Procedural Generation of Quests for Games Using Genetic Algorithms and Automated Planning

Edirlei Soares de Lima
School of Design, Technology and Communication
Universidade Europeia
Lisbon, Portugal
edirlei.lima@universidadeeuropeia.pt

Bruno Feijó, Antonio L. Furtado
Department of Informatics
Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro, Brazil
{bfeijo, furtado}@inf.puc-rio.br

***Abstract* — The production of high-quality commercial games requires the work of a few hundred individuals, including designers, artists, and story writers, to produce game content, such as 3D models, textures, and narratives. Over the last decade, the production of game content has grown to the point of becoming a bottleneck in companies' schedules and budgets. In this context, procedural content generation techniques are increasingly being applied to reduce the work overload of the development teams. Although game developers and academic researchers have extensively explored procedural content generation, there is a lack of techniques to handle procedural generation of quests. In this paper, we present a new quest generation method based on genetic algorithms and automated planning. By combining planning with an evolutionary search strategy guided by story arcs, the proposed method can generate coherent quests based on a specific narrative structure. Preliminary results show that quests created with our method are nearly at par with those created by game design professionals.**

Keywords – quest generation; genetic algorithms; planning; interactive storylling;

I. INTRODUCTION

In the game industry, the production of high-quality commercial games requires the effort of a few hundred individuals, including artists, designers, programmers, and story writers, many of whom work mainly to produce game content, such as 3D models, textures, environments, stories, and quests [10]. Over the last decade, the production of game content has grown to the point that it has become a bottleneck in companies' schedules and budgets [11]. As a solution to this problem, many game development companies are employing Procedural Content Generation techniques (PCG) to reduce the work overload of the development teams. PCG involves the use of algorithms for the generation of game content with limited or no human contribution [12]. In this way, game content is not generated manually by human designers, but by computers executing well-defined procedures that can be adjusted to match the vision and objectives of artists and designers.

With more than three decades of research, PCG methods have been applied for the generation of many types of game content (see [10] and [13] for extensive surveys of PCG techniques). Today, PCG techniques are even presented as key features and selling points of some games, such as *No Man Sky* (2016), which featured a procedurally generated open world universe composed of over 18 quintillion planets with their own ecosystems and unique forms of fauna and flora. Tools for PCG are also very common nowadays, such as the SpeedTree system, which has been used for the generation of trees, grass and other types of vegetation in hundreds of commercial games, such as *Horizon Zero Dawn* (2017), *Far Cry 5* (2018), and *Forza Horizon 4* (2018).

Although PCG has been extensively explored by game developers and academic researchers, there is a lack of techniques to handle procedural generation of quests. Computer Role-Playing Games (RPGs) usually employ quests as a fundamental mechanism for narrative progression, which provides players with concrete goals that guide the gameplay. The computational generation and adaptation of narratives is the objective of a specific type of PCG oriented towards Interactive Storytelling, a promising research area dedicated, since the 1970s, to exploring narrative generation (see [14] for a survey on narrative generation methods). However, most of the existing narrative generation methods are not designed to handle dynamic game environments. Although it is fair to recognize that narrative generation methods have significantly evolved in the last decades, we are still far from having algorithms capable of producing complex and creative stories with the quality of those created by professional writers.

Quest generation for games involves several challenges, especially regarding the dynamic interaction between player, environment, and story, where the logical coherence must be a primary concern. Indeed, even relatively unimportant sidequests (quests that usually have no effects in the main storyline of the game), must affect future sidequests or game events that involve the same characters, objects or places (e.g. if a character dies in one quest, he cannot appear alive in a future quest without any justification). Another essential requirement for a quest generation system is related to its ability to generate diversified plots. The repetition of gameplay elements is known for causing frustration on players [16]; therefore, it is important to guarantee the right level of variety among the generated quests.

In this work, we aim at some of the challenges faced by quest generation methods, especially how to achieve the ability to handle dynamic game environments, guarantee the logical coherence of the story, and generate diversified plots. To accomplish this goal, we propose a new quest generation method based on genetic algorithms and automated planning. By combining planning with an evolutionary search strategy guided by story arcs, the proposed method can generate coherent quests based on a specific narrative structure. The main objective of this paper is to present our method and to validate the produced results by analyzing how close the generated quests are to those created by human game designers.

The paper is organized as follows. Section II describes related work. Section III introduces the proposed method for quest generation and describes the implementation details of the genetic algorithm. Section IV describes an application of the method in a game prototype. Section V presents an evaluation of the method. Section VII offers concluding remarks.

II. Related Work

There are several works on quest generation in the literature, such as the generic framework presented by Sullivan et al. [7]. In their system, a game manager uses the player history and the current state of the world to dynamically generate and alter the structure of quests. Their rule-based system works on a library of quests and can dynamically recombine them upon generation. Another recent framework for quest generation was proposed by Breault et al. [9], who present a quest engine that relies only on automated planning to create quests. Their system takes in a world description represented as a set of facts and generates quests according to the state of the world using a deterministic planning algorithm. A more dynamic solution is presented by Lima et al. [8], who propose a method to generate quests based on hierarchical task decomposition and planning under non-determinism. Their approach combines planning, execution, and monitoring to handle nondeterministic events and support quests with multiple endings that affect the game's narrative and create interactive and dynamic story plots.

Although, to the best of our knowledge, no previous work has directly explored the use of genetic algorithms for quest generation, we can find some related works that use genetic algorithms for general narrative generation. An example is the work of McIntyre and Lapata [2], which describes a story generator system that employs an evolutionary search strategy where each plot is represented by an ordered graph of dependency trees (corresponding to sentences of the narrative text). Since their algorithm works directly with text sentences, the genetic operations were designed to handle the syntax and semantics of the sentences, such as the mutation, which can occur on any verb, noun, adverb, or adjective in the sentence. If a noun, adverb or adjective is chosen to undergo mutation, it will be replaced with a new lexical item that is sufficiently similar. Their algorithm uses a fitness function that scores candidates based on their coherence.

The use of story templates and graphs to provide information to a genetic algorithm was also explored in previous works [1][4]. Ong and Leggett [1] present a system for narrative generation that uses a genetic algorithm to recombine story components created from a set of story templates. In their algorithm, the fitness of a given story is determined by the events, which must be previously rated by the author. The basic structure of the narrative is ensured by rules and conditions described in pre-determined templates. In a similar context, Giannatos et al. [4] present a system that uses an evolutionary algorithm to suggest new story events that, if added to an existing story graph, would improve the quality of traversals of the story space. Their algorithm constructs story graphs with candidate plot points and then samples 100 possible playthroughs from the story graph. The sampled stories are then rated according to three criteria: spatial locality, thought flow and motivation. The final fitness is computed as the average of the fitness values obtained from evaluating all playthroughs.

A different approach is explored by Nairat et al. [5][6], where, instead of following a plot-based approach, they integrate evolutionary methods in a character-based system. They propose a method for generating stories that integrates an agent-based system where the characters are created using an interactive genetic algorithm. In their method, each individual of the population is represented by internal states (resources and emotions) and actions rules that define the personality and behavior of the agent. The author is responsible for observing and selecting the agents whose behaviors are relevant for the intended story. When the author decides that the characters are interesting enough to develop further, the system performs the reproduction process to create a new generation.

The combination of genetic and planning algorithms was also previously explored by Giannatos et al. [3], who proposed a method that uses genetic algorithms to generate plan operators representing possible story actions. As is common in planning-based storytelling systems, they represent the story world using predicates and action operators (defined by pre and post-conditions). The goal of their genetic algorithm is to generate plan operators as narrative units for constructing new story plots. In order to evaluate the solutions, they use a fitness function that estimates the operator's contribution to generate suspenseful stories. Although the work of Giannatos et al. [3] explores the combination of genetic and planning algorithms as proposed in this work, they use genetic algorithms only to create new operators (parameters, preconditions, and effects), without changing the initial state and goals as we do in the present paper. In the context of a game, it would not always be feasible to change existing operators, as each operator results in a different mechanism that must be implemented in the game. An evident difficulty of their approach is that it does not generate meaning for the operators. Their algorithm only generates preconditions and effects that are logically valid, but it is not explicit what kind of action/event the operator represents.

Although quest and narrative generation has been extensively explored by academic researchers in game and non-game contexts, most of the methods require a library of existing quests (e.g.: [1][7]) or highly detailed descriptions of planning problems with predefined goal states (e.g.: [8][9]), which demands extra authorial work and restricts the space of possible quests. In such context, the use of genetic algorithms has been limited to non-game applications, where event structures are not necessary (e.g. [2][5][6]) or other methods are responsible for the actual narrative generation (e.g. [3][4]).

III. Quest Generation

In traditional literature, a quest represents a journey towards a specific mission or goal, where multiple adventures can occur. According to Propp's analysis of folktales [17], the main adventure begins when a villainy is committed or a lack is recognized, after which the hero departs on a perilous journey, culminating with a struggle against some sort of adversary. But other adventures often occur, typically in a preliminary phase wherein the villain makes preparations, and even after the hero's victory the quest may not terminate.

In computer games, quests also represent missions or objectives to be accomplished by avatars (i.e. game characters controlled by human players). However, differently from the well-established theory about quests in literature, there is no general quest theory in computer games, mainly because the concept of narrative in videogames is not properly defined. There are many contradictory opinions about the videogame's narrative: Aarseth [18] describes it as a "post-narrative discourse"; Tosca [19] claims that games are not narratives (and only after the quest has been completed, it can be narrated as a story); and Jenkins [20] defends a hybrid concept between games and narratology.

In this paper, we consider that a quest is defined by a set of tasks to be accomplished by the player (e.g. gathering and delivering items, killing enemies, protecting and saving characters). This set of tasks represents the plot of the quest (i.e. storyline). Although each quest has its own plot (with beginning, middle, and end), all quests take place in the same world; therefore, changes in the world state can have implications in the plot of future quests (e.g. if the player opens the gate to a secrete place as part of the tasks of one quest, there is reason to make it a recurring task in future quests – except if there was an intermediary quest that required the player to close the gate).

Game worlds are often populated by dozens of non-player characters, amid thousands of objects, items, enemies and places. In addition, players can interact with the world by performing dozens of different actions, which increases the complexity of finding a "good" sequence of tasks for a quest. In this type of problem, the search space would normally contain thousands of nodes that could produce tens of thousands of possible quests. Searching in this complex space is a difficult optimization problem that must satisfy several constraints, typically related to the logical coherence of the events, the overall narrative structure, and the length of the quest. In this context, we argue that a genetic algorithm is advantageous as it can search and find good solutions in a more efficient way than traditional search methods.

As illustrated in Figure 1, the architecture of the proposed quest generator system is composed of two subsystems: (1) the Offline Quest Generator (OQG), which is responsible for running the genetic algorithm and handling the results (generated quests); and (2) the Game Manager (GM), which handles the execution of quests while the player interacts in real-time. The OQG runs in a preprocessing phase (offline) and provides the GM with the full set of quests generated for the game. As part of the OQG, the Genetic Algorithm module implements all methods of a traditional genetic algorithm and handles the execution of Quest Planners to validate the generated quests. Quests are generated based on information stored in a Domain Database, which includes the definition of valid quest events, characters, places and objects that are part of the game world. While players interact with the game, their actions are used to update the World State, which is used in turn by the Quest Manager to keep track of the player progression as he/she performs quests. Meanwhile, the Player receives help from the Player's Assistant, who provides tips about the next objectives of the current quest.

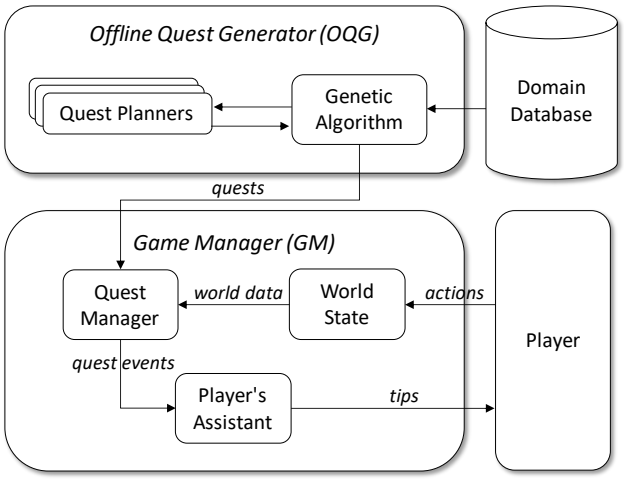

Figure 1. Architecture of the quest generator system.

A. Genetic Algorithm

The theory of evolution by natural selection proposed by Charles Darwin in his famous book "*On the Origin of Species*"[1] states that all life forms that exist today are the result of millions of years of adaptation caused by demands of the environment (cf. [22]). According to this theory, at any given time, several different organisms may co-exist and compete for the same resources in an ecosystem. The organisms that are more capable of acquiring resources and successfully reproducing will have more descendants in the future [15]. Organisms that are less capable, will tend to have few or no descendants (less chance to survive). Over time, the entire population of the ecosystem will evolve to contain organisms that are more fit and adapted to their environment than those of previous generations (promoting the *survival of the fittest*).

A genetic algorithm abstracts these evolutionary principles into computational processes that can be used to search for optimal solutions of a problem. In a traditional implementation, an initial population of individuals (also called chromosomes) is randomly generated, where a genetic structure represents each individual. This structure is usually defined by a fixed-length bit string (but many problems – including ours – require more complex structures), where each position in the string represents a feature of the individual (an analogy to the genes found in DNA of biological organisms). Each individual of the population is evaluated according to a fitness function. After being evaluated, a certain number of individuals are selected to be parents (according to their fitness) and undergo crossover (also called reproduction or recombination) and mutation processes in order to produce a new and evolved population (the new chromosomes are called offsprings). Since offsprings usually combine genes from good individuals, they are expected to be better than their parents. Therefore, offsprings are inserted into the new population to replace the inferior individuals of the previous generation. This process is repeated until a given termination criterion triggers (usually a given number of generations). The repetition leads to the evolution of the population. At the end, the fittest individuals represent the best solution to the problem.

[1] http://www.gutenberg.org/files/1228/1228-h/1228-h.htm

As illustrated in Figure 2, our method involves the main steps of a traditional genetic algorithm (initialize population, evaluation, selection, crossover, mutation) with the addition of an extra control loop to manage the generation of sequential quests. By updating the world state (that is stored in the Domain Database) with information extracted from the final state of the a previously generated quest, the algorithm can guarantee the logical coherence of sequences of quests.

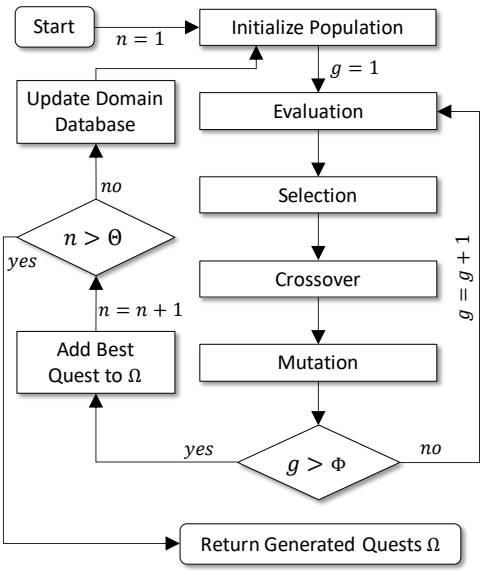

Figure 2. Overview of the proposed genetic algorithm, where Φ is the maximum number of generations and Θ is the number of sequential quests to be generated.

In our genetic algorithm, each individual (i.e., each chromosome) represents a quest, which is encoded as a planning problem:[2]

$$Q = (P, S_0, G, O),$$

where P is a set of *atomic formulas* (or *atoms*, for short), O is a set of planning operators, S_0 is the initial state of the quest, and G is the goal state, such that:

- An *atom* is an expression of the form $pred(r_1, \cdots, r_k)$, where *pred* is a predicate symbol and $r_1, \cdots, r_k$ are variable terms (e.g. `CH1` and `IT`) or ground terms (e.g. `player` and `antidote`);
- A *literal* is an atom p or the negation of an atom, $\neg p$, letting negation represent the deletion of the proposition from the current world state *S* (i.e. we use the *closed-world assumption*: a proposition that is not explicitly specified in a state does not hold in that state); and
- $S_0 \subseteq P$ and $G \subseteq P$ are sets of ground literals.

An operator $o \in O$ is denoted by

$$o = \big(name(o), precond(o), effect(o)\big),$$

where:

- $name(o)$ is the name of the operator, which is an atom $op(x_1, x_2, \ldots, x_k)$, where op is a unique symbol called an operator symbol, and x_i is a variable symbol that occurs anywhere in o;
- $precond(o)$ is a set of literals that define the preconditions of o (i.e. the positive and/or negative literals that must be true in order to use the operator o); and
- $effect(o)$ is a set of literals that define the effects of o (i.e. the positive and/or negative literals the operator o will cause to hold).

The genetic structure of an individual comprises *schematic* and *reactive* elements. Both P and O are schematic and defined as part of the conceptual schema of the domain (game world). While O defines possible types of events that can occur in the course of a quest, P establishes valid atoms to be used to describe states, goals, and operators. These domain elements are schematic and used to compose the planning problems of all individuals. On the other hand, S_0 and G (initial state and goal state) are reactive and can vary from one individual to another, expressing different ways to perceive a situation and the impulse to change it according to character-dependent preferences. Therefore, S_0 and G are the elements that define the distinguishing features of each individual and, consequently, they are the elements submitted to the crossover and mutation operations. Considering that these elements are sets of ground literals with no fixed length, they are represented in the genetic structure of the individuals by two linked lists (in contrast to more traditional implementations that use fixed-length bit strings).

The following example illustrates the representation of an individual in our genetic algorithm (for the sake of simplicity, we omitted operators and atoms that were not relevant for this example):

```
Schematic:
P: at(C, P), has(C, P), hero(C), healthy(C),
   isantidote(I), alive(C), infected(C),
   cured(C), path(L1, L2)
o1:
  name: go(CH, PL1, PL2)
  precond: character(CH), place(PL1),
           place(PL2), at(CH, PL1),
           alive(CH), hero(CH),
           path(PL1, PL2)
  effect: at(CH, PL2), ¬at(CH, PL1)
o2:
  name: get(CH, IT, PL)
  precond: character(CH), item(IT),
           place(PL), at(CH, PL), alive(CH),
           hero(CH), at(IT, PL)
  effect: has(CH, IT), ¬at(IT, PL)
o3:
  name: attack(ZO, CH, PL)
  precond: zombie(ZO), character(CH),
           place(PL), at(CH, PL),
           at(ZO, PL), alive(CH),
           healthy(CH)
  effect: infected(CH), ¬healthy(CH)
o4:
  name: cure(CH1, CH1, IT, PL)
  precond: character(CH1), character(CH3),
           item(IT), place(PL), at(CH1, PL),
           at(CH2,PL), alive(CH1), hero(CH1)
           alive(CH2), infected(CH2),
           has(CH1, IT), isantidote(IT)
  effect: cured(CH2), healthy(CH2),
          ¬infected(CH2), ¬has(CH1, IT)
```

[2] Notation closely adapted from planning theory [21].

```
Reactive (S0 with 20 genes and G with 4):
S0: character(john), character(anne),
    place(village), place(johnhome),
    place(store), item(antidote1),
    isantidote(antidote1), hero(john),
    alive(john), alive(anne), healthy(john),
    healthy(anne), enemy(zombie),
    path(johnhome, village), path(village,
    johnhome), path(store, village),
    path(village, store),at(john, johnhome),
    at(anne, johnhome),at(antidote1, store)
G: cured(anne), healthy(anne),
   at(john, johnhome), alive(john)
```

In the above example, the schematic elements define the domain of the planning problem, which includes the vocabulary of atoms used to describe the problem (P) and four operators based on the well-known STRIPS formalism that represent possible events for the quest (`go`, `get`, `attack`, `cure`). In the reactive part of the individual, we can notice that the initial state of the quest (S_0) defines that `john` and `anne` are characters; `john` is the hero of the story; `johnhome`, `village` and `store` are places; `john` and `anne` are both at `johnhome`, alive and healthy; `antidote1` is an item that is at the `store`; there is a `zombie` that is an enemy; and there is a path connecting `johnhome` with the `village` (amongst other paths connecting places). The goal state (G) establishes that `anne` must be cured and healthy, and `john` must be alive and at `johnhome`.

When the planning problem of an individual is solved by a planner, the plot for the quest is established as a linear sequence of events or tasks to be accomplished by the player. In the above example, the plot comprises:

```
attack(zombie, anne, johnhome), go(john,
johnhome, village), go(john, village,
store), get(john, antidote1, store),
go(john, store, village), go(john, village,
johnhome), cure(john, anne, antidote1,
johnhome).
```

Individuals for the initial population of the genetic algorithm are randomly generated according to the information defined in the Domain Database (DB),[3] which is a set:

$$DB = \{\mathrm{A}, \mathrm{B}, \Gamma, \Delta, \mathrm{E}\},$$

where:

- A is a set of pairs $\alpha_i = (obj_i, objType_i)$ that defines all **objects** of the game world (obj_i) and associate them with a specific object type ($objType_i$). For example, A= {(`john`, `character`), (`home(john)`, `place`), (`antidote1`, `item`)} defines that *john* is a *character*, *home*(*john*) is a *place*, and *antidote1* is an *item*. Currently, for the sake of simplicity, we represent function symbols as constants (e.g., we use *johnhome* instead of *home*(*john*));
- B is a set of ground literals that describe **properties and relations** of objects that exist in the game world, e.g.: B = { `path(johnhome, village)`, `alive(john)`, `at(john, johnhome)`, `isantidote(antidote1)`};
- Γ is a set of **semantic integrity constraints** on predicates $pred_i$, which plays a central role in the chromosome generation process of our algorithm. Currently, we have three constraints types: variable types, contradictory or opposite relations, and existential uniqueness quantification. That is, each member of the set Γ is $\gamma_i = (pred_i, opp_i, (u_1 t_1, \cdots, u_n t_n))$, where $pred_i$ is a predicate symbol with *n* terms (e.g. `at`, `has`, `cured`), opp_i is the opposite predicate of $pred_i$, t_j denotes the object type ($objType$) that is required for the j-th ground term of $pred_i$ to produce a valid instance of the predicate, and u_j indicates a uniqueness quantification $\exists! t_j\ pred_i$ (i.e., there exists exactly one t_j such that $pred_i$ is true). For example, $\gamma_i = (cured, (character))$ indicates that the predicate *cured* has a single term that must be of type *character*. An example of opposite relation is $\gamma_i = (healthy, infected, (character))$, which denotes that either `healthy(john)` holds or `infected(john)` holds, but not both. An example of uniqueness of a term is $\gamma_i = (at, (character, \exists! place))$, which denote that each character can only be at one place at a time. In this later case, the predicate `at(john,…)` can appear only once in a state for the character *john*;
- Δ defines a set of **planning operators** based on the STRIPS formalism (with preconditions and effects) that represents all possible events that can occur in the course of a quest; and
- E is a set of pairs $\varepsilon_i = (o_i, tension_i)$ that establishes how each operator o_i affects the overall **tension of the quest** ($tension_i$), which can be increased (+), decreased (-), or maintained (=). For example, the `attack` operator creates tension and the `cure` represents a resolution that reduces the tension, therefore: E= {(`attack`, +), (`cure`, −), …}.

In order to create the initial population for the genetic algorithm, planning problems are generated to represent the individuals of the population. For the schematic elements of each planning problem, P is defined by the unique atoms found in A ⊆ DB and B ⊆ DB, and O is created with the planning operators defined in Δ ⊆ DB. For the reactive elements, first the initial state of the quest (S_0) is initialized with the schematic elements of the game world (A ⊆ DB), and then a number of new ground literals are randomly generated (according to the semantic integrity constraints established in Γ ⊆ DB) and added to S_0. Similarly, a random number of ground literals are generated for the goal state (G). The process to generate random ground literals automatically avoids adding repeated ground literals to the same state and ensures the uniqueness of literals that contain specific terms that must appear only once in a state (according to Γ ⊆ DB). The number of individuals of the population and the total number of ground literals to be added to S_0 and G are defined through parameters of the genetic algorithm. In our experiments, we use populations

[3] In our implementation, DB is defined in an XML file. An example of database used in our prototype is available at: http://www.icad.puc-rio.br/~logtell/geneticquest_db.xml

of 100 individuals and random values in the range [1, 30] to define the number of ground literals of S_0 and [1, 10] for G.

After generating the initial population, individuals are evaluated according to the story arcs of their plots (see subsection B for details about the fitness function). Then, to select individuals for reproduction, we used the fitness proportionate selection method (also known as roulette wheel selection [22]), which uses the individuals' fitness to calculate a selection probability, allowing candidates to be selected randomly but with a bias towards those with a larger proportion of the population's combined fitness. In addition, to avoid decreasing the quality of solutions from one generation to the next, we also applied the elitist selection strategy [15], where a limited number of individuals with the best fitness values are copied directly to the next generation (in our experiments, we use an elitism factor of 2%).

Once the individuals are selected for reproduction, the next step of the algorithm is the crossover process, which combines the genetic information of two parents to generate new offspring. As illustrated in Figure 3, we use a single point crossover on each reactive element of the individuals (S_0 and G). Therefore, two crossover points are randomly selected, one along the length of the initial state of the parents (S_0) and another one along the length of goal state (G). When the length of the states does not match, the point is selected along the smallest length. Then, ground literals of the parents' states on each side of the crossover points are swapped and copied to two different children (Figure 3). While copying ground literals to a new child, the algorithm automatically removes repeated literals and ensures the uniqueness of ground literals according to $\Gamma \subseteq DB$. At the end, two different children are created, each one carrying genetic information from both parents (Figure 3).

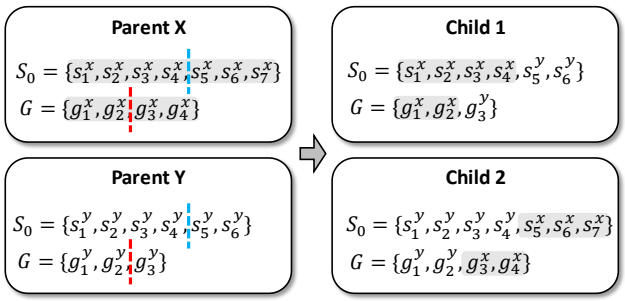

Figure 3. Crossover process (vertical dashed lines are the crossover points and shadowed variables are the parent X's genes).

After crossover, offspring are submitted to mutation, which prevents the algorithm from being trapped in a local minimum and maintains the diversity of the population. Considering that the reactive elements of individuals are represented by sets of ground literals, traditional mutation methods, such as flipping and interchanging genes, cannot be directly applied. Therefore, we employed a mutation procedure that randomly removes or adds ground literals from the initial state (S_0), goal state (G), or both. Given a mutation probability M_p, a random mutation type M_t (add, remove, or add & remove), and a random target M_s (S_0, G, or both), the mutation is performed with probability M_p over state M_s, changing it according to M_t. When performing an add mutation type, a new ground literal is randomly generated according to the rules established in $\Gamma \subseteq DB$. In our experiments, we used $M_p = 20\%$.

All offspring created through the crossover process, which may or may not be affected by mutation, are inserted into a new population. When the size of the new population reaches the maximum number of individuals (100 in our experiments), the current population is replaced by the new population, establishing a new generation. The new generation is then evaluated and the whole process is repeated for a given number of generations (Figure 2).

B. *Fitness Function*

In a genetic algorithm, the fitness function takes an individual as input and returns a numeric value representing the evaluation of the indicated individual. In our method, the utility/quality of a quest is estimated by how well its plot resembles a desired story arc.

The concept of story arc dates back to 1863, when Gustav Freytag, inspired by the ideas of Aristotle about epic poetry and tragedy, proposed a simple plot pattern that represents the narrative structure of a classical five-act tragedy (Freytag, 1900[4]). This structure was enhanced by subsequent theorists towards a more general model, known today as Freytag's Pyramid or Freytag's Triangle. Although modern literary narrative has been increasingly hostile to normative notions of structure, the new entertainment media (e.g. films, comic books, and videogames), have special needs and challenges, because they deal with audiences engaged in short episodes (e.g. 80 min films), visually intensive discourses and short narrative cycles (e.g. comics and graphic novels), and/or interactive experiences (e.g. videogames and interactive storytelling). In these cases, normative notions of dramatic structure are very helpful. This may explain the enormous success of simple variants of the Freytag's Pyramid that are known in the film and videogame industry as story arcs.

The most famous story arc is the three-act structure used by the film industry (Figure 4), which is divided into Setup (1/4 of the story time), Confrontation (2/4 of the story time), and Resolution (1/4 or less of the story time). This structure has 3 notable turning points: Plot Point 1 (Inciting Incident), Plot Point 2 (the stirring turning point, with uncertain outcome – e.g. will the protagonist win, lose, or die?), and Crisis (the climax, the final and dramatic confrontation, which is followed by the denouement). As illustrated in Figure 4, the horizontal axis represents time and the vertical axis represents emotional tension.

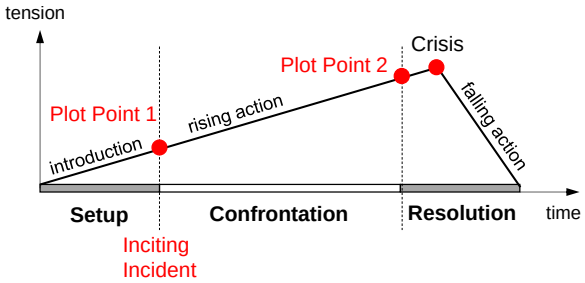

Figure 4. Story arc as a three-act structure.

[4] https://archive.org/details/freytagstechniqu00freyuoft/page/n4

Considering that story arcs are composed of falls and rises in the tension axis, we represent a story arc of a plot p as a sequence of symbols $p_{arc}^{sym} = \{s_1, s_2, \dots, s_n\}$, where each S_i can be: "+" to represent rise; or "−" to represent fall; or "=" to represent the maintenance of the tension level. The number of symbols in the sequence represents the discretized time axis. For example, the three-act story arc illustrated in Figure 4 can be expressed as: $p_{arc}^{sym} = \{+, +, +, -\}$, where: 1/4 of the story time is dedicated to the Setup (first " + " symbol); 2/4 is dedicated to the Confrontation (second and third " + " symbols); and 1/4 of the remaining story time is reserved for the Resolution (last " − " symbol).

The symbolic representation of a story arc can also be converted into a numeric representation $p_{arc}^{num} = \{v_1, v_2, \dots, v_n\}$, where each v_i of p_{arc}^{sym} is a number indicating the current tension value in the vertical axis of the story arc. Letting $v_0 = 0$, the tension value for each element of p_{arc}^{num} is given by:

$$v_i = \begin{cases} v_{i-1} + 1 & if\ s_i = " + " \\ v_{i-1} - 1 & if\ s_i = " - " \\ v_{i-1} & if\ s_i = " = " \end{cases}$$

For example, the symbolic story arc $p_{arc}^{sym} = \{+, +, +, -\}$ yields $p_{arc}^{num} = \{1, 2, 3, 2\}$.

In our method, the fitness function takes as input a desired story arc (provided by the user using the symbolic notation) and an individual of the population. Then, to calculate the fitness of the individual, the function preforms three steps: (1) solves the planning problem of the individual to generate the plot of the quest; (2) estimates the story arc of the quest plot using the information defined in $E \subseteq DB$, which describes how each event affects the overall tension of the quest; and (3) calculates the fitness of the individual by comparing the story arc of the quest plot with the desired story arc.

The process to solve planning problems can be performed by any classical planner. In our implementation, we used the HSP2 planner provided by Bonet and Geffner [23], which is fully compatible with our STRIPS-like formalism. It is important to notice that not all genetic structure characterizations of individuals can be treated as a planning problem leading to proper solutions. Therefore, the planner must handle situations where there is no valid sequence of actions that leads from the initial state to the goal state; or when the searching process exceeds a prescribed time limit (indicating an infinite loop or an excessively complex problem). In these cases, the plot of the quest will be empty.

After obtaining the plot of the quest, the story arc is calculated according to the events that occur along the plot. For example, considering a plot p with the events:

```
attack(zombie, anne, home), go(john,
johnhome, village), go(john, village,
store), get(john, antidote1, store),
go(john, store, village), go(john, village,
johnhome), cure(john, anne, antidote1,
johnhome),
```

and assuming that $E \subseteq DB$ defines the effects of the operators as E = {`attack:"+", get:"+", go:"=", cure:"-",` …}, the story arc for plot p can be expressed in the symbolic notation as $p_{arc}^{sym} = \{+, =, =, +, =, =, -\}$ and converted to its numeric representation: $p_{arc}^{num} = \{1, 1, 1, 2, 2, 2, 1\}$.

In order to estimate how well the story arc of a plot resembles a desired story arc, we calculate the difference of the two arcs. However, considering that story arcs can have different time scales, first we rescale them both to a common time interval. In our experiments, all story arcs are scaled to the interval [1, 10]:

$$\forall_i \in \{1, \dots, 10\} \quad p_{arc_i}^{scaled} = p_{arc_{\left\lfloor \frac{i-1}{10}\left(\overline{p_{arc}^{num}}-1\right)\right\rfloor+1}}^{num}$$

where $\overline{p_{arc}^{num}}$ represents the length of p_{arc}^{num}. For example, the three-act story arc $v_{arc}^{num} = \{1, 2, 3, 2\}$ is scaled to $v_{arc}^{scaled} = \{1, 1, 2, 2, 2, 3, 3, 2, 2, 2\}$. In the same way, the plot of the example presented above $p_{arc}^{num} = \{1, 1, 1, 2, 2, 2, 1\}$ is scaled to $p_{arc}^{scaled} = \{1, 1, 1, 1, 2, 2, 2, 2, \ 1, 1\}$. With both story arcs scaled to the same time interval, their differences can be comparted (as shown in Figure 5).

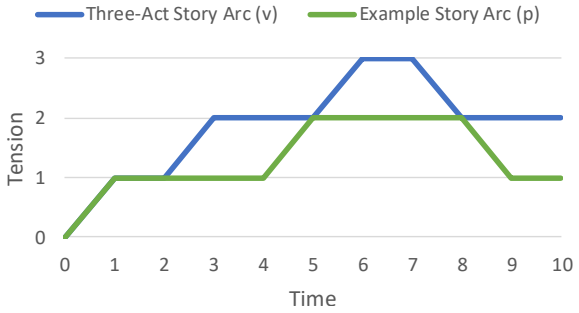

Figure 5. Visual comparasion of two scaled story arcs.

In order to calculate the difference between two scaled story arcs (d and p), we use the mean squared error:

$$mse(p, d) = \frac{1}{n}\sum_{i=1}^{n}\left(p_{arc_i}^{scaled} - d_{arc_i}^{scaled}\right)^2$$

The final fitness of an individual is calculated as:

$$fitness(p, d) = \frac{\bar{p}}{mse(p_{arc}^{scaled}, d_{arc}^{scaled})}$$

where p is the plot of an individual of the population, d is the desired story arc provided by the user, and $\bar{p}$ represents the length of p (i.e., the number of events in the plot).

C. Optimizations

The process to evaluate the individuals of the population is the most computationally expensive part of our genetic algorithm, especially because it involves the execution of a planner to generate the quest plot of each individual. Although a genetic algorithm is naturally an offline process, fast responses are important in our context to allow users to try out different parameters or modifications in the game world. Therefore, we implemented two optimizations for the evaluation process: parallel evaluation and cached plots.

Considering that each individual represents an independent planning problem, we can take advantage of multi-core processors to evaluate multiple individuals in parallel. For this optimization, we use a thread pool that maintains multiple threads waiting for tasks (evaluation of individuals) to be allocated for concurrent execution. As will be described in section V, the use of threads noticeably improves the overall performance of our genetic algorithm.

Another characteristic of the populations of our genetic algorithm is the occurrence of individuals with equal planning problems along the same or different generations, especially when the game world is small. In these cases, the same planning problem would need to be solved multiple times, wasting time and CPU resources. In order to optimize this process, we implemented a memoization (a.k.a. memoisation) technique to store the plots generated from previous planning problems. When a similar planning problem is about to be solved, the cached plot is retrieved and reutilized. The cache uses a hash table structure to efficiently map keys (calculated according to the initial state and goal state of the planning problems) to values (plots).

IV. APPLICATION

In order to test and validate our method, we adapted a 2D RPG developed for a previous project [8][24][25] (Figure 6), in which we incorporated the proposed architecture to use quests generated by our genetic algorithm. The game pertains to a zombie survival genre, telling the story of a family that lives in a world dominated by a zombie plague. The game world comprises 15 different places: a village; a forest; a hospital; a store; the main character's house; a neighbor's house; an island, which includes east, west and central regions; a mountain area that equally includes east, west and central regions; a bridge connecting the village with the island; and another bridge connecting the island with the mountain area. The world is inhabited by 7 characters: the brave husband John (controlled by the player); Anne (John's wife); Maggie (John's daughter); Rick (a shop attendant); Bob (a doctor); Matt (a wood cutter); and Robin (a survival specialist). Several collectable items are scattered through the world, including keys to open locked doors, antidotes to cure infected characters, food to feed starving characters, and piles of wood that can be used in combination with a tool kit to fix broken bridges. In addition, the game world includes several zombies that can attack and infect other characters.

Figure 6. Scene of the prototype game, where the player's avatar (John) is surrounded by zombies.

The gameplay is designed to be driven by the story quests, wherein it is the player's responsibility to collect items, interact with non-player characters, kill zombies, open locked doors, fix broken bridges, cure infected characters, and feed starving characters. In order to fight against the zombies, the player has a gun with a limited amount of ammunition, which is reloaded when the player collects ammunition kits. When the player is attacked by zombies, he/she loses an amount of life (i.e. of the life energy initially attributed to the player), which is only restored when he/she collects medic kits. The game was implemented in Lua using the Löve 2D framework. [5]

In order to generate the quests for the game, we used our quest generation method. To test the genetic algorithm, we employed a fixed population size of 100 individuals, and each run of the algorithm comprised 100 generations. At the end of a run, only the best individual's quest was selected to be included in the game. For all runs, the three-act structure was used as the desired story arc. A total of 3 quests were generated for the game: [6]

Q_1 = `starve(maggie, home), go(john, johnhome, store), ask(john, rick, food2, store), give(rick, john, food2, store), go(john, store, johnhome), feed(john, maggie, food2, johnhome).`

Q_2 = `attack(zombie, anne, johnhome), go(john, johnhome, neighborhouse), kill(john, zombie2, neighborhouse), get(john, hospitalkey, neighborhouse), go(john, neighborhouse, hospitaldoor), opendoor(john, hospitalkey, hospitaldoor), go(john, hospitaldoor, hospital), ask(john, bob, antidote2, hospital), give(bob, john, antidote2, hospital), go(john, hospital, johnhome), cure(john, anne, antidote2, johnhome).`

Q_3 = `attack(zombie, bob, hospital), go(john, johnhome, forest), kill(john, bigzombie, forest), get(john, wood1, forest), go(john, forest, store), get(john, toolkit, store), go(john, store, villageislandbridge), fixbridge(john, toolkit, wood1, villageislandbridge), go(john, villageislandbridge, islandeast), ask(john, matt, antidote3, islandeast), give(matt, john, antidote3, islandeast), go(john, islandeast, hospital), cure(john, bob, antidote3, hospital).`

Although our game world supports more quests, we generated only 3 quests to give human designers more freedom to design their own quests for the user evaluation test (a Turing-like Test) described in section V.

V. EVALUATION

To evaluate the results produced by the proposed method, we performed two tests: (1) a technical test to check the performance and the evolution progress of our genetic algorithm; and (2) a user evaluation test to compare the quests automatically generated by our system with quests manually produced by a human game designer.

A. *Techinical Evaluation*

The technical evaluation of our method comprised two experiments. First, we analyzed the evolution progress of the genetic algorithm and compared it with a random quest generation strategy. For the second experiment, we evaluated the computational performance of the algorithm.

[5] https://love2d.org/

[6] For the sake of simplicity, we condensed sequences of `go` events by describing only the initial and final locations (for example, "`go(john, home, village), go(john, village, forests)`" is represented as "`go(john, home, forest)`").

To evaluate the evolution progress, we performed 25 runs of the genetic algorithm to generate a single quest. For this experiment, we used the world of the game described in section IV and the following settings in the genetic algorithm: population size of 100 individuals; elitism factor of 2%; single point crossover (following the process described in section III (A)); mutation probability of 20%; termination condition of 100 generations; and the three-act structure as the desired story arc. These settings were chosen in the course of preliminary experiments.

In order to compare the results with a random quest generation strategy, we created another version of our system that uses our method to generate a random initial population for the genetic algorithm, but, instead of applying the evolutionary strategy, each new generation is initialized as a new population. As the genetic algorithm, the process is repeated for 100 generations with the population size set to 100 individuals (testing a total of 10.000 random quests). At each generation, the best random quest is selected to be compared to the best, average and worst individuals of the genetic algorithm.

Figure 7 shows the results of this experiment. As can be noticed, the genetic algorithm clearly overcomes the random strategy with just a few generations. In addition, 50 generations proved to be enough for the individuals to converge to the optimal solution for this problem (that is, for this game world configuration and using the three-act structure as the desired story arc).

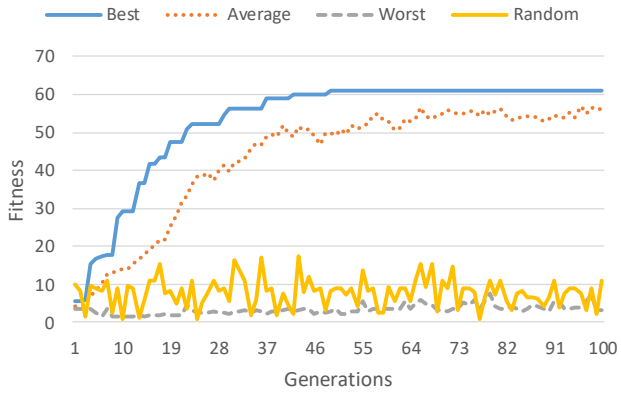

Figure 7. Average evolution progress of the best, average, and worst (valid) individuals of the population during 25 runs of the genetic algorithm in comparsion with the best random quests created with a random strategy.

To evaluate the computational performance of our method, including de efficiency of the optimizations described in Section III (C), we calculated the average time required to process a population of 100 individuals during one generation of the genetic algorithm (using the same settings of the previous experiment). Three versions of our algorithm were tested: (1) a version without optimizations (Base Version); (2) an optimized version with multiple threads for the evaluation process (Parallel Version); and (3) a fully optimized version using multiple threads and cached plots (Optimized Version). The computer used to run the experiments was an Intel Core i7 7820HK, 2.9 GHZ CPU, 16 GB of RAM. Each version was tested for 25 full runs of the genetic algorithm, each one testing 100 generations (generating a total of 2500 time samples per version). Table I shows the results of the tests, which indicate that the optimized version is more than 6 times faster than the base version. Although the optimized version still needs a considerable amount of time to generate a quest plot (average time of 3.59 minutes to run the algorithm for 50 generations), the quality of the solution (in comparison with those generated by a random strategy) fairly compensates the processing time.

TABLE I. AVERAGE TIME TO PROCESS A POPULATION OF 100 INDIVIDUALS DURING ONE GENERATION IN THREE VERSIONS OF OUR GENETIC ALGORITHM: (1) BASE VERSION WITHOUT OPTIMIZATIONS; (2) PARALLEL VERSION USING MULTIPLE THREADS; AND (3) OPTIMIZED VERSION USING MULTIPLE THREADS AND CACHED PLOTS

	Base Version	**Parallel Version**	**Optimized Version**
Time (*sec*)	27.56	10.45	4.31
Standard Deviation	13.21	7.11	3.72

B. *User Evaluation*

In order to compare quests automatically generated by our system with those manually produced by game design professionals, we conducted a simplified Turing-like Test to evaluate if human players would be able to differentiate quests produced by our algorithm from those created by a human game designer.

For this experiment, we asked a professional game designer with over 10 years of game industry experience to design 3 new quests for our game, allowing him to freely add new characters, places, items and actions, but without interfering in the logic and structure of the existing quests previously generated by our algorithm. The quests created by the game designer are:

```
DQ1 = go(john, home, store), ask(rick, john,
killvillagezombies), go(john, store,
village), kill(john, villagezombies,
village), go(john, village, store),
thank(rick, john).

DQ2 = getlost(george, forest), go(john,
home, forest), kill(john, zombie4, forest),
find(john, george, forest), take(john,
george, forest, hospital).

DQ3 = attack(zombie, maggie, home), go(john,
home, islandeast), kill(john, zombie5,
islandeast), get(john, wood2, islandeast),
go(john, islandeast, islandmountainbridge),
fixbridge(john, toolkit, wood2,
islandmountainbridge), go(john,
islandmountainbridge, mountainwest),
attack(zombie, robin, mountainwest),
die(robin, mountainwest), steal(john,
robin, antidote4, mountainwest), go(john,
mountainwest, home), cure(john, maggie,
antidote4, home).,
```

which were integrated with those resulting from our algorithm in the order: {Q1, DQ1, Q2, DQ2, Q3, DQ3}.

After integrating the designer's quests into the game, we asked 34 students (28 male and 6 female, aged 18 to 23) to play our game and classify quests according to whether they were created by a human designer or by a machine. Before the test, we explained to subjects that some quests were created by an algorithm and others by a game designer, but did not tell them how many quests were created by each one. After completing each quest, the game was automatically paused, and the subjects were prompted to judge if the quest was created by a human designer or by a machine. All

participants were able to complete all quests. On average, each session lasted 32.4 minutes (standard deviation of 8.3).

Over the entire test set of 204 data points (34 players, each evaluating 6 quests), the "machine" version was correctly identified in 48 cases (out of a total of 102) and the "human" version was correctly identified in 52 cases (out of a total of 102), leading to an overall accuracy of 49.02%.

An ideal Turing Test is represented by the case where the computer and the human versions are indistinguishable, leading therefore to a random choice of 50% accuracy. The small difference between the achieved accuracy (49.02%) and the ideal Turing Test value (50%) suggests that the computer-generated and the human-designed quests are hardly distinguishable, which is an indication of the capacity of the proposed quest generation method to achieve quest plots closely similar to those created by professional game designers.

VI. Conclusion Remarks

We described in this paper a new quest generation method, combining planning with an evolutionary search strategy guided by story arcs, which can generate coherent and diversified quests based on a specific narrative structure. Our method provides game designers with new ways of imagining and creating narratives for games. Also, game developers can take advantage of this technique to automate the process of designing new quests for games, and, therefore, to reduce the work overload of the development teams.

In our experiments, the proposed method revealed encouraging results. The genetic algorithm showed good overall evolution progress that easily overcomes a random quest generation strategy with just a few generations. Another suggestive evidence comes from the fact that players were not able to distinguish quests created by our method from those created by a professional game designer.

In terms of performance, although some optimizations in our method were already implemented, users still need to wait a few minutes to visualize the results (as described in section V (A)). Accordingly, one of our next research goals is to find alternatives to improve the overall performance of our method, for instance by replacing the planning algorithm by faster methods (e.g., hierarchical task networks) or by implementing a cloud computing architecture.

As another research objective, to be pursued while exploring alternatives to improve performance, we plan to extend the proposed method to support the generation of narratives with branching storylines. The alternative paths can be created by using the final states of the best individuals as initial states for new runs of the genetic algorithm. This form of branching storylines may provide game designers with new ways to expand traditional quests towards new forms of interactive storytelling.

References

[1] T. Ong, and J. J. Leggett, "A genetic algorithm approach to interactive narrative generation," Proceedings of the fifteenth ACM conference on Hypertext and hypermedia, 2004, pp. 181–182.

[2] N. McIntyre, and M. Lapata, "Plot induction and evolutionary search for story generation," Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, 2010, pp. 1562–1572.

[3] S. Giannatos, Y.-G. Cheong, M. J. Nelson, and G. N. Yannakakis, "Generating Narrative Action Schematics for Suspense," Proceedings of the Workshop on Intelligent Narrative Technologies, Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2012), AAAI, 2012, pp. 8–13.

[4] S. Giannatos, M. J. Nelson, Y.-G. Cheong, and G. N. Yannakakis, "Suggesting new plot elements for an interactive story," Proceedings of the Workshop on Intelligent Narrative Technologies, Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE 2011), AAAI, 2011, pp. 25–30.

[5] M. Nairat, P. Dahlstedt, and M. G. Nordahl, "Character evolution approach to generative storytelling," Proceedings of the 2011 IEEE Congress of Evolutionary Computation, 2011, pp. 1258–1263.

[6] M. Nairat, P. Dahlstedt, and M. G. Nordahl, "Story Characterization Using Interactive Evolution in a Multi-Agent System, " In: P. Machado, J. McDermott, and A. Carballal (eds), Evolutionary and Biologically Inspired Music, Sound, Art and Design, Springer, 2013.

[7] A. Sullivan, M. Mateas, and N. Wardrip-Fruin, "Rules of engagement: Moving beyond combat-based quests," Proceedings of the Intelligent Narrative Technologies III Workshop (INT3 '10), Article No. 11, ACM, 2010, doi: 10.1145/1822309.1822320.

[8] S. Lima, B. Feijó, and A. L. Furtado, "Hierarchical Generation of Dynamic and Nondeterministic Quests in Games," Proceedings of the 11th International Conference on Advances in Computer Entertainment Technology, ACM, 2014, Article N. 24, doi: 10.1145/2663806.2663833.

[9] V. Breault, S. Ouellet, and J. Davies, "Let CONAN tell you a story: Procedural quest generation," arXiv:1808.06217, 2018.

[10] M. Hendrikx, S. Meijer, J. V. D. Velden, and A. Iosup, "Procedural content generation for games: A survey," ACM Transactions on Multimedia Computing, Communications, and Applications, Vol. 9 (1), Article No. 1, 2013, doi: 10.1145/2422956.2422957.

[11] A. Amato, "Procedural Content Generation in the Game Industry," In: O. Korn, N. Lee (eds) Game Dynamics, Springer, 2017.

[12] J. Togelius, A. J. Champandard, P. L. Lanzi, M. Mateas, A. Paiva, M. Preuss, and K. O. Stanley, "Procedural Content Generation: Goals, Challenges and Actionable Steps," Artificial and Computational Intelligence in Games, vol. 6, 2013.

[13] J. Freiknecht, and W. Effelsberg, "A Survey on the Procedural Generation of Virtual Worlds," Multimodal Technologies and Interaction, vol. 1 (4), 27, 2017, doi:10.3390/mti1040027.

[14] B. Kybartas, and R. Bidarra, "A Survey on Story Generation Techniques for Authoring Computational Narratives," IEEE Transactions on Computational Intelligence and AI in Games, Vol. 9 (3), 2017, pp. 239-253, doi: 10.1109/TCIAIG.2016.2546063.

[15] S. N. Sivanandam, and S. N. Deepa, "Introduction to Genetic Algorithms," Springer Berlin Heidelberg, New York, 2008.

[16] D. Johnson, "Animated Frustration or the Ambivalence of Player Agency," Games and Culture, vol. 10 (6), pp. 593–612, 2015.

[17] V. Propp, "Morphology of the Folktale," University of Texas Press, Austin, USA, 1968.

[18] E. Aarseth, "Quest Games as Post-Narrative Discourse," In Marie-Laure Ryan (ed.): Narrative Across Media. University of Nebraska Press, pp. 361-76, 2004.

[19] S. Tosca, "The quest problem in computer games," Proceedings of the Technologies for Interactive Digital Storytelling and Entertainment (TIDSE), Fraunhofer IRB Verlag Press, 2003.

[20] H. Jenkins, "Henry Jenkins responds in turn: riposte to Game Design as Narrative Architecture," Electronic Book Review, 2004.

[21] M. Ghallab, D. Nau, and P. Traverso, Automated Planning: Theory and Practice, Morgan Kaufmann Publishers, United States, 2004.

[22] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Boston, 1989.

[23] B. Bonet, and H. Geffner, Planning as Heuristic Search, Artificial Intelligence, vol. 129 (1), pp. 5-33, 2001.

[24] E. S. Lima, B. Feijó, and A. L. Furtado, "Player Behavior and Personality Modeling for Interactive Storytelling in Games," Entertainment Computing, Vol. 28, December 2018, pp. 32-48, 2018, doi: 10.1016/j.entcom.2018.08.003.

[25] E. S. Lima, B. Feijó, and A. L. Furtado, "Player Behavior Modeling for Interactive Storytelling in Games," Proceedings of the XV Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2016), São Paulo, Brazil, p. 1-10, 2016.