Utrecht University

Master Thesis

Generating Sokoban Levels that are Interesting to Play using Simulation

Author:
Simon Karman
ICA-5521904

Supervisor:
Prof. Dr. F.P.M. Dignum
Second Supervisor:
Dr. M. Löffler

Master Game and Media Technology
Department of Information and Computing Sciences

June 28, 2018

UTRECHT UNIVERSITY

Abstract

Faculty of Science
Department of Information and Computing Sciences

Master of Science

Generating Sokoban Levels that are Interesting to Play using Simulation

by Simon Karman

Procedural Content Generation for Games (PCG-G) is the act of generating content for games using a procedure. There are many valid reason why game creators could be interested in using PCG-G. An example is that artists can be aided in creating massive game worlds, since creating all the content by hand is too time consuming. By using pseudo randomness, procedures are able to generate content useful in games, however without the input of an expert artist, procedures struggle to generate content that is interesting to play.

The focus in this work lies on generating interesting game worlds. Generating interesting game worlds can be subdivided into the generation of the space and the mission of the game world. The abstract equivalent of the mission is a puzzle. A puzzle generator that can generate interesting puzzles can therefore be used to generate interesting worlds.

Previous work has shown that challenging Sokoban puzzles can be generated by simulated play. Although these generated puzzles are slightly challenging, they do not come near the interestingness of handcrafted puzzles. There seems to be an upper limit that the current approach cannot (consistently) exceed. This is mainly due to the use of simple metrics that do not seem to be able to capture the underlying concepts of interestingness.

Seven candidate improvements on a replica of the *foundational* work were created. The Push Alteration and the Hard-coded Symmetry Reduction seemed to be the most promising. These two candidate improvements were taken to the test in an user study experiment. The user study showed that these candidate improvements result in generally more interesting puzzles than puzzles generated by the replica of the *foundational* work, but that they are still much less interesting than puzzles generated by expert artists.

Acknowledgements

First, I would like to thank my supervisor, Prof. Dr. F.P.M. Dignum, and my second supervisor, Dr. M. Löffler, for their assistance during my master thesis. They have always been available for comments, feedback and advice.

Next, I would like to thank Bilal Kartal for granting me access to the code base of the original paper. My thesis heavily relied on this access. Without it, I wouldn't have been able to replicate their results.

Next, I would like to thank all the participants of the user study for helping me validate my hypotheses. The participants in order of participation are: *Nik, Jelle, Kim, Rosemarije, Yentl, Kees, Eileen, Maarten, Simardeep, Wouter, Samuel, Robert, David, Roos, Steven, Sera, Niels, Reinier, Bart, Tommy, Matthijs, Lara, Roman, Thomas, Pia, Bianca, Robin, Yorick, Lisa, Jac., Govie, Marcel, Glenn, Annelies, Nelly, Pieternel, Anjo, Pieter and Geanne.*

Next, I would like to thank Nik, Kevin, Rik, and Jorn for their valuable feedback on the earlier versions of this master thesis.

Lastly, I would like to thank my parents, sister, and girlfriend for their loving support.

Contents

1 Introduction

1.1 Procedural Content Generation for Games (PCG-G)

Players of games demand larger and more detailed game worlds than ever before. Such game worlds are getting too large to be created by hand. Instead of manually creating game worlds, game worlds can also be algorithmically created. (Parts of) The creation process can be executed by a procedure that generates the desired content. This is called Procedural Content Generation for Games: **PCG-G**. The essence is easy to understand: PCG-G is the act of generating content for games using a procedure.

Apart from creating large game worlds, procedural generation of content in games can be used for many different reasons. Some other example reasons of using PCG-G are assisting artists, allowing re-playability, saving disk space or satisfying peoples desire to explore. These four examples are described in the subsections below.

1.1.1 Assisting Artists

The amount of content within worlds is growing quickly, because of two factors. First the details within the world are increasing and secondly the overall size of the worlds are increasing too. Some worlds require such high amounts of content that artists can no longer create everything by hand.

Decreasing the amount of time an artist uses to create content, saves production cost and production time. Simple copy and paste tools are generally not enough. This results in worlds that look too repetitive and still need loads of work by the artist tweaking the results. Offline PCG-G techniques can be used in this case. To assist artists in their work PCG-G could: executing their repetitive tasks including variants, generate parts of the world, or altering parts of the world.

1.1.2 Increase Gameplay Time

The amount time a game can be played can be increased by using PCG-G.

Re-playability

A player can spend more time playing the game if he/she can replay the whole game or parts of the game. Re-playability is to what extent (parts of) a game can be played repeatedly by players without the experience becoming repetitive. PCG-G can be used to improve re-playability of a game. An example would be to add unique content that is procedurally generated each time a player plays the game.

End Game Content

A player can spend more time playing the game if he/she can keep playing with a character after the game has finished. Content that is provided to a player after the end of a game is called end game content. PCG-G can be used to generate such end game content.

Infinite Worlds

A player can spend more time playing the game if the world in which the game takes place has an infinite size.

Because no computers exist that have infinite storage or infinite memory, it is impossible to generate a world of infinite size. A workaround to still be able to generate worlds that feel like they are infinite in size is to generate a small region of the world around the player. When a player moves towards any boundary of this region, new regions can be generated around the player. This way the player can never reach any boundary of the playable space and thus the world seems infinite in size.

1.1.3 Limited Disk Space

The amount of available disk space has increased massively over the last years, however historically this has not always been the case. PCG-G was a method used to deal with limited disk space that used to be available. Nowadays it can still be used to limit the use of disk space on mobile devices.

PCG-G can resolve this problem by procedurally generating the content in memory when the specific content is needed. The idea behind this is that algorithms to generate content requires less disk space than the resulting content itself. Instead of shipping a game with all its content, ship a game with only the algorithms to generate the content, in exchange for longer loading times.

1.1.4 Satisfying the desire to explore

Humans are motivated by exploration as discussed in [Mas43]. Since the beginning of mankind, humans have had the desire to explore. Over millennia, human exploration has led to discovering new lands, exploring the deep and vast oceans, analyzing the smallest bits that life is made of, and the exploration of space.

As an example: “Here be Dragons” - is a phrase historically used on world maps to indicate unexplored territory. This phrase hints at hoping to find new and interesting creatures when exploring new regions.

Exploring has become less accessible to most people nowadays. Most land on earth has been discovered, and other forms of exploration are not easily accessible for most people. Nowadays, to be able to satisfy the desire to explore, a different approach has to be taken. Games can be used to satisfy this desire of exploring unknown territories that haven not been seen by any other human, ever. PCG-G is the driving factor behind such games.

1.2 Examples of PCG-G

PCG-G can be used to generate game content such as levels, maps, history, quests, textures, characters, vegetation, and rules. A lot of existing games make use of PCG-G. Either in their development pipeline (offline) or during gameplay (real-time). Some great examples of PCG-G in games are: (see Figure 1.1)

- level layout generation in the Diablo series by Blizzard Entertainment,
- unique item creation in the Borderlands series by Gearbox Software,
- dungeon generation in UnExplored by Ludomotion,

FIGURE 1.1: Existing games that use procedural content generation for games. FLTR: Diablo, Borderlands, UnExplored, Minecraft and, .kkrieger

- infinite world generation in Mincecraft by Mojang,
- and texture generation in .kkrieger by Farbrausch.

1.3 Research Motivation

Nowadays, most content is generated by using pseudo-randomness, this results in content that looks interesting at first glance, but fails in accomplishing similar results to those achieved by expert artists. Generating content that resembles content created by expert artists has shown to be much more difficult.

Combining multiple PCG-G approaches to generate more complex content is not feasible, since most individual PCG-G approaches are not designed to work together. Moreover, most approaches cannot be constrained to guarantee certain output, which also makes them hard to combine. It is therefore still quite difficult to generate content such as worlds with complex, meaningful and interwoven structures, that are interesting to play.

1.4 World Generation

The scope of this master thesis is procedural content generation that is used to create game worlds. In most game worlds, tasks are used to complete certain goals. These tasks are what makes a game world interesting. A task that will result in the completion of a goal is also known as a mission. [Dor10] shows that it is a viable approach to generate game worlds by separating the generation of the mission that has to be completed by the player, from the space in which these missions take place.

The generation of a game world can thus be separated into two distinct parts: (1) the generation of missions, (2) the generation of the space. The research in this thesis focuses on generating an interesting mission for a game world. In most games

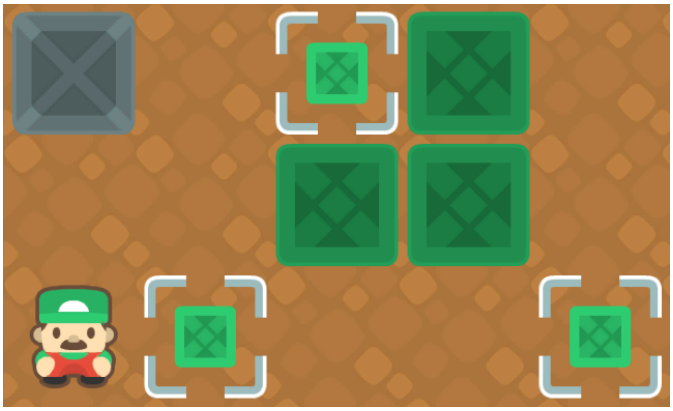


FIGURE 1.2: An example puzzle of the Sokoban puzzle game on a board of 5x3 tiles. The player starts in the lower left corner. Walls are gray, boxes are green, and goal location are green boxes surrounded by a white border.

the mission can be described as an abstract puzzle solving problem. Being able to generate interesting puzzles is being able to create interesting missions for game worlds. This makes the ability to generate interesting puzzles useful. Within this thesis the focus lies mainly on puzzle generation techniques.

1.4.1 Puzzles Generation

Generating interesting puzzles is a crucial component of generating interesting game worlds. Solving puzzles is a notable ingredient of gameplay in most games and most gameplay can be formulated as an abstract puzzle solving problem. A generic puzzle generation tool that can generate puzzles given the elements of a game world does not exist.

Many scientific puzzle generation and puzzle solving techniques use the game 'Sokoban' as a starting point for their research. Within this master thesis the puzzle game Sokoban is used as well.

1.4.2 Sokoban - A Puzzle Game

Sokoban is an easy to play and easy to understand puzzle game. Figure 1.2 shows an example puzzle of the Sokoban game. A metaphor for the game of Sokoban is that you control a robot in a warehouse that is supposed to re-organize boxes into a certain configuration.

Sokoban is played on a board of tiles, where each tile is either a floor or a wall. The player starts on one of the floor tiles and can move horizontally and vertically one tile at a time. The player cannot move to wall tiles and is thus restricted to move on the floor tiles.

Some of the floor tiles contain boxes. These boxes can be pushed. A player can push a box by standing next to it and moving onto the floor tile that has a box on it. The box is then pushed to the next tile. Pushing a box is only possible when the tile the box is pushed onto is also a floor. The floor may not have a box on it already, because the player is only allowed to push one box at a time. Figure 1.3 shows an overview of valid and invalid pushes in Sokoban for a player moving to the right.

Some of the floor tiles of the board contain goal locations. A puzzle is solved when each goal tile is covered by a box. The number of goal tiles is always equal to

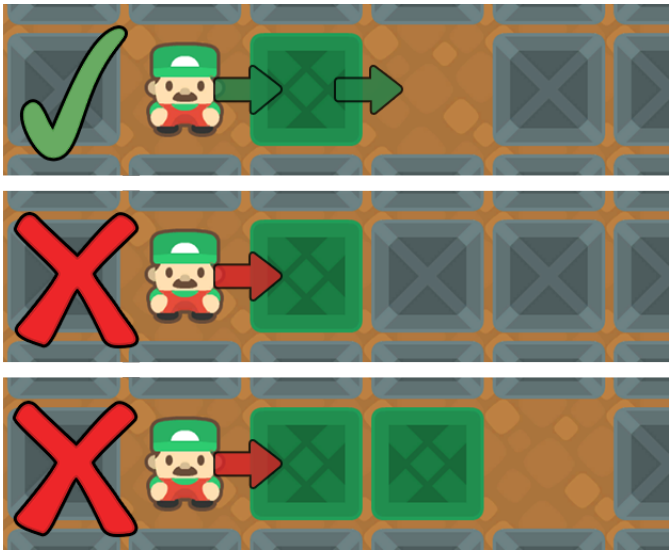


FIGURE 1.3: Overview of valid and invalid pushes in Sokoban when the player makes a move to the right.

the number of boxes. Any goal location may be covered by any box, they are not labeled, however in handcrafted puzzles each box generally has only one feasible goal tile.

The game is considered to be pure and simple, very playable and mentally challenging according to Computer Gaming World magazine. What makes Sokoban tricky is that it is easy to get stuck. Getting stuck in Sokoban means that from the current configuration of the boxes a solution to the puzzle is no longer possible. The puzzles must then be reset to its initial configuration of boxes, so the player can start over. Situations in which the player gets stuck can easily arise. An example of this is that a box is pushed into a corner.

After solving a puzzle, some players will try to solve the puzzle again while trying to minimize the number of moves used.

1.4.3 Reasons for using Sokoban

Many puzzle solving and puzzle generation techniques use the Sokoban puzzle game as a test case for their research. Sokoban suits puzzle solving and generation techniques well, because of the following reasons:

- **Simple to Understand** - The puzzle game is simple to understand,
- **Available Data Sets** - There are datasets of solvable puzzles of varying difficulties available,
- **Large Search Space** - There is a large search space of possible puzzle configurations,
- **Hard to Solve** - Testing whether possible puzzle configuration can be solved is impractical. It has been proven that solving Sokoban puzzles is PSPACE-complete by [Col97].
- **General Applicability** - In [Pos16] puzzle variations of deterministic transportation games are considered. Sokoban falls under this category of deterministic transportation games. This work shows that the approaches used to

generating deterministic transportation puzzles are applicable to other puzzle games too.

1.5 Research Question

Existing work has shown that solvable Sokoban puzzles can be generated using simulated play. This simulated play is called simulation. Although the resulting puzzles are solvable, they are generally less interesting to play in comparison to handcrafted Sokoban puzzles. Other work in puzzle generation shows techniques that can generate puzzles that are interesting to play. By improving the existing work in simulated play it might also be feasible to generate puzzles that are interesting to play using simulated play.

The research question of the master thesis is:
Can simulation be used to generate Sokoban levels that are interesting to play?

1.6 Research Plan

To be able to answer the research question, 'Can simulation be used to generate Sokoban levels that are interesting to play?', the research has been subdivided into multiple smaller steps. The different chapters in this master thesis correspond to these steps.

(a) An overview of **related work** in the field of PCG-G is made in Chapter 2.

(b) The ***foundational* work** that is able to generate Sokoban puzzles using a Monte Carlo Tree Search approach is discussed in Chapter 3. This *foundational* work is the basis of the rest of this thesis.

(c) Seven candidate **improvements** on this *foundational* work are formulated in Chapter 4. These improvements fall into two categories: Alteration Extending and Symmetry Reduction.

(d) An **experiment** is conducted on the work done in this master thesis and the results of this are presented and compared to the *foundational* work in Chapter 5.

(e) The thesis is wrapped up by answering the research question and giving a **conclusion** in Chapter 6. Some discussion, current limitations and future work are also included in this chapter.

2 Related Work

In this chapter existing work within PCG-G is discussed. First an overview of different classes of methods in PCG-G is given. These different classes of methods are then explained and are related to puzzle generation.

2.1 Classes of Methods in PCG-G

Before diving into techniques that specifically aim to generate puzzles, a broad overview of PCG-G approaches is given. This section gives a summary of research that has been done in the field of Procedural Content Generation for Games. [HMV13] gives an comprehensive survey of the field of PCG-G. This survey shows six distinguishable classes of methods for PCG-G.

1. **Pseudo-Random Number Generators (PRNG)** - Used to mimic randomness found in nature,
2. **Generative Grammars (GG)** - Used to create correct objects from elements encoded as letters/words,
3. **Image Filtering (IF)** - Used to emphasize certain characteristics of an image to display (partially) hidden information,
4. **Spatial Algorithms (SA)** - Used to manipulate space to generate game content,
5. **Modeling and Simulation of Complex Systems (CS)** - Used to model a simulation to overcome impracticability to describe natural phenomena with mathematical equations,
6. **Artificial Intelligence (AI)** - Used to mimic animal or human intelligence.

The following sections will describe the classes of methods and will discuss some related work within these classes of methods.

2.1.1 Pseudo-Random Number Generators (PRNG)

Pseudo-Random Number Generators are used to mimic randomness found in nature. PRNG generated pseudo-random sequences of numbers, commonly the generated numbers can be reproduced by using the same starting number, which is the seed. These random sequences of numbers are random noise in which the numbers have no relationship to each other. Other forms of random noise in which the numbers do have relationships to each other also exist.

PRNG is not sophisticated enough to generate puzzles all by itself, since randomness rarely succeeds in generating interesting structures. However, PRNG can aid other puzzle generation techniques in generating variation.

Perlin Noise and Simplex Noise are two examples of PRNG and are described below.

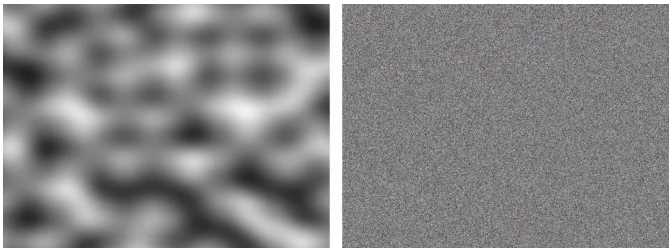


FIGURE 2.1: A comparison of smooth random 2D-noise (on the left) to random 2D-noise (on the right).

Perlin Noise

[Per85] formulates Perlin Noise. A noise algorithm in which the numbers do have a relationship to each other. Perlin Noise is a very well known algorithm within PRNG and is implemented in most game engines. The relationship between the numbers in Perlin Noise is a smooth transition between the values. This makes it extremely suitable for things such as terrain generation, because the terrain height is also some form of a smooth transition. Figure 2.1 shows an example of smooth noise in comparison to random noise. Perlin Noise is mainly used in its two-dimensional form, so two dimensions are used in this explanation for simplicity, but note that Perlin Noise works for any number of dimensions.

Given a 2D-coordinate a value will be calculated within the range of -1 to 1. Using the same coordinate will always give the same result. Perlin Noise generates the value in the following steps steps.

1. **Find rectangle,** - The space is subdivided into a grid of rectangle regions. The rectangle in which the coordinate lies is found. A grid is most commonly made of squares.

2. **Calculate corner gradients,** - For all four corners of the rectangle a random two dimensional gradient is generated. The gradients all have the same magnitude. In most cases an index table of gradients is used. The index within this table of the corner is the same each time the same corner coordinate is provided.

3. **Calculate distance vectors,** - For each corner a distance vector from the corner of the rectangle to the coordinate is calculated.

4. **Calculate dot products,** - For each corner a dot product between the distance vector and the gradient is calculated. The result of each dot product is a number at each corner of the rectangle.

5. **Combine dot products.** - The results of the dot product at each corner have to be linearly interpolated. First the two top corners are interpolated on the x position of the coordinate. Secondly the two bottom corners are interpolated on the x position of the coordinate. We now have a value at the top position of the rectangle and the bottom position of the rectangle at the given x. Finally, these are then interpolated using the y value of the coordinate. This is the result of the Perlin Noise function. Instead of using the actual x and y position within the rectangle a function is first applied to x and y, this is the fade-function. Normally the fade-function is in the form of $f(t) = 6t^5 - 15t^4 - 10t^3$.

Simplex Noise

Another form of smooth noise that was designed by Perlin is Simplex Noise. Simplex Noise is similar to Perlin Noise, but differs from Perlin Noise in some parts. The most important differences are that Simplex Noise uses equilateral triangles, instead of rectangles. This will result in 60 degree artifacts instead of 90 degree artifacts, which are harder to spot by the human eye. Also Simplex Noise sums contributions from each corner and is slightly faster.

2.1.2 Generative Grammars (GG)

Generative grammar is used to create correct objects from elements encoded as letters/words. Generative Grammars are sets of rules that, operating on individual words, can generate only grammatically-correct sentences.

Within the scope of PCG-G more abstract forms of generative grammar are used. Instead of operation on letters/words, the grammars operate on structures such as graphs and shapes. These types of generative grammar are respectively named Graph Grammar and Shape Grammar. These grammars are a sets of rules that change a part of a graph or shape. New possibly interesting, graphs or shapes can be generated by iterativel applying rules to the graph or shape.

GG is well suited to generate puzzles when the puzzles to be generated are made out of fixed elements. In [Dor10], graph grammar is used to generate missions and shape grammars is used to generate space.

Cyclic Graph Grammar

Recent work in Graph Grammar has resulted in a special case of Graph Grammar, called Cyclic Graph Grammar. It can be used to generate missions. Cyclic graph grammar is used to perform cyclic dungeon generation for *UnExplored* in [TWR17].

[Dor17] introduced these cyclic grammars. Traditional approaches of graph grammar have rules that add branches to the graph, the base of these graph grammars are connections between the nodes. Using such graph grammar will result in a graph that grows from a starting point, much like a tree. The branches of these trees end in nodes that are dead ends. Traditional approaches fix these dead ends by trying to connect the nodes of dead ends. Cyclic grammar overcomes dead ends entirely by using cycles as the base of the graph grammar. These cycles are used to create flows that feels more handcrafted and to present the player with alternative solutions.

2.1.3 Image Filtering (IF)

Image Filtering is used to emphasize certain characteristics of an image to display (partially) hidden information. Many techniques are yearly developed for image filtering. Two fundamental image processing techniques are binary morphology and convolution filters.

IF is not most useful when generating puzzles, because it is mainly used to generated smoothness or roughness in terrains or textures. Binary Morphology and Convolution Filter are two IF approaches and are described below.

Binary Morphology

In binary morphology, pixels of a binary images are processed. An image can be transformed into a a binary images by changing each pixel into either 0 or 1 based on some threshold. Examples of techniques are erosion or dilation, which can respectively

add or remove edges from the binary image. These binary morphology techniques can be used to gather certain characteristics of a texture to create new textures or to generate binary fields.

Convolution Filters

Convolution filters are image filters that change a source image into a filter images, by applying a filter to each pixel that takes surrounding pixels into account. This filter could for example be used to blur or sharpen edges of images. Within procedural content generation it can be used to generate variants or whole new textures from existing textures.

2.1.4 Spatial Algorithms (SA)

Spatial Algorithms are used to manipulate space to generate game content. Output is generated by using a structured input, for example a grid, or self-recurrence. Tiling and layering, grid subdivision, fractals, and Voronoi diagrams are all spatial algorithms that can be used in PCG.

SA can be very useful when generating puzzles. Most of the time puzzles can be described as an abstract problem in some possibility space. For example a graph that shows all possible routes through a dungeon. Being able to generate such spatial structures can aid puzzle generation. Below Tile-based Methods for PCG are described and we discuss how SA can be used to generate Infinite Worlds.

Tile-based Methods for PCG

[Mau16] shows that there are many tile-based approaches within PCG. Tile-based approaches are applicable for many different types of content creation, including levels. Tile-based approaches conveniently subdivide the content creation process into smaller parts, making it easier to define local features. Another advantage of tile-based generation is that the tiles themselves can be generated offline, while the tile configuration can be generated in real-time. This makes sure lengthy content generation methods do not have to be executed during run-time.

Infinite Worlds

An infinite world is something that can result from PCG-G methods by generating content using Spatial Algorithms. It is used to create the feeling for the player of a world that is infinite in size.

Because it is impossible to generate a world of infinite size, only a portion of the world can be generated, ideally only the part of the world that can be seen by the viewer is generated. This part is the local area. The local area moves along with the viewer. To generate the local area, information about the surroundings might be necessary. For example: Generating a town could need information of neighboring towns to determine the road-layout to those towns. The challenge in this problem is to generate all surrounding content that is strictly necessary to generate the local area.

Layer-Based Regionalized Infinite World Generation To tackle this problem [Joh13] presents layered-based approach to infinite world generation. The world space is subdivided into regions, in which each region can have up to M layers. For the local

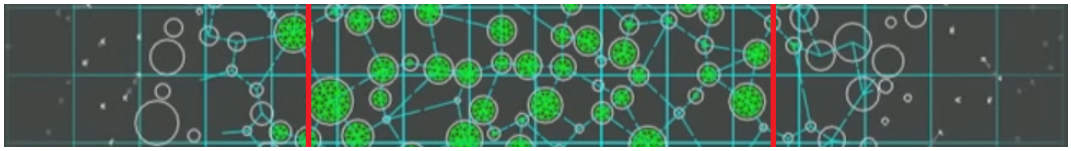


FIGURE 2.2: An abstract render of an infinite world in which the circles are towns and the connections between those circles are roads.

area all information is needed, which means that the local area consists of regions that all have M layers.

A layer adds more information to the region based on the information in the surrounding layers below it. The information of a layer must be in a layer as low as possible. In the 1st layer (base layer) all information that is independent of the surroundings or other layers is generated. All information in the 2nd layer must be based on information from the surrounding base layers, because if it was not based on information from the base layer it could just as well have been generated in the base layer itself. The general rule is: The *nth* layer of a region uses some of the information from the *(n-1)th* layer of the surrounding regions. The shape and/or size of the regions can differ for the different layers.

In Figure 2.2 an example is shown. This is an abstract render of a infinite world in which the circles are towns and the connections between those circles are roads. In this example the world is subdivided into square grid regions. The local area is the area between the red lines. This example world uses five layers. For the local area all available layers must be generated. All regions within the local area must be surrounded by regions with 4 layers. This will inevitably include some regions outside of the local area, which will result in a band of regions surrounding the center regions. This band is then also surrounded by another band of regions with 3 layers. This process is repeated until the base layer.

Non-Popping LOD-Chunks Terrain Visualizer Another approach to deal with infinite worlds is to use Spatial Algorithms to generate high detail content close to the viewer, and content with decreasing detail further from the view of the player.

[Ulr02] describes a method that visualizes existing, possibly infinite, terrain data. Within the context of this approach a terrain is the combination of a heightmap (xy-plane to z-height) and the textures that map onto this heightmap. Visualizing small heightmaps is not a problem, but it becomes problematic when the terrain you are trying to visualize is massive. In this context massive means, it contains way more data than a computer is able to visualize at once. This paper describes a way to render such a massive terrain using chunks.

A chunk is a subsection of the xy-plane. Each chunk contains local preprocessed mesh and texture data up to some Level of Detail (LOD) to represent that the terrain on its own subsection of the xy-plane. This LOD mesh has a certain maximum offset error. The chunks can contain other chunks, up to some arbitrary depth. This forms a chunk-tree. The further down the chunk-tree, the smaller the size of the xy-plane of the chunk, the higher the precision and, the lower the maximum offset error from the terrain data.

Rendering is based on the viewer. Chunks closer to the viewer need higher precision when rendered, than chunks far away. During rendering each chunk calculates the maximum allowed offset error based on the distance to the viewer and the field of view of the viewer. The chunk-tree is then traversed to select the correct depth in the

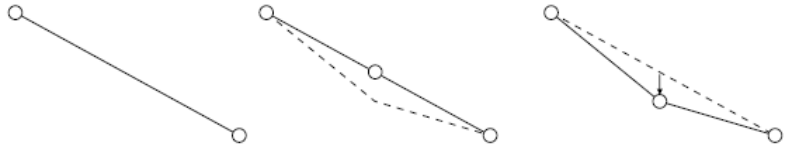


FIGURE 2.3: The initial line (left) first pops to a higher detail line (middle) that has the same shape. As the player comes closer, the higher detail line (middle) will slowly morph into the final line (right).

chunk-tree based on the maximum offset error of each chunk. This process results in chunks with higher precision closer to the viewer and chunks that have lower precision farther away.

This however gives two major problems. (1) Neighboring chunks with different LOD meshes have cracks where they meet and (2) when the viewpoint approaches a chunk, the chunk will split into child chunks, and the mesh shape will suddenly pop.

Cracks can be fixed by adding a vertical border around all chunks that have the texture from the chunk itself applied to them.

The vertex popping can be fixed by slowly morphing to the next LOD mesh based on the distance from the viewer to the previous and next LOD swap distances as can be seen in Figure 2.3.

2.1.5 Modeling and Simulation of Complex Systems (CS)

Modeling and Simulation of Complex Systems is used to model a simulation to overcome impracticability to describe natural phenomena with mathematical equations. Well known methods are cellular automata, flocking, tensor fields, and agent-based simulation. Many other methods of complex systems are part of procedural content generation. For example acting (for quest generation) and semantic models (for entity behavior).

CS can be useful in puzzle generation. Most puzzles can be modeled as a complex system. This complex system can be build based on the key-elements of a certain type of puzzle. In some cases this complex system can then be used to generate puzzles of this type. Two examples, a semantics-based PCG approach and a flocking simulated height brushing approach, are discussed below.

Semantics-based PCG

The PhD thesis [Tut12] uses semantics to procedurally generate content. Semantics can be used to model a complex system, which can be used to generate content in a meaningful and logical manner. The framework described uses a semantic database of information about the objects that should be place in a world. The generated worlds are meaningful and can be interacted with by using the semantic data that is now present for the objects in the scene.

The procedure of generating content using semantics is described in a Semantic Layout Solver. The semantic layout solver uses properties of the objects that are placed to create the scene. Objects have preferred positions relative to other objects, take up space that can not be occupied by other objects and can demand other objects to be placed in the scene.

The procedure to build a scene is to place each object that has to be placed into that scene sequentially. Each object is placed on a valid location. Whether its placement is valid is based on its clearance-regions, off-limits regions and geometric rules. Clearance regions should be empty but cannot overlap, off-limits regions must be empty and cannot overlap with clearance-regions and geometric rules specify objects that should be close to or far away from each other. Other rules such as how many chairs should be placed around a table, or how much weight a table can hold can also be incorporated into the metrics which are used by the layout solver.

Flocking Simulated Height Brushing

[CGG07] uses simulation to create the surface of a spherical world and texture the resulting geometry.

To generate a spherical world, the approach starts with a perfect sphere. This sphere is represented as a parametrized heightmap along its surface. Brushes are defined that can conditionally raise or lower parts of this spherical heightmap. Artists can create new brushes, these bushes can represent planet like features such as mountains, canyons and oceans. Combinations of these brushes can be swarmed (which is the flocking-like tool the developers created) over this planet to deform the sphere. This results in a height map.

To texture the planet a color and a detail texture are generated and combined using a control texture. A shader then combines those textures and changes the color based on the curvature of the terrain, the height of the water level and the atmospheric settings (such as the temperature) of the planet. Combinations of brushes, and all texture and atmospheric settings can be considered as a planet type.

2.1.6 Artificial Intelligence (AI)

Artificial Intelligence is used to mimic animal or human intelligence. Examples include speech recognition, planning, and execution of physical tasks by robots. Within PCG, AI is used in Genetic Algorithms, Neural Networks and Constraint Satisfaction and Planning.

AI can be used in puzzle generation to (1) try to model the task that a puzzle designer would perform by hand, (2) to search in a smart manner through many different puzzles to determine puzzles that fulfill certain requirements, and (3) it could also be used to simulate gameplay to generate puzzles. A Monte Carlo Tree Search (MCTS) approach that generates Sokoban puzzles is described below. Important to note is that although AI looks promising, using generic AI to generate puzzles is still in its infancy.

Monte Carlo Tree Search Puzzle Generation

[KSG16b] tries to generate Sokoban Puzzles by formulating the generation process of Sokoban puzzles as a search (and optimization) problem in a tree. The search technique used is Monte Carlo Tree Search (MCTS). MCTS is a search method that combines the precision of tree search with the generality of random sampling. It has shown great success in the world of Go, and this paper shows that it can also be used to generate Sokoban puzzles.

An MCTS algorithm iteratively expands the whole search tree. During each iteration a decision is made where to expand the search tree and a random simulation is made to a leaf node of the tree to check whether the result looks promising. The expansion process is made so a balance is found between exploitation and exploration.

The results of a simulation are fed back into the tree and are used to determine which areas of the tree are most interesting.

By describing the Sokoban generation process as a stepwise process, the generation can be seen as a tree search problem for which the MCTS can be used. The steps are chosen such that they simulate gameplay and therefore ensure that all generated puzzles are valid, because the generation process adheres to the rules of the game. Heuristics are used to update the tree after each simulation. The paper shows that, if these heuristics make more interesting puzzles score higher, the search tree slowly converges to finding more interesting puzzles.

2.2 Applicability in Puzzle Generation

As can be concluded from this chapter, puzzles can be generated in many different ways. Pseudo-Random Number Generators (PRNG) and Image Filtering (IF) have shown to be useful in other area's of PCG-G, but are not so suitable for puzzle generation. Generative Grammars (GG), Spatial Algorithms (SA), Modeling and Simulation of Complex Systems (CS) and Artificial Intelligence (AI) do show promising application within puzzle generation.

3 Foundational Work

This chapter discusses the *foundational* work done in search-based Sokoban puzzle generation. [KSG16b; KSG16a] present a way of generating puzzles of varying difficulty for Sokoban. [KSG16b] is the initial work of the method and [KSG16a] proposes an improvement of this method using a data driven approach. The complementary work of these two papers will be referred to within this master thesis as the *foundational* work, because it lies the foundation and basis for the work done in this thesis.

The *foundational* work shows very promising results. The method that is described heavily relies on a Monte Carlo Tree Search (MCTS) which is used frequently in AI research. At the time of writing it is the only puzzle generation technique that was found that uses an MCTS to generate puzzles. Given the promising results, the brand-new topic, and the state-of-the-art techniques: this paper is the foundation of the research in this master thesis.

A good understanding of the foundation is necessary, therefore it is crucial to have a detailed overview of the *foundational* work. Although a rough summary was already presented in Section 2.1.6, a more in-depth summary is given in this chapter. First, an introduction of the work is given, secondly it is shown how the generation of a Sokoban puzzle can be mapped to a tree, thirdly a estimated difficulty function is given, fourthly the use of the MCTS to traverse this tree is explained and lastly the conclusion of the work is discussed.

3.1 Introduction

The *foundational* work proposes a method to generate Sokoban puzzles using simulated play. It does this by mapping the generation process of all Sokoban puzzles (of a fixed board-size) to a simple hierarchical tree structure. Each branch from the root to a leaf node in this tree represents the construction of a single Sokoban puzzle. This tree that has branches for all possible Sokoban puzzles is, consequently, too large to explore completely. However, clever techniques that have been developed to traverse large trees, for example for game-trees used in Chess or Go playing AIs, can be used.

The clever technique that is used to traverse the tree is the Monte Carlo Tree Search (MCTS). This technique expands the tree one node at a time. The node that is expanded is selected based on an exploitation vs exploration dilemma. The *foundational* work shows that by traversing the search space using an MCTS, somewhat challenging puzzles can be found within a fixed amount of exploration time.

The MCTS uses random roll-outs to leaf nodes of the tree to determine the estimated difficulty of puzzles in certain regions of the search space. A roll-out is a random sequence of alterations that results in a leaf node. A branch from the root to any leaf node represent a Sokoban puzzle. Each leaf node has only one branch back to the root, therefore each leaf node corresponds to a Sokoban puzzle. Each time a leaf node is traversed, an estimated difficulty is computed for its corresponding puzzle. The puzzle that resulted in the highest estimated difficulty is kept track of. At any time during the traversal of the tree, the process can be halted after which the puzzle

that scored the highest on the estimated difficulty function is returned. Increasing the exploration time will increase the likelihood of finding more difficult puzzles.

3.2 Sokoban as a Tree

As explained in the introduction, the *foundational* work is based on the ability to map the generation of all Sokoban puzzles as a tree. In this section this mapping is explained.

The tree that is used to described the generation process of all Sokoban puzzles is an hierarchical non-cyclic tree. A branch from the root node to any leaf node represents the generation process of a Sokoban puzzle. The nodes do not represent Sokoban puzzles themselves, the nodes represent a step in the construction process of a Sokoban puzzle. A step in the construction process in the context of this tree is called an **alterations**.

The **root node** is the entry point of the tree. The root node is the node that provides the initial state of a Sokoban puzzle. The initial state is a puzzle in which all tiles, except the center tile, are obstacles. At the center tile the player starts. This initial state is visualized at the left-top in Figure 3.1. A **leaf node** is an exit point of the tree. A leaf node is a node that describes a construction state which has no potential alterations available. A **branch** from the root node to a leaf node describes the sequence of alterations that is applied to the initial state that results in the generation of a Sokoban puzzle. Because the tree is hierarchical and non-cyclic, each leaf node has exactly one path to the root of the tree. Each leaf node therefore corresponds to a single Sokoban puzzle.

To ensure that all the generated puzzles adhere to the rules of the game, the tree structure is build up out of alterations that adhere to the rules of the game. Due to this, all branches from root to a leaf node are puzzles that adhere to the rules of the game. Consequently, puzzles that do not adhere to the rules of the game, such as puzzles that are not solvable, cannot be generated using these alterations.

Each alteration has to define two important procedures. The first is **finding** the available alterations given the current state of construction. The second is **applying** an alteration to a state of construction. These find and apply procedures of the alterations should be executable within a very short execution time. The faster these procedures are executed, the less time is spent on a single branch, which means more time is available for visiting new branches.

The following five types of alterations are used.

1. **Delete Obstacle Alteration** - If the level has not yet been frozen, each obstacle adjacent to an empty square can be removed.

2. **Place Box Alteration** - If the level has not yet been frozen, on each empty square a box can be placed.

3. **Freeze Level Alteration** - If the level has not yet been frozen, the level can be frozen.

4. **Move Player Alteration** - If the level has been frozen, the player can move (up, down, left or right) based on the rules of the game.

5. **Evaluation Alteration** - If the level has been frozen, some post-processing is applied to the puzzle, resulting in a leaf node.

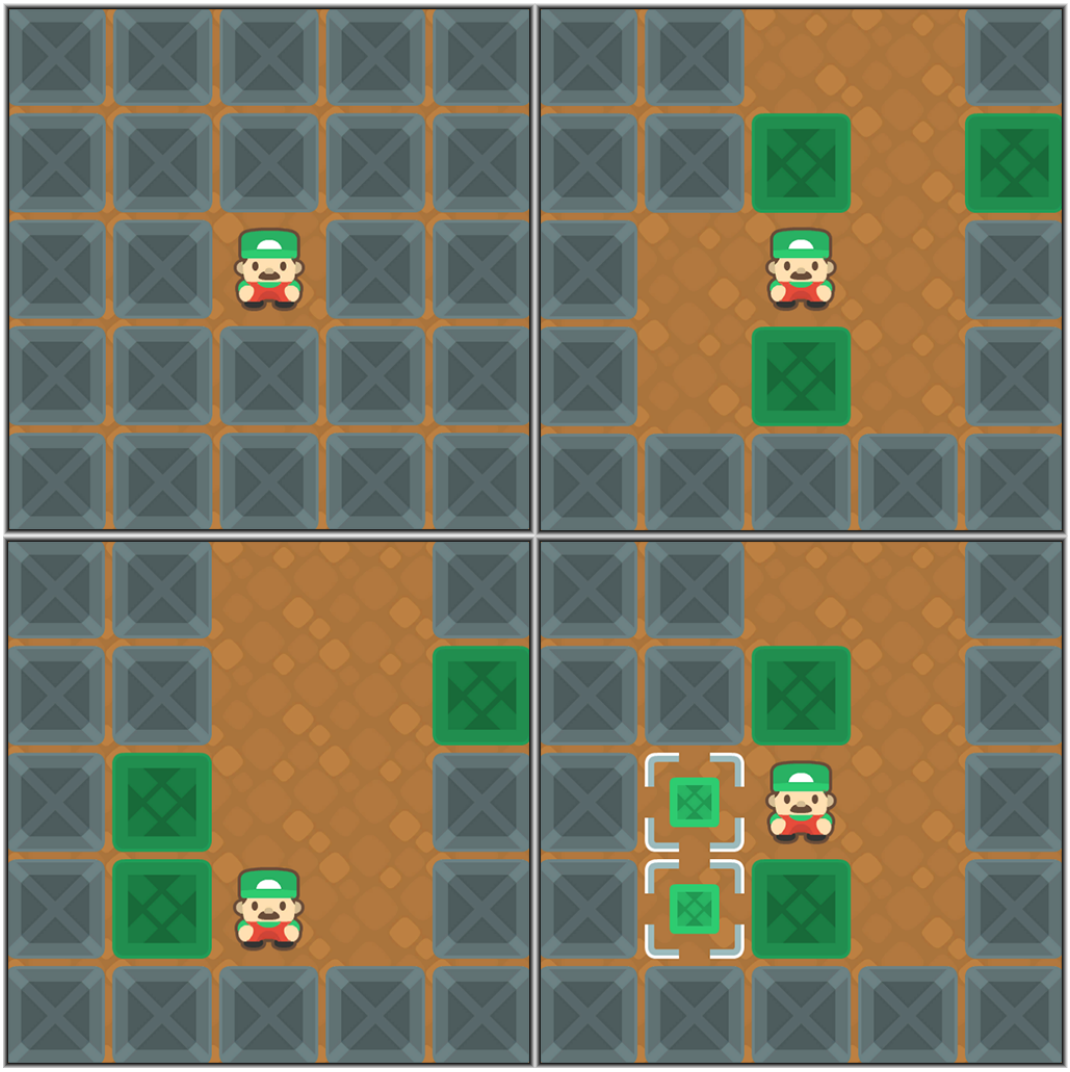


FIGURE 3.1: An example of the states of a puzzle. *left-top*: initial state, *right-top* state at freeze, *left-bottom*: state before evaluation and *right-bottom* state after evaluation.

These alteration types divide the tree into two phases. The first phase is from root until the freeze, the second phase is from the freeze until the end. During the first phase the starting configuration is created. This starting configuration contains the positions of the player, boxes and obstacles. During the second phase, play is simulated on the puzzle defined during the first phase. This simulated play only consists of alterations that are legal moves within the game. The second phase always ends with the evaluation alteration.

The evaluation alteration is the most complicated alteration. It only happens once each branch and it is always the alteration that results in a leaf node of the branch. The evaluation alteration applies some post-processing. The player and all boxes are moved back to their starting positions. The starting position of the player and boxes are their positions at the moment the freeze alteration occurred. The positions where the boxes ended up are their goal locations, at these position a goal is created. An example of this can be seen in Figure-3.1. In this figure you can also find a box that has not been pushed by the player and is turned into an obstacle. Boxes that have not been pushed by the player do not add any value to the puzzle, these boxes could have just as well been obstacles. These boxes are therefore changed into an obstacle without invalidating the puzzle.

3.3 Estimated Difficulty Function

As explained in the introduction, the *foundational* work uses a function to estimate the difficulty of solving a certain puzzle. The estimated difficulty function maps a Sokoban puzzle to an estimated difficulty, which is a number in the range from zero to two.

In [KSG16b] this function is based on two metrics, a terrain-metric and a congestion-metric. The terrain-metric will return higher scores for puzzles with not too many open obstacle-free spaces, and the congestion-metric will return higher scores for puzzles in which box-path are congested with other boxes, obstacles and goals. The results of these metrics are combined using the geometric mean. Therefore, when one of the metrics approaches zero, the result approaches zero too, meaning that both metrics have to be improved to result in good puzzle scores.

In [KSG16a], the follow up work of the main approach, a data-driven approach was used to reformulate these metrics. For over 200 hand-crafted puzzles a difficulty was estimated by users playing the puzzles. Metrics were formulated that showed correlation between the user-estimated and computer-estimated difficulty of the hand-crafted puzzles. Three promising metrics arose: a new congestion-metric, a non3X3UniformBlock-metric and a boxCount-metric. The new congestion-metric is almost the same as the original one, except it takes the size of the area of the box-path into account. The non3X3UniformBlock-metric counts each tile that is not part of either a 3x3 region of obstacles or a 3x3 region of non-obstacles. And the boxCount-metric simply counts the number of boxes. The results of these metrics are scaled using individual weights and then combined using the arithmetic mean, after which the result is divided by a normalizer to get the results within the range of zero to two.

3.4 Traversing the Tree

To traverse the search space a Monte Carlo Tree Search (MCTS) is used. An MCTS expands the traversed space one node at a time. Each expansion is a single iteration

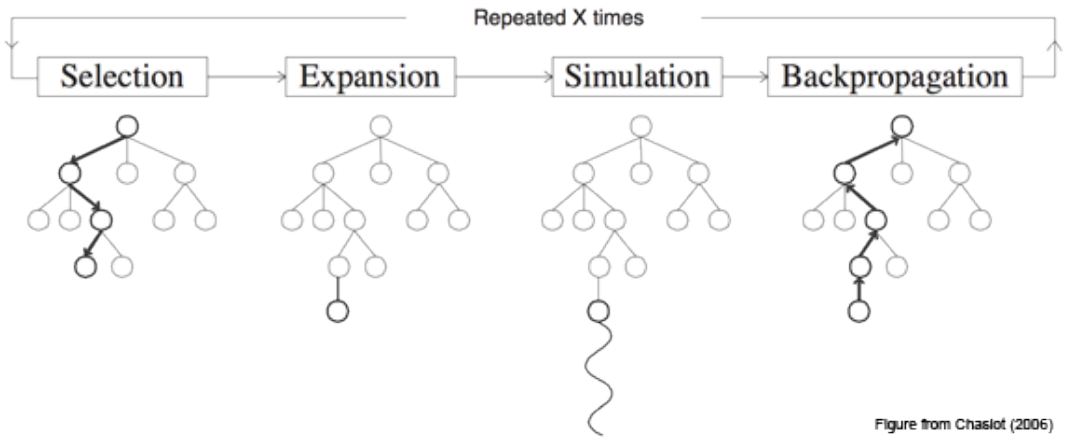


FIGURE 3.2: Visual overview of the different phases in the Monte Carlo Tree Search algorithm.

of the MCTS. Each iteration executes four steps: selection, expansion, simulation and backpropagation. An MCTS is a tree that slowly expands towards higher scoring nodes as more iterations are performed. Figure 3.2 gives a visual overview of the different phases of the MCTS.

The selection-phase traverses the tree until a node is found that does not have all its children expanded. In the expansion-phase one of the non-expanded child nodes is expanded. In the simulation-phase a random roll-out is run from the newly expanded node, until a leaf node is reached. The leaf node corresponds to a puzzle for which the estimated difficulty is computed. The nodes created for the roll-out are then discarded. All nodes on the branch from the expanded node to the root node are updated with the estimated difficulty in the backpropagation-phase.

During the selection-phase the results of the back-propagation-phase are used to chose how to traverse the tree. Choosing the child node to traverse to, is an exploitation vs exploration dilemma. On the one hand picking child nodes that have promising results seems logical (exploitation) and on the other hand exploring new nodes that have not shown promising results yet is also necessary (exploration). The UCB formula provides a good balance between these two.

$$e_i + C\sqrt{\frac{\ln v_p}{v_i}}$$

in which e_i is the estimated value of the child node between 0 and 1, C is the tunable bias parameter, v_p is the visit count of the parent and v_i is the visit count of the child node.

3.5 Conclusion

The *foundational* work shows that it can generate somewhat challenging puzzles within minutes for varying board sizes given no human designed input and that it is guaranteed to produce solvable puzzles. Figure 3.3 shows four example puzzles generated by the *foundational* work. The authors note that the box path interactions in their puzzles usually correspond to determining a small number key moves, after which the puzzle is easily solved. In contrast, human designed puzzles can require

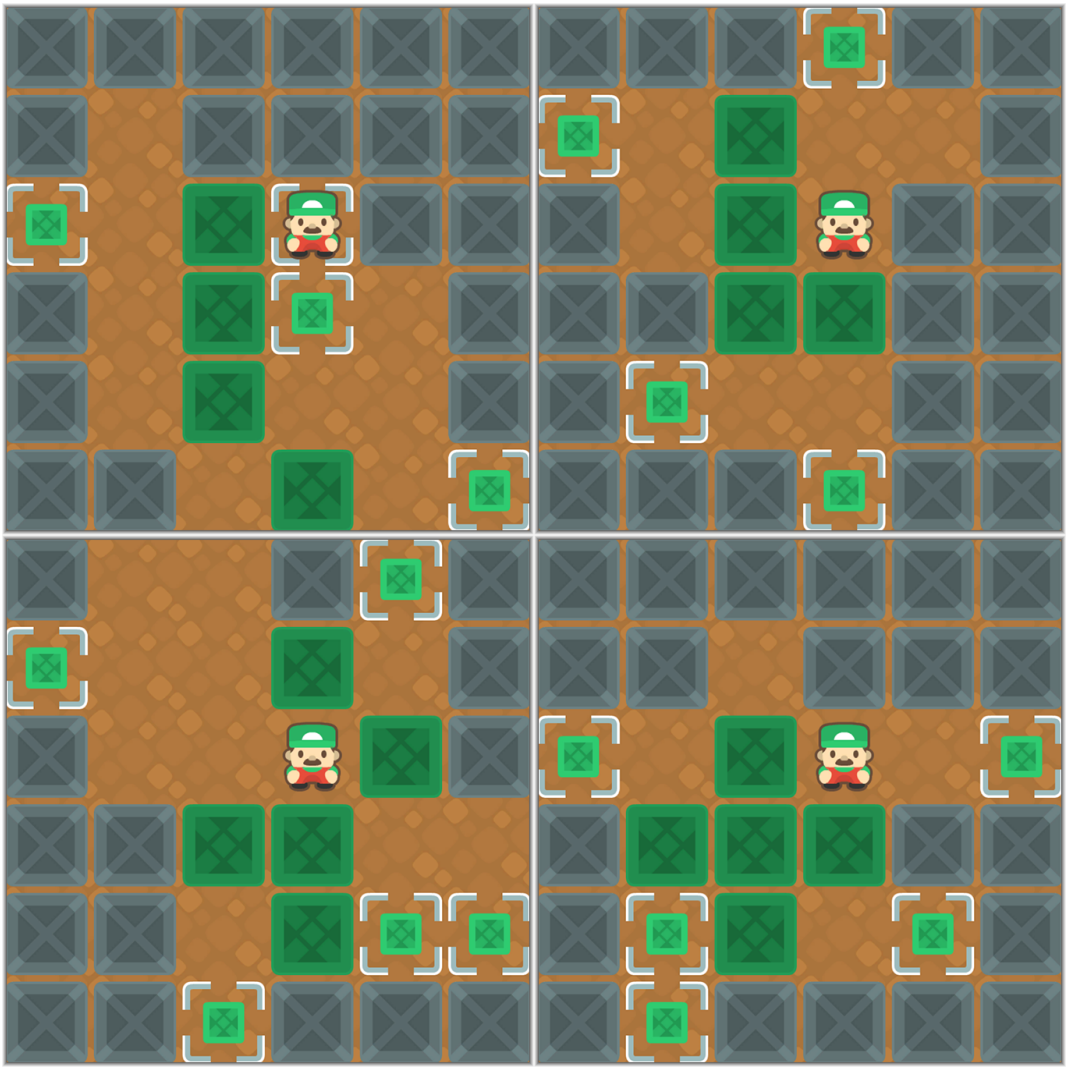


FIGURE 3.3: Puzzle examples generated by the foundational work requiring at least *left-top*: 15, *right-top* 15, *left-bottom*: 18 and, *right-bottom* 15 moves to solve on boards with a size of 6x6.

the player to move boxes carefully together throughout the entire solution. There is thus room for improvement.

4 Candidate Improvements

In this chapter an overview of candidate improvements on the *foundational* work is given. The candidate improvements that were implemented all fall into one of two categories: Alteration Extending or Symmetry Reduction. Candidate improvements in Alteration Extending try to improve the *foundational* work by altering or extending the alterations that are used to build up the search space. Candidate improvements in Symmetry Reduction try to improve the *foundational* work by removing the symmetry that is present within the search space.

The following sections describe the candidate improvements. For each candidate improvement the motivation, concept, implementation and results are discussed. The setup that was used to gather the results can be found in Section 5.3.

4.1 Alteration Extending

In this section candidate improvements that fall into the category Alteration Extending are described. These candidate improvements try to improve the *foundational* work by altering or extending the alterations that are used to build up the search space. The alterations that are used in the *foundational* work are described in Section 3.2.

4.1.1 Push Alteration

Motivation

In the *foundational* work a move alteration is used. This move alteration simulates input for left, right, up, and down moves. The idea is that this move alteration will sometimes push a box when moving the player. These box pushes are used by the evaluation alteration to finalize the puzzle. The evaluation alteration is always the last iteration that is applied to the board. This alteration moves all boxes and the player back to their starting positions and adds goals at the positions where the boxes ended. The end position of the player does not affect the outcome of this evaluation alteration. Therefore, moves that do not push a box, do not alter the outcome of the puzzle and are uninteresting.

The move alteration is essentially a way to search for box pushes. By using an alteration to perform this search the MCTS is executing this search algorithm internally. Although this seems to work in the *foundational* work, there might be a better way to execute this search. An external search algorithm can be employed to search for available box pushes instead of using the MCTS for this.

Concept

The push alteration is the substitute for the move alteration. The traditional moves used by the move alteration are replaced with sequences of moves that end with a box push. Such a sequence of moves is a path. The push alteration searches for all paths that end with a box push from the current position of the player. It does this

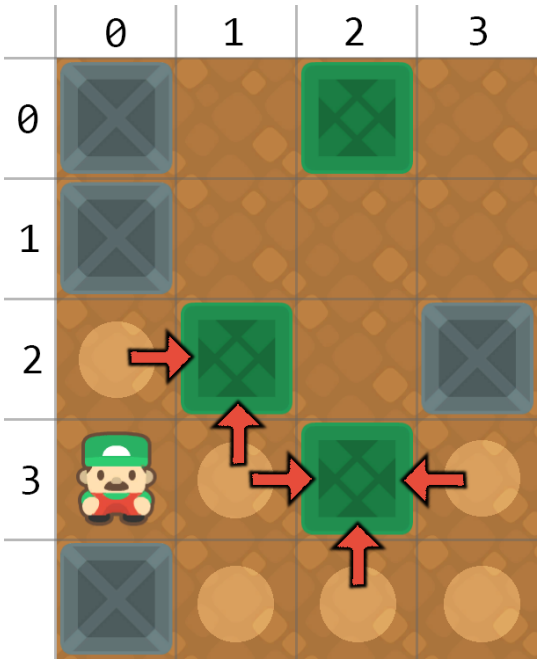


FIGURE 4.1: An example of all reachable tiles (white circle) and all available box pushes (red arrow) from those reachable tiles.

by first finding all the floor tiles that the player can reach and then looking for all the box pushes that can be made from these reachable floor tiles. Figure 4.1 shows an example of this.

In the *foundational* work the move alteration marks tiles that are moved to as used. During the evaluation alteration tiles that have not been used are turned into an obstacle. The push alteration should make sure that tiles are marked as used when a path crosses them. Moreover it is preferable that the paths that are found try to avoid tiles that are not marked as used unless unavoidable. When it is unavoidable for a path to cross a tile that is not marked as used there might be multiple solutions to achieve this. In the implementation, the first solution that is found is picked and all other solutions are discarded.

A benefit of the push alteration is that it drastically reduces the number of cycles in the search space. In the *foundational* work, alterations can create cycles in the search space. Any sequence of alterations that will result in the same configuration of the board as that was started with is a cycle. As an example: the move alteration could move the player left and right indefinitely. There are many more examples of such cycles, and the move alteration is involved in most of them. Although the push alteration does not completely get rid of such cycles, it does reduce the amount of cycles that can occur drastically.

There is also a drawback in using the push alteration. It does not scale well with an increasing board size. The complex push alteration is a more costly to compute than the simple move alteration. Moreover, the expected time to find all push alterations depends on the size and state of the board and the expected time of applying the push alteration depends on the length of the path. Therefore, the time in which the push alterations can be found and applied is not constant. On large boards this computation time could get too large and neglect the benefits of the candidate improvements.

Implementation

The implementation of an alteration requires two components. The first is finding the available alterations on a board state, and the second one is applying an alteration to a board state.

To find the available push alterations all floor tiles that are reachable from the current position of the player are first found. Figure 4.1 shows all reachable tiles marked with a white circle. A flood-fill algorithm is used to find the reachable tiles. The flood-fill is performed on empty floor tiles. The flood-fill is slightly altered to prioritize the exploration of tiles that have been marked as used. This ensures that tiles not yet marked as used are only used when necessary. The flood-fill is executed breath-first. This makes sure that path that is found to a reachable tile is the shortest possible path. This path is used to apply the alteration.

Once all reachable tiles have been found, checks are performed from each reachable tile into all four directions (up, down, left and right). These checks test for the presence of a box in a neighboring tile and for an empty floor tile behind it. If this is the case, a push alteration is created. The push alteration contains the sequences of moves to the reachable tile and includes a move into the direction of the box, to push the box. Figure 4.1 shows an example of all available push alteration on a board. In this example, the player could for instance move to tile $(3, 3)$ and from there push the box on $(2, 3)$ to $(1, 3)$.

Applying the push alteration is now trivial. The box is pushed from its original position in the direction that the player pushed it in, then all tiles from the player to the box are marked as used, and lastly the player is moved to the original position of the box.

Results

Figure 4.2 shows the average results over generating ten puzzles with and without the push alteration. The results show that the push alteration generates higher scoring puzzles on average.

Although the push alteration itself takes more time to compute, on average less time is spent to execute 500.000 iterations. This seems to have two main reasons. First, when the push alteration finds no available pushes, the branch is immediately stopped and therefore a leaf node is immediately reached. Secondly, the push alteration is less likely to find redundant moves that result in identical positions. As an example: A move alteration that moves the player can be followed up by a move alteration that moves the player into the opposite direction back onto its previous position. This is a redundant sequence of moves. Such an example is less likely to occur when the push alteration is used.

4.1.2 No Separate Freeze Alteration

Motivation

The freeze alteration is used to separate the deletion of walls and placement of boxes from the simulated play. This is necessary since deleting an obstacle or adding a box during simulated play could break the solution to the puzzle. However, a separate freeze alteration is in essence an unnecessary alteration that increases the size of the search space. Removing the separate freeze alteration decreases the search space and might therefore improve the results.

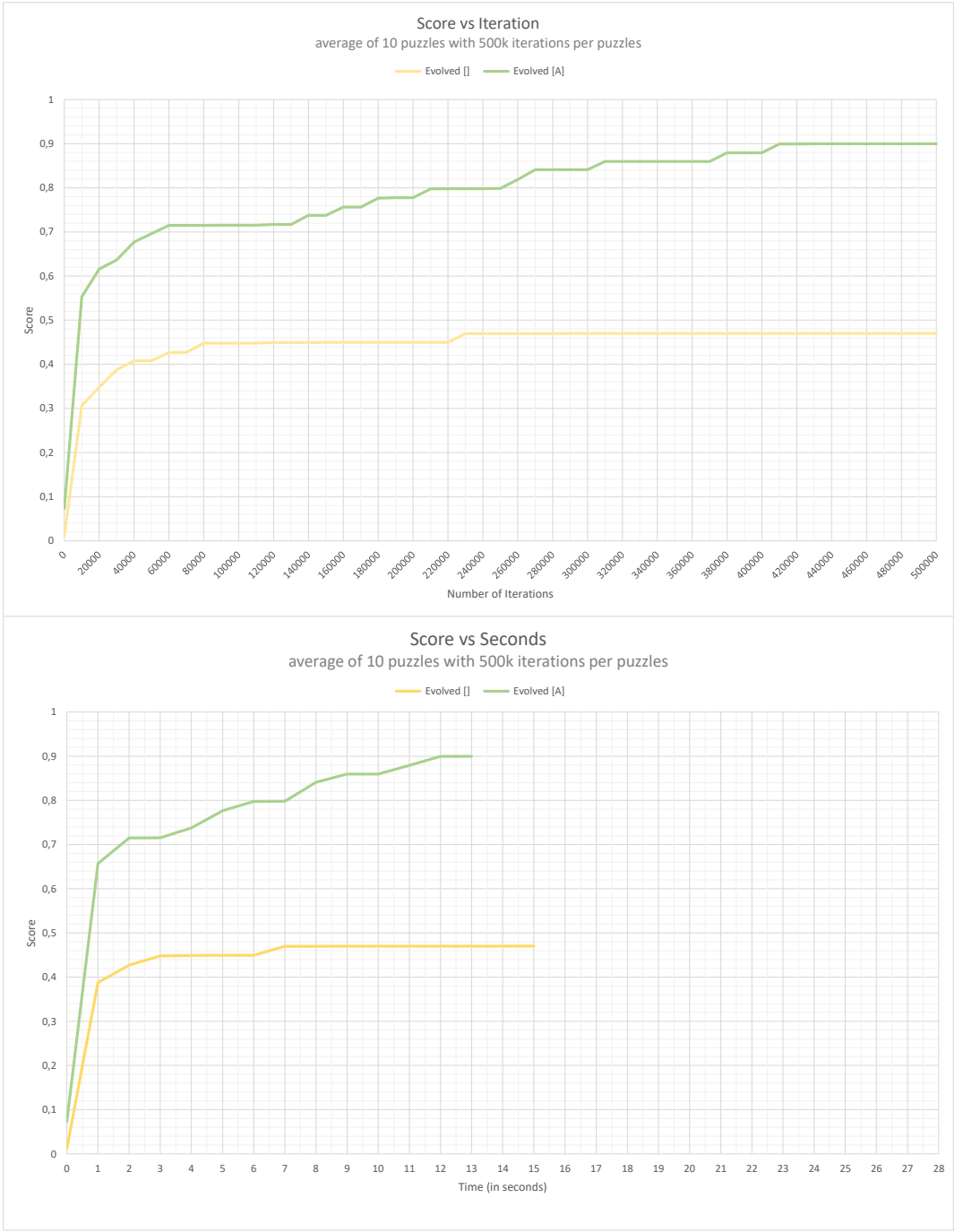


FIGURE 4.2: Two graphs that show the score over an average of 10 puzzles at the seconds and the iterations used for puzzles generated with (green) and without (yellow) Alteration Extending.

Concept

The freeze alteration does not have to be a separate alteration, the concept of this candidate improvement is to always allow the move alteration but to disallow delete and place alteration after a move alteration has been applied. The move alterations are then no longer restricted to only work on boards that have been frozen, instead, a move alteration freezes the board and delete and place alterations are restricted on frozen boards.

Results

Removing the separate freeze alteration has shown not to be successful. The average score of the puzzles dropped, when no separate freeze alteration was used.

The bad results are probably due to the fact that the separate freeze alteration does serve a function. Its function is to separate the delete and place alterations from the simulated play. The freeze alteration hereby decreases the number of iterations that are spend on exploring states that simulate play in the first few layers of the tree. This is beneficial since simulated play is not interesting there.

4.2 Symmetry Reduction

In this section candidate improvements that fall into the category Symmetry Reduction are described. These candidate improvements try to improve the *foundational* work by removing the symmetry that is present within the search space.

In the following subsections, first the motivation behind symmetry reduction is given, then a visualization of the first layers of the search space is given, then simple search space reduction techniques are described and lastly improvements that try to reduce symmetry are described.

4.2.1 Motivation

The search space that is defined by the alterations of the *foundational* work contains symmetry. Spending computer power on exploring configurations is a waste of time when a configuration symmetrical or identical to it has already been explored. The goal is to minimize the time spent on exploring configurations that have already been seen before, and thereby maximize the time spent on new and unique configurations. Removing this symmetry will reduce the search space while preserving the existence of all possible configuration in the search space.

The symmetry that has been found in the search space of the *foundational* work can be categorized into the following two categories:

- an equal board configuration reached by symmetrical branches within the search tree
- a board configuration that is symmetrically equal to another board configuration elsewhere in the tree

An example of symmetry in the search tree: Given a configuration that has two empty floor tiles A and B, where a box can be placed by a place alteration. The search space contains a branch in which a box is placed on tile A and then on tile B, and also a branch in which a box is placed on tile B and then on tile A. Although these branches seem like completely different configuration to the MCTS, the resulting

configuration are equal, due to the symmetry in the branches. This symmetry can be avoided by only allowing one of these two branches to exist.

Ideally, there would be a system that prunes out all symmetry from the search space while preserving the existence of at least one unique instance of each possible configuration. Building and verifying such a system is however very complex. Some explanations of this complexity are:

- Visualizing the search space to find examples of symmetry is hard.
- Programmatically defining which configurations are symmetrically identical is not trivial.
- Verifying whether the symmetry reduction algorithm preserves all possible configurations is hard to verify.

A candidate improvement does not have to be perfect. If it can very rapidly prune out half of the symmetry, the overall results might be better than a very slow candidate improvement that removes all of the symmetry. Moreover, it is not that bad if ten percent of the possible puzzle configuration are lost, if that means the other ninety percent of the puzzles can be found more quickly. Finding a good candidate improvement that reduces symmetry is thus a balance between speed and effectiveness.

4.2.2 Understanding the Search Space

To reduce symmetry a good understanding of the search space is essential. The insights gained from this subsection can be used to understand the structure of the search space and can be used to understand what possible changes to the search space can be made by candidate improvements.

Visualization Tools

To get a better understanding of the search space two visualization tools have been made. Figure 4.3 shows a preview of these visualizations. One preview of the configurations that arise in the top layers of the search tree, and one preview of the visualization of the UCB values in the MCTS. The visualizations of the search space showed where symmetry in the search space existed. These insights were used to come up with candidate improvements.

Measurements

To get a better understanding of the search space, measurements have been done on the size of the search space.

- In Table 4.1 the size of the tree, the number of nodes, and the average growth of a node from the previous depth can be seen for depths one to nine. As can be conclude each node has approximately **nine** children at depth five, which slightly increases as the depth increases.
- After five hundred thousand iterations in the MCTS the average depth at which a child node is expanded is **5.45**, the average depth of a roll out is **14.04**, and the average depth of a roll out that resulted in an improvement is **34.99**.

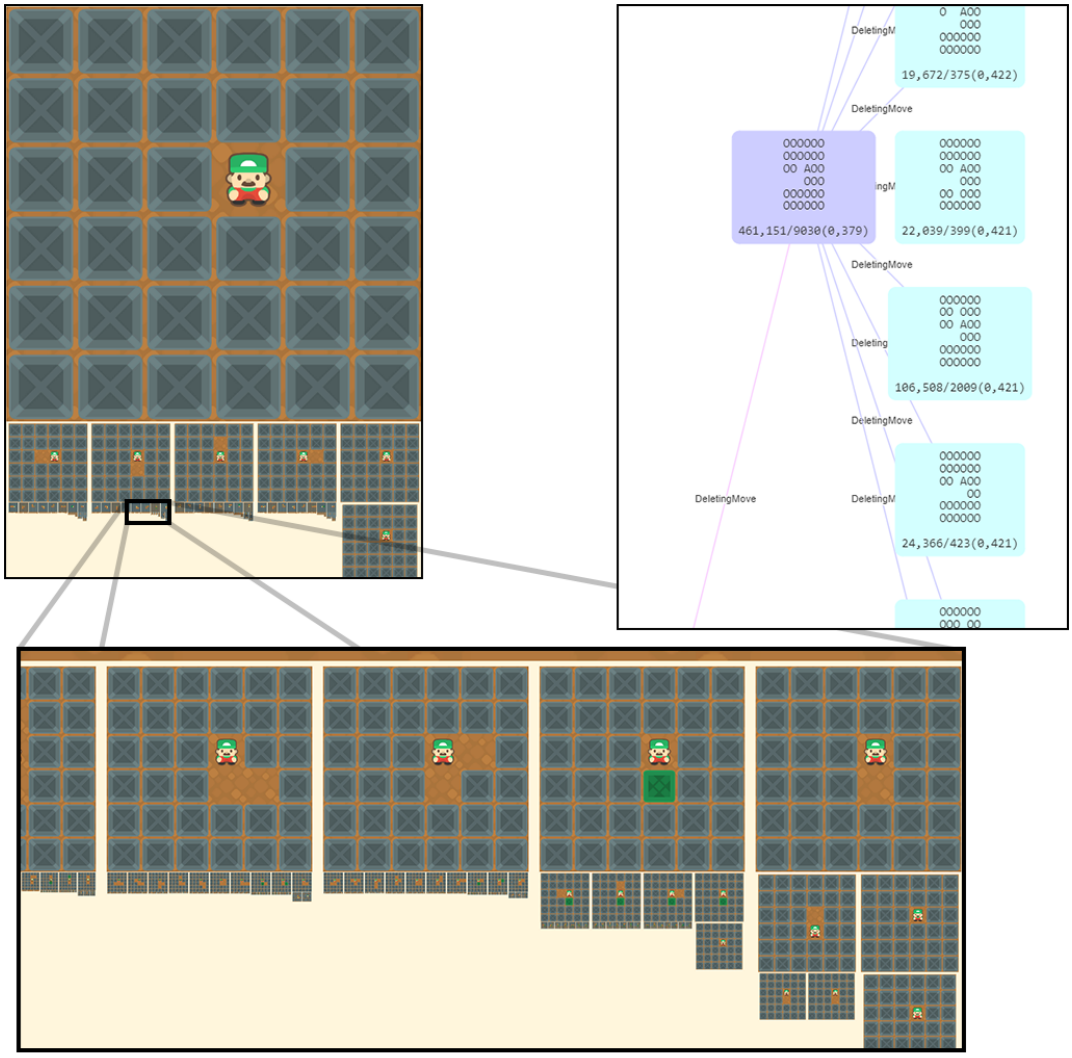


FIGURE 4.3: A preview of the visualization of the configurations of the search tree (on the left and bottom) and a preview of the visualization of the UCB values of the MCTS after five-hundred-thousand iterations (on the right).

Depth	**Tree Size**	**Number of Nodes**	**Growth**
1	1	1	-
2	6	5	x5.00
3	38	32	x6.40
4	268	230	x7.19
5	2.454	2.186	x9.43
6	22.990	20.536	x9.39
7	229.866	206.876	x10.07
8	2.412.794	2.182.928	x10.55
9	26.275.178	23.862.384	x10.93

TABLE 4.1: Tree Size, Number of Nodes, and Growth at different depths in the tree

4.2.3 Search Space Reduction

While trying to get a better understanding of the search space, some insightful observations were made. These resulted in two simple approaches to reduce the size of the search space. These approaches do not relate to symmetry reduction, but reduce the size of the search space in a different manner.

- Applying the freeze alteration in the first few layers is not useful. In the first few layers the freeze alteration will always result in puzzles that have an estimated difficulty score of zero.

  The approach used to counter this problem is to only allow freeze alteration from the **ninth** layer. When no delete or place alterations are available, the freeze alteration can always be applied to ensure that the play-phase is reached.

- There should always at least be one box before the simulation part of the search space is reached. Otherwise resulting puzzles have a near zero estimated difficulty score.

  The approach used to counter this problem is to ensured that at least **three** and at most **seven** boxes are placed before the freeze alteration is applied.

Figure 4.4 shows a graph of the average score over ten puzzles per iteration for not applying search space reduction (Evolved [A]) and a graph for applying the search space reduction (Evolved [A + SSR]). The graphs show that these simple reductions of the size of the search space have a slightly positive effect on the speed by which higher scoring puzzles are found. These effects are most notable during the first two hundred fifty thousand iterations of the search. Due to the positive results, these two improvements are also applied in all symmetry reduction improvements described in the following subsections.

4.2.4 Hard-coded Initial Layers

Concept

Before trying to implement a symmetry reduction approach that dynamically prunes parts of the search space, a hard-code symmetry reduction test is done. In this test the first five layers of the search tree have been collapsed into one layer containing all unique configurations that would be available after five layers of only applying delete alterations. The number of hard-coded layers was empirically set at five.

Symmetry

The symmetry that is pruned out using this approach is symmetry in the shape of the cut-outs of the delete alterations. Cut-out shapes are considered symmetrical when they can be either rotate or flipped onto each other.

Implementation

Figure 4.5 shows the available variants. Each row is a layer of cutout from the search space, the first column shows the unique cut-outs from the previous row, and the second column are the possible cut-outs after applying one more delete operation adjacent to the existing cut-out. Unique variants are underlined by a green line. As can be seen in the second column of the last row there are twelve unique variants after a depth of five layers.

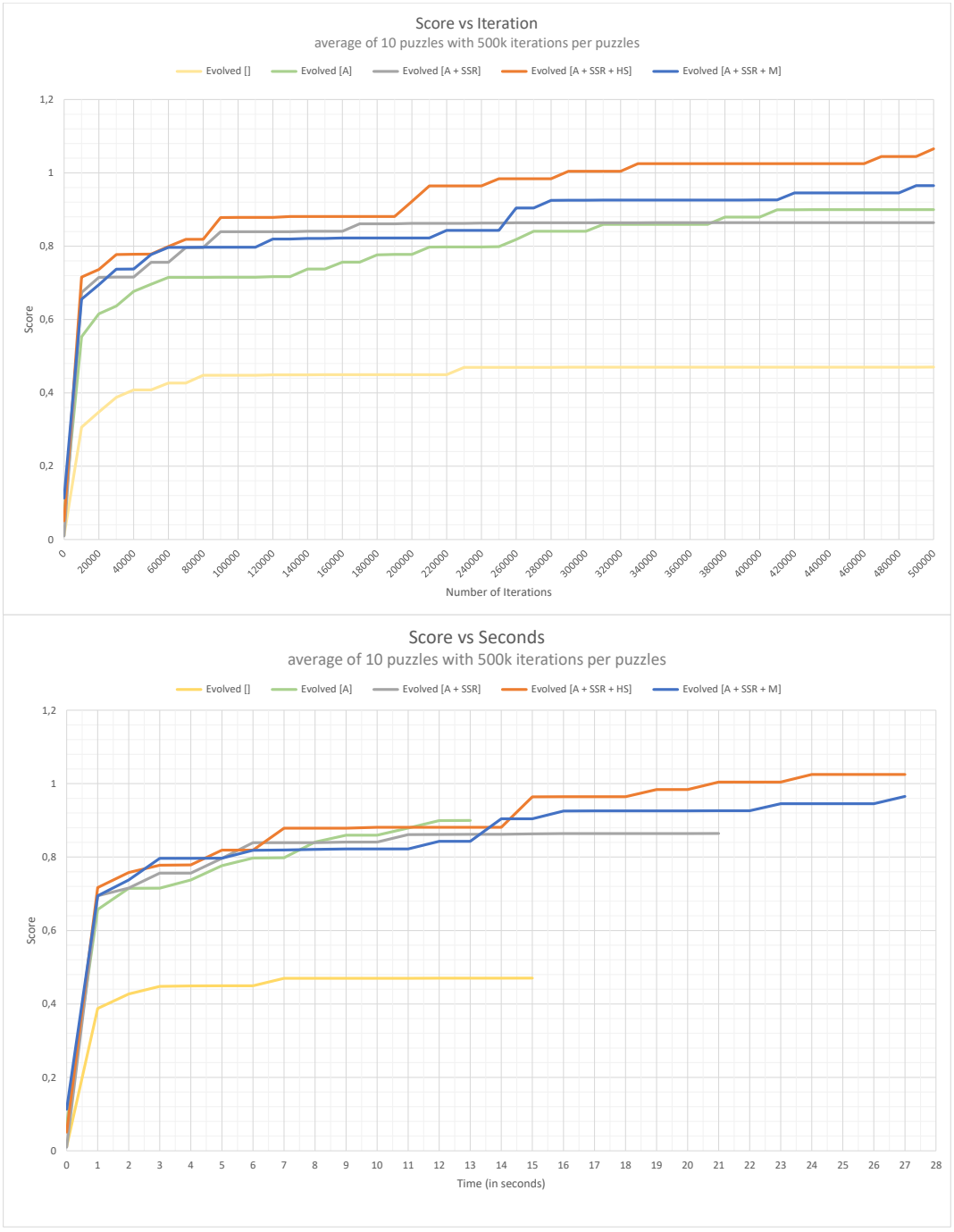


FIGURE 4.4: Multiple graphs of the average score over ten puzzles per iteration for different symmetry reduction techniques.
The **yellow** graph shows the results of the replica of the *foundational* work. The **green** graph shows the results of Alteration Extending. The **gray** graph shows the results of Alteration Extending and Search Space Reduction. The **orange** graph shows the results of Alteration Extending, Search Space Reduction, and Hard-coded Symmetry Reduction. The **blue** graph shows the results of Alteration Extending, Search Space Reduction, and Mirror Symmetry Reduction.

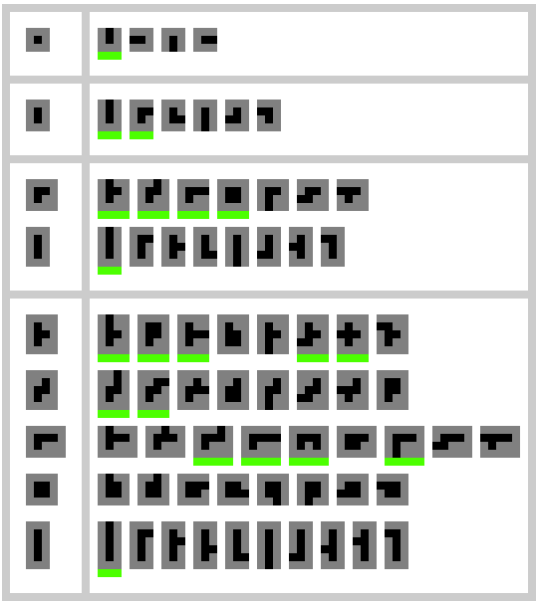


FIGURE 4.5: An overview of the unique hard-coded cut-out shapes after five layers of delete alterations.

A new type of alteration is used, the cut-out alteration. This cut-out alteration has twelve variants, one for each of the unique cut-outs. The first alteration made in the search tree is always one of these twelve cut-out alterations. They are not used in the rest of the search tree.

Results

Table 4.1 shows that the original size of the search tree at depth five is around 2454 nodes of which 2168 are at depth five itself. Using this implementation the number of nodes at depth five is brought down to twelve. This is a reduction of the search space at depth five from 2454 to 13 nodes. Which is a 99.5% reduction.

These twelve nodes do not contain all distinct configurations possible after five layers of the original search tree. This has two important reasons. The first reason is that only delete alterations are considered. Place, freeze, push, and evaluation alterations are not considered. The second reason is that all cut-out shapes are applied at the center of the puzzle, whereas the in the original approach they could arise at any position within the puzzle.

Figure 4.4 shows a graph of the average score over ten puzzles per iteration for not applying the hard-coded initial layers (Evolved [A + SSR]) and a graph for applying the hard-coded initial layers (Evolved [A + SSR + HS]). These graphs shows that the hard-coded initial layers perform better than the original approach.

Removing symmetry from the branches is therefore very promising. Because by only reducing a small fraction of the symmetry, a notable effect on the outcome can be seen. It is a small fraction, because although pruning out 99.5% of the first five layers seems like a lot, it pales into insignificance when compared to the whole search tree that has depths of around 40.

The idea was to test whether symmetry reduction is useful at all, based on this hard-coded test. The results show that symmetry reduction that only operates on the first few layers certainly is already worth it and that pruning out symmetrical branches throughout the whole tree might even improve the results further.

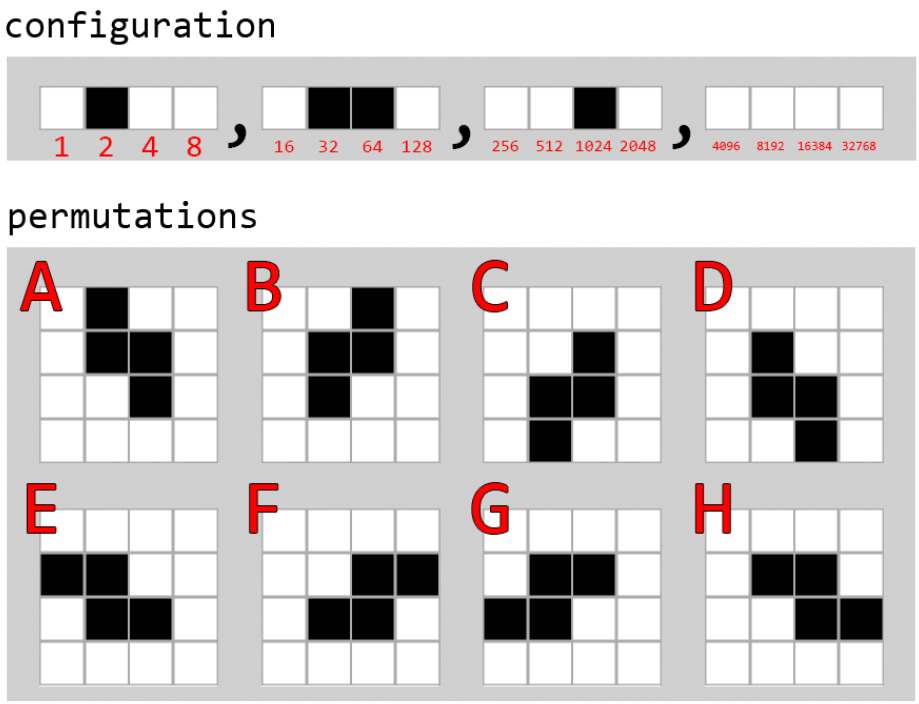


FIGURE 4.6: Different permutations of the same example configuration.

4.2.5 Lexicographical Ordered

Concept

[Wal06] shows that lexicographical ordering can be successfully used to prune symmetry from constraint solving problems. This concept can also be applied to the structure of the search space. To prune the search space, this approach only allows the search space to expand into configurations that are the smallest possible lexicographical ordering of all symmetrical permutations.

Symmetry

Within this candidate improvement symmetry is only reduced in the shape of the cut-out that results after applying delete alterations. Configurations are considered symmetrical identical if they are different permutation of each other. Different permutations of the same configuration can be seen in Figure 4.6. In this figure the tiles resemble a Sokoban puzzle, the white tiles are obstacles, and the black tiles are places were a delete alteration has deleted an obstacle. The black tiles form the cut-out shape of the configuration. This cut-out shape can be described in eight different lexicographical orderings. The description can start at the left or right side of the puzzles, the description can start at the top or bottom of the puzzle, and the description can describe the puzzle row-first or column-first. This gives $2 * 2 * 2 = 2^3 = 8$ distinct lexicographical orderings.

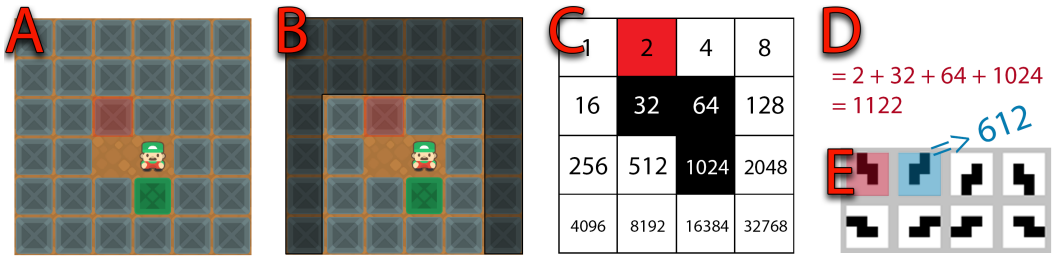


FIGURE 4.7: Example process that filters out lexicographical non-minimal permutations.

Implementation

The implementation of this candidate improvement is a filter. The filter is applied when a child is expanded into its possible delete alterations. The expansion of a child into a delete alteration is only allowed when the resulting cut-out shape is lexicographically minimal. To do this, all possible permutations of the cut-out shape are compared with the actual one. Figure 4.7 shows an example of this process. It consists of five steps, from A to E.

Step A shows the input of the process. The input consists of the board state before the candidate delete alteration is applied and the location of the candidate delete alteration. The candidate delete alteration is indicated by a red square.

Step B shows how the board is cropped to the size of the cut-out shape, including a one tile wide border around it. This is the area of the cut-out shape in which all possible delete alterations could be applied.

Step C shows the visualization of the cut-out shape. The white tiles correspond to obstacles, black tiles correspond to non-obstacles, and the red tile indicates the candidate delete alteration. The red and black tiles together form the cut-out shape after the candidate delete alteration would be applied.

Step D shows the calculation of the permutation-value of the cut-out shape from last step. Its value is 1112.

Step E shows that this permutation-value is then compared to all other possible permutations of the same cut-out shape. Since the cut-out shape in blue has a value of 612, which is smaller than 1112, the **candidate delete alteration is not minimal and it is therefore rejected**.

This approach does have a limitation. It is limited in the size of the board. To calculate the permutation-value the size of the board is limited to the size of the data type that is used to store numbers. The number of tiles in the minimal area that is created at Step B can be as large as the size of the board. 2^{Nt} is the maximum number that the data type needs to store in which Nt is the number of tiles. If a 32-bit integer is used, which can represent values up to 2^{32}, the maximum supported board size is 32 tiles. For a 64-bit integer the maximum supported board size is 64, which corresponds to a board size of 8x8 tiles.

The search space will collapse to zero nodes if the lexicographical ordering is applied on each depth in the tree. This happens because the minimal permutations are not necessarily children of each other within the search tree. The lexicographical

Depth	Without LOSR	With LOSR
1	1	1
2	6	4
3	38	12
4	268	39
5	2.454	141
6	22.990	603
7	229.866	2.464
8	2.412.794	25.594
9	26.275.178	215.709

TABLE 4.2: Size of the search space with and without Lexicographical Ordered Symmetry Reduction (LOSR)

permutation that are used are therefore slightly altered. On each odd depth all eight permutations of the configuration are considered, however on each even depth only the first four permutations are considered. Although this will result in symmetrical permutation of symmetrical position occuring in the search space, the algorithm still reduces a lot of symmetry by only using a small amount of time.

Table 4.2 shows the difference in search space with and without Lexicographical Ordered Symmetry Reduction.

An important aspect of the MCTS is that similar configurations should be close together in the search space. Given the MCTS finds a high scoring configuration in some part of the search space and starts to exploit that part of the search space, the MCTS expects to find more high scoring configuration nearby. If this is not the case, using an MCTS to explore the search space is no longer useful. Using the approach described in this subsection, similar configurations are no longer guaranteed to be close to each other in the search space. At the beginning of the search tree this is not so much of a problem, however when diving deeper into the tree the alterations should not be filtered to disallow exploration of configurations that might have a symmetric configuration somewhere else in the tree. These symmetric configurations might have not been encountered yet, and during the current search we might not find them. To avoid this problem the symmetry reduction approach is only applied for the first eleven layers of the search tree.

Results

Before the Evolved project was written an experimental C# project was created. This candidate improvement was only implemented on the C# project. This C# project ran 20 times slower than the Evolved project written in C++. The results are therefore not comparable to the C++ project to which the other approaches are compared.

In Figure 4.8 the results of this improvement in the C# project can be seen. This approach results in a slight improvement.

4.2.6 Node Jumping

Concept

This candidate improvement tries to use node jumping to reduce the symmetry in the search space. The idea is that when a configuration is reached that has already been identically, or symmetrically reached somewhere else in the search space, that

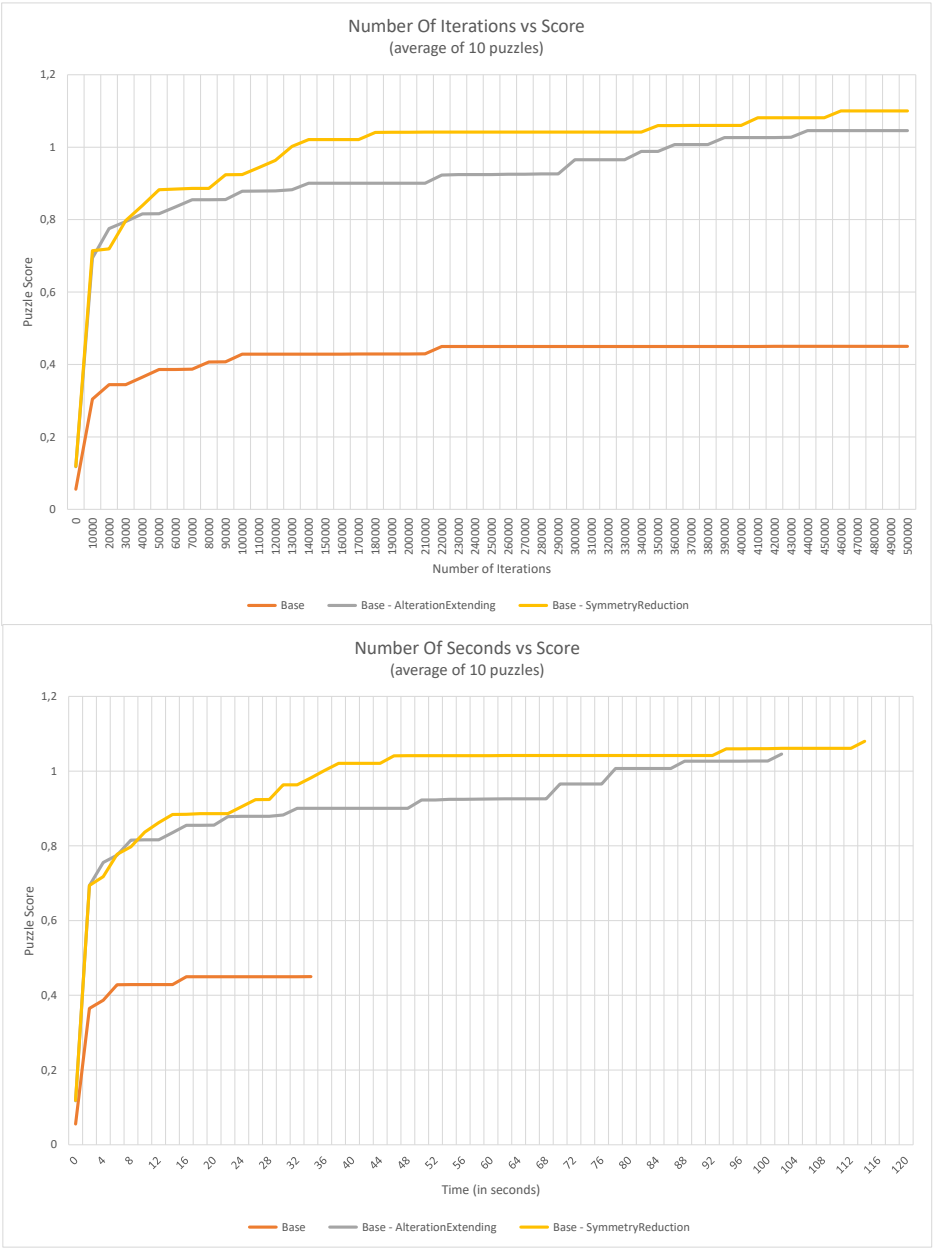


FIGURE 4.8: A graph of the average score over ten puzzles per iteration in the C# project. Red is the base graph, gray is the graph with alteration extending, and yellow is the graph with alteration extending and with the lexicographical ordering improvement.

instead of adding a new node, a reference to the previously reached node is created. In essence, instead of adding a new node, a jump is made to the node where the configuration was first seen. This way, existing information that has been collected at that node can immediately be used instead of starting exploration of the essentially same configuration all over again.

Implementation

When a node is expanded into children, for each of the children a search has to be performed to find whether the resulting configuration already exists somewhere else in the tree. This search cannot compare the configuration to each node in the tree since this would take to much time. To avoid this, a hash map is used. Hash maps have a constant add and find speed. To use a hash map a hashing function must be created for configurations. Configurations that are symmetrically equal should return the same hash.

A node jump can only be added when the configuration is 100% identical or symmetrical to the node that is jumped to. The hash function should therefore take the cut-out shape, used tiles, box placements, box origins, and player location into consideration. Moreover, since the hash has to be calculated for each node, the hash function should be fast. A hashing method exists that is both accurate and fast, Zobrist Hashing. This hashing method is very fast because given the hash of the node before the last alteration the new hash can be calculated in constant time. The changed that the alterations made simply have to be hashed-in to the hash.

The hashing function is used when a child node is created. The hash of the child node is calculated and a check is performed whether this hash already is present in the search tree. If the hash does not exist, the node is created and it is added to the hash map. If the hash does exist, a jump to the already existing node is inserted instead of creating a new child.

Results

Unfortunately after implementing the approach, it did not work. The structure of the search space is heavily altered by using this approach. The resulting structure is no longer a hierarchical non-cyclic tree. Directed cycles were able arise in the resulting search space when jumps to nodes are added. Unfortunately, no way around this problem was found.

4.2.7 Mirror

Concept

The mirror symmetry reduction approach tries to remove symmetry over the whole depth of the search space. The mirror approach does not try to remove all symmetry. The mirror approach only removes some of the symmetry, but tries to do this very efficiently. The mirror approach removes symmetry in the creation of child nodes. When multiple alterations on a node are found that will result in symmetrical identical children, only one of these nodes is accepted, while the others are rejected.

Symmetry

Configurations that this approach considers symmetrical are configurations in which the cut-out shape is symmetrical. When a configuration is horizontally, vertically

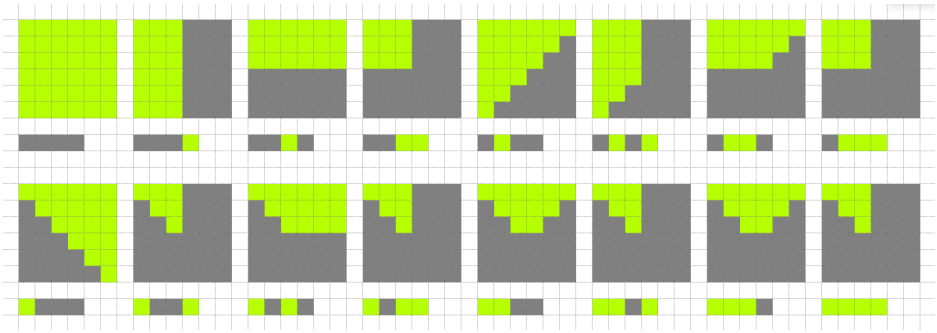


FIGURE 4.9: All possible variants of mirrors.

or diagonally (for square boards) symmetrical, alterations on only one side of the symmetry line are allowed.

Implementation

Figure 4.9 shows all possible variants of mirrors. The green tiles underneath each 6x6 board describe on which axis the board is symmetrical: (1) diagonal right, (2) diagonal left, (3) vertical, and (4) horizontal. Gray means the board is not symmetrical on that particular axis and green means the board is symmetrical on that axis. On each board the green tiles indicate the tiles on which a delete alteration is allowed. The gray tiles indicate that no delete alteration is allowed. These are not allowed since these would create a symmetric alteration that could also be created from within the green area. On each board there are always some green tiles, meaning some delete alterations are always allowed.

Just before the delete alterations are added, the mirror variant that corresponds to the board configuration is picked. Using this mirror variant only tiles that lie within the green area are allowed and the other delete alterations are rejected.

Results

Figure 4.4 shows that the mirror approach (A + SSR + M) is a slight improvement in comparison to not using the mirror approach (A + SSR).

5 Experiment

To answer the research question a comparison between the *foundational* work and the *improved* work is made. An experiment is used to make this comparison. The experiment is a user study that tries to verify the assumption that the *improved* work, on average, results in more interesting puzzles. Within this chapter the hypothesis, sample size, setup, datasets, comparison and analysis of the experiment are outlined.

5.1 Hypothesis

The user study of this experiment compares two datasets. The first dataset is a set of puzzles from the *foundational* work. The second dataset is a set of puzzles from the work that has been done in this master thesis. The most promising candidate improvements were chosen and combined to generate the second dataset. This is referred to as the *improved* work. The goal of the experiment is to compare these two datasets on their interestingness. Apart from the interestingness, the tools that are used for this comparison, also try to gather as much additional data as possible during the experiment.

The hypothesis is that the puzzles from the *improved* work, on average, are more interesting than the puzzles from the *foundational* work. The *improved* work tries to achieve more work in the same amount of iterations, without spending much more time. Since both datasets have been generated with the same number of cycles, the puzzles from the *improved* work data set should generally be more interesting.

5.2 Sample Size

Due to time constraints an experiment with strong confidence cannot be conducted. The experiment is therefore only an initial guideline to whether the *improved* work might be more interesting than the *foundational* work.

To be able to compare the interestingness of the puzzles, the puzzles must be of the same difficulty in both data sets. If the puzzles are not of the same difficulty, then the fact that people dislike or like difficult or easy puzzles would influence the interestingness of them. For this experiment we can assume that the puzzles in both data sets are homogeneously difficult. This assumption is verified during the experiment.

Given a confidence level of 80% and a margin of error in the result of 10%, the sample size of the user study is calculated to be 40 samples per comparisons.

To eliminate bias from the users: (1) each sample is provided by a unique user and (2) the order in which the puzzles from both sets are presented to the user is random.

To eliminate bias from the generation methods: each puzzle that is pre-generated is only used once.

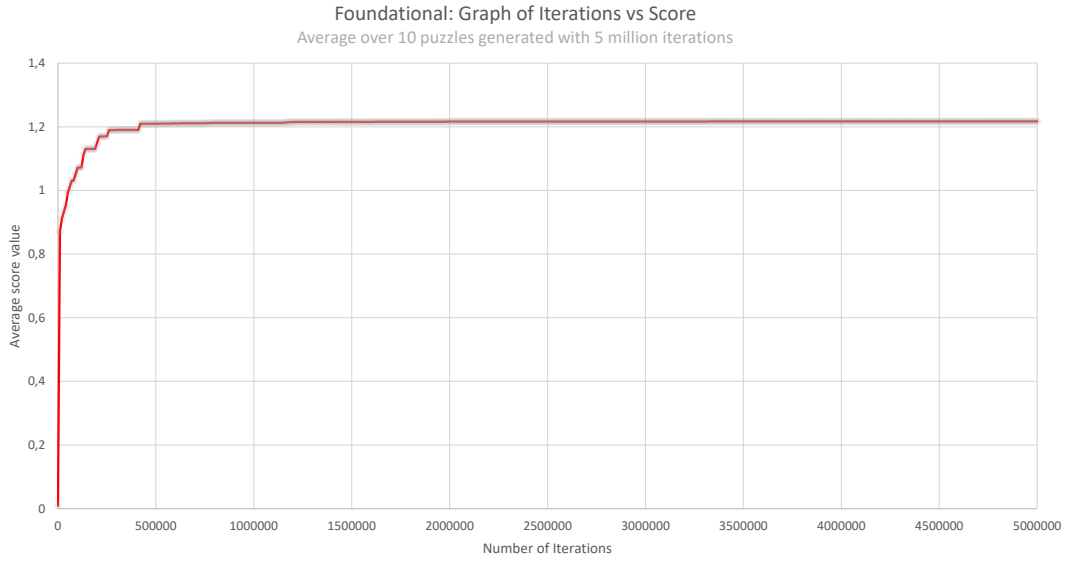


FIGURE 5.1: Graph of the average score of ten puzzles from the *foundational* work over the number of iterations.

5.3 Setup

Within this section the experimental setup is described. In the following subsections an explanation for the chosen number of iterations and the chosen board size is given. Replicating the *foundational* work was not trivial, this resulted in differences between the *foundational* paper and the implementation used by this experiment. These differences are discussed in the last subsection.

Number of Iterations

The *foundational* work describes how using more iterations should result in more interesting puzzles. Figure 5.1 shows a graph of the average score over ten puzzles over the number of iterations. This graph shows that after around 500.000 (five hundred thousand) iterations, the result is about as good as after 5.000.000 (five million) iterations. In this experiment all puzzles have been generated using 100.000 (one hundred thousand) iterations or 500.000 (five hundred thousand) iterations.

The average score reached by performing a certain number of iterations is not the only interesting property to look at when trying to improve a iteration based system. The duration of each iteration is also very important. Just looking at the number of iterations is biased, because an approach could execute more time consuming algorithms in a single iterations to achieve better results. Improving an iteration-based approach is a choice between executing more complex algorithms per iteration, or doing as many simpler iterations as quickly as possible. Therefore the comparisons do not just show the average score after a certain amount of iterations but also show the amount of seconds it took to execute these iterations. The execution time can heavily differ between different implementations. Therefore, comparing executing time is only insightful when comparing within the same implementation.

Board Size

In this experiment comparisons between different puzzles are made. To make a fair comparison, the sizes of the boards of the puzzles should be the same. Therefore a fixed board size was set. The board size was set to six by six tiles. The choice for this board size is mainly based on the following three reasons:

- **6x6** puzzles are generated in the *foundational* work,
- **6x6** is not too large. Therefore puzzles can be generated within 30 seconds and with reasonable memory usage of around two gigabytes,
- **6x6** is not too small. Therefore interesting puzzles can still arise within 6x6 board configurations. As an example Figure 5.2 shows highly interesting puzzles on 4x6 boards.

Parameter Differences in Foundational Work

Replicating the *foundational* work was not trivial. The *foundational* work was not reproducible from the paper describing it. The results of the *foundational* work were different from the replica. Fortunately, the development project of the *foundational* work was made available by the authors. A comparison between the development project of the *foundational* and the development project of this master thesis was made. This showed some differences between the *foundational* work and the replica were found. These are set out in this subsection. This might not be all, since there might be more differences left that were not found during this master thesis.

Estimated Value of Nodes Within the *foundational* work the estimated value that is used when computing the UCB is based partly on the highest scoring node found in its children. The UCB function is therefore slightly altered. The estimated value is calculated by the following formula $E(n) = (S_m + S_a)/2$ in which $E(n)$ is the estimated value of a node, S_m is the score of the highest scoring child seen so far and S_a is the average score of its children. This seemed to have no significant effect on the outcome and it is therefore not used in the *foundational* work replica used by this master thesis.

Ordered Child Expansion In a default MCTS implementation a random child is expanded during the child expansion step. However, within the *foundational* work, the children are expanded in order of appearance to save execution time. A costly randomized pick of a non-expanded child can be avoided, instead a simple shift to the next child can be used. It does change the outcome of the algorithm slightly since alterations that appear first are picked first. This seemed to have a significant impact on the results and is therefore used in the *foundational* work replica used by this master thesis.

Weights of Metrics The weights used for the metrics in the paper of the *foundational* work differ from the weights used in the development project of the *foundational* work. The weights that were found in the development project seemed to work best and were therefore used in the *foundational* work replica used by this master thesis.

- Box Count Metric: **10**,
- Congestion Metric: **4***,

- 3x3 Block Metric: **1 over the number of tiles in the board**.

*The Congestion Metric uses an additional multiplier, the square multiplier. This multiplier is calculated for each box individually. The rectangle that encapsulates the box path is taken into account. The value is bigger on a rectangle that more closely resembles a square. The multiplier is calculated by the following formula: $squareMultiplier = (R^w + R^h)/max(R^w + 1, R^h + 1)$ in which R^w is the width and R^h is the height of the rectangle.

Tunable Constant of UCB The MCTS is unstable in the selection phase when used with a tunable constant C of $\sqrt{2}$ as described by the paper of the *foundational* work. In the development project of the *foundational* work a value of $5\sqrt{2}$ is used that seems to work better. This value is also used in the *foundational* work replica used by this master thesis.

Upper Limit on Number of Boxes The *foundational* work adds an upper limit to the amount of boxes that can be placed. This upper limit was not described in the paper, but is present in the development project. An upper limit on the number of boxes is not used in the *foundational* work replica used by this master thesis.

5.4 Datasets

The experiment compares two datasets to empirically determine which dataset has puzzles that are more interesting. The datasets consist of 80 pre-generated puzzles. Half of each dataset has been generated with 100.000 (one hundred thousand) iterations and the other half has been generated using 500.000 (five hundred thousand) iterations. Although the puzzles are pre-generated to save time while conducting the experiment, the puzzles are assumed to be generated right before a user plays them.

During the experiment a quick comparison is also made between a dataset of handcrafted puzzles and the *improved* work.

The following subsection describe the *improved* work dataset and the handcrafted dataset.

5.4.1 Improved Work

Chapter 4 described seven candidate improvements falling in two categories: Alteration Extending and Symmetry Reduction. In both categories, one candidate improvement seemed to work best. For the Alteration Extending category this is the '**Push Alteration**'-improvement and for the Symmetry Reduction category this is the '**Hard-coded Symmetry Reduction**'-improvement. This includes the Search Space Reduction that is included in all symmetry reduction candidate improvements.

These candidate improvements have shown to improve the *foundational* work. The *foundational* work including these two improvements is the *improved* work proposed by this master thesis. Figure 4.4 shows the results of the *improved* work in the graph (Evolved [A + SSR + HS]). It shows that the average score after 500.000 iterations is significantly higher than that of the *foundational* work (Evolved []).

5.4.2 Handcrafted dataset

A dataset of handcrafted Sokoban puzzles was collected. These puzzles can be compared to puzzles generated by the work in this master thesis. This is useful to get and understanding in which ways generated puzzles still differ from handcrafted puzzles.

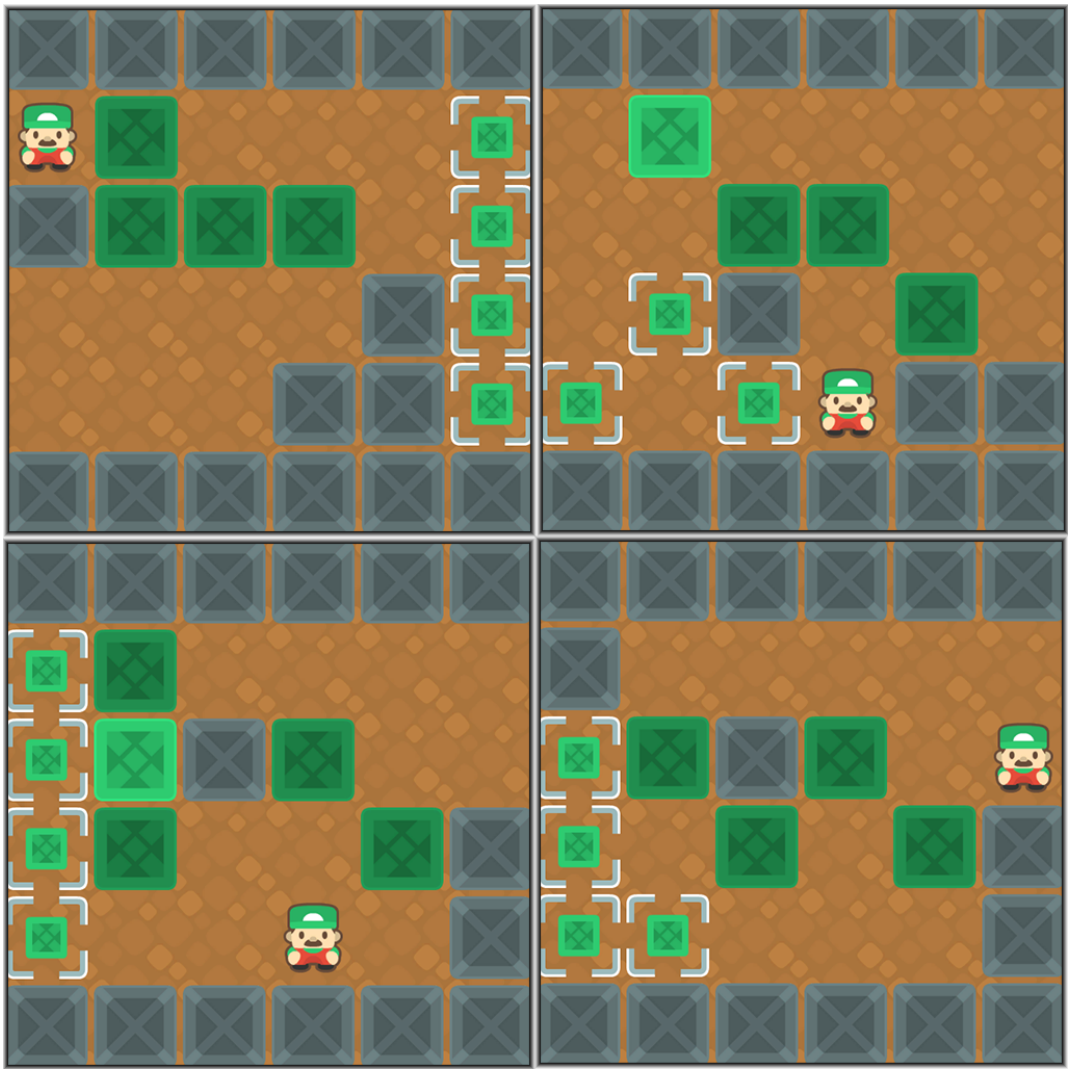


FIGURE 5.2: Handcrafted puzzle examples requiring at least *left-top*: 74, *right-top* 116, *left-bottom*: 119 and, *right-bottom* 105 moves to solve on boards with a size of 6x4.

Figure 5.2 shows examples of puzzles that are highly interesting. The puzzles require certain 'key'-moves to be found through-out the entire solution of the puzzle. As pointed out by the *foundational* work, most puzzles generated with it only have a few key moves, while the handcrafted (high quality) puzzles seem to have key moves until the end of the solution.

5.5 Comparison

The comparison of the two datasets will be done using a comparison tool that was created for this master thesis. This section explains the process a user makes while using the tool.

The user will start with a tutorial. The tutorial familiarizes the user with the controls and the gameplay. It is important that the user has a good understanding of the gameplay before beginning the experiment.

Next, the comparison tool presents the user with two puzzles, one from each datasets, that was generated with 100.000 thousand iterations. The user will play both these puzzles in a random order without knowing to which dataset the puzzles belong. There is no restriction on the number of resets, number of solves, or play time that the user has while playing the puzzles. During the playthrough of each puzzle, the following data is collected in the background:

- The number of seconds a player played a certain puzzle. The timer starts when the player makes the first move (or after 3 seconds) and stops when the player clicks on the continue/next button to either go to the next puzzle or go to the questions.
- A log of all the actions the player took. Containing the following actions:
  - Moves: Logging of each move (per direction). A move is added to the log when the input resulted in the player successfully moving on the board.
  - Resets: Logging of each reset of the puzzle to its original state.
  - Solves: Logging of each time the puzzle was solved. The puzzle can remain unsolved, solved once, or solved multiple times.
- The minimum number of moves it took a player to solve the puzzle. Or -1, indicating that the puzzle was not solved.

After playing the first two puzzles, the user decides which puzzle it found more interesting to play and which puzzle it found more difficult to play. The player also assigns from 1 to 5 stars for the following four questions:

- interestingness of the first puzzle,
- difficulty of the first puzzle,
- interestingness of the second puzzle,
- difficulty of the second puzzle.

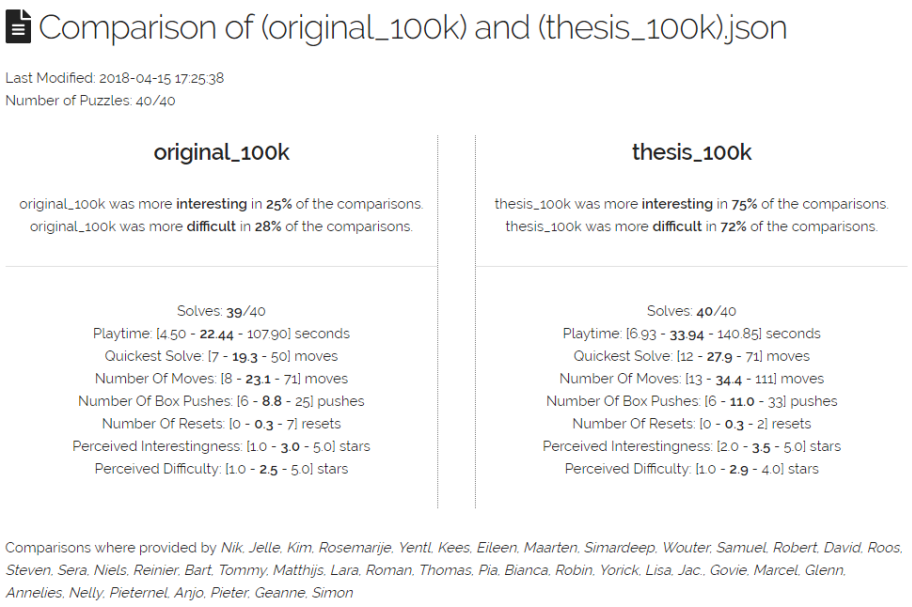
Next the process starts over, skipping the tutorial and picking two puzzles that were generated with 500.000 thousand iterations from the datasets. After playing these two puzzles and answering the questions the user has finished the experiment.

5.6 Analysis

In this section the output from the experiment is analyzed. Figure 5.3 and Figure 5.4 show the results of the user study.

5.6.1 Assumption

In Section 5.2 the assumption was made that the puzzles in both datasets are of homogeneous difficulty. This was verified during the experiment. The user study shows that all datasets are of homogeneous difficulty. People rate both datasets with an average of 2.9 stars out of 5 for difficulty. The exception are the puzzles of the *foundational* work at 100.000 iterations, these puzzles have a slightly lower difficulty rating of 2.5 stars out of 5.


Comparison of (original_100k) and (thesis_100k).json

Last Modified: 2018-04-15 17:25:38
Number of Puzzles: 40/40

original_100k	thesis_100k
original_100k was more **interesting** in **25%** of the comparisons. original_100k was more **difficult** in **28%** of the comparisons.	thesis_100k was more **interesting** in **75%** of the comparisons. thesis_100k was more **difficult** in **72%** of the comparisons.
Solves: **39**/40 Playtime: [4.50 - **22.44** - 107.90] seconds Quickest Solve: [7 - **19.3** - 50] moves Number Of Moves: [8 - **23.1** - 71] moves Number Of Box Pushes: [6 - **8.8** - 25] pushes Number Of Resets: [0 - **0.3** - 7] resets Perceived Interestingness: [1.0 - **3.0** - 5.0] stars Perceived Difficulty: [1.0 - **2.5** - 5.0] stars	Solves: **40**/40 Playtime: [6.93 - **33.94** - 140.85] seconds Quickest Solve: [12 - **27.9** - 71] moves Number Of Moves: [13 - **34.4** - 111] moves Number Of Box Pushes: [6 - **11.0** - 33] pushes Number Of Resets: [0 - **0.3** - 2] resets Perceived Interestingness: [2.0 - **3.5** - 5.0] stars Perceived Difficulty: [1.0 - **2.9** - 4.0] stars

Comparisons where provided by *Nik, Jelle, Kim, Rosemarije, Yentl, Kees, Eileen, Maarten, Simardeep, Wouter, Samuel, Robert, David, Roos, Steven, Sera, Niels, Reinier, Bart, Tommy, Matthijs, Lara, Roman, Thomas, Pia, Bianca, Robin, Yorick, Lisa, Jac., Govie, Marcel, Glenn, Annelies, Nelly, Pieternel, Anjo, Pieter, Geanne, Simon*

FIGURE 5.3: User study results of the puzzles generated with 100.000 iterations

5.6.2 Outcome

The hypothesis defined in Section 5.1 is that puzzles of the *improved* work dataset should be more interesting on average. The user study shows that puzzles generated by the *improved* work are indeed generally more interesting. The puzzles also generally have more playtime, more moves to solve the puzzles, more overall moves and more box pushes. The puzzles of 100.000 iterations from the *improved* work dataset were found to be more interesting by 75% of the users and the interestingness is on average 0.5 stars higher than of the *foundational* work. The puzzles of 500.000 iterations from the *improved* work dataset were found to be more interesting by 60% of the users and the interestingness is on average 0.2 stars higher than of the *foundational* work.

5.6.3 Correlation

Figure 5.5 gives an overview of the correlation between different variables of the results of the user study. A value close to 0 means that two variables are not correlated and a value closer to 1 means that the two variables are correlated. The figure consists of five tables. The first table is the correlation overview of all the puzzles from all datasets. The other four tables show the correlation overview of a specific dataset with a specific number of iterations. There seems to be no significant difference in the correlation tables of the specific tables. The main focus is thus on the first table which shows the correlation for all datasets.

The correlation overview shows some obvious correlation. The number of moves correlates with the playtime. This is obvious since a user that makes more moves use more time.

Notable is that the perceived interestingness does not correlate with any of the other variables. The interestingness does only partly correlate with the perceived

FIGURE 5.4: User study results of the puzzles generated with 500.000 iterations

difficulty (0.57). The fact that interestingness does not seem to correlate with simple variables indicates that interestingness is not easily captured by simple metrics.

5.6.4 Handcrafted

The first time the experiment was conducted, the users compared puzzles form the handcrafted dataset and the *improved* work dataset. During this experiment the difference in difficulty between the two datasets was too large. The puzzles from the handcrafted dataset were significantly harder to solve. The users would take around fifteen to twenty minutes to solve a handcrafted puzzle, sometimes even without success. On the other hand the puzzles from the *improved* work data set were solved in under a minute. This shows that the puzzles from the *improved* work dataset still lack the quality of handcrafted puzzles.

all

	Playtime	*Quickest Solve*	*Number Of Moves*	*Number Of Box Pushes*	*Number Of Resets*	*Perceived Interestingness*	*Perceived Difficulty*
Playtime	1,00						
Quickest Solve	0,40	1,00					
Number Of Moves	0,67	0,84	1,00				
Number Of Box Pushes	0,71	0,59	0,87	1,00			
Number Of Resets	0,62	0,42	0,63	0,72	1,00		
Perceived Interestingness	0,17	0,25	0,28	0,28	0,15	1,00	
Perceived Difficulty	0,39	0,32	0,40	0,41	0,40	0,57	1,00

original_100k

	Playtime	*Quickest Solve*	*Number Of Moves*	*Number Of Box Pushes*	*Number Of Resets*	*Perceived Interestingness*	*Perceived Difficulty*
Playtime	1,00						
Quickest Solve	0,54	1,00					
Number Of Moves	0,78	0,88	1,00				
Number Of Box Pushes	0,90	0,72	0,90	1,00			
Number Of Resets	0,82	0,67	0,72	0,83	1,00		
Perceived Interestingness	0,15	0,31	0,36	0,24	0,05	1,00	
Perceived Difficulty	0,50	0,45	0,52	0,47	0,39	0,65	1,00

thesis_100k

	Playtime	*Quickest Solve*	*Number Of Moves*	*Number Of Box Pushes*	*Number Of Resets*	*Perceived Interestingness*	*Perceived Difficulty*
Playtime	1,00						
Quickest Solve	0,42	1,00					
Number Of Moves	0,56	0,91	1,00				
Number Of Box Pushes	0,41	0,62	0,82	1,00			
Number Of Resets	0,49	0,50	0,76	0,73	1,00		
Perceived Interestingness	0,09	0,10	0,15	0,19	0,16	1,00	
Perceived Difficulty	0,40	0,38	0,44	0,38	0,45	0,34	1,00

original_500k

	Playtime	*Quickest Solve*	*Number Of Moves*	*Number Of Box Pushes*	*Number Of Resets*	*Perceived Interestingness*	*Perceived Difficulty*
Playtime	1,00						
Quickest Solve	0,43	1,00					
Number Of Moves	0,62	0,80	1,00				
Number Of Box Pushes	0,57	0,47	0,88	1,00			
Number Of Resets	0,63	0,27	0,79	0,88	1,00		
Perceived Interestingness	0,25	0,19	0,39	0,49	0,44	1,00	
Perceived Difficulty	0,34	0,27	0,44	0,46	0,46	0,60	1,00

thesis_500k

	Playtime	*Quickest Solve*	*Number Of Moves*	*Number Of Box Pushes*	*Number Of Resets*	*Perceived Interestingness*	*Perceived Difficulty*
Playtime	1,00						
Quickest Solve	0,28	1,00					
Number Of Moves	0,74	0,68	1,00				
Number Of Box Pushes	0,80	0,44	0,91	1,00			
Number Of Resets	0,81	0,20	0,72	0,81	1,00		
Perceived Interestingness	0,16	0,18	0,17	0,13	0,04	1,00	
Perceived Difficulty	0,51	0,24	0,39	0,39	0,38	0,54	1,00

FIGURE 5.5: Correlation overview of the results of the user study

6 Conclusion

A puzzle generator that can generate interesting puzzles, can be used to generate interesting worlds. The Sokoban puzzle game is a puzzle game that suits research into puzzle generation well. It is interesting to know whether it is possible to generate Sokoban puzzles that are interesting to play using simulated play.

Generative Grammars (GG), Spatial Algorithms (SA), Modeling and Simulation of Complex Systems (CS), and Artificial Intelligence (AI) show promising application within puzzle generation. [KSG16b] uses CS and AI to generate Sokoban puzzles using simulated play, this is the *foundational* work. The *foundational* work showed that slightly challenging Sokoban puzzles can be generated using simulated play that are guaranteed to be solvable.

Within this master thesis it is tested whether the *foundational* work can be improved to generate Sokoban puzzles that are interesting to play. This resulted in the research question: "Can simulation be used to generate Sokoban puzzles that are interesting to play?". The concluding answer is:

No. Although the generated puzzles are slightly challenging, they do not come near the interestingness of handcrafted puzzles. It might be achievable by using a different implementation, but highly interesting puzzles are not achievable with the current approach. There seems to be an upper limit that the current approach cannot (consistently) exceed. This is mainly due to the use of simple metrics that do not seem to be able to capture the underlying concepts of interestingness.

What the puzzles seem to lack are key moves through-out the whole solution of the puzzle. After one or a few key moves the solution to most puzzles becomes trivial. The simple metrics that are used do not capture such key moves. The definition of a key move is not trivial either. Examples of key moves are:

- a (sequence of) move(s) that is hard to spot or that is overlooked at first sight,
- a (sequence of) move(s) that might feel like locking the configuration, but is actually the right way to progress,
- a sequence of moves that feels like a short puzzle itself.

Such key moves can be identified by humans, however a solid definition of what a key move is is quite hard to establish. As is instructed by the given examples, key moves have something to do with look and feel of the puzzle, these concepts are hard to grasp for a computer. Key moves make a puzzle interesting. Due to the complex nature of key moves, it can be concluded that simple metrics will probably not be able to capture the definition of them. If a definition for key moves can be defined for Sokoban, then the definition will probably rely on specific knowledge and insights of the Sokoban puzzle game. Which makes it hard to generically use it for other puzzle games.

Although the approach was unsuccessful in generating Sokoban puzzles that are interesting to play, some interesting candidate improvements on a replica of the *foundational* work were created. Two candidate improvements stood out in their performance. The first one being the **Push Alteration**. The Push Alteration replaced the

simple move alteration by a more complex alteration used during simulated play. It simulates multiple consecutive alterations at once by only focusing on the alterations that have an impact on the result. The second one is **Hardcoded Symmetry Reduction**. The Hardcoded Symmetry Reduction prunes out symmetry in the upper layers of the search space, which decreases the size of the search space and therefore increases the chance of exploring unique and promising Sokoban puzzles.

These most promising candidate improvements were taken to the test in an experiment. The *improved* work was defined to be the replica of the *foundational* work including the Push Alteration and the Hardcoded Symmetry Reduction. A puzzle dataset from the replica of the *foundational* work was compared to a puzzle dataset of the *improved* work. An user study is used to make this comparison. The user study showed three interesting results:

- The user study showed that the puzzles generated by simulated play are notably less interesting than handcrafted puzzles.
- The user study showed that puzzles from the *improved* work are generally more interesting than puzzles from the replica of the *foundational* work.
- The user study showed that there seems to be little to no correlation in the perceived interestingness of the puzzles with simple metrics such as the quickest solve, number of box pushes, and playtime. Indicating that interestingness is not easily captured by simple metrics.

6.1 Limitations

The current approach has some limitations. These limitations are described in the following subsections.

6.1.1 Congestion Metric

The congestion metrics that calculates an estimated difficulty score of a puzzle configuration has a limitation. Although most of the times the metrics are correct there are situations in which the estimated difficulty score is far off the actual score. This is due to the fact that boxes are not labeled. In the estimated difficulty function the boxes are expected to go to a specific goal tile, however sometimes a simpler solution exists. The result is that although a puzzle seems difficult to the MCTS, it can actually be trivial to solve.

6.1.2 Search Space Structure

There are two important limitations in the structure of the search space. The first one is that similar puzzle should be close together in the tree. When the MCTS is exploring puzzles it has expected to be interesting, it should be able to find similar puzzles in that part of the tree. The second limitation is that puzzles that are similar in configuration should also be similar in interestingness. If highly interesting puzzles are scattered randomly around the search space, then looking for interesting puzzles by checking variants of those puzzles that are nearby in the search space does not make sense.

6.1.3 Max Evaluation Score

The evaluation score seems to be capped to a max value of around 1.2. Of all puzzles seen so far, no puzzle has exceeded this number. This indicates that it is very rare for highly interesting puzzles, such as hand made ones, to arise from this generation technique. And that, although somewhat interesting puzzles can be found, with the current metrics, better results are not to be expected.

6.2 Future Work

In the following subsections some potential future work is discussed.

6.2.1 Applicability to Other Puzzle Games

In the introduction the assumption was made that the *foundational* work can be used to generate puzzle configuration for other puzzle games. Due too time constraints, this lies outside the scope of this master thesis, future research is necessary to verify this assumption. Apart from the metrics and the alteration steps, the *foundational* work is generally applicable to other puzzle games. The metrics and alterations have to be designed for each puzzle game individually. It is important to know whether the *foundational* work can be used to generated puzzles for different types of puzzle games too. If this is the case then research on Sokoban puzzle generation can be applied to many different games.

6.2.2 Key Moves

In the conclusion some examples of key moves are given. It can be insightful to try to define what a key move in the game Sokoban actually is. Based on this definition new metrics can be formed. These metrics should reward puzzles with key moves and penalize puzzle configurations that lack key moves. After it has been shown that using key moves can generate interesting puzzles for the puzzle game Sokoban, other puzzles games could also be tried. Different games might have key moves based on similar definitions. Showing that patterns do, or do not exist in the key moves of different games can be very insightful.

6.2.3 Restricted Push Alteration

A limitation of the proposed push alteration in this thesis is that the expected time to find and execute a push alteration is not constant. The move alteration that this push alteration replaced is constant. A middle ground might be feasible that searches for nearby box pushes and that allows a simple move alteration to reach boxes further away.

6.2.4 Heat-map

The conclusion states that the metrics are too simple to capture interestingness. A new idea for a more complex metric that might be used to estimate the difficulty score of a puzzle configuration is to create a heat-map of the usage of the tiles of the puzzle. The state of the heat-map can be kept track of when applying the alterations, making sure not much additional execution time is spent on updating the heat-map. This heat map could positively reward usages of all sections of the puzzle and it could also positively reward parts of the puzzle that are highly congested.

6.2.5 Recursive MCTS

A possible way to improve the results of the MCTS could be to recursively step down the MCTS after a fixed amount of iterations have been applied. For example after every ten-thousand iterations, the MCTS could change the root of the search space to the most promising child of the current root (based on the UCB). If at every ten-thousand iteration the next alteration is fixed, then after five-hundred-thousand iterations the puzzle will be 50 alterations deep. This could simply be implemented by breaking out of the the back propagation step when a node with a depth smaller or equal to $currentIteration/10.0000$ is reached. All existing information that is already present in the branch of the search space from the first ten-thousand iterations will be preserved.

6.2.6 Flexible Board Size

Another way to possibly improve the results of the MCTS is that the board could be made of a flexible width and height. This could be implemented by starting with a board of 1x1 with just the player on it, and allowing the delete alterations to increase the size of the board. This possible improvement would make it easier to reduce symmetry, since the puzzle are no longer positioned on a board, which removes a way in which puzzles can be symmetrical. This type of symmetry is already tried to be removed by the candidate improvement 'Lexicographical Symmetry Reduction', but would be made much easier when a flexible board size is used. To make sure that size of the board does not get too large an upper limit on the width and height could be added.

6.2.7 Player Starting Position

The starting position of the player can be made variable. In the current situation the player always starts in the center of the board. It might be hard to generate very interesting puzzles that have a player in the center of the board. In most handcrafted puzzles the player does not seem to start at the center of the board. Figure 5.2 shows that in a lot of highly interesting puzzles the player start at the side of the puzzle.

6.2.8 Embed Flood-fill in Search Space

Although the push alteration seems to work very good, there might still be room for improvement. An idea is to embed the flood-fill algorithm of the push alteration within the move alteration. The move alteration would then, apart from moving the player, also mark all neighboring tiles from the tile that was moved from as not walkable. Only when a push is made by this simple move alteration the walkable tiles will be reset. This will execute the flood-fill algorithm within the search space itself. It will also leave the opportunity for all possible paths open, and not just the shortest ones.

To ensure that not all tiles have to be enumerated to reset the empty tiles that are available for box pushes, a counter could be kept of the number of box pushes since the start of the simulated play. When the player moves to a floor tile the number of box pushes since the start of the simulated play can be saved. A move can then only be allowed to floor tiles that have a value lower than this number.

References

[Mas43] Abraham H Maslow. "A theory of human motivation." In: *Psychological review* 50.4 (1943), p. 370.

[Dor10] Joris Dormans. "Adventures in level design: generating missions and spaces for action adventure games". In: *Proceedings of the 2010 workshop on procedural content generation in games*. ACM. (2010), p. 1.

[Col97] Joseph C. Colberson. *Sokoban is PSPACE-complete*. Tech. rep. TR 97-02, Dept. of Computing Science, University of Alberta. (1997).

[Pos16] Jelle O.M. Postma. "Generic Puzzle Level Generation for Deterministic Transportation Puzzles". MA thesis. Utrecht University, (2016).

[HMV13] Mark Hendrikx, Sebastiaan Meijer, and Joeri Van Der Velden. "Procedural content generation for games: A survey". In: *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)* 9.1 (2013), p. 1.

[Per85] Ken Perlin. "An image synthesizer". In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[TWR17] Mike Treanor, Nicholas Warren, and Mason Reed. "Playable Experiences". In: *AIIDE* (2017).

[Dor17] Joris Dormans. *A Handcrafted Feel: Unexplored Explores Cyclic Dungeon Generation*. (2017). URL: http://ctrl500.com/tech/handcrafted-feel-dungeon-generation-unexplored-explores-cyclic-dungeon-generation/.

[Mau16] David Maung. "Tile-based Method for Procedural Content Generation". PhD thesis. The Ohio State University, (2016).

[Joh13] Rune Skovbo Johansen. *Layer-Based Procedural Generation for Infinite Worlds*. Youtube. (2013). URL: https://www.youtube.com/watch?v=GJWuVwZO98s.

[Ulr02] Thatcher Ulrich. "Rendering massive terrains using chunked level of detail control". In: *Proc. ACM SIGGRAPH*. (2002).

[Tut12] Tim Tutenel. "Semantic game worlds". ISBN 978-94-6203-259-0. PhD thesis. Delft, The Netherlands: Delft University of Technology, (2012).

[CGG07] Kate Compton, James Grieve, and Ed Goldman. "Creating spherical worlds." In: *SIGGRAPH Sketches*. (2007), p. 82.

[KSG16b] Bilal Kartal, Nick Sohre, and Stephen Guy. "Generating sokoban puzzle game levels with monte carlo tree search". In: *The IJCAI-16 Workshop on General Game Playing*. (2016), p. 47.

[KSG16a] Bilal Kartal, Nick Sohre, and Stephen J Guy. "Data-driven sokoban puzzle generation with monte carlo tree search". In: *Twelfth Annual AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. (2016).

[Wal06] Toby Walsh. “General Symmetry Breaking Constraints”. In: *Principles and Practice of Constraint Programming*. Ed. by Frédéric Benhamou. Springer Berlin Heidelberg, (2006), pp. 650–664. ISBN: 978-3-540-46268-2.