Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Corrêa Fontoura de Oliveira

Desenvolvimento de uma Ferramenta de Criação de Grafos para Storytelling Interativo

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao Programa de Engenharia da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Engenharia da Computação.

Orientador: Prof. Augusto Cesar Espíndola Baffa

Rio de Janeiro
Julho de 2024

Pontifícia Universidade Católica
do Rio de Janeiro

Eduardo Corrêa Fontoura de Oliveira

Desenvolvimento de uma Ferramenta de Criação de Grafos para Storytelling Interativo

Relatório de Projeto Final, apresentado ao Programa de Engenharia da Computação, do Departamento de Informática da PUC-Rio como requisito parcial para a obtenção do titulo de Bacharel em Engenharia da Computação.

Prof. Augusto Cesar Espíndola Baffa
Orientador
Departamento de Informática – PUC-Rio

Rio de Janeiro, 19 de Julho de 2024

Eduardo Corrêa Fontoura de Oliveira

Graduando em Engenharia da Computação na PUC-Rio

Ficha Catalográfica

Eduardo Corrêa Fontoura de Oliveira

Desenvolvimento de uma Ferramenta de Criação de Grafos para Storytelling Interativo / Eduardo Corrêa Fontoura de Oliveira; orientador: Augusto Cesar Espíndola Baffa. – 2024.

50 f: il. color. ; 30 cm

Projeto Final - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2024.

Inclui bibliografia

1. Informática – Teses. 2. Planejamento Automático. 3. Storytelling Interativo. 4. Planejamento em Grafos. 5. Ferramenta de Desenvolvimento. I. Baffa, Augusto. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Aos meus pais, por me incentivarem a ser melhor todos os dias

Agradecimentos

Aos meus pais pelo apoio nos momentos difíceis e presença nos bons, pela oportunidade de estudar na PUC-Rio e carinho incondicional, essencial para essa trajetória.

À minha irmã por me ensinar a relativizar os problemas e resolvê-los com calma.

Aos meus tios e primos por me acolherem de braços abertos em sua casa por um período desse caminho.

À minha namorada por ser meu porto seguro tanto em dias de mares tranquilos como turbulentos.

Aos amigos que fiz em todos esses anos que tornaram essa jornada bem mais leve e prazerosa.

Ao meu grupo de RPG por entender que seria um momento conturbado e por não desistir de mim.

Ao meu orientador pelas sessões de *brainstormings*, conversas *off topic* e por guiar esta tese.

Por fim, agradeço a PUC-Rio pela oportunidade de ter estudado como bolsista em uma instituição de altíssima qualidade.

Resumo

Eduardo Corrêa Fontoura de Oliveira; Baffa, Augusto. **Desenvolvimento de uma Ferramenta de Criação de Grafos para Storytelling Interativo**. Rio de Janeiro, 2024. 50p. Relatório de Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Em jogos interativos, desde videogames à livros jogos, as escolhas apresentadas ao usuário podem ser visualizados como grafos. Motivado por isso, esse trabalho se propõem a implementar uma ferramenta de criação de grafos para apoiar o desenvolvimento de tais histórias. A partir de *inputs* do usuário, busca-se a estruturação de um grafo direcionado, com garantia de caminho para o objetivo determinado, assim como possibilidade de alteração em sua estrutura durante a execução do programa. Este trabalho traz como resultado, além da ferramenta implementada, três arquivos de apoio aos autores: uma visualização do grafo gerado, uma descrição textual da estrutura e uma descrição textual contendo todas as soluções encontradas para ela.

Palavras-chave

Planejamento Automático; Storytelling Interativo; Planejamento em Grafos; Ferramenta de Desenvolvimento.

Abstract

Eduardo Corrêa Fontoura de Oliveira; Baffa, Augusto (Advisor). **Development of a Graph Creation Tool for Interactive Storytelling**. Rio de Janeiro, 2024. 50p. Relatório de Projeto Final – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

In interactive games, from video games to gamebooks, the choices presented to the user can be visualized as graphs. Motivated in that sense, this work proposes to implement a graph creation tool to support the development of such stories. Using inputs from the user, the aim is to structure a random directed graph, with a guaranteed path to the determined objective, as well as the possibility of changing its structure. This work brings as a result, in addition to the implemented tool, three files to support the authors: a visualization of the generated graph, a textual description of the structure and a textual description containing all the solutions found for it.

Keywords

Automated Planning; Interactive Storytelling; Graph Planning; Development Tool.

Sumário

Lista de figuras

Lista de tabelas

Lista de algoritmos

Lista de Códigos

Lista de Abreviaturas

PDDL – *Planning Domain Definition Language*

STRIPS – *Stanford Research Institute Problem Solver*

NPC – *Non-Player Character*

Não faz sentido só lutar.
Nem só sobreviver.
Eu quero vencer!

Kurosaki Ichigo, *Bleach*.

1
Introdução

A definição mais simples de *storytelling* é sua tradução direta, isto é, contar histórias. Seu principal objetivo é criar uma conexão entre a narrativa e o público para transmitir uma mensagem, sentimento ou entretenimento. (DOMINGOS, 2009) Esse método está presente por toda a existência da humanidade, seja através da passagem de conhecimento de forma oral, na leitura de textos ou mais recentemente no consumo de jogos digitais.

A possibilidade de modificar a narrativa através de suas ações como usuário, proporcionada pelo *storytelling* interativo, quebrou o paradigma do cenário tradicional. A história deixou de ser contada unidirecionalmente, abrindo espaço para maior imersão e enredos mais diversos (CAVAZZA; CHARLES; MEAD, 2002). A série de livros-jogos Fighting Fantasy (Steve Jackson and Ian Livingstone), assim como os jogos digitais Until Dawn (Supermassive Games, 2015), Detroit: Become Human (Quantic Dream, 2018), Cyberpunk 2077 (CD Projekt RED, 2020) e Baldur's Gate III (Larian Studios, 2023) são exemplos de obras que já implementam ramificações durante suas narrativas.

Essa transformação do *storytelling* tradicional, linear para o interativo trouxe consigo novas necessidades e desafios, especialmente no que diz respeito ao desenvolvimento e gestão de histórias complexas e ramificadas. Para melhor gerenciar essas narrativas, ferramentas que permitam a visualização e manipulação de suas estruturas são essenciais.

Visualmente, essas histórias podem ser apresentadas como um grafo conectado, onde cada nó representa um evento e seu caminho para o vizinho uma decisão que pode ser tomada. Assim, esse trabalho tem como objetivo a viabilização de uma ferramenta de criação de grafos para auxiliar o desenvolvimento dessa estrutura narrativa.

O presente projeto consiste em gerar inicialmente um grafo direcionado baseado em dois *inputs* do usuário: número de nós e valor máximo de habilidade do personagem, variável utilizada para gamificação da estrutura. Determinado a quantidade de nós, as arestas iniciais são criadas de forma aleatória, conforme restrições predefinidas. Além disso, a fim de proporcionar flexibilidade autoral, é possível realizar customizações durante a execução, como criar novas arestas, garantir caminhos entre dois nós distintos e gerar arestas aleatoriamente.

Esse trabalho foi estruturado de forma a apresentar uma sequencia lógica desde a sua motivação até a sua conclusão. Assim, o capítulo 2 apresenta a

situação atual desse nicho e os trabalhos relevantes ao desenvolvimento da solução do problema. O capítulo 3 versa sobre os objetivos do projeto em si, estabelecendo metas a alcançar e requisitos funcionais para ferramenta desenvolvida. O capítulo 4 aborda os conceitos preliminares, essenciais ao entendimento do trabalho. O capítulo 5 apresenta e justifica as funcionalidades da ferramenta, mostrando através de pseudocódigos sua implementação. Por fim, o capítulo 6 encerra o trabalho com uma conclusão sobre este projeto e propondo melhorias futuras para o software.

2
Situação Atual

O uso da inteligência artificial no campo da narrativa atualmente concentra-se na contribuição para a coesão do texto gerado ou na construção da história. Diversos estudos exploram a geração automática de eventos ou a adaptação de estruturas pré-existentes, reagindo às ações ou informações fornecidas pelo jogador. Embora muitos desses trabalhos utilizem grafos para orientar a construção ou apenas para explicar visualmente o processo, nenhum deles tem como objetivo principal fornecer esses grafos como um produto final para os autores das histórias.

No trabalho intitulado *"Adaptive Branching Quests Based on Automated Planning and Story Arcs"* de Lima, Feijó e Furtado (2021), dada uma missão existente, são calculadas as tensões esperadas para um arco dramático. Com base nas decisões do jogador durante a missão, um planejador em tempo real introduz eventos para alinhar as tensões modificadas pelas ações do jogador com as tensões esperadas para aquele arco de história. Neste estudo, as missões e conexões são representadas por nós e arestas, respectivamente.

Em *"Interactive Narrative: An Intelligent Systems Approach"* de Riedl e Bulitko (2013), são apresentadas diferentes abordagens para manter uma progressão de história coerente com as ações do usuário. Através da modelagem em três eixos principais (autoral, grau de autonomia de NPCs e modelagem do jogador), um gerente de experiência adapta a narrativa ao usuário. Sistemas com propostas distintas apresentam diferentes distribuições de relevância entre esses pilares. As ações e consequências são frequentemente representadas por grafos direcionados.

No artigo *"Interactive Storytelling: A Player Modelling Approach"* de Thue et al. (2021), o modelo PaSSAGE (*Player-Specific Stories via Automatically Generated Events*) é desenvolvido com o objetivo principal de manter e atualizar em tempo real um modelo da personalidade do jogador, dividido em cinco categorias (*Fighter*, *Method-Actor*, *Storyteller*, *Tactician*, *PowerGamer*). A partir desse modelo, são selecionadas missões pré-definidas que possam ser mais apelativas ao tipo de jogador. As possibilidades apresentadas ao usuário são exibidas como uma árvore de escolhas.

No estudo *"NGEP: A Graph-based Event Planning Framework for Story Generation"* de Tang et al. (2022), foi proposta a ferramenta NGEP (*Neural Graph-based Event Planning*) com o objetivo de prever sequências de eventos para a geração de histórias. Como sugerido pelo nome, um grafo de eventos

é automaticamente construído, gerando vizinhos a partir da probabilidade condicional de determinados cenários ocorrerem. Novamente, a construção do grafo é uma etapa secundária do projeto, não sendo prioridade o auxílio aos autores.

No artigo *"Automated Storytelling via Causal, Commonsense Plot Ordering"* de Ammanabrolu et al. (2021), o sistema C2PO (*Commonsense, Causal Plot Ordering*) é utilizado para gerar enredos a partir de relações causais esperadas pelo senso comum. O sistema cria um espaço ramificado de possíveis caminhos a serem seguidos a partir de cada evento apresentado. Grafos são novamente utilizados como forma de armazenamento e visualização dos eventos tratados. Neste estudo, a participação humana ocorre apenas para a verificação e validação dos resultados.

3
Objetivos

Nesta seção, destacam-se os principais objetivos deste trabalho, bem como, os da ferramenta proposta.

3.1
Objetivos da Ferramenta

Como visto no capítulo anterior, os trabalhos desenvolvidos utilizam grafos como uma etapa intermediária, seja conectando textos com maior probabilidade de virem seguidos uns dos outros (AMMANABROLU et al., 2021), seja mapeando etapas de uma história (THUE et al., 2021). No entanto, há uma lacuna no fornecimento desses grafos como ferramentas diretamente utilizáveis pelos autores. Os métodos atuais focam em processos internos e visualizações auxiliares, sem entregar o grafo como um produto final que possa ser facilmente integrado e manipulado pelos criadores de histórias.

A ferramenta desenvolvida no presente trabalho visa preencher essa lacuna, proporcionando uma solução que gere grafos para o auxílio da construção de narrativas, oferecendo um produto final robusto e customizável para os autores. Com a capacidade de criar, analisar e visualizar grafos, a ferramenta se torna um auxílio valioso tanto para o desenvolvimento de jogos quanto para outras formas de narrativas interativas.

Com esse software, deseja-se apoiar desenvolvedores, game designers, criadores de aplicativos de storytelling, escritores, roteiristas e complementar ferramentas educacionais que necessitem de uma estrutura flexível e personalizável.

O sistema oferece adaptabilidade na criação de grafos narrativos, permitindo a geração aleatória e a personalização dessas estruturas, com diversas opções de conexão entre nós. A partir da escolha de número de nós pelo usuário, o algoritmo determina quantas conexões um nó apresentará e escolhe aleatoriamente seus destinos, fornecendo uma estrutura diferente a cada interação.

Além disso, proporciona interatividade na personalização, permitindo que os usuários adaptem os grafos gerados, criando novas conexões ou ajustando as já existentes. A partir de um menu, o usuário poderá escolher entre criar conexões diretas, garantir que haja caminho entre dois nós ou criar diversos caminhos aleatórios para um destino.

Por fim, o sistema gera uma representação visual do grafo acompanhada de documentação detalhada que descreve os nós, suas conexões e as soluções

para percorrê-las do nó inicial ao final, facilitando a análise e o planejamento de histórias complexas.

3.2 Requisitos Funcionais

A ferramenta desenvolvida deve atender a uma série de requisitos funcionais para garantir sua eficácia e usabilidade. Primeiramente, ela deve permitir a criação de uma estrutura inicial de nós conectados, assegurando a existência de um caminho entre o nó inicial e o nó final. Além disso, deve possibilitar ao usuário especificar a quantidade desejada de nós no grafo, oferecendo uma estrutura personalizada desde o início do processo.

Adicionalmente, a ferramenta deve ser capaz de receber um nível de habilidade máximo alcançável pelo jogador. Com base nesse nível, dificuldades são atribuídas aleatoriamente a cada nó, contribuindo para a gamificação da estrutura. Caso o nível de dificuldade não seja aplicável à história do usuário, basta que ele atribua o valor zero, ajustando a ferramenta às necessidades específicas de diferentes tipos de narrativa.

O sistema deve calcular e exibir a quantidade de caminhos possíveis entre os nós inicial e final, proporcionando ao usuário uma visão clara das opções narrativas disponíveis. Deve também habilitar a criação de conexões adicionais entre os nós, que podem ser geradas de forma aleatória ou específica, conforme as necessidades do usuário.

Para facilitar a visualização das conexões e dos caminhos narrativos, a ferramenta deve gerar uma imagem do grafo criado. Além disso, deve produzir um arquivo de texto detalhando os nós e suas respectivas conexões, oferecendo uma documentação completa da estrutura do grafo. Por fim, a ferramenta deve gerar um arquivo de texto listando todas as possíveis soluções narrativas, ou seja, os caminhos entre o nó inicial e o nó final, permitindo ao usuário explorar todas as opções de progressão narrativa disponíveis.

Esses requisitos funcionais são essenciais para assegurar que a ferramenta atenda às expectativas dos usuários, proporcionando uma interface robusta e flexível para a criação e análise de grafos narrativos, ao mesmo tempo em que incorpora elementos de gamificação para enriquecer a experiência do usuário.

4
Conceitos Preliminares

Neste capítulo serão apresentados conceitos básicos necessários para a compreensão de diferentes etapas do projeto.

4.1
Grafo

Como apresentado por Chang e Nichols (2006) em sua publicação "*Introduction to Graph Theory*", um grafo pode ser definido como uma estrutura matemática $G = (V, E)$, onde V é o conjunto de todos os vértices e E é o conjunto de todas as arestas. Uma aresta $e \subset E$ conecta quaisquer dois vértices, mesmo que sejam iguais, desde que $v_i \subset V$ e $v_j \subset V$. Em particular, uma aresta é classificada como própria quando conecta dois vértices distintos $v_i \subset V$ e $v_j \subset V$, onde $v_i \neq v_j$.

Para o escopo deste projeto, também é útil definir um grafo simples, que é um grafo onde nenhum vértice possui arestas para si mesmo ou arestas múltiplas conectando os mesmos pares de vértices.

Um caminho entre o vértice v_0 e o vértice v_n é uma sequência alternada de vértices e arestas da forma

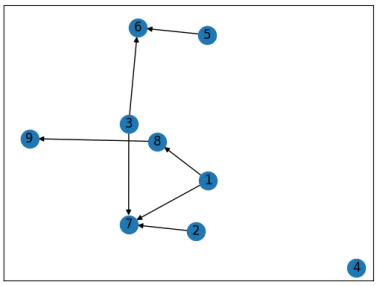
$$P = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \rangle$$

Por fim, um caminho que começa e termina no mesmo vértice configura um ciclo. Consequentemente, um grafo sem ciclos é denominado acíclico.

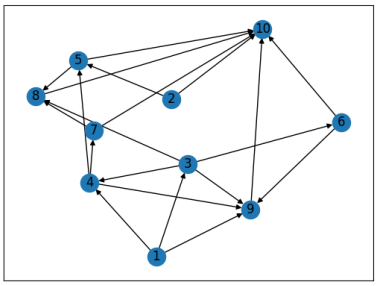
4.1.1
Grafo Direcionado

Antes de apresentar o conceito de grafo direcionado, é necessário entender o que é uma aresta direcionada. Esta é uma aresta onde um vértice, digamos v_i, é denominado cauda e o outro, v_j, é denominado cabeça. Ou seja, a aresta sai de v_i e chega em v_j.

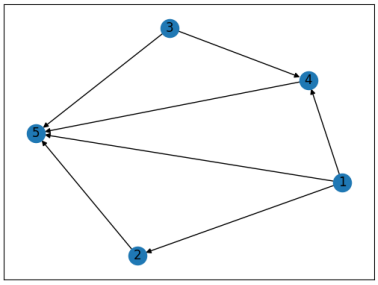
Assim, um grafo é classificado como direcionado quando todas as suas arestas são direcionadas. A figura 4.1(à seguir) apresenta exemplos de grafos direcionados.



(a) Grafo com 9 vértices



(b) Grafo com 10 vértices



(c) Grafo com 5 vértices

Figura 4.1: Exemplos de Grafos Direcionados

4.2 Planejador

Como definido nos livros "*Artificial Intelligence: A Modern Approach*"(RUSSEL; NORVIG, 2021) e "*Automated Planning: Theory and Practice*"(GHALLAB; NAU; TRAVERSO, 2004), o planejamento clássico consiste em encontrar sequências de ações que permitem a transição de um estado inicial para um estado objetivo em um ambiente discreto, determinístico, estático e totalmente observável.

Adicionalmente, utiliza-se a linguagem PDDL, sigla em inglês para *Planning Domain Definition Language*, para modelar os fatos conhecidos e as possíveis ações a serem tomadas, juntamente com suas pré-condições e pós-condições. Embora esse projeto não se utilize de PDDL, esta é útil para compreender o funcionamento da biblioteca WARBETA, a ser apresentada na próxima seção.

Em PDDL, um estado é representado pelo conjunto de fatos válidos em determinado momento da execução. Os fatos devem ser atômicos, referindo-se apenas a um único predicado, não podem conter variáveis e podem mudar ao

longo do tempo. Além disso, por utilizar semântica de banco de dados, qualquer fato desconhecido é assumido como falso pela hipótese do mundo fechado.

Os esquemas de ação representam as operações previstas no domínio. Neles, são definidas as ações permitidas, as pré-condições necessárias para sua execução, isto é, fatos que devem ou não estar presentes, e os efeitos ou pós-condições da ação, ou seja, novos fatos adicionados e removidos em consequência da execução. Um exemplo de esquema de ação é exibido em 4-1:

$$
\begin{aligned}
&Action(goTo(Player, NodeFrom, NodeTo),\\
&\quad \text{PRECOND:}\\
&\qquad \neg Visited(NodeFrom, NodeTo) \wedge\\
&\qquad Path(NodeFrom, NodeTo) \wedge\\
&\qquad At(Player, NodeFrom) \qquad (4\text{-}1)\\
&\quad \text{EFFECT:}\\
&\qquad At(Player, NodeTo) \wedge\\
&\qquad \neg At(Player, NodeFrom) \wedge\\
&\qquad Visited(NodeFrom, NodeTo))
\end{aligned}
$$

Assim, uma ação A só pode ser executada em um estado E se E satisfizer todas as pré-condições de A. A execução da ação resulta na transição do estado E para o estado E', onde os efeitos de A são adicionados e removidos da lista de fatos.

4.3 WARBETA

WARBETA é um sistema desenvolvido por Furtado e Casanova (1990). Implementado em Prolog, seu propósito é a geração de planos em domínios similares aos apresentados pelo Stanford Research Institute Problem Solver (STRIPS). Baseado no sistema WARPLAN(WARREN, 1974), a modelagem conceitual do WARBETA é estruturada em três esquemas: Estático, Dinâmico e Comportamental, dos quais apenas os dois primeiros serão utilizados neste projeto.

No esquema estático, são definidos os fatos. De forma semelhante à teoria de banco de dados, são estabelecidas as entidades, os relacionamentos e os atributos pertinentes ao problema. O conjunto de fatos definidos em determinado momento indica o estado do planejador e suas possibilidades de ação.

No esquema dinâmico, são definidos os eventos, com operações especificadas por suas pré-condições e pós-condições, aproximando-se do método STRIPS. Um evento só pode ser produzido por uma operação se todas as suas pré-condições forem satisfeitas pelos fatos válidos no estado corrente. Sua execução resulta em um novo estado, onde as pós-condições adicionam ou removem fatos do esquema estático.

Após definido o domínio, as chamadas ao sistema são feitas através do procedimento $plans(G, P)$, onde G é um input com o objetivo que se quer atingir e P é um output com o plano encontrado. Mais detalhes sobre os procedimentos serão abordados a seguir.

4.3.1 Operador plans

O operador especial *plans* irá processar os objetivos contidos em G, passando para o predicado *plan* a lista formatada como argumento. Ao final, verificará se o plano obtido é consistente com as restrições definidas pelo domínio.

4.3.2 Predicado plan

Duas instâncias do predicado *plan* são apresentadas através do Algoritmo 1. Ambas recebem como entrada uma lista C com os objetivos a resolver, o plano parcial T e uma lista P com os objetivos já alcançados.

Na primeira instância, o predicado itera sobre os elementos da lista de objetivos, passando um por vez como argumento para *solve*. Em seguida, *plan* é chamado novamente com o próximo objetivo a ser alcançado, configurando uma recursão.

Na segunda instância, que constitui o caso base, como há apenas um objetivo a ser resolvido em G, o predicado *plan* passa-o para *solve* como argumento.

Seu retorno é o plano parcial atualizado T_1, que contém T, preserva os objetivos presentes em P e alcança os objetivos contidos em C.

Algoritmo 1: Predicado *plan*

Entrada: Objetivos a resolver C, Objetivos alcançados P, Plano parcial T

Saída: Plano parcial atualizado T_1

```
1 se C.length() = 1 então
2 |  T_1 ← solve(C[0], P, T)
3 senão
4 |  elem ← C[0]
5 |  (achievedGoals, updatedPlan) ← solve(elem, P, T)
6 |  newGoals ← C − {elem}
7 |  T_1 ← plan(newGoals, achievedGoals, updatedPlan)
8 retorna T_1
```

4.3.3 Operador solve

O operador especial *solve* é apresentado no Algoritmo 2 e conta com três argumentos de entrada, sendo eles o objetivo atômico que se quer alcançar X, um plano parcial T e um conjunto de objetivos já alcançados P. Sua implementação conta com algumas possibilidades de saída.

Caso o objetivo já tenha sido alcançado no estado atual, há o retorno imediato do plano parcial T e da lista de objetivos alcançados P. Uma consideração importante é que o objetivo pode ser tanto positivo, isto é, adicionar um fato, ou negativo, remover um fato.

Caso contrário, é verificado se existe uma operação que adicione ou remova o objetivo em questão, então chama-se o predicado *achieve* que eventualmente retornará um plano parcial T_1 e uma lista de objetivos atualizados P_1.

4.3.4 Predicado achieve

Há duas implementações para o predicado *achieve*, das quais apenas a primeira é utilizada no projeto e apresentada no Algoritmo 3. Ambas recebem como entrada o objetivo atômico X, uma operação U que resulta nele, uma conjunção de fatos válidos P para o estado e um plano parcial T que levou o planejador até esse estado.

Na primeira implementação, também conhecida como extensão, a ação que satisfaz o objetivo é colocada no final do plano. Primeiro, as pré-condições do objetivo são processadas e transformadas numa lista C. Em seguida, é verificado se o objetivo atual não gera *loop* nessa lista. Então, é checada a consistência entre as pré-condições C e os fatos já alcançados P. O predicado *plan*

Algoritmo 2: Predicado *solve*

Entrada: Objetivo atômico X, Objetivos alcançados P, Plano parcial T

Saída: Lista de objetivos atualizada P_1, Plano parcial atualizado T_1

```
1 se X ⊆ P então
2     retorna (P, T)
3 senão
4     se ∃ operação U que resulta em X então
5         T_1 ← achieve(X, U, P, T)
6         P_1 ← {X} + P
7         retorna (P_1, T_1)
8     senão
9         Falha
```

é chamado novamente para as pré-condições determinadas. Tendo executado o planejador, verifica-se se todas as pré-condições foram alcançadas no plano parcial atualizado T_1. Ainda, o sistema averigua se o objetivo atômico X, ao ser executado, tem o resultado esperado de adição ou deleção e determina se ele preserva os fatos previamente alcançados P, finalmente retornado o novo plano parcial T_1.

A segunda implementação, também conhecida como inserção, se diferencia por colocar o objetivo X no meio do plano parcial T. Em suma, ele retraça o plano até um ponto onde a inserção é possível e não gera conflitos e tenta inseri-lo nesse estado. Essa abordagem não é utilizada no projeto devido ao fato de o planejador ser usado somente para encontrar um caminho que sai de um ponto inicial à um final. Dessa forma, os objetivos são sempre alcançados de forma progressiva, não havendo necessidade de inserir objetivos intermediários em um plano em andamento.

Para melhor visualização o pseudocódigo para o primeiro método é exibido abaixo:

Algoritmo 3: Predicado *achieve*

Entrada: Objetivo atômico X, Operação U, Objetivos alcançados P, Plano parcial T

Saída: Plano parcial atualizado T_1

```
1 C ← preconditions(U)
2 se noloop(X, C) ∧ consistent(C, P) então
3 |   T_1 ← plan(C, P, T)
4 |   se no_goal(C, T_1) então
5 |   |   se productive(U, T_1) ∧ preserves(U, P) então
6 |   |   |   retorna T_1
7 retorna Falha
```

5
Implementação

5.1
Escopo

Para alcançar os objetivos estabelecidos, o projeto utiliza técnicas de geração de grafos para criar a estrutura das histórias e o planejador WARBETA para percorrer os grafos gerados, determinando quantos caminhos existem entre o nó inicial e o nó final.

São utilizados grafos direcionados, acíclicos e simples. Grafos direcionados permitem maior controle sobre os possíveis caminhos a serem seguidos. Grafos acíclicos evitam que o planejador entre em *loop* durante a busca por soluções. Grafos simples, além de evitarem ciclos por não possuírem nós conectados a si mesmos, permitem identificar exatamente quais arestas foram percorridas.

Como o problema se enquadra em um domínio discreto, determinístico, estático e totalmente observável, é possível utilizar o planejamento clássico para sua resolução.

O diagrama que exibe as conexões entre as diferentes partes do projeto é apresentado na figura 5.1.

O fluxo de execução da ferramenta inicia-se com os *inputs* do usuário para a criação do grafo. Em seguida, o grafo é gerado aleatoriamente e verificado

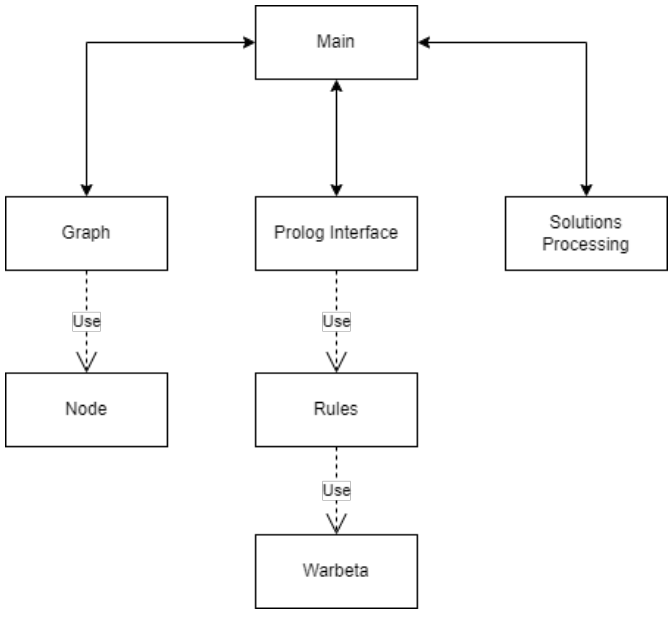


Figura 5.1: Diagrama de conexões

para garantir pelo menos um caminho entre o nó inicial e o nó final. As possíveis soluções são analisadas e apresentadas ao usuário, mostrando a quantidade de caminhos encontrados e uma imagem da estrutura gerada. Neste ponto, um menu é exibido, permitindo ao usuário salvar e sair, sair sem salvar, modificar a estrutura apresentada ou gerar um novo grafo aleatório.

5.2 Bibliotecas

O trabalho se apoia em três bibliotecas externas de Python apresentadas nas seções abaixo.

5.2.1 Biblioteca PySwip

PySwip (TEKOL; CONTRIBUTORS, 2020) é uma biblioteca, ainda em desenvolvimento e incompleta, que estabelece uma conexão entre Python e SWI-Prolog, permitindo a execução de consultas em SWI-Prolog a partir de programas escritos em Python. Esta ferramenta fornece uma interface de linguagem estrangeira para SWI-Prolog, uma classe utilitária que simplifica a realização de consultas em Prolog, e uma interface em estilo Python, tornando sua utilização mais intuitiva para desenvolvedores.

Por meio dessa biblioteca, realiza-se a inserção de dados relacionados ao grafo na base de dados de Prolog.

5.2.2 Bibliotecas networkx e matplotlib.pyplot

NetworkX (HAGBERG; SCHULT; SWART, 2008) é uma biblioteca em Python voltada para a criação, manipulação e estudo da estrutura, dinâmica e funções de redes complexas. Esta ferramenta oferece recursos para o estudo de redes sociais, biológicas e de infraestrutura, além de proporcionar uma interface de programação padrão e uma implementação de grafos adequada para diversas aplicações. NetworkX facilita o desenvolvimento rápido de projetos colaborativos e multidisciplinares e integra-se a algoritmos numéricos existentes e códigos escritos em C, C++ e FORTRAN. A biblioteca também permite o manuseio eficiente de grandes conjuntos de dados não padronizados.

Utilizando NetworkX, é possível carregar e armazenar redes em vários formatos de dados, tanto padrão quanto não padrão, gerar diferentes tipos de redes aleatórias e clássicas, analisar suas estruturas, construir modelos, desenvolver novos algoritmos e visualizar redes, entre outras funcionalidades.

A biblioteca Matplotlib (HUNTER, 2007) possibilita a geração de visualizações estáticas, animadas e interativas em Python.

Neste trabalho, a biblioteca NetworkX, em conjunto com matplotlib.pyplot, é utilizada para transformar os grafos criados em imagens.

5.3 Estrutura Node

A estrutura mais elementar do projeto são os nós que compõem o grafo. Cada nó é inicializado com um identificador numérico, um grau de dificuldade e uma lista de vizinhos inicialmente vazia.

O identificador numérico serve para distinguir cada nó, enquanto o grau de dificuldade é utilizado para auxiliar na construção de histórias gamificadas, nas quais um evento está associado a um desafio que pode ou não ser superado pelo jogador.

A lista vazia de vizinhos identifica as conexões existentes para aquele nó. Como os grafos gerados são direcionados, os elementos presentes na lista recebem uma aresta partindo desse nó. Além disso, como o grafo é acíclico, qualquer caminho selecionado nunca retornará ao nó sendo analisado. Por fim, sendo também um grafo simples, o identificador do próprio nó nunca estará presente na sua lista de vizinhos.

O único método presente nessa classe (Algoritmo 4) tem como objetivo adicionar uma escolha à lista de vizinhos de um nó. Em outras palavras, o método *addChoice* cria um caminho saindo do nó em questão e chegando em algum vizinho. Após isso, remove qualquer duplicata que possa ter sido inserida na lista.

Algoritmo 4: Método *addChoice* da classe *Node*

Entrada: Lista de vizinhos *choices*, Escolha *choice*
Saída: Lista de vizinhos *choices* atualizada

1 $choices \leftarrow choices + \{choice\}$
2 *Remove duplicatas de choices*
3 **retorna** *choices*

5.4 Classe Graph

A classe *Graph* utiliza diversos objetos *Node*, conforme descrito anteriormente. Ela é inicializada com uma lista vazia destinada a armazenar os nós que serão inseridos. Além disso, contém um *layout* de distribuição e uma estrutura de nós, ambos inicialmente vazios, que serão utilizados para a exibição.

O método *clearGraph* (Algoritmo 5) tem como função realizar um *reset* nas variáveis mantidas pela classe. Dessa forma, permite que a estrutura armazene outro grafo diferente do que estava em sua memória.

Algoritmo 5: Método *clearGraph* da classe *Graph*

Entrada: Lista de nós *nodeList*, Layout de Distribuição dos nós *pos*, Estrutura de nós *DG*

Saída: Lista de nós *nodeList*, Layout de Distribuição dos nós *pos*, Estrutura de nós *DG*

```
1 nodeList ← Lista vazia
2 pos ← Layout vazio
3 DG ← Estrutura vazia
```

O método *generateRandomGraph* (Algoritmo 6) cria um grafo aleatório com base na quantidade de nós especificada pelo usuário. Os nós são criados progressivamente, conectados a vizinhos e armazenados na lista de nós.

A geração do grafo conta com quatro possíveis tratamentos. Se o número de nós solicitado pelo usuário for zero, retorna-se uma estrutura vazia. Se for um, apenas esse nó é adicionado à estrutura. Se for dois, ambos os nós são adicionados à lista e uma conexão é estabelecida entre o primeiro e o segundo nó. Nos demais casos, a aleatoriedade influenciará a criação do grafo.

Primeiramente, é atribuída uma dificuldade a cada nó, variando de zero ao valor máximo de habilidade informado pelo usuário. Caso esse atributo não seja relevante para o contexto da história, basta inserir o valor zero.

Em seguida, a quantidade de vizinhos é determinada com base na posição do nó na sequência da história. Nós mais próximos do início têm maior probabilidade de possuir vários vizinhos, enquanto aqueles próximos do final tendem a ter menos ramificações. Isso é feito porque no início da história, há uma ampla gama de opções disponíveis, porém, à medida que as decisões são tomadas, essas opções tornam-se progressivamente mais limitadas. É necessário comprometer-se com as escolhas feitas, o que reflete a natureza do processo decisório. Conforme a narrativa avança, o número de alternativas diminui, simbolizando o caminho escolhido ao longo do percurso.

Os vizinhos são selecionados aleatoriamente, sempre com um número maior que o do próprio nó, garantindo assim que o grafo seja acíclico, pois as arestas são direcionadas, permitindo a navegação apenas em ordem crescente de nós. Devido à aleatoriedade dessa seleção, em muitos cenários nem todos os nós serão alcançáveis a partir do nó inicial. Embora esse problema pudesse ser facilmente resolvido realizando uma busca a partir do nó inicial e mantendo no grafo apenas os nós alcançados, optou-se por manter os nós não alcançados pelo inicial devido ao seu potencial narrativo. Um nó inacessível pelo primeiro

pode representar uma ação realizada por um NPC, outro jogador ou um evento secundário à linha principal de missões.

Algoritmo 6: Método *generateRandomGraph* da classe *Graph*

Entrada: Lista de nós *nodeList*, Quantidade de nós *nodeQtd*, Habilidade máxima *maxAbility*
Saída: Lista de nós *nodeList* atualizada

```
1  clearGraph()
2  se nodeQtd = 0 então
3      retorna nodeList vazia
4  node.number = 1
5  node.difficulty = 0
6  se nodeQtd = 1 então
7      nodeList ← nodeList + {node}
8      retorna nodeList
9  qtdNeighbors ← randomFromTo(1, 5)
10 para i = 0 até qtdNeighbors - 1 faça
11     node.addChoice(randomFromTo(node.number + 1, nodeQtd))
12 nodeList ← nodeList + {node}
13 se nodeQtd = 2 então
14     node.number = 2
15     node.difficulty = 0
16     nodeList ← nodeList + {node}
17     retorna nodeList
18 senão
19     para nodeNum = 2 até nodeQtd - 1 faça
20         node.number = 2
21         node.difficulty = randomFromTo(0,maxAbility)
22         qtdNeighbors ←
             número de vizinhos ponderado por ponto da história
23         para i = 0 até qtdNeighbors - 1 faça
24             node.addChoice(randomFromTo(node.number + 1,
               nodeQtd))
25         nodeList ← nodeList + {node}
26     node.number = nodeQtd
27     node.difficulty = 0
28     nodeList ← nodeList + {node}
29     retorna nodeList
```

O método *createPath* (Algoritmo 7) cria uma ligação direta entre dois nós. Após realizar testes para determinar que ambos os nós existem e que a criação do caminho não infringirá a propriedade de aciclicidade, o nó de destitino é adicionado como uma possibilidade na lista de vizinhos do nó de partida.

Algoritmo 7: Método *createPath* da classe *Graph*

Entrada: Nó de partida $nodeFrom$, Nó de destino $nodeTo$
Saída: Lista de nós $nodeList$ atualizada

```
1 se nodeFrom ⊆ nodeList então
2     se nodeTo ⊆ nodeList então
3         se nodeFrom < nodeTo então
4             nodeFrom.addChoice(nodeTo)
```

A função *ensurePath* (Algoritmo 8) assegura a existência de um caminho entre dois nós. Inicialmente, realiza os mesmos testes do método *createPath*. Em seguida, executa uma busca parcial no grafo a partir do nó inicial fornecido como parâmetro.

Durante a busca, verifica-se primeiramente se o nó inicial possui vizinhos. Se não houver, o nó de destino é adicionado à lista. Caso contrário, verifica-se se o nó de destino já está presente entre os vizinhos. Se estiver, nenhuma ação adicional é necessária. Se não estiver, realiza-se uma verificação adicional. Se todos os nós na lista de vizinhos tiverem numeração maior que o nó de destino, este é adicionado à lista. Caso contrário, seleciona-se o primeiro nó na lista que tenha numeração menor, e a função é chamada recursivamente, passando como argumentos o nó selecionado e o nó de destino original.

Essa busca parcial é justificada pela aleatoriedade envolvida, uma vez que a lista de vizinhos não necessariamente está em ordem crescente de numeração, além de permitir a criação de mais caminhos entre os nós. Como o próprio nome do método indica, sua intenção é garantir a existência de pelo menos um caminho, e não verificar se ele já existe.

Algoritmo 8: Método *ensurePath* da classe *Graph*

Entrada: Nó de partida $nodeFrom$, Nó de destino $nodeTo$

Saída: Lista de nós $nodeList$ atualizada

```
1  se nodeFrom ⊆ nodeList então
2      se nodeTo ⊆ nodeList então
3          se nodeFrom < nodeTo então
4              se ¬empty(nodeFrom.choices) então
5                  se nodeTo ⊆ nodeFrom.choices então
6                      retorna
7                  senão se nodeTo < all(nodeFrom.choices) então
8                      nodeFrom.addChoice(nodeTo)
9                  senão
10                     para choice ∈ nodeFrom.choices faça
11                         se choice < nodeTo então
12                             retorna ensurePath(choice, nodeTo)
13             senão
14                 nodeFrom.addChoice(nodeTo)
```

O método *createRandomPaths* (Algoritmo 9) recebe como argumentos o nó de destino e o número de tentativas de criação de novos caminhos. A cada iteração, um nó com número menor que o nó de destino é selecionado aleatoriamente, e tenta-se criar um caminho utilizando o método *createPath*.

O objetivo principal é aumentar a aleatoriedade e diversificar os caminhos no grafo, sem se prender rigidamente ao número de tentativas especificado. Essa abordagem previne erros caso o número de tentativas seja maior que a quantidade de nós no grafo ou superior ao número do nó de destino escolhido.

Algoritmo 9: Método *createRandomPaths* da classe *Graph*

Entrada: Nó de destino $nodeTo$, Número de tentativas $numberOfTries$

Saída: Lista de nós $nodeList$ atualizada

```
1  para i = 1 até numberOfTries faça
2      nodeFrom ← randomNode(1, nodeTo − 1)
3      nodeFrom.addChoice(nodeTo)
```

O método *saveGraphText* (Algoritmo 10) é responsável por salvar o grafo gerado em um arquivo de texto. Seu funcionamento é direto: ele abre um

arquivo denominado *generatedGraph.txt* e itera sobre a lista de nós, escrevendo em cada linha o número do nó, seu grau de dificuldade e seus vizinhos. Se um arquivo com esse nome já existir, ele será sobrescrito, se não, um novo arquivo será criado.

Algoritmo 10: Método *saveGraphText* da classe *Graph*

Entrada: Lista de nós *nodeList*
Saída: Arquivo de texto *generatedGraph.txt*

```
1 Abre ou cria arquivo generatedGraph.txt
2 para node ∈ nodeList faça
3     arquivo.Write(node.number, node.difficulty, node.choices)
4 Fecha arquivo generatedGraph.txt
```

A classe *Graph* também é encarregada da representação gráfica do grafo armazenado. Para isso, foram implementados dois métodos: *drawGraph* e *saveGraphPNG*. Ambos dependem da biblioteca NetworkX para seu funcionamento correto.

O método *drawGraph* transforma o grafo contido na lista de nós em uma estrutura própria da biblioteca NetworkX, armazenando tanto seus nós quanto arestas. Em seguida, define a disposição a ser seguida pelos nós em sua exibição e gera a visualização através da biblioteca Matplotlib.

Já o método *saveGraphPNG* salva a representação gráfica do grafo com base na estrutura e nas posições previamente definidas por *drawGraph* e criando um arquivo chamado *generatedGraph.png* utilizando a função *savefig* da biblioteca Matplotlib.

5.5 Rules

O arquivo *rules.pl* (Código 1) estabelece o domínio e as operações permitidas ao planejador.

Na primeira seção, mencionada no Capítulo 4 como esquema Estático, são definidas as entidades, atributos e relações presentes no problema.

Na segunda seção, denominada esquema Dinâmico, é especificada a operação permitida ao planejador: *goTo*. Este predicado recebe o nome do jogador, o número do nó de partida e o número do nó de destino, e verifica se as pré-condições são satisfeitas. As condições exigem que o jogador esteja no nó de partida, possua habilidade suficiente para superar o grau de dificuldade do evento, exista um caminho entre o nó de partida e o nó de destino, e que a aresta não tenha sido visitada anteriormente. Se todas essas condições forem atendidas, o estado do planejador é atualizado: o jogador é movido para o nó

de destino, sua posição anterior é removida da memória e a aresta percorrida é marcada como visitada.

Código 1: rules.pl

```
% domain specification

:- dynamic pAt/2, path/2, notVisited/2.

entity(player, name).
entity(node, number).
attribute(player, ability).
attribute(node, difficulty).
relationship(pAt, [player, node]).
relationship(path, [node, node]).
relationship(notVisited, [node,node]).


% operation

operation(goTo(Player, NodeFrom, NodeTo)).
added(pAt(Player, NodeTo), goTo(Player, NodeFrom, NodeTo)).
deleted(pAt(Player, NodeFrom), goTo(Player, NodeFrom, NodeTo)).
deleted(notVisited(NodeFrom, NodeTo), goTo(Player, NodeFrom, NodeTo)
    ).
precond(goTo(Player, NodeFrom, NodeTo),
        (notVisited(NodeFrom, NodeTo),
         path(NodeFrom, NodeTo),
         (difficulty(NodeFrom,DifficultyValue), ability(Player,
             AbilityValue), AbilityValue>=DifficultyValue),
         pAt(Player, NodeFrom))).
```

5.6 Prolog Interface

A classe *prologInterface* é responsável por instanciar o SWI-Prolog e administrar as chamadas e respostas dele. É através dela que é feita a tradução dos nós presentes no grafo, com suas conexões e níveis de dificuldade para o Prolog. Outra responsabilidade dessa interface é coletar e armazenar as respostas fornecidas para que posteriormente sejam formatadas pelo módulo *SolutionsProcessing*.

O método *consultProlog* (Algoritmo 11) tem como objetivo inserir na base de conhecimento do Prolog o domínio definido no arquivo *rules.pl*.

O método *inputProlog* (Algoritmo 12) recebe um grafo e o traduz para Prolog. Ele define, sempre inserindo os fatos ao final da base de conhecimento, as informações sobre o jogador, e sua posição inicial. Em seguida, percorrendo

Algoritmo 11: Método *consultProlog* da classe *prologInterface*

Entrada: Arquivo Prolog *rules.pl*

```
1 Consulta arquivo rules.pl
```

todos os elementos do grafo, cria o nó, define sua dificuldade e insere todos as arestas que partem dele. Por último, marca elas como não visitadas.

Algoritmo 12: Método *inputProlog* da classe *prologInterface*

Entrada: Lista de nós *nodeList*

```
1 Insere no Prolog a entidade Jogador
2 Insere no Prolog a posição inicial do Jogador
3 para node ∈ nodeList faça
4     Insere no Prolog a entidade Nó(node.number)
5     Insere no Prolog a dificuldade do Nó
6     para vizinho ∈ node.choices faça
7         Insere no Prolog o caminho entre o Nó e o Vizinho
8         Insere no Prolog que o caminho não foi visitado
```

Outra funcionalidade essencial deste módulo é o procedimento *findSolutions* (Algoritmo 13). Este recurso tem como objetivo identificar todas as soluções possíveis para o grafo, dado um valor de habilidade do jogador. Para que um jogador tenha sucesso no evento representado por um nó, sua habilidade deve ser igual ou superior ao nível de dificuldade do nó. Dessa forma, o procedimento oferece uma visão sobre o balanceamento da estrutura do grafo, permitindo avaliar quantas formas existem de alcançar o nó final para diferentes níveis de competência. Se o valor de habilidade não for relevante para o contexto, como em uma novela, ele pode ser definido como zero.

O algoritmo recebe como entradas o maior valor de habilidade possível para o jogador e o nó de destino desejado. Inicialmente, o valor máximo de habilidade é inserido na base de conhecimento do Prolog. Em seguida, todas as soluções para atingir o nó de destino são buscadas. Após essa busca, o valor de habilidade é removido da base de conhecimento, decrementado em uma unidade, e o ciclo é reiniciado. Este processo continua até que não existam soluções para o valor de habilidade fornecido ou que o valor de habilidade se torne negativo.

Este procedimento é fundamental para a análise e desenvolvimento de jogos e narrativas gamificadas, pois permite ajustar a dificuldade e garantir que existam múltiplos caminhos para alcançar os objetivos, de acordo com as habilidades do jogador.

Algoritmo 13: Método $findSolutions$ da classe $prologInterface$

Entrada: Valor máximo de habilidade $higherAbilityPossible$, Nó de destino $nodeTo$

Saída: Lista de soluções $solList$

```
1 abilityPoints ← higherAbilityPossible
2 enquanto ¬empty(iterationSolutions) ∧ abilityPoints ≥ 0 faça
3     clear(iterationSolutions)
4     Insere no Prolog a habilidade do jogador com valor abilityPoints
5     iterationSolutions ← Soluções para alcançar nó nodeTo
6     solList.append((abilityPoints, iterationSolutions))
7     abilityPoints ← abilityPoints − 1
```

A última tarefa atribuída a este módulo é o método *resetDatabase* (Algoritmo 14), cuja função, como o nome sugere, é limpar a base de conhecimento inserida no Prolog.

Devido à biblioteca *PySwip* ainda estar em desenvolvimento, algumas funcionalidades essenciais não foram implementadas, incluindo a capacidade de reiniciar a conexão com o SWI-Prolog. Portanto, o método *resetDatabase* é necessário para excluir todos os fatos e predicados previamente inseridos, permitindo a geração de um novo grafo. Sem esse método, a continuidade do projeto seria inviável, pois os nós e conexões da primeira estrutura interfeririam nas estruturas subsequentes.

Este procedimento é vital para garantir a integridade e a correção dos dados ao testar diferentes configurações de grafos. Ele assegura que cada execução comece com uma base de conhecimento limpa, evitando conflitos e interferências de dados anteriores, o que é crucial para a precisão dos resultados obtidos.

Algoritmo 14: Método $resetDatabase$ da classe $prologInterface$

```
1 Remove do Prolog o predicado Jogador
2 Remove do Prolog o predicado Posição do Jogador
3 Remove do Prolog todos os predicados Nó
4 Remove do Prolog todos os predicados Dificuldade do Nó
5 Remove do Prolog todos os predicados Caminho
6 Remove do Prolog todos os predicados Nó Não Visitado
```

5.7 Solutions Processing

A biblioteca *Solutions Processing* inclui duas funções auxiliares para a manipulação e armazenamento das soluções geradas.

A função *formatSolutions* recebe um dicionário contendo soluções para diferentes níveis de habilidade e formata essas soluções removendo caracteres específicos. As soluções formatadas são armazenadas em um novo dicionário, que é então retornado. A função utiliza expressões regulares para a formatação e assegura que as soluções para cada nível de habilidade sejam agrupadas corretamente.

Já função *saveSolutions* salva no arquivo de texto *Solutions.txt* as soluções encontradas para diferentes níveis de habilidade. Para cada habilidade, ela escreve o nível de habilidade, a quantidade de soluções encontradas e as próprias soluções, separando os diferentes conjuntos de soluções com uma linha em branco. Este processo assegura que o arquivo resultante seja organizado e fácil de ler.

5.8 Função Main

Por último temos a função *main*. Além de ser a responsável por fazer a conexão entre todos os módulos quando apropriado, ela pode ser vista como um menu que guiará o usuário através da criação da estrutura narrativa.

Seu funcionamento se inicia a partir do momento que o código é executado na linha de comando. Primeiramente, é requerido ao usuário que insira a quantidade de nós desejada e o valor máximo de habilidade alcançável ao jogador. É válido destacar que o último nó será sempre considerado o nó de destino. A partir desses *inputs* um grafo aleatório é gerado. Verifica-se, então, se há mais de um nó presente na estrutura, e, em caso afirmativo, o método *ensurePath* é chamado para garantir solução até o nó final.

Após essas etapas, é estabelecida a conexão com SWI-Prolog a partir do módulo *PrologInterface*. O arquivo *rules.pl* é consultado para que o domínio e a operação permitida sejam adicionados à base de conhecimentos.

O grafo é exibido e inicia-se a etapa de customização, onde um menu com as opções de Salvar e sair, Sair sem salvar, Customizar grafo e Gerar outro grafo aleatório é exibido.

Caso a terceira opção seja escolhida, outras opções são apresentadas. Torna-se possível Criar um caminho, Garantir um caminho, Criar caminhos aleatórios e Recarregar o grafo. Destaca-se que as três primeiras opções são chamadas para os métodos mencionados acima *createPath*, *ensurePath* e

createRandomPaths, respectivamente, enquanto que a última opção sai desse menu, reescreve o grafo na base de conhecimento do Prolog e reexibe sua representação visual.

Algoritmo 15: *Main*

Entrada: Quantidade de nós $numberOfNodes$, Habilidade máxima $maxAbility$

Saída: Estrutura de grafos conectados própria para desenvolvimento de narrativas baseadas em escolhas

```
1  repeat ← True
2  graph ← generateRandomGraph(numberOfNodes, maxAbility)
3  se numberOfNodes ⩾ 2 então
4      ensurePath(1, numberOfNodes)
5  consultProlog()
6  enquanto repeat faça
7      inputProlog(graph)
8      solutions ← findSolutions(maxAbility, numberOfNodes)
9      formatSolutions(solutions)
10     para (ability, plans) ∈ solutions faça
11         Exibe len(plans) para cada valor de ability
12     drawGraph()
13     promptAns ← Opção selecionada pelo usuário
14     se promptAns = Salvar e sair então
15         repeat ← False
16         saveGraphText()
17         saveGraphPNG()
18         saveSolutions(solutions)
19     senão se promptAns = Sair sem salvar então
20         repeat ← False
21     senão se promptAns = Customizar grafo então
22         customAns ← Opção selecionada pelo usuário
23         enquanto customAns ≠ Recarregar o grafo faça
24             se customAns = Criar um caminho então
25                 nodeFrom ← Nó de partida do caminho
26                 nodeTo ← Nó de destino do caminho
27                 createPath(nodeFrom, nodeTo)
28             senão se customAns = Garantir um caminho então
29                 nodeFrom ← Nó de partida do caminho
30                 nodeTo ← Nó de destino do caminho
31                 ensurePath(nodeFrom, nodeTo)
32             senão se customAns = Criar caminhos aleatórios então
33                 nodeTo ← Nó de destino do caminho
34                 numberOfTries ← Número de tentativas de criação
35                 createRandomPaths(nodeTo, numberOfTries)
36         resetDatabase()
37     senão se promptAns = Gerar outro grafo aleatório então
38         resetDatabase()
39         numberOfNodes ← Nova quantidade desejada de nós
40         maxAbility ← Nova habilidade máxima alcançável
41         graph ←
             generateRandomGraph(numberOfNodes, maxAbility)
42         se numberOfNodes ⩾ 2 então
43             ensurePath(1, numberOfNodes)
```

5.9 Resultados

Para avaliar os resultados e determinar a eficácia da solução, foram estabelecidas métricas de desempenho esperadas para o sistema. Ao término da execução, espera-se obter uma saída organizada da estrutura do grafo gerado, detalhando suas conexões e o nível de dificuldade atribuído a cada nó. Todas as possíveis soluções devem ser armazenadas em um arquivo separado, especificando, ação por ação, o caminho do nó inicial ao nó final. Além disso, a ferramenta deve gerar uma imagem do grafo para facilitar sua visualização. Durante a execução, a ferramenta deve permitir a interação com o usuário para a customização da estrutura gerada. Indicadores de sucesso incluem a criação de arestas, a garantia de caminhos conectados e a capacidade de reiniciar a criação sem a necessidade de reiniciar a ferramenta.

Para a análise do funcionamento, a ferramenta foi executada em cinco instâncias distintas, com os resultados compilados na Tabela 5.1.

Foram registrados o número de nós do grafo, a habilidade máxima do personagem testada e a quantidade de soluções possíveis utilizando essa habilidade, tanto na geração do grafo quanto após a execução do comando *createRandomPaths* para o nó de destino. Além disso, a tabela exibe a habilidade mínima do personagem necessária para encontrar uma solução, considerando os mesmos cenários, e a quantidade de soluções encontradas em ambos os casos. Como esperado, a geração de mais caminhos para o nó final através do comando *createRandomPaths* não só aumenta as possibilidades de alcançar o nó final com a habilidade máxima, mas também pode reduzir a habilidade mínima necessária para atingir esse objetivo.

Adiciona-se que o comando *createRandomPaths* foi utilizado para simplificação dos testes. O grafo gerado após sua execução poderia ser criado também utilizando o comando *createPath*, porém de forma manual e mais trabalhosa.

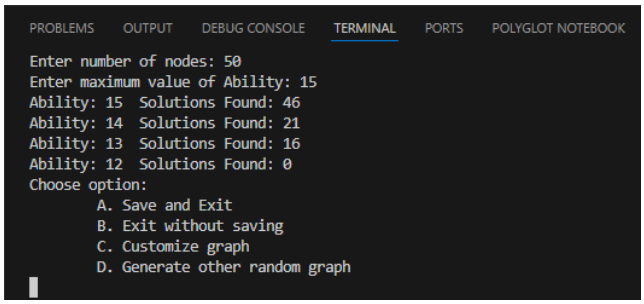
Nº de Nós	Habilidade Máxima	Qtd de Soluções Habilidade Máxima		Habilidade Mínima		Qtd de Soluções Habilidade Mínima	
		Original	Após createRandomPaths()	Original	Após createRandomPaths()	Original	Após createRandomPaths()
10	5	5	8	0	0	1	1
50	10	46	91	4	0	1	1
100	15	52	123	13	6	1	1
200	20	23	40	16	16	5	8
400	20	37	47	10	6	5	1

Tabela 5.1: Resultados obtidos após execução da ferramenta

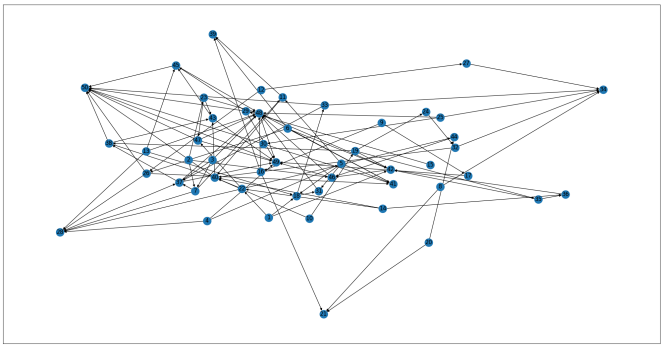
Em outra instância, testaram-se os requisitos de execução. Durante ela, o usuário interage com o *prompt* de comando para orientar a criação dos resultados. Logo após a primeira resposta, são exibidas a visualização gráfica

dos nós e a quantidade de soluções encontradas conforme o nível de habilidade, como na Figura 5.2.

Para o exemplo, optou-se por um grafo de cinquenta nós e valor de habilidade máximo de quinze. Nota-se que para o grau máximo existem quarenta e seis soluções, porém esse número se reduz a menos da metade para apenas um nível de habilidade a menos. Cabe ao autor decidir se essa é a proposta buscada para sua narrativa ou é necessária adaptação do grafo, permitida no menu apresentado logo em seguida.



(a) *Prompt* com *inputs* recebidos e quantidade de soluções encontradas



(b) Grafo gerado a partir dos *inputs*

Figura 5.2: Primeiros *outputs* do programa

Caso o grafo gerado atenda às expectativas, o usuário pode optar por salvar e sair. Esse procedimento resulta na criação de dois arquivos de texto: o primeiro (Figura 5.3a) contém o número dos nós, juntamente com sua dificuldade e conexões, atendendo uma das métricas estabelecidas para saída do programa. Já o segundo (Figura 5.3b) apresenta todas as soluções encontradas para cada nível de habilidade, detalhando o plano a ser seguido para atingir o nó final a partir do nó inicial, atendendo a métrica faltante.

```
Node: 1 Difficulty: 0 Neighbors: [42, 18, 22]
Node: 2 Difficulty: 1 Neighbors: [23, 46, 7]
Node: 3 Difficulty: 11 Neighbors: [48, 37, 7]
Node: 4 Difficulty: 2 Neighbors: [49, 28, 5]
Node: 5 Difficulty: 0 Neighbors: [41, 18, 11, 31]
Node: 6 Difficulty: 3 Neighbors: [41, 26, 42, 39]
Node: 7 Difficulty: 9 Neighbors: [11, 28, 37]
Node: 8 Difficulty: 14 Neighbors: [34, 21]
Node: 9 Difficulty: 1 Neighbors: [16, 17]
Node: 10 Difficulty: 4 Neighbors: [40, 19]
Node: 11 Difficulty: 6 Neighbors: [40, 39]
Node: 12 Difficulty: 10 Neighbors: [26, 27]
Node: 13 Difficulty: 4 Neighbors: [48, 18, 28, 45]
Node: 14 Difficulty: 9 Neighbors: [40, 36, 22]
Node: 15 Difficulty: 4 Neighbors: [29]
Node: 16 Difficulty: 2 Neighbors: [48, 21, 39]
Node: 17 Difficulty: 5 Neighbors: [49]
Node: 18 Difficulty: 13 Neighbors: [33, 31]
Node: 19 Difficulty: 7 Neighbors: [24, 42, 35, 38]
Node: 20 Difficulty: 4 Neighbors: [44, 21]
Node: 21 Difficulty: 2 Neighbors: []
Node: 22 Difficulty: 15 Neighbors: [40, 28, 44, 47]
Node: 23 Difficulty: 6 Neighbors: [48, 43, 47]
Node: 24 Difficulty: 2 Neighbors: [32]
Node: 25 Difficulty: 8 Neighbors: [48, 34, 30]
Node: 26 Difficulty: 12 Neighbors: [41, 50, 28]
Node: 27 Difficulty: 7 Neighbors: [34]
Node: 28 Difficulty: 8 Neighbors: [37]
Node: 29 Difficulty: 12 Neighbors: [49, 45]
Node: 30 Difficulty: 11 Neighbors: [41, 50]
Node: 31 Difficulty: 8 Neighbors: [48]
Node: 32 Difficulty: 6 Neighbors: [40, 34]
Node: 33 Difficulty: 1 Neighbors: [34, 50, 37]
Node: 34 Difficulty: 8 Neighbors: []
Node: 35 Difficulty: 0 Neighbors: [42, 36]
Node: 36 Difficulty: 13 Neighbors: [42]
Node: 37 Difficulty: 7 Neighbors: [48, 49, 38]
Node: 38 Difficulty: 12 Neighbors: [40, 50, 43]
Node: 39 Difficulty: 4 Neighbors: []
Node: 40 Difficulty: 12 Neighbors: [50, 43]
Node: 41 Difficulty: 11 Neighbors: [48]
Node: 42 Difficulty: 14 Neighbors: [48, 46]
Node: 43 Difficulty: 4 Neighbors: [49, 45]
Node: 44 Difficulty: 15 Neighbors: [46]
```

(a) Trecho do arquivo descrevendo a estrutura

```
Ability: 14  Solutions Found: 21
start, goTo(Raygrou, 1, 42), goTo(Raygrou, 42, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 42), goTo(Raygrou, 42, 46), goTo(Raygrou, 46, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 49), goTo(Raygrou, 49,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 31), goTo(Raygrou, 31, 48), goTo(Raygrou, 48, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 42), goTo(Raygrou, 42, 48), goTo(Raygrou, 48, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 31), goTo(Raygrou, 31, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 42), goTo(Raygrou, 42, 46), goTo(Raygrou, 46, 48), goTo(Raygrou, 48, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 48), goTo(Raygrou, 48, 50)
start, goTo(Raygrou, 1, 42), goTo(Raygrou, 42, 46), goTo(Raygrou, 46, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 48), goTo(Raygrou, 48,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 48), goTo(Raygrou, 48,

Ability: 13  Solutions Found: 16
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 31), goTo(Raygrou, 31, 48), goTo(Raygrou, 48, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 49), goTo(Raygrou, 49,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 31), goTo(Raygrou, 31, 48), goTo(Raygrou, 48, 49), goTo(Raygrou, 49, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 48), goTo(Raygrou, 48,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45, 48), goTo(Raygrou, 48,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 38), goTo(Raygrou, 38, 40), goTo(Raygrou, 40, 43), goTo(Raygrou, 43, 45), goTo(Raygrou, 45,
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 50)
start, goTo(Raygrou, 1, 18), goTo(Raygrou, 18, 33), goTo(Raygrou, 33, 37), goTo(Raygrou, 37, 48), goTo(Raygrou, 48, 50)

Ability: 12  Solutions Found: 0
```

(b) Trecho do arquivo descrevendo soluções encontradas

Figura 5.3: Arquivos de texto gerados

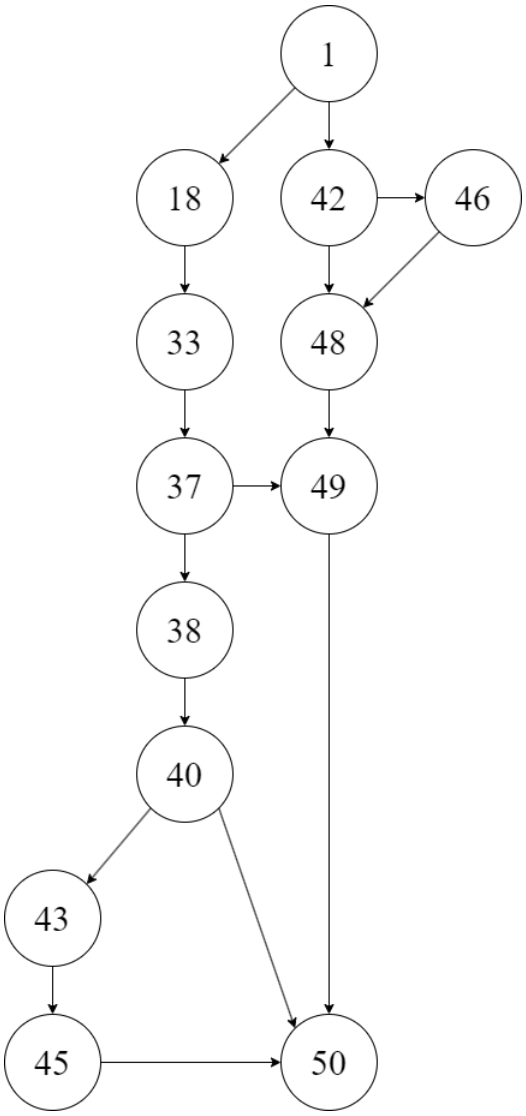


Figura 5.4: Grafo Parcial

A opção de salvar e sair permite ao usuário armazenar os dados gerados para análises futuras, garantindo que os resultados obtidos sejam preservados e possam ser utilizados em outras etapas do desenvolvimento de histórias ou jogos.

Como forma de ilustração, construiu-se o grafo da Figura 5.4 com as cinco primeiras soluções para o nível de habilidade catorze apresentadas na Figura 5.3b. Note que essa forma é apenas uma visualização parcial do grafo, construída manualmente, para facilitar a compreensão do arquivo de texto gerado.

Assim, a ferramenta desenvolvida demonstra ser eficaz na criação, análise e visualização de grafos, permitindo a avaliação detalhada de diferentes cenários de planejamento. A integração das bibliotecas NetworkX e PySwip, juntamente com a implementação de métodos específicos para garantia de conectividade e análise de soluções, proporciona uma ferramenta robusta de au-

xílio a desenvolvedores de histórias, sejam elas voltadas para jogos ou não.

6
Conclusão e trabalhos futuros

Como apresentado anteriormente no Capitulo 2, embora esteja presente em diversos trabalhos, a estrutura de grafos que representa as histórias não é colocada a disposição dos autores. São utilizadas para armazenar os eventos, avaliar a coesão da história ou prever qual evento deve ser concatenado ao atual. No entanto, o usuário não recebe a estrutura como um produto final, como uma ferramento de apoio para autoria de seus próprios textos.

Assim, a ferramenta desenvolvida atende de forma eficaz às necessidades de autores que buscam apoio na criação de narrativas interativas. O software proporciona uma interface intuitiva e robusta para a geração e análise de grafos narrativos, permitindo que autores explorem diferentes cenários de planejamento com facilidade.

Os resultados obtidos demonstraram que a ferramenta é capaz de gerar uma estrutura de grafos detalhada e organizada, com nós e conexões claramente definidos. A visualização gráfica dos grafos facilita a compreensão das conexões e dos caminhos narrativos disponíveis, enquanto os arquivos de texto gerados fornecem uma documentação completa da estrutura e das soluções possíveis.

Em suma, a ferramenta desenvolvida representa uma contribuição para o campo da narrativa interativa, oferecendo uma solução prática e eficiente para a criação e análise de grafos narrativos. A possibilidade de explorar diferentes caminhos e conexões narrativas amplia as opções criativas dos autores, proporcionando suporte para o desenvolvimento de histórias mais complexas e envolventes. Com essa ferramenta, os autores têm à disposição um recurso para facilitar o processo de autoria e enriquecer a experiência narrativa, tanto em jogos quanto em outras formas de narrativa interativa.

Por outro lado, há oportunidade para melhoria e incremento do presente protótipo. Possíveis adições que contribuiriam para robustez da ferramenta são o desenvolvimento de uma interface fora da linha de comando, a possibilidade de atribuir, armazenar e editar textos em um nó, deleção de nós e arestas, geração automática de textos para nós escolhidos pelo autor, destaque visual de caminhos selecionados e capacidade de receber como *input* um grafo parcialmente definido e editá-lo dentro da ferramenta.

7
Referências bibliográficas

AMMANABROLU, P. et al. Automated storytelling via causal, commonsense plot ordering. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 35, n. 7, p. 5859–5867, May 2021. Disponível em: <https://ojs.aaai.org/index.php/AAAI/article/view/16733>.

CAVAZZA, M.; CHARLES, F.; MEAD, S. J. Character-based interactive storytelling. **IEEE Intelligent Systems, vol. 17, no. 4, pp. 17-24**, 2002. Disponível em: <https://ieeexplore.ieee.org/document/1024747>.

CHANG, N.; NICHOLS, J. Introduction to graph theory. 2006. Disponível em: <https://mathcs.pugetsound.edu/~bryans/Current/Journal_Spring_2002/presentation_nchang_Jnichols_2002.pdf>.

de Lima, E. S.; FEIJó, B.; FURTADO, A. L. Managing the plot structure of character-based interactive narratives in games. **Entertainment Computing**, v. 47, p. 100590, 2023. ISSN 1875-9521. Disponível em: <https://www.sciencedirect.com/science/article/pii/S1875952123000459>.

DOMINGOS, A. A. Storytelling: Evolução, novas tecnologias e mídia. 2009. Disponível em: <http://www.intercom.org.br/papers/nacionais/2009/resumos/R4-2427-1.pdf>.

EADES, P.; XUEMIN, L. How to draw a directed graph. In: **1989 IEEE Workshop on Visual Languages**. Los Alamitos, CA, USA: IEEE Computer Society, 1989. p. 13,14,15,16,17. Disponível em: <https://doi.ieeecomputersociety.org/10.1109/WVL.1989.77035>.

FURTADO, A.; CASANOVA, M. Nice: A cooperative environment for the use of information systems. In: . [S.l.: s.n.], 1990.

GHALLAB, M.; NAU, D.; TRAVERSO, P. **Automated Planning: Theory and Practice**. [S.l.]: Morgan Kaufmann, 2004.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using networkx. In: VAROQUAUX, G.; VAUGHT, T.; MILLMAN, J. (Ed.). **Proceedings of the 7th Python in Science Conference**. Pasadena, CA USA: [s.n.], 2008. p. 11 – 15.

HUNTER, J. D. Matplotlib: A 2d graphics environment. **Computing in Science & Engineering**, IEEE COMPUTER SOC, v. 9, n. 3, p. 90–95, 2007.

LIMA, E. E. S. D. et al. A note on process modelling: Combining situation calculus and petri nets. **MONOGRAFIAS EM CIÊNCIA DA COMPUTAÇÃO**, Faculdades Catolicas, fev. 2018. ISSN 0103-9741. Disponível em: <http://dx.doi.org/10.17771/PUCRio.DImcc.59758>.

LIMA, E. S. D. et al. Chatgeppetto - an ai-powered storyteller. In: **Proceedings of the 22nd Brazilian Symposium on Games and Digital Entertainment**. New York, NY, USA: Association for Computing Machinery, 2024. (SBGames '23), p. 28–37. ISBN 9798400716270. Disponível em: <https://doi.org/10.1145/3631085.3631302>.

LIMA, E. S. de; FEIJó, B.; FURTADO, A. L. Adaptive branching quests based on automated planning and story arcs. In: **2021 20th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. [S.l.: s.n.], 2021. p. 9–18.

LIMA, E. Soares de et al. Personality and preference modeling for adaptive storytelling. In: **2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. [S.l.: s.n.], 2018. p. 187–18709.

RIEDL, M. O.; BULITKO, V. Interactive narrative: An intelligent systems approach. **AI Magazine**, v. 34, n. 1, p. 67–77, 2013. Disponível em: <https://onlinelibrary.wiley.com/doi/abs/10.1609/aimag.v34i1.2449>.

RUSSEL, S.; NORVIG, P. **Artificial Intelligence: A Modern Approach**. [S.l.]: Pearson Education, 2021.

TANG, C. et al. **NGEP: A Graph-based Event Planning Framework for Story Generation**. 2022.

TEKOL, Y.; CONTRIBUTORS. **PySwip v0.2.10**. 2020. Disponível em: <https://github.com/yuce/pyswip>.

THUE, D. et al. Interactive storytelling: A player modelling approach. **Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment**, v. 3, n. 1, p. 43–48, Sep. 2021. Disponível em: <https://ojs.aaai.org/index.php/AIIDE/article/view/18780>.

WARREN, D. H. D. **WARPLAN:A System for Generating Plans**. [S.l.], 1974.

YAO, L. et al. Plan-and-write: Towards better automatic storytelling. **Proceedings of the AAAI Conference on Artificial Intelligence**, v. 33, n. 01, p. 7378–7385, Jul. 2019. Disponível em: <https://ojs.aaai.org/index.php/AAAI/article/view/4726>.