Travelling Thief Problem,’ arXiv preprint arXiv:1802.02434, 2018.

SECTION

2. SUMMARY AND CONCLUSIONS

This thesis presented two papers introducing distinct methods for improving the canonical genetic programming algorithm. The first paper introduces Asynchronous Parallel Evolutionary Algorithms (APEA) for Cartesian Genetic Programming (CGP), which reduces runtime by eliminating the bottleneck caused by individuals with exceptionally long evaluation times. This allows more effective use of compute resources, at the cost of increasing the complexity of implementation. The benefits of APEA for CGP were increasing the number of evaluations per second and as a result completing the search more quickly. The decreased runtime was, however, limited to more computationally complex problems; for computationally cheap problems, the overhead of parallelization was too great to overcome. CGP also has more advanced variations than the basic one used that could reap even better benefits from APEA.

The second paper introduces Primitive Granularity Control (PGC) which allows dynamically changing the primitive set during evolution, which can increase solution quality, decrease runtime, or both. PGC, however, does require primitives that can be used to build more complex primitives, making it unsuitable for many simpler problems. The process of building up these primitives is also non-trivial and requires a significant amount of domain-specific expertise. An exhaustive search was performed over a small subset of possible dynamic plans for heuristics for the Traveling Thief Problem. Dynamic plans were found that, compared to static plans, either found solutions of similar quality in less time, or found better solutions in the same amount of time. Areas of expansion would include a way to automatically generate the higher-level primitives, a method for efficiently determining effective plans, and closer examination of the population under PGC.

A combination of these methods would be possible, but presents new technical challenges to overcome. The first of these is that PGC relies on generations as a measure to dynamically change the primitive set. This could be addressed by using different metrics, such as time or number of evaluations. The next problem is specific to CGP and is much more fundamental. PGC relies on being able to build smaller primitives into larger more complex primitives, which clashes with CGP's representation. All individuals in CGP are the same size, making transforming between basic and macro primitives difficult. While this could be easily addressed by allowing individuals to change size, further alterations would almost certainly be required to combat new complications. One such issue is that CGP has no means of combatting bloat aside from the set individual size, so allowing individuals to grow is likely to result in extremely large individuals. Other, more subtle, problems likely exist in combining APCGP and PGC, but successfully utilizing both methods is expected to produce even greater returns than either one individually.

VITA

Adam Tyler Harter was raised in Plainfield, Illinois, graduating from Plainfield North High School in May 2012. From Fall 2012 to Spring 2017, Adam attended Missouri University of Science and Technology, earning a Bachelor of Science in Computer Science and Computer Engineering in May 2017. In the Summer of 2016, Adam worked at Sandia National Laboratories at the Albuquerque, New Mexico location, and then in the Summer of 2018 at Los Alamos National Laboratory. Adam earned a Master of Science degree in Computer Science from Missouri University of Science and Technology in July 2019.