Empirical Evidence of the Effectiveness of Primitive Granularity Control for Hyper-Heuristics

Adam Harter
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, USA
athb79@mst.edu

Aaron Scott Pope
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, USA
Los Alamos National Laboratory
Los Alamos, New Mexico, USA
aaron.pope@mst.edu

Daniel R. Tauritz
Natural Computation Laboratory
Department of Computer Science
Missouri University of Science and Technology
Rolla, Missouri, USA
dtauritz@acm.org

Chris Rawlings
Los Alamos National Laboratory
Los Alamos, New Mexico, USA
crawlings@lanl.gov

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.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; • **Theory of computation** → *Evolutionary algorithms*; • **Mathematics of computing** → *Combinatorial optimization*;

GECCO '19 Companion, July 13–17, 2019, Prague, Czech Republic

ACM ISBN 978-1-4503-6748-6/19/07…$15.00
https://doi.org/10.1145/3319619.3326860

ACM Reference Format:
Adam Harter, Aaron Scott Pope, Daniel R. Tauritz, and Chris Rawlings. 2019. Empirical Evidence of the Effectiveness of Primitive Granularity Control for Hyper-Heuristics. In *Genetic and Evolutionary Computation Conference Companion (GECCO '19 Companion), July 13–17, 2019, Prague, Czech Republic.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3319619.3326860

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 [6]. 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 [14]. 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 [17]. 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 [20, 24]. 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 [5] (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 [2, 11, 16, 26] and the knapsack problem [7, 8, 18, 22] in the past. Additionally, most methods for solving TTP are partially or entirely based on evolutionary methods [4, 21, 27]. 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 [10]. Martin and Tauritz [20] and Pope et al. [24] 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 [12] 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 [1], Hierarchy Locally Defined Modules [3], and Adaptive Representation [25]. 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 [15], 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 Figure 1 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.

4 TRAVELING THIEF PROBLEM

The Traveling Thief Problem [5] (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

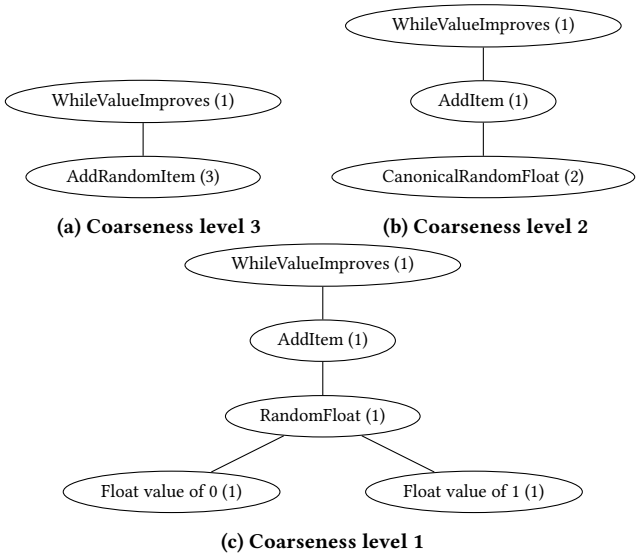


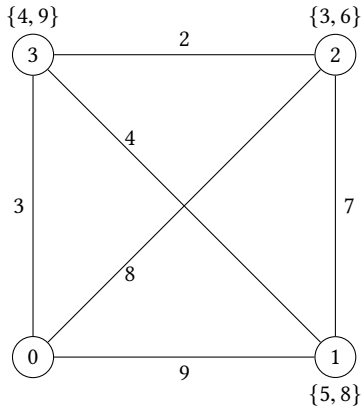
Figure 1: Example TTP heuristic at three primitive coarseness levels. The coarseness of the primitives is indicated by the number in parentheses after the primitive name.

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.

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 [13], 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 [23]. It has also enjoyed a great amount of attention from the field of evolutionary computation in general [4, 9, 21, 27]. These approaches generally start by finding a good starting TSP solution, usually using the Lin-Kernighan heuristic [19], 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



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

Table 1: Terminal Primitive Effective Coarseness

		Coarseness				
		1	2	3	4	5
Generated Type	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*.

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.

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 2: Non-terminal Primitive Effective Coarseness

		Coarseness				
		1	2	3	4	5
Generated Type	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

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, 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 [23] were computationally infeasible due to the exhaustive nature of the search. Therefore, three new, smaller,

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
```

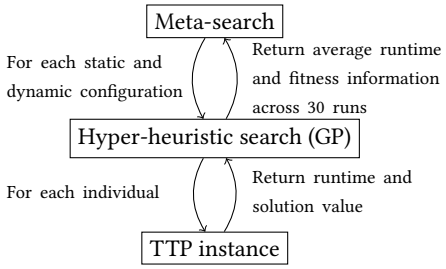


Figure 3: High Level Overview of Experiment

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. 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.

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

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

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 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 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

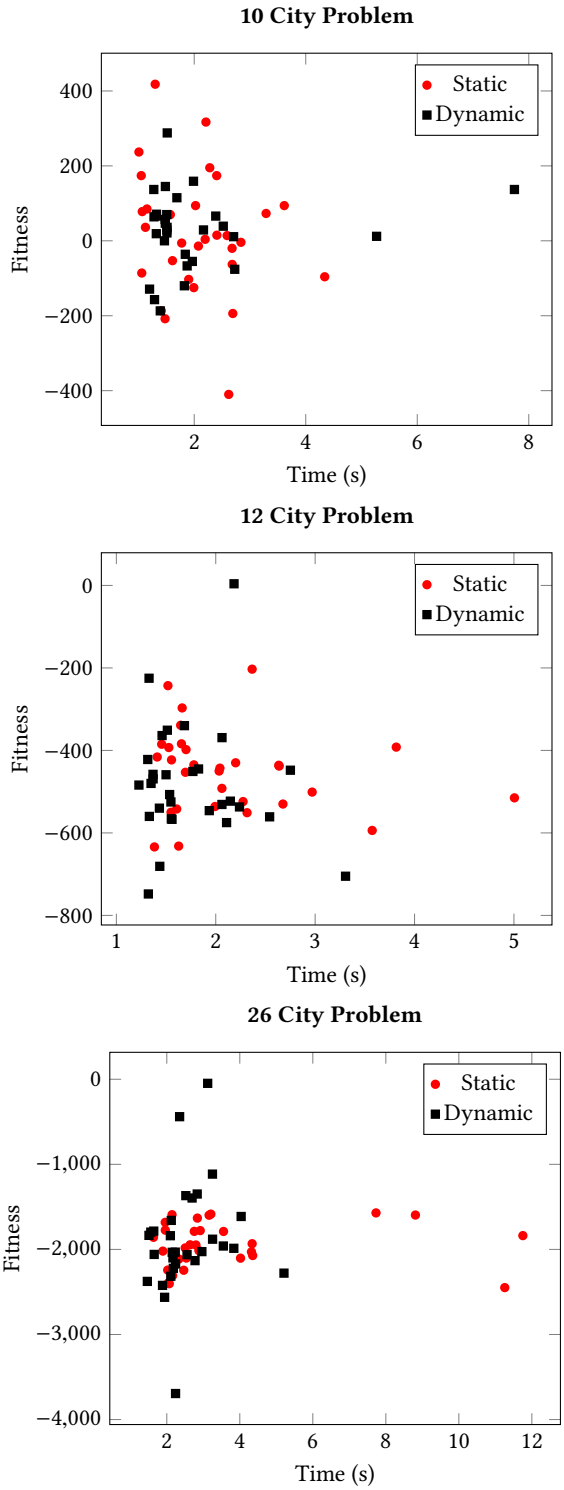


Figure 4: Max Fitness Versus Runtime for Each Run of the Best Dynamic and Static Plans for Maximum Fitness

Table 6: Comparison of Static and Dynamic Configurations

		Mean Value Across 30 Evaluations							
		Mean Fitness		Max Fitness		Runtime		Coarseness Level	
		Static	Dynamic	Static	Dynamic	Static	Dynamic	Static	Dynamic
	10 City Problem								
Best Configuration For	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**.

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

[1] Peter J Angeline and Jordan Pollack. 1993. Evolutionary Module Acquisition. In *Proceedings of the second annual conference on evolutionary programming*. Citeseer, 154–163.
[2] Zalilah Abd Aziz. 2015. Ant Colony Hyper-heuristics for Travelling Salesman Problem. *Procedia Computer Science* 76 (2015), 534–538.
[3] Wolfgang Banzhaf, Dirk Banscherus, and Peter Dittrich. 1999. *Hierarchical Genetic Programming Using Local Modules*. Secretary of the SFB 531.
[4] Julian Blank, Kalyanmoy Deb, and Sanaz Mostaghim. 2017. Solving the Bi-objective Traveling Thief Problem with Multi-objective Evolutionary Algorithms. In *International Conference on Evolutionary Multi-Criterion Optimization*. Springer, 46–60.
[5] Mohammad Reza Bonyadi, Zbigniew Michalewicz, and Luigi Barone. 2013. 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, 1037–1044.
[6] Edmund K. Burke, Michel Gendreau, Matthew Hyde, Graham Kendall, Gabriela Ochoa, Ender Özcan, and Rong Qu. 2013. Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society* 64, 12 (2013), 1695–1724.
[7] Edmund K Burke, Matthew R Hyde, Graham Kendall, and John Woodward. 2012. Automating the Packing Heuristic Design Process with Genetic Programming. *Evolutionary computation* 20, 1 (2012), 63–89.
[8] John H Drake, Matthew Hyde, Khaled Ibrahim, and Ender Ozcan. 2014. A Genetic Programming Hyper-heuristic for the Multidimensional Knapsack Problem. *Kybernetes* 43, 9/10 (2014), 1500–1511.
[9] Mohamed El Yafrani and Belaïd Ahiod. 2016. Population-based vs. Single-solution Heuristics for the Travelling Thief Problem. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. ACM, 317–324.
[10] Mohamed El Yafrani, Marcella Martins, Markus Wagner, Belaïd Ahiod, Myriam Delgado, and Ricardo Lüders. 2018. A Hyperheuristic Approach Based on Low-level Heuristics for the Travelling Thief Problem. *Genetic Programming and Evolvable Machines* 19, 1-2 (2018), 121–150.
[11] David B Fogel. 1993. Applying Evolutionary Programming to Selected Traveling Salesman Problems. *Cybernetics and systems* 24, 1 (1993), 27–36.
[12] Brian W Goldman and Daniel R Tauritz. 2011. Meta-evolved Empirical Evidence of the Effectiveness of Dynamic Parameters. In *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*. ACM, 155–156.
[13] Thomas Helmuth and Lee Spector. 2015. *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.
[14] Patricia D. Hough and Pamela J. Williams. 2006. Modern Machine Learning for Automatic Optimization Algorithm Selection. In *Proceedings of the INFORMS Artificial Intelligence and Data Mining Workshop*. 1–6.
[15] Stephen Kelly and Malcolm I Heywood. 2017. Emergent Tangled Graph Representations for Atari Game Playing Agents. In *European Conference on Genetic Programming*. Springer, 64–79.
[16] Graham Kendall and Jiawei Li. 2013. Competitive Travelling Salesmen Problem: A Hyper-heuristic Approach. *Journal of the Operational Research Society* 64, 2 (2013), 208–216.

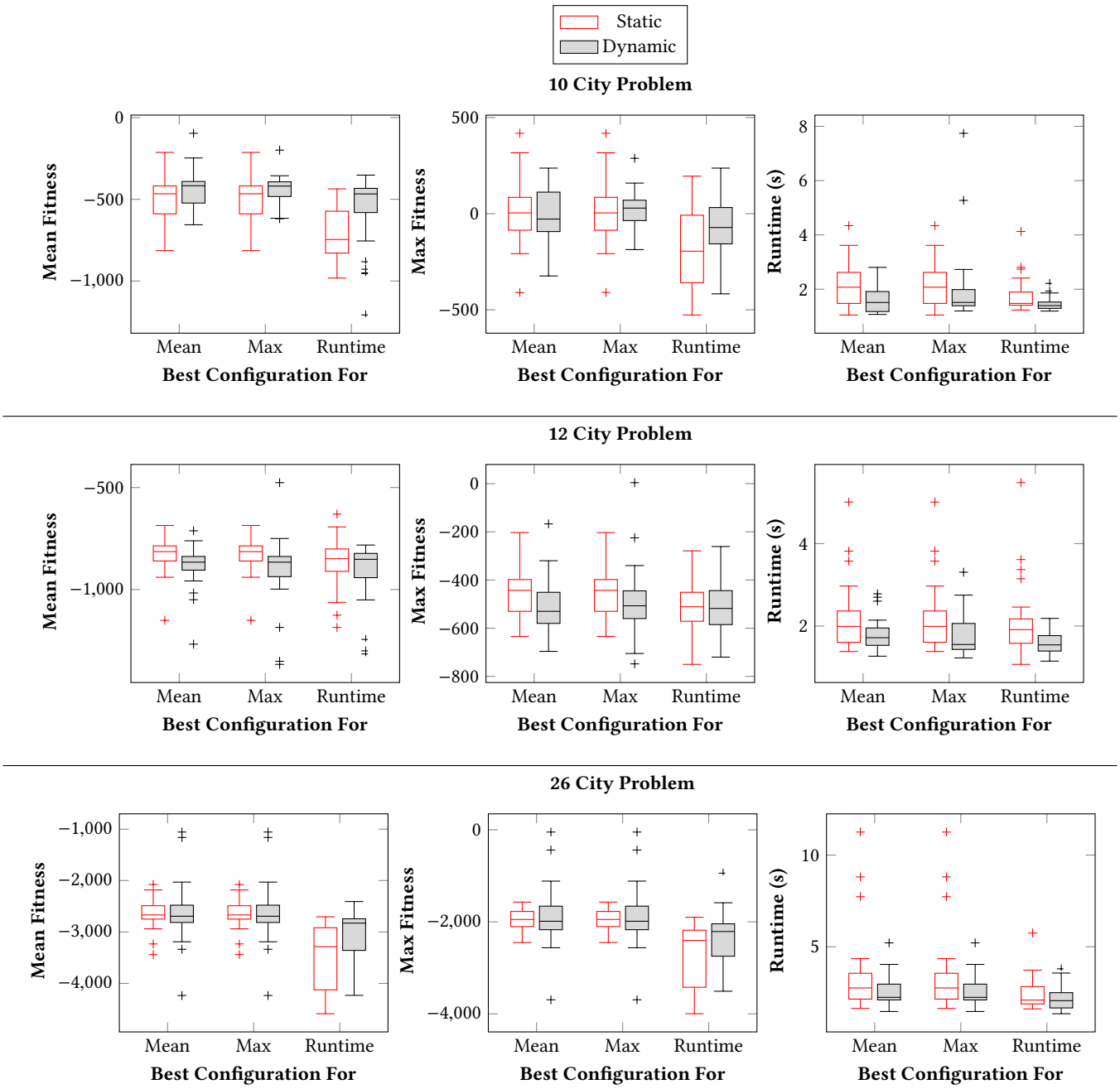


Figure 5: Comparison of Each Best Configuration

[17] John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

[18] Rajeev Kumar, Ashwin H Joshi, Krishna K Banka, and Peter I Rockett. 2008. 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, 1227–1234.

[19] Shen Lin and Brian W Kernighan. 1973. An Effective Heuristic Algorithm for the Traveling-salesman Problem. *Operations research* 21, 2 (1973), 498–516.

[20] Matthew A Martin and Daniel R Tauritz. 2015. 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, 1051–1058.

[21] Yi Mei, Xiaodong Li, Flora Salim, and Xin Yao. 2015. Heuristic Evolution with Genetic Programming for Traveling Thief Problem. In *Evolutionary Computation (CEC), 2015 IEEE Congress on.* IEEE, 2753–2760.

[22] Lucas Parada, Carlos Herrera, Mauricio Sepúlveda, and Víctor Parada. 2016. Evolution of New Algorithms for the Binary Knapsack Problem. *Natural Computing* 15, 1 (2016), 181–193.

[23] Sergey Polyakovskiy, Mohammad Reza Bonyadi, Markus Wagner, Zbigniew Michalewicz, and Frank Neumann. 2014. A Comprehensive Benchmark Set and Heuristics for the Traveling Thief Problem. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation.* ACM, 477–484.

[24] Aaron S. Pope, Daniel R. Tauritz, and Alexander D. Kent. 2016. Evolving Random Graph Generators: A Case for Increased Algorithmic Primitive Granularity. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. IEEE, 1–8. https://doi.org/10.1109/SSCI.2016.7849929

[25] Justinian P Rosca and Dana H Ballard. 1994. Hierarchical Self-organization in Genetic Programming. In *Machine Learning Proceedings 1994*. Elsevier, 251–258.

[26] Patricia Ryser-Welch, Julian F Miller, and Shahriar Asta. 2015. 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, 1067–1074.

[27] Junhua Wu, Sergey Polyakovskiy, Markus Wagner, and Frank Neumann. 2018. Evolutionary Computation plus Dynamic Programming for the Bi-Objective Travelling Thief Problem. *arXiv preprint arXiv:1802.02434* (2018).