Automated video game parameter tuning with XVGDL+

Jorge R. Quiñones
(University of Málaga, Málaga, Spain
https://orcid.org/0000-0003-4348-4853, jorge.ruiz.qui@gmail.com)

Antonio J. Fernández-Leiva
(University of Málaga, Málaga, Spain
https://orcid.org/0000-0002-5330-5217, afdez@lcc.uma.es)

Abstract: Usually, human participation is required in order to provide feedback during the game tuning or balancing process. Moreover, this is commonly an iterative process in which play-testing is required as well as human interaction for gathering all important information to improve and tune the game components' specification. In this paper, a mechanism is proposed to accelerate this process and reduce significantly the costs of it, contributing with a solution to perform the game parameter tuning and game balancing using search algorithms and artificial intelligence (AI) techniques. The process is executed in a fully automated way, and just requires a game specification written in a particular video game description language. Automated play-testing, and game's feedback information analysis, are related to perform game parameters' tuning and balancing, leading to offer a solution for the problem of optimizing a video game specification. Recently, XVGDL, a new language for specifying video games which is based on the eXtensible Markup Language (XML), has been presented. This paper uses XVGDL⁺, an extension of this language that incorporates new components to specify, within the video game specification, desirable goals or requirements to be evaluated after each game execution. A prototypical implementation of a Game Engine (termed XGE⁺) was also presented. This game engine not only enables the execution of an XVGDL⁺ game specification but also provides feedback information once the game has finished.
The paper demonstrates that the combination of XVGDL⁺ with XGE⁺ offers a powerful mechanism for helping solving game AI research problems, in this case, the game tuning of video game parameters, with respect to initial optimization goals. These goals, as one of the particularities of the proposal presented here, are included within the game specification, minimizing the input of the process.
As a practical proof of it, two experiments have been conducted to optimize a game specification written in XVGDL via a hill climbing local search method, in a fully automated way.

Keywords: Video game description language, automated game tuning, automated game balancing
Categories: I.2.8
DOI: 10.3897/jucs.75357

1 Introduction

During the refinement phase of game development in general and, in particular, refinement of the game's components, human intervention has traditionally been a necessary aspect. In a, usually, iterative process, once an executable prototype of a game is available, the designers obtain feedback through the *play-testing* process, considered one of the most important activities within the overall video game design task [Fullerton, 2014]. This information is key in technical aspects such as runtime performance or error detection, but it is also a fundamental point to refine certain parameters or components that affect directly both the gameplay and the user experience. This process, usually

called *game parameters tuning* or *game balancing* [Becker and Görlich, 2020], makes game specifications evolve guided by preset requirements and thanks to the feedback provided by the players, through the aforementioned play-testing process.

This process presents drawbacks and difficulties that mainly have an impact on the cost and use of resources during this phase of development. One of the most important aspects is the cost in terms of time and resources necessary to carry out the refinement or tuning. But also, an important point to keep in mind is that having human users (players) greatly increases the subjectivity factor of the information received during this process. Moreover, this classical approach of game parameters' tuning also has an important limitation: the fatigue of the human tester that is produced by the continuous feedback that the refinement process demands to the user [Cotta and Fernández-Leiva, 2011]. Having an automated solution undoubtedly contributes to improving the parameters refinement process at a global level and reduces the effects of the aforementioned problems, even eliminating some of them.

In this article, an automated process to deal with video game parameter tuning is described. This approach minimizes the effects of the aforementioned problems. The process can be carried out even without needing to have a playable prototype of the game, and without the need for human intervention as it just requires a game specification and goals to accomplish after finishing the game play. This solution is framed into the videogame research area and is not intended to avoid the human factor, that must be present during the game specification tuning phase, but it does contribute significantly to mitigating the effects of the problems that it presents.

Within the community of Artificial and Computational Intelligence in Games, one of the most important and valuable objectives is the definition of a language that allows video games to be specified. Formally, a video game description language (VGDL) is a language providing the required structures to define the essential components of a video game. Those elements include mainly rules, mechanics, events, physics, user interaction, multimedia elements or narrative, to mention some of them. A VGDL also helps to understand the features and mechanics of the game and aids in the task of implementing it. Nowadays, the work in the VGDL area is performed mainly at the research and academic level on artificial intelligence in games, leaving the use of it and the development of related commercial tools outside the scope of them.

In [Ruiz-Quiñones and Fernández-Leiva, 2020a], XVGDL (the eXtensible VGDL based on the standard XML) was introduced as a new implementation of a VGDL providing flexibility and abstraction enough to specify a variety of different kinds of video games. XVGDL provides special tags to specify properties and offers support to define a wide set of game components. The main limitations of existing VGDLs and what this new tool offers that these do not are also discussed. XML can be hard to read and write, especially in case of a lack of knowledge of scripting or programming. However, it provides a number of advantages that can compensate its learning curve. Game specifications written in XVGDL are not simple text files tied to a concrete development framework (specifically implemented ad hoc). In fact, XVGDL specifications can be treated as XML files. This provides a number of advantages that other VGDLs does not have. For instance, XVGDL files can be managed by hundreds of software tools dedicated to XML, e.g., translators to other formats –such as Excel, Access, HTML, text, etc.–, content sorters, validators, debuggers, markers, viewers, schema tools, (possibly visual) editors, and many others. Moreover, all modern high-level programming languages (such as Java, C, or C$^{\#}$), provide support to manage XML files and, as a direct consequence, XVGDL files. Other differences with respect to existing VGDLs are also identified–cf. [Ruiz-Quiñones and Fernández-Leiva, 2020a](Section V)–. In fact, this

proposal was aimed not only to ease the creation/refining of game specifications, but also to investigate in search of a standard VGDL. In addition, the XVGDL Game Engine (XGE), a prototype implementation of a tool that allows the direct execution of game specifications written in XVGDL, was also presented.

The work presented in this paper uses an extension of XVGDL, named XVGDL$^+$, that incorporates new features that allow to specify game goals or requirements that are desirable to be accomplished after a game execution. XGE$^+$ an updated version of XGE, has also taken part in this work. This update not only allows the direct execution of game specifications written in XVGDL$^+$, but also acts as an interface that automatically provides feedback at the end of the video game (specification) execution. The feedback can be analyzed and measure how far the game specification is from the goals that were initially set as desirable in the game specification. The employment of XVGDL$^+$ to write game specifications together with assessable objectives that can subsequently be directly executed in XGE$^+$ constitutes a system to reduce the gap between the attainment of a video game specification and the instant in which its executable (possibly prototype) version is developed.

These advances in the definition of the language make this approach also a proposal for automated video game tuning and balancing as it allows to evaluate directly, by execution, whether the game specification meets the requirements that were initially configured. The whole process can be automated so that it does not strongly depend on the user experience associated to the execution of the video game (specification), saving resources and costs.

As well as mentioning the new features of XVGDL$^+$ and XGE$^+$, this paper offers as primary contribution a new mechanism that allows to automatically perform a process of optimization of the video game components (i.e., tuning game parameters in a fully automated way). One of the main novelties of this paper is that it does not require to have a prototype executable version of the video game but just a game specification written in XVGDL$^+$. Another aspect to highlight is that the desirable requirements or goals are included together with the definition of the game in the XVGDL specification.

The process starts from a video game specification in XVGDL$^+$ which also allows new structures to identify requirements or goals that will be evaluated at the end of each gameplay. This specification can be directly executed by XGE$^+$ that, in its turn, will provide feedback (in form of a number of execution data). The combined employment of XVGDL$^+$ and XGE$^+$ helps to assess video game specifications written in XVGDL$^+$with respect to the expectations (initial defined requirements or goals) without the need of producing previously an executable version of the game. Consequently, this process minimizes costs and simplify the process, making it able to be fully automated so that the human intervention is significantly reduced to a just initial definition phase.

In this mechanism, XGE$^+$ can be viewed as an interface that can be used to reduce the iterative (classical) process of (sequentially) first designing, second developing, and finally testing, inside the game development cycle. In this classical approach, there is a significant effort of programming, even considering a prototyping stage, and programmers have to build an executable version of the game and, afterwards, modify it repeatedly according to the requirements until the the goals are satisfied with the last executable version.

As shown along the paper, with this new approach, the game specification and the desirable execution goals can be both written in an XVGDL$^+$ game specification. Afterwards, the game is played and evaluated directly in an execution in XGE$^+$ and later the feedback information can be analyzed. Potentially, using XVGDL$^+$ language predefined components, the phase of programming can be removed in this approach.

However, note that there might be a number of specific external components (identified in the game specification) that might be required to be implemented. If no external components have to be implemented, then no programming is required. Needless to say that, the feedback received via XGE[+] can be used to modify the game specification and restart the process with the modified version as new input to the XGE[+]. In each iteration of the game development cycle, the objective is to obtain a better game version (with respect to preset execution goals). This is clearly a process of game optimization that is being done usually in a manual way. This is because the automation of video game refinement (as discussed in Section 4) is a challenging task from a technical point of view.

This method presents innovations with respect to other existing approaches, mainly highlighting the following: the game specification contains the definition of the optimization requirements. This makes possible that the whole process can be executed completely automatically in the same execution block (the game specification, optimization goals and game engine combined and feedback). Also, this approach allows to carry out the game tuning without the need for an executable version of the game and avoiding carry out the steps of design, execution and analysis through different processes.

Two experiments have been described to demonstrate the second contribution of the paper, that is to say, showing how XVGDL[+], in combination with XGE[+], can be exploited to help in a complex research problem, in this case, optimizing video games' specifications according to initial requirements. In other words, the paper shows by experiments the viability of this work as a novel automated video game parameters' tuning procedure. In the experiments, starting from a given game specification written in XVGDL[+] a set of game parameters affecting several components are automatically improved. The process is guided by the goals specified at the beginning of the process and is based on the use of search techniques. The paper is structured as follows: Section 2 discusses related work, and Section 3 describes both XVGDL[+] and XGE[+]. Then, section 4 includes the analysis of two experiments carried out to validate and shows how the proposal presented in this paper represents a promising and powerful approach for the automated tuning of video game parameters and video game balancing. The paper ends with the conclusions and open lines of future work.

2   Related work

This section presents the work related to the two fundamental aspects that are dealt with in the article, reviewing the existing work related to game parameter tuning in general and optimization. Following, the main VGDL approaches that share this same purpose are mentioned, finishing with XVGDL, the starting point of the research presented in this paper.

2.1   Game parameters' tuning – Game balancing

Optimization process, as a general concept, has been widely studied in any engineering field and it is extensively present in the literature. Even nowadays, thousands of publications related to this concept are submitted every year[1]. An optimization process includes improvements in several different aspects. It can be considered as a task summarized as follows: Starting from requirements and a set of variables, the process is executed. During

---

[1] More than fifty thousand of publications in IEEE Xplore between 2010 and 2019

the process execution, or once it is finished, feedback is provided and analyzed. When all data have been recollected and analyzed, people involved in the process (designers, engineers, final users, etc.) can carefully study the feedback and check if the process' feedback meets their expectations in terms of performance, time, resources, or any other aspect they would like to improve or check.

Video game design process, as a particular kind of process, is susceptible for those concepts to be applied. Optimizing a video game specification can be understood as a wide range of points, and in fact it is so. A game specification can be optimized focusing on different aspects of the game such as graphics, video game components and its features, or video game performance, to mention a few. Although there is not a clear definition of what makes a video game *good* or *bad*, game parameters can be optimized to get a *better* variation (i.e., an executable version coming from a variant of the original game specification and that improves the user experience). The quality of a game is actually a subjective concept and depends on the requirements, or the player' expectations. In addition, the criteria to evaluate a video game as good or bad, or even as better or worse than other versions of the same game (i.e., the criteria or fitness function to compare distinct versions of the same video game), could also be different for different kinds of games. For instance, the time to complete a video game or a game level as a measure to evaluate its quality (or the user experience) can be considered as to be analyzed, this surely will have distinct weight in an adventure game than in a battle royale game.

Game optimization is not a simple task as exposed in [Ebner et al., 2013]. In general, there are interesting recent researches in the literature for optimization related to games or video games [Yin, 2019, Smirnov and Golkar, 2019, Duarte and Battaiola, 2017], giving a good background in the scope of this paper. The optimization and tuning process of video game parameters and their components, also known as game balancing [Becker and Görlich, 2020], is an aspect that has been addressed mainly in two ways at the research level, detailed next.

On the one hand, there are approaches that aim to improve the game parameters associated with agents or bots in the game, so that they are able of improving their performance during the gameplay. In general, this approach evolves the parameters and, potentially, the behavior associated with these agents, using some type of learning technique. In these models, the initial parameters can even be completely random and evolve according to the learning model developed [Patel and Carver, 2011]. Other works base the parameter tuning process on genetic algorithms, such as the particular case of [David-Tabibi et al., 2010] in which a solution (individual) is encoded as a chromosome and a fitness function is defined, in addition to applying the selection, crossover and mutation operators. The population of individuals (solutions) evolves throughout the game. The automatic balance of parameters has also been presented in real-time strategy games [DeLaurentis et al., 2021], in the same way as the previous examples focused in principle on a particular game but potentially with the ability to be applied to other games.

In this line of research but at an even more ambitious level, game tuning can be applied to General Game Playing (GGP). [Sironi and Winands, 2017] presents a study that applies Monte Carlo Tree search for agent tuning, tested on a set of 14 different games of a heterogeneous nature. This is a good example of a more generic approach and improves on previous proposals that focus on a particular game.

These advances are very significant but, in general, there is no direct relationship with the specification of the game. Instead, they are more focused on the objectives of the player. Note that most of works related to game optimization require to execute the game or, at least, simulate its execution to obtain data that will define the fitness of the game. In

general, these steps are executed independently one from each other. The fitness function is used later to guide the search in the optimization process. In the approach presented in this article, the specification of the game (including its goals) and the parameter tuning are described at the same level–that is, the same file–. In this case, the bibliography is much scarcer than in the case of optimization of agents or bots. The proposal presented in this paper introduces a novelty by being able to perform this optimization from the game specification itself. This scarcity of publications and advances is mainly caused by the technical and functional difficulties that the problem presents (see Section 4).

In existing works, such as [Zook et al., 2019], techniques such as Active Learning are applied to optimize certain game parameters through the training, evaluation and selection of a set of selected parameters. This type of approach has a direct relationship with the game specifications since the optimization of the parameters is more aimed at game design components (for example, projectile size or their speed), although there is a framework and a executable version of the game prior to experimentation.

Recently, a paper has also been presented that works on the rules of the game and mechanics and their optimization (balance of parameters), applying an evolutionary algorithm [Wang and Zhang, 2021], which shows that this area of research continues sparking interest and could have real applications in the video game industry.

In the context of VGDL, there are proposals related to game optimization. As it is detailed in [Ruiz-Quiñones and Fernández-Leiva, 2020a], some existing works related to VGDLs are strongly linked to concrete games or game genres [Schaul, 2014, Love et al., 2008], which makes difficult to generalize the process of refining a video game specification. In some cases, VGLDs do not support important features required to make a deeper study, like a Game Engine (GE) adequate to play the video game. In general, VGDLs have been created just for research or theoretical study purposes. Actually, VGDLs can be used as the a base to optimize a game specifications, but do not provide support to define and evaluate the quality based on pre-defined requirements.

As will be seen in subsequent sections, the approach described here tries to address the main lacks of the current works, contributing with a generic, fully automatic process that works directly on the specification of the game in a given description language.

2.2 Video game description languages

Existing papers have expounded initial ideas and established guidelines of what a VGDL must define and how to incorporate video game elements into the game specifications written in a specific VGDL [Ebner et al., 2013]. Many of the VGDLs proposals that can be found in the literature are very specific of the context in which they were used. Other proposals describe schemes of VGDLs that are based on high-level programming languages in the search of a standard VGDL. Moreover, these approaches try to produce executable versions directly from game specifications, what might be considered the holy grail of a VGDL. The XVGDL proposal, pursues this goal [Ruiz-Quiñones and Fernández-Leiva, 2020a].

2.2.1 VGDLs based on high programming languages

The work presented in [Ebner et al., 2013] establishes the basis for a generic VGDL [Love et al., 2008]. The importance of finding a formal way to specify video games concisely is, nowadays, a well-accepted approach within the AI community [Ebner et al., 2013, Levine et al., 2013]. Considering games generically, the Game Description

Language (GDL) represents the standard of General Game Playing (GGP) for describing games, even dealing with both randomness and imperfect information [Thielscher, 2010]. The GDL consists of a mathematical notation similar to the syntax and semantics of logic programming that cannot be directly extended to cope with video games because, for instance, it does not provide structures or components to describe multimedia elements or user interaction.

Although theoretical issues are not in the scope of this work, there are approaches related to VGDLs primarily focused on mathematical models for describing games [Levine et al., 2013, Jiang et al., 2016].

With respect to video games, no standard VGDL exists. This paper is focused on a practical approach in the sense that it considers those VGDLS that not only allow specifying the game, but also provide an extra as their game specifications can be compiled (or interpreted) into an executable. These can include the use of standard tools or languages. In this line of research, some experiments have been reported in the literature designed to obtain a playable video game from the game specification written in a concrete VGDL. This is the case of PyVGDL, a VGDL based on the programming language Python [Schaul, 2014]. This approach represents a good starting point to compile game specifications, but it is developed in a very specific way to describe video games so its use is very limited. There are some other works that are associated with high-level programming languages such as Ludi [Browne, 2014], although in this case, the solution is restricted to one-player board games. Other interesting proposals suggest the integration with commercial game engines. This is the case of Casanova, a declarative programming language focused in games (not video games) that allows the integration with the well-known Game Engine Unity [Unity Technologies,] and which can also be used with other engines and libraries [Casanova, 2012, Abbadi et al., 2015, Di Giacomo et al., 2017].

From the point of view of game design, it is worth mentioning that there are also approaches, such as Ludocore [Smith et al., 2010], that help to develop rules and their relations, contributing with a flexible game engine for logical games, able to be applied to both GDLs and VGDLs.

However, generally speaking, current implementations of compilers for VGDLs are usually *closed* and developed in a very specific way to describe games or video games. In most of the cases, they are tied to certain features or specific games and therefore are not transferable to different tools or engines. This gap needs to be covered by a more generic and flexible implementation of a VGDL and its corresponding game interpreter or compiler. The following section focuses on a proposal that has been recently reported in the literature to deal with this issue. The interested reader is referred to [Ruiz-Quiñones and Fernández-Leiva, 2020a] (see Section V), where the advantages and benefits of using XVGDL over other existing approaches are widely exposed.

There are also works related to game generation and evaluation based on PuzzleScript description language [Lavelle, 2017]. In particular, [Chong-U Lim and Harrell, 2014] is an interesting approach focused on generating novel games, starting from a defined ruleset and evolving different components to create new mechanics, levels, or end conditions.

2.2.2 XVGDL

Recently, XVGDL[Ruiz-Quiñones and Fernández-Leiva, 2020a] was presented, a VGDL based on XML whose game specifications can be compiled and executed via a proto-type specific Game Engine (termed XGE). XVGDL takes advantages of the potential of XML,including its simplicity, generality, and usability across different platforms

and environments. Games described in XVGDL are XML files that can make use of both libraries dedicated to managing XML files and XML data structures that can be interrelated. These data structures can be useful to define game objects that will interact in the game. Moreover, XVGDL game specifications are not restricted to a concrete programming language or an operating system. In addition, XVGDL files (as any XML file) can be validated by particular XML Schema Definition (XSD).

XVGDL offers a number of components allowing to write game specifications in a simple and clear way. In general, an XVGDL game description (also termed as specification) follows the following code scheme:

```
ZL-\C?CHS^SzSb^N
  ZFCC pC^@CqCq <b^HSL~q-zSb^ CCN
    iii
  ZFCC yS\Co-z <b^HSL~q-zSb^ CCN
    iii
  ZY-%o-zN iii ZwY-%o-zN
  Z\-e iiiN iii Zw\-eN
  Z<b^zqbYs iiiN iii Zw<b^zqbYsN
  ZeY-%Cqs iiiN iii ZweY-%CqsN
  Zb4UC<zsN iii Zwb4UC<zsN
  ZCfC^zsN iii ZwCfC^zsN
  Zq-YCsN iii Zwq-YCsN
  ZC^@ b^@SzSb^sN iii ZwC^@ b^@SzSb^sN
  ZL-\Orz-zCsN iiiZwL-\Orz-zCsN
ZwL-\C?CHS^SzSb^N
```

XVGDL provides tags to cope with both configuration issues and video game components. The specific video game components managed in XVGDL can be listed as: layout, maps, controls, players, objects, events, rules, end conditions, and game states. Each component (and its sub-components) can make use of properties and/or attributes as usually employed in XML. The syntax is as follows:

```
Z<b\eb^C^z f-zzqS4-zCc bq eqbeCqz%ag iii f-zzqS4-zC] bq eqbeCqz%gN
    iiii<b^zC^z
Zw<b\eb^C^zN
```

XVGDL also offers the possibility of linking video game components with customized external components (ad hoc implementations) that can be used for different purposes, for instance, to deal with the game AI or specific rendering issues.

The interested reader is referred to [Ruiz-Quiñones and Fernández-Leiva, 2020a] and [Ruiz-Quiñones and Fernández-Leiva, 2020b] for more information. Suffice it to highlight that XVGDL provides enough flexibility to let easily represent the video game, its components, rules, mechanics, etc. in a standard, flexible and portable format.

3   XVGDL+: The XVGDL Extension

XVGDL has been extended to support the definition of game goals, technically called *objectives*. This update is named XVGDL+. The objectives are defined as components that will be evaluated after the game execution, that is to say, after executing the XVGDL+ game specification in the Game Engine XGE+ (see Section 3.2). These evaluations are intended to provide useful information (i.e., feedback), helping humans or, potentially, automated processes, to refine and improve the game definition. The game engine, in charge of executing the video game, is responsible for providing this information so it can be analyzed as needed once the game has finalized.

Section 3.1 describes how XVGDL[+] offers support to manage requirements or goals. Then, Section 3.2 focuses on XGE[+], that is to say, the Game Engine in charge of executing game specifications written in XVGDL[+], showing how the objectives given in the XVGDL[+]specification are evaluated. Afterwards, and for clarity, Section 3.3 displays an example of a well-known video game (namely Pacman) written in XVGDL[+].

3.1   Objectives in XVGDL[+]

A game objective identifies those aspects of the game that might be considered to be enhanced (or optimized). More formally:

Definition: An objective is an aspect of the game that can be measured or calculated once it finishes and offers significant information to build or improve the game according to initial expectations.

For clarity, examples of possible game objectives are detailed next:

1. Generally speaking, designers usually look for interesting challenges to pose to for the player. It seems reasonable, for instance, to think that the game will be all the more challenging and exciting if the player ends with just one life left. So, one interesting goal is to minimize the number of lives that the player keeps at the end of the game execution.

2. In 2-player games with one virtual player, the it might be interesting to alternate the victories of the human and the virtual player. In this case, it seems adequate to receive feedback, at the end of a game execution, indicating if the player has won the game or not. Therefore, an objective might added to the game specification –associated to an external component– to evaluate this outcome.

XVGDL[+] includes a new component that allows specifying game objectives. Game **b4UC<zSƒCs** can be stated by adding, to the XVGDL code schema shown in Section 2.2.2, the following component (where **ryp** denotes a string):

```
Zb4UC<zSƒCsN
  Zb4UC<zSƒC <Y-ss]-\CEorypo s<bqCEorypo .CSLPzEorypoN Zwb4UC<zSƒCN
  i i i
  Zb4UC<zSƒC <Y-ss]-\CEorypo s<bqCEorypo .CSLPzEorypoN Zwb4UC<zSƒCN
Zwb4UC<zSƒCsN
```

Therefore, a set of objectives is enclosed between the start-tag **Zb4UC<zSƒCsN** and the end-tag **Zwb4UC<zSƒCsN** Each of the specific objectives of this set is specifically described between the tags **Zb4UC<zSƒCN** and **Zwb4UC<zSƒCN** using the following properties:

– **<Y-ss]-\C**: each objective has associated an external component, specified by a **<Y-ss]-\C**, as is usually configured in XVGDL [Ruiz-Quiñones and Fernández-Leiva, 2020a, Ruiz-Quiñones and Fernández-Leiva, 2020b]. This component can be implemented independently and will be in charge of evaluating the objective when the game specification is run in XGE[+]. This component manages the evaluation of the stated objective and returns a score associated to it when its execution finishes. The implementation of this component is precisely one of the primary tasks that the programming team must tackle to enable the execution of the game specification in XGE[+].

- **s<bqC**: is an optional property that defines the default score assigned to the objective when this score is not computed by the external component associated to the property **<Y-ss]-\C** as previously stated.

- **.CSLPz**: is a property that can be used to rank and prioritize the objectives. This property specifies a weight value assigned to the objective. It is represented by a number. It can be thought as the importance given to a concrete objective among others. Analogously to the score property, the weight is as well optional, and will be 1 for all objectives by default if no one is specified by the external component associated to the property **<Y-ss]-\C**.

In addition, in a multiobjective approach, the objectives can be prioritized by assigning them different scores and weights. See the next example of the objective components defined in XVGDL⁺:

```
Zb4UC<zSƒCsN
   Zb4UC<zSƒC <Y-ss]-\CEn,,S^^S^LK-\Ca4UC<zSƒC; PC<W2qn s<bqCEncC5i
      ,! .CSLPzEn|iCEwN
   Zb4UC<zSƒC <Y-ss]-\CEn]~\4CqaHXSƒCsa4UC<zSƒC; PC<W2qn s<bqCEnIn
      ,! .CSLPzEnciCEwN
Zvb4UC<zSƒCsN
```

where the two configured checkers are external components that the XVGDL⁺ Game Engine will invoke to evaluate each of the objectives. Optionally, using the scores and weights, the aggregation of both evaluations will be obtained by the game engine and provided once the game has been completed. A final value for the objectives will be a weighted linear combination (weighted sum), where the scores are the values for each of the objectives, while the weights are the factors applied to each of them to obtain the final result.

3.2 XGE⁺: the XVGDL⁺ Game Engine

The XVGDL Game Engine (XGE) was introduced as a prototype of an XVGDL interpreter that allows XVGDL game specifications to be executed [Ruiz-Quiñones and Fernández-Leiva, 2020a]. XGE⁺ is an extension to deal with the new components of XVGDL⁺. Its main features are the following:

- XGE⁺ allows to play games described in XVGDL⁺, managing the main game loop. It is responsible for applying the game events and game rules, and checking the end conditions at each iteration.

- XGE⁺ is a core implementation of a Game Engine to parse video game specifications written in XVGDL⁺ and is able to run them by invoking the external components specified in the properties of a game specification written in XVGDL⁺.

- XGE⁺ will also be in charge of doing the proper calls to the renderer (if defined) to let the game be presented on the screen according to the concrete renderer specifications.

- As XVGDL⁺ allows to describe (possibly externally implemented) components and AIs for objects in games, XGE⁺ can execute games completely without any human intervention.

  – XGE⁺ allows the evaluation of game objectives stated inside an XVGDL⁺ specification that is executed. This way, feedback is obtained once the game specification has finished its execution in XGE⁺.

With respect to this last functionality, XGE⁺ provides an Application Program Interface (API) which allows obtaining significant information from the game (specification) execution, not only once it has finished but also while it is taking place. This API is used to compute the concrete evaluation of the specified game objectives associated to the execution of the game specification. More specifically, there are two ways of evaluating the objectives:

1. A list of objectives can be defined as previously stated, and each of them will be evaluated separately by the XGE⁺ game engine. Once all are evaluated, XGE⁺ joins all results and obtains a final evaluation of the game. In this case, XGE⁺ implements a weighted sum model, using all the described objectives and their weights (if they are also specified), and returns a compound value. In other words, let gs be a game specification with a list of n objectives $o_1, \ldots, o_n$, and where $w_1, \ldots, w_n$ and $s_1, \ldots, s_n$ are the weights and scores given and computed, respectively, for each of the n objectives. Then, when gs is executed in XGE⁺, its fitness evaluation returned by XGE⁺ is calculated according to the Equation 1:

$$fitness(gs) = \sum_{i \in \{1,\ldots,n\}} w_i \cdot s_i \tag{1}$$

2. The second option is to take advantage of the XVGDL⁺ structures to link with an external component that might be developed to be in charge of evaluating specific game objectives. The XGE⁺ API makes it possible to access all the required information, so the evaluation component just needs to get the corresponding values from XGE⁺ to obtain a final score for the objective. The advantage of using external components is that they are fully customizable and the video game specification is much simpler. An example of the specification for evaluating this kind of objectives could be as simple as:

Ƶb4UC<zSƒC <Y-ss]-\ŒEna4UC<zSƒCsBƒ-Y~-zSb^n wN

where an external component termed **a4UC<zSƒCsBƒ-Y~-zSb^** is in charge of the evaluation process.

Some generic objectives evaluation components have been already implemented and included in the XGE core code, such as, for instance, minimizing the player lives after a game not reaching zero or maximizing the player score after the game execution. These components are ready to be used just by referencing them in an XVGDL⁺ specification.

The XGE⁺ is available online as open source. It can be downloaded from github [Ruiz-Quiñones and Fernández-Leiva", 2022]. It is also packaged inside a runnable jar file. XVGDL⁺ games specifications can be launched using the following command:

U-ƒ- CU-q ‡ƒL@YCL-\CCC^LS^ClU-q Z‡ƒL@YCseC<SHS<-zSb^N

Anyhow, XGE⁺ was created as a prototype game engine for research purposes, and must be considered as that.

3.3 An example: Pacman

This section describes an example of an XVGDL[+] specification for the classical Pacman game. This game has been selected for several reasons: it is widely used in the literature, its mechanics are well-known and it is simple to understand. In addition, it offers many different possibilities to be used as testbed and this serves the purposes of game optimization as described in the following sections. Furthermore, Pacman was used as an example in [Ruiz-Quiñones and Fernández-Leiva, 2020a].

A classical Pacman specification written in XVGDL[+] is available in B. This particular game specification has a number of definitions related to the original game and also other components that are directed to specify technical details associated with the execution of the game. Following, details of this specification are given and refer specific lines or fragments of the Pacman specification shown in B.

- Lines 5-18 are dedicated to give details associated with the rendering of the game (and its graphical user interface) when this is executed. Also, a configured timeout is set before the game is executed. If the timeout is reached, game ends and the player looses the game.

- The game takes place in a two-dimensional map which has walls that avoid **e-<\-^** and **LPbszs** to move freely over it (see Lines 24-27). Note that, when the game is executed, the map will be automatically generated by an external component specified by property **LC^Cq-zbq** in Lines 25-26.

- As usual, there is a **e-<\-^** character located on the map and it is controlled by the player according to Lines 20-22. However, for the experiments conducted in next Section, the **e-<\-^** behavior will be controlled by an AI associated to an external component via the property **-S**. See Line 30.

- All the objects and characters presented in the game are specified in Lines 33-38. As usual, there are: (a) a number of items (**s\-YY @bzs**) distributed on the map at those places not occupied by walls; (b) special items (**4SL @bzs**) that make **e-<\-^** to be powered-up; and (c) a number of **LPbszs** that are trying to catch **e-<\-^**.

- Game rules are stated in Lines 40-81. For instance, when a **LPbsz** catches **e-<\-^**, the player loses a life; see rule **L**hostCatchPacmanLivesDown in Line 62. If the player loses all lives, the game finishes (see **C^@-b^@SzSb^s** in Lines 93-96). For the game execution in XGE[+], an AI implemented as an external component will be in charge of controlling **LPbszs** (see property **-S** in Line 35).

- In this example, game objectives are associated to an external component in Lines 98-101. Here **d-<\-^aezS\Sx-zSb^rbY~zSb^Bf-Y~-zSb^** is an external component that will return the value(s) associated to the execution of the game. The number and nature of these values depend on the requirements, consequently, the external component will be previously programmed according to these.

Given this Pacman XVGDL[+] specification fulfilling the requirements for the aforementioned game rules and objectives, the game is ready to be executed by the XGE[+]. Figure 1 shows a screenshot of the Pacman specification running in XGE[+]. In this screenshot, the game information like the score, player lives and gameplay time are displayed at the top and the bottom of the screen. the walls are represented by the character ▦. Small and big dots are represented by ◙ and ◘ respectively. Pacman is represented by the character ◉ and ghosts are represented by the character ◩

Figure 1: Screenshot of the execution in XGE⁺ of the Pacman specification written in XVGDL⁺.

3.4 Tuning of components and game specification with XVGDL

Before detailing the main features of XVGDL⁺ and regarding the work that presented in this article, components that have been used in the parameter tuning algorithm and that take a direct part of the experiments are detailed here. These components are directly related to the game specification, differentiating this approach from those works aimed at tuning agent or bot parameters.

As will be discussed later, to apply the search and optimization algorithms, a base video game specification (i.e., a germinal specification of a video game) written in XVGDL format is required. For clarity and because this game will be used in the experiments, the XVGDL components that will participate directly in the optimization process are listed below, highlighting their basic specification. For more details about the value ranges that are allowed for each of them during the experiments, see Section 4.1.

1. Number of lives that the player has at the beginning of the game, specified in XVGDL inside the node that represents the player configuration:

   ```
   ZeY-%Gq ^-\ŒEne-<\-^n s<boqŒEnŒ5 YSƒCsEn{n YSƒCdCq<C^z-LŒEncŒŒ5 -S
   ⌐  En<b\iUq|i ‡ƒL@Yie-<\-^i \b@CYi b4UC<zi -Sid-<\-^,RmwN
   ```

2. Number of enemies present on the game, described in the *ghost* game object definition node:

   ```
   Zb4UC<z ^-\ŒEnLPbszn z%&ŒEnC^C\%n @%& \S<Enzq~Cn ƒbY-zSYŒEnzq~Cn
   ⌐  sS< ŒEnc>cn S^sz-^<ŒsEnJn -SEn<b\iUq|i ‡ƒL@Yie-<\-^i \b@CYi
   ⌐  b4UC<zi -SiKPbsz,Rn wN
   ```

3. Map fill ratio, that determines the number of items that appear on the map at the start of the game. Although there is a initial number of *instacnes*, this value will be calculated for each possible solution as detailed in the Section 4.1 and is based on the following specification of the *smallDots* game objects:

```
Zb4UC<z ^-\OEns\-YY?bzn z%&OEnSzC\n @%‰\S<Enzq-On ƒbY-zSYOEnzq-C
n sS‹OEnc>cn S^sz-^<OsEn|n wN
```

4. Number of items called *bigDots* initially placed on the map at the beginning of the game, described in XVGDL in the corresponding game object node. The same as for the *smallDots*, the base specification has a number of instances, described as follows:

```
Zb4UC<z ^-\OEn4SL?bzn z%&OEnSzC\n @%‰\S<Enzq-On ƒbY-zSYOEnzq-On
sS‹OEnc>cn S^sz-^<OsEn|n wN
```

5. Game timeout, the game property that defines the maximum time in seconds that the game will last, configured as a main game property under this *property* node:

```
ZeqbeOqz%‰VC%EnzS\Ob~zn ƒ-Y~OEn|OEEEEi wN
```

Note that the details of the components specification belong to the initial specification that is used to run the algorithms. Programmatically, and at execution time of the tuning algorithm, these values will evolve for each solution.

4 Video game optimization using XVGDL[+]

The importance of a VGDL in the optimization of game specifications is a fascinating issue arisen in both the Software Engineering and the Artificial Intelligence communities. In general, the game is described subjected to a set of initial requirements. Afterwards, other ideas and game components are incorporated to make the game funny, attractive, challenging, enjoyable, and as flexible as possible. The initial specification is improved via an iterative cycle of trial and error. In this cycle, the game design is followed by the implementation of a prototype executable version that is tested by a number of users. The feedback helps to modify the original game specification and the process iterates until the specification satisfies the initial goals or the resources (e.g., development time or financial support) are exhausted.

This paper explores the potential of XVGDL[+] in combination with XGE[+]to evolve (i.e., optimize) the original specification of a game. The goal is to obtain *better* video games (i.e., games that improve the user experience) according to some preset game objectives. The optimization process starts by defining a game specification in XVGDL[+] in which a set of game *objectives* are identified, as detailed in previous sections.

Any game specification obtained within the process can be directly evaluated in XGE[+], as this provides support to compute (and return) values associated to a particular execution. This feedback can be used to modify a game specification and restart the process with the modified version as new input to the XGE[+]. This section is devoted to show by experiments the validity of this approach.

4.1   An experimental case study: Pacman optimization

This section describes an experimental work to demonstrate how the approach presented here, shapes a promising mechanism to optimize games automatically. Pacman has been used as testbed, starting from its game specification written in XVGDL+ and described in Section 3.3.
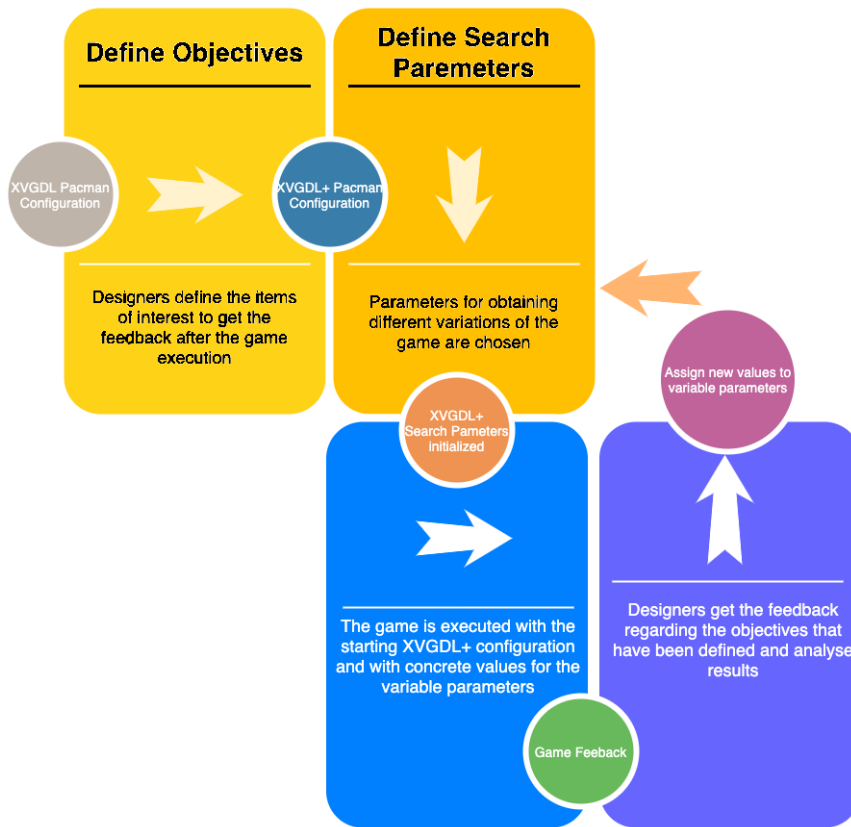

Figure 2: Graphical representation of the video game optimization using XVGDL+

Figure 2 expresses graphically the different steps included in a process of Pacman optimization, in which the inputs and outputs of each phase are identified. Consequently, the video game optimization can be viewed as a repetitive task of this process. This process can be generalized to other video games (specifications). The initial game specification includes the pursued objectives when the game is executed. This definition of the objectives is directly related to the identification of the parameters whose values need to be returned at the end of the game execution. With these values (e.g., number of lives remaining, or execution time to complete the game), the process can assess the quality of the game and revise (evolve) its specification if necessary.

This process can be automated to be applied as a mechanism for optimizing game specifications. Consider, for instance, an initial set of distinct XVGDL⁺ specifications of the same game, or a set of different instances of a parameterized XVGDL⁺ game specification (this second scenario is the case studied in this paper). Then, each of these specifications/instances can be executed and evaluated separately (in XGE⁺) according to the objectives. This enables their comparison so that the 'worst' specifications might be discarded and new ones can be generated for further comparisons. These steps can be repeatedly carried out until reaching a specific termination condition. At the end of the process, the best solution (i.e., specification) can be considered as a (possibly local) optimal solution.

In the following section, an experiment that illustrates the idea for automating a game optimization process is described. For the interested reader, the execution technical details are given in the Appendix A. To perform the whole process, the Pacman specification described in section 3.3 has been considered as the original seed to be optimized, and where:

– A virtual player, in form of an AI to control **e-‹\-^**, has been implemented as an external component (see Line 30 in B). This AI is described in C.

– An AI has also been developed, as external component (see Line 35 in B), to govern the **LPbszs**. The details can be found in D.

– The experiments are executed on the same two-dimensional map, with a fixed size of 15x28 squares (see Lines 24-27). This map only includes the **..-YY** objects. The rest of the game objects used for the experiment (i.e., ghosts, small dots, and big dots; see Lines 33-38) are automatically generated and initially located on the map according to the initial map configuration file (see Lines 25-26). A graphical representation of the map is shown on figure 3. In this screenshot, the **#** icon represents the map walls, while other components of the gameplay as the player or enemies are not included.

– The game has no renderers or layouts configured, as during the search process, game rendering is not needed and it is not an aspect required by optimization requirements.

– The game has two different states (Lines 83-91). In the default state, a **LPbsz** can catch **e-‹\-^** (so the player loses a life), while the **e-‹\-^db.Cq}e** state allows **e-‹\-^** to move faster than ghosts and makes the player character be able to catch ghosts. The ghosts so seized will move to their initial position and freeze for five seconds until they can move again.

Two experiments have been carried out to optimize the Pacman specification with two distinct objectives. (a) In a first experiment, the goal specification considers that a winning game is better, from the user perspective, than a losing game. In this case, the primary goal is precisely to offer a version of the game that prioritizes the victory of the player. (b) In a second case, the goals are more focused on offering a challenging experience to the player. This means that the primary goal is not to make the player win or lose, but to put pressure on the player to obtain a final result under stressful circumstances. Based on previous game experience, this second objective is related to achieve the following sub-objectives at the end of a game:

– Minimize the number of small dots left on the map so that the player finishes or is close to finish the level.
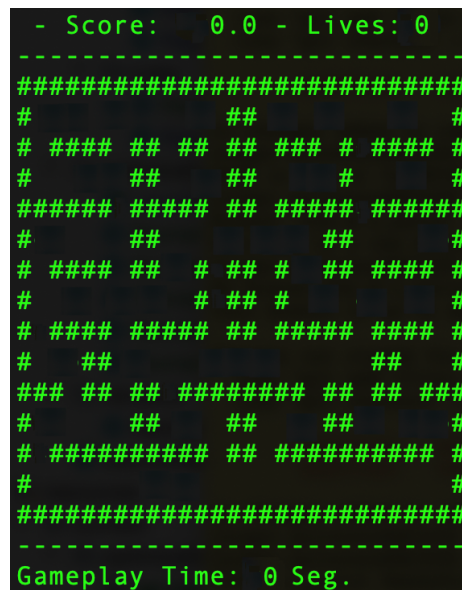
Figure 3: Screenshot of the base Pacman map used in the experiments.

– Minimize the number of remaining lives (not reaching 0 meaning a victory of the player) so that the player is stressed because ghosts are close to win the game.

– Maximize gameplay time with respect to the configured timeout. A game version is considered better if the gameplay time is closest to the configured timeout. That means that the game has finished almost at the allowed time limit.

In fact, the second experiment is related to the concept of game difficulty. Games that are too easy for the players could be considered less interesting, while games won by the player (when the player has only one life left) seem to be more exciting and challenging. Also, a game can be considered to be more exciting when the total gameplay time is closest to the limit time (without actually reaching it).

For clarity and simplicity, the experiments are directed to obtain the optimal configuration (the one that leads to achieve the objectives described above) of an initial game specification that has been parameterized with respect to the following values:

1. Number of lives that the player has at the beginning of the match ($nl \in [1; 5]$, that is to say, 5 possible values).

2. Number of ghosts during the match ($ng \in [1; 5]$, that is to say, 5 possible values).

3. Map fill ratio ($fr \in [10\%; 45\%]$, that is to say, 36 possible values). This parameter sets the percentage of small dots with respect to the empty spaces (those that are not part of the **.-.YYs**, **e- <\- ^** and **LPbszs**) on the map where the game takes place. This is associated to the replacement of the empty spaces on the map by small dots. Based on a set of preliminary experiments, a filling range between 10% and 45% has been considered. The number of small dots initially placed in the game will be denoted by *nsd*.

4. Number of big dots initially placed on the map at the beginning of the match ($bd \in [1;5]$, that is to say, 5 possible values).

5. Game timeout ($to \in [100;400]$ seconds, that is to say, 301 possible values).

According to this set of parameters, the search space contains 1.354.500 possible candidates, that is to say, $nl \cdot ng \cdot fr \cdot bd \cdot to$.

For the experiments covered in this paper, the XGE$^+$ will execute distinct instances of the same Pacman parameterized XVGDL$^+$ specification (denoted by gs) charged with distinct values for the configuration parameters $v_{config} = < nl;ng;fr;bd;to >$. At the end of each game, XGE$^+$ will return feedback containing the following data: a Boolean value $b_{result} \in \{0;1\}$ (i.e., 0 or 1) indicating whether the player loses (then, $b = 1$) or wins (then, $b = 0$) the game, the number of remaining lives the player has ($n_{rl} \in [0;nl]$), the number of small dots left on map ($n_{rsd} \in [0;nsd]$), and the difference in seconds between the time consumed in the gameplay ($time_{gameplay}$) and the timeout for the game (to), i.e., the remaining time of gameplay $time_{rgameplay} \in [0;to - time_{gameplay}]$. Therefore, XGE$^+$ will return, at the end of the execution of the game specification, the following 4-tuple:

$$rvalues = < b_{result};n_{rl};n_{rsd};time_{rgameplay} >$$

So, solutions can be compared with respect to these values returned by independent executions of the solutions. How to compare those values depends on the game objectives. For instance, if the priority is to obtain game versions where the player wins, then a lexicographical ordering can be considered. So, let us assume that gs^i and gs^j are two distinct variants of the same video game (e.g., Pacman) specification whose configurations values are v^i_{config} v^j_{config} respectively, and that the execution of gs^i and gs^j in XGE$^+$ returns, respectively, the tuples $rvalues^i$ and $rvalues^j$. Then, gs^i is better that gs^j if $rvalues^i <_{lex} rvalues^j$ (i.e., $rvalues^i$ is lexicographically less than $rvalues^j$). This means that gs^i guarantees that the player wins, whereas gs^j does not. Or that the two specifications guarantee that the player wins (or loses at the same time) but gs^i returns a smaller number of remaining lives at the end of the game than gs^j, and , in case of returning the same value for this, then the first version guarantees a smaller number of small dots left than the second version. The time left until timeout is the last value to compare (the lower, the better).

4.2 The search algorithm

A local search algorithm has been used for the optimization process. More specifically, a classical multi-start hill climbing (HC) has been employed. This is a local search technique that is simple to understand and has been applied to address with success a number of combinatorial problems [Michalewicz and Fogel, 2004]. Given a current solution gs (i.e., a game specification), its neighborhood $N(gs)$ is explored, and the best solution found is taken as the new current solution, provided it is better than the current best solution. If no such neighboring solution exists, the search is considered stagnated, and can be restarted from a different initial point. This whole process is executed until the computational budget (i.e., a maximum number of evaluations/executions in XGE$^+$) is exhausted.

The initial seed, termed as gs^0, is the Pacman specification described previously, where its configuration values $v^0_{config} = < nl^0;ng^0;fr^0;bd^0;to^0 >$ have been randomly

generated in their corresponding ranges. The representation (i.e., encoding) of any candidate solution gs^i is directly associated with its configuration values v^i_{config}, and its fitness is associated with the values returned by its execution in XGE⁺, following the formula shown in Equation 2.

$$fitness(gs^i) = < b^i_{result}; n^i_{rl}; n^i_{rsd}; time^i_{rgameplay} > \qquad (2)$$

Given a candidate solution $gs = < nl; ng; fr; bd; to >$ (where each of its 5 configuration values as gs_k for $1 \le k \le 5$ are identified), the following neighborhood $N(gs)$ has been considered, according to Equation 3.

$$N(gs) = \{gs^j \mid H(gs; gs^j) = 1 \wedge \exists k \in [1;5] : gs^j_k = gs_k \pm 1\} \qquad (3)$$

where $H(gs; gs^j) = \sum_{j=k}^{5} [gs_k = gs^j_k]$ is the Hamming distance between the configuration values of candidates gs and gs^j (i.e., the number of positions in which their 5-tuple of configuration values differ), and [ ], inside the definition of the Hamming distance, is Iverson bracket (i.e., $[P] = 1$ if P is true, and $[P] = 0$ otherwise). This means that neighbours of a specific game specification are calculated by just adding or subtracting 1 to any of its configuration parameters. That way, for a given solution gs, at most 10 neighbours can be generated (possibly even less, if configuration values are edges cases).

As already mentioned, the optimization process starts from an initial seed configuration gs^0 whose execution in XGE⁺ returns a fitness value $rvalues^0$; gs^0 is considered the best solution at the start. Then, in the i'th iteration of the search, the neighborhood of the best solution found so far (i.e., $N(gs_i)$) is generated and the fitness of all the neighbors is computed via their independent execution in XGE⁺. If the best solution is improved because other solution is better according to its returned values (ties are randomly broken), then the best solution (i.e., gs_i) is replaced by this new solution (let us say gs_{i+1}), and the process continues iteratively with the following iteration. If no such neighboring solution exists, the search is considered stagnated, probably because a local maximum has been reached. In that case, and making it possible to find other maximums, the algorithm is forced to move somewhere else inside the search space, generating again random values for the configuration parameters. The process continues until a maximum number of executions (i.e., evaluations) n_{eval} of game specifications in XGE⁺ is reached.

It is important to mention that this search process does not assure a (global) optimal solution but the initial solution, is surely improved in accordance with the specified objectives.

At the end of the optimization process, the outcome of each experiment is a set of new game specifications that optimize the initial specification as input of the process. These results might be considered as a final solution, as a source of inspiration for changing the initial game seed, or just as proof that the game concept does not match the requirements and can be discarded.

In the following sections, the data obtained for two experiments –one in which the player wins (but is close to lose) and other in which the player loses (but is close to win; in this case the number of remaining lives is zero)– is analyzed and discussed. Both experiments start from the same initial Pacman specification. Also, the experiment has been divided in 10 iterations running 1000 specifications each, being $n_{eval} = 10000$.

4.3 Experiment 1: Finding good variations for Pacman video game

In this case, the experiment searches the most interesting version of the Pacman specification that minimizes its fitness. This means that the player wins the game (i.e., $b_{result} = 0$), whereas the number of remaining player lives, small dots left on the map, and remaining time to reach the timeout tend to be minimal. Table 1 shows the 10 best solutions found in this experiment with respect to the returned values and the initial game objectives specified. The configuration values for each of the solutions are also shown in the table. First column provides an identifier for the specific solutions whose values are shown in the corresponding row. Second column shows the game score obtained at the end of the execution. Columns 3rd to 6th display the values returned by the execution of the game specification: $b_{result} = 0$ if player wins and 1 otherwise), number of remaining lives (n_{rl}), number of small dots left (n_{rsd}), and remaining time of gameplay ($time_{rgameplay}$). Columns 7th to 11th show the values preset to the configuration parameters of the game specification before its execution: number of lives (nl), number of ghosts (ng), map fill ratio (fr), number of big dots (bd), and configured timeout in milliseconds (to).

#	score	b_{result}	n_{rl}	n_{rsd}	$time_{rgameplay}$	nl	ng	fr	bd	to
		Execution Results ($rvalues$)				Configuration Parameters (v_{config})				
1	4647	0	1	0	82382	2	2	26	3	87147
2	4613	0	1	0	108415	1	1	41	5	107065
3	4600	0	1	0	86485	2	3	35	5	88412
4	4595	0	1	0	81953	2	2	25	2	82382
5	4594	0	1	0	81880	1	2	26	2	82382
6	4592	0	1	0	56877	3	1	10	3	57674
7	4563	0	1	0	85725	2	3	35	5	89412
8	4559	0	1	0	78292	1	2	26	2	82382
9	4557	0	1	0	102833	2	2	41	5	107065
10	4521	0	1	0	112318	1	2	39	3	112318

Table 1: Best results obtained in experiment 1.

These 10 best solutions are versions of the parametrized Pacman specification and guarantee the victory of the player and all the small dots have been eaten at the end of the match (as expected). Moreover, all these specifications also assure a certain level of stress (as required). This argument is based on the fact that, at the end of the executions, the player owns only one life.

Paying attention to the configuration values shown in columns 7-11 in Table 1, it can be observed that the initial number of lives for the player is in the range of ⑨⑥②: and the number of ghosts vary between ⑨⑥②: . This is helpful information to decide finally how many lives or ghosts are configured at the beginning of the game. Regarding to the map fill ratio, there is no clear decision. The values ranges between [10%,41%], but the value of 10% might be an outlier, as the rest values are higher than 25%. This suggests the map fill ratio to be in the range of [25%,41%].

In the case of the big dots, it is clear that at least 3 of them must be present. Note also that there are no versions in the list of most promising games using just one big dot. This surely makes the game too difficult for the player to achieve a victory. It is also interesting to point out that the classical value of 4 big dots is not recommended according to this experiment. As for the remaining time of gameplay, the values returned do not provide significant information to be analyzed, as the game is not rendered in an screen and the time consumed is negligible.

Experiment 2										
		Execution Results (*rvalues*)				Configuration Parameters (v_{config})				
#	score	b_{result}	n_{rl}	n_{rsd}	$time_{rgameplay}$	nl	ng	fr	bd	to
1	4696	1	0	1	26685	4	5	20	1	26685
2	4682	1	0	1	27449	5	4	10	4	28150
3	4680	1	0	1	18873	2	4	34	5	19809
4	4677	1	0	2	19503	3	5	33	4	19771
5	4674	1	0	2	16202	3	2	39	5	16748
6	4671	1	0	2	16505	5	4	41	4	17404
7	4669	1	0	2	16005	1	4	25	2	17060
8	4665	1	0	3	13311	3	3	13	1	13795
9	4665	1	0	2	15501	2	1	21	1	17918
10	4661	1	0	3	10201	1	5	14	5	11021

Table 2: Best results obtained in experiment 2.

4.4   Experiment 2: Finding challenging levels for Pacman video game

In this case, the experiment is similar to the previous one, but the game is intended to finish with a narrow victory of the virtual player. This means that the player loses the game (i.e., $b_{result} = 1$) and thus, the remaining number of lives is zero at the end of the game. The rest of values (i.e., small dots left on the map and time remaining until timeout) are minimal to assure a narrow win/lose.

Tables 2 shows the values related to this experiment for the 10 best solutions found. First column provides an identifier for the specific solutions whose values are shown in the corresponding rows. Second column shows the game score obtained at the end of the execution. Columns 3rd to 6th display the values returned by the execution of the game specification: $b_{result} = 0$ if player wins and 1 otherwise), number of remaining lives (n_{rl}), number of small dots left (n_{rsd}), and remaining time of gameplay ($time_{rgameplay}$). Columns 7th to 11th show the values preset to the configuration parameters of the game specification before its execution: number of lives (nl), number of ghosts (ng), map fill ratio (fr), number of big dots (bd), and configured timeout in milliseconds (to).

According to the represented data, all of them assure a narrow victory of the opponent (i.e., the ghosts). This can be perceived by analyzing the reduced number of small dots (between 1 and 3) that remain on the map at the end of the game executions. This means that the player comes near securing victory before being defeated, and this was the main objective pursued in this experiment.

5   Conclusions

This article presents an automated solution to the problem of game tuning or game balancing, minimizing the programming effort for the development of game AI research experiments. This proposal does not require to have an executable version of the game, since it just requires an specification in a game description language. The definition of goals that will be part of the fitness function is also included in the specification itself, reducing the configuration of the entire process to a single file.

Using the new features of XVGDL$^+$and XGE$^+$, this approach is also a demonstration that these tools help significantly when carrying out research experiments of a diverse nature. In this case, the parameter optimization problem is solved, but potentially and in an analogous way, they can be used to carry out other types of experiments, such as the generation of game mechanics or content.

The proposal described in this paper abstracts researchers from making programming tasks and greatly simplifies the current lacks that are inherent to the optimization problem, such as the human fatigue during play-testing or subjectivity in evaluations. Researchers may just focus on the specification of the game and the requirements that they want to meet when evaluating the game executions, while programmers can focus on the implementation of the external components. Note that XVGDL$^+$offers structures to specify the use of external components during the game execution. These components have to be specifically implemented/coded beyond the game specification. However, if no external component is used and only XVGDL$^+$built-in features are employed, then no programming effort is required. In fact, XVGDL$^+$ can be viewed as a video game description language enriched with structures that construct bridges between the game specification and its executable version.

The experiments carried out use a classic 2D game as testbed. Like most existing video game description languages, arcade and 2D-type games are usually used in experiments. However, XVGDL has been shown to have advantages over other existing languages and potentially allows specifications for different types of games and to focus solely on one genre. Not being a scripting language as such, it allows, the development of external components. This gives a great potential when implementing, for instance, AIs for players and other components present in the game, making this approach attractive to be used for other future research.

The presented approach, aims to mitigate some of the disadvantages that exist in some works published to date. The game tuning process is carried out without the need to have a game already developed. Also, this solution is potentially generic. Given a specification of a game, the optimization or balancing process can be applied, without taking into account its nature, since the algorithm works directly with the game specification, instead of working on parameters of an already developed game or in development. This solution could significantly reduce the time and resource cost issues involved in underlying works.

In the detailed experiments, a total of 10000 evaluations inside the local search algorithm have been executed. This means that a significant limited portion of the search space has been explored (just a 0,73%). However, on the one hand, it has been guaranteed that the HC algorithm does not get stuck and explored different regions of the search space, ensuring that it does not find just a local optimal solution. On the other hand, the evolution process has finished with solutions (i.e., Pacman specifications) that clearly hold the objectives. This result not only proves the adequacy and validity of the approach, but also encourages us to persist in the research on game optimization. Several lines of work might be tackled in the future. One of them is to consider more powerful search techniques (perhaps population-based optimization techniques) and other games. In

addition, a crucial step of the automated process is how to generate new or modified versions of the game specifications based on the feedback information returned by XGE⁺. In this paper this step has simplified by considering the optimization of a Pacman specification with 5 parameters. Future works might consider the optimization of other (more complex) video game components such as rules, layouts or end conditions.

In accordance with the primary principles of Open Science, the sources of the XVGDL⁺ game Engine (XGE) and XGE⁺ are publicly available (see [Ruiz-Quiñones and Fernández-Leiva", 2022, Ruiz-Quiñones and Fernández-Leiva, 2022]). In addition, many details of XVGDL⁺ and a number of other video game specifications written in XVGDL⁺ can be found in [Ruiz-Quiñones and Fernández-Leiva, 2020b].

Acknowledgements

References

[Abbadi et al., 2015] Abbadi, M., Di Giacomo, F., Cortesi, A., Spronck, P., Costantini, G., and Maggiore, G. (2015). Casanova: A simple, high-performance language for game development. In *Serious Games*, pages 123–134, Cham. Springer International Publishing.

[Becker and Görlich, 2020] Becker, A. and Görlich, D. (2020). What is game balancing?-an examination of concepts. *ParadigmPlus*, 1(1):22–41.

[Browne, 2014] Browne, C. (2014). Evolutionary game design: Automated game design comes of age. *SIGEVOlution*, 6(2):3–16.

[Casanova, 2012] Casanova (2012). Casanova.

[Chong-U Lim and Harrell, 2014] Chong-U Lim and Harrell, D. F. (2014). An approach to general videogame evaluation and automatic generation using a description language. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8.

[Cotta and Fernández-Leiva, 2011] Cotta, C. and Fernández-Leiva, A. J. (2011). Bio-inspired combinatorial optimization: Notes on reactive and proactive interaction. In Cabestany, J., Rojas, I., and Joya, G., editors, *Advances in Computational Intelligence*, pages 348–355, Berlin, Heidelberg. Springer Berlin Heidelberg.

[David-Tabibi et al., 2010] David-Tabibi, O., Koppel, M., and Netanyahu, N. S. (2010). Genetic algorithms for automatic search tuning. *ICGA journal*, 33(2):67–79.

[DeLaurentis et al., 2021] DeLaurentis, D. A., Panchal, J. H., Raz, A. K., Balasubramani, P., Maheshwari, A., Dachowicz, A., and Mall, K. (2021). Toward automated game balance: A systematic engineering design approach. In *2021 IEEE Conference on Games (CoG)*, pages 1–8.

[Di Giacomo et al., 2017] Di Giacomo, F., Abbadi, M., Cortesi, A., Spronck, P., Costantini, G., and Maggiore, G. (2017). High performance encapsulation and networking in casanova 2. *Entertainment Computing*, 20:25 − 41.

[Duarte and Battaiola, 2017] Duarte, L. C. S. and Battaiola, A. L. (2017). Distinctive features and game design. *Entertainment Computing*, 21:83 − 93.

[Ebner et al., 2013] Ebner, M., Levine, J., Lucas, S. M., Schaul, T., Thompson, T., and Togelius, J. (2013). Towards a video game description language. In [Lucas et al., 2013], pages 85–100.

[Fullerton, 2014] Fullerton, T. (2014). *Game design workshop: a playcentric approach to creating innovative games*. CRC press.

[Jiang et al., 2016] Jiang, G., Zhang, D., Perrussel, L., and Zhang, H. (2016). Epistemic gdl: A logic for representing and reasoning about imperfect information games. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, IJCAI'16, pages 1138–1144. AAAI Press.

[Lavelle, 2017] Lavelle, S. (2017). open-source html5 puzzle game engine, http://www.puzzle-script.net/.

[Levine et al., 2013] Levine, J., Congdon, C. B., Ebner, M., Kendall, G., Lucas, S. M., Miikkulainen, R., Schaul, T., and Thompson, T. (2013). General video game playing. In [Lucas et al., 2013], pages 77–83.

[Love et al., 2008] Love, N., Hinrichs, T., Haley, D., Schkufza, E., and Genesereth, M. (2008). General game playing: Game description language specification. *Stanford University*.

[Lucas et al., 2013] Lucas, S. M., Mateas, M., Preuss, M., Spronck, P., and Togelius, J., editors (2013). *Artificial and Computational Intelligence in Games*, volume 6 of *Dagstuhl Follow-Ups*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

[Michalewicz and Fogel, 2004] Michalewicz, Z. and Fogel, D. B. (2004). *How to solve it - modern heuristics: second, revised and extended edition, Second Edition*. Springer.

[Patel and Carver, 2011] Patel, P. and Carver, N. (2011). Tuning computer gaming agents using q-learning. pages 581–588.

[Ruiz-Quiñones and Fernández-Leiva", 2018] Ruiz-Quiñones, J. and Fernández-Leiva", A. (2018). Xvgdl project.

[Ruiz-Quiñones and Fernández-Leiva, 2020a] Ruiz-Quiñones, J. and Fernández-Leiva, A. (2020a). Xml-based video game description language. *IEEE Access*, 8:4679–4692.

[Ruiz-Quiñones and Fernández-Leiva, 2020b] Ruiz-Quiñones, J. and Fernández-Leiva, A. (2020b). Xvgdl web.

[Ruiz-Quiñones and Fernández-Leiva", 2022] Ruiz-Quiñones, J. and Fernández-Leiva", A. (2022). Xvgdl game engine github repository.

[Ruiz-Quiñones and Fernández-Leiva, 2022] Ruiz-Quiñones, J. and Fernández-Leiva, A. (2022). Xvgdl schema.

[Schaul, 2014] Schaul, T. (2014). An extensible description language for video games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331.

[Sironi and Winands, 2017] Sironi, C. F. and Winands, M. H. (2017). On-line parameter tuning for monte-carlo tree search in general game playing. In *Workshop on Computer Games*, pages 75–95. Springer.

[Smirnov and Golkar, 2019] Smirnov, D. and Golkar, A. (2019). Design optimization using game theory. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pages 1–11.

[Smith et al., 2010] Smith, A. M., Nelson, M. J., and Mateas, M. (2010). Ludocore: A logical game engine for modeling videogames. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pages 91–98.

[Thielscher, 2010] Thielscher, M. (2010). A general game description language for incomplete information games. In Fox, M. and Poole, D., editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

[Unity Technologies,] Unity Technologies. Unity game development platform, https://unity.com/es.

[Wang and Zhang, 2021] Wang, W. and Zhang, R. (2021). Improved game units balancing in game design through combinatorial optimization. In *2021 IEEE International Conference on e-Business Engineering (ICEBE)*, pages 64–69.

[Yin, 2019] Yin, P. (2019). Research on design and optimization of game ui framework based on unity3d. In *2019 International Conference on Electronic Engineering and Informatics (EEI)*, pages 221–223.

[Zook et al., 2019] Zook, A., Fruchter, E., and Riedl, M. O. (2019). Automatic playtesting for game parameter tuning via active learning. *arXiv preprint arXiv:1908.01417*.

A   Technical execution details

The software of both XVGDL⁺and XGE⁺is developed in Java programming language, following new and current methodologies such as Continuous Delivery/Integration (CD/CI) or Test Driven Development (TDD), applying current programming principles, in addition of taking advantage of the advantages offered by new versions of the Java Development Kit (JDK).

For the correct execution of the software, it is necessary to have a Java virtual machine (JVM) installed on the local computer (at least with version 11), in addition to having the project correctly built and the *jar* files (java libraries) accessible from the terminal. The project is fully integrated with the *maven* build tool, so it is also necessary to have this tool if you want to compile and work with its source code.

The parameter optimization and tuning program can be launched through the following command from a terminal:

```
U-ƒ- Q‹e nZe-zPQ‹bO√S4q-q§C‹sNw1n <b\i U∟qli‡ƒL@√i e-<\-^i C‡eQq§\C^zi
‚! bezS\S‹-zSb^i X-~^<Pd-<\-^aezS\S‹-zSb^
```

The entire optimization process is auto-configured (both tuning parameters and the XVGDL specification of the *Pacman* game that is used as a basis for optimization). The process, during its execution, shows information traces in the console/terminal, so that its evolution can be followed as solutions are evaluated.

On the other hand, at the end of it, a file containing each of the solutions found with its associated fitness value is stored in CSV format, so that it is easily interpretable by any spreadsheet-type management software. The file is made up of lines of text, each one represents a row in the CSV format. On each line, the following information is represented, separated by the $ character:

– Execution number.

– Number of enemies.

– Small dots filling percentage.

– Number of big dots.

– Number of initial player lives.

– Game maximum play time.

– Wining game.

– Number of small dots left on the map after finishing the game.

– Number of lives left for the player.

– Final game play time.

– Fitness function evaluation for this solution.

An example of the actual format of the output is as follows:

```
cAl A(_AJA(A(QE‹cDAzq~CAQEA(A_l AJŒ_J
```

Following the open source guidelines, the repository where the XVGDL project files and experiments are stored is available through GitHub. The reader interested in the implementation details, can access through a browser at the corresponding URL [Ruiz-Quiñones and Fernández-Leiva", 2018].

B XVGDL+ specification for Pacman

```
1   Zm4\Y ƒCqsSb^Enci OE5 C^<b@S^LEn}yGODn sz-^@-Yb^OEn^bn nN
2   ZFCQ ?OHS^SzSb^ bH e-<\-^ L-\C Hoq bezS\S<-zSb^ CON

3
4   ZL-\C?OHS^SzSb^N
5     ZeqpeCqz%JO%EnqC^@CqCq; b^HSL~q-zSb^n ƒ-Y~OEnw<b^zC‡zw
              e-<\-^, s<SSpC^@CqCq; b^HSL~q-zSb^i ‡\Yn wN
6     ZeqpeCqz%JO%EnzS\Co~zn ƒ-Y~OEn|OEEE5 wN
7
8     ZY-%o~zN
9       Z<b\eb^C^z S@EnL-\CR^Hoyben Yb<-zSb^EnzbenN
10        Z<b^zC‡zdqpeCqz%S@Ens<bqCn wN
11        Z<b^zC‡zdqpeCqz%S@EnPSr<bqCn wN
12      Zvw<b\eb^C^zN
13
14      Z<b\eb^C^z S@EnL-\CR^Hb?b.^n Yb<-zSb^En4bzzb\nN
15        Z<b^zC‡zdqpeCqz%S@EnYSƒCsn wN
16        Z<b^zC‡zdqpeCqz%S@EneY-%GqRzC\sn wN
17      Zvw<b\eb^C^zN
18    ZvwY-%o~zN
19
20    Z<b^zqpYsN
21      Z<b^zqpY YOH~zEn|un qSLPzEn|vn ~eEn|Jn @b.^En|IwN
22    Zvw<b^zqpYsN
23
24    Z\-e z%o&OEn|?n sS<C‡Encln sS<C~En|Dn zbqpS@-YEnH-YsOn
25      LC^Cq-zbqEn<b\iUq|i‡ƒL@Yie-<\-^i\b@CYi\-ei
              GSYC3-sO@K-\C[-eKC^Cq-zbqn
26      HSYOEnw<b^zC‡zwbezS\S<-zSb^we-<\-^[-eaezS\S<-zSb^iz‡znN
27    Zv\x-eN
28
29    ZeY-%o&qs ^-\4CqEncn \-‡]~\4CqEncn \S^]~\4CqEncnN
30      ZeY-%o&q ^-\OEne-<\-^n s<bqCEnOE5 YSƒCsEn{n YSƒCdCq<C^z-LOEncOEE5
              -SEn<b\iUq|i‡ƒL@Yie-<\-^i\b@CYi b4UC<zi-Sid-<\-^,RnwN
31    ZvweY-%o&qsN
32
33    Zb4UC<zsN
34      Zb4UC<z ^-\OEn.~.YYn z%o&OEn.~.YYn @%o~\S<EnH-YsOn ƒbY-zSYOEnH-YsC
              n sS<C‡Encn sS<C~Encn wN
35      Zb4UC<z ^-\OEnLPbszn z%o&OEnC^C\%o n @%o~\S<Enzq~On ƒbY-zSYOEnzq~C
              n sS<OEnc>cn S^sz-^<OsEnJn -SEn<b\iUq|i‡ƒL@Yie-<\-^i
              \b@CYi b4UC<zi-SiKPbsz,Rn wN
36      Zb4UC<z ^-\OEns\-YY?bzn z%o&OEnSzC\n @%o~\S<Enzq~On ƒbY-zSYOEn
              zq~On sS<OEnc>cn S^sz-^<OsEn|n wN
37      Zb4UC<z ^-\OEn4SL?bzn z%o&OEnSzC\n @%o~\S<Enzq~On ƒbY-zSYOEnzq~C
              n sS<OEnc>cn S^sz-^<OsEn|n wN
38    Zvwb4UC<zsN
39
40    Zq~YOsN
41      Zq~YC ^-\OEnC-zr\-YY?bzn z%o&OEn<bYYSsSb^n L-\Crz-zOEn@OH-~Yz>
              e-<\-^db.Cq}enN
42        Zq~YC, <zSb^sN
43          Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n qCs~YzEns<bqC}en ƒ-Y~OEn
                  cOEE5wN
44          Zq~YC, <zSb^ b4UC<z]-\OEns\-YY?bzn qCs~YzEn@Ss-eeC-qnwN
45        Zvwq~YC, <zSb^sN
```

```
46        Zvvq~YCN
47
48        Zq~YC ^-\OEnC-z3SL?bzn z%&OEn<bYYSsSb^n L-\Orz-zOEn@CH-~Yz>
                 ⌐ e-<\-^db.Cq}enN
49           Zq~YC, <zSb^sN
50            Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n ƒ-Y~OEn{OEEni qCs~YzEmn
                    ⌐ <Y-ss]-\OEn<b\iUq|i‡ƒL@Yie-<\-^i\b@CYiq~YCsi
                    ⌐ d-<\-^db.Cq}ep~YC, <zSb^nwN
51               Zq~YC, <zSb^ b4UC<z]-\OEn4SL?bzn qCs~YzEn@Ss-eeC-qnwN
52           Zvvq~YC, <zSb^sN
53        Zvvq~YCN
54
55        Zq~YC ^-\OEnLPbosz; -z<Pd-<\-^n z%&OEn<bYYSsSb^nN
56           Zq~YC, <zSb^sN
57            Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n qCs~YzEnS^SzS-YdosSzSb^n
                    ⌐ wN
58            Zq~YC, <zSb^ b4UC<z]-\OEnLPboszn qCs~YzEn
                    ⌐ LCzB^C\SCsR^p-@S~sfSzC\gnwN
59           Zvvq~YC, <zSb^sN
60        Zvvq~YCN
61
62        Zq~YC ^-\OEnLPbosz; -z<Pd-<\-^XSƒCs?b..^n z%&OEn<bYYSsSb^nN
63           Zq~YC, <zSb^sN
64            Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n qCs~YzEmn <Y-ss]-\OEn<b\i
                    ⌐ Uq|i‡ƒL@Yie-<\-^i\b@CYiq~YCsi
                    ⌐ d-<\-^;PC<VlXSƒCs?b..^p-YC, <zSb^nwN
65            Zq~YC, <zSb^ b4UC<z]-\OEnLPboszn qCs~YzEmnwN
66           Zvvq~YC, <zSb^sN
67        Zvvq~YCN
68
69        Zq~YC ^-\OEne-<\-^; -z<PKPboszn z%&OEn<bYYSsSb^n L-\Orz-zOEn
                 ⌐ e-<\-^db.Cq}enN
70           Zq~YC, <zSb^sN
71            Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n qCs~YzEns<boqC}en ƒ-Y~OEn
                    ⌐ cOEEEwN
72            Zq~YC, <zSb^ b4UC<z]-\OEnLPboszn qCs~YzEnS^SzS-YdosSzSb^n
                    ⌐ wN
73           Zvvq~YC, <zSb^sN
74        Zvvq~YCN
75        Zq~YC ^-\OEne-<\-^; -z<PKPbosz CqCC.CKPboszn z%&OEn<bYYSsSb^n
                 ⌐ L-\Orz-zOEne-<\-^db.Cq}enN
76           Zq~YC, <zSb^sN
77            Zq~YC, <zSb^ b4UC<z]-\OEne-<\-^n qCs~YzEmn wN
78            Zq~YC, <zSb^ b4UC<z]-\OEnLPboszn qCs~YzEnHqCC.Cn ƒ-Y~OEn
                    ⌐ lOEEEwN
79           Zvvq~YC, <zSb^sN
80        Zvvq~YCN
81     Zvvq~YCsN
82
83     ZL-\Orz-zCsN
84        ZFCO.?CH-~Yz L-\C sz-zC OCN
85        ZL-\Orz-zC ^-\OEn@CH-~YznwN
86        ZFCO.K-\C sz-zC qCeqCsC^zS^L zPC d-<\-^ db.Cq }e
87         Q.R^<qC-sC d-<\-^ reCC@
88         Q.d-<\-^ <-^ <-z<P LPboszs
89        OCN
90        ZL-\Orz-zC ^-\OEne-<\-^db.Cq}enwN
91     ZvvL-\Orz-zCsN
```

```
92
93     ZC^@,b^@SzSb^sN
94       ZC^@,b^@SzSb^ <PC<WCq;Y-ssEn<b\iUq|i‡ƒL@Yi\b@CYi C^@<b^@SzSb^i
              ! ]ba4UC<zsdqCsC^zK-\CB^@,b^@SzSb^n b4UC<z]-\CsEn4SL?bz>
              ! s\-YY?bz.n .S^^S^L;b^@SzSb^Enzq~OnwN
95       ZC^@,b^@SzSb^ <PC<WCq;Y-ssEn<b\iUq|i‡ƒL@Yi\b@CYi C^@<b^@SzSb^i
              ! XSƒCsŠCqbK-\CB^@,b^@SzSb^nwN
96     ZvwC^@,b^@SzSb^sN
97
98     Zb4UC<zSƒCsN
99      ZFCD,,^ CtzCq^-Y <b\eb^C^z zP-z <-Y<-Y-zCs zPC b4UC<zSƒCs s<bqC
              ! CDN
100      Zb4UC<zSƒC <Y-ss]-\CEn<b\iUq|i‡ƒL@Yie-<\-^ib4UC<zSƒCsi
              ! d-<\-^rbY~zSb^Bƒ-Y~-zSb^n wN
101    Zvwb4UC<zSƒCsN
102
103    ZvwL-\C?CH-S^SzSb^N
```

C   Pacman AI

An AI component has been developed for the virtual player (**e-<\-^**) to enable the game to be executed automatically, so that the search algorithm can be run without human intervention.

Algorithm input:

The **e-<\-^** AI takes as input the following parameteres:
- The player position.
- A list of the position of each enemy.
- The positions of all the items and walls on the map.
- A search radius to limit the number of possible movements to explore.

The algorithm: - For each of the items on the map within the specified radius, the algorithm calculates the distance **?** to each of them. This distance is calculated as the minimum number of steps that **e-<\-^** has to take to get to the destination position. This is the distance between the current **e-<\-^** position and the target position, taking into account the walls on the map.

- If there are no items within the specified radius, this is incremented by one in order to try to find the nearest possible item (target) on the map, preventing the algorithm from returning any movement.

- A distance is calculated for each of the items in the search radius, the algorithm calculates the number of enemies and also the enemies surrounding using the same radius input parameter. A score for each of the possible movements is calculated using the distance, the number of enemies in the path to the target item, and the number of enemies surrounding the target item. Each of the enemies in the path and the enemies surrounding the target item penalizes that possible movement.

- Being D the distance from the **e-<\-^** to a target item t for a given movement m, ep and es the number of enemies in the path and enemies surrounding the target item, the final score for that possible movement is calculated as detailed in Equation 4:

$$score(m; t) = D + 100 \quad ep + 10 \quad es \tag{4}$$

- In the case the game is in **d-<\-^ db.Cq }e** state, the enemies factor is not taken into account to calculate the score for a target item.

Algorithm output:
- List of possible movements arranged from the most promising options according to the aforementioned formula.

D   Ghosts' AI

Two different AIs have been implemented for the **LPbszs** (enemies) in the game.
- **Ghosts chasing Pacman:** This AI makes **LPbszs** chase **e-<\-^** all over the map. The aim of each movement will be to get closer to **e-<\-^** using the minimum distance from the **LPbszs** object to the **e-<\-^** object. As **e-<\-^** is continuously moving on the map, this AI tries to avoid using a path that has been used in the previous movements so the **LPbszs** will not get stuck or repeat movements. In the case the game is in **d-<\-^ db.Cq }e** state, then this AI make the ghosts try to escape from **e-<\-^**.
- **Ghosts moving randomly:** This simple AI just makes ghosts to move randomly over the map. The AI makes the **LPbszs** to try to follow a path, preventing them from moving repeatedly from a position to a contiguous cell on the map, but it does not take into account the **e-<\-^** position.