UNIVERSIDADE DO ESTADO DO RIO GRANDE DO NORTE - UERN
FACULDADE DE CIÊNCIAS EXATAS E NATURAIS – FANAT
DEPARTAMENTO DE INFORMÁTICA – DI
CURSO DE CIÊNCIA DA COMPUTAÇÃO

CLAUDIVAN BARRETO DA SILVA

UM MODELO COMPUTACIONAL PARA INTEGRAÇÃO DE PROBLEMAS DE OTIMIZAÇÃO UTILIZANDO BANCO DE DADOS ORIENTADOS A GRAFOS

MOSSORÓ - RN

2017

CLAUDIVAN BARRETO DA SILVA

UM MODELO COMPUTACIONAL PARA INTEGRAÇÃO DE PROBLEMAS DE OTIMIZAÇÃO UTILIZANDO BANCO DE DADOS ORIENTADOS A GRAFOS

Monografia apresentada à Universidade do Estado do Rio Grande do Norte como um dos pré-requisitos para obtenção do grau de bacharel em Ciência da Computação, sob orientação do Prof. Me. Antônio Oliveira Filho.

MOSSORÓ - RN

2017

Ficha catalográfica gerada pelo Sistema Integrado de Bibliotecas
e Diretoria de Informatização (DINF) - UERN,
com os dados fornecidos pelo(a) autor(a)

S586m Silva, Claudivan Barreto da .
UM MODELO COMPUTACIONAL PARA INTEGRAÇÃO DE PROBLEMAS DE OTIMIZAÇÃO UTILIZANDO BANCO DE DADOS ORIENTADOS A GRAFOS / Claudivan Barreto da Silva - 2017.
48 p.

Orientador: Antônio Oliveira Filho.
Coorientadora: .
Monografia (Graduação) - Universidade do Estado do Rio Grande do Norte, Ciência da Computação, 2017.

1. Métodos Heurísticos. 2. Estrutura de dados. 3. Modelo arquitetural. I. Oliveira Filho, Antônio, orient. II. Título.

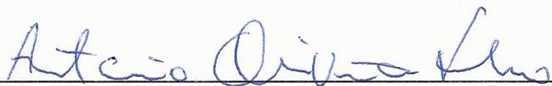
Claudivan Barreto Da Silva

UM MODELO COMPUTACIONAL PARA INTEGRAÇÃO DE PROBLEMAS DE OTIMIZAÇÃO UTILIZANDO BANCO DE DADOS ORIENTADOS A GRAFOS

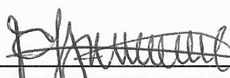
Monografia apresentada como pré-requisito para a obtenção do título de Bacharel em Ciência da Computação da Universidade do Estado do Rio Grande do Norte – UERN, submetida à aprovação da banca examinadora composta pelos seguintes membros:

Aprovada em: ___/___/______

Banca Examinadora



Prof. Me. ANTÔNIO OLIVEIRA FILHO
Universidade do Estado do Rio Grande do Norte - UERN



Prof. Dr. FRANCISCO CHAGAS DE LIMA JÚNIOR
Universidade do Estado do Rio Grande do Norte - UERN

Prof. Dr. CARLOS HEITOR PEREIRA LIBERALINO
Universidade do Estado do Rio Grande do Norte - UERN

Aos meus pais.

Agradecimentos

Primeiramente, agradeço a Deus por tudo que fez para que eu alcançasse esta vitória, sem Ele nada disso valeria a pena.

A minha família por sempre acreditar em mim. Principalmente aos meus pais, Claudioberg e Cleide, por todo suporte que sempre me proporcionaram e ao meu tio e minha vó, Selmo e Maria, pelos ensinamentos que me ajudaram a chegar onde estou.

A minha namorada, Marisa, pela força e por sua importante presença em todos os momentos.

Aos professores do DI/UERN, pela contribuição para o meu desenvolvimento intelectual, em especial aos professores Marcelino Pereira, Lima Júnior, Alysson Mendes e ao meu orientado Antônio Oliveira por todo seu esforço e dedicação.

Aos meus amigos que adquiri ao longo deste curso, obrigado pelos agradáveis momentos que compartilhamos.

Por fim, o meu muito obrigado aos que me ensinaram a pegar no lápis, a entender os números e a não temer a vida.

"Quando o jogo de xadrez termina,
o peão e o rei vão para a mesma caixa".
Ditado Chinês

Resumo

Métodos modernos para resolução de problemas de otimização em Pesquisa Operacional necessitam de mecanismos para armazenamento de grafos. No entanto, as estruturas de dados utilizadas pelos pesquisadores atualmente não oferecem meios eficientes para este fim. O presente trabalho tem como objetivo a criação de um modelo de dados para armazenar grafos e suas soluções. Para tanto, este modelo foi construído em um banco de dados orientados a grafos chamado *ArangoDB*. É apresentado também uma arquitetura de acesso aos grafos pela *web*, possibilitando que os pesquisadores contribuam e compartilhem seus resultados. Esta arquitetura foi construída através de uma API REST implementada em um servidor JavaScript, *Node.js*. Através do modelo proposto, um estudo é exposto demonstrando a possibilidade dos pesquisadores em comparar e combinar resultados advindos de algoritmos de caminho mínimo.

Palavras-chave: Métodos Heurísticos, Estrutura de dados, Modelo arquitetural.

Abstract

Modern methods for solving optimization problems in Operations Research demands mechanisms for storing graphs. However, the data structures used by researchers nowadays, do not provide efficient resources for this purpose. The present work has the goal of creating a data model to store graphs and their solutions. For this, this model was built in a graph database called ArangoDB. An architecture of access to graphs through the web is also presented, allowing the researchers to contribute and share their results. This architecture was built through a REST API implemented in a JavaScript server, Node.js. Through the proposed model, a study is exposed demonstrating the possibility of the researchers to compare and combine results from shortest path algorithms.

Keywords: Heuristic methods, Data structure, Architectural model.

Lista de ilustrações

Figura 1 – Tipos de bancos de dados NoSQL 19
Figura 2 – Grafo de propriedades do *Twitter* 21
Figura 3 – Elementos do modelo 28
Figura 4 – Diagrama de classe do modelo M2P 31
Figura 5 – Diagrama de arquitetura do modelo M2P 32
Figura 6 – Funcionalidade das arestas *partOf* 33
Figura 7 – Grafo base 35
Figura 8 – Geração da BaseSolution 36
Figura 9 – Solução gerada pelo algoritmo guloso 37
Figura 10 – Solução gerada pelo algoritmo aleatório 38
Figura 11 – Grafo base e as soluções geradas 39
Figura 12 – *match* e *mismatch* presentes do grafo base 41
Figura 13 – Geração da *HiperSolution* 42
Figura 14 – Grafo solução referenciado pela HiperSolution 43
Figura 15 – Nova solução 44

Lista de tabelas

Tabela 1 – Métodos HTTP 24
Tabela 2 – Categorias dos códigos de estado do HTTP 25
Tabela 3 – Relação entre as categorias de grafos do modelo M2P e as coleções contidas no *ArangoDB* 33
Tabela 4 – API REST do Modelo M2P 34
Tabela 5 – Funcionalidades extras da API REST do Modelo M2P 34

Lista de abreviaturas e siglas

API	Application Programming Interface
CRUD	Create, Read, Update, Delete
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
NOSQL	Not Only SQL
M2P	Method Program Problem
REST	Representational State Transfer
SQL	Structured Query Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Name
TCP	Transmission Control Protocol
XML	eXtensible Markup Language

Sumário

1 INTRODUÇÃO . . . 13

2 REFERENCIAL TEÓRICO . . . 16
2.1 Grafos, Problemas e Métodos . . . 16
2.2 NoSQL . . . 17
2.2.1 Classificação dos Bancos de Dados NoSQL . . . 18
2.2.1.1 Chave-Valor . . . 19
2.2.1.2 Baseado em Documentos . . . 19
2.2.1.3 Baseado em Colunas . . . 20
2.2.1.4 Orientado a Grafos . . . 20
2.2.2 *ArangoDB* . . . 21
2.3 Representação por Transferência de Estado (REST) . . . 22
2.3.1 Identificador Uniforme de Recurso (URI) . . . 24
2.3.2 Protocolo de Transferência de Hipertexto (HTTP) . . . 24
2.4 Node.js . . . 25
2.4.1 Express.js . . . 26

3 O MODELO . . . 27
3.1 Elementos . . . 28
3.2 Implementação . . . 31

4 PROVA DE CONCEITO . . . 35
4.1 *HiperSolution* . . . 38
4.1.1 *Match* e *Mismatch* . . . 39
4.1.2 Resultados . . . 42

5 CONSIDERAÇÕES FINAIS . . . 45

REFERÊNCIAS . . . 47

1 Introdução

A área de Pesquisa Operacional envolve um série de métodos para resolução de problemas complexos de otimização, dos quais destacam-se as heurísticas, meta-heurísticas e hiper-heurísticas. Estes métodos são utilizados em casos em que não existem estratégias que proporcionem algum tipo de garantia de solução ótima (PAPADIMITRIOU; STEIGLITZ, 1998).

Esses métodos surgiram decorrentes das desvantagens de outros métodos. Por exemplo, para problemas mais complexos as heurísticas não são adaptáveis o bastante para contribuírem com soluções expressivas, as meta-heurísticas originaram-se desta dificuldade. De modo semelhante, as meta-heurísticas apresentam frequentemente resultados insatisfatórios quando não refinados através da aplicação de conhecimentos específicos. As hiper-heurísticas foram concebidas a partir desta necessidade (SUCUPIRA, 2007).

Em poucas palavras, uma hiper-heurística é uma heurística que coordena um conjunto de heurísticas, selecionando uma delas em cada ponto de decisão. Na realidade atual, uma forma de construir esse método seria estudar amplamente grande volumes de dados que permitissem obter uma solução plausível. No entanto, essa possibilidade depara-se com os seguintes problemas:

- Diferentes estruturas para armazenamento dos grafos;
- Compartilhamento árduo das soluções obtidas;
- Algoritmos construídos em diferentes linguagens de programação.

Existem diferentes maneiras de armazenar grafos em sistemas computacionais, cada uma apresenta vantagens e desvantagens. A estrutura de dados utilizada depende da estrutura do grafo e do algoritmo usado para manipulá-lo. Estas estruturas são distinguidas entre matrizes e listas. Estruturas de listas são frequentemente preferidas para grafos densos, elas possuem menores requisitos de memória. Estrutura de matrizes, por outro lado fornecem acesso mais rápido para algumas aplicações, mas podem consumir enormes quantidades de memória (PEDIAPRESS, 2011).

O armazenamento dessas estruturas assemelham-se ao primeiro sistema para armazenamento e manipulação de dados, os sistemas de processamento de arquivos que é um método da década de 1960 para gerenciamento computadorizados de dados comercias. Estes sistemas entraram em desuso por possuírem as seguintes desvantagens: redundância e inconsistências de dados, dificuldade de acesso a dados, isolamento de dados, problemas de integridade, problemas de atomicidade, entre outros (SILBERSCHATZ et al., 2006).

Armazenar grafos usando estas estruturas tornam a manipulação dos dados complexa e difícil de compreender, dado que cada pesquisador implementa seu próprio método em sua linguagem de programação de preferência.

A tarefa torna-se mais enfadonha quando um pesquisador deseja compartilhar seus resultados. Ele precisa utilizar algum meio de comunicação para enviar seus arquivos. Porventura, se o pesquisador que recebe esses dados quiser testá-los afim de comprovar sua autenticidade, este precisa conhecer a estrutura utilizada para executar seus algoritmos, surgindo às vezes a necessidade de adaptadores para reconhecimento dos dados.

Dentre os bancos de dados atuais, surgiram os bancos de dados orientados a grafos. Estes se caracterizam por possuir estruturas de dados no qual os esquemas são modelados como grafos, permitindo uma modelagem mais natural dos dados e provendo estruturas eficientes para que algoritmos executem funções específicas (ANGLES; GUTIERREZ, 2008).

O presente trabalho tem como objetivo definir um modelo de dados para armazenamento de grafos e soluções a ele relacionadas e propor um modelo arquitetural de acesso a estes dados via *web*. O modelo juntamente com a arquitetura auxiliará ao pesquisador na construção de soluções sem as deficiências presentes na estrutura de dados apresentadas anteriormente. E além disto, propor um meio de acesso ao dados de forma que qualquer pesquisador em qualquer lugar pode fazer contribuições para um mesmo problema.

Desta forma, será possível a criação de um sistema que comporte-se como uma grande hiper-heurística, que ao passo que as soluções são armazenadas, o sistema combine resultados advindo dos diversos algoritmos com a finalidade de encontrar a solução ótima.

O modelo engloba um conjunto de tecnologias composto por um banco de dados orientados a grafos chamado *ArangoDB* e uma API (Application Programming Interface) REST (Representational State Transfer) construída em *Node.js* com auxílio do *framework Express.js*.

Os objetivos específicos constituem-se em:

- Definir de uma arquitetura para acesso aos grafos;
- Compartilhar soluções entre os pesquisadores;
- Comprovar soluções dos pesquisadores;
- Comparar soluções de um mesmo problema sobre um grafo;
- Desenvolver algoritmos para prova de conceito;
- Prover independência de linguagem para resolução de problemas.

O presente documento está organizado da seguinte maneira: no capítulo 2 são apresentados os conceitos fundamentais que apoiarão o estudo deste trabalho e as tecnologias empregadas nesse processo. No capítulo 3 é descrito o modelo e a aplicação proposta, seu funcionamento e implementação. No capítulo 4 é apresentado uma prova de conceito mostrando os resultados da implementação. Por fim, no capítulo 5 são expostas as considerações finais e as perspectivas futuras.

2 Referencial Teórico

2.1 Grafos, Problemas e Métodos

Um grafo pode ser definido como um conjunto de vértices e arestas, ou ainda descrito como um conjunto de nós que estão conectados por relacionamentos.

Formalmente, um grado é definido por $G = (V, E)$, onde $V = (v_1, v_2, \ldots, v_n)$ é o conjunto do vértices e $E = (e_1, e_2, \ldots, e_n)$ é o conjunto das arestas.

Os grafos modelam os mais diversos tipos de problemas. O problema do caminho mínimo consiste em buscar o menor caminho entre um par de vértices em um grafo. Este é um problema fundamental com inúmeras aplicações. Por exemplo, ele é utilizado por transportadoras, em que os produtos de um local precisam ser conduzidos para outro lugar com o menor custo possível.

> O problema do caixeiro viajante é um problema de caminho ponderado com restrições fortes sobre a natureza do caminho, de modo que este caminho pode vir a não existir. No problema de caminho mínimo, não existem restrições na natureza do caminho; como o grafo é conexo, sabemos que o caminho existe. Por esta razão, pode esperar que haja um algoritmo eficiente para resolver este problema, ainda que não exista algoritmo eficiente para o problema do caixeiro viajante (GERSTING, 2004).

O algoritmo de Dijkstra's é um dos algoritmos mais conhecidos e utilizados para encontrar o caminho mínimo. Em contrapartida, algoritmos gulosos são bastantes cogitados para resolverem o problema do caixeiro viajante. Um algoritmo guloso sempre realiza a escolha que parece ser a melhor no momento. Isto é, ele faz uma escolha ótima para as condições locais, na esperança de que a escolha leve à uma solução ótima para a situação global.

Outra forma de resolver o problema do caixeiro é por meio da utilização de métodos heurísticos. Uma heurística é uma técnica algorítmica para encontrar soluções para problemas de otimização em tempo hábil, no entanto, sem oferecer garantias quanto à sua qualidade em relação às soluções ótimas.

Segundo Sucupira (2007), as heurísticas específicas frequentemente possuem alta qualidade, contudo na maioria dos casos, não são adaptáveis o suficiente para contribuírem de maneira significativa na resolução dos problemas de otimização. As meta-heurísticas surgiram para solucionar este problema.

> Uma meta-heurística é um conjunto de conceitos que pode ser utilizado para definir métodos heurísticos aplicáveis a uma ampla coleção de

> problemas diversos. Em outras palavras, uma meta-heurística pode ser vista como uma estrutura algorítmica geral que pode ser empregada na resolução de diferentes problemas de otimização, com um número relativamente pequeno de modificações que a adaptem para o tratamento de cada problema específico (ZÄPFEL; BRAUNE, 2010).

As meta-heurísticas tem sido bastante difundidas e utilizadas pelos pesquisadores nos últimos anos para resolverem problemas em inteligência artificial e pesquisa operacional. Burke et al. (2003) afirma que a investigação de meta-heurísticas para uma vasta e diversa gama de áreas de aplicação tem influenciado fortemente o desenvolvimento da tecnologia moderna de busca, tal como, agendamento, mineração de dados, corte de estoque, processamentos de imagens médicas, bio-informática, entre outras. No entanto, o impacto prático em organizações comerciais e industriais não foi tão grande como esperado devido às meta-heurísticas apresentarem resultados pobres na maioria das vezes quando não são aperfeiçoados através do emprego de conhecimentos específicos para o problema em questão.

As hiper-heurísricas surgiram como uma consequência da necessidade de elevar o nível de generalidade dos métodos de otimização. Segundo Sucupira (2007), o termo "hiper-heurística" pode se referir a qualquer processo heurístico que administre outras heurísticas na resolução de problemas de otimização. Em outras palavras, uma hiper-heurística recebe um conjunto de heurísticas e, em cada ponto de decisão, seleciona uma delas para aplicar no passo seguinte.

2.2 NoSQL

A quantidade de dados das organizações tem aumentado de tal modo que as aplicações não conseguem analisar completamente todos os dados ao passo que eles são gerados. McAfee et al. (2012) afirma que são gerados cerca de 2.5 exabytes a cada dia, e que esse número dobraria a cada 40 meses. Isso tem motivado as organizações a procurarem novos métodos de armazenamento e processamento de dados. Com advento do Big Data é possível medir e gerenciar toda essa massa de dados.

Ainda não existe um consenso sobre o significado do termo Big Data, mas pode ser descrito como um conjunto de tendências tecnológicas que permite uma nova abordagem para o análise e compreensão de grandes volumes de dados para fins de tomada de decisões (BRETERNITZ; SILVA, 2013), ou ainda definido como uma grande e complexa coleção de dados no qual banco de dados tradicionais não são capazes de manipular de forma eficiente e viável.

Segundo Zikopoulos et al. (2012) Big Data se caracteriza por quatro fatores: volume, velocidade, variedade e veracidade. A seguir são descritos estes quatros fatores:

- Volume: quantidade de dados aumenta exponencialmente.
- Velocidade: taxa na qual dados são processados ou bem compreendidos.
- Variedade: dados não estruturados providos de fontes diferentes.
- Veracidade: qualidade ou credibilidade dos dados.

O NoSQL (*Not Only SQL*) tem sido bastante explorado quando o assunto é Big Data devido sua capacidade de processar e armazenar grandes volumes de dados, semi-estruturados ou não estruturados, que precisam de alta disponibilidade e escalabilidade.

O termo NoSQL é definido para categorizar os bancos de dados que possuem as seguintes características: não-relacional, distribuído, de código aberto, horizontalmente escalável, ausência de esquema ou esquema flexível, suporte nativo a replicação e acesso via APIs simples (NOSQL, 2009).

Uma linguagem de consulta pode ser definida como uma linguagem que é usada para manipular dados dentro de um banco de dados. NoSQL não usa Structured Query Language (SQL), que é a linguagem mais comumente usada nos bancos de dados relacionais. Em vez disso, o NoSQL não tem uma linguagem de consulta padrão. A maioria dos bancos de dados NoSQL fornecem sua própria linguagem de consulta.

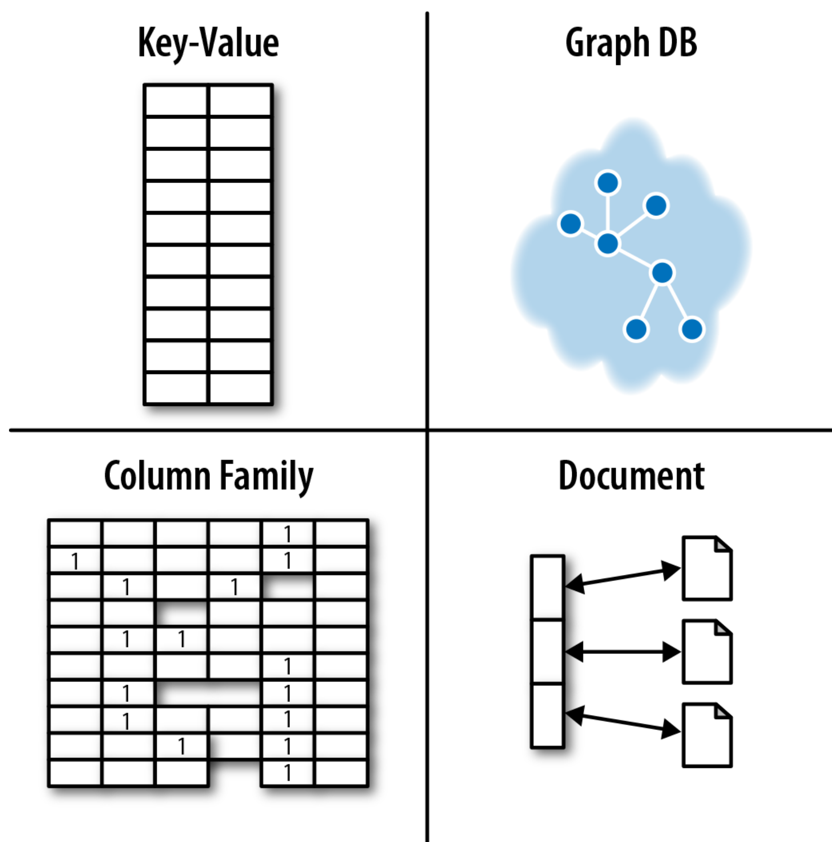
Uma característica importante nos bancos de dados NoSQL é a capacidade de escalar horizontalmente, em que há o aumento do número de máquinas disponíveis para processamento e armazenamento ao invés de escalar verticalmente no qual o são adicionados mais recursos (mais núcleos, mais memória, ou mais um disco rígido) as maquinas já existestes, melhorando o desempenho da aplicação, no entanto essa alternativa geralmente é inviável devido alto custo (VIEIRA et al., 2012).

Geralmente bancos de dados NoSQL são mais rápidos do que bancos relacionais devido a ausência completa ou quase total do esquema que define a estrutura dos dados modelados. Esta ausência de esquema simplifica tanto a escalabilidade quanto contribui para um maior aumento da disponibilidade. O enorme crescimento dos bancos NoSQL, o levou a ser dividido em sub-categorias descritas a seguir.

2.2.1 Classificação dos Bancos de Dados NoSQL

Leavitt (2010), classifica os bancos de dados NoSQL em três tipos: Baseado em Chave-Valor (*Key-Value stores*), Baseado em Coluna (*Column-oriented databases*) e Baseado em Documentos (*Document-based stores*). Nesta seção será adicionado um tipo a mais, o Orientado a Grafos (*Graph databases*). Todos estes bancos de dados estão ilustrados na figura 1.

Figura 1 – Tipos de bancos de dados NoSQL



Fonte: (ROBINSON; WEBBER; EIFREM, 2015)

2.2.1.1 Chave-Valor

Neste modelo, um dado consiste de duas partes, uma string representado a chave e o dado referenciado como valor, deste modo criando um par chave-valor (NAYAK; PORIYA; POOJARY, 2013). Este modelo assemelha-se a um grande tabela *hash* no qual cada valor é indexado pela sua respectiva chave permitindo que os dados sejam recuperados rapidamente.

2.2.1.2 Baseado em Documentos

Os dados são organizados em forma de coleções de documentos de modo que os usuários podem adicionar qualquer número de campos de qualquer tamanho. Estes documentos estão tipicamente sob o formato eXtensible Markup Language (XML) ou JavaScript Option Notation (JSON). Diferente do modelo chave-valor onde há somente um tabela hash, o banco de dados baseado em documentos existe um conjunto de documentos e em cada documento tem um conjunto de chaves e os valores associados a estas chaves. Uma grande vantagem deste modelo é de não possui um esquema, ou seja, os documentos

armazenados não precisam ter uma estrutura em comum.

2.2.1.3 Baseado em Colunas

Diferente dos bancos relacionais, no qual dados ficam disposto em forma de tuplas (linhas), os bancos baseados em colunas armazenam seus registros em colunas separadas, guardando contiguamente os valores de atributos pertencendo à mesma coluna. Um benefício desta abordagem é a capacidade de compressão dos dados pelo fato das informações semelhantes serem armazenadas sequencialmente (ABADI; BONCZ; HARIZOPOULOS, 2009). Este modelo é adequado para aplicações de mineração de dados (*data mining*) no qual o método de armazenamento é ideal para as operações realizadas nos dados.

2.2.1.4 Orientado a Grafos

Os bancos de dados orientados a grafos se caracterizam por possuírem estruturas de dados no qual os esquemas e/ou instâncias são modeladas como grafos que podem ser possivelmente rotulados (ANGLES; GUTIERREZ, 2008). O grafo consiste de vértices e arestas, em que os vértices atuam como entidades e as arestas como os relacionamentos entre as entidades.

Bancos de dados orientados a grafos podem ser usados em uma vasta gama de aplicações como redes sociais, bioinformática, segurança, controle de acesso, gerenciamento de nuvem e etc.

Segundo Angles e Gutierrez (2008), os bancos de dados orientados a grafos permitem uma modelagem mais natural dos dados, provêm uma eficiente estrutura de armazenamento para que algoritmos executem específicas operações e consultas possam utilizar-se da estrutura do grafo. Por exemplo, retornar vértices adjacentes, encontrar o menor caminho e localização de subgrafos.

Estes bancos de dados podem ser classificados como nativos e não-nativos. Os nativos são otimizados e projetados para armazenamento e gerenciamento de grafos. São usadas listas de adjacências no armazenamento possibilitando o acesso rápido ao grafo. Por outro lado, os não-nativos usam outras forma de armazenamento como por exemplo o modelo relacional (em forma de tabelas) que afeta o desempenho de consultas quando diversas junções (*joins*) são necessárias (PENTEADO et al., 2014).

A análise de novos tipos de dados exigem modelos mais flexíveis que suportam dados em diferentes graus de estrutura. Banco de dados de grafos implementam o grafo de prioridade que é uma resposta razoável para esta demanda, pois eles suportam dados em uma estrutura altamente irregular sob a forma de um grafo (VASILYEVA et al., 2016).

Um grafo de propriedades é um multigrafo em que seus nós (vértices) e seus relacionamentos (arestas) possuem atributos para descrever suas propriedades possibilitando

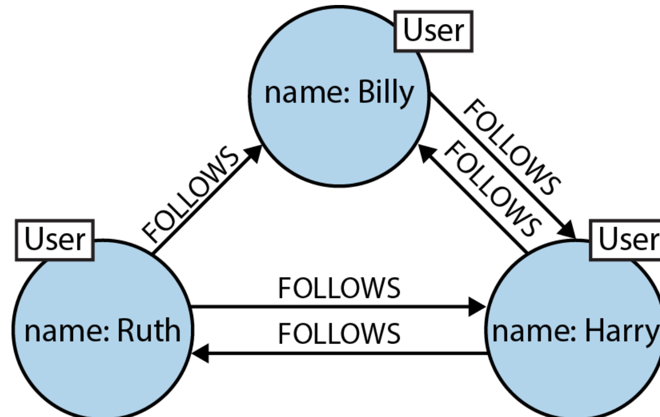
múltiplas relações e um maior poder de modelagem (PENTEADO et al., 2014).

Segundo Robinson, Webber e Eifrem (2015), o grafo de propriedades tem as seguintes características:

- Contém nós e relacionamentos.
- Nós contêm propriedades (pares chave-valor).
- Nós podem ser rotulados com um ou mais rótulos.
- Relacionamentos são nomeados e direcionados, e sempre têm um início e um fim.
- Relacionamentos também contêm propriedades.

Uma vantagem dos multigrafos é que eles são a implementação mais complexa, pois qualquer outro tipo de grafo consiste em um subconjunto da implementação de um grafo de propriedades (MILLER, 2013). Isso significa que um grafo de propriedades pode efetivamente modelar qualquer outro tipo de grafo.

Figura 2 – Grafo de propriedades do *Twitter*



Fonte: (ROBINSON; WEBBER; EIFREM, 2015)

A figura 2 ilustra a rede social *Twitter*. Os nós representam os usuários e a aresta *follows* define um relacionamento entre um usuário que decide seguir outro.

Exemplos deste modelo são o Neo4J, Infinite Graph e o *ArangoDB*.

2.2.2 *ArangoDB*

ArangoDB é um banco de dados NoSQL nativo e híbrido que combina os modelos chave-valor, baseado em documentos e orientados a grafos. Todos os dados podem ser acessados com uma linguagem de consulta. Também é possível combinar diferentes modelos

em uma única consulta. Devido a ser um banco de dados nativo é possível criar aplicações de alta performance e escalabilidade com os três modelos de dados.

Os dados do *ArangoDB* são transportados usando JSON via Transmission Control Protocol (TCP) usando o protocolo HTTP. Uma API REST é fornecida para interagir com o banco de dados. O *ArangoDB* também dispõe de uma interface *web* chamada Aardvark e um *shell* interativo chamado Arangosh. Além disso há diversos *drivers* em várias linguagens de programação que facilita a conexão com o banco de dados. Todas essas ferramentas usam a interface HTTP e removem a necessidade de escrever código de baixo nível na maioria dos casos.

Os documentos no *ArangoDB* podem conter um ou mais atributos, cada atributo tem um valor. Um valor pode assumir os seguintes tipos: número, string, booleano, *null* (nulo), *array* (vetor) e objeto. Os *arrays* e objetos podem conter todos esses tipos, o que significa que estruturas de dados arbitrariamente aninhadas podem ser representadas em um único documento.

Estes documentos são agrupados em coleções. Uma coleção pode conter um ou mais documentos. Comparando com os bancos relacionais, as coleções se assemelham com as tabelas e os documentos com as tuplas (linhas). A diferença é que os bancos relacionais possuem colunas que definem um esquema de dados que deve ser seguido por todas as tuplas, no *ArangoDB* é possível que cada documento detenha seu próprio esquema.

No *ArangoDB* existem dois tipos de coleções:

- Coleção de documentos: representa a coleção de vértices no contexto de grafos.
- Coleção de arestas: também são documentos, mas incluem dois atributos especiais, *_from* (origem) e *_to* (destino) que são usados para criar relações entre os documentos.

Geralmente dois documentos (vértices) são armazenados em um coleção de documentos que são relacionadas por um documento (aresta) armazenado em uma coleção de arestas. As coleções existem dentro de uma base de dados, na qual é possível ter uma ou mais bases de dados. As *queries* (consultas) no *ArangoDB* são escritas usando a linguagem AQL (*ArangoDB Query Language*).

2.3 Representação por Transferência de Estado (REST)

Fielding (2000) define REST como um estilo arquitetural para sistemas distribuídos de hipermídias. REST se trata de um conjunto de restrições a serem utilizadas no processo de construção de aplicações web distribuídas. Inclusive, o modelo REST ignora os detalhes da implementação dos componentes e da sintaxe do protocolo, com o objetivo de enfatizar

os papéis dos componente. Além disso, o REST dispõe de um conjunto de interfaces genéricas com o intuito de proporcionar interações sem estado.

REST trata qualquer tipo de serviço ou informação (páginas, vídeos e documentos) como um recurso. Os recursos são identificados por um Identificador Uniforme de Recurso do inglês Uniform Resource Identifier (URI) que são acessados por meio do Protocolo de Transferência de Hipertexto do inglês Hypertext Transfer Protocol (HTTP).

A seguir serão descritos as principais características do REST:

- Identificação global: um URI identifica exclusivamente um recurso na *web* e simultaneamente o faz endereçável ou capaz de ser manipulado usando um protocolo de aplicação tal como HTTP. O URI de um recurso o distingue de qualquer outro recurso e é através dele que as interações com os recursos acontecem.
- Interface uniforme: os recursos são acessados usando uma interface genérica por meio do protocolo HTTP que utiliza um conjunto de métodos: GET, POST, PUT, DELETE.
- Interações sem estado: o servidor não mantém informação alguma sobre os clientes, ou seja, cada requisição do cliente para o servidor deve conter toda informação necessária para que a requisição seja entendida (DOGLIO, 2015).

Os serviços *web* são servidores especificamente construídos para suportar as necessidades de um site ou de qualquer outro tipo de aplicação. Os programas clientes usam uma Interface de Programação de Aplicativos do inglês Application Programming Interface (API) para comunicar-se com os serviços *web*. De modo geral, uma API dispõe de um conjunto de dados e funções que facilita a interação entre aplicações e permitem que elas troquem informações (MASSE, 2011).

O estilo arquitetural REST é comumente usado para projetar APIs para serviços *web* modernos. Uma API *Web* em conformidade com estilo arquitetural REST é chamado API REST. Basicamente, uma API REST recebe uma requisição HTTP, executa algum processamento e envia sempre uma resposta HTTP.

Normalmente em uma API REST, as URIs tem os mesmos nomes das coleções de dados armazenados no banco de dados. Geralmente existe um conjunto de URIs para cada coleção. Dentro de um conjunto de URIs é preciso cobrir vários tipos de ações, geralmente baseados nas operações CRUD (*Create*, *Read*, *Update*, *Delete*). As principais ações são:

- Criar um novo recurso;
- Ler uma lista de recursos;

- Ler um recurso específico;
- Atualizar um recurso específico;
- Apagar um recurso específico.

Estas ações são mais detalhadas na tabela 1.

2.3.1 Identificador Uniforme de Recurso (URI)

APIs REST utilizam URIs para endereçar seus recursos. Um URI segundo Coulouris, Dollimore e Kindberg (2012) é um conjunto de caracteres que serve para identificar um recurso na internet. O URI é um identificador de recurso geral cujo valor pode ser um Uniform Resource Name (URN) ou Uniform Resource Locator (URL). O URL é reservado para identificadores que são localizadores de recursos. Os URNs são independentes de localização, ou seja, eles contam com um serviço para fazer um mapeamentos para os URLs do recursos.

2.3.2 Protocolo de Transferência de Hipertexto (HTTP)

As APIs REST abrangem todos os aspectos do HTTP, bem como os métodos de requisição, códigos de reposta e cabeçalhos de mensagem. Segundo Kurose et al. (2013), O HTTP é um protocolo da camada de aplicação da *Web* que é executado em dois programas: um cliente e outro servidor. Os dois são executados em sistemas finais diferentes e conversam entre si por meio de troca de mensagens HTTP.

O HTTP pode ser usado para acessar os vários tipos de recursos. Além disso, ele apresenta um conjunto de métodos cuja a semântica define o tipo da ação esperada no recurso.

Supondo que um servidor está hospedado no endereço *m2p.com* e que o nome da coleção do bando de dados é *vertices*. A tabela 1 exemplifica os principais métodos HTTP relacionado-os com as ações de uma API REST descritas na seção 2.3.

Tabela 1 – Métodos HTTP

AÇÃO	MÉTODO	URI	PARÂMETROS	EXEMPLO
Ler uma lista de recursos	GET	/vertices		http://m2p.com/vertices
Ler um recurso específico	GET	/vertices	vertexid	http://m2p.com/vertices/123
Criar um novo recurso	POST	/vertices		http://m2p.com/vertices
Atualizar um recurso específico	PUT	/vertices	vertexid	http://m2p.com/vertices/123
Apagar um recurso específico	DELETE	/vertices	vertexid	http://m2p.com/vertices/123

Fonte – (MASSE, 2011)

Através dos métodos HTTP é possível interagir com os recursos. Desse modo, criando uma interface uniforme que pode ser acessada pelos diversos dispositivos.

Uma característica importante do HTTP é o uso de códigos de estado (*status codes*). Um código de estado é um número de três dígitos que resume a resposta enviada pelo servidor e que ajuda os clientes a interpretá-la.

O código de estado mais familiar é o 404 (*Not found*), que é retornado pelo servidor quando um cliente solicita uma página que não pode ser encontrada. Os códigos de estado estão divididos em cinco categorias baseado em seus significados. A tabela 2 apresenta estas categorias.

Tabela 2 – Categorias dos códigos de estado do HTTP

CATEGORIA	DESCRIÇÃO
1xx:Informativa	Comunica informações do protocolo de transferência.
2xx:Sucesso	Indica que a requisição do cliente foi aceita com êxito.
3xx:Redirecionamento	Indica que o cliente deve tomar alguma ação adicional para completar seu pedido.
4xx:Erro de Cliente	Indica os casos em que o cliente pode ter cometido algum erro.
5xx:Erro de Servidor	Indica um erro do servidor ao processar a requisição.

Fonte – (MASSE, 2011)

2.4 Node.js

O Node.js foi criado por Ryan Dahl com objetivo de resolver os problemas das arquiteturas bloqueantes. Uma arquitetura bloqueante enfileira as requisições e posteriormente processa uma por uma, não possibilitando o processamento de várias delas ao mesmo tempo. Por exemplo, enquanto requisição está executando um I/O, o processador mantém-se ocioso esperando a requisição finalizar.

O Node.js faz uso da *Engine V8* do *Google* para executar código JavaScript no lado do servidor. A V8 é o interpretador JavaScript desenvolvido pelo *Google* que é atualmente usado no navegador *Google Chrome*. Segundo Hughes-Croucher (2010) a V8 é extremamente rápida e tem um bom desempenho em diversas circunstâncias. Além disso, a V8 utiliza uma tecnologia de compilação que permite que o código escrito em uma linguagem de alto nível como JavaScript seja tão rápida quanto um código escrito em uma linguagem de baixo nível como C, com uma pequena fração do custo associada ao desenvolvimento.

JavaScript é uma linguagem orientada a eventos, e o *Node.js* emprega isso como uma vantagem, utilizando uma arquitetura chamada *Event Loop*. O *Event Loop* é o mecanismo interno responsável por escutar e emitir eventos no sistema. Basicamente, ele é um *loop* infinito que em cada interação verifica se existem novos eventos para serem processados em sua fila de eventos. Com isso, o *Node.js* utiliza operações I/O não bloqueantes e assíncronas, apenas registrando uma função de *callback* que é chamada somente quando a operação em causa é concluída, prevenindo assim que a aplicação fique bloqueada (SOUSA, 2015).

As bibliotecas de funções no *Node.js* são chamados de módulos. O *Node.js* é composto por módulos que fazem parte do núcleo (*core modules*) e módulos criados pela comunidade. Através do gerenciador de pacotes do *Node.js* chamado *npm* (*Node Package Manager*) é possível integrar uma aplicação com os diversos módulos já existentes.

Os módulos não participantes do núcleo do Node.js podem estar nas dependências da aplicação. Uma dependência da aplicação, neste contexto, é um módulo que necessita ser instalado para prover funcionalidades para aplicação (CANTELON et al., 2014). As dependências da aplicação são especificadas em um arquivo chamado *package.json*. Este arquivo consiste de uma expressão JSON munida de informações que descrevem a aplicação.

2.4.1 Express.js

Um módulo bastante usado para a construção de serviços *web* utilizando o estilo arquitetural REST é o *Express.js*. O *Express.js* é um *framework* que tem por objetivo simplificar o desenvolvimento de APIs para aplicações *web* com *Node.js*, permitindo a criação de servidores que receba requisições HTTP de forma simples. Além disso fornece maneiras de reutilizar código e provê uma estrutura semelhante ao modelo *Model View Controller* (MVC). De acordo com Hahn (2016), O *Express.js* abstrai a complexidade do servidor HTTP do *Node.js* adicionando um número significativo de funcionalidades. Por exemplo, para enviar uma imagem JPEG em *Node.js* puro é bastante complexo. O *Express.js* reduz isso para um linha.

3 O Modelo

Atualmente, os pesquisadores em Pesquisa Operacional executam localmente seus programas utilizando uma estrutura dados específica, tipicamente sob forma de matrizes ou listas. Geralmente os programas implementam métodos heurísticos ou meta-heurísticos para resolverem problemas de otimização, tais como, roteamento de veículos, caixeiro viajante, coloração de grafos, entre outros.

Utilizar uma estrutura de dados específica torna-se um problema quando um pesquisador deseja executar seu programa sobre uma base de dados de um outro pesquisador. Diversas vezes, há a necessidade de criação de adaptadores para reconhecimento dos dados. Além disso, quando o grafo é demasiadamente grande, as matrizes tornam-se uma opção inviável pelo alto espaço de armazenamento. As vezes, é preciso que um programa opere sobre um subconjunto de um grafo (subgrafo). Realizar isto com o modelo de dados convencional (listas ou matrizes) é uma tarefa complexa e custosa.

Para solucionar estes problemas, é necessário um modelo onde sua estrutura de dados se aproximasse o máximo possível da estrutura de um grafo. Da mesma forma, é preciso um meio de armazenar as diversas instâncias de soluções de um dado programa e que cada parte da solução possua características que levaram ela a ser escolhida. Inclusive, é essencial que os pesquisadores tenham acesso a cada versão de um programa e tivessem conhecimento de quais métodos foram utilizados para gerar tal solução.

Todos esses dados deveriam ser compartilhados para comparação e comprovação. Comparação entre as soluções de diferentes programas que usaram métodos distintos afim de analisar quais os aspectos foram levados em consideração para que uma parte de uma solução sobressaísse de outra e comprovação da autenticidade de um estudo que fora realizado sobre um determinado grafo.

O modelo trata-se de uma estrutura de dados que agrupa todas essas características sob a forma de um grafo de propriedades, atendendo os seguintes requisitos:

- Disponibilidade de um modelo para armazenamento de grafos;
- Fornecimento de uma interface de acesso aos grafos;
- Registro de programas, problemas e métodos;
- Registro de soluções provenientes do programa;
- Acesso a resultados compartilhados;

Baseando-se nos requisitos citados acima, pode-se destacar algumas vantagens:

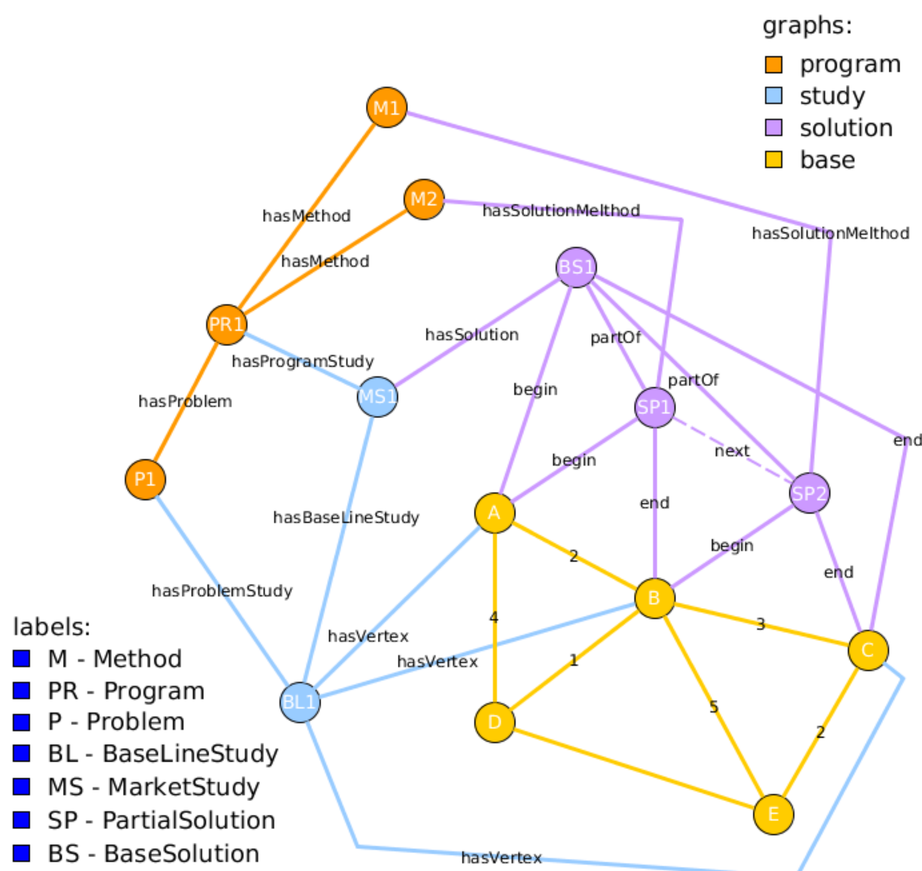
- Criação estudos sobre um grafo;
- Comparação e comprovação soluções;
- Criação de adaptadores desnecessária;
- Combinação de soluções.

Para que este modelo torna-se possível é preciso a criação de elementos que ampare sua estrutura. Estes elementos foram propositalmente nomeados em inglês.

3.1 Elementos

O modelo engloba quatro categorias de grafos interconectados, como exposto na figura 3, possibilitando que todos os elementos do modelo fossem definidos sob a forma de grafos. A seguir serão descritos os elementos vértices.

Figura 3 – Elementos do modelo



Fonte: Autoria própria.

- *Program*: representa um programa que gera soluções.
- *Method*: representa o método de resolução de problemas que um programa implementa.
- *Problem*: representa o problema que um programa tenta resolver. Ex: Caminho mínimo e Caixeiro viajante.
- *BaseLineStudy*: define um estudo que é realizado sobre um conjunto especifico de vértices.
- *BaseSolution*: indica o início e o fim de uma solução
- *MarketStudy*: serve para agrupar vários BaseSolutions em função de um BaseLineStudy.
- *PartialSolution*: é uma solução parcial que tem início e fim. Além disso, indica qual a próxima solução parcial.
- *Vertices*: são os vértices do grafo.

A seguir serão descritos os elementos arestas.

- *HasProblem*: relaciona *Program* a *Problem.*
- *HasProgramStudy*: relaciona *Program* a *BaseLineStudy.*
- *HasProblemStudy*: relaciona *Problem* a *BaseLineStudy.*
- *HasBaseLineStudy*: relaciona *MarketStudy* a *BaseLineStudy.*
- *HasMethod*: relaciona *Program* a *Method.*
- *HasSolution*: relaciona *MarketStudy* a *BaseSolution.*
- *HasSolutionMethod*: relaciona *PartialSolution* a *Method.*
- *HasVertex*: relaciona *BaseLineStudy* a um vértice do grafo base. Serve para definir o conjunto de vértices em que o *MarketStudy* deve operar.
- *Begin*: relaciona uma solução a um vértice. Indica qual o vértice de início da solução.
- *End*: semelhante ao *begin*, exceto por indicar o vértice de término da solução.
- *Next*: relaciona um *PartialSolution* a outro. Serve para indicar sequência.
- *Edges*: são as arestas do grafo.

- *PartOf*: relaciona *BaseSolution* as suas *PartialSolutions*. Serve fundamentalmente para que a *BaseSolution* tenha acesso direto a cada uma das *PartialSolution*.

Baseando-se nos elementos e nas caraterísticas citadas anteriormente. O modelo foi nomeado como *Method Program Problem*. Doravante, o denominaremos como modelo M2P.

A principal função dos elementos com prefixo "*has*" é criar uma referência entre um elemento e outro.

É importante notar que os elementos *Method*, *Program* e *Problem* se tratam de abstrações de métodos, programas, e problemas respectivamente. Em outras palavras, isso significa por exemplo que um *Program* não apresenta o código fonte em suas propriedades e sim as características atreladas a ele como versão, data etc.

As contribuições feitas pelos pesquisadores são distinguidas pelo *Problem* que o programa pretende resolver e pelo *BaseLineStudy* que define o conjunto de vértices em estudo. Assim, não existe possibilidade de uma solução ser cogitada melhor do que outra, tendo em vista que ela não resolve o mesmo problema e/ou não opera sobre o mesmo grafo.

Como mencionado, o modelo M2P é formado por quatro categorias de grafo:

- *program*: denota as relações entre métodos, programas e problemas.
- *study*: indica o conjunto ou sub-conjunto de vértices a serem estudados e mantém as referências de todas as soluções geradas.
- *solution*: expressa a solução gerada pelo programa.
- *base*: é de fato o grafo em si com seus vértices e arestas.

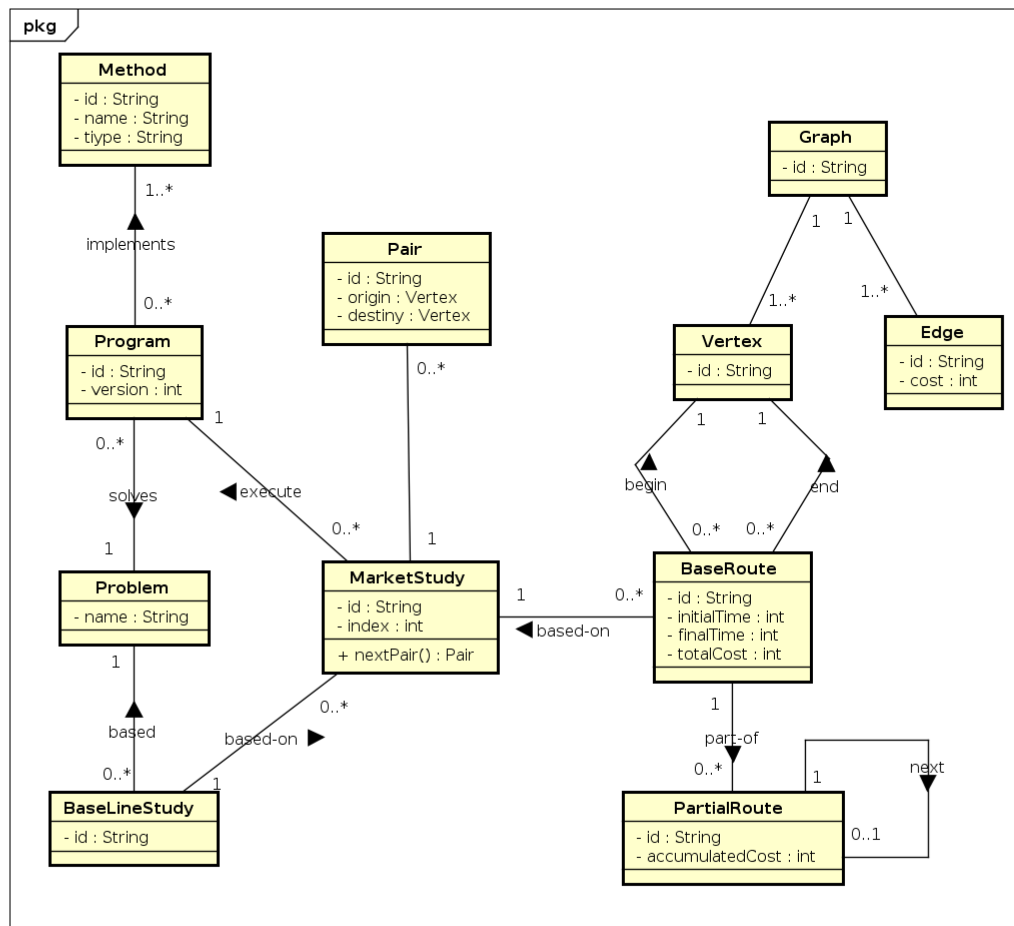
O diagrama de classes apresentado na figura 4 exibe as cardinalidades, funções e atributos de cada elemento. Entretanto, nada impede que mais atributos e funções possam ser adicionadas.

A classe *Pair* é uma propriedade do elemento *MarketStudy* que define um par de vértices: origem e destino. Este par indica uma solução que será gerada pelo programa e é solicitado pelo programa quando está em execução através no método *nextPair*.

A classe *BaseSolution* é responsável por indicar os vértices de início e término de uma solução. Nela contém a informação de tempo do programa e do custo da solução.

A classe *partialSolution* expressa uma aresta que é usada como parte da solução. Nela contém o custo da aresta e custo acumulado do início até ela. Além disso, a *partialSolution* indica qual é a próxima *partialSolution* através do elemento aresta *next*.

Figura 4 – Diagrama de classe do modelo M2P



Fonte: Autoria própria.

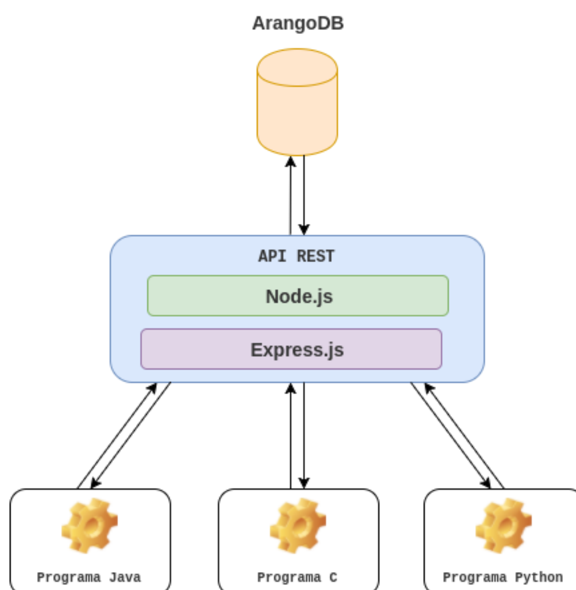
3.2 Implementação

O modelo M2P foi desenvolvido utilizando o bando de dados NoSQL *ArangoDB*. A interação do programa com o *ArangoDB* ocorre através de uma API REST construída em *Node.js* utilizando o *framework Express.js*. A API REST é utilizada para criar uma interface de maneira que os diversos programas acessem os dados armazenados no *ArangoDB*.

A figura 5 exemplifica a arquitetura proposta para o modelo M2P. A arquitetura possibilita que vários pesquisadores, usando qualquer linguagem de programação que possua uma biblioteca HTTP, possam acessar os grafos armazenados no *ArangoDB* por meio da API REST. Desta forma, os pesquisadores em qualquer lugar que tenha acesso a *Internet* podem contribuir com soluções para um mesmo problema em cima de um grafo.

A seção 2.2.2 apresenta dois tipos de coleções existentes no *ArangoDB*: coleção de documentos e coleção de arestas. Cada elemento descrito na seção 3.1 detém uma coleção específica na base de dados do *ArangoDB*. Inclusive, estas coleções possuem mesmo nome

Figura 5 – Diagrama de arquitetura do modelo M2P



Fonte: Autoria própria.

e função dos elementos do modelo.

Abaixo é descrito todas estas coleções categorizadas pelo tipo de coleção.

- Coleção de documentos (vértices): *programs*, *methods*, *problems*, *baseLineStudys*, *marketStudys*, *baseSolutions*, *partialSolutions* e *vertices*.
- Coleções de arestas: *hasBaseLineStudy*, *hasProblem*, *hasProblemStudy*, *hasProgramStudy*, *hasMethod*, *hasSolution*, *hasSolutionMethod*, *hasVertex*, *partOf*, *bebin*, *end*, *next* e *edges*.

Como mencionado na seção 3.1, o modelo M2P é constituído por quatros grafos. Isto é possível devido a capacidade do *ArangoDB* definir grafos sobre um conjunto específico de coleções de vértices e arestas. Por exemplo, o grafo *study* é definido pelas coleções *baseLineStudies*, *marketStudies*, *hasVertex*, *hasProgramStudy*, *hasProblemStudy* e *hasBaseLineStudy*. A tabela 3 exibe as coleções pertencentes a cada grafo.

A coleção *partOf*, responsável por relacionar diretamente uma *BaseSolution* a uma *PartialSolution*, apesar de ser uma aresta redundante, é definida afim de diminuir o tempo de consulta. A figura 6 apresenta dois cenários no qual um grafo possui as arestas *partOf* e o outro não. Percebe-se que se não houvesse as arestas *partOf* seria preciso navegar entre as arestas *next* a procura de um *partialSolution* especifica. Por exemplo, no cenário da

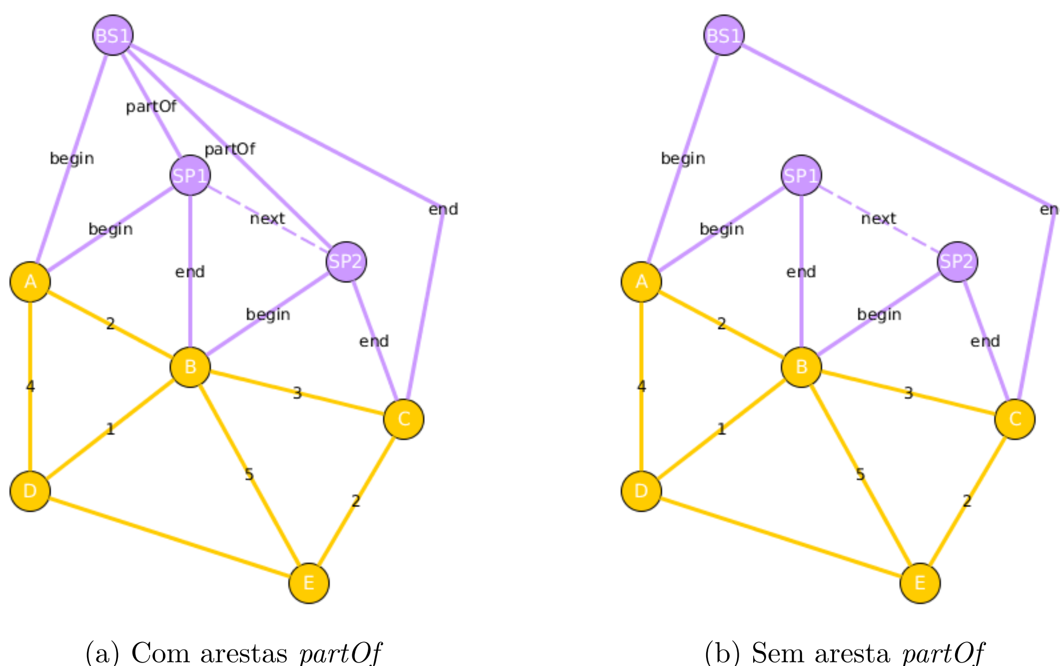
Tabela 3 – Relação entre as categorias de grafos do modelo M2P e as coleções contidas no *ArangoDB*

GRAFO	COLEÇÕES DE VÉRTICES	COLEÇÕES DE ARESTAS
program	*methods*, *programs*, *problems*	*hasProblem*, *hasMethod*
study	*baseLineStudies*, *marketStudies*	*hasVertex*, *hasProgramStudy*, *hasProblemStudy* e *hasBaseLineStudy*
solution	*baseSolutions*, *partialSolutions*	*begin*, *end*, *next* e *partOf*
base	*vertices*	*edges*

Fonte – Autoria própria

figura 6a, caso fosse preciso acessar o vértice *SP2* a partir de *SB1* seria necessário passar por três arestas. Por outro lado, no cenário da figura 6b o acesso é de forma direta.

Figura 6 – Funcionalidade das arestas *partOf*



(a) Com arestas *partOf* (b) Sem aresta *partOf*

Fonte: Autoria própria.

A API REST comunica-se com o *ArangoDB* através do módulo chamado *arangojs* que é o driver JavaScript oficial do *ArangoDB* para *Node.js*. Neste módulo as estão presentes as diversas funcionalidades do *ArangoDB*, bem como manipular as base de dados para acessar as coleções dos grafos e criar consultas.

Como visto na seção 2.3, uma API REST precisa de URIs para acessar, criar, atualizar e excluir seus recursos. A API REST do modelo M2P possui os URI's descritos na tabela 4 juntamente com seus recursos associados. Neste caso, os recurso são as coleções do armazenadas na base de dados do *ArangoDB*.

As URIs citadas na tabela 4 conforme mencionado na seção 2.3 tem as seguintes

Tabela 4 – API REST do Modelo M2P

URI	RECURSO
/vertices	Vértices
/edges	Arestas
/programs	Programas
/problems	Problemas
/methods	Métodos
/marketstudies	MarketStudies
/baselinestudies	BaseLineStudies
/basesolutions	BaseSolutions
/partialsolutions	PartialSolutions

Fonte – Autoria própria

funcionalidades implementadas:

- Obter uma lista dos recursos;
- Obter um recurso dado um *id*;
- Criar um novo recurso;
- Atualizar um recurso dado um *id*;
- Apagar um recurso dado um *id*.

Tabela 5 – Funcionalidades extras da API REST do Modelo M2P

MÉTODO	URI	DESCRIÇÃO
GET	/vertices/neighborhood/{id}	Obtém os vértices vizinhos de um vértice dado um id.
GET	/marketstudies/{id}/pairs	Obtém o próximo par de vértices de um MarketStudy dado um id.
POST	/baselinestudies/hasvertex/{id}	Associa um BaseLineStudy a um vértice

Fonte – Autoria própria

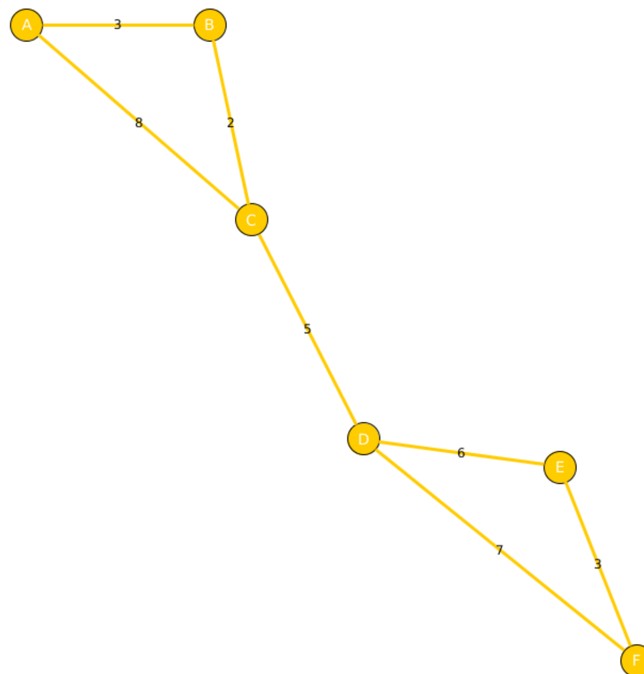
Uma das maiores vantagens desta implementação é a possibilidade de usar somente uma linguagem de programação, o JavaScript. O que também é uma grande vantagem na utilização um banco de dados NoSQL como *ArangoDB* que trabalha diretamente com objetos JSON possibilitando que o JavaScript seja utilizado desde do banco de dados até o servidor.

4 Prova de Conceito

Como consequência do modelo M2P neste capítulo adicionaremos uma extensão ao modelo de forma que pesquisadores contribuam para um mesmo problema. Neste cenário, implementado em Java utilizando a biblioteca Apache HttpComponents Client 4.5.2 para realizar as requisições HTTP à API REST, dois programas de caminho mínimo serão executados em cima de um grafo base (figura 7): um programa que implementa um algoritmo guloso e outro programa que gera um caminho aleatório. Intencionalmente, ambos gerarão soluções não ótimas. A solução do algoritmo guloso por um trecho do caminho terá custo menor comparado ao o algoritmo aleatório e vice-versa. Ao final é gerado uma solução ótima reunindo as melhores partes da solução de cada um dos programas.

Ao decorrer deste estudo, os termos vértice origem e vértice destino são os vértices apontados pela aresta *begin* e *end* respectivamente. As arestas *partOf* entre *BaseSolutions* e *PartialSolutions*, que servem para criar um ligação direta entre as mesmas, foram omitidas para facilitar o entendimento.

Figura 7 – Grafo base

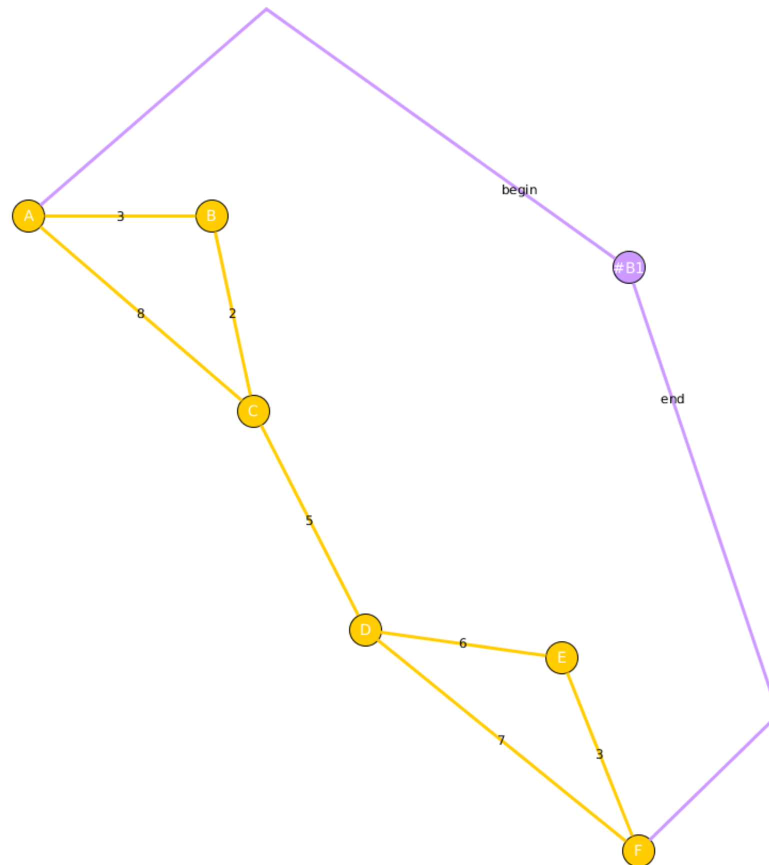


Fonte: Autoria própria.

A figura 7 apresenta um grafo no qual serão gerados duas soluções objetivando o

caminho mínimo tendo como origem o vértice A e destino o vértice F.

Figura 8 – Geração da BaseSolution



Fonte: Autoria própria.

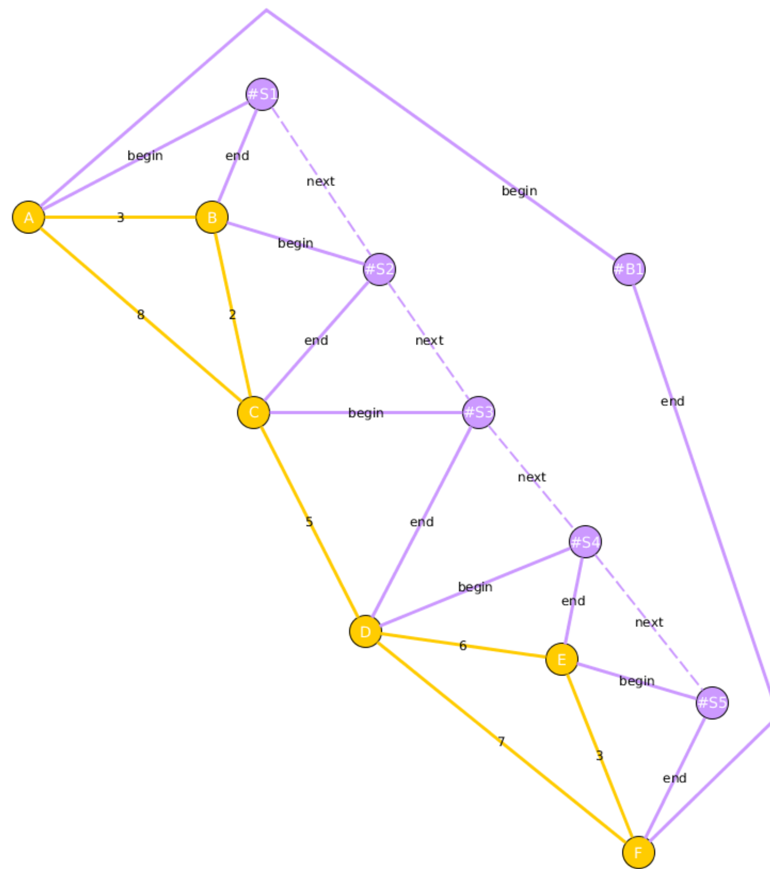
Inicialmente, quando um programa requisita um par de vértice ao *MarketStudy* é criado um *BaseSolution* definido pelo início e fim do par, como ilustrado na figura 8.

A figura 9 exibe a solução gerada pelo algoritmo guloso. O algoritmo guloso é considerado um algoritmo cego porque escolhe sempre a melhor opção local sem analisar vértices posteriores. Neste caso, ele sempre escolherá a aresta com menor custo.

A partir do vértice A, existem duas opções: ir para o vértice C com custo 8 ou para o vértice B com custo 3. A aresta que tem por destino o vértice B possui custo menor e por isso o algoritmo à elege como uma *PartialSolution*, que na figura 9 está representada pelo vértice *#S1*. Em seguida, são escolhidos C, D, E e F.

No momento da criação da *PartialSolution #S4* (figura 9) que possui origem no vértice D, o algoritmo guloso elege aresta que tem por destino o vértice E por dispor de menor custo comparado a aresta com destino F embora, aumente o custo total posteriormente.

Figura 9 – Solução gerada pelo algoritmo guloso



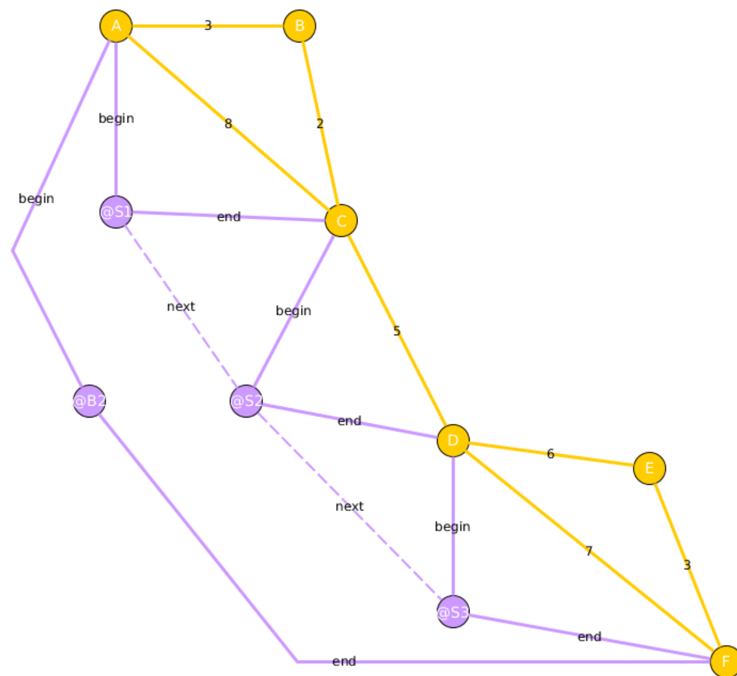
Fonte: Autoria própria.

O segundo algoritmo também teve como origem *A* e destino *F*, porém o caminho é gerado de forma aleatória. O vértice *@B2* refere-se ao *BaseSolution* gerado pelo programa. O caminho gerado é *A*, *C*, *D* e *F* como ilustrado na 10.

A figura 11 apresenta as duas soluções uma ao lado da outra. Percebe-se que do vértice *A* para o vértice *C* a melhor solução é dada pelo programa guloso, de *C* para *D*, ambos programas apresentam a mesma e óbvia solução e de *D* para *F* a melhor solução é dada pelo programa aleatório.

A partir das soluções geradas foi possível a criação de um novo elemento do modelo M2P chamado *HiperSolution*. A *HiperSolution* é passível de ser criada somente quando um programa é executado sobre um mesmo par de vértices, origem e destino, que já tenha uma solução armazenada que fora gerada por outro programa. A *HiperSolution* seleciona as melhores partes de uma solução com finalidade de criar a solução ótima.

Figura 10 – Solução gerada pelo algoritmo aleatório



Fonte: Autoria própria.

4.1 *HiperSolution*

A *HiperSolution* é executada somente quando as soluções geradas tentam resolver um mesmo tipo de problema, neste caso o problema é encontrar o caminho mínimo.

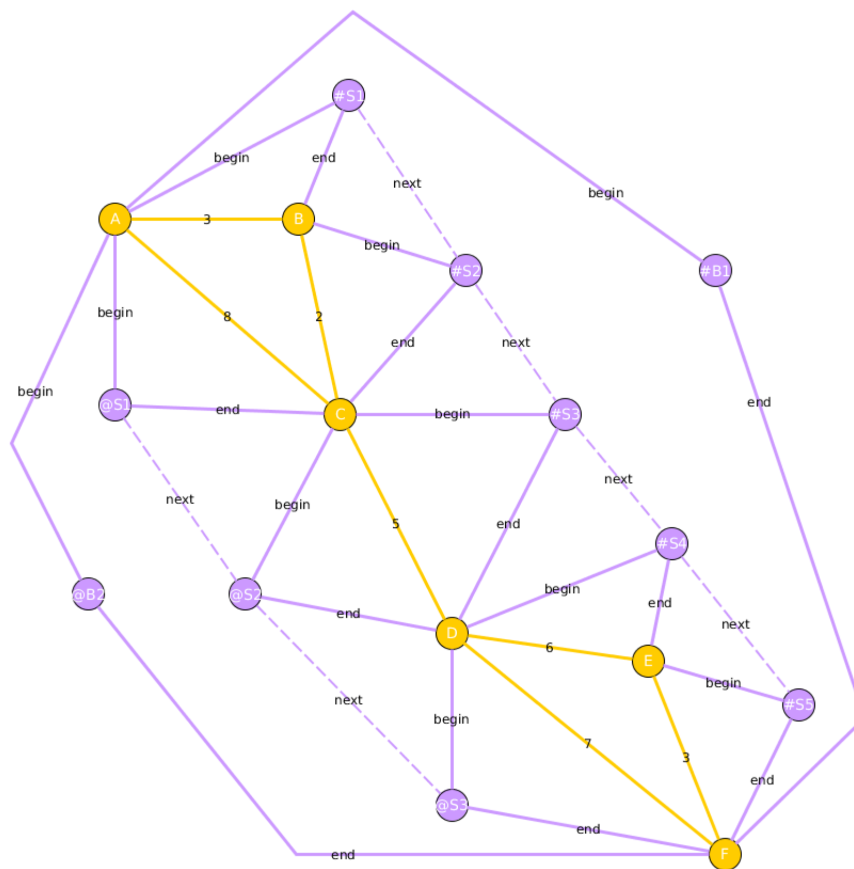
Para a geração da *HiperSolution* fez-se preciso a criação de artifícios que à subsidiassem descritos a seguir:

- Definição de *BaseSolution* ativa e *BaseSolution* corrente;
- Identificação de *PartialSolutions* coincidentes e divergentes.

A *HiperSolution* atua sobre duas soluções: Uma ativa, correspondente a melhor solução encontrada para o objetivo do problema até o momento e uma solução corrente, que denota a solução que fora criada no momento que a *HiperSolution* será executada.

É importante atentar para o fato de que entre duas soluções distintas que resolvem o problema do caminho mínimo podem existir trechos de caminhos iguais ou diferentes quando estas são comparadas. Baseando-se nisto podemos definir dois conceitos: *match* e *mismatch*.

Figura 11 – Grafo base e as soluções geradas



Fonte: Autoria própria.

4.1.1 *Match* e *Mismatch*

A idealização do termo *match* e *mismatch* tem o intuito de localizar trechos de caminhos de menor custo em ambas soluções.

- *match*: é o conjunto de *PartialSolution* coincidentes no grafo de soluções, isto é, possuem mesmo vértice de origem e de destino contiguamente.
- *mismatch*: é o conjunto de *PartialSolution* divergentes, ou seja, possuem vértices de origem ou destino diferentes contiguamente.

Nota-se na figura 11 que as *PartialSolution @S2* e *#S3* são coincidentes, ou seja, elas possuem o mesmo vértice origem(*begin*) e destino (*end*), com a diferença de que pertencem à *BaseSolutions* distintas. Por outro lado, as *PartialSolution @S1* e *#S1* são divergentes, pois não possuem o mesmo vértice origem e destino.

O algoritmo abaixo demonstra a função geral para geração de *HiperSolutions*.

Passo 1. Este passo trata-se das definições das variáveis que serão usadas ao decorrer do algoritmo.

1. Definir A como o conjunto das *PartialSolutions* da *BaseSolution* ativa;
2. Definir B como o conjunto das *PartialSolutions* da *BaseSolution* atual;
3. Definir H como o conjunto das *PartialSolutions* que farão parte da *HiperSolution*;
4. Definir At como a quantidade de elementos de A;
5. Definir Bt como a quantidade de elementos de B;
6. Definir i como índice para iteração sobre A;
7. Definir j como índice para iteração sobre B.

Passo 2. Este passo trata-se da iteração sobre os elementos dos conjuntos A e B à procura de trechos de caminhos coincidentes (*match*) e divergentes (*mismatch*)

1. Enquanto $i < At$ e $j < Bt$ execute os passos 3 e 4.

Passo 3. Este passo é responsável por encontrar um *match* e definir um trecho de caminho padrão já que os trechos são iguais em ambos conjuntos, A e B. Neste caso, é escolhido o trecho de caminho de A como padrão.

1. Se $A[i] = B[j]$ então;
2. Marcar pontos iniciais do *match* com base em nos índices i e j;
3. Enquanto $A[i] = B[j]$, incrementar i e j;
4. Marcar pontos finais do *match* com base nos índices i e j;
5. Com base nos pontos iniciais e finais, adiciona o trecho de A em H.

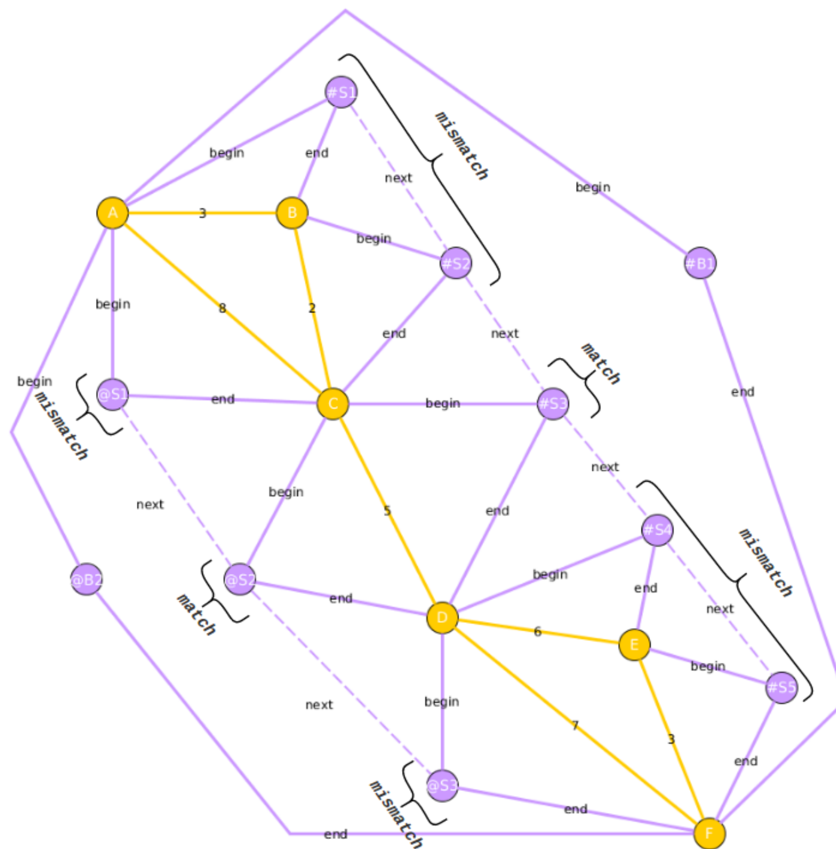
Passo 4. Este passo é responsável por encontrar um *mismatch* e adicionar trecho de caminho com menor custo em H.

1. Se $A[i] \neq B[j]$ então;
2. Marcar pontos iniciais do *mismatch* com base nos índices i e j;
3. Enquanto $A[i] \neq B[j]$, incrementar i e j;
4. Marcar pontos finais do *mismatch* com base nos índices i e j;

5. Com base nos pontos iniciais e finais, calcula custos dos trechos de caminhos de A e B;

6. Adicionar o trecho de menor custo em H.

Figura 12 – *match* e *mismatch* presentes do grafo base



Fonte: Autoria própria.

A figura 12 exemplifica os conceitos de *match* e *mismatch*. As *PartialSolutions* pertencentes a *BaseSolution #B1* formam o conjunto A e as *PartialSolutions* pertencentes a *BaseSolution @B2* formam o conjunto B.

Embora as *PartialSolutions @S1* e *#S1* possuam o mesmo vértice origem, eles não se caracterizam como um *match*, visto que não possuem o mesmo vértice de destino. De modo semelhante, as PartialSolution *@S1* e *#S2* possuem o mesmo destino, contudo não apresentam mesma a origem. Isso se caracteriza como um *mismatch*, pois as *PartialSolutions* possuem origem ou destino diferentes.

Ainda na figura 12, a *PartialSolution @S2* e *#S3* são coincidentes, pois apresentam origem e destino iguais, portanto indicam um *match*.

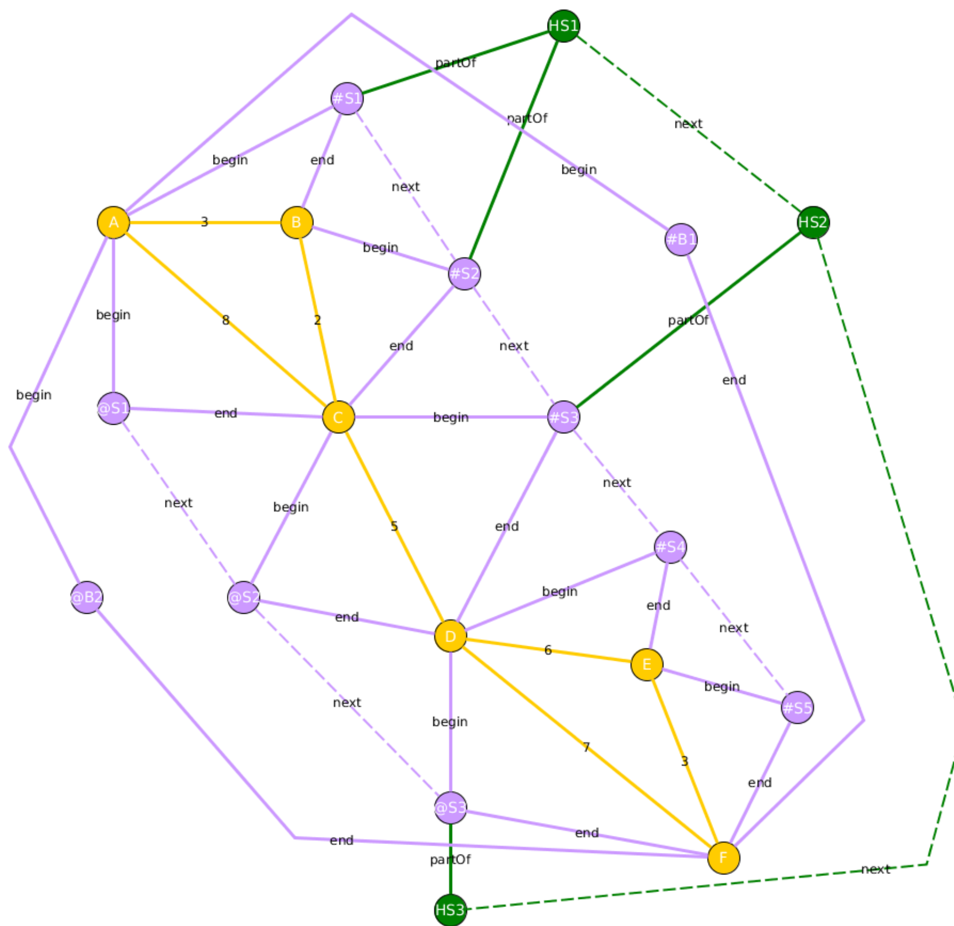
Por fim, há outro *mismatch* entre as *PartialSolutions @S3*, *#S4* e *#S5*.

Um *mismatch* ou um *match* está sempre vinculado a duas soluções. O número de vértices *PartialSolutions* em um *mismatch* podem ser igual ou diferente em ambas soluções. Porém no *match* o número de vértices *PartialSolutions* sempre é igual em ambas soluções.

4.1.2 Resultados

A figura 13 apresenta o comportamento da *HiperSolution* para o grafo em estudo. Um vértice *HiperSolution* é gerado para cada *match* e *mismatch* encontrados no grafo.

Figura 13 – Geração da *HiperSolution*



Fonte: Autoria própria.

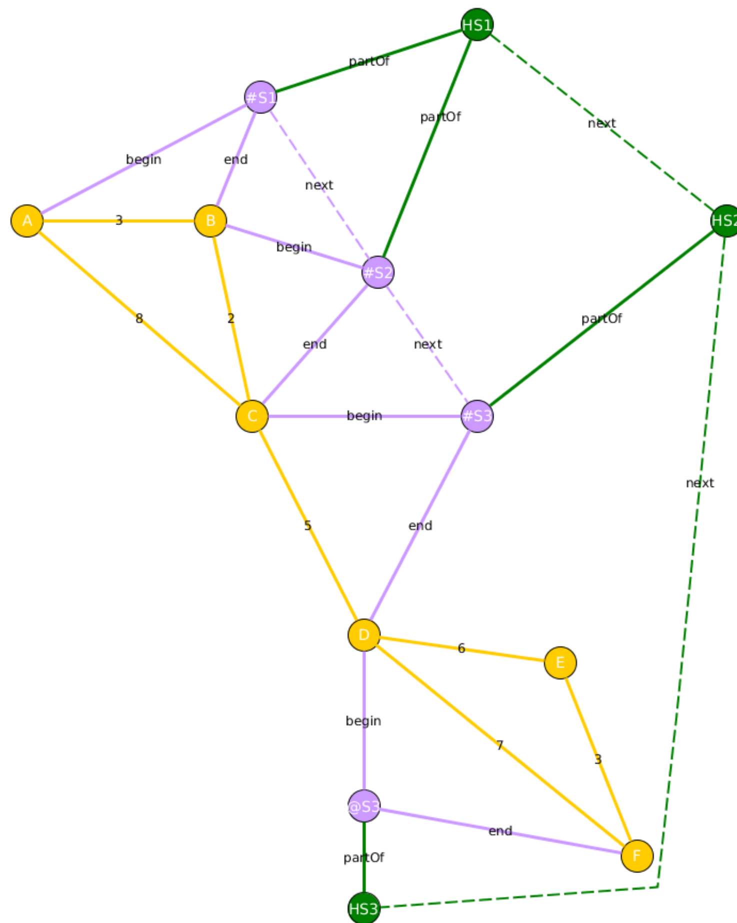
As *PartialSolutions* de cada solução que estão envolvidas em um mesmo *match* possuem sempre o mesmo custo. Portanto, em um *match* qualquer solução pode ser selecionada. Em contrapartida, na maioria das vezes em um *mismatch* o custo é diferente em cada solução, neste caso a solução com menor custo é escolhida.

As arestas *partOf* relacionam a *HiperSolution* à uma *PartialSolution* com intuito de indicar quais soluções parciais serão utilizadas para gerar a solução final. A aresta *next*

funciona da mesma forma que nas *PartialSolutions*, isto é, indicam a sequência entre as *HiperSolutions*.

No grafo apresentado na figura 14 é ilustrado o grafo solução referenciado pelas arestas *partOf*. Os vértices *PartialSolutions* não pertencentes as *HiperSolutions* geradas são omitidos a fim de exemplificar a solução ótima.

Figura 14 – Grafo solução referenciado pela HiperSolution

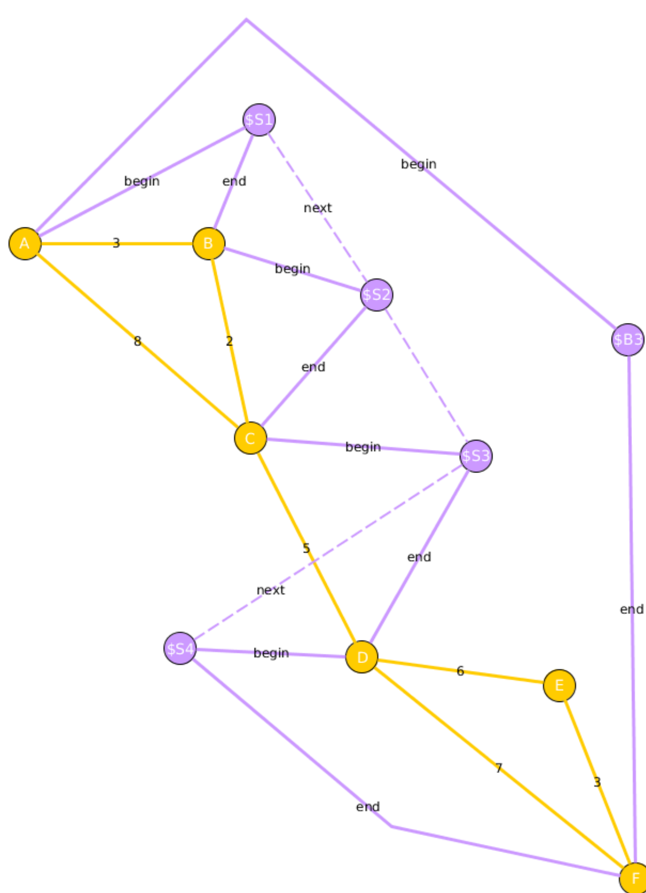


Fonte: Autoria própria.

Por fim, é gerada uma nova solução criando uma *BaseSolution* copiando todos os vértices *PartialSolutions* referenciados pelas *HiperSolutions* como exemplifica a figura 15. Esta torna-se a nova *BaseSolution* ativa que seria mesclada com outra solução contribuída por outro pesquisador usando um método qualquer.

Neste estudo, abordou-se a criação de um algoritmo que analisasse as soluções submetidas pelos programas de caminho mínimo. Isto foi possível em consequência das funcionalidades que o modelo e arquitetura proporcionou.

Figura 15 – Nova solução



Fonte: Autoria própria.

5 Considerações Finais

Este trabalho proporcionou um modelo de dados e uma arquitetura que permite que os pesquisadores executem seus algoritmos e submetam suas soluções através de uma API REST para uma base de dados disponível na web. Além disso, possibilitou uma estrutura de dados em que as soluções armazenadas podem servir para comprovação de sua autenticidade e comparação.

Uma desvantagem está no fato da arquitetura para ser utilizada requerer que o pesquisador elabore funções em sua linguagem de programação para acessar a API REST. Para isto, é preciso que a linguagem escolhida disponha de uma biblioteca HTTP para realizar as requisições.

O estudo realizado no capítulo 4 apresenta limitações, bem como, abordar somente problemas de roteamento e apenas analisar as distâncias entres os vértice. Apesar de existir algoritmos polinomiais para resolução do caminho mínimo, o capítulo 4 evidencia as funcionalidades e possibilidades que o modelo oferece. Além disto, é apresentado a criação de um coordenador genérico para problemas de caminho mínimo. Através dos conceitos *match* e *mismatch* foi possível que o sistema se comportasse como um método hiper-heurístico por meio da comparação de resultados, e deste modo, elegendo trechos de menor caminhos entre as soluções armazenadas.

Por fim, a realização desse trabalho faz surgir novas possibilidades de pesquisas futuras, como a aplicação do modelo para problemas de otimização dos quais não existem algoritmos polinomiais que o resolvam, como problema do caixeiro viajante. A caracterização dos conceitos *match* e *mismacth* pode ser estudada para avaliar se eles ajustam-se aos diversos problemas de roteamento ou quais alterações seriam necessárias para adequarem-se a este tipo de problema.

O elemento *Problem* poderia possuir regras em suas propriedades que quando instanciadas restringissem as soluções à medida que são resolvidas. Deste modo, o pesquisador não cometeria equívocos à respeito da solução gerada se de fato cumpriu seu objetivo. Por exemplo, o problema do caixeiro viajante não permite que um vértice seja visitado mais de uma vez. No momento em que as soluções fossem armazenadas o elemento *Problem* verificaria se as regras não foram infringidas.

Para suprir a inconveniência de escrever código para acessar a API REST, poderiam ser criadas bibliotecas de acesso ao modelo em linguagens comumente usadas pelos pesquisadores.

O modelo M2P demonstra a importância da disposição de uma estrutura eficiente

para armazenamento de grafos e a interação entre os pesquisadores para resolução de problemas de otimização do mundo real.

Referências

ABADI, D. J.; BONCZ, P. A.; HARIZOPOULOS, S. Column-oriented database systems. *Proceedings of the VLDB Endowment*, VLDB Endowment, v. 2, n. 2, p. 1664–1665, 2009.

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. *ACM Computing Surveys (CSUR)*, ACM, v. 40, n. 1, p. 1, 2008.

BRETERNITZ, V. J.; SILVA, L. A. Big data: Um novo conceito gerando oportunidades e desafios. *Revista Eletrônica de Tecnologia e Cultura*, v. 2, n. 2, 2013.

BURKE, E. et al. Hyper-heuristics: An emerging direction in modern search technology. In: *Handbook of metaheuristics*. [S.l.]: Springer, 2003. p. 457–474.

CANTELON, M. et al. *Node. js in Action*. [S.l.]: Manning, 2014.

COULOURIS, G. F.; DOLLIMORE, J.; KINDBERG, T. *Distributed systems: concepts and design*. [S.l.]: Pearson Education, 2012.

DOGLIO, F. *Pro REST API Development with Node.js*. [S.l.]: Apress, 2015.

FIELDING, R. T. *Architectural styles and the design of network-based software architectures*. Tese (Doutorado) — University of California, Irvine, 2000.

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação: um tratamento moderno de matemática discreta*. [S.l.]: Livros Técnicos e Científicos, 2004.

HAHN, E. M. *Express in Action Writing, building, and testing Node.js applications*. [S.l.]: Manning, 2016.

HUGHES-CROUCHER, T. H. *Up and Running with Node.js*. [S.l.]: O'Reilly, 2010.

KUROSE, J. F. et al. *Redes de Computadores e a Internet: uma abordagem top-down*. [S.l.]: Pearson Education, 2013.

LEAVITT, N. Will nosql databases live up to their promise? *Computer*, IEEE, v. 43, n. 2, 2010.

MASSE, M. *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces*. [S.l.]: "O'Reilly Media, Inc.", 2011.

MCAFEE, A. et al. Big data. *The management revolution. Harvard Bus Rev*, v. 90, n. 10, p. 61–67, 2012.

MILLER, J. J. Graph database applications and concepts with neo4j. In: *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*. [S.l.: s.n.], 2013. v. 2324, p. 36.

NAYAK, A.; PORIYA, A.; POOJARY, D. Type of nosql databases and its comparison with relational databases. *International Journal of Applied Information Systems*, v. 5, n. 4, p. 16–19, 2013.

NOSQL. *NOSQL Databases*. 2009. Disponível em: <http://nosql-database.org/>.

PAPADIMITRIOU, C. H.; STEIGLITZ, K. *Combinatorial optimization: algorithms and complexity*. [S.l.]: Dover Publications, 1998.

PEDIAPRESS. *Computer Science, An Overview*. [S.l.]: PediaPress, 2011.

PENTEADO, R. R. et al. Um estudo sobre bancos de dados em grafos nativos. *X ERBD-Escola Regional de Banco de Dados*, 2014.

ROBINSON, I.; WEBBER, J.; EIFREM, E. *Graph databases: new opportunities for connected data*. [S.l.]: O'Reilly Media, Inc., 2015.

SILBERSCHATZ, A. et al. *Sistema de banco de dados*. [S.l.]: Elsevier, 2006.

SOUSA, F. P. de. Criação de framework rest/hateoas open source para desenvolvimento de apis em nodejs. 2015.

SUCUPIRA, I. R. *Um Estudo Empírico de Hiper-Heurísticas*. Tese (Doutorado) — Universidade de São Paulo, 2007.

VASILYEVA, E. et al. Answering "why empty?" and "why so many?" queries in graph databases. *Journal of Computer and System Sciences*, Elsevier, v. 82, n. 1, p. 3–22, 2016.

VIEIRA, M. R. et al. Bancos de dados nosql: conceitos, ferramentas, linguagens e estudos de casos no contexto de big data. *Simpósio Brasileiro de Bancos de Dados*, 2012.

ZÄPFEL, G.; BRAUNE, R. *Metaheuristic search concepts: A tutorial with applications to production and logistics*. [S.l.]: Springer Science & Business Media, 2010.

ZIKOPOULOS, P. et al. *Harness the power of big data The IBM big data platform*. [S.l.]: McGraw Hill Professional, 2012.