Video Game Motion Planning reviewed NP-complete

Cory Gabrielsen

June 4, 2012

Contents

1 Introduction

A number of recent papers have proved NP-hardness of a variety of "sliding block" puzzles [2] and another, [1], has expanded this classification to some of Nintendo's most cherished video game franchises—Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. We consider the problem of motion planning, where a robot or virtual avatar plans a path in the presence of movable objects. Namely, we prove that motion planning in generalized versions of these classic video games is NP-hard. In particular, our results for Mario apply to the NES games Super Mario Bros., Super Mario Bros.: The Lost Levels, Super Mario Bros. 3, and Super Mario World; our results for Donkey Kong apply to the SNES games Donkey Kong Country 1-3; our results for Legend of Zelda apply to all Legend of Zelda games except the side-scrolling Zelda II: The Adventure of Link; our results for Metroid apply to all Metroid games; and our results for Pokémon apply to all Pokémon role-playing games.

For these games, we consider the decision problem of reachability; that is, given a stage or dungeon of a game world, is it possible to reach the goal point t from start point s? Our results apply to generalizations of the games, but we limit our generalizations to map size, leaving all other game mechanics unchanged. All of our NP-hardness proofs are by reduction from SAT. Further, the proofs for Mario, Donkey Kong, Legend of Zelda, and Metroid rely on a common construction. We do not review the proof for Legend of Zelda or Pokémon, which are based off a reduction to Push-1 [2] .

2 Review

We review basic concepts from first-order logic and theory of computation, establishing the language for the paper. We assume the reader is familiar with only the most basic concepts, deferring a full introduction to [3] and [4].

2.1 Definitions

- A *variable* corresponds to a decision and may be assigned the value 'True' or 'False'.
- A *logical connective* is a binary operator between two variables that returns True or False. We consider the operators AND ($\wedge$), OR ($\vee$), and NOT ($\neg$), all defined in the usual manner.
- A *literal* is a variable or its negation. We focus on evaluation over notation, so this shortcut suffices.
- A *clause* is a finite disjunction of literals, which we write as $l_1 \vee \cdots \vee l_n$, where l_i are literals.
- A clause is *satisfied* whenever one of its variables is assigned True; or equivalently, the clause evaluates to True given an assignment of its variables.
- A *boolean formula* is expressed as logical connectives between literals and clauses, evaluating True or False. Every boolean formula can be represented as a conjunction of clauses, called *conjunctive normal form.*
- A set of objects is called *recursive*, *computable*, or *decidable* if there is an algorithm which terminates after a finite number of steps and correctly decides whether or not a given object belongs to the set.
- An algorithm runs in *polynomial time* if given input size n, the algorithm terminates in $2^{O(log(n))} = O(p(n))$ steps were p is a real-valued polynomial.

- A problem is *intractable* if it is solvable in theory, but in practice, takes too long to be useful. More precisely, we say that a problem is intractable for all but the smallest inputs if it lacks a polynomial-time solution.
- An algorithm is *deterministic* if it behaves predictably. Given a particular input, the algorithm will always proceed through the exact same sequence of steps and produce the same output.
- Similarly, an algorithm is *nondeterministic* if it can behave differently on different runs of the same input. Given a specific input, a nondeterministic algorithm is neither guaranteed to proceed through the same sequence of steps nor guaranteed to produce the same output. Nondeterminisim is a generalization of determinism, so every deterministic algorithm is automatically a nondeterministic algorithm.
- *Motion planning* is the process of detailing a task into discrete motions. We will show that this problem is NP-complete in the video game problem variants we consider.

2.2 Decision Problems

A *decision problem* is an arbitrary question on an infinite set of inputs for which the answer is "yes" or "no". For instance, the question, "is this number prime?" is a decision problem. Contrarily, the question, "how many primes are less than 2^{32}?" is not a decision problem. It is conventional to interpret, or even define, a decision problem to be equivalent to the set of inputs for which the problem returns yes. As such, we say that a decision problem A is *decidable*, *solvable*, or *computable* if A is a decidable set.

2.3 P vs. NP

P and NP are both sets of decision problems. P is the set of decision problems that can be solved by a deterministic algorithm in polynomial time. NP is the set of decision problems that can be solved by a nondeterministic algorithm in polynomial time. Recall from earlier that every determinisitic algorithm is also nondeterministic. Hence, P $\subseteq$ NP. The question, "Is P = NP ?" is considered a major unsolved problem in computer science. In fact, it is one of the seven Millenium Prize Problems selected by the Clay Mathematics Institute to carry a US$ 1,000,000 reward for the first correct solution, which has helped generate significant interest and research regarding the subject. It is conjectured that P $\neq$ NP.

2.4 NP-hard and NP-complete

NP-hard is the set of problems for which a solution algorithm can be translated into an algorithm for solving any NP problem. It is intuitivive to think of NP-hard as the set of problems that are at least as hard as the hardest problems in NP, though they may in fact be more difficult. We note that NP-hard problems may be decision problems, search problems, or optimization problems.
With this in mind, we define NP-complete as the set of decision problems for which a solution can be translated in polynomial-time into one for solving any NP problem. As such, NP-complete is the intersection of NP and NP-hard. In particular, if any NP-complete problem can be solved in polynomial-time, then every problem in NP can be solved in polynomial-time, which would imply P = NP. Clearly, no such solution to any of the more than 3000 known NP-complete problems has been found despite decades of exhaustive research.

2.5 Boolean Satisfiability Problem

SATISFIABILITY (SAT) is the problem of determing if given a boolean formula, there exists an assignment of variables such that the formula evaluates to True. Of course, the complimentary problem of determining whether no such assignments exists is equally important. This would imply the formula evaluates to False for all possible variable assignments. Stephen Cook proved Cook's theorem in 1971, which states SAT is NP-complete. It was the first known example of an NP-complete problem at the time, sparking some of the most notable and active research in computer science since. Importantly, a wide array of natural or interesting decision and optimization problems can be transformed or reduced to an instance of SAT. Likewise, k-SAT is a special case of SAT where each clause contains exactly k literals. We will reduce our motion planning problems to 3-SAT.

3 Framework

We consider the general decision problem of determining whether it is possible to travel from a start point to a given goal point in a platform game world. The platform game genre is characterized by virtual avatars maneuvering around movable enemies and obstacles with the goal of reaching the end of each level. If the player completes every level, they win the game, so reachability is a natural problem to consider.

3.1 SAT Reduction

We review the reduction from SAT posed in [NVG]. We begin with a boolean formula in conjunctive normal form, $C_1 \vee \cdots \vee C_n$, and consider its satisfiability problem. We will reduce the problem to a set of "gadgets" and/or "units". We note that these gadgets and units will represent an instance of the problem we are reducing to, rather than the original problem itself. In this sense, our reduction maps an instance of SAT to an instance of a game-world reachability problem by mapping individual clauses in the original boolean formula to gadgets and units in the game. By showing we can implement these gadgets in our games, we will have achieved the desired classification.

In all the generalized game worlds, players control an avatar in a bounded world. We constrain the map sizes and shapes but leave all other gameplay mechanics unchanged. Intuitively, we aim to construct "corridors" surrounded by fixed regions to guide the avatar's activity. The basic idea is to define Variable units that force the avatar to make a choice between two paths, which corresponds to assigning variable x_i to True or False. Every Variable unit connects to each Clause unit in which it appears. The Variable units are arranged in a linked chain that must be traversed in order, after which the Clause units must be visited in order. We wish to translate the each clause in the original boolean formula to a Clause unit impassable unless it was earlier visited from a Variable unit, hence the corresponding clause is satisfied by our definition. The only paths from s to t force the avatar to traverse all Variable units and then all Clause units, so every clause must be satisfied if t is reachable. A complete construction of four clauses is shown in Figure 1. The left is unsatisfiable; the right is satisfiable:

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \tag{1}$$

$$(x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_3) \tag{2}$$

Figure 1 diagrams the atomic components of the construction. We will duplicate the functionality in the games we consider.

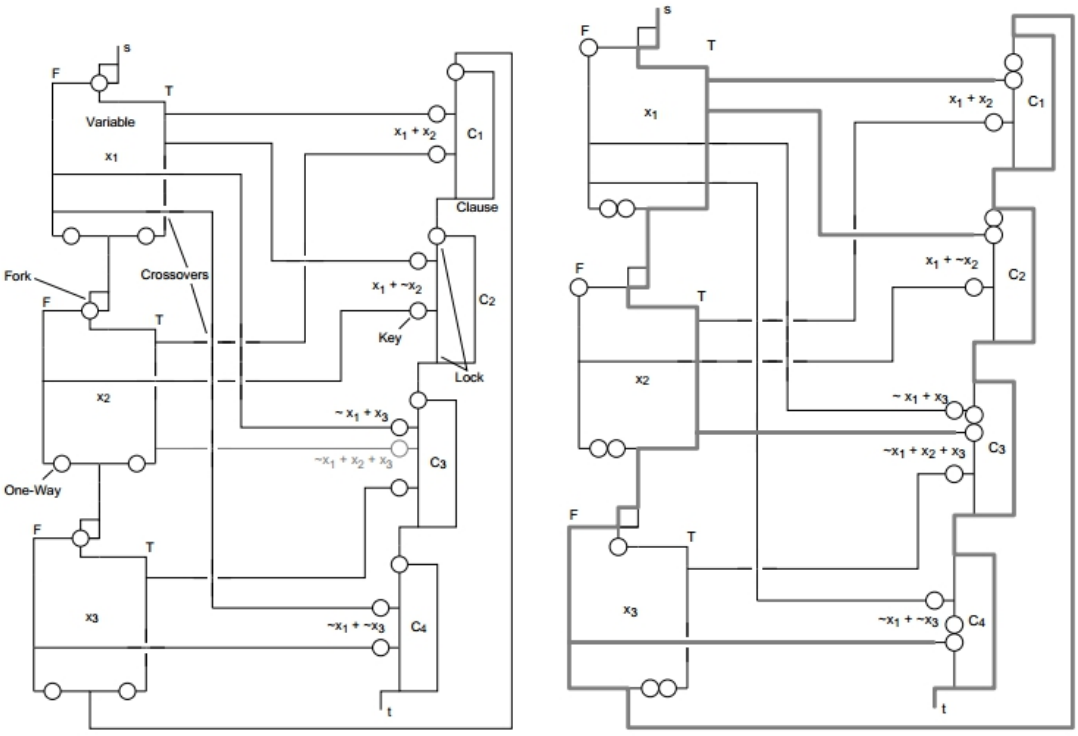

Figure 1: This is a complete construction for a simple game of pushing blocks discussed in [2]. As the avatar traverses the path, it may push each circlular obstacle once in one direction. So here, the avatar's goal is to move from s to t by moving blocks in the correct direction amongst two options. Variable gadgets are on the left with paths leading towards the Clause gadgets on the right. For any Clause unit, if any of its literals are satisfied, the corresponding obstacle will slide right, thereby blocking the obstacle encountered later in the solution path. This allows traversal through the Clause units. The right image shows a solution path.

1. *Variable* units where passage by an avatar assigns True or False.
2. *Fork* gadgets, which force the variable-setting binary choice.
3. *One-Way* gadgets that permit passage in only one direction.
4. *Clause* units traversable only if one of their literals is True.
5. *Lock and Key* mechanisms that prevent passage unless a key has been "unlocked".
6. *Crossover* units that allow two corridors or paths to cross without the possibility of leakage from one to the other.

This completes the reduction. We have mapped an instance of SAT into the task of implementing these constructions in a game world.

3.2 Platform Game Framework

Motivated by the previous reduction, we use a general framework for proving the NP-hardness of platform games, as illustrated in Figure 2. By invoking the framework, we need only construct the necessary gadgets for each game. The framework reduces from 3-SAT. Here, the player's virtual avatar begins at the position denoted Start, proceeds to the Variable gadgets wherein the player assigns the variable True (x) or False ($\neg x$). Within the game world, these paths of assignment may cross, so we implement Crossover gadgets that prevent the avatar from switching paths. By visiting a Clause gadget, the avatar can "unlock" the clause (a permanent state change), but cannot reach any of the other paths connecting to the Clause gadget. Lastly, after the traversing through all of the Variable gadgets, the avatar must traverse an often long "check" path, which passes through each Clause gadget, to reach the Finish position. Hence, the avatar may proceed through the entire check path if and only if each clause has been unlocked by some literal. So, it suffices to implement Start, Variable, Clause, Finish, and Crossover gadgets to prove NP-hardness of our games.

We will also show our problems are in NP, but the proofs are independent of the framework.

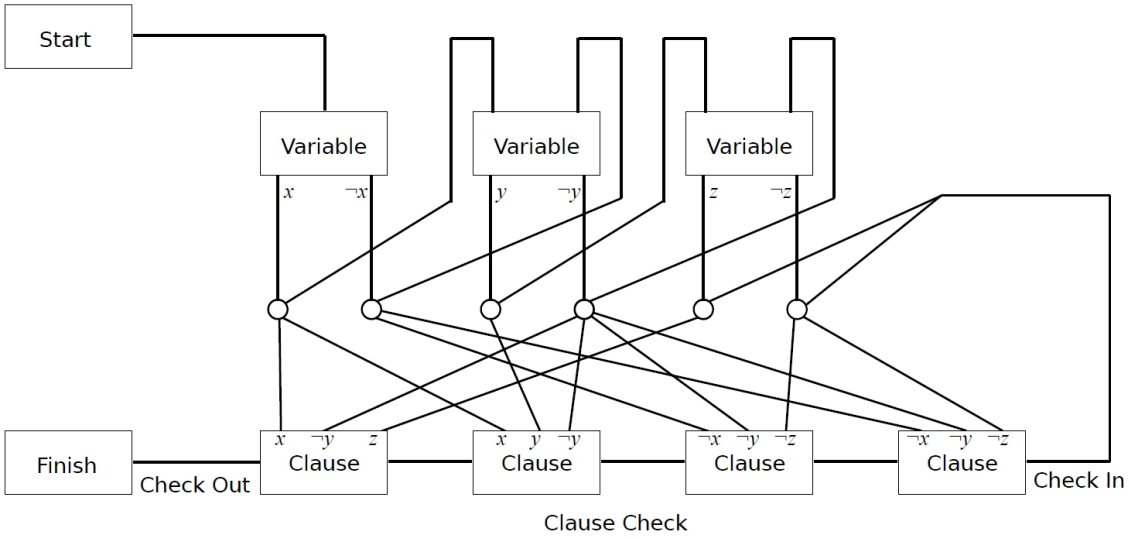

Figure 2: General framework for platform games

3.3 Gadgets

Start and Finish. The Start gadget contains the starting point for the avatar. The Finish gadget contains the goal for the avatar. In most of our reductions, these prove trivial to implement. In some cases, however, we require that the avatar maintain a certain in-game state throughout the construction. We force this requirement by making the Finish accessible only if the avatar is in a desired state and allowing this state to be entered at the Start. Specifically, for our generalized version of Mario, we require Mario to be "big" (which we define later) throughout the stage. So, we put a Super Mushroom power-up at the Start, allowing Mario to enter the Super Mario state, and a brick block at the Finish, which may only be broken by Super Mario.

Variable. We force the player to make a choice between two paths. Once one path is taken, the other path is forever inaccessible. The paths will correspond with variable x_i or its negation $\neg x_i$ being chosen as the satisfied literal. Each Variable gadget is accessible only from the previous

Variable gadget, independent of the choice of path made in the previous gadget, and connected in such a way that entering one literal disallows traversal back into its negation, as required.

Clause and Check. Each Clause gadget is accessible from and only from the literal paths corresponding to the literals appearing in the clause in the original boolean formula. Further, when the avatar visits the Clause gadget, they may perform an action which "unlocks" the Clause. The Check must pass through every Clause so that the avatar may pass through the Clause via the Check if and only if the Clause has been unlocked. The Check is accessible only if all the Clause gadgets have been visited from Variables. If the avatar traverses the entire Check, then they may access the Finish.

Crossover. The Crossover must allow traversal via two paths that cross each other in such a way that there is no leakage between the two unless both passages have already been traversed. We remark that the Crossover gadgets need only be *unidirectional*. That is, each of the two paths need only be traversed in one direction. This suffices because, for every path visiting a clause from a literal, we may reroute directly to the next clause instead of backtracking to the literal and then traveling back out to the next clause. So, the avatar will never be required to traverse the clause-literal path in both directions. Of course, implementing a bidirectional Crossover would also work, but one direction suffices.

4 Super Mario Bros.

Created by Shigeru Miyamoto, developed by Nintendo, and published for the Nintendo Entertainment System (NES) in 1985, Super Mario Bros. was designed as a pseudo-sequel to the 1983 game Mario Bros. The game was one of eighteen games to debute alongside the launch of the NES and has since become the best-selling video game of all time, having sold more than 40.23 million copies worldwide as of 2003, until recently when it was surpassed by Wii Sports, which is included with the purchase of every Nintendo Wii home video game console. The player controls an Italian plumber named Mario, the avatar, as he travels through worlds within the Mushroom Kingdom. The game is organized as a sequence of self-contained levels, each filled with enemies and obstacles that Mario must navigate through in order to rescue the kidnapped Princess Toadstool (later known as Princess Peach) from Bowser (also known as King Koopa), the antagonist. The franchise has spawned dozens of sequels and spin-offs on all of Nintendo's entertainment systems. Mario's character became so popular that he became Nintendo's mascot and appeared in over 200 games, including racing, puzzle, party, role-playing, fighting, and sports games.

4.1 Gameplay Mechanics

Mario is limited to walking, running, crouching, and jumping. Through a combination of these actions, Mario may perform more complex movement. For instance, if Mario crouches while running, he will crouch-slide a short distance, allowing him to slide into or fit through small corridors. Mario defeats most enemies by jumping and landing on the top of the enemy avatar. Most of the time, and by default, Mario is non-harmful, occupies one unit space in the game world, and dies instantly from harmful contact with an enemy. Jumping on an enemy is typically non-harmful, although there are exceptions not encountered in our proof. If Mario obtains a Super Mushroom, he becomes Super Mario, growing big to occupy two units vertically. Mario reverts back to his default form upon harmful contact, typically through physical contact with an enemy avatar or projectile. Some enemies and game objects will be particularly important.

Koopa Troopas play special roles as turtle-like enemies in Mario's adventure. Green Koopas

walk off cliffs, whereas Red Koopas turn around when they reach a cliff edge. Unique to Koopas, when hit by a jump attack from Mario, they will retreat into their shell, leaving it for Mario to use. Mario can then "kick" or walk into the Koopa Shell. Upon contact, the shell would be launched at a constant velocity, defeating all enemies in its path, bouncing off surfaces, activating blocks it rebounds off, and falling off cliffs (including red shells). Further, a shell in motion can be stopped by jumping on it. After a short time, the Koopa will come back out of its shell and resume its previous mechanics. Only a fireball shot by Fire Mario can destroy a Koopa Shell, which we do not consider in our proof. In our construction, we only use Red Koopas.

In addition, blocks are a central construct in the games. Super Mario activates blocks by jumping into them from below, thereby hitting them with the top of his head. There are three types of blocks: Normal blocks are inert, item blocks release an item upon activation, brick blocks are destroyed upon activation. Countless other enemies and obstacles present themselves throughout the game, but none play a role in our proofs so we refer the reader to [6] for additional information.

4.2 NP-completeness

In this section, we prove the following:

Theorem 3.1 *It is NP-complete to decide whether the goal is reachable from the start of a stage in generalized Super Mario Bros.*

4.2.1 NP-hard

We first prove our problem is NP-hard. We use the general framework from Section 3. So, it remains only to implement the gadgets. We begin with the Start and Finish gadgets. The Start gadget includes an item block containing a Super Mushroom which allows Mario the option to enter the Super Mario state (see Figure 3). This serves two purposes. Recalling that Super Mario occupies two units, he is prevented from fitting into narrow horizantal corridors, a property essential to our other gadgets. Second, only Super Mario may destroy bricks, unlike Mario. We force the player to choose taking the Super Mushroom by blocking the Finish gadget with a brick block (see Figure 3).

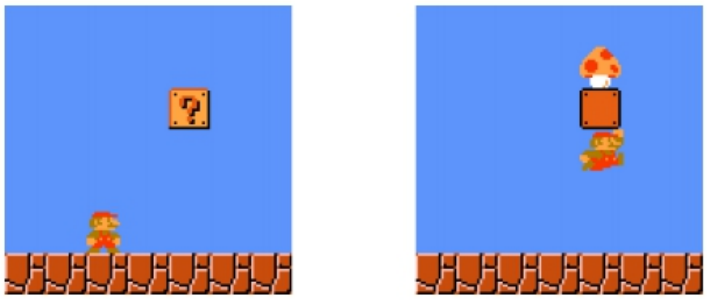

Figure 3: Left: Start gadget for Mario. Right: The item block contains a Super Mushroom.

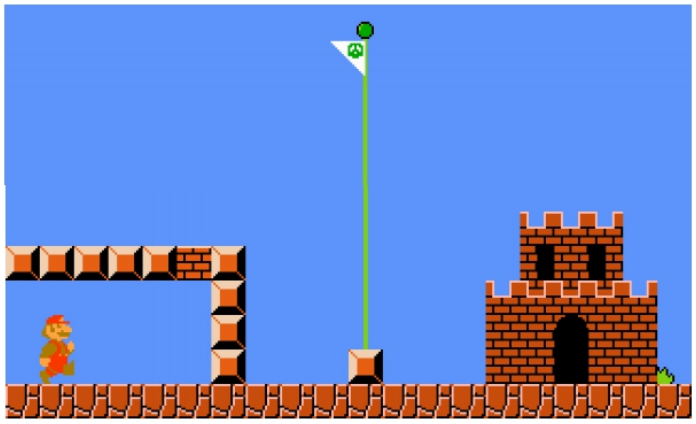

Figure 4: Finish gadget for Super Mario Bros.

The Variable gadget is pictured in Figure 5. We create two entrances, one stemming from each literal of the previoius variable. That is, for variable x_i, one entrance comes from x_{i-1} and the other from $\neg x_{i-1}$. Mario's jump is limited, so once he falls down, he is unable to jump back up onto the top ledges. So, Mario cannot re-visit any variable. In particular, Mario cannot traverse the negation of the literal he previously chose. To assign a value to the variable, Mario may fall down either the left or right passage.

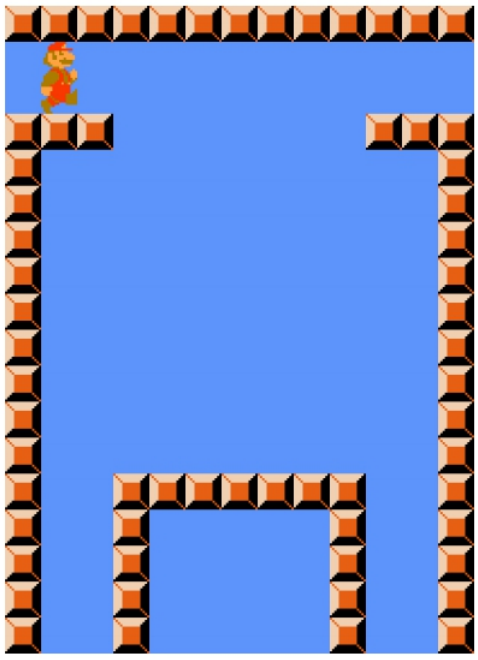

Figure 5: Variable gadget for generalized Super Mario Bros.

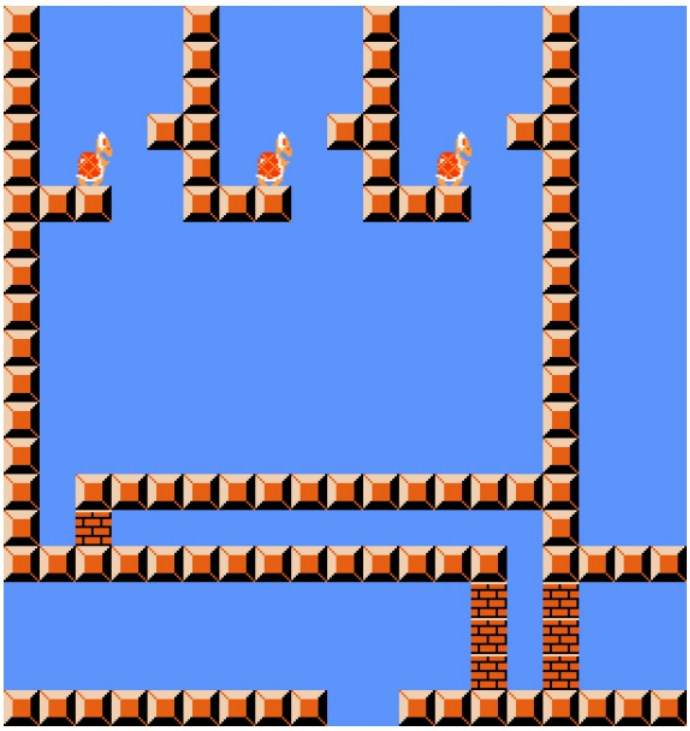

Figure 6: Clause gadget for generalized Super Mario Bros.

Now, we implement the Clause gadget, pictured in Figure 6. We know each clause contains exactly $k = 3$ literals. Hence, the three entrances at the top originate from the three literals appearing in the clause. To unlock the clause, Mario enters through on of the three entrances, jumps onto the respective right platform, and then onto the Red Koopa beneath him. He then kicks the shell, launching it off the platform, rebounding off the right wall, and continuing through until it breaks all the bricks in the bottom passage, opening the corridor for later travel. We note that falling down serves no purpose for Super Mario because he cannot fit into the one-unit space of the raised lower corridor. Harming himself and reducing to small Mario prevents him from proceeding through the Finish gadget at the end. Moreover, only one shell is need to satisfy the clause, corresponding to boolean disjunction. The gap at the bottom serves to force the Koopa Shell to fall into a bottomless pit and leave the game.

Lastly, we discuss the Crossover gadget, pictured in Figure 7. There are four entrances/exits: top left, top right, bottom left, and bottom right. We design the crossover unidirectionally, which suffices as described earlier. So, without loss of generality, suppose Mario enters from the top left. He cannot jump across to the top right due to the vertical line of normal blocks, nor can he fall down and exit through the bottom left because the passage is blocked. He may only jump on the Red Koopa and kick its shell down rightward, thereby breaking the lone brick block on the bottom right. Mario may then fall down and crouch-slide through the now-open doorway. A symmetrical argument holds for the top right to bottom left passage.

By invoking the framework proved in Section 3, we have proved Super Mario Bros. NP-hard.

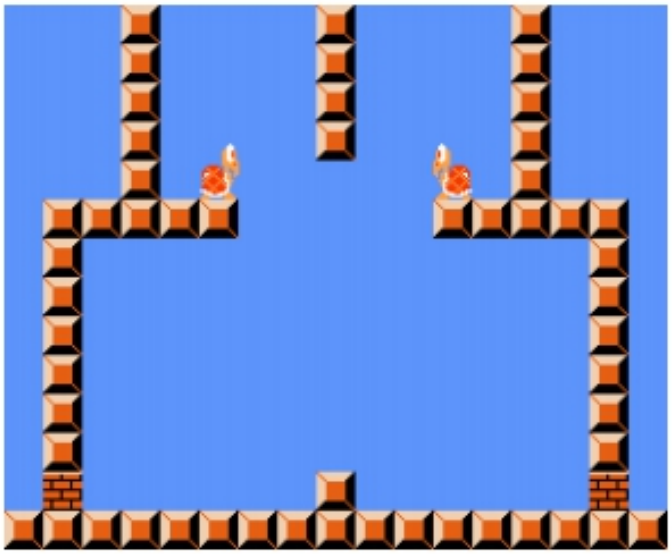

Figure 7: Crossover gadget for generalized Super Mario Bros.

4.2.2 NP

We now show that reachability in Super Mario Bros. is in NP. It suffices to show that, if a solution exists to the problem, then there is a path-finding solution algorithm with a running time that is at most polynomial in the input size.

So, suppose there exists a solution. Consider any path Mario takes through the stage. Since enemies do not respawn after dying, the path need only visit every enemy at most once. This will provide sufficient opportunity for Mario to defeat or avoid every enemy and advance through the level. The only other permanent objects in the game are Koopa Shells, but these only ever slide and fall. So, every shell must either fall down into a bottomless pit and leave the game or eventually enter a rebound cycle where the bounce back and forth between two walls infinitely. In both cases, Mario is not obstructed any further in his path, so the shells can be effectively ignored. Mario gains nothing from jumping on the shell to stop it. Hence, there exists a solution that is polynomial in the size of input.

4.3 Other Mario games

The NP-hardness from Theorem 4.1 holds for the NES sequel, Super Mario Bros.: The Lost Levels (Super Mario Bros. 2 in Japan), because the gameplay mechanics are exactly the same.

4.3.1 Modified Crossover Gadget

In later games, Mario can pick up Koopa shells, which we recall played a role in both the Clause and Crossover gadgets. The Clause gadget is unaffected, but the Crossover gadget requires modification since Mario can now pick up and throw the a Koopa shell to break both brick blocks in just one traversal. Instead, consider the modified Crossover gadget in Figure 8 which avoids the use of Koopas. The enemy on the left is a Goomba, Mario's most frequent foe. They are a fungus-based species that resembles a small, brown mushrooms and are physcially weak. Mario may defeat a Goomba by jumping on it. No other features of a Goomba are encountered. [6]

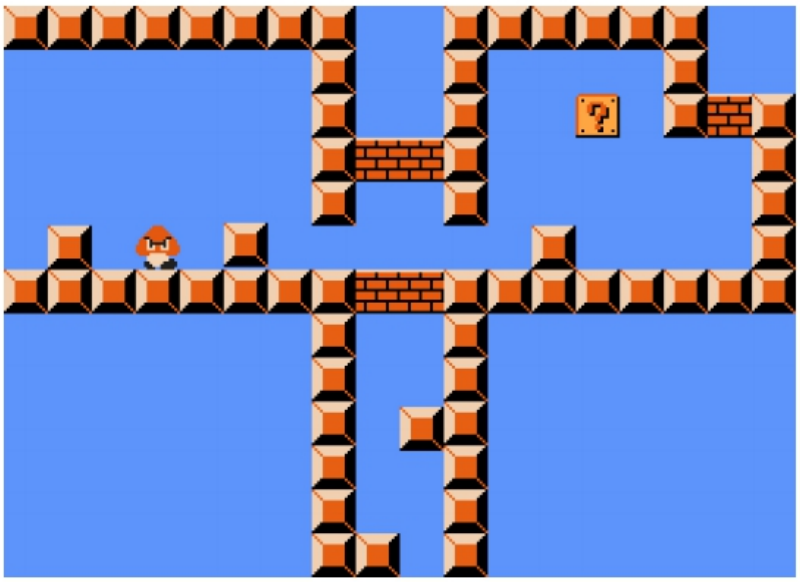

Figure 8: Modified Crossover gadget for Super Mario Bros. 3 and Super Mario World

This unidirectional Crossover allows left-to-right and bottom-to-top traversal as follows. From the left, Super Mario may choose to get hurt by the Goomba and revert back to his small state. After he is small, Mario may fit through the narrow corridor in the center, pick up a Super Mushroom from the item block on the other side, and break through the brick to exit on the right. Clearly, there will be no leakage on the bottom nor leakage to the top since small Mario cannot break bricks. Likewise, there is not enough space for Super Mario to run and crouch-slide through the narrow corridor's entrance. So, the gadget allows left-to-right traversal without leakage.

From the bottom, Super Mario may break the right brick on the lower level, jump onto the left brick of the lower level, break the left brick on the upper level, jump onto the right brick of the upper level, and exit upwards. Since Mario is big throughout the traversal, he cannot fit through the narrow corridor's entrance or exit, thereby preventing horizantal leakage. So, the gadget allows bottom-to-top traversal without leakage. Lastly, we note that the gadget does not originally allow entrance from the top or the right. This completes the modified Crossover construction.

4.3.2 Classifications

Using this new Crossover gadget, our full construction extends to Super Mario Bros. 3 and Super Mario World, so our motion-planning problem is NP-hard in all Mario games considered. NP-completeness holds for Super Mario Bros.: The Lost Levels because the mechanics are identical. However, it is unclear whether our problem remains in NP under the new game mechanics. Koopa shells are permanent, so Mario's newfound ability to pick up Koopa shells negates our previous assumption that each enemy need only be visited once. So, importantly, our existential claim regarding a polynomial-time, path-finding solution algorothm does not follow. In fact, a clever reader who has played Mario before can easily imagine a level that requires exponentially-many Koopa shell re-visits. Current methods cannot show that such a situation is either necessary or impossible. So, we leave it as an open question whether Super Mario Bros. 3 and Super Mario World are in NP and thus NP-complete.

5 Donkey Kong Country

Developed by Rare, Donkey Kong Country is a platform video game published for the Super Nintendo Entertainment System (SNES) in 1994. Similar to Mario, the Donkey Kong character was created by Shigeru Miyamoto and appears in many other games. Almost humorously, the entire premise of Donkey Kong Country is isomorphic to that of Super Mario Bros. This time, King K. Rool has stolen Donkey Kong's banana hoard. The player controls Donkey Kong (and once in a while, his small nephew Diddy Kong) as he navigates through levels filled with enemies and obstacles in order to eventually defeat K. Rool.

5.1 Gameplay Mechanics

Donkey Kong Country's gameplay is very simple. We review only the mechanics necessary for our framework construction, directing the curious towards [5]. Donkey Kong can walk, run, roll, crouch, and jump as well as pick up Barrels and throw them, after which they will roll along the ground (similar to Koopa Shells) until they hit a wall or an enemy, thereby killing it. Further, Donkey Kong can jump into Barrel Cannons. This allows him to defy gravity by firing in a straight line. Manual Barrel Cannons rotate through a preset set of angles at which Donkey Kong may be fired. A skilled player can pick a desired angle with precise timing. Automatic Barrel Cannons fire Donkey Kong in a static, preset direction as soon as he jumps inside. To differentiate between the two, Manual Barrel Cannons appear with a white, 8-sided-star decal, whereas Automatic Barrel Cannons appear with a single white line on them. Zingers are giant wasp enemies that can only by killed by Barrels and kill Donkey Kong on contact.

5.2 NP-completeness

In this section, we prove the following

Theorem 5.1 *It is NP-complete to decide whether the goal is reachable from the start of a stage in generalized Donkey Kong Country.*

5.2.1 NP-hard

As before, we first prove generalized Donkey Kong Country is NP-hard. Again, we use the general framework from Section 3 to reduce to 3-SAT.

This time, the Start and Finish gadgets are trivial; they are simply the beginning and end of the level with no trickery required. Likewise, the Variable gadget is completely analgous to the Variable gadget for Super Mario Bros, except this time replacing blocks with the appropriate terrain from Donkey Kong Country.

Following the common theme, the Clause gadget shown in Figure 8 is again similar to Mario. When visiting the Clause gadget from a Variable, Donkey Kong must pick up the Barrel and throw it down onto the platform below, after which it rolls off the right edge to hit and kill the Zinger. This unlocks the Clause. Likewise, the Barrel Cannons above each platform exist to return upward and each platform is sufficiently high so that Donkey Kong cannot fall down and then jump back up onto one of the platforms. This disallows traversal to any of the other Variables, as desired.

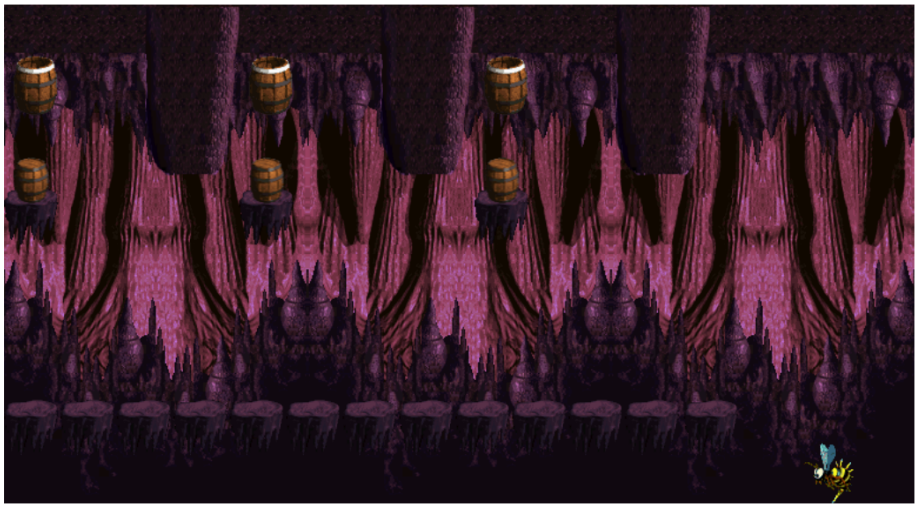

Figure 9: Clause Gadget for Donkey Kong Country

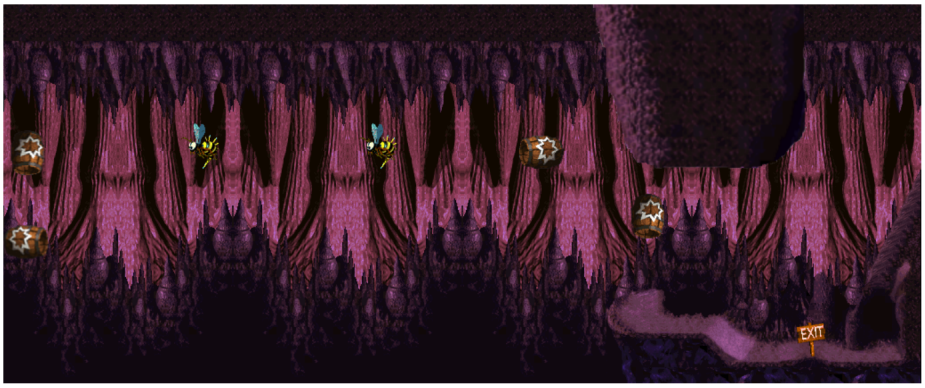

Figure 10: Check Gadget for Donkey Kong Country

Next, we implement the Check gadget. The concept is illustrated in Figure 10, but it is not drawn to scale. Each Zinger represents a Zinger from a Clause, although the corresponding Clause-gadget-length platform is not shown between Zingers due to size constraints. The idea is that after all Variable and Clause Gadgets have been traversed, Donkey Kong will fall off a ledge (not shown) into the Automatic Barrel Cannon pictured in the top right. He then blasts towards the Automatic Barrel Cannon on the top left, then bottom left, and then bottom right, finally landing on the platform and proceeding towards the goal at the Exit. There are Zingers positioned in the first blast path, one per clause (the same Zingers that are in the Clause gadgets) so that Donkey Kong may reach the goal if and only if every Zinger is killed; this corresponds to each clause being satisfied.

Finally, the Crossover gadget in Figure 11 is simple given the game mechanics. At every entrance and exit, there is both a forward-blasting and backwards-blasting Automatic Barrel Cannon. Clearly, there is no vertical or horizantal leakage.

By invoking the framework from Section 3, we have proved generalized Donkey Kong Country is NP-hard. Now we complete the NP-complete classification by proving it is NP.

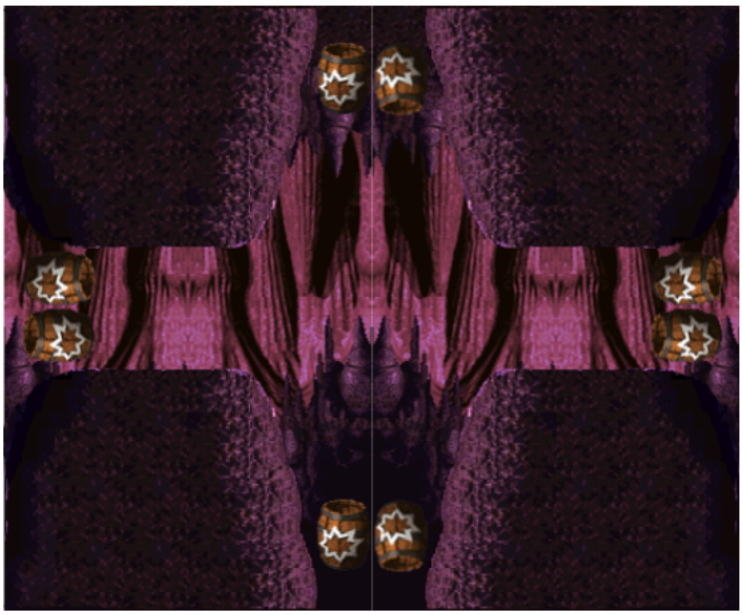

Figure 11: Crossover Gadget for Donkey Kong Country

5.2.2 NP

Again, like Super Mario Bros., Donkey Kong Country is in NP because any solution path need only visit each enemy at most once. Since levels are bounded, all other objects that affect the environment (Barrels) make monotonic progress until the are destroyed and leave the game upon colliding with a wall or falling off a ledge. So, the length of a solution path is polynomially bounded in the input size, and thus Donkey Kong Country is in NP.

5.3 Other Donkey Kong Games

Unlike Mario, Theorem 5.1 holds for the sequels Donkey Kong Country 2 and 3 since Zingers, Barrels, and both Barrel Cannons are all present in these games.

6 Metroid

Metroid is an action-adventure video game co-developed by Nintendo Research and Development 1 division and Intelligent Systems. It was released in Japan in 1986, North America in 1987, and Europe in 1988 for the NES. The action follows the female bounty hunter Samas Aran as she travels across the planet Zebes in search of Metroid—creatures stolen by Space Pirates who, of course, have world domination in mind. Unlike our previous side-scrolling platform games, Metroid is nonlinear and the player must acquire new items and power-ups in order to access previously inaccessible areas. The game world, Zebes, is essentially one very large overworld, which is much less structured at first glance. As opposed to a series of start-to-finish levels, the game focuses on exploration and the search for abilities.

6.1 Game Mechanics

Gameplay is simple. Samus can walk, jump, or shoot missles. She may shoot left, right, or upwards but not downwards. When standing on the ground, Samus can enter Morph Ball mode which imposes a state change on Samus by shrinking her from two tiles to one and allowing her to roll. This removes her ability to attack or jump, however, and she may only roll left or right and fall.

We only encounter the Zoomer enemy in our proof. Zoomers are native creatues of Zebes that conveniently walk along whatever surface they adhere to and deal damage to Samus upon contact. Samus kills Zoomers using her normal weapons so no power-ups or items are required. Similar to Mario, the game contains breakable blocks that can be destroyed by Samus' normal weapon. Contrarily, the blocks regenerate after a short while. Additional information can be found in [7].

6.2 NP-complete

We prove the familiar theorem:

Theorem 6.1 *It is NP-complete to decide whether a given target location is reachable from a given start location in generalized Metroid.*

6.2.1 NP-hard

As before, we use the same framework from Section 3. In our construction, Samus has only acquired the Morph Ball and has only 1 Energy point left, so she must not get hurt, else she will die. Similar to Donkey Kong, the Start and Finish gadgets are trivial, and the Variable gadget is identical to both our previous constructions, replacing the relevant terrain.

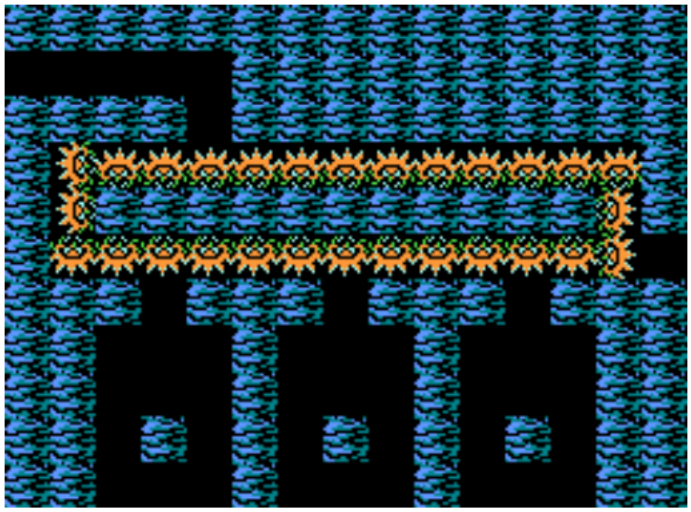

Figure 12: Clause Gadget for generalized Metroid

So, we begin with the Clause gadget above. This time, Variables enter from below and are checked on top. Arriving from below, Samus may shoot upward and kill all the Zoomers as they move clockwise about the platform. This frees up space for later traversal using Morph Ball mode and only Morph Ball mode because Samus cannot fit through otherwise. Since Samus cannot jump in Morph Ball mode, she cannot traverse any Variable from any other Variable, preventing

leakage.

Lastly, we implement the Crossover gadget below. The horizantal passage clearly prevents vertical leakage. On the other hand, the vertical passage is unidirectional from bottom to top, which we recall suffices from Section 3. The two blue blocks in the center are breakable, and Samus cannot jump and fit into the horizontal passage for the same reasons she can't jump into the small corridor in the Clause gadget. She won't fit without Morph Ball mode but can't jump if she enters it so it remains impassable.

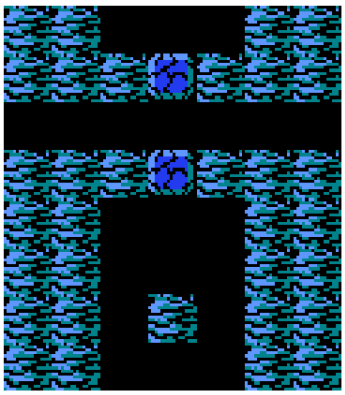

Figure 13: Crossover Gadget for generalized Metroid

Again, we've constructed all nontrival gadgets so we conclude Metroid is NP-hard.

6.2.2 NP

Metroid is in NP because any solution path only needs to visit each enemey at most once and therefore is polynomially bounded in length.

6.3 Other Metroid Games

Thereom 6.1 holds for all other Metroid games because the Morph Ball, Zoomers, and breakable rocks appear in all sequels. Even for three-dimensional games, our construction can be modified to have very small depth, making the third dimension irrelevant.

7 Summary

We conclude by summarizing our results. In short, we've shown that motion planning in generalized Super Mario Bros. is NP-hard, and the classification extends to NP-completeness for Super Mario Bros. and Super Mario Bros.: The Lost Levels. Likewise, we've shown that motion planning in both generalized Donkey Kong Country and generalized Metroid is NP-complete. We have also developed a general framework which can be used to prove NP-hardness for any game given that the gadgets can be implemented in the respective game world. We encourage the reader to consider games they have enjoyed over the years and think about how a reduction might be proved. Lastly, we have demonstrated a standard proof that motion planning in a video game is in NP whenever a solution path need only visit each enemy or obstacle at most once.

8 Acknowledgments

This work has largely been a review of results proved in [1] and [2]. Finally, of course, we thank Nintendo and the associated developers for creating these timeless classics.

References

[1] G. Aloupis, E. D. Demaine, and A. Guo. Classic Nintendo Games are (NP-)Hard. Computing Research Repository, 2012. http://arxiv.org/abs/1203.1895.

[2] E. D. Demaine, M. L. Demaine, and J O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *Proceedings of the 12th Canadian Conference on Computational Geometry*, pp. 211-219, 2000.

[3] J. Barwise and J. Etchemendy, *Language, Proof and Logic*. CSLI Publications, 2nd Edition, 2011.

[4] Michael Sipser, *Introduction to the Theory of Computation*. Course Technology, 2nd Edition, 2005.

[5] http://donkeykong.wikia.com/wiki/Donkey_Kong_Country

[6] http://www.mariowiki.com/Super_Mario_Bros.

[7] http://www.metroidwiki.org/wiki/Metroid_(game)