Multiple Pass Monte Carlo Tree Search

Cameron McGuinness

***Abstract*—A new variant of Monte Carlo Tree Search called Multiple Pass Monte Carlo Tree Search is introduced in this study. It is then applied to two test domains: Map Generation and Real Parameter Optimization. For Map Generation it was found to create maps that are more intuitive given the fitness function than those created with evolutionary algorithms. For real parameter optimization a novel algorithm called Iterated Splitting was tested using the functions from the CEC 2014 competition and compared against results from a similar algorithm. The Iterated Splitting method performed better on nearly half of the problems, often obtaining an improvement measured in orders of magnitude. Using multiple passes of MCTS enables a simple decomposition of problems, leading to improved performance.**

Keywords: Search based procedural content generation, automatic game content generation, monte carlo tree search, level generation, scalable content generation, multiple pass MCTS, real optimization.

I. Introduction

Monte Carlo Tree Search (MCTS) [5] is a best-first search algorithm used for tasks in which there are sequential decisions being made. MCTS was developed for and is mostly used in the playing of games. There are four main parts to the MCTS algorithm: selection, expansion, simulation, and back-propagation. The selection step traverses the tree to find a node that has not been expanded yet, making the moves in the game that occurs along the tree. The way that the tree gets traversed is controlled by the selection method. The expansion step adds one or more nodes to the game tree from the node reached in the selection step by selected amongst the children randomly. After the expansion step is performed, one of the new nodes is chosen and the simulation step occurs from that node. The simulation is done by finishing the game from the current state. After the simulation, the score of the game (for example, win or lose) is recorded and back-propagated up the tree to all of the nodes that were visited in the selection step. Each node has it's average score updated and the number of times it has been visited incremented. These steps continue until a set stopping point has been reached and the best next move is returned. There are many variants of MCTS due to the need to manage the trade off between exploration and exploitation while traversing the tree.

A. Multiple Pass MCTS

For Multiple Pass MCTS (MP-MCTS) the MCTS algorithm is run multiple times with the search space reduced in size each time it is run. This can only be used with tasks that can be broken up into multiple stages. There are two main parameters to this algorithm that need to be tuned. Those are the number of passes to make before returning the final result, and how much progress to make each pass. These parameters greatly depend on the application Multiple Pass MCTS is used for. This algorithm is similar to the *restarting-recentering* algorithm designed for use in evolutionary algorithms in [1]. Restarting-recentering is an algorithm in which the starting point for a generative chromosome is changed to the best found so far after a set number of generations in the evolutionary algorithm.

B. Map Generation

This study introduces a novel application of Monte Carlo Tree Search (MCTS) to generating level maps building on the previous work described in [2], [3], [8], [9], [7]. This is a type of *automatic search-based procedural content generation*. Automatic search-based procedural content generation is usually game content that is generated using an optimization technique such as evolutionary algorithms, or MCTS ([12]).

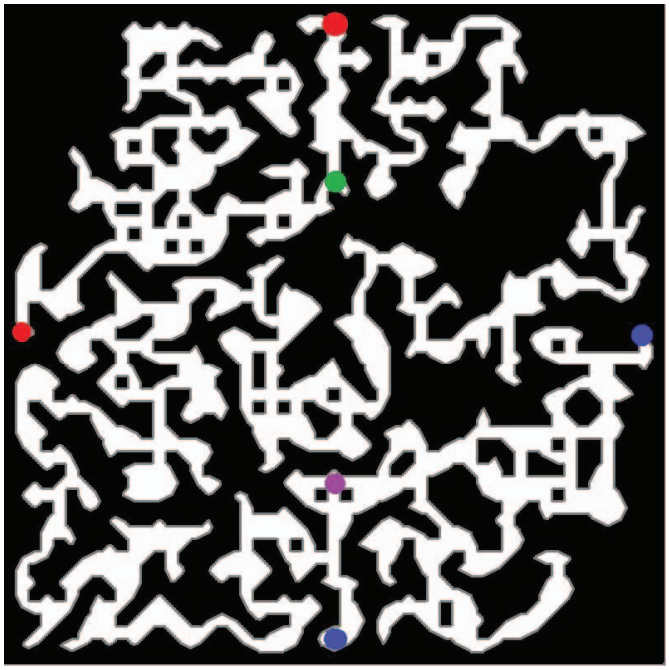

Fig. 1. An example map generated with the direct representation. Checkpoints are given by colored circles.

1) Direct Representation for Maps: A *direct* representation, in which open and blocked squares within a rectangular grid are specified directly as a long, binary gene. This representation is primarily used to contrast different fitness

Cameron McGuinness is at the Department of Mathematics and Statistics at the University of Guelph in Guelph, Ontario, Canada, N1G 2W1. email: cmcguinn@uoguelph.ca

The author thanks the Natural Sciences and Engineering Research Council of Canada for their support of this research.

978-1-5090-0623-6/16/$31.00 ©2016 IEEE

functions. For an $X \times Y$ board the direct representation is a string of XY bits, with 0 representing an empty square and 1 representing a blocked square. An example map evolved using this representation on a 51×51 board is given in Figure 1.

2) Definitions:

Definition 1: The set of *checkpoints* of a grid is a predetermined subset of the squares of the grid, this set is denoted by C.

Definition 2: The length of a minimal length path between any square x and another square y is denoted as $|x, y|$.

3) Fitness Function: There are many fitness functions that can be used to design mazes. The fitness function used for the maps in this study is called *Distances Between Checkpoints*.

```
0 1 2
1 0 2
2 2 0
```

Fig. 2. An example 3×3 specification of checkpoint distance requirements.

Distances Between Checkpoints: This fitness function gives the user good control over how an evolved level looks. For each pair of checkpoints the user must choose whether the distance between them should be minimized, maximized, or neither. To do this we create a $|C| \times |C|$ matrix in which a value of 0,1 or 2 in the i, jth element denotes neither, minimize, and maximize between checkpoints i and j respectively. An example matrix with $|C| = 3$ is given in Figure 2.

If M is the sum of distances between checkpoints whose mutual distance is to be maximized, and m is the sum of distances between checkpoints whose mutual distance is to be minimized then the fitness function is then given by

$$F_2 = \frac{M + 1}{m + 1} \tag{1}$$

4) Sparse Initialization: It was found that a technique called *sparse initialization* was required for some of the representations. Since almost all of the randomly generated maps using the direct representation with a probability of 0.5 of filling a square were invalid (no path between checkpoints) a method that allowed most of the initial maps generated was necessary. For the direct representation this was done by only filling 5% of the map with filled squares and leaving the rest of the map open. This allowed there to be a path between the checkpoints at the start of evolution and the evolutionary algorithm filled in the remaining squares as needed. Using sparse initialization creates maps with low fitness for evolution to built off of to create maps with higher fitness. If sparse initialization was not used, the evolutionary algorithm would eventually find viable maps, however, they would be less varied since all of the descendants would be coming from the one population member that happened across a viable map. Using sparse initialization allows the evolutionary algorithm to start with a population containing many viable maps to build from.

Even with using the exact same fitness function, each representation creates maps with a very different look and feel. This allows a tool to be created that permits a designer to choose a representation to use based on what style of map they are looking for. This can be considered analogous to a paint brush tool in an image editing program.

C. Real Parameter Optimization

A similar approach to MCTS called *Bandit optimization* was used in [11] with good results. The algorithm that was used is called Simultaneous Optimistic Optimization (SOO) introduced in [10]. It is a deterministic algorithm that assumes the objective functions it is being used on is locally smooth near the global optima. SOO starts with the whole search domain as one *cell*. It then splits the cell into 3 partitions and checks the objective function value at the centre of each partition. A search tree is created of each cell created. The next step of the algorithm checks all of the cells at each depth that haven't been split and splits the one with the best objective function value. This continues until a set number of objective function evaluations have occurred and the best found value is returned.

The novel Iterated splitting algorithm requires there to be a defined search space as it starts with the whole space and iteratively reduces the search space based on a set of moves. An example of the Iterated Splitting algorithm for two dimensions is given in Figure 3. Start with full bounded domain, in this case [-100,100] in each dimension. Split into 3 parts in the x dimension and choose one. For this example, the leftmost part is chosen. The new search domain is now just that part. Then repeat in the y direction. This would continue as many times as deemed necessary.

II. Experimental Design

A. Map Generation

The nodes in the game tree created by the MCTS algorithm were any position in the map that is currently empty. This means that the MCTS algorithm will be choosing one wall to place for each move it is making. The selection method employed is UCB1[4] with parameter setting 2. The simulation step of MCTS continues adding walls until a set percentage of the board is filled. That percentage value is the stages for MP-MCTS. This gives us two parameters we need to adjust: how the percentage steps up per stage and how many iterations of MCTS to perform for each stage. The values of each of those that were tested are given in Table I. The minimum value for percentage filled steps is given by $\frac{1}{\texttt{width} \times \texttt{height}}$. This allows the algorithm to add just one point at a time if it improves fitness. After each step, the MCTS implementation is given the previously best found map to start with and attempt to fill to improve the fitness. The experiments generate 51×51 maps. The fitness function used for the maps is the one described in Section I-B3 with requirement matrix given in Figure 4. The checkpoints are located at $[(0, 25), (25, 0), (50, 25), (25, 50), (25, 13), (25, 37)]$.

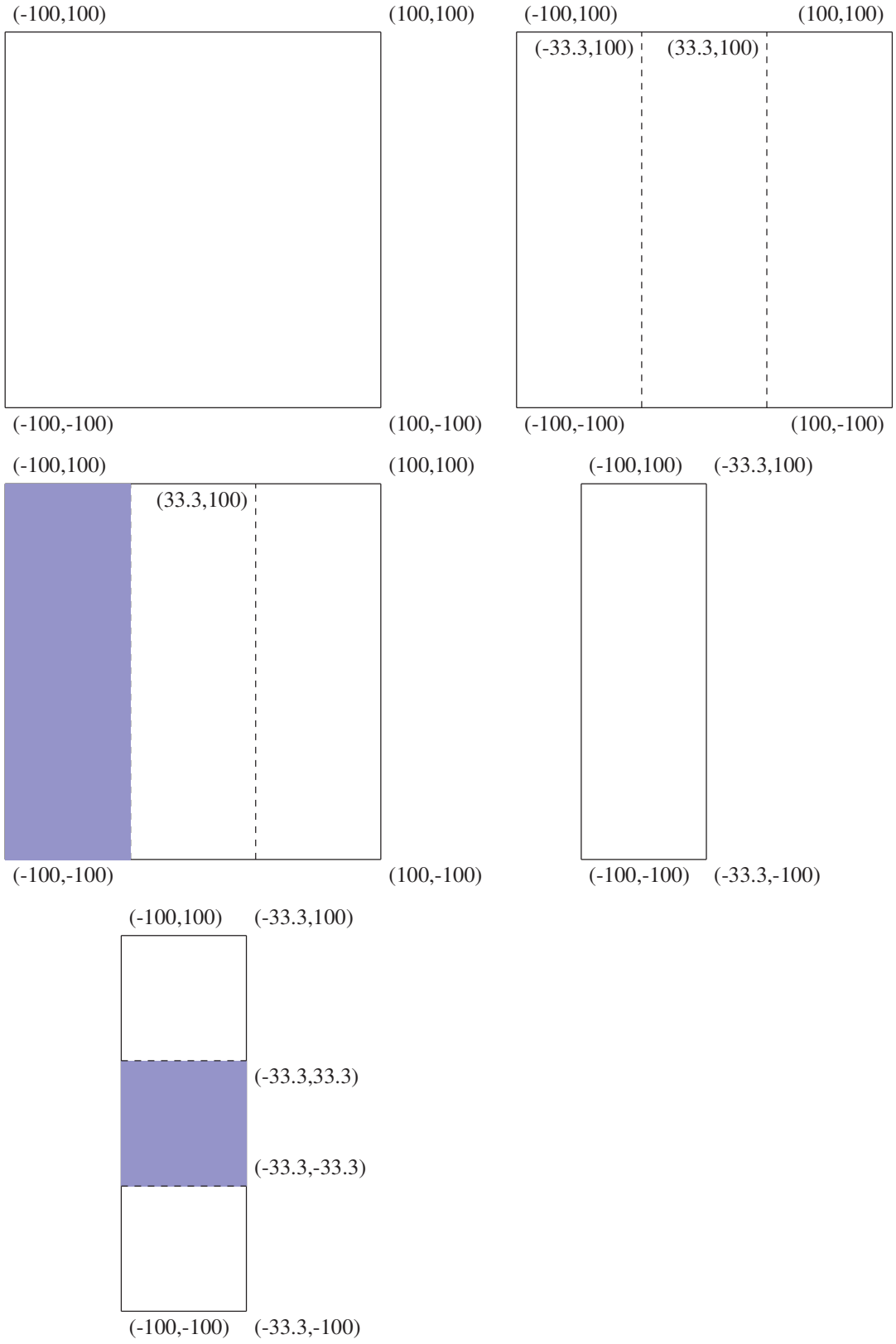

Fig. 3. Example of two-dimensional Iterated Splitting. Read from top to bottom, left to right.

Parameter	values
Percentage Filled Steps	minimum,0.05,0.1,0.2
Iterations	1,10,100,1000

TABLE I
PARAMETER VALUES TO BE TESTED.

This fitness matrix means we want the $(0, 25)$, $(25, 0)$, checkpoints to be far from the $(50, 25)$, and $(25, 50)$ checkpoints, the $(0, 25)$ and the $(25, 0)$ checkpoints to be close to each other, the $(50, 25)$ and the $(25, 50)$ checkpoints to be close to each other, the $(0, 25)$ and $(25, 13)$ close to each other, the $(25, 0)$ and $(25, 13)$ close to each other, the $(50, 25)$ and $(25, 37)$ close to each other, and the $(25, 13)$ and $(25, 37)$ checkpoints close to each other. All other combinations of checkpoints have no restrictions on them.

$$\begin{matrix} 0 & 1 & 2 & 2 & 1 & 0 \\ 1 & 0 & 2 & 2 & 1 & 0 \\ 2 & 2 & 0 & 1 & 0 & 1 \\ 2 & 2 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{matrix}$$

Fig. 4. The requirement matrix used for the experiments.

B. Real Parameter Optimization

The moves for the MCTS implementation will be which piece of the split dimension to take as the next area of focus. This continues for a set number of moves each iteration of the MCTS run. There are three parameters that need to be tuned for the iterated splitting algorithm: number of moves to perform per pass, the number of passes to perform and the number of ways to split each dimension. The values that were tested for these are $[10, 20, 50, 100]$, $[2, 3, 5, 10]$ and $[1, 2, 5, 10]$ respectively. For each of the 64 combinations of the parameters 30 runs were performed with the average and best fitness values recorded. A total of 100,000 function evaluations were performed split equally among each of the passes.

The best values found for each of the 30 functions in 10 dimensions defined in [6] were found over 51 runs and compared to each other and the results obtained in [11]. The results will be compared to [11] since the Iterated Splitting method is based off of the method they created.

III. RESULTS

A. Map Generation

The fitness vs. fill percentage graphs for all step sizes are given in Figures 5, 6, 7 and 8. It can be seen from the graphs that minimum step size results in the highest fitness at 1000 iterations. It's possible that more iterations will result in a higher fitness. However, the run time of more iterations make it a less viable option for map generation. It can also be seen that the fitness does not increase noticeably after fill percentage of 40%. This is because as the map gets more filled, there are fewer places to put walls that do not block off checkpoints from each other. The minimum step size allows the algorithm to test adding just one wall each iteration of MCTS if it has found a new best map in the previous step. If it hasn't found a new best in the past N steps, it will add N walls each iteration of MCTS. With larger step sizes the complexity of the optimization performed by MCTS increases as more walls must be added to finish a map and get a fitness estimate. The larger step sizes require trees with larger search depth. By increasing the number of iterations given to each step, the algorithm is allowed more simulations to find a better map.

Example maps generated with 1000 iterations and minimum step size are given in Figure 9. These maps look very different than the ones generated by evolution (Figure 1). In the evolution-generated maps, there tend to be paths from checkpoint to checkpoint, short paths when the checkpoints are meant to be close and long winding ones when they are meant to be far. In the MCTS-generated maps, These maps make more intuitive sense, given the fitness function, than the maps generated by evolution, the checkpoints that are meant to be close to one another are in a large room together and the ones that are meant to be far away from each other are separated by long winding corridors. These maps could make decent single player first-person shooter maps as there are lots of large spaces intermixed with corridors that have plenty of chokepoints.

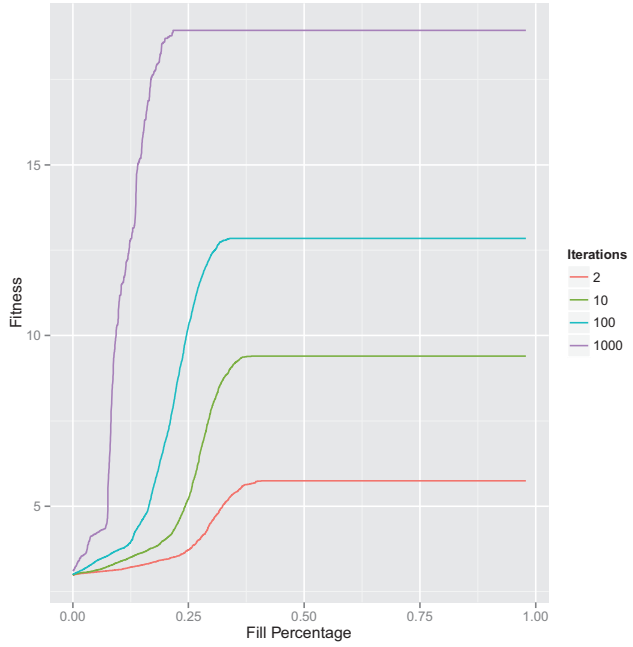

Fig. 5. Fitness vs. Fill Percentage for minimum step size. From this graph it can be seen that the higher iteration values create maps will better fitness.

B. Real Parameter Optimization

The best parameter values are given in Tables II and III for average fitness and best fitness respectively.

The results of all algorithms and from [11] are given in Table IV. For 2 out of the 3 unimodal functions Iterated

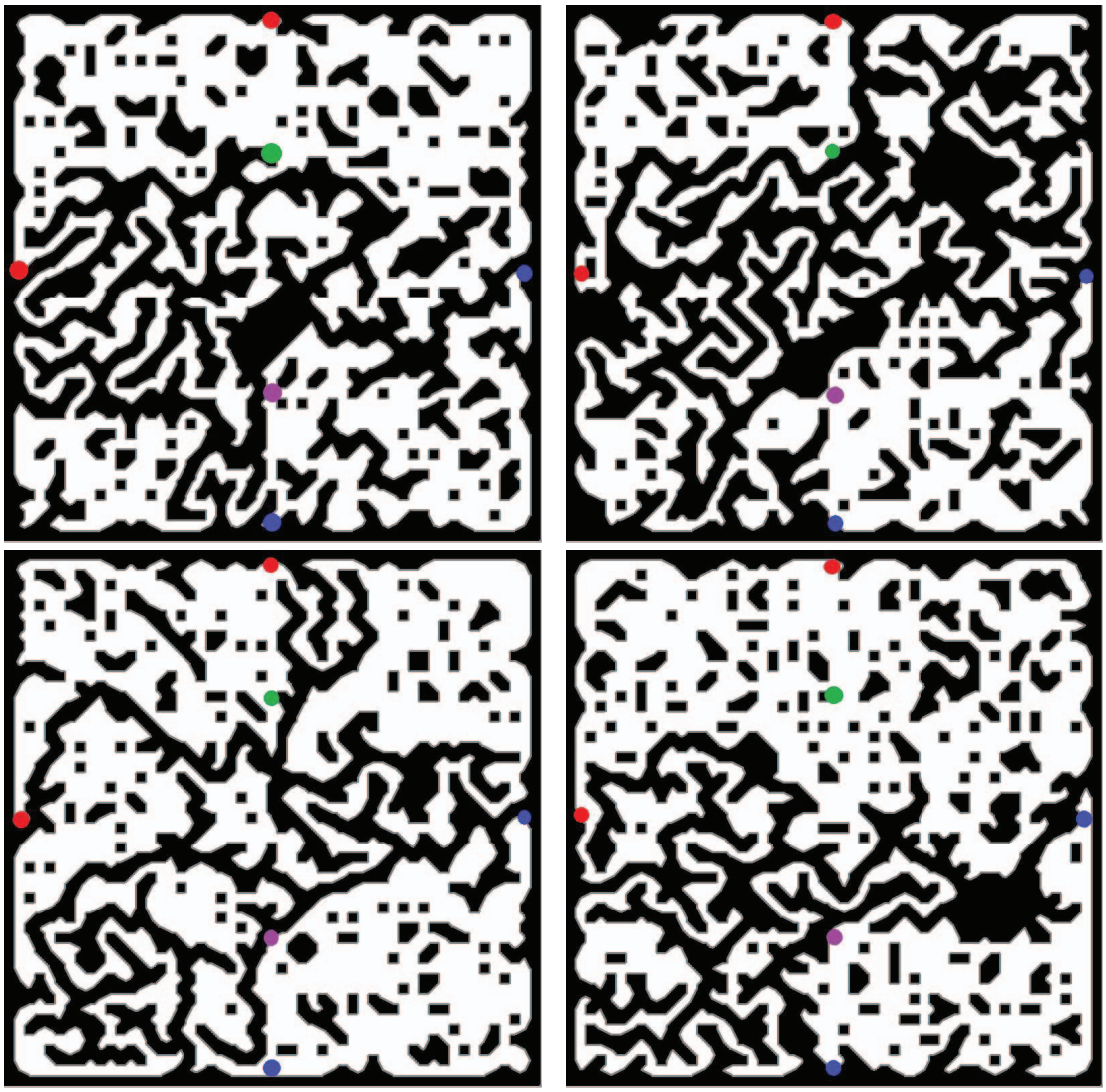

Fig. 9. Examples maps for 1000 iterations and minimum step size. The red checkpoints are required to be close to the green checkpoint, the blue ones, close to the purple one, and the red ones are required to be far from the blue ones. All other combinations have no requirements.

Function Class	#Moves	#Passes	#Splits
Unimodal	10	5	3
Multimodal	10	10	3
Hybrid	10	2	10
Composition	10	1	3

TABLE II
BEST PARAMETER SETTINGS FOR ITERATED SPLITTING BASED ON AVERAGE FITNESS.

Function Class	#Moves	#Passes	#Splits
Unimodal	10	10	5
Multimodal	10	10	5
Hybrid	20	5	10
Composition	10	Any	3 or 5

TABLE III
BEST PARAMETER SETTINGS FOR ITERATED SPLITTING BASED ON BEST FITNESS.

Splitting had the best results and Preux et al had the other best result. For 5 out of 13 of the multimodal functions Iterated Splitting had the best results with Preux et al performing the best on the other 8. For 5 out of the 6 hybrid functions, Iterated Splitting had the best result and Preux et al had the other best result. On the functions that Iterated Splitting had the better results on, it was by at least one order of magnitude each time. For the composition functions, Iterated Splitting had the best result on 1 out of 8 of the functions and Preux et al had the best on 3 out of the 8 with 4 of the functions being tied between all 3 algorithms. In total, Iterated Splitting did the best on 13 out of the 30, Preux et al did the best on 13 out of the 30, and both tied on 4 out of the 30.

IV. Conclusions and Next Steps

For map generation, a new algorithm based on MCTS was introduced called Multiple Pass Monte Carlo Tree Search. This algorithm permits the build up of an object piece by piece by only filling a percentage of the object each pass

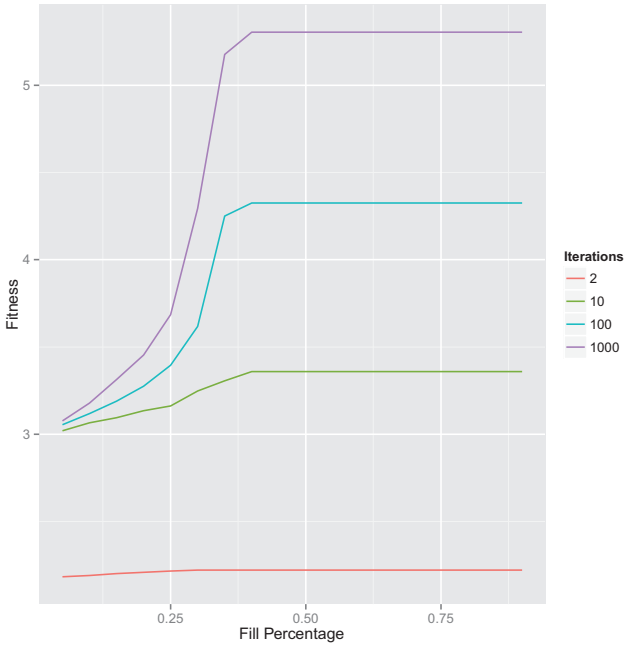

Fig. 6. Fitness vs. Fill Percentage for 0.05 step size. From this graph it can be seen that the higher iteration values create maps will better fitness. The fitness value for 2 iterations is very flat likely due to not having enough simulations to find a decent map.

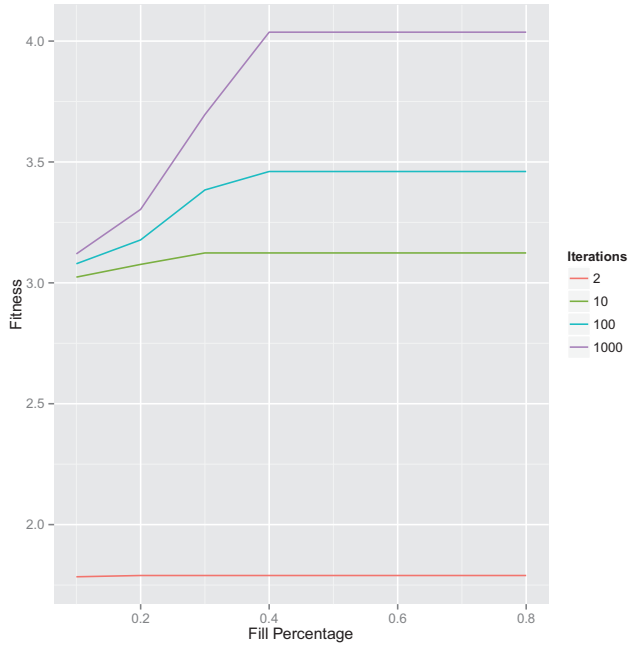

Fig. 7. Fitness vs. Fill Percentage for 0.1 step size. From this graph it can be seen that the higher iteration values create maps will better fitness. The fitness value for 2 iterations is very flat likely due to not having enough simulations to find a decent map.

of the algorithm. This was then used to create maps that followed requirements on distances between checkpoints in the map. It was found that it created maps that were more intuitive and qualitatively different than those created by evolutionary algorithms.

For real parameter optimization, Multiple Pass MCTS was used again with a novel way to discretize a search space using MCTS called Iterated Splitting. This method was tested on a number of functions from [6]. It was found that Iterated Splitting did better on 13 out of the 30 functions, the method from [11] did better on 13 out of the 30 functions, and both methods tied on 4 of the functions.

A. Next Steps

Testing the algorithm with other representations for map generation will be an early extension of this study. Another next step will be testing the algorithm with other MCTS selection methods to see if they give different results.

One possible representation to try for map generation would be to put down *seeds* spaced throughout the map and have MCTS decide which seed to grow and in which direction. The walls would grow until the either hit a barrier or are within a set distance away from another wall. This would be a complicated system to implement with evolutionary algorithms but much simpler with MCTS. With a string-based evolutionary algorithm, it would be a string of indeterminant length (or set length, and cyclic through it until no moves can be made) with an 8N character alphabet where N is the number of seeds placed. For MCTS, the stopping point is easier to determine, as it would be when no seeds can grow any further.

Extending these techniques to larger dimensions in real parameter optimization will be the first priority of this work. It is not yet clear if the techniques will do as well at higher dimensions.

Extending the Multiple Pass MCTS algorithm to other generative representations is another task that will be done in the near future. One area that will be tested will be to use it for multi-criteria optimization. The first pass creates an object using one of the criteria as the fitness function, the next pass alters the object using another criteria, etc. until all criteria have been iterated through.

Another way to use Multiple Pass MCTS would be to use it for a complete role-playing game dungeon generator. The first pass would be creating the level using one of the representations presented earlier, the next task would be placing monsters using a different fitness function for monster placement. The next could be placing treasures or traps using yet another fitness function.

References

[1] Daniel Ashlock and Colin Lee. Characterization of extremal epidemic networks with diffusion characters. In *Computational Intelligence in Bioinformatics and Computational Biology, 2008. CIBCB'08. IEEE Symposium on*, pages 264–271. IEEE, 2008.

[2] Daniel Ashlock, Colin Lee, and Cameron McGuinness. Search-based procedural generation of maze-like levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):260–273, 2011.

[3] Daniel Ashlock, Colin Lee, and Cameron McGuinness. Simultaneous dual level creation for games. *Computational Intelligence Magazine, IEEE*, 6(2):26–37, 2011.

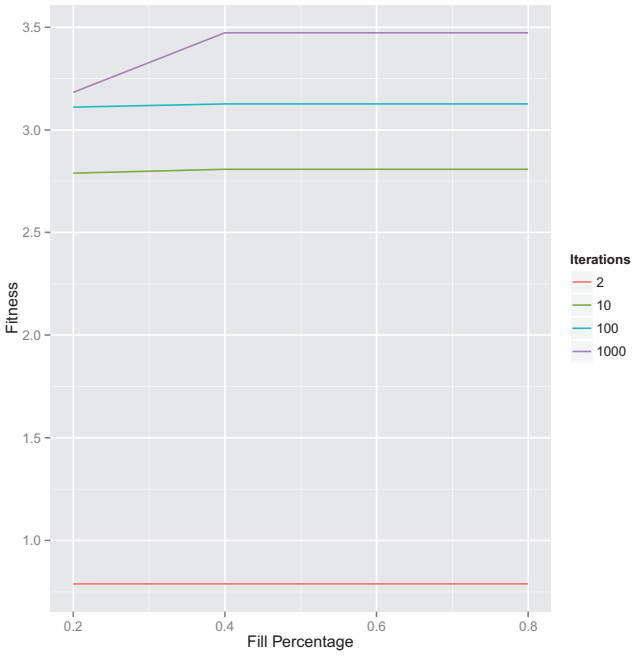

Fig. 8. Fitness vs. Fill Percentage for 0.2 step size. From this graph it can be seen that the higher iteration values create maps will better fitness. The fitness value for 2 iterations is very flat likely due to not having enough simulations to find a decent map.

[4] Cameron Browne, Edward J. Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Trans. Comp. Intell. AI Games*, 4(1):1–43, 2012.

[5] Guillaume Maurice Jean-Bernard Chaslot, Sander Bakkes, István Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, pages 216–217, Stanford Univ., California, 2008.

[6] JJ Liang, BY Qu, and PN Suganthan. Problem definitions and evaluation criteria for the cec 2014 special session and competition on single objective real-parameter numerical optimization. *Computational Intelligence Laboratory, Zhengzhou University, Zhengzhou China and Technical Report, Nanyang Technological University, Singapore*, 2013.

[7] Cameron McGuinness. Statistical analyses of representation choice in level generation. In *Computational Intelligence and Games*. IEEE, 2012.

[8] Cameron McGuinness and Daniel Ashlock. Decomposing the level generation problem with tiles. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, pages 849–856. IEEE, 2011.

[9] Cameron McGuinness and Daniel Ashlock. Incorporating required structure into tiles. In *Computational Intelligence and Games (CIG), 2011 IEEE Conference on*, pages 16–23. IEEE, 2011.

[10] Rémi Munos. Optimistic optimization of deterministic functions without the knowledge of its smoothness. In *Advances in neural information processing systems*, 2011.

[11] Philippe Preux, Rémi Munos, and Michal Valko. Bandits attack function optimization. In *IEEE Congress on Evolutionary Computation*, Beijing, China, July 2014.

[12] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.

Function	Preux	IS-avg	IS-best
1	8.8×10^6	5.3×10^5	$\mathbf{3.8 \times 10^5}$
2	**6.343**	2.9×10^5	7.0×10^7
3	6643.670	**868.717**	9564.13
4	**0.678**	4.923	0.895
5	**20.0**	20.117	20.02
6	**0.002**	0.581	2.987
7	**0.049**	0.530	3.474
8	18.904	22.736	**7.374**
9	8.955	18.484	**8.917**
10	**130.39**	261.359	148.700
11	349.05	**237.610**	291.808
12	**0.0**	0.198	0.0344
13	**0.03**	0.143	0.151
14	0.13	**0.058**	0.111
15	**0.44**	1.120	0.599
16	2.52	2.325	**1.897**
17	3.1×10^6	7.8×10^5	**4148.993**
18	12932.1	5.0×10^6	**167.714**
19	**0.55**	5.100	5.764
20	9364.20	5.1×10^5	**133.213**
21	24694.9	27405.513	**799.768**
22	126.46	221.9	**27.765**
23	200.0	200.0	200.0
24	**115.65**	196.11	196.11
25	**145.16**	179.7	179.7
26	**100.05**	101.55	101.55
27	200.0	**134.036**	**134.036**
28	200.0	200.0	200.0
29	200.0	200.0	200.0
30	200.0	200.0	200.0

TABLE IV

BEST FITNESS FOUND FOR EACH OF THE 30 BOUNDED DOMAIN FUNCTIONS FOR ITERATED SPLITTING AND IN [11]. BOLDED ENTRIES ARE THE BEST FOUND AMONGST ALL 3 ALGORITHMS.