Classic Nintendo Games are (Computationally) Hard

Greg Aloupis*  Erik D. Demaine†  Alan Guo‡  Giovanni Viglietta§

February 8, 2015

Abstract

We prove NP-hardness results for five of Nintendo’s largest video game franchises: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. Our results apply to generalized versions of Super Mario Bros. 1–3, The Lost Levels, and Super Mario World; Donkey Kong Country 1–3; all Legend of Zelda games; all Metroid games; and all Pokémon role-playing games. In addition, we prove PSPACE-completeness of the Donkey Kong Country games and several Legend of Zelda games.

1 Introduction

A series of recent papers have analyzed the computational complexity of playing many different video games [1, 4, 5, 6], but the most well-known classic Nintendo games have yet to be included among these results. In this paper, we analyze some of the best-known Nintendo games of all time: Mario, Donkey Kong, Legend of Zelda, Metroid, and Pokémon. We prove that it is NP-hard, and in some cases PSPACE-hard, to play generalized versions of most games in these series. In particular, our NP-hardness results apply to the NES games Super Mario Bros. 1–3, Super Mario Bros.: The Lost Levels, and Super Mario World (developed by Nintendo); to the SNES games Donkey Kong Country 1–3 (developed by Rare Ltd.); to all Legend of Zelda games (developed by Nintendo);1 to all Metroid games (developed by Nintendo); and to all Pokémon role-playing games (developed by Game Freak and Creatures Inc.).2 Our PSPACE-hardness results apply to to the SNES games Donkey Kong Country 1–3, and to The Legend of Zelda: A Link to the Past. Some of the aforementioned games are also complete for either NP or PSPACE. All of these results are new.3

*Chargé de Recherches du FNRS, Département d’Informatique, Université Libre de Bruxelles, aloupis.greg@gmail.com. Work initiated while at Institute of Information Science, Academia Sinica.
†MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar St., Cambridge, MA 02139, USA, {edemaine,aguo}@mit.edu
‡Partially supported by NSF grants CCF-0829672, CCF-1065125, and CCF-6922462.
§School of Electrical Engineering and Computer Science, University of Ottawa, Canada, viglietta@gmail.com.
1We exclude the Zelda CD-i games by Philips Media, which Nintendo does not list as part of the Legend of Zelda series.
2All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Sprites are used here under Fair Use for the educational purpose of illustrating mathematical theorems.
3A humorous paper (http://www.cs.cmu.edu/~tom7/sigbovik/mariok.pdf) and video (http://www.youtube.com/watch?v=IlhGIGqAK9c) by Vargomax V. Vargomax claims that “generalized Super Mario Bros. is NP-complete”, but both versions have no actual proof, only nonsensical content.
For these games, we consider the decision problem of reachability: given a stage or dungeon, is it possible to reach the goal point \( t \) from the start point \( s \)? Our results apply to generalizations of the games where we only generalize the map size and leave all other mechanics of the games as they are in their original settings. Most of our NP-hardness proofs are by reduction from 3-SAT, and rely on a common construction. Similarly, our PSPACE-completeness results for Legend of Zelda: A Link to the Past and Donkey Kong Country games are by a reduction from True Quantified Boolean Formula (TQBF), and rely on a common construction (inspired by a metatheorem from [5]). In addition, we show that several Zelda games are PSPACE-complete by reducing from PushPush-1 [3].

We can obtain some positive results if we bound the “memory” of the game. For example, recall that in Super Mario Bros. everything substantially off screen resets to its initial state. Thus, if we generalize the stage size in Super Mario Bros. but keep the screen size constant, then reachability of the goal can be decided in polynomial time: the state space is polynomial in size, so we can simply traverse the entire state space and check whether the goal is reachable. Similar results hold for the other games if we bound the screen size in Donkey Kong Country or the room size in Legend of Zelda, Metroid, and Pokémon. The screen-size bound is more realistic (though fairly large in practice), while there is no standard size for rooms in Metroid and Pokémon.

**Membership in PSPACE.** Most of the games considered are easy to show belong to PSPACE, because every game element’s behavior is a simple (deterministic) function of the player’s moves. Therefore, we can solve a level by making moves nondeterministically while maintaining the current game state (which is polynomial), and use that \( \text{NPSPACE} = \text{PSPACE} \).

Some other games, such as Legend of Zelda and its sequels, also include enemies and other game elements that behave pseudorandomly. As long as the random seed can be encoded in a polynomial number of bits, which is the case in all reasonable implementations, the problem remains in PSPACE.

In general, we may safely assume that any generalization of a commercial single-player game is in PSPACE. Indeed, these games are inherently designed to run smoothly in real time, so they must rely on a game engine that is at least in PSPACE, if not in P. Again, once a PSPACE game engine is given, the game can be played nondeterministically, maintaining a polynomial-size game state and using that \( \text{NPSPACE} = \text{PSPACE} \).

**Game model and glitches.** We adopt an idealized model of the games in which we assume that the rules of the games are as (we imagine) the game developers intended rather than as they are implemented. In particular, we assume the absence of major game-breaking glitches (for an example of a major game-breaking glitch, see [16], in which the speed runner “beats” Super Mario World in less than 3 minutes by performing a sequence of seemingly arbitrary and nonsensical actions, which fools the game into thinking the game is won). We view these glitches not as inherently part of the game but rather as artifacts of imperfect implementation.

**Organization.** In Section 2, we present two general schematics used in almost all of our NP-hardness and PSPACE-hardness reductions. We show that, if the basic gadgets in one of the two constructions are implemented correctly, then the reduction from either 3-SAT or TQBF follows. In Section 3, we prove that generalized Super Mario Bros. is NP-hard by constructing the appropriate
gadgets for the construction given in Section 2. In Sections 4, 5, 6, and 7, we do the same for general-ized Donkey Kong Country, Legend of Zelda, Metroid, and Pokémon, respectively. In addition, Sections 4 and 5 show that the Donkey Kong Country games and some Legend of Zelda games are PSPACE-complete, again by constructing the appropriate gadgets introduced in Section 2.

2 Frameworks for Platform Games

The general decision problem we consider for platform games is to determine whether it is possible to get from a given start point to a given goal point. This is a natural problem because the main challenge of platform games is to maneuver around enemies and obstacles in order to reach the end of each stage.

2.1 Framework for NP-hardness

We use a general framework for proving the NP-hardness of platform games, illustrated in Figure 1. This framework is similar to the NP-hardness proof of PushPush-1 [2]. With this framework in hand, we can prove hardness of individual games by just constructing the necessary gadgets.

![General framework for NP-hardness](Figure 1: General framework for NP-hardness)

The framework reduces from the classic NP-complete problem 3-SAT: decide whether a 3-CNF Boolean formula can be made “true” by setting the variables appropriately. The player’s character starts at the position labeled Start, then proceeds to the Variable gadgets. Each Variable gadget forces the player to make an exclusive choice of “true” \(x\) or “false” \(\neg x\) value for a variable in the formula. Either choice enables the player to follow paths leading to Clause gadgets, corresponding to the clauses containing that literal \((x\ or \ \neg x)\). These paths may cross each other, but Crossover gadgets prevent the player from switching between crossing paths. By visiting a Clause gadget,
the player can "unlock" the clause (a permanent state change), but cannot reach any of the other paths connecting to the Clause gadget. Finally, after traversing through all the Variable gadgets, the player must traverse a long "check" path, which passes through each Clause gadget, to reach the Finish position. The player can get through the check path if and only if each clause has been unlocked by some literal. Therefore, it suffices to implement Start, Variable, Clause, Finish, and Crossover gadgets to prove NP-hardness of each platform game.

The specific properties our gadgets must satisfy are the following.

**Start and Finish.** The Start and Finish gadgets contain the starting point and goal for the character, respectively. In most of our reductions, these gadgets are trivial, but in some cases we need the player to be in a certain state throughout the construction, which we can force by making the Finish accessible only if the player is in the desired state, and the desired state may be entered at the Start. For example, in the case of Super Mario Bros., we need Mario to be big throughout the stage, so we put a Super Mushroom at the start and a brick at the Finish, which can be broken through only if Mario is big.

**Variable.** Each Variable gadget must force the player to choose one of two paths, corresponding to the variable $x_i$ or its negation $\neg x_i$, being chosen as the satisfied literal, such that once a path is taken, the other path can never be traversed. Each Variable gadget must be accessible from and only from the previous Variable gadget, independent of the choice made in the previous gadget, in such a way that entering from one literal does not allow traversal back into the negation of the literal.

**Clause and Check.** Each Clause gadget must be accessible from (and initially, only from) the literal paths corresponding to the literals appearing in the clause in the original Boolean formula. In addition, when the player visits a Clause gadget in this way, they may perform some action that "unlocks" the gadget. The Check path traverses every Clause gadget in sequence, and the player may pass through each Clause gadget via the Check path if and only if the Clause gadget is unlocked. Thus the Check path can be fully traversed only if all the Variable gadgets have been visited from literal paths. If the player traverses the entire Check path, they may access the Finish gadget.

**Crossover.** The Crossover gadget must allow traversal via two passages that cross each other, in such a way that there is no leakage between the two.

**Remark 2.1.** The Crossover gadget only needs to be unidirectional, in the sense that each of the two crossing paths needs to be traversed in only one direction. This is sufficient because, for each path visiting a clause from a literal, instead of backtracking to the literal after visiting the clause, we can reroute directly to visit the next clause, so the player is never required to traverse a literal path in both directions.

**Remark 2.2.** It is safe to further assume in a Crossover gadget that each of the two crossing paths is traversed at most once, and that one path is never traversed before the other path (i.e., if both paths are traversed, the order of traversal is fixed). This is sufficient because two literal paths either are the two sides of the same Variable (and hence only one gets traversed), or they come from different Variables, in which case the one from the earlier Variable in the sequence is
Pokémon

Pokémon is a series of overhead role-playing games developed by Game Freak for various handheld Nintendo consoles, including Game Boy, Game Boy Color, Game Boy Advance, and the Nintendo DS. There are various versions of Pokémon, but the core mechanics of the game are invariant throughout. The player controls a young teenager and wanders through the land capturing and training creatures called Pokémon (short for Pocket Monsters). At any one time, the player may hold up to six Pokémon in their team. Each Pokémon has a Level, which indicates roughly how experienced it is, as well as battle stats: attack, defense, speed, special attack, and special defense. In addition, each Pokémon has hit points (HP), which indicate how much damage it can take before “fainting”, as well as power points (PP) for each move, which indicate how many times it may use that move. Pokémon battles are two-player simultaneous move games, and in each round the Pokémon with higher speed attacks first. The battle ends when all of one trainer’s Pokémon have fainted.

Pokémon is NP-hard because it has blocks that the player can push according to the rules of Push-1 [2]. Therefore we immediately have

Theorem 7.1. It is NP-hard to decide whether a given target location is reachable from a given start location in generalized Pokémon.

In this section, we give an alternate proof that Pokémon is NP-hard using no elements of the game except enemy Trainers and the game’s battle mechanics (and thus no blocks). Therefore we obtain the following stronger result:

Theorem 7.2. It is NP-complete to decide whether a given target location is reachable from a given start location in generalized Pokémon in which the only overworld game elements are enemy Trainers.

Proof. We briefly describe the mechanism for enemy Trainer encounters in the Pokémon games. Each enemy Trainer has an orientation (which direction they are facing), a range of sight, and a set of Pokémon. If the player walks into the line of sight of the Trainer (and if such a line of sight is not occluded by some other element, e.g., another Trainer), the player is forced to stop, the Trainer walks toward the player until they are adjacent, and then battle ensues. Additionally, if the player approaches a Trainer from outside the Trainer’s line of sight (i.e., from behind or the sides), the player may talk to the Trainer, activating the battle. If the player wins the battle, the Trainer will not move or attack again.

We prove NP-hardness by using the framework in Section 2. Start and Finish gadgets are trivial, hence it suffices to implement the Variable, Clause, and Crossover gadgets.

In our implementations, we use three kinds of objects. Walls, represented by dark grey blocks, cannot be occupied or walked through. Trainers’ lines of sight are indicated by translucent rectangles. We have two types of Trainers. Weak Trainers, represented by red rectangles, are Trainers whom the player can defeat with certainty without expending any effort, i.e., without consuming
PP or taking damage. Strong Trainers, represented by blue rectangles, are Trainers against whom the player will always lose.

We can implement weak and strong Trainers as follows (a construction due to Istvan Chung). Weak Trainers each hold a Level 100 Electrode with maximum Speed and equipped with only the Self Destruct move. Strong Trainers each hold two Snorlaxes, with Speed of 30. The player has no items, and only one Pokémon in his team. For Generation I and II games (Red/Blue/Yellow and Gold/Silver/Crystal versions respectively), the player holds a Gastly which has learned Self Destruct using TM36, and its PP for its other moves have all been expended, so it can only use Self Destruct in battle. When the player encounters a weak Trainer, the enemy Electrode will move first and use Self Destruct, which deals no damage to Gastly because Self Destruct is a Normal type attack and Gastly is Ghost type, so the weak Trainer immediately loses. When the player encounters a strong Trainer, Gastly moves first and uses Self Destruct, causing the player to lose (even if it defeats the enemy Snorlax, the opponent holds another one). This implementation only works in Generations I and II because TM36 exists only in Generation I and the Time Capsule feature in Generation II allows a Gastly with Self Destruct to be traded from Generation I to Generation II. In Generations III, IV, and V, Gastly can be replaced by Duskull, which is allowed to learn the move Memento, which serves the same purpose as Self Destruct, via breeding.

We now implement each gadget. The Variable gadget, illustrated in Figure 33, must force the player to choose either an a-to-b traversal or an a-to-c traversal. We show that this is the case. The player enters through a. If the player wants to exit through b, they walk into the Trainer’s line of sight, luring the Trainer down and opening up the passage to b, but closing the passage to c. On the other hand, if the player wants to exit through c, they walk up to the Trainer and talk to him, disabling the Trainer, so that they may walk around and exit through c.

![Figure 33: Variable gadget for Pokémon](image)

The Clause gadget, depicted in Figure 34, is unlocked whenever the player enters from one of the top paths and talks to one of the three weak Trainers, disabling him. When the Check path is traversed, every weak guard who has not been disabled is lured down by the player, thus getting out of the line of sight of the rightmost strong Trainer. Eventually the player must cross that line of sight, and is reached by the strong Trainer if and only if no weak Trainer blocks him, which happens if and only if the Clause gadget is still locked. The leftmost strong Trainer prevents leakage from the Check path to the literals, in case the leftmost weak Trainer has been lured down.

Before presenting the Crossover gadget, we introduce the Single-use path, a path that can only be traversed once, and only in one direction. This is implemented by the gadget in Figure 35, which can be traversed only once, from a to b. Clearly, the player cannot enter via b, because that lures the weak Trainer to block the passage. Suppose the player enters through a. They can safely walk to b, because the weak Trainer is blocking the bottom strong Trainer’s line of sight. However, to reach b, the player must lure the weak Trainer out of the line of sight of the strong Trainer, hence
the player may never return in the reverse direction. Also, there is a strong Trainer above the weak Trainer, preventing the player from disabling the weak Trainer by entering from $a$.

Finally, the Crossover gadget is shown in Figure 36. Each of its two passages, $x$-to-$x'$ and $y$-to-$y'$, is unidirectional and may be traversed only once, which suffices due to Remarks 2.1 and 2.2.

Because the Crossover gadget is symmetric, we assume, without loss of generality, that the player attempts to traverse the $x$-to-$x'$ passage first. Upon entering from $x$, they go down and disable the bottom weak Trainer, then go back up and proceed to the right. Observe that the top-left part of the gadget is a “crossroads” made of four isometric copies of the Single-use path of Figure 35. Upon reaching it, the player is forced to go either down or right. If they go down, they eventually get stuck, because the crossroads is now unreachable due to the just traversed Singe-use path, and the $y'$ exit is unreachable too, because the top weak Trainer is lured all the way down to block the player, as soon as they attempt to exit. On the other hand, if the player proceeds to the right upon entering the crossroads, they can safely reach the $x'$ exit, because the leftmost weak Trainer has been previously disabled. Note that, by doing so, the player incidentally crosses the line of sight of the top weak Trainer, luring him down and disabling him.

Now, if the player wishes to traverse the $y$-to-$y'$ passage, they must take the vertical Single-use paths of the crossroads, because the other two have already been traversed. At this point, the player has no choice but to exit from $y'$, which is safely reachable because the top weak Trainer has already been disabled during the first traversal of the gadget, and is not blocking the way out.

**NP.** To show that Pokémon with only enemy Trainers is in NP, note that once a Trainer has been battled, they become inert. Moreover, each actual battle is bounded in length by a constant, because eventually all Pokémon must expend all their PP for their moves and use Struggle, a standard physical attack which hurts the opponent as well as the user. Moreover, each Pokémon only has four different attacks, so the branching factor of the game tree for each battle is constant (and hence the size of the game tree is also constant). Therefore, the player may nondeterministically
guess a solution path through the overworld, and for each battle compute the winning strategy in constant time.

Other Pokémon games. Theorem 7.2 actually holds for all Pokémon role-playing games, because the only actual Pokémon used in the construction were from the first generation, which is present in all the games, and enemy Trainers are of course present in all the games as well.