Proceedings of the Sixteenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE-20)

Tree Search versus Optimization Approaches for Map Generation

Debosmita Bhaumik,[1*] **Ahmed Khalifa,**[1*] **Michael Cerny Green,**[1,2] **Julian Togelius**[1,2,3]
[1]New York Univeristy, [2]OriGen.AI, [3]Modl.AI
debosmita.bhaumik01@gmail.com, ahmed@akhalifa.com, mike.green@nyu.edu, julian@togelius.com

Abstract

Search-based procedural content generation uses stochastic global optimization algorithms to search for game content. However, standard tree search algorithms can be competitive with evolution on some optimization problems. We investigate the applicability of several tree search methods to level generation and compare them systematically with several optimization algorithms, including evolutionary algorithms. We compare them on three different game level generation problems: Binary, Zelda, and Sokoban. We introduce two new representations that can help tree search algorithms deal with the large branching factor of the generation problem. We find that in general, optimization algorithms clearly outperform tree search algorithms, but given the right problem representation certain tree search algorithms performs similarly to optimization algorithms, and in one particular problem, we see surprisingly strong results from MCTS.

Introduction

Generating levels for games is a research problem with broad relevance across most game genres and many domains outside of games. Video games, from shooters to role-playing games to puzzle games, need level generation in order to create larger and more replayable games, adapt games to players, simplify game development, and enable certain kinds of aesthetics. Domains such as architecture, urban planning, military simulation and logistics need scenario and environment generation for similar reasons, and these problems are often very similar to game level generation. In reinforcement learning, level generation allows for creating variable environments which helps with generalization (Justesen et al. 2018). For these reasons, the past decade has seen considerable interest in research on level generation and other forms of procedural content generation (PCG) (Shaker, Togelius, and Nelson 2016).

One approach to the generation of levels as well as other types of game content is to use evolutionary algorithms or similar global stochastic optimization algorithms to search for good levels. This approach, called search-based PCG, requires that the levels are represented in such a way that the level space can be efficiently searched, and that there is a fitness function which can reliably approximate content quality (Togelius et al. 2011).

As an alternative to using evolutionary methods, tree search methods such as Monte Carlo Tree Search (MCTS) (Browne et al. 2012) have been suggested for PCG. While it seems that both stochastic optimization and tree search can be used for level generation (and many related generative tasks), we have only been able to find a handful of papers doing PCG by tree search (see next section). Given the very different ways in which these algorithm types search, it stands to reason that they should differ sharply in performance depending on the objective and representation. Maybe there are domains where tree search significantly outperform optimization methods?

This paper systematically compares several tree search algorithms with optimization algorithms on three different 2D level generation problems, across three different representations. We attempt to answer the question which tree search algorithms can have similar or better performance with optimization for level generation, and when.

Background

Procedural content generation (PCG) is the automatic generation of game content, be it game levels, characters, quests, storylines, game elements like trees and rocks, or even entire games themselves (Shaker, Togelius, and Nelson 2016). Search-based PCG is a subset of PCG methods that relies on search or optimization methods (Togelius et al. 2011). In practice, evolutionary algorithms are most commonly used. This section describes previous research in the areas of tree search and evolution as well as procedural content which can be generated using these methods.

Tree Search

Tree search algorithms try to find solutions by starting at a root node and expanding child nodes in a systematic way. Popular techniques include Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best First Search (GBFS), and Monte Carlo Tree Search (MCTS) (Russell and Norvig 2016; Browne et al. 2012). Tree search agents

[*]Both authors contributed equally to this research.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

are commonly used in game-playing agents, like for the Mario AI Benchmark (Togelius, Karakovskiy, and Baumgarten 2010), Chess (Campbell, Hoane Jr, and Hsu 2002), Go (Gelly et al. 2006; Silver et al. 2016), and general video games (Perez-Liebana et al. 2016) among many others.

In the area of PCG, few examples exist of using tree search to generate game content. Browne (2013) first explored this concept by using a variant of the Upper Confidence Bound for Trees equation (UCT) called *Upper Confidence Bounds for Graphs* (UCG) to develop biominoes, simple polyomino[1] shapes and the Pentominoes puzzle domain. Summerville, Philip, and Mateas (2015) generated levels for Super Mario Bros (Nintendo 1985) using Markov Chains where the exploration was guided using Monte Carlo Tree Search. Kartal, Koenig, and Guy (2013) used MCTS to generate stories, taking advantage of MCTS' ability to successfully navigate the large search spaces associated with possible character actions and reactions within narratives. Kartal, Sohre, and Guy (2016) also used MCTS to generate *Sokoban* (Imabayashi 1981) levels. At each node in the MCTS tree, the level generator is given choices to take to modify the level, such as deleting/adding objects and moving an agent around within the level to simulate gameplay. Graves and others (2016) experimented with using MCTS to generate Angry Birds (Rovio Entertainment 2009) levels. At each node in the tree, the level generator can place/remove structures/pigs or do nothing at all. Finally, exhaustive search can also be used to create all possible content artifacts in some space (Sturtevant and Ota 2018).

Optimization

Global optimization algorithms are algorithms which focus only on finding a good solution, which maximizes or minimizes some objective, not on the path leading from an origin state to that solution. Evolutionary algorithms, a family of stochastic population-based algorithms, are a good representative of this class. Such algorithms are popular choices for PCG because it is easy to frame the PCG as a single-point or population-based optimization problem, where the fitness functions/objectives can be cleanly mapped to game elements like difficulty, time, physical space, level variety, etc (Togelius et al. 2011). Ashlock did this several ways, such as optimized puzzle generation for different difficulties (Ashlock 2010), or stylized cellular automata evolution for cave generation (Ashlock 2015). McGuinness and Ashlock (2011) created a micro-macro level generation process, using a wide variety of fitness functions based on level elements. Shaker, Shaker, and Togelius (2013) evolved levels for *Cut the Rope* (ZeptoLab 2010) using constrained evolutionary search where the fitness measures the playability using playable agents. In addition to evolving level elements in GVGAI (Khalifa et al. 2016), PuzzleScript (Khalifa and Fayek 2015a), and Super Mario Bros (Khalifa et al. 2019), Khalifa and Fayek (2015b) offers a literature review of search based level generation within puzzle games.

[1] Orthogonally connected sets of squares (Golomb 1996).

Methods

In this paper, we compare tree search algorithms to optimization algorithms. We decided to compare them over three different problems (Binary, Zelda and Sokoban) which were introduced as part of the PCGRL framework (Khalifa et al. 2020). We settled on these problems as they cover different types of games and heuristic/fitness functions already exist. Also, variants of these generation problems had been tackled before using optimization algorithms (Ashlock, Lee, and McGuinness 2011; Ashlock 2018; Khalifa et al. 2016; Charity et al. 2020; Khalifa and Fayek 2015a), meaning that we already know of effective problem representations for optimization. For all the used algorithms, the tree search algorithm stops either when it finds a solution to the problem or time runs out. In this section, we will talk about the different algorithms, representations, problems used in this work.

Tree Search Algorithms

We compare four simple and commonly used tree search algorithms, two of which are uninformed search (Breadth First Search and Depth First Search) and two are informed search (greedy best first search and Monte Carlo Tree Search). For the first three algorithms we use the canonical versions as defined in (Russell and Norvig 2016).

Breadth First Search(BFS) is a simple uninformed search algorithm that expands a full tree level before exploring deeper nodes using a queuing system.

Depth First Search (DFS) is another uninformed search algorithm that always expands one of the nodes at the deepest level of the tree using a stack system. When the search hits a dead end, it goes back and expands nodes at shallower levels.

Greedy Best First Search (GBFS) is an informed search algorithm which uses the cost to reach the goal as heuristic function and a priority queue to select the most promising nodes in the search tree first.

Monte Carlo Tree Search (MCTS) is a stochastic tree-search algorithm (Browne et al. 2012) that creates asymmetric trees by expanding the most promising branches of the search space using random sampling (rollout). There are many variants of MCTS (Browne et al. 2012); we use UCT (Kocsis, Szepesvári, and Willemson 2006); arguably the most widely used version. UCT uses the UCB1 equation to balance between exploration and exploitation:

$$UCB1_i = \frac{V_i}{n_i} + c\sqrt{\frac{\ln N}{n_i}}$$

where V_i is the total accumulated rewards for that node, n_i is the total number of visits for that node, N is the total number of visits of the parent node, and c is a constant balancing between the exploitation term (first term) and exploration term (second term).

Optimization Algorithms

For the optimization algorithms, we explore two single-point optimization algorithms (hill climbing and simulated annealing) and two population-based optimization algorithms (evolution strategy and genetic algorithm).

Hill Climbing (HC) is a single-point optimization algorithm (Russell and Norvig 2016) that initializes a random solution and keeps improving the solution (by comparing it to all the possible neighbors) until a local optimal solution is found.

Simulated Annealing (SA) is a single-point global optimization algorithm (Russell and Norvig 2016) that tries to find a global optimum in the presence of several local optima. Instead of always accepting a better neighbor, it can accept less optimal neighbors with probability less than 1. The probability is calculated by $P = exp(-d/T)$, where d is the absolute difference between the current solution's score and the new solution's score and T is temperature. Temperature is initially given a high value which slowly decreases every iteration using a cooling rate c ($T = T * c$).

Evolution Strategy (ES) is a nature inspired population-based optimization algorithm (Brownlee 2011). It applies selection and mutation operators to a population, that contains solutions, to evolve better and better solutions. The process begins with a random population of $\mu + \lambda$ individuals and calculates the fitness of the entire population using a fitness function. λ worst individuals are removed from the population and replaced with mutated version of the top μ individuals.

Genetic Algorithm (GA) is a population-based optimization technique inspired by the Darwinian principle of evolution (Brownlee 2011). Like ES, it uses nature inspired operators like mutation and selection as well as a crossover operation to generate high quality solutions. Starting with a random population, it selects individuals based on their fitness for reproduction. These individuals produce a new solution using crossover and mutation operators (with different probabilities) which is inserted into the next population. Additionally, the most fit individuals are immediately inserted into the next generation in a process known as *elitism*. The reproductive process goes on until the new population is fully created.

Representations

Problem representation is critical in search-based PCG (Ashlock 2018) as it impacts the speed of generation and the style of the generated content. For example, one way of representing a level generation problem is with a 1 dimensional integer array. This representation lends itself directly to the optimization algorithms where the algorithm is able to modify the level by directly changing a single index in that array. Tree search algorithms need to model this representation within a graph of nodes and connections. We can imagine every map as a node in the space of all possible maps in an entire game, and the connection between these nodes are based on the changes needed to reach that map from the other map. We can then limit these connections to mimic the behavior of optimization algorithms: only one tile is modified to move from node to node. Thus, we can replicate problem representation regardless of the search method used.

In this section, we introduce three different representations (*Narrow*, *Turtle*, and *Wide*) that can transform the generation process into a graph which can be easily traversed using Tree Search Algorithm. Before applying a tree search algorithm, the root node has to be selected. In this work, we select that node randomly, similar to the random starting maps of the optimization algorithms.

Narrow Representation is defined as changing one specified tile at a time. In this approach, the tiles that the algorithm can modify are randomly ordered, and the algorithm can only modify the map in that particular order. This means that each tree node represents the current map and the current modified tile, while the branches represents the modifications that can be done to that tile. This modification decreases the branching factor to be n actions (where n is the number of different game tiles) but it increased the state space by adding the current modified tile as part of the state representation.

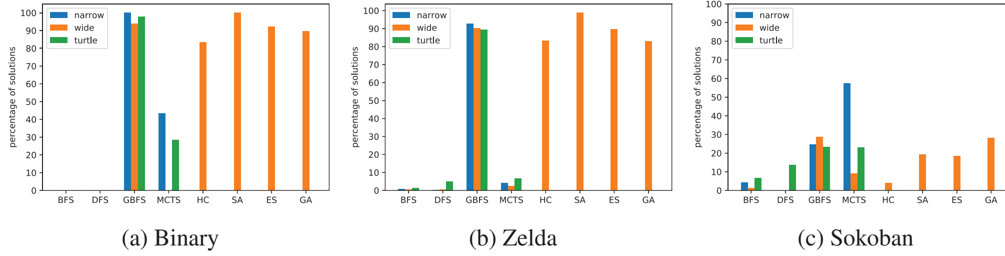
Turtle Representation draws parallels to the Turtle Graphics module in the Logo programming language. In this representation, algorithms are given a random initial position within the map. They are allowed to either change its position by moving to any of the neighboring tiles in the four cardinal directions (unless a direction would take them "out-of-bounds") or modify this tile to another tile value, and the process repeats. This means that a node in the tree represents the map and the turtle's current position, while the branches are the directional movement and the map modification decisions. Similar to the narrow representation, the action space decreased to be $4+n$ actions (4 cardinal movements and n tile modification actions) but it increased the state space by adding the current modified tile as part of the state representation.

Wide Representation is inspired from the optimization algorithms' representation. In this representation, the algorithm itself can decide exactly which tiles to modify in any order. For tree search, this means that a node only represents the current map while the branches are equal to the map size multiply by number of possible tiles where it identifies which tile location can be changed and what is the new tile type.

Problems

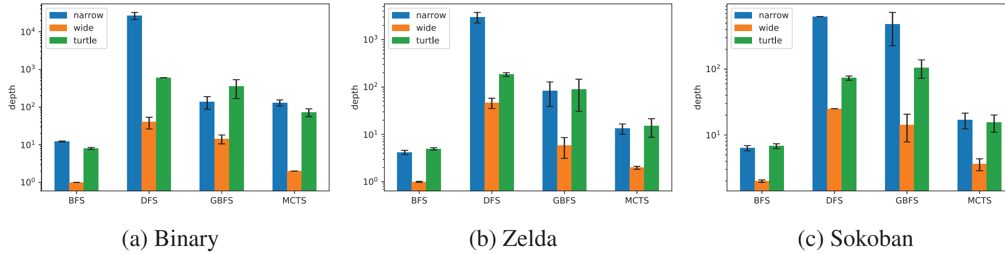
The problems that we are using for the comparison are taken from the PCGRL framework[2]. The framework supports six different problems. In this work, we will focus only on three problems, the same ones used in (Khalifa et al. 2020). Each problem is represented as 2D array of tiles. The heuristic/fitness function is provided through the PCGRL framework for each problem. To evaluate any problem state, the PCGRL framework compares the current state with a reference state (the root state in case of tree search algorithms and a random initial chromosome in case of optimization algorithms). For example, if the goal is to create 2D maze with a long path. If the initial state has a path length of 5 and the current state has a path length of 10, then the fitness/heuristic will be 5. On the other hand, if the current state has a path length of 2, the fitness/heuristic will be -3. If there are multiple goals for the problem, the heuristic/fitness from each

[2]https://github.com/amidos2006/gym-pcgrl



(a) Binary (b) Zelda (c) Sokoban

Figure 1: The playability percentage for all the algorithms over Binary, Zelda, and Sokoban problems.



(a) Binary (b) Zelda (c) Sokoban

Figure 2: The average maximum depth for all the tree search algorithms over 2500 runs. The actual bars are the average value while the black vertical lines show the standard deviation.

goal is combined as a weighted sum[3].

Binary is the simplest problem: the goal is to create a binary map layout where all the empty tiles are connected and the longest shortest path between any two points in the map increases by at least X (X = 20 in this work).

Zelda is inspired by the GVGAI (Perez-Liebana et al. 2016) version of the dungeon system in The Legend of Zelda (Nintendo, 1986). The player has to collect a key and the reach the door without dying from the moving enemies. The generator must take into consideration the goal of the game and must try to generate a playable level. A playable level must have 1 player, 1 key, 1 door, all tiles fully connected, enemy should be Y step away from player and the path between the player and the key as well as the key and the door must be at least X steps (Y = 5,X = 20 in this work).

Sokoban is a port of the famous Japanese game by the same name (Thinking Rabbit, 1982). The goal of the game is to push every crate to a goal location. To achieve that goal, the generated levels has to have 1 player, number of crates equal to number of targets, and can be solved using A* algorithm with at least X steps (X = 20 in this work).

Experiments

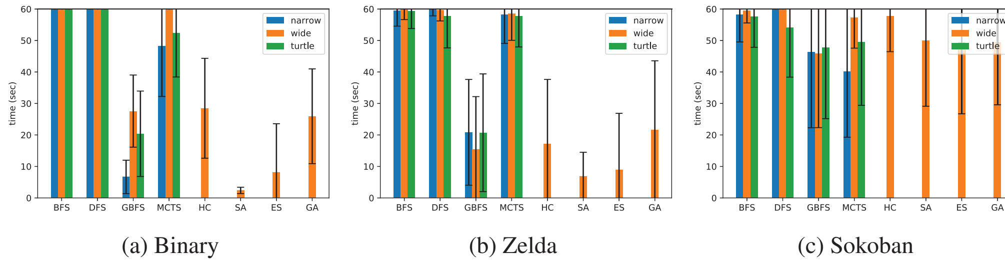
Our generators are configured to create maps of size 14x14 for Binary, 11x7 for Zelda and 5x5 for Sokoban excluding boundaries. We run all the eight algorithms on all the three problems using different representations. Optimization algorithm are only run using wide representation as it is the direct representation used for optimization algorithms in all the previous work. We introduce the narrow and turtle representations to assist Tree Search algorithm manage the large branching factor in the graph (392 for binary, 616 for Zelda, and 125 for Sokoban) which optimization algorithm does not have problem with.

We run each experiment for 2500 runs, where each run was capped at 60 seconds. If the algorithm finds the solution before hand, it terminates and returns the solution, otherwise it continues till time out. For MCTS, the C value is set to 5 to balance with the big values of exploitation term. We also assign the rollout length as 40% of the size of the problem (78 for Binary, 40 for Zelda, and 10 for Sokoban). This variable length is just to allow the MCTS rollout to have an effect on the map, as the bigger the maps gets the deeper they need to be explored randomly to have an effect on the level output. For SA, we start with temperature equal to 10 and cooling rate equal to 0.99. For ES, we use μ equal to 10 and λ equal to 20. Finally, GA uses population of size 30 with elitism of size 1; new individuals are generated either by crossover (80% chance) followed by mutation (5%), or only mutation. Rank selection is used to select parents. For mutation (for all optimization algorithms), we are using single point mutation where a single tile is picked and changed randomly. For crossover (only GA), we are using two point crossover where two points on the map (as a 1D string) are picked and the values between these points are swapped. These values are chosen by earlier experiments that lead to the best result.

Results

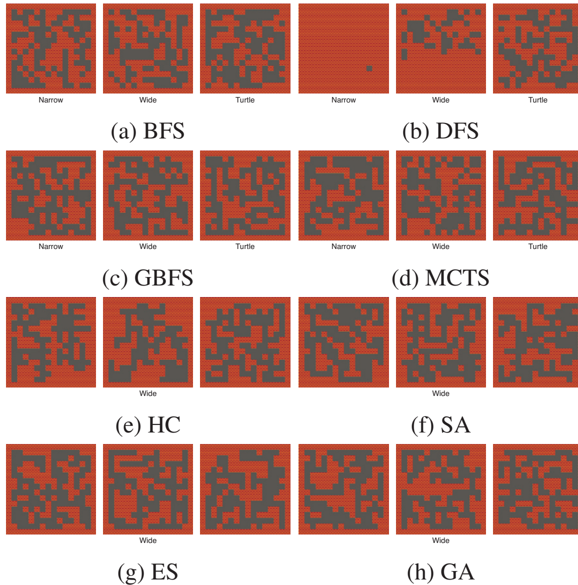
Figure 1 shows the performance of all the different algorithms on our 3 different problems using the possible representations. For the tree search algorithms, Greedy algorithms such as GBFS outperformed most of the other algorithms, its performance was the same between the different representations on all the problems. This is likely due

[3]check the repository for detailed implementation of the fitness/heuristic functions: https://github.com/amidos2006/tsxoa



(a) Binary (b) Zelda (c) Sokoban

Figure 3: The average time in seconds that each algorithm takes to run over all 2500 runs. The actual bars are the average value while the black vertical lines show the standard deviation.



(a) BFS (b) DFS

(c) GBFS (d) MCTS

(e) HC (f) SA

(g) ES (h) GA

Figure 4: Binary Examples

to the sensitive heuristic function that we are using for our problems. Any change in the map usually corresponds to a change in the heuristic.

On the other hand, BFS and DFS performed extremely low (almost zero on all the problems and representations) which was expected. BFS is unable to search very deep within the tree where most solutions reside (see Figure 2). BFS depth is very low compared to most of the other algorithms. On the other hand, DFS can search very deep within the tree but still performs badly. This is due to DFS' tendency to trap itself in bad initial branch paths, leading to wasted computational time before exploring other branches that could lead to the solution. Figure 2 shows the depth of the deepest found node by the algorithm during search regardless it found a solution or not. If the algorithm found a solution, it will terminate the search at that node (which will make it the deepest node found); if it did not find a solution, it will continue exploring until 60 seconds expire.

MCTS was a different case: it did not search very deep in the tree compared to GBFS due to the exploration factor and the random rollouts. MCTS performed badly on the Binary problem, especially when using wide representation (due to the low depth). On Zelda, MCTS performed worse as well: its performance was almost as bad as DFS and BFS. Looking at the depth, we can see that MCTS built a very shallow tree. We believe this occured due to the big map space (11x7) and the large amount of possible tiles (8 different tiles). The large branching factor lead MCTS to waste most of its computation time on low level nodes as MCTS cannot continue exploring until it expands/simulates/back-propagates all the previous children (which is not the case in GBFS). In Sokoban, MCTS using Narrow representation outperformed all the other algorithms, and at the same time it did not explore deeply in the tree. This is likely due to the deceptive the reward heuristic/fitness landscape of Sokoban (Anderson et al. 2018). Deceptive landscapes were not a issue in the first two problems (Binary and Zelda) as the heuristic was usually provided accurate guidance toward success. In Sokoban, the heuristic sometimes does not always lead the generator toward playable levels. For example: the number of crates equalling the number of targets and that each of these are reachable from the player's location are a good indicators for playability. However, neither guarantee that the level is beatable. Crates could be reachable but be stuck beside a wall or another crate. GBFS might waste more time on searching a branch that leads to a non playable level compared to MCTS (which estimates how good or bad a branch is before committing to it). On the other hand, the MCTS turtle representation did not perform well. The random rollouts are likely a leading factor here: they are less effective as 4 of the selected actions are agent movements, leading to less effective sampling and changes in the heuristic estimation of that branch.

The optimization algorithms generally perform similarly on all the problems. We were surprised to see that SA, in spite of being a single point optimization algorithm, performs slightly better than the other optimization algorithms. We think the temperature is helping it to explore the space faster and reach a good result, while HC, ES, and GA converge pretty fast to a local optimal solution and cannot escape it. We think this can be easily solved by increasing the population size for ES and GA.

An advantage GBFS has over greedy HC algorithm is that GBFS does not get stuck since it can always roll back to any other path to explore if it reaches a local maxima, while HC will always get stuck as it does not keep track of previous

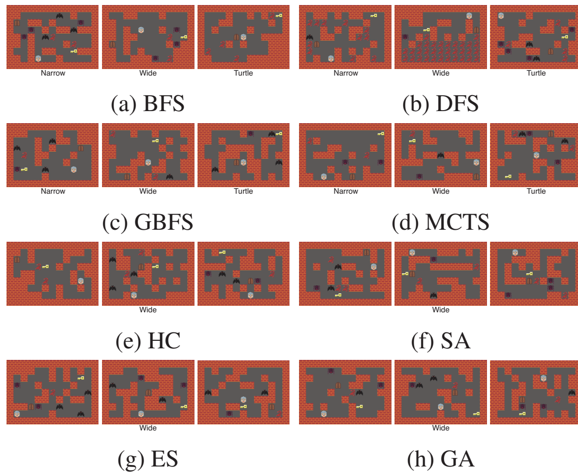


Figure 5: Zelda Examples

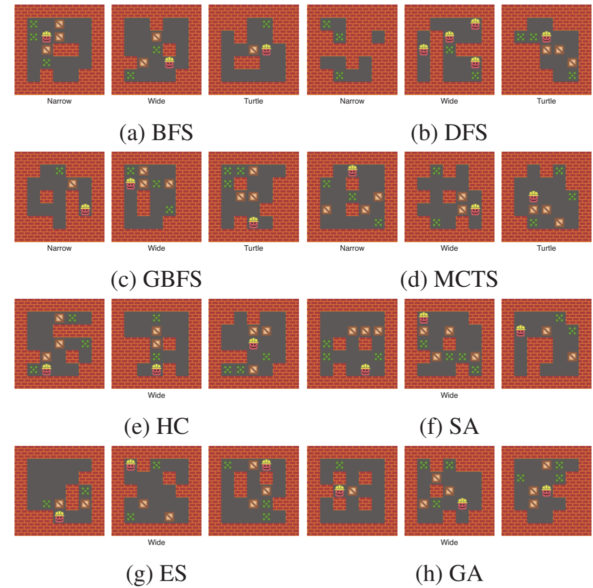


Figure 6: Sokoban Examples

explored solutions. This is a classic trade off between memory and performance where GBFS uses more memory which leads better performance than HC algorithm.

Figure 3 shows the average time in seconds that each algorithm needs to find solution. The optimization algorithms take less time on average to find the results, which was not surprising looking at their performance. GBFS is the only algorithm that can be compared in time due to its greedy nature of visiting the nodes that leads it to the solution quickly (except for Sokoban where it easily gets deceived and does not find the solution). BFS, DFS and wide MCTS are always near the cut off time (60 seconds): they are never able to find a solution before timeout.

Figure 4, 5, and 6 shows examples of the best found levels for every different algorithm on all the three problems. BFS and DFS did not find any solution in most of the problems but it is interesting to look at their best found level. Since the DFS algorithm sticks with a certain action until the end of the tree it forces it to repeat a certain element a lot.

In Figure 4, The BFS maps looks like almost finished very few tiles are not connected, while the DFS algorithm best solutions cover the map with solid tiles such that it ends with having a single connected region which is interesting. Similar behavior can be seen in Zelda (figure 5) and Sokoban (figure 6) regarding the unfinished DFS levels. Another notable change is SA solutions, they are usually have a different structural aesthetics compared to the rest. It can be easily seen in Zelda, the levels looks more horizontally connected than other levels. We theorize that temperature variable allows SA to reach areas that is not easily reached by being greedy or take longer time during hierarchical exploration.

Conclusion

In this paper, we compared four tree search algorithms to four optimization algorithms in level generation for three different problems (Binary, Zelda, and Sokoban). We introduced two new representations (Narrow and Turtle) to battle the problem of the high branching factor that can affect tree search performance badly. We found that GBFS performed very similar to optimization algorithms. While MCTS did not perform well on Binary and Zelda problem, it exceeded our expectations on the Sokoban problem. We think that the random sampling capabilities allows MCTS to avoid getting stuck in a local optima.

In most cases, it is hard to notice the difference between the generated levels. However, there are subtle differences, suggesting that each algorithm has its own "style" as an effect of the way the algorithm searches for a solution. While the optimization algorithms typically reach quantitatively better results, the right tree search algorithms paired with the right representation can solve the same content generation problem in a different way.

We would like to take this work further and investigate population-based tree search algorithms, treating the domain as a graph with multiple starting points similar to the work by Browne (2013). We would also like to extend this work by using bigger maps and more complex problems (such as Super Mario Bros level generation) and compare new state of the art algorithms in both tree search and optimization algorithms. Another direction is to test these techniques on different domains and generative problems and see how well they translate (narrative generation, character generation, sprite generation, rule generation, etc).

Acknowledgements

Ahmed Khalifa acknowledges the financial support from NSF grant (Award number 1717324 - "RI: Small: General Intelligence through Algorithm Invention and Selection."). Michael Cerny Green acknowledges the financial support of the SOE Fellowship from NYU Tandon School of Engineering.

References

Anderson, D.; Stephenson, M.; Togelius, J.; Salge, C.; Levine, J.; and Renz, J. 2018. Deceptive games. In *International Conference on the Applications of Evolutionary Computation*, 376–391. Springer.

Ashlock, D.; Lee, C.; and McGuinness, C. 2011. Search-based procedural generation of maze-like levels. In *IEEE Transactions on Computational Intelligence and AI in Games*, 260–273. IEEE.

Ashlock, D. 2010. Automatic generation of game elements via evolution. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, 289–296. IEEE.

Ashlock, D. 2015. Evolvable fashion-based cellular automata for generating cavern systems. In *Computational Intelligence and Games (CIG), 2015 IEEE Conference on*, 306–313. IEEE.

Ashlock, D. 2018. Exploring representation in evolutionary level design. *Synthesis Lectures on Games and Computational Intelligence* 2(1).

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

Browne, C. 2013. Uct for pcg. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, 1–8. IEEE.

Brownlee, J. 2011. *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu.

Campbell, M.; Hoane Jr, A. J.; and Hsu, F.-h. 2002. Deep blue. *Artificial intelligence* 134(1-2):57–83.

Charity, M.; Green, M. C.; Khalifa, A.; and Togelius, J. 2020. Mech-elites: Illuminating the mechanic space of gvgai. *arXiv preprint arXiv:2002.04733*.

Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modification of {UCT} with patterns in {M} onte-{C} arlo {G} o. *INRIA*.

Golomb, S. W. 1996. *Polyominoes: puzzles, patterns, problems, and packings*, volume 16. Princeton University Press.

Graves, M., et al. 2016. *Procedural content generation of Angry Birds levels using monte carlo tree search*. Ph.D. Dissertation, University of Texas at Austin.

Justesen, N.; Torrado, R. R.; Bontrager, P.; Khalifa, A.; Togelius, J.; and Risi, S. 2018. Illuminating generalization in deep reinforcement learning through procedural level generation. In *NeurIPS Workshop on Deep Reinforcement Learning*.

Kartal, B.; Koenig, J.; and Guy, S. J. 2013. Generating believable stories in large domains. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Kartal, B.; Sohre, N.; and Guy, S. 2016. Generating sokoban puzzle game levels with monte carlo tree search. In *The IJCAI-16 Workshop on General Game Playing*, 47.

Khalifa, A., and Fayek, M. 2015a. Automatic puzzle level generation: A general approach using a description language. In *Computational Creativity and Games Workshop*.

Khalifa, A., and Fayek, M. 2015b. Literature review of procedural content generation in puzzle games. http://www.akhalifa.com/documents/LiteratureReviewPCG.pdf.

Khalifa, A.; Perez-Liebana, D.; Lucas, S. M.; and Togelius, J. 2016. General video game level generation. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, 253–259. ACM.

Khalifa, A.; Green, M.; Barros, G. A.; and Togelius, J. 2019. Intentional computational level design. In *Proceedings of The Genetic and Evolutionary Computation Conference*. ACM.

Khalifa, A.; Bontrager, P.; Earle, S.; and Togelius, J. 2020. Pcgrl: Procedural content generation via reinforcement learning. *arXiv preprint arXiv:2001.09212*.

Kocsis, L.; Szepesvári, C.; and Willemson, J. 2006. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep* 1.

McGuinness, C., and Ashlock, D. 2011. Decomposing the level generation problem with tiles. In *Evolutionary Computation (CEC), 2011 IEEE Congress on*, 849–856. IEEE.

Perez-Liebana, D.; Samothrakis, S.; Togelius, J.; Schaul, T.; and Lucas, S. M. 2016. General video game ai: Competition, challenges and opportunities. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Russell, S. J., and Norvig, P. 2016. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Shaker, N.; Shaker, M.; and Togelius, J. 2013. Evolving playable content for cut the rope through a simulation-based approach. In *AIIDE*.

Shaker, N.; Togelius, J.; and Nelson, M. J. 2016. *Procedural content generation in games*. Springer.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of go with deep neural networks and tree search. *nature* 529(7587):484.

Sturtevant, N. R., and Ota, M. J. 2018. Exhaustive and semi-exhaustive procedural content generation. In *AIIDE*, 109–115.

Summerville, A. J.; Philip, S.; and Mateas, M. 2015. Mcmcts pcg 4 smb: Monte carlo tree search to guide platformer level generation. In *Eleventh Artificial Intelligence and Interactive Digital Entertainment Conference*.

Togelius, J.; Yannakakis, G. N.; Stanley, K. O.; and Browne, C. 2011. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3):172–186.

Togelius, J.; Karakovskiy, S.; and Baumgarten, R. 2010. The 2009 mario ai competition. In *IEEE Congress on Evolutionary Computation*, 1–8. IEEE.