Improving the Tartarus Problem as a Benchmark in Genetic Programming

Thomas D. Griffiths and Anikó Ekárt

Aston Lab for Intelligent Collectives Engineering (ALICE)
Aston University, Aston Triangle, Birmingham B4 7ET, UK
grifftd1@aston.ac.uk, a.ekart@aston.ac.uk

Abstract. For empirical research on computer algorithms, it is essential to have a set of benchmark problems on which the relative performance of different methods and their applicability can be assessed. In the majority of computational research fields there are established sets of benchmark problems; however, the field of genetic programming lacks a similarly rigorously defined set of benchmarks. There is a strong interest within the genetic programming community to develop a suite of benchmarks. Following recent surveys [7], the desirable characteristics of a benchmark problem are now better defined. In this paper the Tartarus problem is proposed as a tunably difficult benchmark problem for use in Genetic Programming. The justification for this proposal is presented, together with guidance on its usage as a benchmark.

Keywords: Genetic Programming, Benchmark, Tartarus

1 Introduction

The Genetic Programming (GP) research community has recently recognised that there is a lack of serious and structured benchmark test problems [3]. The need to define and establish a suite of benchmark problems is becoming greater as the popularity of GP increases and the field matures. The most prominent step so far has been the use of community surveys to establish which problems and tests are most popular and widely used [3,7]. There are certain attributes and characteristics that all benchmark problems are expected to exhibit. In this paper we analyse how these apply to Genetic Programming and argue that the *Tartarus problem* satisfies all the relevant characteristics, and therefore can be used as a reliable benchmark problem.

A benchmark is defined as a point of reference, against which given artifacts can be compared and contrasted. In computer science, entire computer programs or specific software components are often tested and compared using so-called benchmark problems. Many fields in computer science have established suites of benchmark problems on which proposed methods and approaches are tested.

As a research field matures, it is common for a suite of benchmark problems to emerge. This is important for the field as it provides a standardised way to compare and measure the relative performance of different solution methods and approaches. When a new algorithm is developed and tested, its performance must be compared to other state-of-the-art algorithms within the field. The only practical way to do this is using a suite of benchmark problems.

For example, in evolutionary algorithms, there is a benchmark repository [4] to which newly created benchmarks are continuously added, via benchmarking competitions and special sessions at conferences. However, for GP specific problems this has not been the case, there is a lack of agreement within the GP community about which problems should be used. Often the problems on which new methods are demonstrated are outdated, with little applicability to real world situations, or they are so-called toy problems of trivial difficulty.

In this paper we propose that the Tartarus problem become adopted as a GP benchmark and we justify how it satisfies all required characteristics for a benchmark. The paper is organised as follows: Section 2 details the desirable characteristics of a GP benchmark problem, Section 3 outlines previous attempts at GP benchmarks and the issues with the usage of toy problems in GP. Section 4 justifies the usage of the Tartarus problem as a benchmark for GP and Section 5 draws conclusions.

2 Desirable GP Benchmark Characteristics

For many years there has been no agreement on what makes an effective benchmark in GP. More recently, surveys of the GP community by White et al. [7] have outlined some of the important characteristics that must be present across a suite of benchmark problems. The intention was not to create an exhaustive list of every desirable feature of any benchmark suite, but a list of the key features that must be present in order for the set of benchmark problems to be useful and effective for GP. These desirable characteristics are discussed below.

Tunable Difficulty One of the most important characteristics of an effective benchmark problem is tunable difficulty. A problem is said to be tunably difficult if there are methods by which the difficulty of instances can be changed and altered relative to each other. This provides scope for the benchmark to be used across a wide range of GP methods, while maintaining comparability between the results. The creation of instances of increasing difficulty is essential in order to push the boundaries of current research [3].

Precisely Defined A benchmark should be well-defined and documented, outlining the problem constraints and boundaries. It is common for a benchmark to be accompanied by a set of recommended resource constraints, such as an upper limit on the number of available evaluations or placing a time limitation on the program execution.

Accommodating to Implementors In order for a benchmark problem to be accepted by the research community, it must be accommodating to the practitioners who implement it, and straight forward to use. The benchmark problem must be self-contained and all its elements must be open-source and accessible, to ensure universal access without the need for specific domain knowledge.

Representation Independent An effective benchmark should attempt, as far as it is reasonably practical, to be representation independent in terms of the programming language used and the programming style. As the field of GP expands and matures, we expect that the number of programming languages and representations being used will increase. Benchmarks should be flexible enough to allow adaptation between various languages and representations, while still being effective. Attempts should be made to ensure that the benchmark does not rely on any specific attributes from a language.

Easy to Interpret and Compare It is also important that the results generated by the benchmark are easy to interpret and can be compared without ambiguity. This clarity in understanding the results is vital, it allows for trends and relationships to be established between different sets of results, and reliable conclusions to be drawn on the data.

Relevant A benchmark problem should contain elements which are directly relevant to the wider field in which it operates. Ideally for GP this would be the wider machine learning and optimisation communities. A well structured benchmark should not be vulnerable to paradigm shifts in the underlying application domain, rendering it irrelevant.

Fast Due to the fact that GP individuals are programs which must be executed, often many times per individual, fitness evaluations can be slow. Therefore a GP benchmark should be fast enough to allow the large number of runs required to create meaningful comparisons between approaches, to be carried out in a reasonable time frame.

In order for a problem to be considered an effective benchmark candidate it must satisfy the majority of the aforementioned characteristics, combining them together to make an effective benchmark. However, many of the problems currently being used as de-facto benchmarks in GP only satisfy a small number of these characteristics. It is important to define a suite of benchmark problems, which collectively satisfy the entire range of desirable characteristics. The main benefit of having a suite of benchmark problems instead of using just a single problem is that it allows for different types of problems from various areas to be tested and the approaches compared. The range of problem domains also allows for the portability and scalability of a solution approach to be tested across the field of research.

3 GP Benchmarks

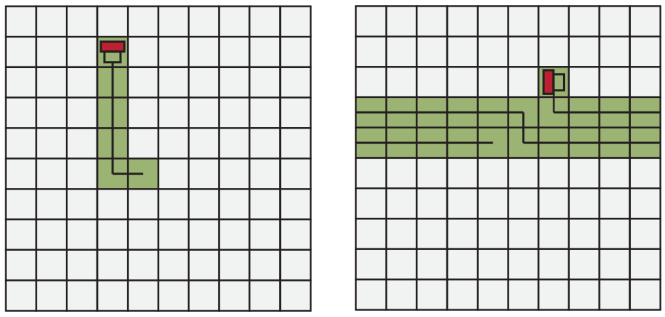
In order to demonstrate how GP worked, in 1992 Koza defined a set of test problems, including the *k-even parity*, *symbolic regression*, *artificial ant* and *lawnmower* problems [2]. Over the following 20 years these problems became widely used within the GP community, emerging as the de-facto problems for GP experimental research. More recently, however, the GP community has started to expand and has attempted to use a more varied set of test problems. As a consequence, there is a growing desire within the GP community to establish a suite of well-defined benchmark problems for universal use [7].

One of the main issues with Koza's problems, as outlined by Vanneschi [6], is the fact that many of them lack rigorous evaluation methods, and more importantly, the majority of the problems have their own level of *fixed structural complexity* built into the problem. It is often hard to create several instances of these problems with a predictably varied level of complexity and difficulty. The inability to create a range of difficulties across instances by altering the parameters of the problem is a major drawback for many of Koza's problems. Many of these problems are referred to as 'toy problems' as they are usually simple in nature and are trivially easy to solve using modern GP methods (as well as manually, by a human).

It has been said that 'GP has a toy problem problem' and as a field of research it is often too reliant on these simple, trivially easy problems. It has been argued that the use of toy problems as de-facto benchmarks does little to advance the field, and may in fact be responsible for holding it back in some regards [3]. As mentioned previously, one of the main drawbacks of using toy problems as de-facto benchmarks is that many of the instances may be disproportionate to each other in terms of difficulty. This can lead to poor coverage of the problem space in which they operate, limiting their effectiveness as a benchmark [1]. Therefore, the results obtained from toy problems may be misleading and create the illusion of success, when the relative performance of an approach may actually be measurably worse. In order to have practical significance it is often important for a benchmark to be able to simulate some aspects of real world problems. However, this is one area in which toy problems are rather weak: they are often unreliable at predicting the success of a methodology or solution in real world situations [8].

3.1 The Lawnmower Problem

One of Koza's test problems which has been widely used is the *Lawnmower problem* [10]. In this simple problem a $n \times m$ toroidal grid representing a lawn of grass and a controllable agent representing a lawnmower are given. The essence of the problem is to find a program to control the movement of the lawnmower so that it traverses the entire lawn. The lawnmower has a state consisting of its current location in the $n \times m$ grid and its current orientation (North, East, South or West). Figure 1 shows examples of the Lawnmower problem of size 10×10.



(a) Example State of an Instance (b) Systematic Approach

Fig. 1: The Lawnmower Problem

The shaded area shows the traversed lawn, while the line within indicates the actual trajectory taken. The lawnmower has no sensors with which to collect information from its surroundings. This lack of sensory information has little impact on the agent, as there are no obstacles in the environment and the toroidal nature of the grid effectively removes any boundaries which the agent might otherwise have encountered. There are four operations which are used to change the state of the lawnmower: *turn left*, *turn right*, *move forward one square* and *move forward k squares*.

The Lawnmower problem has been used to test and prove the effectiveness of different GP methods and representations. Current GP methods are very effective and can solve the problem easily. For this problem the only way in which the difficulty can be changed is by altering the size of the instance. Given that there are no constraints to satisfy, an increase in the size of the problem results in a proportional increase in the time taken to solve the problem. Often solutions to the Lawnmower problem can simply be scaled up along with the instance size with no increase in complexity.

The solutions that are most successful at solving the Lawnmower problem tend to be systematic in their approach, an example of which is shown in Figure 1(b). They repeatedly execute a relatively small set of instructions until the problem's termination criteria are met, an example GP solution that leads to the simple systematic approach used in Figure 1(b) is shown in Figure 2. There are two reasons why systematic controllers are successful at solving the Lawnmower problem. Firstly due to the fact that the grid is toroidal in nature, a solution can move freely in any direction without being impeded. This allows for a randomly generated solution, which would otherwise have become stuck against a wall or corner, to exploit the toroidal nature of the grid and generate adequate solutions. Secondly, there is little external information available to the lawnmower, and no information relating to the state of its surrounding environment, and in fact no need for such information.

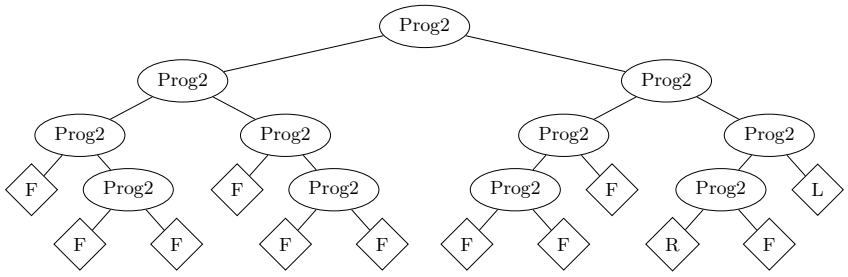


Fig. 2: Simple systematic solution to the Lawnmower problem

The Lawnmower problem can be seen as part of a wider family of grid-based problems, including the *Maze problem* and the *Tartarus problem*. There are many parallels that can be drawn across these, they all operate in a grid environment and involve an agent following some path, but according to different objectives:

- *Lawnmower Problem* The objective of the agent is to cover as much of the lawn as possible.
- *Maze Problem* The objective of the agent is to find a trajectory from one position to another, given obstacles (walls).
- *Artificial Ant Problem* The objective of the agent is to locate all food pellets on a trajectory, possibly with gaps.
- *Tartarus Problem* The objective of the agent is to explore the environment in order to find the blocks. The agent must then move the found blocks to prescribed areas of the grid.

The Lawnmower problem in its default setting is clearly the simplest of the three problems. It is possible however to alter the problem definition slightly and add obstacles, increasing the complexity of the problem instance. Another method of increasing the complexity of the Lawnmower problem is to use a non-toroidal grid. Removing the toroidal nature of the grid will force the agent to take the edges of the grid into consideration when creating solutions. This new constrained Lawnmower problem becomes a special case of the Maze problem where there are multiple potential start and finish locations. The Maze problem can be seen as a generalisation of a constrained Lawnmower problem. On the other hand, the Tartarus problem involves locating blocks and moving them to the sides of the grid, which is similar to finding routes in the maze, while picking up blocks on the way. The added complexity comes from the fact that the locations of the blocks are not known to the agent. The problems do differ however in terms of their applicability to real world situations.

In unconstrained problems such as the lawnmower, with limited available external information, it is hard to create a strategic controller that is able to strategically plan ahead the actions of the agent. Therefore, systematic controllers become dominant in terms of performance. However as the problems

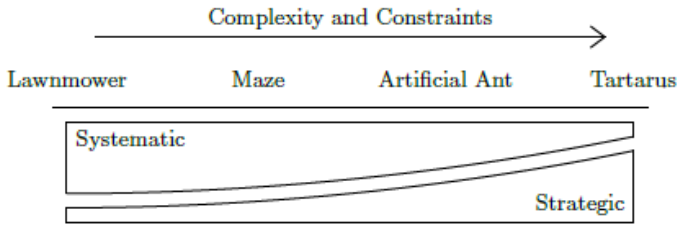


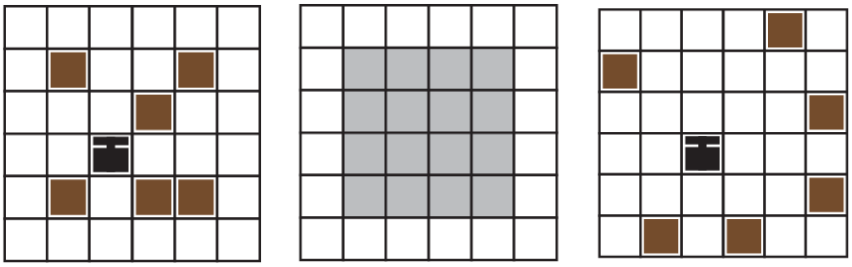
Fig. 3: Systematic and strategic controllers for grid-based problems

become more constrained (i.e. via the addition of obstacles) strategic controllers become more appropriate, utilising the higher level of external information available to them to create successful and effective solutions, as shown in Figure 3.

4 The Tartarus Problem

The Tartarus problem, originally introduced by Teller [5] is a grid-based optimisation problem. In the Tartarus problem a $n \times n$ non-toroidal grid representing an enclosed environment, a number of movable blocks and a controllable agent representing a bulldozer are given. The aim of the problem is to locate and move the blocks to the edge of the environment. Unlike in the Lawnmower problem, the agent in the Tartarus problem has no initial knowledge of its location in the grid or its orientation. The agent does however, have eight sensors, which are able to sense the environment in the surrounding eight gridsquares. There are three operations which can be used to change the state of the dozer: *turn left*, *turn right* and *move forwards one square*.

The canonical Tartarus instance, an example initial state of which is shown in Figure 4(a), consists of a 6×6 grid, six blocks and one dozer. The blocks and the dozer must be placed in a valid start position in the central 4×4 gridsquares, shown by the shaded area in Figure 4(b). During execution of a 6×6 instance, the agent is allowed a total of 80 moves to attempt to create a viable solution,



(a) Example Initial State (b) Valid Start Positions (c) Example Final State

Fig. 4: Example Tartarus states

with each of the three operations counting as an individual move. The final state of a successfuly completed Tartarus instance, where all of the blocks have been pushed to the edges of the grid, is shown in Figure 4(c).

4.1 Satisfying the Desirable Benchmark Characteristics

With the modified state evaluation proposed in Section 4.3, the Tartarus problem satisfies all of the characteristics outlined in Section 2:

Tunable Difficulty The difficulty of the Tartarus problem can be altered by changing the parameters of the problem (grid size, number of blocks) and the restrictions placed upon the agent (number of allowed moves). This leads to a predictable shift in the complexity and difficulty of the generated instance. A detailed explanation of this process and guidelines on the tuning of the difficulty in the Tartarus problem is outlined in Section 4.6.

Precisely Defined The Tartarus problem is a constrained problem with well defined boundaries for both the problem instance and the agent. The original problem definition outlined a maximum number of allowed moves, in this paper we suggest minimum and maximum constraints for allowed number of moves, number of blocks and the size of the instance, providing a comprehensive list of the operating constraints for the Tartarus problem.

Accommodating to Implementors Due to the fact that the Tartarus problem is a grid based problem it is simple to implement and use. There are no specialist skills or software tools that are required to create and use an implementation of the Tartarus problem.

Representation Independent As the implentation of the Tartarus problem is simple, it does not require any specific languages or software tools. The problem is representation independent and it should be possible to implement in any modern programming language and paradigm.

Easy to Interpret and Compare In order to increase the ease with which Tartarus problem instances and solutions could be interpreted and compared we suggest an improved method of state evaluation in Section 4.3. The improved method of state evaluation allows for a more fine-grained evaluation of instances, allowing for a more accurate comparison to be made.

Relevant The Tartarus problem has scope for real applications outside of GP. For example, the recently announced Emergency Robots competition[1] (building upon the success of EuRathlon[2]), inspired by the 2011 Fukushima accident focuses on realistic emergency response scenarios. In these scenarios missing workers have to be found, critical hazards have to be identified in limited time.

[1] http://eu-robotics.net/robotics_league
[2] http://www.eurathlon.eu An outdoor robotics challenge for land, sea and air

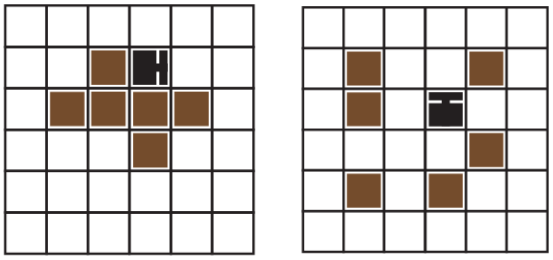
Fast Given the improved method of state evaluation for the Tartarus problem outlined in this paper, together with the simple nature of the implementation, tests can be carried out and meaningful comparisons can be made in a reasonable time frame. The fitness evaluation is fast, allowing for multiple executions per individual to be carried out.

4.2 Current State Evaluation

For the Tartarus problem a solution consists of a series of instructions, which control the actions of the agent in the environment. In order to test the efficacy of a solution, there needs to be a way to measure its outcome on an instance environment. This is usually done by evaluating the end position (state) after executing the complete series of instructions. In fact, the same evaluation method can be used to evaluate any state, for different purposes:

- evaluate the initial state, in order to evaluate the problem instance
- evaluate an intermediate state, after a set period of time or number of moves, in order to measure progress
- evaluate the end state in order to evaluate solution quality.

The current method of state evaluation only rewards the number of blocks, which are located at the edges of the grid. This rather binary success or fail approach works well for many benchmark problems where the absolute score achieved by a candidate solution is the only desired success measure. However, for GP, rewarding part-way solutions is essential during evolution, so that better solutions can evolve. For example, the cluster of Figure 5(a) is very different from the dispersed blocks of Figure 5(b). Both these states would have the same evaluation score of zero. The dispersed blocks example in Figure 5(b) is visibly closer to an optimal end state when compared to the cluster example in Figure 5(a) as the blocks are closer to the edges of the grid. Specifically, from the state in Figure 5(a) 32 moves would be required to move all blocks to an edge position, while to do the same from the state in Figure 5(b) 27 moves would be needed. Therefore a solution that arrived at the latter position had made more progress than one that arrived at the earlier position.



(a) Cluster (b) Dispersed

Fig. 5: Example Tartarus states

The current binary assignment of success or fail is too coarse. It misses differences in performance between candidate solutions in the population. Solutions that move no blocks at all would be evaluated the same as solutions that make some progress, but fall short of actually moving a block to the edge of the grid (see Figure 5(b)). Solutions that have not actually moved blocks to the edges of the grid could still have made some progress by moving blocks *closer* to the edges and this should be recognised and rewarded in some manner. According to the same rationale, initial states with the same number of blocks are not equivalent, as some require less moves than others to solve. Therefore the difficulty of the problem instance is not only dependent on the grid size, number of blocks or allowed number of moves, but also the initial distribution of blocks on the grid.

4.3 Proposed Improved State Evaluation

We propose a new evaluation method that rewards states according to how close they are to the desired final states, by including how close to the edge each block in the given state is:

$$State\_value = \mathrm{C}_1 \cdot \left(B - \frac{2}{n} \sum_{i=1}^{B} d_i - \mathrm{C}_2 \right) \tag{1}$$

where B is the total number of blocks, n is the size of the grid and d_i is the distance of block i from an edge in the given problem instance. C_1 and C_2 are scaling and translation constants based on B and n in order to make the value range consistent with the current evaluation method. For a 6×6 instance with 6 blocks $\mathrm{C}_1 = 1.8$ and $\mathrm{C}_2 = \frac{8}{3}$.

We generated 1000 random states for grid size 6×6 containing 6 blocks and evaluated them with the two methods (there was no need to generate candidate solutions, as the candidate solutions would be evaluated on the basis of the end position that they lead to). In Table 1 we present 30 examples of the state evaluation. As shown, the new method of state evaluation allows for a more fine-grained evaluation.

Table 1: Fitness data comparing evaluation methods

Individual	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Old Value	0.0	0.0	0.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0
New Value	1.2	1.2	1.8	0.6	1.8	1.8	1.8	1.8	2.4	2.4	2.4	2.4	2.4	3.0	3.0

Individual	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
Old Value	2.0	2.0	2.0	2.0	2.0	2.0	3.0	3.0	3.0	3.0	3.0	3.0	4.0	4.0	4.0
New Value	3.0	3.0	3.0	3.6	3.6	3.6	3.0	3.6	3.6	4.2	4.2	4.2	3.6	4.2	4.8

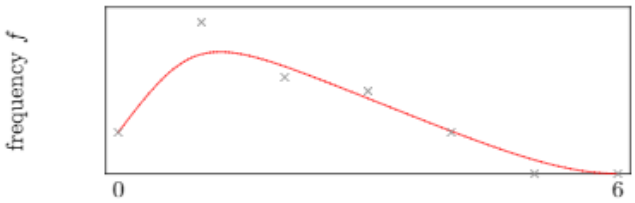


Fig. 6: Current evaluation distribution

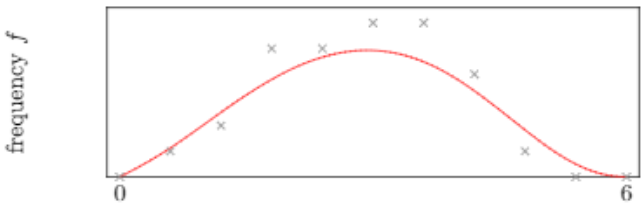


Fig. 7: Proposed evaluation distribution

The distributions of the results are shown in Figures 6 and 7. The current evaluation method rates all states with no blocks at the edges as equally poor, when in reality some can lead to end states that are just a few moves away from optimal end states. The proposed evaluation method rewards these. The distribution in Figure 7 better reflects the actual performance, at a much more fine grained level.

4.4 Baseline Values for Tartarus Instances

A baseline is a set of instances that are generated corresponding to a set of values for the parameters defining difficulty. This is structured to allow clean, controlled comparison of difficulty between generated instances. In the Tartarus problem the difficulty increases with size, therefore when comparing difficulty of problem instances, by fixing the number of moves and blocks, the only variable altered in the tuning of the baseline difficulty is the grid size n. When establishing the functions for the number of moves $m(n)$ and the number of blocks $B(n)$, we endeavour to maintain the difficulty relative to grid size.

An actual solution will need to include moving to a block, then pushing the block to an edge and repeating these steps for all blocks, taking into account obstacles (non-movable blocks). Let us consider the step of pushing a block to an edge. The expected distance of a block randomly placed on the inner grid of size $n-2$ to an edge is:

$$\text{Dist} = \frac{1}{(n-2)^2} \sum_{i=1}^{\frac{n-2}{2}} 4(n-1-2i)i = \frac{n(n-1)}{6(n-2)} \quad , \tag{2}$$

where $4(n-1-2i)$ is the number of positions on the inner grid of size $n-2$ that are at distance i from an edge of the grid of size n. For grid size 6, the agent can expect to have to move 1.25 for each block, and the number of moves increasing with grid size will be 1.55 for grid size 8, and 4.35 for grid size 25. At the same time, if the agent *does not know the direction in which to move* to get to the closest edge, the expected distance to have to push a block randomly placed on the grid at location (x, y) to any edge becomes:

$$\text{Dist} = \frac{x + (n-1-x) + y + (n-1-y)}{4} = \frac{n-1}{2} \quad . \tag{3}$$

Comparatively, for grid size 6, the number of moves becomes 2.5, for grid size 8, 3.5 and for grid size 25, 12. The fraction of the grid that the agent can be expected to travel to reach the edge will be:

$$\frac{\text{Dist}}{n} = \frac{n-1}{2n} \quad . \tag{4}$$

For grid size 6 the agent without global vision can expect to have to travel a proportion of 0.416 of the grid size n in order to move one block to the edge (not accounting for obstacles and travelling from the edge after successful move of one block to the next block). As n increases, this proportion approaches 0.5. A size 6×6 instance is thus solvable with less effort than a 7×7 instance (0.429) and substantially less effort than an instance of size 25×25 (0.48). This does not account for the occurrence of impossible to move blocks in an instance. We can conclude that the effort expected to move one block to the edge is increasing with the increase in grid size. So far the tunability of Tartarus is similar to the tunability of the Lawnmower problem. However, when the number of blocks and their location is considered, more fine tuning becomes possible with the Tartarus problem.

The number of blocks in the environment which must be moved, B, contributes to the overall difficulty of the problem instance. A baseline was determined through regression, following generation and evaluation of a set of 1000 instances each of the sizes (6,7,8,16) and with varying numbers of blocks, but fixed numbers of moves (80,109,142,569). This baseline with C=$\frac{2}{9}$ is:

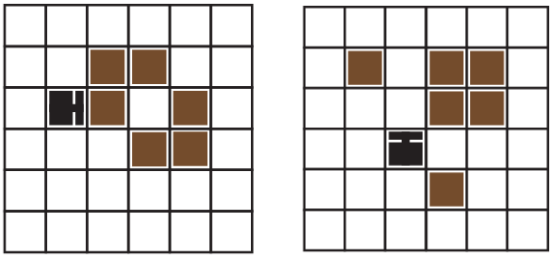
$$B_{baseline} = \left\lceil (n-1)^2 \cdot C \cdot \left(\frac{3}{2} - \frac{n-1}{2n} \right) \right\rceil \quad . \tag{5}$$

The number of allowed moves m can be used to tune the difficulty of the Tartarus problem. It makes sense to link this resource limitation to the grid size, as the number of steps required for simply traversing the grid, or moving one block to an edge, depends on the grid size. In order to establish the relationships between the problem parameters and produce reliable results, we determined that the following quadratic function would be suitable:

$$m_{baseline} = \left\lfloor 10\, C \cdot n^2 \right\rfloor \quad . \tag{6}$$

4.5 Generating Tartarus Instances

When generating Tartarus instances it is important to note that not all generated instances can be solved perfectly. As the placement of the blocks in the instance is random, it is likely that some instances may contain blocks, which are impossible to move, making the instance partly or completely impossible to solve. The two most common of these configurations are shown in Figures 8(a) and 8(b). In the Wilson configuration in Figure 8(a), although it is possible to move some of the blocks, doing so will create a cluster of four blocks which cannot be moved any further. An example of an instance containing a four-block cluster is outlined in Figure 8(b). Although the four block square cluster is the most common situation in a 6×6 instance, it is also possible for all six blocks to be placed together in a non-movable cluster.



(a) Wilson Configuration (b) 4-Block Cluster

Fig. 8: Partially Solvable Tartarus Instances

In previous work on the Tartarus problem [11,12,13], the authors identified the instances that contained unmovable block configurations, and removed them from the study. Their argument for removing the impossible to solve instances was that $\frac{2}{3}$ of all the blocks in the instances were impossible to move and studying these would not lead to solutions. However, with the increase in size of the problem, the impact of the four-block clusters becomes more negligible. For example, in the case of a 32×32 instance with over 200 blocks, the presence of 5 four-block clusters still allows for a substantial proportion of blocks to be moved. It is for this reason that we advise against discarding Tartarus instances which are partially solvable for instances above the canonical size of 6×6.

We next identified a reasonable working range of parameters for the Tartarus problem based on a set of 1000 generated instances. Beyond this range the results that are produced are of little to no value, due to the fact that the instances become too easy or too hard to reasonably solve. For example, a Tartarus instance with $m = 20$ allowed moves would be extremely difficult, if not impossible, to reliably find an effective solution. Therefore a set of *baseline* values were established. It is recommended that these be used when creating instances of the Tartarus problem. We located a usable working range of approximately $\pm$ 25% around the baseline values. It is proposed that Table 2 should be used

Table 2: Reference guide for generating Tartarus instances

n	Moves			Blocks		
	-25%	$m_{baseline}$	$+25\%$	-25%	$B_{baseline}$	$+25\%$
6	60	80	100	4	6	8
7	82	109	136	7	9	11
8	107	142	177	9	12	15
16	427	569	711	39	52	65
32	1707	2276	2845	163	217	271

as a reference when tuning the individual difficulty of a Tartarus instance, in order to maintain a relative level of practicality and usability for any instances created.

4.6 Tuning Difficulty

In addition to the established baseline for difficulty for each given grid size n, the difficulty can be adjusted in a more fine grained manner to achieve a more complete set of problem instances of different levels of increasing difficulty. This can be achieved by modifying the number of blocks B and the number of moves m away from the baseline values. It is best to systematically make changes and follow a Design of Experiments approach[9]. We propose a generic method to estimate the relative difficulty of a Tartarus instance, outlined in Equation 7:

$$D = \begin{cases} 0.5 \cdot \dfrac{m_{baseline}}{m} + 0.5 \cdot \dfrac{B_{baseline}}{B} + \dfrac{B_I}{B} & \textbf{if} \quad B_I < B \\ \textbf{impossible} & \textbf{if} \quad B_I = B \end{cases} \tag{7}$$

where $m_{baseline}$ is the baseline number of moves defined in Equation 6, m is the user set number of allowed moves, $B_{baseline}$ is the baseline number of blocks defined in Equation 5, B is the user set number of blocks, B_I is the number of impossible-to-move blocks.

The formula ensures a clear separation between instances that only contain impossible-to-move blocks and instances that have some movable blocks. In the case when there are movable blocks, it accounts equally for the relative changes in number of moves away from the baseline and number of blocks away from the baseline. It also factors in the difficulty added by having some impossible-to-move blocks. It has been designed to indicate difficulty of approximately 1 for baseline values. The value 1 cannot be ensured with exactness due to the small number of impossible-to-move blocks that could be included in randomly generated problem instances.

Table 3 provides a range of settings for difficulty levels from very easy to very hard for each grid size. These examples reflect how reducing the number of allowed moves makes an instance harder and also how this coupled with reducing the number of blocks could make an instance even harder (intuitively it takes longer to locate blocks which are spaced further apart).

Table 3: Example instances and their difficulty

		Very Easy	Easy	Baseline	Hard	Very Hard
6×6	m	96	88	80	66	55
	B	7	7	6	6	6
	D	0.85	0.88	1	1.11	1.23
8×8	m	170	156	142	116	104
	B	14	13	12	11	11
	D	0.85	0.92	1	1.11	1.22
16×16	m	704	662	569	486	422
	B	58	55	52	50	47
	D	0.85	0.90	1	1.11	1.23

In order to demonstrate equivalent difficulty levels across different grid sizes, we ran 100 experiments of standard linear GP for each grid size and corresponding difficulty level according to Table 3. For every difficulty level, we performed the experiments on randomly generated problem instances with the proposed number of moves and number of blocks. The instructions *left, right, forward*, crossover rate 0.5, mutation rate 0.05, population size 100 over 50 generations were used. A snapshot of the fitness values at the end of the runs is presented in Figure 9. The confidence intervals at 95% confidence level ranged between 0.03 and 0.06. This reinforces the hypothesis that as the difficulty and grid size increase, the quality of solutions and associated fitness obtained with equivalent resources decrease.

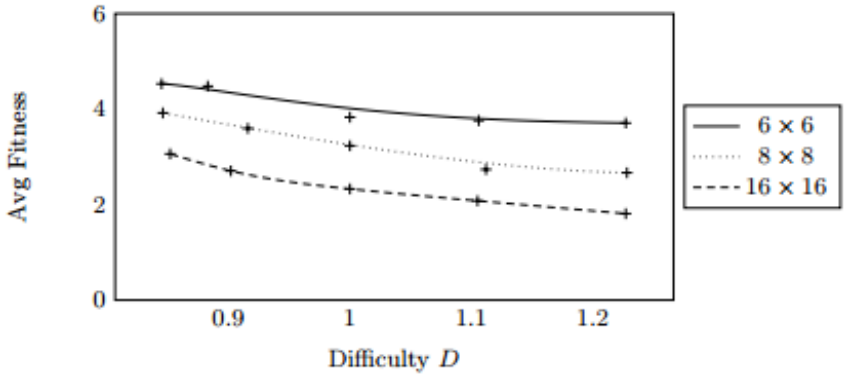


Fig. 9: Average fitness vs difficulty in linear GP experiments

5 Conclusion

In this paper we proposed using the Tartarus problem for benchmarking purposes in GP. We first proposed a more fine grained fitness evaluation measure. Then we demonstrated its tunable difficulty character, introduced a method to estimate the difficulty of instances and proposed suitable ranges of parameters to tune the difficulty. We are planning to publish methods to automatically generate suites of problem instances of desired varied difficulty levels.

A remaining aspect of an effective benchmark to be satisfied is to explore in more detail how the Tartarus problem can cross the gap between the theoretical and practical domains. This is one essential feature distinguishing between toy problems and effective benchmarks. We are confident that real life robotics scenarios (such as handling dangerous equipment) can be found where theoretical results can be directly applied. Therefore future work should focus upon ways in which results of the Tartarus problem can be extended to a real world context. Once this is achieved, the Tartarus problem will be set apart from other seemingly easy toy problems within GP.

References

1. Korkmaz, E.E. and Ucoluk, G.: Design and Usage of a New Benchmark Problem for Genetic Programming. In: Yazıcı A. et al., (ed.) Proceedings of the 18th International Symposium on Computer and Information Sciences, ISCIS XVIII. pp.561-567 (2003)
2. Koza, John R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection (1992)
3. McDermott, J., White, D.R., Luke, S., Manzoni, L., Castelli, M., Vanneschi, L., Jaskowski, W., Krawiec, K., Harper, R,. De Jong, K. and O'Reilly, U.M.: Genetic Programming Needs Better Benchmarks. In: Soule, T. et al., (ed.) Proceedings of the 14th International Conference on Genetic and Evolutionary Computation, GECCO '12. pp.791-798 (2012)
4. Sendhoff, B., Roberts, M. and Yao, X.: Evolutionary Computation Benchmarking Repository. IEEE Computational Intelligence Magazine 1, pp.50-60 (2006)
5. Teller, A.: The Evolution of Mental Models. In: Kinnear, Jr, K.E. (ed.) Advances in Genetic Programming. pp.199-217 (1994)
6. Vanneschi, L., Castelli, M. and Manzoni, L.: The K Landscapes: a Tunably Difficult Benchmark for Genetic Programming. In: Krasnogor, N. et al., (ed.) Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, GECCO '11. pp.1467-1474 (2011)
7. White, D.R., McDermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M. and Luke, S.: Better GP Benchmarks: Community Survey Results and Proposals. In: Genetic Programming and Evolvable Machines 14(1). pp. 3-29 (2013)
8. Woodward, J., Martin, S. and Swan, J.: Benchmarks That Matter for Genetic Programming. In: Woodward, J. et al., (ed.) 4th Workshop on Evolutionary Computation for the Automated Design of Algorithms, GECCO '14. pp. 1397-1404 (2014)
9. Jiju, A.: Design of Experiments for Engineers and Scientists, Elsevier (2003)
10. Koza, J.R.: Scalable Learning in Genetic Programming Using automatic Function Definition. In: Kinnear, K.E. Jr., (ed.) Advances in Genetic Programming, pp.99-117 (1994)
11. Ashlock, D. and Joenks, M.: ISAc lists: a Different Program Induction Method. In: Koza, J.R. et al, (ed.) Proceedings of the Second Annual Conference on Genetic Programming, pp.18-26 (1998)
12. Ashlock, D. and Freeman, J.: A Pure Finite State Baseline for Tartarus. In: Evolutionary Computation, Proceedings of the 2000 Congress on, pp.1223-1230 (2000)
13. Dick, G.: An Effective Parse Tree Representation for Tartarus. In: Blum, C. et al, (ed.) Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, GECCO '13. pp.1397-1404 (2013)