Technology
Arts Sciences
TH Köln

TH Köln – University of Applied Sciences
Faculty of Computer Science and Engineering Science

MASTER-THESIS

How to Go Fast

Ways to Model and Approach Speedrun Routing Algorithmically

Submitted to:
TH Köln, Campus Gummersbach
Course of Study:
Informatik / Computer Science (Master)
Specialization: Software Engineering
Advisors:
Prof. Dr. Boris Naujoks,
Prof. Dr. Dietlind Zühlke

Elaborated by:
Matthias Groß

Gummersbach, in April 2022

Abstract

Speedrunning in general means to go fast in a video game. This simple concept has an immense impact on the ways players engage with games. The inherent aspect of optimization within this esports niche makes it a good application field of optimization methods. This work gives an overview of speedrunning as an esports discipline. Prior works on this subject are discussed and assessed and relevant nomenclature is introduced. Using this information, routing — the procedure of planning a speedrun — is picked up as a graph optimization problem.

Nintendo's iconic game *The Legend of Zelda: Ocarina of Time* from 1998 is used as a working example to assess previous works and to explore more of this mainly uncharted field of research. To do so, the process of speedrun modeling is conducted as exhaustive as possible within the limits of a work like this. All relevant steps to obtain a faithful model are lined out. This procedure yields a partial game graph with 6764 nodes and approx. 321,022 edges — some uncertainty included. Current pathfinding techniques are discussed and assessed regarding their applicability to the presented speedrun routing problem. The resulting graph model and algorithmic approaches present a good reference to identify the most promising points of improvement. Challenges, flaws and possible solutions to still standing problems are discussed and assumptions from prior works are assessed. Finally, further tracks of scholarly and community work in this area are suggested and possible extensions to the approach are lined out.

It is shown that speedrun routing is not a trivial shortest path problem that can be solved with conventional methods. Furthermore, it is made clear that different games can have vastly differing routing circumstances. Assumptions of other works have been assessed using a working example and many interesting challenges and fields of further study and research in the field of speedrunning have been identified. Some probabilistic pathfinding methods and current works on AI agents for games pose interesting approaches which can be extended on by utilizing the findings of the presented work. Specialized modeling and optimization techniques have to be employed in order to have a positive effect on the speedrunning community.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Speedrunning in general means to go fast in a video game. As simple as this sounds, earning a top entry on the official leaderboards can take thousands of hours of planning and practicing. Over the last decade, an extensive community has formed around this esports discipline, and it is not only growing in size but also in professionalism. Many speedrunners stream full-time on the video live streaming website [twitch.tv]. and in 2018 *Counter Logic Gaming*, one of the biggest esports teams, contracted a cadre of speedrunners [CLG-creators].

Despite this growing popularity and professionalism, scientific works on the topic of speedrunning are still found only sparsely. And while there has been a number of works concerned with sociological and narrative impacts of the art of speedrunning [BB07; New08; Scu14; Scu16; Scu18; For18; New19; Hay20; Hem20; Ric21] scholarly work to support speedrunning ambitions with computer science is rare.

As this field is mainly uncharted yet, this work is to be understood as a first tap into this field of research. As a first vision and overview paper, [GZN22] formulates and assesses some basic principles with regard to algorithmic speedrun modeling and routing. Building on the findings and compilations of that overview paper, these insights are applied to a working example, discovering weaknesses and strengths of the proposed modeling techniques and deepening the general knowledge about speedrun routing as a field of computational optimization. Relevant contents of [GZN22] are reproduced where necessary. While not yielding an exhaustive model faithful to real speedruns, many interesting challenges and promising tracks of further improvement and research are identified.

The example chosen is the famous Nintendo title *The Legend of Zelda: Ocarina of Time* (hereafter *OoT*), developed and published by Nintendo in 1998 for their home video game console *Nintendo 64*. This game has been chosen mainly for three reasons. First, the game plays an iconic role both as a casual and as a speedrunning game, which has led to many tools and resources being available. Second, it has been the working example for [GZN22] so these findings can more closely be applied to this work. And third, the author of this work has some experience with the game, helping greatly in general understanding and interpreting the collected data.

Following this introduction, section *2 A Speedrunning Overview* presents basic relevant knowledge about speedrunning and introduces some speedrun terms. Section *3 Related Work* covers the existing range of works on the same topic of speedrun routing. After these basic points are settled, section *4 Modeling an* OoT *Speedrun* is focused on the actual working example.

The procedure of crafting a game graph representation of *OoT* is presented, starting out with an overview of the game and its speedrunning relevant mechanics in section *4.1 About* OoT. With this basic knowledge, section *4.2 Graph Representation* defines what a game graph of *OoT* will consist of. Section *4.3 Data collection* describes all tools, resources and procedures utilized to acquire all necessary data for the next steps. Before starting to construct a graph, a *4.4 Size estimation* is performed to gauge the possible dimensions of a game graph using the acquired data. This provides a basis for further decisions. Then, the actual graph construction is described in detail in section *4.5 A Sample Game Graph.* After the sample graph is constructed, section *5 Solving the Routing Problem* is concerned with routing algorithms. Existing pathfinding methods are discussed and their applicability to the speedrun routing problem is assessed. First approaches for specialized speedrun routing algorithms and supporting customizations to the model are lined out. Some challenges are already pointed out in the respective sections they emerged in. Being the main contribution of the work at hand, section *6 Identified Challenges* reiterates and summarizes the encountered challenges and discovered potentials, providing solution approaches if available. Where possible, the results are generalized to a context outside the working example. Finally, the findings of the work are summarized and multiple further lines of research that can pursue or support this endeavor are outlined in section *7 Summary and Outlook.* All referenced game titles will be credited in the *Ludography* at the end of the document, where a reference to *Code and Data Availabilty* can be found as well.

2 A Speedrunning Overview

In order to establish a proper model of a speedrun, some basic speedrunning concepts need to be introduced. This section will do so by summarizing the findings of [GZN22], specifically section *2 Speedruns and Categories*.

As it was mentioned before, speedrunning in general means to go fast in a video game. While this might sound trivial at first, really using all means at one's disposal to reduce the completion time for a game as much as possible can be a quite challenging task. As the community around this concept grew and began to form its own esports niche, the need arose to document rules for comparability and leaderboards for actual comparison. Over time, [Speedrun.com] became the de facto standard for speedrun leaderboards, which will also be used as reference point in this work.

There are some site-wide rules [SR.C-SiteRules], for example any usage of cheat codes or hardware manipulation is usually forbidden. One important note is the broad acknowledgment and usage of *glitches* throughout many speedruns. Generally speaking, a glitch is an irregularity or inconsistency in the game, such as programming weaknesses, oversights, or plain errors in the game. Such glitches, if reproducible, can often be exploited to make faster progress. Different existing definitions and categorizations of speedrunning techniques — e.g from Newman [New19], Scully-Blaker [Scu14] and Ricksand [Ric21] — are compiled and assessed in [GZN22]. Differing from the definition made there, the work at hand will adhere to the definition of glitches as it is formulated in the ruleset of the relevant speedrun *category* being discussed.

Different games' communities may define rulesets for a specific game. To differentiate between forms of speedruns for the same game, the concept of *categories* was introduced. Each speedrun category defines its own *ruleset* to which any run of this category must conform, making runs of the same category comparable to each other. Popular categories for many games include *any%*, *100%* and *Glitchless*.

any% runs are the least restrictive, often only denoting the ending state that has to be reached. This is essentially an "anything goes" category, where only the most basic rules like banned hardware manipulation are in place. This often leads to a lot of specialized techniques and glitches being used to reduce the completion time to a minimum, and this also often leads to a very unconventional playthrough. The name is derived from the fact that any percentage of the game might be completed before reaching the end, and the run will still be a valid *any%* run.

100% runs in a sense are the opposite to any% runs, as these rulesets require 100% of the game to be completed before reaching the ending state. Exact definition of what 100% means is to be stated in the corresponding category's ruleset. Games with an in-game percentage counter often use this counter as reference but any definition is possible.

Glitchless While any% and 100% qualifiers of a category mostly denote the amount of content that needs to be beaten, collected or otherwise included in a run to count as valid, the Glitchless qualifier specifies the rules in terms of allowed and banned techniques. Glitchless, as the name suggests, often refers to the ban of exploiting glitches in a category. What exactly is considered a glitch is to be specified by the ruleset, preferably unambiguously as to prevent misunderstandings in ruling.

Ruling and definition decisions are settled through community polls. The work at hand will focus on the *OoT Glitchless* categories and therefore adhere to their ruleset and glitch definition [OoTGlitchless], summarizing the matter as follows:

> "If the means are not a glitch, neither are the ends." So if you do not perform a glitch, you may skip any part of the game, regardless of how important. [OoTGlitchless]

This ruling is accompanied by an extensive list of permitted and prohibited techniques. It's important to note that there are multiple sub-categories of *OoT Glitchless* speedruns in place. Which one is referenced will be pointed out in the relevant sections.

Speedrunning a game can be formulated as reaching a defined ending state from a defined starting state [GZN22]. While these starting and ending states and the game contents in between differ heavily between games and categories, the underlying concept stays the same. In most cases, players cannot simply move directly to a game's end — if the gameplay mechanics include movement at all — but instead have to traverse its progression mechanics and collect items, avoid hindrances or meet other requirements to reach the ending state.

Optimizing this process usually takes a lot of practice to master what [GZN22] refers to as the *operational* level of speedrunning, i.e. giving the right inputs at the right times. Another identified layer of speedrunning are *tactical* considerations, encompassing decisions like which techniques to apply to efficiently defeat an enemy or circumvent other barriers of a game. The focus of this work however will be the *strategic* level of speedrunning: Planning the way through a game in a way that it takes minimum time, also known as *routing*, by optimizing the order in which to traverse the required game progression mechanics.

3 Related Work

Although rare, there have been other works on algorithmic approaches to speedrunning. One of the most recent works on this topic is the vision and overview paper by Groß, Zühlke and Naujoks [GZN22]. Besides the terminology already presented in the previous section *2 A Speedrunning Overview*, this work also provides basic concepts to take on algorithmic speedrun modeling and routing. A graph model is presented and major challenges are identified. The findings of this work will serve as a basis for the modeling decisions being made and the identified challenges will be assessed using the results of the modeling process. The relevant contents of [GZN22] will appropriately be reproduced at the respective points of the work at hand.

Lafond [Laf18] formulates the routing problem of the action-platform title *Mega Man* as well as its successors *Mega Man 2* and *Mega Man 3* on a graph model. One of the main progression mechanics of the Mega Man Series is the acquisition of power-ups by defeating stage bosses. Collected power-ups can then be used to speed up subsequently encountered stages. As the player has some degree of freedom in picking the next stage, the progression is not strictly linear and some combinations are faster to play through than others. By defining a function that maps each previously cleared stage to a specific amount of time save in another stage, and defining a dependency graph of stages, a route through this graph is searched that maximizes the total time save.

Another, more informal work by Iškovs [Išk18] discusses the "*all factions*" category of the Role-Playing-Game (RPG) *The Elder Scrolls III: Morrowind*. As a typical RPG, this title features many quests that can be completed and the selected category requires completing many of them. An exhaustive quest dependency graph is formed and through appropriate use of game knowledge, statistical data, node purging, evolutionary algorithms, visual computing methods and handcrafted customizations, a route is generated. Because this is a very complex category, at the time of writing only one run has been submitted and accepted to the leaderboards, which was based on the route discovered through Iškovs' work [Vol18].

Speedrunner *JaV* took on the popular car racing game *TrackMania Nations Forever* [Jst19]. The racing track they chose requires the player to visit a number of checkpoints on the track in any order. This allowed JaV to model the checkpoints as nodes of a graph and use a customized genetic algorithm to treat the problem as a slightly altered traveling salesperson problem, producing near-optimal solutions. The route they presented from this procedure ended up being fast enough to yield very good times and even a world record run.

Although not directly concerned with speedrun routing, the three best scoring contributions of the 2009 Super Mario AI competition [TKB10] used knowledge from the game engine of a *Super Mario World* variation to create a graph model and then decide on the most promising next input to advance the game. The resulting agents ended up playing the game in a very speedy fashion. This form of routing however is more in the realm of operational level optimization and the overall success of the agents can be attributed to the very simple progression mechanic to go right.

Aaronson [Aar18] employs a model with multiple layers called *planes of existence*. Each progression impacting item is tracked individually in a set-like collection. For every possible combination of items collected, a plane is created, generating up to n^2 planes for n items. Each of those planes contains the possible routing options for the given combination of items.

The other project of interest is by Mottare [Mot20]. They introduce the concept of requirements as additional attributes for graph edges and rewards as additional attributes for graph nodes to model the progression dynamics. Then, a customized Dijkstra's Algorithm [Dij59] is applied with requirement checking in place to find valid routes.

The first three works mentioned [Išk18; Jst19; Laf18] focus on model creation and solving the routing problem for specific games. Consequently, they yield very purpose specific tools specialized for the game and situation in question. The latter two projects [Aar18; Mot20] focus on solving the problem of routing using a very generic model in order to find a more general solution for multiple games.

None of these related works established a reliable and generalized framework to model and approach speedrun routing computationally that is supported by scientific research. Based on these works, the work at hand approaches the routing problem in *OoT*. The findings of the presented works are used as indications for modeling decisions and to assess the results. Besides finding a way to model this specific game, efforts are made to generalize the findings to a broader variety of games and categories.

4 Modeling an *OoT* Speedrun

In order to support the routing process algorithmically as detailed in [GZN22], the underlying problem has to be defined and formalized. Due to the problem's combinatorial nature, the similarities to pathfinding, and the general consent of other related work, this work will focus on graph representations of the speedrun routing problem. This section will present the process of modeling a speedrun on the working example of *OoT* in detail. Encountered challenges will be pointed out and discussed in more detail and in a broader context in section *6 Identified Challenges*.

4.1 About *OoT*

In order to model a graph for a specific game title, it is advisable to have good knowledge about the game. This section covers the game knowledge required to follow the course of the work. To help build a general understanding for the working example of *OoT*, some of the most important elements of the game are depicted in Figure 2 in a notation similar to the UML class diagram.

Figure 1 depicts an in-game screenshot of the game. The player controls the protagonist *Link* on his quest to defeat the evil *Ganon* and save the land of *Hyrule* and its princess, *Zelda*. However, the narrative of a game is of little interest to a speedrunner, having beaten the game thousands of times and more. Therefore, this overview focuses on the gameplay mechanics of *OoT*. This is done from a conceptual rather than a technical point of view. A more technical examination of relevant game elements will be conducted later on in section *4.3 Data collection*. To get a better grasp of the game world, it is very insightful to utilize the online interactive map at [OoTmap.com]. The interactiveness unfortunately is not fit for presentation in a work like this, but Figure 3 shows an overview.

Figure 1: A screenshot of *OoT*.

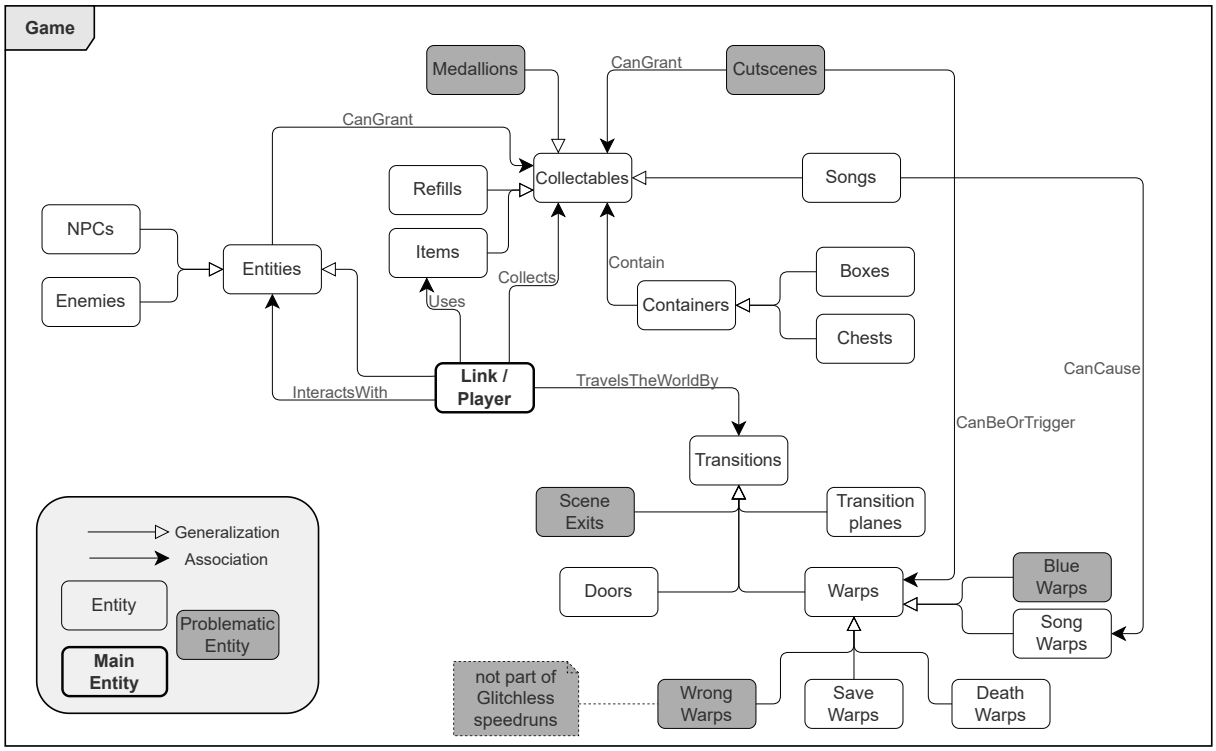

Figure 2: An overview diagram of *OoT* game elements as entities connected by relations in UML-style notation. Problematic entities are marked gray.

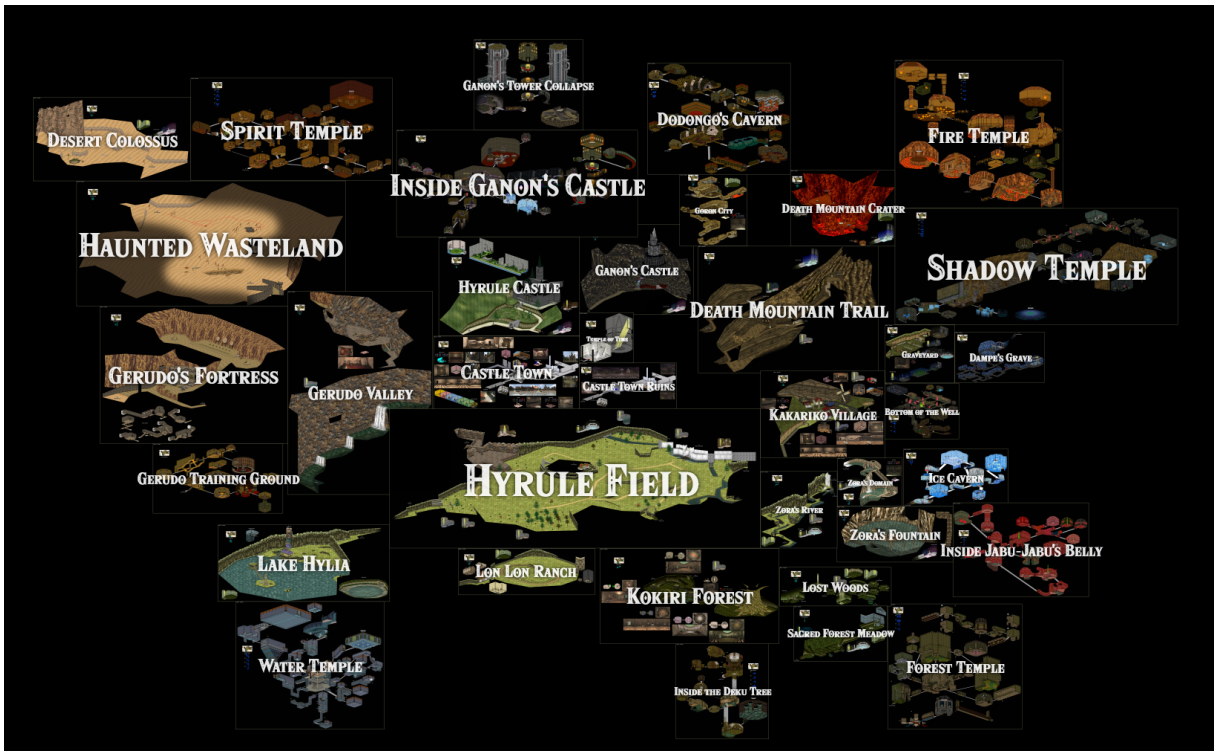

Figure 3: Screenshot of [OoTmap.com] on lowest zoom scaling, showing the game world of *OoT*. Scenes are arranged approximately accurate to geographic setting of the game world and labeled with white, game themed font; Copyright © 2017 Daniel Eck, MIT License [Eck21]

Looking at Figure 2 in detail, the protagonist *Link* is at the center. Many parts of the game revolve around the acquisition of many different items in order to reach places, beat enemies or finish quests. This is reflected by the many connections of the *collectibles* entity. The kind of those collectibles varies widely: They can be relatively unimportant *refills* like Rupees (the game's currency) found all over the world, or very important progression locking items found in *chests* deep inside dungeons. In terms of game progression, some of the most important collectibles are *medallions*. These are not simply found and picked up in the world, but rather are granted upon victory over one of the game's bosses. Defeating a boss or stepping into the activation zone of a *cutscene* invokes specific functions in the code, which in turn add the item in question to the players inventory, without encountering a representation of it in the game world. This dynamic ties these items more closely to the game's code than to its data. This poses a challenge, as code for the game is only available as a reverse engineered, manual decompilation project operated by the fan base [ZeldaRET]. Even with the source code available, code is much less searchable than organized data. The same reasoning holds for *Blue Warps*, which provide the player with a way back out of a dungeon after defeating a boss, usually in conjunction with a cutscene granting the reward.

Warping in general is a mechanic employed in many games and usually refers to the instantaneous transportation of the player to a remote location. Warps can be part of the intentional game design, like a fast traveling option in many games. Some warps result in the exploitation of other game mechanics not inherently intended for transportation.

Besides Blue Warps, there are three more types of warps with some import in *OoT* Glitchless runs: *Save Warps*, *Death Warps* and *Song Warps*. While there are only eight Blue Warps in the game — one for each boss fight — these other kinds of warps are more dynamic in behavior. Save and Death Warps are common in many other games' speedruns as well.

Save Warps are performed simply by saving and resetting the game. Many games do not place the player at the exact same spot they saved in when loading the game, *OoT* being no exception. This provides a quick method of transportation to specific places. When determining the spawn location upon loading a save file, the game checks Link's age and his last position. Link's age refers to the fact that one of the game mechanics includes Link being frozen in time for seven years. After doing this for the first time, the player is able to freely switch between the two timelines. These two states are commonly referred to as adult Link and child Link. If the player saved as child Link and not in a dungeon area, Link will respawn in his house in the *Kokiri Forest* area. When saving as adult Link outside of dungeon areas, he will respawn inside the *Temple of Time* — also the only place where the player can switch between timelines. The only exception from this rule is the case of saving inside Link's house as adult Link. This will spawn Link in his house again after loading the save file. If the player saves inside a dungeon

area, saving and resetting will spawn Link at the entrance of this area, regardless of age.

Death Warps are another method of quick transportation. As the name suggests, it involves Link's death. While this sounds cruel, after reducing the remaining hearts of Link's health gauge to zero and watching a short death animation, the game then asks the player if they want to continue the game, respawning Link upon the player's confirmation. The location of resurrection will always be the last entrance that the current area was entered by.

Finally, Song Warps refer to six songs Link learns to play on his Ocarina — a vessel flute instrument of great importance throughout the whole game. These magical songs have the ability to warp Link to one specific place each when played. This is obviously an advantage while traversing the game world and backtracking for newly reachable places. Save and Death Warps can technically be performed at arbitrary places, while Song Warps have some restrictions imposed on their usage. Incorporating these warps into a game graph is necessary as they present easy, fast and therefore heavily used transportation techniques. However, they will increase the problem size considerably. Section *4.5 A Sample Game Graph* will cover this in more depth.

4.2 Graph Representation

This section will cover the basic concept that is applied to design a graph for *OoT*.

The overview paper [GZN22] presents different ways to model speedruns as graphs. This work will elaborate on the *Weighted Game Event Digraph* detailed in section 4.1 of that paper, and then refer to the challenges identified in the same section when applied to *OoT*. The reference paper states:

> [...] [A] weighted game digraph is assumed
>
> $$G = (V, E, w), \tag{1}$$
>
> with nodes $V = \{v_1, ..., v_n\}$ as the set of all events relevant to the game's progression. Each possible traversal between these events make up the set of directed edges $E = \{e_1, ..., e_m\}$ between the nodes representing the given events. Edges are weighted with a function $w : E \to \mathbb{R}$, assigning each edge the time it takes to traverse between the in-game events in the given direction. [GZN22]

For the sake of simplicity, this definition of a weighted game event digraph will be referred to as a *game graph* from here on out.

The paper continues by identifying five major challenges of this model:

1. *Defining the Nodes*: "Events relevant to the game's progression" is not defined and it is nontrivial to do so.
2. *Defining the Edges*: The extent of "each possible traversal between events" has underlying restrictions that have to be defined as well.
3. *Dynamic Weights*: Edge weights are not consistent but rather change with graph traversal.
4. *Repeatable Events*: A subset of the events can be repeated once triggered, while others can not, further increasing the complexity.
5. *Multiobjective Optimization*: The model does not account for any dimension other than time in a possible route. [GZN22]

Regarding the first listed challenge of *Defining the Nodes*, the nodes represent at least "*all events $V_r \subseteq V$ deemed required by the category's ruleset*" [GZN22], as well as all events that can potentially speed up the run. Due to excessive community efforts, in the case of *OoT* there are a lot of data sources available for modeling purposes, which will be discussed in detail in *4.3 Data collection*. From these sources, it's possible to extract a representation of the entire game world. The geometry of the world has been excluded from modeling at this point. Using the game world's geometry can be important for operational or tactical considerations. On a strategical level however, only actual times of traversal between events are considered.

As to the second listed item, traversal between events is intuitively modeled as edges between nodes. The process of *Defining the Edges E* needs careful attention. As traversal between two game events can differ depending on the direction, directed edges are necessary. An intuitive option is to connect nodes after manually assessing the reachability in either direction. However, even with as few as one hundred nodes, this would be very time consuming, requiring $100(100-1) = 9900$ manual checks. As *4.4 Size estimation* will show, this is not feasible in the case of *OoT*. Using geometrical distances or even line of sight between nodes is not feasible as well, as this does not account for actors blocking the direct way like moving walls, enemies, gates etc. Geometrical data also does not benefit other means of pathfinding, which will be discussed in detail in *5 Solving the Routing Problem*. Another aspect of game world coherence can be utilized instead — the technical segmentation of the world into *scenes* and *rooms*. Section *4.3 Data collection* will cover this in detail.

Among others, all challenges listed in [GZN22] will be discussed and assessed by the results of this modeling process in section *6 Identified Challenges*. One of the biggest challenges identified by this work is the inevitable need for actual edge weights. Timings of every edge in a game graph that is to be incorporated into a route have to be supplied in order for many algorithmic approaches to be applicable (see section *5 Solving the Routing Problem* for details). Possible approaches to face this problem are outlined in section *7 Summary and Outlook*.

Other than that, data sources for *OoT* are plenty and the process of data collection is covered in the following section.

4.3 Data collection

One of the reasons *OoT* has been selected as the working example is the amount of effort the speedrunning community has already put into understanding the game on a technical level. As a byproduct of these efforts, there are a lot of resources available regarding technical details. Being the main means of communication, the *OoT* Speedrunning Discord server [OoTDiscord] lists many resources useful for speedrunning in its `#resources` channel. Some of these resources are also useful for modeling the speedrun routing problem [gla21; Dan14; mzx19; UltOoTSheet; SRMTables]. To minimize error potential, very close attention was paid to collect data that is as faithful as possible, by staying as close as possible to original data and its structure. In order to do so, data sources have been selected that draw their data directly from the game files whenever possible.

It's important to note at this point that *OoT* was released in multiple different versions. Because of technical preconditions, there are different releases for regions using the National Television Systems Committee (NTSC) or the Phase-Alternating-Line (PAL) analog television format. These releases have in turn been distributed in different versions. The re-release for the Nintendo Gamecube console is another version as well. All officially released versions are valid speedrunning options. The differences between versions are partly quite impactful on the speedrunning techniques that are used, adding another choice that has to be made by speedrunners.

Due to technical conditions, the PAL versions of *OoT* run at lower framerates. As the game's simulation engine is coupled closely to the framerate, this causes many speedrunning techniques to fail or differ drastically [OoTVersions]. Because of this, speedruns are almost exclusively performed on NTSC versions. For this work, the NTSC 1.0 version has been selected as it is frequently used for speedrunning. Besides the aforementioned reasoning, any version

could have been picked. The methods outlined in this work have been tested for other versions and are likely to work on any version of the game.

The main tool used for data collection developed by the speedrunning community member *mzxrules* is *VerboseOcarina* [mzx19]. This tool reads image files of *OoT* game cartridges — so called ROM files, or just *ROM*s — and outputs internal game data. This data can be used to create a model of the entire game world. To understand the importance of the retrieved data, the VerboseOcarina output is now explained briefly.

Figure 4 shows an excerpt of the VerboseOcarina output for a single *OoT* scene. Relevant lines for data collection are highlighted. The main element used to organize and represent *OoT*'s game world data are *scenes* (highlighted black ■). Generally speaking, each self-contained area of the game is represented as one scene. A scene can be as small as a single room of a house, or as big as an entire dungeon. If the scene needs to adapt to different game states (day or night, child Link or adult Link as explained in *4.1* etc.), this is done through different *setups* (highlighted yellow ■), which will be discussed later on in more detail. Each scene setup has a distinct set of *spawn points* (highlighted orange ■), also often referred to as positions, entrances, spawn actors or just spawns. These are mostly used to determine Link's location when entering a scene from another scene. Some scenes are subdivided into *rooms* (highlighted red ■). Unless glitches are used, exactly one room is loaded at any given time. Rooms also have setups associated with them. Room setups are congruent with the setups of their scene, so each room setup can be considered belonging to the corresponding scene setup. Rooms in turn contain *actors* (highlighted purple ■), which account for the majority of the game's interactions. Enemies, NPCs, collectibles, interactable objects etc. are represented by these actors. To reduce any ambiguities that might arise with the designations of different kinds of actors, henceforth these room-level actors are what is referred to as actors unless the type of actor is explicitly stated (e.g. spawn actors). To transit between the rooms of a scene, *transition actors* (highlighted light blue ■) come into play. Transition actors are things like doors or loading planes that exist in a given scene and load an adjacent room when Link passes through them. To do so, each transition actor holds the information of both adjacent rooms. Some instances of transition actors point to the same room twice. These transition actors don't load rooms but instead lead to *loading zones*. Loading zones, once entered, load other scenes.

As for setups, each scene has different states it can be in, each represented as a separate setup. To give an example, one setup can be used to represent the corresponding scene by day, another one for the same scene by night. Scenes have up to 4 different gameplay setups, labeled by integers from 0 to 3. In most cases, these correspond to the game world states as young Link by day (0), young Link by night (1), adult Link by day (2) and adult Link by night (3), respectively. Exceptions from this rule are explained in detail in section *6 Identified Challenges*.

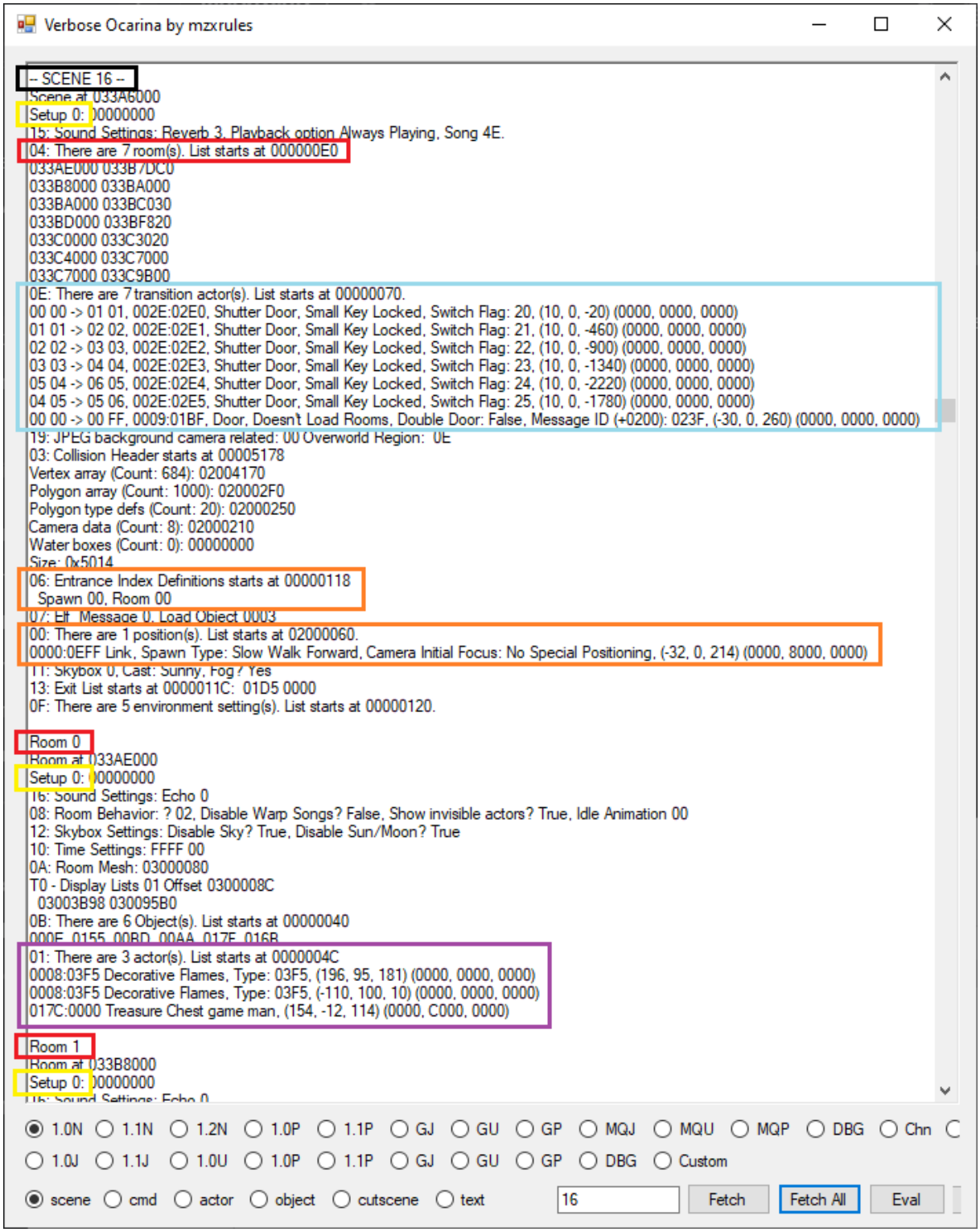

Figure 4: VerboseOcarina's user interface. Output field shows excerpt of text dump for scene 16. ■: scene index, ■: setup indices, ■: room data, ■: actor data, ■: spawn data, ■: transition actor data

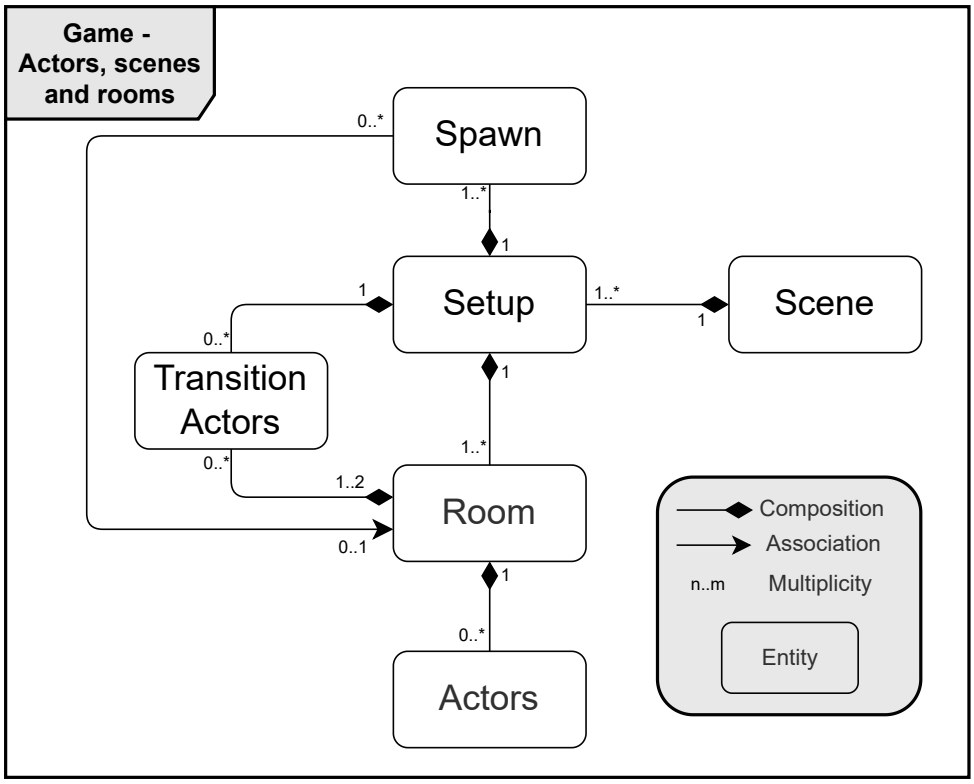

Figure 5: Technical Domain Model of *OoT* game data in UML-like notation. Diamond shapes depict owning end of composition relation. Arrow shapes depict navigation direction of association. Numbered line tags depict multiplicities.

Additionally a scene can have many different cutscene setups, the maximum used being 11, labeled by integers from 0 to 10. As cutscene setups unsurprisingly only come into play during cutscenes, they will not be taken into further consideration in this work.

Excluding loading zones, Figure 5 depicts the relations between the aforementioned game elements with UML-like associations.

The data dumped by VerboseOcarina can be used to define three types of actors: spawn actors, transition actors and room actors, or just actors. Scenes, setups and rooms on the other hand present hierarchical structures that contain these actors. These concepts can be used to create a game graph. As actors are used for everything the player can interact with, such a model encompasses the entire game world and, regarding static data, is exhaustive.

When interpreting the different actors as compound objects containing all information available, table 1 shows the data fields and types an actor can be associated with. The data types for the spawn actors are almost identical. The only exception is the `room` field not indicating the room the spawn belongs to, but rather the room being loaded and consequently, the `idx` field denotes the index of the spawn in the containing scene setup. Transition actors, having slightly different fields, are described by table 2.

Table 1: Data fields with types and descriptions for actors.

Data field	Data type	Description
scene	Integer	Index of scene containing the actor
is_cutscene	Boolean	Whether or not the containing setup is a cutscene setup
setup	Integer	Index of scene setup containing the actor
room	Integer	Index of room containing the actor
idx	Integer	Index of actor in its room
params	String	Parameters of the actor
description	String	Descriptive text
(pos_x, pos_y, pos_z)	(Integer, Integer, Integer)	X, Y and Z coordinates of the actor in the scene's coordinate system
(rot_x, rot_y, rot_z)	(Integer, Integer, Integer)	X, Y and Z axis rotation of the actor, sometimes used as additional parameter if rotation not needed [Eck22]

All of this information is obtainable through VerboseOcarina directly from a game ROM. Loading zones, however, present a deviation from the aforementioned game elements. They are represented as properties of the geometry. Entering a geometry polygon with a loading zone property triggers the transition. This circumstance makes discovering and identifying loading zones very hard without extensive knowledge about the game's geometry encoding.

During data research and development of the sample graph, other tools have been used for validation and exploration. The *OoT Interactive Map* website [OoTmap.com] — which also draws its data from VerboseOcarina outputs [Eck21] — was used to validate scene graphs, scene connections and actor placement. *SceneNavi* [Dan14] is a tool for displaying and editing *OoT* scenes with a 3D editor interface. This facilitated data exploration and validation massively. *The Ultimate OoT Spreadsheet* [UltOoTSheet] and the *OoT SRM / GIM Tables* [SRMTables] are community-built, extensive sources about the game's data and inner mechanics. A number of situations also had to be tested in-game. The *Ocarina of Time Practice ROM* [gla21; Mon] — also known as *gz* — developed by community member *glank* and other contributors, provides a multitude of features to facilitate this task, like free movement and teleportation trough the game world, arbitrary setting of inventory slots and many more. Another valuable source has been the *OoT* Speedrunning Discord server [OoTDiscord], where users have been very helpful and supportive in answering questions about item usability as well as Death and Save Warps (see Figures 21 and 22 in Appendix *Excerpt of* OoT *Speedrunning Discord conversations*).

Table 2: Data fields with types and descriptions for transition actors.

Data field	Data type	Description
scene	Integer	Index of scene containing the transition actor
is_cutscene	Boolean	Whether or not the containing setup is a cutscene setup
setup	Integer	Index of scene setup containing the actor
idx	Integer	Index of actor in its room
exiting_room	Integer	Index of room that is being exited by the transition actor
entering_room	Integer	Index of room that is being entered by the transition actor
transition	String	Original data describing the exiting and entering room with additional parameters. Format: XX PP -> EE PP; X: exiting room digits, E: entering room digits, PP: additional parameters
params	String	Parameters of the transition actor
description	String	Descriptive text
(pos_x, pos_y, pos_z)	(Integer, Integer, Integer)	X, Y and Z coordinates of the actor in the scene's coordinate system
(rot_x, rot_y, rot_z)	(Integer, Integer, Integer)	X, Y and Z axis rotation of the transition actor.

4.4 Size estimation

To get a grasp of the dimensions of a game graph of *OoT* built from the aforementioned data, it is advisable to perform a size estimation. The basis for this estimation is the data extracted from a *OoT* NTSC v1.0 ROM using VerboseOcarina [mzx19].

Table 3: Total and relevant counts of different game elements.

	Total count	Relevant count
Scenes	101	99
Setups	189	127
Rooms	494	417
Spawns	806	658
Actors	7031	5639
Transition Actors	638	456

Table 3 shows the total counts of different game elements and structures. The right column "Relevant count" displays the corresponding count excluding two scenes, which are used only for cutscene purposes, as well as all cutscene setups in other scenes. Spawns, actors and transition actors will serve as nodes in the game graph. Conforming to the explanations in section *4.3 Data collection*, scenes, setups and rooms act as hierarchical containers of these elements. This hierarchical structure will facilitate the graph design in section *4.5 A Sample Game Graph*. It's important to note at this point that the listed counts depict maximum values. E.g. there are some actors that only provide visual or sound effects. Other actors and spawns are not used in the game at all. Finding and purging these from the data can further decrease the resulting graph's size without contradicting its capabilities for routing. This will be discussed in more detail in section *6 Identified Challenges*.

Another insightful metric is the size of rooms and setups to expect. The scenes in *OoT* are sparsely interconnected. There are few ways to enter or exit a scene, and every scene is only connected to a few others. As a result, additional scenes will have minimal effect on the entire graph's size and density, regarding the connections with other scenes. A single scene and the containing rooms with their actors and spawns on the other hand can be expected to become very dense partial graphs. Figure 6 depicts the quantitative population of actors and spawn points per room. If a room has different setups, they are counted individually. Figure 7 depicts the quantitative population of actors, spawn points and transition actors per scene setup. Transition actors per room cannot be counted as they connect two rooms rather than belonging to only one. The designation Q_n is used for the nth quartile of the population and the interquartile range is defined as $IQR = Q_3 - Q_1$.

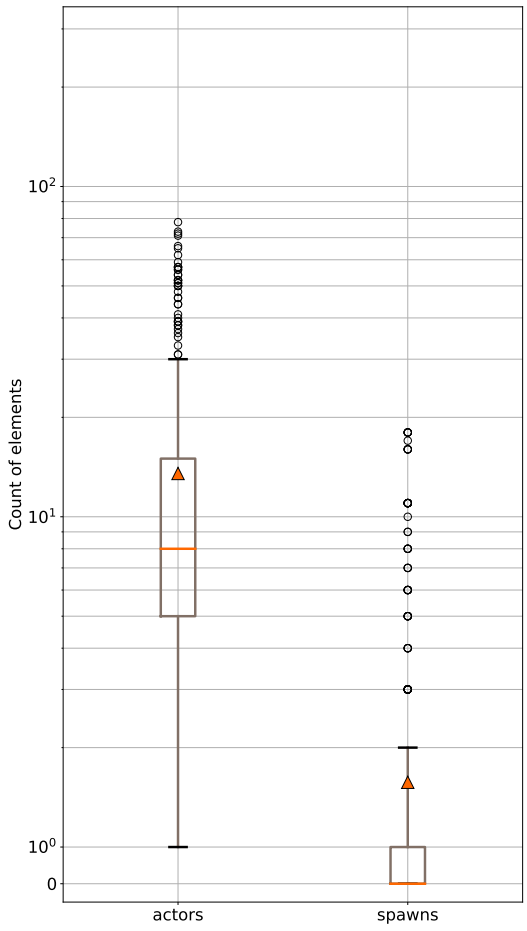

Figure 6: Boxplot of quantities of actors and spawns per room; 417 total. Orange ■ triangle: mean; Orange line: Q_2; Box: Q_1 to Q_3; Whiskers: $1.5 \times IQR$, or min / max if in this range; Circles: outliers.

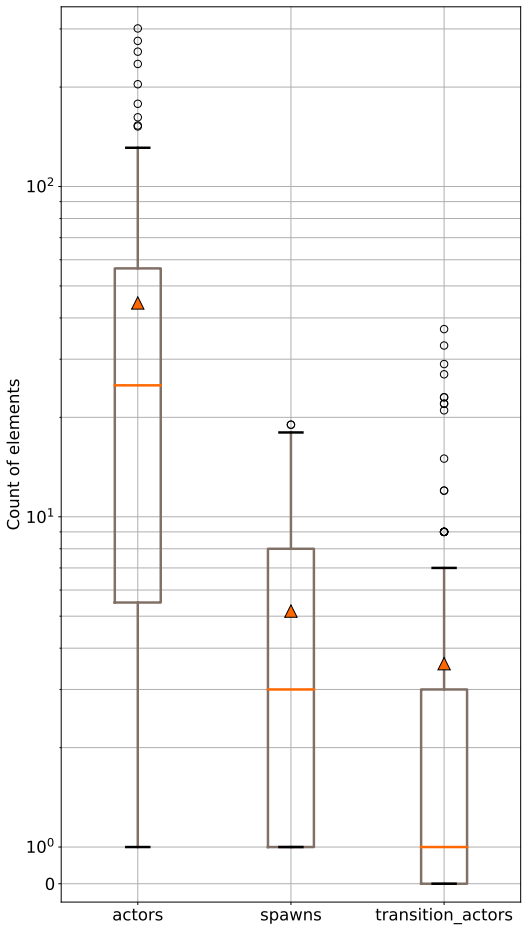

Figure 7: Boxplot of quantities of actors, spawns and transition actors per setup; 127 total. Orange ■ triangle: mean; Orange line: Q_2; Box: Q_1 to Q_3; Whiskers: $1.5 \times IQR$, or min / max if in this range; Circles: outliers.

Tables 4 and 5 show the corresponding data of Figures 6 and 7 respectively. Standard symbols are used, so N denotes the population size, μ is used for the mean value and σ denotes the standard deviation. Note that the actors make up by far the greater part of a room's population. Also note that most rooms (>50%) contain fewer than ten actors, with a Q_3 of only 15, and hardly more than five spawns. This reflects the fact that a lot of rooms represent NPC houses, shops or similar places, with only a few things to interact with.

Table 4: Selected statistics for quantities of actors and spawns per room.

	actors	spawns
N	417.00	417.00
μ	13.52	1.57
σ	14.25	3.32
min	1.00	0.00
Q_1	5.00	0.00
Q_2	8.00	0.00
Q_3	15.00	1.00
max	78.00	18.00

The rooms with a higher actor count represent more lively places, like overworld hubs and dungeon rooms. For example, the room with the maximum actor count of 78 is the Kokiri Forest main room. This is the first room the player will enter during gameplay after exiting Link's house, which itself only contains 3 actors.

Table 5: Selected statistics for quantities of actors, spawns and transition actors per scene setup.

	actors	spawns	transition_actors
N	127.00	127.00	127.00
μ	44.40	5.18	3.59
σ	57.30	5.02	7.06
min	1.00	1.00	0.00
Q_1	5.50	1.00	0.00
Q_2	25.00	3.00	1.00
Q_3	56.50	8.00	3.00
max	301.00	19.00	37.00

Assuming that all nodes n inside a defined bound of aggregation will form a complete directed graph, each of these bound's subgraphs will contain $n(n-1)$ edges, exponentially growing with each added node. It is advisable to keep these bounds in which nodes are aggregated and connected by edges as small as possible, while maintaining a feasible coherence between them. This will be elaborated on in more detail in *4.5 A Sample Game Graph.*

To get a better understanding of the practical traversal depth of an actual speedrun, a current speedrun has been chosen for reference. As this work's scope is confined to glitchless speedruns, the category for reference is "Glitchless Any% Unrestricted", which defines a ruleset disallowing most glitches but allowing some gray area techniques that are not allowed in the similar "Glitchless Any% Restricted" ruleset [OoTGlitchless]. For the sake of simplicity, this work will not differentiate between those rulesets but only cover Any% glitchless speedruns in general. The chosen speedrun is the former world record of *OoT* speedrunner *dannyb* [Dan21]. This reference run visits 143 of the 417 unique rooms in 50 of the 99 unique gameplay scenes of the game, changing rooms 223 times during the course of the run. This is a significant part of the game, as to be expected

of a speedrun category that does not allow for major glitches but requires the final boss to be beaten. Consequently a game graph has to represent an exhaustive mapping of the game world to be of any use for speedrun routing.

4.5 A Sample Game Graph

This section will cover the process of generating a game graph of *OoT* using the data sources listed in section *4.3 Data collection*. Underlying data structures are presented and critical parts of the resulting code is shown. All listings will be explained briefly in succession. Necessary manual customizations are pointed out. This section will have a strong focus only on modeling the working example of *OoT*. Arising challenges and implications are discussed in section *6 Identified Challenges* Conclusions and interpretations for other games and contexts are presented in section *7 Summary and Outlook*.

4.5.1 Preparations

In order to construct a game graph algorithmically, a data structure has to be decided on. To have a better concept of what needs to be included in a game graph as detailed in section *4.2 Graph Representation*, a UML class diagram is created and shown in Figure 8. The concepts of rewards and requirements from [Mot20] are included as well. Utilizing the possiblilities of Python, some details are not completely worked out in the beginning, e.g. the type of the `amount` field in the `Resource` class. This field is intended to keep track of amounts of any countable item, like bombs, arrows or Rupees. There are resources that can assume fractional values. The smallest possible unit of Link's health for example is $\frac{1}{16}$ of a heart of his life gauge. Python's dynamic type system can be utilized to mitigate this problem by using the relevant type when needed. In the case of errors arising, all amounts can be represented by the Float type.

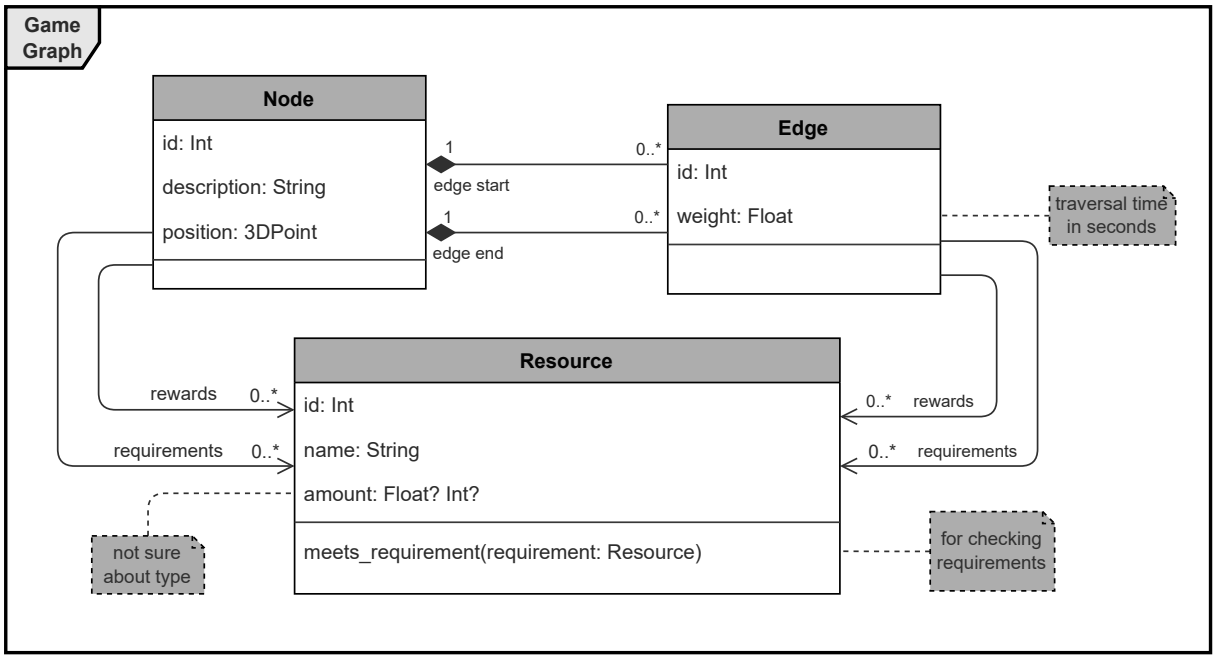

Figure 8: Initial game graph class diagram in UML notation. Diamond shapes depict owning end of composition relations. Arrow shapes depict navigation direction of associations. Numbered line tags depict multiplicities. Text line tags specify relation ends.

The data sources listed in section *Data collection 4.3* and the results of section *Size estimation 4.4* led to the following workflow:

First, VerboseOcarina is used to dump all relevant scene and room data from a *OoT* NTSC v1.0 ROM into a text file. The resulting text file accounts for 25,967 lines and therefore is not suitable for display in this work, but Figure 4 depicts an excerpt. The entire output file `all_scenes.txt` can be inspected in the GitHub repository [Gro22] in directory `/verbose-ocarina-parser/resources/`.

The VerboseOcarina dump is then parsed and processed by a Python script. Relevant data is extracted and transferred to CSV tables, which are much easier to work with in Python due to standard libraries. Actors, spawns and transition actors are listed in separate files and the columns of the tables correspond to the data structures displayed in Tables 1 and 2. The Python code produced for this task can be inspected in the `verbose-ocarina-parser` subproject of the GitHub repository [Gro22]. The resulting CSV files can be found in directory `/verbose-ocarina-parser/output/data/`.

With these CSV files, the Python libraries *pandas* [McK10; Reb+22] and *NetworkX* [HSS08; Hag22] are used to create the game graph. The pandas library provides data analysis and manipulation tools. The `pandas.DataFrame` class is used most and can be inspected in the pandas

documentation [pandasDoc]. NetworkX provides graph models and algorithms. During the course of this work, the data structure of NetworkX has been found to meet all upcoming needs. Used data structures can be inspected at the NetworkX documentation [NetworkX-Doc]. By using these libraries, well known, documented and tested frameworks can be used to minimize implementation load and error proneness.

The NetworkX library uses a structure relying heavily on the Python built-in class `dict`, which is a basic collection of `key:value` pairs with unique keys and standard dictionary operations. The `networkx.MultiDiGraph`, which is deemed the most fitting for a *OoT* game graph, employs a dict-of-dict-of-dict-of-dict structure. All graph classes of NetworkX can assign arbitrary data to both nodes and edges as attributes. This was utilized to implement the data fields of the respective classes shown in Figure 8. Nodes can be any hashable Python object. As Integers and Strings are immutable Objects and therefore hashable types in Python, each actor, spawn and transition actor is identified by their index in their respective list, which is then used as base for the node object. To prevent collisions between the lists, all nodes are converted to their String representation, spawns are prefixed with `'s-'` and transition actors are prefixed with `'t-'`. The resulting String objects serve as nodes.

The code of all listings in this section can be found in the `oot_graph_builder.py` file inside the `oot-graph-builder` sub-project of the GitHub repository [Gro22]. It is advisable to familiarize oneself with the built-in data structures of Python, especially the `dict` class as the presented code makes heavy use of this class. The same is true for the concept of Python's *comprehensions*. Both are explained in Python's documentation [PythonDoc].

To create a game graph, several conditions have to be considered. First, single scenes have to be modeled. Let the set of all 127 routing relevant scene setups in *OoT* be $S = \{s_0, s_1, \dots, s_{126}\}$. Following the suggestions of [GZN22], for any setup s in this set a *scene graph* G_s can be defined as

$$G_s = (V_s, E_s, w_s) \quad . \tag{2}$$

The set of nodes V_s is defined as the "*events relevant to the game's progression*" [GZN22]. In the case of *OoT* speedrunning, this definition relies heavily on the chosen category. All categories have in common that there can be very subtle reasons for a specific in-game element to be relevant. A single bush for example is not required to be cut in order to progress the run, but doing so can yield a bomb item pickup, potentially speeding up the rest of the route. To account for all possible cases, each of the actors in a scene is considered one node in the scene graph. There are obvious exceptions from this, like actors that provide sounds, visuals or other purely aesthetic effects, but most of the actors can be interacted with. Spawns are included into V_s as well, as they resemble the starting point for each entry of the scene. Later on, they

will also serve as connection points to other scenes. All nodes then are augmented with the information from VerboseOcarina, including the scene, setup and room they belong to, their description and instance parameters as well as their geometrical position and rotation in the scene.

For the set of edges E_s, the category plays an even more important role. As the Glitchless category, which is the situation in this working example, disallows the use of any glitches as defined in the category ruleset [OoTGlitchless], it can be assumed that the runner stays inside the intended boundaries of movement.

Edge weights are defined as the time it takes a player to traverse the game world from one node to another. Unfortunately, when it comes to the weighting function w_s, there is no trivial way to computationally assign weights to the edges. Section *6 Identified Challenges* will cover this in detail. Without a good way to compute weights, it is only possible to determine which edges exist and which do not, thus creating unweighted graphs. Actual edge weights only come into play within sample graphs for verification and testing. For these purposes, for all edges without weight information in a graph, the weight will be considered positive infinity. Also note that an edge in a finalized graph with actual weightings can have more than one weight, representing multiple forms of movement between nodes. For example a runner can move across large distances very quickly with a technique called "*Hyper Extended Super Slide*" (*HESS*), under the expense of at least one explosive, or traverse the space through other means, trading speed for item count. Having multiple edges per node pair and per direction, the respective graph is then considered a *multigraph*.

4.5.2 Constructing Room Sub-Graphs

For each of the j rooms in a scene setup, all n actors , as well as the m spawns that lead to this room, are connected by edges in a way that results in a complete digraph. This yields a room sub-graph $G_r = (V_r, E_r)$ with $|V_r| = n + m$ nodes and $|E_r| = n + m(n + m - 1)$ edges. This process is repeated so that there are j complete sub-digraphs, one for every room in the scene setup.

Transcribed to Python code, after providing all actor data in a `DataFrame` object, referenced by variable `scene_actors_df`, the statements in Listing 1 create a Python `list` with one graph object per room containing only actors.

```
rooms = {x['room']
         for _, x in scene_actors_df.iterrows()}
room_actors_dfs = [scene_actors_df.loc[(scene_actors_df['room']) == r]
                   for r in rooms]
g_rooms = [nx.complete_graph(x.index, nx.MultiDiGraph)
           for x in room_actors_dfs]
room_actors_dicts = [df.to_dict('index')
                     for df in room_actors_dfs]
for g_room, room_actor_dict in zip(g_rooms, room_actors_dicts):
    nx.set_node_attributes(g_room, room_actor_dict)
```

Listing 1: Python code to create graph components as complete sub-graphs for each room in a provided scene setup `DataField`. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `build_room_actors_components()` [Gro22]

Note that `nx` is the alias often used for `networkx`, as it will henceforth be referred to in this work. The first statement of Listing 1 composes a Python `set` of all room indices in the given `DataFrame`. The second statement of Listing 1 then splits this into a Python `list` of one `DataFrame` object per room. The third statement of Listing 1 then iterates over these objects and uses their `index` field — which is a `list` of all indices contained in a `DataFrame` — to create a complete graph for each room, using the entries of the index `lists` as nodes. The function `nx.complete_graph()`, as the name suggests, creates a complete graph from a container of nodes and a specified base class, using `nx.MultiDiGraph` is this case. The remaining statements then add the remaining actor data as node attributes.

The variable `g_rooms` now references a `list` of all room sub-graphs of a given scene setup. These sub-graphs so far only contain actors though. This process is repeated given a `DataField` object of all spawn data. When all resulting graphs are united by the function `nx.union()`, spawns and actors can be connected through the following code:

```
for spawn, s_room in spawns:
    for actor, a_room in actors:
        if s_room == a_room:
            g.add_edge(spawn, actor)
            g.add_edge(actor, spawn)
```

Listing 2: Python code to connect spawns and actors of a single scene setup. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `union_spawns_and_actors()` [Gro22]

Prior to Listing 2, all actors' and all spawns' nodes are coupled with their room id and referenced by `spawns` and `actors` as `lists` of `tuples`. The code than iterates over both `lists` and

if both nodes are in the same room, one edge per direction is added to the room graph `g`. This yields the complete sub-graphs of actors and spawns for each room.

4.5.3 Constructing Scene Setup Sub-Graphs

During the construction of scene graphs, different scene setups have to be accounted for. As explained earlier, different setups resemble different states of the game world — such as day and nighttime, or Link's childhood and adulthood worlds. As such, they must be treated as distinct scenes. This is taken into account by performing the described procedure, including the room sub-graphs, on all relevant scene setups individually.

To construct the scene setup sub-graphs, the rooms need to be connected using the transition actors. The transition actors are prepared to be injected in the graph by first adding them as nodes in a separate graph. Each transition actor is connected to each other transition actor with which it shares at lest one room connection. Expressed in Python code, this yields:

```
for it, it_data in nodes:
    for other, other_data in nodes:
        if it ≠ other:
            it_targets = {it_data['exiting_room'],
                          it_data['entering_room']}
            other_targets = {other_data['exiting_room'],
                             other_data['entering_room']}
            if not it_targets.isdisjoint(other_targets):
                g_transit.add_edge(it, other)
```

Listing 3: Python code to connect transition actors with each other. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `build_transition_actors()` [Gro22]

At the beginning of Listing 3, `nodes` is referencing a `list` of all transition actor nodes of a single scene setup with their corresponding data and `g_transit` references the graph object containing these nodes. As the two `list` iterations are nested, self comparisons must be prevented by the `if` clause in the third line. Then, two `sets` are created with the room connections of both transition actors. If these are disjoint, no room connection is shared. Otherwise, an edge is added. Only one direction needs to be added, again because of the nested iteration loops. The process of creating and comparing `sets` is a design decision, as Python `sets` have very good performance at containment checks like the one in Listing 3 and these checks are performed frequently during graph creation. Then, all the nodes of each adjacent room are connected to the transition actor node by an edge. This employs logic very similar to the

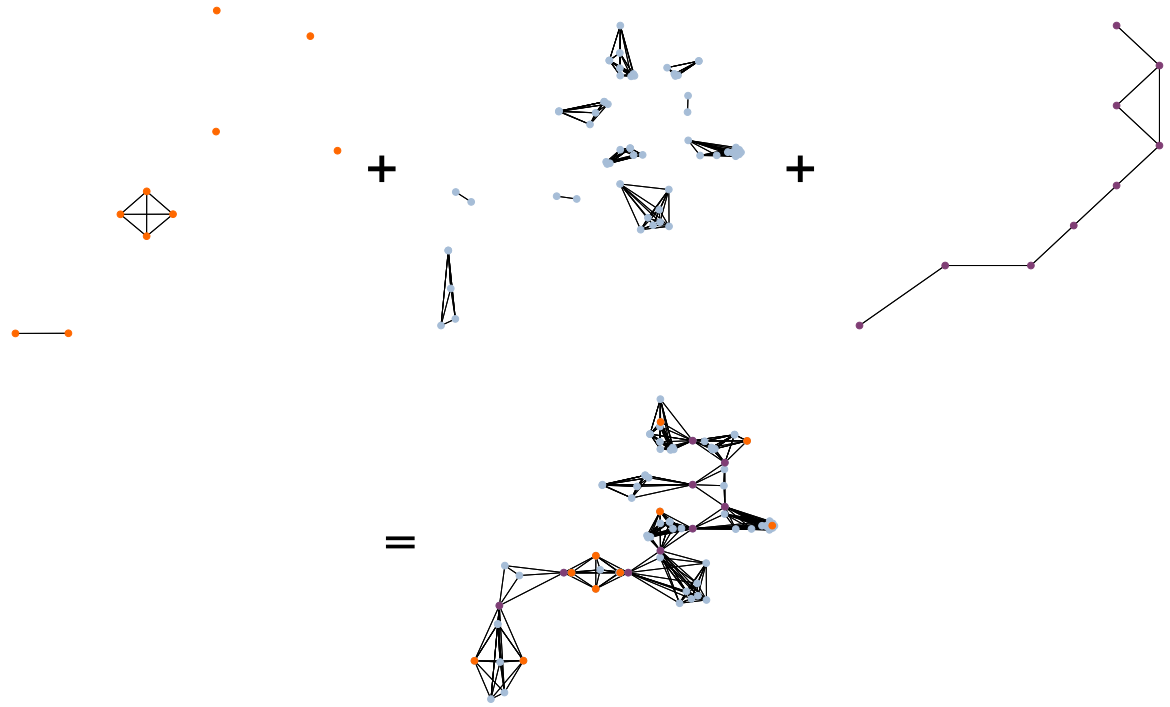

Figure 9: Visualization of graph construction. Upper graphs from left to right depict the sub-graphs of spawns, actors and transition actors of scene 91 setup 0 (Lost Woods, child Link). The lower graph depicts the full scene graph as the union of the sub-graphs. Purple ■ nodes: transition actors; Orange ■ nodes: spawn locations; Light blue ■ nodes: actors.

code of Listing 2 and is therefore not shown in a separate listing. This process is then followed by instructions to give the nodes a `'node_type'` attribute, labeling them as `'actor'`, `'transition_actor'` or `'spawn'`.

Following this procedure for all rooms and transition actors, the resulting scene graph comprises only one connected component. The steps of the process are visualized in Figure 9.

Figure 10 depicts setup 0 of Scene 85 (Kokiri Forest, child Link) in its graph representation. Note the two red circled nodes in the bottom left of the figure. The light blue colored node is part of room 0, but is placed in the region of room 2; Vice versa for the yellow colored node. This is because these nodes represent objects — a wooden sign and a bush — that are only used to prevent an often undesired effect in video games called *pop-in*. A pop-in effect is observed when an object or texture suddenly ”pops” into the vision of the player. This is often due to performance issues. In *OoT* this often is a side effect of how rooms are loaded. As soon as a transition actor, like a loading plane or a door, is passed, the next room will load immediately, with all of its actors and textures. To prevent pop-ins, some actors are placed as one additional instance in one or multiple rooms from which they can be seen. Therefore they serve a sole aesthetic purpose and are not of interest for the problem of routing.

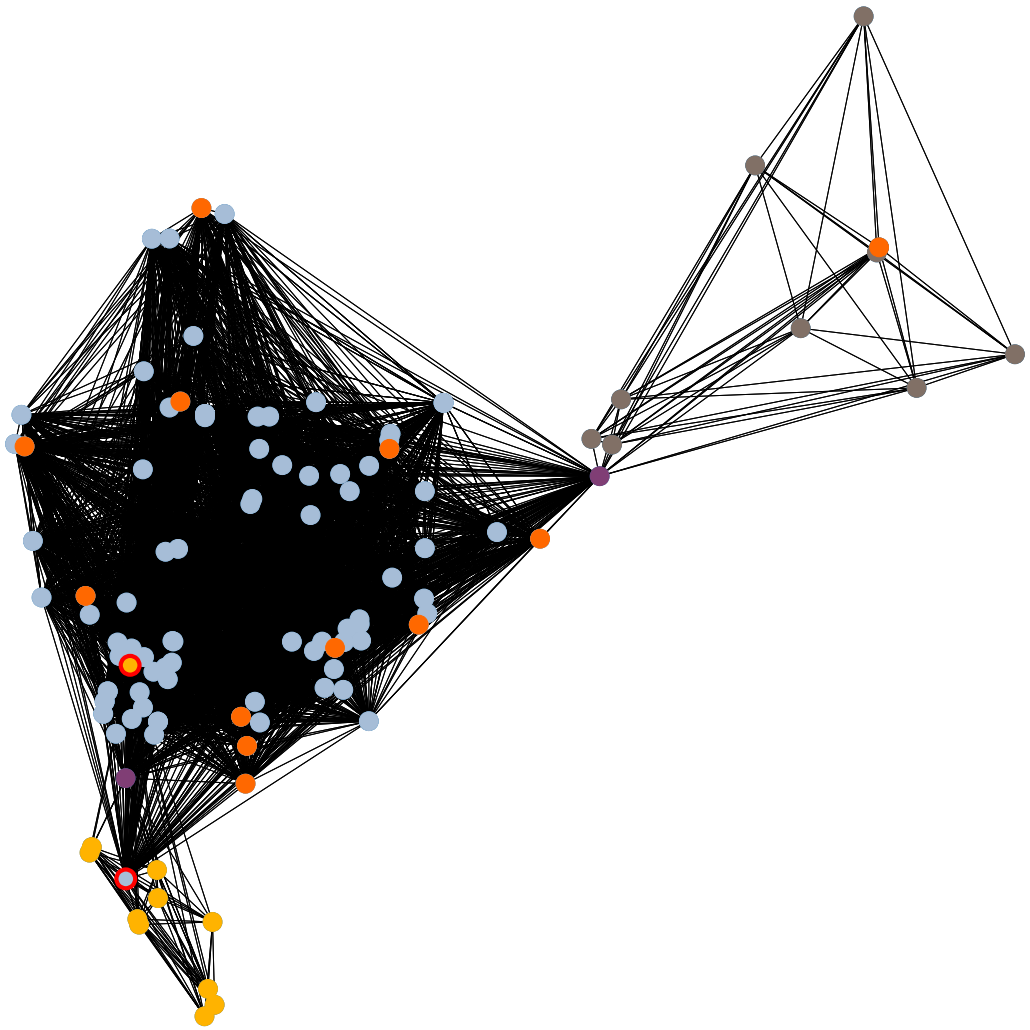

Figure 10: Setup 0 of Scene 85 (Kokiri Forest, child Link) as a graph representation, arranged by in-game X and Z coordinates. Light blue ■ nodes: room 0; Gray ■ nodes: room 1; Yellow ■ nodes: room 2; Orange ■ nodes: spawn locations; Purple ■ nodes: transition actors

Again, purging the data from nodes like these can reduce the problem size and will be discussed in more detail in the *6 Identified Challenges* section. An exemplary result for scene 85, setup 0 is shown in Figure 11, drawn on top of the scene's map from [vgmaps.com], which also provided the base maps for [OoTmap.com].

Figure 11: Setup 0 of Scene 85 (Kokiri Forest, child Link) as a graph representation, arranged by in-game X and Z coordinates; Purged from surplus nodes; Transparently drawn on top of the scene's map from [OoTmap.com]. Light blue ■ nodes: room 0; Gray ■ nodes: room 1; Yellow ■ nodes: room 2; Orange ■ nodes: spawn locations; Purple ■ nodes: transition actors. Map assets ©1998 Nintendo; Map composed by Peardian; cropped by author.

Figure 12 depicts two different scene setups with two different but equivalent graph layouts each. The graph of scene 85 setup 2 (Kokiri Forest, adult Link) at the top consists of 61 nodes spread out in 2 rooms, connected with 2508 edges. The graph for scene 91 setup 0 (Lost Woods, child Link) on the bottom consists of 80 nodes divided among 10 rooms, connected by only 792 edges. This comparison clearly underlines the need to aggregate nodes to complete graphs on a per room basis rather than on a per scene setup basis. The right layouts are equivalent to their left counterparts, but arranged by the NetworkX implementation of the Kamada-Kawai algorithm [KK89], a force-directed graph drawing algorithm based on the physical simulation of edges as springs. Using this layout, the nodes are spaced out much more evenly and nodes otherwise very near to each other do not obscure each other's edges, making the greater amount of edges in the upper scene 85 graphs obvious to see.

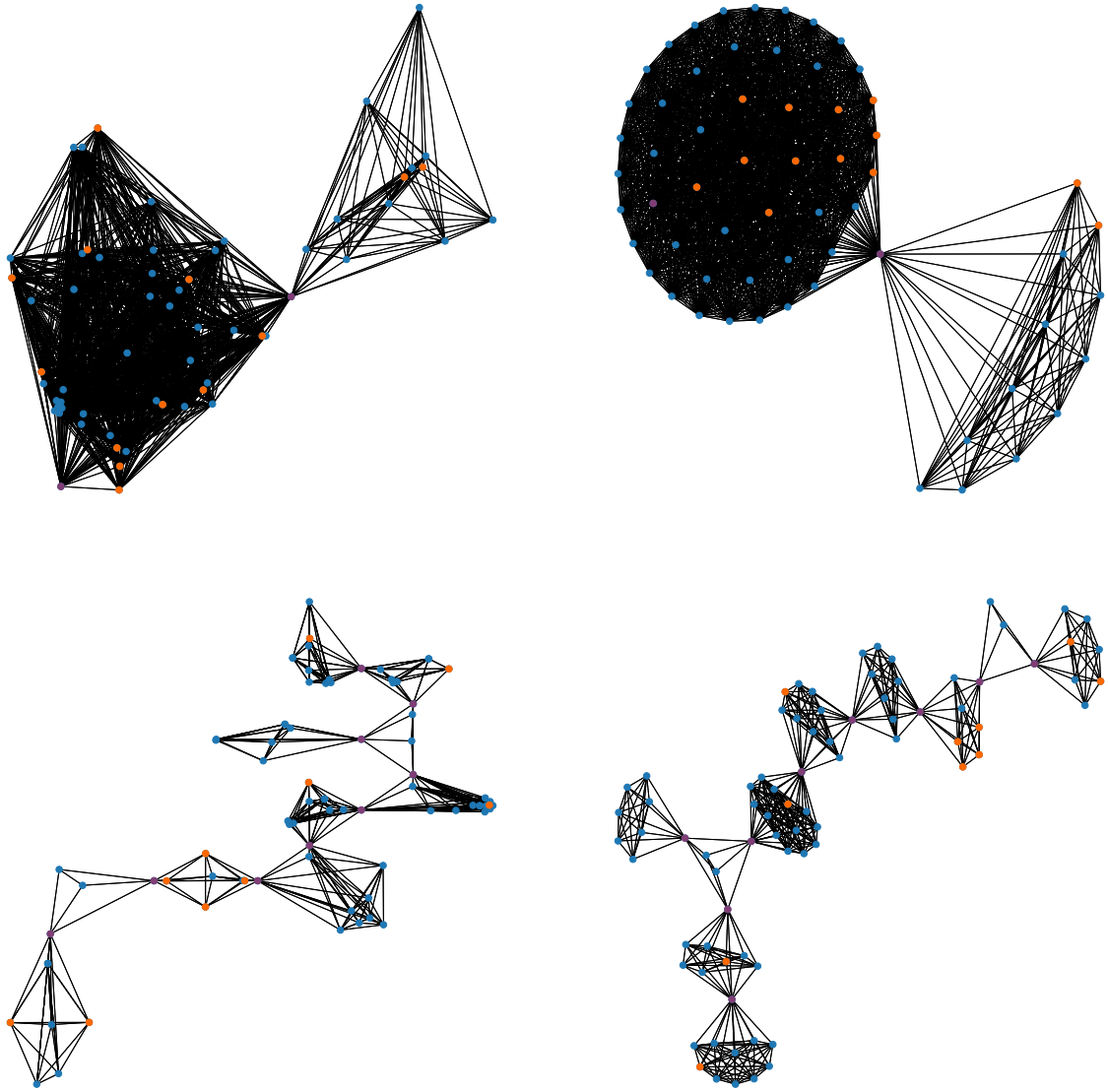

Figure 12: Two scene setups' graphs arranged with two different layouts. Upper row shows Scene 85, setup 2 (Kokiri Forest, adult Link); lower row shows scene 91, setup 0 (Lost Woods, child Link). Left layouts arranged by in-game X and Z coordinates; right layouts arranged by Kamada-Kawai algorithm [KK89]. Purple ■ nodes: transition actors; Orange ■ nodes: spawn locations; Blue ■ nodes: actors.

4.5.4 Composing the Game Graph

With the procedure to model single scene setups at hand, these sub-graphs have yet to be merged into a corporate game graph. This however poses a major challenge: As already briefly mentioned in *4.3* there is no trivial way to connect the scenes. All scenes have their own coordinate system and the exits of a scene are part of the scene geometry. This issue will be addressed more thoroughly in section *6 Identified Challenges*. For the exemplary purposes of this work, a number of sample connections can be added manually. Assuming mutual transitions, i.e. each transition from scene A to scene B also works vice versa, spawn points can be used. The spawn used when entering scene B from scene A is connected with the spawn used when entering A from B.

When all scene setups are built in this fashion and sample connections are in place, an approximate representation of the whole routing relevant game is created, consisting of **6764** nodes and **231,884** edges. Unfortunately, a force-directed layout of this graph doesn't produce useful outputs because of the missing scene transitions. A visualization of the graph with all scene setups lined up in rectangular cells and arranged by in-game coordinates is shown in Figure 13. All scenes are scaled to fill up their cells.

On closer inspection, this image presents some oddities, most obvious in the first column, second row from the bottom. This scene among others seems to consist of more than one connected component. In fact, when further inspecting the resulting graph data, it is found that, contrary to the earlier statement that each scene setup graph comprises only one connected component, four scenes do not adhere to this:

- Scene 12 (Thieves' Hideout): 6 components
- Scene 14 (Ganon's Tower (Collapsing)): 4 components
- Scene 62 (Grottos): 14 components
- Scene 76 (Ranch House & Silo): 2 components

All of these scenes feature only a single setup labeled 0. These four exceptions all have the same reason. They are scenes that are composed of multiple areas, which are entered through different spawns and from different other scenes. During normal gameplay — excluding the use of glitches —, it is also not possible to move from one of these areas to another one in the same scene, effectively making them isolated sub-scenes. Consequently, treating them as isolated sub-graphs is perfectly fine, provided they are properly connected to the entering and exiting scenes.

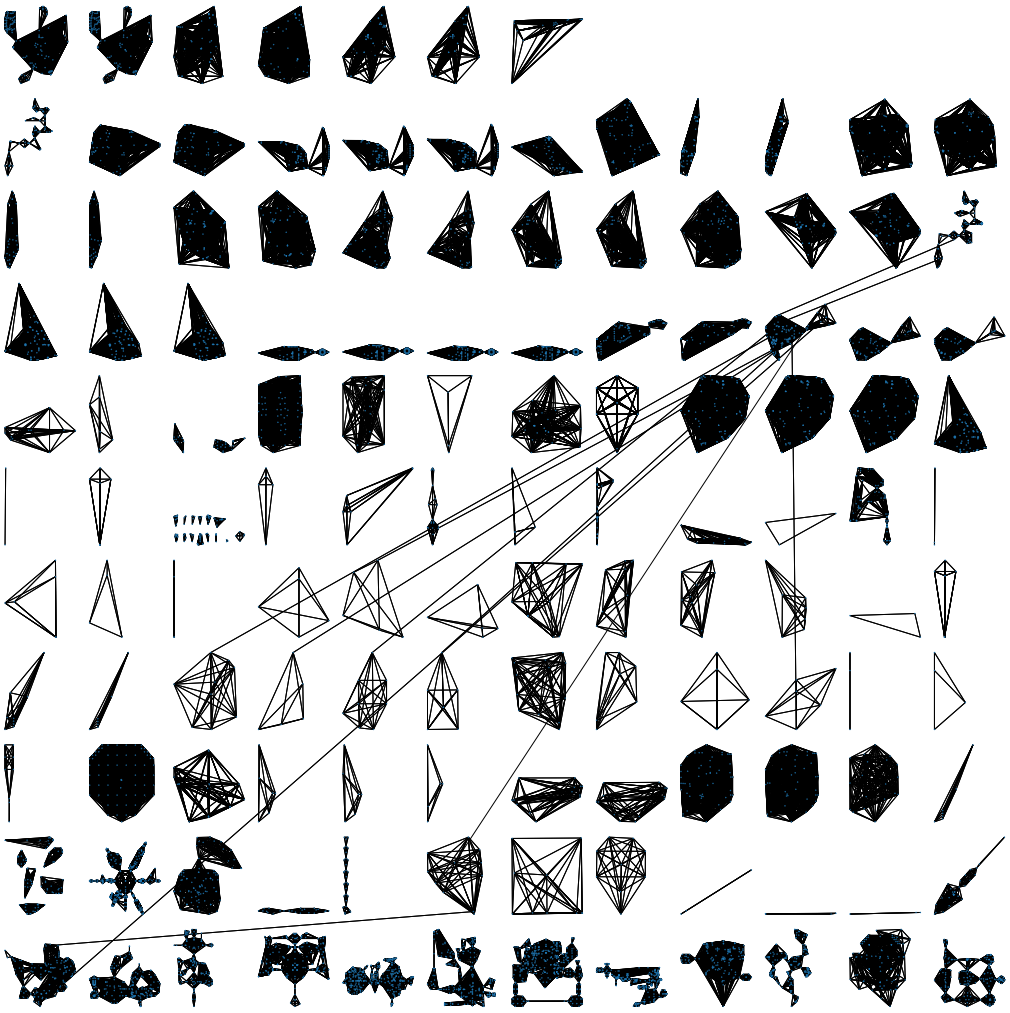

Figure 13: All 127 routing relevant *OoT* scene setups as graph components. Ordered by scene index, then by setup index. Arranged from left to right, then bottom to top. Scene 85 setup 0 (Kokiri Forest, child Link) on the upper right side has edges to all its connected scene setups. The dungeon scene 0 setup 0 (Deku Tree) in the bottom left corner is also connected to the separate boss room scene. Components are scaled to fit their cell. All nodes are blue ■.

4.5.5 Adding Warps to the Game Graph

A critical mechanic for speedrunning *OoT* that hasn't been included in the graph design yet is the warping mechanic. As described in section *4.1 About* OoT, the term warp aggregates a number of different instant transportation techniques useful for speedrunning.

Save Warps One of the most heavily used warps is the Save Warp, partly because it can be performed at any place in the game. As described in section *4.1 About* OoT, where Link ends up spawning after a Save Warp though depends on two factors: The location and age of Link when the game is saved. When saving in a *dungeon* scene, saving and loading the game spawns Link at the dungeon's entry. If saved in an *overworld* scene, i.e not in a dungeon scene, Link spawns either in his house (child) or in the Temple of Time (adult). This logic is expressed in the following Python code:

```
targets = {item
           for sublist in SW_SCENE_TO_TARGET_SCENES.values()
           for item in sublist}
sw_scene_to_spawn = {s: n
                     for n, data in nodes
                     if ((s := data['scene']) in targets
                         and data[NODE_TYPE_LABEL] == SPAWN_LABEL
                         and data['idx'] == (0 if s ≠ ADULT_OW_SW[0]
                                             else ADULT_OW_SW[1])}
for n, data in nodes:
    if (s := data['scene']) in SW_SCENE_TO_TARGET_SCENES:
        for target in SW_SCENE_TO_TARGET_SCENES[s]:
            g.add_edge(n, sw_scene_to_spawn[target])
```

Listing 4: Python code to add Save Warp edges to a given graph. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `add_save_warps()` [Gro22]. Displayed code is slightly simplified for better readability.

Prior to this code, a `dict` is prepared, mapping all source scenes to a `list` of Song Warp target scenes. There can be multiple target scenes, because a single node can lead to two different targets based on Link's age. This `dict` is referenced by `SW_SCENE_TO_TARGET_SCENES`. Also, a list of all nodes of the graph object `g` is referenced by `nodes`. The first statement of Listing 4 creates a set of all valid target scenes by iterating through said prepared `dict` and then through all the `lists` inside. The second statement comprises a `dict` comprehension that maps every Save Warp target scene to its very first spawn node. An exception is made for the Temple of Time, the target scene for adult Save Warps in the overworld, which uses a

different index for the spawn. Its scene index 67 and the used spawn index 7 are referenced by the `tuple ADULT_OW_SW`. This exception renders the comprehension somewhat hard to read, but this way the graph nodes only need to be iterated through once. One more iteration of the nodes is of course necessary to actually add the edges, which is done in the remaining lines of Listing 4. All nodes of graph object `g` are iterated and an edge is added for every Save Warp target scene found in `SW_SCENE_TO_TARGET_SCENES`. The actual nodes to connect to the edge's end are taken from the previously compiled `dict`. Only one direction must be considered as a Save Warp only works one way. Another exception needs to be implemented, which is the case when Save Warping inside Link's house as adult Link. This does not respawn him in the Temple of Time, but back in his house again. This is done by treating Link's house as a dungeon in the `SW_SCENE_TO_TARGET_SCENES` dictionary, with the effect of Save Warps leading to spawn 0 of the same scene, regardless of age, which is correct for this case. Applied to the previously created game graph, the total amount of Save Warp edges is **10,430**.

It's important to mention at this point, that the target nodes of Save Warps are not presented by the collected data. It can be assumed that because of the dynamic nature and contextual differences of save warps, their targets are defined in the respective code, rather than in the static data. Because of this, Save Warp targets had to be defined manually and cannot safely be applied to other versions of the game.

This procedure, as well as most other forms of warping, is guaranteed to create parallel edges, as many edges that are added through Save Warps already existed in the graph. Tracking these individual edges is important as they can differ vastly in traversal time and thus in edge weight, presenting distinct routing options. Here, the multigraph capabilities of NetworkX's `MultiDiGraph` class come into play. Adding identical edges, i.e. edges with identical starting and ending nodes, causes parallel edges to be created, which can be labeled by arbitrary keys. The weights of all Save Warps on the other hand can be assigned in bulk during creation, as the save and reset operations always take the same amount of time. Edge keys are not altered from the default behavior of increasing `int` labels, starting at 0.

Death Warps Death Warps are less complex to implement but add more edges to the graph than Save Warps. A Death Warp always places Link at the spawn by which he entered the current scene. The following listing depicts the respective code excerpt:

```
spawn_dict = collections.defaultdict(list)
node_dict = collections.defaultdict(list)

for n, data in nodes:
    key = (data['scene'], data['is_cutscene'], data['setup'])
    node_dict[key].append(n)
    if data[NODE_TYPE_LABEL] == SPAWN_LABEL:
        spawn_dict[key].append(n)

for k, nodes in node_dict.items():
    for node in nodes:
        for spawn in spawn_dict[k]:
            g.add_edge(node, spawn)
```

Listing 5: Python code to add Death Warp edges to a given graph. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `add_death_warps()` [Gro22]. Displayed code is slightly simplified for better readability.

Again, all nodes of graph object `g` are referenced by `nodes` as a list. The first `for` loop of Listing 5 iterates over all the graph's nodes and creates a `tuple` with its `scene`, `is_cutscene` and `setup` values. This tuple is used as a key. The node is then added to a pool of this key. Note that the dictionary used for this pooling is a `collections.defaultdict`. When trying to write to a non-existent key, this subclass of `dict` creates an object of a specific class as the value of that key instead of raising an error. In this case, a `list` is created to serve as a pool that the respective node is appended to. If the node happens to be a spawn node, it is also added to a dedicated spawn pool. This pooling procedure is performed to prevent more node iterations than necessary. The second `for` loop then iterates both `dict` objects and for every node adds one edge to each spawn node from the equally keyed pool. This process produced one loop, i.e. edge with identical start and end nodes, per spawn node. This is intentional as Death Warps can have beneficial side effects, like skipping a cutscene, even if the scene has just been entered. This procedure amounts to **43,303** Death Warp edges in the *OoT* graph.

Song Warps Song Warps, after being acquired, present a fast and easy way to travel to six fixed points in the game, but can only be used in specific scenes. The *OoT SRM / GIM Tables* [SRMTables] list all scenes in the game on sheet `Item Usability by scene`, stating the usability of different items for each. This sheet features a dedicated column for `warpSongs`, presenting the desired information. Any number being present in a row denotes a restriction of use for that row's scene, with the exact number specifying the type of restriction. This however only changes texts and similar behavior, so that the presence of any content is the only information that matters here. Listing 6 shows the implementation of this logic.

```
for n, data in nodes:
    if data['scene'] not in NO_WARP_SONGS_SCENES:
        for dest in SONG_WARP_DESTINATION_LIST:
            g.add_edge(n, dest)
```

Listing 6: Python code to add Song Warp edges to a given graph. Excerpt of file `/oot-graph-builder/oot_graph_builder.py`; Function `add_song_warps()` [Gro22]. Displayed code is slightly simplified for better readability.

This code simply iterates over all the nodes of a given graph `g`, which are once again referenced by `nodes` as a `list`. For each node, after checking that the node is not in one of the scenes where Song Warps are prohibited, the list of Song Warp destinations is iterated and a corresponding edge is added. This aggregates to a total of **35,364** Song Warp edges in the game graph.

Blue Warps Blue Warps are a special case. There are only eight Blue Warps in the game. As for cutscenes, these cannot be discovered computationally from game data, but easily added manually to the graph.

Incorporating Warps into the Game Graph When all of these warp edges are added to the *OoT* game graph, the total amount of nodes stays at **6764** but the edge count increases from 231,892 to **320,915**. Regarding the scene transitions, *The Ultimate OoT Spreadsheet* [UltOoTSheet] lists 294 of the spawns on the sheet of the same name as "usable". The terms "spawn", "entrance" and "exit" are used inconsistently in the different sources, and while it is not entirely clear what this number means, it can be assumed that this is the amount of entrances in the game. This assumption is reinforced by similarities between this list and other sources for entrance data like wiki entries [CM-ET; ZSR-ET] and the SceneNavi [Dan14] "Exit Table" feature. Combined with the facts that each individual entrance covers four of the 1556 rows in these tables, and that some entrances are not used in the game, a number of 294 scene transition edges for the game graph is likely. Even if all 1556 rows would represent a single edge in the graph, this would not increase the problem size by a significant amount.

When drawn, unfortunately the graph layout becomes very cluttered with this many edges both with an in-game coordinate based layout and with a force-directed layout. Scene transition edges might help a little in the latter case. Both can be seen in Figure 14.

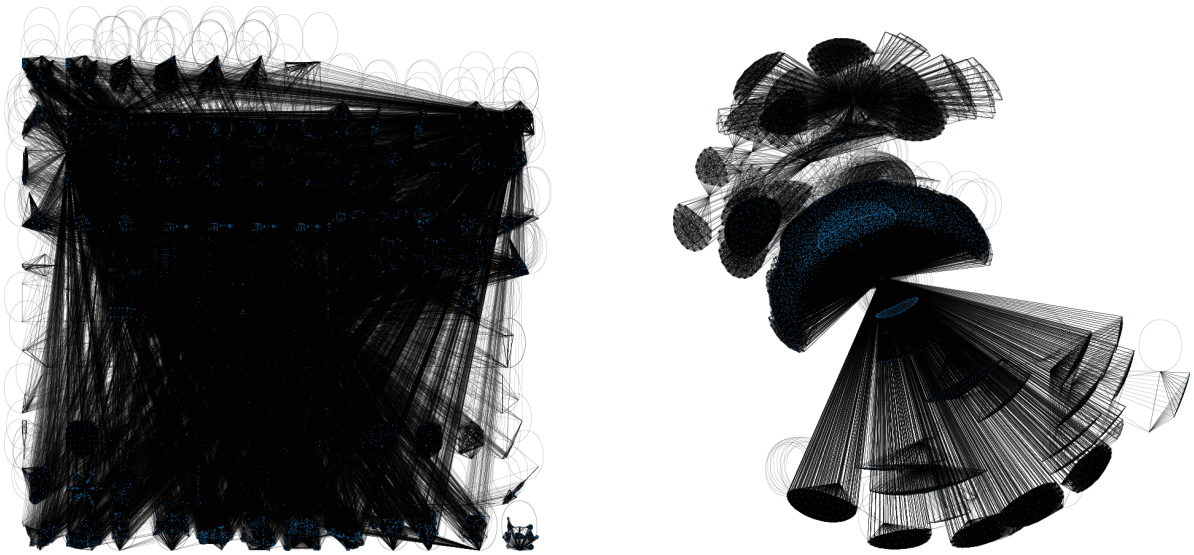

Figure 14: All 127 routing relevant *OoT* scene setups as graph components; Connected by sample edges and with added Save, Death, Blue and Song Warp edges; All nodes are blue ■. Left: Scenes arranged by in-game X and Z coordinates, ordered by scene index, then by setup index; Arranged from left to right, then bottom to top. Right: Graph arranged by Kamada-Kawai algorithm [KK89]

More Warps In a broader sense, the term warp can also include the usage of *Sun's Song* — a magical song that when played on the ocarina turns day into night and vice versa — and specific cutscenes, that when finished spawn Link in another location than where the cutscene was triggered, similar to Blue Warps. Another, more obscure kind of warp is the *Wrong Warp*, which exploits the inner warping mechanics of the game — mostly used with Blue Warps. This however is with broad consent considered a glitch and therefore not relevant for this work. Void Warps, as the resetting of Links position is sometimes called when he falls into a pit or otherwise goes out of the bounds of the game world, can also be utilized while speedrunning. Finally, the magic spell of *Farore's Wind*, once acquired, presents the ability to place arbitrary checkpoints one can return to at will. All these warp-like mechanics pose some challenges of themselves and will be discussed among many other things in section *6 Identified Challenges*.

After a game graph has been composed using the procedure presented in this section, a path through the graph has to be found that minimizes the cumulative weights of traversed edges. Although edge weights are not yet incorporated into the model, first considerations can be made as to how this task can be approached.

5 Solving the Routing Problem

Given the presented graph modeling methods, solving the speedrun routing problem comes down to a shortest-path problem, i.e. finding a path between two nodes — the nodes representing the starting and ending events of a speedrun — so that the cumulative edge and node weight in the path is minimized. The shortest path problem in networks is an extensively studied topic. This section will first elaborate on well understood and broadly applied methods of pathfinding and shortest path algorithms and put them in relation to the speedrun routing problem. Then, first concept ideas for a speedrun routing algorithm are presented.

5.1 The Shortest Path Problem and Speedrun Routing

Generally, Dijkstra's algorithm [Dij59] solves the shortest path problem in a network without negative edge weights. A common extension is the use of heuristics to speed up the search, as used in the popular A* algorithm [HNR68]. However, these algorithms do not account for any restrictions — like game state or item requirements in speedruns — imposed on edge traversal or edge weight changes during traversal. Defining an admissible heuristic, i.e. one that never overestimates the cost of reaching the goal, for A* is also non-trivial. A heuristic often used for A* is the Euclidean distance to the target node. Although there are coordinates available for all nodes in the *OoT* game graph, every scene uses its own coordinate space, unrelated to each other. Even when translated into a common space manually, seeing how the graph also includes shortcuts such as teleports and Save Warps or Death Warps, it is easy to see that the Euclidean distance is not an admissible heuristic. As an alternative, one could compute the shortest path to the target node from all other nodes, ignoring restrictions, as a heuristic. A speedrunner who does not need to adhere to any restrictions on movement through the game world always traverses the shortest possible path. Any restriction will lengthen this path, rendering this heuristic admissible. However, this still leaves the problem of traversing the game graph while respecting the imposed restrictions.

Tow further assess existing pathfinding algorithms, two general approaches for game graph traversal are discussed:

- Considering the game graph as topologically *static*, which would need a reordering of the graph data to incorporate the game state in the topology, or
- Considering the game graph as topologically *dynamic*, which would raise the need of tracking the game state for each path while traversing the graph.

Dynamic graphs have been subject to extensive research [HG97; CH66; Mic16; Lou83; Raj+15]. For example Cooke and Halsey define dynamic edge weights as follows:

> Given the matrix function $G(t) = (g_{ij}(t))$, not necessarily symmetrical, where $g_{ij}(t)$, the time required to travel from city *[i.e. node] i* to city *[i.e. node] j*, varies as a function of the starting time t at i. [CH66]

In fact most of the listed research regarding dynamic graphs treats the graph dynamic as a function of time or as stochastic values. This is due to the fact that many works in this field aim for the application in computer networks or driving directions for road networks, where this assumption holds true. Edges in a game graph as in *OoT* however do not vary as a function of time (for the most part), but rather as a function of the player's current inventory, progress and general game state, most of which is entirely in control by the player.

Incremental heuristic search methods for the shortest path problem in dynamic graphs can incorporate newly acquired knowledge of the graph on the fly. Examples of incremental heuristic search algorithms are D* [Ste93] and its extension Focused D* [Ste95], as well as LPA* [KLF04] and its extension D* Lite [KL02; KL05]. These algorithms are capable of finding a shortest path in a graph, incorporating newly found information about the problem space during traversal. Here, changes in edge weights due to new items being acquired can be interpreted as new information about the graph being acquired. The aforementioned problem of defining an admissible heuristic persists however. Furthermore, these algorithms focus on applications in unknown terrain that is explored in real-time, which facilitates applications in robotics and AI environments. A speedrun game graph on the other hand is either exhaustively known, or lacks some information which cannot be explored during real-time pathfinding. Applications in other settings could be promising though. A popular modification of the game is the *randomizer*, shuffling all items in the game randomly. Here, real-time suggestions for progression based on found items could be given.

When considering a graph as static, different speed-up techniques can be applied [WW07; Bau+10]. A common technique employed to reduce computational load is to preprocess the input graph in various ways, e.g. using Contraction Hierarchies (CH) [Gei+08], Labeling [Abr+11] or Landmarks (as in the ALT algorithm) [GH05]. The *Customizable Route Planning* algorithm [Del+11] for finding paths in road networks explicitly considers the topological information as static and exploits this circumstance by applying relatively slow preprocessing steps in order to be able to respond to queries on continental road networks in about a millisecond.

Although real-time application is not needed for speedrun routing, preprocessing can be used to transform the game graph into a multi-dimensional network in order to encode game state

into the topology. Each layer of the network would only contain the edges that are traversable with a specific set of items or requirements. Each node granting a reward to the player will then lead to a layer with edges unlocked by the added reward. This approach will be covered in more detail in *5.2 Algorithm Design*.

In addition to the already stated challenges when applying the presented methods to speedrun routing, two more major caveats can be observed. First, all of the presented methods require complete awareness — or at least explorability — of the search space, i.e. the game graph. While theoretically possible, for many games this would account for tens of thousands of individual graph weights — more than 321,022 in the case of the presented *OoT* model — that would need to be timed in order to produce such an exhaustive graph model. While this issue could be addressed by employing crowd sourcing mechanisms for edge timings and reducing the complexity through purging or contracting unimportant edges and nodes, other solutions are investigated in section *7 Summary and Outlook*.

Another main caveat is the assumption that all presented methods need a fixed target node. While a final target can easily be determined — the final boss actor in the case of most *OoT* categories — this target is not directly reachable from the beginning of the game due to edge restrictions. Other, intermediate targets need to be reached instead in order to meet edge traversal requirements, i.e. traverse the game mechanics towards the ending event. In a static, layer based graph model as described before, this can be accomplished, but would still need exhaustive information.

Another approach to engage the problem is the use of techniques such as ant colony optimization (ACO) [DD99; DG97]. ACO is a probabilistic multi-agent optimization algorithm, mimicking the behavior of ant colonies. In a nutshell, multiple ant agents move through a given graph with a given strategy, which can just be a random walk. Agents that reach the target mark their path with artificial pheromones, biasing other agents in following iterations to more likely follow certain edges. In the context of speedrun routing, a probabilistic approach, although not necessarily optimal, can potentially help speedrun routing by giving new inspiration and suggestions for potentially better routes. It could also make up for missing information in the game graph by letting the ant agents traverse unconfirmed edges with a given probability, allowing them to explore the search space more thoroughly, while accepting a degree of uncertainty. Routes found with methods like these would have to be checked manually by the speedrunning community, but can give new insights and ideas. There are extensions on ACO to enhance exploration of the search space [SH00], which seems to be a desirable feat in speedrun routing. An ACO algorithm with emotionally biased decision making has been shown to improve realism in virtual entities [Moc+10]. Biasing the pathfinding decisions e.g. to emphasize utility nodes that speedup the route as well as progress nodes

that are required for run progression could further enhance exploration and exploitation of promising solutions. As discussed in [GZN22], multiple metrics, such as speedrun trick difficulty, can be modeled as multiple dimensions in edge weights. With regard to multi-objective optimization, ACO has been extensively researched and successfully applied to this setting as well [Fal+22; LS12].

5.2 Algorithm Design

As can be seen by the previous section, the problem of speedrun routing poses some fairly unique challenges, most notably the traversal dependent edge dynamics.

The layer based idea introduced in the previous section is inspired by the approach of another project concerned with similar questions [Aar18]. This work uses a set-like collection of "keys" to model all progression locking game mechanics. For every subset of keys, a "plane of existence" is created, modeling the possible traversals with the given set of keys. This can be combined with the reward and requirement concept of yet another project [Mot20]. Rewards and requirements are similar to the concept of keys, rewards being granted upon visiting a node and requirements being assigned to edges as conditions. Only if a requirement has been received as a reward beforehand, an edge can be traversed.

Incorporating rewards and requirements from [Mot20], the established graph model can be extended to

$$G = (V, E, w, r_{req}, r_{rew}, \Sigma), \tag{3}$$

where V is the set of all nodes, E is the set of all edges, w is the weighting function. r_{req} and r_{rew} define functions similar to w, but mapping a set $R \subset \Sigma$ of requirements and rewards to nodes and edges, with $\Sigma = \{r_0, \dots, r_n\}$ being the alphabet of n possible requirements and rewards.

With this model, a routing algorithm working with priority queues needs to track the rewards that are collected in a path together with said path and check them against fulfillment of the edge and node requirements. When a new reward is collected, all previously computed paths containing or connected to edges that are impacted by the new reward would have to be marked as unvisited, significantly increasing computational load.

One way to facilitate this is to interpret the gathered rewards as different graph layers, similar to the approach in [Aar18]. Assuming one layer for each possible subset of Σ, this yields 2^n layers, which is unfavorable. To limit the potentially massive amount of identical edges created in different layers that way, it can be assumed that any edge on a layer with requirement set

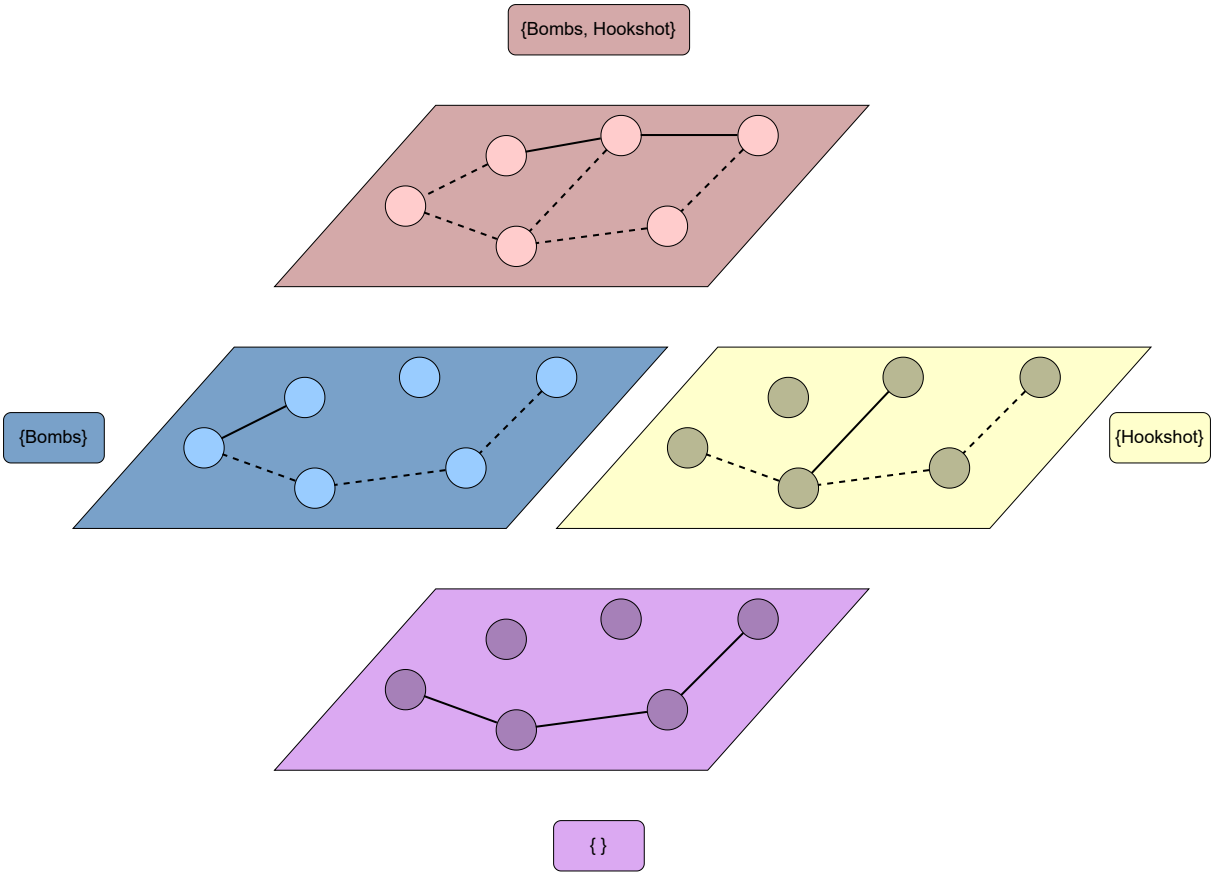

Figure 15: Visualization of a layered game graph model. Different colors depict different graph layers. Circles represent nodes and lines represent edges on the given layer. Boxes next to the layers denote the set of required items to traverse the edge. Dashed lines in a given layer represent implicit edges, derived from another layer with a subset of required items.

R also exists in any layer with requirement set $R' \supseteq R$. This reflects the fact that if a certain set of items is required to reach a specific point in the game, it doesn't matter if the player already acquired more than the required items. For a visualized example, refer to Figure 15. It depicts an exemplary graph with four layers. The set of required items of the graph layer on the bottom of the figure is the empty set, i.e. no items are required to traverse any of its three edges. The layer on the right side requires the *Hookshot* item in order to traverse the edge on this layer. The three edges from the bottom layer can still be traversed however, as the set {Hookshot} is a superset of the empty set and the requirements are therefore met. The edges are an implicit part of the layer. The same is true for the left layer with the *Bombs* item. On the left layer however, the edges of the right layer can not be traversed, as {Bombs} is not a superset of {Hookshot}. In the top layer, all edges from the other layers are implicitly included, as {Bombs, Hookshot} is a superset of all the other requirement sets. When collecting rewards during graph traversal, the current layer is changed.

6 Identified Challenges

During this first effort to create a game graph for *OoT* as exhaustive as possible, many challenges were identified. Some of these have been solved and presented in the previous sections. This section takes a look at the challenges that have been overcome as well as those yet to be overcome in the future. Where possible, the findings are extended to contexts outside the working example of *OoT*.

6.1 Defining the Nodes

One of the main challenges of a game graph identified by [GZN22] is the following:

> *Defining the Nodes*: "Events relevant to the game's progression" is not defined and it is nontrivial to do so. [GZN22]

This work showed that in the specific case of *OoT*, using actors, spawns and transition actors, every interactable element in the entire game can be modeled as a graph node. Although this is partly due to the extensive data sources available, it can be assumed that identifying the nodes of a game graph generally does note pose a substantial problem. Some games feature in-game overviews or maps that can be utilized, which is of particular help for modeling 2D games. Furthermore, many speedrunning communities compose such overviews themselves to facilitate manual routing. In a similar manner, modeling data for nodes can be compiled.

In the case of *OoT*, there are actors present in the game data, that do not have any importance for speedrunning, like the already mentioned purely aesthetic actors, e.g. causing sound or visual effects. Purging these can be facilitated by using the *actor ID*, which denotes the type of actor and is part of the VerboseOcarina outputs and therefore part of the graph nodes. Filtering these — or the data before constructing the graph — by specific IDs will reduce the graph size. Other uses for the actor ID or actor parameters, which are also included in the nodes, are possible, but have not yet been investigated.

Generally, the nodes of a game graph do not need to be minimized to only the relevant ones. In fact it can be quite challenging to do so, as nodes can have very subtle reasons to have an impact on a given route. But while surplus nodes are not inherently detrimental to the routing process, as identified in the example at hand, purging a graph of these nodes reduces the problem's complexity. A more challenging task is to connect modeled nodes with edges.

6.2 Defining the Edges

As is stated in [GZN22], another main modeling challenge is the definition of a graph's edges.

> *Defining the Edges*: The extent of "each possible traversal between events" has underlying restrictions that have to be defined as well. [GZN22]

This challenge was minimized in this work by selecting the *OoT* Glitchless category [OoT-Glitchless]. This category forbids the use of any glitches as defined in its ruleset. This restriction critically simplifies the rules by which the nodes are connected with edges. In a Glitchless setting, the intended game cohesion can be utilized.

In the case of *OoT* this results in the applied aggregation of actors and spawns inside rooms to form complete sub-graphs, further composed to scenes by connecting the rooms using transition actors. The connections of scenes however, having some dynamics to them, can not be easily derived from the game data and have to be set manually, unless other ways are found.

Applied to other contexts, the difficulty of defining edges between nodes of a game graph is difficult to assess, as different games and categories can have very different progression mechanics. Glitched *OoT* speedruns for example can, depending on the category, be very convoluted, following obscure rules in game world traversal and introduce only partially understood dynamics. It can be said that the more "*game breaking glitches*" — as the more impactful and powerful glitches are often called — are known in a given game, the harder the definition of edges for that game gets. Accordingly, the more restrictions are imposed on a category through its ruleset, the easier it is to map the rules of game traversal to edges in a game graph.

Generally speaking, when modeling a game as a graph, the structures of game world cohesiveness are good indications on how to aggregate and connect the nodes and to layout the general structure of the graph. The game world of the Super NES title *Super Mario World* for example comprises an overworld that is composed of many levels. The overworld can be modeled as a graph, with the levels as nodes. Edges would intuitively be placed representing the paths connecting the levels with each other. The nodes then can refer to individual sub-graphs for every level in the overworld. This way, the overworld represents the outer layer of cohesion, holding together the individual levels' graphs. Details of the graph layout have to be considered and designed individually for every game, but the overall game cohesiveness must be respected. This of course includes the softening of said cohesiveness through glitches and techniques allowed by the category's ruleset. If critical elements of cohesion are missing, like the scene transitions in *OoT*, other ways must be found to meaningfully construct a graph.

6.3 Edge Weights

One of the most obvious and problematic challenges left unaccounted for is the assignment of weights to edges. To specify the meaning of this, the weight of the edge (v_a, v_b) is considered the least amount of time possible to reach the game element represented by node v_b from the game element represented by node v_a. The least possible time is taken because this is the time a speedrun is aiming for. This approach abstracts from the operational and tactical decisions and execution insofar as optimal execution is assumed. If there are different techniques that can be used and that pose different challenges to the runner, multiple edges can be introduced, possibly with a difficulty rating assigned to them. Alternatively, the average value of the top timings from multiple runners, or average times in general can be imagined to be a helpful approximation as well.

By extension of edge weights, nodes can also be considered to have weights assigned to them, e.g. the duration of a cutscene or of an NPC's monologue. In some cases this will be 0, as the collection of a pickup or similar does not take any time other than reaching its location, but even opening a chest takes a couple of seconds. This time can be modeled as node weights.

In the presented example of *OoT*, there are geometrical coordinates incorporated into the nodes. These however are not fit for calculations or even assumptions of edge weights. Even if two nodes are very close to each other, the player can be blocked off by terrain, a gate or just an enemy that has to be defeated before proceeding. When considering more than one scene, the difference in coordinate spaces turns any calculation into a mere approximation, and as soon as warps are involved, distance only has little meaning, if any at all.

In context of other games, spatial conditions might be a good indicator for edge weights. Especially games in which movement is of major importance to the general progress can benefit from such a metric. Simple platform titles like the *Super Mario* games have been approached like this with great success [TKB10]. More thoughts on this are presented in section *7 Summary and Outlook*.

That being said, figuring out edge weights in somewhat complex settings is a major open issue. But without edge weights, a game graph simply denotes the general cohesiveness and structure of the game world, which a speedrunner is already very familiar with. This alone does not present any advantage. It does however greatly facilitate such ventures in the future. Having a sophisticated model at hand can be an inspiration or even a basis for more modeling and routing algorithms developed in the future. Suggestions on how this could be approached are given in section *7 Summary and Outlook*.

6.4 Dynamic Edges

Another, very important aspect stated by [GZN22] that is not yet implemented in the model is the dynamic nature of edges.

> *Dynamic Weights*: Edge weights are not consistent but rather change with graph traversal. [GZN22]

Edges change in weight, become available for traversal or even cease to be available. This dynamic has to be accounted for in order to reflect the dynamics present in most speedruns. Another project that is also concerned with an algorithmic approach to speedrun routing [Mot20] introduces *rewards* and *requirements*. This has also been picked up and conceptually extended by [GZN22] as well as in section *5 Solving the Routing Problem* of this work. In a nutshell, rewards are granted upon the player when visiting a node or traversing an edge, representing items being acquired or other progress being made. Respectively, edges and nodes can also present requirements that need to be acquired prior to traversal or visit.

Using this approach, modifiable world state, like Link's age in *OoT*, can be modeled as two rewards, starting out the game with a `child` reward, then when becoming adult losing this reward and gaining an `adult` reward instead. With these concepts in place, a potential algorithm will have to track the rewards collected during traversal and check for requirements before any edge traversal. A layered graph model as presented in section *5 Solving the Routing Problem* can help to circumvent this problem by encoding the inventory state in the graph topology.

6.5 Repeatable Events

Another challenge a graph model introduces, as identified by [GZN22], are repeatable events:

> *Repeatable Events*: A subset of the events can be repeated once triggered, while others can not, further increasing the complexity. [GZN22]

To address this issue in a game graph, both of these cases need to be accommodated.

By default, all nodes represent repeatable events, as all nodes can technically be visited an arbitrary amount of times. Special cases are nodes that yield rewards upon visit. Repeatable events can be implemented into a graph by granting their reward every time a specific node is

visited or a specific edge is traversed. If an event is repeatable immediately multiple times in succession, a loop, i.e. an edge with the respective node as the starting and ending node, can be placed. This way, the loop can be traversed as often as needed. A practical example for this is repeatedly playing a minigame or performing a glitch to accumulate a large amount of money in a game. This process is often called *farming* and encountered frequently, even outside of speedrunning contexts. Sometimes, a node must be "rearmed" before being able to collect a reward again, for example if an area has to be reloaded to regrow bushes or similar. In simple cases, the incoming edges of the node in question can be provided with the respective reward. This however is only possible if all adjacent nodes can rearm the reward. In more complex settings, like in *OoT*, another mechanic has to be implemented in order to detect when an event can yield its reward again. One possible way is to utilize the layered graph concept as described in section *5 Solving the Routing Problem* and change to the corresponding layer once the conditions for rearming the node are fulfilled.

Non-repeatable events can change the topology of a graph during traversal, just as dynamic edges do. Most progression items can only be acquired once for example. This case can also be addressed by a layered graph layout. After visiting a non-repeatable event, the layer can be changed to one that does not include the corresponding node. Another option is to remove all incoming edges once the node has been visited. A third option is to leave the node as-is and remove the reward, so the node can still be visited during the remaining routing process.

As with many modeling challenges, how to deal with repeatable and non-repeatable events in a graph depends heavily on the game and category in question. One possible approach for *OoT* — and many games that behave in a similar way in this regard — can be to rearm specific types of rewards after leaving a room or a scene. This can be a valid method regarding item drops from enemies, bushes, rocks etc.

6.6 Multiobjective Optimization

The last major challenge of the game graph model listed by [GZN22] is referencing other objectives than completion time:

> *Multiobjective Optimization*: The model does not account for any dimension other than time in a possible route. [GZN22]

The prime directive of a speedrun is speed. However, this issue refers to other objectives a speedrunner might aim for. These other objectives might manifest as a certain difficulty a

runner's skill does not allow them to exceed, or a certain degree of consistency in an otherwise very volatile run. As this issue is a more future directed consideration, it will be covered in more detail in the upcoming *7 Summary and Outlook* section.

6.7 Other Versions and Categories

One other obvious disadvantage of the presented modeling is that it only is faithful to Glitchless speedruns of the NTSC 1.0 version of *OoT*. Categories allowing glitches will most certainly yield differing models. In *OoT*, most types of edges will still prevail, as other categories usually have less restrictions imposed on them. Wrong Warps for example exploit the design and mechanics of the way Blue Warps are implemented in such a way that they can lead to entirely different scenes. In other instances, a player might bypass a transition actor in order to prevent a room from loading. As this includes all of the room's actors, potential doors or gates in this room will no longer exist and hence not block the player. In such a case, the assumption that a player must traverse from room to room by transition actors is no longer valid, yielding a different graph model.

The nodes of the graph however can be assumed to stay the same, as the underlying data is not changed. This also means that for most games, different categories only manifest in differing graph edges, ending requirements as well as starting and ending states of the run, while the nodes are kept.

This again is only valid if the same game version is used. As discussed in section *4.3 Data collection*, different versions of the game might be slightly different in their data. In the case of "*Master Quest*", an *OoT* version distributed in a special collector's edition for Nintendo's *Game Cube* console, the data is very different, as all of the dungeons have been reworked to pose a greater challenge. This is a problem, as many of the manual adjustments to the *OoT* graph — like the Blue Warp transitions or the target nodes for Save Warps — use hard coded positions in the actor lists and thus are not robust to changes in the data. If programmatic solutions for these hard coded adjustments can be found, this would facilitate the applicability of the presented procedure to other game versions immensely.

All of these assumptions are void if *Arbitrary Code Execution* (*ACE*) is considered, a very powerful technique possible in many games. As the name suggests, ACE is capable of executing arbitrary code inside a game engine. However, because of this very power, ACE is banned in all categories except for *Any%*, which per definition has no restrictions.

6.8 Game State

In [GZN22] an alternative modeling approach is suggested, modeling states of the game, rather than events or points in the game, as nodes. Though this has not been covered here, some aspects of game state are still modeled into the graph. Most *OoT* scenes only have one setup and thus only one state. Scenes with more than one setup automatically represent another game state in each setup. In most cases, this is the time of day (day vs. night time) and Link's age (child Link's world vs. adult Link's world). In some cases there is some other state encoded in scene setups. Scene 85 (Kokiri Forest) for example has one setup for child Link (setup 0) and two for adult Link (setups 2 and 3). This resembles the "curse" that lies on the forest and the village within. Before lifting the curse (scene 2) the village is overrun by monsters and there are almost no Kokiri (the childlike inhabitants of Kokiri Village) outside. After cleansing the Forest Temple and beating its Boss, the curse vanishes together with the enemies and the Kokiri roam their village freely again. Every node of these setups contains this information — Is Link child or adult? Has the Forest Temple been beaten yet? — simply by belonging to a specific setup. Nevertheless, all three scenes must be connected to the sole scene setup of, e.g., the Deku Tree, being the very first dungeon and directly connected to Kokiri Forest, just like any of the NPCs' houses. This in itself does not present any challenges, but when it comes to algorithmic concerns, there has to be some logic in place to prevent a traversal — even a transitive one — from a child Link setup to an adult Link setup. The only place where such a traversal is intended in the game is the Temple of Time. Here, Link can interact with the Master Sword's pedestal and either place or remove the sword, switching timelines upon doing so. Considering a layered graph with rewards and requirements as lined out in section *5 Solving the Routing Problem*, at this point an *adult* reward can be granted an a *child* reward can be withdrawn. This way the available edges in the resulting graph layer reflects the circumstances of the corresponding timeline.

Similar reasoning is valid for the time of day, which is partly an even more challenging case. Time of day behaves somewhat inconsistently in *OoT*. In some places, like most of the dungeons, cities and villages, time of day does not pass. In other places, like most of the overworld, it does. Scenes without passing time represent day vs. night, if there is any difference at all, with different scene setups, which can be modeled with the procedure presented in this work. Some scenes however represent the time of day by actor *behaviour*. In scene 84 (Zora's River) for example there are four Gold Skulltulas, an enemy granting the collectible item "*Gold Skulltula Token*". Even though these items are not of concern in the Any% Glitchless category, they present a suitable sample case for similar considerations. Two of those Gold Skulltulas only appear at night as adult Link, but they are still part of the same scene setup (setup 2) as the rest of the actors. In this case, the scene setup does not reflect the time of day in its data, unlike the

aforementioned scene setups of most villages. This inconsistency can lead to difficulties.

A trivial solution would be to discover all of these actors that are only active at nighttime and manually assign them to a separate additional scene setup. There may even be a way to distinguish these actors by analyzing the data even further, or their behavior in the code. This still leaves the challenge to determine when to change between these setups, as nightfall could happen anytime while roaming these scenes. One could track the exact time of day passed by keeping track of the actual time passed, which would be available by the cumulative weight of edges traversed in a scene. Some actions however prevent time from passing, like talking to NPCs, — which can also be part of routing decisions by intentionally dodging or triggering NPC encounters — rendering this approach potentially error prone and in need of testing.

Another problematic note on setups is their inconsistent indexing. As already mentioned, most of the time Link's childhood timeline is represented by setup 0 by day, setup 1 at night if applicable, and Link's adulthood timeline is represented by setup 2 by day, setup 3 at night if applicable. There are exceptions, like the case of Kokiri Forest covered in the previous paragraph. Looking at Table 6, there are only 127 setups for the 99 scenes, with the majority of 2-setup-scenes being the default child and adult setups. Therefore, manually finding and handling these exceptions might be within reasonable scope.

Table 6: Distribution of the 127 scene setups among the 99 scenes.

# of scenes with ...	... # of setups
81	1
11	2
4	3
3	4

Transferred to other contexts, dynamically changing game and world state can be a major issue. Many games, especially RPGs, employ day-night-cycles, parallel worlds, weather effects or other means of dynamic world behavior. Some games provide multiple characters to explore the game world with, each providing individual means of doing so. These circumstances hold the potential to introduce very complex rules that need to be mapped to a game graph. A layered graph model can help in circumventing some of these, by providing layers for specific game states. This would however further increase the complexity of a given graph.

6.9 Even more Warps

While section *4.5 A Sample Game Graph* deals with the different types of warps in *OoT* quite thoroughly, some warps remain undealt with.

6.9.1 Wrong Warps

Wrong Warps have been mentioned briefly before, and though they are irrelevant for glitchless categories, they pose an interesting case. Wrong Warping has been a cornerstone of *OoT* speedrun optimization since it was found. It is so important and quite complex in fact, that there are multiple entries on the ZeldaSpeedRuns Wiki [ZSR-WW; ZSR-WWExp; ZSR-WWTable] and a dedicated calculator for different outcomes of the technique [mzx-WWCalc]. Explaining the exact functionality is outside the scope of this work, but in short, Wrong Warping lets the player enter a warp — a Blue Warp in many cases — and after performing a series of very specific manipulations, Link will spawn in an unintended place, like another cutscene than intended, an entirely different scene, or even both.

Here lies one of the greatest strengths of Wrong Warping: It is entirely possible, as is performed frequently by many runners, to Wrong Warp from the Fire Temple Blue Warp directly into the game's ending cutscene, showing the development credits and thus prematurely finishing the game and reaching the ending condition for many speedrun categories. A Wrong Warp from the Deku Tree, which is the very first dungeon in the game, to Ganon's Tower Collapse, the last section of the game, is possible as well. Other uses for Wrong Warps are to simply skip lengthy cutscenes or even to prepare for other glitches. This versatility, as well as the fact that "*the game ends up in an interesting state*" [ZSR-WW] after performing Wrong Warps likely makes them very hard to incorporate into a model such as the presented one in any meaningful way. That being said, no testing or further consideration has been conducted to verify this assumption.

6.9.2 Void Warps

One more kind of warp is the Void Warp. While this term also encompasses a series of in part very sophisticated and complex glitches, this discussion will be limited to the process of voiding out, i.e. leaving the bounds of the game or falling into a pit, and consequently resetting Link's position. The result of this position reset depends on multiple dynamic factors.

Some overworld areas reset Link to the last spawn by which he entered the scene, just like a Death Warp would. If available, this is usually faster than a Death Warp, as no death animation is played and the game over screen is not shown. Either way, an edge to represent this kind of Void Warp is already implemented through Death Warp edges. In other locations, Link's position is reset to a checkpoint, e.g. where he entered a room. As this is usually done through a transition actor or a spawn, these edges also already exist in a graph as described in this work.

There are special kinds of pits, setting Link's position in yet another way. However, in a Glitchless setting, most of these resets do not move Link outside of the current room. In such cases, the already existing edges can be used to describe a Void Warp.

It can even be argued that these movements are more on a tactical than a strategic level and should not be included in the model, but rather only be reflected in the timing of an edge and possibly a note that a timing is based on a Void Warp.

6.9.3 Farore's Wind

An intended but also much more dynamic warp is the magic spell *Farore's Wind*. Once received from one of the great fairies in the game, Farore's Wind can be used in any dungeon to place a warp point at the last checkpoint, similar to where a void warp would place Link. When used again anywhere in the dungeon or even in another dungeon, the player can return Link to this warp point. This obviously creates edges between arbitrary nodes of all the dungeons, which can only be traversed under very specific circumstances. Therefore adding all possible combinations as individual edges is unfeasible and this mechanic still needs a good modeling solution.

When looking at Glitched speedruns, advanced techniques make use of this dynamic to transport Link to unintended coordinates in the target scene. Through a series of complex memory manipulations the return point of Farore's Wind can be set to over 60,000 different locations, reading unintended data as spawn information and warping Link almost arbitrarily. Obviously, this would pose quite a challenge for speedrun modeling.

6.9.4 Sun's Song

Another warp-like technique worth mentioning is the effect of *Sun's Song*. While Sun's Song turns night into day and vice versa, this happens in different fashions, depending on the current scene.

In scenes that have different setups for nighttime and daytime and where the time of day does not normally pass, playing Sun's Song reloads the scene with the corresponding new setup, with the time of day changed to noon or midnight, respectively. Link will then reappear in the spawn by which he last entered the scene.

In certain scenes that do not differentiate between day and night, a very similar behavior can be observed, but the setup stays the same after reloading the scene, behaving similar to a Death Warp.

Other scenes without day / night differences do not react to Sun's Song at all, e.g. dungeons. A possible reason for this is a certain kind of enemy in the game, the undead *Gibdos* and *ReDeads*, which can be temporarily paralyzed by playing Sun's Song in their proximity. Supposedly, scene reload is suppressed in order to make use of this other effect of Sun's Song (see Figure 22 in Appendix *Excerpt of* OoT *Speedrunning Discord conversations*).

A fourth variation is observable in those scenes in which time of day naturally passes continuously. Here, a time-lapse effect is triggered, causing a very quick transition to either daybreak or nightfall. Then, time continues to pass normally.

Finally, a last variation are scenes in which the use of the Ocarina item is restricted, consequently making it impossible to play Sun's Song. The `Item Usability by scene` sheet of the *OoT SRM / GIM Tables* [SRMTables] lists all those restricted scenes in column `sunsSong` (value `1`), as well as some of the scenes that do not reload or react in any way (value `3`). The dungeons however, despite also not reacting to Sun's Song, do not find a mention here. The other variations are not documented at all.

This leaves the case of Sun's Song hard to model. The reloading scenes could be modeled in a similar fashion to Death Warps. The other variations however pose a challenge of their own and — given there is no other source that can be used for this purpose or even a way to programmatically extract this information from game data or code — still require some experimentation and documentation.

6.9.5 Cutscenes

A last warp-like game mechanic are cutscenes. Cutscenes can cover a wide range of uses, from informing the player about the plot — which is of little interest to speedrunners — to granting items, transport Link or progress the game. Cutscenes are often triggered by entering a specific area in a scene while fulfilling its trigger conditions. These conditions also are very versatile, ranging from items in possession to general game progression or no condition at all. Additionally, cutscenes can change time of day, or change the timeline, in the case of the Master Sword cutscene. This variety as well as the dynamic nature of cutscenes pose a modeling challenge. Some of the more speedrun impacting cutscenes can be modeled by manually adding edges and potential rewards for the respective cutscene. The number of these cutscenes can be supposed to range somewhere in the double digits. Other cutscenes with a more complex behavior have to be considered more thoroughly.

6.9.6 Warps in other Games

Outside *OoT*, some of these warps are present and frequently used in other games' speedruns, as well as Save and Death Warps as described in section *4.1 About* OoT. Notably, cutscenes repositioning the player or instant traveling options are well-established concepts in many games. Fast traveling through a map e.g. is part of many open-world-games. If the target points for such an option are fixed and few, a procedure similar to Save and Death Warps can be employed. The more freedom the player has in instant transportation, the harder this transportation is to express in game graph edges.

As with many aspects of speedrunning mentioned so far, the more dynamic a given movement on the game graph is, and the more complex and rule-based it is, the harder it is to include in a model. All warps, intended or not, seek to circumvent the cohesiveness of a game world by breaking or ignoring the rules imposed on its traversal. This deviation from otherwise consistent rules is just as detrimental to the modeling process as complex dynamics are.

6.10 Scene Transitions

A challenge that has been mentioned multiple times by now is the connection of scenes by transition edges. More specific, the selection of nodes to connect with an edge, to represent the game mechanic of leaving a scene through a specific exit and spawning Link in the corresponding spawn position in another scene. This seemingly very basic and static game feature has some dynamic to it. As explained earlier, the game scenes have different setups attached to them, representing different states. When traveling from one scene with only one setup to a scene with multiple setups, the right setup has to be selected for loading so that it reflects the current game state. It can be assumed that this is also the reason for the four entries per spawn in the exit tables. Like other dynamics that are reflected by code behavior rather than data, this has not been part of this work and therefore still might cause difficulties. However, being a critical part of the game world cohesion, it needs a workaround before the model can serve any meaningful purpose.

One workaround that has is part been applied during the presented procedure is to manually identify the transitions and to add the respective edges into the graph. This is tedious and error prone. The process can be facilitated by clustering and shortcutting nodes. As an example, some of the NPC houses only house NPCs that tell tales or give hints and story information. This does not constitute any contribution to any speedruns and is therefore irrelevant to the model. These scenes can be condensed to single nodes, or even eliminated from the graph entirely. Other scenes, like the last part of the last dungeon in the game, which presents itself very linearly progression-wise, can also be condensed to few or single nodes. Even speedrun relevant scenes that are very small could be condensed to single nodes. Also, preprocessing like this is a well established method of reducing graph complexity for faster processing and shortest-path computation [WW07; Bau+10] and similar techniques can be suitable here. Section *5 Solving the Routing Problem* covered this approach in some depth.

Regarding programmatical support in finding scene transitions, one source to facilitate this task can be *The Ultimate OoT Spreadsheet* [UltOoTSheet] with its sheet `Spawns`, an excerpt of which is displayed in Figure 16. Column `C` shows the spawn index of every spawn in a scene denoted by column `B` (note the non-indicated hexadecimal format here), with descriptions of the spawn scene in column `H` and most importantly the *exited scene* in column `I`. This at least gives a hint towards which spawn of a scene to use when coming from a specific other scene, but the actual exit position or other information to identify a corresponding exit node of that other scene is not documented. This is especially unfortunate in cases, where there are multiple connections between two specific scenes. If a way can be found to programmatically assign scene transition edges, most of these issues can be resolved.

A	B	C	D	E	G	H	I	J
Ent.	Map	Ent.	Var.	Trn.	Index	Spawn At	Having Exited From	Used
0000	00	00	41	02	Gameplay	Inside the Deku Tree	Kokiri Forest	✓
0001	00	00	41	02				
0002	00	00	41	02				
0003	00	00	41	02				
0004	01	00	41	02	Gameplay	Dodongo's Cavern	Death Mountain Trail	✓
0005	01	00	41	02				

Figure 16: Excerpt of the `Spawns` sheet in *The Ultimate OoT Spreadsheet* [UltOoTSheet], showing relevant columns for entrance identification.

6.11 Level of Detail

One more potentially problematic note regards the level of detail of the model. In the case of *OoT*, some actors, when modeled as single nodes, present a very high detail level. As an example, there is a section in many runs that involves Link being asked to return seven *Cuccos* — chicken-like animals — to their pen. Link can only carry one Cucco at a time, but he can manipulate the behavior of roaming Cuccos in the speedrunner's favor. This is usually done in the same way every single time as this method provides a fast and consistent way to complete the task. The graph model has no concept of quests or tasks. It is also not possible to simulate the Cucco manipulation. The timings between collecting the individual Cuccos can be added to their respective edges, but that seems tedious and redundant if the time of the entire process is known. As this task has to be completed in one go anyway, it can be modeled as a single node, or multiple nodes for different strategies, with their timing encoded in either the incoming or outgoing edges. If node weights are considered, this would be the intuitive choice.

Situations like this present an interesting modeling decision space. Providing a timing for connected activities prevents a potential optimization algorithm from finding better solutions for tasks like these. This reduction however has the advantage of moving the model closer to a strategic level. One could argue that the collection order of individual Cuccos, Rupees or similar seems a rather tactical or even operational consideration. Decisions on specific levels of detail in a game graph need close attention in order to keep the model on an appropriate routing level.

6.12 Data Sources

A last challenge to be pointed out is the fact that the data sources concerned for this work have been of very different natures. Much information has been accumulated by the community to facilitate learning speedrunning techniques, routing and understanding the game more thoroughly. But as there have not yet been any somewhat broadly used systems for algorithmic

routing, there are no data sources designed for this intent. Some sources could be translated to the routing modeling problem very well, like the VerboseOcarina [mzx19] outputs. Others, like the OoT Interactive Map [OoTmap.com] or SceneNavi [Dan14] are very insightful, but mainly designed for human consumption and manipulation and therefore hard to use as data basis. That being said, all sources have helped greatly in the presented work and without the enormous community effort to extract each and every possible information from the game, this endeavor would not have been possible.

There are many very sophisticated speedrun communities, and while it may be hard to find a game that is quite as well documented, understood and generally explored as *OoT*, many of these communities provide a multitude of data and advice for runners of all skill levels. As was shown in the work at hand, data does not have to be specifically crafted to facilitate modeling a game graph. Carefully examining all available data and assessing its modeling capabilities can yield unexpectedly positive results.

7 Summary and Outlook

During the course of the presented work, the art of speedrunning was briefly introduced together with relevant terms. Other approaches to algorithmic speedrun routing were discussed and assessed. The video game title *The Legend of Zelda: Ocarina of Time* was introduced and speedrun routing relevant mechanics explained. With this game as a working example, the findings of [GZN22] were applied to formulate a graph model. After collecting all necessary data and conducting a size estimation for the resulting graph, a procedure to construct a game graph was developed and applied to said data, yielding a graph covering as much of the entire game as possible. Then, the actual speedrun routing was examined and identified as a shortest path problem on the constructed game graph. Different shortest path algorithms were discussed and assessed for their applicability on the routing problem. Thoughts on the adaptation of both the model and an algorithm to facilitate formulating a routing algorithm were presented. Some aspects however are not yet faithfully represented in the graph. Being the main outcome of this work, reasons for the encountered modeling flaws and other challenges were presented and possible solution approaches described where possible while transferring findings to contexts outside the working example of *OoT*.

Summarizing the findings, it can be said that the formulation of a game graph with a method like the one presented here is a highly game-specific procedure. The amount of data available for the working example of *OoT* is not to be considered representative for other games. Even with the extensive data sources at hand, some aspects that are important for speedrun routing could not be incorporated into the graph. Furthermore, some manual customizations and hard coded exceptions had to be made in order to construct a meaningful graph.

That being said, *OoT* is a game with fairly complex dynamics, and being able to construct a graph — even with the presented flaws — is not a trivial task. Using the game actors and the presented procedure to model a graph that comprises as much speedrunning relevant content of the game as possible works well for the presented case of Glitchless *OoT* speedruns. Many of the encountered challenges hold some relevance with regard to other contexts and games.

Future ventures to elaborate on these matters can draw on the presented findings, yet regarding to *OoT* it is recommended to focus on the more impactful flaws of the model first. With a deeper understanding of the scene transition dynamics for example it can be assumed that a programmatic discovery of these transitions is very much possible, increasing the practical viability and faithfulness of the model significantly and reducing error proneness. This assumption stands equally for other manual customizations made, such as warps, and missing representations, such as time of day dynamics. How well such a model fares when being

subjected to the corresponding routing problem is yet to be found, as well as the respective algorithms.

Looking beyond the scope of *OoT*, the viability of this procedure is hard to assess. There exist speedruns for almost any kind of game and as different as these games are, as different are the speedruns. *The Legend of Zelda: Majora's Mask* (*MM*) for example, being *OoT*'s direct successor, could be expected to yield very similar results. The game was also developed by Nintendo for the game console Nintendo 64, just as *OoT*, only two years later. It even is based on the same engine, so it is no surprise that there is almost as much data available for *MM* as there is for *OoT*. Amongst them there is *The Ultimate MM Spreadsheet* [UltMMSheet], with a very similar structure to that of the *OoT* spreadsheet, containing all actors, transition actors and spawns in the game. There is a warping system in place similar to Song Warps and the Save Warp mechanic is much simpler. The progression mechanics also revolve around acquiring items to traverse the game world. Other than in *OoT* however, a lot of the game mechanics in *MM* also revolve around a three-day-cycle that is repeated over and over again. The game world heavily changes during the course of these three days and even the time of day to the hour is important for some events. This introduces a much more complex dynamic that would have to be mapped in a model. Seeing how dynamics like this are one of the main weaknesses of the presented model, the assumption of similar results is at least questionable.

On the other hand, there are games without or with very few of such dynamics. At the time of writing, one of the most popular *speedgames* — as games are often called that have good speedrunning potential — is *Celeste* with 172 active players ranking in 7th place in terms of activeness on [Speedrun.com]. Figure 17 shows a screenshot of the game. Celeste is a 2D platform game with very limited but fast paced movement options. The player can run, jump, climb and dash. Later on, an extension for the dash ability is unlocked. Some of the elements in the different levels are designed to provide more interesting ways to design mazes, puzzles and other challenges, but they all are interacted with through the use of these four actions. For the Any% speedrun, there is no progression mechanic in place other than moving through the levels to reach the end, occasionally with the need to collect a key prior to passing a locked door. This poses much less complex behavior and dynamic elements than *OoT* or *MM*.

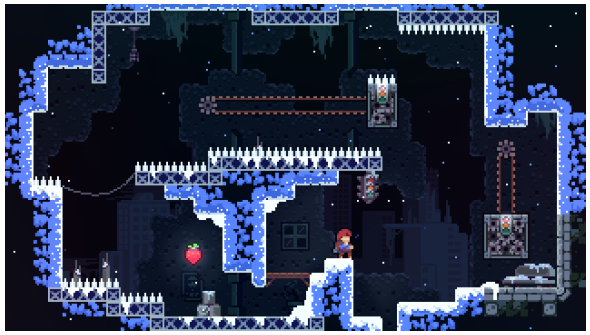

Figure 17: A screenshot of Celeste.

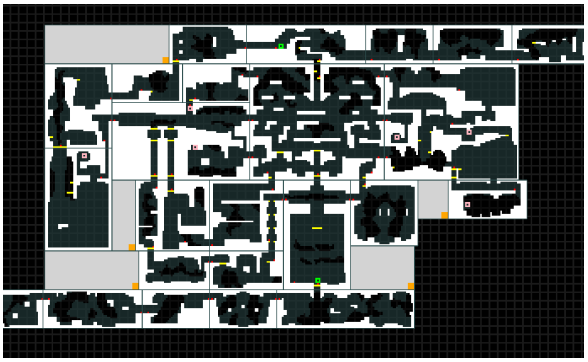

Figure 18: A screenshot of the Celeste debug map.

As for data collection, the 2D nature of Celeste can be utilized. In the debug mode of the game, there are map overviews available for each level of the entire game (see figure 18). Potentially after some preprocessing, Visual Computing techniques like line detection [Hou62] template matching [Bru09; Umb17] or simple color mappings can be used to derive a graph model from these maps. Due to Celeste's popularity and its speedrunning suitability, there also exist many modding projects. For example, [Ahorn22] is an open source map editor for Celeste, so it can be assumed that the map data can be extracted directly from the game's files. In the case of simple maps like the first level of the game, routing might be trivial. The fourth area of the fifth level of the game on the other hand has a more open layout, with two keys that need to be collected in order to activate a contraption at the center of the level that allows further progression. Maps like this might profit from a speedrun routing algorithm in order to find the fastest sequence of events. Another category "*All Red Berries*" requires the player to collect all 175 red berries in the game — a collectible without any further effect — and complete the game afterwards. The order of collection is irrelevant and therefore another point of possible optimization, as these berries are spread across the levels and the player often has to go out of their way to collect them. And even for the more trivial examples, an algorithmic process to assess manual routes can be beneficial.

This example demonstrates the differences that occur between games of different design. Obviously, data collection will be different with games being different. If the game is well documented and understood, this process will be much easier. A game's complexity is very important for data collection as well. The nature of the data available will also dictate the possibilities for automated model generation. But while the methods of data collection might differ substantially, a model as described in this work can be imagined to help routing efforts for many games.

Regarding data collection, another approach can be to reach out to game developers to provide useful data. Some game developers, especially small and independent studios, actively collaborate with speedrunners to meet their special needs. A lot of modern games contain speedrun features like accurate timers or even dedicated speedrun modes that remove cutscenes or other downtimes during gameplay and provide other quality of life improvements for speedrunning (e.g. *Celeste*, *Loop Hero*, *A Hat in Time*, *Fury*, *Dragon's Dogma* etc.). If algorithmic routing becomes a viable method in the future, developers caring for their speedrun communities might be inclined to provide dedicated data and tools to facilitate speedrun modeling.

Another important possible track of continuation is the gathering of meaningful model data, mostly in form of edge weights. One common argument of rejection against algorithmic speedrun routing is the need for extensive timing data. This has also been pointed out in this work. One approach to facilitate this task can be data crowdsourcing. If a public platform is provided to easily contribute data to a routing model, this can greatly help with data collection. The presented work can be used as a basis for such a platform. This way, runners would have reference points to which they can easily assign information like timings, requirements and rewards. See figure 19 for an exemplary website wireframe. A central data collection like this can also help to improve the general documentation of speedrunning related data, possibly also supporting tasks different from routing. This would also help in making the modeling process more accessible to those without computer science or graph theory knowledge, provided the user interface of the platform is designed appropriately.

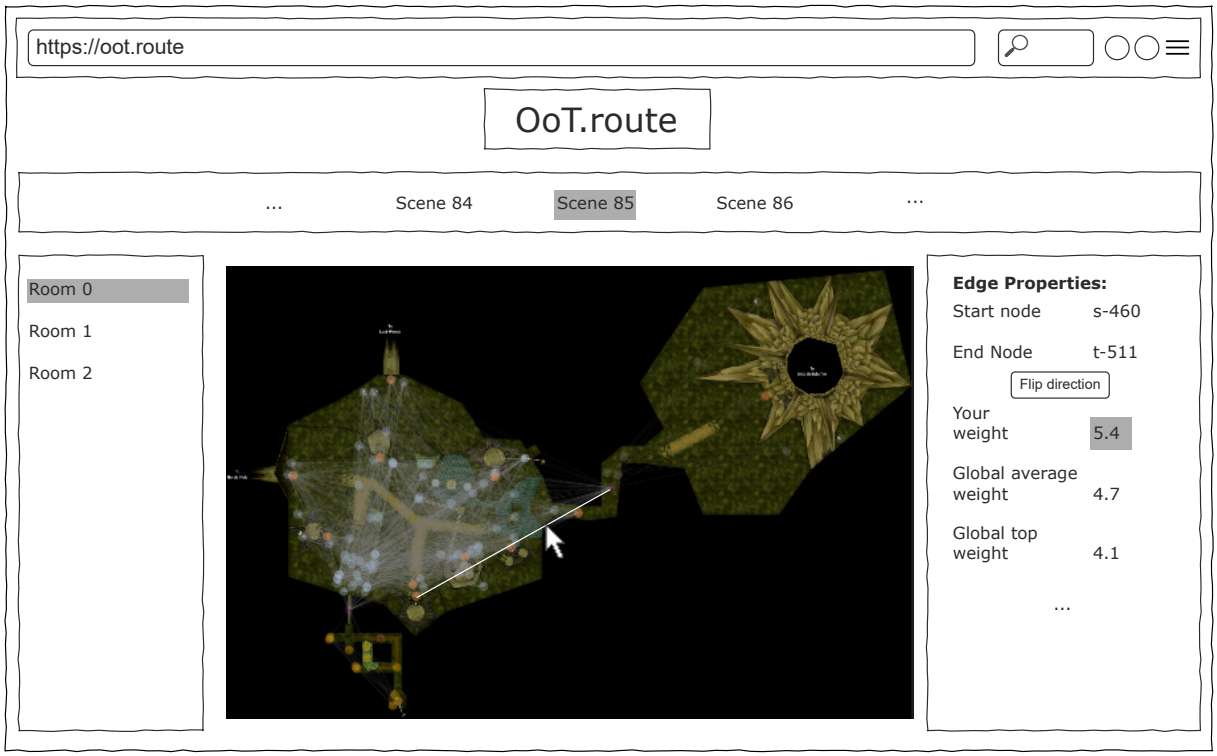

Figure 19: Wireframe of a possible data crowdsourcing website for edge weights. Through scene (top) and room (left) selectors, a scene graph can be displayed (center). By selecting an edge, its data (right) can be viewed and edited. Using an account system, data of individual users can be tracked and global average or top values etc. can be used as feedback for validation, or to assess one's performances in competition with others.

A more future directed approach can be the use of AI. A simple first idea might be to train an AI agent to speedrun *OoT* from scratch, as has already been done outside of speedrun contexts [TKB10; LC17; Com18; Vin+19; Ye+20]. The community driven reverse-engineering project [ZeldaRET] might be utilized by supplying the AI with the entire simulation engine of the game — provided that the decompilation of *OoT* is in a progression state to be able to do this.

Given that an open source PC port of *OoT* has been recently released in an alpha version [Har22], this is very likely the case.

An approach like this has been followed successfully by the winner of the *2009 Mario AI competition* [TKB10]. Although this was a 2D platform title and therefore had much fewer dimensions of freedom just as Celeste, it can be assumed with a degree of certainty that access to inner game engine information will have an impact on such an AI approach. The more recent work of Lample and Chaplot [LC17] uses partial internal game engine information to substantially increase the performance of a Deep Reinforcement Learning AI agent in a deathmatch scenario of *Doom*.

More specifically targeting speedrun routing, one could imagine a better informed AI agent — for example supplied with knowledge about how to efficiently move, interact with the game mechanics or even perform speed techniques — to be trained to traverse specific edges of a game graph as fast as possible. Such an AI agent could then be used to automatically time edge weights en masse. This could be optimized further with evolutionary algorithms [Bäc96] by evaluating multiple agent's performances. If available, agents could utilize direct information of the (modified) game engine. If it is possible to detach the drawing of the game world graphics from the actual simulation loop, as is the case in most modern games, AI agents could operate on the simulation alone to minimize computational load and reduce training times. Besides finding edge weights, an AI agent trained like this could also be used to perform a *Monte Carlo Tree Search* (*MCTS*) [Cou07]. In a nutshell MCTS, when applied to games, works by randomly sampling next moves in a game tree, playing out the game to the end and assessing the selected moves by the outcome. Edges in the game tree representing these moves are then given a weight so that better moves are chosen more frequently. This would remove the need for exhaustive graph knowledge, but also require an agent to quickly and accurately simulate every possible move. This is not a problem in the fields where MCTS is often used like the adversarial board games chess, shogi or Go [Cou07; Sil+16; Sil+17]. The rules that are in place to traverse a speedrun routing game graph display much more complexity and to play out a whole speedrun can take up to multiple hours, rendering an MCTS approach like this unfeasible. It can still be worth to look into this line of research further to facilitate the work with incomplete game graphs.

There is an important differentiation that is being made here and that needs some clarification: The difference between operational AI agents and strategic AI agents. As explained in [GZN22], speedrun routing is a strategical process. Most video game AI agents perform well on an operational level, i.e. deciding on next inputs on a frame to frame basis or over a limited window of frames, given a set of input variables, like screen pixels, objects detected on screen or internal engine information as in [TKB10] and [LC17]. On a strategic scale as routing for

an RPG game with some degree of complexity to it, these approaches have not been tested yet. A purely operational AI agent however is not likely to perform well at routing tasks as any routing relevant progress would have to emerge from many operational actions by chance, which on its own might not seem beneficial. The same assumption is made by the authors of the *Starcraft II* AI *AlphaStar* [Vin+19], calling it "*the exploration problem*":

> Discovering new strategies without any guide would be a "needle in a haystack problem," Prof Silver said, with the agent required to stumble upon a series of steps with a beneficial outcome. "You'd have to do so many unlikely things, each of which in turn looks really bad from where you are." [Kel19]

Their solution to this problem is to provide the AI with a database of top-level players' matches to learn from by imitating their moves. Following this principle, a speedrun routing AI could be provided with previous manually created routes to learn from. For most games however, such a database does not yet exist in a usable form and would most likely not be a large enough dataset to actually be usable for AI training. Hausknecht and Stone [HS15] implemented a Recurrent Deep Reinforcement Learning AI agent, that is able to remember information for an arbitrarily long time, which can be helpful in tactical decision making. Lample and Chaplot [LC17] already introduce tactical considerations into their approach by using separate networks for different phases of the game *Doom*: One network for the navigation phase to discover enemies and items (tactical) and one network for the action phase to fight encountered enemies (operational). Dedicated networks or layers to make strategic decisions and provide directions for lower-level networks or layers to realize tactical and/or operational tasks can be a promising development towards routing-capable AI agents.

One aspect covered in [GZN22] that has not yet been discussed is *Multi-Objective Optimization* (*MOO*) [Deb14; ED18]. In general, MOO describes an optimization problem with multiple (conflicting) objective functions to optimize. This often leads to a set of multiple solutions, none of which being strictly better than any other with regard to all the objectives — also called *dominating*. If a solution cannot be further optimized with regard to any objective without impairing another objective, it is called nondominated or Pareto optimal. A set of such solutions is then called the Pareto front. Applied to speedrunning, while the assignment of rewards and requirements to edge weights might be considered an approach similar to multiple objectives, a more direct form of this would be to assign e.g. a difficulty rating to edges or nodes. This way a potential routing algorithm could yield different routes for runners with different levels of skill and experience. Another possible metric would be a utility or "hidden gain" rating, denoting the amount of possible route speedup or consistency a node can provide. Bomb and bombchu pickups in *OoT* for example would possess a high utility rating, as the count of explosives is a common issue while routing. This way, safer or more consistent routes could be

found that do not emerge when optimizing only for timing based edge weights. Regarding the routing algorithm, section *5 Solving the Routing Problem* presented thoughts on an ACO approach, letting the ant agents stochastically traverse edges with unknown weights. When doing so, an uncertainness rating could be assigned to the route or just the corresponding edge. This rating can then be used either just to identify those uncertain routes, or even to minimize this rating, if the approach yields too many invalid routes.

Many speedruns have to deal with randomness, in the speedrunning community often called *RNG*, derived from the term *Random Number Generation*. RNG elements are notoriously difficult to include in a route for obvious reasons. Fortunately, most occurrences of RNG regard enemy behavior and therefore tactical or operational concerns. It is possible though for RNG to influence routing decisions. One possible way to engage this problem is the use of statistical data if available. A form of data often available are *droptables*. Droptables are tables that denote the probability of specific outcomes, like item drops, of certain game elements or events, like cut grass patches, smashed rocks or defeated enemies. Figure 20 depicts an overview of the drop tables of *OoT*.

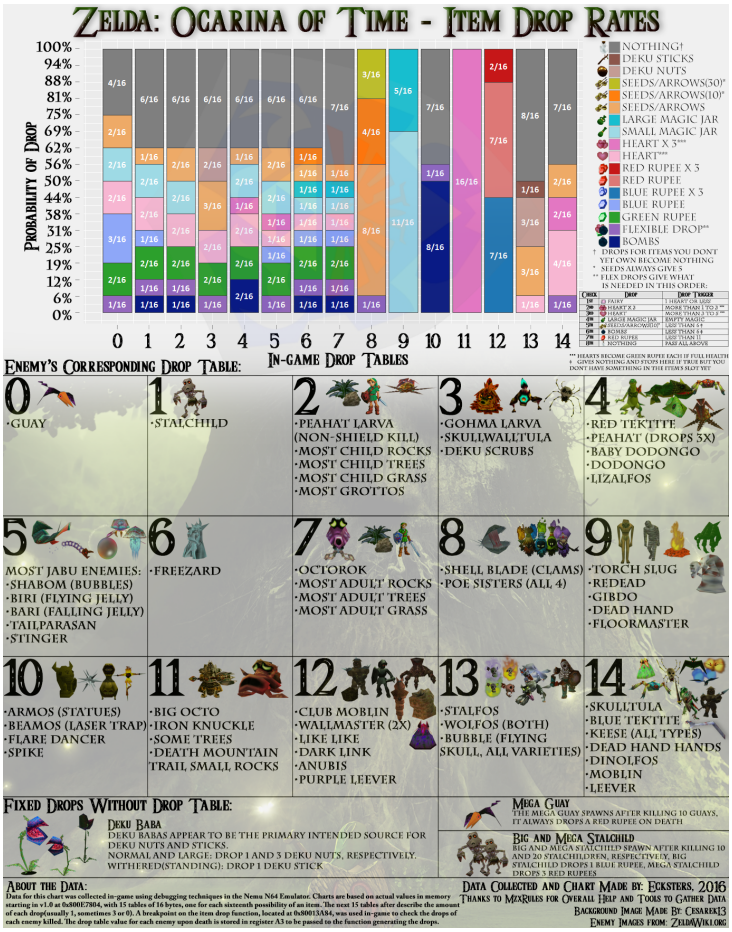

Figure 20: *OoT* drop tables overview showing probabilities of different items to be granted by different game elements and events; by Daniel "Ecksters" Eck [DropRates].

To improve the consistency of generated routes, drop tables or similar statistical information could be used as an additional metric for multi-objective optimization, for biasing traversal decisions of ACO agents or just for convenience by displaying drop chances in the user interface of a crowdsourcing data collection platform.

It was shown that, while the current *OoT* model is incomplete and flawed, it can still be of importance in other speedrun supporting endeavors to come. Despite of the provisional bridges that needed to be put in place, the presented procedure can be considered a viable method of modeling a game, with the potential to be useful in multiple different speedrun related contexts. The suggestions of [GZN22] have been assessed and most importantly, many interesting challenges and possible fields of further study and research regarding the modeling and algorithmic approach of speedrun routing have been identified. If nothing else, this work proves the complexity and dynamics a speedrun model must represent, as well as the need to support ventures like this by scientific methods.

Ludography

Celeste (2018)

Developer: Extremely OK Games

Publisher: Extremely OK Games

Platforms: PC (Linux, macOS, Windows), Nintendo Switch, PlayStation 4, Xbox One, Stadia

Doom (1993)

Developer: id Software

Publisher: id Software

Platforms: Originally MS-DOS. Ported to numerous systems, see [DoomPorts].

Dragon's Dogma (2012)

Developer: Capcom

Publisher: Capcom

Platforms: PC (Windows), PlayStation 3, PlayStation 4, Xbox 360, Xbox One, Nintendo Switch

The Elder Scrolls III: Morrowind (2002)

Developer: Bethesda Game Studios

Publisher: Bethesda Softworks

Platforms: PC (Windows), Xbox

Furi (2016)

Developer: The Game Bakers

Publisher: The Game Bakers

Platforms: PC (Windows), PlayStation 4, Xbox One, Nintendo Switch

A Hat in Time (2017)

Developer: Gears for Breakfast

Publisher: Humble Bundle

Platforms: PC (macOS, Windows), PlayStation 4, Xbox One, Nintendo Switch

The Legend of Zelda: Majora's Mask (2000)

Developer: Nintendo EAD

Publisher: Nintendo

Platforms: Nintendo 64, GameCube

The Legend of Zelda: Ocarina of Time (*OoT*) (1998)
Developer: Nintendo EAD
Publisher: Nintendo
Platforms: Nintendo 64, GameCube, iQue Player

Loop Hero (2021)
Developer: Four Quarters
Publisher: Devolver Digital
Platforms: PC (Linux, macOS, Windows), Nintendo Switch

Mega Man / 2 / 3 (1987 / 1988 / 1990 resp.)
Developer: Capcom
Publishers: Capcom (Mega Man 3: Capcom, JP/NA; Nintendo, EU)
Platforms: NES, PlayStation, Mobile phone, Android, iOS

StarCraft II (2010)
Developer: Blizzard Entertainment
Publisher: Blizzard Entertainment
Platforms: PC (macOS, Windows)

Super Mario World (1990)
Developer: Nintendo EAD
Publisher: Nintendo
Platforms: Super NES, Game Boy Advance

TrackMania Nations Forever (2008)
Developer: Nadeo
Publishers: Focus, France; Enlight, USA; Deep Silver, Germany; Digital Jesters, UK
Platforms: PC (Windows)

Code and Data Availabilty

The code produced as well as data extracted and produced during the course of this work are available in the GitHub repository [Gro22]. The version relevant for the statements made in this work is **`v0.1.0-alpha`**, released on Apr. 2, 2022, which is also the version referenced by [Gro22].

References

[Abr+11] Ittai Abraham et al. "A Hub-Based Labeling Algorithm for Shortest Paths in Road Networks." In: *Experimental Algorithms*. Ed. by Panos M. Pardalos and Steffen Rebennack. Berlin, Heidelberg: Springer, 2011, pp. 230–241. DOI: 10.1007/978-3-642-20662-7_20.

[Bäc96] Thomas Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. New York: Oxford University Press, 1996. 329 pp. DOI: 10.1093/oso/9780195099713.001.0001.

[Bau+10] Reinhard Bauer et al. "Combining Hierarchical and Goal-Directed Speed-up Techniques for Dijkstra's Algorithm." In: *ACM Journal of Experimental Algorithmics* 15 (2010), 2.3:2.1–2.3:2.31. DOI: 10.1145/1671970.1671976.

[BB07] Wilma Alice Bainbridge and William Sims Bainbridge. "Creative Uses of Software Errors: Glitches and Cheats." In: *Social Science Computer Review* 25.1 (2007), pp. 61–77. DOI: 10.1177/0894439306289510.

[Bru09] Roberto Brunelli. *Template Matching Techniques in Computer Vision*. Chichester, UK: John Wiley & Sons, Ltd, 2009. DOI: 10.1002/9780470744055.

[CH66] Kenneth L Cooke and Eric Halsey. "The Shortest Route through a Network with Time-Dependent Internodal Transit Times." In: *Journal of Mathematical Analysis and Applications* 14.3 (1966), pp. 493–498. DOI: 10.1016/0022-247X(66)90009-6.

[Cou07] Rémi Coulom. "Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search." In: *Computers and Games*. Ed. by H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers. Berlin, Heidelberg: Springer, 2007, pp. 72–83. DOI: 10.1007/978-3-540-75538-8_7.

[DD99] M. Dorigo and G. Di Caro. "Ant Colony Optimization: A New Meta-Heuristic." In: *Proceedings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No. 99TH8406)*. Vol. 2. 1999, pp. 1470–1477. DOI: 10.1109/CEC.1999.782657.

[Deb14] Kalyanmoy Deb. "Multi-Objective Optimization." In: *Search Methodologies: Introductory Tutorials in Optimization and Decision Support Techniques*. Ed. by Edmund K. Burke and Graham Kendall. Boston, MA: Springer US, 2014, pp. 403–449. DOI: 10.1007/978-1-4614-6940-7_15.

[Del+11] Daniel Delling et al. “Customizable Route Planning.” In: *Experimental Algorithms*. Ed. by Panos M. Pardalos and Steffen Rebennack. Berlin, Heidelberg: Springer, 2011, pp. 376–387. DOI: 10.1007/978-3-642-20662-7_32.

[DG97] M. Dorigo and L.M. Gambardella. “Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem.” In: *IEEE Transactions on Evolutionary Computation* 1.1 (1997), pp. 53–66. DOI: 10.1109/4235.585892.

[Dij59] E. W. Dijkstra. “A Note on Two Problems in Connexion with Graphs.” In: *Numerische Mathematik* 1 (1959), pp. 269–271. ISSN: 0029-599X; 0945-3245/e. URL: https://eudml.org/doc/131436 (visited on Apr. 1, 2022).

[ED18] Michael T. M. Emmerich and André H. Deutz. “A Tutorial on Multiobjective Optimization: Fundamentals and Evolutionary Methods.” In: *Natural Computing* 17.3 (2018), pp. 585–609. DOI: 10.1007/s11047-018-9685-y.

[Fal+22] Jesús Guillermo Falcón-Cardona et al. “Multi-Objective Ant Colony Optimization: An Updated Review of Approaches and Applications.” In: *Advances in Machine Learning for Big Data Analysis*. Ed. by Satchidananda Dehuri and Yen-Wei Chen. Vol. 218. Intelligent Systems Reference Library. Singapore: Springer, 2022, pp. 1–32. DOI: 10.1007/978-981-16-8930-7_1.

[For18] Dom Ford. “Speedrunning: Transgressive play in digital space.” In: *Proceedings of Nordic DiGRA 2018*. Bergen, Norway, 2018. DOI: 10.13140/RG.2.2.12357.91369.

[Gei+08] Robert Geisberger et al. “Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks.” In: *Experimental Algorithms*. Ed. by Catherine C. McGeoch. Berlin, Heidelberg: Springer, 2008, pp. 319–333. DOI: 10.1007/978-3-540-68552-4_24.

[GH05] Andrew V. Goldberg and Chris Harrelson. “Computing the Shortest Path: A* Search Meets Graph Theory.” In: *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’05. USA: Society for Industrial and Applied Mathematics, 2005, pp. 156–165. ISBN: 978-0-89871-585-9. URL: https://dl.acm.org/doi/10.5555/1070432.1070455 (visited on Mar. 2, 2022).

[GZN22] Matthias Groß, Dietlind Zühlke, and Boris Naujoks. “Automating Speedrun Routing: Overview and Vision.” In: Submitted and accepted to the Special Session on Soft Computing Applied to Games of EvoApplications 2022

as part of EvoStar 2022, to be published in its proceedings. 2022. arXiv: 2106.01182 [cs].

[Hay20] Jonathan Hay. "Fully Optimized: The (Post)Human Art of Speedrunning." In: *Journal of Posthuman Studies* 4.1 (2020), pp. 5–24. DOI: 10.5325/jpoststud.4.1.0005.

[Hem20] Michael Hemmingsen. "Code Is Law: Subversion and Collective Knowledge in the Ethos of Video Game Speedrunning." In: *Sport, Ethics and Philosophy* 15.3 (2020), pp. 1–26. DOI: 10.1080/17511321.2020.1796773.

[HG97] F. Harary and G. Gupta. "Dynamic Graph Models." In: *Mathematical and Computer Modelling* 25.7 (1997), pp. 79–87. DOI: 10.1016/S0895-7177(97)00050-2.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/TSSC.1968.300136.

[Hou62] Paul V. C. Hough. "Method and Means for Recognizing Complex Patterns." U.S. pat. 3069654A. Paul V C Hough. 1962. URL: https://patents.google.com/patent/US3069654A/en (visited on Mar. 27, 2022).

[HS15] Matthew Hausknecht and Peter Stone. "Deep Recurrent Q-Learning for Partially Observable MDPs." In: *2015 AAAI Fall Symposium Series*. 2015. URL: https://www.aaai.org/ocs/index.php/FSS/FSS15/paper/view/11673 (visited on Mar. 25, 2022).

[HSS08] 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. URL: https://www.osti.gov/biblio/960616 (visited on Mar. 2, 2022).

[Kel19] Leo Kelion. "DeepMind AI Achieves Grandmaster Status at Starcraft 2." In: *BBC News. Technology* (Oct. 30, 2019). URL: https://www.bbc.com/news/technology-50212841 (visited on Mar. 25, 2022).

[KK89] Tomihisa Kamada and Satoru Kawai. "An Algorithm for Drawing General Undirected Graphs." In: *Information Processing Letters* 31.1 (1989), pp. 7–15. DOI: 10.1016/0020-0190(89)90102-6.

[KL02] Sven Koenig and Maxim Likhachev. "D*lite." In: *Eighteenth National Conference on Artificial Intelligence*. USA: American Association for Artificial Intelligence, 2002, pp. 476–483. ISBN: 978-0-262-51129-2. URL: https://aaai.org/Library/AAAI/2002/aaai02-072.php (visited on Mar. 6, 2022).

[KL05] Sven Koenig and Maxim Likhachev. "Fast Replanning for Navigation in Unknown Terrain." In: *IEEE Transactions on Robotics* 21.3 (2005), pp. 354–363. DOI: 10.1109/TRO.2004.838026.

[KLF04] Sven Koenig, Maxim Likhachev, and David Furcy. "Lifelong Planning A*." In: *Artificial Intelligence* 155.1 (2004), pp. 93–146. DOI: 10.1016/j.artint.2003.12.001.

[Laf18] Manuel Lafond. "The Complexity of Speedrunning Video Games." In: *9th International Conference on Fun with Algorithms (FUN 2018)*. Ed. by Hiro Ito et al. Vol. 100. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 27:1–27:19. DOI: 10.4230/LIPIcs.FUN.2018.27.

[LC17] Guillaume Lample and Devendra Singh Chaplot. "Playing FPS Games with Deep Reinforcement Learning." In: *Proceedings of the AAAI Conference on Artificial Intelligence* 31.1 (1 2017). ISSN: 2374-3468. URL: https://ojs.aaai.org/index.php/AAAI/article/view/10827 (visited on Apr. 1, 2022).

[Lou83] Ronald Prescott Loui. "Optimal Paths in Graphs with Stochastic or Multidimensional Weights." In: *Communications of the ACM* 26.9 (1983), pp. 670–676. DOI: 10.1145/358172.358406.

[LS12] Manuel López-Ibáñez and Thomas Stützle. "An Experimental Analysis of Design Choices of Multi-Objective Ant Colony Optimization Algorithms." In: *Swarm Intelligence* 6.3 (2012), pp. 207–232. DOI: 10.1007/s11721-012-0070-7.

[McK10] Wes McKinney. "Data Structures for Statistical Computing in Python." In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman. Austin, Texas, 2010, pp. 56–61. DOI: 10.25080/Majora-92bf1922-00a.

[Mic16] Othon Michail. "An Introduction to Temporal Graphs: An Algorithmic Perspective*." In: *Internet Mathematics* 12.4 (2016), p. 1606. DOI: 10.1080/15427951.2016.1177801.

[Moc+10] Jose A. Mocholi et al. “An Emotionally Biased Ant Colony Algorithm for Pathfinding in Games.” In: *Expert Systems with Applications* 37.7 (2010), pp. 4921–4927. DOI: 10.1016/j.eswa.2009.12.023.

[New08] J. Newman. *Playing with Videogames*. London: Routledge, 2008. 207 pp. ISBN: 978-0-415-38523-7. URL: http://www.routledge.com/books/details/9780415385237/ (visited on Apr. 1, 2022).

[New19] James Newman. “Wrong Warping, Sequence Breaking, and Running through Code.” In: *Journal of the Japanese Association for Digital Humanities* 4.1 (2019), pp. 7–36. DOI: 10.17928/jjadh.4.1_7.

[Raj+15] Mojtaba Rajabi-Bahaabadi et al. “Multi-Objective Path Finding in Stochastic Time-Dependent Road Networks Using Non-Dominated Sorting Genetic Algorithm.” In: *Expert Systems with Applications* 42.12 (2015), pp. 5056–5064. DOI: 10.1016/j.eswa.2015.02.046.

[Ric21] Martin Ricksand. “‘Twere Well It Were Done Quickly’: What Belongs in a Glitchless Speedrun?” In: *Game Studies* 21.1 (2021). ISSN: 1604-7982. URL: http://gamestudies.org/2101/articles/ricksand (visited on Apr. 1, 2022).

[Scu14] Rainforest Scully-Blaker. “A Practiced Practice: Speedrunning Through Space With de Certeau and Virilio.” In: *Game Studies* 14.1 (2014). ISSN: 1604-7982. URL: http://gamestudies.org/1401/articles/scullyblaker (visited on Apr. 1, 2022).

[Scu16] Rainforest Scully-Blaker. “Re-Curating the Accident: Speedrunning as Community and Practice.” MA thesis. Concordia University, 2016. 108 pp. URL: https://spectrum.library.concordia.ca/982159/ (visited on Apr. 1, 2022).

[Scu18] Rainforest Scully-Blaker. “The Speedrunning museum of accidents.” In: *Kinephanos* (Preserving Play, Special Issue Aug. 2018), pp. 71–88. ISSN: 1916-985X. URL: https://www.kinephanos.ca/2018/the-speedrunning-museum-of-accidents/ (visited on Apr. 1, 2022).

[SH00] Thomas Stützle and Holger H. Hoos. “MAX–MIN Ant System.” In: *Future Generation Computer Systems* 16.8 (2000), pp. 889–914. DOI: 10.1016/S0167-739X(00)00043-1.

[Sil+16] David Silver et al. “Mastering the Game of Go with Deep Neural Networks and Tree Search.” In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[Sil+17] David Silver et al. “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm.” 2017. arXiv: 1712.01815 [cs]. (Visited on Mar. 25, 2022).

[Ste93] Anthony Stentz. “Optimal and Efficient Path Planning for Unknown and Dynamic Environments.” In: *International Journal of Robotics and Automation* 10 (1993), pp. 89–100. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.3683 (visited on Mar. 4, 2022).

[Ste95] Anthony Stentz. “The Focussed D* Algorithm for Real-Time Replanning.” In: *In Proceedings of the International Joint Conference on Artificial Intelligence*. 1995, pp. 1652–1659. URL: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.8257 (visited on Mar. 4, 2022).

[TKB10] Julian Togelius, Sergey Karakovskiy, and Robin Baumgarten. “The 2009 Mario AI Competition.” In: *IEEE Congress on Evolutionary Computation*. 2010, pp. 1–8. DOI: 10.1109/CEC.2010.5586133.

[Umb17] Scott E. Umbaugh. *Digital Image Processing and Analysis: Applications with MATLAB® and CVIPtools*. 3rd ed. Boca Raton: CRC Press, 2017. 897 pp. DOI: 10.1201/9781351228374.

[Vin+19] Oriol Vinyals et al. “Grandmaster Level in StarCraft II Using Multi-Agent Reinforcement Learning.” In: *Nature* 575.7782 (2019), pp. 350–354. DOI: 10.1038/s41586-019-1724-z.

[WW07] Dorothea Wagner and Thomas Willhalm. “Speed-Up Techniques for Shortest-Path Computations.” In: *Annual Symposium on Theoretical Aspects of Computer Science (STACS) 2007*. Ed. by Wolfgang Thomas and Pascal Weil. Berlin, Heidelberg: Springer, 2007, pp. 23–36. DOI: 10.1007/978-3-540-70918-3_3.

[Ye+20] Deheng Ye et al. “Mastering Complex Control in MOBA Games with Deep Reinforcement Learning.” In: *Proceedings of the AAAI Conference on Artificial Intelligence* 34.04 (04 2020), pp. 6672–6679. DOI: 10.1609/aaai.v34i04.6144.

Unpublished Sources

[Eck22] *Interview with Daniel ”Ecksers” Eck, Author of the Online OoT Interactive Map and the Drop Table Overwiew*. In collab. with Daniel ”Ecksters” Eck. Oct. 6, 2021.

Software

[Aar18] Gregory Aaronson. *Gdaaronson/Thesis*. 2018. URL: https://github.com/gdaaronson/Thesis (visited on Feb. 3, 2022).

[Ahorn22] CelestialCartographers. *Ahorn*. 2022. URL: https://github.com/CelestialCartographers/Ahorn (visited on Mar. 27, 2022).

[Dan14] Daniel "xdanieldzd" R. *SceneNavi*. Version v1.0 Beta 9b. Apr. 11, 2014. URL: https://github.com/xdanieldzd/SceneNavi/releases/tag/v1.0-beta9b (visited on Feb. 19, 2022).

[Eck21] Daniel "Ecksters" Eck. *Zelda: Ocarina of Time - Interactive Map*. 2021. URL: https://github.com/Ecksters/OoT-Interactive-Map (visited on Feb. 19, 2022).

[gla21] glank. *Glankk/Gz*. In collab. with Krimtonz et al. Version gz-0.3.6. Aug. 31, 2021. URL: https://github.com/glankk/gz/releases/tag/v0.3.6 (visited on Feb. 19, 2022).

[Gro22] Matthias Groß. *OoT-Graph-Model*. Version v0.1.0-alpha. Apr. 2, 2022. URL: https://github.com/DarkMuesli/OoT-Graph-Model/releases/tag/v0.1.0-alpha (visited on Apr. 2, 2022).

[Hag22] Aric Hagberg. *NetworkX*. Version 2.7.1. Mar. 5, 2022. URL: https://networkx.org/ (visited on Apr. 2, 2022).

[Har22] HarbourMasters. *Ship of Harkinian*. Mar. 25, 2022. URL: https://github.com/HarbourMasters/Shipwright (visited on Mar. 25, 2022).

[Mot20] Marcus Mottare. *Hzck/Speedroute*. 2020. URL: https://github.com/hzck/speedroute (visited on Apr. 1, 2022).

[mzx19] mzxrules. *VerboseOcarina*. Nov. 26, 2019. URL: https://wiki.cloudmodding.com/oot/App:VerboseOcarina (visited on Jan. 25, 2022).

[Reb+22] Jeff Reback et al. *Pandas-Dev/Pandas: Pandas*. Version 1.4.1. Feb. 12, 2022. DOI: 10.5281/zenodo.6053272.

Web Sources

[CLG-creators] *CLG Content Creators*. CLG. URL: https://www.clg.gg/clg-creators (visited on Mar. 15, 2022).

[CM-ET] *Entrance Table (Data)*. CloudModding OoT Wiki. URL: https://wiki.cloudmodding.com/oot/Entrance_Table_(Data) (visited on Mar. 18, 2022).

[Com18] Mauro Comi. *How to Teach an AI to Play Games: Deep Reinforcement Learning*. Medium. Nov. 15, 2018. URL: https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a (visited on Apr. 1, 2022).

[Dan21] Daniel Baamonde, director. *[WR] Glitchless Any% Unrestricted Speedrun in 2:18:47!* May 15, 2021. URL: https://www.youtube.com/watch?v=OOHLEP0Ew4c (visited on Mar. 12, 2022).

[DoomPorts] *List of* Doom *Ports*. Mar. 19, 2022. URL: https://en.wikipedia.org/w/index.php?title=List_of_Doom_ports&oldid=1077953637 (visited on Mar. 28, 2022).

[DropRates] *Item Drop Rate Chart by Ecksters - Guides - The Legend of Zelda: Ocarina of Time - Speedrun.Com*. URL: https://www.speedrun.com/oot/guide/prwj5 (visited on Mar. 20, 2022).

[Išk18] Artjoms Iškovs. *Travelling Murderer Problem: Planning a Morrowind All-Faction Speedrun with Simulated Annealing*. Apr. 18, 2018. URL: https://www.kimonote.com/@mildbyte/travelling-murderer-problem-planning-a-morrowind-all-faction-speedrun-with-simulated-annealing-part-1-41079/ (visited on Apr. 1, 2022).

[Jst19] JstAnothrVirtuoso, director. *Finding the Optimum Nadeo Cut... With Science!!* May 28, 2019. URL: https://www.youtube.com/watch?v=1ZsAjvO9E1g (visited on Apr. 1, 2022).

[Mon] Kieran "Volvagia" Monaghan. *Ocarina of Time - Practice ROM*. URL: https://www.practicerom.com/ (visited on Feb. 19, 2022).

[mzx-WWCalc] mzxrules. *Wrong Warp Calculator*. URL: https://mzxrules.github.io/zelda64/ocarina/ww/ (visited on Mar. 18, 2022).

[NetworkXDoc] *Reference — NetworkX Documentation*. URL: https://networkx.org/documentation/stable/reference/ (visited on Mar. 26, 2022).

[OoTDiscord] *OoT Speedrunning Discord Server*. URL: https://discord.gg/7FYGh3d (visited on Mar. 21, 2022).

[OoTGlitchless] *The Legend of Zelda: Ocarina of Time - speedrun.com*. URL: https://www.speedrun.com/oot#Glitchless (visited on Feb. 21, 2022).

[OoTmap.com] Daniel Eck. *OoT Interactive Map*. URL: https://ootmap.com/ (visited on Feb. 10, 2022).

[OoTVersions] *Version Differences - ZeldaSpeedRuns*. URL: https://www.zeldaspeedruns.com/oot/generalknowledge/version-differences (visited on Feb. 10, 2022).

[pandasDoc] *API Reference — Pandas Documentation*. URL: https://pandas.pydata.org/docs/reference/ (visited on Mar. 26, 2022).

[PythonDoc] *5. Data Structures — Python 3.10.4 Documentation*. URL: https://docs.python.org/3/tutorial/datastructures.html (visited on Mar. 27, 2022).

[Speedrun.com] *Speedrun.com*. URL: https://www.speedrun.com/ (visited on Apr. 1, 2022).

[SR.C-SiteRules] *Site Rules - Knowledge Base - Speedrun.Com*. URL: https://www.speedrun.com/knowledgebase/site-rules (visited on Mar. 21, 2022).

[SRMTables] *OoT SRM / GIM Tables*. URL: https://docs.google.com/spreadsheets/d/1SLJzamokLb7wDOaJh5x8DsxmMBy9oIYawyDN3dAWppw (visited on Feb. 19, 2022).

[twitch.tv] *Twitch*. URL: https://www.twitch.tv (visited on Mar. 31, 2022).

[UltMMSheet] *The Ultimate MM Spreadsheet*. URL: https://docs.google.com/spreadsheets/d/1J-4OwmZzOKEv2hZ7wrygOpMm0YcRnephEo3Q2FooF6E/edit?usp=embed_facebook (visited on Mar. 24, 2022).

[UltOoTSheet] *The Ultimate OoT Spreadsheet*. URL: https://docs.google.com/spreadsheets/d/1Sl_ay1qPxTrs6xBTpVeqhKFrJjOGYyh3Tg1W0Ch5aHw (visited on Jan. 25, 2022).

[vgmaps.com] *The Video Game Atlas - OoT Maps*. URL: http://www.vgmaps.com/Atlas/N64/index.htm#LegendOfZeldaOcarinaOfTime (visited on Mar. 27, 2022).

[Vol18] Volvy. *Reddit Post about the Morrowind All Factions Speedrun Route*. r/speedrun. Nov. 4, 2018. URL: www.reddit.com/r/speedrun/comments/9u1r9o/using_ai_to_grind_out_routes/e91dg6w/ (visited on Apr. 1, 2022).

[ZeldaRET] *ZeldaRET*. URL: https://zelda64.dev/ (visited on Mar. 17, 2022).

[ZSR-ET] *Entrance Table - ZeldaSpeedRuns*. URL: https://www.zeldaspeedruns.com/oot/wrongwarp/entrance-table (visited on Mar. 18, 2022).

[ZSR-WW] *Wrong Warp - ZeldaSpeedRuns*. URL: https://www.zeldaspeedruns.com/oot/wrongwarp/wrong-warp (visited on Mar. 18, 2022).

[ZSR-WWExp] *Wrong Warp Explained - ZeldaSpeedRuns*. URL: https://www.zeldaspeedruns.com/oot/wrongwarp/wrong-warp-explained (visited on Mar. 18, 2022).

[ZSR-WWTable] *Wrong Warp Table - ZeldaSpeedRuns*. URL: https://www.zeldaspeedruns.com/oot/wrongwarp/wrong-warp-table (visited on Mar. 18, 2022).

Appendix

Excerpt of *OoT* Speedrunning Discord conversations

Figure 21: Save Warp explanation by *DannyB*.

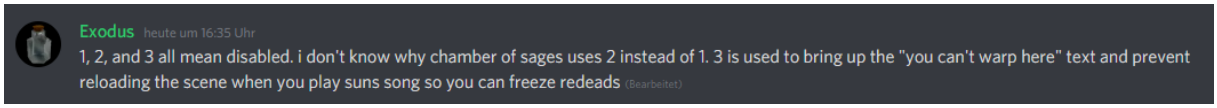

Figure 22: Item usability explanation by *Exodus*.

Declaration in lieu of oath

I hereby declare in lieu of an oath that this work is my own and that I have not used any sources other than those listed in the bibliography. Content from published or unpublished works that has been quoted directly or indirectly or paraphrased is indicated as such. The work has not been submitted in the same or similar form or in part for any other academic award. The electronic version I have submitted is completely identical to the hard copy version submitted.

I am aware that my work may be checked for unmarked copying of others' intellectual property for the purpose of a plagiarism check using plagiarism detection software. I am aware of the punishability of a false Declaration in lieu of an oath, namely the threat of punishment according to § 156 StGB up to three years imprisonment or fine for intentional committal of the offense or according to § 161 Abs. 1 StGB up to one year imprisonment or fine if committed by negligence.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder Ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden. Ich versichere, dass die eingereichte elektronische Fassung der eingereichten Druckfassung vollständig entspricht.

Mir ist bekannt, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs mittels einer Plagiatserkennungssoftware auf ungekennzeichnete Übernahme von fremdem geistigem Eigentum überprüft werden kann. Die Strafbarkeit einer falschen eidesstattlichen Versicherung ist mir bekannt, namentlich die Strafandrohung gemäß § 156 StGB bis zu drei Jahren Freiheitsstrafe oder Geldstrafe bei vorsätzlicher Begehung der Tat bzw. gemäß § 161 Abs. 1 StGB bis zu einem Jahr Freiheitsstrafe oder Geldstrafe bei fahrlässiger Begehung.

Place, Date, Signature

Ort, Datum, Unterschrift