Entertainment Computing 43 (2022) 100491

Contents lists available at ScienceDirect

Entertainment Computing

journal homepage: www.elsevier.com/locate/entcom

Procedural generation of branching quests for games

Edirlei Soares de Lima [a,b,*], Bruno Feijó [c], Antonio L. Furtado [c]

[a] Universidade Europeia, Faculty of Design, Technology, and Communication, Lisbon, Portugal
[b] UNIDCOM/IADE - Design and Communication Research Unit, Lisbon, Portugal
[c] Pontifical Catholic University of Rio de Janeiro, Department of Informatics, Rio de Janeiro, Brazil

ARTICLE INFO

Keywords:
Quest Generation
Genetic Algorithms
Automated Planning
Branching Narratives
Interactive Storytelling

ABSTRACT

The production of high-quality digital games usually requires a few hundred individuals, including designers, artists, and story writers. Currently, development teams are increasingly applying procedural content generation techniques to reduce their work overload. However, there is a lack of methods to handle the procedural generation of branching quests that can support the richness of the digital games' mutable and highly interactive virtual worlds. The challenges to maintaining a rich and coherent game narrative are enormous. In this work, we present a novel procedural quest generation method that can produce coherent quests with branching storylines for dynamic and interactive game worlds. By combining automated planning with a genetic algorithm guided by story arcs, the proposed method can generate coherent branching quests based on a narrative structure. Preliminary results show that the branching quests created with our approach are nearly at par with those made by professional game designers.

1. Introduction

Procedural content generation (PCG) creates video game content (such as maps, background scenery, music, rules, quests, and characters) using computer algorithms with limited or no human intervention. There is a vast literature on the subject elsewhere [1–4]. PCG is a promising approach to reduce the work overload of the development teams, which involve hundreds of individuals among designers, engineers, artists, and story writers producing new content continuously [5]. Although game developers and academic researchers extensively explore PCG, there is a lack of techniques to handle the procedural generation of quests - a fundamental mechanism for narrative progression. This type of procedural generation is the central subject of the present paper.

Quests (also called missions) are tasks that avatars (i.e., game characters controlled by human players) may gain rewards and keep the story going on. We find them most commonly in RPGs (Role-Playing Games), such as Assassin's Creed Valhalla (Ubisoft, 2020) and Cyberpunk (CD Projekt RED, 2020). A more in-depth discussion of video games' narratives and quests can be found elsewhere [6].

Quests and narrative generation methods face enormous problems in interactive storytelling, i.e., in stories built through the intervention of the reader or player (see [7] for a survey on narrative generation methods). A common approach to RPGs' quests and plot generation is to define a branching storyline that contains all possible courses of the narrative described by the story writer. However, interactivity in the storyline of RPGs' missions is remarkably complex because they affect quests generated by algorithms and those manually created by game designers. Although many recent games include some quests with branching storylines, they produce most of their missions based on linear storylines. These linear quests often fail to provide the players with the ability to interfere in the game's story, which reduces the players' sense of agency (i.e., the players' assurance of being the agents of their actions). Examples of these games are Mass Effect 2 (BioWare, 2010), The Witcher 3: Wild Hunt (CD Projekt RED, 2015), Assassin's Creed Valhalla (Ubisoft, 2020), and Cyberpunk 2077 (CD Projekt RED, 2020). In this scenario, there is a need for more effective and robust methods for procedural quest generation.

In this work, we propose a new quest generation method that can produce coherent quests with branching storylines for dynamic and interactive game worlds. By combining automated planning with a genetic algorithm guided by story arcs, the proposed method can generate coherent branching quests based on a narrative structure.

This work is a considerable extension of our paper "Procedural Generation of Quests for Games Using Genetic Algorithms and Automated Planning", published in the *XVIII Brazilian Symposium on*

* Corresponding author at: Universidade Europeia, Faculty of Design, Technology, and Communication, Lisbon, Portugal.
E-mail address: edirlei.lima@universidadeeuropeia.pt (E.S. de Lima).

https://doi.org/10.1016/j.entcom.2022.100491
Received 14 May 2021; Received in revised form 28 November 2021; Accepted 8 April 2022
Available online 14 April 2022

Computer Games and Digital Entertainment (*SBGames 2019*), where we proposed a method for the procedural generation of *linear* quests using a genetic algorithm and automated planning [8]. In the present work, we extend the proposed method to support the generation of *nonlinear* quests through *branching storylines*. Our approach resulted in a completely new method that uses tree structures to represent the individuals of the genetic algorithm (see Section 3). This article also presents a new user study and a new technical evaluation, which were conducted to assess the results produced by the new quest generation method in a game prototype (see Section 5). These contributions are new results from a sequence of previous research works on procedural quest generation [6–10].

The text is organized as follows. Section 2 presents related works. Section 3 describes the proposed method for the generation of branching quests. Section 4 presents an application of the proposed method in a game prototype. Section 5 describes an evaluation of the method. Section 6 offers concluding remarks.

2. Related work

There are several works on quest generation in the literature, such as the generic framework presented by Sullivan et al. [11]. In their system, a game manager uses the player history, a library of quests, and the current state of the world to dynamically recombine and alter the structure of quests. A more recent framework based on automated planning was proposed by Breault et al. [12], who use a deterministic planning algorithm and a logical description of the game world to generate linear quests that are consistent with the given game world. A similar approach is explored by Chongmesuk and Kotrajaras [13], but focusing on the analysis of the alternative paths that can be generated for a given type of quest.

A more dynamic solution is presented by Lima et al. [6], who uses hierarchical task decomposition and automated planning to generate non-deterministic quests. Their approach combines planning, execution, and monitoring to handle nondeterministic events and generate quests offering multiple alternative outcomes, in an attempt to achieve highly interactive dynamic story plots. A different approach is explored by Ammanabrolu et al. [14], who present a framework to generate cooking quests for text-adventure games that uses Markov chains and a neural language model to generate recipes. Their system uses a weighted ingredient graph (Markov chain), which is learned from a large-scale knowledge base of recipes. Quests are generated by probabilistically walking along the ingredient graph and applying a language model based on the GPT-2 framework [15] to generate a description for the sequences of ingredients required by the recipe.

There are also a few related works that explore the application of genetic algorithms for narrative generation. McIntyre and Lapata [16] propose a story generator system that uses a genetic algorithm where plots are represented by ordered graphs of dependency trees, which correspond to narrative text sentences). Their algorithm works directly with text sentences; therefore, the genetic operations handle the syntax and semantics of the sentences, including the “mutation, which can occur on any verb, noun, adverb, or adjective in the sentence” [16]. Ong and Leggett [17] propose a genetic algorithm to recombine story components created from a set of pre-determined story templates. In their algorithm, generated stories are evaluated according to their events, which are previously rated by the author. Giannatos et al. [18] present a system that uses a genetic algorithm to generate interactive narratives by incrementally adding candidate plot points to a story graph, which has its possible playthroughs rated according to three criteria: spatial locality, thought flow, and motivation. A different approach is explored by Nairat et al. [19,20], who use an interactive genetic algorithm to generate narratives in an agent-based system. In their method, the author is responsible for observing and selecting the agents whose behaviors are considered relevant for the intended story. Based on the author’s selections, the system proceeds with the genetic algorithm to perform the reproduction process and create new generations of agents.

There are also applications of genetic algorithms for narrative generation in games, such as the work of Utsch et al. [21], who propose a drama manager based on a genetic algorithm that generates plots by selecting, from a pool, the subset of plot points that should be presented to the player according to a player model. The individuals of their genetic algorithm are represented by a set of plot points (events), which are subject to traditional crossover and mutation operations. In their algorithm, the fitness of an individual is given by the difference between the set of plot points and the player model, which is based on the three dimensions of the *Anthropological Structures of the Imaginary* proposed by Durand [22].

The combination of genetic and planning algorithms was also previously explored by Giannatos et al. [23], who proposed a method that uses genetic algorithms to generate plan operators representing possible story actions. Their algorithm uses a fitness function that estimates the operator’s contribution to achieve suspenseful stories. Although the work of Giannatos et al. [23] 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). An evident difficulty of their approach is that it does not clearly attribute meaning to the operators (i.e. the human author must interpret the generated operator and define its meaning for the narrative).

Most of the previous works on quest generation require a library of existing quests (e.g.: [11,17]) or highly detailed descriptions of planning patterns with predefined goal states (e.g.: [12,13,6]), which demands extra authorial work and restricts the space of possible quests. In such context, the use of genetic algorithms has been mostly limited to non-game applications, where event structures are not necessary (e.g.: [16,19,20]) or other methods are responsible for the actual narrative generation (e.g.: [18,23]). Another important limitation of previous works on quest generation is the fact that most of them are designed to generate only linear quests (e.g.: [12,20,21]), which reduces the player’s sense of agency.

3. Quest generation

3.1. The quest generation system

In this paper, a quest is defined as a set of tasks (e.g. killing enemies, escorting and saving characters, collecting and delivering items) which the player must accomplish. The success in achieving these tasks or the different decisions made by the players while performing those tasks will lead them to experience a unique and customized plot for the quest. This unique plot is named *storyline* in the present work. Therefore, quests are modeled as tree structures, where branches represent the different ways in which the quest plot can develop (more details about the tree structure of quests are presented in Section 3.2).

Considering that all quests take place in the same world, changes in the world state caused by player actions while performing previous quests can have implications in the plot of future quests. For example, if the player constructs a bridge to access a certain location of the world as part of the tasks of one quest, there is reason to make it a recurring task in future quests – except if the bridge was destroyed during an intermediary quest. As a strategy to maintain the logical coherence between consecutive quests, a tree structure, which we call *Game Tree* (Ω), represents the structure of the game’s narrative. As illustrated in Fig. 1, each node of the Game Tree is a branching quest (ψ_i), which is also a tree structure. The number of final states of ψ_i (i.e. the number of outcomes) determines the number of possible new quests that can follow ψ_i. The

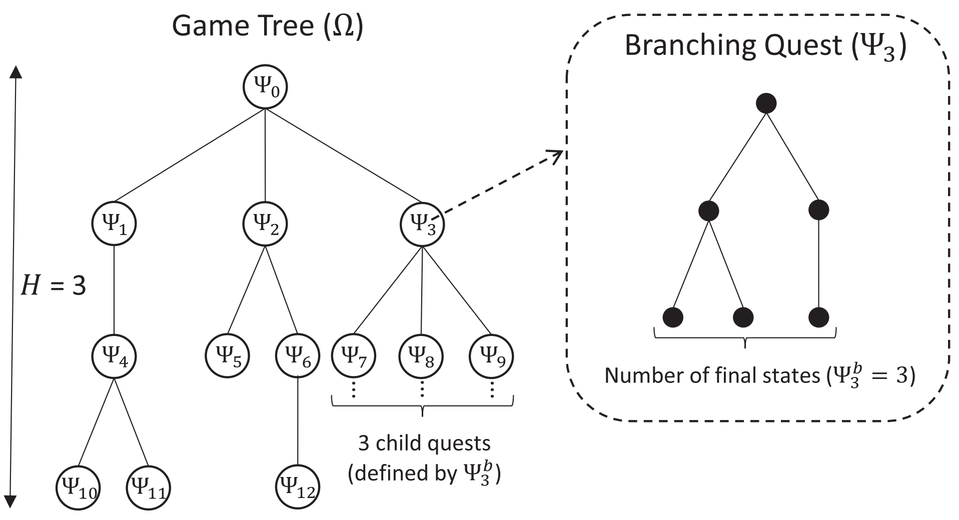
Game Tree (Ω)
Branching Quest (Ψ_3)
$H = 3$
Ψ_0 Ψ_1 Ψ_2 Ψ_3 Ψ_4 Ψ_5 Ψ_6 Ψ_7 Ψ_8 Ψ_9 Ψ_{10} Ψ_{11} Ψ_{12}
Number of final states ($\Psi_3^b = 3$)
3 child quests (defined by Ψ_3^b)

Fig. 1. Structure of the narrative as a tree of branching quests.

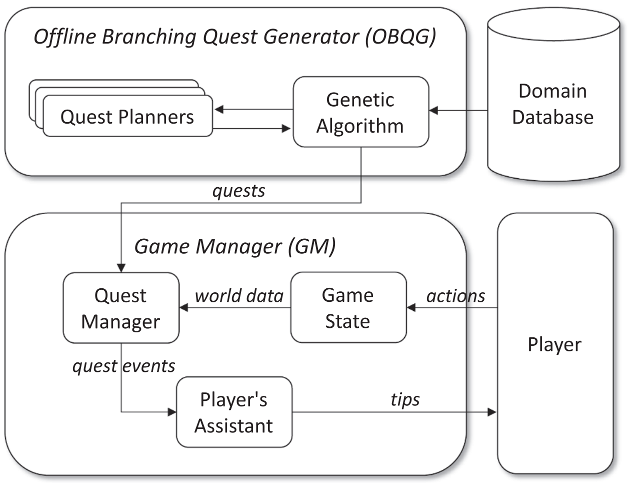

Fig. 2. Architecture of the quest generator system.

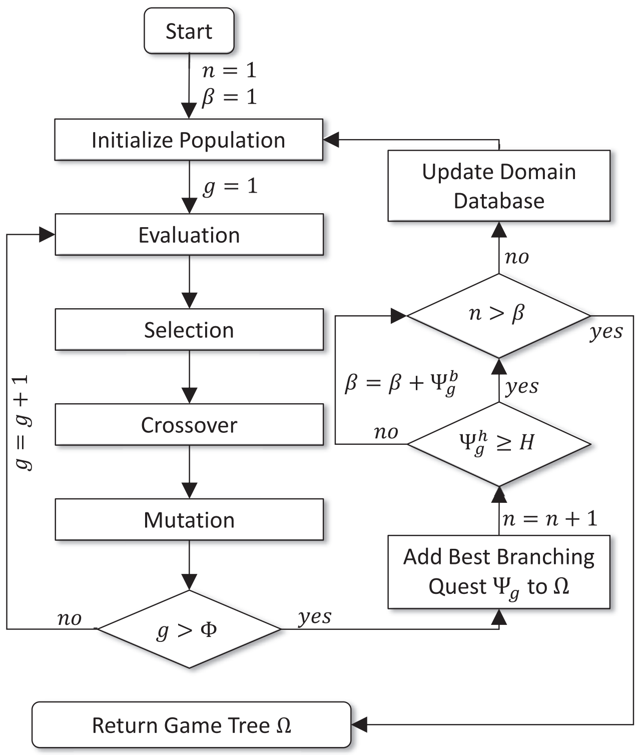

Fig. 3. The proposed genetic algorithm. Φ is the maximum number of generations, H is the intended height for the Game Tree (Ω), ψ_g^h is the current height of node ψ_g in Ω, ψ_g^b is the number of final states (outcomes) in the generated quest ψ_g, and β is the number of sequential quests.

proposed method adds each of these new quests to Ω as a child of ψ_i (Fig. 1).

As illustrated in Fig. 2, 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 cause updates on the *Game State*, which is used by the *Quest Manager* to keep track of the player progression accomplishing quests. Meanwhile, the player receives help from the *Player's Assistant*, who provides tips about the next objectives of the current quest.

3.2. The basic elements of the proposed genetic algorithm

As illustrated in Fig. 3, our method involves the main steps of a standard genetic algorithm [24,25] with the addition of an extra control loop to manage the generation of sequential quests that will compose the Game Tree structure representing all quests of a game. The world state,[1] which is stored in the Domain Database, is updated with information extracted from the final states of previously generated quests. In this way, the algorithm can guarantee the logical coherence of sequences of

[1] Term used in automated planning theory to describe the current state of the world.

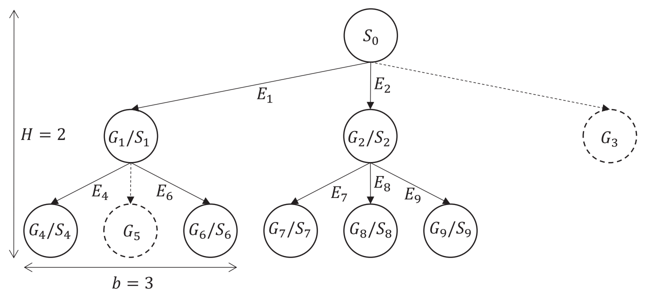

Fig. 4. Tree structure of a branching quest with height $H = 2$ and branching factor $b = 3$. Dotted arrows and dotted circles represent invalid branches with unachievable goal states.

quests in the Game Tree.

The algorithm shown in Fig. 3 seems similar to the one we proposed in our previous work [8], but there is a substantial change in how we manage the generation of sequential quests.

Each individual in our new genetic algorithm represents a candidate branching quest to be added to the Game Tree, encoded as a tree structure (Fig. 4), where:

- The root node represents the initial state of the quest (S_0);
- Internal nodes define intermediate goal states (G_i) and intermediate states (S_i);
- Leaf nodes define final goals and final states for the quest;
- A branch (E_i) is composed of a pair of nodes (S_j, G_i) and an edge ($S_j \rightarrow G_i$), $j \prec i$, where the edge comprises a sequence of events to achieve the intermediate or final goal G_i (and state S_i) from the initial state or the intermediate one that precedes S_i (i.e., S_j). For example, E_9 is composed of (S_2, G_9) and ($S_2 \rightarrow G_9$).

Each branch of the *quest tree* is encoded as a planning problem:[2]

$E_i = (F, S_j, G_i, O), \ j \prec i,$

where F is a set of *atomic formulas* (or *atoms*, for short), O is a set of planning operators, $S_j \subseteq F$ is the initial state of E_i, and $G_i \subseteq F$ is the goal state in the form of a *ground literal*. In this definition, an operator $o \in O$ is denoted by:

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

where $name(o)$ is the name of the operator in the form of an atom $op(x_1, x_2, \cdots, x_k)$, and $precond(o)$ and $effect(o)$ are sets of literals that define the preconditions and the effects of o, respectively. A *literal* is an atom f or the negation of an atom ($\neg f$). A detailed explanation of these basic terms used in automated planning can be found in our previous works [8,9].

The structure of branches comprises *schematic* elements (F and O) and *reactive* elements (S_j and G_i). The interpretation of these elements is similar to the case of the single quest as a chromosome found in our previous work [8]. In the present case, however, the schematic elements are used to compose the planning problems of all branches and quests. The reactive elements can vary in the different branches of a current generated quest, expressing different paths that the story can take depending on different situations in the game world and actions caused by character-dependent preferences.

The following example illustrates the representation of a branch E_i as a planning problem:

Schematic elements of E_i:

F: character(CH), place(PL), item(IT), at(CH, PL), has(CH, PL),
hero(CH), alive(CH), path(PL0, PL1),know-request(CH0, CH1, IT),
know-need(CH0, CH1, IT).
o1:
name: go(CH0, PL0, PL1).
precond: character(CH0), place(PL0), place(PL1), at(CH0, PL0),
alive(CH0), hero(CH0), path(PL0, PL1).
effect: at(CH0, PL1), ¬at(CH0, PL0).
o2:
name: request(CH0, CH1, IT0, PL0).
precond: character(CH0), character(CH1), item(IT0),
place(PL0), at(CH0, PL0), at(CH1, PL0), alive(CH0),
alive(CH1), hero(CH1).
effect: know-request(CH1, CH0, IT1).
o3:
name: ask(CH0, CH1, IT0, PL0).
precond: character(CH0), character(CH1), item(IT0), place(PL0),
at(CH0, PL0), at(CH1, PL0), alive(CH0), alive(CH1),
hero(CH0), has(CH1, IT0).
effect: know-need(CH1, CH0, IT0).
o4:
name: give(CH0, CH1, IT0, PL0).
precond: character(CH0), character(CH1), item(IT0), place(PL0),
at(CH0, PL0), at(CH1, PL0), alive(CH0), alive(CH1),
hero(CH1), has(CH0, IT0), know-need(CH0, CH1, IT0).
effect: has(CH1, IT0), ¬has(CH0, IT0).
o5:
name: deliver(CH0, CH1, IT0, PL0).
precond: character(CH0), character(CH1), item(IT0), place(PL0),
at(CH0, PL0), at(CH1, PL0), alive(CH0), alive(CH1),
hero(CH0), has(CH0, IT0), know-request(CH0, CH1, IT0).
effect: has(CH1, IT0), ¬has(CH0, IT0).

Reactive elements of E_i:

Sj: character(john), character(rick), character(anne),
place(village), place(johnhome), place(store),
item(antidote2), hero(john), alive(john), alive(anne),
alive(rick), path(johnhome, village),
path(village, johnhome), path(store, village),
path(village, store), at(john, johnhome), at(anne, johnhome),
at(rick, store), has(rick, antidote2).
Gi: has(anne, antidote2), at(john, johnhome).

By solving the planning problem using a planning algorithm, a linear sequence of events is generated from the initial state S_j. In the above example, the generated plot is very simple:

request(anne, john, antidote2, johnhome), go(john, johnhome, village), go(john, village, store), ask(john, rick, antidote2, store), give(rick, john, antidote2, store), go(john, store, village), go(john, village, johnhome), deliver(john, anne, antidote2, johnhome).

In this simple plot, Anne requests the antidote to John at home. Then he goes to the village and the store, where Rick gives him the remedy. He comes back home and delivers the antidote to Anne.

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

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

The elements of this set are similar to the case found in our previous conference paper [8], except for two new types of *semantic integrity constraints* in Γ: $qualif_i$ and $motif_i$. However, we repeated all of them here in a more precise way to understand better and adapted them to the examples of this section. These elements are as follows:

- A is a set of pairs $\alpha_i = (obj_i, objType_i)$ that defines all **objects** of the game world (obj_i) and associates them with a specific object type ($objType_i$). For example, $\mathrm{A} = \{(anne, \ character), \ (home(john), \ place)$

[2] Notation closely adapted from planning theory [27,28].

, $(antidote2,\ item)\}$ defines that *anne* is a *character*, *home*(*john*) is a *place*, and *antidote2* is an *item*. Currently, for the sake of simplicity, we reduce all function symbols with arity greater than 0 to constants, which are function symbols with arity 0 (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.: $B = \{alive(anne),\ at(anne,\ hospital),\ path(hospital,\ village),\ isantidote(antidote2)\}$;
- Γ is a set of **semantic integrity constraints** on predicates, which plays a central role in the branch generation process of our algorithm. Currently, we have six constraint types: variable types (t), contradictory or opposite relations (*opp*), existential uniqueness quantification (u), recurrence importance (r), qualification types (*qualif*), and goal motifs (*motif*). Each member of Γ is a 5-tuple $\gamma_i = (pred_i, (u_1r_1t_1, \cdots, u_nr_nt_n), opp_i, qualif_i, motif_i)$, in which the second element indicates the *n* terms of the predicate $pred_i$ affected or not by the constraints *u* and *r*. More specifically, the elements of γ_i are:
  - o $pred_i$ is a predicate symbol with *n* terms (e.g. *path*, *alive*, *knowrequest*);
  - o 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. For example, $\gamma_i = (cured, (character), , ,)$ indicates that the predicate *cured* has a single term that must be of type *character*. Another example is $\gamma_i = (love, (character, character), , ,)$, meaning that *love* is a predicate with two terms of type *character*;
  - o 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). An example of uniqueness of a term is $\gamma_i = (at, (character, \exists! place), , ,)$, which denotes that each character can only be at one place at a time. In this case, the predicate *at*(*john*,...) can appear only once in a state for the character *john*;
  - o r_j defines the recurrence importance of the *j*-th ground term of $pred_i$ in the plot of a quest, which is used by the fitness function to evaluate the continuity factor of multiple branches of the same quest. An interrogation mark (?) indicates this type of constraint. For example, in $\gamma_i = (request, (character, character, ?item, place), , ,)$, the term of type *item* is identified as an important recurrent element ($?item$) for the *request* event, where a character requests an item from another character. Therefore, recurrences of the item object in the events of descendant branches of the same quest contribute to the general continuity of the quest (i.e., the future use of an item that was previously requested contributes to the general perception of continuity in a branching quest);
  - o opp_i is the opposite predicate of $pred_i$. An example of opposite relation is $\gamma_i = (healthy, (character), infected, ,)$, which determines that either *alive*(*anne*) or *dead*(*anne*) can hold, but not both;
  - o $qualif_i$ defines whether $pred_i$ can be dynamically added to the initial state or goal state of a quest without affecting the logical integrity and consistency of the quest. For example, in $\gamma_i = (at, (item, \exists! place), , initial, \ )$ the keyword *initial* defines that predicates of type *at* with terms of type *item* and *place* can be added to initial state of a quest (i.e., items can be dynamically spawned in the beginning of a quest without affecting the logical integrity of the game world). An alternative keyword for $qualif_i$ is *goal*, which is used to establish predicates that can be used to compose goal states;
  - o $motif_i$ denotes a motif name for predicates of related goals. For example, $\gamma_i = (hasreceived, (character, character, \exists! item), , , askitem\ )$ and $\gamma_j = (knowfailedtoget, (character, character, item), , , askitem\ )$ define that *hasreceived* (a character has received an item from another character) and *knowfailedtoget* (a character knows that another character failed to get an item) are related predicates of the goal motif *askitem*. Therefore, both predicates can be used to establish alternative goals for quests that are based on the recurring motif *askitem*.
- Δ defines a set of **planning operators** based on the STRIPS formalism [28] that represents all possible events that can occur in the course of a quest; and
- T 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: $T = \{(kill,\ +),\ (save,\ -),\ \}$.

3.3. Generating the initial population

The process to create an individual (i.e., a candidate branching quest) for the initial population of the genetic algorithm is divided into three steps: the *quest-giver* selection, the *quest motif* selection, and the tree structure generation.

First, a ***quest-giver*** is randomly selected, which, according to Howard [29], is the most essential character in role-playing and adventure games. The quest-giver is the character that will offer the quest to players, give them information on how to embark on the tasks of the quest, and offers rewards when the quest is completed. The quest-giver corresponds to Propp's dispatcher character role [30]. When creating a new individual for the genetic algorithm, the quest-giver is randomly selected among all characters that exist in the game world. The selected character will then be used by the branch generation algorithm to define the main character for the predicates that establish intermediate and final goal states.

The second step of the individual generation process is the selection of a ***quest motif***, which will guarantee the logical coherence among the goals defined for the different branches of the quest. By using goals that share the same motif, it is ensured that the generated quest will be composed of related tasks. The quest motif avoids situations in which branches lead to completely unrelated tasks, such as gathering an item in one branch while the other branches concern only what is involved in the act of killing an enemy (although the task of killing an enemy will still be possible to occur when motivated by the need to collect an item that is protected by an enemy). The possible quest motifs are defined in $\Gamma \subseteq DB$, where predicates used to define goal states are associated with a motif name. When creating a new individual for the genetic algorithm, the quest motif is randomly selected among all possible motifs defined in Γ.

After selecting the quest-giver and the quest motif, the third and final step to create an individual for the genetic algorithm is the generation of the tree structure to represent the branching quest (Fig. 4). The process starts from the root node of the tree, which is defined by the initial state of the quest (S_0). For any node of the tree, a number *b* of new branches E_i are generated and added to the tree, where *b* is a parameter of our algorithm that defines the maximum branching factor of the tree. For the schematic elements *F* and *O* of the planning problem $E_i = (F, S_j, G_i, O)$ of each branch, *F* is defined by the unique atoms found in $A \subseteq DB$ and $B \subseteq DB$, and *O* is created with the planning operators defined in $\Delta \subseteq DB$. For the reactive elements, first the initial state of the quest S_0 is initialized with the schematic elements of the game world ($A \subseteq DB$), and then a number of new ground literals are randomly generated and added to S_0 (according to the semantic integrity constraints established in $\Gamma \subseteq DB$). These new ground literals represent different ways in which the game world can be adapted and prepared for the upcoming events of the quest, which involve changes that do not affect the logical consistency of the game world, such as spawning temporary items, enemies, or characters. These changes to the initial states of branches will add more diversity to different branches of the same quest and the initial population of quests used in the genetic algorithm. A similar process is performed to define the goal state G_i for each branch, but using the *quest motif* and *quest-giver* to filter the possible ground literals that can be generated for the goal state. In this process, the quest motif is used to define the possible predicates that can be selected, which must have the same recurring motif that was selected for the quest. In addition, the main character of the ground literals for the goal states will always be the quest-giver, which reinforces the relationship between different branches of the same quest. The numbers of new ground literals to be added to S_j and G_i are defined through parameters of the genetic algorithm. In our experiments, we use random values in the range [1,30] to define the number of ground literals of S_j and [1,10] for G_i.

Once all branches of a node are generated, the planning problem of each branch is solved by a planner to generate its storyline. In our implementation, we used the HSP2 heuristic search planner provided by Bonet and Geffner [31], which is fully compatible with our STRIPS-based formalism. It is essential to notice that no computational method can treat all branches as a planning problem leading to proper solutions. Therefore, the planner must adequately handle situations in which 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 such failure cases, our method removes the branch from the tree structure. On the positive case, when our algorithm successfully solves a planning problem, the resulting plan defines the branch's storyline. Specifically, the system uses the final state of this plan to establish the intermediate state of the branch's child node. This child node can then be used as the initial state to continue the story by generating new tree branches.

Branch generation is a recursive process that starts at the root node of the tree and ends when all leaf nodes have height H, which is a parameter of our algorithm that defines the expected height for the tree of each quest. The process is repeated for all new child nodes until the expected height of the tree is achieved. When the process ends, the resulting tree will be a candidate branching quest to be added to the initial population of individuals. The entire process is iterated n times to generate the initial population of the genetic algorithm. In our experiments, we work with populations of 100 individuals.

3.4. Selection, Crossover, and mutation steps

To select individuals for reproduction, we use the *fitness proportionate selection method* (also known as roulette wheel selection [24]). In addition, to avoid decreasing the quality of solutions from one generation to the next, we also apply the *elitist selection strategy* [25].

Once the individuals are selected for reproduction, the next step of the algorithm is the crossover process. Giving as input two parents (P_1 and P_2), the first step of the crossover consists of finding compatible states in the nodes of P_1 and P_2. The notation P_kS_i denotes the state S_i of the branching quest P_k. A pair of states from two parents (P_1S_i and P_2S_j) is considered *compatible* when:

1. Both states are composed of identical ground literals ($P_1S_i = P_2S_j$), named *equal* type. Therefore, all branches originating in P_1S_i can also occur in P_2S_j and vice versa without violating the logical coherence of the quest. The crossover operation resulting from this compatibility type is identified by $P_1S_i \leftrightarrow P_2S_j$ (meaning that branches can be exchanged in both nodes);
2. P_1S_i is a subset of P_2S_j ($P_1S_i \subset P_2S_j$), named *sub* type. In this case, all branches that originate from P_1S_i can also occur in P_2S_j, but not the opposite. The crossover operation that results from this type of compatibility is identified by $P_1S_i \rightarrow P_2S_j$ (meaning that branches from the node in the left side of the expression can be copied to the node in the right side of the expression);
3. P_1S_i is a superset of P_2S_j ($P_1S_i \supset P_2S_j$), named *super* type. Consequently, all branches originating from P_2S_j can also occur in P_1S_i, but not the other way around. The crossover operation for this compatibility type is identified by $P_1S_i \leftarrow P_2S_j$ (meaning that branches from the node in the right side of the expression can be copied to the node in the left side of the expression).

After identifying the types of compatible states, the tree structures of both parents (P_1 and P_2) are traversed in order to produce two new child quests (C_1 and C_2). Nodes that are not part of any pair of compatible states, are directly copied to child quests (non-compatible nodes of P_1 are copied to C_1 and non-compatible nodes of P_2 are copied to C_2). As illustrated in Fig. 5, the crossover operation is performed on each pair of compatible nodes according to their compatibility types. For $P_1S_i \leftrightarrow P_2S_j$ (*equal* type), branches are swapped and copied from both parents' nodes to C_1 and C_2 using a uniform crossover method (i.e., each branch of P_1S_i and P_2S_j has an equal probability of going to its respective node in C_1 or C_2). For $P_1S_i \rightarrow P_2S_j$ (*sub* type), each branch that originates from P_1S_i has a probability of 50% of being copied to the compatible node of P_2S_j in the new child C_2. When a branch is not copied, the original branch from P_2S_j is used in its place (according to the order in which branches are established in P_2S_j and P_1S_i). The other branches of the new child C_1 in the node of P_1S_i are a direct copy of the original branches from P_1S_i. Similarly, for $P_1S_i \leftarrow P_2S_j$ (*super* type), branches originating in the node of P_2S_j are copied, with 50% of probability, to the node of P_1S_i in the new child C_1. As illustrated in Fig. 5, the crossover process generates two different children.

After the crossover operation, the generated children are submitted to a mutation procedure, where we randomly select a node from the tree and then remove that node or try to generate new branches starting from it. Considering a mutation probability M_p, a random mutation type M_t (add or remove), and a random target node M_s, the mutation is performed with probability M_p over node M_s, changing it according to M_t. When performing an "add" mutation type, a new branch is randomly generated using the *quest-giver* and the *quest motif* originally defined for the quest.

The control parameters of our genetic algorithm (population size, selection strategy, elitism factor, crossover method, mutation probability, and the number of generations) were defined through a series of experiments conducted during the development of the algorithm. As ought to be expected in a general genetic algorithm, variations in parameters such as the crossover method, elitism factor, and mutation probability affect the balance between exploitation and exploration. While exploitation takes advantage of the existing individuals to create the next generation, exploration expands the scope of the search space by using mutations to add more variation to the population. By systematically testing different control parameters while evaluating the evolution progress of the genetic algorithm and the quality of the generated quests, we were able to establish the best control parameters for our testing domain (described in Section 4), which are: population size of 100 individuals, elitism factor of 2%, the crossover method described above, mutation probability of 20%, and termination condition of 100 generations. These parameters are also in accordance with the study conducted by Mills et al. [26].

3.5. Fitness function

In our method, the utility/quality of a quest is estimated by how well its plot fits into a desired story arc [8]. Fig. 6 illustrates one of the most popular story arcs: the three-act structure.

Considering that story arcs are composed of falls and rises in the tension axis, we represent a story arc of a plot d as a sequence of symbols $d_{arc}^{sym} = \{s_1, s_2, \cdots, s_n\}$, where each s_i can be: "+" to indicate rise; or "-" to indicate fall; or "=" to indicate that the tension level is maintained. The number of symbols in the sequence represents the discretized time axis. For example, the three-act story arc d illustrated in Fig. 6 can be expressed as: $d_{arc}^{sym} = \{+,+,+,-\}$.

The symbolic representation of a story arc can also be converted into a numeric representation $d_{arc}^{num} = \{v_1, v_2, \cdots, v_n\}$, where each v_i of d_{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 d_{arc}^{num} is given by:

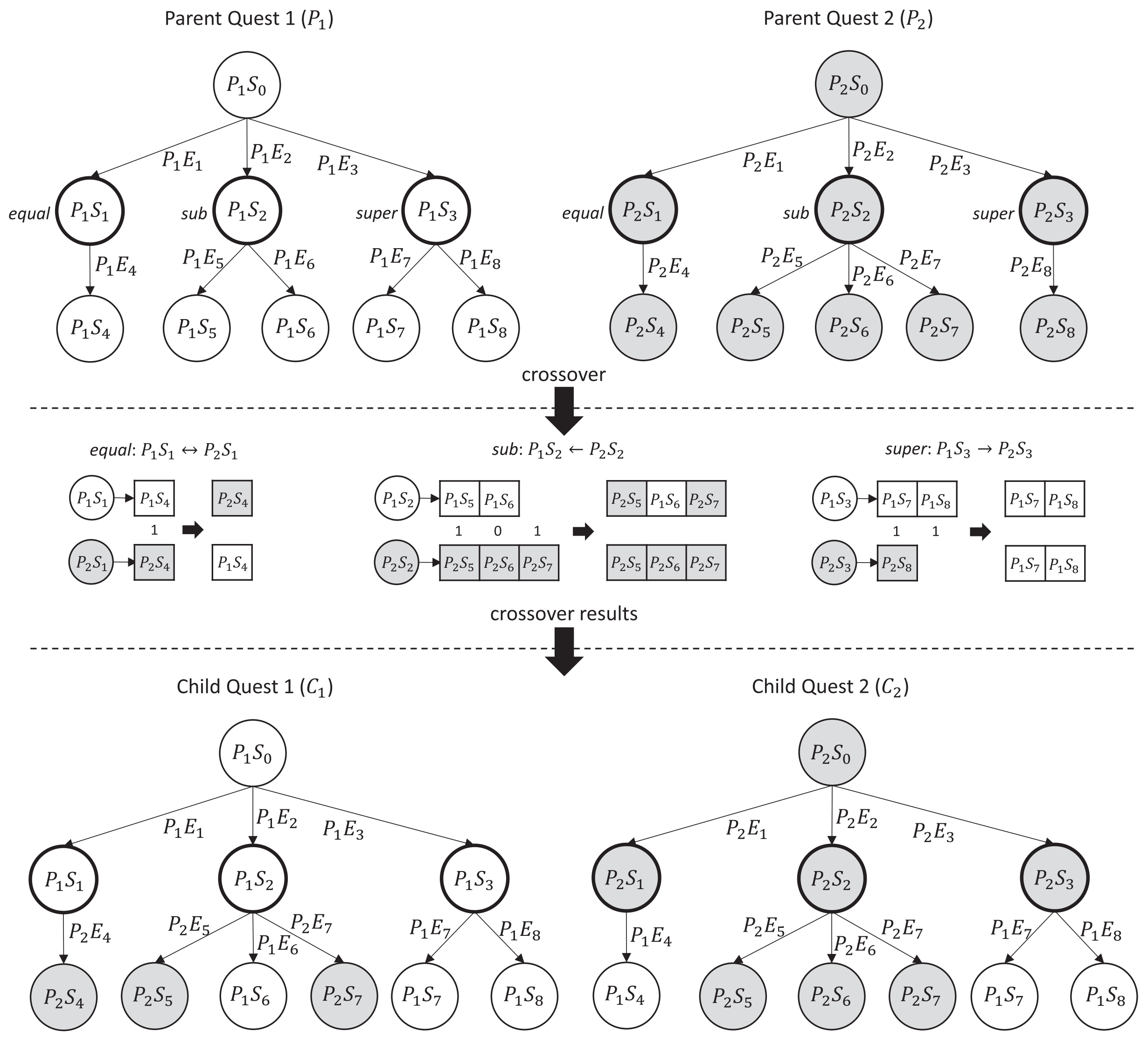

Fig. 5. Example of a crossover operation between two parents (P_1 in blank circles and P_2 in gray circles), with the three types of compatible nodes (*equal*, *sub* and *super*).

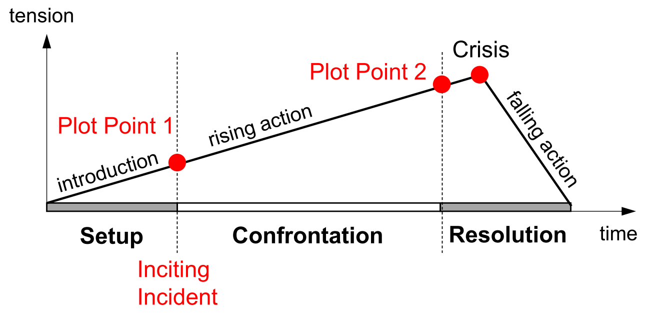

Fig. 6. Story arc as a three-act structure.

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

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

In our method, the fitness function takes as input a desired story arc (provided by the user using the symbolic notation) and the tree structure of an individual of the population. Then, to calculate the fitness of the individual, the function performs four steps: (1) traverse the individual's tree structure and extract all plot variants that can occur in the course of the quest; (2) estimate the story arcs of all quest variants using the information defined in $\mathrm{T} \subseteq DB$, which describes how each event affects the overall tension of the quest; (3) compare the story arcs of all plot variants with the desired story arc to calculate the average similarity between the branching quest and the desired story arc; and (4) calculate the fitness of the individual based on the average similarity with the desired story arc and a set of assessment factors that can be used to prioritize quests with certain characteristics, such as the average length of the possible plots, the total number of branches (i.e., possible interaction/decision points in the plot), and the recurrence of certain terms in the events of several branches (an indication of continuity in the different branches of the quest).

Fig. 7 illustrates a branching quest p with 5 branches: E_1 to E_5 (see Appendix A).

When traversing p, three plot variants can be composed: $p_1 = E_1 + E_3$; $p_2 = E_2 + E_4$; and $p_3 = E_2 + E_5$. Assuming that $\mathrm{T} \subseteq DB$ defines the effects of the operators as $\mathrm{T} = \{$ *request:*"+", *go:*"=", *ask-item:*"+", *give:*"+", *deliver:*"-", *cure:*"-", *get:*"+", *attack:*"+", *lose-item:*"+", *report-failed-to-get:*"+", *request-escort:*"+", *go-two:*"=", *ask-cure:*"+", *kill:*"+",

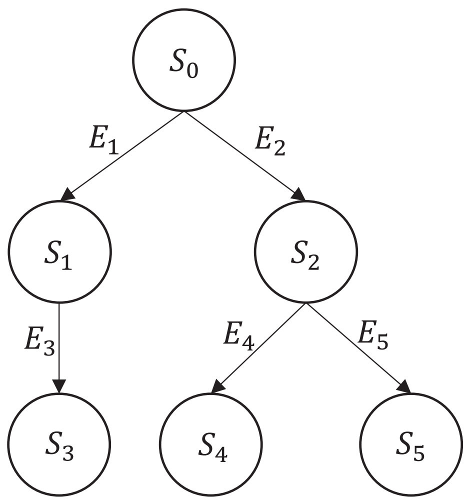

Fig. 7. Example of a branching quest p with 5 branches (see Appendix A).

loot:"+", *make-antidote:*"-",...}, the story arc for each plot variant of p can be expressed in the symbolic notation as:

$$p_{1_{arc}}^{sym} = \{+,=,=,+,+,=,=,-,-\}$$

$$p_{2_{arc}}^{sym} = \{+,=,=,+,+,+,=,=,+,+,=,=,+,+,+,=,=,+,+,=,\\ =,-,-\}$$

$$p_{3_{arc}}^{sym} = \{+,=,=,+,+,+,=,=,+,+,+,=,=,+,+,+,=,=,+,=,\\ =,-,-,-,-\}$$

and converted to its numeric representation:

$$p_{1_{arc}}^{num} = \{1,1,1,2,3,3,3,2,1\}$$

$$p_{2_{arc}}^{num} = \{1,1,1,2,3,4,4,4,5,6,6,6,7,8,9,9,9,10,11,11,11,10,9\},$$

$$p_{3_{arc}}^{num} = \{1,1,1,2,3,4,4,4,5,6,7,7,7,8,9,10,10,10,11,11,11,10,9,8,7\}$$

The estimation of how well the story arc of a plot resembles the desired story arc is similar to the method we proposed in [8]: we calculate the difference of the two arcs after scaling them. However, we improved the calculation of the scaled arcs. The first step is the same, where we consider that story arcs can have different time and tension scales, and we perform normalization by scaling both arcs to standard intervals. In our experiments, time is scaled to the interval [1,10], and tension is scaled to the interval [0,1]. In the present work, we improved the normalization formula by considering the extreme values of the tension:

$$\forall_j \in \{1,\cdots,10\} p_{i_{arc_j}}^{scaled} = \frac{p_{i_{arc}}^{num}\left\lceil \frac{j-1}{10}\left(\overline{p_{i_{arc}}^{num} - 1}\right)\right\rceil + 1 - \min(p_{i_{arc}}^{num})}{\max\left(p_{i_{arc}}^{num}\right) - \min(p_{i_{arc}}^{num})}$$

where $\overline{p_{i_{arc}}^{num}}$ represents the length of $p_{i_{arc}}^{num}$, and $\min(p_i)$ and $\max(p_i)$ are functions that return the minimum and maximum tension values of p_i. For example, the three-act story arc $d_{arc}^{num} = \{1,2,3,2\}$ is scaled to:

$$d_{arc}^{scaled} = \{0.33, 0.33, 0.66, 0.66, 0.66, 1.00, 1.00, 0.66, 0.66, 0.66\}$$

In the same way, the plots of the examples presented above can be scaled to:

$$p_{1_{arc}}^{scaled} = \{0.33, 0.33, 0.33, 0.66, 1.00, 1.00, 1.00, 1.00, 0.66, 0.33\}$$

$$p_{2_{arc}}^{scaled} = \{0.09, 0.18, 0.36, 0.36, 0.54, 0.54, 0.81, 0.81, 1.00, 0.81\}$$

$$p_{3_{arc}}^{scaled} = \{0.09, 0.18, 0.36, 0.45, 0.63, 0.63, 1.00, 1.00, 0.81, 0.63\}$$

With the story arcs scaled to the same intervals, their differences can be revealed in a visual comparison (as shown in Fig. 8).

The difference between two scaled story arcs (d and p_i) is given by the mean squared error, as proposed in [8]:

$$mse(p_i, d) = \frac{1}{n}\sum_{j=1}^{n}\left(p_{i_{arc_j}}^{scaled} - d_{arc_j}^{scaled}\right)^2$$

where n is the maximum value used in the scaled time interval of the story arc (in our experiments, $n = 10$).

Considering that a branching quest comprises multiple storylines (each one with a different story arc), the differences between the desired story arc and the story arcs of possible plots that can occur must be combined to evaluate their average similarity. Therefore, the fitness of a branching quest p, according to a desired story arc d, is calculated as:

$$fitness(p, d) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\overline{p_i}}{mse\left(p_{arc}^{scaled}, d_{arc}^{scaled}\right)}\right) \times f_1 \times f_2 \cdots \times f_k$$

where m is the number of plot variants of p, $\overline{p_i}$ represents the length of p_i (i.e., the number of events in the plot), and f_i are assessment factors that can be used to prioritize certain characteristics in the generated quests.

Although several assessment factors are supported, such as the average length of the possible plots (prioritizing quests with long or short storylines) or the total number of branches (prioritizing quest with more/less decision points in the plot), we used in our experiments only one assessment factor that accounts for the general continuity of the events that take place in different branches of the quest. The *continuity factor* is defined by the number of recurrences of important terms in the events of descendant branches of the quest. Considering that terms whose recurrence contributes to the continuity factor are defined in the semantic integrity rules $\Gamma \subseteq DB$, the process to calculate the value of the *continuity factor* for a quest p involves the task of traversing the tree structure of p counting the number of occurrences of each term identified by $\Gamma \subseteq DB$ in different descendant branches of p (recurrences in the same branch are not counted).

For example, considering the branching quest p (illustrated in Fig. 7) and the three-act story arc d as the desired story arc (Fig. 6), the mean squared errors for each plot variant p_i are:

$$mse(p_1, d) = \frac{1}{10}\sum_{j=1}^{10}\left(p_{1_{arc_j}}^{scaled} - d_{arc_j}^{scaled}\right)^2 = 0.044$$

$$mse(p_2, d) = \frac{1}{10}\sum_{j=1}^{10}\left(p_{2_{arc_j}}^{scaled} - d_{arc_j}^{scaled}\right)^2 = 0.067$$

$$mse(p_3, d) = \frac{1}{10}\sum_{j=1}^{10}\left(p_{3_{arc_j}}^{scaled} - d_{arc_j}^{scaled}\right)^2 = 0.048$$

In this example, if we consider that the semantic integrity rules $\Gamma \subseteq DB$ identify the following recurrence importances of the item type term in the predicate symbols request, deliver, and cure:

$$\gamma_1 = (request, (character, character, ?item, place))$$

$$\gamma_2 = (deliver, (character, character, ?item, place))$$

$$\gamma_3 = (cure, (character, character, ?item, place))$$

then, the recurrence of item *antidote2* can be identified in branches E_2 and E_4 as follows:

E_2 = ***request***(*anne*, *john*, ***antidote2***, *johnhome*), *go*(*john*, *johnhome*, *village*), ….

E_4 = …, ***request***(*bob*, *john*, ***antidote2***, *hospital*), *give*(*bob*, *john*, *gun1*, *hospital*), …, *go*(*john*, *village*, *hospital*), ***deliver***(*john*, *bob*, ***antidote2***, *hospital*), ***cure***(*bob*, *anne*, ***antidote2***, *hospital*).,

where the item ***antidote2*** has its first occurrence in E_2 and then has 3

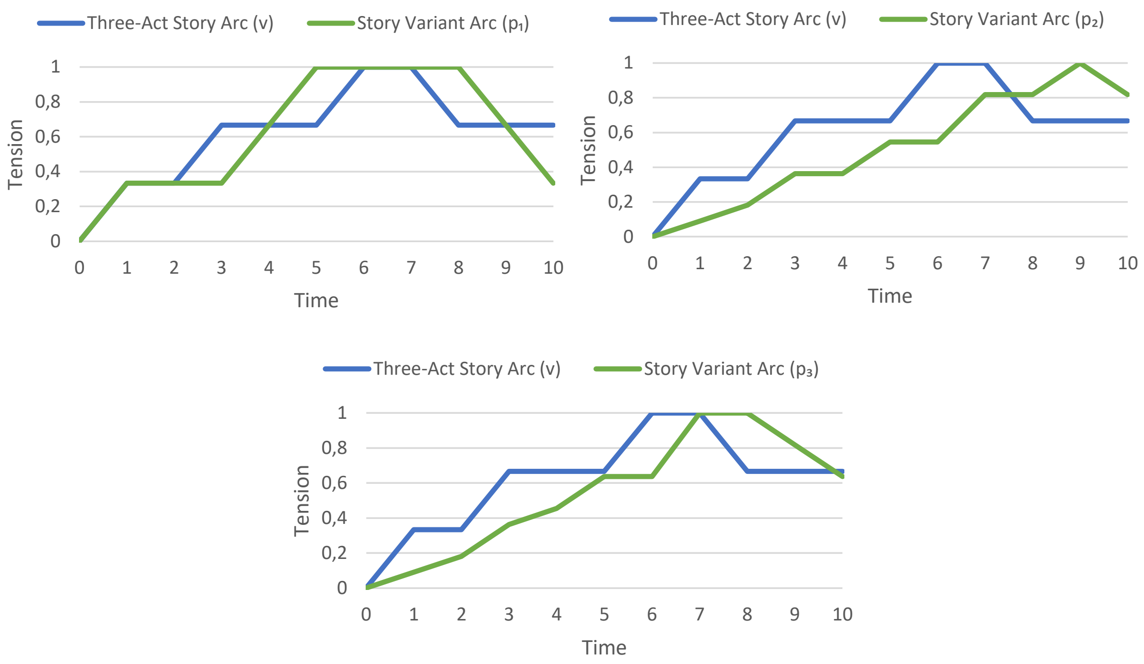

Fig. 8. Visual comparison of the desired story arc (v) and the story arcs of three plot variants of quest p (p_1, p_2, and p_3).

recurrences in the plot of the descendant branch E_4 (considering only the events identified in $\Gamma \subseteq DB$). Therefore, the *continuity factor* is defined by $f_1 = 3$.

Considering that the number of events of each plot variant p_i are: $\overline{p_1} = 9, \overline{p_2} = 23$, and $\overline{p_3} = 25$ (the full list of events of each branch E_i is presented in Appendix A), the fitness of p can be calculated as:

$$fitness(p,d) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\overline{p_i}}{mse\left(p_{i\,arc}^{scaled}, d_{arc}^{scaled}\right)}\right) \times f_1 \times f_2 \cdots \times f_k$$
$$= \frac{1}{3}\left(\frac{9}{0.044} + \frac{23}{0.067} + \frac{25}{0.048}\right) \times 3 = 1068.66$$

3.6. Optimizations

The process to generate branches for the individuals of the population is the most computationally expensive part of our genetic algorithm, mainly because it involves executing a planner to create the storyline for each branch. To optimize this process, we implemented two optimization features: parallel branch generation and cached plots.

Considering that each branch represents an independent planning problem, we can take advantage of multi-core processors to calculate the storyline for multiple branches in parallel. For this purpose, we use a thread pool that maintains multiple threads waiting for tasks (solve the planning problem of a branch) to be allocated for concurrent execution.

The occurrence of branches with equal planning problems along with the same or different generations is another characteristic of the populations of our genetic algorithm, especially for small game worlds. In these cases, the same planning problem would need to be solved multiple times. To optimize this process, we implemented a memorization (a.k.a. memoization) technique to store the plots generated from previous planning problems. When an already encountered 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 (the generated plots).

4. Application

The game used to test and validate our method is a 2D zombie survival RPG developed for a previous project [6,8,9,10] (Fig. 9), in which we incorporated the proposed architecture to use branching quests generated by our genetic algorithm. The game was implemented in Lua[3] using the Löve 2D framework.[4] See [8] for more details about the game world.

The branching quests to compose the game tree structure of the prototype game were created by combining quests produced by the proposed method with quests manually created by a professional game designer. This approach will be helpful to evaluate whether human players can differentiate quests produced by our algorithm from those created by a human game designer (a kind of Turing Test, described in Section 5.2).

In order to create this hybrid game tree, our quest generation method was used in a more interactive way: (*i*) first, we applied our algorithm to generate just a single branching quest for the root node of the game tree (the first quest of the game); (*ii*) then, we asked a professional game designer – with over 12 years of game industry experience – to design

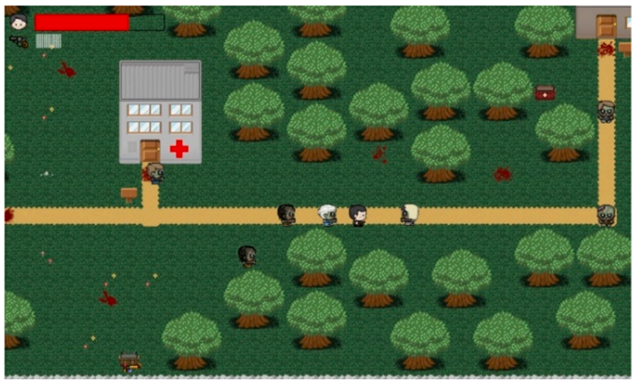

Fig. 9. Scene of the prototype game.

[3] Lua is the well-known programming language developed at the Pontifical Catholic University of Rio de Janeiro, Brazil (https://www.lua.org/).

[4] https://love2d.org/.

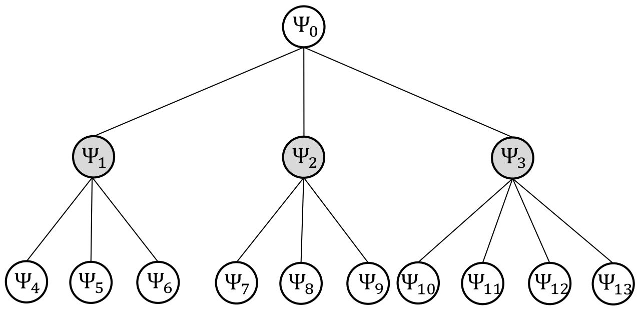

Fig. 10. Game tree with the branching quests used in the prototype game. Quests ψ_1, ψ_2, and ψ_3 were created by a professional game designer and the other quests were generated by the proposed genetic algorithm.

new branching quests for our game using the final states of the previous quest as the starting point for the new quests; (*iii*) after adding the designer's quests to the game tree, we executed our algorithm again, but using the final states of the designer's quests as the initial state for the new quests. We also limited the height of the game tree to 2, so only one new level of quests was generated by the algorithm. Although our method could generate more quests for this domain, we limited the height of the game tree to 2 in order to reduce the length of the prototype game and allow all players to complete the game during the Turing Test (described in Section 5.2). The resulting game tree is illustrated in Fig. 10, where shaded nodes represent the branching quest created by the professional game designer and non-shaded nodes are branching quests generated by our genetic algorithm. By having all designer's quests in the intermediate level of the game tree, all players can experience two quests generated by our algorithm and one quest manually created by the designer, independently of the chosen path.

When running the genetic algorithm to generate the branching quests for the game, we employed a fixed population size of 100 individuals. Each run of the algorithm comprised 100 generations, and the complexity of the generated branching quests was limited by a maximum height $H = 2$ and a maximum branching factor $b = 2$. For all runs, the three-act structure was used as the desired story arc.

As illustrated in Fig. 10, the game comprises 14 quests (3 created by a professional game designer and 11 generated by our genetic algorithm). The branching quest ψ_0 is the same quest described in the example presented in Section 3.5, which has its structure illustrated in Fig. 7. In general, a quest has several branches, and we denote the branch E_i of the quest ψ_k as $\psi_k E_i$. One example of quest created by the game designer is quest ψ_3, which has the structure illustrated in Fig. 11 – *a*, and its branches are composed of many sequences of events. For example:[5]

$\psi_3 E_5$ = *request-kill*(*rick*, *john*, *villagezombie*, *store*), *go*(*john*, *store*, *village*), *kill*(*john*, *villagezombie*, *oldgun*, *village*), *go*(*john*, *village*, *store*), *report-kill*(*rick*, *john*, *villagezombie*, *store*), *give*(*rick*, *john*, *food1*, *store*), *go* (*john*, *store*, *johnhome*), *deliver*(*john*, *anne*, *food1*, *johnhome*), *feed*(*anne*, *maggie*, *food1*, *johnhome*).

The events for all the branches shown in Fig. 11 – *a* can be found in Appendix B.

An example of a quest generated by our algorithm to follow the outcome of branch $E_5 \in \psi_3$ is quest ψ_{11} (Fig. 11 – *b*). This quest has six branches and each branch has several events. For instance:

$\psi_{11} E_1$ = *request-escort*(*anne*, *john*, *johnhouse*, *islandeast*), *go-two*(*john*, *anne*, *johnhome*, *forest*), *get*(*john*, *wood1*, *forest*), *go-two*(*john*, *anne*, *forest*, *store*), *get*(*john*, *toolkit*, *store*), *go-two*(*john*, *anne*, *store*, *villageislandbridge*), *fixbridge*(*john*, *toolkit*, *wood1*, *villageislandbridge*), *go-two*(*john*, *anne*, *villageislandbridge*, *islandeast*).

The events for all the branches presented in Fig. 11 – *b* can be found

[5] Sequences of go events were condensed (for example, “go(john, johnhome, village), go(john, village, store)” is represented as “go(john, johnhome, store)”).

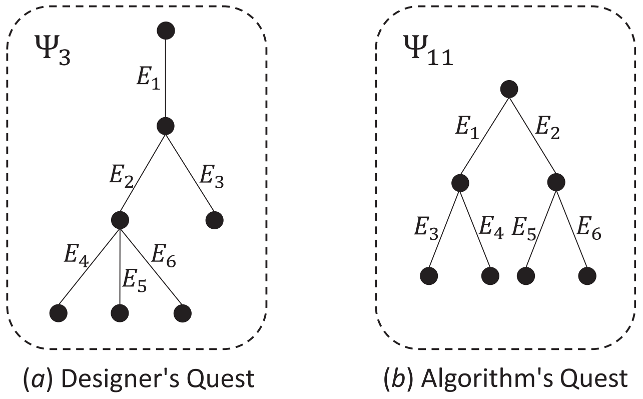

Fig. 11. Examples of branching quests: (*a*) structure of quest ψ_3 created by the professional game designer; and (*b*) structure of the quest ψ_{11} created by the genetic algorithm.

in Appendix C.

Dialogs and descriptive interaction options for branching points were automatically generated by our game according to the plot events and a set of templates created to translate logical ground propositions into generic natural language sentences. More details about this process are presented in our previous work on quest generation [6].

The source code of the proposed genetic algorithm is available on the website of our project: https://www.icad.puc-rio.br/~logtell/genetic-quests/.

5. Evaluation and results

The evaluation of our method comprised two tests: (1) a technical test to assess the evolution progress and the performance of the proposed genetic algorithm; and (2) a simplified Turing Test to evaluate if human players would be able to differentiate quests produced by our algorithm from those created by a professional game designer.

5.1. Technical evaluation

To evaluate the evolution progress, we performed 30 runs of the genetic algorithm to generate a single branching quest for the game world of our prototype (described in Section 4). For this experiment, the following settings were used in the genetic algorithm: population size of 100 individuals; elitism factor of 2%; the crossover method described in Section 3.4; mutation probability of 20%; termination condition of 100 generations; complexity of the generated branching quests limited by a maximum height $H = 3$ and a maximum branching factor $b = 2$; and the three-act structure as the desired story arc.

To compare the results produced by the genetic algorithm with a “random quest generation strategy”, we implemented a new algorithm that creates random quests using the method to generate new individuals described in Section 3.3. However, no evolutionary strategy was employed. For every generation, 100 new random quests were created and evaluated.

The results of this experiment are shown in Fig. 12. As can be noticed, the genetic algorithm overcomes the random strategy within just a few generations. In addition, less than 35 generations proved to be enough for the individuals to converge to the optimal solution for this problem.

To evaluate the computational performance of our method, we calculated the average time required to process a population of 100 individuals during one generation of the genetic algorithm. 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

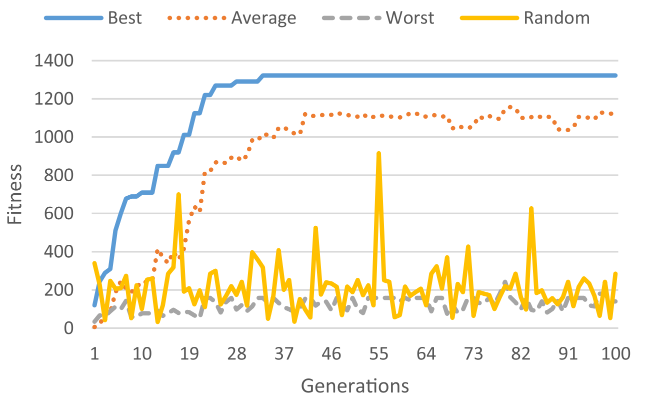

Fig. 12. Average evolution progress of the best, average, and worst individuals of the population during 30 runs of the genetic algorithm in comparison with the best random quests created with a random strategy.

Table 1
Average time (in seconds) to generate the initial population and process a subsequent generation of 100 individuals 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**
Initial Population	Time (sec)	307.32	224.56	90.87
	Standard Deviation	56.75	29.63	25.44
Nth-Generation	Time (sec)	11.17	10.46	8.21
	Standard Deviation	14.68	12.93	11.32

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 30 full runs of the genetic algorithm, each one testing 100 generations. Considering that the generation of the initial population for our algorithm naturally requires more time than subsequent generations,[6] we analyzed the performance of these steps separately. Table 1 shows the results of the tests, which indicate that the optimized version is more than 3 times faster than the base version for the generation of the initial population.

5.2. *Turing Test*

For the simplified Turing Test, we asked 38 students (30 male and 8 female, aged 17 to 24) to play our prototype game and classify quests according to whether they were created by a human designer or by a machine. This test was part of the planned activities in training students in mastering game development techniques at IADE, Universidade Europeia. Subjects were informed that some quests were created by an algorithm and others by a game designer, but they were not informed which quests, nor even how many quests, were created by each one. As described in Section 4, our prototype game uses a hybrid game tree (Fig. 10) that allows players to experience in a single playthrough two quests generated by our algorithm and one quest manually created by a game design professional. At the end of each quest, subjects were prompted to judge if the quest was created by a machine or by a human game designer. All participants were able to complete the game. On average, each session lasted 16.31 min (standard deviation of 5.78).

[6] When generating the initial population for our algorithm, all branches of all individuals comprise new planning problems that need to be solved by a planner, which is the most time-consuming task of our algorithm. In subsequent generations, new individuals are created by recombining branches that were generated for the initial population. New plans are only required when a mutation operation produces new branches from a node.

Over the entire test set of 114 data points (38 players, each evaluating 3 quests), the “machine” version was correctly identified in 34 cases (out of a total of 76) and the “human” version was correctly identified in 20 cases (out of a total of 38), leading to an overall accuracy of 48.68%.

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 (48.68%) and the ideal Turing Test value (50%) suggests that the computer-generated and the human-designed quests are hardly distinguishable. This high value of accuracy indicates the capacity of the proposed quest generation method to achieve quest plots closely similar to those created by professional game designers.

6. Concluding remarks

This article presented a novel quest generation method that combines planning with an evolutionary search strategy guided by story arcs, generating coherent and diversified branching quests based on a specific narrative structure. As discussed in Section 2, previous works on procedural quest generation for games focus mainly on the generation of quests as linear sequences of events, failing to provide the player with the ability of interfering in the narrative of the game. Our method expands the boundaries of traditional linear quests and focuses on the generation of interactive branching quests. In addition, our method maintains the rigor of the formulation of interactive narratives as planning problems.

Although the proposed method presented good results in our experiments, some methodological considerations and current limitations of our research work must be pointed out. First, since our primary focus was on the technical aspects of the proposed method, we have not yet conducted a rigorous user study to evaluate how the generated quests affect the overall game experience. Although the simplified Turing Test indicates that players are unable to distinguish the generated quests from those created by a professional game designer, we still need to evaluate the impacts of the branching quests in the general player experience, which surely remains a mandatory task in the short term. In particular, an evaluation of the replayability aspects of the generated branching quests is an important subject for future studies.

Secondly, the performance of our method is directly affected by the complexity of the game world, because generating plots with branching event sequences by way of classical planning algorithms is an EXPSPACE-complete problem [32]. Indeed, this remains a concern even after our adoption of the HSP2 heuristic search planner (mentioned in Section 3.3), and finding alternative algorithms to improve the computational performance of our method is also part of our future research goals. On this matter, the use of parallel planning algorithms designed to run on GPUs (Graphics Processing Units), such as the implementation proposed in [33], can offer a viable alternative to solve more complex domains in a reasonable time without further modifications to the general structure of our method. Furthermore, the genetic algorithm itself can also be extended to run on GPUs [34].

Apart from the time required to generate branching quests, we did not observe in our experiments any restrictions related to the complexity of the game world or the complexity of the generated quests. In fact, we had to limit the height of the game tree generated for our prototype game to reduce the game’s length and avoid excessively long testing sessions during the Turing Test. However, when defining the constraints to the quest generation algorithm, such as the branching factor and the maximum height for the branching quests, it is essential to balance these constraints not only according to the expected length for quests but also taking into account the possibilities of events and goals offered by the game world. For example, excessively increasing the branching factor or the maximum height of the generated quests in a game world that does not offer many options of events can lead to the generation of a single branching quest that explores all possible events that can occur in the game world, which will leave no space for the generation of consecutive quests. Although more studies to evaluate the quality of branching quests with distinct complexities are still needed, the preliminary tests conducted during the development of our method seem to indicate that it can produce similar results independently of the quest complexity.

Also, applying any schema-based method, like ours, to game development involves difficulties. First, one must recall that specifying the Domain Database is a manual and time-consuming task, in the course of which game developers must face a variety of challenges. For example, they must create a logical description of the game world, define the possible types of events that can occur as planning operators, and establish all semantic integrity constraints that shall guide the branch generation process of the genetic algorithm. Further research on this subject involves the automatic extraction of such information from existing game structures to simplify the application of our method. Another difficulty comes from implementing the game itself, which has to be robust enough to deal with all story variants in all branching quests. An example is spawning and controlling the instances of items and characters in the locations specified in quest plans, while guaranteeing the consistency between the game world and the logical state described by the quest. In this respect, the adoption of supplementary procedural content generation methods that could automatically adapt the game world to quests is a promising direction for future research.

Although we explored only one fitness function in this work (story arcs), our method, as most genetic algorithms, is open to alternative fitness functions, which can be used to guide the generation process towards different types of quests. Examples of alternative methods to evaluate story plots include the use of specific metrics to assess suspense [35,36,37], flow and motivation [38,39], diversity and choice variety [40], and structural features [41]. Analyzing the results produced by alternative fitness functions also constitutes an interesting point for future research work.

Lastly, we firmly believe that quests with real interactive plots dynamically generated by algorithms are one of the key ingredients for a new and more immersive generation of digital games. The positive feedback we received from players, especially the enthusiasm demonstrated by them while exploring the different options offered by the generated branching quests, is a welcome stimulus for continuing our work.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We would like to thank CNPq (National Council for Scientific and Technological Development) and FINEP (Brazilian Innovation Agency), which belong to the Ministry of Science, Technology, and Innovation, for the financial support.

Appendix A

Examples of branches in Fig. 7 (E_1 to E_5):

E_1 = *request*(*anne, john, antidote1, johnhome*), *go*(*john, johnhome, village*), *go*(*john, village, store*), *ask-item*(*john, rick, antidote1, store*), *give*(*rick, john, antidote1, store*), *go*(*john, store, village*), *go*(*john, village, johnhome*), *deliver*(*john, anne, antidote1, johnhome*).

E_2 = *request*(*anne, john, antidote2, johnhome*), *go*(*john, johnhome, village*), *go*(*john, village, forest*), *get*(*john, antidote2, forest*), *attack*(*zombie1, john, forest*), *lose-item*(*john, antidote2, zombie1, forest*), *go*(*john, forest, village*), *go*(*john, village, johnhome*), *report-failed-to-get*(*john, anne, antidote2, johnhome*).

E_3 = *cure*(*anne, anne, antidote1, johnhouse*).

E_4 = *request-escort*(*anne, john, johnhome, hospital*), *go-two*(*john, anne, johnhome, village*), *go-two*(*john, anne, village, hospital*), *ask-cure*(*anne, bob, hospital*), *request*(*bob, john, antidote2, hospital*), *give*(*bob, john, gun1, hospital*), *go*(*john, hospital, village*), *go*(*john, village, forest*), *kill*(*john, zombie1, gun1, forest*), *loot*(*john, antidote2, zombie1, forest*), *go*(*john, forest, village*), *go*(*john, village, hospital*), *deliver*(*john, bob, antidote2, hospital*), *cure*(*bob, anne, antidote2, hospital*).

E_5 = *request*(*anne, john, ingredient1, johnhome*), *request*(*anne, john, ingredient2, johnhome*), *go*(*john, johnhome, village*), *go*(*john, village, darkforest*), *get*(*john, oldgun, forest*), *kill*(*john, bigzombie1, oldgun, darkforest*), *get*(*john, ingredient2, darkforest*), *go*(*john, darkforest, village*), *go*(*john, village, store*), *get*(*john, ingredient1, store*), *go*(*john, store, village*), *go*(*john, village, johnhome*), *deliver*(*john, anne, ingredient2, johnhome*), *deliver*(*john, anne, ingredient1, johnhome*), *make-antidote*(*anne, ingredient1, ingredient2, johnhome*), *cure*(*anne, anne, antidote1, johnhouse*).

Appendix B

The branches and events of Fig. 11a (created by the professional game designer), and used in the application of Section 4, are the following:

ψ_3E_1 = *starve*(*maggie, johnhome*), *request*(*anne, john, food1, johnhome*), *go*(*john, johnhome, store*).

ψ_3E_2 = *ask*(*john, rick, food1, store*).

ψ_3E_3 = *steal*(*john, rick, food1, store*), *go*(*john, store, johnhome*), *deliver*(*john, anne, food1, johnhome*), *feed*(*anne, maggie, food1, johnhome*).

ψ_3E_4 = *request-payment*(*rick, john, store*), *pay*(*john, rick, store*), *give*(*rick, john, food1, store*), *go*(*john, store, johnhome*), *deliver*(*john, anne, food1, johnhome*), *feed*(*anne, maggie, food1, johnhome*).

ψ_3E_5 = *request-kill*(*rick, john, villagezombie, store*), *go*(*john, store, village*), *kill*(*john, villagezombie, oldgun, village*), *go*(*john, village, store*), *report-kill*(*rick, john, villagezombie, store*), *give*(*rick, john, food1, store*), *go*(*john, store, johnhome*), *deliver*(*john, anne, food1, johnhome*), *feed*(*anne, maggie, food1, johnhome*).

ψ_3E_6 = *request*(*rick, john, hospitalstoragekey, store*), *go*(*john, store, hospital*), *steal*(*john, bob, hospitalstoragekey, hospital*), *go*(*john, hospital, store*), *give*(*john, rick, hospitalstoragekey, store*), *give*(*rick, john, food1, store*), *go*(*john, store, johnhome*), *deliver*(*john, anne, food1, johnhome*), *feed*(*anne, maggie, food1, johnhome*).

Appendix C

The branches and events of Fig. 11b (created by the genetic algorithm), and used in the application of Section 4, are the following:

$\psi_{11}E_1$ = *request-escort*(*anne, john, johnhouse, islandeast*), *go-two*(*john, anne, johnhome, forest*), *get*(*john, wood1, forest*), *go-two*(*john, anne, forest, store*), *get*(*john, toolkit, store*), *go-two*(*john, anne, store, villageislandbridge*), *fixbridge*(*john, toolkit, wood1, villageislandbridge*), *go-two*(*john, anne,* *villageislandbridge, islandeast*).

$\psi_{11}E_2$ = *request-escort*(*anne, john, johnhouse, mountain*), *go-two*(*john, anne, johnhome, forest*), *get*(*john, wood1, forest*), *get*(*john, wood2, forest*), *go-two* (*john, anne, forest, store*), *get*(*john, toolkit, store*), *go-two*(*john, store, villageislandbridge*), *fixbridge*(*john, toolkit, wood1, villageislandbridge*), *go-two*(*john, anne, villageislandbridge, islandmountainbridge*), *fixbridge*(*john, toolkit, wood2, islandmountainbridge*), *go-two*(*john, anne, islandmountainbridge, mountainwest*).

$\psi_{11}E_3$ = *request-shelter*(*anne, john, islandeast*), *go*(*john, islandeast, islandwest*), *ask-item*(*john, matt, wood4, islandwest*), *give*(*matt, john, wood4, islandwest*), *go*(*john, islandwest, islandeast*), *build-shelter*(*john, wood4, toolkit, islandeast*).

$\psi_{11}E_4$ = *request-shelter*(*anne, john, islandeast*), *go*(*john, islandeast, hospital*), *steal*(*john, bob, hospitalstoragekey, hospital*), *go*(*john, hospital, hospital-storagedoor*), *open*(*john, hospitalstoragedoor, hospitalstoragekey*), *go*(*john, hospitalstoragedoor, hospitalstorage*), *get*(*john, metalplate1, hospitalstorage*), *go* (*john, hospitalstorage, islandwest*), *build-shelter*(*john, metalplate1, toolkit, islandeast*).

$\psi_{11}E_5$ = *request*(*anne, john, supplieskit1, mountainwest*), *go*(*john, mountainwest, islandwest*), *ask-item*(*john, matt, supplieskit1, islandwest*), *ask-item* (*matt, john, oldgun, islandwest*), *give*(*john, matt, oldgun, islandwest*), *give*(*matt, john, supplieskit1, islandwest*), *go*(*john, islandwest, mountainwest*), *deliver* (*john, anne, supplieskit1, mountainwest*).

$\psi_{11}E_6$ = *request*(*anne, john, supplieskit2, mountainwest*), *go*(*john, mountainwest, mountaineast*), *get*(*john, supplieskit2, mountaineast*), *attack*(*zombie6, john, mountaineast*), *lose-item*(*john, supplieskit2, zombie6, mountaineast*), *go*(*john, mountaineast, mountainwest*), *report-failed-to-get*(*john, anne, supplieskit2, mountainwest*).

References

[1] M. Hendrikx, S. Meijer, J. Van Der Velden, A. Iosup, Procedural content generation for games: A survey, ACM Trans. Multimedia Comput. Commun. Appl. 9 (1) (2013) 1–22.

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

[3] S. Risi, J. Togelius, Increasing generality in machine learning through procedural content generation, Nature Machine Intelligence 2 (8) (2020) 428–436, https://doi.org/10.1038/s42256-020-0208-z.

[4] J. Togelius, A.J. Champandard, P.L. Lanzi, M. Mateas, A. Paiva, M. Preuss, K. O. Stanley, Procedural Content Generation: Goals, Challenges and Actionable Steps, Artificial and Computational Intelligence in Games 6 (2013) 61–75, https://doi.org/10.4230/DFU.Vol6.12191.61.

[5] A. Amato, Procedural Content Generation in the Game Industry, in: O. Korn, N. Lee (eds) Game Dynamics, Springer, 2017. https://doi.org/10.1007/978-3-319-53088-8_2.

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

[7] B. Kybartas, R. Bidarra, A Survey on Story Generation Techniques for Authoring Computational Narratives, IEEE Trans. Comput. Intell. AI Games 9 (3) (2017) 239–253, https://doi.org/10.1109/TCIAIG.2016.2546063.

[8] E. S. Lima, B. Feijó, A. L. Furtado, Procedural Generation of Quests for Games Using Genetic Algorithms and Automated Planning, in: Proceedings of the XVIII Brazilian Symposium on Computer Games and Digital Entertainment (SBGames 2019), Rio de Janeiro, Brazil, 2019, pp. 495-504. https://doi.org/10.1109/SBGames.2019.00028.

[9] E.S. Lima, B. Feijó, A.L. Furtado, Player Behavior and Personality Modeling for Interactive Storytelling in Games, Entertainment Computing 28 (2018) 32–48, https://doi.org/10.1016/j.entcom.2018.08.003.

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

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

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

[13] T. Chongmesuk, V. Kotrajaras, Multi-Paths Generation for Structural Rule Quests, in: Proceedings of the 16th International Joint Conference on Computer Science and Software Engineering (JCSSE), Chonburi, Thailand, 2019, pp. 97-102. https://doi.org/10.1109/JCSSE.2019.8864168.

[14] P. Ammanabrolu, W. Broniec, A. Mueller, J. Paul, M. O. Riedl, Toward Automated Quest Generation in Text-Adventure Games, arXiv:1909.06283 [cs.CL], Sep. 2019.

[15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, Language models are unsupervised multitask learners, OpenAI Blog 1 (2019) 8.

[16] N. McIntyre, M. Lapata, Plot induction and evolutionary search for story generation, in: Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics, Uppsala, Sweden, 2010, pp. 1562-1572.

[17] T. Ong, J.J. Leggett, A genetic algorithm approach to interactive narrative generation, in, in: Proceedings of the fifteenth ACM conference on Hypertext and hypermedia, 2004, pp. 181–182, https://doi.org/10.1145/1012807.1012856.

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

[19] M. Nairat, P. Dahlstedt, M.G. Nordahl, Character evolution approach to generative storytelling, in, in: Proceedings of the 2011 IEEE Congress of Evolutionary Computation, 2011, pp. 1258–1263, https://doi.org/10.1109/CEC.2011.5949760.

[20] M. Nairat, P. Dahlstedt, 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. https://doi.org/10.1007/978-3-642-36955-1_15.

[21] M.N.R. Utsch, G.L. Pappa, L. Chaimowicz, R.O. Prates, A new non-deterministic drama manager for adaptive interactive storytelling, Entertainment Computing 34 (2020) 100364, https://doi.org/10.1016/j.entcom.2020.100364.

[22] G. Durand, The Anthropological Structures of the Imaginary, Boombana Publications (1999).

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

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

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

[26] K.L. Mills, J.J. Filliben, A.L. Haines, Determining Relative Importance and Effective Settings for Genetic Algorithm Control Parameters, Evol. Comput. 23 (2) (2015) 309–342, https://doi.org/10.1162/EVCO_a_00137.

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

[28] R.E. Fikes, N.J. Nilsson, STRIPS: A new approach to the application of theorem proving to problem solving, Artif. Intell. 2 (3–4) (1971) 189–208, https://doi.org/10.1016/0004-3702(71)90010-5.

[29] J. Howard, Quests: Design, Theory, and History in Games and Narratives, A K Peters/CRC Press, Natick, Massachusetts, 2008.

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

[31] B. Bonet, H. Geffner, Planning as Heuristic Search, Artif. Intell. 129 (1) (2001) 5–33, https://doi.org/10.1016/S0004-3702(01)00108-4.

[32] J. Rintanen, Complexity of planning with partial observability, in, in: Proceedings of the Fourteenth International Conference on International Conference on Automated Planning and Scheduling (ICAPS), 2004, pp. 345–354.

[33] D. Sulewski, S. Edelkamp, P. Kissmann, Exploiting the Computational Power of the Graphics Card: Optimal State Space Planning on the GPU, in, in: Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling (ICAPS), 2011, pp. 242–249.

[34] K. Wang, Z. Shen, A GPU-Based Parallel Genetic Algorithm for Generating Daily Activity Plans, IEEE Trans. Intell. Transp. Syst. 13 (3) (2012) 1474–1480, https://doi.org/10.1109/TITS.2012.2205147.

[35] S. Giannatos, Y.-G. Cheong, M.J. Nelson, G.N. Yannakakis, Generating Narrative Action Schemas for Suspense, in, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2012, pp. 8–13.

[36] B. O'Neill, M. Riedl, Dramatis: A Computational Model of Suspense, in: Proceedings of the AAAI Conference on Artificial Intelligence, AAAI, 2014, pp. 944-950.

[37] Y.-G. Cheong, R. M. Young, Narrative Generation for Suspense: Modeling and Evaluation, in: Proceedings of the Joint International Conference on Interactive Digital Storytelling, Springer, 2008, pp. 144-155. https://doi.org/10.1007/978-3-540-89454-4_21.

[38] S. Giannatos, M. Nelson, Y.-G. Cheong, G. Yannakakis, Suggesting New Plot Elements for an Interactive Story, in, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2011, pp. 25–30.

[39] P. Weyhrauch, Guiding Interactive Drama, School of Computer Science, Carnegie Mellon University, Pittsburgh, 1997. Ph.D. Thesis.

[40] N. Szilas, I. Ilea, Objective Metrics for Interactive Narrative, in, in: Proceedings of the Seventh International Conference on Interactive Digital Storytelling, 2014, pp. 91–102, https://doi.org/10.1007/978-3-319-12337-0_9.

[41] N. Partlan, E. Carstensdottir, S. Snodgrass, E. Kleinman, G. Smith, C. Harteveld, M. Seif El-Nasr, Exploratory Automated Analysis of Structural Features of Interactive Narrative, in, in: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment, 2018, pp. 88–94.