EasyChair Preprint

№ 12462

Randomizers in Video Games: Algorithms and Complexity Analysis

Alexandre-Quentin Berger

March 12, 2024

Randomizers in Video Games: Algorithms and Complexity Analysis

Alexandre-Quentin Berger

February 27, 2024

Video games have become the biggest entertainment industry in the world, gathering important communities of players and modders (people that customize games). Some of these games either include randomizers natively or are modded to include randomizers. The goal of these randomizers is to change some parts of the game randomly, while keeping it playable and ensuring the game can be finished. In this paper, we formalize the most frequent forms of randomizers, analyze the complexity of generating a random configuration that still allows the game to be finished, and propose algorithms to create such configurations efficiently.

1 Introduction

In the last decades, video games have risen from a niche product to the number one cultural industry in the world [19, 25]. Most games can be basically described as being in one or multiple categories: racing, FPS and TPS (First Person Shooter and Third Person Shooter respectively), fighting, action, adventure, RPG (Role Playing Game), plat-formers, sports, etc.

Some of these games are very different from one playthrough to another. Some others, like most action, adventure, and role-playing games tend to tell a story, which makes replaying them repetitive and thus less interesting. Randomizers allow us to change that by introducing more uncertainty in the game, most of the time by shuffling the place of the important elements needed to progress in the game (items, abilities, allied characters, etc) between their respective locations (other types of randomizers also shuffle enemies, or even parts of the world itself).

1.1 Complexity and algorithms in video games

The complexity of video games is a well explored domain: there have been some results showing some games complexity depending on key properties of these games, mostly applied to 2D games [1, 9, 24] and a few 3D games [5], that allow to classify many games, and

other results about specific games (like Lemmings [4], generalized Super Mario Bros [6], puzzles in Final Fantasy XIII-2 [12]). In most cases, these games are shown to be NP-Hard or PSPACE-Hard.

As for algorithms for video games, there has been significant contributions on algorithms to win games. In particular, the specific category of Real-Time Strategy games is well developed [7, 13, 18, 22, 26]. We can also mention search games [2], which have basically the same goal of finding an end point in a graph, but generally work with simpler structures and without the resource concept which will be essential to our model.

In this landscape, more specific results about modding, speedrunning, or randomizing games are almost inexistent: these concepts are barely mentioned in a few works [3, 17], but there is nothing to formalize or analyze these fundamental aspects in the gaming community, and this is where our work starts.

1.2 Contributions

In this paper, we start with a description of randomizers, the conditions under which we can apply one to a game, and how to formalize the problem of whether a randomized game can be finished or not. Then, we show that these problems are NP-Complete. We also give algorithms that randomize games while ensuring the game can be completed, and show some statistics for these algorithms applied to the "Metroid Prime" video game.

Note that in this work, we do not analyze the complexity of a specific game: we are considering the inherent graph problem, and we show that this theoretical problem itself is an NP-Complete problem, without any consideration to how the game itself actually works or what its complexity is. The same is true for the algorithms we present: these algorithms are applied to the graph problem and can be applied to any game being randomized.

2 Randomizers in Video Games

Let us start by introducing the basics of randomizers in video games, their rationale and inner-workings, and finally their effect in the game.

2.1 Randomizers effects

Many single player games tell a story: the game makes you play through the story as a character or group of characters, and makes you progress following a path, towards the endgame. When you replay this type of game, you already know where the game wants you to go, what is possible or not at each moment of the game, in which order you will recruit your companions, find your new powers, or the items needed to complete the game. In other words, there is no room for the unexpected, which takes out part of the amusement for a lot of players.

This is where randomizers were born: what if we could change the game each time we play it? This uncertainty will not affects the main characters, the fundamental story, the

endgame or the game mechanics, but will still randomize some parts of it, so we can replay the same adventure, the same role while introducing different experiences and challenges.

However, the main difficulty with randomizers is to ensure the game can still be played to the end and finished, and that randomizing these elements does not prevent to finish the game.

2.2 Different types of randomizers

There are a lot of randomizers that modify various aspects in many types of games: items, power-ups, allies, enemies, statistics of the player/allies/enemies, even the world of the game itself by changing the placement of rooms (most randomizers actually combine many of these aspects at the same time).

In our case, we focus on what we call ”item randomizers”, which randomize the position of items in a game, as it allows us to model almost every other randomizer: by considering power-ups, allied characters, all statistics, and even story progression (with flags indicating an event has been realized and unlocking the next event), as different kinds of items to be collected, we can model almost any kind of randomizer. This also includes randomizing enemies, if we model these enemies as obstacles for which we need some items or minimal stats to get through.

Formally, we distinguish four variants of item randomizers. As some games only allow the player to gain items (these are definitively acquired), while others also include items that are lost once used (dungeon keys in Zelda games for example), we consider two variants: one allowing item loss and one that does not. We also consider two different completion goals: some communities and randomizers consider that for the game to be playable, we just need the endgame to be reachable (this may allow some of the items to be unreachable, as long as the endgame itself can be reached), while others consider that all items need to be reachable. We will call these ”lossy game”, ”lossless game”, ”finishable game” and ”completable game” respectively.

3 Formalizing Randomized Game Problems

First, we need a model to express this problem formally.

Graphs are a good starting point, as nodes and edges can easily be identified as locations and paths between locations respectively. We considered other models, like Petri nets and algebraic Petri nets [14, 20, 21]. We chose graphs for multiple reasons: graphs are more widely known (thus allowing an easier use for the communities of players), and are very intuitive in this context as we can model a location in the game as a node, which allows the graph to visually, ”physically” represent the game’s world.

We considered dynamic graphs [8, 11] and temporal graphs [15], which would have been relevant for games with an evolving world (games procedurally generated for example), but in this case the game does not change after the randomization and stays static (only the player’s ability to access parts of the world does change), so a static graph makes more sense. Labeled graphs allow to easily translate requirements to go from one location to

another as labels on edges, but that is still not enough: we need some kind of resources system to model items being picked up.

Thus we decided to use graphs and add a resource system to formalize these problems, by defining our own new graph model, which could be described as a "labeled-multi-directed graph" that also includes two different types of nodes and a resource system. We call that new model a **resource graph**:

Definition 1. *A **Resource Graph** is defined as a 5-uplet $G_R = (V, E, R, F, L)$, with:*
- $V = (V_F; V_R)$ (Nodes) is a pair of disjoint sets of nodes V_F, V_R (i.e. such that $V_F \cap V_R = \emptyset$). V_F is a set of nodes with fixed resources (this could be no resources, or positive/negative resources that respectively give/take resources). V_R is a set of nodes that may have varying resources.
- R (Resources) is a multiset of resources.
- F (Fixed Resources) is a multiset of resources.
- $L \subseteq V_F \times F \times \{+, -\}$ (Fixed Resources Locations) is a set of triplets indicating the locations of fixed resources and whether it is a positive or negative resource.
- $E \subseteq V \times V \times P(R \cup F)$ (Edges) is a set of directed and labeled edges, where labels are a subset of all resources.

The idea is as follows: R and F are multisets (allowing multiple copies of an item), edges labels are subsets of all items (which may include fixed items), E allows to have multiple edges between a pair of nodes (to allows different paths with different conditions, or even one path that can be followed with different sets of items), and L includes a positive/negative sign (to allow for item loss, which happens in a lot of games with consumable items).

Finally, we split the nodes into two sets because it allows for an easier randomization: V_F, F and L are the fixed parts, and the action of randomizing item locations becomes a simple assignment of the items from R to (some of) the nodes in V_R. Such an assignment is called a **"seed"** (this is the standard terminology used by communities of players, not to be confused with the usual meaning of seed in computer science as an initialization value for pseudo-random generation):

Definition 2. *A **Seed** S over a resource graph $G_R = (V = (V_F, V_R), E, R, F, L)$ is a assignment of a subset of resources from R to a subset of resource nodes in V_R, i.e. a partial function $S : R \to V_R$.*

Note that S is not necessarily injective or surjective: there is no obligation to place all items from R (for example, you can remove some non essential items to make the game harder), and there is no obligation to fill all locations (we may have some of these locations left empty). In short, a seed is just a possible way to place items in these locations.

Finally, we call a **valid path** over a resource graph G_R with seed S a path, i.e. a list of consecutive edges, such that we satisfy the requirements for every edge along the path if we collect every resource encountered in the nodes we go through (i.e. it is a path that the player could actually execute in the game, as at any point on this path, the player is actually in possession of all necessary items to to through the next edge).

Using this model, we can define our randomizer problem:

Definition 3. ***Randomizer Seed Problem (Lossless, Completable Game)****:*
Considering a game without item loss with a given resource graph $G_R = (V, E, R, F, L)$ (i.e. such that L contains no negative resources), and given a starting node $Start$, an end node End, and a seed S, the goal is to find whether it is possible to find a valid path from $Start$ to End that collects all resources (the path passes at least once in each node of V_R).

We can similarly define the variant where we just want the game to be finishable (without the obligation to collect all items), and the variants where we can lose items (L has no restriction). These definitions are given in appendix A.

We show that these problems are NP-Complete, and thus that we cannot blindly generate seeds and then verify if the game can be finished; We need specific algorithms that only generate seeds that can be finished.

4 Complexity of Randomized Game Problems

Let us start with the complexity analysis of these randomizer problems.

4.1 Item randomizers are NP-complete

We show that all four versions of our randomizer problem are NP-Complete.

Theorem 1. *The Randomizer Seed Problem (Lossless, Completable Game) is NP-Complete.*

Proof. The idea of the proof is the following (full proof in given in appendix B): first, we build an non-deterministic Turing machine that will non-deterministically choose an order to pick up the items, and then a path collecting all the items in the chosen order. Checking if the chosen path is indeed valid is done in polynomial time ($O(N^2 \cdot M + N^3)$ where N is the number of nodes and M the number of edges).

Then for the NP-Complete part, we do a reduction from the NP-Hard problem 3-SAT: the main idea is to create two items for each variable in 3-SAT, representing the choice of value 0 and 1 for the variable, and place each one in a separate node, and create the edges as a long directed path forcing you to get either one or the other for each variable. Then a second part of the graph has a path of nodes representing each clause, and edges with the variables in the clause as requirements, to reach the end node.

□

Let us note that the same proof works for all other variants of our randomizer problem: completing the game is a particular case of finishing the game, and both lossless cases are just a particular case of both lossy cases. This reduction works as a proof for all four variants:

Corollary 2. *The Randomizer Seed Problems (Lossless, Finishable Game), (Lossy, Finishable Game) and (Lossy, Completable Game) are NP-Complete as well.*

Thus all these randomizer seed problems, whether it is possible to lose items or not, whether we want to just finish the game or complete it, are NP-Complete.

5 Algorithms to Randomize Games Efficiently

As the randomizer seed problem is NP-Complete, we cannot just generate random seeds and then verify if the game can be finished, as it would take an exponential time (and that is not even considering that with most games, the proportion of fully random seeds that can be finished is very small). Thus to randomize games, we need to build seeds in such a way that the game can always be completed. For this algorithm, we will consider that games do not allow item losses, and that the goal is for the game to be completable.

The basic idea behind these algorithms is the following: We first search for all currently accessible resource locations (where an item can be placed). Then we choose an essential item (an item that locks at least one location in the game) that allows us to reach at least one new resource location, and we place this item in one of the currently accessible resource locations.
We can then update the accessible resource locations, and repeat this whole process until all resource locations are reachable. At this point, we know we can get to any resource location in the game, and we can just randomly place every remaining item in the remaining available locations.

By placing items this way, we ensure that there is a solution to complete the game (which is to pick the items in the order they were placed by the algorithm), as we always place items in locations accessible at the moment.

5.1 Algorithm (first version)

The first version of the algorithm implements the idea previously described (with a few tricks to ensure it works):

Let $G_R = (V, E, R, F, L)$ be the resource graph modeling the game (reminder: $V = (V_F, V_R)$ with V_F the set of fixed/no resources nodes and V_R the set of resource nodes), and $Start$ and End be the start and ending nodes respectively. Finally, let I be the set of starting items.

The pool of items to be randomly placed R is split into two disjoint subsets, one subset P_I of primary items (items that are needed to reach some resource nodes), and another subset S_I of secondary items containing the remaining items.

Note that we consider without loss of generality that every location can only contain one item (as this can be circumvented and still allow for multiple items in one location of the game by having multiple neighboring resources nodes linked to each other to model one location). This allows to ensure the number of resources is at most equal to the number of nodes.

The goal is to build a seed (which is an assignment of the items from R to resource nodes in V_R), for which the algorithm proceeds as follows:

Note: In this algorithm, "reachable" means strongly connected to the starting node, i.e. we can reach the location from the "Start" node and then come back to it.

1. Build the set of reachable resources nodes from the $Start$ node with items I in our possession. We note this set of resource nodes as $Reach$.

2. Randomly choose one item $Item$ in P_I and verify whether it would allow to reach more resource nodes. If not, choose another item from P_I and repeat until we find an item that allows to reach at least one new resource node (if no items from P_I currently allow to reach new resource nodes, just keep one randomly).

3. Choose a random location Loc in $Reach$ and assign the chosen item to this location (we add a pair $(Item, Loc)$ to the seed). Then add the item $Item$ to I and remove it from P_I. The location Loc is now considered a non-resource node.

4. With the updated item set I, we update $Reach$, by computing the set of reachable resource nodes from $Start$ again.

5. We now choose another item by repeating from point b) again and again until $Reach$ contains all remaining resource nodes in the graph (at which point it means we can reach every item location in the game).

6. When $Reach$ contains all remaining resource nodes, just randomly place every remaining item in P_I and all items in S_I in the remaining resource locations (adding each pair $(Item, Location)$ to the seed), and return the seed.

This algorithm has a few issues though: it can only manage games without item loss (so for example, it would not work with dungeon keys in Zelda games); it can only place items individually (so if a group of items locks places as a group, for example a door locked by three keys, the algorithm places these keys after all other items that are useful on their own, and this group of items will never lock anything essential); and finally it gives too much importance to early places and items (places reachable early in the game have a very high probability to get an essential item).

This means that even if this algorithm can generate only seeds that can be completed, it also potentially misses a lot of other seeds that can be finished as well. Thus, the idea of the future versions of these algorithms is to eliminate these problems to improve the randomization and cover as much completable seeds as we can.

5.2 Parametrized algorithm

The second version of our algorithm builds on the first one by keeping the same structure, but adds some parameters to allow both a better randomization and a personalization of the seeds. Here are these parameters and how they affect the algorithm:

- α is a parameter that allows to modulate the linearity of the seed. Items are placed in a location newly reachable (location just made available by the last item we collected). This has two benefits: it increases the probability to have essential items in later locations, and makes the seed more "realistic" (most games when they are not randomized tend to have a mostly linear progression).
  This parameter is a real number in the $[0, 1]$ interval: $\alpha = 0$ has no effect, $\alpha = 1$ value makes the seed fully linear (essential items are always placed in locations newly reachable). Any value in between ($0 < \alpha < 1$) places linearly with probability α, and in any reachable location with probability $1 - \alpha$.

- β has a similar effect than α but in a more progressive way. Each location has a probability of being chosen, and this probability decreases over time (when a location is accessible but not chosen to place an item, its probability decreases). This allows to decrease over time the probability of placing an essential item in earlier locations. It is also a value chosen in the $[0, 1]$ interval, and is used as a multiplicative factor to the probability of already opened places each time an item is placed.
  In particular, this means $\beta = 0$ forces to place items in new places (same effect as $\alpha = 1$), and $\beta = 1$ has no effect. In between, the lower the value β is, the faster the probability for old places to get essential items decreases.

- γ works differently: it does not affect the choice of the location, but the choice of the item. We give a bigger chance to be chosen to items if they allow to reach more new locations (which should result in a higher number of possible seeds).
  This parameter is also chosen in the $[0, 1]$ interval: $\gamma = 0$ has no effect, $\gamma = 1$ affects to each item a probability that is almost proportional to the number of resource locations they would immediately allow to reach.
  γ also has another effect: if we currently have multiple reachable locations, we allow with a small probability the placement of an essential item even if the item would not yet allow to reach any new resource location.
  Here are the relative (sum not equal to 1) probabilities p_I given to items I, where N_I is the number of new locations reachable with I, and W the number of locations already reachable where we can place items):
  $p_I = 1 + N_I \cdot \gamma$ if $N_I \geq 0$
  $p_I = \gamma$ if ($N_I = 0$ and $W \geq 2$)
  $p_I = 0$ if ($N_I = 0$ and $W = 1$)

Thanks to these three parameters, we are able to control the seed generation to personalize seeds and/or optimize their randomness.

5.3 Complexity

The main reason we need this algorithm is to avoid the NP-Complete problem of verifying if a game with a completely random seed can be finished. This means our algorithm needs to terminate in polynomial time. To prove that, we analyze the complexity of our algorithm to show that it runs in polynomial time.

The main idea is as follows: let n be the total number of nodes (and thus the upper bound for the number of items), and m be the total number of edges. In the worst case, with or without the parameters, we need to place $O(n)$ items, and for each placement, we build for each item ($O(n)$) the complete reachability graph ($O(n^2 \cdot m)$). This makes a total complexity of $O(n^4 \times m)$, which is polynomial. The step by step analysis of the complexity is available in appendix C.

Note that this is an upper bound: with carefully thought data structures it may be possible to improve it. But it is sufficient for showing that our algorithm terminates in polynomial time.

As a result, both algorithms have polynomial complexity.

Next, we apply both variants to the ”Metroid Prime” game.

6 Application and Experimental Results

Now that we have our algorithm, we need to analyze the effects of these parameters on the generated seeds. Let us start with a small description of the game used as our case study.

6.1 Metroid Prime

The Metroid franchise in its entirety is a prime example of the types of games that randomizers were developed for in the first place, as the genre of ”Metroidvanias” games lends itself perfectly to a randomization of items (the first ”Metroid” game from 1986 is generally considered as the pioneer of this genre, being based in part on openness and exploration [16, 23]). In particular, the ”Metroid Prime” game (initially released in 2002 on the Nintendo Gamecube) is one of the most revered games of its era and has been ported recently on current consoles (”Metroid Prime Remastered” for Nintendo Switch in early 2023). This game also has no item loss, has a very active randomizer community. This makes ”Metroid Prime” the perfect case study to test our algorithm.

To resume shortly: the player explores a planet and has to reach the core. The player can collect up to 100 items on this planet to upgrade its equipment, including ammunition for weapons, items giving new abilities (double jump, beam weapons with electrical/ice/fire properties, suits with resistances to heat/gravity/radiations, etc), and artifacts locking the access to the core of the planet where the end goal is.

Our algorithm randomizes the game by shuffling those 100 items between their respective locations.

6.2 Modeling the game and implementation

All the implementation, from the resource graph to the algorithms and simulations, has been done with Python 3 (using mainly the ”networkx” library [10] to model the graph). We have created the full resource graph that models the ”Metroid Prime” game, and then ran both algorithms, including multiple different sets of parameters (both testing the effect of one parameter at a time, and then all three parameters together) to create multiple seeds for each parameter set. We then analyze these seeds by comparing distances and variances between all seeds to evaluate how the algorithms behave.

6.3 Experimental results

Before diving into the results, we need to explain a few more details: as a lot of items are identical (twelve artifacts, fifty missile expansions, etc) or not essential to finish the game or not locking other item locations, we only consider where the essential items (unique items unlocking new abilities) are placed, as these items are the ones that will dictate how the player progresses. We compute distances between seeds based on these essential items (the number of essential items not in the same place between the two seeds), and also analyze how often each location receives essential items.

As there are 17 such essential unique items out of a total 100 items, distances show how many of the 17 items are in different places between each pair of two seeds.
Each different set of parameters has been ran with 1000 seeds, and then distances are computed between each pairs of seeds.

First, a quick word about the time differences between with and without parameters: they have the same order of complexity, but there is still a constant difference in terms of speed: with my personal computer (ASUS Vivobook X515EA, Intel Core i5-1135G7 2.4 GHz, with 16 Go of RAM), most seeds are generated in between 0.2s and 0.4s seconds per seed for the non-parametrized version, and between 0.3s and 0.7s per seed for the parametrized version.

First, here are the distances between pairs of seeds (see fig. 1):

Distance	0	1	2	3	4	5	6	7	8	9	10
No parameters	0	0	0	0	0	0	0	0	0	0,00004	0,00092
One parameter	0	0	0	0	0	0	0,00014	0,00014	0,00029	0,00186	0,01357
All parameters	0	0	0	0	0	0	0	0	0	0	0,00033

Distance	11	12	13	14	15	16	17
No parameters	0,01638	0,20946	2,1834	17,0917	94,99858	333,00858	552,39094
One parameter	0,07229	0,455	2,84786	17,71086	91,43886	323,93643	562,52271
All parameters	0,00422	0,071778	0,965	9,79567	68,88444	303,16433	616,11422

Figure 1: Distances distribution with and without parameters (mean over 1000 distances)

The interesting fact we can see is that even with thousands of seeds, there are no seeds too close to each other (even with all different parameters variations tested, we barely found a few pairs that have a distance inferior to 10).

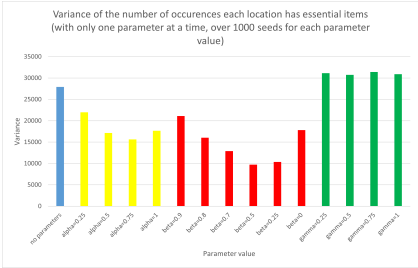
The graph shows two important properties: having only one parameter at a time has a very limited effect, but combining all three parameters improves the differences between seeds, as we have more pairs with max distance and less pairs for every other value.

One reason it does not show great results with only one parameter at a time is the following: α has slightly worse results on its own compared to the algorithm with no parameters, particularly with high values, which can cause parts of some seeds to be similar. γ is the best of the three parameters on this aspect, but still nowhere as good as all three parameters combined.

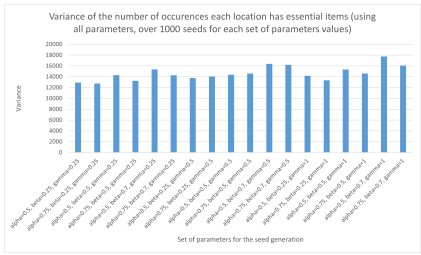
Now, let us focus on how often each location receives items. To this end, we count for each location how many times it receives an essential item, and then we compute the variance of these results. The lower the variance, the better the essential items are distributed along all locations. The results are shown in fig. 2.

The first effect we can observe for individual parameters is that it is the opposite of the distance: α and β both greatly improve the variance compared to the no parameters version, but γ is the worst and even makes the variance slightly higher than without parameters.

However, there is another very important effect of γ: there are a few locations (6 in total out of the 100) that are locked by a pair of items (”Charge Beam” and ”Super Missile”)



(a) Variance of the number of instances each location has essential items, with one parameter



(b) Variance of the number of instances each location has essential items, with all parameters

Figure 2: Variance of essential items in each location

that are only useful as a pair. As our algorithm only places items that allow to reach at least one new location by themselves, these 6 locations can never have any important item without the possibility given by γ to place items when they do not open new locations at the moment.
Even though γ slightly worsens the variance, it also gives items to these 6 locations a fair amount of times.

This is where the conjunction of all three parameters shines: we have items in these 6 locations quite often, we have the best distances (and thus the more varied seeds), and we have a very good variance (only beaten by some of β values when used alone).

We can also note one important result: as long as we do not go to the extreme of having a parameter maximized/minimized, all these sets of values for α, β and γ have very close results. When we add the fact that other games than "Metroid Prime" would have a different distribution of items and locations (number of locations and items and proportion of essential items would change, existence or not of groups of items, etc), it seems hard to find a real "optimized" value for each parameter that would work for every game. This is a positive as it means you can modify the parameters to personalize seeds (for example to have a seed more or less linear), and still have a very good randomness compared to the non-parametrized version. More recommendations can be found in appendix D.

7 Conclusion

Thanks to our new graph model, we were able to model how a randomizer in video game works, and to prove randomizer problems are NP-Complete. This implies we cannot really solve randomizer problems, and we need carefully crafted algorithms to reach a good randomness while avoiding falling into exponential complexity. We also showed an efficient algorithm to randomize games, that both ensures the game can be finished and provides a good level of randomness, improved with parameters at essentially no cost.

The best part is that it can also be applied to other types of randomizers like locations randomizers (if we model loading zones as nodes, we can randomize how these loading zones are linked to each other) or enemies randomizers (modeling enemies positions in the graph as well, and requirements to beat them if needed), to cite a few.

There is an important community of gamers that like to play games in their own way: mods, speedruns, randomizers, etc. We hope to initiate a promising research avenue between both the scientific and gamers communities by formalizing these domains and hopefully improving the quality of mods and games. To the gamers, we aim at helping randomizers to be applied easily to more games and to have a proved good randomization, which is not easy to obtain since these are NP-Hard problems. To the computer science community, we open the door to a new side of video game complexity and analysis. Finally we propose a new model based on graphs that could be of more general interest for video games and other domains.

References

[1] Greg Aloupis, Erik D. Demaine, and Alan Guo. "Classic Nintendo Games are (NP-)Hard". In: *CoRR* abs/1203.1895 (2012). arXiv: 1203.1895. URL: http://arxiv.org/abs/1203.1895.

[2] Steve Alpern. "Search games on trees with asymmetric travel times". In: *SIAM Journal on Control and Optimization* 48.8 (2010), pp. 5547–5563.

[3] W. Bailey. "Hacks, Mods, Easter Eggs, and Fossils: Intentionality and Digitalism in the Video Game." In: (2008), pp. 69–90.

[4] Graham Cormode. "The hardness of the Lemmings game, or Oh no, more NP-completeness proofs". In: *In Proceedings of the 3rd International Conference on Fun with Algorithms*. 2004, pp. 65–76.

[5] Erik D. Demaine, Joshua Lockhart, and Jayson Lynch. "The Computational Complexity of Portal and Other 3D Video Games". In: *CoRR* abs/1611.10319 (2016). arXiv: 1611.10319. URL: http://arxiv.org/abs/1611.10319.

[6] Erik D. Demaine, Giovanni Viglietta, and Aaron Williams. "Super Mario Bros. is Harder/Easier Than We Thought". In: *FUN*. 2016.

[7] Ethan Dereszynski et al. "Learning probabilistic behavior models in real-time strategy games". In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*. Vol. 7. 1. 2011, pp. 20–25.

[8] David Eppstein, Zvi Galil, and Giuseppe F Italiano. "Dynamic graph algorithms". In: *Algorithms and theory of computation handbook* 1 (1999), pp. 9–1.

[9] Michal Forišek. "Computational Complexity of Two-Dimensional Platform Games". In: *Fun with Algorithms*. Ed. by Paolo Boldi and Luisa Gargano. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 214–227. ISBN: 978-3-642-13122-6.

[10] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[11] Frank Harary and Gopal Gupta. "Dynamic graph models". In: *Mathematical and Computer Modelling* 25.7 (1997), pp. 79–87.

[12] Nathaniel Johnston. “The Complexity of the Puzzles of Final Fantasy XIII-2”. In: *CoRR* abs/1203.1633 (2012). arXiv: 1203.1633. URL: http://arxiv.org/abs/1203.1633.

[13] Froduald Kabanza et al. “Opponent behaviour recognition for real-time strategy games”. In: *Workshops at the Twenty-Fourth AAAI Conference on Artificial Intelligence*. Citeseer. 2010.

[14] Chieh-Ying Kan and Xudong He. “High-level algebraic Petri nets”. In: *Information and Software Technology* 37.1 (1995), pp. 23–30.

[15] Vassilis Kostakos. “Temporal graphs”. In: *Physica A: Statistical Mechanics and its Applications* 388.6 (2009), pp. 1007–1023. ISSN: 0378-4371. DOI: https://doi.org/10.1016/j.physa.2008.11.021. URL: https://www.sciencedirect.com/science/article/pii/S0378437108009485.

[16] *Metroidvania (Wikipedia)*. 20..-2023. URL: https://en.wikipedia.org/wiki/Metroidvania (visited on 10/23/2023).

[17] Iantorno Michael. “ROM Hacks, Randomizers, and Retro Games: Challenging Copyright and Remixing Zelda”. In: *DiGRA &#3919 - Abstract Proceedings of the 2019 DiGRA International Conference: Game, Play and the Emerging Ludo-Mix*. DiGRA, Aug. 2019. URL: http://www.digra.org/wp-content/uploads/digital-library/DiGRA_2019_paper_91.pdf.

[18] Chris Miles and Sushil J Louis. “Co-evolving real-time strategy game playing influence map trees with genetic algorithms”. In: *Proceedings of the International Congress on Evolutionary Computation, Portland, Oregon*. IEEE Press. 2006, pp. 0–999.

[19] Sean Monahan. *Video games have replaced music as the most important aspect of youth culture*. January 11, 2021. URL: https://www.theguardian.com/commentisfree/2021/jan/11/video-games-music-youth-culture (visited on 09/19/2023).

[20] James L Peterson. “Petri nets”. In: *ACM Computing Surveys (CSUR)* 9.3 (1977), pp. 223–252.

[21] Wolfgang Reisig. “Petri nets and algebraic specifications”. In: *Theoretical Computer Science* 80.1 (1991), pp. 1–34.

[22] Thiago A Souza, Geber Lisboa Ramalho, and Sergio RM Queiroz. “Resource management in complex environments: Applying to real time strategy games”. In: *2014 Brazilian Symposium on Computer Games and Digital Entertainment*. IEEE. 2014, pp. 21–30.

[23] *The undying allure of the metroidvania*. 2015. URL: https://www.gamedeveloper.com/design/the-undying-allure-of-the-metroidvania (visited on 10/23/2023).

[24] Giovanni Viglietta. “Gaming is a hard job, but someone has to do it!” In: *CoRR* abs/1201.4995 (2012). arXiv: 1201.4995. URL: http://arxiv.org/abs/1201.4995.

[25] Wallace Witkowski. *Video games are a bigger industry than sports and movies combined thanks to the pandemic*. Jan 2, 2021. URL: https://www.marketwatch.com/story/videogames-are-a-bigger-industry-than-sports-and-movies-combined-thanks-to-the-pandemic-11608654990 (visited on 09/19/2023).

[26] Quanjun Yin et al. "A semi-Markov decision model for recognizing the destination of a maneuvering agent in real time strategy games". In: *Mathematical Problems in Engineering* 2016 (2016).

Appendices

A Randomizer Problem Definitions

These are the formal definitions for the other variants of the randomizer problem:

Definition 4. ***Randomizer Seed Problem (Lossless, Finishable Game)**:*
Considering a game without item loss with a given resource graph $G_R = (V, E, R, F, L)$ (i.e. such that L contains no negative resources), and given a starting node $Start$, an end node End, and a seed S, the goal is to find whether it is possible to find a valid path from $Start$ to End.

Definition 5. ***Randomizer Seed Problem (Lossy, Finishable Game)**:*
Considering a game with a given resource graph $G_R = (V, E, R, F, L)$, and given a starting node $Start$, an end node End, and a seed S, the goal is to find whether it is possible to find a valid path from $Start$ to End.

Definition 6. ***Randomizer Seed Problem (Lossy, Completable Game)**:*
Considering a game with a given resource graph $G_R = (V, E, R, F, L)$, and given a starting node $Start$, an end node End, and a seed S, the goal is to find whether it is possible to find a valid path from $Start$ to End that collects all resources (the path passes at least once in each node of V_R).

B NP-Completeness Proof

Here is the full proof for the NP-Completeness of randomizer problems, as well as an example to illustrate it:

B.1 Proof

Proof. First, we show that the problem is NP.

Let us define N as the number of nodes and M as the number of edges. We select non-deterministically a path by selecting non-deterministically an order in which to pick-up

items (all items, both fixed and randomized, which for a graph with N nodes has maximum N nodes containing items) and then a path collecting these items in the selected order.
Each edge choice takes at worst M steps (going through all edges), which we have maximum N-1 edges to choose from one item node to the next, and we repeat that N+1 times maximum (from Start to N items to pick to End). That makes a complexity of $N^2 \cdot M$.
Then we have to verify that the path is valid (which means following maximum $N-1 \cdot N+1$ edges, and each time comparing the list of requirements of the edge with the list of items currently in the player's possession (comparing two lists of maximum length N takes maximum $O(N)$ if the items are always ordered the same way), which makes a maximum of N^3.
That makes a complexity of $N^2 \cdot M + N^3$, which is clearly polynomial to the entry (which is of length $N + M$).

Now for the NP-Hard part, we give a proof that the randomizer problem (Lossless, Completable Game) variant is NP-Hard by reducing from another known NP-Hard problem, 3-SAT:

We consider the entry for 3-SAT as such: variables are noted x_1 to x_N, and clauses (containing three variables each) are noted c_1 to c_K.
We build the graph for the Randomizer problem as follows:

- Create three nodes, "Start", "End" and "Satisfied", placed in V_F.
- Create 2N items, two for each variable x_i, that we name T_i (T for True) and F_i (F for False). These items are in R.
- Similarly, we create 2N nodes, two for each variable x_i, named P_i (P for positive) and N_i (N for negative) respectively. These nodes are in V_R.
- The seed S assigns each item T_i to the node P_i, and each item F_i to the node N_i.
- Create K nodes (one for each clause c_i), that are named C_i. These nodes are in V_F.
- We create two edges (without any conditions) from Start to P_1 and N_1 respectively.
- For each pair N_i and P_i for $1 \leq i < N$, we create four edges (without conditions) linking this pair to the next (from P_i to P_{i+1}, from N_i to P_{i+1}, from P_i to N_{i+1}, and from N_i to N_{i+1}).
- We create three edges from P_N to C_1, with three different conditions that are the three variables in c_1. Similarly, we create three edges from N_N to C_1 with the same conditions.
- For each C_i for $1 \leq i < K$, add three edges from C_i to C_{i+1}, with three different conditions that are the three variables from c_{i+1}.
- We create one edge from C_K to $Satisfied$ (no conditions), one edge from $Satisfied$ to $Start$ (with no conditions), and one last edge from $Satisfied$ to End with the condition that we need all items.

So we have our entry for the Randomizer Seed problem, which is $G = (V = (V_R = \{P_1, ..., P_N, N_1, ..., N_N\}, V_F = \{Start, End, Satisfied, C_1, ..., C_K\}), E, R = \{T_1, ..., T_N, F_1, ..., F_N\}, F = \emptyset, L = \emptyset)$ (with E containing all edges described), the seed S described earlier and the "Start" and "End" nodes.

In this graph, we can only ever reach the satisfied node if and only if the formula can be satisfied: from Start, we can choose either to go collect item T_1 or F_1 (this corresponds to choosing true or false for the first variable). We are then presented with the same choice for each other variable: pick either item T_2 or F_2, then either T_3 or F_3, and so on. Thus we pick exactly half the items (corresponding to an assignment of True/False to each variable).

To reach C_1 from N_N or P_N, we need to satisfy the condition on at least one of the three edges (i.e. we need to have at least one True variable in the clause). The same goes for each clause, thus to reach the "Satisfied" node, we need at least one item to validate each clause and reach the corresponding node (one variable True in each clause), and thus the "Satisfied" node can be reached if and only if the formula can be satisfied.

When we reach the $Satisfied$ node, we can then go back to the $Start$ node, take another path to collect the other half of items (those we did not collect the first time), and thus go back to Satisfied with all items and reach the "End" node. There is a valid path allowing us to reach the "End" node (and allowing us to get all items) if and only if there is a assignment of variables that satisfies the formula.

And that reduction is clearly polynomial: we build $2N + K + 3$ nodes, $4N - 2 + 3K + 3$ edges and $2N$ items, which is linear (and thus polynomial) to the number of variables and clauses. □

B.2 Example

Let us take the following boolean formula: $F = (x_1 \vee x_2 \vee \neg(x_4)) \wedge (\neg(x_2) \vee x_3 \vee x_4) \wedge (x_5 \vee \neg(x_1) \vee x_3) \wedge (x_5 \vee \neg(x_3) \vee x_2)$. The corresponding graph is given in fig. 3.

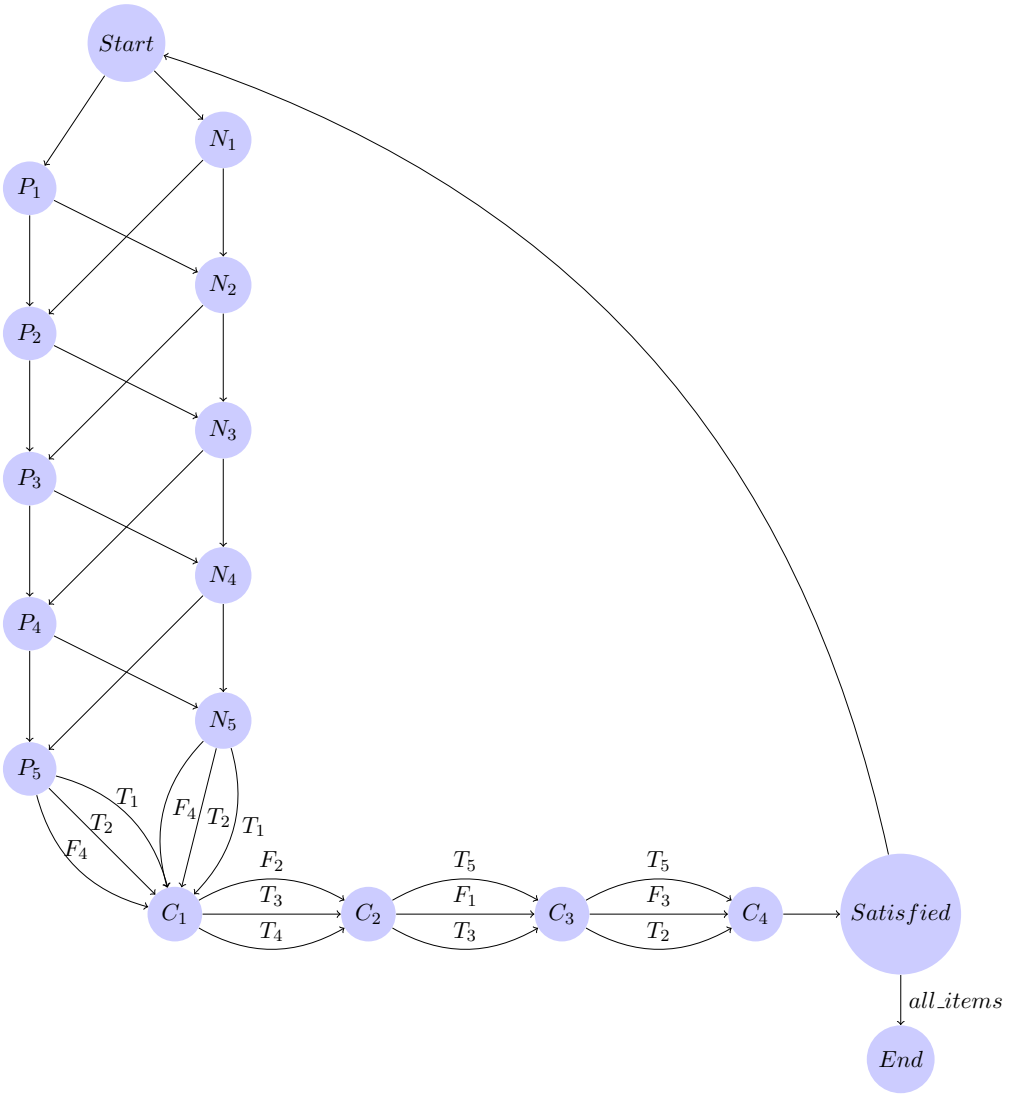


Figure 3: Graph created from formula F.

C Complexity Analysis

Here is the step by step complexity analysis of our algorithm. Let n be the total number of nodes, and m be the total number of edges in our resource graph:

1. The first step of building the set of reachable resources nodes works as follows: we go through all edges and check for all edges if it changes the state of at least one node, from unreached to one-way reached (semi-connected) to reachable (strongly connected), so for each edge we need to go through these lists of nodes. Thus we have a maximum complexity of n for each edge, thus $n \cdot m$ for all edges.
   And we repeat this process of visiting all edges as long as at least one node was updated, so in the worst case we may go through all edges, only update one node, and repeat this process 2n times (each node can switch categories twice: from unreached

to reached one-way, and from reached one-way to reachable).
This results in a maximum complexity of $O(n \cdot m \cdot 2n) = O(n^2 \cdot m)$.

2. The second step consists of choosing an item, computing the reachable resources with this new item, and in the worst case repeat for all items. So it has the same complexity as before multiplied by the number of items (which is at worst as many as the nodes), which gives us $O(n^3 \cdot m)$.

3. Adding the item to a location and removing it from the item list, we only need to go through the list of nodes once, so complexity of this step is $O(n)$.

4. We compute again the set of reachable resource locations with the new item added, so we have the same complexity as the initial step, $O(n^2 \cdot m)$.

5. Now we repeat steps 2, 3 and 4, once for each item to be placed (there are maximum n items for n nodes), so we do at worst n times the complexity of steps 2 to 4, which results in a total complexity of $O(n^4 \times m)$.

6. Then the last step consists in placing every remaining item in random places which means the same complexity as step 3, repeated once for each remaining item so maximum n times, which results in a complexity of $O(n^2)$.

And thus the final complexity for the whole algorithm is the sum of all that, which means it is $O(n^4 \cdot m)$ which is clearly polynomial to the entry (which is of size n + m).

Note that the parameters do not add to this maximum complexity: α and β just add different probabilities when choosing the location, which stays $O(n)$, and γ only forces to compute the full reachable graph for each item in each round, but that was already the case in the worst case scenario where we needed to go through each item. So the maximum complexity for that step stays the same and thus the order of complexity with parameters is exactly the same as without parameters: it is $O(n^4 \cdot m)$.

D Parameters Recommendations

Here are a few additional notes regarding the parametrized version of the algorithm:

- The $\alpha = 1$ value prevents a lot of possible seeds (as it only allows for fully linear seeds), so to optimize the randomness we recommend to have $\alpha < 1$.

- α and β interact with each other: with $\alpha = 1$ forcing linearity, β then has no effect. If $\beta = 0$, it also forces linearity, and α has no effect. But they complement each other as well: α works on a "binary" level in the sense that the choice is either made fully randomly, or only in newly opened locations, while β takes a more progressive approach.

- γ allows to generate some seeds that were not generated with only α and β (thanks to the placement of items when they do not yet allow to reach locations), but if its value is too high, it may also generate seeds that are very similar, if it always places the same items first (the items that allow to immediately reach a lot of locations, and thus have a higher probability).

- α and β have a small side effect: items that only open a few locations that are very far into the game that have a lot of other requirements will rarely be placed in early reachable locations (as an example, the ”Phazon suit” in Metroid Prime is only necessary to reach two locations very far into the game, which have a lot of other requirements). This can be balanced with γ ability to place items even when they do not yet allow to reach any new locations.