Modelo para o artigo do Módulo 5

André Luís Lessa Junior, Arthur Alberto Cardoso Reis, Cristiane Andrade Coutinho, Giovan

Janeiro de 2023

Abstract

Como parte das atividades do módulo 5, cada grupo deverá redigir um texto descrevendo os resultados do projeto no formato de um artigo científico. Este arquivo no formato markdown contém a estrutura básica deste artigo. Cada grupo deverá editar este arquivo com a descrição do projeto que desenvolveu.

Introdução

O meio militar envolve operações que visam o planejamento estratégico, a fim de definir uma direção ou caminho almejado pela corporação e que gastos de recursos sejam minimizados e a eficiência do trabalho para alcançar o objetivo seja maximizada(Asplan,2011). Para que esse planejamento estratégico consiga êxito no futuro, diversas empresas ou autarquias necessitam de tecnologias que potencializem essa ideia de estratégia, além de desempenhar um papel na persecução dos objetivos organizacionais e garantir o alinhamento da empresa com esses mesmos objetivos(PETI,2019). Dentro do planejamento estratégico existem diversas missões que necessitam das mais variadas logísticas e estratégias, entre elas existem missões aeroespaciais de reconhecimento, resgate e ataque em percursos próximos ao solo. Assim sendo, uma das ferramentas aeroespaciais militares utilizadas nesse contexto é o radar *Terrain-Following*, permitindo que o piloto, ao fazer o reconhecimento de uma região (muitas vezes desconhecida), voe em baixa altitude de modo constante, ou seja, sem variação da elevação da aeronave. A importância desse sistema se deve ao impedimento de uma possível detecção de um radar inimigo, além de facilitar o reconhecimento de um alvo ou até mesmo compreender a estrutura do relevo local.

Entretanto, a tecnologia *Terrain-following* necessita de segurança e ponderações para a sua utilização durante as missões aéreas. O sistema se baseia na fusão de dados provenientes do mapeamento radar e de um banco de dados de terreno e obstáculos(Portal de Comunicações AEl,2023). Assim sendo, é de suma importância que ocorra um reconhecimento prévio da região e a escolha de um melhor caminho possível. Todavia, a tecnologia *Terrain-following* não supre essas necessidades de reconhecimento, gerando uma desvantagem inerente ao sistema. Portanto, o problema de pesquisa está relacionado à geração de trajetória,

durante o evento do *Terrain-Following*, representada por grafos, no espaço bidimensional. Esses pontos representam uma região que aeronave deve passar ou não e possui atributos geográficos reais que facilitam sua localização(latitude e longitude), além da visualização de rotas viáveis e inviáveis, a fim de fornecer uma orientação para o piloto durante a viagem. O cálculo desse percurso atribui ao desenvolvimento de um modelo matemático computacional que, através do conhecimento prévio da região (dados armazenados nesse banco), construa possíveis rotas que levem o avião de um ponto ao outro e especifique o caminho preferível para o seu deslocamento. Existem muitos trabalhos que aboradaram sobre tema, priciplamente relacionado às áreas de logística e transporte e utilizava o conhecimentos de algoritmo de *Dijkstra* e estrutura de dados para otimizar o transporte público de Criciúma/SC (A L GUIZZO; C P ENGELMANN; J G ZANNETE; SIMÕES P.W.T.A; G V SILVA; C V SCARPATO,2015). O nosso trabalho contem similiaridades com o descrito anteriormente, contudo com ênfase na área de defesa, visto que a criação do algoritmo propicia que empresas e as forças armadas realizem o *Terrain-following*, em conjunto com a tecnologia planejadora de voos, sem a preocupação de não percorrerem um trajeto viável. O resultado esperado é que o *software* consiga destacar uma ou mais trajetórias viáveis através de dados de *input* das cidades de saída e chegada feitos pelo usuário.

Como mencionado anteriormente, o projeto envolve a compreensão de algoritmo e estrutura de dados e enteder o funcionamento basilar de grafos e estrutura de dados, e como esses conceitos fundamentais em Ciência da Computação se relacionam com o projeto (JUDITH GERSTING,2017). Uma das estrutura de dados mais utilizadas para problemas de caminhos mínimos é a fila de prioridades(ou *heap*) que permite inserir elementos com uma determinada prioridade e remover o elemento de maior (ou menor) prioridade. Nos algoritmos de caminhos mínimos, a prioridade é determinada pela distância (ou custo) do caminho atual até o vértice adjacente(CARLA NEGRI LINTZMAYER;GUILHERME OLIVEIRA MOTA,2020). Ademais,para que tal solução fosse viável, os principais materiais que foram utilizados para a construção do algoritmo de alta performance, capaz de identificar o melhor trajeto possível, são a linguagem de programação Java, necessária para a construção desses modelos computacionais, o banco de dados NoSql, que contém as informações de relevo da região diponibilizadas pela empresa AEL. Desse modo, a linguagem java com o *framework Spring Boot Web* como arquitetura MVC (*Model*, *View*, *Control*) para a realização do CRUD (*Create*, *Read*, *Update*, *Delete*), nescessário para manipular a estrutura do banco de dados orientado à grafos através de uma *API RESTful*. Além disso, esses dados estão na extensão de arquivo DTED e por isso se faz necessário o uso da biblioteca *Geospatial Data Abstraction Library* (GDAL), disponível na linguagem Java, capaz de extrair essas informações geográficas. De tal maneira, convertemos os dados do arquivo DTED com essa biblioteca e convertemos em objetos Java. Por outro lado, para o NoSql, utilizamos outro banco de dados denominado Neo4j, orientado a grafos (manipulação dos dados através da linguagem Cypher), reponsável pela modelagem matemática e a construção

do raciocínio algorítmico dos fluxos. Por fim, para a visualização das rotas e a identificação do melhor caminho possível, utilizamos a biblioteca do Javascript chamada D3js. Desse modo, para que tais ferramentas fossem possíveis, todas essas tecnologias foram hospedadas através da aplicação Docker que isola as mesmas em contêineres únicos, justamente para garantir a padronização do ambiente de desenvolvimento.

No entanto, é de suma importância a compreensão do funcionamento do algoritmo e que tipo de modelo foi desenvolvido para construção lógica do projeto planejador de voos. Assim sendo, utilizamos um algoritmo que é conhecido por muitos estudiosos na área da computação. Ele foi elaborado Edsger *Dijkstra*, cientista da computação holandês, no final da década de 50 e até hoje é aplicado para propor rotas mais otimizadas. O algoritmo de *Dijkstra* começa em um vértice de origem e visita todos os outros vértices do grafo para encontrar o caminho mais curto de origem para todos os outros vértices. Ele mantém uma lista de distâncias conhecidas a partir da origem para cada vértice do grafo, que são atualizadas à medida que o algoritmo avança. O algoritmo também mantém uma lista de vértices que ainda precisam ser visitados, e a cada iteração, escolhe o vértice com a menor distância conhecida para visitar em seguida.O algoritmo de *Dijkstra* funciona apenas para arestas ponderadas não negativas e é implementado usando uma fila de prioridade para manter a lista de vértices a serem visitados, o que torna o algoritmo eficiente em termos de tempo. Ele é amplamente utilizado em aplicações práticas, como sistemas de roteamento em redes de computadores, sistemas de navegação em mapas e jogos de estratégia. Todavia, o algoritmo de *Dijkstra* possui alguns problemas que no ponto de vista do projeto foram negativos pelo seu uso em específico. O funcionamento da aplicação são as requisições que o usuário faz no *front-end* e essas mesmas requisições são passadas para o *SPRINGBOOT* pelo protocolo *Rest*. Se é a primeira vez que a aplicação está sendo rodada, o usuário terá que popular no banco de dados com as informações do arquivo *DTED*. Para popular no banco de dados é utilizada a biblioteca *GDAL* e a aplicação manda os dados manipulados para o *Neo4j* que retorna um objeto *Json* para a aplicação no *SpringBoot* que, por sua vez, envia para o *front-end*. Existiam muitos dados no arquivo e a repesentação das regiões por nós desse grafo eram muito pesadas e exigiam uma alta capaciadade de processamento, assim sendo a preferência por um outro algoritmo que possua a mesma lógica do Dijkstra e resolvesse os problemas de consumo de tempo era considerável, portanto, o algoritmo escolhido foi o _A*_(A-star) com algumas determinações heurísticas.

De maneira mais precisa, nos baseamos no _Hierarchical A_ *que é uma variante do algoritmo A* que usa uma estrutura de grafo hierárquico para melhorar o desempenho e a eficiência do _A*_. Esse algoritmo possui a capacidade de suprir as necessidades do algoritmo de *Dijkstra*. Dessa maneira, as principias vantagens do _A*_ sobre o *Dijkstra* são eficiência, visto que pode ser mais rápido do que o algoritmo de *Dijkstra* em grafos grandes. Isso ocorre porque o _H-A*_ usa uma estrutura de grafo hierárquico para reduzir o número de nós visitados. Isso pode levar a uma economia significativa de tempo em comparação com o algoritmo de

Dijkstra, especialmente em grafos grandes e complexos. Também exige menos memória, pois o _H-A*_ usa uma estrutura de grafo hierárquico, ele pode ser mais eficiente em termos de uso de memória do que o algoritmo de *Dijkstra*. Isso ocorre porque o _H-A_ *armazena informações sobre subgrafos em vez de armazenar informações sobre cada nó individualmente. A terceira vantagem é a adaptabilidade do algoritmo, já que O H-A* é mais adaptável do que o algoritmo de *Dijkstra*. Isso ocorre porque o _H-A_ *pode ser facilmente ajustado para lidar com diferentes tipos de heurísticas, enquanto o algoritmo de* Dijkstra *usa apenas uma medida de distância (peso da aresta). Por último existe a pecisão do modelo, que geralmente é mais preciso do que o algoritmo de* Dijkstra*, pois usa uma heurística que leva em consideração informações adicionais sobre o grafo (CARLA NEGRI LINTZMAYER;GUILHERME OLIVEIRA MOTA,2020). A heurística usada pelo H-A* pode ser personalizada para levar em conta fatores como a topologia do grafo, a localização do destino e a natureza das arestas.

Motivação

A gama de utilidades de voos de baixa altitude é bastante grande, abrangindo desde a aplicação de fertilizantes e defensivos agrícolas em grandes extensões de plantio até operações militares de busca, resgate, reconhecimento de terreno, dentre outras missões. Por sua vez, a operação desse tipo de voo apresenta riscos inerentes de colisão com o solo ou CFIT do inglês Controlled Flight Into Terrain (DA COSTA, 2019). Atualmente, a empresa parceira do projeto, AEL Sistemas, provê sistemas de Terrain Following, todavia o planejamento de uma rota possível é feita à mão. A elaboração dessa rota pode ser caracterizada como o problema do Caminho Mínimo da teoria de grafos, tendo como objetivo encontrar a rota com menor peso entre dois vértices (inicial e final) em um dado grafo G com pesos. Para suprir a lacuna de planejamento de rotas de voos de baixa altitude não automatizado pela AEL e, consequentemente, diminuir os riscos de colisão com o solo, a presente pesquisa trouxe um sistema que automatiza todas essas funções, desde carregar um mapa suprido pelo banco de dados, gerar um grafo a partir de suas coordenadas geográficas, adicionar pesos às arestas e, por fim, apontar o melhor caminho mínimo para a missão. # Descrição do problema

Trabalhos relacionados

Descrição da estratégia adotada para resolver o problema

Análise da complexidade da solução proposta

A complexidade de um algoritmo é uma medida de quão "difícil" ou "custoso" ele é em termos de recursos computacionais, como tempo e espaço de memória. Em

outras palavras, a complexidade de um algoritmo é uma medida da quantidade de recursos que ele precisa para resolver um determinado problema, e essa medida geralmente é expressa como uma função do tamanho da entrada do problema.

Existem diferentes tipos de complexidade, como a complexidade de tempo e a complexidade de espaço, que medem a quantidade de tempo ou espaço de memória necessária para executar o algoritmo. A complexidade de tempo é geralmente medida em termos do número de operações básicas executadas pelo algoritmo em relação ao tamanho da entrada do problema. Já a complexidade de espaço é medida em termos da quantidade de espaço de memória necessário para armazenar os dados usados pelo algoritmo em relação ao tamanho da entrada do problema.

A análise de complexidade é uma etapa importante na avaliação de algoritmos, pois permite comparar diferentes algoritmos para um mesmo problema e escolher o que é mais adequado para cada situação. Algoritmos com menor complexidade geralmente são mais eficientes em termos de tempo e espaço de memória, o que é especialmente importante em aplicações que lidam com grandes volumes de dados ou que exigem uma resposta rápida. O modelo utilizado para o desenvolvimento de nosso projeto é o algoritmo _A*_, que é utilizado para encontrar o caminho mais curto entre dois pontos em um grafo ponderado. Ele é uma extensão do algoritmo *Dijkstra*, com a adição de uma heurística que estima a distância restante para o destino.

Antes de explicar a análise da complexidade do algoritmo em si, precisamos entender o que leva o algoritmo _A*_ ter uma capacidade de processamento de memória mais rápido ou mais demorado, dependendo de algumas circunstâncias. O algoritmo utiliza uma estrutura de dados, que veremos posteriormente, chamada fila de prioridades. Essa estrutura de dados organiza seus elementos de acordo com uma determinada ordem de prioridade. Em uma fila de prioridades, os elementos com maior preferência são sempre os primeiros a serem processados (STUART JONATHAN RUSSEL e PETER NORVIG, 2010). A seguir, segue um passo a passo do funcionamento de uma fila de prioridades: 1. Inicialização: A fila de prioridades é criada e inicializada vazia. 2. Inserção: Quando um novo elemento é inserido na fila, ele é adicionado de acordo com sua prioridade. Os elementos com maior prioridade são colocados no topo da fila e os elementos com menor prioridade são colocados no final. 3. Remoção: Quando um elemento é removido da fila, o elemento com maior prioridade é sempre o primeiro a ser removido. Após a remoção, os elementos restantes são reorganizados para manter a ordem de prioridade. 4. Atualização: Em alguns casos, o valor de um elemento na fila de prioridades pode mudar. Quando isso acontece, a posição do elemento na fila também deve ser atualizada para manter a ordem de prioridade. Isso é feito removendo o elemento da fila e inserindo-o novamente com seu novo valor. 5. Consulta: É possível consultar o elemento de maior prioridade na fila sem removê-lo. Isso é útil quando se quer saber qual o próximo elemento que será processado. 6. Tamanho: É possível saber o tamanho atual da fila, ou seja, quantos elementos estão armazenados na fila.

Outro ponto importante a se comentar estão nas heurísticas, que são funções que ajudam a estimar a distância ou o custo de chegar de um determinado nó a um objetivo final em um grafo ou rede. As heurísticas são uma parte importante do algoritmo _A*_, pois ajudam a determinar a próxima etapa de busca a ser tomada, com base na distância ou no custo estimado de chegar ao objetivo. No algoritmo _A*_, a heurística é usada para estimar o custo total de alcançar o objetivo a partir de um determinado nó. Isso é feito adicionando o custo real (calculado) de chegar ao nó atual com uma estimativa heurística do custo de chegar ao objetivo. O resultado é um valor que representa o custo total estimado de chegar ao objetivo a partir do nó atual (STUART JONATHAN RUSSEL e PETER NORVIG, 2010). A heurística usada no algoritmo _A*_ deve atender a dois requisitos: 1. A heurística deve ser admissível, o que significa que ela não pode superestimar o custo de chegar ao objetivo. Em outras palavras, a heurística deve sempre subestimar o custo total de chegar ao objetivo a partir do nó atual. 2. A heurística deve ser consistente, o que significa que o custo estimado de chegar a qualquer nó sucessor deve ser menor ou igual ao custo estimado de chegar ao nó atual mais o custo real de mover-se do nó atual para o sucessor.

A fórmula de Haversine é uma das principais heurísticas que utilizamos para calcular a distância geodésica entre dois pontos na superfície da Terra. Essa heurística é particularmente útil em aplicações que envolvem roteamento de veículos, como sistemas de navegação por GPS. O comportamento da heurística de Haversine no algoritmo _A*_ é semelhante a outras heurísticas usadas no algoritmo. A heurística de Haversine é usada para estimar o custo mínimo de movimento do nó atual ao objetivo final. Ela calcula a distância geodésica entre o nó atual e o objetivo final (Stuart Jonathan Russell e Peter Norvig, 2010). O nosso projeto envolve justamente a criação de melhores rotas para que o avião percorra uma determinada região com mais facilidade.

Continuando com a descrição de outro conceito primordial que envolve o funcionamento do algoritmo, temos o fator de ramificação. Ele é denominado pelo número médio de sucessores de cada nó em um grafo, além de ser um indicador da complexidade do próprio grafo, e afeta diretamente o desempenho do algoritmo _A*_. Quanto maior o fator de ramificação, mais difícil é encontrar o caminho ótimo, pois há mais vértices para serem explorados. Por isso, a escolha da heurística pode ser crucial para o desempenho do algoritmo em grafos com diferentes fatores de ramificação. O fator de ramificação também pode afetar a capacidade de processamento e memória do algoritmo _A*_. Quanto maior o fator de ramificação, mais vértices o algoritmo precisa avaliar, o que aumenta o tempo de processamento e a quantidade de memória necessária para armazenar as informações da busca. Além disso, o fator de ramificação também pode afetar a qualidade da solução encontrada pelo algoritmo. Quando o fator de ramificação é muito alto, pode ser difícil encontrar a solução ótima em um tempo razoável. Isso ocorre porque o algoritmo _A*_ pode ficar preso explorando caminhos que não levam a uma solução, o que é conhecido como “ramificação falsa”. Quando o fator de ramificação é muito alto, muitos nós serão adicionados

à fila de prioridade, aumentando seu tamanho e, consequentemente, o tempo de processamento do algoritmo. Além disso, quando muitos nós são adicionados à fila de prioridade, pode haver um aumento na possibilidade de que o algoritmo percorra caminhos desnecessários, o que também leva a um aumento no tempo de execução e na utilização de memória (STUART JONATHAN RUSSEL e PETER NORVIG, 2010).

O algoritmo _A*_ é um algoritmo clássico e está presente na literatura da ciência da computação . De acordo com muitos estudos e pesquisas de diversos autores e especialistas em algoritmos de caminhos mínimos, como é o caso dos autores Stuart Jonathan Russell e Peter Norvig que publicaram um livro em 2010 e intitulado como "*Artificial Intelligence: A Modern Approach*", podemos afirmar que sua complexidade assintótica representa o tempo e espaço de um algoritmo à medida que o tamanho de entrada se aproxima do infinito, e é dada por $O(nlogn)$. Essa complexidade se caracteriza pela sua taxa de crescimento, que é proporcional a n multiplicado por uma função logarítmica de n. Em outras palavras, a complexidade cresce de maneira logarítmica com o aumento do tamanho de entrada n. Essa função logarítmica é baseada em algoritmos que dividem o problema em subproblemas menores, geralmente pela metade, e resolvem cada subproblema recursivamente. Esse tipo de algoritmo também é comum em algoritmos de ordenação, como o *merge sort* e o *quicksort*, bem como em algoritmos de busca, como o próprio algoritmo _A*_. As características matemáticas da complexidade assintótica $O(nlogn)$ incluem o fato de que o tempo de execução aumenta de forma logarítmica com o tamanho da entrada n, o que significa que, para entradas grandes, o tempo de execução é relativamente pequeno em comparação com complexidades de ordem superior, como $O(n^2)$ e $O(n^3)$, mas possui um tempo de execução maior e pode não ser tão eficiente comparado à complexidade assintótica $O(n)$, $O(logn)$ e $O(1)$. Ainda assim, a complexidade $O(nlogn)$ é considerada bastante eficiente, com um tempo de execução razoável para a maioria dos problemas práticos.

Diante dessas informações, podemos descrever o comportamento da complexidade assintótica do algoritmo para o pior e melhor caso. Com relação ao pior caso, se a heurística for admissível e consistente, sua complexidade é propriamente $O(nlogn)$. Vale ressaltar que utilizamos a notação O para o estudo do pior caso (THOMAS H. COEMEN, CHARLES E. LEISERSON, RONALD L. RIVEST e CLIFFORD STEIN, 2012). Isso significa que se o ponto de partida do nosso algoritmo estiver em uma ponta inferior da região delimitada (área que representamos por um retângulo) e o ponto de chegada estiver na outra extremidade superior do retângulo, significa que teremos a maior distância possível da região delimitada. Além disso, considerando o fator de ramificação do grafo e se o algoritmo _A*_ que calcula a melhor rota tiver um fator de ramificação médio de 3 (cada nó possui em média 3 filhos), mais dados são armazenados na fila de prioridades. Assim sendo, tendo a maior distância possível e com fator de ramificação 3, o gráfico dessa função cresce ao máximo possível para esse caso, e portanto, tende à própria função e a própria complexidade assintótica do algoritmo $O(nlogn)$.

O melhor caso para o algoritmo _A*_ ocorre quando o nó de destino é o primeiro nó visitado pelo algoritmo. Nesse caso, o algoritmo encontra a solução sem precisar expandir nenhum outro nó, o que resulta em uma complexidade de tempo constante $\Omega(1)$ (representação do melhor caso através da letra grega *Omega*). Ou seja, há apenas um valor na entrada (THOMAS H. COEMEN, CHARLES E. LEISERSON, RONALD L. RIVEST e CLIFFORD STEIN, 2012). No entanto, é importante destacar que esse é um caso muito específico e pouco comum na prática. Na maioria dos casos, o algoritmo _A*_ precisará expandir vários vértices antes de encontrar a solução. Além disso, a qualidade da heurística utilizada pode afetar significativamente o desempenho do algoritmo, mesmo nos melhores casos.

Análise de corretude do algoritmo

A corretude de um algoritmo é a garantia de que ele produz o resultado correto para todos os casos de entrada possíveis. Em outras palavras, um algoritmo é considerado correto se ele atende a sua especificação para todas as entradas possíveis. A análise da corretude do algoritmo é uma das principais preocupações dos cientistas da computação durante o processo de desenvolvimento de um algoritmo (Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest e Clifford Stein, 2012).

A invariante do laço é uma técnica usada na análise da corretude do algoritmo, que consiste em encontrar uma propriedade que permanece verdadeira em todas as iterações do laço do algoritmo. A invariante do laço é uma expressão matemática que descreve a propriedade e é usada para demonstrar que o algoritmo atende a sua especificação. Essa técnica é amplamente utilizada na análise de algoritmos, especialmente em algoritmos iterativos, como ordenação, busca e muitos outros. Em nosso caso, como vimos anteriormente, aplicaremos essa técnica no algoritmo A*.

A invariante do laço do algoritmo A* é a seguinte: a cada iteração, o nó com a menor estimativa de custo total ($f = g + h$) é escolhido para expansão.

Onde:

- g = o custo do caminho percorrido do nó inicial ao nó atual;
- h = heurística que estima o custo do caminho mais barato do nó atual ao nó objetivo;
- f = soma de g e h, ou seja, a estimativa do custo total do caminho do nó inicial ao nó objetivo passando pelo nó atual.

O algoritmo A* mantém uma lista de nós abertos e uma lista de nós fechados. A lista de nós abertos contém os nós que ainda não foram expandidos, enquanto a lista de nós fechados contém os nós que já foram expandidos. A invariante do laço garante que, a cada iteração, o nó com a menor estimativa de custo total na lista de nós abertos será escolhido para expansão, como já mencionado.

Essa invariante garante que, quando o nó objetivo é encontrado, a solução

encontrada é ótima, ou seja, não há outro caminho mais curto. Além disso, a invariante garante que o algoritmo A* é completo, ou seja, ele encontra uma solução se uma solução existe.

Para provar a invariante do laço, utilizaremos a técnica de indução.

Primeiro elemento do conjunto

No início do algoritmo, o nó inicial é definido como nó atual e $g_{nóinicial} = 0$, pois o custo do caminho do nó inicial até ele mesmo é zero. Assim, temos que:

$f_{nó\ inicial} = g_{nó\ inicial} + h_{nó\ inicial} = 0 + h_{nó\ inicial} = h_{nó\ inicial}$

Ou seja, $f_{nó\ inicial}$ é igual à heurística do nó inicial.

Hipótese

Suponha que a invariante $f = g + h$ seja verdadeira para todos os nós visitados pelo algoritmo até o momento.

Indução

Vamos considerar a o próximo nó a ser visitado pelo algoritmo, chamado de nó atual. Seja p o nó predecessor do nó atual e $w_{p,a}$ o peso da aresta que liga p a a.

Para esse novo nó atual, o algoritmo A* calcula duas estimativas:

- g_a = custo do caminho do nó inicial até o nó atual, passando por p
- h_a = estimativa do custo do caminho do nó atual até o nó final.

Com base nisso, o algoritmo A* atualiza o valor de f_a como $f_a = g_a + h_a$ e verifica se o nó atual já foi visitado antes.

O algoritmo verifica se a já foi visitado antes. - Se a já foi visitado antes, o algoritmo verifica se o novo caminho até ele é melhor que o anterior, comparando os valores de g do caminho anterior e do novo caminho. - Se o novo caminho for melhor, o algoritmo atualiza o nó com o novo valor de g e o nó é adicionado novamente na lista de nós a serem visitados.

Analisando o passo da indução, podemos ver que a invariante $f = g + h$ continua sendo verdadeira para a. Isso ocorre porque o valor de g_a é atualizado para o novo valor, mas a soma f_a como $f_a = g_a + h_a$ continua sendo a mesma.

Assim, podemos concluir que a invariante $f = g + h$ é válida para todos os nós visitados pelo algoritmo A*, ou seja, a cada iteração do laço do algoritmo, a soma $f = g + h$ é mantida como invariante.

Resultados obtidos

Primeiramente, o backend foi capaz de carregar os dados dos arquivos *DTED* de forma eficiente, o que é fundamental para o sucesso da aplicação. O processo

de carregamento dos dados é ativado através de uma requisição *POST*, o que permite que a solução seja facilmente integrada a outras aplicações que exigem a obtenção de rotas eficientes.

Em seguida, foi feita uma requisição *GET*, passando os parâmetros sourceLat, SourceLon, targetLat e targetLon(que são respectivamente a latitude e longitude dos pontos de partida e chegada) para definir os nós de início e fim e rodá-los no algoritmo _A*_. O algoritmo _A*_ é capaz de encontrar o caminho mínimo em um grafo com custos positivos, e neste caso, levou em consideração fatores como elevação, tipo de solo e outros elementos topográficos para encontrar a melhor rota possível entre os dois pontos definidos.

Após a execução do algoritmo, o caminho gerado foi salvo no banco de dados *Neo4j* e retornado como resposta à requisição *GET*. Isso significa que a solução proposta é escalável e pode ser facilmente adaptada para trabalhar com diferentes tipos de terrenos. A requisição *GET*, com os parâmetros de entrada necessários, permite que a aplicação encontre a rota mais eficiente entre dois pontos, o que é útil em diversas aplicações que envolvem o planejamento de rotas, como entregas, transporte, entre outras. Além disso, a solução proposta pode ser facilmente integrada a outros sistemas, tornando-se uma opção viável para a otimização de rotas em diversas áreas de atuação.

Conclusão

A tecnologia apresentada nesta pesquisa possui grande relevância como solução para entidades públicas de defesa e soberania da pátria, especialmente para operações militares de alta complexidade e sigilosidade. É válido ressaltar que essa solução proposta tem impactos diretos nos problemas iniciais enfrentados nessas operações, seja para reconhecimento, salvamento ou invasão de território inimigo. Essas operações são de alto risco e dependem de tecnologias que possam torná-las possíveis com a agilidade e execução requisitadas. Além disso, as soluções propostas até então são insuficientes, como é o caso da tecnologia *Terrain-Following*, que consiste em uma tecnologia de mapeamento 3D do terreno para a escolha do percurso com a menor altitude, de forma mais manual e sem análise do terreno como um todo, mas sim do local de sobrevoo da aeronave no momento em questão. Essa insuficiência torna difícil a previsibilidade de um caminho mínimo viável, o que foi solucionado com a tecnologia apresentada nesta pesquisa.

Logo, o impacto potencial da tecnologia desenvolvida, em alternativa e complemento da tecnologia *Terrain-Following*, é significativo. Esta solução possui alta tecnologia e complexidade de resolução, o que gera grande valor e interesse. Dessa forma, a resolução apresentada através do algoritmo _A*_ com a heurística proposta se torna uma solução viável de caminho mínimo entre o ponto de partida e destino apresentado pelo usuário para uso aeronáutico. Este algoritmo traz maior previsibilidade de um caminho viável, coisa que a tecnologia

de *Terrain-Following* não traz, completando-a. Muitas das imprevisibilidades do caminho e do terreno podem ser calculadas e descobertas por essa tecnologia. Em suma, as tecnologias, quando utilizadas em conjunto, são capazes de trazer mais confiabilidade e segurança na missão.

Para solucionar este problema, foi arquitetada uma solução composta por várias tecnologias harmoniosas que, em conjunto, oferecem uma solução eficiente no processamento e armazenamento dos dados geográficos necessários. Uma estrutura foi montada com o objetivo de padronizar este ambiente através do *Docker*, armazenando rotas em grafos pelo banco de dados *NoSQL Neo4j*, extração dos dados geográficos, manipulação dos mesmos através de tipos abstratos de dados e processamento de rotas por estes tipos abstratos através de algoritmos de caminho mínimo na linguagem *Java*. Por fim, os resultados foram processados e exibidos através de tecnologias de *front-end* como *React Native*, *TypeScript* e a biblioteca *D3*.

Dentre os resultados que foram levantados nesta pesquisa, é importante ressaltar a eficiência do backend da solução na leitura e armazenamento dos dados geográficos do *DTED*. Além disso, a *API* de consumo e escrita dos dados foi pensada de forma eficiente para lidar com um grande número de dados. O processamento da rota pelo algoritmo _A*_ também trouxe excelentes resultados. Em especial, o uso desse algoritmo foi determinante para encontrar uma solução viável para o problema apresentado, levando em conta fatores como elevação, tipo de solo e outros elementos topográficos para encontrar a melhor rota possível entre os dois pontos definidos. O caminho retornado foi armazenado de maneira inteligente no banco *NoSQL Neo4j*, um banco orientado a grafos, especialmente porque o caminho é menor que todo o grafo processado e seu armazenamento se torna mais viável no banco. Por fim, a requisição *GET* trabalhou para que o *front-end* consuma os dados processados e exiba em uma tela de fácil visualização a rota gerada.

Em suma, a solução e pesquisa apresentadas neste artigo trazem uma grande contribuição para as ciências exatas. A solução agrega diversas tecnologias poderosas, tornando-a viável e vendável para qualquer instituição interessada.

Referências Bibliográficas

AEL, Portal de comunicações. **Terrain Following**. Disponível em: https://www.ael.com.br/terrain-following.html. Acesso em: 20 fev. 2023.

ASPLAN. **Sistema de Planejamento Estratégico e Defesa**. Importância do Planejamento Estratégico,cap.3. Disponível em: https://portal.tcu.gov.br/lumis/portal/file/fileDownload.jsp?fileId=8A8182A24F0A728E014F0AC5DCB653E4. Acesso em: 20 fev. 2023

DA COSTA, Rodolfo Lobo. **Causa e prevenção de acidentes resultantes da colisão controlada com o solo – CFIT**. Universidade do Sul de Santa

Catarina – UNISUL, Palhoça, SC. 2019.

GERSTING, Judith L. **Fundamentos matemáticos para Ciência da Computação**. Rio de Janeiro: LTC – Livros Técnicos e Científicos Editora, 2017.

GRAMANI, Cristina M; **Abordagem de Caminho Mínimo para problemas de otimização**, XXXVI - SBPO, São João Del-Rei, MG. 23 a 26 de novembro de 2004.

L., A. et al. **Utilização do Algoritmo de Dijkstra para Cálculo de Rotas no Trabalho Público do Município de Criciúma/SC**. Disponível em: https://periodicos.unesc.net/ojs/index.php/sulcomp/article/download/1815/1717/5477. Acesso em: 20 fev. 2023.

LINTZMAYER, CN; OLIVEIRA MOTA, G. **Análise de Algoritmos e Estruturas de Dados**. Disponível em: https://www.ime.usp.br/~mota/livros/livro_AAED.pdf. Acesso em: 20 fev. 2023.

LUIZ, O. **PETI**. Intodução,cap.2. Disponível em: https://www.gov.br/inpi/pt-br/acesso-a-informacao/tecnologia-da-informacao/arquivos/documentos/peti_20162019.pdf. Acesso em: 20 fev. 2023.

PRESTES, E. **Introdução à Teoria dos Grafos**. Fundamentação Teórica,cap.1. Disponível em: https://www.inf.ufrgs.br/~prestes/Courses/Graph%20Theory/Livro/LivroGrafos.pdf. Acesso em: 20 fev. 2023.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. **Algoritmos: Teoria e Prática**. 3. ed. Rio de Janeiro: Elsevier, 2012.

RUSSELL, Stuart Jonathan; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. 3rd ed. Upper Saddle River, NJ: Prentice Hall, 2010.