The complexity of speedrunning video games

Manuel Lafond[1]
Department of Mathematics and Statistics, University of Ottawa, Canada
mlafond2@uOttawa.ca

Abstract

Speedrunning is a popular activity in which the goal is to finish a video game as fast as possible. Players around the world spend hours each day on live stream, perfecting their skills to achieve a world record in well-known games such as Super Mario Bros, Castlevania or Mega Man. But human execution is not the only factor in a successful speed run. Some common techniques such as *damage boosting* or *routing* require careful planning to optimize time gains. In this paper, we show that optimizing these mechanics is in fact a profound algorithmic problem, as they lead to novel generalizations of the well-known NP-hard knapsack and feedback arc set problems.

We show that the problem of finding the optimal damage boosting locations in a game admits an FPTAS and is FPT in $k + r$, the number k of enemy types in the game and r the number of health refill locations. However, if the player is allowed to lose a life to regain health, the problem becomes hard to approximate within a factor 1/2 but admits a $(1/2 - \epsilon)$-approximation with two lives. Damage boosting can also be solved in pseudo-polynomial time. As for routing, we show various hardness results, including $W[2]$-hardness in the time lost in a game, even on bounded treewidth stage graphs. On the positive side, we exhibit an FPT algorithm for stage graphs of bounded treewidth and bounded in-degree.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms, Theory of computation → Approximation algorithms analysis, Theory of computation → Parameterized complexity and exact algorithms

Keywords and phrases Approximation algorithms, parameterized complexity, video games, knapsack, feedback arc set

Digital Object Identifier 10.4230/LIPIcs.FUN.2018.27

1 Introduction

The study of the complexity of video games has been a relatively popular area of research in the recent years. This line of work first started in the early 2000s with puzzle-oriented video games such as *Minesweeper*, *Tetris* or *Lemmings* [22, 11, 25][2]. More recently, platforming games were subjected to complexity analysis [17], and it is now known that for a wide variety of such games (including *Super Mario Bros*, *Donkey Kong Country* or *Zelda*), it is NP-hard [5] or sometimes PSPACE-hard [13] to decide whether a given instance of the game can be finished. Notably, Viglietta proposed in [24] a series of *meta-theorems* that describe common video game mechanics under which a game is NP-hard or PSPACE-hard.

Of course, few games are (computationally) hard to finish, as there is little incentive for publishers to release an unfinishable game. Here, we take a different perspective on the complexity of video games, and rather ask *how fast can a game be finished*? This question is of special interest to the adepts of *speedrunning*, in which the goal is to finish a video game

[1] The author acknowledge the Natural Sciences and Engineering Research Council of Canada (NSERC) for the financial support of this project.

[2] All products, company names, brand names, trademarks, and sprites are properties of their respective owners. Video game screen-shots and sprites are used here under Fair Use for educational purposes.

9th International Conference on Fun with Algorithms (FUN 2018).
Editors: Hiro Ito, Stefano Leonardi, Linda Pagli, and Giuseppe Prencipe; Article No. 27; pp. 27:1–27:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

as fast as possible. This has been a relatively obscure activity until the last decade, during which speedrunning has seen a significant increase in popularity. This is especially owing to video game streaming websites, where professional speedrunners can spend hours each day on camera trying to earn a world record whilst receiving enough donations from viewers to make a living. Games Done Quick, one of the most popular events in this discipline, is a speedrunning marathon in which professional gamers take turn on live stream to go through a wide range of games as fast as possible [2]. Performances are broadcast 24 hours a day for a whole week, and viewers are invited to provide donations which are then given to charitable organizations. The event went from raising $10,000 during its first event in 2010 to amassing over $2 million in its January 2018 event.

Owing to this popularity, speedrunning is now an extremely competitive area, and having near-perfect execution is mandatory to obtain reasonable times. A single misplaced jump, or an attack that comes a split-second late, can cost a player a world record. There is, however, a category of speedrunning that can circumvent these harsh execution requirements: Tool Assisted Speedruns (TAS). In a TAS, the player is allowed to use any tool provided by emulators, which include slowing down the game, rewinding the game, saving multiple states and reloading them, etc. In the end, the final speedrun is presented in a continuous segment, as if played by a human. In a TAS, execution is therefore not the main challenge, as the player can retry any portion of the game hundreds of times if necessary. But speedrunning remains a challenging task, as difficult optimization problems arise.

In this paper, we are interested in the algorithmic challenges underlying some common mechanics that are unique to speedrunning. We first formulate the problem of speedrunning by modeling a game as a series of punctual *time-saving events*, which can be taken or not. This is in contrast with the natural formulation "given a video game X, can X be finished in time t", as it was done for *Mario Kart* in [7]. This allows our results to be applicable to any game that can be described by time-saving events, and also enables us to avoid dealing with unfinishable games.

We then study the approximation and parameterized complexity aspects of the techniques of *damage boosting* and *routing stages*. Damage boosting consists in taking damage intentionally to go through some obstacles quickly. The amount of damage that can be taken in a game is limited, and it is possible to regain health using items, or by losing a life (this is called *death abusing*). This can be seen as a generalization of the *knapsack* problem in which the items come in a specific order and some of them have a negative weight. We show that if no life can be lost, optimizing damage boosts in a game admits the same FPTAS as knapsack, and is fixed-parameter tractable (FPT) in the number of possible damage sources and healing locations. If lives can be lost to regain health, we show that damage boosting cannot be approximated within a factor $1/2$ or better, but can be approximated within a factor $1/2 - \epsilon$ with two lives and can be solved in pseudo-polynomial time.

Routing applies to games in which the player is free to choose in which order a set of stages is to be completed. This includes the *Mega Man* games, for example. Each completed stage yields a new ability to the player, which can then be used in later stages to gain time on certain events, such as defeating a boss more quickly. The time saved in an event depends on the best ability currently available. As we shall see, this makes Routing a generalization of the well-known *feedback arc set* (FAS) problem, as the time-gain dependencies can be represented as a directed graph D. Unlike FAS though, we show that Routing is $W[2]$-hard in the time lost in a game, even if D has treewidth 1, and that it is also hard to approximate within a $\mathcal{O}(\log n)$ factor. We then show that Routing is FPT in the maximum in-degree of D plus its treewidth.

The paper is structured as follows. In Section 2, we provide a non-technical summary of the speedrunning mechanics that are discussed in this work and present our general model of speedrunning. In Section 3, we formally define the problem of optimizing damage boosting and present our algorithmic results. Then in Section 4, we define our routing optimization problems and provide the underlying algorithmic results.

2 Models, speedrunning mechanics, and problems

In this section, we first motivate our model of speedrunning, and how we depart from the traditional formulation of deciding whether a stage can be finished. We then describe the two speedrunning mechanics that we study in more detail.

As mentioned before, perhaps the most natural formulation of speedrunning is the following: given a set of stages, we are asked whether they can be completed in time at most t [7]. However for many games, it is NP-hard to decide whether a given set of stages can be completed at all (e.g. [24]). It follows that for these games, speedrunning is NP-hard even for $t = \infty$. But in reality, video games that are played by speedrunners are always known to be completable. We will therefore assume that an initial way of finishing the game is known, which yields an upper bound t on the time required to complete the game. This time t is usually the time taken to finish the game "normally", as intended by the developers. The problem of speedrunning now becomes: given an initial way of completing the game in time t, can this be improved to time $t' < t$?

To simplify further, stages are often linear and time saves usually consist of punctual events that allow the player to save a few seconds over the developer-intended path. For example, the player may exploit a glitch to go through a wall, or use a certain item to defeat an enemy faster than usual. These punctual events are assumed to occur one after another, and therefore, we will model a stage as a sequence $S = (e_1, \dots, e_n)$ of *time-saving events*. For each such event e_i, the player has a choice of taking the time save from e_i or not. If the event only has positive consequences, then of course the player must take it and we will assume that all events in S offer some sort of trade-off. A notable advantage of this formulation is that it does not depend on a specific video game. For instance, if the events of S model damage boosting, then our hardness results apply to any game that allows damage boosting as a mechanic. We now describe this latter notion.

Damage boosting

The idea of damage boosting is to take damage to save time. This is a common technique that is useful in one of the following ways. In many games, the player is given some invulnerability time after taking damage. This invincibility period allows unintended behavior such as walking on deadly spikes or going through a horde of enemies quickly. Also, when taking damage, the player often loses control and gets knocked back, regardless of the current location and status of the character. If damage is taken at the apex of a jump, say, then this back-knocking can extend the jump higher and farther than normal, allowing the player to access unintended locations. An example of this is illustrated in Figure 1.

Damage-boosting is not without cost. In a game, the player has a limited number of *hit points*, or HP for short. In the top-left of Figure 1, one can see that the player has a maximum of 16 HP, but has 14 remaining after hitting a bat. Each time damage is taken, the player's HP decreases by a certain amount and a life is lost when it reaches 0. Suppose that each time-saving event is described by a pair (d, t), where d is the damage taken and the time gained t. Then it is easy to see that this is exactly the knapsack problem. Indeed,

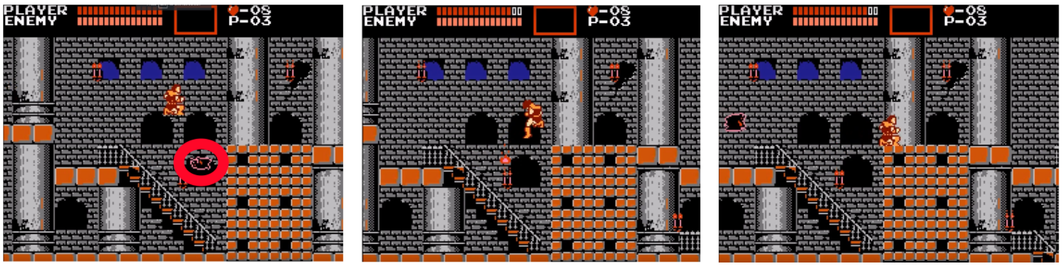

Figure 1 A well-known example of damage boosting in the NES game Castlevania. In this portion of Stage 1, the developer-intended path is to go downstairs, go through an underground section, go up and reappear on the right side of the screen. Here, Simon Belmont can skip the underground section by passing over the wall on the right side of the screen. To do this, the player times a precise jump while facing right, switches direction in mid-air to face left, and lands on a bat passing by at the right moment, damaging the player. When Simon Belmont takes damage, he says "Ow" and gets knocked back (middle figure). This back-knocking allows him to extend his jump farther right and reach the ledge of the wall. This saves 30-40 seconds over taking the normal path.

if *hp* is the starting HP of the player, speedrunning with damage boosting asks for the set of time-saving events that can be taken such that a maximum total time gain is achieved, and such that the total damage of these events does not exceed *hp*.

The problem can be made more interesting by considering the possibility of regaining health during a stage. For instance in Castlevania, there is chicken hidden inside walls and candles across the castle. Fetching these chickens is usually time-consuming, and the player must decide whether the additional damage boosts that this allows is worth it. In Figure 2, Richter Belmont from Castlevania X takes a detour to a dead-end to grab a chicken and regain health. Another way to regain HP is to lose a life. When the player runs out of HP, a life is lost and the game restarts at the last checkpoint with full health. This can be beneficial if the checkpoint is not too far away and new damage boosts are to be taken. It is worth mentioning that the idea of losing lives is used in [13] to establish the hardness of completing a Super Mario Bros game (although losing lives is not used as a health refill mechanism).

Routing

In many games, the player has the freedom to choose the order in which to clear a set of stages or to visit a set of locations. For example, in many Metroid games, a certain set of items scattered across a map must be obtained before reaching the end of the game, and the goal of *routing* is to obtain these items in the optimal order[3].

As another example, in the Mega Man games, the order in which stages are visited is fundamental from a speedrunning perspective. When a stage is cleared after defeating its robot master, Mega Man gains a new weapon/ability that can be used in the latter stages. This has an impact on how fast a stage S can be completed, as previously obtained weapons can be used to gain time – notably during the boss fights. For instance, in Mega Man 2, the boss Crash Man takes 2 damage from the regular weapon, but 20 damage from the weapon left by Air Man. Thus Crash Man can be defeated 10 times faster if the Air Man stage is

[3] The astute reader will observe that *Metroid* games do restrict the ordering of locations that can be visited. However, many sequence-breaking glitches have been found in the last years, and this ordering restriction is often irrelevant. For example in *Super Metroid*, the *Norfair* boss *Ridley* is often defeated first in speedruns, whereas this boss is normally supposed to be reached last.

Figure 2 "This candle chicken certainly tastes great, but it is time-consuming..." - Richter Belmont going out of his way for meat.

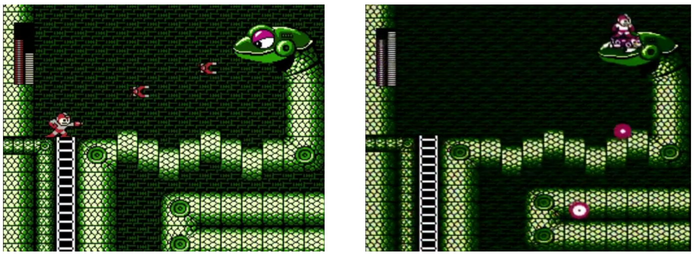

Figure 3 A event in *Mega Man 3*: facing Big Snakey. If Mega Man has Magnet Man's weapon, Big Snakey can be defeated in 5 shots, which saves about 8 seconds. But if Mega Man has acquired Rush Jet from Needle Man, he can simply fly over Big Snakey, which saves about 15 seconds.

cleared beforehand. Bosses are not the only time-saving events in the game though, as the stages themselves also offer many opportunities. See Figure 3 for an example. Here, Mega Man must face Big Snakey and can achieve various time gains depending on the weapons currently at his disposal from the previous stages.

The problem of routing is to determine the order in which to clear the stages so as to save a maximum amount of time. If we represent stages as a graph, with an arc from S' to S weighted by the time saved in stage S by having cleared S' first, this is similar to the feedback arc set problem, which asks for an ordering of the vertices of a graph in a way that the number of forward edges is maximized. One notable difference is that each event within a stage can make use of a different previous stage.

Before proceeding, we introduce some notation that we will use throughout the paper. Given an ordered list $S = (s_1, \ldots, s_n)$, we write $s_i <_S s_j$ if $i < j$ (and $s_i \leq_S s_j$ if $i \leq j$). We denote by *head*(S) and *tail*(S) the first and last element of S, respectively. A *subsequence* of S is another ordered list $S' = (s'_1, \ldots, s'_k)$ in which $s'_i <_S s'_{i+1}$ for every $i \in [k-1]$. Suppose that each element in S is distinct. Let X be the set underlying S. We call S a *linear ordering* of X, or simply an *ordering* for short. Abusing notation slightly, we may treat S as the set X whenever convenient (e.g. we may write $s \in S$ if s occurs in S).

3 Damage boosting

We first study the damage boosting mechanics when losing a life is forbidden. That is, the player can only take damage or refill health, without ever letting health drop to zero. An *event* is an opportunity to gain time by taking damage, and is represented by a pair $e = (d, t)$ where d is the damage to take to save t units of time. We will assume that d and t are integers, possibly negative. If both d and t are negative, we call e a *chicken event*, as d

represents a health refill and t the time lost to regain this health. We denote by $d(e)$ and $t(e)$ the damage and time-save components of e, respectively.

A *stage* $S = (e_1, \ldots, e_n)$ is an ordered list of events. A *solution* $\hat{S} = (\hat{e}_1, \ldots, \hat{e}_k)$ to a stage S is a subsequence of S. We say that we *take* event e_i if $e_i \in \hat{S}$. A given integer hp represents the player's *hit points* (HP) at the start of the stage. The player's hit points can never exceed hp. Each event $\hat{e}_i \in \hat{S}$ leaves the player with a number of hit points $h_{\hat{S}}(\hat{e}_i)$ after being taken. We define $h_{\hat{S}}(\hat{e}_1) = \min(hp, hp - d(\hat{e}_1))$ and, for $i \in \{2, \ldots, k\}$, $h_{\hat{S}}(\hat{e}_i) = \min(hp, h_{\hat{S}}(\hat{e}_{i-1}) - d(\hat{e}_i))$. A solution is *valid* if $h_{\hat{S}}(\hat{e}_i) > 0$ for every $\hat{e}_i$ in $\hat{S}$.

Given a stage S and maximum hit points hp, the objective in the DAMAGE BOOSTING problem is to find a valid solution $\hat{S}$ for S that maximizes $t(\hat{S}) = \sum_{e \in \hat{S}} t(e)$.

As mentioned before, DAMAGE BOOSTING can be viewed as a knapsack instance in which each item is given in order, and we have some opportunities to (partially) empty the sack (which corresponds to our chicken events). It is not hard to show that the well-known pseudo-polynomial time algorithm and FPTAS for knapsack can be adapted to DAMAGE BOOSTING. The proof is essentially the same as in the knapsack FPTAS - we include it here for the sake of completeness.

▶ **Theorem 1.** *DAMAGE BOOSTING can be solved in pseudo-polynomial time $O(n^2 T)$, where T is the maximum time gain of an event. Moreover, DAMAGE BOOSTING admits an FPTAS, and can be approximated within a factor $1 - \epsilon$ in time $\mathcal{O}(n^3/\epsilon)$ for any $\epsilon > 0$.*

Proof. Let (S, hp) be an instance of DAMAGE BOOSTING, $S = (e_1, \ldots, e_n)$. Let $H(i, t)$ denote the highest HP value achievable when gaining a time of exactly t by taking a subset of the events $\{e_1, \ldots, e_i\}$. Define $H(i, t) = -\infty$ if this is not possible, and define $H(0, 0) = hp$ and $H(0, t) = -\infty$ for $t > 0$. Then

$$H(i, t) = \min\{hp, \max\{H(i-1, t), H(i-1, t - t(e_i)\} - d(e_i))\}$$

$H(i, t)$ needs to be computed for each $i \in [n]$ and each $t \in [nT]$. We then look at the maximum value of t such that $H(n, t) > 0$, which leads to a dynamic programming algorithm with the claimed complexity.

To get an FPTAS, we scale the time gains as in the knapsack FPTAS. Let $\epsilon > 0$ and let $c = \epsilon T/n$. Let $S' = (e'_1, \ldots, e'_n)$, where $e'_i = (d(e_i), \lfloor t(e_i)/c \rfloor)$. Let $\hat{S}$ (resp. $\hat{S}'$) be a subsequence of S (resp. S') that maximizes the time gain $t(\hat{S})$ (resp. $t(\hat{S}')$). Observe that for each $e_i \in S$, we have $t(e_i)/c - 1 \leq t(e'_i) \leq t(e)/c$ whether e_i is a chicken event or not. Hence, $t(\hat{S}') \geq \sum_{e_i \in \hat{S}} t(e'_i) \geq t(\hat{S})/c - n$ (where the first inequality is due to the optimality of $\hat{S}'$ on S'). Note that $\hat{S}'$ is a valid solution for S, since the damage values were unchanged from S to S'. The time gained by taking the events of $\hat{S}'$ as our solution for S is

$$\sum_{e'_i \in \hat{S}'} t(e_i) \geq \sum_{e'_i \in \hat{S}'} c \cdot t(e'_i) \geq c \cdot (t(\hat{S})/c - n) = t(\hat{S}) - cn = t(\hat{S}) - \epsilon T \geq (1 - \epsilon) t(\hat{S})$$

where we use $t(\hat{S}) \geq T$ in the last inequality. The algorithm takes time $\mathcal{O}(n^2 T/(\epsilon T/n)) = \mathcal{O}(n^3/\epsilon)$. ◀

From the point of view of parameterized complexity, Theorem 1 implies that DAMAGE BOOSTING is FPT in t, the total time that can be gained (due to results of [8]). However t is typically high, and alternative parameterizations are needed. In the context of video games, although stages can be large, the number of types of enemies and damage sources is usually limited. Likewise, there are usually only a few healing items in a stage. The time gained or lost per event can vary widely though.

We would therefore like to parameterize DAMAGE BOOSTING by the number k of values that $d(e)$ can take in the events of S. It was shown in [15] that knapsack can be solved in time $\mathcal{O}(2^{2.5k \log k} poly(n))$, where k is the number of distinct weights that appear in the input. The algorithm does not seem to extend directly to DAMAGE BOOSTING, and we leave the FPT status of the problem open for k. We do show, however, that if the number of chicken events is also bounded by some integer r, one can devise an FPT algorithm in $k + r$ based on the ideas of [15]. We make use of the result of Lokshtanov [23, Theorem 2.8.2], which improve upon Kannan's algorithm [21] and state that a solution to an Integer Linear Program (ILP) with ℓ variables can be found in time $\mathcal{O}(\ell^{2.5\ell} poly(n))$.

▶ **Theorem 2.** *DAMAGE BOOSTING is FPT in $k + r$, where k is the number of possible damage values and r the number of chicken events. Moreover, an optimal solution can be found in time $\mathcal{O}(2^r (2k(r+1)+r)^{2.5(2k(r+1)+r)} poly(n))$.*

Proof. Let C be the set of chicken events of S, and suppose $r = |C|$. We simply "guess" which of the 2^r subsets of C to take. That is, for each subset $C' \subseteq C$, we find the maximum time gain achievable under the condition that the chicken events taken are exactly C', hence the 2^r factor in the complexity. For the rest of the proof, assume $C = \{c_0, c_1, \ldots, c_r, c_{r+1}\}$ is a set of chicken events such that $c_i <_S c_{i+1}$ for $0 \leq i \leq r$, each of which must be taken. For notational convenience, we have added chicken $c_0 = c_{r+1} = (0, 0)$, where c_0 (respectively c_{r+1}) is a chicken event that occurs before (resp. after) every event of S.

Let $d_1, \ldots, d_k$ be the possible damage values. For $i \in [k]$ and $j \in \{0, \ldots, r\}$, let n_{ij} be the number of events of damage value d_i that occur after chicken c_j, but before chicken c_{j+1}. Note that to obtain a solution, it suffices to know how many events of damage value d_i we take for each i and j. That is, let $(e^1_{ij}, \ldots, e^{n_{ij}}_{ij})$ be the events of damage value d_i that occur between chickens c_j and c_{j+1} in S, sorted in non-increasing order of time gain. If we know that, say, $x_{ij} \in \{0, \ldots, n_{ij}\}$ events of damage value d_i must be taken between chickens c_j and c_{j+1}, then we simply take the first x_{ij} events of maximum time gain, i.e. $e^1_{ij}, \ldots, e^{x_{ij}}_{ij}$. The time gain with respect to x_{ij} is $f_{ij}(x_{ij}) := \sum_{h=1}^{x_{ij}} t(e^h_{ij})$.

This lets us formulate an ILP with at most $2(r+1)k + r$ variables. For each $j \in [r]$, a variable h_j represents the player's HP right after taking the j-th chicken. We add the constant $h_0 := hp$ for convenience. For $i \in [k], j \in \{0, \ldots, r\}$, there is a variable x_{ij} for the number of events of damage value d_i to take between chicken c_j and c_{j+1}, and a variable g_{ij} for the time gained by events of damage value d_i within this range. The ILP is the following.

$$
\begin{array}{lll}
\text{maximize} & \displaystyle\sum_{i=1}^{k}\sum_{j=0}^{r} g_{ij} & \\
\text{subject to} & \displaystyle h_{j+1} \leq h_j - \sum_{i=1}^{k} x_{ij} d_i - d(c_{j+1}) & j \in \{0, \ldots, r-1\} \\
& h_j \leq hp & j \in \{1, \ldots, r\} \\
& \displaystyle h_j - \sum_{i=1}^{k} x_{ij} d_i > 0 & j \in \{0, \ldots, r\} \\
& g_{ij} \leq f_{ij}(x_{ij}) & i \in [k], j \in \{0, \ldots, r\} \\
& h_j \in \mathbb{N} & j \in \{1, \ldots, r\} \\
& x_{ij} \in \{0, \ldots, n_{ij}\}, g_{ij} \in \mathbb{N} & i \in [k], j \in \{0, \ldots, r\}
\end{array}
$$

The first constraint ensures that the player's HP after taking chicken c_{j+1} never exceeds the HP after taking chicken c_j and taking the damage boosts in-between. The second constraint ensures that we do not exceed the maximum hit points. The third constraint ensures that the player never dies. The fourth constraint bounds the total time gained by the damage boosts taken. The correctness of the above ILP is then straightforward to verify. The functions $f_{ij}(x_{ij})$ are however not guaranteed to be linear. But they are convex since they consist of the partial sums of a non-increasing sequence of integers. The authors of [15] have shown that the constraint $g_{ij} \leq f_{ij}(x_{ij})$ can easily be replaced (in polynomial time) by a set of linear constraints $g_{ij} \leq p_{ij}^{(\ell)}(x_{ij})$, for $\ell \in [n_{ij}]$. We refer the reader to [15, Lemma 2] for more details. The complexity follows from the aforementioned result of [23]. ◀

Damage boosting with lives

In the rest of this section, we consider the *death abuse* speedrunning strategy. In most games, when the player reaches 0 hit points, a life is lost and the player restarts with full health at the last predefined revival location traversed. We call such a location a *checkpoint*. The game is over once the player does not have any lives remaining. Death abusing is a common way of replenishing health, at the cost of having to re-traverse the portion of the stage from the last checkpoint to the location of death.

We modify the DAMAGE BOOSTING problem to incorporate death abuse as follows. A DAMAGE BOOSTING WITH LIVES instance is a 5-tuple (S, hp, ℓ, C, p) where $S = (e_1, \dots, e_n)$ is a sequence of events, hp is the maximum hit points, ℓ is the starting number of lives, $C \subseteq \{e_1, \dots, e_n\}$ is the set of checkpoints and $p : \{e_1, \dots, e_n\} \to \mathbb{N}$ is the *death penalty*, where $p(e_i)$ is the time lost by dying at event e_i and having to re-do the stage from the last checkpoint to e_i. For an event $e_i \in S$, let $c(e_i) \in C$ be the latest checkpoint of S that occurs before e_i. If the player reaches 0 HP at event e_i, the game restarts right before event $c(e_i)$ (so that taking the event $c(e_i)$ is possible after dying). If $c(e_i) = e_j$, we assume that $p(e_i) \geq \sum_{h=j}^{i} t(e_h)$, as otherwise it might be possible to gain time by dying.

A solution to S is a list of $k \leq \ell$ event subsequences $(S_1, \dots, S_k)$ that describes the events taken in each life used by the player. For $i \in [k-1]$, the i-th life of the player must end exactly after taking the last event of S_i. That is, $S_i = (e_1^i, \dots, e_r^i)$ must satisfy $h_{S_i}(e_j^i) > 0$ for each $j \in [r-1]$ and $h_{S_i}(e_r^i) \leq 0$. As for S_k, it must simply be valid, since the player's hit points can never go below 0 in the last life. Finally, we require that for $i \in \{2, \dots, k\}$, S_i starts at the checkpoint assigned to the event at which the player died in S_{i-1}. In other words, the first event of S_i must occur after the appropriate checkpoint, so that $c(tail(S_{i-1})) \leq_S head(S_i)$.

For $i < k$, the time gained $t(S_i)$ at life S_i is defined as before, except that $t(tail(S_i))$ is replaced by the penalty $p(tail(S_i))$ of dying at the last event. That is, $t(S_i) = \sum_{e \in S_i} t(e) - t(tail(S_i)) - p(tail(S_i))$. Our objective is to find a solution $(S_1, \dots, S_k)$ to S that maximizes $\sum_{i \in [k-1]} t(S_i) + \sum_{e \in S_k} t(e)$.

In this section, we show that having even only one life to spare removes the possibility of having a PTAS for DAMAGE BOOSTING WITH LIVES (unless P = NP). Despite this, we show that the problem still admits a pseudo-polynomial time algorithm. Beforehand, we state a simple approximabiltiy result.

▶ **Proposition 3.** *For any $\epsilon > 0$, DAMAGE BOOSTING WITH LIVES can be approximated within a factor $\frac{1}{\ell} - \epsilon$ in time $\mathcal{O}(n^3/\epsilon)$.*

Proof. Let (S, hp, ℓ, C, p) be a given instance of DAMAGE BOOSTING WITH LIVES, and let t be the maximum time gain achievable in stage S without losing a single life. By Theorem 1, t can be approximated within a factor $1 - \epsilon$ for any $\epsilon > 0$. Now, each life of the player can be used to gain at most t time, implying that at most ℓt time can be gained. The Lemma follows, since $(1-\epsilon)t \geq \frac{1-\epsilon}{\ell} \cdot \ell t \geq (\frac{1}{\ell} - \epsilon)\ell t$. ◀

We then present our inapproximability result. Note that this implies that the above approximation is tight in the case that the player has two lives.

▶ **Theorem 4.** *DAMAGE BOOSTING WITH LIVES is hard to approximate within a factor* $1/2$*, even if the player has two lives, and there is no chicken event.*

Proof. We show that having an algorithm with approximation factor $1/2$ or better would allow solving SUBSET SUM in polynomial time. Let (B, s) be a SUBSET SUM instance, with $B = \{b_1, \ldots, b_n\}$ a (multi)-set of n positive integers and s the target sum. Define a DAMAGE BOOSTING WITH LIVES instance (S, hp, ℓ, C, p) as follows. Put $hp = s + 1$ and $\ell = 2$. Also let $S = (e_1, \ldots, e_n, x, y)$. Here the e_i events correspond to the b_i integers, and x and y are two additional special events. For each $i \in [n]$, put $e_i = (b_i, b_i)$, and put $x = (1, 0)$, $y = (s, s-1)$. Set x as the only checkpoint, i.e. $C = \{x\}$. The only relevant death penalties are $p(x) = 0$ and $p(y) = 10s$. We show that if (B, s) is a YES instance, then it is possible to gain a total of $2s - 1$ time units, and if (B, s) is a NO instance, then at most $s - 1$ time units can be gained.

Suppose that (B, s) is a YES instance, and that there is a subset $B' = \{b_{i_1}, \ldots, b_{i_k}\}$ of B whose elements sum to s. Then the player can take the damage boosts $e_{i_1}, \ldots, e_{i_k}$ before arriving at x. At this point, s time units have been gained and s damage has been taken. Hence there is only 1 HP remaining. The player can take the 1 damage at event x, lose a life, and reappear at event x at full health with no time penalty. With this new life, the player then skips x, and takes the y damage boost, saving an additional $s - 1$ time units. The total time gain is $2s - 1$.

Now suppose that (B, s) is a NO instance. Assume that the player uses an optimal strategy on the constructed DAMAGE BOOSTING WITH LIVES instance. Consider the situation when the player arrives at event x, before deciding whether to take it (for the first time, if more than one). Let h_x and t_x be the remaining HP of the player and the time gained at this point, respectively. Observe that since all the e_i events have equal damage and time gain, we have $h_x = hp - t_x$. We must have $t_x \neq s$, since otherwise the events taken so far would provide a solution to the SUBSET SUM instance. Moreover, we cannot have $t_x > s$, since otherwise $h_x = hp - t_x = s + 1 - t_x \leq 0$, i.e. the player would have died before event x, and would have restarted at the beginning of the stage. Thus, $t_x < s$, and therefore $h_x > 1$. Since only 1 HP can be lost at event x, the player cannot die at event x. Thus the player arrives at y with a time gain of at most $s - 1$. Note that there is no point in dying at the y event, as the time lost is too high. Moreover, the only way the player can gain time from the y event is by being at full health. Since the player did not die at event x and there is no chicken, full health is only possible if the player has taken no damage boost before getting to y. It follows that there are then only two possibilities: if the player takes some events prior to x, he can save at most $t_x < s$ time units, and otherwise, he can skip every damage boost prior to x and save $s - 1$ time units by taking event y. We conclude that the time gain is at most $s - 1$.

Now, observe that if there is a factor $1/2$ approximation algorithm, it returns a time gain of at least $(2s - 1)/2 = s - 1/2$ on YES instances, and a time gain of at most $s - 1$ on NO instances. This gap can be used to distinguish between YES and NO instances. ◀

We do not know whether there exists a constant-factor approximation algorithm for DAMAGE BOOSTING WITH LIVES that holds for all values of ℓ. However, the problem does admit a pseudo-polynomial time algorithm. The dynamic programming is not as straightforward as the one for the knapsack, since the player can die and come back at checkpoints. The idea is to optimize the first life for each possible death, then the second life depending on the first, and so on.

▶ **Theorem 5.** *DAMAGE BOOSTING WITH LIVES can be solved in time* $\mathcal{O}(n^2 \cdot hp^2 \cdot \ell)$*.*

Proof. Let (S, hp, ℓ, C, p) be a given DAMAGE BOOSTING WITH LIVES instance, with $S = (e_1, \ldots, e_n)$. For simplicity, we assume that $e_1 = (0,0)$. Denote by $T(i,h,l)$ the maximum time gain that can be achieved by exiting event e_i (i.e. after deciding whether to take it or not) with exactly h hit points and l lives. Note that event e_i might have been visited in a previous life. Define $T(i,h,l) = -\infty$ if $h > hp$, $h \leq 0$, $l > \ell$ or $l \leq 0$. Our goal is to compute $\max_{1 \leq h \leq hp, 1 \leq l \leq \ell} T(n,h,l)$.

For $i = 1$, set $T(1, hp, \ell) = 0$ and $T(1,h,l) = -\infty$ whenever $h \neq hp$ or $l \neq \ell$ (we assume that we will never return to e_1 by losing a life, as this would be pointless).

For $i > 1$ such that $e_i \notin C$, note that we can only enter e_i through e_{i-1} with the same number of lives. If e_i is not a chicken event, we thus have

$$T(i,h,l) = \max\{T(i-1,h,l), T(i-1, h+d(e_i), l) + t(e_i)\}$$

(observe that invalid values of $h + d(e_i)$ yield a time gain of $-\infty$)

If e_i is a chicken event, the above recurrence applies unless taking event e_i would refill the player's health above hp. Thus $T(i, hp, l)$ is a special case, which we handle as follows (recall that $d(e_i)$ and $t(e_i)$ are now negative):

$$T(i,hp,l) = \max\left\{T(i-1,hp,l), \max_{hp+d(e_i) \leq d \leq hp}\{T(i-1, hp-d, l)\} + t(e_i)\right\}$$

Now suppose that $i > 1$ is such that $e_i \in C$. We can either enter e_i through e_{i-1} with the same number of lives, or through some e_j with $j > i$ by dying while having $l+1$ lives. In the latter case, we must enter e_i with health equal to hp. Therefore, if $h \notin \{hp, hp - d(e_i)\}$, it is impossible to enter e_i by dying and exiting with exactly h hit points. Hence, if $h \notin \{hp, hp - d(e_i)\}$, the above recurrence from the $i > 1$ case applies. Moreover, if $l = \ell$, the player cannot have died yet and the same recurrence also applies. Assume that $h \in \{hp, hp - d(e_i)\}$ and $l < \ell$. We must compute a temporary value for $T(i,h,l)$. Let e_k be the latest event that leads to checkpoint e_i upon death. That is, $c(e_k) = e_i$ but either $k = n$ or $c(e_{k+1}) \neq e_i$. Define

$$D_{i,l} = \max_{i \leq j \leq k}\left\{\max_{h' \leq d(e_j)}\{T(j,h',l+1) - p(e_j)\}\right\}$$

which is the maximum time gain achievable by losing the player's $(l+1)$-th life and respawning at e_i. Then it follows that

$$T(i,hp,l) = \max\{T(i-1,hp,l), D_{i,l}\}$$

if e_i is not a chicken event. If e_i is a chicken event, then similarly as we did above,

$$T(i,hp,l) = \max\left\{T(i-1,hp,l), \max_{hp+d(e_i) \leq d \leq hp}\{T(i-1,hp-d,l)\} + t(e_i), D_{i,l}\right\}$$

Finally, for the case $h = hp - d(e_i)$, we have

$$T(i, hp - d(e_i), l) = \max\{T(i-1, hp - d(e_i), l), D_{i,l} + t(e_i)\}$$

Note that to compute $T(i,h,l)$, one only needs values of $T(j,h',l')$ with either $j < i$ and $l' = l$, or with $l' = l+1$. It is not difficult to see that one can compute the $T(i,h,l)$ values in decreasing order of values of l, starting at $l = \ell$, and in increasing order of i. Each $T(i,h,l)$ value depends on at most $\mathcal{O}(n \cdot hp)$ values. There are $(n+2) \cdot |hp| \cdot |\ell|$ possible $T(i,h,l)$ values, resulting in a $\mathcal{O}(n^2 \cdot (hp)^2 \cdot \ell)$ time algorithm. ◀

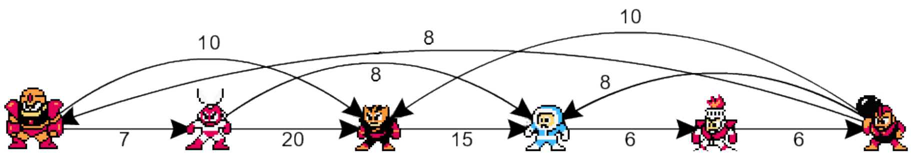

Figure 4 The dependency graph for the game Mega Man (with approximate time gains in seconds according to [1]), where the only event considered is defeating the boss.

4 Routing

We now turn to the problem of *routing*, in which the player may visit a set of locations or stages in any order. Clearing a stage yields a new weapon to the player. Each stage has a set of time-saving events, and each weapon can be used to gain some amount of time in an event. The time saved on an event depends on the best weapon available. Figure 4 represents this notion in Mega Man as a weighted directed graph. For instance, defeating Guts Man first (far left) allows saving 7 seconds against Cut Man (second), and 8 seconds could be gained by defeating Bomb Man (far right) before Guts Man.

In this section, a *game* is a set of stages $\mathbb{S} = \{S_1, \ldots, S_n\}$. A *stage* $S_i = \{e_1, \ldots, e_k\}$ is a set of *events*, where here an event $e_j : \mathbb{S} \to \mathbb{N}$ is a function mapping each stage to an integer. The event e_j is interpreted as follows: if stage S_i is cleared, then a time of $e_j(S_i)$ can be saved while going through e_j using the weapon gained from S_i. Let $C \subseteq \mathbb{S}$ and let e be an event. We will write $e(C) = \max_{S \in C} e(S)$. That is, if C is the set of cleared stages, we will assume that event e will be cleared using the best option available. Given C, the time gained in a stage S becomes $t(S, C) := \sum_{e \in S} e(C)$.

In the ROUTING problem, we are given a set of stages $\mathbb{S} = \{S_1, \ldots, S_n\}$. The objective is to find a linear ordering π of $\mathbb{S}$ such that $\sum_{i \in [n]} t(S_i, \{S_j : S_j <_\pi S_i\})$ is maximum. Later on, we shall consider the minimization version of ROUTING.

We define the notion of a *dependency digraph* $D(\mathbb{S})$ for a set of stages $\mathbb{S}$. The digraph $D(\mathbb{S}) = (\mathbb{S}, A, w)$ has one vertex for each stage, and for every ordered pair (i, j), an arc from S_i to S_j of weight $w(S_i, S_j) = \sum_{e \in S_j} e(S_i)$. The *underlying undirected graph* of $D(\mathbb{S})$ is the graph obtained by removing the arcs of weight 0, ignoring the other weights and the direction of the arcs.

We start with two easy special cases. The first case is when each stage contains only one event, which could for example correspond to the case in which we only consider the fastest way to defeat all bosses. This reduces to finding a *maximum weight branching* in $D(\mathbb{S})$, where a branching of a digraph D is an acyclic subdigraph of D in which every vertex has in-degree 0 or 1. The second case is when each event depends on only one stage. The Routing problem then becomes equivalent to finding a maximum weight directed acyclic sub-digraph of $D(\mathbb{S})$. This is the *maximum weight sub-DAG* problem, the maximization version of the feedback arc set problem.

▶ **Theorem 6.** *The following properties of Routing hold:*

1. *If each stage contains a single event, ROUTING can be solved in time $\mathcal{O}(|A| + |\mathbb{S}| \log |\mathbb{S}|)$.*
2. *If, for each event e, there is only one $S_i \in \mathbb{S}$ such that $e(S_i) > 0$, then ROUTING is equivalent to the maximum weight sub-DAG problem on $D(\mathbb{S})$.*

Proof. (1) For a stage S_i, denote by e_i the single event of S_i. Given an ordering π of $\mathbb{S}$ and a stage $S_i \neq \mathit{tail}(\pi)$, denote by $p_\pi(S_i)$ the stage prior to S_i that allows a maximum time gain on e_i, breaking ties arbitrarily. That is, $p_\pi(S_i) = \arg\max_{S_j <_\pi S_i} e_i(S_j)$.

Observe that for any ordering π and any stage $S_i \neq tail(\pi)$, because S_i has only one event there is at most one stage prior to S_i that can be useful to clear it, namely $p_\pi(S_i)$. Recalling that $D(\mathbb{S}) = (\mathbb{S}, A, w)$, the time gain for a given π is $t = \sum_{S_i \in \mathbb{S}} w(p_\pi(S_i), S_i)$. Consider the set of arcs $A' = \{(p_\pi(S_i), S_i) : 1 < i \leq n \text{ and } e_i(p_\pi(S_i)) > 0\}$. Then the subdigraph of $D(\mathbb{S})$ formed by the arc set A' contains no directed cycle, and each vertex has at most one incoming arc, with the exception of the first the vertex of π which has none. Thus A' forms a branching, and its weight is t. Conversely, let B be a branching of $D(\mathbb{S})$ with arc set A'. Then it is not hard to see that B can be converted to an ordering π of $\mathbb{S}$ such that the total time gained is $\sum_{(u,v)\in A'} w(u, v)$. Indeed, as B is acyclic, a topological sorting of B yield a linear ordering of $\mathbb{S}$ in which each event e_{S_i} can be completed using the in-neighbor of S_i in A' (if any). A maximum weight branching can be found in time $\mathcal{O}(|A| + |\mathbb{S}| \log |\mathbb{S}|)$ by reduction to the maximum weight spanning arborescence problem (see e.g. [10, Chapter 6]), and using Gabow & al.'s algorithm [18].

(2) If every event depends on exactly one stage, we show that ROUTING and maximum weight sub-DAG reduce to one another with the same optimality value. We start by reducing ROUTING to maximum weight sub-DAG. Consider the $D(\mathbb{S}) = (\mathbb{S}, A, w)$ digraph. Because each $e \in S_j$ depends only on one stage, $w(S_i, S_j)$ corresponds exactly to the time gain contribution of S_i to stage S_j if S_i is completed before S_j (which might not be the case if an event could be completed by more than one stage). Thus given an ordering π of $\mathbb{S}$, the total time gain is $t = \sum_{S_i <_\pi S_j} w(S_i, S_j)$ (where $w(S_i, S_j) = 0$ if $(S_i, S_j) \notin A$). Moreover, the arcs $\{(S_i, S_j) \in A : S_i <_\pi S_j\}$ cannot form a cycle in $D(\mathbb{S})$. It follows that an ordering π of time gain t can be used to find a sub-DAG of $D(\mathbb{S})$ of weight t. Conversely, a topological sorting of a sub-DAG of $D(\mathbb{S})$ with total weight t gives an ordering of the stages with total time gain t.

The reduction from the maximum weight sub-DAG problem to the routing problem goes along the same lines. Given a maximum weight sub-DAG instance $H = (V, A, w)$, it suffices to create a stage S_u for each $u \in V$, and add one event e^u_v in S_u for each v such that $(v, u) \in A$. We put $e^u_v(S_v) = w(v, u)$. It is easy to see that a total time of t can be gained if and only if H has a sub-DAG of weight t. ◀

The above implies that every known hardness result for the maximum weight sub-DAG problem transfers to ROUTING. In particular, ROUTING is NP-hard even if the maximum degree of the $D(\mathbb{S})$ is 4 (this follows from the hardness of vertex cover in cubic graphs [3]). Also, the maximum weight sub-DAG problem cannot be approximated within a ratio better than 1/2, assuming the Unique Games Conjecture [19]. On the positive side, it is trivial to attain this bound, just as in the maximum weight sub-DAG problem: take any ordering π. Either π or its reverse will attain 1/2 of the maximum possible time save.

▶ **Proposition 7.** *ROUTING admits a factor* $1/2$ *approximation algorithm.*

Proof. Note that $\sum_{i\in[n]} \sum_{e\in S_i} e(\mathbb{S})$ is an obvious upper bound on the maximum time gain achievable. Pick a random ordering $(S_1, \ldots, S_n)$ of $\mathbb{S}$, and let $(S_n, \ldots, S_1)$ be the reverse ordering. One of these two must achieve a time gain of $e(\mathbb{S})$ for at least half the events e that are in $\mathbb{S}$. ◀

Minimizing time loss

We now turn to the minimization version of the ROUTING problem. That is, consider the upper bound $\mu := \sum_{i\in[n]} \sum_{e\in S_i} e(\mathbb{S})$ on the possible time gain. Ideally, one would like to get as close as possible to μ, which amounts to finding a time gain t that minimizes $\mu - t$. Given an ordering π of $\mathbb{S}$, denote by $cost(\mathbb{S}, \pi) := \mu - \sum_{i \in [n]} t(S_i, \{S_j : S_j <_\pi S_i\})$.

We define the MIN-ROUTING-LOSS as follows: given a set of n stages $\mathbb{S}$, find a linear ordering π of $\mathbb{S}$ that minimizes $cost(\mathbb{S}, \pi)$.

By Theorem 6, this is at least as hard as the *feedback arc set* (FAS) problem, where the goal is to delete a set of arcs of minimum weight from a digraph to obtain a DAG (these deletions correspond to time losses in $D(\mathbb{S})$). FAS is APX-hard [20], but determining if there is a constant factor approximation appears to be open. A factor $\mathcal{O}(\log n \log \log n)$ approximation algorithm is presented in [16], but does not appear to apply to MIN-ROUTING-LOSS.

We will show that MIN-ROUTING-LOSS cannot be approximated with a ratio better than $\mathcal{O}(\log n)$. As for parameterized complexity, FAS is known to be FPT in k, the weight of the edges to remove (assuming weights in $poly(n)$) [9]. FAS is also known to be FPT in the treewidth of the underlying undirected graph [6]. As we show here, both parameters are not applicable to MIN-ROUTING-LOSS.

▶ **Theorem 8.** *MIN-ROUTING-LOSS is W[2]-hard with respect to the time loss k and hard to approximate within a factor $\mathcal{O}(\log n)$. This holds even on instances in which the underlying undirected graph of $D(\mathbb{S})$ is a tree and only one stage has more than one event.*

Proof. We reduce from DOMINATING SET, which is known to be W[2]-hard for parameter k, the number of vertices in the dominating set [14]. Let (G, k) be an instance of DOMINATING SET. Denote $V(G) = \{v_1, \ldots, v_n\}$. Create a set of stages $\mathbb{S} = \{S_1, \ldots, S_n, X\}$. For $i \in [n]$, stage S_i has only one event e_i, whereas X has n events $\{x_1, \ldots, x_n\}$. For each edge $v_i v_j \in E(G)$, set $x_j(S_i)$ very high, say $x_j(S_i) = kn^{10}$. Also set $x_j(S_j) = kn^{10}$ for all $j \in [n]$. Then for each $i \in [n]$, set $e_i(X) = 1$. All other event completion times are set to 0. Note that the upper time bound on $\mathbb{S}$ is $\mu = n + (kn^{10})n$. We show that G has a dominating set of size at most k if and only if a time gain of at least $\mu - k$ is possible.

Let $B = \{v_{i_1}, \ldots, v_{i_k}\}$ be a dominating set of G of size k, and denote $\{v_{i_{k+1}}, \ldots, v_{i_n}\} = V(G) \setminus B$. Order the stages of $\mathbb{S}$ as follows: $\pi = (S_{i_1}, \ldots, S_{i_k}, X, S_{i_{k+1}}, \ldots, S_{i_n})$. For any $x_j \in X$, either $v_j \in B$ or there is some $v_i \in B$ such that $v_i v_j \in E(G)$. Since one of $S_j <_\pi X$ or $S_i <_\pi X$ holds, event x_j can be cleared with time gain kn^{10}. Also, every event in $S_{i_{k+1}}, \ldots, S_{i_n}$ can be cleared with a time gain 1 using stage X. Only the events in stages $S_{i_1}, \ldots, S_{i_k}$ do not yield a time gain, and the total time gain is therefore $\mu - k$.

Conversely, suppose that there is an ordering π of $\mathbb{S}$ that achieves a time gain of at least $\mu - k$. For this to be possible, every event of X must be cleared with a time gain kn^{10}. Consider the set $B = \{S_{i_1}, \ldots, S_{i_h}\}$ that precedes X in π. None of these stages can yield a time gain, which implies $h \leq k$. Moreover, B must be a dominating set, for if not, there is an event $x_j \in X$ that cannot be cleared with a time gain of kn^{10}.

As for the inapproximability result, DOMINATING SET is hard to approximate within a factor $\mathcal{O}(\log n)$ (see [4]). It is not hard to see that the above reduction is approximation preserving: from a dominating set of size k, one can obtain a time loss of at most k and vice-versa. As the number of stages in $\mathbb{S}$ is $n + 1$, the $\mathcal{O}(\log n)$ inapproximability follows. ◀

Observe that in addition to treewidth, the number of stages with more than one event is also not an option for parameterization, as well as the maximum degree of $D(\mathbb{S})$ (due to Theorem 6 and the remark after). In the rest of this section, we show that Routing is FPT when combining the treewidth and maximum in-degree parameters.

Parameterization by treewidth and maximum in-degree

In this section, we assume that the in-degree of a vertex in $D(\mathbb{S})$ is bounded by d and the treewidth of the underlying undirected graph of $D(\mathbb{S})$ is bounded by t. We devise a more or less standard dynamic programming algorithm on the tree decomposition of $D(\mathbb{S})$. We introduce the essential notions here, and refer the reader to [14, 12] for more details.

A *tree decomposition* of a graph $G = (V, E)$ is a tree T in which each node x is associated with a *bag* $B_x \subseteq V$ such that $\bigcup_{x \in V} B_x = V$. Moreover, the two following properties must hold: (1) for any $uv \in E$, there is some $x \in V(T)$ such that $u, v \in B_x$, and (2) for any $v \in V$, the set $\{x \in V(T) : v \in B_x\}$ induces a connected component of T. The *width* of T is the size of the largest bag of T minus 1, and the *treewidth* of G is the minimum width of a tree decomposition of G.

A tree decomposition T for G is *nice* if each $x \in V(T)$ is of one of the following types:

- *Leaf node*: x is a leaf of T and $B_x = \emptyset$.
- *Introduce node*: x has exactly one child y and $B_x = B_y \cup \{v\}$ for some $v \in V(G)$.
- *Forget node*: x has exactly one child y and $B_x = B_y \setminus \{v\}$ for some $v \in V(G)$.
- *Join node*: x has exactly two children y, z and $B_x = B_y = B_z$.

We also assume that T is rooted at a vertex r such that $B_r = \emptyset$. The root defines the ancestor/descendant relationship between nodes of T. It is well-known that a nice tree decomposition T' of width t can be constructed from a tree decomposition T of width t in polynomial time (see [12, 14]).

The routing algorithm

Assume that we have constructed a nice tree decomposition T from $D(\mathbb{S}) = (V, A, w)$. For convenience, we shall treat stages of $\mathbb{S}$ as vertices (hence, each $v \in V$ is a set of events). For $v \in V$, denote by $N^-(v) = \{u \in V : (u, v) \in A\}$ and $N^-[v] = N^-(v) \cup \{v\}$. Under our assumptions, $|N^-(v)| \leq d$ for all $v \in V$. Roughly speaking, at each node $x \in V(T)$, we would like to "try" each ordering of B_x and compute a time cost for each stage $v \in B_x$ based on the children of x. This is essentially the idea in the bounded treewidth FPT algorithm for feedback arc set [6]. This however does not work directly, as the cost of a stage $v \in B_x$ depends on $N^-(v)$, which may or may not be included in B_x. To solve this problem, we also include all the in-neighbors of the stages in B_x in the set of orderings to consider. One way to do this would be to consider all orderings of $\bigcup_{v \in B_x} N^-[v]$ at every bag B_x and assign a cost to every vertex in B_x or in a bag below. This would lead to a relatively simple $\mathcal{O}((dt)!poly(n))$ algorithm. However, this complexity can be improved (at the expense of more technicality) by considering, instead of every permutation of $\bigcup_{v \in B_x} N^-[v]$, only the subsets of $N^-(v)$ that occur before v for each $v \in B_x$.

To formalize this notion, let $P = \{\pi_1, \ldots, \pi_s\}$ be a set of orderings of (possible different) subsets of V. We say that P is *realizable* if there exists an ordering π of V such that for each $i \in [s]$, $u <_{\pi_i} v$ implies $u <_\pi v$. We then say that π *realizes* P (or for short, π realizes π' if $P = \{\pi'\}$). Note that the existence of π can be verified in polynomial time.

Let V_x be the subset of vertices of V appearing in the bags under x, i.e. $v \in V_x$ if and only if x has a descendant y such that $v \in B_y$ (noting that x is a descendant of itself). For $x \in V(T)$, we denote by $\Pi(x)$ the set of all $|B_x|!$ possible orderings of B_x (with $\Pi(x) = \{()\}$ if $B_x = \emptyset$). Denote by $\Lambda(x)$ the set of all combinations of subsets of in-neighbors of vertices in B_x. That is, if $B_x = \{v_1, \ldots, v_s\}$ with $1 \leq s \leq t$, then

$$\Lambda(x) = \mathcal{P}(N^-(v_1)) \times \ldots \times \mathcal{P}(N^-(v_s))$$

where $\mathcal{P}(X)$ denotes the powerset of X. Let $\Lambda(x) = \{()\}$ contain the empty sequence if $B_x = \emptyset$. Observe that $|\Lambda(x)| = \mathcal{O}(2^{dt})$. For $P_x = (P_1, \ldots, P_s) \in \Lambda(x)$, we interpret P_i as "all elements of P_i occur before v_i, and those of $N^-(v_i) \setminus P_i$ occur after v_i". We thus denote the set of two-elements orderings implied by P_x by

$$s(P_x) = \bigcup_{v_i \in B_x} \{(u, v_i) : u \in P_i\} \cup \{(v_i, u) : u \in N^-(v_i) \setminus P_i\}$$

We now define a time cost $D(x, \mu_x, P_x)$ over all $x \in V(T)$, $\mu_x \in \Pi(x)$ and $P_x = (P_1, \ldots, P_s) \in \Lambda(x)$. Given an ordering π of V and $v \in V$, let $cost(v, \pi) = \sum_{e \in v_i}(e(V) - e(\{u : u <_\pi v\}))$ be the time lost in stage v. Let $V'_x = V_x \cup \bigcup_{v \in B_x} N^-(v)$. Then

$$D(x, \mu_x, P_x) := \min\{\sum_{v \in V_x} cost(v, \pi) : \pi \text{ is an ordering of } V'_x \text{ that realizes } \{\mu_x\} \cup s(P_x)\}$$

In words, $D(x, \mu_x, P_x)$ is the minimum cost for the set of stages in V_x in an ordering of V'_x, with the obligation of using the partial orderings prescribed by μ_x and P_x. If r is the root of T, our goal is to compute $D(r, (), ())$ (recall that $B_r = \emptyset$). For $v \in V$ and $P \subseteq N^-(v)$, let $cost(v_i, P) = \sum_{e \in v_i}(e(V) - e(P))$ the time lost at stage v_i if precisely the elements of P occur before v. We claim that $D(x, \mu_x, P_x)$ can be computed as follows.

- If x is a leaf node, then V_x and B_x are empty and we simply set $D(x, (), ()) = 0$;
- If x is an introduce node with child y, let v_i be the new node in B_x and P_i be the subset of $N^-(v_i)$ present in P_x. Then

$$\begin{aligned} D(x, \mu_x, P_x) = \min\{&D(y, \mu_y, P_y) : \mu_y \in \Pi(y), P_y \in \Lambda(y) \text{ and} \\ &s(P_x) \cup s(P_y) \cup \{\mu_x, \mu_y\} \text{ is realizable}\} + cost(v_i, P_i) \end{aligned}$$

- If x is a forget node with child y, then

$$\begin{aligned} D(x, \mu_x, P_x) = \min\{&D(y, \mu_y, P_y) : \mu_y \in \Pi(y), P_y \in \Lambda(y) \text{ and} \\ &s(P_x) \cup s(P_y) \cup \{\mu_x, \mu_y\} \text{ is realizable}\} \end{aligned}$$

- If x is a join node with children y and z, then

$$D(x, \mu_x, P_x) = D(y, \mu_x, P_x) + D(z, \mu_x, P_x) - \sum_{v_i \in B_x} cost(v_i, P_i)$$

  where P_i is the ordering of P_x for $N^-[v_i]$, for each $v_i \in B_x$.

The above yield the following result. The main difficulty is to show that an ordering at node x can be obtained from the ordering of its child/children.

▶ **Theorem 9.** *ROUTING can be solved in time $\mathcal{O}(2^{t(d+\log t)}(2^d + md) \cdot nt)$, where m is the maximum number of events in a stage.*

Proof. We prove the complexity first, then proceed with the correctness of the dynamic programming recurrences. There are $\mathcal{O}(nt!2^{dt}) = \mathcal{O}(n2^{t \log t}2^{dt}) = \mathcal{O}(n2^{t(d+\log t)})$ possible x, μ_x and P_x combinations for the values of $D(x, \mu_x, P_x)$. To compute a specific $D(x, \mu_x, P_x)$, in the worst case we need to consider all the possible $D(y, \mu_y, P_y)$ values for the child y of x, in the case of introduce and forget nodes. However in these situations, μ_x and μ_y differ only by one element, and so given μ_x, there are only $\mathcal{O}(t)$ orderings of B_y such that $\{\mu_x, \mu_y\}$ are realizable. Similarly, $s(P_x) \cup s(P_y)$ are realizable only if they have the same sets of in-neighbors for each $v \in B_x \cap B_y$. Therefore, only one subset of $N^-(v_i)$ can differ between P_x and P_y, where here v_i is the introduced or forgotten vertex. It follows that only $\mathcal{O}(t2^d)$ combinations of μ_y and P_y need to be checked from $D(y, \mu_y, P_x)$. One still needs to check whether $\mu_x, \mu_y, s(P_x)$ and $s(P_y)$ are realizable. This can easily be done in time $O(n)$, for instance by constructing the directed graph on vertex set V and adding an arc from u to v whenever u is immediately before v in an ordering of $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$. This graph has $O(td)$ arcs and it suffices to check that it is acyclic. In the case of introduce nodes, the value of $cost(v_i, P_i)$ can be computed in time $\mathcal{O}(md)$. At most t such values need to be computed. It follows that the total complexity is $\mathcal{O}(n2^{t(d+\log t)}(t2^d + tmd))$.

It remains to show that our recurrences for $D(x, \mu_x, P_x)$ are correct, which we do by induction over the nodes of T from the leaves to the root. As a base case, this is true for the leaves, so assume $x \in V(T)$ is an internal node of T. For the remainder of the proof, given an ordering π of some set X, let $\pi|X'$ denote the ordering on $X' \subseteq X$ of π restricted to X' (i.e. $\pi|X'$ is the unique ordering of X' such that π realizes $\pi|X'$).

Before proceeding with the correctness, we first claim that for any child y of $x \in V(T)$, $V'_y \subseteq V'_x$. Suppose this is not the case. Because $V_y \subseteq V_x$, by the definition of V'_y and V'_x, there must be some $v \in B_y \setminus B_x$ and $u \in N^-(v)$ such that $u \notin V'_x$. In particular, $u \notin V_x$. Since T is a tree decomposition, there must be a node $z \in V(T)$ such that $u, v \in B_z$. But since $u \notin V_x$, z cannot be in the subtree rooted at x, as otherwise $u \in V'_x$ would hold. This is a contradiction, as this implies that the vertices with bags containing v do not form a connected component of T, which proves our claim.

We now treat each possible node type separately to prove our recurrences correct.

Introduce nodes. Suppose x is an introduce node with child y and new vertex v_i. For each $v_j \in B_x$, let $P_j \in P_x$ be the subset of $N^-(v_j)$ for v_j. Let $u \in N^-(v_i)$. It is straightforward to check that $u \notin V_y \setminus B_x$, since a bag of T must contain u and v, v_i was introduced in bag B_x and u is not in B_x. Similarly, let $u \in V_y \setminus B_y$. One can check that $N^-(u) \subseteq V_y$, as otherwise a neighbor of u outside of V_y would lead to the same type of contradiction.

We first show that $D(x, \mu_x, P_x) \geq \min_{y,\mu_y,P_y}\{D(y, \mu_y, P_y) + cost(v_i, P_i)\}$, where P_i is the subset of $N^-(v_i)$ for v_i in P_x, and $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ are realizable. Let π be an ordering of V'_x such that $\sum_{v \in V_x} cost(v, \pi) = D(x, \mu_x, P_x)$ and such that π realizes μ_x and $s(P_x)$. Let $\pi_y := \pi|V'_y$ (note that π_y is well-defined since $V'_y \subseteq V'_x$), and let $\mu_y = \pi_y|B_y$ and $P_y \in \Lambda(y)$ be such that π_y realizes $s(P_y)$. Clearly, $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ is realizable (as witnessed by π). Moreover, $\sum_{w \in V_y} cost(w, \pi_y) \geq D(y, \mu_y, P_y)$, by the definition of $D(y, \mu_y, P_y)$. As we also have $cost(v_i, \pi) = cost(v_i, P_i)$, it follows that

$$\sum_{v \in V_x} cost(v, \pi) \geq D(y, \mu_y, P_y) + cost(v_i, P_i) \geq \min_{y', \mu'_y, P'_y}\{D(y', \mu'_y, P'_y)\} + cost(v_i, P_i)$$

as desired.

As for the converse bound, take any ordering π_y of V'_y of cost $D(y, \mu_y, P_y)$ that realizes μ_y and P_y on B_y such that $\{\mu_x, \mu_y\} \cup s(P_x) \cup s(P_y)$ is realizable. We start from π_y and construct an ordering of V'_x. If v_i is not in π_y, insert v_i in π_y anywhere so that it realizes μ_x (this is possible since π_y realizes $\mu_y = \mu_x|(B_x \setminus \{v_i\})$. Then let $\pi'_y := \pi_y|(V_y \cup \{v_i\})$. Note that since any $u \in V_y \setminus B_x$ has no in-neighbor outside of V_y, the cost of u is entirely defined by π'_y, and hence unchanged from π_y. We now want to insert the elements of $V'_x \setminus (V_y \cup \{v_i\})$ so as to realize $s(P_x)$. Let $\hat{\pi}$ be any ordering of $\bigcup_{v_i \in B_x} N^-[v_i]$ that realizes $s(P_x) \cup s(P_y) \cup \{\mu_x\}$, and let $\pi' := \hat{\pi}|((V'_x \setminus V_y) \cup B_x)$. Since the elements of π'_y and π' coincide only on B_x and both realize μ_x, it is easy to see that there is some ordering π that realizes π'_y and π'. Note that π is an ordering of V'_x. Moreover, π realizes μ_x, and therefore also realizes μ_y.

We must now argue that π realizes $s(P_x)$ (which also implies that π realizes $s(P_y)$). First consider $P_j \in P_x$, where $i \neq j$ so that $v_j \in B_x \cap B_y$ and P_j is the subset of $N^-(v_j)$ for v_j in P_x. Let $u \in P_j$. If $u \in V_y$, then $u <_\pi v_j$ since π realizes π'_y (which is a subordering of π_y that realizes $s(P_y)$). If $u \in V'_x \setminus V_y$, then $u <_\pi v_j$ because π realizes π' (which is a subordering of $\hat{\pi}$ that realizes $s(P_x)$). By a similar argument, one can check that all $u \in N^-(v_j) \setminus P_j$ occur after v_j. Now consider $P_i \in P_x$, the subset of $N^-(v_i)$ for v_i. Let $u \in P_i$, and recall that $u \notin V_y \setminus B_x$. If $u \in B_x$, then $u <_\pi v_i$ because π realizes μ_x (and we may assume $u <_{\mu_x} v_i$ as otherwise μ_x and $s(P_x)$ are not possibly realizable together). If $u \in V'_x \setminus B_x$, then $u <_{\mu_x} v_i$ because π realizes π', as above. The case $u \in N^-(v_i) \setminus P_i$ can be verified in a similar manner.

Since the costs of the $v \in V_y$ are unchanged from π_y to π, it follows that $D(x, \mu_x, P_x) \leq \sum_{v \in V_x} cost(v, \pi) = \sum_{v \in V_y} cost(v, \pi_y) + cost(v_i, \pi_y) = D(y, \mu_y, P_y) + cost(v_i, P_i)$, which yields the complementary bound.

Forget nodes. Suppose that x is a forget node with child y. In this case, $V_x = V_y$ and $V'_x = V'_y$. It is not hard to see that it suffices to inherit the time costs computed at the y node.

Join nodes. Suppose x is a join node with children y, z, in which case $B_x = B_y = B_z$. Denote $B_x = \{v_1, \ldots, v_s\}$. For each $v_i \in B_x$, let $P_i \in P_x$ be the subset of $N^-(v_i)$ for v_i. Note that if $v \in V_y \setminus B_x$, then $v \notin V_z$ (otherwise, the bags containing v would not be connected). Similarly, if $v \in V_z \setminus B_x$ then $v \notin V_y$. Hence $V_y \cap V_z = B_x$. Let π be an ordering of V'_x that realizes μ_x and $s(P_x)$ of cost $D(x, \mu_x, P_x)$. Let $\pi_y := \pi | V'_y$ and $\pi_z := \pi | V'_z$. Note that both π_y and π_z must realize μ_x and $s(P_x)$. Hence $\sum_{v \in V_y} cost(v, \pi_y) \geq D(y, \mu_x, P_x)$ and $\sum_{v \in V_z} cost(v, \pi_z) \geq D(z, \mu_x, P_x)$. Since $V_y \cap V_z = B_x$, it follows that $D(x, \mu_x, P_x) \geq D(y, \mu_x, P_x) + D(z, \mu_z, P_x) - \sum_{v_i \in B_x} cost(v_i, P_i)$.

For the converse bound, let π_y (respectively π_z) be orderings of V'_y (V'_z) that realize μ_x and $s(P_x)$ of cost $D(y, \mu_x, P_x)$ ($D(z, \mu_x, P_x)$). Note that if $u \in V_z \setminus B_x$, then $N^-(u) \subseteq V_z$ (using tree decomposition arguments) and if $u \in V_y \setminus B_x$, then $N^-(u) \subseteq V_y$. Let $\pi'_y := \pi_y | (V'_y \setminus V_z) \cup B_x$. Then for all $u \in V_y \setminus B_x$, the cost of u is unchanged from π_y to π'_y. Then, let $\pi'_z := \pi_z | V_z$, with the same remark on $u \in V_z \setminus B_x$. Let π be an ordering of V'_x that realizes π'_y and π'_z. Note that π exists, since π'_y and π'_z coincide only on B_x and both realize μ_x.

We argue that π realizes $s(P_x)$. Let $v_i \in B_x$ and $u \in P_i$. If $u \in B_x$, then $u <_\pi v_i$ because π realizes μ_x (which, as we may assume, is realizable with $s(P_x)$). If $u \in V'_y \setminus V_z$, then $u <_\pi v_i$ because π realizes π'_y (which is a subordering of π_y which realizes $s(P_x)$). Finally if $u \in V_z \setminus B_x$, then $u <_\pi v_i$ because π realizes π'_z (which is a subordering of π_z which realizes $s(P_x)$). A similar argument shows that $v_i <_\pi u$ for $u \in N^-(v_i) \setminus P_i$.

It remains to argue that $D(x, \mu_x, P_x) \leq \sum_{v \in V_x} cost(v, \pi) = D(x, \mu_x, P_x) + D(y, \mu_x, P_x) - \sum_{v \in B_x} cost(v, P_i)$. For $v \in V_y \setminus B_x$ or $v \in V_z \setminus B_x$, the cost is unchanged from π_y and π_z, respectively, as we mentioned above. If $v \in B_x$, the cost is the same as in π_y and π_z, since π, π_y and π_z all realize $s(P_x)$. Therefore, $\sum_{v \in V_x} cost(v, \pi) = \sum_{v \in V_y} cost(v, \pi_y) + \sum_{v \in V_z} cost(v, \pi_z) - \sum_{v_i \in B_x} cost(v, \pi)$ (as we double-counted the B_x elements). The correctness follows, since $\sum_{v \in V_y} cost(v, \pi_y) = D(y, \mu_x, B_x)$ and $\sum_{v \in V_z} cost(v, \pi_z) = D(z, \mu_x, B_x)$. ◀

5 Conclusion

The hardness results presented in this work apply to any game that allows damage boosting or routing in its speedrunning mechanics. However, the positive results ignore other possible aspects of the game, which could be incorporated in our problem models in the future. For instance, some games may offer multiple possible paths that in turn offer different sets of events. Also, role-playing games such as *Final Fantasy* are notorious for the calculations needed for manipulating the game's random number generator, which leads to other optimization problems. We also leave the problems of approximating damage boosting with lives and minimum-loss routing open, as well as determining their precise FPT status.

References

1 Classic damage data charts mega man 1 damage data chart. http://megaman.wikia.com/wiki/Mega_Man_1_Damage_Data_Chart. Accessed: 2018-02-21.

2 Games done quick. https://gamesdonequick.com/. Accessed: 2018-02-21.

3 Paola Alimonti and Viggo Kann. Hardness of approximating problems on cubic graphs. In *Italian Conference on Algorithms and Complexity*, pages 288–298. Springer, 1997.

4 Noga Alon, Dana Moshkovitz, and Shmuel Safra. Algorithmic construction of sets for k-restrictions. *ACM Transactions on Algorithms (TALG)*, 2(2):153–177, 2006.

5 Greg Aloupis, Erik D Demaine, Alan Guo, and Giovanni Viglietta. Classic nintendo games are (computationally) hard. *Theoretical Computer Science*, 586:135–160, 2015.

6 Marthe Bonamy, Lukasz Kowalik, Jesper Nederlof, Michal Pilipczuk, Arkadiusz Socala, and Marcin Wrochna. On directed feedback vertex set parameterized by treewidth. *arXiv preprint arXiv:1707.01470*, 2017.

7 Jeffrey Bosboom, Erik D Demaine, Adam Hesterberg, Jayson Lynch, and Erik Waingarten. Mario kart is hard. In *Japanese Conference on Discrete and Computational Geometry and Graphs*, pages 49–59. Springer, 2015.

8 Liming Cai and Jianer Chen. On fixed-parameter tractability and approximability of NP optimization problems. *Journal of Computer and System Sciences*, 54(3):465–474, 1997.

9 Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon. A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM (JACM)*, 55(5):21, 2008.

10 William Cook, László Lovász, Paul D Seymour, et al. *Combinatorial optimization: papers from the DIMACS Special Year*, volume 20. American Mathematical Soc., 1995.

11 Graham Cormode. The hardness of the lemmings game, or oh no, more NP-completeness proofs. In *Proceedings of Third International Conference on Fun with Algorithms*, pages 65–76, 2004.

12 Marek Cygan, Fedor V Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized algorithms*, volume 4. Springer, 2015.

13 Erik D Demaine, Giovanni Viglietta, and Aaron Williams. Super Mario Bros. is harder/easier than we thought. In *Proceedings of Third International Conference on Fun with Algorithms*, 2016.

14 Rodney G Downey and Michael Ralph Fellows. *Parameterized complexity*. Springer Science & Business Media, 2012.

15 Michael Etscheid, Stefan Kratsch, Matthias Mnich, and Heiko Röglin. Polynomial kernels for weighted problems. *Journal of Computer and System Sciences*, 84:1–10, 2017.

16 Guy Even, J Seffi Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.

17 Michal Forišek. Computational complexity of two-dimensional platform games. In *International Conference on Fun with Algorithms*, pages 214–227. Springer, 2010.

18 Harold N Gabow, Zvi Galil, Thomas Spencer, and Robert E Tarjan. Efficient algorithms for finding minimum spanning trees in undirected and directed graphs. *Combinatorica*, 6(2):109–122, 1986.

19 Venkatesan Guruswami, Rajsekar Manokaran, and Prasad Raghavendra. Beating the random ordering is hard: Inapproximability of maximum acyclic subgraph. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 573–582. IEEE, 2008.

20 Viggo Kann. *On the approximability of NP-complete optimization problems.* PhD thesis, Royal Institute of Technology Stockholm, 1992.

21 Ravi Kannan. Minkowski's convex body theorem and integer programming. *Mathematics of operations research*, 12(3):415–440, 1987.

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

23 Daniel Lokshtanov. New methods in parameterized algorithms and complexity. *University of Bergen, Norway*, 2009.

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

25 Giovanni Viglietta. Lemmings is PSPACE-complete. *Theoretical Computer Science*, 586:120–134, 2015.