Gabriel Brandão de Carvalho
Kevin Taiyo Onishi

Multi-Hop Question Answering with Knowledge Graphs

São Paulo, SP

2024

Gabriel Brandão de Carvalho
Kevin Taiyo Onishi

Multi-Hop Question Answering with Knowledge Graphs

Trabalho de conclusão de curso apresentado ao Departamento de Engenharia de Computação e Sistemas Digitais da Escola Politécnica da Universidade de São Paulo para obtenção do Título de Engenheiro.

Universidade de São Paulo – USP

Escola Politécnica

Departamento de Engenharia de Computação e Sistemas Digitais (PCS)

Orientador: Anarosa Alves Franco Brandão

São Paulo, SP

2024

Resumo

Os grafos de conhecimento são amplamente utilizados em diversos contextos, como question answering e sistemas de recomendação, devido à sua capacidade de representar e revelar relações entre conceitos. Este trabalho tem como objetivo desenvolver um algoritmo para responder perguntas utilizando grafos de conhecimento, explorando saltos baseados em contexto. Para atingir esse propósito, iniciamos descrevendo o modelo empregado para identificar informações contextuais, o BERT, e os quatro estágios que compõem nossa abordagem: a revisão da literatura sobre busca em grafos de conhecimento, a implementação de uma adaptação do algoritmo MultiHop KG (YU; HE; GLASS, 2021), a realização de uma bateria de testes no algoritmo e, por fim, a integração do algoritmo em uma interface interativa. O MultiHop KG (YU; HE; GLASS, 2021) destaca-se de outros métodos de busca em grafos por ser desenvolvido com base em documentos não estruturados, além de superar, com uma margem significativa, a precisão de outros métodos aplicados a question answering. Assim, o projeto teve início fundamentado nesse algoritmo, e, ao longo do desenvolvimento, investigamos diferentes estratégias para realizar buscas no grafo, culminando na definição do método mais adequado ao nosso contexto. Posteriormente, conduzimos uma série de testes exaustivos para determinar o modelo mais eficaz de geração de embeddings de palavras, bem como os parâmetros ideais para o sistema. Finalmente, desenvolvemos uma interface interativa projetada para facilitar o uso da ferramenta de question answering pelos usuários, permitindo visualização do percorrimento do grafo, assegurando uma experiência prática e acessível.

Palavras-chave: Grafos de conhecimento, Question Answering, Tuplas, Documentos não-estruturados, BERT, Similaridade de Cosseno, PLN, Embbedings.

Abstract

Knowledge graphs are widely used in various contexts, such as question answering and recommendation systems, due to their ability to represent and reveal relationships between concepts. This study aims to develop an algorithm to answer questions using a knowledge graph, leveraging context-based hops. To achieve this, we begin by describing the model employed to identify contextual information, BERT, and the four stages of our approach: a literature review on search in knowledge graphs, the implementation of an adaptation of the Multihop KG algorithm (YU; HE; GLASS, 2021), the execution of a comprehensive set of tests, and, finally, the integration of the algorithm into an interactive interface. Multihop KG (YU; HE; GLASS, 2021) stands out from other graph search methods as it is designed based on unstructured documents and significantly surpasses the accuracy of other approaches in knowledge graph search for question answering. Thus, the project began with a foundation in this algorithm, and during its development, we explored different strategies for performing searches in the graph, culminating in the identification of the most suitable method for our context. Subsequently, we conducted an exhaustive series of tests to determine the most effective word embedding generation model and the optimal parameters for the system. Finally, we developed an interactive interface designed to facilitate user interaction with the question-answering tool, allowing the user to vizualyze the paths traversed, ensuring a practical and accessible experience.

Keywords: Knowledge Graphs, Question Answering, Tuples, Unstructured Documents, BERT, Cossine Similarity, NLP, Embbedings.

Lista de ilustrações

Lista de tabelas

Lista de abreviaturas e siglas

AI	Artificial Inteligence
BERT	Bidirectional Encoder Representations from Transformers
BLAB	Blue Amazon Brain
KG	Knowledge Graph
LLM	Large Language Model
LM	Language Model
MLM	Masked Language Modeling
NLP	Natural Language Processing
NSP	Next Sentence Prediction
NN	Neural Network
QA	Question Answering
ZEE	Zona Econômica Exclusiva

Sumário

1 Introdução

1.1 Motivação

Com o avanço da inteligência artificial (IA) e das técnicas de processamento de linguagem natural (PLN), novos horizontes têm se aberto para a exploração e disseminação de conhecimento em áreas estratégicas e de alta relevância. A aplicação de IA em PLN permite a análise, compreensão e geração de linguagem humana, utilizando técnicas avançadas, como redes neurais profundas (NNs). Isso viabiliza a construção de agentes conversacionais sofisticados e a extração de informações a partir de grandes volumes de texto, ferramentas fundamentais em projetos de grande impacto, como o Blue Amazon Brain (BLAB) (FRANCO, 2022), uma iniciativa do Centro de Inteligência Artificial (C4AI), que visa criar um agente conversacional no domínio Amazônia Azul, com o objetivo de auxiliar na educação básica e em pesquisas sobre o tema.

A Amazônia Azul (WIKIPEDIA, 2024) é um exemplo claro de como a tecnologia pode transformar a gestão do conhecimento em regiões de importância estratégica. A Amazônia Azul abrangem as águas interiores, mar territorial, zona contígua, zona econômica exclusiva (ZEE) e águas sobrejacentes à plataforma continental (PC). Este território é rico em biodiversidade e recursos como petróleo, gás natural, minerais e uma infinidade de espécies marinhas. O termo foi cunhado em 2004 pelo almirante Álvaro Alberto, destacando a importância dessa região para o país, semelhante à da Amazônia Verde. Apesar de sua importância, informações organizadas e acessíveis sobre a Amazônia Azul ainda são escassas, dificultando a conscientização do público geral e o avanço de pesquisas mais especializadas.

Dado que o projeto está sendo realizado dentro do contexto do BLAB, temos que o projeto propõe a criação de um algoritmo capaz de percorrer um grafo de conhecimento sobre a Amazônia Azul e responder perguntas multicontextuais. Este grafo atua como uma estrutura que organiza informações de maneira hierárquica e interconectada, facilitando a resolução de perguntas complexas e permitindo conexões entre dados dispersos. Por exemplo, um sistema assim pode responder a perguntas como: “Quais os impactos ambientais da exploração de petróleo na biodiversidade da Amazônia Azul?” ou “Como os manguezais contribuem para a preservação da região?”.

Graças aos avanços em question answering (QA) com salto de contexto e na construção de grafos de conhecimento, o projeto busca não apenas consolidar informações sobre a Amazônia Azul, mas também visa, usando a combinação de PLN com grafos de conhecimento, oferecer uma base de dados estruturada e acessível para diferentes aplicações

sobre a Amazônia Azul, desde a educação até a pesquisa científica, ajudando a preencher lacunas críticas de informação.

Além de contribuir para o avanço do BLAB, nosso projeto reforça a importância da IA na construção de ferramentas que impactam diretamente o entendimento e a gestão de patrimônios estratégicos, como a Amazônia Azul. Essa abordagem ressalta o papel da tecnologia na ampliação do acesso ao conhecimento, na preservação ambiental e no desenvolvimento econômico sustentável do Brasil.

Figura 1 – Infográfico da Amazônia Azul retirado de um artigo da Universidade Federal de Tocantins

1.2 Justificativa

Este trabalho propõe a criação e o aprimoramento de técnicas para percorrer grafos de conhecimento, com o objetivo de realizar raciocínios em múltiplas etapas e chegar a conclusões a partir de um ponto de partida específico. Ao desenvolver uma inteligência capaz de responder perguntas que envolvem múltiplos contextos, abre-se a possibilidade de gerar impactos significativos em diversas áreas, como educação, economia, e a experiência do usuário. Essas melhorias têm o potencial de transformar desde sistemas de atendimento ao cliente, tornando-os mais eficientes e personalizados, até a criação de ferramentas educacionais interativas e sistemas médicos que realizem interpretações mais profundas de dados complexos.

A utilização de grafos de conhecimento é um aspecto central deste trabalho, dada a capacidade desses grafos de representar informações de forma estruturada e organizada. Essa estruturação permite não apenas a análise detalhada de dados, mas também a visualização de conexões e a identificação de problemas que poderiam passar despercebidos em formatos de dados menos organizados. Além disso, o uso de grafos de conhecimento potencializa a criação de sistemas que oferecem interpretações complexas, conectando informações aparentemente distantes e facilitando a solução de problemas multifacetados.

O foco em algoritmos capazes de realizar saltos de contexto em grafos de conhecimento traz uma contribuição especialmente relevante. Essa capacidade é essencial para responder perguntas de alta complexidade, nas quais a resposta requer a conexão de informações oriundas de diversas fontes ou níveis de conhecimento. Por exemplo, em ferramentas de busca avançadas ou *chatbots*, essa abordagem pode transformar a experiência do usuário, entregando respostas mais completas e contextualizadas.

Ao considerar o crescente uso de grafos de conhecimento em aplicações como mecanismos de busca, sistemas de recomendação e agentes conversacionais, o desenvolvimento deste trabalho ganha ainda mais relevância. Ele demonstra como a combinação de grafos de conhecimento com técnicas avançadas de IA pode expandir as fronteiras do que sistemas inteligentes são capazes de realizar, melhorando a interação entre humanos e máquinas em múltiplos domínios. Dessa forma, o projeto não só contribui para o avanço acadêmico, mas também tem potencial de impacto direto em aplicações práticas amplamente utilizadas na sociedade atual.

1.3 Objetivos

Este trabalho de conclusão de curso tem como objetivo investigar, desenvolver e avaliar algoritmos de QA com salto de contexto, aplicando-os especificamente à criação de um agente conversacional no domínio da Amazônia Azul. Esse projeto visa integrar técnicas de IA e PLN para abordar questões que demandam raciocínio em múltiplas etapas, permitindo que o sistema responda a perguntas complexas de maneira precisa e relevante.

A pesquisa será estruturada em três etapas principais. Primeiramente, será conduzida uma revisão abrangente da literatura atual, com o intuito de identificar os conceitos fundamentais e explorar os avanços mais recentes em algoritmos de QA com salto de contexto. Essa revisão abrangerá técnicas como *transformers*, modelos de linguagem pré-treinados (e.g., *Bidirectional Encoder Representations from Transformers* (BERT)), e métodos de raciocínio em grafos, buscando entender os desafios e as oportunidades no desenvolvimento dessas soluções.

Na etapa seguinte, serão analisadas e comparadas diferentes abordagens para o desenvolvimento de algoritmos capazes de navegar por grafos de conhecimento e estabelecer conexões entre múltiplos contextos. O foco estará em técnicas que maximizem a precisão e a eficiência do raciocínio, garantindo que o sistema consiga lidar com perguntas que exijam um entendimento profundo das relações entre os nós do grafo.

Finalmente, os algoritmos mais promissores serão implementados e testados em grafos de conhecimento previamente definidos, relacionados à Amazônia Azul. Esses testes incluirão a avaliação de desempenho utilizando métricas como precisão, recuperação de informações e eficiência computacional, além de cenários práticos para validar a aplicabilidade dos algoritmos. Essa abordagem permitirá identificar não apenas a eficácia técnica das soluções, mas também sua capacidade de atender às necessidades de usuários finais, como educadores e pesquisadores e, finalmente, integrar dentro de um sistema com uma interface onde tais usuários poderão usufruir do sistema construído.

Com a conclusão deste trabalho, espera-se contribuir significativamente para o avanço das aplicações de QA em domínios complexos e, simultaneamente, apoiar iniciativas como o BLAB, que visam conscientizar e disseminar o conhecimento sobre a importância estratégica e ambiental da Amazônia Azul.

1.4 Organização do Trabalho

A estrutura desta monografia foi cuidadosamente planejada para apresentar os fundamentos, o desenvolvimento e as conclusões do projeto de forma clara e organizada. Cada capítulo aborda aspectos específicos do trabalho, guiando o leitor desde os conceitos básicos até os resultados obtidos e as perspectivas futuras. A seguir, apresenta-se uma visão geral do conteúdo de cada capítulo:

No Capítulo 2, são introduzidos os conceitos fundamentais utilizados na construção do sistema. Este capítulo tem como objetivo explicar, de maneira clara e objetiva, os principais termos técnicos que embasam o trabalho, como entidade, grafo, BERT, *Embbeding* e QA. Essas definições fornecem ao leitor a base teórica necessária para compreender as etapas subsequentes do projeto.

No Capítulo 3, é descrito o planejamento metodológico do projeto. Este capítulo detalha como as atividades foram estruturadas para alcançar resultados satisfatórios, apresentando as etapas organizadas em um cronograma lógico. O objetivo é fornecer uma visão clara de como o trabalho foi conduzido, destacando os métodos e estratégias utilizadas em cada fase. Além disso, o capítulo também detalha de materiais e tecnologias utilizados para alcançar tais resultados.

O Capítulo 4 é dedicado à especificação do sistema, detalhando as funcionalidades e características que orientaram seu desenvolvimento. Nesse capítulo, são apresentados os requisitos funcionais e não funcionais que guiaram a construção do sistema, assim como sua arquitetura, garantindo que ele atenda às demandas do domínio em que será aplicado.

O Capítulo 5 foca no desenvolvimento técnico do sistema, apresentando os códigos e algoritmos responsáveis por navegar no grafo de conhecimento e realizar o raciocínio *multi-hop*. São explicadas as abordagens adotadas, bem como os métodos de avaliação das respostas geradas. Adicionalmente, o capítulo documenta as etapas de testes realizadas em cada fase do algoritmo, incluindo comparações entre diferentes abordagens e os resultados obtidos.

Por fim, o Capítulo 6 apresenta as considerações finais do trabalho. Este capítulo faz um balanço dos resultados atingidos e das metas que não foram alcançadas, fornecendo justificativas para os desafios enfrentados. Também são destacadas as contribuições do projeto, com ênfase no que foi desenvolvido pela equipe. Por último, são exploradas possibilidades de continuidade do trabalho, sugerindo o uso de Redes Neurais e Large Language Models (LLMs) como uma extensão futura para aprimorar o sistema.

Essa organização permite que o leitor acompanhe, de forma sequencial e lógica, todos os aspectos do projeto, desde sua concepção até suas implicações práticas e potenciais futuras.

2 Aspectos Conceituais

Para compreender a construção e o funcionamento do sistema desenvolvido neste trabalho, é essencial explorar os conceitos fundamentais que sustentam sua base teórica e prática. Este capítulo tem como objetivo apresentar os termos e ideias centrais que servem como alicerce para as discussões e implementações descritas nos capítulos subsequentes.

Inicialmente, serão introduzidos os conceitos de entidade e grafo, fundamentais para a estruturação de informações em forma de relações organizadas e navegáveis. Em sequência, será discutido o conceito de grafos de conhecimento, destacando sua importância na representação de dados de forma estruturada e na facilitação de análises complexas, como o raciocínio de salto de contexto.

O capítulo também abordará técnicas modernas de representação de dados, como os *embeddings*, que permitem mapear palavras ou conceitos em espaços vetoriais, preservando relações semânticas. Em especial, será introduzido o modelo BERT, amplamente utilizado em tarefas de PLN por sua capacidade de capturar contextos em múltiplas direções e como será aplicado essas técnicas para realizar a semelhança entre sentenças.

Além disso, serão apresentados os principais algoritmos de navegação em grafos, que desempenham um papel crucial no raciocínio sobre relações complexas e na descoberta de caminhos relevantes para responder a perguntas em múltiplos contextos. Esses algoritmos são ferramentas essenciais para explorar as conexões presentes nos grafos de conhecimento de maneira eficiente e precisa, funcionando em conjunto com avaliadores de semelhança de frases.

Por fim, será feita a conexão entre esses conceitos e o objetivo do projeto, destacando como cada um deles contribui para o desenvolvimento de um sistema capaz de responder a perguntas sobre a Amazônia Azul, combinando técnicas de IA e estruturas de conhecimento organizadas.

2.1 Entidade

Uma entidade é um conceito fundamental no campo da representação de conhecimento e da IA, sendo definida como qualquer objeto ou coisa que possui uma existência independente e pode ser identificado de forma única dentro de um contexto. Em outras palavras, uma entidade pode ser uma pessoa, lugar, objeto, evento, ou até mesmo uma ideia abstrata que tenha relevância para o domínio de interesse.

No contexto de grafos de conhecimento, uma entidade é representada como um nó (ou vértice) dentro do grafo. As entidades estão conectadas por arestas, que representam

as relações entre elas. Por exemplo, em um grafo de conhecimento sobre a Amazônia Azul, entidades podem ser conceitos como “Amazônia Azul”, “bioma marinho”, “biodiversidade”, “recursos naturais”, “Brasil”, entre outros.

Essas entidades são associadas a atributos ou características que descrevem suas propriedades. Por exemplo, a entidade “Brasil” pode ter atributos como “capital” (Brasília), “área” (8,5 milhões de km²), e “população” (cerca de 210 milhões de habitantes).

2.2 Relacionamento

Um relacionamento (ou relação) é a conexão entre duas ou mais entidades que descreve como elas estão associadas ou interagem dentro de um determinado contexto. Em um grafo de conhecimento, o relacionamento é representado como uma aresta ou link entre dois nós (entidades), refletindo o tipo de associação ou vínculo que existe entre essas entidades.

Por exemplo, em um grafo de conhecimento sobre a Amazônia Azul, algumas entidades podem ser relacionadas por vários tipos de relacionamentos, como:

- Localização: A entidade “Brasil” pode estar relacionada à entidade “Amazônia Azul” por um relacionamento de “está localizado em”.
- Recursos Naturais: A entidade “Amazônia Azul” pode estar relacionada a “petróleo” e “gás natural” através de um relacionamento como “contém recursos”.
- Ecossistema: A entidade “Amazônia Azul” pode estar relacionada à “biodiversidade marinha” com um relacionamento de “abriga”.
- Temporalidade: Uma entidade “descoberta do pré-sal” pode estar relacionada à entidade “Amazônia Azul” por um relacionamento de “ocorre em”.

Além disso, os relacionamentos podem ter diferentes características, como a direção (um relacionamento pode ser unidirecional ou bidirecional), o tipo (por exemplo, “é parte de”, “pertence a”, “causa”, “depende de”), e o peso (que pode indicar a força ou importância da relação).

Esses relacionamentos são essenciais para dar sentido às entidades dentro de um grafo de conhecimento. Ao organizar as entidades e suas interações de forma estruturada, os relacionamentos permitem que um sistema de IA entenda e raciocine sobre como diferentes conceitos se interconectam e como podem ser usados para responder perguntas complexas ou realizar inferências.

2.3 Grafos de Conhecimento

Um grafo de conhecimento (KG) é uma estrutura de dados que organiza informações de forma gráfica, representando entidades como nós (ou vértices) e os relacionamentos entre elas como arestas (ou links). Cada nó no grafo representa uma entidade, que pode ser um objeto, pessoa, conceito ou local relevante para um determinado domínio, enquanto as arestas descrevem as relações entre essas entidades. Além disso, as entidades e os relacionamentos podem ter atributos, que são propriedades que detalham características das entidades ou das relações, como a data de criação de um objeto ou a intensidade de uma conexão entre duas entidades.

Esses grafos de conhecimento são usados principalmente em sistemas de QA, onde o algoritmo percorre o grafo para encontrar respostas a perguntas que envolvem múltiplas etapas de raciocínio, conectando diversas entidades. Além disso, são amplamente aplicados em sistemas de recomendação, onde a análise das conexões entre as entidades pode sugerir novos itens ou informações ao usuário. Eles também são essenciais em tarefas de análise de dados, ajudando a identificar padrões e relações que podem não ser imediatamente visíveis.

A principal vantagem dos grafos de conhecimento é sua capacidade de representar informações de maneira estruturada e interconectada, o que facilita a análise, a identificação de problemas e a visualização de conexões complexas entre diferentes entidades, como pode ser visto em (SINGHAL, 2012). Em sistemas de IA e PLN, como os utilizados em *chatbots* e assistentes virtuais, o grafo de conhecimento desempenha um papel fundamental ao permitir que o sistema compreenda e responda a perguntas de múltiplos contextos e camadas de raciocínio.

Como exemplo, temos o grafo de conhecimento temos alguns como citados no artigo (SULLIVAN, 2020) ou também na figura 2, que relaciona seres vivos.

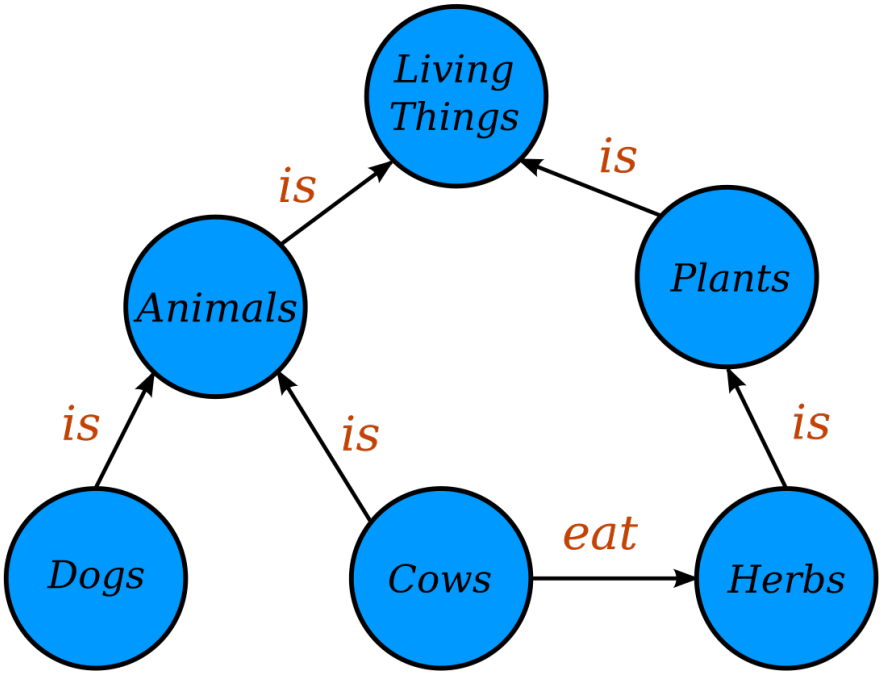

Figura 2 – Grafo de conhecimento exemplificado retirado da Wikipedia

Neste grafo de exemplo, podemos ver entidades como “Dogs”, “Cows”, “Plants”, e assim por diante. Junto disso, podemos ver também as relações entre as entidades como *Cows “eat” Herbs*, sendo comer(*eat*) a relação entre as entidades “Cows” e “Herbs”.

2.4 Question Answering

Question Answering (QA) é uma área do PLN que se concentra em desenvolver sistemas capazes de responder a perguntas formuladas em linguagem natural. O objetivo do QA é gerar respostas precisas e relevantes a partir de fontes de dados, como textos, documentos, ou grafos de conhecimento, onde o sistema precisa identificar a informação correta que responde à consulta do usuário.

O processo de QA envolve várias etapas, incluindo:

1. Compreensão da Pergunta: O sistema deve analisar e entender a pergunta feita pelo usuário. Isso envolve a identificação de entidades, relações e intenções na consulta. Perguntas podem ser simples (ex: “Qual é a capital do Brasil?”) ou complexas, exigindo múltiplos saltos de contexto para chegar à resposta.

2. Busca de Informação: Após entender a pergunta, o sistema busca a informação relevante nas fontes de dados disponíveis, como textos ou grafos de conhecimento. Nesse momento, técnicas como recuperação de informações e busca por padrões são aplicadas.

3. Raciocínio e Inferência: Para perguntas mais complexas, o sistema precisa realizar um raciocínio sobre as informações encontradas. Isso pode envolver a navegação por múltiplos pontos de dados, como em QA com salto de contexto, onde a resposta pode depender de informações dispersas em diferentes partes de um grafo de conhecimento ou de um conjunto de textos.

4. Geração da Resposta: Após identificar a informação relevante, o sistema gera uma resposta adequada, podendo ser uma resposta direta, uma explicação mais detalhada ou até mesmo uma recomendação.

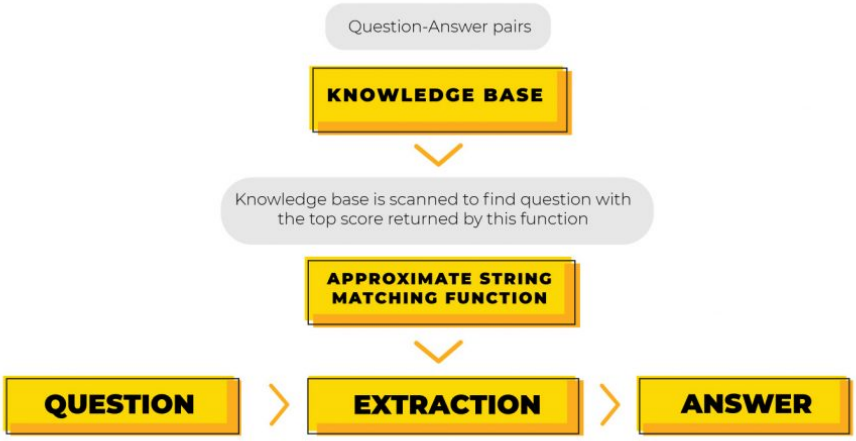

Figura 3 – Funcionamento genérico de um sistema de QA retirado de um blog de Intersog

Existem duas abordagens principais em QA:

- QA Extrativa: O sistema responde extraindo diretamente a resposta de um texto ou documento existente. Exemplos incluem sistemas que pegam trechos específicos de uma página da web ou de um artigo para responder à pergunta.

- QA Abstrativa: O sistema gera uma resposta original baseada em sua compreensão do contexto, podendo reescrever ou sintetizar a resposta em suas próprias palavras. Essa abordagem é mais desafiadora, pois exige um entendimento mais profundo do conteúdo.

Para nosso projeto, planejamos gerar uma QA Extrativa apenas, buscando e extraindo informação e dados a partir do grafo de conhecimento e gerando uma resposta direta sem qualquer tipo de compreensão ou interpretação por parte do algoritmo.

2.5 Embedding

Embedding é uma técnica fundamental no campo de aprendizado de máquina, especialmente em tarefas que envolvem linguagem natural, imagens e grafos. Ele refere-se à transformação de dados complexos, como palavras, frases ou entidades de um grafo, em representações numéricas de baixa dimensão chamadas vetores densos. Esses vetores são projetados para preservar as relações semânticas ou estruturais existentes nos dados originais, tornando-os mais manejáveis para modelos computacionais.

A ideia central de um *embedding* é representar informações de forma compacta e significativa. Ao invés de trabalhar diretamente com palavras ou entidades como símbolos únicos (uma abordagem que geralmente ignora relações entre eles), os *embeddings* posicionam esses elementos em um espaço vetorial contínuo, onde a proximidade entre vetores reflete semelhanças contextuais ou semânticas. Por exemplo, no caso de palavras palavras com significados semelhantes, como "rei" e "rainha", terão representações numéricas (vetores) próximas no espaço vetorial e palavras sem relação, como "cachorro" e "prédio", estarão mais distantes. Além disso, os *embeddings* são capazes de capturar relações semânticas lineares. Um exemplo clássico é:

$$\textbf{rei} - \textbf{homem} + \textbf{mulher} = \textbf{queen}$$

Esse tipo de propriedade é essencial para tarefas de raciocínio semântico, como aquelas presentes em sistemas de QA ou em grafos de conhecimento.

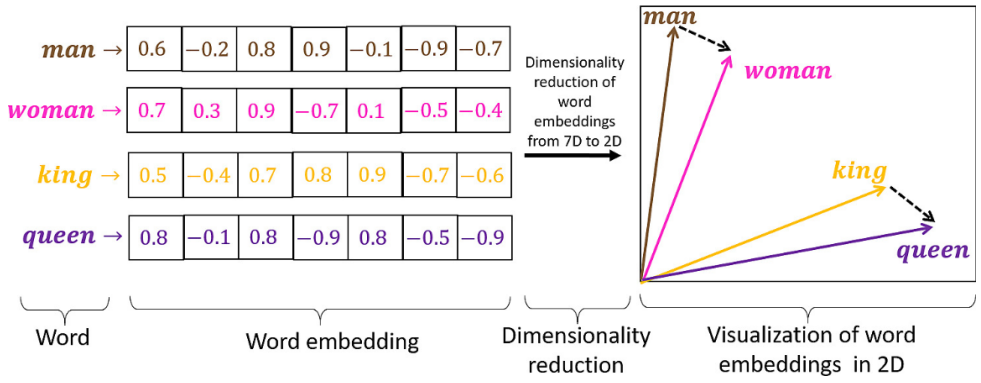

Figura 4 – Processo de gerar *embedding* de palavras retirado de um artigo sobre *embedding*

Os *embeddings* desempenham um papel central em muitas áreas de aprendizado de máquina e inteligência artificial. Alguns exemplos incluem:

- *Processamento de Linguagem Natural* (PLN):
  - *Word Embeddings*: Representam palavras em contextos semânticos (e.g., *Word2Vec, GloVe, FastText*).

  - *Contextual Embeddings*: Modelos como BERT e GPT geram representações específicas para cada palavra, considerando o contexto em que aparecem.
  - Tarefas de QA: Permitem comparar perguntas e respostas com base em suas similaridades semânticas.

- Grafos de Conhecimento: *Embeddings* de entidades e relações ajudam a navegar grafos complexos, extraindo informações relevantes para raciocínios em múltiplos contextos. Modelos como TransE ou RotatE aprendem a representar entidades e relações para facilitar a inferência no grafo.

- Busca Semântica e Sistemas de Recomendação: *Embeddings* melhoram a relevância de resultados ao analisar o significado subjacente das consultas e itens, em vez de apenas correspondências textuais exatas.

- Representação Multimodal: Transformam imagens, áudios ou vídeos em vetores que podem ser analisados em conjunto com texto ou outras modalidades.

2.6 BERT

2.6.1 Definição

BERT, acrônimo de *Bidirectional Encoder Representations from Transformers*, é um modelo de linguagem baseado em *transformers* desenvolvido pelo Google em 2018. Ele revolucionou o campo do PLN ao introduzir uma abordagem que considera o contexto de uma palavra não apenas à sua direita ou esquerda, mas em ambas as direções simultaneamente, oferecendo uma compreensão mais rica e precisa do texto.

Antes do BERT, muitos modelos utilizavam abordagens unidirecionais, analisando palavras apenas no contexto anterior ou posterior. Essa limitação tornava difícil capturar nuances em frases ambíguas ou altamente contextuais. O BERT superou essa barreira ao adotar um mecanismo bidirecional, permitindo interpretações mais sofisticadas e alinhadas à semântica do texto.

O modelo foi treinado em grandes corporações, como a Wikipédia e o BooksCorpus, utilizando tarefas como *Masked Language Modeling* (MLM) e *Next Sentence Prediction* (NSP). Essas tarefas permitiram ao BERT aprender representações linguísticas universais, que podem ser adaptadas a uma ampla gama de aplicações, como QA, classificação de textos e análise de sentimentos.

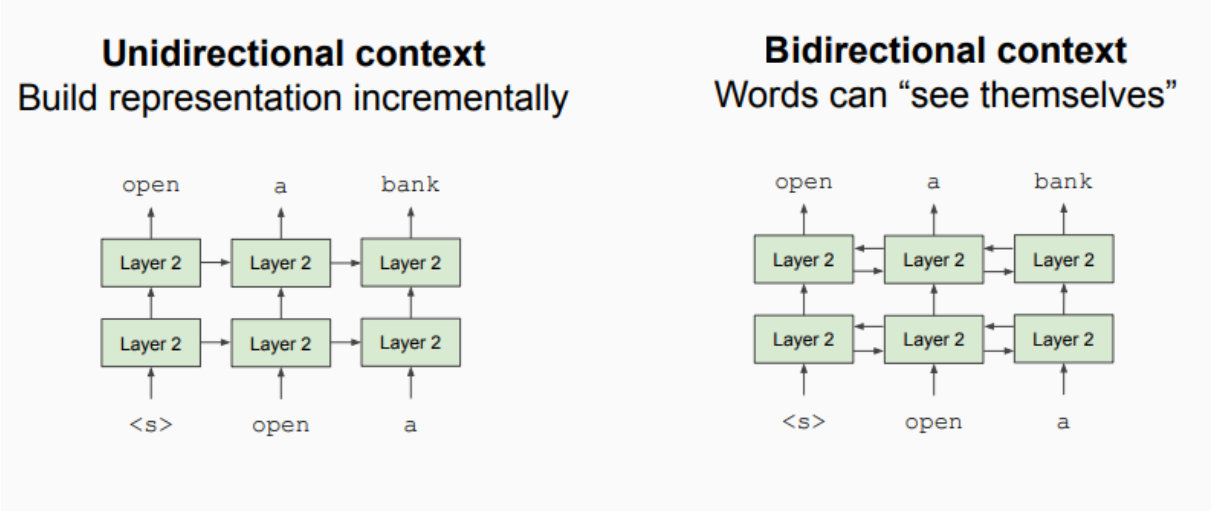

Figura 5 – Diferença do processamento do BERT em relação a outras LM retirado de uma apresentação da Princeton University

2.6.2 Justificativa

O uso do BERT neste projeto se justifica pelo fato deste ser capaz de capturar nuances contextuais em uma frase, especialmente aquelas que exigem raciocínio contextual complexo. Como o foco é navegar e raciocinar em grafos de conhecimento, o BERT se destaca pela sua capacidade de capturar relações semânticas e contextuais em textos variados, tornando-o ideal para lidar com perguntas que envolvem múltiplos contextos.

A capacidade bidirecional do BERT é particularmente útil ao interpretar perguntas complexas que exigem inferências em várias etapas. Por exemplo, em uma pergunta como "Qual é a função de um recife de coral na Amazônia Azul?", o modelo precisa compreender não apenas os termos isolados, mas também as relações entre eles, algo que o BERT faz com excelência.

Além disso, o BERT possui versões pré-treinadas que podem ser ajustadas (*fine-tuned*) para tarefas específicas. Essa característica reduz significativamente o tempo e os recursos necessários para desenvolver um modelo personalizado, permitindo concentrar os esforços na criação de grafos e otimização de algoritmos de navegação.

2.6.3 Funcionamento

Os *embeddings* gerados por um modelo BERT pré-treinado são representações numéricas que capturam as características contextuais de palavras ou frases em um texto. Esses *embeddings* são altamente contextuais, o que significa que a representação de uma palavra depende do texto ao seu redor, isso pode ser visto claramente no artigo (ALAMMAR, 2018). A seguir, detalhamos o processo de como um modelo pré-treinado do BERT realiza essa tarefa:

1. Tokenização

   Antes de processar o texto, o modelo BERT utiliza uma tokenização específica chamada *WordPiece Tokenizer*.

   O texto de entrada é dividido em partes menores chamadas tokens. Isso permite lidar com palavras raras ou desconhecidas, quebrando-as em subpalavras frequentes.

   Além disso, tokens especiais são adicionados: [CLS] (indica o início da sequência) e [SEP] (separa sentenças ou marca o final do texto).

   Por exemplo, a frase "O ecossistema marinho é vasto" pode ser tokenizada como:

Token de Início	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**Token Separador**
[CLS]	O	e	###cos	##sist	###ema	ma	###rin	###ho	é	vast	###o	[SEP]

Tabela 1 – Exemplo de tokenização BERT utilizando bert-base-cased

Cada token é transformado em um vetor inicial utilizando três componentes principais. Token *embeddings* que é a representação inicial de cada token, *segment embeddings* que diferencia sentenças em casos de múltiplas entradas, como no treinamento do modelo e, por fim, *positional embeddings*, no qual inclui informações sobre a posição de cada token na sequência, já que o BERT não possui ordem inerente por ser um modelo baseado em *transformers*. Esses vetores são combinados e somados para formar a entrada que será processada pelo BERT.

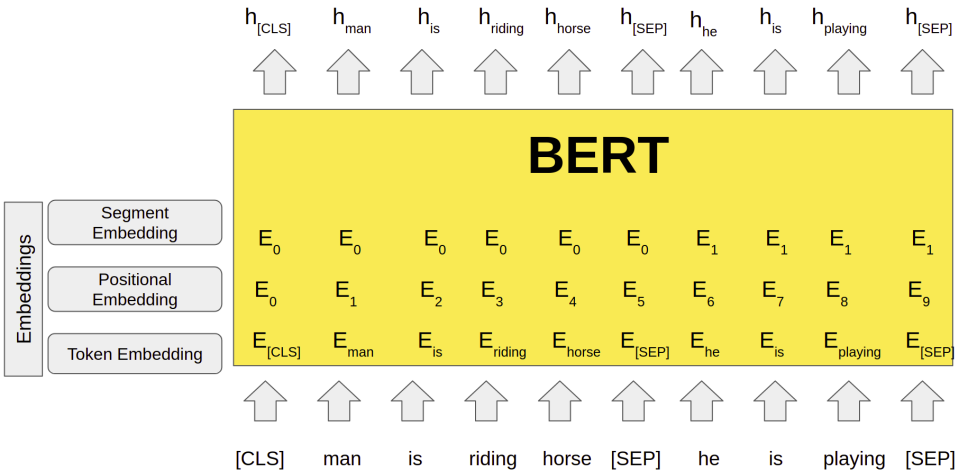

Figura 6 – Camadas iniciais de *embeddings* retirado de Analytics Vidhya

2. Mecanismo de atenção

   O núcleo do BERT é o mecanismo de atenção bidirecional. Aqui, cada token é analisado em relação a todos os outros na sequência, tanto antes quanto depois. Utilizando do mecanismo de *self-attention* permite que o modelo atribua pesos diferentes às palavras ao redor, dependendo de sua relevância. Por exemplo, na frase

"A navegação é essencial no ecossistema marinho", o modelo pode entender que "marinho" está relacionado mais fortemente a "ecossistema" do que a "navegação".

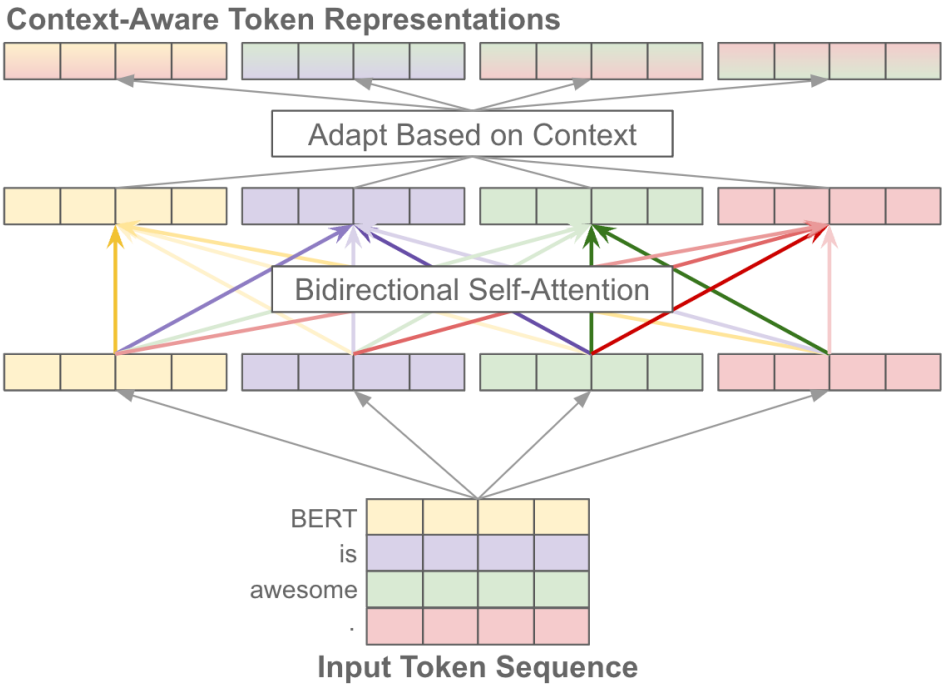

Figura 7 – Mecanismo de atenção bidirecional do BERT retirado de um artigo no Medium

3. Extração dos *embeddings*

   Após passar por várias camadas de atenção, o modelo gera representações vetoriais para cada token. Esses vetores podem ser utilizados para diferentes propósitos:

   - Palavras individuais: Os vetores dos tokens correspondentes representam as palavras no contexto.
   - Frases ou sentenças completas: O vetor associado ao token [CLS] é frequentemente usado como uma representação da sequência inteira.

   Por exemplo, na frase "Os recifes são importantes para a biodiversidade", o vetor do token [CLS] pode encapsular o significado geral dessa sentença, enquanto os vetores dos tokens "recifes" e "biodiversidade" trazem nuances contextuais específicas. Por fim, as representações geradas por BERT pré-treinado podem ser usadas diretamente como *embeddings* ou ajustadas para tarefas específicas, como QA, classificação de texto ou análise de similaridade semântica.

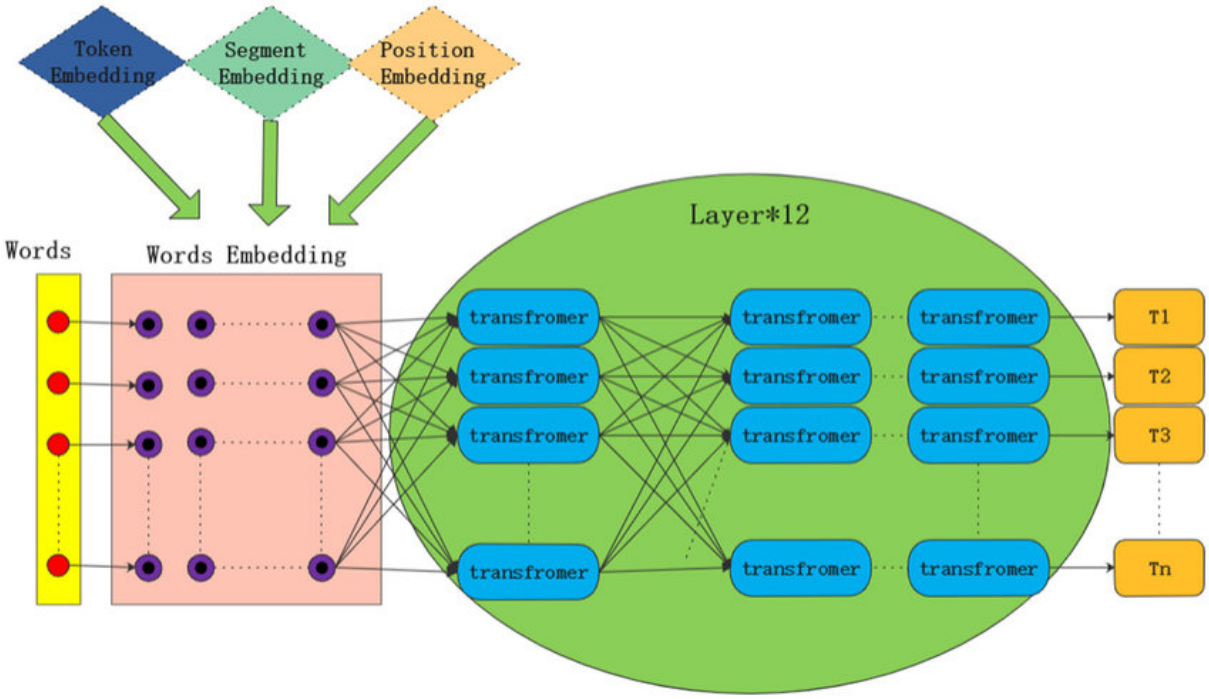

Figura 8 – Funcionamento geral do BERT retirado do artigo Span identification and technique classification of propaganda in news articles

3 Materiais e Métodos do trabalho

O capítulo de Materiais e Métodos apresenta o processo de desenvolvimento deste trabalho, dividido em diferentes fases que incluem a especificação de requisitos, projeto, implementação e testes. Cada uma dessas etapas foi essencial para garantir a clareza, a organização e a viabilidade do projeto, assegurando que os objetivos propostos fossem atingidos de maneira estruturada. Embora os detalhes específicos de cada fase sejam abordados nos capítulos subsequentes, este capítulo oferece uma visão geral das ferramentas, recursos e metodologias empregadas.

Além disso, este capítulo também contextualiza a escolha dos materiais e métodos com base em trabalhos relacionados previamente consultados. Estudos sobre modelos pré-treinados como BERT e suas variantes, além de implementações de sistemas baseados em grafos de conhecimento, influenciaram diretamente o desenvolvimento deste projeto. Dessa forma, os métodos aqui apresentados refletem a integração de soluções consolidadas com abordagens inovadoras, aplicadas ao domínio da Amazônia Azul.

3.1 Materiais

Nesta seção, apresentamos os materiais utilizados para o desenvolvimento do projeto, detalhando os recursos computacionais, bibliotecas, linguagens de programação e *datasets* empregados ao longo das etapas. A combinação desses materiais foi cuidadosamente selecionada para atender às demandas de processamento, desenvolvimento e análise, garantindo um ambiente eficiente e adequado à complexidade do projeto.

Inicialmente, abordamos as especificações do ambiente de desenvolvimento utilizado, destacando o Google Colab, uma plataforma que oferece suporte a hardware acelerado, como GPUs, essencial para o treinamento e a execução de modelos baseados em deep learning. Em seguida, exploramos os modelos pré-treinados da biblioteca Transformers, da Hugging Face, que desempenharam um papel fundamental na construção e aplicação do modelo BERT para gerar *embeddings* e realizar QA no contexto do grafo de conhecimento da Amazônia Azul.

Além disso, descrevemos as ferramentas e linguagens de programação que integraram os diferentes componentes do sistema. O Python foi utilizado no desenvolvimento do *backend*, responsável pelo processamento e execução das lógicas do grafo de conhecimento, enquanto o JavaScript, aliado a *frameworks* modernos como React, foi aplicado na construção de uma interface *front-end* interativa e intuitiva. Por fim, apresentamos o *dataset* sobre a Amazônia Azul, cuidadosamente estruturado para atender aos requisitos

do modelo e viabilizar uma análise precisa e contextualizada.

3.1.1 Hardware

O hardware utilizado, como dito anteriormente, foram unidades de processamento providos pelo Google Colab, especificamente a unidade de Python Backend, cuja as especificações são a seguir:

Processador	Intel(R) Xeon(R) CPU @ 2.20GHz
Memória RAM	13 GB
Disco	100 GB
Unidade de Processamento Gráfico	NVIDIA Tesla K80 12 GB

Tabela 2 – Tabela de especificações do Google Colab

3.1.2 Software

Os softwares utilizados para o desenvolvimento do *Backend* e do *Frontend* estão mostrados a seguir, junto das tecnologias e *frameworks* utilizados em cada um deles.

Python (*Backend* + algoritmo)	
Versão	3.10
***Frameworks*/Bibliotecas**	Transformers (4.46.2) Numpy (1.26.4) Pandas (2.2.2) SentenceTransformer (3.2.1) Scikit-Learn (1.5.2) Torch (2.5.1) Flask (3.0.3)

Tabela 3 – Tabela de especificações dos principais Softwares utilizados para o *Backend*

Javascript (*Frontend*)	
Versão	Node 22.11.0
***Frameworks*/Bibliotecas**	React (18.3.1) papaparse (5.4.1) react-pro-sidebar (1.1.0) material-ui (6.1.9) reagraph (4.20.1)

Tabela 4 – Tabela de especificações dos principais Softwares utilizados para o *Frontend*

3.1.3 Data

O *dataset* utilizado neste projeto é um grafo de conhecimento construído a partir de informações extraídas de um texto descritivo sobre as atividades e avanços tecnológicos da Petrobras no desenvolvimento *offshore*. Este texto, que inclui detalhes sobre iniciativas de pesquisa e desenvolvimento, parcerias estratégicas e projetos específicos, serviu como base para a criação inicial do grafo.

A partir do trecho selecionado, construímos uma estrutura inicial contendo 28 relacionamentos e 10 perguntas projetadas para avaliar a capacidade do modelo em realizar inferências contextuais e responder a questões com múltiplos saltos de contexto. As entidades principais incluem termos como Petrobras, FMC Technologies Inc., Marlim, e programas de pesquisa como Procap 3000 e Future Vision Procap, enquanto os relacionamentos mapeiam conexões relevantes, como "colabora com", "implementa", e "realiza".

Após observar resultados promissores no desempenho do modelo, decidimos expandir o grafo utilizando o texto completo de onde o trecho inicial foi extraído. Com isso, aumentamos significativamente a quantidade de entidades e relações representadas, o que proporcionou uma maior riqueza contextual ao grafo. Essa expansão permitiu não apenas cobrir mais informações, mas também explorar interconexões mais complexas entre os elementos, reforçando a capacidade do sistema em realizar raciocínios avançados e responder a perguntas mais desafiadoras.

O grafo resultante é um modelo estruturado e abrangente do domínio relacionado à Petrobras e ao desenvolvimento *offshore* na bacia de Campos. Ele reflete informações técnicas, como a instalação de sistemas de separação submarina em profundidades de 899 metros, além de dados estratégicos, como as parcerias com empresas como a FMC Technologies. Essa estruturação facilita o treinamento do modelo de QA, que pode explorar as relações no grafo para responder perguntas complexas de forma precisa e eficiente.

A escolha desse texto como base do *dataset* se justifica pela relevância do domínio no contexto da Amazônia Azul, destacando atividades econômicas, científicas e tecnológicas de alta importância. Além disso, a possibilidade de expandir o grafo dinamicamente com novos dados permite que o sistema se mantenha escalável e flexível enquanto podemos gerar um grafo de forma ideal sem relações redundantes ou incompletas, adaptando-se a informações adicionais ou mudanças no contexto.

3.2 Métodos

O capítulo de método descreve o processo metodológico adotado para desenvolver e validar o sistema de QA baseado em grafos de conhecimento. O trabalho foi estruturado em

quatro etapas principais, que vão desde estudos preliminares sobre conceitos fundamentais até a integração completa do sistema com uma interface simples de *chatbot* e visualização gráfica. Cada uma dessas etapas foi cuidadosamente planejada para assegurar o alcance dos objetivos do projeto, garantindo tanto a robustez técnica quanto a usabilidade do sistema desenvolvido.

Inicialmente, foi realizada uma extensa revisão bibliográfica sobre grafos de conhecimento e algoritmos relacionados, a fim de construir uma base teórica sólida para o desenvolvimento do sistema. Em seguida, foi implementado um algoritmo baseado no modelo AutoKG como principal referência, buscando extrair e estruturar conhecimento a partir de documentos não estruturados. Para avaliar a eficiência e a precisão do algoritmo, foram conduzidos testes rigorosos utilizando o grafo gerado. Por fim, foi desenvolvida a integração do sistema com um *frontend* e *backend* que permitiu a criação de um *chatbot* interativo e a visualização do percurso pelo grafo. A seguir, cada uma dessas etapas será detalhada.

3.2.1 Estudos de Bibliografia

Nesta etapa, foi realizada uma pesquisa aprofundada sobre os fundamentos de grafos de conhecimento, incluindo conceitos essenciais como entidades, relações e os algoritmos utilizados para navegar e consultar essas estruturas. A literatura revisada incluiu artigos acadêmicos, livros e documentos técnicos que abordam modelos amplamente utilizados em tarefas de QA baseadas em grafos, como o RDF, o SPARQL e abordagens de aprendizado de máquina.

Além disso, exploramos trabalhos recentes que aplicam inteligência artificial para a construção de grafos de conhecimento a partir de dados não estruturados. Esse levantamento foi essencial para identificar lacunas e desafios no estado da arte, que influenciaram diretamente as decisões no desenvolvimento do sistema. A análise crítica de algoritmos existentes também permitiu escolher o modelo (YU; HE; GLASS, 2021) como a principal base para a implementação.

3.2.2 Implementação do Algoritmo

Após a fundamentação teórica, o próximo passo foi o desenvolvimento de algoritmos para percorrer grafos de conhecimento, com foco na resolução de questões que envolvem múltiplos saltos de contexto. Esses algoritmos foram projetados para explorar as conexões entre entidades e relações, navegando pelo grafo de forma eficiente e precisa para localizar as informações necessárias para responder perguntas complexas.

Durante o desenvolvimento, foram realizadas adaptações para alinhar os algoritmos às necessidades do projeto, considerando as especificidades do domínio da Amazônia

Azul e termos não recorrentes. Modificações foram introduzidas para melhorar a capacidade do sistema em lidar com questões que exigem navegação por caminhos inversíveis e interdependentes no grafo. Como resultado, foi implementado um sistema capaz de identificar trajetórias relevantes no grafo e utilizá-las para fornecer respostas coerentes e fundamentadas.

3.2.3 Testes de Precisão e Eficiência

Com o algoritmo implementado, foi desenvolvida uma bateria de testes para avaliar sua precisão e eficiência na navegação do grafo. Esses testes incluíram métricas como precisão na identificação de entidades e relações, além do tempo médio de resposta para consultas de múltiplos saltos. Foram utilizados tanto o grafo inicial quanto a versão expandida do *dataset* para assegurar que o sistema funcionasse de maneira robusta em diferentes condições.

Além disso, também foram realizados testes para avaliar a qualidade das respostas fornecidas pelo sistema de QA. Isso incluiu a análise qualitativa das respostas geradas para as 10 perguntas iniciais e outras questões derivadas da expansão do grafo. Os resultados obtidos ajudaram a identificar pontos de melhoria e realizar ajustes no modelo.

3.2.4 Integração com *Frontend* e *Backend*

A última etapa envolveu a integração do sistema com um *frontend* e *backend*, criando uma interface funcional que combina interatividade e visualização. No *backend*, utilizamos Python para implementar uma API que gerencia as consultas ao grafo e executa o modelo de QA. No *frontend*, foi desenvolvido um *chatbot* simples em JavaScript, capaz de receber perguntas e exibir as respostas geradas pelo sistema.

Além disso, foi incorporada uma funcionalidade para visualizar o percurso realizado pelo grafo durante a resposta a uma pergunta. Essa visualização permite que os usuários compreendam melhor como o sistema navega pelas conexões no grafo para gerar respostas. A integração dessas componentes torna o sistema mais intuitivo e acessível, destacando sua aplicabilidade em contextos reais.

4 Especificação de Requisitos e Projeto de Arquitetura

Durante o desenvolvimento do projeto, foi implementado um algoritmo de QA que busca respostas percorrendo um grafo de conhecimento. Paralelamente, foi desenvolvido um sistema com um *backend* que receba perguntas dos usuários, processando-as por meio do algoritmo, e uma interface que exibe, em tempo real, o percurso realizado no grafo para encontrar a resposta. Essa abordagem visa proporcionar maior transparência ao usuário sobre o processo de busca de respostas, contribuindo para a interpretabilidade do sistema.

4.1 Requisitos

O algoritmo, como requisito fundamental, deve ser capaz de reproduzir os resultados do Multihop KG (YU; HE; GLASS, 2021), servindo como base para a avaliação comparativa dos resultados. Isso implica que, ao receber uma pergunta, o algoritmo deve não apenas identificar os caminhos no grafo que levam à resposta, mas também apresentar esses caminhos ao usuário de maneira visual e compreensível. Além disso, o tempo máximo para concluir a operação deve ser de até dez segundos, garantindo a responsividade e usabilidade do sistema.

Para assegurar a portabilidade e acessibilidade, o sistema deve ser projetado para executar em ambientes computacionais modestos, como notebooks no Google Colab, equipados com até 12 GB de RAM e um processador Intel(R) Xeon(R) CPU @ 2.20GHz ou equivalente. Essa exigência ressalta a necessidade de otimizações no uso de memória e processamento, especialmente considerando que algoritmos de QA baseados em grafos podem demandar operações complexas de busca e inferência.

Outro aspecto crucial é o suporte a modelos de linguagem natural baseados em *embeddings*, que deverão ser integrados ao sistema para melhorar a compreensão semântica das perguntas. Isso exige que o *backend* seja compatível com *frameworks* de aprendizado de máquina populares, como TensorFlow ou PyTorch, e que permita a utilização de *embeddings* pré-treinados, como BERT ou FastText, para enriquecer a representação vetorial das perguntas e relações no grafo.

Como requisito funcional essencial, o sistema deve incluir uma interface que proporcione uma interação eficiente e intuitiva para o usuário. Essa interface deve permitir que o usuário insira perguntas em linguagem natural e receba, de maneira clara e organizada, as respostas geradas pelo sistema. Além disso, é fundamental que a interface apresente os

caminhos percorridos no grafo de conhecimento durante o processo de busca, destacando as melhores opções selecionadas pelo algoritmo.

Já entre os requisitos não funcionais, o sistema deve ser escalável, possibilitando a utilização de grafos de maior dimensão sem prejuízo significativo no desempenho. A interface deve ser intuitiva e acessível. Além disso, o sistema deve incluir logs detalhados das operações realizadas, tanto para fins de auditoria quanto para análise e melhoria contínua.

Por fim, a implementação do projeto deve seguir rigorosos padrões metodológicos e boas práticas de engenharia de software, incluindo a utilização de testes automatizados para validar a integridade do código, documentação clara e detalhada, e o uso de metodologias ágeis para garantir entregas incrementais e adaptáveis às necessidades do projeto. Esses elementos são fundamentais para garantir a qualidade, a robustez e a confiabilidade do sistema desenvolvido.

4.2 Arquitetura

Neste projeto, foi desenvolvido um sistema arquiteturalmente organizado em componentes especializados, com destaque para o módulo de busca. Esse módulo é responsável por receber uma pergunta do usuário e, a partir dela, percorrer grafos de conhecimento para identificar a melhor resposta. Nesta seção, descreveremos detalhadamente o funcionamento do sistema como um todo, destacando a integração entre seus componentes e as interações que ocorrem para viabilizar o processo de QA.

A arquitetura do sistema é composta por três camadas principais: a interface do usuário, o *backend* e o módulo de processamento de grafos. A interface é responsável por coletar as perguntas dos usuários e apresentar visualmente os resultados, incluindo a trajetória percorrida no grafo em tempo real. O *backend* atua como um intermediário, recebendo as entradas da interface, comunicando-se com o módulo de busca, e retornando as respostas de forma estruturada e eficiente. O módulo de busca, por sua vez, foi integrado ao *backend* e tem acesso direto ao grafo de conhecimento, permitindo uma navegação precisa e otimizada.

No nível interno, o módulo de busca foi projetado com uma arquitetura modular que incluirá subcomponentes responsáveis por etapas específicas, como pré-processamento da pergunta, tradução da linguagem natural para representações vetoriais (usando *embeddings*), e mecanismos de busca no grafo. Cada uma dessas subpartes foi interligada por meio de pipelines bem definidos, que garantirão que os dados fluam de forma consistente e eficiente entre as etapas. Para otimizar a comunicação entre os componentes, foi utilizado um modelo que salva o grafo previamente, permitindo processos paralelos aceleram a busca por respostas, respeitando o limite de tempo estabelecido.

Por fim, a integração entre as diferentes camadas do sistema foi projetada para ser altamente coesa e de fácil manutenção. A comunicação entre a interface e o *backend* será realizada via APIs, garantindo interoperabilidade e facilidade de implementação em diferentes plataformas. O módulo de busca será conectado ao *backend* por meio de chamadas diretas a bibliotecas internas, minimizando latências e garantindo um desempenho adequado mesmo em ambientes com recursos limitados. Essa arquitetura, cuidadosamente planejada, permite a implementação de um sistema robusto, escalável e apto a atender os requisitos.

4.2.1 Arquitetura de Alto Nível do Sistema

O sistema é composto de três partes principais:

1. Interface: a interface do sistema é o ponto de contato direto com o usuário, sendo responsável por receber as perguntas e exibir as respostas de forma clara e interativa. Desenvolvida utilizando Python com o *framework* Flask, a interface foi projetada para ser leve e eficiente, adotando uma abordagem baseada em renderização de *templates* HTML para apresentar as informações ao usuário. Além de exibir as respostas, a interface incorpora elementos visuais que demonstram, em tempo real, os percursos realizados no grafo de conhecimento, proporcionando maior transparência ao processo de busca;

2. *Backend*: o *backend* atua como o orquestrador central do sistema, sendo responsável por gerenciar a comunicação entre a interface e os componentes de processamento. Implementado com foco na modularidade e escalabilidade, ele recebe as perguntas enviadas pela interface e as encaminha para o Módulo de Resposta. Além disso, o *backend* integra serviços externos, como APIs para formatação de texto ou enriquecimento semântico, garantindo que as respostas sejam apresentadas de maneira clara e bem estruturada. Para isso, utiliza uma arquitetura baseada em APIs, promovendo interoperabilidade e facilidade de integração com outras ferramentas;

3. Módulo de resposta: O Módulo de Resposta é o componente responsável por processar as perguntas dos usuários e retornar as respostas mais apropriadas. Este projeto utiliza especificamente um algoritmo de QA baseado em grafos de conhecimento, que permite navegar por relações entre entidades para encontrar respostas relevantes. O módulo é dividido em subcomponentes especializados, como o pré-processador de perguntas, que transforma as consultas em formatos adequados para a busca, e o motor de busca, que percorre o grafo e retorna os caminhos e respostas. Sua arquitetura foi projetada para ser eficiente, com suporte a operações paralelas e integração com modelos de PLN para melhorar a compreensão semântica das perguntas.

Esses componentes, trabalhando de forma integrada, garantem que o sistema atenda aos requisitos de desempenho, usabilidade e escalabilidade definidos no projeto, oferecendo uma experiência de alta qualidade para os usuários.

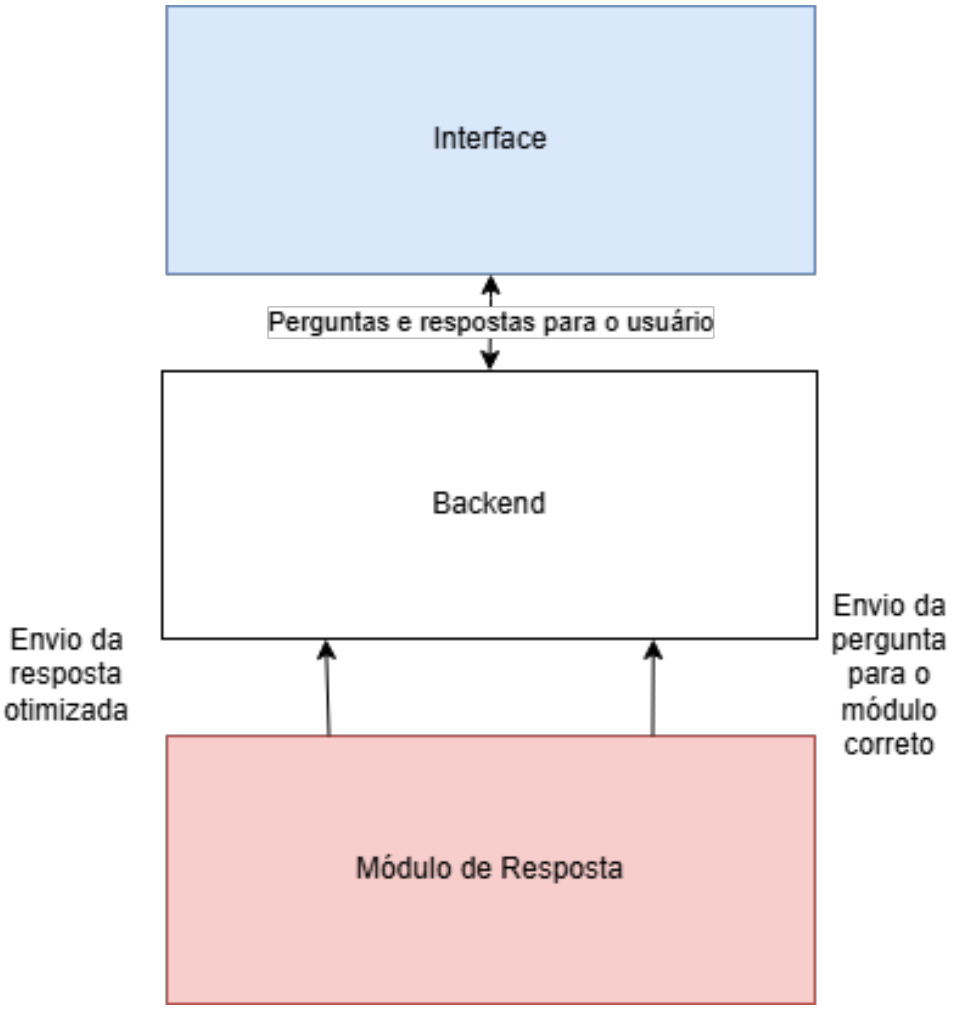

Figura 9 – Arquitetura do Sistema

4.2.2 Arquitetura do Módulo de Resposta

O módulo de resposta foi concebido como o núcleo do sistema, sendo responsável por processar as perguntas dos usuários e buscar respostas adequadas no grafo de conhecimento. Ele é composto por três componentes principais que operam de maneira integrada: Leitor de Grafos, Criador de *embeddings* e Podador. Cada um desempenha funções específicas que, em conjunto, asseguram a eficiência e precisão do sistema.

1. Leitor de Grafos: o Leitor de Grafos é o componente responsável pela ingestão inicial do grafo de conhecimento, garantindo que sua estrutura seja devidamente carregada e processada para posterior navegação. A leitura do grafo inclui a verificação da consistência dos dados, sendo que esse processamento inicial reduz significativamente o tempo necessário para o algoritmo percorrer o grafo durante a busca de respostas;

2. Criador de *embeddings*: o Criador de *embeddings* é o módulo responsável por transformar tanto a pergunta do usuário quanto os elementos do grafo (como nós e relações) em representações vetoriais no espaço contínuo. Esse processo é realizado utilizando modelos de PLN, como BERT, que permitem capturar a semântica das expressões de maneira eficiente. Ao gerar *embeddings* vetoriais, o sistema pode calcular similaridades entre perguntas e relações do grafo com maior precisão, permitindo que os caminhos mais relevantes sejam priorizados durante a busca. Além de gerar *embeddings* para a pergunta, este módulo processa os caminhos percorridos no grafo e calcula sua representação vetorial cumulativa, levando em consideração o contexto semântico das relações. Isso é essencial para que o sistema consiga estabelecer conexões significativas entre a pergunta do usuário e os dados do grafo, mesmo quando a correspondência exata de palavras não está presente;

3. Podador: este componente desempenha um papel crítico no desempenho do sistema, sendo responsável por limitar as expansões do algoritmo durante a navegação no grafo. Ele utiliza a métrica de similaridade de cosseno para avaliar a proximidade entre os *embeddings* da pergunta e as representações vetoriais dos caminhos percorridos. A similaridade de cosseno, definida como o cosseno do ângulo entre dois vetores em um espaço multidimensional, é amplamente utilizada em sistemas de recuperação de informação por sua capacidade de medir a similaridade sem depender da magnitude dos vetores (A., 2001). Este módulo implementa critérios de corte baseados em limiares configuráveis, descartando os k caminhos cuja similaridade seja menor. Essa poda seletiva reduz significativamente o espaço de busca, mantendo apenas os caminhos mais promissores e, consequentemente, acelerando a identificação da resposta. Além disso, o Podador contribui para a escalabilidade do sistema, permitindo que ele lide com grafos de maior complexidade sem comprometimento significativo no desempenho.

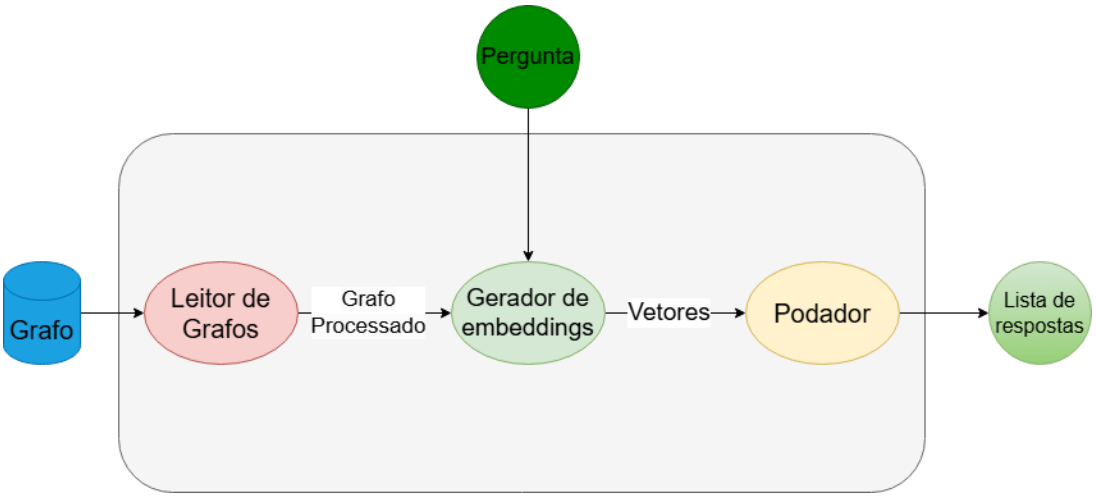

Figura 10 – Arquitetura do Módulo de Resposta

5 Implementação e Testes

5.1 Tecnologias Utilizadas

Para o desenvolvimento do projeto, foi empregada uma gama diversificada de tecnologias que, em conjunto, facilitam tanto a implementação quanto o teste do algoritmo responsável por percorrer o grafo de conhecimento e fornecer respostas precisas. Cada ferramenta desempenha um papel essencial, contribuindo significativamente para a eficiência, a robustez e a escalabilidade da solução proposta. A seleção dessas tecnologias foi feita considerando não apenas suas capacidades técnicas, mas também sua adequação aos requisitos do projeto, incluindo facilidade de uso, integração e desempenho.

Python Notebooks: Um dos pilares do ambiente de desenvolvimento são os notebooks interativos, como o Jupyter. Esses notebooks proporcionam um espaço de trabalho que combina código executável, visualizações e documentação, permitindo que o desenvolvimento ocorra de maneira incremental e iterativa. Essa abordagem facilita a validação e a depuração da lógica de percurso no grafo, permitindo análises imediatas dos resultados e ajustes rápidos. Além disso, a capacidade de compartilhar notebooks entre os membros da equipe promove a colaboração e acelera o ciclo de desenvolvimento, tornando o processo mais ágil e integrado.

Pandas e NumPy: Para a manipulação de dados, as bibliotecas Pandas e NumPy desempenham papéis complementares e fundamentais. O Pandas é utilizado para estruturar, organizar e manipular grandes volumes de dados no formato de *dataframes*, fornecendo uma base eficiente para trabalhar com informações complexas provenientes do grafo de conhecimento. Já o NumPy é empregado para operações matemáticas e manipulações em arrays multidimensionais, garantindo eficiência e alto desempenho no processamento de dados. Essas ferramentas são cruciais para a execução de cálculos vetoriais e outras operações essenciais para a lógica do algoritmo.

Scikit-learn: A biblioteca Scikit-learn (sklearn) é um elemento-chave na implementação das métricas de similaridade textual, como a similaridade de cosseno, que é amplamente utilizada para comparar a proximidade semântica entre as palavras presentes nas perguntas dos usuários e os relacionamentos do grafo de conhecimento. Essa técnica é indispensável para o projeto, pois permite identificar, com precisão, os nós mais relevantes no grafo para cada consulta realizada. Além disso, o Scikit-learn oferece uma ampla gama de ferramentas para análise e manipulação de dados que podem ser utilizadas para aprimorar ainda mais o desempenho do sistema.

Transformers e TensorFlow: As bibliotecas Transformers, da Hugging Face, e

TensorFlow são exploradas para incorporar redes neurais ao sistema, ampliando suas capacidades de PLN. O Transformers é utilizado para integrar modelos pré-treinados, como o BERT e variantes mais recentes, que oferecem alto desempenho em tarefas de compreensão semântica. Por outro lado, o TensorFlow fornece uma infraestrutura flexível para treinar modelos personalizados de redes neurais, caso surja a necessidade de soluções mais especializadas. Essas tecnologias avançadas adicionam uma camada de sofisticação ao sistema, permitindo a associação inteligente entre perguntas e respostas no grafo.

A combinação dessas ferramentas cria uma base tecnológica sólida para a construção de um sistema robusto e escalável. Essa integração possibilita não apenas um alto grau de precisão nas respostas fornecidas, mas também um desempenho eficiente, mantendo a experiência do usuário fluida e intuitiva. O uso dessas tecnologias reflete uma abordagem moderna e inovadora no desenvolvimento de sistemas baseados em grafos de conhecimento.

5.2 Projeto e Implementação

Nesta seção, será explicado o algoritmo utilizado para conseguir fazer a escolha da resposta correta em nosso grafo de conhecimento.

5.2.1 Importação de bibliotecas

Instalação de Bibliotecas Necessárias: O primeiro bloco de código instala as bibliotecas essenciais para o desenvolvimento do projeto. Aqui, estão incluídas bibliotecas como transformers, torchvision, pandas, numpy, sentence-transformers e scikit-learn. Essas bibliotecas fornecem as ferramentas fundamentais para o processamento de dados, manipulação de matrizes, processamento de linguagem natural (PLN) e algoritmos de aprendizado de máquina. A instalação dessas dependências é crucial para garantir que o ambiente de desenvolvimento tenha os pacotes apropriados para manipular grandes volumes de dados textuais e realizar cálculos de similaridade entre palavras.

Listing 5.1 – Importação de Bibliotecas

```
1  import os
2  print(os.path)
3  from google.colab import drive
4  drive.mount('/content/drive')
5  import numpy as np
6  import pandas as pd
7  import torch
8  import csv
9  from google.colab import drive
10 import nltk
```

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity
from sentence_transformers import SentenceTransformer
from transformers import BertTokenizer, BertModel
from transformers import BertForQuestionAnswering
from transformers import BertTokenizer
model = SentenceTransformer('bert-base-nli-mean-tokens')
```

5.2.2 Função Get Surface Entities

Após isso, a função *get_surface_entities* tem como objetivo identificar as entidades relevantes em uma consulta (ou pergunta) com base em um grafo de conhecimento, retornando entidades que correspondam às palavras-chave presentes na pergunta, desde que estas tenham mais de três caracteres.

A função começa definindo as colunas do grafo em que a busca será realizada. Neste caso, as colunas selecionadas são "*Subject*" e "*Object*", que correspondem às entidades e objetos no grafo. O grafo de conhecimento normalmente possui uma estrutura relacional, onde as entidades e os objetos (relacionamentos ou descritores) são armazenados nessas colunas.

A consulta fornecida pelo usuário, armazenada em *query*, é dividida em palavras individuais utilizando o método *split*(). Em seguida, é criada uma nova lista, *key_words*, onde são adicionadas apenas as palavras que têm mais de três caracteres. O critério de tamanho é utilizado para evitar palavras muito curtas, como artigos ou preposições, que geralmente não são informativas para a busca de entidades.

Depois desse tratamento inicial, a busca é realizada no grafo. Para cada coluna especificada (no caso, *Subject* e *Object*), a função percorre todas as entidades do grafo. Em cada linha, a função verifica se alguma das palavras-chave da consulta está contida no valor da célula correspondente à coluna, ou se o valor da célula está contido na palavra-chave (ignorando diferenças de maiúsculas e minúsculas com o método *lower*()). Se uma correspondência for encontrada, o valor correspondente à entidade (ou objeto) é adicionado à lista *results*.

Após a busca, a função converte a lista de resultados em um conjunto (*set*) para remover duplicatas. Isso garante que cada entidade seja retornada apenas uma vez, mesmo que tenha aparecido em múltiplas correspondências. Por fim, retorna a lista de resultados.

Listing 5.2 – Função Get Surface Entities

```
#Funcao que seleciona as entidades da pergunta(tamanho > 3)
def get_surface_entities(query, graph):
    # Define as colunas em que deseja fazer a busca
```

```
    search_columns = ['Subject', 'Object']
    # Cria uma lista de palavras-chave a partir da pergunta
    key_word = query.split()
    key_words = []
    for element in key_word:
      if len(element) > 3:
        key_words.append(element)
    # Faz a busca no grafo pelas entidades
    results = []
    for search in search_columns:
      for index, row in graph.iterrows():
          for word in key_words:
              if word.lower() in str(row[search]).lower() or str(
                 row[search]).lower() in  word.lower():
                  results.append(row[search])

    results = list(set(results))
    return results
```

5.2.3 Função Get Triples

Além disso, temos a função *get_triples*, que tem como objetivo extrair todas as triplas do grafo de conhecimento. Essas triplas são fundamentais na representação de grafos de conhecimento, pois descrevem as relações entre diferentes entidades por meio da estrutura (Sujeito, Relação, Objeto).

A função começa selecionando as três colunas do grafo de conhecimento, que são: *Subject*, *Relation*, e *Object*. Estas colunas formam o núcleo de uma tripla, onde o *Subject* (Sujeito) é a entidade principal, o *Relation* (Relação) é o tipo de vínculo ou interação entre as entidades, e o *Object* (Objeto) é a entidade associada. A operação de extração dessas colunas cria um subconjunto dos dados chamado *triples_dataset*, que contém apenas as informações necessárias para a criação das triplas.

Neste passo, a função inicializa uma lista vazia chamada triples, que armazenará as triplas formadas a partir do grafo. O método *iterrows*() é utilizado para percorrer cada linha do subconjunto de dados *triples_dataset*. Para cada linha, uma tupla contendo os valores das colunas *Subject*, *Relation* e *Object* é criada e adicionada à lista *triples*. A cada iteração, a função captura a tripla de uma linha específica, garantindo que todas as relações entre entidades sejam registradas. Por fim, as triplas são retornadas

Listing 5.3 – Função Get Triples

```
#Essa funcao retorna todas as triplas que temos no grafo
```

```
2 def get_triples(graph):
3     #Leitura do grafo e selecao das colunas pertinentes
4     triples_dataset = graph[['Subject','Relation','Object']]
5     #Criacao das triplas
6     triples=[]
7     for index, row in triples_dataset.iterrows():
8         triples.append((row['Subject'], row['Relation'], row['
              Object']))
9     return triples
```

5.2.4 Função Get Link

Neste passo, a função começa filtrando as triplas fornecidas, procurando aquelas em que a entidade de superfície (fornecida pelo parâmetro *surface_entity*) aparece como sujeito. Ou seja, a função verifica a primeira posição de cada tripla (que corresponde ao sujeito) e, se a entidade coincide com essa posição, a tripla é adicionada à lista *filtered_triples_subject*. Esse procedimento identifica todas as relações onde a entidade atua como sujeito.

A função então realiza a mesma filtragem, mas desta vez buscando as triplas onde a entidade de superfície atua como objeto, ou seja, na terceira posição da tripla. Se a entidade coincide com o objeto da tripla, a tripla é invertida usando [::-1] e adicionada à lista *filtered_triples_object*. A inversão é feita para garantir que todas as triplas retornadas apresentem a entidade como sujeito, mantendo uma estrutura consistente no resultado, facilitando o uso das triplas posteriormente.

Após filtrar as triplas onde a entidade aparece como sujeito e objeto, as duas listas (*filtered_triples_subject* e *filtered_triples_object*) são combinadas em uma única lista. Isso assegura que todas as triplas relacionadas à entidade, independentemente de sua posição, sejam incluídas no conjunto final.

Para evitar duplicações e garantir que cada tripla seja única no resultado, a lista combinada é convertida em um conjunto (*set*). A conversão para conjunto elimina qualquer ocorrência repetida de uma mesma tripla. O conjunto é, então, retornado.

Listing 5.4 – Função Get Link

```
1    def get_link(surface_entity, triples):
2    # Filtramos as triplas pegando apenas as que possuem as
        entidades como objeto ou sujeito
3    filtered_triples_subject = [triple for triple in triples if (
        triple[0] == surface_entity )]
4    filtered_triples_object = [triple[::-1] for triple in triples
         if (triple[2] == surface_entity)]
5    filtered_triples_subject += filtered_triples_object
6    # Implementacao para obter o link de uma entidade de
        superficie
7    threshold_triples = set(filtered_triples_subject)
8    return threshold_triples
```

5.2.5 Função Calculate Cosine Similarity with Predicate

Para enriquecer sua explicação, podemos detalhar o funcionamento interno e a lógica de cada etapa da função, além de contextualizar o uso da similaridade de cosseno em problemas de Processamento de Linguagem Natural (PLN) e em sistemas de recuperação de informação:

A função *calc_cosine_similarity_with_predicate* foi projetada para calcular a similaridade de cosseno entre uma pergunta formulada pelo usuário e uma sequência de relações em um grafo de conhecimento, possibilitando a identificação de uma correspondência semântica entre ambas as entradas. Esse processo é especialmente útil em sistemas que dependem da recuperação de conhecimento, pois a similaridade de cosseno é uma métrica robusta para avaliar a proximidade semântica entre vetores de alta dimensionalidade, frequentemente usados para representar sentenças ou palavras no espaço vetorial.

A função inicia avaliando o tipo da entrada *query_enc*, que representa a pergunta do usuário. Esse procedimento é necessário pois a pergunta pode estar representada como uma lista de tokens, onde cada elemento é uma palavra. Para garantir a consistência e que o texto seja tratado como uma única unidade semântica, a função une esses tokens em uma *string* com um delimitador de espaço. Essa normalização permite que o modelo de *embeddings* possa processar a entrada de forma padronizada.

Para a representação da sequência de relações do grafo (*path_enc*), a função seleciona apenas as relações (elementos nos índices ímpares), que são então concatenadas em uma *string* única. Esse procedimento transforma a sequência de relações em um formato textual contínuo, preservando a ordem e estrutura semântica do caminho no grafo.

Após preparar as entradas como strings, a função utiliza um modelo de transformação de linguagem (como BERT ou outro modelo de *embeddings*) para gerar os *embeddings*

das duas sentenças. Cada sentença é transformada em um vetor de alta dimensionalidade que captura suas características semânticas e contextuais. Neste caso, a função *model.encode* é utilizada para esse propósito, convertendo a pergunta e o caminho do grafo em representações vetoriais que refletem suas semelhanças e distinções semânticas.

Com os vetores de *embeddings* das duas sentenças em mãos, a função aplica o método *cosine_similarity* da biblioteca Sklearn para calcular a similaridade de cosseno entre os vetores. Essa métrica mede o ângulo entre os dois vetores no espaço vetorial, onde valores mais próximos de 1 indicam uma alta similaridade (vetores próximos no espaço), enquanto valores próximos de 0 indicam vetores ortogonais, ou seja, sem relação semântica aparente.

Em contextos de recuperação de informação, a similaridade de cosseno permite quantificar a relevância entre a pergunta e as relações no grafo, possibilitando uma comparação precisa e eficaz entre a intenção do usuário e o conhecimento estruturado. Dessa forma, o sistema pode identificar as relações mais pertinentes no grafo de conhecimento para responder à pergunta do usuário de maneira precisa e contextualmente adequada.

Listing 5.5 – Função Calc Cosine Similarity with Predicate

```
1  def calc_cosine_similarity_with_predicate(path_enc, query_enc
       ):
2      sentences = []
3      if isinstance(query_enc,list):
4        delimitador = "␣"
5        query_enc = delimitador.join(query_enc)
6      sentences.append(query_enc)
7      sentences.append("␣".join(path_enc[1::2]))
8      sentence_embeddings = model.encode(sentences)
9      arr = cosine_similarity(sentence_embeddings[0].reshape(1,-1),
           sentence_embeddings[1].reshape(1,-1))
10     return arr[0][0]
```

5.2.6 Função Prune Paths

A função *prune_paths* tem como objetivo podar um conjunto de caminhos P com base na similaridade de cosseno entre esses caminhos e uma codificação de consulta qenc. A poda é feita para garantir que o número de caminhos seja limitado a um fator de ramificação máximo (*beam size*), preservando os caminhos mais relevantes. A técnica de *beam search* é frequentemente utilizada em algoritmos de busca e otimização para limitar o número de candidatos que são explorados em cada iteração, mantendo apenas os mais promissores.

Incialmente, a função primeiro transforma cada caminho em P de uma lista para uma tupla, de modo a garantir que os caminhos sejam imutáveis e, em seguida, cria um *set* para remover duplicatas, já que conjuntos (*sets*) não permitem duplicação de elementos. Posteriormente, as tuplas são convertidas de volta para listas. Isso garante que P contenha apenas caminhos únicos, evitando comparações desnecessárias com caminhos repetidos durante o processo de poda.

A seguir, a função verifica se o número de caminhos P já é menor ou igual ao fator de ramificação. Se for, não há necessidade de podar, e a função retorna o conjunto de caminhos sem modificações. Este é um caso de base que otimiza o tempo de execução, evitando a execução de cálculos adicionais desnecessários.

Se o número de caminhos for maior que *beam size*, a função passa para o processo de poda. Primeiro, ela inicializa uma lista *cossimilarity* para armazenar a similaridade de cosseno entre cada caminho e a codificação da consulta qenc. A função *calc_cosine_similarity_with_predicate* é usada para calcular a similaridade entre cada caminho e *qenc*, e o resultado é armazenado na lista como uma tupla contendo o caminho e sua similaridade correspondente.

Depois de calcular as similaridades de cosseno, a função ordena os caminhos com base nesses valores de similaridade, do maior para o menor (*reverse=True*). Isso significa que os caminhos mais semelhantes à consulta *qenc* (aqueles com maior similaridade de cosseno) aparecem primeiro na lista. Esta ordenação é crucial, pois permite à função podar os caminhos menos relevantes, que têm menor similaridade.

O processo de poda ocorre dentro de um laço *while*, que continua até que o número de caminhos em P seja igual ao valor do fator de ramificação. Em cada iteração, a função remove o caminho com a menor similaridade de cosseno, que está localizado no final da lista ordenada (*cossimilarity*[-1]). O caminho correspondente é removido de P, e a tupla com esse caminho e sua similaridade é removida de *cossimilarity* com o método *pop*(). A função termina retornando P.

Listing 5.6 – Função Prune Paths

```
1     def prune_paths(P, qenc, beam_size):
2     # Implementacao para podar P com base na similaridade do
          cosseno com um tamanho de feixe B
3     # Caso o total de caminhos B nao tenha sido ultrapassado,
          retorna o set de caminhos sem alteracoes
4     P = [list(t) for t in set(tuple(path) for path in P)]
5     if len(P) <= beam_size:
6       return P
7     # Caso contrario faz a poda, armazenando as cosseno-
          similaridades em um dicionario
```

```
8     else:
9       cossimilarity = []
10      for path in P:
11        similarity = calc_cosine_similarity(path,qenc)
12        cossimilarity.append((path, similarity))
13      cossimilarity = sorted(cossimilarity, key=lambda x: x[1],
           reverse=True)
14      # Procedimento de poda,enquanto o numero de feixes nao for
           o desejado remove a menor cosseno-similaridade do Set
15      while len(P) > beam_size:
16          aux = cossimilarity[-1]
17          P.remove(aux[0])
18          cossimilarity.pop()
19
20    return P
```

5.2.7 Função Get answer

A função *get_answer* tem como objetivo encontrar os melhores caminhos em um grafo de conhecimento a partir de entidades de superfície fornecidas como entrada. Esses caminhos são usados para inferir uma resposta relacionada à consulta fornecida (*query*). A função utiliza uma técnica chamada *beam search* (busca em feixe), que mantém apenas os caminhos mais promissores em cada etapa de expansão, limitando o número de caminhos considerados a um valor máximo, *beam size*. O algoritmo também itera sobre um número pré-definido de "saltos" no grafo (*number_of_hops*), expandindo os caminhos com base nas conexões entre entidades do grafo. A seguir, está a explicação detalhada dos componentes dessa função.

A função começa um laço que itera pelo número de "saltos" (*hops*) definidos em *number_of_hops*. Um salto pode ser entendido como um passo ou conexão entre duas entidades no grafo. A cada iteração, a função expande os caminhos atuais (*paths*) explorando as conexões das entidades.

No início da busca (quando *paths* está vazio), a função cria caminhos iniciais para cada entidade de superfície (*seed_surface_entities*). Para isso, ela chama a função *get_link*, que retorna as entidades conectadas ao *seed* (entidade inicial). A função constrói caminhos formados por triplas (sujeito, relação, objeto) e adiciona esses caminhos à lista *paths*. Em seguida, os caminhos são podados utilizando a função *prune_paths*, que limita o número de caminhos ao fator de ramificação, mantendo apenas os mais promissores com base na similaridade com a consulta.

Se já houver caminhos em *paths*, a função expande esses caminhos explorando novas

entidades conectadas à última entidade de superfície de cada caminho. A função *get_link* é chamada para obter as conexões da última entidade no caminho atual (*last_surface_entity)*. Para evitar *loops* ou redundâncias no caminho, a função garante que a entidade recém-adicionada (*connected_surface_entity*[2]) não seja a mesma que a entidade de três posições atrás no caminho (*path*[-3]). Esses novos caminhos são adicionados à lista *new_paths*, e em seguida, eles são podados para manter apenas os mais relevantes.

Após a iteração sobre os saltos, a função classifica os caminhos em *paths* com base na similaridade de cosseno entre o caminho e a consulta (*query*). A função *calc_cosine similarity_with_predicate* é usada para calcular essa similaridade. Os caminhos são classificados em ordem decrescente de similaridade (os mais semelhantes primeiro), e apenas os k melhores caminhos são retornados na variável top k.

Listing 5.7 – Get Answer

```
1    def get_anwser(k, beam_size, number_of_hops,
        seed_surface_entities, paths, query):
2   #Fazemos uma iteracao para cada numero pre definido de hops
3   for hop in range(number_of_hops):
4     # Se nao temos caminhos, para cada entidade de superficie
        seed i, criamos um caminho novo
5     if not(paths):
6       for seed in seed_surface_entities:
7         for connected_surface_entity in get_link(seed, triples):
8           path = [ connected_surface_entity[0],
                connected_surface_entity[1], connected_surface_entity
                [2]]
9           paths.append(path)
10      paths = prune_paths(paths, query, beam_size)
11    else:
12      new_paths = []
13      # Para cada caminho p em P, faca o seguinte:
14      for path in paths:
15          # Obtenha a ultima entidade de superficie em p
16          last_surface_entity = path[-1]
17          # Para cada entidade de superficie conectada a
                last_surface_entity, pega os links:
18          for connected_surface_entity in get_link(
                last_surface_entity, triples):
19            if(connected_surface_entity[2] != path[-3]):
20                new_path = path + [ connected_surface_entity[1],
                    connected_surface_entity[2]]
21                new_paths.append(new_path)
```

```
22          paths += prune_paths(new_paths, query, beam_size)
23
24      # Retorna o conjunto de caminhos final
25      top_k = sorted(paths, key=lambda path: calc_cosine_similarity(
            path, query), reverse=True)[:k]
26      return top_k
```

5.3 Testes e Avaliação

O processo de testes e avaliação do componente desenvolvido foi conduzido de maneira minuciosa e estruturada, abrangendo a aplicação de diversos tipos de testes, modelos pré-treinados e algoritmos, com o objetivo de assegurar o máximo de precisão e confiabilidade no funcionamento do algoritmo de QA. Cada fase do processo foi cuidadosamente planejada, envolvendo a execução de testes exaustivos para verificar o comportamento dos modelos em diferentes cenários e condições, sempre com foco na robustez da solução. Além disso, foi considerada a variabilidade de desempenho entre os diferentes algoritmos utilizados, o que permitiu uma análise detalhada da capacidade de resposta e da adequação do sistema a perguntas complexas e com salto de contexto.

5.3.1 Testes iniciais

Inicialmente, os testes focaram em analisar o desempenho computacional, no caso, foi utilizado o tempo de reposta total, a eficiência na geração de respostas em perguntas com salto de contexto e na acurácia das respostas. Durante esses testes, foram utilizadas duas abordagens distintas para obtenção dos valores das palavras, aplicadas a um pequeno conjunto de perguntas geradas sobre a Amazônia Azul.

O pequeno conjunto de perguntas sobre a Amazônia Azul consistia de 10 perguntas acompanhadas de um grafo de conhecimento de 28 relações extraídas de um paragrafo de um documento sobre a Amazônia Azul. Diante desses dados extraídos do documento, foram realizados essas 10 perguntas para cada modelo pré-treinado diferente e utilizando abordagens diferentes.

Essas abordagens diferem apenas na forma na qual o modelo pré-treinado extrai o valor de semelhança entre frases e palavras, uma das abordagens utilizadas foi a comparação entre os valores gerados para as frases inteiras, ou seja, era extraído o valor tanto da frase da pergunta como da frase da resposta na qual era gerada concatenando o caminho percorrido pelo grafo de conhecimento. A outra abordagem consiste em gerar o valor de palavra por palavra e fazer uma média do valor resultado do modelo e comparar com o valor da frase da pergunta.

A seguir é possível ver o resultado desses testes iniciais, onde possui duas colunas principais para ambas abordagens de frase inteira e palavra por palavra, e abaixo das colunas possuem os diferentes modelos pré-treinados utilizados para gerar a semelhança da resposta e da pergunta. Assim, para cada pergunta é possível ver um quadro verde, amarelo ou vermelho para cada par de método e modelo. O verde representa que o par método-modelo acertou a questão, o amarelo representa que a resposta correta estava entre as 5 respostas mais semelhantes a pergunta e, por fim, o vermelho representa um erro pois a resposta não aparecia nem nas melhores 5 respostas.

Método	**Frase inteira**						**Palavra por palavra**		
Modelo	bert-base-nli-mean-tokens	bert uncased	bert cased	big bert uncased	mpnet-base	multi-qa mpnet	bert-base-nli-mean-tokens	bert uncased	bert cased
Pergunta #1									
Pergunta #2									
Pergunta #3									
Pergunta #4									
Pergunta #5									
Pergunta #6									
Pergunta #7									
Pergunta #8									
Pergunta #9									
Pergunta #10									

Tabela 5 – Tabela de resultados dos testes iniciais

Os resultados obtidos nos testes iniciais indicaram que o desempenho dos modelos avaliados, como BERT *uncased*, *Big* BERT e outros pré-treinados, não foi satisfatório. Nenhum dos modelos conseguiu superar a marca de pelo menos 50% de acertos no conjunto de perguntas testado, conforme ilustrado na Tabela 1. As cores indicam que, na maioria dos casos, os modelos falharam em apresentar a resposta correta, com muitos erros (marcados em vermelho) e algumas poucas aproximações aceitáveis (amarelo), refletindo uma baixa precisão geral. Um dos fatores que pode ter contribuído para esses resultados insatisfatórios foi a limitação na quantidade de perguntas e dados utilizados para os testes, o que possivelmente comprometeu a capacidade dos modelos de gerar respostas mais precisas. Além disso, a complexidade das perguntas, com saltos de contexto, pode ter acentuado as dificuldades dos modelos pré-treinados, evidenciando a necessidade de mais ajustes e uma base de dados maior para otimizar a performance do sistema.

Diante dos resultados insatisfatórios dos testes iniciais, onde nenhum dos modelos alcançou 50% de acertos, decidimos prosseguir com uma nova fase de testes utilizando um banco de dados maior e mais robusto e ao mesmo tempo verificar a consistência e coesão do grafo de conhecimento. Considerando que a quantidade limitada de perguntas e dados pode ter influenciado negativamente o desempenho dos modelos, optamos por testar as mesmas abordagens utilizando grandes corpora, como o banco de dados da Google e a Wikipedia. Esses conjuntos de dados ampliados nos permitiram explorar uma gama mais rica de informações e contextos, oferecendo aos modelos uma base mais sólida

para a geração de respostas. Com isso, esperamos obter uma melhora significativa no desempenho dos algoritmos de QA, especialmente em perguntas com salto de contexto, onde é necessário lidar com informações distribuídas em múltiplas fontes.

5.3.2 Análise do grafo de conhecimento e nova forma de calcular similaridade

Após os testes iniciais com o sistema de QA utilizando saltos de contexto sobre o grafo de conhecimento, os resultados mostraram uma acurácia abaixo do esperado, indicando possíveis limitações no modelo e na estrutura de dados disponível. Dado o impacto dessa baixa acurácia na capacidade do sistema de fornecer respostas coerentes, foi necessário realizar uma análise minuciosa do grafo de conhecimento para entender se a estrutura e o conteúdo das entidades e relações estavam adequados para o propósito do projeto. Esse estudo foi essencial para identificar eventuais lacunas ou inconsistências que pudessem estar comprometendo o desempenho do sistema, antes de partir para um maior banco de dados com uma solução possivelmente ineficiente.

Durante essa análise, observamos que o grafo, em grande parte, estava bem estruturado e adequado para responder às perguntas esperadas no contexto da Amazônia Azul. A quantidade de entidades e as relações entre elas eram amplamente suficientes para cobrir o domínio de conhecimento necessário. Contudo, encontramos alguns casos em que relações específicas apresentavam inconsistências ou redundâncias que poderiam confundir o algoritmo na hora de selecionar a resposta correta. Essas relações problemáticas, embora poucas, tinham o potencial de afetar negativamente o processo de *multi-hop* e, consequentemente, a qualidade das respostas geradas para algumas perguntas específicas.

Diante dessa descoberta, optamos por manter o grafo existente, mas aprimorar a maneira como o modelo processa e interpreta as informações nele contidas. Para isso, decidimos adotar uma nova abordagem para gerar os valores de similaridade entre as perguntas e o grafo. Essa abordagem consiste em um novo método para calcular a similaridade entre a consulta do usuário e os nós do grafo, focando na estrutura das perguntas e nas relações entre as entidades para garantir uma representação mais precisa dos contextos envolvidos. A nova metodologia busca reforçar a precisão dos cálculos, compensando eventuais lacunas nas relações do grafo.

Com essa estratégia, obtivemos uma melhoria significativa na precisão das respostas, já que a nova forma de calcular a similaridade não apenas aprimora a detecção de relações relevantes, mas também minimiza os efeitos de inconsistências no grafo. Essa abordagem nos permite utilizar o grafo de maneira otimizada, mantendo-o atualizado e funcional para responder a perguntas complexas com múltiplos contextos, o que é fundamental para alcançar a qualidade almejada no sistema de *multi-hop* QA.

Os resultados após a implementação do novo método foram muito satisfatórios e

superaram definitivamente os testes iniciais, com um aumento significativo na acurácia das respostas. A mudança para calcular a similaridade utilizando apenas o *embedding* gerado pelo predicado em cada tripla, em vez de considerar toda a estrutura (sujeito + predicado + objeto) semelhante a forma em que (M. JAY PRAKASH, 2022) realiza, revelou-se uma abordagem eficaz. Essa modificação permitiu ao modelo focar exclusivamente na relação principal entre as entidades, eliminando ruídos no cálculo de similaridade e tornando o processo de seleção de respostas mais preciso. O uso do predicado como elemento central reforçou a capacidade do modelo de captar a essência do relacionamento específico contido na pergunta, facilitando o retorno de respostas mais relevantes.

Além dessa alteração, realizamos uma atualização nos modelos pré-treinados de BERT utilizados para as tarefas de *multi-hop* QA. Alguns dos modelos empregados anteriormente estavam desatualizados ou *deprecated*, o que impactava tanto o desempenho quanto a compatibilidade com as novas bibliotecas de PLN. Substituímos esses modelos por versões mais recentes e robustas, ajustadas às melhorias implementadas em bibliotecas modernas de *deep learning* e PLN, como o Hugging Face Transformers. Essas atualizações otimizaram o processamento e garantiram que as métricas de similaridade fossem calculadas com base nos padrões mais avançados de PLN, aumentando ainda mais a acurácia geral do sistema. Abaixo, incluímos uma tabela com os resultados detalhados para cada modelo pré-treinado testado:

Método	**Embedding do predicado**				
Modelo	all-mpnet -base-v2	multi-qa-mpnet -base-dot-v1	msmarco-bert -base-dot-v5	multi-qa-mpnet -base-cos-v1	multi-qa-distilbert -cos-v1
Pergunta #1					
Pergunta #2					
Pergunta #3					
Pergunta #4					
Pergunta #5					
Pergunta #6					
Pergunta #7					
Pergunta #8					
Pergunta #9					
Pergunta #10					

Tabela 6 – Tabela de resultados dos testes após mudança de método

Com essa combinação de melhorias—a alteração no cálculo de *embeddings* e a atualização dos modelos – o sistema demonstrou uma performance superior ao responder perguntas que exigem saltos de contexto complexos. A nova abordagem reduziu significativamente o tempo de processamento e aumentou a confiabilidade das respostas, com o sistema agora alcançando altos índices de precisão mesmo em perguntas mais desafiadoras. No entanto, mesmo com as mudanças implementadas, algumas perguntas ainda não foram respondidas corretamente. Ao analisar essas falhas, identificamos que o problema residia na estrutura do grafo de conhecimento, que, em certos casos, fornecia predicados muito

superficiais, sem detalhar adequadamente os relacionamentos entre as entidades. Esse nível superficial de informação nas relações limita a capacidade do sistema de captar nuances específicas da pergunta, dificultando a identificação da resposta correta. Assim, fica evidente que, para alcançar uma cobertura mais completa, será necessário enriquecer o grafo com predicados que aprofundem os relacionamentos, permitindo ao modelo uma compreensão mais robusta e precisa do contexto necessário para responder a questões complexas.

5.3.3 Expansão do grafo e novo teste para decisão do modelo

Após a execução inicial dos testes com o objetivo de identificar o método mais eficiente para a comparação entre os *embeddings* do grafo e a pergunta, foi necessário realizar uma segunda etapa para definir o modelo de BERT mais adequado ao caso de uso. Para garantir resultados mais representativos e robustos, optamos por realizar os testes em um grafo expandido e mais complexo, eliminando as limitações apresentadas pela versão simplificada utilizada previamente.

A construção de um grafo mais robusto e consistente foi essencial para alcançar a principal meta do projeto, que é identificar o melhor algoritmo e modelo para percorrer grafos de conhecimento, assegurando uma alta acurácia no sistema de QA. O processo de expansão do grafo foi realizado manualmente, com a inclusão de relações adicionais, assegurando que os dados mantivessem relevância semântica e conexão lógica entre as entidades. Essa expansão reflete a necessidade de um ambiente de teste mais próximo de cenários reais e desafiadores.

Especificamente, o número de relações no grafo foi ampliado de 28 para mais de 70, representando um crescimento substancial em complexidade. Além disso, o conjunto de perguntas utilizado nos testes também foi revisado e ampliado, passando de 10 para 20 perguntas distintas. Esse aumento permitiu avaliar não apenas a capacidade dos modelos de lidar com um grafo maior, mas também sua eficiência em situações com maior diversidade de questionamentos e possíveis caminhos.

Os modelos avaliados foram variantes de BERT, incluindo diferentes configurações e ajustes específicos para tarefas de QA. Cada modelo foi analisado em termos de sua habilidade em priorizar corretamente os caminhos relevantes dentro do grafo. A classificação dos resultados foi feita com base em três critérios: respostas corretas classificadas em primeiro lugar (verde), respostas corretas classificadas entre os cinco primeiros resultados (amarelo) e casos em que o caminho correto não apareceu entre os cinco primeiros (vermelho). Essa categorização permitiu uma análise qualitativa e quantitativa do desempenho de cada modelo.

Com base nesses resultados, conclui-se que a expansão do grafo foi um passo crítico

para a avaliação mais precisa do desempenho dos modelos. A análise comparativa revelou as vantagens e limitações de cada abordagem, permitindo identificar o modelo mais adequado para integrar o sistema final. Esses resultados reforçam a importância de utilizar dados robustos e variados para testar sistemas de QA baseados em grafos de conhecimento.

Método	**Embedding do predicado**				
Modelo	all-mpnet -base-v2	multi-qa-mpnet -base-dot-v1	msmarco-bert -base-dot-v5	multi-qa-mpnet -base-cos-v1	multi-qa-distilbert -cos-v1
Pergunta #1					
Pergunta #2					
Pergunta #3					
Pergunta #4					
Pergunta #5					
Pergunta #6					
Pergunta #7					
Pergunta #8					
Pergunta #9					
Pergunta #10					
Pergunta #11					
Pergunta #12					
Pergunta #13					
Pergunta #14					
Pergunta #15					
Pergunta #16					
Pergunta #17					
Pergunta #18					
Pergunta #19					
Pergunta #20					

Tabela 7 – Tabela de resultados dos testes após expansão do grafo

5.3.4 Msmarco-bert-base-dot-v5

Com uma taxa de acerto exato de 45% (verde) e a menor taxa de erro total de apenas 10% (vermelho), o **msmarco-bert-base-dot-v5** destacou-se como o modelo mais eficiente entre os testados. Sua escolha como modelo final para o sistema foi motivada pela consistência com que identificou os caminhos corretos no grafo, mesmo em perguntas que apresentavam maior complexidade semântica. Esse desempenho pode ser atribuído à sua arquitetura otimizada para tarefas de recuperação de informações, utilizando *embeddings* ajustados para capturar nuances contextuais de maneira mais precisa.

Além disso, o modelo demonstrou maior resiliência às alterações no contexto do grafo, especialmente em perguntas que dependiam de múltiplas relações ou de conexões menos diretas entre entidades. Isso indica que o **msmarco-bert-base-dot-v5** é capaz de integrar informações contextuais de maneira eficiente, priorizando caminhos mais relevantes mesmo em cenários mais desafiadores (I., 2019). Essa robustez é essencial para sistemas

de QA baseados em grafos, pois reduz a dependência de dados perfeitamente estruturados, tornando o sistema mais adaptável e confiável em situações práticas.

5.3.5 Multi-qa-mpnet (base-dot-v1 e base-cos-v1)

Os modelos **multi-qa-mpnet-base-dot-v1** e **multi-qa-mpnet-base-cos-v1** também apresentaram resultados satisfatórios, com taxas de acerto total semelhantes ao modelo msmarco. No entanto, ambos apresentaram uma taxa de erro total de 20% (vermelho), o que indica uma maior vulnerabilidade em perguntas onde o contexto ou as conexões no grafo eram mais complexas. Apesar disso, esses modelos demonstraram potencial em cenários onde o foco principal não é exclusivamente a precisão, mas sim a escalabilidade ou o tempo de execução, já que seu desempenho computacional é geralmente superior em tarefas de similaridade vetorial.

Uma característica notável desses modelos é sua capacidade de lidar com questões mais diretas ou menos dependentes de múltiplos níveis de relações no grafo (T.-Y., 2020). Embora tenham sido descartados para o escopo atual do projeto devido ao menor desempenho em cenários complexos, eles poderiam ser reaproveitados em futuras aplicações do sistema, especialmente em contextos onde a simplicidade e velocidade de processamento sejam prioridades.

5.3.6 Multi-qa-distilbert-cos-v1

O modelo **multi-qa-distilbert-cos-v1** (T., 2019) apresentou resultados abaixo do esperado, com um desempenho inconsistente em perguntas que exigiam compreensão contextual mais profunda. Isso pode ser atribuído à sua arquitetura simplificada, baseada no DistilBERT, que embora mais eficiente em termos de tempo de execução, sacrifica parte da capacidade de capturar nuances semânticas detalhadas nas relações do grafo.

Além disso, o modelo apresentou dificuldades em perguntas que exigiam maior conectividade entre diferentes níveis do grafo, resultando em uma taxa de erro total considerável. Embora tenha potencial para aplicações que priorizem eficiência computacional, seu uso em sistemas de QA robustos e baseados em grafos de conhecimento é limitado, sendo, portanto, descartado para o presente projeto.

5.3.7 All-mpnet-base-v2

Por fim, o **all-mpnet-base-v2** (I., 2020), um modelo generalista, apresentou a maior taxa de erro total entre os modelos avaliados, com 30% (vermelho). Isso reflete sua falta de especialização para tarefas de QA em grafos de conhecimento, já que sua arquitetura foi projetada para cenários mais amplos de recuperação de informações. Embora tenha demonstrado bom desempenho em algumas perguntas mais simples, sua performance

caiu significativamente em questões mais complexas, que demandavam maior compreensão do contexto e relações do grafo.

Apesar disso, o **all-mpnet-base-v2** pode ser útil em aplicações onde o foco seja generalidade e flexibilidade, como sistemas que integram múltiplas fontes de dados ou realizam tarefas menos específicas. No entanto, para o escopo deste projeto, ele não atende aos requisitos de precisão e confiabilidade necessários, sendo descartado como opção para o modelo final.

5.3.8 Testes de parâmetros

Os testes finais de parâmetros representam uma etapa crucial no desenvolvimento e validação do sistema, pois permitem ajustar as configurações de maneira a maximizar a eficiência e a precisão do algoritmo. O objetivo principal desses testes é garantir que o sistema opere de forma robusta e eficiente, fornecendo respostas precisas em tempo hábil, mesmo quando submetido a grafos mais complexos e perguntas desafiadoras. Além disso, a análise de parâmetros como o número de saltos e o fator de ramificação é essencial para determinar os limites de desempenho do sistema e para identificar os melhores compromissos entre custo computacional e qualidade das respostas.

A metodologia adotada para esses testes baseia-se em cenários cuidadosamente projetados que simulam diferentes condições de uso. Para cada parâmetro testado, serão avaliados tanto o tempo médio de execução por pergunta quanto a precisão das respostas retornadas. A escolha de métricas claras, como a posição da resposta correta no conjunto retornado e o percentual de falhas, permite realizar comparações consistentes entre as configurações. Esses testes são fundamentais não apenas para refinar o desempenho do sistema, mas também para validar o design do modelo e sua aplicabilidade em contextos reais de perguntas e respostas baseados em grafos de conhecimento.

Análise dos Testes com Diferentes Números de Saltos

Os testes realizados visaram determinar o impacto do número de saltos no desempenho do sistema, tanto em termos de acurácia quanto de tempo de execução. Cada valor de hops foi avaliado utilizando o modelo **msmarco-bert-base-dot-v5**, com um *beam_size* fixo de 10. Para cada configuração, coletamos a posição da resposta correta no conjunto retornado e o tempo médio de execução por pergunta. Segue os resultados abaixo:

Número de hops	1	2	3	4
Pergunta #1				
Pergunta #2				
Pergunta #3				
Pergunta #4				
Pergunta #5				
Pergunta #6				
Pergunta #7				
Pergunta #8				
Pergunta #9				
Pergunta #10				
Pergunta #11				
Pergunta #12				
Pergunta #13				
Pergunta #14				
Pergunta #15				
Pergunta #16				
Pergunta #17				
Pergunta #18				
Pergunta #19				
Pergunta #20				

Tabela 8 – Tabela de resultados dos testes de variação do número de saltos

Os resultados mostram que, com 1 salto, o sistema apresenta limitações severas na busca, retornando a resposta correta em apenas 30% dos casos (6 respostas corretas na primeira posição) e falhando completamente em 45% dos cenários. Apesar do tempo médio de execução ser o mais baixo, de 2,9 segundos, a baixa acurácia torna essa configuração inadequada para nosso objetivo.

Com 2 saltos, o desempenho melhora significativamente, alcançando a melhor acurácia geral. O sistema consegue retornar a resposta correta no topo em 45% dos casos e entre os cinco primeiros em 90%, com uma taxa de falha de apenas 10%. O tempo médio de execução aumenta para 7,35 segundos, ainda dentro de um intervalo aceitável. Essa configuração balanceia eficiência e desempenho, sendo a escolhida para o sistema final.

Para 3 saltos, a acurácia diminui levemente, com 35% de falhas e uma maior taxa de caminhos incorretos classificados. Embora a capacidade do sistema de explorar relações mais profundas no grafo seja ampliada, o tempo médio de execução aumenta para 15,95 segundos, o que compromete a escalabilidade.

Por fim, com 4 saltos, a acurácia é prejudicada ainda mais, com 45% de falhas e um tempo médio de 27,65 segundos. Este aumento drástico no tempo de execução, combinado com a menor precisão, demonstra que aumentar o número de *hops* além de

2 não é vantajoso para o contexto analisado. O gráfico acima ilustra claramente como o tempo de execução cresce de forma não linear com o número de *hops*.

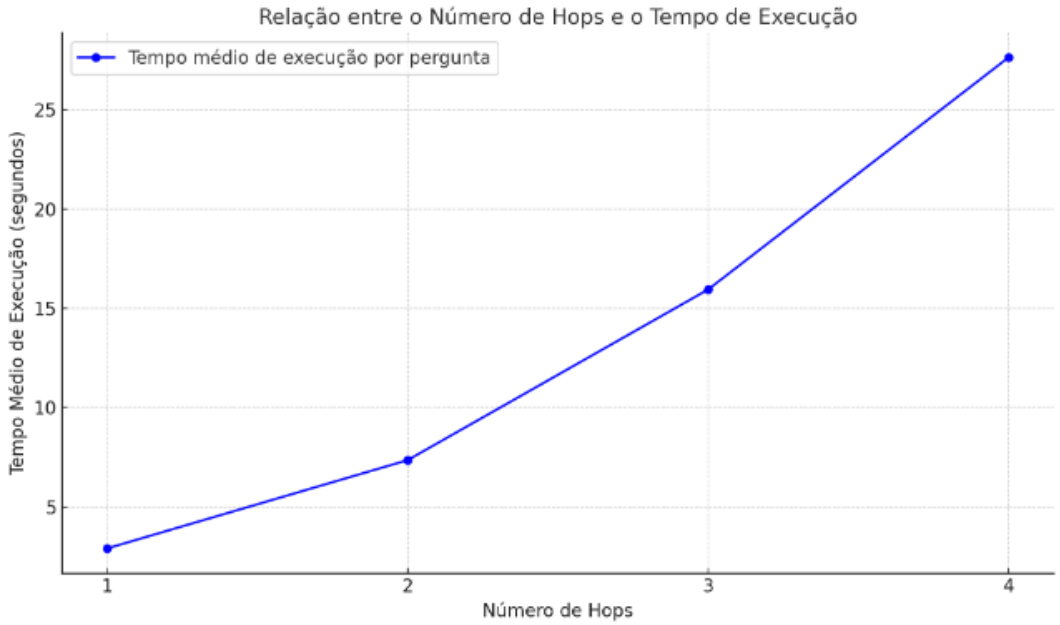

Figura 11 – Gráfico de tempo de execução por saltos

Análise de variação do fator de ramificação(*beam size*)

O parâmetro fator de ramificação é um dos elementos mais críticos no processo de busca em grafos, pois determina o número de caminhos que serão mantidos após cada expansão (salto), enquanto os demais são podados com base nos critérios de relevância do sistema. Esse fator influencia diretamente o equilíbrio entre desempenho e custo computacional, uma vez que um *beam size* pequeno pode limitar a exploração de caminhos potencialmente corretos, enquanto um *beam size* grande aumenta significativamente o tempo de processamento devido ao maior número de ramificações analisadas. Realizar testes com diferentes valores de fator de ramificação é essencial para identificar a configuração que proporciona a melhor combinação entre taxa de acerto, eficiência e escalabilidade, garantindo que o sistema seja capaz de responder perguntas com precisão em um tempo adequado, mesmo em cenários mais complexos. Segue os resultados abaixo:

Fator de Ramificação	1	2	3	4	5	6	7	8	9	10
Pergunta #1										
Pergunta #2										
Pergunta #3										
Pergunta #4										
Pergunta #5										
Pergunta #6										
Pergunta #7										
Pergunta #8										
Pergunta #9										
Pergunta #10										
Pergunta #11										
Pergunta #12										
Pergunta #13										
Pergunta #14										
Pergunta #15										
Pergunta #16										
Pergunta #17										
Pergunta #18										
Pergunta #19										
Pergunta #20										

Tabela 9 – Tabela de resultados dos testes de fator de ramificação

O teste com *beam size* = 1 apresentou os piores resultados dentre todas as configurações testadas, com 6 respostas corretas (valores 1 no vetor), 9 erros totais (valores -1) e apenas 5 caminhos corretos retornados no top 5. Essa baixa performance pode ser explicada pela natureza restritiva desse parâmetro, que limita o fator de ramificação a apenas um caminho por salto. Essa abordagem minimiza drasticamente a exploração do grafo, resultando em uma perda significativa de informações relevantes que poderiam auxiliar na escolha do caminho correto. Em contrapartida, o tempo de execução foi o mais rápido, com uma média de apenas 2,6 segundos por pergunta, indicando que essa configuração prioriza eficiência computacional em detrimento da precisão, tornando-a inadequada para aplicações que demandam maior acurácia.

A configuração com *beam size* = 2 destacou-se por apresentar um equilíbrio surpreendente entre desempenho e eficiência. Foram retornadas 8 respostas corretas, 4 erros totais e 8 caminhos corretos no top 5, superando significativamente o desempenho do *beam size* 1. Essa melhoria pode ser atribuída à maior exploração do grafo, o que permite que o sistema capture mais contextos úteis sem aumentar significativamente o tempo de execução, que foi de 3,15 segundos por pergunta. No entanto, apesar dos resultados promissores, essa configuração pode enfrentar desafios de escalabilidade em grafos maiores ou em cenários com maior complexidade, uma vez que o aumento do *beam size* tende a

impactar a complexidade de busca à resposta correta. Por esse motivo, essa configuração, embora interessante, não foi escolhida como ideal.

As configurações com *beam size* = 3, 5 e 6 apresentaram resultados similares, com 8 respostas corretas e 4 erros totais, além de manterem 8 caminhos corretos no top 5. A principal diferença entre elas está no tempo médio de execução, com 3,7, 5,35 e 5,55 segundos por pergunta, respectivamente. Embora esses valores demonstrem que o aumento do *beam size* não resultou em ganhos significativos na acurácia, ele impactou diretamente a eficiência temporal. Isso indica que, após certo limite, o aumento no número de ramos explorados não traz benefícios expressivos no desempenho do modelo, tornando essas configurações menos ideais para cenários que demandam alta eficiência.

A configuração com *beam size* = 4 destacou-se por apresentar uma das melhores combinações entre precisão e tempo de execução. Foram retornadas 9 respostas corretas, 3 erros totais e 8 caminhos corretos no top 5, com um tempo médio de 4,2 segundos por pergunta. Esse desempenho pode ser atribuído à maior diversidade de caminhos explorados, sem que o custo computacional ultrapasse os limites aceitáveis. Dada essa combinação de fatores, essa configuração é uma excelente escolha para aplicações que demandam tanto acurácia quanto eficiência, embora não tenha sido a opção final.

Caso o sistema precisasse ser utilizado por uma grande quantidade de usuários em tempo real, como no caso de um *chatbot open source*, provavelmente este fator de ramificação seria a escolha correta a ser considerada, já que equilibra perfeitamente um bom desempenho temporal com uma ótima acurácia, porém como o projeto não irá lidar com uma alta demanda de requisições, a escolha feita foi por um fator de ramificação com maior acurácia possível.

Os *beam sizes* de 7 a 10 apresentaram o maior número de respostas corretas, com 9 acertos, apenas 2 erros totais e 9 caminhos corretos no top 5. Esses resultados refletem o aumento significativo na exploração do grafo, permitindo que mais informações contextuais fossem capturadas. No entanto, o custo computacional também aumentou consideravelmente, com tempos de execução variando de 5,95 a 7,35 segundos por pergunta, sendo o *beam size* 10 o mais custoso. Dentre essas configurações, o *beam size* 7 foi escolhido como ideal por apresentar o melhor equilíbrio entre precisão e tempo de execução, atingindo alta acurácia com um custo computacional ainda controlado.

A escolha do fator de ramificação 7 reflete um compromisso estratégico entre desempenho temporal e precisão, adequado para cenários onde a robustez do sistema é essencial, mas a eficiência computacional também deve ser considerada.

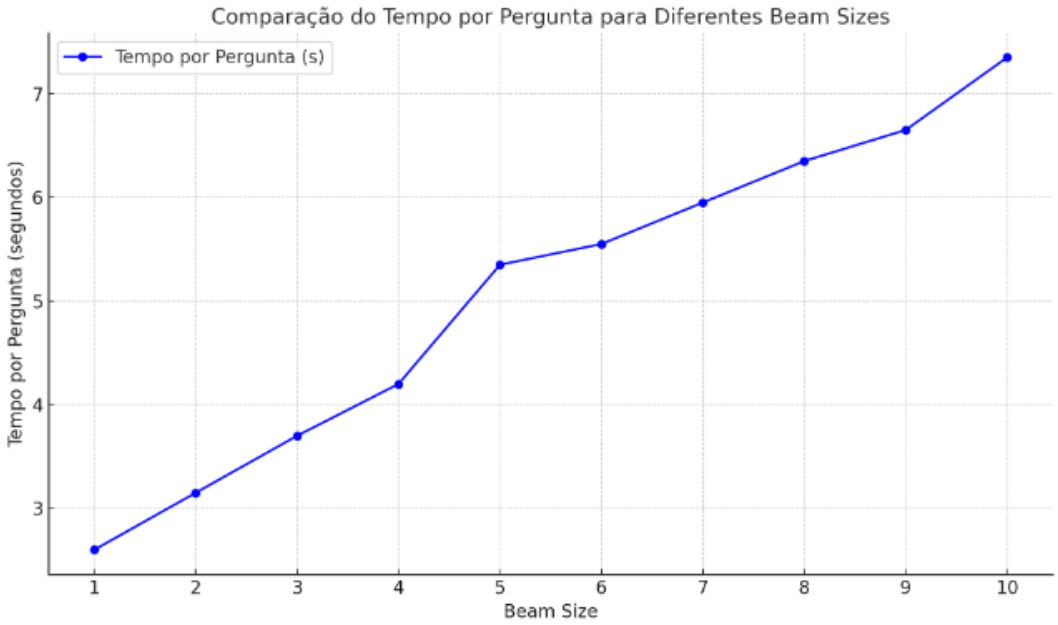

Figura 12 – Gráfico de tempo de execução por fator de ramificação

5.4 Desenvolvimento e testes da interface para demonstração

Por fim, foi feito o desenvolvimento da interface para a demonstração do nosso projeto, que compreende o desenvolvimento de um Frontend e um Backend, o qual pode ser encontrado no repositório público do Github, além da integração de ambos para demonstrar o funcionamento do algoritmo de Multi-Hop Question Answering em um grafo de conhecimento sobre a Amazônia Azul. O objetivo desta interface foi permitir a visualização dinâmica do grafo, a interação com perguntas e respostas, e a classificação das melhores respostas de acordo com os resultados do modelo.

Desenvolvimento do Backend

O backend foi implementado em Flask, uma escolha que permitiu criar uma API leve e modular. A principal funcionalidade foi fornecer suporte ao algoritmo de Multi-Hop Question Answering, utilizando modelos pré-treinados baseados em BERT (como BERT Uncased e Big BERT) para analisar perguntas e respostas e determinar o caminho mais adequado dentro do grafo.

O Backend era composto das seguintes funcionalidades:

1. Percorrer Grafos: Como comentado na seção de desenvolvimento do algoritmo, o Backend era responsável por realizar a execução desses algoritmos e avaliar a melhor resposta conforme é enviado uma pergunta a partir do Frontend. Junto disso, era necessário também que o Backend integrasse com os modelos pré-treinados do BERT o que desacelerava um pouco a responsividade do mesmo.

2. API REST: Os endpoints permitem ao Frontend enviar perguntas, recuperar o caminho sugerido no grafo e obter respostas classificadas, além de poder várias alguns parâmetros caso deseje como o número de saltos e o beam size.

Desenvolvimento do Frontend

O Frontend foi desenvolvido em React com TypeScript, utilizando a biblioteca Material-UI para o design e a Reagraph para a visualização dinâmica do grafo. O foco foi criar uma interface responsiva e interativa, que integra visualização do grafo, perguntas do usuário e classificação de respostas.

Os principais componentes são:

1. GraphCanvas: Este componente exibe o grafo e permite interações, como o clique nos nós para explorar detalhes. Ele utiliza a biblioteca Reagraph, que oferece suporte a grafos dinâmicos e layouts ajustáveis.

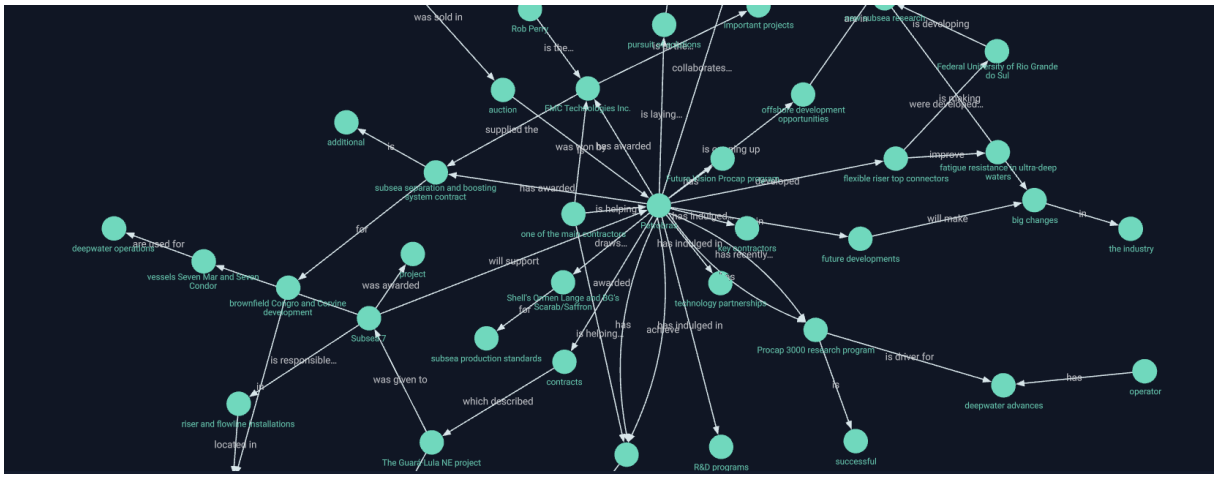

Figura 13 – Visualizado de grafos do Frontend

2. ChatBox: Permite ao usuário enviar perguntas ao backend e exibe as respostas de forma interativa. Um indicador de carregamento foi implementado para exibir quando uma mensagem está sendo processada e também o usuário pode definir quantas respostas deseja receber.

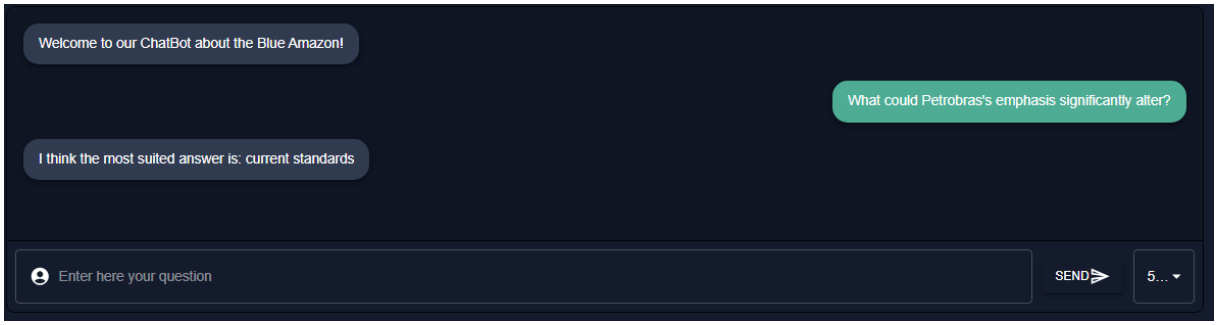

Figura 14 – Chat de QA com o servidor no Frontend

3. AnswersRanking: Apresenta um ranking das respostas mais relevantes, permitindo ao usuário selecionar as respostas e visualizar dinamicamente no GraphCanvas. Este componente utiliza listas com checkboxes, facilitando a seleção e verificação de quais são todas as melhores n respostas definido pelo usuário.

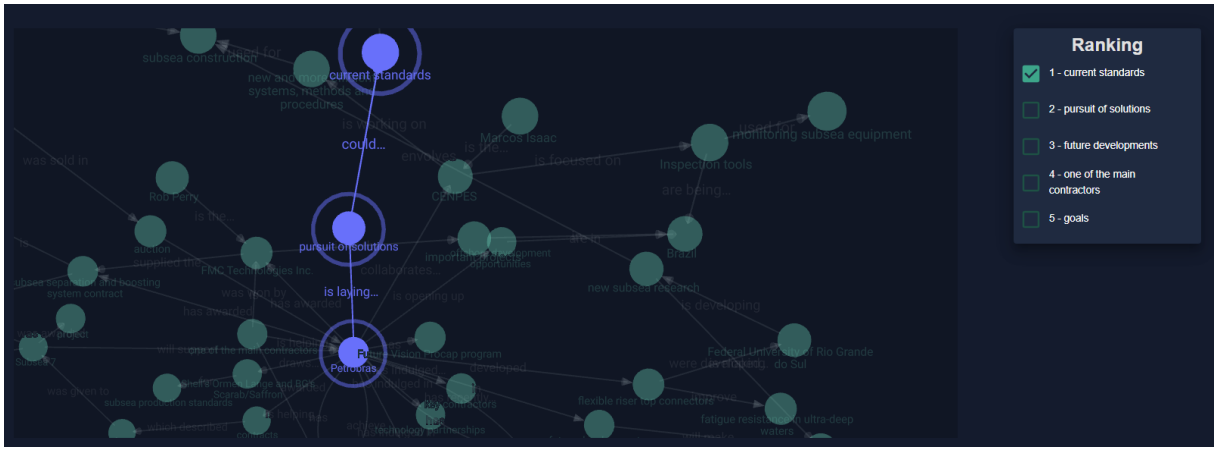

Figura 15 – Classificação de respostas e seletor para visualização no grafo

Integração e testes do Backend e Frontend

A comunicação entre Frontend e Backend foi realizada através de requisições HTTP POST. O uso de bibliotecas nativas do próprio React simplificou as chamadas às APIs do Flask, enquanto as respostas foram renderizadas dinamicamente no React. Junto disso, foi necessário a instalação de CORS Policy no Backend para poder realizar os testes de integração dentro de uma máquina local. Por fim, foram realizados diversos testes para verificar a precisão dos caminhos sugeridos pelo Backend em relação às perguntas fornecidas, avaliar a fluidez e a simplicidade da interface, especialmente na exibição e manipulação do grafo e garantir que o ranking das respostas refletisse corretamente a análise feita pelo Backend e é exibido de forma correta na visualização do grafo.

6 Considerações Finais

Nesta seção, serão apresentadas as conclusões obtidas, discutindo a relevância dos resultados e a maneira como eles atendem aos objetivos iniciais do trabalho. Além disso, exploraremos as contribuições específicas do projeto, tanto em termos técnicos quanto aplicados, destacando soluções implementadas e o diferencial do sistema em diferentes contextos.

Por fim, serão abordadas perspectivas de continuidade, sugerindo caminhos para expandir o alcance e a eficiência do sistema, seja por meio de novas abordagens tecnológicas, como o uso de redes neurais mais avançadas, ou pela aplicação prática em ambientes reais, integrando o projeto a sistemas escaláveis e interativos.

6.1 Conclusões do Projeto de Formatura

Com o avanço das técnicas de IA e PLN, novas possibilidades emergem para a organização, disseminação e utilização do conhecimento em domínios de alta relevância estratégica. Este trabalho, inserido no contexto de aplicação da IA, tem como foco a Amazônia Azul, uma região marítima de enorme importância ambiental e econômica para o Brasil, cuja relevância é equiparada à da Amazônia Verde. Apesar de sua significância, o acesso a informações estruturadas sobre a Amazônia Azul ainda é limitado, dificultando a conscientização pública e o desenvolvimento de pesquisas mais aprofundadas.

O projeto propõe a criação de um sistema de QA baseado em grafos de conhecimento, com a capacidade de realizar raciocínios em múltiplos saltos de contexto. Essa abordagem permite que o sistema conecte informações dispersas para responder a perguntas complexas, como os impactos ambientais da exploração de petróleo ou o papel dos manguezais na preservação da região. A aplicação desse sistema visa preencher lacunas críticas de informação, integrando dados sobre a Amazônia Azul de maneira acessível e estruturada, e potencializando seu uso em contextos educativos e científicos.

Para isso, o trabalho explora avanços recentes em IA e PLN, como modelos pré-treinados baseados em *transformers* (e.g., BERT) e algoritmos especializados na navegação e inferência em grafos de conhecimento. Além de investigar técnicas já consolidadas, o projeto inova ao adaptar e aplicar essas ferramentas ao domínio específico da Amazônia Azul, criando um sistema capaz de gerar respostas relevantes e precisas a partir de um grafo construído com dados sobre atividades e avanços tecnológicos relacionados à Petrobras e à região marítima brasileira.

O desenvolvimento do sistema foi estruturado em etapas que incluem revisão

bibliográfica, implementação de algoritmos, testes rigorosos e integração com uma interface funcional. O grafo de conhecimento, que serve de base para o sistema, foi criado a partir de textos descritivos sobre a Petrobras, inicialmente com 28 relações e 10 perguntas, posteriormente ampliados para aumentar a complexidade e o alcance do modelo. A integração com um frontend interativo permite que o sistema seja acessível a diferentes usuários, desde educadores até pesquisadores.

Com base nos objetivos estabelecidos inicialmente, observa-se que os requisitos definidos no início do projeto foram, de modo geral, plenamente atendidos. Foi desenvolvido um algoritmo de saltos de contexto para QA em grafos de conhecimento que apresentou uma taxa de acerto satisfatória e eficiência temporal adequada. Além disso, foi implementado com sucesso um sistema de software que facilita o uso do sistema pelos usuários.

Após a conclusão deste trabalho de final de curso, constatou-se que o sistema desenvolvido tem potencial para aplicação em contextos reais, especialmente em cenários onde os modelos de linguagem ampla (LLMs) atuais ainda apresentam limitações, como em domínios específicos com grandes volumes de dados esparsos. Conforme discutido no início deste documento, este projeto pode contribuir para ampliar a acessibilidade e a disponibilidade de conhecimentos essenciais.

6.2 Contribuições

O ponto de partida deste trabalho foi fundamentado no algoritmo do Multihop KG (YU; HE; GLASS, 2021), que serviu como base teórica e prática para o desenvolvimento inicial. A partir dessa base, o grupo contribuiu significativamente ao explorar diferentes métodos para a geração de *embeddings* tanto para os grafos quanto para as perguntas. Essa exploração incluiu a experimentação com diversos modelos de PLN, o que permitiu não apenas ampliar o entendimento do comportamento dos algoritmos, mas também refinar a precisão e a acurácia do sistema desenvolvido. Essas inovações foram essenciais para superar as limitações observadas no modelo original e adaptar o sistema aos objetivos propostos.

Outro ponto de contribuição relevante foi o desenvolvimento de uma interface gráfica de fácil utilização, projetada para permitir que os usuários interajam de forma intuitiva com o sistema. Essa interface, também de autoria da equipe, permite a entrada de perguntas e a visualização dinâmica do grafo de conhecimento, facilitando a compreensão e a navegação pelos dados apresentados. Para sua implementação, foram combinados diferentes *frameworks* e componentes tecnológicos, escolhidos estrategicamente para garantir uma experiência amigável ao usuário, sem comprometer a performance do sistema.

Por fim, as contribuições deste trabalho vão além do aspecto técnico, pois abordam a integração de soluções inovadoras em PLN e visualização com aplicações práticas em

grafos de conhecimento. O sistema resultante reflete o esforço conjunto da equipe em unir teoria, experimentação e design centrado no usuário, resultando em um produto alinhado aos desafios propostos no início do projeto.

6.3 Perspectivas de Continuidade

Uma das possibilidades de continuidade é o uso de LSTMs para identificação do tipo de pergunta, as *Long Short-Term Memory* (LSTM) *networks* são um tipo de rede neural recorrente (RNN) projetada para lidar com dados sequenciais, como texto ou séries temporais, capturando dependências de longo prazo. Diferentemente das RNNs tradicionais, as LSTMs utilizam células de memória e portas de controle para decidir quais informações manter ou descartar, permitindo que reconheçam padrões complexos em sequências extensas. Essa capacidade as torna ideais para tarefas como classificação de texto, tradução automática e análise de linguagem natural.

Integrar LSTMs ao projeto pode enriquecer significativamente seu desempenho ao melhorar a interpretação das perguntas dos usuários no sistema de QA baseado em grafos. O uso de LSTMs pode ajudar a capturar relações semânticas mais profundas nas perguntas, especialmente quando estas envolvem saltos de contexto ou linguagem ambígua. Modelos como o Stacked LSTM podem ser utilizados para explorar camadas hierárquicas de abstração no texto, enquanto variações como Bi-LSTM podem oferecer uma visão bidirecional sobre a sequência, tornando a análise mais robusta. Além disso, a aplicação de LSTMs alinhada a *embeddings* previamente desenvolvidos no projeto poderia melhorar a precisão da correspondência entre perguntas e relações no grafo, fortalecendo o potencial do sistema em contextos mais desafiadores.

Uma outra possibilidade de continuidade para este projeto seria a implementação do sistema em um ambiente de servidor, permitindo sua utilização por clientes reais. Essa etapa envolveria a adaptação do componente desenvolvido para uma arquitetura de backend escalável, integrando-o a um servidor que pudesse processar múltiplas solicitações simultaneamente. Esse processo possibilitaria a avaliação do sistema em condições de uso real, onde o volume de dados e a diversidade de consultas poderiam evidenciar novos desafios, como a necessidade de ajustes na eficiência computacional e na capacidade de lidar com diferentes padrões de entrada.

Além disso, a interação com clientes reais permitiria uma análise prática do desempenho do sistema e a identificação de demandas específicas do usuário, como melhorias na interface ou na personalização das respostas. A coleta de métricas de uso e feedback contínuo forneceria insumos valiosos para refinar a acurácia do sistema, garantindo que ele atendesse de forma robusta aos requisitos do domínio de aplicação. Essa etapa, embora desafiadora, seria viável para uma equipe reduzida e alinhada com a proposta de

escalabilidade e aplicabilidade prática deste trabalho.

Referências

A., I. D. E. B. S. *Modern Information Retrieval: A Brief Overview*. 2001. Citado na página 34.

ALAMMAR, J. *The Illustrated BERT, ELMo, and co. (How NLP Cracked Transfer Learning)*. 2018. Disponível em: <http://jalammar.github.io/illustrated-bert/>. Citado na página 21.

FRANCO, P. . P. . C. . J. G. M. . B. . A. A. *The BLue Amazon Brain (BLAB): a modular architecture of services about the Brazilian maritime territory (2022)*. 2022. Repositório do BLAB. Disponível em: <https://repositorio.usp.br/item/003144894>. Acesso em: 12 nov 2024. Citado na página 9.

I., R. N. . G. *Sentence-BERT: Sentence embeddings using Siamese BERT-networks*. 2019. Disponível em: <https://arxiv.org/abs/1908.10084>. Citado na página 51.

I., R. N. . G. *Making Monolingual Sentence Embeddings Multilingual using Knowledge Distillation*. 2020. Disponível em: <https://arxiv.org/abs/2004.09813>. Citado na página 52.

M. JAY PRAKASH, P. K. S. S. C. *Question Answering Over Knowledge Graphs Using Bert Based Relation Mapping*. 2022. Disponível em: <https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4102838>. Citado na página 49.

SINGHAL, A. *Introducing the Knowledge Graph: things, not strings*. 2012. 2020-11-13. Disponível em: <https://www.blog.google/products/search/introducing-knowledge-graph-things-not/>. Citado na página 16.

SULLIVAN, D. *A reintroduction to our Knowledge Graph and knowledge panels*. 2020. Disponível em: <https://www.blog.google/products/search/introducing-knowledge-graph-things-not/>. Citado na página 16.

T., S. V. . D. L. . C. J. . W. *DistilBERT, a distilled version of BERT: smaller, faster, cheaper, and lighter*. 2019. Disponível em: <https://arxiv.org/abs/1910.01108>. Citado na página 52.

T.-Y., S. K. . T. X. . Q. T. . L. J. . L. *MPNet: Masked and Permuted Pre-training for Language Understanding. Advances in Neural Information Processing Systems (NeurIPS), 33, 16857–16867*. 2020. Disponível em: <https://arxiv.org/abs/2004.09297>. Citado na página 52.

WIKIPEDIA. *Águas jurisdicionais brasileiras*. 2024. Wiki da Amazônia Azul. Disponível em: <https://pt.wikipedia.org/wiki/%C3%81guas_jurisdicionais_brasileiras>. Acesso em: 12 nov 2024. Citado na página 9.

YU, S.; HE, T.; GLASS, J. *AutoKG: Constructing Virtual Knowledge Graphs from Unstructured Documents for Question Answering*. 2021. Citado 5 vezes nas páginas 2, 3, 28, 30 e 62.

Apêndices

.1 routes.py - Código de endpoints do Backend

```python
# app/routes.py

from flask import Blueprint, request, jsonify
from .services.query_handler import process_query

# Blueprint para os endpoints
bp = Blueprint('main', __name__)

@bp.route('/health', methods=['GET'])
def health_check():
    return jsonify({"status": "online"}), 200

@bp.route('/info', methods=['GET'])
def info():
    return jsonify({
        "model": "tcc-multi-hop-question-answering",
        "version": "1.0.0"
    }), 200

@bp.route('/process_query', methods=['POST'])
def handle_query():
    data = request.get_json()
    query = data.get("query")
    if not query:
        return jsonify({"error": "Parâmetro 'query' é obrigatório
            "}), 400

    result = process_query(query)
    return jsonify(result), 200

@bp.route('/process_bulk', methods=['POST'])
def handle_bulk_query():
    data = request.get_json()
    queries = data.get("queries", [])
    if not queries:
        return jsonify({"error": "Parâmetro 'queries' é obrigató
            rio e deve ser uma lista"}), 400

    results = [process_query(query) for query in queries]
    return jsonify(results), 200
```

```
39
40 @bp.route('/process_query_first', methods=['POST'])
41 def handle_query_first():
42     data = request.get_json()
43     query = data.get("query")
44     if not query:
45         return jsonify({"error": "Parâmetro 'query' é obrigatório
           "}), 400
46
47     result = process_query(query)
48     print(result)
49     first_item = result[0] if result else []
50     return jsonify(first_item), 200
51
52 @bp.route('/process_query_n', methods=['POST'])
53 def handle_query_n():
54     data = request.get_json()
55     query = data.get("query")
56     k = data.get("k")
57     number_of_hops = data.get("number_of_hops")
58
59     if not query:
60         return jsonify({"error": "Parâmetro 'query' é obrigatório
           "}), 400
61     if k is None:
62         return jsonify({"error": "Parâmetro 'k' é obrigatório"}),
            400
63     if number_of_hops is None:
64         return jsonify({"error": "Parâmetro 'number_of_hops' é
           obrigatório"}), 400
65
66     result = process_query(query,k,number_of_hops)
67
68     result_k = result[:k] if k > 0 else []
69     return jsonify(result_k), 200
```

.2 query_handler.py - Código do algoritmo

```
1 import os
2 import numpy as np
3 import pandas as pd
```

```
4  from sklearn.metrics.pairwise import cosine_similarity
5  from sentence_transformers import SentenceTransformer
6
7  model = SentenceTransformer('Msmarco-bert-base-dot-v5')
8
9
10 #Função que seleciona as entidades da pergunta(tamanho > 3)
11 def get_surface_entities(query, graph):
12   # Define as colunas em que deseja fazer a busca
13   search_columns = ['Subject', 'Object']
14   # Cria uma lista de palavras-chave a partir da pergunta
15   key_word = query.split()
16   key_words = []
17   for element in key_word:
18     if len(element) > 3:
19       key_words.append(element)
20   # Faz a busca no grafo pelas entidades
21   results = []
22   for search in search_columns:
23     for index, row in graph.iterrows():
24         for word in key_words:
25             if word.lower() in str(row[search]).lower() or str(
                   row[search]).lower() in  word.lower():
26                 results.append(row[search])
27
28   results = list(set(results))
29   return results
30
31 def get_link(surface_entity, triples):
32   # Filtramos as triplas pegando apenas as que possuem as
        entidades como objeto ou sujeito
33   filtered_triples_subject = [triple for triple in triples if (
        triple[0] == surface_entity )]
34   filtered_triples_object = [triple[::-1] for triple in triples
        if (triple[2] == surface_entity)]
35   filtered_triples_subject += filtered_triples_object
36   # Implementação para obter o link de uma entidade de superfície
37   threshold_triples = set(filtered_triples_subject)
38   return threshold_triples
39
40 #Essa função retorna todas as triplas que temos no grafo
41 def get_triples(graph):
```

```
42    #Leitura do grafo e seleção das colunas pertinentes
43    triples_dataset = graph[['Subject','Relation','Object']]
44    #Criação das triplas
45    triples=[]
46    for index, row in triples_dataset.iterrows():
47        triples.append((row['Subject'], row['Relation'], row['
          Object']))
48    return triples
49
50  # Define uma função para calcular a similaridade do cosseno
51  def calc_cosine_similarity_predicate(path_enc, query_enc):
52    sentences = []
53    if isinstance(query_enc,list):
54      delimitador = "␣"
55      query_enc = delimitador.join(query_enc)
56    sentences.append(query_enc)
57    sentences.append("␣".join(path_enc[1::2]))
58    sentence_embeddings = model.encode(sentences)
59    arr = cosine_similarity(sentence_embeddings[0].reshape(1,-1),
        sentence_embeddings[1].reshape(1,-1))
60    return arr[0][0]
61
62  # Define uma função para podar P com base na similaridade do
     cosseno com um tamanho de feixe B
63  def prune_paths(P, qenc, beam_size):
64    # Implementação para podar P com base na similaridade do
       cosseno com um tamanho de feixe B
65    # Caso o total de caminhos B não tenha sido ultrapassado,
       retorna o set de caminhos sem alterações
66    P = [list(t) for t in set(tuple(path) for path in P)]
67    if len(P) <= beam_size:
68      return P
69    # Caso contrário faz a poda, armazenando as cosseno-
       similaridades em um dicionário
70    else:
71      cossimilarity = []
72      for path in P:
73        similarity =  (path,qenc)
74        cossimilarity.append((path, similarity))
75      cossimilarity = sorted(cossimilarity, key=lambda x: x[1],
          reverse=True)
```

```
76      # Procedimento de poda,enquanto o número de feixes não for o
            desejado remove a menor cosseno-similaridade do Set
77      while len(P) > beam_size:
78          aux = cossimilarity[-1]
79          P.remove(aux[0])
80          cossimilarity.pop()
81
82    return P
83
84  def get_anwser(k, beam_size, number_of_hops,
        seed_surface_entities, paths, query, triples):
85    #Fazemos uma iteração para cada número pré definido de hops
86    for hop in range(number_of_hops):
87      # Se não temos caminhos, para cada entidade de superfície
            seed i, criamos um caminho novo
88      if not(paths):
89        for seed in seed_surface_entities:
90          for connected_surface_entity in get_link(seed, triples):
91            path = [ connected_surface_entity[0],
                connected_surface_entity[1], connected_surface_entity
                [2]]
92            paths.append(path)
93        paths = prune_paths(paths, query, beam_size)
94      else:
95        new_paths = []
96        # Para cada caminho p em P, faça o seguinte:
97        for path in paths:
98            # Obtenha a última entidade de superfície em p
99            last_surface_entity = path[-1]
100           # Para cada entidade de superfície conectada a
                last_surface_entity, pega os links:
101           for connected_surface_entity in get_link(
                last_surface_entity, triples):
102             if(connected_surface_entity[2] != path[-3]):
103                 new_path = path + [ connected_surface_entity[1],
                        connected_surface_entity[2]]
104                 new_paths.append(new_path)
105       paths += prune_paths(new_paths, query, beam_size)
106
107   # Retorna o conjunto de caminhos final
108   top_k = sorted(paths, key=lambda path:
        calc_cosine_similarity_predicate(path, query), reverse=True)
```

```
    [:k]
  return top_k

def process_query(query, k=5, number_of_hops=2):
  grafo_path = os.path.join(os.path.dirname(__file__), '../data/
    grafo.csv')
  grafo = pd.read_csv(grafo_path)
  triples = get_triples(grafo)
  paths = []
  beam_size = 7
  seed_surface_entities = get_surface_entities(query, grafo)
  return get_anwser(k, beam_size, number_of_hops,
    seed_surface_entities, paths, query, triples)
```

.3 Código da interface principal do Frontend

```
import { Box, Grid2, useTheme } from '@mui/material';
import { useEffect, useState } from 'react';
import { GraphCanvas, GraphNode, GraphEdge } from 'reagraph';
import Header from '../../components/header';
import { csvParser } from '../../utils/csvParser';
import AnswersRanking from '../../components/ranking';
import ChatBox from '../../components/chat';
import { Answers, Message } from '../../interfaces';
import { getGraphCanvaTheme } from '../../theme';

const GraphVisualizer = () => {
  const [nodes, setNodes] = useState<GraphNode[]>([]);
  const [edges, setEdges] = useState<GraphEdge[]>([]);
  const [highlightedPath, setHighlightedPath] = useState<string
    []>([]);
  const [uniqueNodes, setUniqueNodes] = useState<Map<string,
    GraphNode>>(
    new Map<string, GraphNode>(),
  );
  const [messages] = useState<Message[]>([
    { sender: 'server', content: 'Welcome to our ChatBot about
      the Blue Amazon!' },
  ]);
  const [answers, setAnswers] = useState<Answers[]>([]);
  const theme = useTheme();
```

```
23    const canvaTheme = getGraphCanvaTheme(theme.palette.mode);
24
25    useEffect(() => {
26      csvParser()
27        .then((result) => {
28          setNodes(result.nodes);
29          setEdges(result.edges);
30          setUniqueNodes(result.uniqueNodes);
31        })
32        .catch((error) => console.error('Erro ao carregar nodes e
            edges:', error));
33    }, []);
34
35    return (
36      <>
37        <Box sx={{ mr: '20px', ml: '20px' }}>
38          <Box display="flex" justifyContent="space-between"
              alignItems>
39            <Header
40              title={'Knowledge Graphs'}
41              subtitle={'Multi-Hop Question Answering with
                  Knowledge Graphs'}
42            />
43          </Box>
44        </Box>
45        <Grid2 container spacing={2} sx={{ mr: '50px', ml: '50px',
             mt: '10px', mb: '10px' }}>
46          <Grid2 size={10} height="50vh">
47            <Box position="absolute" width="65vw" height="50%">
48              <GraphCanvas
49                theme={canvaTheme}
50                nodes={nodes}
51                edges={edges}
52                selections={highlightedPath}
53                layoutType="forceDirected2d"
54                cameraMode="pan"
55                actives={highlightedPath}
56                labelType="all"
57                edgeLabelPosition="inline"
58                onNodeClick={(node) => console.log('Nó clicado:',
                    node)}
59                draggable
```

```
60              />
61            </Box>
62          </Grid2>
63          <Grid2 size={2}>
64            <AnswersRanking
65              answers={answers}
66              setHighlightedPath={setHighlightedPath}
67              setAnswers={setAnswers}
68            />
69          </Grid2>
70          <Grid2 size={9}>
71            <ChatBox
72              messages={messages}
73              uniqueNodes={uniqueNodes}
74              setHighlightedPath={setHighlightedPath}
75              setAnswers={setAnswers}
76            />
77          </Grid2>
78        </Grid2>
79      </>
80    );
81  };
82
83  export default GraphVisualizer;
```