Making Mario Work Hard

Aaron Ceross
Supervisor: Dr. Benjamin Sach

MSc Thesis Project
COMSM3201 MSc Project, Computer Science

August 2, 2015

Executive Summary

Blank at the moment

Acknowledgements

Blank at the moment

Contents

Chapter 1

Aims and Objectives

> "The obvious objective of video games is to entertain people by surprising them with new experiences."
>
> Shigeru Miyamoto

1.1 Background and motivation

An NP-complete problem is one with a solution that can be *verified* quickly in polynomial time, though an existing solution is not readily known or *identifiable* [10]. It has been proven that certain classic video games, such as *Super Mario Bros* [1], are computationally hard in that the character could be made to solve an arbitrary instance of a Boolean satisfiability problem (SAT). SAT is a type of NP-complete problem in which the variables of the formula may be consistently replaced by the values TRUE or FALSE in a way that the formula evaluates to TRUE, or 'satisfiable'.

The primary motivation of this research is to investigate the feasibility of the video game medium as a teaching tool to visualise and demonstrate concepts in computational complexity. This could be achieved through the evaluation of different instances of SAT in a video game setting.

Computer science and mathematics can be quite daunting subjects for new students and quite often the concepts may not be clear. The utilisation of video games as a medium to introduce the topic and concepts could be

There are several challanges that need to be addressed in order for this method to be feasible.

1.2 Research aims and objectives

The aim of this project is to produce a game engine that is capable of visualising and demonstrating concepts in computational complexity through the evaluation of different instances of SAT. The game engine would be able to take an instance of SAT

and generate a human-playable level that is computationally hard. This level would then be able to be solved by a SAT solver. The solution to the level would then be visualised on screen. In order to fulfil this aim, the research project will implement a game engine through achieving the following objectives:

1. Develop a platform game in which the mechanics are NP-hard;
2. Develop a level generator which converts an instance of SAT into a playable level;
3. Develop a visualisation of the SAT solving these levels by choosing a suitable path;
4. Evaluate level-generation and display of the SAT solver's decisions for efficiency and success against an established criteria or one developed during the project.

As shown from the range of topics above, the areas being investigated are expansive and multi-faceted. Therefore, this research review surveys those elements most relevant for the implementation and development of the game engine software.

1.3 Thesis Structure

This research project is divided between 66% Type I (software development) and 34% Type II (investigatory, research). The structure of this document reflects this division in that the theoretical concepts are broadly introduced and discussed insofar as relevant to the research aim. The discussion strongly focuses on the implementaiton and development challenges in realising these concepts in the project.

Chapter 2 covers the fundamental concepts in computational complexity, identifying which characteristics render a task difficult. The chapter discusses the well-known P vs NP question and explains the NP-hardness complexity class. Other classes which are later explored within the project are also demonstrated and analysed.

The overview of computational complexity provides the basis for the discussion in Chapter 3 on Boolean Satisfiability problems and SAT solving software. In addition to explaining the major features of SAT solvers, the chapter also identifies SAT solvers which had been considered during the project's development.

Chapter 4 relates to how video games can be classified as 'hard', through use of Boolean satisfiability solvers. An overview of relevant research explores the models and proofs that have been utilised in making such classification. The frameworks highlighted in this chapter will form the basis of a model for the implementation of the game engine. After which, Chapter 5 turns the focus of the research review to the generation of game content, particularly centred on game levels. This chapter critically examines methodologies and techniques.

In Chapter 6, the research review will consider the initial design specifications and features of the implementation of the game engine. Discussion will focus on utilising the outputs from this review in a viable game engine model. Finally, Chapter 7 will draw conclusions from this initial research review and discuss future areas of investigation.

Chapter 2

Computational complexity theory

> "In general, it is much harder to *find* a solution to a problem than to *recognize* one when it is presented"
>
> Stephen A Cook [10]

2.1 Fundamental concepts

2.1.1 Definitions

The goals of the research project are to visualise computational complexity through video games. Therefore, the literature review begins with identifying the relevant key concepts in computational complexity theory in order to understand what characteristics makes a problem computationally hard.

Computational complexity theory is concerned with assessing and classifying the difficulty of defined tasks as well as the relationship between such tasks. Within computational complexity, there exist different types of problems, termed *complexity classes*. Complexity classes rank the difficulty of differing types of problems. Of the many types of problems which can exist in computational complexity theory, the two fundamental types this research project is interested in are (*i*) decision problems and (*ii*) search problems. The importance of the distinction between these two types will be made clearer when discussing Boolean satisfiability (Chapter 2) and procedural content generation (Chapter 5).

Decision problems and search problems

A decision problem is a binary choice one wherein, on an infinite set of inputs, the question may be answered '*yes*' or '*no*' It is common to define the decision problem equivalency as: "the set of inputs for which the problem returns *yes*". Most problems may be reduced to a decision problem [37], which is important in that the determination

of whether a solution exists or not is required in order to solve the complementary search problem [26].

Search problems represent a significant area of research in computer science (citation needed), covering topics such as sorting and identifying the shortest path [26]. A search problem consists of the identification of a solution from a set of solutions (possibly infinite or empty). Therefore, given an instance of a search problem, a solution must be found or return a determination that no solution exists in such instance.

2.1.2 Survey of complexity classes

P and NP

The two fundamental complexity classes relevant for this research project are **P** and **NP**. The P (polynomial) class of problems contain those decision problems that are often described as 'easy' problems [37] as the time required to solve such problem has an upper- bound that scales by a polynomial function of the input [53]. The other complexity class, NP (non-deterministic polynomial), relates to those problems that do not have an efficient way of finding a solution but can have a solution verified in polynomial time [26, 49].

P vs NP Problem

The P vs NP problem refers to search problems to which there exists an efficient algorithm that given a solution to a given instance determines whether or not the solution is correct[53, 26, **?**, 21].[1] Although any given solution to an NP-complete problem can be verified quickly (in polynomial time), there is no known efficient way to locate a solution in the first place. A distinguishing characteristic of of NP- complete problems is that there is no known efficient solution to them [26]. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows.

Proving that P = NP or P $\neq$ NP is one of the most important open questions in theoretical computer science and has been described as one of seven principal unsolved problems in the field of mathematics [17]. The implications of P = NP are far-reaching. If it were possible to demonstrate that P = NP (i.e. all NP problems can be solved in polynomial time), then it would then be possible to solve difficult problems in polynomial time (see Figure 2.1). For every problem that has an efficiently verifiable solution, that same solution could also be efficiently identified. While the problem is still being researched and open to debate, there is wide belief that P $\neq$ NP is more likely than P = NP [24].

NP-hard

A problem is considered to be classed as **NP-hard** if the solution's algorithm may be adapted to solve any problem in NP. By extension, this will then include all problems

[1] This is also sometimes described as the P vs NP Question in literature

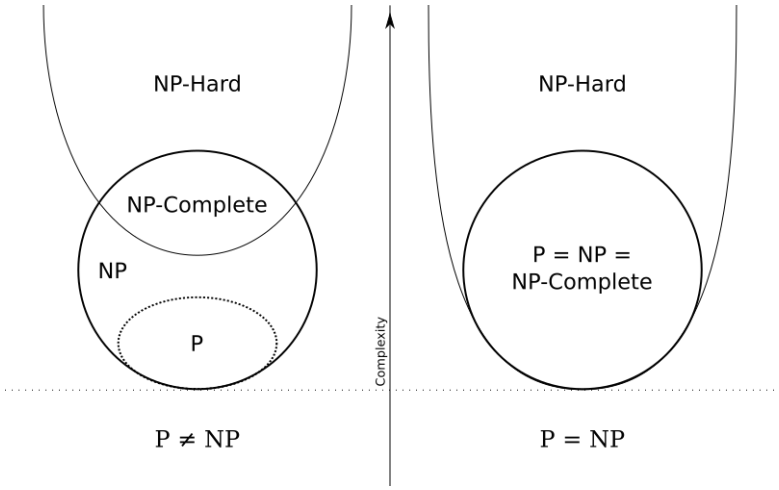

Figure 2.1: Euler diagram illustrating the relationship between P and NP [22].

in P, which are contained within NP. It should be noted that not all NP-hard problems fall within NP nor are NP-hard problems necessarily decidable problems [53, 26, 37, 21].

NP-complete

When a problem is classed as both NP and NP-hard is said to be **NP- complete** and are considered to be the hardest problems in NP [53, 26, 37, 21]. A problem that is NP- complete can have every other problem in NP reducible to it [50]. 'Reduction' in this sense that there exists an algorithm in P that change a problem into another instance [21].

PSPACE

Another relevant complexity class to this research review is **PSPACE**, as some games have been proven to be in this class [23, 15].[2] PSPACE refers to decision problem which may be solved with an amount of space polynomial in the size to its input [53].

2.2 Chapter summary

This chapter has identified the relevant complexity classes and problems in order to establish that video games are computationally complex in Chapter 4. In the next chapter, these concepts will be used to show how problem instances may be proven to be NP-complete through the use of Boolean expressions. This will provide the foundation for being able to develop a game engine which produces computational hard levels in a video game.

[2]Please refer to Chapter 3 of this research review for further information.

Chapter 3

Boolean satisfiability problems

> "It is not of the essence of mathematics to be conversant with the ideas of number and quantity."
>
> George Boole

This chapter provides insight into the hardness of Boolean satisfiability problems (or SATs). SATs are the basis for proving the hardness of video games so understanding how they work is important. During the implementation, SAT solvers (programs that solve instances of SAT) will be used to run tests through the constructed game.

3.1 Principle concepts and definitions

A Boolean formula is a logic expression with variables that have a value of either TRUE or FALSE. These formulae also contain logical connectives, which subscribe to an order of precedence: negation ($\neg$), conjunction($\wedge$), disjunction ($\vee$), implication ($\leftarrow$), and equivalence ($\leftrightarrow$) [3, 27].

Formulae are constructed through literals, which is a variable (e.g. x or y) or its negation (e.g. x or $\neg x$) joined together in clauses (e.g. x). Of these constructions, the conjunctive normal form (CNF) is of particular importance, especially in relation to finding solutions to satisfiability problems, particularly those solving methods based on DPLL algorithm [27] (See Section X below for further discussion). To be in CNF, a formula must a conjunction of clauses, wherein those clauses are a disjunctive of literals. For example:

$$(\text{x} \vee \neg\text{y}) \wedge (\neg\text{x} \vee \text{z} \vee \text{y}) \wedge (\neg\text{z} \vee \text{x})$$

The example above is a conjunctive normal form with three clauses (joined together by $\wedge$ (AND)) containing three literals (x, y, z).

3.2 Boolean Satisfiability problem and NP-completeness

The Boolean satisfiability problem (SAT) is the problem of determining if there exists an assignment that satisfies a given Boolean formula. In order for such assignment to

be *satisfiable*, the variables' values are replaced constantly with TRUE or FALSE until the formula evaluates to TRUE. Where no such assignment is possible (i.e. the formula evaluates to FALSE in all combinations), the formula is determined to be *unsatisfiable* [3, 27]. Therefore, given a CNF formula F, SAT may be posed as: does F have a satisfying assignment?

This question has an important place in computer science as it was the first problem proven to be NP-complete [9].[1] In solving a SAT problem, the question involves not only providing a yes or no, but also in finding an actual solution to the assignment [68]. Specialised software, SAT solvers (discussed in Section 3.2.2 below), provide these solutions, if one exists.

3.2.1 Examples of SAT reductions

Three-satisfiability or 3SAT is the SAT reduction is most relevant for the research project as it has been used to demonstrate the NP- completeness of 2D platform games [1]. In 3SAT, there may only ever be three literals in each clause and at least of of the literals contain a value of TRUE in order to achieve satisfiability [3].There is a variant of the 3SAT, exactly 1-3-satisfiability or 1-in-3SAT above reduction wherein the each clause must contain *exactly* one literal [3, 21].

3.3 SAT solvers

SAT solvers are software that given a SAT instance can either find a solution, which would be a satisfying variable assignment or prove that no solution exists [68]. There are also stochastic methods based on local search may be able to find satisfiable instances in a fast manner but are unable to prove that an instance is in fact unsatisfiable [27].

3.3.1 The DPLL algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [12] is a complete, systematic search algorithm which is based on backtracking. The DPLL algorithm (see Figure 3.1) is able to decide the satisfiability of a CNF SAT reduction as well as identify the satisfying assignment. It can also show that a given Boolean formula is unsatisfiable. The algorithm achieves these results through the use of branching and where clauses are found to be FALSE, removes these from its search space.

The DPLL algorithm works with the CNF of a propositional logic expression (as described in Section 3.1 above). Given that any formula may be converted into CNF through the addition of new variables corresponding to the sub-formulae [64, 26], DPLL may used to in all cases.

[1] While this is often described as *Cook's Theorem* it had been independently verified in [39] and is more accurately termed the *Cook- Levin Theorem*

```
Input  : A CNF formula F and an initially empty partial assignment ρ
Output : UNSAT, or an assignment satisfying F
begin
    (F, ρ) ← UnitPropagate(F, ρ)
    if F contains the empty clause then return UNSAT
    if F has no clauses left then
        Output ρ
        return SAT
    ℓ ← a literal not assigned by ρ                    // the branching step
    if DPLL-recursive(F|ℓ, ρ ∪ {ℓ}) = SAT then return SAT
    return DPLL-recursive(F|¬ℓ, ρ ∪ {¬ℓ})
end

sub UnitPropagate(F, ρ)
begin
    while F contains no empty clause but has a unit clause x do
        F ← F|x
        ρ ← ρ ∪ {x}
    return (F, ρ)
end
```

Figure 3.1: DPLL in pseudo-code. Adapted from [3, 27].

3.4 Available SAT solvers

The following SAT solvers have been identified as being potentially viable for the research project: MiniSAT[2] [58]; zChaff[3] [40]; jerusat [46]; BerkMIN [25]; Glucose[4] [2]. These may be scope for the research to simultaneously compare the efficacy of different solvers in varying conditions in the research project (e.g. size of level).ow

Features of modern DPLL-based SAT solvers

Modern DPPL-based SAT solvers are readily available and useful for this project. There are currently a number of highly scalable SAT solvers, all based on the classic DPLL search framework. These solvers, now also known as conflict-driven clause learning (CDCL) solvers, can generally handle problem instances with several million variables and clauses [35]. In the section below, a collection of important features have been identified and have relevance for the research project.

1. **Selection heuristic** : Also termed the 'decision strategy', this is the procedure of *how* variables are selected and assigned values. This aspect varies the most between different solvers can significantly impact the efficiency of a solver [40, 27, 68]. There are a range of strategies that can be employed including: Maximum occurrence in clauses of minimum size heuristic [34], Bohm's heuristic [40], dynamic largest individual sum heuristic [41], variable state independent decaying sum [44].

2. **Clause learning** : This method allows the solver to learn the causes of unsatisfiability (sometimes termed 'conflict') in clauses, using this information to minimise the search space [4]. When a conflicting clause is encountered, the

[2]http://minisat.se
[3]http://www.princeton.edu/ chaff/zchaff.html
[4]http://www.labri.fr/perso/lsimon/glucose/

solver will needs try to identify the cause of a conflict and attempt to resolve it. This is achieved by asserting that a solution does not exist in the particular search space, backtrack (undoing the previous decisions) and continuing the search in a new one. [68]. This feature has been identified as one that improves the underlying DPLL algorithm [68, 27].

3. **Conflict clause minimization** Een and Sorensson [58] had first explored the use of this method in their `MiniSat` solver. By utilising subsumption resolution, the size of the learned conflict is minimised through the removal of literals that are implied to be FALSE when the rest of the literals in the clause are FALSE [68]. This increased efficiency come with an increased computational cost [27].

4. **Conflict-directed backjumping** : Stallman and Sussman [60] proposed a method to allow a solver to backtrack directly to a decision-point p if the variables at level p or lower are causing conflict. The assumption is therefore that there is no solution to be found in this search space. It is agreed that this provides greater efficiency and completeness to the procedure [27].

5. **Fast backjumping** : Gomes *et al.* [27] describe this feature as allowing for a solver to directly go to a lower decision level if even one branch in the search space is in conflict. The authors note that this may not always increase efficiency however. The branch at depth level d is not marked as unsatisfiable but rather a new variable and value is selected for that level, continuing with a new clause. There has been some experimental work that suggests an increase in solving efficiency [27, 38].

6. **Watched literals scheme**: This feature monitors two classes of literals in an unsatisfied clause: TRUE or unassigned [44]. As an empty clause will cause the DPLL to stop, This feature had been introduced in the the `zChaff` solver and is utilised by many other solvers due to the efficiency of the constraint propagation [27]. This somewhat of an advancement of the 'lazy data structures' which had been introduced by the `Sato` solver [67]. This feature has furthered development of clause learning [68, 27].

7. **Assignment stack shrinking** : In the `Jerusat` SAT solver [46], Nadel had introduced this feature which is based on conflict clauses. Assignment stack shrinking servers to make the search area more local by 'shrinking' the conflict clause once it reaches a set threshold [47]. The 'shrinking' is achieved by finding the lowest decision level that is less than the immediate higher level by at 2. The solver will then backtrack to this level and where possible, sets unassigned literals of the clause to FALSE [47].

8. **Randomized restarts** : This provides that the clause learning algorithm can restart the branching process from decision level 0 while retaining all the learned clauses *et al.* [28]. Pioneered `zChaff` [44] many modern SAT solvers utilise highly aggressive restarts strategies, even below 20 backtracks, as it demonstrably lowers the solution time [27].

3.5 Chapter summary

Boolean satisfiability is the foundation for proving NP-hardness and NP- completeness for the project, as will be demonstrated in the next chapter. The availability of SAT solvers is also useful in order to try and test different ones with the research project's game engine. Currently in the available literature, there is very little research in this regard to the subject of linking SAT solvers and video game engines, therefore representing a fertile ground for development with this research project. The issue may be that these solvers will not scale well with the game engine. In that respect, Brummayer et al. [6] developed 'fuzz testing' techniques which allow for debugging errors in larger SAT instances. These techniques are useful in ensuring the scalability of the SAT solver when attempting to solve large levels.

Chapter 4

Computational complexity and the hardness of (video) games

> "I strongly approve the study of games of reason, not for their own sake, but because they help to perfect the art of thinking."
>
> Gottfried Wilhelm Leibniz

4.1 Overview

Having established the concepts and means of testing for certain classes of computational complexity in the previous chapters, this chapter will address the assessment of complexity in video games. There is a growth of interest and research in analysing 'mainstream' video games, including two- dimensional (2D) platform games [65, 23, 1, 54]. This chapter will identify the characteristics and tests required to class a game as computationally hard. This will inform the design considerations of the research project in Chapter 6.

Puzzles and games have long been a subject of complexity research with many games being found to be at least within NP-hard [37]. This is often based on the assumption that P $\neq$ NP [13]. The list includes a variety of grid and block placement games, wherein a player manoeuvres a block around a map to a specified location, such as *Minesweeper*[36] and *Tetris*[16]. Titles such as *Sokoban*, *Push* and *PushPush*, which have been shown to be not only NP- hard [14], but also PSPACE-hard [11, 18]. In the case of *PushPush*, there have been proofs of NP-hardness not only in 2D but in 3D as well [48]. More recently, Gualà *et al.* have demonstrated that a number of 'match-three' games, grid-based games which are popular on mobile devices, such as *Candy Crush Saga* and *Bejewled* are NP- hard [30].

The current trend in this field is to assess classic franchises such as *Prince of Persia*, and *Donkey Kong Country*. These studies are more relevant to the research project as visualisation will utilise a 2D platfomer resembling a game like *Super Mario Bros.*

4.2 2D platform games

The research project focuses on developing a two-dimensional (2D) platform game and thus the proof for NP-completeness is of particular interest. These games are (very often) single-player games wherein the player traverses levels in order to proceed to the next level. A 'level' is a virtual spaces in which the player character can manoeuvre and interact [7]. The player's game-play is influenced by the physics of the world, which often mean a limitation of jump height and distance [7].

Characteristics of 2D games

In an examination of 2D platform games, a list of common puzzle elements found within the genre [54, 23]. These include:

- **long fall**: the maximum height from which the player character may fall without receiving damage.
- **door opening**: the game world may include a variable number of doors and suitable mechanisms to open them
- **door closing**: the game world contains a mechanism to close doors and a way for the player to trigger the mechanisms
- **collecting items**: a set of items the player collects as part of the game. These may be necessary or optional content.
- **enemies**: Navigating the level may be made more challenging by having 'enemy' characters that must either be avoided or defeated by the player

4.2.1 Characteristics of NP-hard games

These above elements can be modeled into Boolean satisfiability problems, which are known to NP- complete [23, 9]. The proofs of video games rely on this analogy and if a proof then removes certain game elements, it is expected that the complexity of that game is reduced [65].

Viglietta [65] surveyed several classic video games games published between 1980 and 1998, identifying general common elements and schemes in video games that class them as computationally hard. These include collectable items, activation of pathways (e.g.by key or button) by, or the ability to destroy paths. Forišek [23] further identifies several features specific to 2D platform video games which provide for complexity, with particular focus on *Prince of Persia*. In the work, he posits a number of meta theorem, of which four are particularly relevant for the research project:

- **Meta-theorem 1**: A 2D platform game where the levels are constant and there is no time limit is in P, even if the collecting items feature is present.
- **Meta-theorem 2**: A 2D platform game where the collecting items feature is present and a time limit is present as part of the instance is NP-hard.

- **Meta-theorem 3**: Any 2D platform game that exhibits the features long fall and opening doors is NP-hard.
- **Meta-theorem 4**: Any 2D platform game that exhibits the features long fall, opening doors and closing doors is PSPACE-hard.

Following the criteria above, it can be demonstrated that many platform games are not only able to be classified as NP-hard but even PSPACE-hard in some cases.

Aloupis *et al* [1] surveyed the complexity of classic Nintendo franchises by considering that the games are essentially a decision problem of reachability: "given a level, is it possible to reach the goal point t from the start point s"? The authors also review subsequent *Super Mario Bros* titles as well as other platform games such as *Donkey Kong Country*.

In the work, the authors had generalised the map size and left all other elements of the game in the original settings. The proofs in [1] rely on a reduction of the game to 3SAT and the 'level' was modelled to the problem instance. This represents two potential points of difference from the current research project. First, the research goal is to develop a means to generate large levels and scale accordingly. Secondly, further reductions aside from 3SAT may be explored.

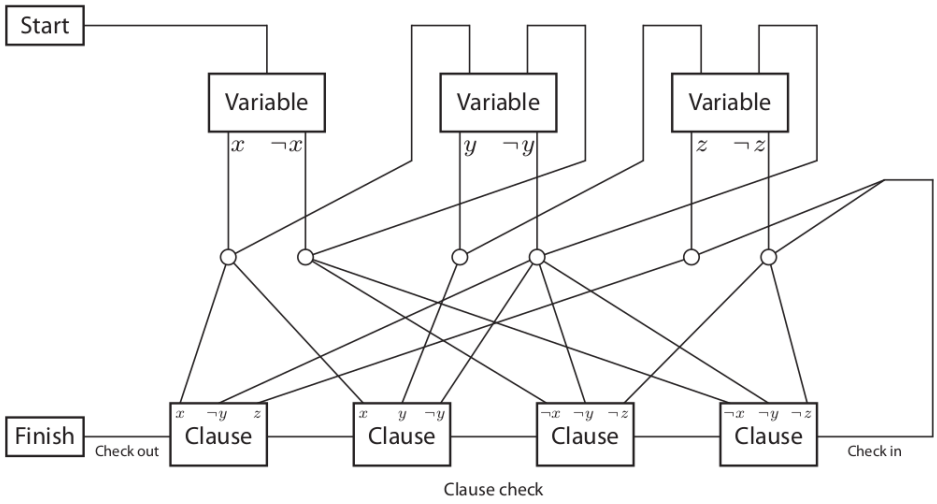

Figure 4.1: Framework for NP-hardness [1].

The framework for proving NP-hardness of games (see Figure 4.1) is reduced from 3SAT (described in Section 3.2.1 of this review). The player begins the level at the `START` gadget and then traverses through variable gadgets. A *variable gadget* requires the player to make an choice, which amounts to TRUE (x) or FALSE ($\neg x$) for the values in the formula. These decisions allows the player to follow paths leading to *clause gadgets*. A clause gadget creates a permanent state change and as a consequence, the player cannot access other paths connecting to that gadget. As can be seen in the framework above, there are several areas of potential crossover. To prevent such occurrence, *crossover gadgets* prevent a player from switching paths.

Once the player has passed through all the variable gadgets, the player can proceed to the `FINISH` through a clause check path. This can only be successfully traversed only if every clause gadget has been unlocked.

4.3 Chapter summary

The frameworks provided by [1] give a comprehensive model for the project's implementation. Modelling these gadgets to SAT clauses will form a significant area of the research project's work. By utilising the metatheorms provided by Forišek [23], it may be possible to incorporate other complexity levels including PSPACE-hard.

Having established how a 2D platformer may be classed as NP-hard (or even PSPACE- hard), the focus of the literature review shifts to the details of how these levels/problem instances will be generated, explored in the next chapter.

Chapter 5

Content generation

"Human beings are never more ingenious than in the invention of games."

Gottfried Wilhelm Leibniz

5.1 The fundamentals of content generation

One of the stated research project goals is to be able to generate large problem instances within the game engine. As mentioned in the previous chapter, within the research project's game engine, a 'level' is a problem instance for the SAT solver to assess if there is a satisfying assignment [1] .

Procedural content generation (PCG), refers to automating the creation of content for a game through the use of an algorithm [32, 63]. *Game content* refers to all game elements excluding the player's own actions and the game engine [7]. This can, and often does, include the layout of a level. As will be discussed in the sections below, there sizeable research interest in this field with many wide ranging impacts.

5.1.1 Overview

Content generation in video games combines a broad range of dynamic subject areas including graphics, image processing, mathematics, artificial intelligence, and ludology [32]. PCG is increasingly popular with recent titles employing elements of PCG to varying degrees. This is due to being able to reduce the prohibitive expense of manually creating game content [32] and the ability to generate new types of games based around content- generation [63]. From a game player perspective, a game that has infinite levels has greater potential replay value, especially if the game adapts to the player's playing style [66].

There are other, more practical reasons such as memory consumption. The content. This has been used to tremendous effect in *.kkrieger*, a first-person shooter game, whose executable was a mere 95 KiB [32].

5.2 Taxonomy of game elements

The literature in this area has its own particular definitions, which will be identified below and briefly discussed below.

Online vs offline generation

There is a distinction between. *Online content generation* occurs during a video game's runtime. This is in contrast to *offline content generation* which is performed during the development of the game. The requirements for effective online content generation are (i) that generation is fast (ii) the content is reliable and correct [63].

Necessary vs optional content

Another set of distinctions is between *necessary* and *optional* content. Necessary content is required for the progress and completion of the game. Optional content is content with which the player may wish to engage but does not prevent basic gameplay or completion of the game itself.

The importance is that necessary content must always be generated in a correct manner. For example, a level must not be unsolvable nor should the game have an undefeatable enemy, if such things prevent essential gameplay or render the game impossible to complete.

5.2.1 Challenges in PCG

Content generators can automatically create a large amount of differing levels in a short amount of time but this is not indicative of the *quality* of the generated content [55]. Much of the literature surveyed has identified a lack of reliability and consistency in quality as a main challenge area and barrier to wider adoption of PCG in commercial game offerings. Zafar and Mujtaba [66] assert that the majority of PCG techniques are characterised by *catastrophic failures* which make them unsuitable for wider commercial deployment. These failures affect the reliability and accuracy of the game. The work identifies the following as failures in 2D platformers:

- height of ground;
- height of ground and optional content;
- simple level generation
- unplayable levels
- placement of the character's start position

The generation algorithms may be tested through the use of *fitness functions*, which are objectively evaluates how closely a proposed solution achieves a particular design aim. There is a question about which fitness functions are useful and what sort of rules should be employed when generating level. For example, Togelius *et al.* [61] developed

personalised fitness functions that would automatically generate different racetracks based on the player's actions and skill.

5.3 Content generation methodologies

There are several approaches available for content generation which will be considered for the design implementation of the research project. Three differing approaches will be surveyed and compared and evaluated for suitability for use in the research project (see Figure 5.1).

The first approach is a constructive approach wherein an algorithm generates content based around a series operations that ensure that content is produced according to the definitions of rules. In this way, the content generated is 'correct' insofar as adherence to the specified rules [5]. The algorithm does not test that the content has in fact adhered to the rules however. While this method is restrictive, it has been used to generate in-game content such as terrain [43].

Another approach is 'generate-and-test'. As suggested by the name, there are two parts to this algorithm: After the content is generated, it is then tested against constraints and assessed for compliance [63]. Content that does not pass assertion against these criteria is discarded and is regenerated until it passes [63].

Search based generation is a specialised type of generate-and-test procedure. Rather than merely accept or reject the generated content, the candidate content is tested through a fitness function. The generation of new content is dependent on the fitness value assigned to the prior evaluated instances. The goal is to produce content that is qualitatively of increased value [63].

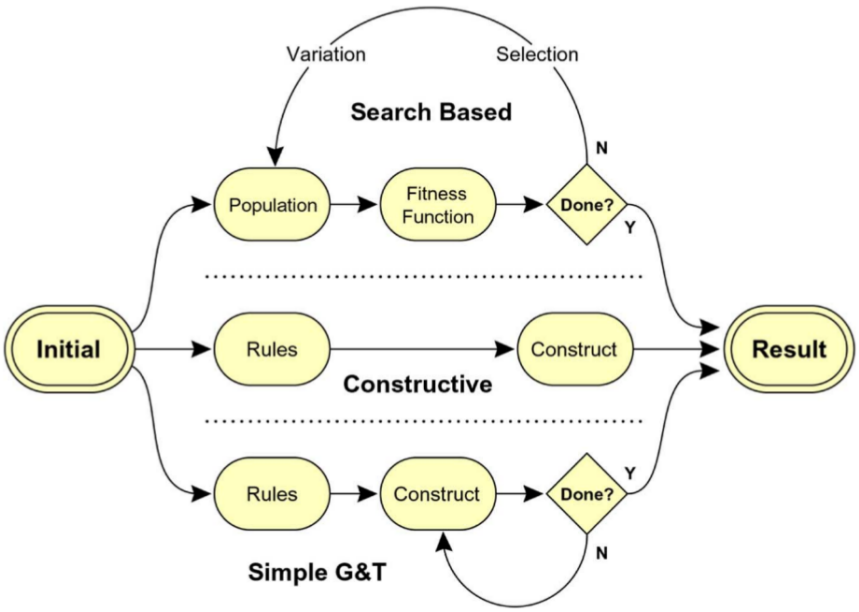

Figure 5.1: Representation of three types of level generation [63].

5.3.1 PCG heuristic techniques

In each of the approaches above, there exits a difference between deterministic and stochastic methods of generation. In the former, the generation tool will generate specified content on each output. In a stochastic method, the results are somewhat random.

Dormans and Bakkes [20] presented a generative grammar which allows for the generation of levels which are 'correct' on a syntactic level. This was achieved by dividing the content into mission and spaces. The missions (or game tasks) are generated through the use of recursive, non-linear graphs. From this output, the structure and shape of the level are constructed from a separate 'shape' grammar. The authors also utilised differing player models in order to dynamically adjust difficulty, resulting in more adaptive levels.

To ensure efficient level generation, both Compton and Mateas [8] and Smith *et al.* [55] generate levels for 2D platform games by first splitting the levels into smaller sections. From a platform tileset, those smaller sections could be generated. This is the concept of 'rhythm'-based generation: the pattern is derived from how the game is to be played. This allows for a range of the types of generated levels as well as maintaining the pacing of the game [55]. The Figure 5.2 below illustrates the process.

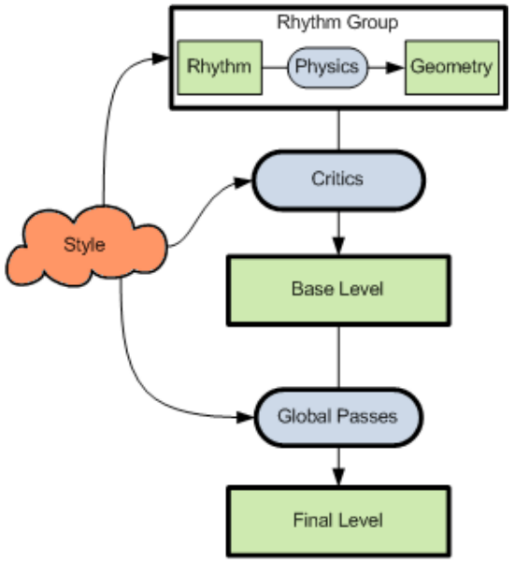

Figure 5.2: Level generation algorithm from [55]. The authors describe the green items as the generated content and blue boxes as constraints in order to ensure quality generation that adheres to specified criteria.

In a somewhat different fashion, the research from Jennings-Teats *et al.* [33] and Shaker *et al.* [52] demonstrate that a player's actions can be used for online generation of platform levels. This would entail that the game would model itself on the user's own style of playing. This is a more focused application of the approach used in Dormans' implementation [19], which uses a pre-defined grammar to generate the mission structure. This is then implemented into the 2D-level using another grammar

to translate this structure.

In a survey on the the evolution of game designs, Togelius and Schmidhuber [62] posit that 'fun' in a game is derived from learning how to play and ultimately mastering the mechanics. While this may be true to an extent [7], the declaration and incorporation of these constraints into an algorithm for generation is limited insofar as the design matches human experience [57].

Mourato *et al.* [45] proposed using a genetic algorithm in order to generate game content for human-players. A genetic algorithm is a search heuristic that generates approximations through selection and mutation [59]. The authors achieved modest but positive results including increased level variety and ability to add necessary and optional content correctly. This method is not entirely suitable for the current research project as the authors' goals has been to

Smith *et al.* [56] presented a design tool (TANAGRA) that merged the input from the game designer and the PCG algorithm to produce levels for a 2D platform game. The technique allowed the designer to establishes rules and constraints to specify specific content, but allows the algorithm to decide that which is unspecified. The authors content that this provides for more diversity in level design as well as enhanced entertainment value.

A rhythm-based technique is common approach to automatic level generation. Mawhorter and Mateas [42] devised the 'occupancy regulated extension' algorithm which stitches together samples (referred to 'chunks') authored by human developers into a playable level. This method described here could be combined with. This is further elaborated upon in the next chapter on design considerations.

5.4 Chapter summary

Level generation is a dynamic, growing area of video games research. There exist a number of different methodologies and approaches appropriate for the stated goals of the research project. For the purposes of the research project, many of these approaches are not necessarily applicable in initial design and implementation. Many studies, such as [57], [62], and [56] focus on creating varied content that is focused on player experience. This may not be an immediate priority in the research project as the focus is to be able to generate large levels that correctly represent SAT expressions.

To this end, the rhythm-based approach [8, 55] and 'chunk approach' [42] are most pertinent as it would allow various gadget clauses to be pre-created and joined together to make larger scale levels to be solved. The creation of an efficient algorithm (even a genetic one [45]) represents an area of future research. This will be further explored in the next chapter on design considerations.

Chapter 6

Level Design and Generation

In the previous chapters, the basis for making a 2D platform game NP-hard was established. In this chapter, the focus will be on designing a playable level that incorporates those elements.

In previous literature, the level design has focused on the gadgets that correspond to 3SAT reductions in order to prove that the game is in fact NP-hard.. In these models, the actual gameplay is generalised and left-otherwise unaddressed.

This provides a challenge when

6.1 Designing NP-hard levels

For the purposes of this project, the size of the levels are n X n, which means that the size is essentially unlimited.

Figure 6.1demonstrates how the gadgets may be phsyically linked together in a three variable, three clause 3SAT reducation.

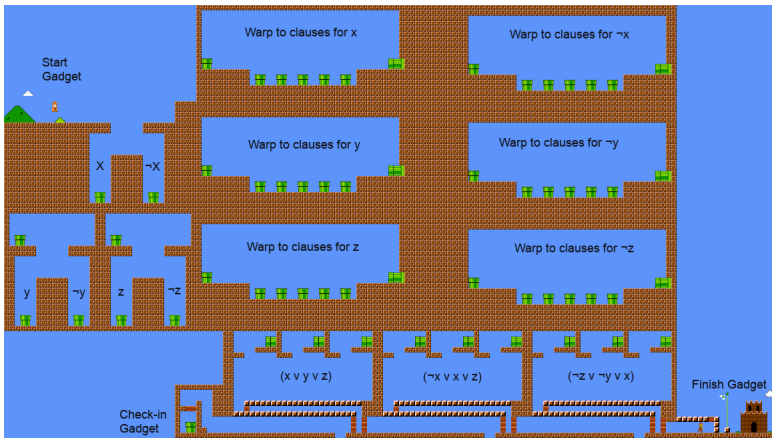

Figure 6.1: Representation of physical linking of gadgets.

This will be briefly described below.

The player starts as in Aloupis et al [1] with a start gadget that turns 'small' Mario (composed of a single 16x16 tile) into Super Mario. Super Mario can jump under bricks

and break certain types of blocks. Once sufficiently progressed to the right, the screen view will bring the first variable gadget into focus.

The variable gadget relies on the meta-theorem provided by Forišek [23] (long fall). In Aloupis et al [1], the authors created a cross-over gadget that prevented the player from by- passing or accessing areas of the level which impact the reducation to a 3SAT instance. The authors are not entirely clear exactly how the gadgets are physically linked together. This presentes a number of challenges when scaling a level. In Aloupis, the 3SAT instance has three clauses. However, when dealing the amount of cross over increases.

There are a number of issues which arise when scaling a level.

In *Super Mario Bros.*, the player may discover that certain pipes are able to be utilised in order to travel to different hidden areas or even other levels. This game mechanic provides a useful solution to scalability.

In the project's implementation, after the player picks a variable by falling and using the pipe, the player is taken to a clause choosing gadget. Here there are a series of pipes, each leading to a particular clause where the literal appears.

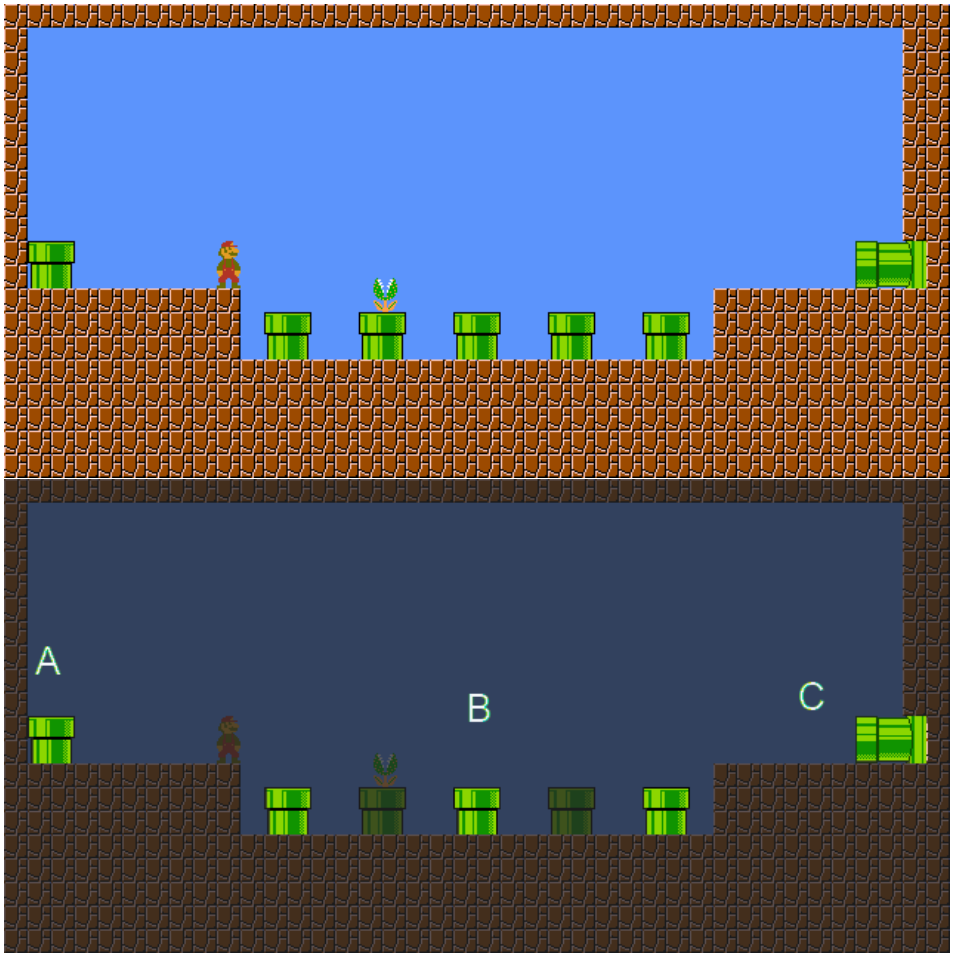

Figure 6.2: Crossing into other clause areas.

In the clause-crossing gadget, the player enters on the pipe on the left (marked A).

The player is not able to return through this pipe. Even if it was the case, the player cannot overcome the variable gadget.

There are five pipes of which three lead to clauses and the other two contain *Piranha plants*, venus-fly trap like enemies in *Super Mario Bros* (Marked B). Once the clauses have been visited and unlocked, the player returns through the pipe on the right side of the screen (marked C) and can proceed to the next variable assignment. If the player has no more variable assignments to make, the pipe will lead to the check-in gadget, illustrated in Figure 6.3. The check-in gadget allows the player to progress through the check-out and ultimately to the finish gadget.

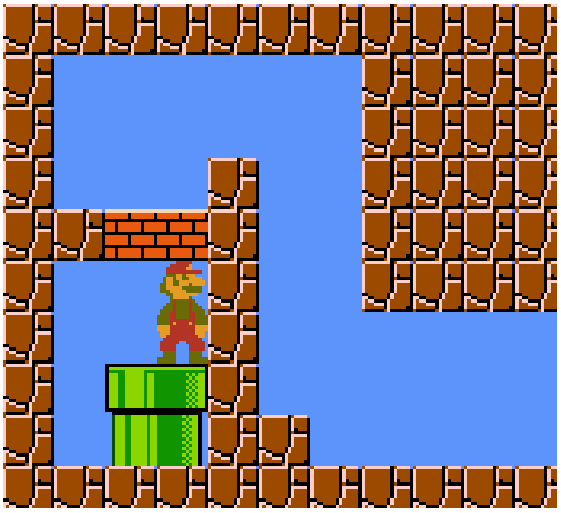

Figure 6.3: The check-in gadget. Note the player can only advance as Super Mario.

Once the player checks-in, the player then advances through the check-out of the level. The check-out phase is only passable if the clauses have been opened. The only obstacle at this point of the game is the drop in the floor. The player then proceeds to the finish gadget, shown in Figure 6.4.

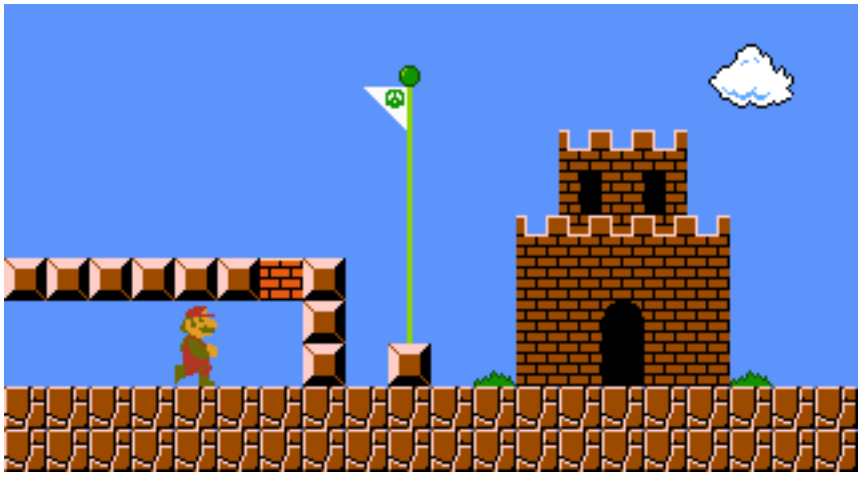

Figure 6.4: The check-in gadget. Note the player can only advance as Super Mario.

Chapter 7

Game Engine Design

7.1 Design considerations

7.1.1 Development language and libraries

The game was developed in C++ using the C++11 standard. The C++ language was selected because this is the same language used in the source code for SAT solvers. This ensures that the code base can easily be adapted into the game engine without the need to develop an API or wrapper, which would have detracted from time developing the game engine.

The game engine uses the Simple and Fast Media Library (SFML) for handling graphics and player input. There are a number of libraries which can provide this functionality. The primary criteria in selecting a suitable multimedia library was based on ease of use and ease of integration.

The candidate libraries were SFML, SDL, and Allegro, as these were the most popular choices for C++ game development. The decision to use SFML was based on the use of C++. SFML distinguishes itself from the other two libraires in that it is written for C++ and thereby provinding a way to use the multimedia library in an object-orientated fashion. Another strong choice was SDL. However, SDL lacks the support for object-orientation that SFML provides. At any rate, the multimedia library is not of any significant impact or consequence.

The game was developed on Ubuntu Linux 15.04 with SPECS. It was compiled using LLVM/Clang (clang++). There is no appreciable difference when using the GNU Compiler Collection (gcc). Instructions to switch the makefile to gcc rather than clang++ are contained in the wiki as well as the comments within the project's makefile.

The coding style used for the development was the Google C++ style guide. The intention is that this research will continue to be developed after this project's completion and that future work may be continued on the same code base. This particular style guide was chosen because it is widely available and provides good references.

7.2 Basic game engine mechanics

7.2.1 Game loop

7.2.2 Player movement

7.2.3 Collision management

7.3 Level generation

The levels are drawn using a tile-based method. There are a number of methods that could be used

7.4 SAT solver integration

The Game State Manager class has a private member called `SAT_Manager_` which integrates the zchaff SAT solver into the game engine.

7.4.1 Reading in the CNF file

7.4.2 Accessing variables and clauses

7.5 Project wiki and references

A project wiki has acted as a journal and reference guide for the game engine's development. The rationale behind keeping this documentation was to allow for future development work to be carried out on the game engine.

7.6 Unit testing

The game engine source code also includes a testing framework for the various modules. In this project, googletest, Google's C++ Testing Framework was used.

Chapter 8

Evaluation

Chapter 9

Future Work

9.1 Constrained level size

9.2 Additional complexity classes

9.2.1 PSPACE

9.2.2 EXSPACE

9.3 Web Deployment

Bibliography

[1] G. Aloupis, E. D. Demaine, and A. Guo. Classic Nintendo games are (NP-)Hard. *CoRR*, abs/1203.1895, 2012.

[2] G. Audemard and L. Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[3] T. Balyo. *Solving Boolean satisfiability problems.* PhD thesis, Charles University, Prague, 2010.

[4] A. Biere, M. Heule, H. van Maaren, and T. Walsh. Conflict-driven clause learning sat solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[5] C. Browne. *Automatic generation and evaluation of recombination games.* PhD thesis, Queensland University of Technology, 2008.

[6] R. Brummayer, F. Lonsing, and A. Biere. Automated testing and debugging of SAT and QBF solvers. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 44–57. Springer, 2010.

[7] K. Burgun. *Game Design Theory: A New Philosophy for Understanding Games.* Taylor & Francis, 2012.

[8] K. Compton and M. Mateas. Procedural level design for platform games. In *AIIDE*, pages 109–111, 2006.

[9] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

[10] S. A. Cook. Can computers routinely discover mathematical proofs? *Proceedings of the American Philosophical Society*, pages 40–43, 1984.

[11] J. Culberson. Sokoban is PSPACE-complete. In *Proceedings in Informatics*, volume 4, pages 65–76, 1999.

[12] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[13] E. D. Demaine. Playing games with algorithms: Algorithmic combinatorial game theory. In *Mathematical Foundations of Computer Science 2001*, pages 18–33. Springer, 2001.

[14] E. D. Demaine, M. L. Demaine, and J. O'Rourke. PushPush and Push-1 are NP-hard in 2d. *arXiv preprint cs/0007021*, 2000.

[15] E. D. Demaine, R. A. Hearn, and M. Hoffmann. Push-2-f is PSPACE-complete. In *CCCG*, pages 31–35, 2002.

[16] E. D. Demaine, S. Hohenberger, and D. Liben-Nowell. Tetris is hard, even to approximate. In *Computing and Combinatorics*, pages 351–363. Springer, 2003.

[17] K. Devlin. *The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time*. Basic Books, 2002.

[18] D. Dor and U. Zwick. Sokoban and other motion planning problems. *Computational Geometry*, 13(4):215–228, 1999.

[19] J. Dormans. Adventures in level design: generating missions and spaces for action adventure games. In *Proceedings of the 2010 workshop on procedural content generation in games*, page 1. ACM, 2010.

[20] J. Dormans and S. Bakkes. Generating missions and spaces for adaptable play experiences. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):216–228, 2011.

[21] D.-Z. Du and K.-I. Ko. *Theory of computational complexity*, volume 58. John Wiley & Sons, 2011.

[22] B. Esfahbod. Euler diagram for P, NP, NP-complete, and NP-hard problem. `http://commons.wikimedia.org/wiki/File:P_np_np-complete_np-hard.svg`, 2011. This file is licensed under the Creative Commons Attribution-Share Alike 3.0 Unported and 2.5 Generic and 2.0 Generic and 1.0 Generic license.

[23] M. Forišek. Computational complexity of two-dimensional platform games. In *Fun with Algorithms, 5th International Conference, FUN 2010, Ischia, Italy, June 2-4, 2010. Proceedings*, pages 214–227, 2010.

[24] W. I. Gasarch. Guest Column: The Second P =?NP Poll. *SIGACT News*, 43(2):53–77, June 2012.

[25] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat-solver. *Discrete Applied Mathematics*, 155(12):1549 – 1561, 2007. {SAT} 2001, the Fourth International Symposium on the Theory and Applications of Satisfiability Testing.

[26] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, New York, NY, USA, 1 edition, 2008.

[27] C. P. Gomes, H. Kautz, A. Sabharwal, and B. Selman. Satisfiability solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[28] C. P. Gomes, B. Selman, H. Kautz, et al. Boosting combinatorial search through randomization. *AAAI/IAAI*, 98:431–437, 1998.

[29] J. Gregory. *Game Engine Architecture*. Taylor & Francis Ltd., First edition, 2009.

[30] L. Gualà, S. Leucci, and E. Natale. Bejeweled, candy crush and other match-three games are (np-)hard. *CoRR*, abs/1403.5830, 2014.

[31] S. Hahn. *NIkki and the Robots*. `https://github.com/nikki-and-the-robots`, 2012. Date Accessed: 2015-04-28.

[32] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, Feb. 2013.

[33] M. Jennings-Teats, G. Smith, and N. Wardrip-Fruin. Polymorph: dynamic difficulty adjustment through level generation. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games*, page 11. ACM, 2010.

[34] R. G. Jeroslow and J. Wang. Solving propositional satisfiability problems. *Annals of mathematics and Artificial Intelligence*, 1(1-4):167–187, 1990.

[35] H. Katebi, K. A. Sakallah, and J. P. Marques-Silva. Empirical study of the anatomy of modern sat solvers. In *Theory and Applications of Satisfiability Testing-SAT 2011*, pages 343–356. Springer, 2011.

[36] R. Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22(2):9–15, 2000.

[37] G. Kendall, A. J. Parkes, and K. Spoerer. A survey of NP-complete puzzles. *ICGA Journal*, 31(1):13–34, 2008.

[38] S. Kottler. Sat solving with reference points. In *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 143–157. Springer, 2010.

[39] L. A. Levin. Universal sequential search problems. *Problemy Peredachi Informatsii*, 9(3):115–116, 1973.

[40] J. Marques-Silva. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence*, pages 62–74. Springer, 1999.

[41] J. P. Marques-Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, 1999.

[42] P. Mawhorter and M. Mateas. Procedural level generation using occupancy-regulated extension. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 351–358. IEEE, 2010.

[43] G. S. P. Miller. The definition and rendering of terrain maps. *SIGGRAPH Comput. Graph.*, 20(4):39–48, Aug. 1986.

[44] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[45] F. Mourato, M. P. dos Santos, and F. Birra. Automatic level generation for platform videogames using genetic algorithms. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*, page 8. ACM, 2011.

[46] A. Nadel. The jerusat SAT solver. *Master's thesis, Hebrew University of Jerusalem*, 2002.

[47] A. Nadel and V. Ryvchin. Assignment stack shrinking. In *Proceedings of the 13th International Conference on Theory and Applications of Satisfiability Testing*, SAT'10, pages 375–381, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] J. O'Rourke and T. Group. PushPush is NP-hard in 3D. *arXiv preprint cs/9911013*, 1999.

[49] C. H. Papadimitriou. Computational complexity. In *Encyclopedia of Computer Science*, pages 260–265. John Wiley and Sons Ltd., Chichester, UK.

[50] C. H. Papadimitriou. *Computational complexity*. John Wiley and Sons Ltd., 2003.

[51] F. Richter. *Secret Maryo Chronicles*. `http://www.secretmaryo.org`, 2013. Date Access: 2015-04-27.

[52] N. Shaker, G. N. Yannakakis, and J. Togelius. Towards automatic personalized content generation for platform games. In *AIIDE*, 2010.

[53] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 2012.

[54] G. Smith, M. Cha, and J. Whitehead. A framework for analysis of 2d platformer levels. In *Proceedings of the 2008 ACM SIGGRAPH Symposium on Video Games*, Sandbox '08, pages 75–80, New York, NY, USA, 2008. ACM.

[55] G. Smith, M. Treanor, J. Whitehead, and M. Mateas. Rhythm-based level generation for 2d platformers. In *Proceedings of the 4th International Conference on Foundations of Digital Games*, FDG '09, pages 175–182, New York, NY, USA, 2009. ACM.

[56] G. Smith, J. Whitehead, and M. Mateas. Tanagra: Reactive planning and constraint solving for mixed-initiative level design. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):201–215, 2011.

[57] N. Sorenson, P. Pasquier, and S. DiPaola. A generic approach to challenge modeling for the procedural creation of video game levels. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):229–244, 2011.

[58] N. Sorensson and N. Een, N.n. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005:53, 2005.

[59] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *Computer*, 27(6):17–26, 1994.

[60] R. M. Stallman and G. J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial intelligence*, 9(2):135–196, 1977.

[61] J. Togelius, R. De Nardi, and S. M. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259. IEEE, 2007.

[62] J. Togelius and J. Schmidhuber. An experiment in automatic game design. In *Computational Intelligence and Games, 2008. CIG '08. IEEE Symposium On*, pages 111–118, Dec 2008.

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

[64] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pages 466–483. Springer, 1983.

[65] G. Viglietta. Gaming is a hard job, but someone has to do it! *Theory of Computing Systems*, 54(4):595–621, 2014.

[66] A. Zafar and H. Mujtaba. Identifying catastrophic failures in offline level generation for mario. In *Frontiers of Information Technology (FIT), 2012 10th International Conference on*, pages 62–67, Dec 2012.

[67] H. Zhang. Sato: An efficient prepositional prover. In *Automated Deduction—CADE-14*, pages 272–275. Springer, 1997.

[68] L. Zhang and S. Malik. The quest for efficient Boolean satisfiability solvers. In *Computer Aided Verification*, pages 17–36. Springer, 2002.