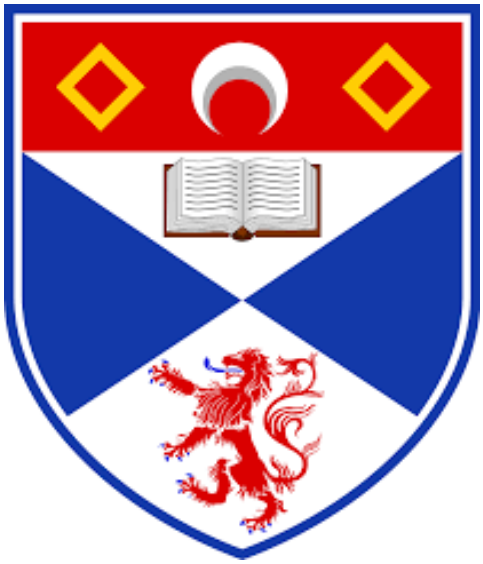
Sokoban Procedural Instance Generation

A report submitted in partial fulfilment of the requirements for
the award of the degree of

B.Sc. (Computer Science)

by

Sami Hatoum



Department of Computer Science
University of St Andrews
2024-2025

Abstract

Procedural content generation is an alternative approach to development that replaces the intricate design choices of human level creation with automated, stochastic techniques. This dissertation explores the development and evaluation of a procedural content generation system for the 2D single-agent puzzle game, "Sokoban". The creation of candidate puzzles can be broken down into three main components, all intertwined with one another: generation, checking, and evaluation. Resulting instances are sorted based on scoring metrics, and trivial or uninteresting candidates are disposed by the evaluator. This combined process is repeated over extended periods of time, iteratively finding more optimal solutions to keep for a given set of objectives.

Ultimately, the degree of uniqueness and difficulty of puzzles is left to the player, however the objective of this project is to "exploit" the properties of a solver to act as a ground truth. Results show the metrics which aim to maximise the total number of diversions, redirections, and switches between the keeper's objectives, produce the most engaging levels for a solver. The findings also indicate a degree of minimalism in the level design increases the branching rate of solver. Reaching a balance between these has led to the production of puzzles which are both immersive, and challenging; a property which translates analogously to human players as well.

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is 14783 words long, including project specification and plan. In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Acknowledgements

I would like to express my heartfelt gratitude to my supervisors, Dr. Ruth Hoffmann and Dr. Joan Espasa Arxer, for their invaluable support throughout this dissertation. Over the past academic year, our meetings have been an inspiring blend of brainstorming, exploring ideas, and problem-solving. Your guidance and enthusiasm have not only shaped this project but have also made the journey thoroughly enjoyable and enriching.

I would also like to thank my parents, Omar and Joumana Hatoum, for their unwavering support and the countless opportunities they have provided me. Their guidance and belief in me have been instrumental in my journey, and I am deeply grateful for everything they have done to help me reach this point.

Contents

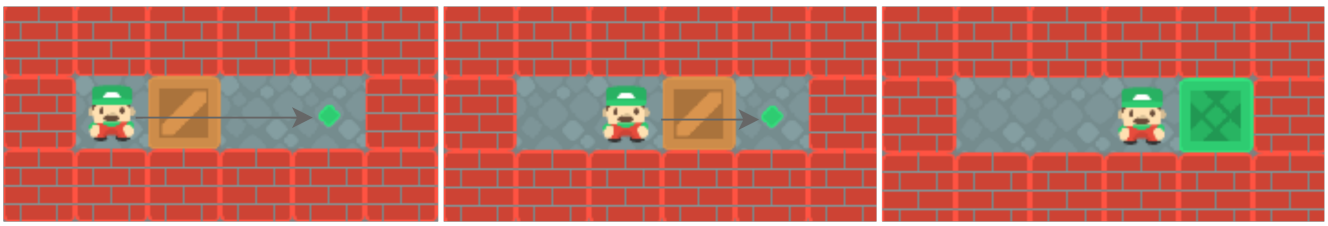
List of Figures

CHAPTER 1

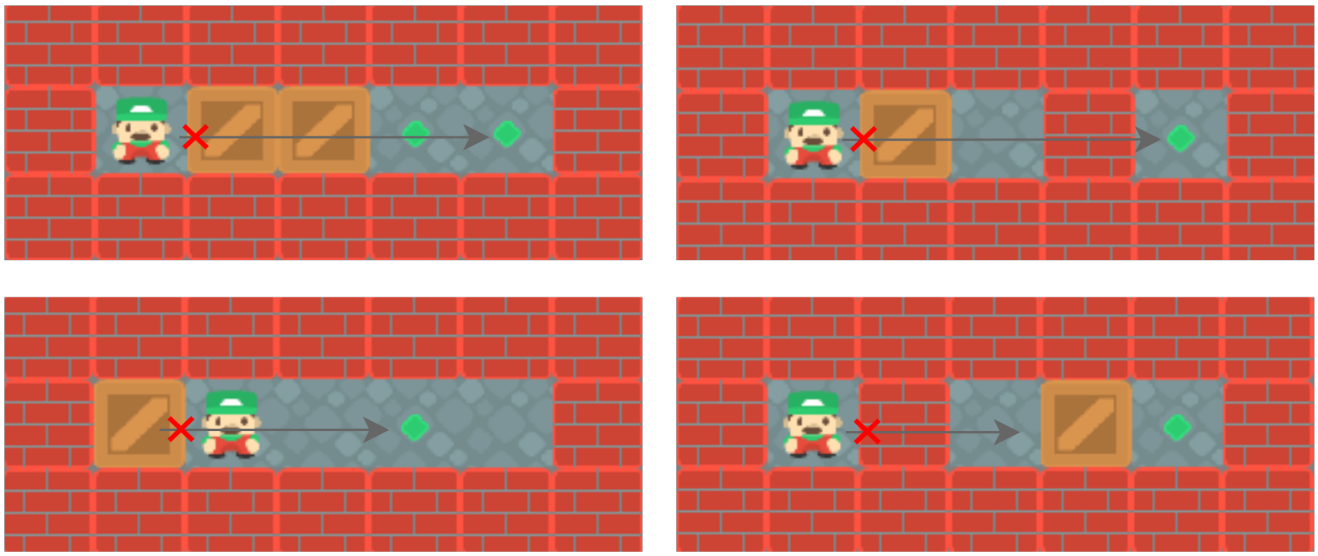
Introduction

"Sokoban", a Japanese puzzle game first released in 1982, tasks players with pushing boxes (or crates) around a warehouse to fill all designated storage locations [25]. The player, depicted most commonly as the warehouse keeper, is constrained to single-square horizontal and vertical movements across a 2D plane, unable to pass through crates or walls, as shown in Figure 1.1a.

In order to move a box, the player must be adjacent to the box with an available space on the opposite side for both to shift in the direction of. Similar to the keeper, a box cannot pass through walls or other boxes (Figure 1.1b), and there must be an equal number of goal locations and boxes to complete the level.



(a) Valid Moves



(b) Invalid Moves

Figure 1.1: Sokoban Tutorial

The simplicity of the rules masks the deep complexity of the game; seemingly insignificant details in a level's design can be the deciding factor between a puzzle that is entirely trivial and one that the vast majority of players and solvers would struggle to complete within a reasonable time-frame. Sokoban has been shown to be PSPACE-complete [3], meaning it is computationally at least as challenging as most single-player puzzle games. As a result, solving it typically requires exponential time, and potentially exponential space as well. This complexity is further highlighted by the fact that optimal solutions for difficult levels often involve hundreds of moves, compounded by a high branching factor, making exhaustive search approaches particularly demanding.

Together, this makes Sokoban a suitably challenging candidate for procedural instance generation, one which is recognised as being hard not only for humans, but also for artificial intelligence applications [7]. Completely random Sokoban levels are extremely likely to be unsolvable or entirely trivial; metrics and heuristics must be put into place to assemble and assess levels with some degree of objectivity.

1.1 Problem Statement

This project explores an alternate path to instance creation for Sokoban, capable of producing solvable levels of varying densities autonomously. Levels are evaluated by a range of metrics and criteria, relative to a pre-defined solver of the game. Instances should be interesting and generated within a reasonable amount of time (or the equivalent, in the number of instances made), defined as such:

- **Interesting**: Challenging to a dedicated solver (and by extension, experienced players), but not due to unnecessarily large sizes, omitting 16x16 grids or greater from consideration
- **Reasonable amount of time**: Twenty minutes of instance generation ($\approx$ 200 instances) to find an optimal solution for a given grid size and entity count

1.2 Report Overview

The following chapter (Context Survey) delves deeper into the background of procedural content generation, both within the broader field of game development and in prior implementations specific to Sokoban. This is followed by a detailed discussion of the artefact's design and implementation, outlining the core components of the generation process and the methodology used for scoring puzzles. The report concludes with an evaluation of the system, assessing its effectiveness against the original project objectives and its relevance within the wider Sokoban procedural generation community.

CHAPTER 2

Context Survey

This chapter defines the core components of the project and reviews relevant literature, including early methods of instance generation, techniques for detecting deadlocks and dead-ends, and the application of evolutionary algorithms.

2.1 Related Work

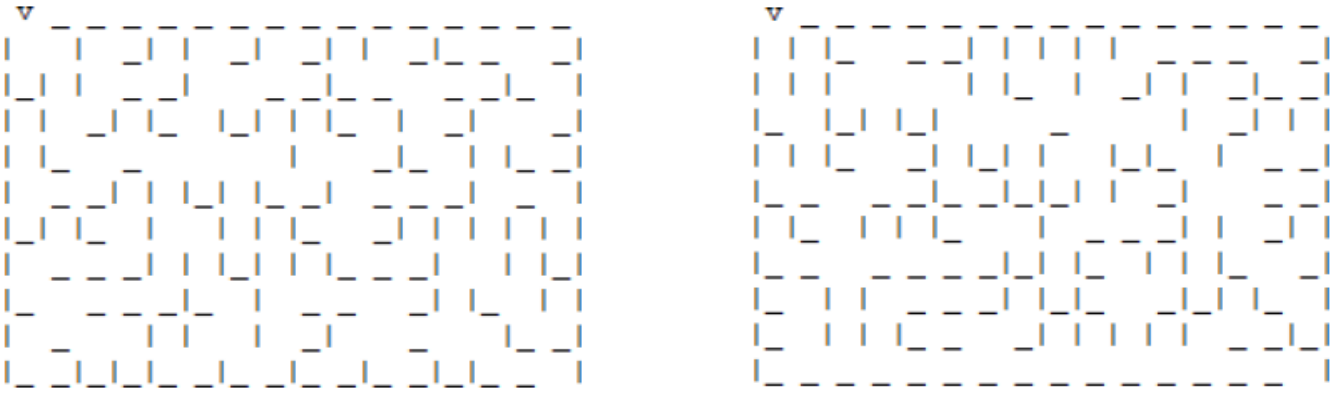
2.1.1 Procedural Content Generation

Procedural Content Generation (PCG) is an approach used to algorithmically produce content (instances) instead of creating it manually. It depends significantly on random number generation (RNG) to produce content, which is subsequently assessed against rules and constraints [20]. PCG often employs pseudo-random number generators (pseudo-RNGs), which offer a computationally efficient way to produce sequences that appear random, while remaining entirely determined by an initial value known as a "seed". Using pseudo-RNG enables reproducibility, as identical seeds consistently produce identical sequences of generated content.

PCG facilitates the automatic creation of rich and dynamic game environments, from intricate labyrinths to expansive natural landscapes, using algorithmic techniques that ensure diversity and scalability in instance creation. PCG is essential for games which do not require a specific layout but rather a specific set of criteria to be met, orientating the components of the level in a near infinite set of ways to achieve that goal, resulting in an unpredictable range of possible game play spaces [20]. Each style of game adopts a certain form of generation which can be adopted in the creation of Sokoban levels:

- **Mazes**: A form of puzzle which tasks the player to navigate through a complex network of paths, often with branching corridors and dead ends, to reach a specific goal. Gabrovšek [22] proposes a generation technique starting with a rectangular graph; initially all the edges (connections) begin as walls. The algorithm converts random walls into passages on each iteration, and the result of this is a sub-graph made out of passages representing a tree-like structured maze. Based on the search algorithm chosen, dead-ends and intersections can be interacted with, and recovered from, differently. One common search algorithm is Depth First Search (DFS) recursive backtracking, which will attempt to explore as deep through the maze as possible, and reset the position of the agent to the first branch that has unvisited paths upon encountering a dead-end. This will repeat until all nodes are searched, marking all instances where a branching decision (an intersection) had to be made by the solver, and where a previously marked node is re-encountered (a loop)[12].

The result of this search exploration will output a maze which can then be "walked through" using a different solver, providing metrics related to the difficulty of the maze, which will be optimised over many generation instances. Alternate algorithms will produce variations in grid related to how the search tree branches. The mazes in Figure 2.1 are a comparison between DFS backtracking and Prim's algorithm—a greedy algorithm, derived from randomised Breadth First Search, which results in far more short dead ends.



(a) Maze generated using DFS Backtracking

(b) Maze generated using Prim's Algorithm

Figure 2.1: Maze Generation Comparison. Source: [22]

- **Dungeons**: Dungeons represent a distinct form of level design, characterised by a labyrinth of interconnected rooms and narrow corridors that contain puzzles and challenges designed around player progression and reward. In contrast to mazes, dungeons exhibit a stronger and more deliberate relationship between gameplay and spatial design, as noted by van der Linden, Lopes, and Bidarra [15], who suggest that this relationship can be controlled explicitly, for example through difficulty parameters, or implicitly, via topology. To effectively maximise difficulty and carefully control gameplay experiences, a constraint-based creation approach is useful, filtering out dungeon configurations that fail to meet predefined criteria for puzzle or enemy placements. Green et al. [5] demonstrate this clearly through object-placement constraints tailored to dungeon complexity and difficulty in Figure 2.2. Topological aspects such as room connectivity, dungeon density, loops, and symmetry further define dungeon design, as done by Shaker et al. [24].

  Unlike puzzle-based PCG approaches used in games such as Sokoban, which typically rely on more computationally expensive generate-and-test methods, dungeon generation often employs constructive algorithms like cellular automata or diggers [24]. These methods are computationally efficient and allow continuous, real-time generation without iterative evaluations or regeneration of outputs. For instance, digger algorithms start at a random grid tile and incrementally build outwards, placing rooms (a 3x3 or larger template) or corridors (a 1xL path, where L is a random number representing the length of the corridor) sequentially in available spaces, ensuring non-overlapping placements, all while maintaining the topological and difficulty constraints established for the game. The steps of the algorithm's progress are illustrated in Figure 2.3.

Entrance	Always within 8 tiles of one end of the **LP**
Exit	Within 5 tiles of the opposite end of the **LP** from the entrance
Treasure Chests	Surrounded by 2 or more walls, with a preference towards 3 walls
Potions	Scattered randomly across the map
Portals	One placed 5-10 tiles from Entrance, the other 5-10 tiles from Exit; at least 10 tiles from each other
Traps	On or around (within 1 tile) the shortest path between the Entrance and Exit
Goblins	One side be a wall
Goblin Mages	Must be adjacent to a Goblin
Ogres	4-8 tiles away from a Treasure chest in LOS
Blobs	4-8 tiles away from a Potion in LOS
Minitaur	4-8 tiles away from Entrance

Figure 2.2: Dungeon Generation Object Placement Constraints. Source: [5]

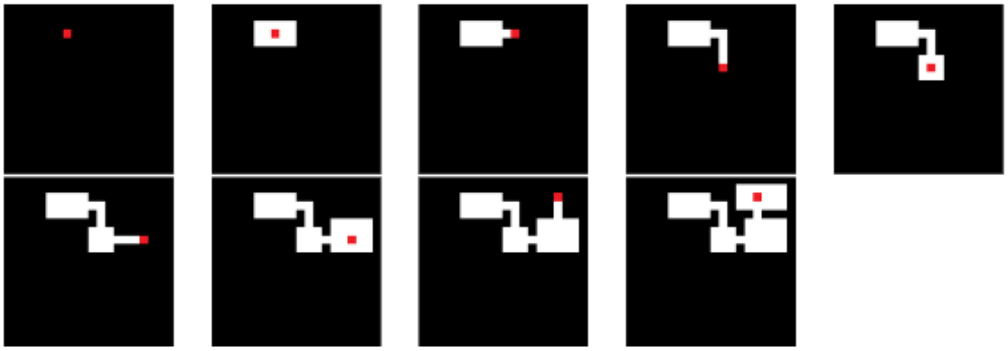
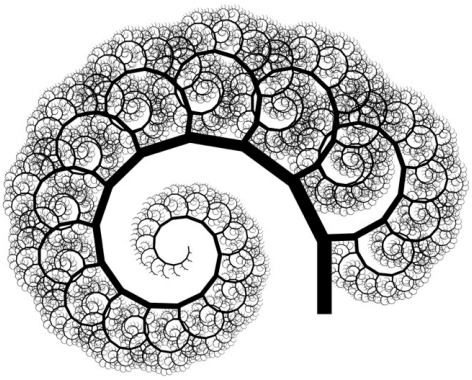


Figure 2.3: Dungeon Generation Using Digger. Source: [24]

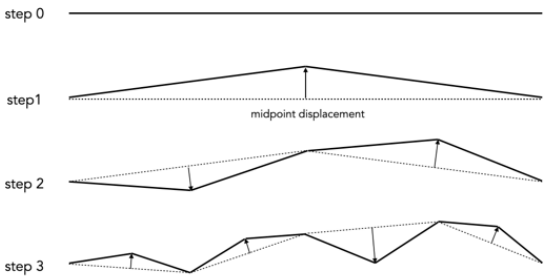
- **Terrain**: Autonomously creating terrain can present a unique challenge to PCG because standard processes of random number generation can struggle to mimic the fragmented shapes and patterns found in the natural world. Rose and Bakaoukas [23] describe these patterns as non-Euclidean shapes. Unlike what is found in standard geometry such as squares and triangles, "nature is built in fractals (seen in Figure 2.4a)—a higher level of geometrical complexity altogether" [23]. Fractals have two distinct properties, they are self-similar and chaotic, meaning they can be broken down into smaller versions of themselves while still remaining infinitely complex on each recursion.

  Fractal Brownian Motion (fBM) is a technique used for generating height-maps of landscapes on a 2D grid which posses fractal properties. fBM is constructed by summing multiple frequencies (or octaves) of noise—seemingly random but structured numerical patterns, each scaled and adjusted in amplitude [23]. This controls the roughness of the resulting terrain; lower values yield more rugged landscapes with sharp peaks and valleys, whereas higher values create smoother, rolling hills. By iteratively combining noise functions, each adding finer details to the height-map, terrain features maintain consistent levels of detail regardless of zoom level, closely mimicking natural geological formations regardless of the scale.

fBM can however be considered too slow for general purpose use [14]. Perhaps the most well established noise function, and an approximation of fBM, is midpoint displacement. The algorithm begins with a straight line spanning two points. At the midpoint (± random offset determined by the length of the current line segment) the line is split into two new segments, calling this recursively produces smaller segments on each iteration, resulting in a distribution between the larger, flatter aspects of the plane—these being abstracted segments of terrain, such as mountains, valleys, and hills—and the more precise detail of smaller segments. This can be scaled easily to higher dimensions as well.



(a) Fractal Pattern. Source: [26]



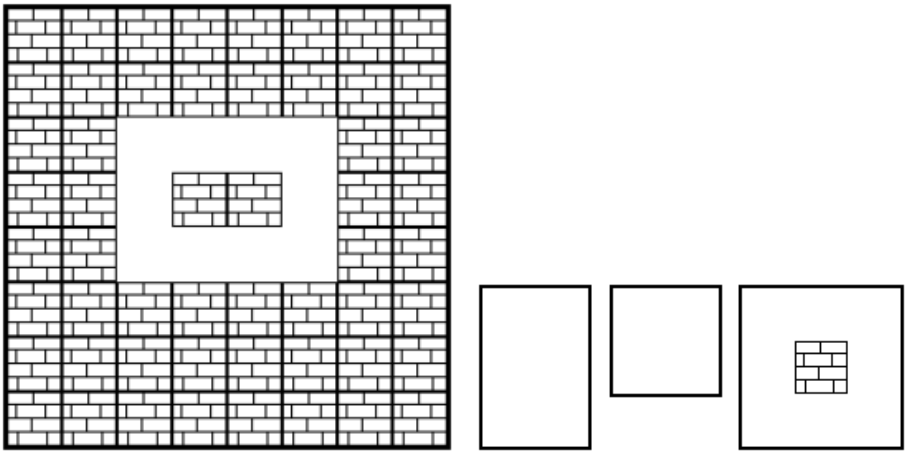
(b) Midpoint Displacement. Source: [28]

Figure 2.4: Terrain Generation

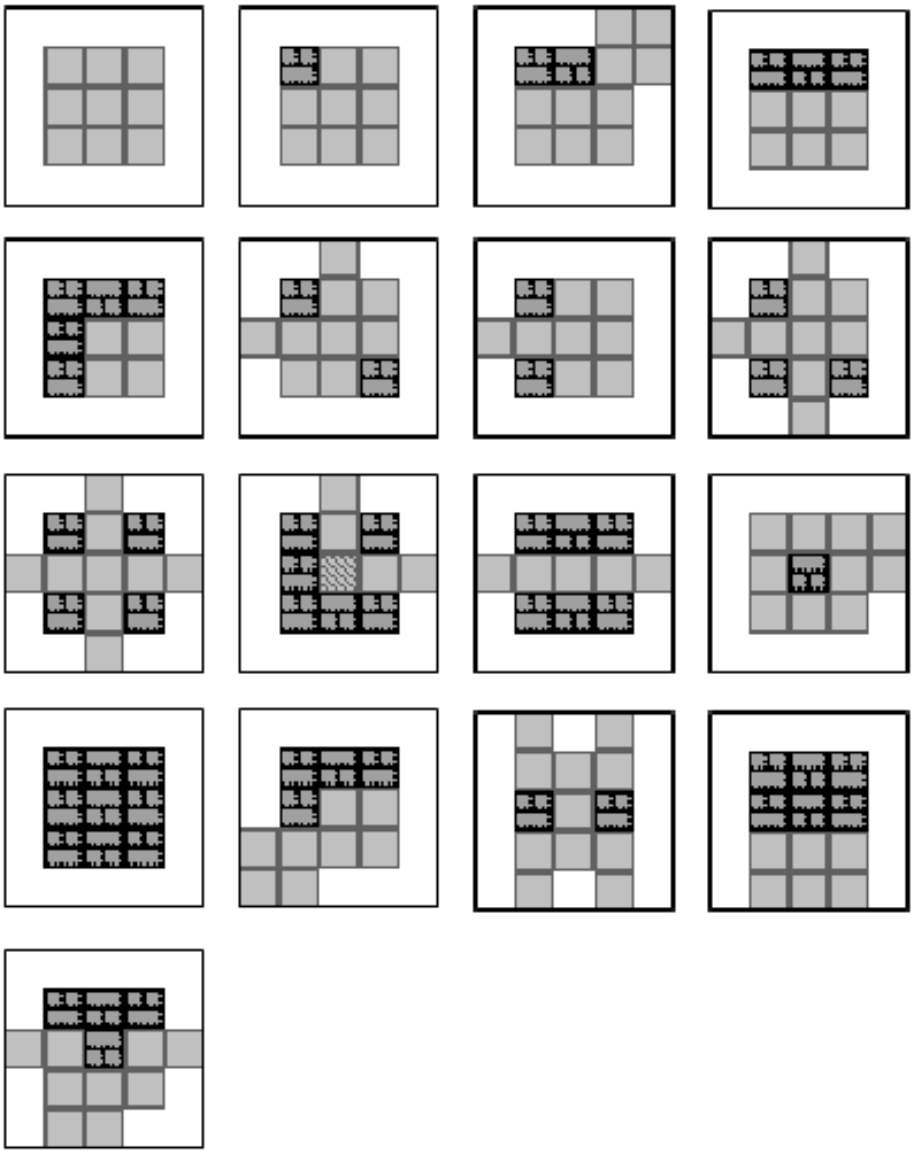
2.1.2 Sokoban PCG History

Sokoban generation is a topic with a long history. One of the first generation techniques for the game was introduced by Murase, Matsubara, and Hiraga in 1996 [18]. Here, instance generation adopted placement ideas of dungeon PCG which began with the placement of "templates" (a smaller grid of floor and wall tiles) on top of a grid composed entirely of walls. Templates are placed at random, overriding any of the original terrain of the larger grid. In doing so, over many iterations, an island of open space is constructed, ideally fitted with interesting obstacles and intersections. Working in conjunction with this is a solver, used to eliminate all unsolvable levels. Repeated over an extended period of time, eventually some composition of these basic templates would work to produce a level, which can then be fitted with the boxes, goals, and the keeper.

This implementation can be considered limited due to its heavy reliance on trial-and-error; though not an alien characteristic of PCG, this method lacks the supporting constraints prior to introducing the solver to determine whether or not a grid would be feasible for level usage; in an experiment of hundreds of trials, only a small handful could be considered interesting. Inspired by this is Taylor and Parberry's algorithm [19], which continues to make use of templates for the grid generation the empty room but through a more deterministic approach. Templates consist of interesting and varied orientations of floors, walls, and null tiles, each overlapping with the neighbouring templates, and fitted into the pre-divided grid (there exists a spot for each randomly chosen template). An approach like this will produce perhaps a more limited (though still seemingly infinite) series of grids, however these are more likely to contain interesting and solvable solutions upon the placement of the entities as the segments are designed to connect together. The two template styles are detailed in Figure 2.5.



(a) Prototype with templates. Source: [18]



(b) Template variations. Source: [19]

Figure 2.5: Empty Grid Building Comparison

2.1.3 Deadlock

Deadlocks are the crux of Sokoban's difficulty, it is a state of the game in which the level is no longer solvable regardless of the user's actions. The only way to complete the level would be to undo moves or restart the level entirely. This game state arises due to the player's limitation of being unable to pull a box. An example of this is pushing a box into a non-goal corner. One or more boxes will never reach their goal. Identifying deadlocks thus becomes a critical aspect of any refined solver of Sokoban, if done correctly, it is capable of pruning significant portions of the search tree early on. Deadlocks can come in the form of many permutations of the grid, thus the most suited approaches to correctly identifying these are ones which pre-compute results: deadlock tables and pattern databases [21].

As done for the Rolling Stone solver [8], deadlock tables store explicit deadlocking configurations for a fixed grid size (5x4). Every possible combination of walls, boxes, and keeper positions are enumerated and checked. If a permutation leads to a deadlock, it is stored. During search, sliding windows of the same size can be used to scan the grid for localised deadlock, and identify failed states of the game prematurely. Generating these tables is a very computationally taxing process, but once done, this brute-force approach can be used to quickly filter bad states during search.

Pattern databases (PDBs) help identify deadlocks by constructing the set of all reachable abstract states (i.e., a subset of the boxes) from abstract goal configurations. During search, a current state is mapped to its abstract form; if this abstract state is missing from the PDB, it indicates that no goal state is reachable, or in effect, the lower bound on the solution length is infinite [21]. Similar to the deadlock tables, by using PDBs to recognise deadlocked states, the search can immediately prune them, avoiding unnecessary computation and improving the chances of finding a solution.

2.1.4 Planning with Abstraction

"Abstract Sokoban" is a planning approach that has been adopted by Botea, Müller and Schaeffer [1] that simplifies the solving of Sokoban puzzles by breaking them into high-level components. The grid is divided into distinct rooms (open areas) and tunnels (narrow passageways connecting rooms), effectively turning the level into a graph of interconnected regions. Each room is treated as a "black box" with various internal configurations (arrangements of walls and entities). At the abstract level, planning actions become macro-moves; instead of single-step pushes, an action might move a box from one room to another through a tunnel in one macro-step. By operating on this coarse grid of rooms and tunnels, the planner works with a much smaller state space, focusing on moving boxes between regions rather than every individual interaction with a tile.

Solving levels is split into a multilayer approach, a "global planner" is used to determine which room a box should be moved into, and a "local planner", to handle the unabstracted moves of a box within a room which will feedback to the higher level transitions. A key aspect is that each room's internal moves are computed or analysed independently in a preprocessing phase, then reused whenever the global plan requires entering or exiting that room—ensuring in doing so, moving to that room will not result in deadlock. By dividing Sokoban into these sub-problems, the approach dramatically reduces the branching factor and search depth needed at the top level, since many low-level details are abstracted away; the overall difficulty no longer scales exponentially with the full level size, but with the largest room or the interactions between rooms. Whether or not abstraction benefits the solvability on the level depends on how much of the level space is being "used at once"; if the grid cannot be effectively divided into sub-rooms, the global planner becomes obsolete, and

search space remains as it did before abstraction.

2.1.5 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS), as put by Karman [10], is a search algorithm which "combines precision of tree search with the generality of random sampling techniques", typically known for its success in simulating games of Go and Chess. MCTS can help explore large design spaces of puzzle layouts by systematically sampling possible moves, simulating outcomes, and propagating the value of each decision up the search tree over thousands or potentially millions of iterations. Each round of MCTS consists of four steps:

- **Selection**: Viewing Sokoban as a stepwise problem, start from the root node (the initial state) and descend through the tree by choosing child nodes according to a policy like UCB (Upper Confidence Bound) until reaching a leaf node, or a node that has unvisited children.
- **Expansion**: If the leaf node has unvisited children (i.e., possible actions/decisions not yet explored, such as moving a box or placing a wall), select one and create a new child node in the tree.
- **Simulation**: From that new node, run a simulation (using a solver or heuristic) to check the validity of the move and measure the puzzle quality from the current partial design; examples of which are the number of moves required or puzzle complexity. The outcome of this is the "reward".
- **Backpropagation**: Propagate the reward back through the visited nodes, updating their statistics. This makes future selections more likely to choose promising branches. If these heuristics make more interesting puzzles score higher, the search tree will slowly converge to finding more of these.

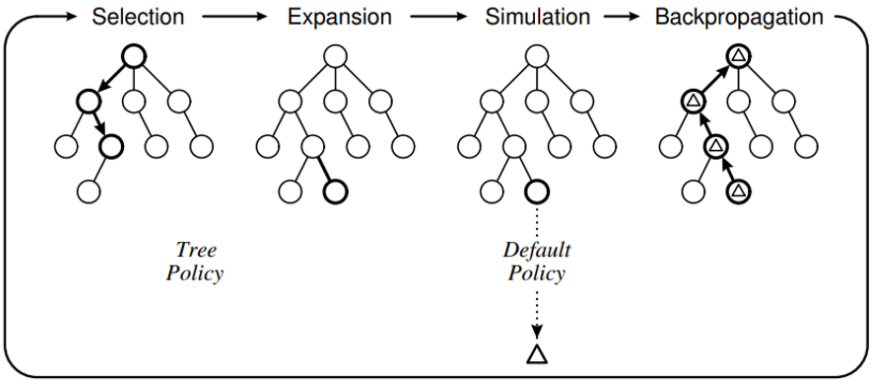


Figure 2.6: Monte-Carlo Tree Search. Source: [2]

2.1.6 Large Language Models

Perhaps the most seemingly obvious choice for procedural content generation is the use of generative artificial intelligence (AI). Where manual instance creation would require complete human intervention, these models are capable of mimicking human thought processes and tasks, and are able to do so on a much larger scale. In the same way standard techniques of generation can be checked

against a solver for valid layouts, so can AI-generated levels. The limitation here lies in the substantial dataset required to train these models (which are small and difficult to acquire), and more pressingly, training the model to identify patterns and structures of difficulty, and correctly translate these into functional, spatial constraints of good game levels [27]. Large language models can be suited to PCG of Sokoban, as instances can be represented by streams of characters representing different tiles and the dimensions of the grid. However, this introduces its own set of challenges; grids of the same size will not be represented with an equal number of characters, thus misrepresenting spatial relationships of the grid unless significant preprocessing is performed [27]. Overcoming these obstacles, there have been successful cases of instance generation, however not without a considerable computational commitment to the training, nor with the guarantee that levels are considered difficult.

2.1.7 Dead-end Detection

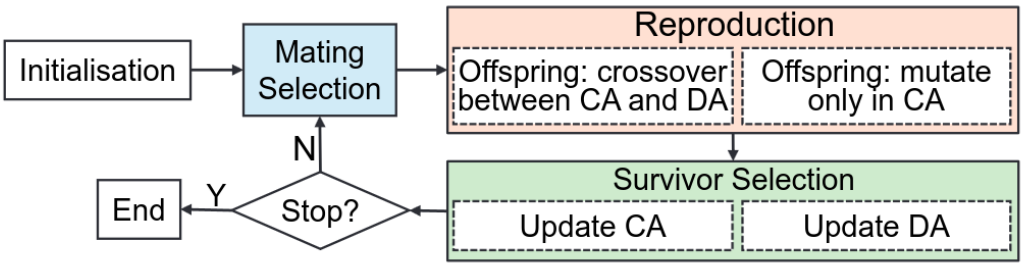
Dead-end detection has become an increasingly prevalent topic in the realm of instance generation. It is critical that levels remain interesting throughout their exploration, though a common by-product of PCG is the production of insignificant spacing, which has no impact on the level or the player(s). The reason this is so difficult to combat is due to the non-conformity of dead-ends to any particular layout; these rooms can be long narrow labyrinths spanning the full length of a level, one-block protrusions from an otherwise useful room, or the room itself entirely. With no consistent way to detect these, nor an exact definition, they become a unique challenge to solve.

In the context of Sokoban, a dead-end can be seen as an area of the grid which the player would not logically enter for the completion of the level (inherently, this means the area also contains no entities). In the simplest case, empty 1x1 tunnels can be quickly eliminated; Tarjan's bridge-finding algorithm [4] offers a graph based representation of the grid, in which bridges are the nodes on the graph such that their removal will disconnect the graph. Working from the outer most nodes of the graph, with only a single connection (from a non-dead-end), the algorithm can be called recursively, marking the nodes dependant on a single bridge. The appeal of this algorithm is its capable of marking all dead-end areas without mistakenly marking bridges which have a high betweenness-centrality. From here, it is up to the designer whether or not these are filled in or the level is scrapped entirely, dependant on the number of dead-ends. An alternative approach, taken by Le [13], adopts a two-array strategy, counting the incoming and outgoing nodes of each cell on the grid, and using a queue to dictate which dead-ends should be removed and in what order, ensuring no same point is covered twice.

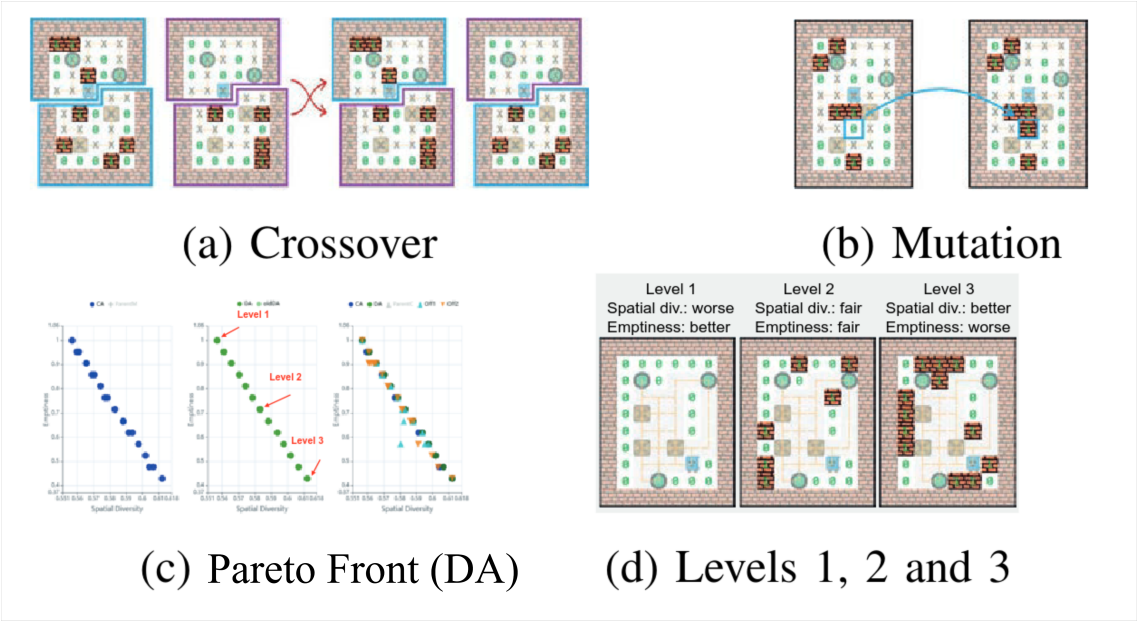
Dungeon-based environments greatly benefit from this method of dead-end detection, it allows for uninteresting areas of the grid to be populated or restructured to match the remainder of the level. This advantage of dead-end detection is not present in puzzle games such as Sokoban to the same extent for one critical reason—purpose. There is no way to determine the intention of the solver or the player completing the level; dead-ends can be used as redirection points, a common example of this would be temporarily moving a box into the dead-end to make way for another box to be moved to its goal, returning to it later. Understanding intention requires the level to be solved. It also requires many permutations of the solution to be found, as there does not exist a single way to solve any level (unless specifically designed to do so, which does not fall under PCG), and from there it requires these solutions to be examined to understand the significance of that dead-end, a computationally expensive process when done in bulk.

2.1.8 Evolutionary Algorithms

Evolutionary algorithms are one of many paths which can be taken in design optimisation. The purpose of these algorithms is to graft the best levels, and the best components of each level into a selection of superior levels, paralleling biological genetic modification. Zhang et al. [29] proposes an algorithm which populates two archives of levels based on different metrics. First, the Convergence Archive (CA) is populated and sorted with levels which balance two conflicting metrics on the Pareto Front, emptiness and diversity. Emptiness relates to how open the spaces are, the minimalism of the level, while diversity relates to the obstacles present. These are inversely related, optimal levels will find the correct balance between these metrics. The purpose of minimalism, or open space, is to introduce more options to the solver (out of the possible 4 moves) at every step, increasing the branching rate as much as possible, while diversity serves to be the effective difficulty of the level. The Divergence Archive (DA) is a collection of random levels with a high diversity score, this servers as a source of randomness to avoid local optima in the level design. Offspring instances are created by taking levels in the CA and DA and grafting them onto one another, ensuring grids still maintain solvability in the act, a process known as "mutation". The full theoretical series of operations is detailed in Figure 2.7.



(a) Flowchart



(b) Example Instances

Figure 2.7: Evolutionary Algorithm Pipeline. Source: [29]

CHAPTER 3

Requirements Specification

Having an open-ended project allowed me more flexibility in terms of what the final project was to look like. I considered many different forms of instance generation constraints, the kind of metrics being evaluated, and what would be considered a successful implementation, whether that would be testing against human or machine participants, and whether speed or difficulty would be the primary goal to optimise. The eventual main requirement for this project was to create a piece of software that autonomously outputs interesting levels which a solver, or a user, can interact with. This main idea was then divided into smaller requirements to inform the technical requirements for the project.

3.1 Objectives

The project will be done under an iterative development cycle. This will involve first establishing all the interactivity of the model in order for progress to be better visualised. From here, areas of complete randomness can be eliminated and replaced with more informed generation and evaluation.

3.1.1 Primary Objectives

- Analyse the structure of existing Sokoban levels to understand patterns, rules, difficulty progression, and solvability
- Establish a set of constraints and heuristics to guide instance generation
- Develop a baseline algorithm using constraint satisfaction to produce simple, solvable, and likely randomised levels
- Utilise a pre-defined solver to ensure generated instances are solvable
- Develop a GUI to visualise and interact with instances

3.1.2 Secondary Objectives

- Improve algorithm to rely less on absolute-random instance generation and more on stochastic, or pseudo-random, grid creation and entity placement
- Modify algorithm for constraint optimisation—create a set of weighted scoring metrics to evaluate the difficulty of the levels, including but not limited to:

  These metrics have been adapted, as discussed in the objectives evaluation (Section 6.1)

  - Number of tiles (instance size)
  - Number of crates/storage locations

  - Dead-ends, corners, tunnels, and areas of backtracking
  - Time and moves required by the predetermined solver
  - Alter existing solver to allow levels to be generated based on a specified difficulty

3.1.3 Tertiary Objectives

- Compare evaluation results against existing research
- Develop a Sokoban solver. As this can be approached as a graph problem, the solver will likely rely on one of four search algorithms: Breadth First Search, Depth First Search, Uniform Cost Search or A* Search

CHAPTER 4

Software Engineering Process

4.1 Approach

Building the artefact adopted an iterative test-driven-development (TDD) process. Based on the research at the inception of this project, and research throughout, a set of long terms goals have been established. This is broken down further into small, realistic goals for each weekly sprint where the work is implemented, tested, and the findings discussed with my supervisors. The result of this feedback will then determine the goals for the subsequent week(s). Choosing a TDD approach ensures that at each stage of development all code is functioning as expected (to the extent of all the edge cases tested), meaning mistakes can be caught early in the process as opposed to cascading failures later in development.

Sokoban PCG has been approached using a generate-and-test philosophy, this means that at every stage of the design, some aspect of the level is being constructed, through pseudo-random means, and if it fits a certain criteria it is kept and evaluated, otherwise it is discarded and generated anew. This approach is more computationally expensive than techniques displayed in dungeon generation but the result is a more interconnected level design.

A large breadth of approaches were available to procedurally generate Sokoban instances, found across many styles of games. However, I eventually settled on a layered approach, starting with the generation of the empty grid, ensuring it is capable of being manoeuvred around fully. This was followed by the placement of entities on to the grid; after discussions with my supervisor, this adopted a reverse-engineered approach, starting with the goals and working to the boxes. The outcome of this is a single instance which fits our requirements of being solvable and interesting; this is then repeated over extended periods of time, eventually selecting the "best" levels based on a range of scoring metrics.

4.2 Tools and Resources

A crucial aspect of software development are the tools chosen, most notably: the language, and the game's solver(s). These had to be picked in the beginning weeks of the project to ensure compatibility with all remaining tools.

4.2.1 Java

Java (JDK17) is a strong choice for PCG because it performs well in multiple key areas essential for generating large numbers of puzzles quickly and reliably. First, its platform independence makes it easy to run the same code when testing across multiple operating systems. Additionally, Java's Just-In-Time compilation can efficiently handle the intense computational demands of PCG, especially

when scaling up to hundreds of generated instances at a time and the object-oriented design facilitates modular, maintainable code. Java was chosen over other high level languages such as Python as performance matters here when attempting to produce as many instances as possible in a given time. Finally, the decision was reinforced by existing solver implementations in Java, ensuring straightforward integration within the PCG pipeline and minimising the overhead of rewriting or reconfiguring solver code.

4.2.2 JSoko - Solver

JSoko [17] is a Java based Sokoban player and solver, available under the GNU General Public License. This solver will be used to find optimal solutions of instances when assessing the generator's puzzles. JSoko also offers a large selection of handcrafted puzzles to compare against.

4.2.3 Marwes - Solver

Another Java based Sokoban solver by developer - Marwes [16]. This code is open-source, unlike JSoko, allowing the search heuristics to be analysed, and worked against by our generation algorithms. The Marwes solver and the generation will feed information into one another to influence the resulting instances.

4.2.4 Kenney - Graphics

Kenney's Sokoban game assets [11] are licensed under the Creative Commons CC0 license, placing them in the public domain. This will serve as the graphic assets for the GUI.

4.3 Ethics

There are no ethical considerations directly tied to the development of this artefact, as it neither involves sensitive personal data nor introduces risks related to privacy or security, and it does not have broader societal implications that might typically require ethical scrutiny.

CHAPTER 5

Design and Implementation

Design of the project can be broken down into the generation of grid, then its entities. Each valid level is then tested for solvability and simulated, gathering a range of metrics which are assessed against other instances, finding best puzzle for a given grid size and entity count.

5.1 Grid Generation

Almost as crucial as the entities themselves is the grid they are situated on. This aspect of the design adopts an approach introduced in earlier papers by the likes of Parberry and Taylor [19] which makes use of templates to build the grid section by section, in a similar manner to how puzzles pieces fit together. This process was adapted for my implementation as it possess an edge over other generation techniques in regard to difficulty through its added determinism. This design contains a selection of 17 templates of varied weightings which are chosen from at random; each puzzle piece is assigned a random seed from the set {0,1,2,3} to determine its orientation (0°, 90°, 180°, or 270°), ensuring both variety and reproducibility in the final arrangement. The element of determinism here is in regard to what the templates contain. Though being no larger than a 5x5 grid, templates are able to exhibit features that introduce areas of challenge to the solver. These range from being simple one- or two-block obstacles (blocks referring to walls on the grid) surrounded by floor tiles to more intricate designs involving corridors and intersections.

To increase the total number of grid outcomes the generator is able to produce, these 5x5 grids are placed in the centre of 3x3 divisions which the grid is comprised of. These means that templates can overlap, and override one another wherever floor or wall tiles appear in a template introducing many increased variations for a grid of the same size while still maintaining the intended difficulty the templates introduce. Figure 5.1 shows nine example templates, the grey squares denote null entries, these simply fill in the gaps and will not be placed onto the grid, the remaining squares, floors (white) and walls (black) will however. When choosing a grid size, the generator will take a random value in a pre-defined range which will determine the number of divisions in the rows and columns separately. Figure 5.2 is an example of a 6x6 grid (or 2x2 subdivisions), the red lines indicating their intersections, where the protruding tiles of one template can override another.

An alternative approach suggested in the literature proposes a more sporadic approach to template generation by Murase, Matsubara, and Hiraga [18]. This method begins with a grid composed entirely of walls, the templates, conversely made predominantly of floor tiles, are placed at random anywhere on the grid, with the intention of overlapping frequently. The end result of this will be a floor-plan with similarities to an island, with few obstacles within. The way this differs from the current implementation is through its lack of redundant space; the grid, if generated successfully, will likely be a single open area, with little added difficulty; this will come later, when placing the

entities. While aesthetically this appears to be a better solution, it does restrict the total possibilities for entity placement, and will likely miss more interesting solutions.

Following the placement of the templates in either implementation, is checking for connectivity. This step is carried out to ensure the player is capable of reaching all floor tiles present on the grid, if not the level is discarded and the process is repeated again. The step is done via a Breadth First Search (BFS) algorithm. An array, the same size of the grid is initialised all to false. The first floor tile of the grid is added to a queue. The process will repeat, removing an item of the queue, marking it as visited in the array, and appending its unvisited neighbouring floor tiles to the queue, thus processing the neighbours in order they are added to the queue. Once the queue has been emptied, if the number of visited floor tiles is not equal to the total floor tiles, the grid is invalid. Figure 5.3 illustrates the breakdown of this process in more detail.

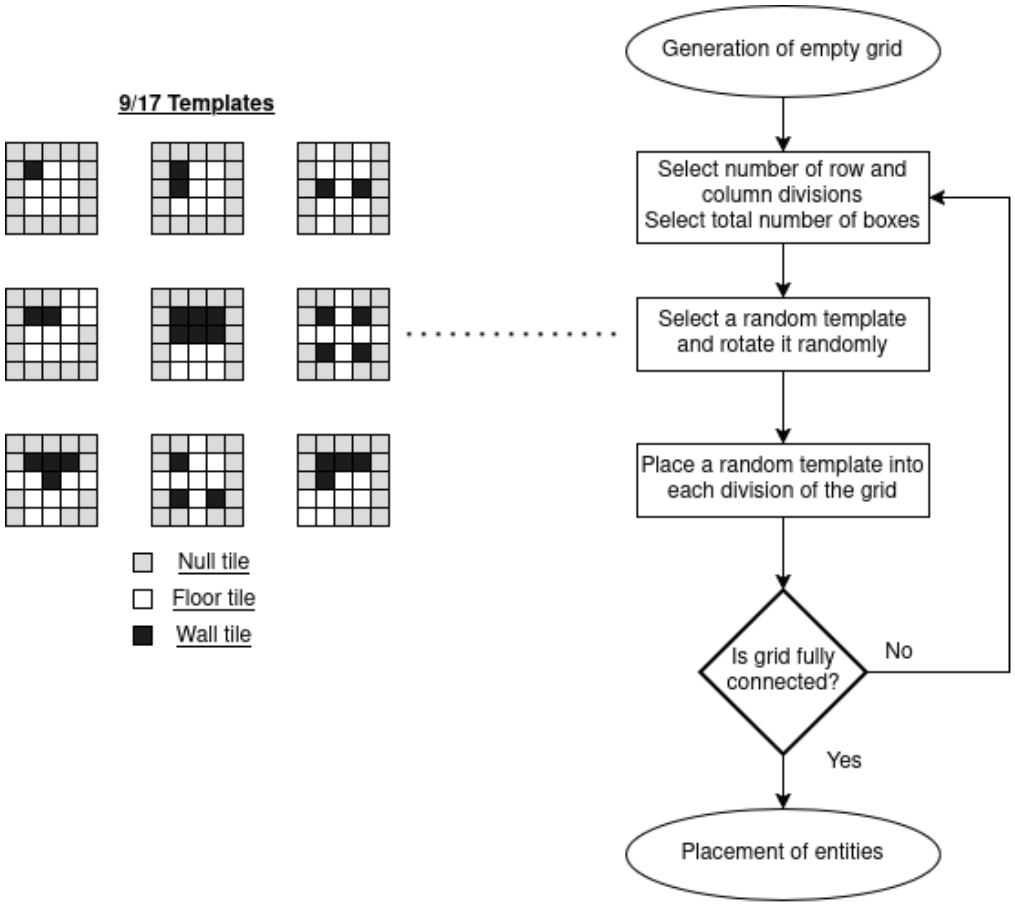


Figure 5.1: Template Placement Flowchart

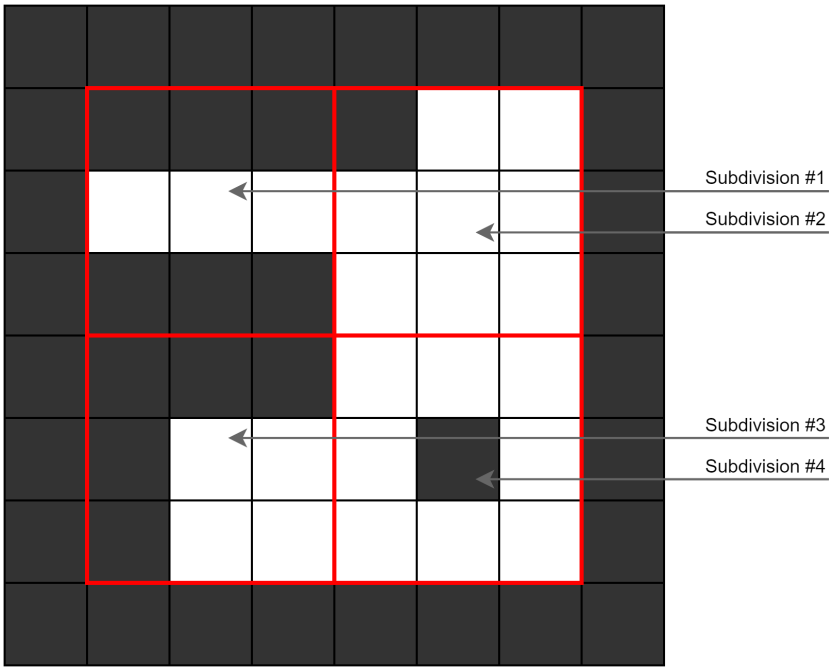


Figure 5.2: Grid Subdivisions

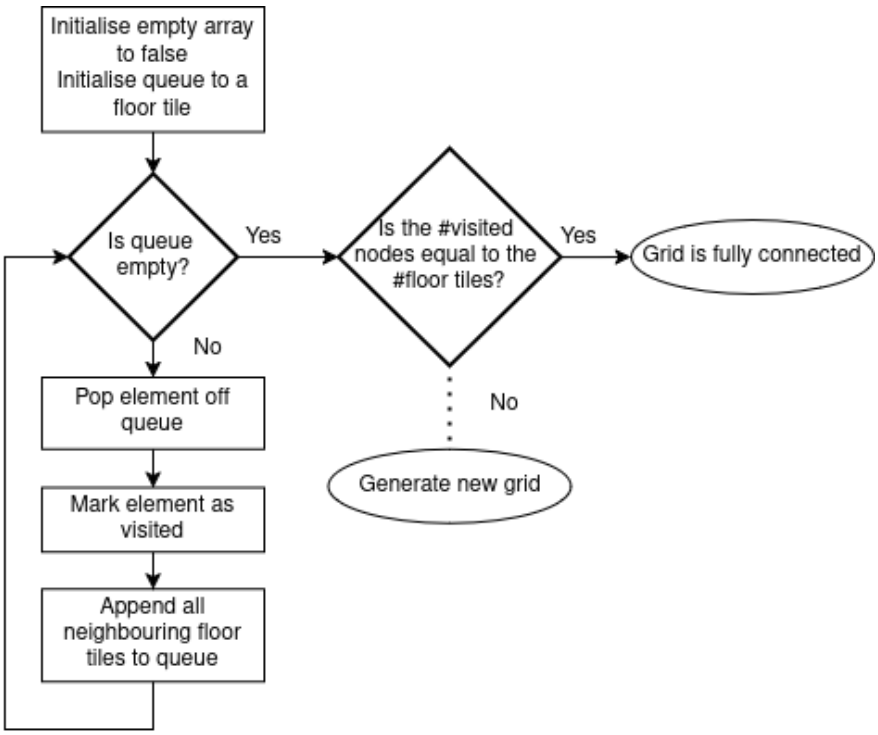


Figure 5.3: BFS Flowchart

5.2 Entity Generation

The next stage of generation is the placement of entities, these being the goals, boxes, and the keeper. Entity placement posses a unique challenge to procedural generation as it requires a considerably more constrained algorithm than that of the grid generation, as not only are valid positions required for the grid to be solvable, but there are many seemingly unobvious placements which can be make a level trivial, or conversely, impossible (this is not accounting for deadlock, Section 2.1.3).

A reverse-engineered approach was taken for this problem, that is, to begin with finding appropriate locations to place the goals, and subsequently place the boxes working outwards from the goals; this process of working backwards is known as a "Pull Strategy (Backward Direction Selection)" [18]. Between the placement of each pair of objects is a series of constraints, filtering the total possible locations of the box (as part of a box-goal pair). Lastly, a suitable position for the keeper is determined.

5.2.1 Definitions

Select terminology has been used to describe different nodes, or sets of nodes, which make up the grid:

- **Passway**: The total floor space, all the tiles the keeper can traverse
- **Access Point(s)**: The adjacent floor tiles to any particular point. These neighbours can be used to "access" a node
- **Access Range**: The set of points on the grid from which a box can reach a specific goal

5.2.2 Pull Strategy

Inspired by Murase, Matsubara, and Hiraga [18], the pull strategy first involves selecting appropriate locations for goals. This requires a single constraint—a goal must be "reachable", defined as a location on the board which both the player and a box can be moved into. Thus a reachable point has at least two traversable tiles protruding outwards in any given direction. Figure 5.4 details two board layouts with a single goal; assuming all tiles outside of the 4x4 section are floor tiles, goal (a) cannot be "reached" as there is insufficient space for the player and box, however goal (b) does not exhibit this same limitation, since the player can utilise the tiles extending left from the goal.

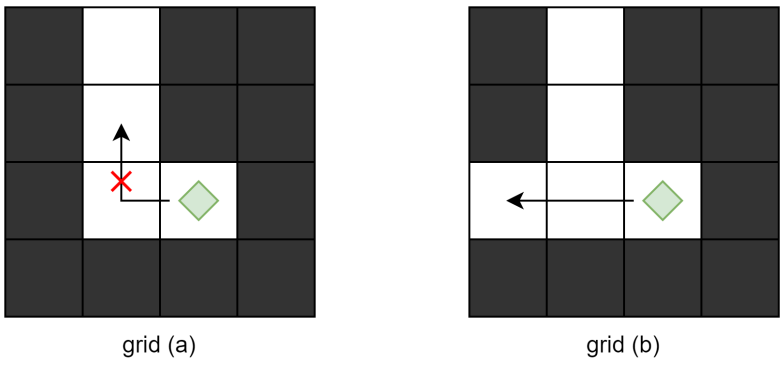


Figure 5.4: Reachable Goal Comparison

Finding all reachable goal locations involves scanning the passway and checking for reachability in four directions, for all tiles. The result of this is all locations on the board which could be accessed by an adjacent tile. Alone, no new insight is gained, however it provides a significant reduction in the total tiles to be evaluated for valid goal placement. Selecting one of these locations at random, the check is called recursively; on each iteration, the access points of the node are determined, beginning with the goal. This emulates a BFS, recursively populating a queue with access points, and filling an array with the same points, expanding outwards until the queue is empty. The result of this is the total available locations for a box to be placed which can reach that goal. Figure 5.5 demonstrates the complete BFS; on every iteration the access points become the next "goal", and its access points are found. The intuition is to treat every access point as if it is the target location for the box. Working backwards, this eventually expands out to all tiles which create a valid path to the original goal.

An important distinction to be made here is in regard to how dead-ends are defined. Previously (Section 2.1.7: Dead-end Detection), dead-ends did not confine to any specific structure or representation; this idea persists, however, in context of the pull strategy, a dead-end includes any point on the grid which a box cannot be pushed out of. This refers to the box's immediate position and not its position in regard to the passway. This definition can be more accurately stated as the corners on the grid—nodes with two adjacent wall tiles at 90° to one another—since these are guaranteed to be deadlock positions (if it is not a goal tile). When checking for reachability, the original constraint

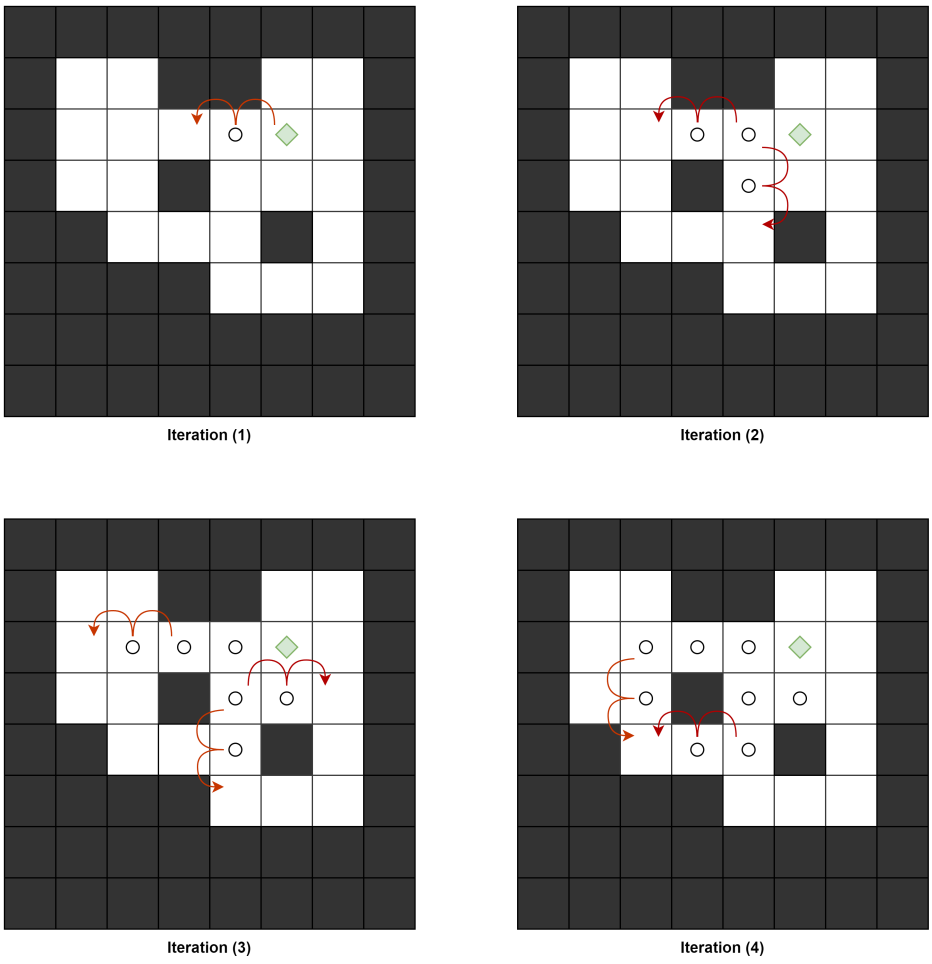


Figure 5.5: Finding Access Points. Adapted from [18]

required two tiles in any direction to be available—that is, the tiles are traversable tile and in bounds. This has since been updated to account for dead-ends. Now, the outer tile also cannot be a corner, to avoid a common edge-case shown in the same example of Figure 5.5. Iteration 4 introduces two new access points which do not check for corners. If either point were to be selected for the box, upon placing the keeper, there would only exist a single location (the corner) in which the keeper could be placed which would allow the boxes to be moved out of their starting locations. This heavy restriction on the keeper significantly reduces the complexity of the level by forcing the player/solver to begin with a fixed series of moves to push that box into a better location. Ignoring these nodes as potential box locations (ending at iteration 3) will introduce more choice, and result in a more interesting level.

5.2.3 Bijective Mapping

The cycle of selecting a goal tile at random, and finding the access points associated with it, is a process repeated from scratch for every goal-box pair. The reason for this is two-fold:

- **Each box is a new obstacle**: Placing a box on the grid obstructs part of the passway; reusing the same list of access points for placing the subsequent entity-pairs will result in many instances containing unsolvable orientations, this could be as simple as two boxes placed adjacent in deadlock, or something far more elaborate; in either case, this would place unnecessary reliance on the solver, for an otherwise avoidable problem. Thus on each iteration, the available goal locations are recalculated ($new\_goal\_tiles \subset total\_goal\_tiles$), and its access range is found anew.

- **Reduced choice**: A system enforcing a one-to-one mapping between each box and its goal location arises naturally from the pull strategy but is also an intended element of the game design. The intention here is that levels are produced in such a way that the there exists only one valid location for a box to be placed in order to complete the level. Though it seems this would produce more trivial levels, and not the inverse, due to limitations on the solver's choice, in practice, this limited choice can prove beneficial for this aspect of the design. The harder the end state of the game is to reach, the larger the search space becomes; limited choice results in harder levels if there exists fewer end states of the game. This is not to say that the levels produced will only have a singular end state, in many of the output levels, there exists multiple options for a singular goal location, though the total possibilities are reduced.

5.2.4 Placement Constraints

In order for the mapping to succeed, the supporting constraints must avoid encouraging trivial placements, for example, selecting a goal's adjacent tile to place a box would be considered a failure. At any point if the constraints leave no available box locations for a specific goal, it is removed from the list of total goal locations and another is selected; if no goal locations remain, level generation beings from scratch. The placement constraints are as follows:

- **Manhattan Distance**: At the most fundamental level, the initial constraint considers the Manhattan distance between a box and its corresponding goal. In the current implementation, this distance is set to a minimum threshold of three tiles, meaning a box can only be placed if it is positioned at least three tiles away from its goal. Though this number appears arbitrary, it eliminates many conditions of the grid, regardless of the size, in which the bijection is evident. This includes far denser layouts as well (many entities in a compact grid). A Manhattan distance of two or lower encourages no educated choices by the player/solver, as the distance is so low, it is unlikely to promote any interesting redirections or displacements.

- **Direct Line of Sight**: Not all cases of trivial mappings can be solved based on proximity alone, another potential case occurs when a box, of arbitrary distance from its goal, is placed along the same column or row of the grid with no obstruction in between. These locations are removed from the access range as solutions can be easily spotted. This condition primarily concerns heuristics which do not consider the nearest goal as the best choice (and humans as well), as they would be able to assess large stretches of the grid at once. Locating these box-goal pairs checks if the "Manhattan product" between the two points (i.e., the product of the absolute differences of their Cartesian coordinates) is 0, and if so examines the tiles in-between.

- **Farthest State Manhattan Distance**: Expanding on this first constraint is the idea of the farthest state, inspired by Taylor and Parberry [19]. The condition requires the Manhattan distance to not only be checked against its respective goal but against all goals, with the same minimum distance required. This condition aims to avoid common cases of a box being placed adjacent to a goal that is not its own, making it the new best contender for that box. Here, all potential box locations in the access range are iterated through, and the Manhattan distance to every goal currently on the board is calculated; if the furthest goal away does not exceed the minimum Manhattan distance for that box, it is removed from the access range. The remaining available positions are then sorted in descending order. Upon placing the box, it will randomly take from the first few available positions in the list, the furthest tile(s) away from any goal.

- **Overlapping Paths**: At first glance, this constraint appears to conflict with the other three; however, its intention is actually to increase grid density by placing entities closer together, thus promoting more intricate move sequences where the player must strategically reposition certain boxes to access others. The way this can be achieved without explicitly designing the levels to do so is through the elimination of box positions in an access range which do not overlap with any of the previously calculated access ranges.

  To achieve this, the combined access range is stored as a single set, and the new access range retains only elements within this combined set ($new\_accessl\_range \subseteq total\_access\_range$).

  Without this constraint an uncommon case exists in which an isolated box, and its goal, are placed in one area of grid, and the remaining entities in the other, leading to no interesting choices made when interacting with the lone box, as it possess no obstacles.

Figure 5.6 provides examples for each of these constraints in isolation; figures (a), (b), and (c) each detail the restricted positions for a box to be placed. Figure (c) in particular also indicates the optimal placements for a box following the sorting. Image (d) is an example of invalid goal placement, as there exists no overlap between the two access ranges.

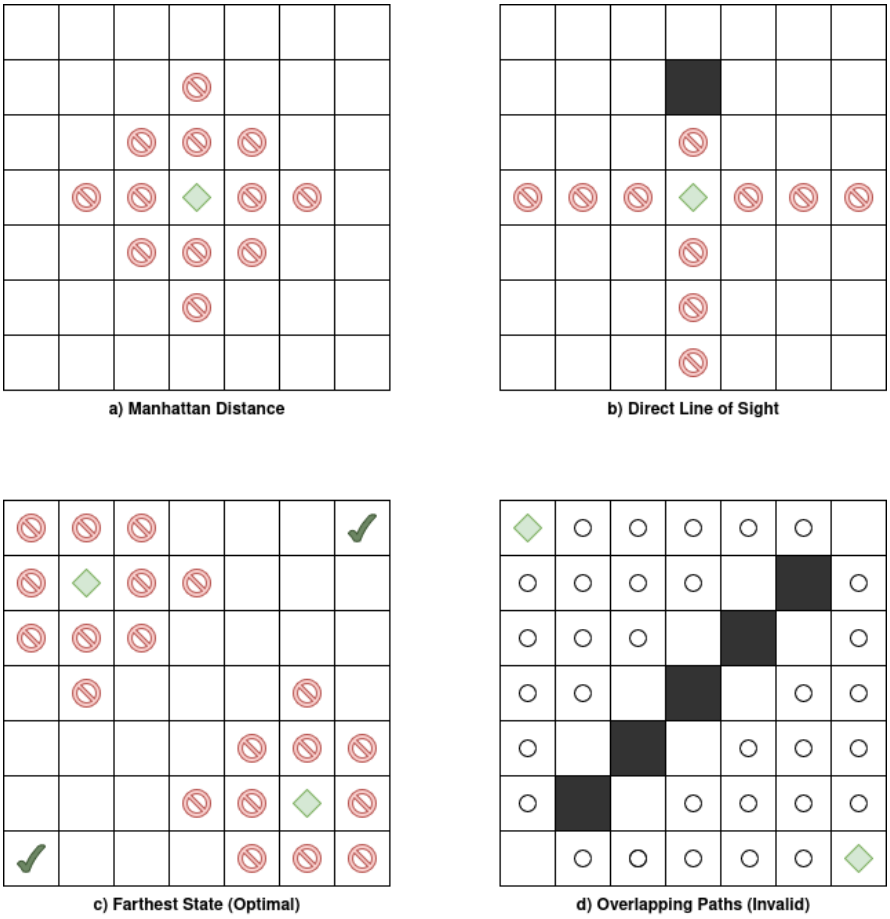


Figure 5.6: Isolated Placement Constraints

5.2.5 Keeper Placement

Perhaps the most difficult problem to give a concrete solution for is the constraints on keeper placement. This problem is far more open-ended than it seems; at first glance, placing the keeper anywhere on the passway is a valid rationale, however this introduces two problems of opposite concern: players can be put into a state of immediate deadlock, or alternatively, the first move(s) of the level are stripped of their choice, reducing the difficulty of the level by reducing the initial search space. Expanding on these in more detail, grid (a), of Figure 5.7, is an example of a deadlocked level, which would otherwise be solvable if the player was given a valid placement (the green location instead of the red). Conversely, grid (b) details another issue, the first move is guaranteed to be 'DOWN'. In more complex examples this would propagate into a search tree with significantly fewer nodes and provide an overall less interesting level. While this issue can be combatted through generating instances and filtering in bulk, it would be more beneficial to the generation to handle this issue proactively, rather than miss potentially good instances.

Initially, I considered adapting the pull strategy, rather than working from goals to boxes, it would work outwards from boxes to the keeper. Through testing it became clear this was not a viable strategy for the same reasons as before—it tied the keeper to a single box, making it less interesting, as this dictates the order of operations, and it had no guarantee of being viable. Referring to grid (a) again, a 'valid' placement here could be atop the rightmost goal, which would be immediate deadlock. The subsequent strategies all resulted in different variations of this same problem, ultimately it

was concluded that the keepers correct placement could not exactly be determined valid until after the solution was checked using a solver.

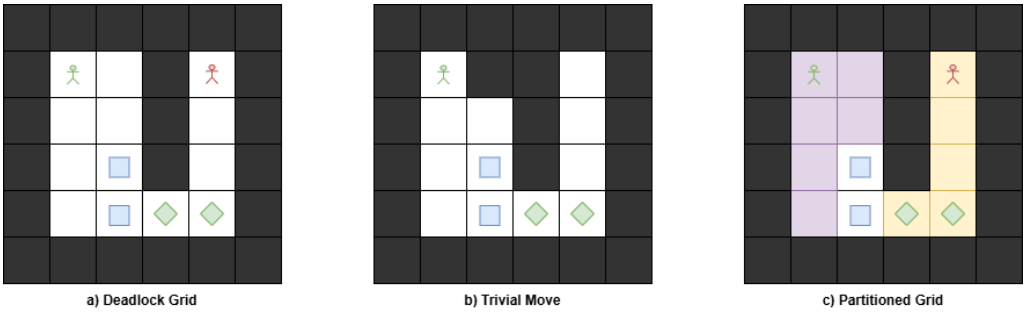


Figure 5.7: Keeper Placement Grids

However, this should not be considered a failed endeavour. While finding an exact optimal location cannot be determined, eliminating invalid locations in a computationally inexpensive way is possible. Proceeding this, all locations on the grid would no longer result in immediate deadlock, and uninteresting locations can be filtered out with ease. This relies on a technique known as 'oil spillage segmentation'. Similar to the BFS used in earlier aspects of the algorithm, a random point on the grid is selected and propagated outwards until the queue is empty. In this case, we are expecting the passway to not be entirely connected (due to boxes), and so the algorithm will persist until the grid has been fully partitioned into 'oil spills'. For the keeper, this means that any floor tile in that section of the grid is accessible, and thus can be selected at random for solvability purposes. This takes a problem which conceptually required running the solver on each tile of the passway, to a problem which would require a maximum of a few run—between 2-3 from testing. grid (c) (Figure 5.7) details the final partitioned grid; finding a valid keeper placement would require two uses of the solver, once in the yellow section (invalid) and once in the purple (valid). As stated previously, 'uninteresting' locations are prioritised last, thus the keeper position with more available starting moves (purple) would be checked before alternate position, meaning this problem has been reduced to a single use of the solver.

5.3 Evolutionary Algorithms

At this stage of the design, solvable and constrained levels are capable being generated autonomously. In theory, evolutionary algorithms (Section 2.1.8), appear as the next most optimal step in raising difficulty—capable of extracting the best aspects of levels in the convergence archive (CA) and mutating them with a selection of diverse levels in the Divergence Archive (DA)—resulting in levels which take the balanced density and minimalism of the CA while avoiding local optima using the DA. This style of algorithm mimics genetic modification of organisms found in biology, going through iterations of mating selection, crossover, and mutation until the archives are exhausted of possibilities and the optimal solution is found. In practice, procedurally evolving levels encounters a considerable number of issues due to the lack of explicit characteristics to favour, and to excise.

Defining which section of a grid is worthy of crossover requires extensive knowledge of how the grid is utilised by the solver. This could be achieved using a heat-map, the darker shades indicating the player traversing those areas more. However, complex solutions of the game could take hundreds of moves, and will likely redden significant portions of the heat-map, making it unable to accurately distinguish what part of the level constitutes the 'most interesting'. If this issue were ignored however, and the perimeter of an area be chosen, it would be impossible to know exactly how this grafted

part of the grid would interact with its new surroundings; it could be that the original solution required large detours spanning the passway, is now blocked off by an obstacle, or more critically, the level contains no tile-space to detour in. Sokoban problems also commonly require "sequential" solutions. Many of the subgoals interact, making it difficult to divide the problem into independent subgoals [9], and move them over to a new instance. None of these situations are uncommon, and will occur on almost every attempt at this algorithm. This is because levels are intentionally designed to be dense, interactive, and to provide some sort of utility in every corner of the grid; introducing new untested environments only breaks the rigorous constraining and evaluation each level initially went through, thus regressing rather than the opposite.

This only covers the crossover stage of the algorithm. New areas of difficulty arise when considering mutation, the process of eliminating obstacles in order to make otherwise unsolvable levels now solvable. Purpose is again the underlying issue, unless performed manually, it would be impossible to know which obstacles are necessary to remove. Blockades could simply be localised to an area of the grid, which would require the keeper to be redirected rather than the level to be considered unsolvable. If performing the same techniques found in the keeper placement (Section 5.2.5)—identifying isolated segments of the grid, to then connect—it now has to be considered whether that connection should be able to fit the keeper alone, or the keeper and a box. Any combination of these solutions could trivialise the level. Ultimately, this strategy exhibits too many failures to consider implementing it into the existing algorithm. Strategies such as efficient scoring and competition based selection have been utilised instead.

5.4 Solver Analysis

When autonomously producing levels, the type of solver and its complexity matter considerably, as this will ultimately dictate how difficult the level is. As this artefact makes no use of human participants, the solver becomes our ground truth and thus must be tailored towards when assessing the challenge a level presents. The evaluation of instances as part of the overarching algorithm for generation relies on a lightweight solver (Marwes), one which I can extract the exact details of solutions from, specifically the moves of the keeper.

The heuristics discussed thus far give insight as to why they would be useful in the general sense of generating hard levels. However, their development was also done intentionally, as to also exploit heuristics of the solver.

Marwes' heuristic strategy is a mix of distance-based guidance and goal completion incentives; it evaluates a state's quality largely by the Manhattan distances of boxes to goals, favouring states that appear closer to completion. This is achieved by summing the distance from each box to its nearest goal (distance in grid steps, ignoring obstacles)—the closer a box is to the goal, the less negative its scoring—it is a greedy estimate. This is the first area of potential exploit. Two constraints developed as part of entity placement aim to combat this exact reward system. By having the boxes take the "farthest point" from all goals, the Manhattan distance is directly acted against. This constraint takes the furthest point from any goal on the grid when deciding where to place a box, ideally resulting in a highly negative scoring for the initial state. Additionally, removing box placements from the direct line of sight of goals means that in the majority of cases the solver will underestimate the true effort to get a box to a goal.

Another way in which the solver is worked against is through the very nature of greedy algorithms.

As boxes and goals are placed as part of a bijective mapping—such that each box is intended to be put into a specific goal—a common occurrence is the corresponding goal not being the closest one to a box. The solver will push each box toward the closer goal, possibly locking them in a suboptimal arrangement, only to later discover it doesn't lead to a solution. This forces the solver to redo work assigning boxes to alternate goals.

The solver uses BFS to determine whether the keeper can reach a push position for a given box. Knowing this, the search space is deliberately made as large as possible; techniques as early as the empty grid generation aim to maximise this by giving a higher weighting to more open templates (which prioritise floor tiles) as to increase the likelihood of more minimalist levels. The solver re-runs the BFS every time the keeper needs to reach a slightly different push position even if the majority of the reachable area remains unchanged. This is done as opposed to caching partial results, thus the inefficiency is taken advantage of.

This speaks on a common issue amongst heuristics. They are uninformed about Sokoban's complexities and do not understand when a seemingly good move, such as pushing a box into a corner, is actually deadlock, or when a momentary setback is necessary for long-term success. Without specialised heuristics—such as utilising pattern databases to store solution costs to certain sub-problems of a larger puzzle [21], or deadlock tables to prune impossible states earlier [17])—the solver faces the fundamental struggles of Sokoban's search space. The game has an enormous branching factor and depth, making exhaustive search infeasible [9]. This solver's approach, while guided, can still suffer from combinatorial explosion on larger or harder puzzles.

5.5 Scoring Instances

In order to produce truly difficult and interesting levels, it must be approached as an optimisation problem. Procedural instance generation can be performed in bulk, extracting a range of insightful metrics as to give that instance a final normalised score. Repeating this process over extended periods of time results in the level which finds the optimal balance of these metrics (some beneficial, and others unfavourable) for that set of levels. Scoring metrics have been inspired by the findings of Taylor and Parberry [19] and Jarušek and Pelánek [6], these range in complexity and implications for the difficulty. Unlike the constraints for entity placement, scoring metrics cannot tailor towards specific constrained ideas of difficulty; they must be far more generalisable to all levels of Sokoban, as to not encourage any one style of level over another. Upon selection, determining which of these was more significant required evaluations of each metric in isolation, from there appropriate coefficients can be assigned relative to their significance. Below are the initial assumptions for each metric:

Solver Metrics

- **Moves** (Figure 5.8): The most intuitive metric to assess the difficulty of a level is based solely on the total number of moves to reach the goal state, this metric is extracted directly from the solver. The basic intuition is that more moves would generally indicate a higher complexity, since more steps are needed to arrange boxes correctly. This intuitively corresponds to a harder puzzle. A very high number of moves may also suggest intricate interactions between boxes and obstacles, meaning that multiple strategic considerations are required. The issue with this metric is it does not take into account all kinds of levels which result in high numbers of moves if prioritised as the main metric. Difficult levels would only be a small subset of the total, with the majority taking the form of long labyrinths, comprised of unintuitive decisions which

merely take long to complete. An example of this is a large grid with boxes and goals split into two corners of the grid, requiring many trips between the two locations.

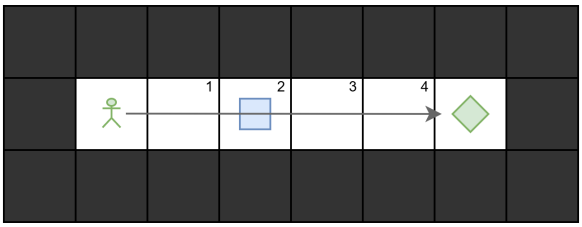


Figure 5.8: Moves Count (=4)

- **Pushes** (Figure 5.9): The subsequent keeper metrics are all derived from simulations of solutions. The first abstraction away from the moves is the number of box pushes. Following the same intuition as the moves, a level with a high number of pushes can be the product of moving boxes around the grid in interesting ways as to make room for others. This however attracts the same style of levels as before though is still an important consideration.

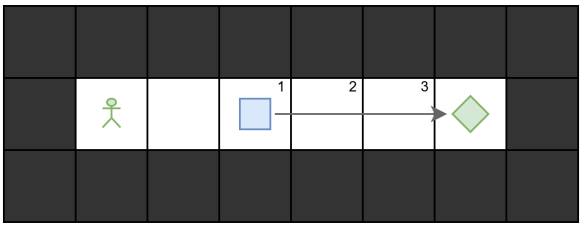


Figure 5.9: Push Count (=3)

- **Directional Pushes** (Figure 5.10): To combat the common occurrence of long unintuitive levels, directional pushes aim to reduce all consecutive moves in a single direction to one; a level solution comprised of many moves from end to end will thus score poorly if that is all the player is required to do. Levels which score better here will need the keeper to push a box from multiple angles, which can help encourage more interesting sets of moves.

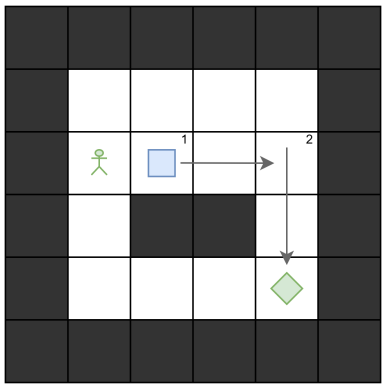


Figure 5.10: Directional Push Count (=2)

- **Reverse Pushes** (Figure 5.11): Assessing the mapping of boxes to goals, this metric counts the total pushes which are in the opposite direction of the goal (in both planes). Sokoban often requires moves that momentarily undo progress or move a box away from its goal to eventually solve the puzzle. The number of such "counter-intuitive" steps (moves that go against a greedy search heuristic) captures this exact philosophy. My first intention with this metric was that it would be an extension of the directional pushes (i.e., the total reverse-directional pushes) as this was a more insightful metric. However, situations that require the box to be pushed towards the goal (correct direction), and past it (incorrect direction) in a single directional push, will skew the calculations, and same for the reverse scenario.

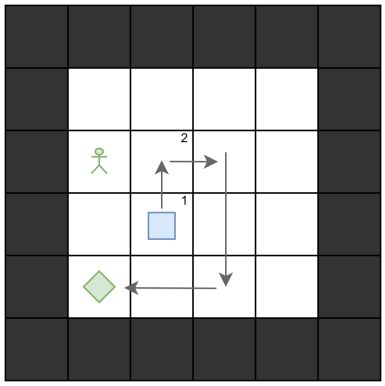


Figure 5.11: Reverse Push Count (=2)

- **Box Changes** (Figure 5.12): Perhaps the most complex metric to calculate is the number of changes between boxes, this is also seemingly the most valuable metric. Optimising the number of switches from pushing one box to pushing another directly correlates to the most widely supported ideas of a difficultly in Sokoban. Counting these changes is synonymous with the concept of puzzle decomposability, as put by Jarušek and Pelánek [6]. A completely decomposable puzzle, one that can be broken down into sub-problems of one or two boxes, would have low switch frequency and be more straightforward. Difficulty grows when the puzzle is not decomposable, requiring a greater number of intertwined moves. The study showed puzzles that forced frequent alternations between box groups were significantly harder (correlating even higher with human difficulty).

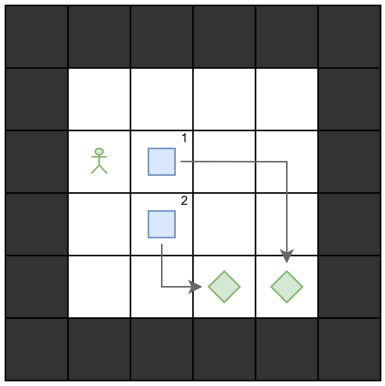


Figure 5.12: Box Change Count (=1)

Entity Metrics

- **Box-Grid Ratio**: This metric acts against the overall score of the level; a level is penalised less if there is a high number of boxes with respect to the grid size. This encourages denser layouts, and as the number of boxes on a grid is restricted to a random range (as is the rows of the grid), this metric will neither be too favourable nor too detrimental to the level score, though the resulting levels will be more tightly packed, which works well in conjunction with other metrics, such as box changes.
- **Corner Goals**: Another penalising metric is the total number of corner goals. Placing a box in a corner, whether it is due to deadlock or to reach a goal, restricts all potential future utility of that box, and thus prematurely prunes parts of the search tree, making the level "easier".
- **RNG**: The final metric is a small positive random value between [0.00 - 0.05] as to increase the variety of levels being favoured.

Each absolute metric which contributes towards the total score is normalised to a range of [0.00 - 1.00], this is done to standardise results across all sizes of grids, and number of entities. Each metric behaves as subset of other characteristics of the grid. Beginning with the number of pushes, this is taken as a fraction of the total moves, which provides the necessary upper-bound on the range the pushes can take. The formula utilised is simplistic but provides a sufficient result for the characteristic: $(metric\_count - lower\_bound)/(upper\_bound - lower\_bound)$. This upper-bound applies the same to the total box changes, and to the number of (total and reverse) directional pushes. Similarly, the number of corner goals is a subset of the total goals on the grid, and the boxes count is a fraction of the grid size. Each of these metrics had their weightings adjust through coefficients which were manually tweaked when assessing importance which was determined as part of a much larger evaluation of metric performance in isolation.

The sum of these provided another un-normalised score which underwent the same calculation for a final result. This approach was taken for normalisation as opposed to more conventional techniques such as $(score - mean)/(standard\_deviation)$ due to a lack of fixed (theoretical or historical) reference results to work against. Conventional methods such as this also centre the data around 0, which can be useful if wanting to interpret whether something is "above average" (positive) or "below average" (negative) by how many standard deviations it is from the mean. However, this difficulty scale is more linear, levels which approach 1 have reached the optimum for each metric, and conversely approach 0 when metrics are at their worst. Using the normalised values, the rating (un-normalised) can be expressed as:

$$\begin{aligned} \text{rating} = {} & a \cdot \text{moves} \\ & + b \cdot \text{pushes} \\ & + c \cdot \text{directional\_pushes} \\ & + d \cdot \text{reverse\_pushes} \\ & + e \cdot \text{box\_changes} \\ & - f \cdot \text{box\_grid\_ratio} \\ & - g \cdot \text{corner\_goals} \\ & + \text{Random float} \end{aligned}$$

Over an extended period of time, levels are generated, simulated, and evaluated against this scoring technique, only keeping the instance with the highest rating for that set of levels.

CHAPTER 6

Evaluation and Critical Appraisal

6.1 Objectives Evaluation

The objectives defined in the DOER set out the goals and progression of the project. The extent to which these were met are evaluated below:

6.1.1 Primary Objectives

- Analyse the structure of existing Sokoban levels to understand patterns, rules, difficulty progression, and solvability
- Develop a GUI to visualise and interact with instances
- Develop a baseline algorithm using constraint satisfaction to produce simple, solvable, and likely randomised levels
- Establish a set of constraints and heuristics to guide instance generation
- Utilise a pre-defined solver to ensure generated instances are solvable

These objectives have been met to the full extent; they acted as the foundation on which all other work would be built off. The GUI was the first sub-goal to be achieved, allowing all future work to be clearly visualised and interacted with. Grid generation then followed, starting with entirely random techniques and progressing towards higher stochastic techniques, through template selection and experimentation with different constraints, primarily grid connectivity (BFS). At this stage entity placement was entirely random, and though infrequent (commonly the solver would timeout), solvable levels could still be generated, thus allowing a solver to be researched and integrated into the generation pipeline.

6.1.2 Secondary Objectives

- Improve algorithm to rely less on absolute-random instance generation and more on stochastic, or pseudo-random, grid creation and entity placement
- Modify algorithm for constraint optimisation—create a set of weighted scoring metrics to evaluate the difficulty of the levels, including but not limited to:
  - Number of tiles (instance size)
  - Number of crates/storage locations
  - Dead-ends, corners, tunnels, and areas of backtracking

  - Time and moves required by the predetermined solver
  - Alter existing solver to allow levels to be generated based on a specified difficulty

These objectives have also been met to the full extent, though with modifications as the project progressed; goals were redefined to better suit the PCG. Stochastic generation has been successfully achieved in the full regard of the game. Each level generated is guaranteed to be solvable if all the constraints are "passed". This is excluding the keeper placement, which will require some utilisation of the solver (as discussed in Section 5.2.5). This process is near instantaneous, and any unsolvable or uninteresting instances, are discarded as early on in the generation as possible.

With solvable and interesting instances able to be generated, they were then assessed against one another. Each level's solution is simulated through, tallying a range of metrics in the process. The evaluation of results (Section 6.3) details the reasoning behind the weightings of each metric. Of the original list of proposed metrics, for differing reasons, each has been refactored or replaced entirely:

- **Number of tiles (instance size)**: Replaced with the box-grid ratio; the size of the instance still matters when scoring difficulty, but as an assessment of the level's density, not just total space.
- **Number of crates/storage locations**: Merged with the first metric.
- **Dead-ends, corners, tunnels, and areas of backtracking**: Due to their lack of uniformity, dead-ends could not be characteristic as a level metric. Instead, the definition has been redefined for the context of Sokoban, and integrated into the entity placement constraints. Corner (goals) remained as a metric, as did areas of backtracking in the form of the reverse-pushes metric. Finally, counting tunnels lacked significant implications for difficulty, and so was removed.
- **Time and moves required by the predetermined solver**: Remained, the solution's moves is the overarching upper bound to most other metrics.
- **Alter existing solver to allow levels to be generated based on a specified difficulty**: Remained, the "specified difficulty" refers to density. The size of the grid, and number of entities can both be adjusted to reflect this. This however, does not require the solver to be modified.

6.1.3 Tertiary Objectives

- Compare evaluation results against existing research
- Develop a Sokoban solver. As this can be approached as a graph problem, the solver will likely rely on one of four search algorithms: Breadth First Search, Depth First Search, Uniform Cost Search or A* Search

Tertiary objectives acted as an extension to the original project. While I did compare by results extensively to existing work throughout the project, I choose to forgo building a solver, and instead, focus on PCG, as this would be a big undertaking, and not allow me to refine the generator to the extent I would like.

6.2 Generation Evaluation

6.2.1 Density Evaluation

This section of the evaluation aims to identify the limits of instance generation by focusing on puzzle density, a key factor influencing puzzle difficulty, especially given its importance in explaining the unique challenge presented by handcrafted levels. It explores both the range of grid dimensions the system can reliably handle and the maximum achievable density of entities.

Beginning with the smallest feasible size allowed without manual tuning, a 3x3 grid. Naturally, the only valid placement for a goal could be in a corner, and the box must be in the immediate neighbouring tiles, so we ignore this. Figure 6.1 details the upper limits of generation with respect to density. Slowly incrementing the grid size and box count, it becomes clear how tightly packed the boxes can be placed together before the constraints of entity placement interfere. The trend appears to be mainly proportional, extrapolating this graph will detail a similar trend for larger grid sizes and box counts. The reason grids of sizes 6x6 to 7x7 show no increases is entirely due to luck—pseudo-random placement of walls can either favour, or make generation more difficult, depending on the templates selected. The 7x7 grid, if given more time, would likely be able to score higher. Density appears to be the core contributor to time delays (or potential timeout) when generating a grid; larger grids below this limit exhibit no struggle instantaneously producing puzzles.

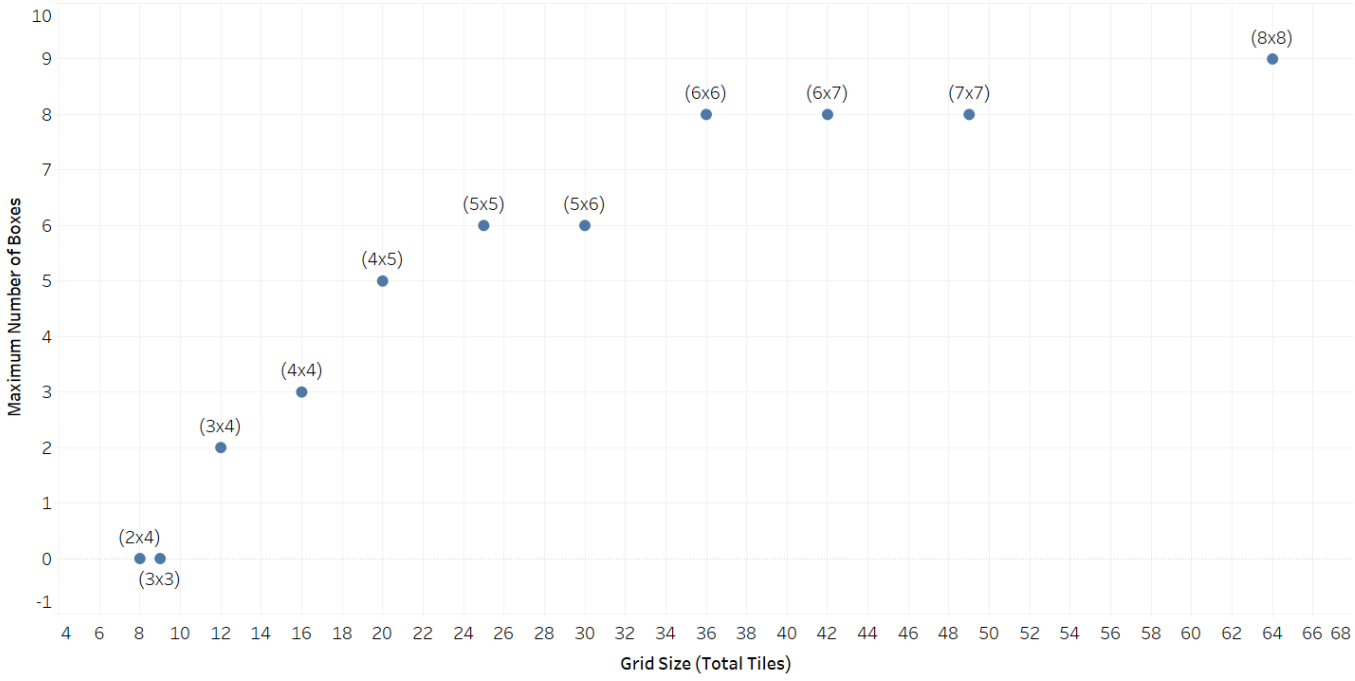


Figure 6.1: Density Limits for Different Grid Sizes (Plot)

A common trend across the most tightly packed levels is the boxes following a U-shape, as seen in Figure 6.2. This is simply a by-product of the constraints paired with the small grid sizes; in order for a box to satisfy the Manhattan Distance(s), the box is overwhelmingly likely to appear in the centre of the grid. The more boxes which follow this placement, the more their access ranges will be localised to the centre of the grid, encouraging future boxes to also be placed nearby. If the density were to be reduced slightly below this threshold, this pattern does not hold as consistently, and placements appear more random. Examples of which are shown in Figure 6.3.

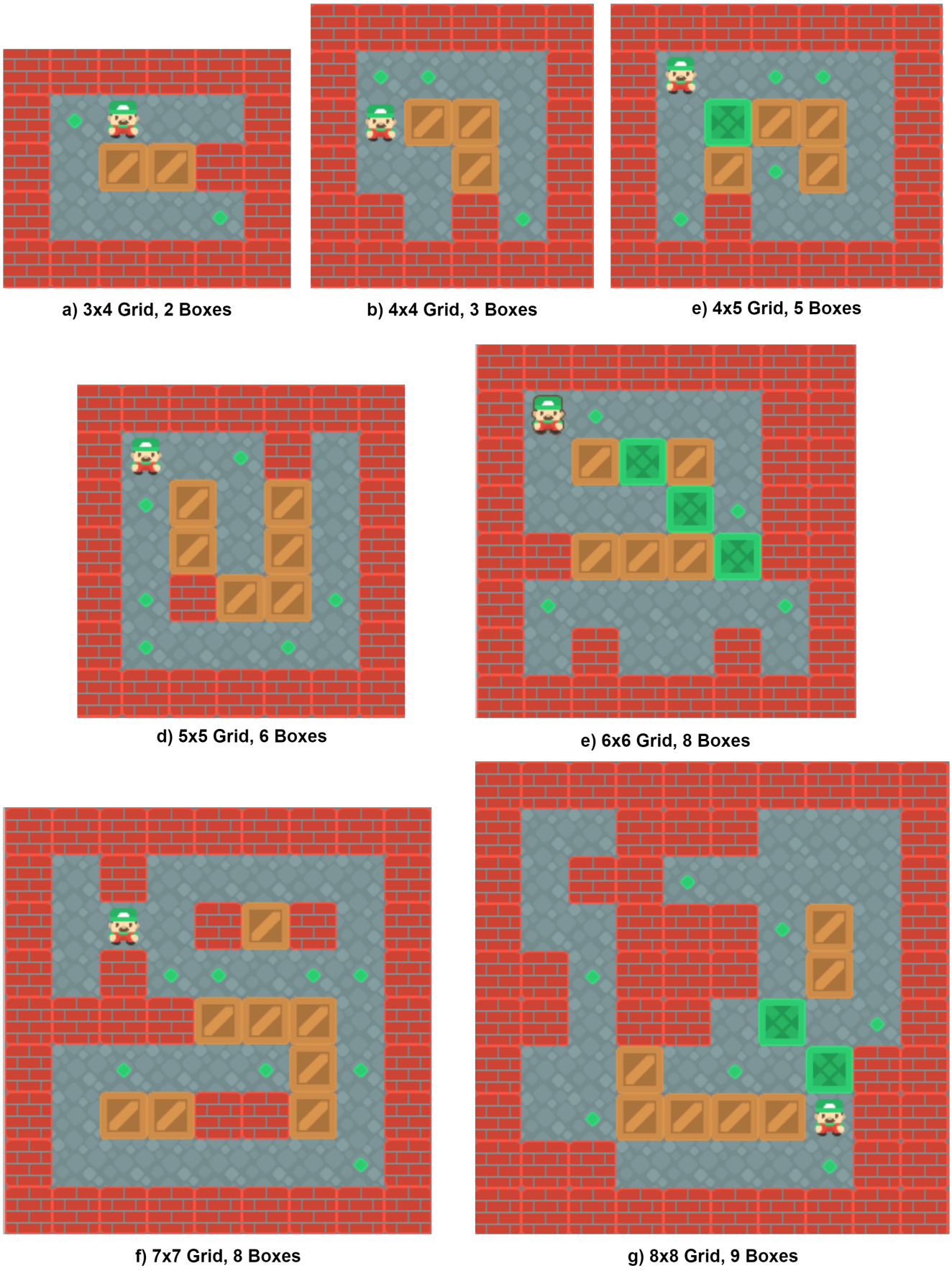


Figure 6.2: Density Limits for Different Grid Sizes (Instances)

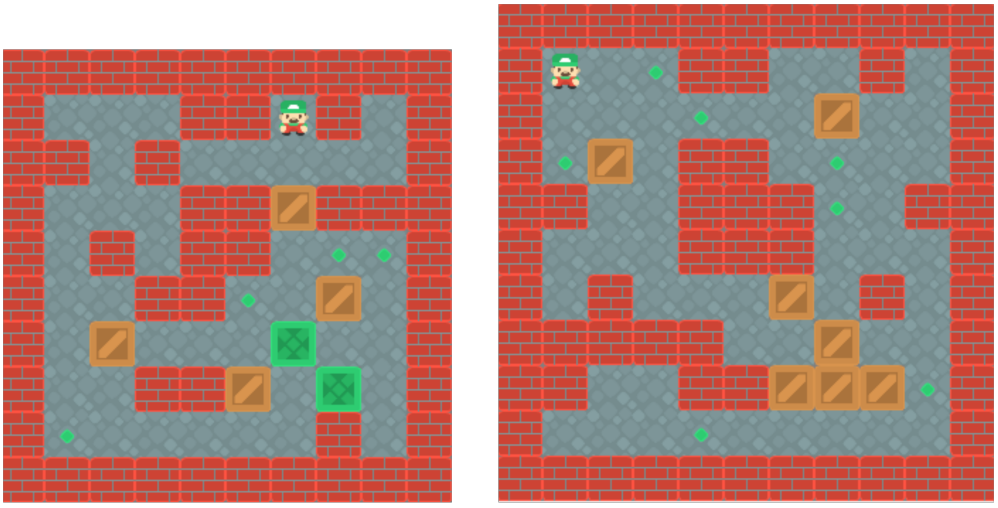


Figure 6.3: Below Density Limits for 8x8 and 9x9 Grids

6.2.2 Diversity Evaluation

Although density is a significant consideration in procedural content generation, diversity (both inter- and intra-level) plays an equally essential role. To maintain long-term engagement, levels should vary significantly in structure and challenge. To assess inter-level diversity, levels with identical grid sizes, box counts, and scores have been compared—examining the common features they share and the differences that set them apart. To avoid the typical characteristics that arise when a grid approaches its "density limit", the parameters have been adjusted to maintain a slightly lower density than the maximum allowed. Figure 6.4 is an example of 4 grids (all 9x9 with 6 boxes), scoring with a standard deviation of ± 0.00142 points. Visually, these grids exhibit drastically different features, with varying distributions of tunnels, box clusters, and areas of redirection.

A calculation of used by Zhang et al. [29], allows intra-level diversity to also be quantised via their 'fDiv' score; puzzles with similar distributions of "challenge" throughout the grid would score similarly. This formula, derived from Shannon-entropy, assesses each row's density (p_i) as a portion of the total rows (n) on a scale of [0.00 - 1.00], with a score closer to one indicating an equal distribution of the content amongst the rows:

$$-\frac{1}{\log n}\sum_{i=1}^{n}\frac{\alpha p_i}{n}\log\left(\frac{\alpha p_i}{n}\right)$$

The metric rewards balance of a level's challenge (to its empty space) across the grid and penalises clustering of content to a localised area of the grid. While this does not directly correlate to more difficult levels, as in either instance, it is unclear what the empty space would be used for, it does allow us to quantise the utilisation of the space on the grid and determine how consistently puzzles utilise their space. These levels all scored high, within ± 0.0053 points of each other, indicating a good use of their space; paired with a visual examination, it is clear these levels share few similarities in their design, as intended.

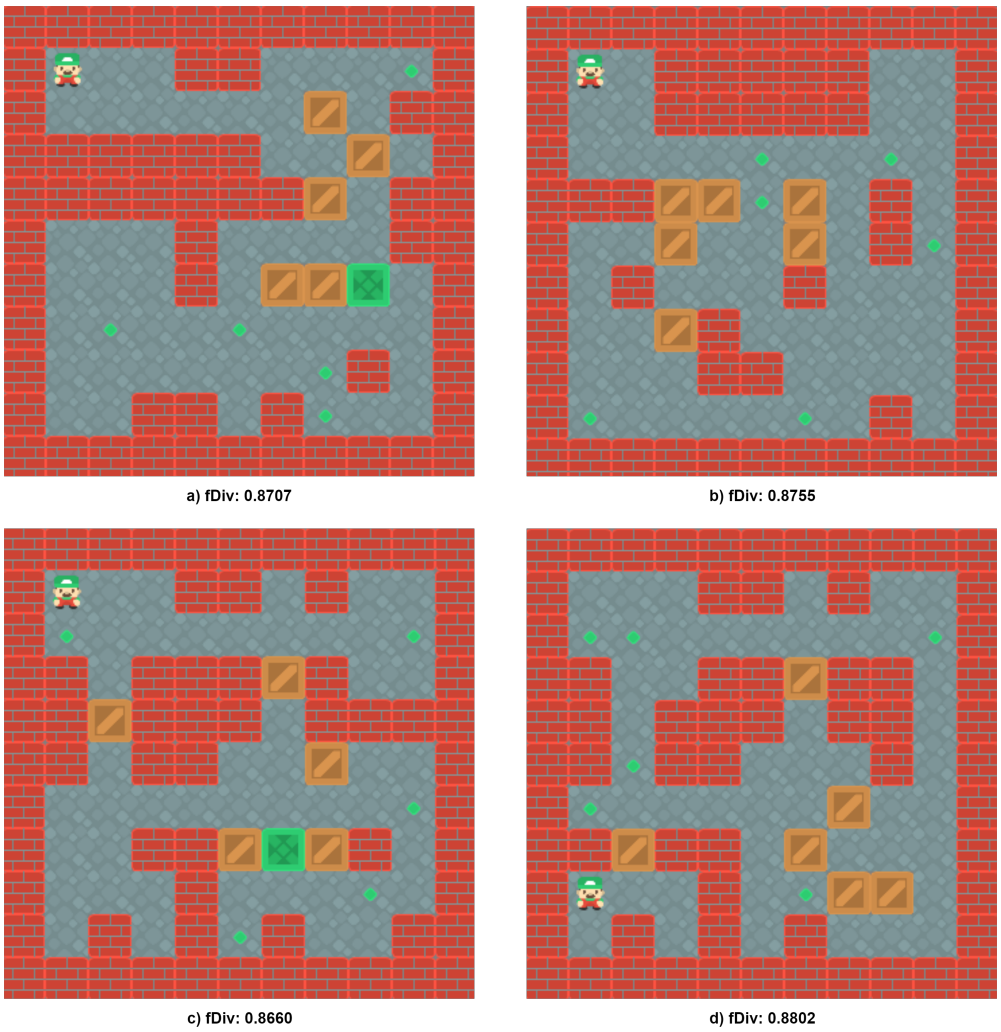


Figure 6.4: Diversity Comparison for Equally Scored Levels

6.3 Scoring Evaluation

This section of the evaluation aims to assess the efficacy of each scoring metric individually, as well as their combined performance. Additionally, it will provide detailed justification for the weightings chosen in the composite score. For the former tests, each metric is considered in isolation, contributing solely to the level's score. At the same time, additional metrics are calculated to explore potential correlations between a metric's maximum value and the others.

For uniform testing conditions, the following tests will all be given a fixed grid size and entity count, 9x9 tiles and 6 boxes/goals respectively. Additionally, 200 instances were generated per metric, and 250 instances generated for the composite score—The top puzzles were taken from each for comparison

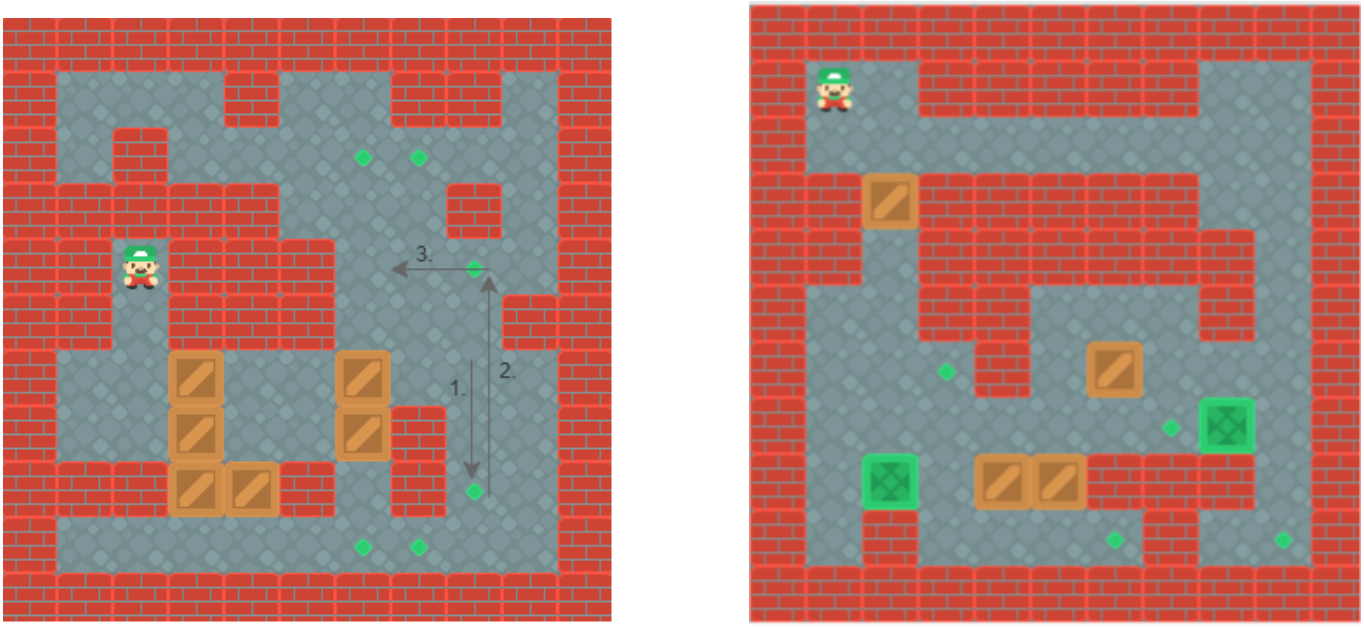
6.3.1 Individual Rating Evaluation

Levels optimised solely for maximum moves often result in repetitive sequences that do not necessarily increase puzzle complexity. Consequently, testing based purely on move count was excluded from the evaluation, focusing instead on pushes (including directional and reverse pushes), and box

changes for deeper insights. The quantitative impacts of optimising each of these four individual metrics is quantised in Tables 6.1 - 6.4.

Across the 200 instances, as expected, optimising the pushes predominantly extends the number of push actions required, often at the expense of puzzle complexity. Such puzzles appear difficult due to their length and repetitive nature, but may lack genuine cognitive challenges. This suggests that high scores in this metric alone do not necessarily equate to high logical difficulty.

Representative puzzles optimised for pushes were visually examined to identify structural patterns. Of the 200, the most pushes overall (a) and the most proportional pushes to the total moves (b) were taken, Figure 6.5 illustrates the puzzle features resulting from this optimisation. Examining the first of these, at a glance, it would appear to present characteristic features of high difficulty; this can be seen through the density of the boxes, tight gaps to work around, and even a large restriction on the order in which goals can be filled (the player must start with the upper left goal and work downwards to avoid deadlock). However, once this final condition can be overcome, 4/6 boxes have near identical routes for reaching their goal state (the abstracted push sequence: [1-3]), thus promoting pushes over challenge. The latter style of level addressed a different concern; if assessing levels by their proportional pushes, it could be the case the puzzle offered little challenge (as can be seen by the 3 corner goals and 2 already-placed boxes) though it is still a solution predominantly made up of pushes. On its own, this presents high volatility in the challenge of levels; a solution could be made up of 500 moves with a high percentage of pushes or 50 moves. However, absolute values cannot be combined with other metrics when attempting to maximise a score so this must be used regardless. Tuning the metric via normalisation and weighting will help regulate its impact.



(a) Optimised total pushes (77/317)

(b) Optimised proportional pushes (20/60)

Figure 6.5: Comparison of Top Push Instances

Table 6.1: Quantitative Analysis (Pushes Maximised, Top 10)

Statistic	Value (mean ± SD)
Moves	338 ± 78.2
Pushes	76 ± 10.2
Directional Pushes	44 ± 7.8
Reverse Pushes	27 ± 5.7
Box Changes	28 ± 7.0
Corner Goals	2 ± 0.9

Table 6.2: Quantitative Analysis (Directional Pushes Maximised, Top 10)

Statistic	Value (mean ± SD)
Moves	321 ± 65.3
Pushes	69 ± 11.9
Directional Pushes	42 ± 3.7
Reverse Pushes	26 ± 9.2
Box Changes	26 ± 4.2
Corner Goals	1 ± 1.2

Table 6.3: Quantitative Analysis (Reverse Pushes Maximised, Top 10)

Statistic	Value (mean ± SD)
Moves	340 ± 79.4
Pushes	75 ± 22.2
Directional Pushes	45 ± 13.6
Reverse Pushes	31 ± 9.8
Box Changes	31 ± 10.5
Corner Goals	3 ± 0.5

Table 6.4: Quantitative Analysis (Box Changes Maximised, Top 10)

Statistic	Value (mean ± SD)
Moves	278 ± 49.8
Pushes	54 ± 11.4
Directional Pushes	36 ± 5.8
Reverse Pushes	17 ± 2.6
Box Changes	27 ± 4.8
Corner Goals	2 ± 0.9

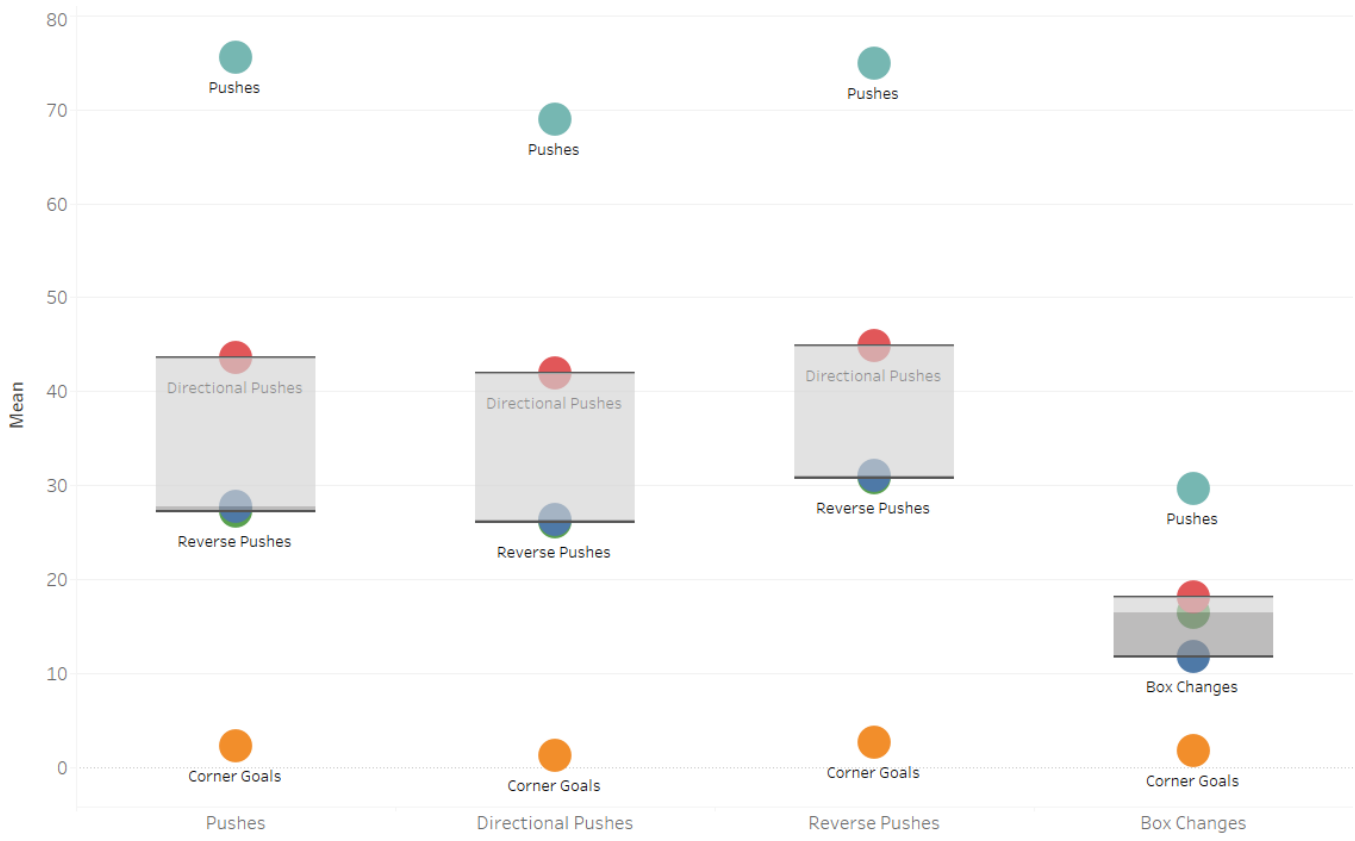


Figure 6.6: Quantitative Analysis (Box Plot)

6.3.2 Composite Rating Evaluation

When combining an array of metrics, there are two conflicting outcomes: the dynamics can be complex, or they may exhibit minimal correlations. The first possibility suggests metrics may have reinforcing or conflicting relationships each other. For example, considering total pushes versus box changes. A level with many total pushes might involve propelling one box back-and-forth repeatedly (high pushes, but low box changes since it is the same box), whereas a level with many box changes encourages alternating between boxes, potentially limiting how many pushes (per box) can happen. Similarly, a level with a high number of directional pushes also constitutes a proportionally high number of pushes. Prioritising one metric through its weighting could inadvertently have one of these impacts on a range of other metrics. The alternative is that truly difficult Sokoban puzzles will score reasonably well on all metrics, but perhaps not extreme on one single metric. For example, a puzzle that is very hard for a solver may have a moderately long solution, requires many distinct push segments, many box switches, and a high number of counter-intuitive pushes—a balanced difficulty profile.

Upon examining the quantitative results from the isolated metrics in Tables 6.1 - 6.4, and their visual comparison in Figure 6.6, both of these predictions are true to an extent. The relationship between box changes and all push-related metrics show an inverse correlation when attempting to solely maximise the box changes, whereas when maximising any of the alternate parameters, their secondary metrics exhibit distributions that are too broad and too consistent across the remaining tables to draw definitive conclusions about causal relationships with the primary metric (including the box changes). A case-by-case analysis of top-performing levels reveals that many high-scoring outliers can be attributed to stochastic factors, such as the pseudo-random placement of entities, and their constraints, which encourage density and interesting series of moves. These random factors tend to inflate secondary metrics independently, rather than reflecting the direct influence of the primary metric itself. Based on these results, coefficients were tuned in regard to how much that metric was to be prioritised, as opposed to what impact it could have on others.

Beginning with seemingly random assignments of coefficients, my intention was to favour the metrics with the greatest contributions to the difficulty as predicted when first discerning how to score levels. Most notably, these are box changes, and directional pushes, while also making the impact of undesirable features such as corner goals more prevalent. This involved frequent manual tuning of each coefficient until the ratio between the metrics appeared representative enough of their importance. Another important consideration was the realistic upper bound on the number of moves a level could have. Since each metric needed to be expressed as a proportion of a defined range, the resulting formula, though requiring frequent tuning, addressed this challenge well, while attempting to not be influenced by high outliers:

$$\max\left(\text{moves}, \sqrt{(\text{total\_tiles})^2 \cdot \text{boxes}}\right)$$

Assessing the performance of the top 10 puzzles (from a set of 250) against an optimised solver, JSoko, contextualises the performance of puzzles in an environment without the intentional exploitation of the inbuilt solver. JSoko makes use of deadlock tables, dedicated heuristics, state caching, and will continue search until an optimal solution is found. As expected, this has considerably lowered any existing counts for the range of metrics (that JSoko tallies) but in exchange provided a ground truth assessment of these levels. The comparison between the Marwes solver and JSoko is detailed in Tables 6.5 and 6.6, with the Table 6.6 also detailing the optimal search results of the built-in 107 handcrafted levels JSoko offers for further comparison. These results show that even for an optimised

solver, the constraints pose a notable challenge, performing similarly in terms of optimal moves and pushes compared to the handmade levels, and in some instances (though highly volatile) surpassing the solve time of the baseline levels by a significant margin. The analysis of density, diversity, and the optimal performance of the puzzles, all illustrate the efficacy of this PCG system across some of the most noteworthy areas of assessment.

Table 6.5: Quantitative Analysis (Composite Score Results, Top 10)

Statistic	**Value (mean ± SD)**
Moves	182 ± 102.9
Pushes	30 ± 20.0
Directional Pushes	24 ± 15.1
Reverse Pushes	9 ± 7.8
Box Changes	19 ± 12.0
Corner Goals	2 ± 1.2

Table 6.6: Quantitative Analysis (JSoko Results: This Work vs. Built-In)

Statistic	**This Work**	**Built-In**
Optimal Moves	70 ± 19.3	94 ± 54.6
Best Pushes	21 ± 6.3	26 ± 14.2
Solve Time	11.0 ± 10.1	2.4 ± 5.2

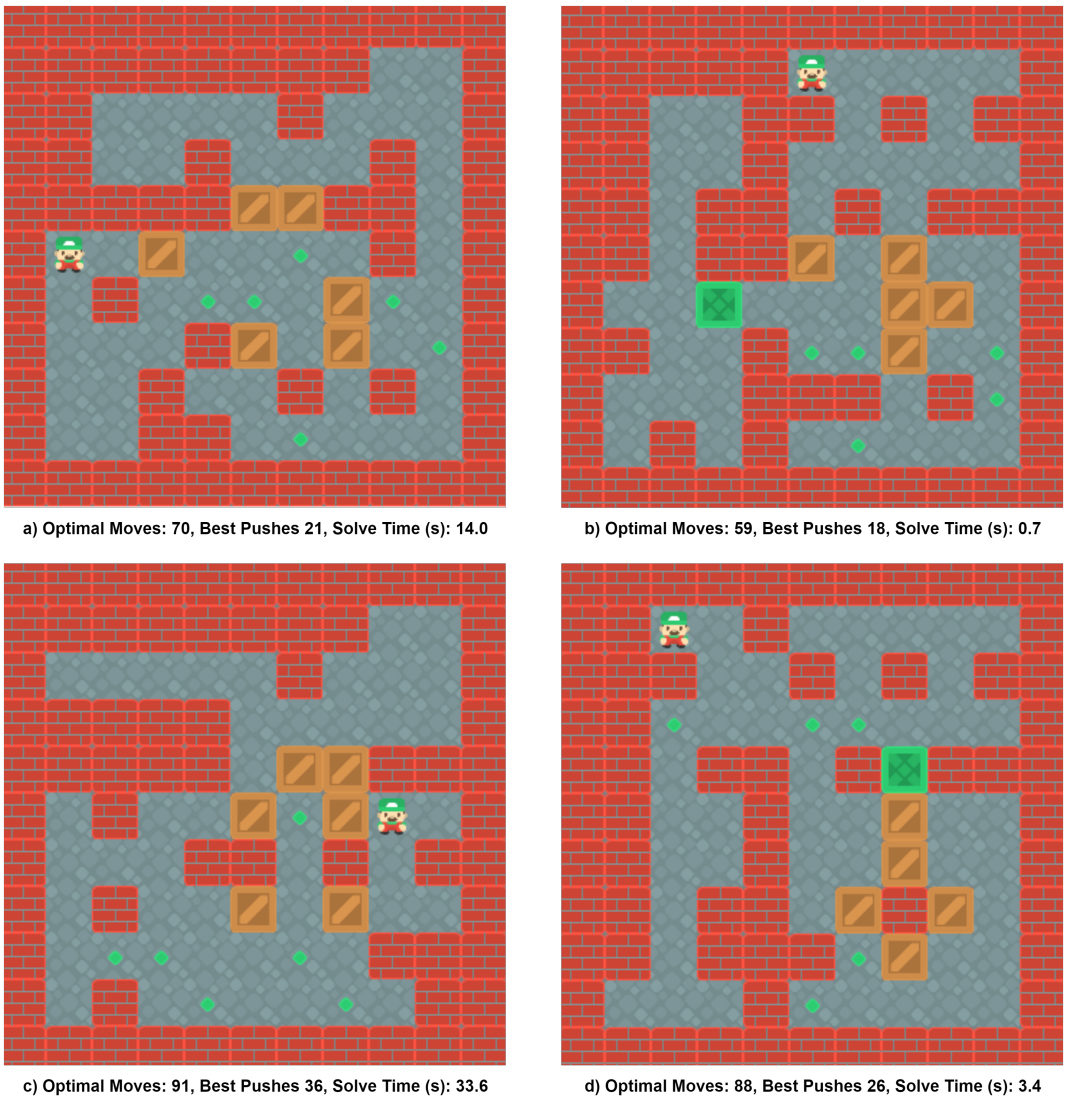


Figure 6.7: Top 4 Composite Scores, JSoko Results

6.4 Critical Appraisal

The critical appraisal focuses specifically on assessing this PCG system against established systems within the field, evaluating its contributions, strengths, and limitations. By assessing the characteristics of of the system, whether that is consistency of the generation or (average) total moves/pushes, this appraisal will determine whether the proposed PCG approach meaningfully advances existing capabilities or simply replicates established techniques.

A key challenge when comparing research against existing studies arises from the varying objectives and methodological differences across the literature. Even in cases where studies seem to align broadly in their goals, comparisons become problematic due to the lack of a standardised definition for core concepts such as "difficulty"—in some instances, difficulty is not considered altogether. Each paper tends to approach puzzle complexity through distinct metrics, such as solver depth or time taken to solve, or subjective interpretations provided by human participants, making direct evaluation challenging and limiting the extent to which meaningful conclusions can be drawn from cross-study comparisons.

Among the studies suitable for comparison, each evaluated their algorithms using different baseline grid sizes and entity counts. Where applicable, these parameters, and recorded metrics, were copied on a case-by-case basis to ensure fairness and consistency in testing conditions. The corresponding results are presented in Table 6.7.

6.4.1 Results

What can be seen across the results of Table 6.7, is a general improvement from the existing baselines across the majority of test cases. While this is representative of the performance to a large extent, this evaluation is unable to mimic the testing conditions exactly, as the generation styles differ. For instance, Karman's tree-based generation [10], and Taylor and Parberry's generate-and-test approach [19], both extend the duration of the PCG such that it will rely more on the generation and less on the scoring for producing a set of levels. However, my system favours the alterative. In such cases, a direct comparison of a random instance (or 10), will result in my puzzles performing worse, whereas a comparison of levels generated over an extended period of time will show the opposite.

There exist other discrepancies in the results as well, the evaluation performed by Murase et al. [18] provided no details on the testing conditions, nor what is considered an "interesting puzzle"—I assumed a score of 0.66 or greater would be sufficient for my comparison. Additionally, my system ensures solvability by working backwards from a goal state, thus a comparison of how consistently solvable levels are will reward my PCG far more.

In summary, while the comparative results suggest that my system performs competitively—and often favourably—against established baselines, it is important to recognise the limitations in the evaluation methodology. Differences in generation strategy, evaluation criteria, and the availability of testing details all contribute to potential biases in direct comparisons. Nonetheless, the consistent improvements observed under my system's own scoring criteria, particularly in terms of solvability and puzzle quality, highlight its strength as a viable and effective approach to procedural level generation.

Study	Prerequisites	Results Comparison
Karman [10]	Grid Size: 6x4 Box Count: 4 Quartiles across 40 instances	**Baseline**: Human Participants Number of Moves: 13 - 34.4 - 111 Number of Pushes: 6 - 11.0 - 33 **This Work**: Marwes Solver Number of Moves: 35 - 41 - 56.5 Number of Pushes: 8 - 10.0 - 12
Taylor and Parberry [19]	Grid Size: 3x6 Box Count: 3 58 sec, Top 10	**Baseline**: Average Moves: 38 **This Work**: Average Moves: 27
	Grid Size: 6x6 Box Count: 3 2.7 min, Top 10	**Baseline**: Average Moves: 69 **This Work**: Average Moves: 64
	Grid Size: 6x9 Box Count: 3 1.1 h, Top 10	**Baseline**: Average Moves: 98 **This Work**: Average Moves: 112
	Grid Size: 9x9 Box Count: 3 1.1 h, Top 10	**Baseline**: Average Moves: 115 **This Work**: Average Moves: 134
Murase, Matsubara, and Hiraga [18]	Grid Size: 6x6 Box Count: 3 (Assumed)	**Baseline**: Trails: 500 Generation failures: 7 No solution: 245 Solvable: (Removed: 204 — Outputted: 44) Interesting Puzzles: 14 **This Work**: Trails: 500 Generation failures: 0 No solution (After entity placement): 0 Solvable: (Removed: N/A — Outputted: N/A) Interesting Puzzles: 31 (Scored ≥ 0.66)

Table 6.7: PCG Comparison

CHAPTER 7

Conclusion

The result of this project is a puzzle generator capable of autonomously generating, constraining, and evaluating instances of Sokoban. When treated as an optimisation problem, these puzzles can present substantial amounts of challenge, rivalling other PCG systems currently available. The foundation of this project was built on the works of some of the earliest techniques of Sokoban instance generation, which follow a generate-and-test development [18][19], being the most well-covered style of Sokoban PCG.

The project set out to discover the different ways constraints can be embedded into the PCG pipeline to capitalise on known pitfalls of the solver and its chosen search heuristic. Additionally, generated levels are evaluated relative to each other through a set of scoring metrics designed to highlight fundamental attributes that contribute to the interest and complexity of Sokoban levels.

A secondary goal of this dissertation aimed to explore the idea of interesting levels, and to what extent this has been achieved. Answering this depends not only on the metrics used internally within the PCG system but also on how these levels compare to those observed in all Sokoban levels created since the game's inception. Whether these historical levels were meticulously handcrafted or generated through pseudo-random processes influences how comparatively interesting newly generated levels are perceived to be. When evaluating against handcrafted puzzles, it becomes evident that certain qualities cannot be reliably reproduced by the current approach. This limitation arises primarily from the simplicity of the constraints and metrics employed. Though effective, these fail to capture the deeper design principles hidden within handcrafted puzzles—most notably, a nuanced concept of "core moves". I define these moves as seemingly counter-intuitive (sequences of) moves which revert a significant amount of progress or appear to be (dead-)locking but are required in order for the level to be progressed—a sub-puzzle within the problem.

While this concept cannot be accounted for in the final product, this PCG system does succeed in generating difficulty through other means. The pull strategy (Section 5.2.2) paired with an array of constraints successfully builds on and leverages Sokoban's most known characteristics for describing difficulty; this is supported by the level ratings as part of the built-in evaluation.

7.1 Limitations

This instance generation system, and a flaw consistent across PCG, is its inability to replicate the same intricacies of manual design, primarily due to the struggle for this tool to "plan long-term". Looking on the actual performance of this system, it is limited in a few other areas as well:

- **Grid size**: Grids are constrained to simple rectangular layouts, and additionally, only sizes that are a multiple of 3 (for both rows and columns), limiting the potential intrigue of a level. However, this can be adjusted manually.

- **Evaluation inaccuracies**: It is not always guaranteed a level with the best rating is the best level in that trial. Though metrics are tuned, certain attributes of the rating could inflate the score more than intended, thus taking precedence over a more deserving level with a different distribution of metrics.

- **Density upper limit**: Examining some of the most popular handcrafted levels, a common characteristic emerges: a high density of entities occupying up to 50% of the grid's tiles. Attempts to replicate this feature were unsuccessful, primarily due to constraints enforcing absolute minimum Manhattan distances between boxes and their goals, combined with grid templates that inherently contain numerous wall tiles (this is more detrimental the smaller the grids become). These factors collectively limit the available space for successful generation. While achieving such dense configurations is not entirely impossible with the current system, it would require manual adjustments to constraints and extensive iterative trials—efforts which have yet to yield success.

- **Core moves undefined**: In the same way it was difficult to explicitly define dead-ends, core moves also cannot be restricted to a specific concept due to their open-ended nature, and thus cannot be implemented as a constraint in any consistent way. Merely comparing multiple solutions of an instance for equal (core) strings of moves, creates a breadth of new issues, primarily related to its computational feasibility, and whether or not this has any significance to the final answer. This alone opens up another study entirely and would perhaps require a significant structuring of the current approach to accommodate for, such as a new solver heuristic which will prioritise these segment(s) of a level.

7.2 Future Work

This section details potential future directions in which the work could have been expanded, further developed or restructured entirely, highlighting opportunities for continued research and improvement:

- **Symmetry breaking**: An aspirational goal for building levels was the integration of symmetry breaking; whether it was through the process of penalising symmetry in the evaluation, or avoiding it in the generation, eliminating symmetry could be a significant benefit in raising the difficulty.

  Lexicographic ordering (specifically Lex^2) can be used for pruning search spaces within constraint satisfaction problems. States of the game are comprised of 2D grids and so lexicographic ordering can be applied here, capable of restricting exploration exclusively to configurations representing the smallest lexicographical ordering amongst all symmetrical permutations. The process of identifying (and breaking) symmetry would allow grids which lack symmetry to be prioritised; the implication would be that levels which display less symmetry have fewer ways to reach a solution, suggesting the solution could be harder to find.

  Adopting this strategy would entail overhauling the existing generate-and-test pipeline and transitioning toward a tree-based search method, such as Monte Carlo Tree Search.

- **Deadlock Tables**: Another approach with strong potential to accelerate level generation involves the use of deadlock tables as an intermediate validation step. By incorporating deadlock tables before invoking the solver, levels containing static deadlock patterns arising from keeper placement (Section 5.2.5) could be swiftly identified and discarded, significantly enhancing efficiency.

Additionally deadlock tables could be embedded into the simulation process, as a tool for identifying "near-deadlock" states. Instead of only checking for outright unsolvable configurations, the simulation can use deadlock tables to detect areas in a level that are close to known deadlock patterns. For example, if a level has a configuration where a single move could lead into a deadlock (or force a recovery move), that "proximity" to a deadlock, and the amount of these, can serve of boosts to the difficulty. Though the implementation of this has not been solidified by any paper in particular, and would likely be a computationally taxing process to find sub-grids which nearly match a deadlock orientation, it is still worth consideration.

- **Additional solver comparisons**: Embedding more than one solver into to the evaluation process, would provide more insights into the universally difficult qualities of the levels, and not simply the ones which leverage the solver's properties.

Bibliography

[1] Adi Botea, Martin Müller, and Jonathan Schaeffer. "Using Abstraction for Planning in Sokoban". In: *Lecture Notes in Computer Science* (2003), pp. 360–375. DOI: https://doi.org/10.1007/978-3-540-40031-8_24.

[2] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[3] Joseph Culberson. "Sokoban Is PSPACE-complete". In: *In Proc. International Conference on Fun with Algorithms* (Jan. 1997), pp. 65–76. DOI: https://doi.org/10.7939/r3jm23k33.

[4] *Geeks for Geeks: Bridges in a Graph*. May 2024. URL: https://www.geeksforgeeks.org/bridge-in-a-graph/.

[5] Michael Cerny Green et al. "Two-step constructive approaches for dungeon generation". In: *Proceedings of the 14th International Conference on the Foundations of Digital Games*. FDG '19. San Luis Obispo, California, USA: Association for Computing Machinery, 2019. ISBN: 9781450372176. DOI: 10.1145/3337722.3341847. URL: https://doi.org/10.1145/3337722.3341847.

[6] Jarušek and Pelánek. "Difficulty Rating of Sokoban Puzzle." In: vol. 222. Jan. 2010, pp. 140–150. DOI: 10.3233/978-1-60750-675-1-140. URL: https://www.researchgate.net/publication/220774696_Difficulty_Rating_of_Sokoban_Puzzle.

[7] Andreas Junghanns. "Pushing the limits: new developments in single-agent search". PhD thesis. University of Alberta, 1999, p. 5. DOI: https://doi.org/10.7939/R3W95103S.

[8] Andreas Junghanns. *Rolling Stone Solver*. Ed. by Jonathan Schaeffer. 2024. URL: http://sokobano.de/wiki/index.php?title=Rolling_Stone_solver.

[9] Andreas Junghanns and Jonathan Schaeffer. "Sokoban: improving the search with relevance cuts". In: *Theoretical Computer Science* 252.1 (2001). CG'98, pp. 151–175. ISSN: 0304-3975. DOI: https://doi.org/10.1016/S0304-3975(00)00080-3. URL: https://www.sciencedirect.com/science/article/pii/S0304397500000803.

[10] Simon Karman. "Generating Sokoban Levels That Are Interesting to Play Using Simulation". Master Thesis. Utrecht University, June 2018, pp. 13–14. URL: https://studenttheses.uu.nl/bitstream/handle/20.500.12932/29729/Master%20Thesis%20-%20Simon%20Karman%20-%20Version%201.0.pdf?sequence=2&isAllowed=y.

[11] Kenney. *Sokoban · Kenney*. Jan. 2024. URL: https://kenney.nl/assets/sokoban.

[12] Paul Hyunjin Kim and Roger Crawfis. "Intelligent Maze Generation Based on Topological Constraints". In: *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*. 2018, pp. 867–872. DOI: 10.1109/IIAI-AAI.2018.00176.

[13] Hung Le. *How to Find Dead Ends Efficiently*. Feb. 2019. URL: https://hunglvosu.github.io/res/deadend.pdf.

[14] Fotis Liarokapis. "Fractal Terrain Generation". BSc Dissertation. University of Sussex, 1998, pp. 4–11. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=abf168ab83c8512b09823dd5e02f6dc2efe586a6.

[15] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. "Procedural Generation of Dungeons". In: *IEEE Transactions on Computational Intelligence and AI in Games* 6.1 (2014), pp. 78–89. DOI: 10.1109/TCIAIG.2013.2290371.

[16] Marwes. *sokoban/src at Marwes/sokoban*. 2025. URL: https://github.com/Marwes/sokoban/tree/master/src.

[17] Matthias Meger. *JSoko*. 2025. URL: https://jsokoapplet.sourceforge.io/.

[18] Yoshio Murase, Hitoshi Matsubara, and Yuzuru Hiraga. "Automatic making of Sokoban problems". In: *Proceedings of the 4th Pacific Rim International Conference on Topics in Artificial Intelligence*. PRICAI'96. Cairns, Australia: Springer-Verlag, 1996, pp. 592–600. ISBN: 3540615326. DOI: 10.1007/3-540-61532-6_50. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=af6c8e280946f019c7f632c8281a35ddf1fdc9b0.

[19] Ian Parberry and Joshua Taylor. *Procedural Generation of Sokoban Levels*. Texas, USA, Feb. 2011, pp. 5–12. URL: https://ianparberry.com/techreports/LARC-2011-01.pdf.

[20] *PCG Wikidot*. 2019. URL: http://pcg.wikidot.com/.

[21] André G. Pereira, Marcus Ritt, and Luciana S. Buriol. "Optimal Sokoban solving using pattern databases with specific domain knowledge". In: *Artificial Intelligence* 227 (2015), pp. 52–70. ISSN: 0004-3702. DOI: https://doi.org/10.1016/j.artint.2015.05.011. URL: https://www.sciencedirect.com/science/article/pii/S0004370215000867.

[22] Gabrovšek Peter. "Analysis of Maze Generating Algorithms". In: *IPSI Transactions on Internet Research* 15.1 (2019), pp. 23–30. URL: http://www.ipsitransactions.org/journals/papers/tir/2019jan/p5.pdf.

[23] Thomas J. Rose and Anastasios G. Bakaoukas. "Algorithms and Approaches for Procedural Terrain Generation - A Brief Review of Current Techniques". In: *2016 8th International Conference on Games and Virtual Worlds for Serious Applications (VS-GAMES)*. 2016, pp. 1–2. DOI: 10.1109/VS-GAMES.2016.7590336.

[24] Noor Shaker et al. "Constructive generation methods for dungeons and levels". In: *Procedural Content Generation in Games*. Cham: Springer International Publishing, 2016, p. 41. ISBN: 978-3-319-42716-4. DOI: 10.1007/978-3-319-42716-4_3. URL: https://doi.org/10.1007/978-3-319-42716-4_3.

[25] *Sokoban Wiki*. 2024. URL: http://www.sokobano.de/wiki/index.php?title=Main_Page.

[26] Matthew James Taylor. *Fractal Art (Exploring Fractals & Art Based on Fractals)*. Jan. 2023. URL: https://matthewjamestaylor.com/fractal-art.

[27] Graham Todd et al. "Level Generation Through Large Language Models". In: *Proceedings of the 18th International Conference on the Foundations of Digital Games*. FDG '23. Lisbon, Portugal: Association for Computing Machinery, 2023. ISBN: 9781450398558. DOI: 10.1145/3582437.3587211. URL: https://doi.org/10.1145/3582437.3587211.

[28] Michael Wirth. *Fractal Brownian Motion Using Random Midpoint Displacement*. June 2021. URL: https://craftofcoding.wordpress.com/2021/06/29/fractal-brownian-motion-using-random-midpoint-displacement/.

[29] Qingquan Zhang et al. “Interpreting Multi-objective Evolutionary Algorithms via Sokoban Level Generation”. In: *2024 IEEE Conference on Games (CoG)*. 2024, pp. 1–2. DOI: 10.1109/CoG60054.2024.10645559.

Appendix

A.1 User Manual

This appendix includes instructions and a sample script for running the project.

Grant permissions to script (from the 3 below)

$ chmod +x ⟨*scriptname*⟩

Running the GUI, play a 'levelset' by giving its id or '-1' for a random level

The current configurations has the generator set to a 9x9 grid with 6 boxes, this is level configuration that has been used for testing purposes. Entering "1" into the pop-up prompt will begin a series of pre-generated levels to interact with (the same levels of Section 6.3.2).

$ ./run_gui.sh

Running the Instance Generator once

$ ./run_gen

Running the Instance Generator for generation in bulk, the number of iterations is hardcoded into the script

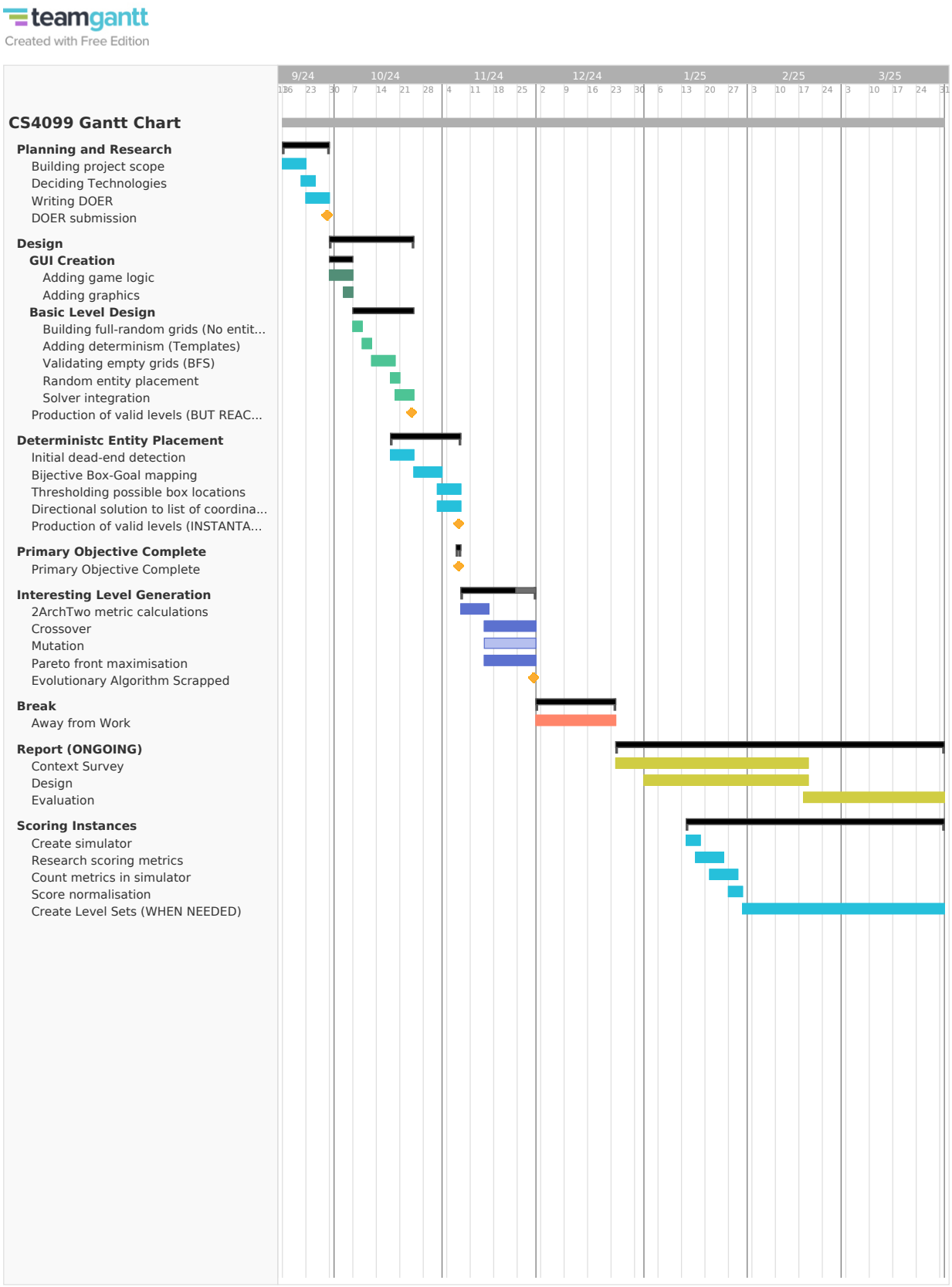
$ ./run_solutions.sh

Code Structure:

- **src**: Source Code (Scripts/README are outside of this folder)
- **out**: Class files
- **graphics**: GUI tile PNGs
- **results**: Generation files for individual and composite metric evaluation. Specific grid size/box count tests placed in folders with the respective configuration as the name: $\langle rows \rangle X \langle cols \rangle \_ \langle boxes \rangle B$

  This also contains search results of JSoko's handcrafted levels
- **levels**: Levelsets, each containing a list of seeds to run
- **diagrams**: Report diagrams
- **assets**: Evaluation CSV files, Tableau files, and Gantt Chart

A.2 Gantt Chart



A.3 Ethics

UNIVERSITY OF ST ANDREWS
TEACHING AND RESEARCH ETHICS COMMITTEE (UTREC)
SCHOOL OF COMPUTER SCIENCE
PRELIMINARY ETHICS SELF-ASSESSMENT FORM

This Preliminary Ethics Self-Assessment Form is to be conducted by the researcher, and completed in conjunction with the Guidelines for Ethical Research Practice. All staff and students of the School of Computer Science must complete it prior to commencing research.

This Form will act as a formal record of your ethical considerations.
Tick one box

- ☐ **Staff Project**
- ☐ **Postgraduate Project**
- ☑ **Undergraduate Project**

Title of project

Sokoban Instance Generator

Name of researcher(s)

Sami Hatoum

Name of supervisor (for student research)

Ruth Hoffman

OVERALL ASSESSMENT (to be signed after questions, overleaf, have been completed)

Self audit has been conducted **YES** ☐ **NO** ☐

There are no ethical issues raised by this project

Signature Student or Researcher

Print Name

Sami Hatoum

Date

26/09/2024

Signature Lead Researcher or Supervisor

Print Name

Date

This form must be date stamped and held in the files of the Lead Researcher or Supervisor. If fieldwork is required, a copy must also be lodged with appropriate Risk Assessment forms. The School Ethics Committee will be responsible for monitoring assessments.

Research with secondary datasets

Please check UTREC guidance on secondary datasets (https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/secondary-data/ and https://www.st-andrews.ac.uk/research/integrity-ethics/humans/ethical-guidance/confidentiality-data-protection/). Based on the guidance, does your project need ethics approval?

YES ☐ **NO** ☑

** If your research involves secondary datasets, please list them with links in DOER.*

Research with human subjects

Does your research involve collecting personal data on human subjects?

YES ☐ **NO** ☑

If YES, full ethics review required

Does your research involve human subjects or have potential adverse consequences for human welfare and wellbeing?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Will you be surveying, observing or interviewing human subjects?
Does your research have the potential to have a significant negative effect on people in the study area?

Potential physical or psychological harm, discomfort or stress

Are there any foreseeable risks to the researcher, or to any participants in this research?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Is there any potential that there could be physical harm for anyone involved in the research?
Is there any potential for psychological harm, discomfort or stress for anyone involved in the research?

Conflicts of interest

Do any conflicts of interest arise?

YES ☐ **NO** ☑

If YES, full ethics review required
For example:
Might research objectivity be compromised by sponsorship?
Might any issues of intellectual property or roles in research be raised?

Funding

Is your research funded externally?

YES ☐ **NO** ☑

If YES, does the funder appear on the 'currently automatically approved' list on the UTREC website?

YES ☐ **NO** ☑

If NO, you will need to submit a Funding Approval Application as per instructions on the UTREC website.

Research with animals

Does your research involve the use of living animals?

YES ☐ **NO** ☑

If YES, your proposal must be referred to the University's Animal Welfare and Ethics Committee (AWEC)

University Teaching and Research Ethics Committee (UTREC) pages
http://www.st-andrews.ac.uk/utrec/