DM

Um Estudo sobre Derrotas na Automatização de um Sistema de Navegação Marítima

DISSERTAÇÃO DE MESTRADO

Inês Sousa dos Reis

MESTRADO EM MATEMÁTICA, ESTATÍSTICA E APLICAÇÕES

UNIVERSIDADE da MADEIRA

A Nossa Universidade

www.uma.pt

Setembro | 2024

Um Estudo sobre Derrotas na Automatização de um Sistema de Navegação Marítima

DISSERTAÇÃO DE MESTRADO

Inês Sousa dos Reis
MESTRADO EM MATEMÁTICA, ESTATÍSTICA E APLICAÇÕES

ORIENTAÇÃO
Paulo Sérgio Abreu Freitas

Agradecimentos

Este trabalho só teria sido possível com o apoio, suporte e orientação de algumas pessoas.

Primeiramente, gostaria de agradecer aos meus pais, aos meus avós e à minha tia por me terem dado a oportunidade de voltar a estudar e pelo apoio sem o qual não teria sido possível tirar este mestrado.

Ao meu orientador, professor Paulo Freitas, por ter visto potencial no tema, pelo tempo despendido na realização do mesmo ao longo destes dois anos e meio e pelos conhecimento técnicos que fizeram evoluir o trabalho.

Aos professores da ENIDH, Sérgio Tomé e Emídio Torrão, por todo o material fornecido e sugestões. E, em particular, ao professor Pedro Silveira, antigo professor da ENIDH pelo apoio e sugestões de como deveria resolver alguns problemas da área da navegação que foram surgindo.

Aos meus amigos, em particular à Sara Teixeira, à Marta Rodrigues e ao Pedro Duarte, por não me ter deixado desistir.

A todos vocês um sincero OBRIGADO!!

Resumo

A falta de marítimos, a necessidade de aumentar a segurança da navegação e a evolução tecnológica têm contribuído para um aumento do estudo de navios autónomos e não tripulados. O crescente número de estudos realizados, um pouco por todo o mundo, vem reforçar o desenvolvimento nesta matéria não apenas do ponto de vista da navegação, mas também na gestão portuária e na legislação. Estamos perante uma revolução no transporte marítimo.

O foco deste trabalho foi a navegação costeira numa abordagem extremamente simplificada da realidade. Selecionou-se dois algoritmos amplamente usados e estudados na bibliografia, o algoritmo Dijkstra e o do Polígono Convexo, pelas suas características e simplicidade. No entanto, e apesar de vários estudos realizados com eles separadamente, não se encontrou nenhum estudo que punha estes algoritmos a trabalhar em simultâneo complementando-se mutuamente.

Concluiu-se que os algoritmos propostos minimizam a distância e os pontos de mudança de rumo e que, pela sua versatilidade, podem ser usados em simultâneo, em função de outras variáveis que não apenas a distância.

Abstract

The shortage of seafarers, the need to enhance navigation safety and technological advancements have contributed to an increase in the study of autonomous and unmanned ships. The growing number of studies conducted worldwide has reinforced development in this area, not only from the perspective of navigation, but also in port management and legislation. We are witnessing a revolution in maritime transport.

The focus of this work was coastel navigation through an extremely simplified approach to reality. Two widely used and studied algorithms were selected from the literature: Dijkstra's algorithm and the Ruber Band algorithm, due to their characteristics and simplicity. However, despite several studies conducted on them separately, no study was found that employed those algorithms simultaneously to complement each other.

It was concluded that the proposed algorithms minimize distance and course change points and that, due to their versatility, they can be used concurrently, depending on other variables beyond just distance.

Índice

Índice de Figuras iii

Índice de Tabelas vii

Lista de Símbolos e Abreviaturas ix

1 Introdução 1

1.1 Desafios 1

1.2 Estado da arte 4

1.2.1 Navegação oceânica 4

1.2.2 Navegação costeira 6

1.2.3 Navegação mista 9

2 Navegação marítima 11

2.1 Navios autónomos 11

2.2 Projetos Mundiais 14

2.2.1 Europa 14

2.2.2 Ásia e Pacifico 19

2.2.3 Rússia 20

2.2.4 Portugal 21

2.3 Sistemas de navegação 22

2.4 Matemática da navegação 25

3 Métodos para a determinação automática de derrotas 33

3.1 Algoritmo do polígono convexo 33

3.2 Algoritmo de Dijkstra 37

3.3 Algoritmo A* 42

3.4 Métodos baseados em grelhas dinâmicas 43

3.5 Método proposto 46

3.6 Dados AIS 62

4 Experiências computacionais **69**

4.1 Esquema de separação de tráfego exterior sul 69

4.2 Esquema de separação de tráfego exterior sul, com entrada no porto 71

4.3 Esquema de separação de tráfego exterior norte 72

4.4 Esquema de separação de tráfego exterior norte, com saída do porto 74

4.5 Esquema de separação de tráfego interior 75

4.6 Resultados 78

5 Notas finais e trabalhos futuros **81**

Referências **83**

Anexos 87

Índice de Figuras

2.1 Diferentes níveis de autonomia segundo MUNIN (Fonte: [33]). 12
2.2 Definição de navio autónomo segundo MUNIN (Fonte: [33]). 12
2.3 Navio ferry *Falco*, o primeiro navio de teste não tripulado (Fonte: www.theengineer.co .uk/falco- autonomous- ferry-rolls-royce). 16
2.4 Viagem de teste do navio *Falco* (Fonte: [35]). 16
2.5 Estação de Controlo remoto - SVAN (Fonte: [35]). 17
2.6 *Zulu4* (Fonte: [22]. 18
2.7 Navio ferry *Basto Fosen VI*, o primeiro navio preparado para uma navegação autónoma (Fonte: www.bairdmaritime.com/ work-boat-world/passenger-vessel-world/ro-pax/basto-fosen-ferries-to-undergo-battery-retrofit). 18
2.8 Navio porta-contentores *Yara Birkeland*, o navio de propulsão elétrica preparado para uma navegação autónoma (Fonte:www.offshore-energy.biz/worlds-1st-zero-emission-container-vessel-yara-birkeland-delivered). 19
2.9 Navio elétrico *Jin Dou Yun O Hao* (Fonte: trends.nauticexpo.com/project-331143.html). 20
2.10 Navio cargueiro *Iris Leader* (Fonte:www.fleetmon.com/vessels/iris-leader). 20
2.11 Navio tanque *Prism Courage* (Fonte:https://newatlas.com/transport/first-autonomous-ocean-passage-prism-courage-tanker-hyundai/). 21
2.12 Drone Português *UOPV* (Fonte: https://www.jornaldenegocios.pt/negocios-em-rede/mar-sustentavel/detalhe/uopv-um-projeto–de-id-desafiante–para-navios–nao-tripulados). 22
2.13 Ecrã de um Radar (Fonte: [30]). 24
2.14 Limitação do Radar para objetos a pequena distância (Fonte: [30]). 24
2.15 Imagem com ruído (Fonte: www.nauticexpo.com/pt/prod/mi-simulators /product-26918-429207.html). 25
2.16 Derrota loxodómica (Fonte: [28]). 26
2.17 Navegação de latitude constante (Fonte: [28]). 27
2.18 Triângulo de navegação plano (Fonte: [28]). 27
2.19 Derrota Ortodrómia (Fonte: [28]). 28

2.20 Triângulo de navegação esférico (Fonte: [28]). 29

3.1 Conjunto de pontos com o seu polígono convexo (Fonte: [6], pág.1029). . . 34
3.2 Matriz de projeção de obstáculos (Fonte: [46]). 35
3.3 Rota primária (Fonte: [46]). 35
3.4 Rota final (Fonte: [46]). 36
3.5 Expansão do grafo resultante do algoritmo do polígono convexo (Fonte: [16]). 36
3.6 Grelha predefinida subjacente a uma derrota ortodrómica (Fonte: [43]. . . 39
3.7 Definição da grelha com 72 vértices (Fonte: [32]). 40
3.8 Estrutura *quadtree* (Fonte: [24]). 40
3.9 Estrutura das variáveis utilizadas para definir o custo das arestas (Fonte: [12]). 42
3.10 Cálculo do tempo mínimo (Fonte: [42]). 44
3.11 Distância que o navio consegue navegar, num dado intervalo de tempo, com uma quantidade de combustível predefinida (Fonte: [42]). 45
3.12 Determinação das isócronas usando a rota de referência (Fonte: [42]). . . . 46
3.13 Carta náutica 3427 (Fonte: Arquivo Escola Superior Náutica Infante D. Henrique). 47
3.14 Carta náutica $n°$2655 (Fonte: Arquivo Escola Superior Náutica Infante D. Henrique). 50
3.15 Simplificação da Carta náutica $n°$2655. 51
3.16 Esquema de Separação de Tráfego: Sentido Norte. 53
3.17 Esquema de Separação de Tráfego: Sentido Sul. 54
3.18 Ilustração de uma área de navegação, com obstáculos predefinidos. 55
3.19 Vértices sucessores de s, de acordo se são seus vizinhos mais próximos em direção aos pontos cardeais e colaterais. 55
3.20 Rota ótima obtida no final da primeira fase do método proposto usando ambos os pontos cardeais e os colaterais. 57
3.21 Rota ótima obtida no final da primeira fase do método proposto usando apenas os pontos cardeais. 58
3.22 Segunda fase do método proposto: A, B e C colineares. 58
3.23 Segunda fase do método proposto: A, B e C não colineares, com AC livre de obstáculos. 59
3.24 Segunda fase do método proposto: A, B e C não colineares, com o segmento entre A e o novo B a intersetar obstáculos. 59
3.25 Segunda variante da segunda fase do método proposto: quando o segmento entre A e o novo C interseta obstáculos. 61

3.26 Comparação entre os vários caminhos obtidos pelas três variantes da segunda fase do método proposto, assim como o caminho obtido na primeira fase. 63
3.27 Dados de AIS referentes ao navio *Patrick*. 67

4.1 Esquema de separação de tráfego exterior sul: Dados de AIS referentes a viagens de 5 navios diferentes. 70
4.2 Esquema de separação de tráfego sul: Derrotas obtidas pelo método proposto. 70
4.3 Esquema de separação de tráfego exterior sul, com entrada no porto: Dados de AIS referentes a viagens de 3 navios diferentes. 71
4.4 Esquema de separação de tráfego exterior sul, com entrada no porto: Derrotas obtidas pelo método proposto. 72
4.5 Esquema de separação de tráfego exterior norte: Dados de AIS relativos a 9 viagens de 8 navios diferentes. 73
4.6 Esquema de separaçao de tráfego exterior norte: Derrotas obtidas pelo método proposto. 73
4.7 Esquema de separação de tráfego exterior norte, com saída do porto: Dados de AIS relativos a viagens de 5 navios diferentes. 74
4.8 Esquema de separação de tráfego exterior norte, com saída do porto: Derrotas obtidas pelo método proposto. 75
4.9 Esquema de separação de tráfego interior: Sobreposição da dados AIS relativos a 70 viagens de 16 navios diferentes. 76
4.10 Esquema separação de tráfego interior, com entrada ou saída do porto: Sobreposição de dados AIS relativos a 15 viagens de 11 navios diferentes. . 76
4.11 Esquema de separação de tráfego interior, de norte para sul: Derrotas obtidas pelo método proposto. 77
4.12 Esquema de separação de tráfego interior, com estrada no porto: Derrotas obtidas pelo método proposto. 78

Índice de Tabelas

2.1 Níveis de autonomia de acordo com Sheridan [34]. 13

3.1 Pontos de linha de costa associados à Carta náutica $n°2655$. 52
3.2 Tipos de mensagens AIS presentes nos dados recolhidos. 64
3.3 Parâmetros selecionados (Fonte: AIS_Data). 65
3.4 Atributos dos dados (Fonte: AIS_Data) 65
3.5 Estados de navegação [18]. 66

4.1 Sumário dos resultados das experiências computacionais. 78

Lista de Símbolos e Abreviaturas

AAWA: Advanced Autonomous Waterborne Application Iniciative
ADF: Automatic Directional Finder
AEGIS: Advanced, Efficient and Green Intermodal System
AIS: Automatic Information System
ARPA: Automatic Radar Plotting Aid
ATON: Aids TO Navigation
AUTOSHIP: Autonomous Shipping Initiative for Europeans Waters
BIMCO: Baltic and Internacional Maritime Council
CAS: Collision Avoidance System
CENTEC: Centro de Engenharia e Tecnologia Naval e Oceânica
COLREG/RIEM: Internacional Regulations for Preventing Collisions at Sea
DBSCAN: Density-Based Spacial Clustering of Application with Noise
DGRM: Direção Geral dos Recursos naturais, segurança e serviços Marítimos
DSC: Digital Selective Calling
E: Este
ECDIS: Electronic Chart Display Information System
EMSA: European Maritime Safety Agency
EPIRB: Emergency Position Indicating Radio Beacon
EST: Esquema de Separação de Trafego
GFS: Global Forecast System
GMDSS: Global Maritime Distress Signal System
GNSS: Global Navigation Satellite System
GPS: Global Position System
H2H: Hull-to-Hull
IBS: Integrated Bridge System
IMO: Internacional Maritime Organization
INS: Integrated Navigation System
ITU: International Telecommunication Union
LIDAR: Laser Imaging Detection And Renging
LOA: Levels Of Autonomy
LRIT: Long Range Identification and Tracking System
MASS: Maritime Autonomous Surface Ship
ML: Machine Learning
MMSI: Maritime Mobile Service Identity
MUNIN: Maritime Unmanned Navigation through Intelligence in Networks

MoniRisk: Traffic Monitoring and Maritime Risk Assessment
N: Norte
NE: Nordeste
NO: Noroeste
NYK: Nippon Yusen Kabushiki
O: Oeste
PSO: Particle Swarm Optimization
RADAR: RAdio Detection And Ranging
RRT: Rapidly exploring Random Tree
S: Sul
SAR/SART: Search And Rescue operation / Search And Rescue Transponder
SE: Sudeste
SO: Sudoeste
SOTA: Self Organizing Tree Algorithm
SRID: Spatial Reference System
SVAN: Safer Vessel with Autonomous Navigation
UOPV: Unmanned Oceanic Patrol Vessel
UTC: Universal Time Coordinated
VHF: Very High Frequency
VTMS: Vessel Tracking Management System
VTS: Vessel Traffic Services
WGS: World Geodetic System
WW3: Wave Watch III

1 Introdução

For as long as we can remember, humans had build ships to conquer the seven seas but who have thought that ships may one day conquer those seas without us!

Vincent Van Quickenborne

O número de pessoas necessárias para operar um navio depende das suas dimensões, do tipo de carga, da natureza da operação e das tecnologias a bordo. Com o desenvolvimento tecnológico, abriu-se caminho para um futuro com navegação autónoma e não tripulada, permitindo reduzir não só o número de marítimos nas diferentes funções a bordo como também o número de acidentes marítimos [14].

A navegação divide-se em quatro processos bem definidos:

- *Appraisal*: Consiste na recolha de informação referente à viagem, tais como cartas de navegação, lista de faróis, descrição morfológica e oceanográfica da área, características do navio, informação meteorológica, etc;
- *Planeamento*: Tem por objetivo planear toda a viagem com base na informação disponível. Depois de concluída, é obtida uma lista de pontos de mudança de rumo, rumos a seguir, distâncias a navegar, no total e em cada troço, e tempo estimado de viagem;
- *Execução* e, simultaneamente, *Monitorização*: Permite uma constante observação sobre o progresso da viagem, de forma a que seja o mais fiel possível ao planeado, tendo em conta os demais imprevistos, tais como encontros com outros navios.

1.1 Desafios

O navegador é o ponto central em todas as fases do processo de navegação, e nenhum dos sistemas usados atualmente a bordo têm a capacidade de o substituir completamente.

Na fase do planeamento, o ECDIS (*Electronic Chart Display Information System*) auxilia na deteção de obstáculos e no cálculo de rumos, distâncias e tempos de viagem, mas, para uma viagem mais eficiente, cabe ao operador tomar a decisão da melhor distribuição dos pontos de mudança de rumo [17].

Neste contexto, o desafio consiste no desenvolvimento de um algoritmo que permita determinar, entre uma origem e um destino pré-definidos e respeitando as características individuais de cada navio, a melhor solução possível de derrota, isto é, calcular uma rota segura (no sentido de serem evitados quaisquer obstáculos e perigos ao longo da mesma) com o mínimo de pontos de mudança de rumo possíveis, com vista a otimizar uma função objetivo, função de custo (minimização) ou função utilidade (maximização), definida, por exemplo, em termos de tempo de viagem, distância percorrida, custos (de combustível, ou outro), emissões, etc.

Na fase de monitorização e execução, os sistemas descritos no próximo capítulo, integrados no IBS (*Integrated Bridge System*), fornecem dados em tempo real da situação da viagem da área circundante. No entanto, mesmo coordenados entre si, estes sistemas não são suficientes para substituir o navegador na tomada e execução da decisão [14]-[1].

Em seguida, são descritos diversos desafios que surgem no contexto da navegação marítima.

Recolha de dados:

Como veremos no Capítulo 2, o AIS (*Automatic Information System*) e o RADAR (*RAdio Detection And Ranging*) ajudam a sintetizar e a analisar toda a informação recolhida. Contudo, nem todas as embarcações estão equipadas com emissores de AIS, tais como navios à vela, embarcações de pesca rudimentares, entre outros, dificultando o seu reconhecimento por parte dos navios equipados com recetores de AIS. O RADAR consegue detetar o eco desses objectos a determinadas distâncias, mas os ruídos devido à neve, chuva ou outras condições de mar adversas criam mais ecos do que aqueles que realmente existem como obstáculos à navegação. O papel do navegador na recolha de informação é analisar, caso a caso e sob uma constante observação visual e auditiva, os dados fornecidos pelos equipamentos, comparando-os sempre com a sua perceção da realidade circundante, procedendo, então, à tomada de decisão, gestão e filtração dos dados disponíveis.

Neste processo, o desafio consiste em equipar o navio com diferentes tipos de câmaras e microfones e explorar outras bandas de RADAR e LIDAR (*Laser Imaging Detection And Ranging*), com vista a aumentar a qualidade da informação recolhida, com sobreposição da mesma, e criar módulos que realizem a filtração destes dados.

Manobras de anti-colisão:

Perante a necessidade de executar manobras para evitar a colisão, os equipamentos disponíveis não sugerem, infelizmente, rotas possíveis à manobra. O ARPA (*Automatic Radar Plotting Aid*) e o ECDIS ajudam o navegador a estudar possíveis alterações ao rumo e à velocidade, fazendo os cálculos e fornecendo informações sobre o impacto de uma determinada manobra sugerida pelo próprio navegador. Portanto, cabe ao navegador fazer o estudo das manobras possíveis, seguindo o COLREG (*Internacional Regulations for Preventing Collisions at Sea*), e tomar a decisão de qual delas realizar.

Neste processo, o desafio passa pela criação de algoritmos capazes de estudar e decidir qual a melhor manobra a executar em cada ocasião, respeitando o regulamento.

Alterações de rumo e velocidade:

Uma vez decidida a manobra a ser executada cabe ao navegador realizá-la, de forma manual utilizando o leme, ou de forma "automática", com recurso ao piloto automático. Neste caso, após o navegador fornecer o rumo sob o qual vai navegar, o piloto automático ajusta o leme para que a mudança de rumo seja realizada.

O desafio nesta fase consiste na criação de algoritmos e procedimentos que permitam ao navio executar dinamicamente mudanças de rumo sem a intervenção do navegador. Esta função seria também importante nos pontos intermédios da viagem para que a rota definida no planeamento fosse cumprida o mais fielmente possível. Estes procedimentos operariam não apenas em situações de evitar abalroamentos, mas também sempre que as mudanças de rumo inerentes à viagem assim o justificassem. Na navegação em águas restritas, as características especiais de cada navio também teriam de ser consideradas nesses procedimentos.

O presente trabalho foca-se, principalmente, na fase do planeamento, onde é proposto um novo algoritmo que resulta da conjugação de dois diferentes algoritmos descritos na literatura, com o objetivo de melhorar o planeamento de uma derrota ótima. Na próxima secção, com o intuito de dar a conhecer um pouco do panorama geral acerca do tema, apresenta-se uma breve descrição da literatura sobre o planeamento da viagem em diferentes contextos de navegação, considerando obstáculos estáticos.

No Capítulo 2, é realizada uma breve contextualização dos diferentes projetos desenvolvidos em diferentes partes do mundo, com vista à automação de navios, de forma a termos uma ideia geral de como a área está a evoluir. Seguidamente, é dada uma breve descrição sobre o funcionamento dos aparelhos usados atualmente a bordo, com as suas

vantagens e limitações. Por fim, são descritos diferentes tipos de navegação e respetivas determinações matemáticas de derrotas (rotas).

No Capítulo 3, é apresentado um estudo mais aprofundado dos algoritmos e das abordagens mais promissoras, seguido de uma descrição detalhada da abordagem proposta neste trabalho.

No Capítulo 4, são descritas as experiências computacionais realizadas, o tratamento efetuado a um conjunto de dados de AIS e uma comparação, tanto quanto possível, entre este e os resultados obtidos.

No Capítulo 5, são apresentadas as notas finais deste estudo e possíveis trabalhos futuros.

1.2 Estado da arte

Nesta secção, é feito uma apanhado geral sobre diferentes abordagens, organizada por tipo de navegação estudada (oceânica, costeira e mista) e por abordagens de métodos de otimização matemática.

1.2.1 Navegação oceânica

O principal objetivo da navegação oceânica consiste na minimização do consumo de combustível. Este consumo depende essencialmente das condições meteorológicas e das características do navio.

A abordagem proposta por Wang [43] teve como objetivo adicional a minimização da fadiga do material em que foram consideradas como variáveis a resistência ao movimento do navio, provocada pela variação da ondulação e do vento, a propulsão da máquina, a resistência física do material e as dimensões do navio. O autor sugeriu um algoritmo híbrido, onde é gerado um grafo através da implementação de uma grelha, gerada a partir da ortodrómia, decompondo-a em perpendiculares que contêm os pontos, seguida da aplicação do algoritmo de Dijkstra para encontrar o candidato ótimo, usado como população inicial do algoritmo genético que suaviza e aprimora a solução ótima. Foi verificado que a diminuição da velocidade no início da viagem contribuía para que o navio evitasse o mau tempo a meio da viagem, minimizando o consumo do combustível e os esforços do navio.

O trabalho realizado por Freitas [12] pretendeu otimizar os recursos energéticos da marinha permitindo um aumento das eficiências das derrotas planeadas. Foi sugerido um

estudo sobre as perdas de velocidade, analisando os efeitos das condições meteorológicas num modelo matemático para navios do tipo corvetas. Estas perdas de velocidade foram a base para a atribuição dos pesos das arestas como o consumo da máquina. No final, o algoritmo de Dijkstra foi usado para minimizar o consumo de combustível.

No estudo conduzido por Avgouleas e Sclavounos [3], foram consideradas como variáveis a resistência ao movimento do navio, provocada pelas condições meteorológicas, e os coeficientes de propulsão[1], com o objetivo único de minimizar o consumo de combustível.

No trabalho de Kuhlemann e Tierney [23], foram consideradas como variáveis a perda de velocidade, resultante das condições climatológicas, e as áreas de pirataria, com o objetivo adicional de evitar navegar em áreas reconhecidas pela pirataria. Na impossibilidade de o fazerem, levarem em conta as particularidades de navegação nessas áreas. Por exemplo a IMO (*Internacional Maritime Organization*) define uma velocidade mínima de navegação como forma de evitar a abordagem pirata. Nesse trabalho, foram utilizados algoritmos genéticos no processo de otimização, onde as operações de mutação e cruzamento foram baseadas nos dados climatológicos.

No caso particular na navegação no Ártico, Li [26] salientou a importância do rigor das previsões meteorológicas e da definição de áreas seguras à navegação, sendo que, após o processo de otimização, seria necessário avaliar os resultados obtidos em termos de segurança, adaptabilidade e economia.

Em [10] Fan e Nie, o problema da otimização de uma viagem oceânica foi abordado como uma generalização do processo de decisão de Markov. Esta linha de pensamento tem como objetivo a maximização da probabilidade de chegar ao destino em um determinado tempo pré-estabelecido, sem penalizações por chegadas antecipadas. O algoritmo utilizado foi o **SOTA** (*Self Organizing Tree Algorithm*), no qual é considerado um grafo de n-vértices com uma função densidade de probabilidade, onde as arestas são calculadas usando a função Gamma.

Ainda no contexto da navegação oceânica, podemos encontrar na literatura alguns trabalhos de otimização meta-heurística baseada no comportamento social, tais como Cheng e Tsou [40], que usaram o algoritmo por colónia de formigas, e Zhao et al [47], que propõem uma combinação do algoritmo PSO (*Particle Swarm Optimization*) e do algoritmo genético. A abordagem consiste numa avaliação multicritério em que, em cada interação,

[1] Como o torque e o impulso da hélice.

a solução é avaliada de forma a minimizar o tempo de viagem, o consumo de combustível e o risco devido às condições meteorológicas. Ambos os trabalhos minimizaram o consumo de combustível através da minimização do tempo de navegação, e tendo como base para a área de procura a derrota ortodrómia.

1.2.2 Navegação costeira

Devido às características próprias deste tipo de navegação e ao rigor exigido pela aproximação de costa e pelo aumento da quantidade de tráfego, a navegação costeira tem sido estudada sob duas abordagens distintas: uma abordagem em que os dados AIS são o ponto de partida e uma abordagem de modelação matemática.

AIS

Nesta abordagem, os dados obtidos por AIS são processados de modo a fornecerem informações valiosas sobre o tráfego marítimo, podendo depois serem usados para extrair padrões de trajetórias.

No trabalho de Wen et al [45], foi proposto a construção de uma rede neuronal artificial com o objetivo de prever pontos de mudança de rumo para navios de diferentes características e dimensões. O ruído contido nos dados foram suavizados utilizando o algoritmo DBSCAN (*Density-Based Spacial Clustering of Application with Noise*), sendo estes depois reduzidos a uma dimensão usando os valores próprios Laplacianos. Cada trajetória daí extraída foi usada para representar um vértice do grafo. A matriz do grafo resultante foi depois processada usando a função gaussiana e a função de Kernel. A rede neuronal construída foi treinada com os dados obtidos para então fornecer previsões e recomendações sobre rotas para diferentes navios.

Usando dados de AIS cartesiados(mapeados), Rong et al [36] desenvolveram um método probabilístico para a previsão de trajetória. A linha central do movimento foi calculada usando o algoritmo *Dynamic Time Warping*, sendo aquele decomposto em movimentos longitudinal e lateral, que depois foram combinados numa única função densidade de probabilidade, resultante da multiplicação das duas funções densidades de probabilidade individuais. Em continuação ao seu estudo, os mesmos autores propõem, em [37], o uso da mineração de dados AIS para uma caracterização probabilística do tráfego marítimo e deteção de anomalias. Os dados foram agrupados por tipo de navio, ponto de chegada e outras características que influenciavam a rota escolhida. As mudanças de rumo eram depois detetadas usando o algoritmo *Douglas and Peucker*.

No estudo realizado por Silveira et al [39], os dados de AIS foram usados na construção de dois grafos. No primeiro, os vértices eram células de grelha, tendo como base as posições, e o peso de cada aresta era definido pelo número de vezes que cada uma era utilizada. No segundo, os vértices eram criados da mesma forma, mas os pesos eram dados pela velocidade de transição entre aqueles. Finalmente, o algoritmo de Dijkstra era usado não só para identificar a rota mais curta entre dois pontos como também a velocidade adotada em cada ponto do percurso.

Com o objetivo de extrair trajetórias usando os dados de AIS e planear trajetórias que minimizem a distância, Peng Han [15] construíram uma rede de rotas possíveis usando o algoritmo DBSCAN. Os dados AIS foram agrupados de acordo com a localização e as características geométricas, nomeadamente as trajetórias. Baseando-se nestas, os autores criaram uma rede de rotas possíveis de modo a calcularem a distância mais curta entre dois vértices, usando o algoritmo de Dijkstra. O algoritmo de Theta* foi depois usado de forma a suavizar os ângulos de mudança de rumo.

No estudo realizado por Jorgensen et al [21], foi proposto um modelo híbrido de machine learning, que combina modelos físicos do navio, tais como as relações físicas entre as variáveis posição, velocidade, rumo e condições meteorológicas, e o consumo de energia, com os dados obtidos pela técnica de machine learning. Com esta abordagem, foi possível obter uma redução de $3,7$ % de energia, quando comparada com modelos atuais resultantes de pequenas variações de rumo e distância.

Modelação matemática

No contexto da navegação, a modelação matemática procura criar modelos e utilizar algoritmos que minimizem o tempo de viagem, a distância percorrida e o número de pontos de mudança de rumo.

Este problema foi abordado de forma semelhante em ambos os trabalhos de Jian Bo Xu et al [46] e Hong [16] na determinação do caminho mais curto entre dois vértices, diminuindo o número de pontos usados para mudanças de rumo. A ideia por trás do algoritmo usado na resolução do problema consistiu em usar uma espécie de banda elástica (*rubber band*) que "estica" no contorno dos obstáculos, de modo que seja construído um polígono regular com o menor número de vértices possível - algoritmo do polígono convexo. Esta ideia é descrita com maior detalhe no Capítulo 3.

Em outros trabalhos como Jia et al [19] e Li Xie [25], foi usado o algoritmo RRT (*Rapidly exploring Random Tree*) como meio para a obtenção de uma rota ótima. Enquanto que, por um lado, Jia et al [19] focaram-se no processo de fusão de dados referentes a obstáculos presentes em diversas cartas náuticas de diferentes escalas, onde, por meio do algoritmo RRT, era decidido entre agrupar vários obstáculos num único ou pegar num deles e dividi-lo em múltiplos obstáculos, permitindo aumentar a área de navegação às necessidades e assim diminuir o tempo de viagem, por outro, Ping Lie et al [25] usaram o algoritmo genético na procura da rota ótima que minimizasse a distância percorrida, usando como solução primária a obtida pelo algoritmo RRT, sendo esta considerada como uma delimitação da área navegável, ou seja, a área que o algoritmo genético atua.

No trabalho de Lee et al [24], o objetivo principal consistiu na minimização do custo computacional para a construção de um grafo. Esta minimização foi realizada com recurso à representação *quadtree*, isto é, dividindo repetidamente um quadrado de grelha em quatro quadrados até que nenhum objeto se encontre na grelha. Usando linhas que não intersetam obstáculos entre pontos de mudança de rumo, foi construido um grafo de visibilidade, cujos os pesos das arestas eram definidos como os tempos de viagem, tendo em conta as perdas de velocidade devido às condições meteorológicas. O algoritmo de Dijkstra foi depois utilizado para encontrar o caminho que minimiza o tempo de viagem.

O algoritmo de Dijkstra foi também usado por Novac et al [32] na determinação da rota mais curta entre dois pontos no Mar Negro, considerando as alterações de velocidade provocadas pela ondulação e pelo vento. Os pesos das arestas nas áreas navegáveis foram definidos como 20 milhas náuticas, segundo a direção dos pontos cardeais, e 28 milhas náuticas, segundo as diagonais. Nas áreas não navegáveis, os pesos foram de 99 milhas, garantindo que estes não fossem escolhidos durante a aplicação do algoritmo referido.

No trabalho de Novac e Rusu [31], o algoritmo de Dijkstra foi aperfeiçoado, designando-se de algoritmo A*. Este algoritmo conjuga numa única função a soma da função custo entre o ponto inicial e o ponto atual e da função heurística entre o ponto atual e o ponto final, minimizando o tempo de viagem. A grelha foi construída por quadrados de lado 2 milhas náuticas, sendo os pesos das arestas dados pelos tempos de viagem, tendo em conta as reduções de velocidade resultantes da altura e direção da ondulação.

O algoritmo A* foi também abordado por Grifoll et al [13], com a única diferença dos quadrados da grelha terem de lado 1,5 milhas.

No trabalho desenvolvido por Wang et al [44], é tido em conta os movimentos do pró-

prio navio e de que forma estes influenciam o rigor na obtenção da rota ótima.

1.2.3 Navegação mista

Mais recentemente, a problemática da navegação mista tem incentivado o aparecimento de abordagens mais criativas, tentando procurar por soluções mais generalizadas, tanto no tipo de navegação como na não diferenciação entre obstáculos cinéticos e estáticos.

Neste contexto, surgiram trabalhos com vista a minimizar o número de pontos de mudança de rumo e da distância navegada. É o caso, por exemplo, do trabalho de Zhenping et al [48], onde a programação dinâmica foi usada para as decisões serem tomadas passo-a-passo, criando-se uma sequência de decisões otimizadas.

Esta mesma questão é abordada por Emiris [9], onde foram usados diagramas de Voronoi. Estes diagramas resultam numa decomposição do espaço em segmentos equidistantes a dois obstáculos mais próximos. Os vértices resultam da intersecção de dois segmentos representado um ponto equidistante a três ou mais obstáculos. Assim, cada vez que o navio entra em uma dessas áreas saberemos automaticamente qual o obstáculo mais próximo.

Para obstáculos dinâmicos pode-se construir diagramas de Voronoi cinéticos, em que os obstáculos são pontos de movimento do espaço.

No trabalho de Liu et al [27], foi utilizada a noção de campo de Energia Potencial para minimizar a distância navegável e maximizar a segurança de navegação. Isto permitiu descrever o espaço de navegação com uma maior precisão. O uso desta noção baseia-se em duas premissas: a de que o navio navega sempre em direção ao ponto de destino, existindo, portanto, um campo de atração entre o navio e aquele, e de que os obstáculos devem ser evitados, assumindo a existência de um campo de repulsão entre o navio e os obstáculos. O algoritmo A* foi depois usado como um algoritmo de procura heurística.

2 Navegação marítima

Como pudemos constatar no capítulo anterior, este tema tem sido amplamente estudado e analisado sob diferentes abordagens. Com o intuito de serem discutidas com maior rigor no capítulo 3, introduzimos, neste capítulo, alguns conceitos no contexto da navegação marítima, em particular, o de navios autónomos. São descritos não só alguns dos projetos desenvolvidos um pouco por todo o mundo como também algumas particularidades dos diferentes tipos de navegação, incluindo os equipamentos usados a bordo para navegação.

2.1 Navios autónomos

O projeto MUNIN (*Maritime Unmanned Navigation through Intelligence in Networks*) introduz a realidade de automação no transporte marítimo de forma a aumentar a competitividade, segurança e sustentabilidade no meio marítimo. São vários os sistemas terrestres (como veículos de condução remota em terminais de contentores) e aéreos onde já estão presentes diferentes níveis de automação [33].

A Waterborne TP - uma plataforma de inovação e pesquisa Europeia para industrias marítimas - descreve um navio automático da seguinte forma:

> *"A próxima geração modular de controlo de sistemas e comunicação será monitorizada e operada a bordo e em terra. Isso incluirá sistemas avançados de suporte à decisão que fornecerá a capacidade de operar um navio remotamente sobre um semi ou completamente controlo automático."*

O MUNIN interpretou esta definição como uma implicação de duas alternativas genéricas que combinariam um navio autónomo (ver Figura 2.1):

- Um navio que operaria em controlo remoto e;
- Um navio que funcionaria completamente de forma autónoma.

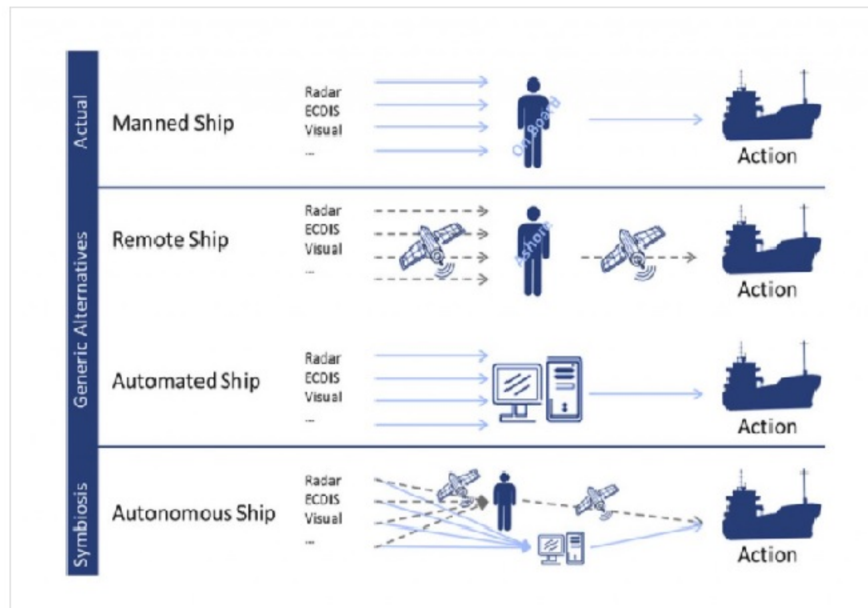

Figura 2.1: Diferentes níveis de autonomia segundo MUNIN (Fonte: [33]).

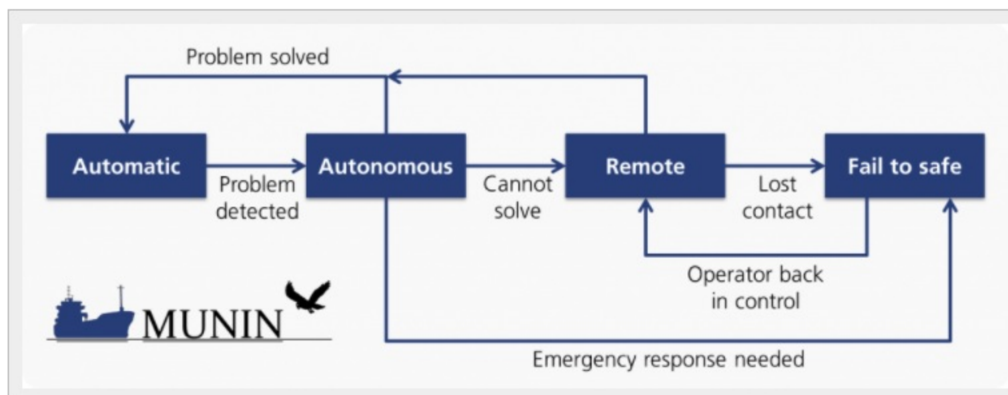

Figura 2.2: Definição de navio autónomo segundo MUNIN (Fonte: [33]).

Esta ideologia culminou num conceito holístico combinado, desenvolvendo e validando uma mistura de tecnologia automática e remota (ver Figura 2.2).

Numa situação normal, o navio navegaria de forma completamente autónoma, equipado com sistemas de sensores que detetassem situações problemáticas, tais como, objetos inesperados, condições meteorológicas adversas ou risco de colisão. Se uma situação inesperada ocorrer, o controlo autónomo tentaria resolver a situação. Se o sistema não conseguisse chegar a uma solução, pediria suporte a um operador remoto ou começaria um procedimento de "fail to safe", se o operador não estivesse disponível.

O projeto AAWA (*Advanced Autonomous Waterborne Application Iniciative*), que deu continuidade ao projeto de MUNIN [34], define um navio autónomo e os seus níveis de

autonomia partindo dos níveis de autonomia criados por Thomas Sheridan[1] para máquinas autónomas. Estes níveis são usualmente usados para descrever até que ponto um sistema consegue operar de forma independente.

A escala de Sheridan contém uma amplitude de definições desde um controlo completamente humano até uma máquina ser completamente autónoma, não necessitando de nenhuma ação humana (Tabela 2.1).

Tabela 2.1: Níveis de autonomia de acordo com Sheridan [34].

Nível	Descrição
10	O computador decide se atua e se informa ou não o operador.
9	O computador atua e decide se informa ou não o operador.
8	O computador atua e informa o operador apenas se este pedir.
7	O computador atua e informa o operador do que foi feito.
6	O computador seleciona a ação e informa o operador, no caso deste querer cancelar a ação.
5	O computador seleciona a ação mas, para a implementar, necessita da aprovação do operador.
4	O computador seleciona a ação e o operador decide se deve ser feita ou não.
3	O computador determina e sugere opções e o operador pode escolher seguir as recomendações.
2	O computador determina opções.
1	O operador faz a tarefa e o computador apenas a implementa.

Os intervenientes deste projeto concluíram que, no caso dos navios, a necessidade de interação humana dependerá do estado do navio e da tarefa a ser executada, isto é, os navios teriam uma autonomia dinâmica: em casos de navegação em mar aberto, o navio poderá ser completamente autónomo, ao passo que, noutras partes mais restritas da viagem, o navio navegaria sob controlo remoto. Esta linha de pensamento vem dar continuidade às conclusões retiradas do projeto de MUNIN.

A IMO define navios autónomos como navios que, sob variados graus de autonomia, podem operar independentemente da interação humana (MASS: *Maritime Autonomous Surface Ship*).

- **Grau 1:** Um navio com processos autónomos e de suporte à decisão, onde a tripulação a bordo do navio opera e controla os diferentes sistemas e funções. Algumas operações poderão ser automatizadas.

[1]Professor americano de engenharia mecânica do Instituto de Tecnologia de Massachusetts. Foi pioneiro em tecnologia robótica e controlo remoto na década de 50.

- **Grau 2:** O navio opera em controlo remoto com alguma tripulação a bordo, que estará disponível para tomar o controlo, caso seja necessário.

- **Grau 3:** O navio opera em controlo remoto sem tripulação a bordo.

- **Grau 4:** O navio é completamente autónomo.

2.2 Projetos Mundiais

Nos anos 2000, os navios começaram a ser equipados com sistemas automáticos de ajuda à tomada de decisão, tais como: AIS, GNSS receivers (*Global Navigation Satellite System*), ECDIS, ARPA, IBS, VTMS (*Vessel Tracking Management System*) e GMDSS (*Global Maritime Distress Signal System*). No entanto, apesar destes novos sistemas de equipamento, a percentagem de acidentes marítimos não diminuiu. Por essa razão, Rivkin [1] defende que os navios não tripulados tornar-se-ão "a solução perfeita" para a segurança à navegação.

De acordo com Allianz Global Corporate & Speciality (seguradora de risco mundial), cerca de 75% a 96% doa acidentes registados deve-se ao fator humano.

Por sua vez, a BIMCO (*Baltic and Internacional Maritime COuncil*) - organização privada formada por armadores e operadores que atuam no transporte marítimo internacional - prevê uma diminuição de marítimos causada pelo envelhecimento das tripulações e da relutância dos jovens irem para o mar.

Assim, com vista a colmatar a falta de tripulações e a diminuir os acidentes marítimos, têm sido desenvolvidos vários projetos focados na automação do comércio marítimo.

2.2.1 Europa

O MUNIN (projeto em vigor de 2012 a 2015) [33] - foi o primeiro projeto, suportado pela Comissão Europeia, com resultados notáveis em desenvolvimento tecnológico para a automatização de navios.

Neste projeto, foi desenvolvido o conceito para a operação de navios não tripulados, abordado anteriormente, e avaliaram a fiabilidade técnica, económica e legal deste conceito. Assim, foi proposto o desenvolvimento dos seguintes sistemas:

- Um módulo de sensores de navegação combinado com sensores de dados (recolha de informação);

- Um sistema de navegação para navegação oceânica (em mar aberto), que segue um planeamento de viagem e ajusta a rota para esse efeito (navegação autónoma);

- Um sistema de controlo e manutenção de propulsão autónoma, capaz de detetar e prevenir possíveis falhas no sistema de propulsão;
- Centros de controlo em terra com uma equipa de engenheiros e oficiais náuticos, que poderão controlar o navio remotamente sempre que necessário.

Concluíram que, por exemplo, um navio de carga geral, nestas condições, em 25 anos, pouparia cerca de 7 milhões de dólares. A probabilidade de colisão reduziria em 10 vezes (o fator fadiga da tripulação seria eliminado) e ajudaria a resolver o problema da falta de pessoal especializado na área, porque, provavelmente, atrairia uma população mais jovem e uma mesma equipa poderia operar mais que um navio em simultâneo.

O projeto AAWA (em vigor de 2015 a 2017) [34] deu continuidade ao projeto MUNIN. Liderado pela Rolls-Royce, reuniu quatro empresas europeias de tecnologia marítima, quatro universidades, o centro de pesquisa tecnológica Finlandês e o operador de Ferries Finferries. Foi proposto que um navio não tripulado deveria conter os seguintes sistemas:

- Um sistema de controlo de propulsão;
- Um sistema de posição dinâmica;
- Um sistema de navegação autónoma;
- Um sistema de alertas;
- Um sistema de controlo e seleção de dados;
- Um sistema de controlo remoto.

Cada um destes sistemas controlariam vários subsistemas que, combinados entre si, forneceriam informação ao Comandante Virtual, ou seja, um módulo que assumiria as ações de um navio de forma autónoma e enviaria as informações, de dados e das ações, ao operador remoto.

A Rolls-Royce trabalhou também no design de navios não tripulados de carga geral. Sem tripulação, todo o espaço associado à mesma pode ser aproveitado, retirando-se a ponte, as cabinas de tripulação, os reservatórios de água potável e os reservatórios de esgotos, o que contribui para uma redução de 5% do peso e, consequentemente, uma redução de gasto de combustível de 15%. Vale a pena mencionar ainda que cerca de 44% dos custos operacionais de uma navio de carga mercante está associada à tripulação. Este espaço “novo” pode ser reutilizado para aumento do transporte de carga, resultando também num aumento da margem de lucro.

Estas ideias foram praticamente confirmadas pela Rolls-Royce e Finferries no projeto SVAN (*Safer Vessel with Autonomous Navigation* (2018) [35]) com o teste do primeiro navio ferry *Falco* (ver Figura 2.3), que navegou entre Parainen e Nauvo de forma completamente autónoma.

Figura 2.3: Navio ferry *Falco*, o primeiro navio de teste não tripulado (Fonte: www. theengineer.co .uk/falco- autonomous- ferry-rolls-royce).

Na Figura 2.4 está descrito a viagem de teste feita pelo navio *Falco*. Foram realizadas duas viagens. A primeira com o navio completamente autónomo: a docagem em Parianen e em Nauvo foi sem intervenção humana assim como as manobras para evitar a colisão de três navios (obstáculos) encontrados no caminho. A segunda em controlo remoto controlado a partir da Estação de Controlo Remoto (ver Figura 2.5) nos escritórios de Finferries.

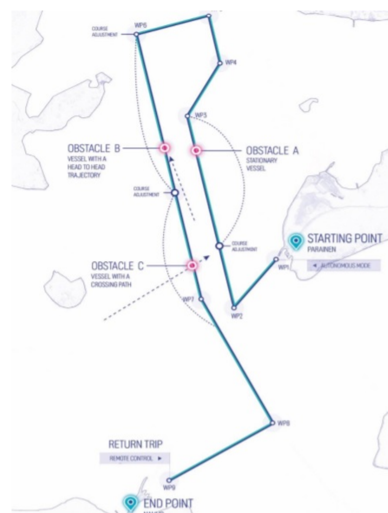

Figura 2.4: Viagem de teste do navio *Falco* (Fonte: [35]).

Em simultâneo com o AAWA, o projeto Autosea [1] focou-se no desenvolvimento de sistemas de evitar colisão para navios não tripulados. O CAS (*Collision Avoidance*

Figura 2.5: Estação de Controlo remoto - SVAN (Fonte: [35]).

System) recolhia os dados de navegação e aplicaria o COLREG baseando-se num modelo preditivo de controlo.

O H2H (*Hull-to-Hull*) deu continuidade a este projeto. O principal objetivo foi a necessidade de uma navegação segura nas proximidades de objetos estacionários e em movimento. Para isso, a recolha de dados da área circundante ao navio deve ser redundante e proveniente de vários sistemas (de posição, visuais (câmeras) e auditivos (microfone)) e com uma contínua troca de informações entre o navio, outros navios e terra.

Em 2019, a União Europeia deu inicio a mais dois projetos: o AUTOSHIP e AEGIS. O projeto AUTOSHIP (*Autonomous Shipping Initiative for Europeans Waters* [4] e [22]) reuniu cinco empresas de cinco países (Itália, Bélgica, Noruega, Finlândia e Reino Unido), com o objetivo de construir dois navios e realizar testes que comprovam as capacidades operacionais pela:

- redução de custos operacionais e de termos de entrega, fazendo-os mais competitivos com o transporte rodoviário;
- redução do consumo de combustível;
- simplificação dos procedimentos de logística; e
- providenciar a cibersegurança à navegação.

O navio *Zulu4* (ver Figura 2.6) navegou, manobrou e atracou em navegação autónoma evitando obstáculos cinéticos e estáticos ao longo de um circuito de 16.5 km, no rio Rupel na Bélgica.

O projeto AEGIS (*Advanced, Efficient and Green Intermodal Systems*), que reúne 12 empresas da Noruega, Dinamarca, Finlândia e Alemanha, tem como objetivo, em paralelo

Figura 2.6: *Zulu4* (Fonte: [22].

com o AUTOSHIP, o desenvolvimento de tecnologias para navios autónomos em navegação costeira com a redução de poluição sonora e aérea nestas áreas e na automação de operações portuárias.

Em Fevereiro de 2020, a Kongsberg Maritime -a empresa Marítima Norueguesa- anunciou que o primeiro navio completamente automatizado *Basto Fosen VI* (ver Figura 2.7) da companhia marítima *Basto Fosen*, já está pronto para navegar na linha Horten-Moss. Contudo, ainda leva tripulação a bordo e a única função ainda não automatizada é a navegação. Se o navio entra em rumo de colisão, o comandante toma o controlo do navio e realiza a manobra [1].

Figura 2.7: Navio ferry *Basto Fosen VI*, o primeiro navio preparado para uma navegação autónoma (Fonte: www.bairdmaritime.com/ work-boat-world/pas senger-vessel-world/ro-pax/basto-fosen-ferries-to-undergo-battery-retrofit).

Kongsberg contruíu, num estaleiro na Roménia, um navio autónomo de propulsão

elétrica e de carga contentorizada: o *Yara Birkeland*(ver Figura 2.8). Começou a operar na primavera de 2022 entre Porsgrunn, Breivik e Larvik. Atualmente, na primeira fase do projeto, navega com uma ponte removível para a tripulação treinar o algoritmo. Depois, operará em controlo remoto e, numa terceira fase, de forma completamente autónoma.

Figura 2.8: Navio porta-contentores *Yara Birkeland*, o navio de propulsão elétrica preparado para uma navegação autónoma (Fonte:www.offshore-energy.biz/worlds-1st-zero-emission-container-vessel-yara-birkeland-delivered).

Em todas estas iniciativas, o software para a navegação autónoma foi desenvolvido para um determinado tipo de navio (o navio de teste). O projeto Autoplan - fundado pela Alemanha e Turquia - tem como objetivo a uniformização de um sistema de assistência à navegação que poderá ser usado nos navios não tripulados [1].

2.2.2 Ásia e Pacifico

A China é um dos países líderes no desenvolvimento de tecnologia de navegação autónoma. Em 2019, concluíram os testes do navio cargueiro *Jin Dou Yun O Hao* (ver Figura 2.9), desenvolvido pela empresa tecnológica Yunzhou Tech, a Universidade de Wuhan e a Sociedade Classificadora da China. É um navio pequeno elétrico e foi acerca de 20% mais barato de construir que um navio tripulado com as mesmas dimensões. A viagem de teste foi entre Hong Kong, Zhuhai e Macao.

No Japão, a NYK line (*Nippon Yusen Kabushiki*) tem desenvolvido inteligência artificial assistida para navios autónomos. A Mitsui Engineering & Shipbuilding Co, Ltd, o ministério do Transporte, Infraestruturas e Turismo do Japão e a Universidade de Tokyo têm desenvolvido tecnologia para a construção de navios não tripulados. O *Iris Leader* (ver figura 2.10) de grande porte, é um exemplo dessa tecnologia, tendo completado uma viagem de teste de dois dias de forma autónoma.

Figura 2.9: Navio elétrico *Jin Dou Yun O Hao* (Fonte: trends.nauticexpo.com/project-331143.html).

Figura 2.10: Navio cargueiro *Iris Leader* (Fonte:www.fleetmon.com/vessels/iris-leader).

Por fim, na Coreia do Sul (líder mundial na construção de navios) a Samsung Heavy Industries começou em 2020, uma série de testes para o desenvolvimento de tecnologia para navios autónomos com rebocadores. Os testes realizados com tecnologia Samsung confirmaram as capacidades tecnológicas para evitar abalroamentos, cumprindo o COLREG. E a Hyundai anunciou o sucesso da primeira viagem oceânica, entre o golfo do México e Chungcheong, pelo navio tanque *Prism Courage* (ver Figura 2.11) em que metade da viagem foi realizada em autonomia total.

2.2.3 Rússia

A Rússia tem tido uma abordagem diferente. Os testes começaram em 2016 e, em vez de projetarem um navio com os sistemas necessários à automação e desenvolver os respetivos sistemas, o foco passa pelo desenvolvimento dos sistemas nos navios existentes de forma a que eles possam começar a operar de forma remota. O desenvolvimento tecnológico está então na uniformização dessas tecnologias.

Figura 2.11: Navio tanque *Prism Courage* (Fonte:https://newatlas.com/transport/first-autonomous-ocean-passage-prism-courage-tanker-hyundai/).

2.2.4 Portugal

O Jornal de Negócios de 02 de Fevereiro de 2022 publicou uma notícia sobre um projeto português: o projeto UOPV (*Unmanned Oceanic Patrol Vessel*) (ver Figura 2.12), a TecnoVeritas associada à NAUTIBER Shipyards (estaleiro algarvio) construíram um drone marítimo, em que o casco é de fibras resistentes às condições severas da operação. O objetivo do drone consiste em missões de patrulha nas áreas de soberania portuguesa, onde se encontram as rotas comerciais mais importantes do mundo. O drone foi desenhado para operar por longos períodos de tempo (mais de 6 meses) sem custos e restrições de combustível, a sua propulsão é por ondas.

No campo civil, poderá ser utilizado para pesquisas de biologia marinha e oceanografia.

O MoniRisk (*Traffic Monitoring and Maritime Risk Assessment*) [29] é um projeto em desenvolvimento pelo Centro de Engenharia e Tecnologia Naval e Oceânica (CENTEC) com o principal objetivo de desenvolver um sistema integrado para a monitorização do tráfego e avaliação do risco marítimo e contribuir para o aumento da segurança e da eficiência marítima.

Atualmente, debruça-se sobre os seguintes tópicos:

- Caracterização do tráfego marítimo e de rotas seguras para navios;
- Métodos para deteção de anomalias e de colisão;
- Desenvolvimento para um índice individual dinâmico de risco para navios e o con-

Figura 2.12: Drone Português *UOPV* (Fonte: https://www.jornaldenegocios.pt/negocios-em-rede/mar-sustentavel/detalhe/uopv-um-projeto–de-id-desafiante–para-navios–nao-tripulados).

trolo de tráfego marítimo;

- Análise detalhada de acidentes de colisão.

2.3 Sistemas de navegação

A *Marine Insight* - líder em notícias e informações marítimas - descreve a ponte de navegação como "o centro de operações de um navio". É equipada com sistemas e subsistemas que fornecem informações relativas à navegabilidade e ao estado do navio, auxiliando o navegador na tomada de decisão para uma navegação segura.

O IBS é definido pela Wartsila Encyclopaedia como um sistema de gestão à navegação que, conectado com outros sistemas, providencia, num único sítio centralizado, detalhes da navegação e de outros sistemas da ponte.

É composto por elementos do INS (*Integrated Navigation System* [2]) ou repetidores do mesmo, sistema de alarmes (gerais e específicos), sistemas de comunicação (de curta distância - DSC (*Digital Selective Calling*), VHF (*Very High Frequency*)), sistema de luzes de navegação e buzina, sistema de propulsão (estado da máquina), de operações de carga (controlo de água do lastro), RADAR e ARPA e AIS.

Para efeitos deste trabalho, focaremos-nos apenas no INS, no ECDIS, RADAR e ARPA, AIS e LRIT (*Long Range Identification and Tracking System*) que, não pertencendo ao IBS, são auxiliares de navegação de grande importância.

INS:

O INS é defenido pelo *American Pratical Navigator* [5] como um conjunto de equipamentos e módulos interconectados e ecrãs para apresentar de forma compreensiva e sucinta a informação relativa à navegação ao navegante.

Este sistema é composto por:

- **Agulha Magnética:** indica o Norte Magnético. É constituído por uma rosa dos ventos graduada ($000° - 360°$), com um ponteiro que gira dentro de um líquido anticongelante.
- **Agulha Giroscópica:** tem a vantagem de não ser influenciada pelo magnetismo terrestre nem pelos ferros do navio, indicando assim o Norte Verdadeiro uma vez que mede o ângulo entre a proa do navio e o eixo de rotação do giroscópio.
- **Electromagnetic Speed Log:** mede a velocidade do navio em relação à agua (velocidade que a máquina está a fornecer).
- **Doppler Speed Log:** mede a velocidade do navio em relação ao fundo (velocidade real do navio).
- **Anemómetro:** mede a direção e a velocidade aparente do vento.
- **Turn Indicator:** informa o ângulo do leme.
- **Piloto Automático:** é um controlador de direção que, de forma automática controla o leme, procurando diminuir o erro entre a proa de referência e a proa verdadeira. Permite controlar de forma manual o leme do navio através de uma combinação de sistemas elétricos, mecânicos e hidráulicos na ponte.
- **ADF** (*Detetor automático de direção*): é um aparelho que determina a direção de um sinal eletromagnético recebido.
- **GNSS** (*Global Navigation Satellite System*): processa o sinal transmitido por satélites determinando a posição, a velocidade e o tempo preciso.
  - **GPS** (*Global Position System*): é o usado atualmente a bordo.

LRIT:

A IMO define LRIT como um sistema que providência uma identificação e monitorização dos navios global.

AIS:

Definido pela IMO, o sistema de identificação automática troca informações de posição, velocidade, rumo e identificação de forma automática a terra e navios circundantes. Utiliza comunicação por VHF.

RADAR e ARPA:

O **RADAR** é descrito como "um sistema de deteção que usa ondas rádio para determinar a distância e o ângulo dos objetos" [30]. Um sistema de radar consiste em um emissor e recetor de ondas eletromagnéticas.

As ondas rádio do transmissor reflete nos objetos e retorna para a recetor, dando a informação do azimute e distância num ecrã. Assegura a segurança da navegação fornecendo informações obtidas da área circundante ao navio (ver Figura 2.13).

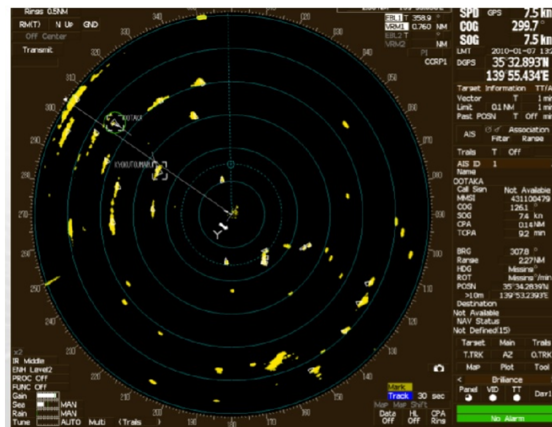

Figura 2.13: Ecrã de um Radar (Fonte: [30]).

Os navios operam nas bandas de frequência S e X pois são robustos em diferentes condições de tempo. Contudo, em distâncias muito pequenas, o radar poderá ter alguma dificuldade em encontrar pequenos objetos (ver Figura 2.14) e a informação pode ter alguma interferência (pelas condições atmosféricas) que dificulte a interpretação dos ecos (ver Figura 2.15).

Figura 2.14: Limitação do Radar para objetos a pequena distância (Fonte: [30]).

O **ARPA**, utiliza os sucessivos ecos captados pelo RADAR e calcula a velocidade e o rumo destes complementado as informações do RADAR. As informações de ambos os sistemas são fornecidas no mesmo ecrã de forma a facilitar a interpretação dos perigos à navegação e, consequentemente, a tomada de decisão [30].

Figura 2.15: Imagem com ruído (Fonte: www.nauticexpo.com/pt/prod/mi-simulators /product-26918-429207.html).

ECDIS:

Compila num ecrã toda a informação obtida por todos os sistemas anteriormente descritos, tendo como "tela" a informação geográfica contida nas cartas eletrónicas [17].

Auxilia o navegador no planeamento e monitorização da viagem de forma simplificada e intuitiva. À medida que o navegador insere os pontos de mudança de rumo que melhor se adequa à viagem, o sistema avalia e informa acerca dos perigos à navegação através de alarmes e de desvios ao planeamento que podem ocorrer.

Além disso, não requer que o operador realize correções às cartas, o sistema atualiza as mesmas automaticamente. Nas cartas em papel, o navegador precisa de as corrigir cada vez que uma correção/atualização ocorre.

2.4 Matemática da navegação

Nesta secção, procura-se descrever teoricamente os tipos de navegação e de derrotas. Podemos considerar quatro áreas distintas em que a navegação deve ser adaptada [5]:

- Cabotagem: Navegação em canais, rios e estuários;
- Portuária: Navegação em aproximação a portos, em portos e baías;
- Costeira: Navegação até 50 milhas da costa;
- Oceânica: Navegação em mar aberto.

Numa mesma viagem, podem ser realizados todos estes tipos de navegação e o tipo de derrota deve ser ajustada a cada área [20]. Os tipo de derrota podem ser divididos em quatro: loxodrómica, ortodrómica, mista e meteorológica e climatológica.

Loxodrómia:

É uma linha que faz o mesmo ângulo com todos os meridianos, sendo representada por uma linha reta nas cartas de Mercator. A principal vantagem da derrota loxodrómica é que esta mantém constante o rumo a navegar, isto é, um navio não precisa de alterar o seu rumo quando navega entre dois pontos sobre esta derrota [5].

A principal desvantagem é ser uma curva logarítmica espiral da terra, logo não é a distância mais curta entre dois pontos [20].

No entanto, para viagens de cabotagem, portuárias e costeiras, sendo viagens de poucas milhas, a distância de navegação não é significativa, logo esta é a forma mais prática de navegar.

Matematicamente, dados o F (ponto de partida) e o T (ponto de chegada) o rumo e a distância entre dois pontos podem ser calculados através da resolução de um triângulo plano [28] (ver Figura 2.16).

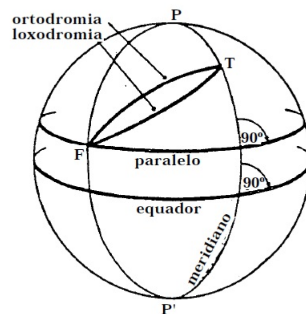

Figura 2.16: Derrota loxodómica (Fonte: [28]).

Considerando, por exemplo, uma navegação de latitude constante (ver Figura 2.17), temos que:

- $\overline{FT}$ é um arco paralelo, isto é, a distância ao longo do paralelo - *ap (apartamento)*[2]-, entre os meridianos que passam por F e por T;
- $\overline{AB}$ é a distância ao longo do equador entre os mesmos meridianos - $\Delta\lambda$ *(diferença de longitude).*

Temos, então, as secções DFT e CAB, que são paralelas e equiangulares, pelo que:

$$\frac{\overline{FT}}{\overline{AB}} = \frac{\overline{DF}}{\overline{CA}} \tag{2.1}$$

Tem-se também que:

[2] Distância entre dois meridianos sob o mesmo paralelo.

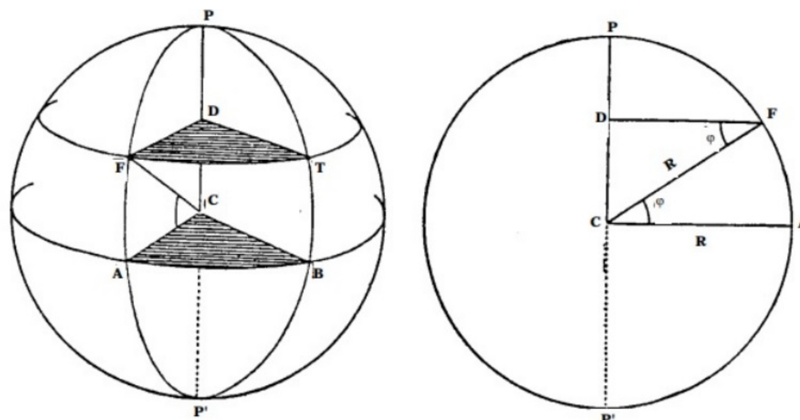

Figura 2.17: Navegação de latitude constante (Fonte: [28]).

$$\begin{aligned} \overline{DF} &= \overline{CF}\cos(\varphi) \\ \overline{DF} &= \overline{CA}\cos(\varphi) \end{aligned} \tag{2.2}$$

onde φ é a latitude.

Como $\overline{CF} = \overline{CA}$, por corresponderem ambos ao raio da terra, substituindo em 2.1, obtém-se

$$\frac{\overline{FT}}{\overline{AB}} = \frac{\overline{CA}\cos(\varphi)}{\overline{CA}} \Leftrightarrow FT = \Delta\lambda\cos(\varphi) \tag{2.3}$$

Quando as diferenças de latitude não são muito grandes (até 600 milhas) nem muito próximas dos pólos ($\varphi \leq 55°$), temos que:

$$ap = \Delta\lambda\cos(\varphi_m) \Leftrightarrow \Delta\lambda = ap.\sec(\varphi_m) \tag{2.4}$$

onde φ_m é a latitude média.

Assim, com recurso às relações trigonométricas, podemos calcular qualquer variável do triângulo de navegação (ver Figura 2.18), tais como:

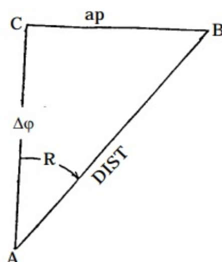

Figura 2.18: Triângulo de navegação plano (Fonte: [28]).

$$R = \arctan\left(\frac{\Delta\varphi}{ap}\right) \tag{2.5}$$

$$Dist = \sqrt{\Delta\varphi^2 + ap^2} \tag{2.6}$$

em que R é o rumo e $Dist$ a distância entre os pontos A e B.

Ortodrómia:

É feita sob o círculo máximo de navegação.

Um círculo máximo é a interseção entre a superfície da esfera e o plano que passa pelo centro da esfera. Corresponde à distância mais curta entre dois pontos, ao longo da superfície [5].

É usada especialmente em viagens oceânicas em que a diferença da distância entre dois pontos por uma derrota loxodrómica e por uma derrota ortodrómica é significativa.

Ao contrário da derrota loxodrómica, os ângulos formados com os meridianos não são constantes, de modo que a derrota ortodrómica é representada por uma linha curvilínea nas cartas de Mercator. Na prática, a implementação desta derrota é difícil, procedendo-se alternativamente pela sua divisão em pequenos segmentos loxodrómicos para uma fácil navegação [20].

Na ortodrómia, considera-se o triângulo esférico formado na superfície da Terra [28], como se ilustra na Figura 2.19:

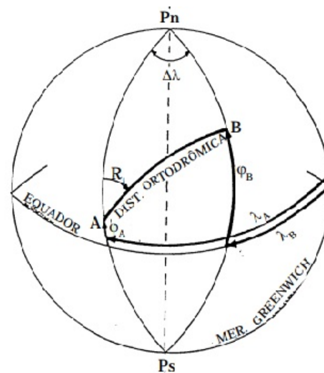

Figura 2.19: Derrota Ortodrómia (Fonte: [28]).

Os vértices representados são: o ponto de partida (A), o ponto de chegada (B) e o pólo elevado do ponto de partida (P_n). Todos os lados são arcos de círculo máximo (ortodrómias), sendo a linha AB a ortodrómia entre o ponto de partida e o ponto de chegada.

Com recurso às relações trigonométricas podemos calcular qualquer variável do triângulo de navegação (ver Figura 2.20)

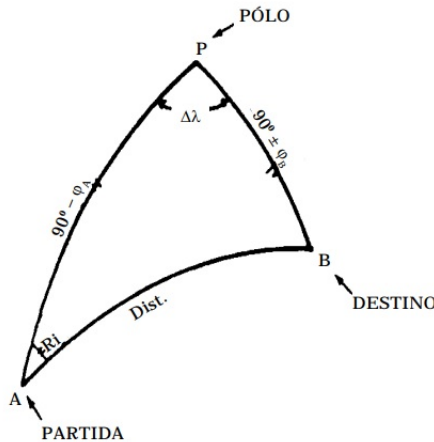

Figura 2.20: Triângulo de navegação esférico (Fonte: [28]).

Tem-se:

$$\cos(Dist) = \sin(\varphi_A)\sin(\varphi) + \cos(\varphi_A)\cos(\varphi_B)\cos(\Delta\lambda) \tag{2.7}$$

$$\cos(R_i) = \frac{\sin(\varphi_B) - \cos(Dist)\sin(\varphi_A)}{\sin(Dist)\cos(\varphi_A)} \tag{2.8}$$

onde R_i é o rumo inicial e $Dist$ é a distância entre os pontos A e B.

Derrota mista:

Uma derrota mista é uma combinação de ambas as derrotas discutidas anteriormente, isto é, a navegação é feita, primeiro, ao longo de uma linha ortodrómica até a um paralelo limite, depois, ao longo de uma linha loxodrómica sob um paralelo e, novamente, ao longo de uma linha ortodrómica até ao ponto de chegada [20].

A escolha do paralelo referido é feita de forma a evitar áreas perigosas como áreas de gelo, arquipélagos extensos e mau tempo em latitudes mais elevadas. O objetivo passa, portanto, por encontrar um equilíbrio entre uma derrota mais curta (a ortodrómia) e uma derrota mais segura evitando paralelos maiores [5].

Derrota meteorológica e climatológica (weather routing)

Estas derrotas originam uma rota ótima para viagens oceânicas dependendo das condições meteorológicas e climatológicas [5].

Condições de tempo adversas são condições meteorológicas com potencial de causar estrago, reduções de velocidade significativas e, consequentemente, o aumento de tempo de viagem, do consumo do combustível e do desconforto da tripulação. Neste tipo de

derrota, o objetivo é não só maximizar a segurança e o conforto da tripulação como também minimizar o consumo de combustível e o tempo de viagem.

Na derrota climatológica são usados dados históricos das condições de tempo. Nas publicações *Pilot Charts*[3] e *Sailing Directions*[4], são recomendadas rotas baseadas nas probabilidades de encontrar uma certa condição de tempo para determinadas alturas do ano e, na derrota meteorológica, são usados dados de previsão de tempo.

Os fatores ambientais que influenciam fortemente a navegação são:

- **Vento:** O efeito da velocidade do vento no desempenho do navio é difícil de determinar. Para ventos até 20 nós, o navio perde velocidade se o vento estiver de proa e ganha velocidade se o vento estiver de popa. Para velocidades maiores, o navio perde velocidade qualquer que seja a direção do vento devido ao aumento da ondulação. A influência do vento depende também do tipo de navio. Por exemplo, ventos fortes têm um efeito adverso maior num navio porta-contentores cheio que num navio tanque de dimensão semelhante devido à enorme área exposta ao vento.

- **Altura da ondulação:** é o fator que mais influência o desempenho do navio. A ação da ondulação é responsável pelos movimentos do navio, causando a redução da propulsão e o aumento do arrasto do navio. A relação entre o navio e a ondulação é semelhante à relação do navio e o vento: ondulação de proa diminui a velocidade do navio e ondulação de popa aumenta-a até certo ponto, diminuíndo-a a partir daí.

- **Nevoeiro:** apesar de não afetar diretamente o desempenho do navio, deve ser evitado sempre que possível, de forma a manter uma velocidade normal em condições seguras. Para evitar navegar em áreas extensas de nevoeiro, durante certas alturas do ano, deve-se reduzir a latitude máxima de navegação. Apesar da rota ficar mais longa, evitar o nevoeiro, reduzirá o tempo de navegação por não haver necessidade de reduzir a velocidade, e diminuirá a fadiga da tripulação.

- **Efeito de parede do Norte:** No Atlântico e Pacífico Norte, durante o inverno, a pressão atmosférica diminui consideravelmente, aumentando imprevisivelmente os ventos e a ondulação. Esta área deve ser evitada reduzindo a latitude máxima.

- **Correntes oceânicas:** Não constituem um problema de roteamento significativo, mas podem ser determinantes na seleção da rota, especialmente em baixas latitudes.

[3]Cartas que contém a média da prevalência das condições meteorológicas numa área para diferentes alturas do ano.

[4]Manuais que contém informação essencial sobre todos os aspetos da navegação.

- **Gelo:** A navegação em áreas de gelo deve ser evitada porque aumenta a probabilidade de colisão devido à dificuldade de deteção.

3 Métodos para a determinação automática de derrotas

Podemos afirmar que um navio autónomo não implica que não tenha tripulação a bordo. No entanto, o inverso é necessário mas não suficiente. Nos capítulos anteriores, não só vimos a evolução de navios autónomos e navios tripulados mas também as vantagens e limitações de diferentes aparelhos a bordo.

Na primeira parte deste capítulo, descrevemos alguns dos algoritmos existentes na literatura para a determinação automática de derrotas, considerando os seus bons resultados em várias temáticas, a sua fácil compreensão e implementação, assim como as suas limitações.

Tendo por base algumas das ideias usadas no desenvolvimento dos algoritmos descritos, propomos, na segunda parte deste capítulo, uma nova abordagem, com o intuito de serem determinadas derrotas ótimas entre quaisquer dois pontos predefinidos num mapa. Uma das vantagens desta abordagem é a capacidade de definir uma derrota que contorne qualquer área não navegável, desde que esta seja identificada antes da partida do navio. Numa situação dinâmica, em que despontam obstáculos que dificultam ou impossibilitam a navegação do navio após a partida deste, a derrota poderá já não ser válida, tendo de ser ajustada em conformidade.

3.1 Algoritmo do polígono convexo

Dados n pontos num plano, o objetivo consiste em encontrar o "casco convexo" desses pontos. Entenda-se por "casco convexo" como polígono convexo mais pequeno que contém esses pontos. Se visualizarmos cada ponto como pregos a sobressair de uma tábua, o casco convexo pode ser representado por uma banda elástica (do inglês, *rubber band*) que compreenda todos os pregos [6]. Por exemplo, a Figura 3.1 ilustra o polígono convexo que engloba todos os pontos representados, de P_0 a P_{12}. Como se pode observar, nem todos esses pontos são vértices do polígono, de onde é necessário definir uma forma de os identificar. De seguida, descrevemos um algoritmo, dividido em passos simples, que

permite determinar, para um dado conjunto de n pontos não colineares (com $n > 2$), quais os que devem ser escolhidos como vértices e os que devem ser rejeitados [6].

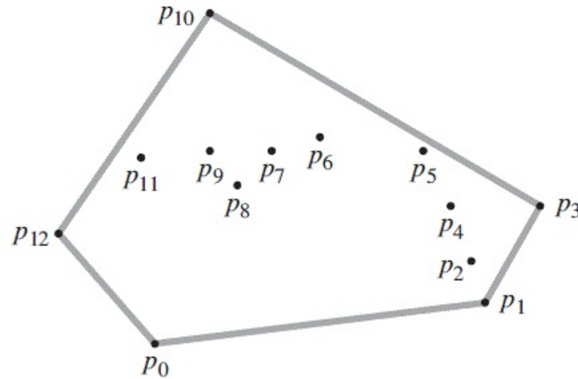

Figura 3.1: Conjunto de pontos com o seu polígono convexo (Fonte: [6], pág.1029).

1. Ler os dois primeiros pontos da aresta inicial (P_0 e P_1).
2. Se o P_1 for o ponto final, parar.
3. Se não, ler o próximo ponto (P_2).
4. Se P_0, P_1 e P_2 forem colineares, rejeitar P_1 e renomear P_2 como P_1. Voltar ao passo 2.
5. Caso contrário, verificar se existe algum obstáculo entre P_0 e P_2.
6. Se não existir, rejeitar P_1, renomear P_2 como P_1, unir os vértices P_0 e P_1 (renomeado) e voltar ao passo 2.
7. Repetir o processo até que o ponto final seja o ponto de chegada.

Xu et al. [46] utilizou este algoritmo para abordar o problema da seguinte forma: são retiradas das cartas náuticas vetoriais[1] e raster[2] informação referente aos obstáculos estáticos, a partir da qual é criada não só uma zona de segurança em redor dos mesmos como também é extraída uma matriz de projeção dos obstáculos, usando o *Polygon Raster Processing*, categorizando cada obstáculo como um polígono (ver Figura 3.2). Depois, tendo como dados de entrada dois pontos, um de partida e outro de chegada, digamos A e B, respetivamente, estes são unidos por uma reta. Se a reta resultante intersetar um

[1] A informação está disposta em diferentes camadas de informação, o que permite, ao navegador, escolher a informação que necessita para operação a utilizar, mas que torna o planeamento da derrota mais complexa devido aos polígonos côncavos.

[2] Com uma estrutura de dados mais simples e, consequentemente, um esforço computacional menor.

obstáculo, são criados dois novos pontos, ambos na interseção desta reta com os limites do obstáculo, digamos, C na entrada e E na saída deste. Assim, o caminho resultante livre de obstáculos será constituído pela linha reta que une os pontos A e C, a linha que circunda o obstáculo entre os pontos C e E (pela esquerda ou pela direita) e a reta entre os pontos E e B (ver Figura 3.3). Daqui, é possível definir um grafo que resulta da ramificação sucessiva de caminhos e do aparecimento de novos pontos, ou vértices, à medida que aqueles intersetam obstáculo estáticos.

Figura 3.2: Matriz de projeção de obstáculos (Fonte: [46]).

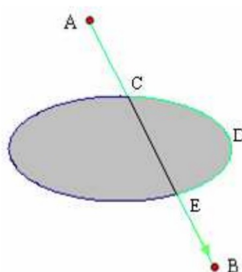

Figura 3.3: Rota primária (Fonte: [46]).

Para minimizar este percurso, é calculado e comparado o perímetro entre os pontos C e E em ambos os sentidos e selecionado o de menor valor. Nas Figuras 3.3 e 3.4, o ponto D corresponde ao ponto mais extremo do segmento de perímetro selecionado. Para terminar, é usado o algoritmo do polígono convexo para suavizar esta curva, minimizar o número de pontos de interseção e a distância percorrida (ver Figura 3.4).

No trabalho [46], os autores concluíram que, em zonas de muitos obstáculos, este algoritmo consegue gerar uma rota segura e curta devido à sua flexibilidade.

Não obstante, uma das desvantagens deste procedimento é apontada no artigo [16], onde o desempenho do algoritmo tende a se degradar com o aumento da quantidade de obstáculos, uma vez que a expansão do grafo resultante é de tal ordem que acaba por incluir obstáculos que não terão impacto no percurso (Figura 3.5).

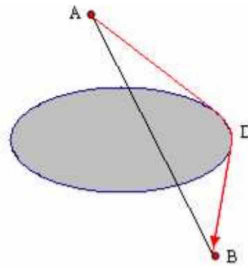

Figura 3.4: Rota final (Fonte: [46]).

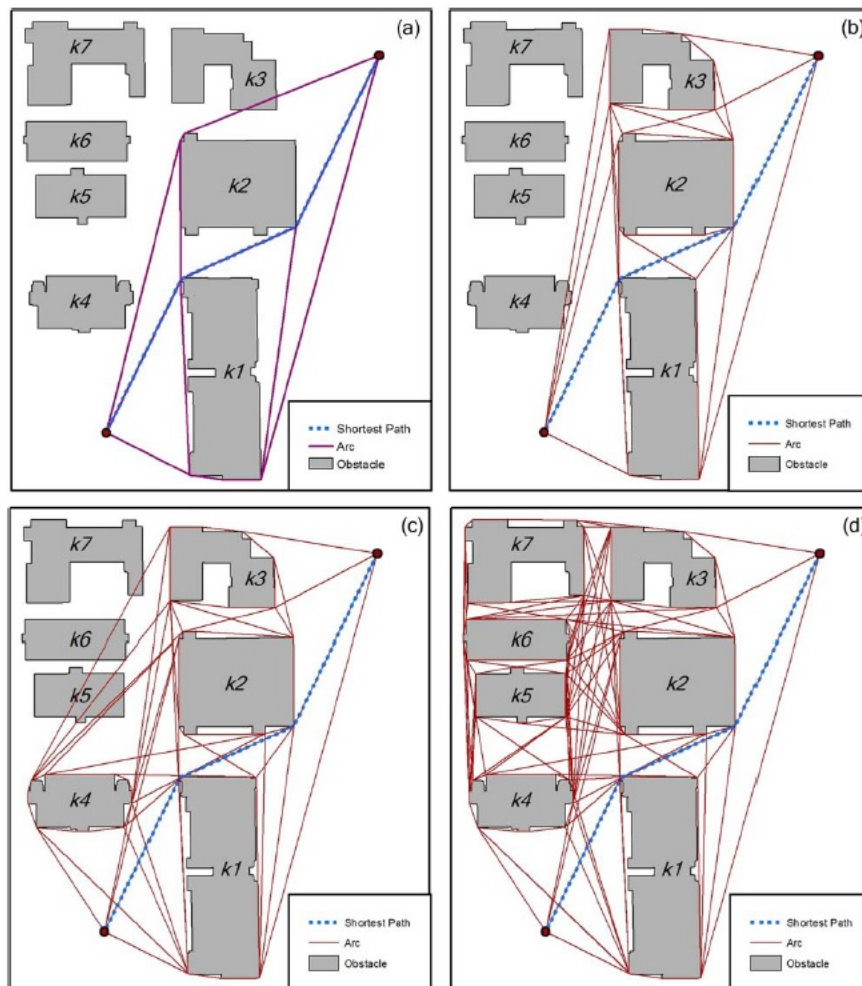

Figura 3.5: Expansão do grafo resultante do algoritmo do polígono convexo (Fonte: [16]).

A imagem (a) da Figura 3.5 resulta da execução do algoritmo do polígono convexo em torno dos obstáculos k_1 e k_2. Como um dos troços dos caminhos possíveis interseta o obstáculo k_3, é necessário uma nova ramificação de caminhos e de criação de novos pontos (imagem (b)). Eventualmente, esses caminhos intersetarão outros obstáculos e, novamente, serão necessárias novas ramificações e mais pontos (imagens (c) e (d)), sendo este procedimento repetido até que nenhum troço intersete nenhum dos obstáculos. Na

ultima imagem da Figura 3.5, é mostrado o caminho mais curto do grafo resultante, cujos vértices que fazem parte do mesmo já se encontravam na imagem inicial, antes das sucessivas ramificações e consequente aumento do grafo. Portanto, as arestas resultantes das interações subsequentes aumentam não só a complexidade do problema como também o custo computacional.

Os autores referem que a eficiência do algoritmo pode ser melhorada com a eliminação de alguns obstáculos sob certas condições, desde que não invalidem o grafo ou a escolha da solução ótima.

Como o caminho mais curto são os limites mínimos de cada interseção, é computacionalmente mais eficiente avaliar apenas os obstáculos que impedem o caminho mais curto em cada interseção e não em todas as arestas. Isto é, quando em uma interação já for obtido o caminho mais curto livre de obstáculos, qualquer outra interação adicional ramificará arestas que não melhorarão a solução já descoberta.

Na abordagem que propomos, utilizaremos este algoritmo para suavizar e minimizar a distância total percorrida, assim como o número de pontos de mudança de rumo, em conjunto com o algoritmo de Dijkstra, descrito mais à frente, pela sua flexibilidade e facilidade de implementação. Como o conjunto dos caminhos possíveis já é livre de obstáculos, a probabilidade de aumentar o custo computacional pelo crescimento exponencial das ramificações é mínima.

3.2 Algoritmo de Dijkstra

Com o objetivo de encontrar, num grafo conexo, o caminho mais curto entre dois pontos P e Q, desde que exista, Dijkstra [8] propõe o seguinte: se R é um vértice pertencente ao caminho mais curto entre P e Q, implica que o caminho mais curto entre P e R é conhecido. O objetivo é aumentar iterativamente o comprimento do caminho mais curto conhecido até o ponto Q ser atingido.

Os vértices são divididos em três conjuntos disjuntos:

A: os vértices serão adicionados a este conjunto de forma a aumentar o caminho mínimo desde P.

B: compreende todos os vértices que estão conectados a pelo menos um vértice do conjunto **A**, com o potencial de serem os próximos selecionados.

C: os restantes vértices.

As arestas são também divididas em três conjuntos:

I: as arestas usadas no caminho mínimo desde o vértice P até aos vértices do conjunto **A**.

II: as arestas seguintes a serem selecionadas para o conjunto **I**.

III: as arestas restantes.

Antes do algoritmo se iniciar, os vértices estão no conjunto **C** e todas as arestas no conjunto **III**. O algoritmo começa colocando o vértice inicial P no conjunto **A**. Os conjuntos de vértices **B** e **C** ficam logo definidos, assim como os conjuntos de arestas **I**, **II** e **III**.

Passo 1: sejam todas as arestas r conectadas ao ultimo vértice transferido para o conjunto **A**, com o vértice R pertencente ao conjunto **B**:

- analisamos se esta aresta r nos leva por um caminho mais curto entre P e R que a aresta já conhecida do conjunto **II**:
  - se não, a aresta r é rejeitada;
  - se de r resulta um caminho mais curto entre P e R, até agora obtido, substitui a aresta correspondente do conjunto **I** e rejeitar a aresta inicial.

Passo 2: Cada vértice do conjunto **B** tem associada uma distância total ao vértice inicial. O vértice com a distância mínima é transferido para o conjunto **A** e retornamos ao passo 1 repetindo o processo até que o vértice Q seja transferido para o conjunto **A**. A solução é então encontrada.

Lee et al. [24], Novac et al. [32] e Wang [43] utilizam o algoritmo de Dijkstra nas suas abordagens. O maior desafio deste algoritmo passa pela construção do grafo em que o algoritmo correrá, assim como na definição da função objetivo a minimizar.

Wang [43] constrói um grafo baseado no círculo máximo de referência entre o ponto de partida e o ponto de chegada. A região do espaço é dividida em n passos ao longo da derrota ortodrómica. Em cada passo i $(1 \leq i \leq n)$, são predefinidos pontos perpendiculares ao ponto de referência (ver Figura 3.6). O peso de cada aresta corresponde ao tempo de viagem que o navio demora a percorrer esse percurso. Nesta abordagem, o tempo de viagem depende das condições meteorológicas, que tem muita influência na velocidade de navegação. O algoritmo de Dijkstra fornece uma rota ótima primária que serve de base para a algoritmo genético.

Novac et al. [32] utiliza este algoritmo num caso de estudo para a navegação no Mar Negro entre Sinop $(42,35283°N; 035,14250°E)$ e Sevastopol $(44,68614°N; 032,47586°E)$.

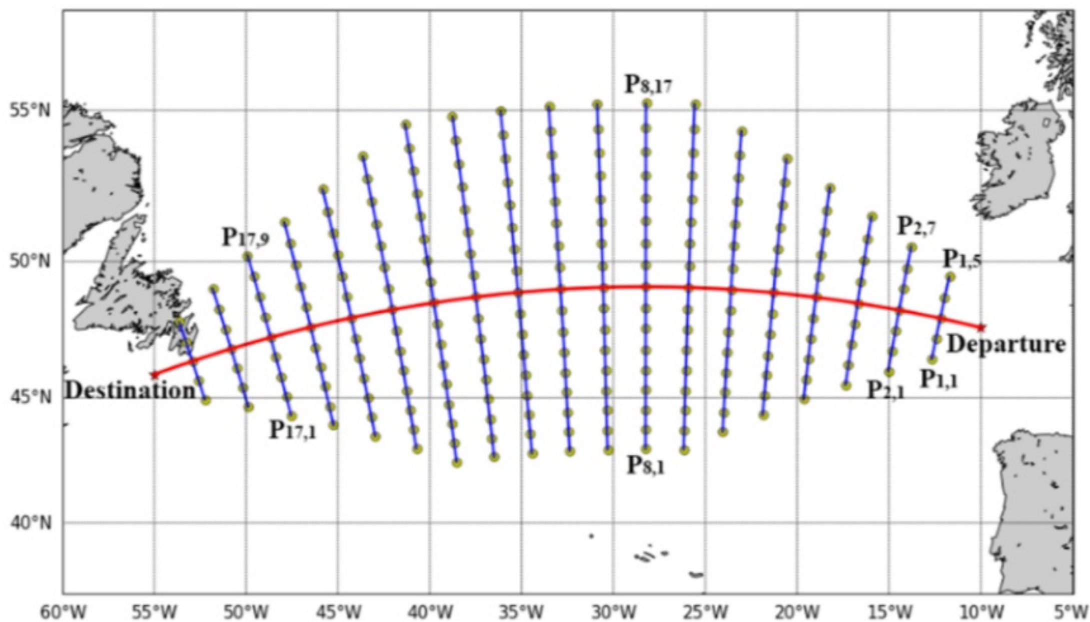

Figura 3.6: Grelha predefinida subjacente a uma derrota ortodrómica (Fonte: [43].

O grafo é construído usando uma malha com 72 vértices com arcos que conectam os vértices em ângulos de 000°, 045°, 090°, 135°, 180°, 225°, 270° e 315°. A resolução espacial do vértice é de $0,33333°$, sendo atribuído um peso igual a 1 a todas as arestas não diagonais. As arestas diagonais têm um peso de $\sqrt{2}$ e as arestas não navegáveis um peso de 99. As arestas são definidas como navegáveis ou não navegáveis a depender das condições meteorológicas o permitirem ou não. Em arestas em que a ondulação é superior a 4m, são consideradas como não navegáveis. Uma aresta de peso 1 corresponde a uma distância de 19,9998 milhas náuticas (ver Figura 3.7).

Os autores concluíram que a diferença obtida entre o caminho com bom tempo e o caminho com mau tempo, obtidos pelo algoritmo, não é muito grande. O caminho gerado pelo algoritmo tem um maior nível de emissões, mas é o mais seguro, por evitar as arestas definidas como não navegáveis.

Em [24], o algoritmo de Dijkstra é usado para encontrar o caminho mais curto entre o ponto de partida e o ponto de chegada. Para a construção do grafo, é usada uma estrutura *quadtree* (ver Figura 3.8). Esta estrutura de dados reduz o espaço necessário para a busca. Este processo de representação divide (ou não) uma área em subestruturas. Esta divisão é feita tendo em conta se as arestas de um bloco são livres de obstáculos ou não. Se for livre de obstáculos, o bloco e as respetivas arestas são aceites. Se não for, o bloco é dividido em quatro sub-blocos iguais e cada bloco é analisado individualmente, repetindo o processo até que todos os blocos sejam construídos por arestas que não intersetam

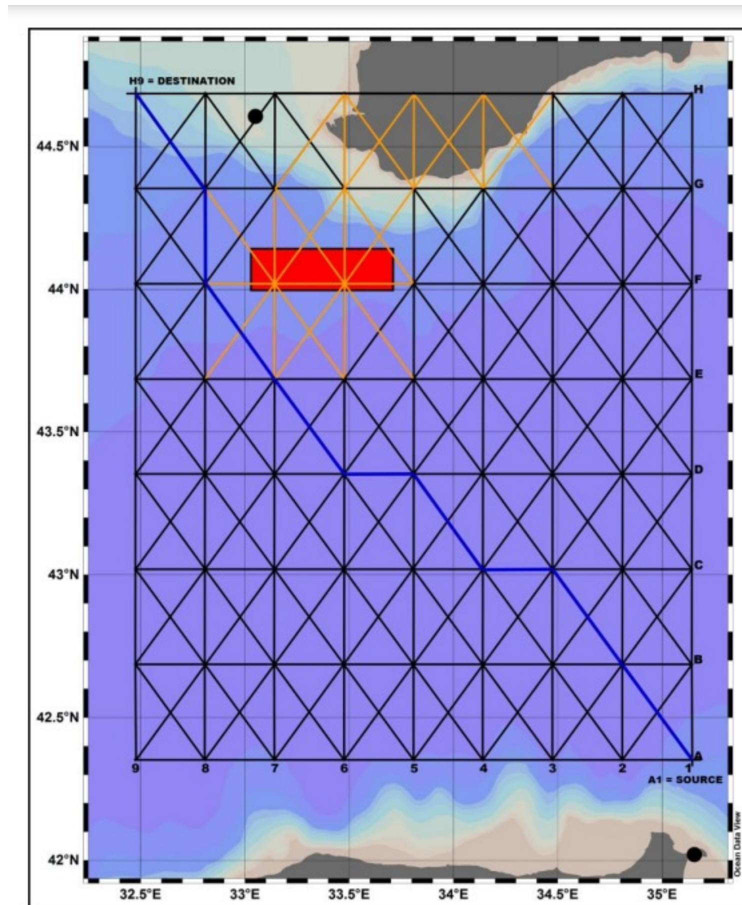

Figura 3.7: Definição da grelha com 72 vértices (Fonte: [32]).

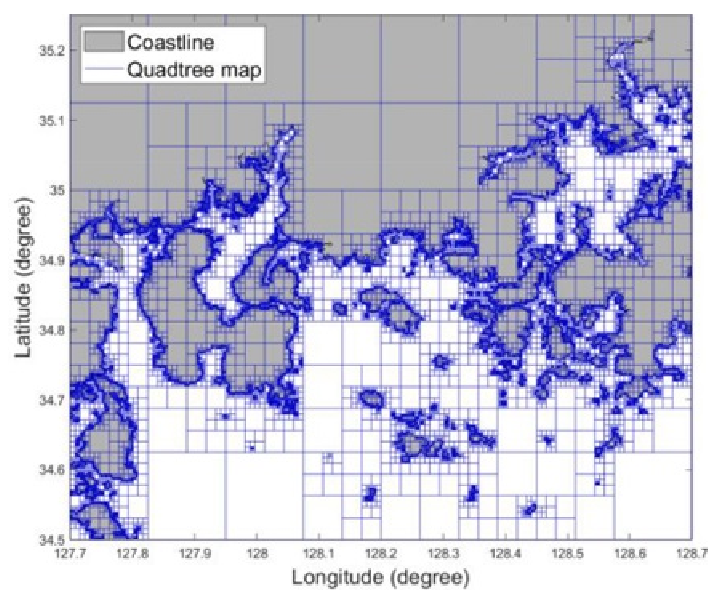

Figura 3.8: Estrutura *quadtree* (Fonte: [24]).

nenhum obstáculo.

Um vértice *quadtree* tem as seguintes propriedades:

- fronteira;
- vértice vizinho;
- 4 vértices vizinhos;
- propriedade transitável;
- arestas dos obstáculos do bloco *quadtree*.

Assim, cada vértice é conectado em 4 direções. Para o algoritmo de Dijkstra, dois tipos de arestas são removidos: as arestas que intersetam as linhas de costa e as que estão conectadas a vértices na linha de costa. Esta linha de costa é definida previamente usando uma extensão de fronteira de 100m. Deste passo, resulta um grafo "mais limpo". Os pesos associados às arestas correspondem aos tempos de navegação entre os vértices correspondentes, considerando as variações de velocidade devido às condições meteorológicas.

Freitas [12] utiliza o algoritmo de Dijkstra com o objetivo de otimizar os recursos energéticos, aumentando a eficiência das derrotas planeadas. Começa por modelar o navio do tipo corveta de forma a calcular as resistências ao avanço proporcionadas pelos ventos e pela ondulação. Assim, consegue monitorizar as perdas involuntárias do navio. Para os dados meteorológicos, o autor trabalha com dados do modelo GFS (*Global Forecast System*), que têm uma resolução maior ($0,5^\circ \times 0,5^\circ$) mas pouca informação relativa à ondulação e do modelo WW3 (*Wave Watch III*), que contém toda a informação necessária mas com uma resolução menor ($1^\circ \times 1,25^\circ$). As duas matrizes são então convertidas numa única matriz (passo 1, da Figura 3.9). Assim, cada vértice do grafo contém informação de posição e meteorológica. Em cada vértice, é registada a altura da ondulação. Se não for um número, é automaticamente atribuído o valor infinito. Aos arcos com uma extremidade em um desses vértices são também atribuídos um peso infinito para que o algoritmo nunca os considere na determinação da rota ótima. Para cada vértice, é calculado uma distância para todos os 8 vértices adjacentes pertencentes ao pontos cardeais e colaterais (passos 2 e 3, da Figura 3.9). Posteriormente, são calculadas as marcações de vento e mar de modo a poderem ser obtidos valores para a perda de velocidade. O consumo referente ao regime da máquina está associado à velocidade adquirida em cada troço como o custo do arco (passo 4, da Figura 3.9). O algoritmo de Dijkstra é depois usado para minimizar o custo total ao longo do percurso. Para os testes de validação, são usados os Diários de Bordo com as posições e os dados meteorológicos retirados em cada viagem.

Figura 3.9: Estrutura das variáveis utilizadas para definir o custo das arestas (Fonte: [12]).

3.3 Algoritmo A*

O algoritmo A* é uma extensão do algoritmo de Dijkstra, onde são usados o custo estimado do vértice atual (representado por n) até ao ponto final, denotado por $h(n)$, e o custo estimado do ponto inicial até ao vértice atual, denotado por $g(n)$.

Em cada instante, é escolhida a melhor rota analisando a função $f(n) = h(n) + g(n)$ no vértice atual n [7].

O algoritmo A* é usado nos trabalhos de Grifoll et al. [13] e Novac e Rusu [31].

No grafo utilizado em [13], os vértices resultam da interseção de paralelos e meridianos de $1,5 \times 1,5$ minutos da área a navegar (Barcelona-Palma de Mallorca) e o peso das arestas que ligam os vértices corresponde à distância.

A função a minimizar é:

$$f(n) = g(n) + h(n) \tag{3.1}$$

e corresponde ao tempo de viagem de navegação. Este tempo de viagem está associado à velocidade de navegação e, consequentemente, às condições meteorológicas que contribuem

para as reduções de velocidade.

A variável meteorológica selecionada para este estudo foi a ondulação, usando os dados de previsão entre dezembro de 2016 e março de 2017. A previsão para a redução da velocidade foi feita usando a fórmula de Bowditch:

$$V = V_0 - f(\theta) \times W^2 \tag{3.2}$$

onde:
W: a altura da ondulação;
$f(\theta)$: direção relativa entre o navio e a ondulação;
V_0: velocidade sem ondulação.

Os autores concluíram que a comparação das rotas dos navios (tempo de viagem ótima vs distância mínima) evidencia a relevância dos efeitos da ondulação para a redução das velocidades. Mesmo para distâncias curtas, esta redução pode chegar a 7% do tempo total. Consequentemente, o benefício do roteamento dos navios dependerá da altura da ondulação, da direção e da sequência espacial da tempestade.

No grafo utilizado por Novac e Rusu [31], os vértices resultam da interseção de paralelos e meridianos de 2×2 minutos da área a navegar, desta feita, no Mar Negro, resultando em um grafo com 99×271 vértices, categorizados como navegáveis ou não navegáveis, sendo os não navegáveis vértices correspondentes a terra, a baixios e a ondulações superiores a 4m. A função a ser minimizada corresponde à definida na Equação 3.1.

O processo foi semelhante nos dois artigos, diferenciando-se apenas na abordagem das condições meteorológicas. Novac e Rusu consideram os pontos cuja altura da ondulação é superior a 4m como área não navegável e Grifoll et al. consideram-a como navegável mas tem em conta a influência da ondulação na redução da velocidade pela Equação 3.2.

3.4 Métodos baseados em grelhas dinâmicas

Genericamente, os métodos baseados em grelhas dinâmicas podem ser descritos como uma minimização do consumo de combustível, do tempo de viagem, de distâncias percorridas, entre outras, considerando as especificações da máquina e as variações de proa e/ou de rumo[3]. As variáveis de controlo, nomeadamente, o poder da máquina e a proa/rumo são

[3]Rumo é o ângulo formado entre a loxodrómia e os meridianos. A proa é o ângulo com que o navio tem de navegar de forma a fazer cumprir o rumo. Em condições perfeitas em que todas as variáveis fossem consideradas pelo algoritmo e/ou não existisse nada que fizesse com que o navio se desviasse da

consideradas constantes entre duas iterações consecutivos (ilustrado, como exemplo, na Figura 3.10). O estudo realizado por Walther et al. [42] compara diferentes abordagens destes métodos.

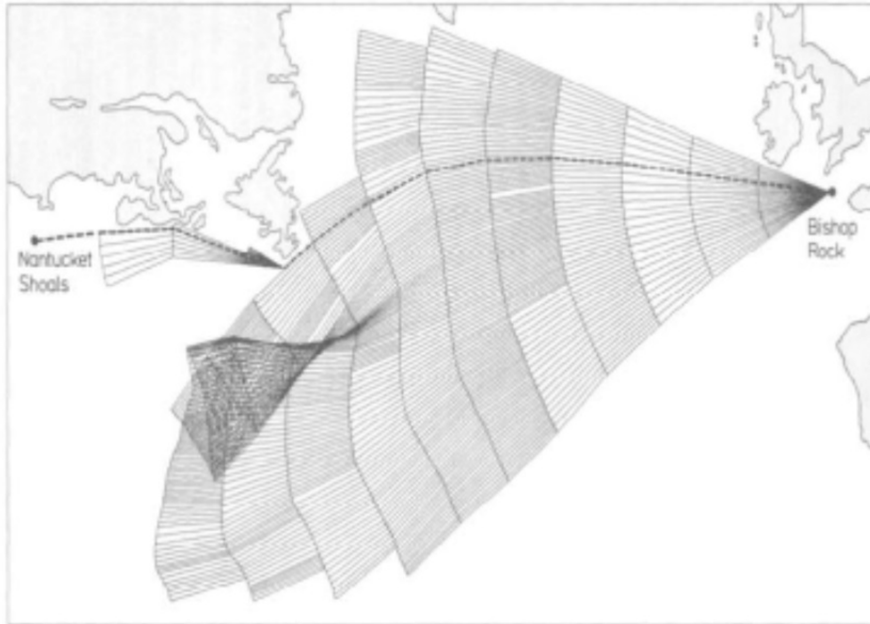

Figura 3.10: Cálculo do tempo mínimo (Fonte: [42]).

O processo começa com pequenas variações de proa/rumo num ponto inicial a cada ponto da grelha. As arestas que violam restrições, como interseção de obstáculos, são abandonadas. Os dados de cada aresta são guardados e comparados em termos da função a minimizar, seja o combustível, o tempo ou a distância. No fim, são escolhidas as arestas de valores mínimos.

Método das isopones

As isopones representam planos de igual consumo de combustível, estando definidas em latitude, longitude e tempo.

A primeira isopone é determinada calculando os pontos fronteiros alcançáveis, partindo do ponto inicial, com uma quantidade fixa de combustível, tendo como referência a ortodrómia (ilustrado na Figura 3.11).

Depois, todos os pontos resultantes da primeira iteração são considerados como pontos iniciais e para cada um deles, resulta um conjunto de pontos fronteiriços em que a aresta corresponde a uma quantidade fixa de combustível gasto.

O procedimento é repetido até ao ponto de destino. O caminho do consumo mínimo é reconstruído seguindo o reverso das proas e velocidades usadas em cada ponto, ou seja, o

rota, a proa e o rumo corresponderiam ao mesmo ângulo.

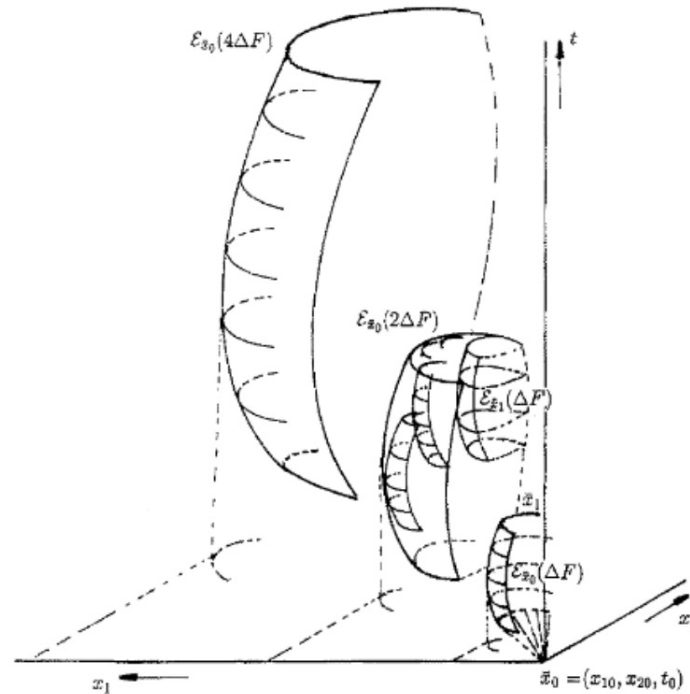

Figura 3.11: Distância que o navio consegue navegar, num dado intervalo de tempo, com uma quantidade de combustível predefinida (Fonte: [42]).

percurso com menos vértices. Se o objetivo for minimizar o tempo de viagem, as arestas escolhidas são aquelas em que a velocidade é maior para o mesmo consumo de combustível.

Método das isócronas

Genericamente, as isócronas caracterizam-se por linhas que unem pontos de rota correspondentes ao mesmo tempo de viagem. As primeiras isócronas são formadas por linhas criadas no mapa partindo de um ponto inicial até um ponto a que o navio chegaria ao fim de um tempo constante numa mesma proa/rumo. Assim, os diferentes pontos possíveis resultam em variações de proa/rumo (ver Figura 3.12).

As variáveis são a velocidade do navio e a proa/rumo, sendo que cada ponto é definido pela localização, tendo em conta as condições meteorológicas e as reduções de velocidade, voluntárias ou não.

O trabalho de Wang [43] considera que a grelha construída usando o método das isócronas é mais flexível que outros métodos, porque muitos fatores podem influenciar o "cálculo iterativo" e, consequentemente, a criação da grelha em si. Por exemplo, reduzindo o intervalo de tempo entre cada iteração, a qualidade, a precisão dos resultados e o custo computacional aumentam consideravelmente.

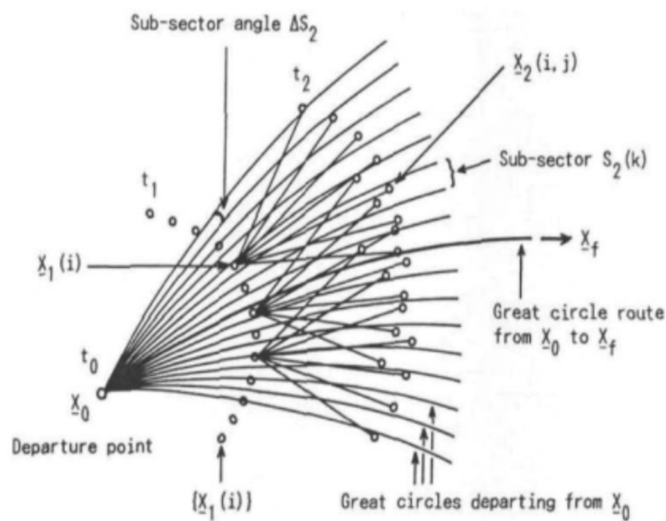

Figura 3.12: Determinação das isócronas usando a rota de referência (Fonte: [42]).

Em cartas náuticas de escala muito grande, onde a área representada é pequena mas com um nível de pormenor maior (por exemplo, a aproximação ao porto de Brest como o ilustrado na Figura 3.13), acreditamos que, uma abordagem semelhante à encontrada nos artigos [43], [48] e [42], embora com algumas adaptações, poderia ser interessante. Assumindo que a velocidade seria constante, uma vez que não estamos a considerar os efeitos meteorológicos e que queremos otimizar a distância navegável, e partindo de um ponto dado pertencente ao diagrama da Figura 3.13, os próximos pontos possíveis são os pontos pertencentes a uma distância fixa que variam em 1° de variação de rumo. O comprimento fixo das arestas foi pensado usando conceito de pontos não retorno [4] e curvas evolutivas [5] de um navio.

Embora esta abordagem não tenha sido explorada neste trabalho, consideramos que é algo a considerar futuramente.

3.5 Método proposto

Nesta secção, propomos um novo método para o planeamento de derrotas automáticas ótimas, com vista à minimização da distância navegada e do número de pontos de mudança

[4]Definido como uma posição onde o navio entra em águas tão restritas que não haverá espaço para regressar ou onde não poderá reentrar devido à alteração da maré ou a quantidade de água disponível abaixo da quilha [17].

[5]É uma característica de navegabilidade definida e padronizada pelo IMO, que sumariza o comportamento de um navio sob mudanças de rumo.

Figura 3.13: Carta náutica 3427 (Fonte: Arquivo Escola Superior Náutica Infante D. Henrique).

de rumo. Todos os algoritmos descritos nesta secção, assim como o código listado em anexo, são originais. Os conceitos foram criados e desenvolvidos por mim, com auxílio do orientador da dissertação.

Esta proposta tem por base algumas das ideias apresentadas na literatura, tais como a construção de grelha de pontos, a aplicação do método de Dijkstra para o caminho mais curto e o método do polígono convexo.

Uma das características da abordagem que se propõe aqui (e que consideramos ser uma das suas vantagens) é a criação de uma grelha de pontos com um nível de refinamento (ou precisão) predefinido, ajustada ao tipo de navegação. Isto é, numa área de navegação mais condicionada, como a aproximação a um porto, considera-se uma grelha com um nível de refinamento mais elevado o que permite aumentar a segurança à navegação. Para áreas de navegação mais abertas, ou "mais livres", a grelha é ajustada para uma menor precisão, com a vantagem de se ver reduzido o custo computacional. Deste modo, é possível obter um equilíbrio entre a precisão, a segurança à navegação e o custo computacional.

A descrição do método proposto é acompanhada com a apresentação de um caso estudo, descrito a seguir.

Dados de entrada:

Para a aplicação do método proposto, são necessárias as coordenadas do ponto de partida e de chegada, a carta de navegação da área a navegar e as coordenadas dos pontos que constituem a grelha, com uma precisão predefinida e ajustada ao tipo de navegação.

A área de navegação em estudo é a entrada Oeste do canal da Mancha (situada na costa Oeste Francesa), entre *Brest* e um ponto a Norte do Esquema de Separação de Tráfego (EST) de *Ouessant*. Desta forma, a área abrange e navegação costeira passando pelo EST do *Ouessant*, que regulamenta e organiza o tráfego marítimo nesta área. A área de navegação baseia-se na carta naútica $n°2655$, representada na Figura 3.14, intitulada *English Channel Western Entrace* e uma escala de 1 : 325000.

Os pontos de partida e de chegada, pertencentes a esta zona, deverão ser tais que a derrota final passe pelas áreas acima citadas. Assim, um deles será a norte do EST de *Ouessant* e o outro a Sul.

Caracterização da área a navegar

Cada ponto de uma carta náutica pode ser definido como navegável ou não navegável, podendo haver uma simplificação das informações presentes na carta, isto é, toda a informação relativa às ajudas à navegação pode ser omitida. Para que o método proposto funcione, basta saber que pontos e arestas são possíveis de navegar e quais não. Neste

sentido, é suficiente proceder-se a uma simplificação da representação da carta náutica.

A carta náutica $n°2655$ (Figura 3.14) foi simplificada na Figura 3.15.

Nesta simplificação estão representados:

- a área compreendida entre:

$$Canto\ inferior\ direito = \begin{cases} \varphi = 48°\ \ 00,0'\ N \\ \lambda = 004°\ \ 20,0'\ W \end{cases} \tag{3.3}$$

$$Canto\ superior\ esquerdo = \begin{cases} \varphi = 49°\ \ 20,0'\ N \\ \lambda = 006°\ \ 10,0'\ W \end{cases} \tag{3.4}$$

  onde φ é a latitude e λ a longitude.

- a linha de costa: com pontos retirados ao longo da costa em intervalos de 2' de latitude e, sempre que se justificasse por uma irregularidade da costa, algum ponto específico (ver Tabela 3.1);

- *Île D'Ouessant*: estende-se por 10 milhas a Oeste da extremidade Noroeste da costa francesa. Ao redor da ilha, num raio de 1,5 milhas, existem diversos perigos à navegação. Assim, este obstáculo pode ser definido pelos pontos: $(48°27,5'N; 005°07,5'W)$, $(48°29,0'N; 005°04,0'W)$, $(48°28,0'N; 005°01,5'W)$ e $(48°25,5'N; 005°08,0'W)$;

- *Île de Molene* e arredores: definida pelos pontos: $(48°26,0'N; 005°01,5'W)$, $(48°25,5'N; 004°53,5'W)$, $(48°22,0'N; 004°47,5'W)$ e $(48°17,5'N; 004°49,0'W)$;

- *La Parquette*: conjunto de ilhas que constituem um perigo à navegação numa área definida pelos pontos: $(48°16,0'N; 004°44,0'W)$, $(48°15,0'N; 004°48,0'W)$, $(48°13,0'N; 004°45,5'W)$, $(48°11,5'N; 004°37,5'W)$ e $(48°08,0'N; 004°36,0'W)$;

- *Chaussée de Sein*: conjunto de ilhas, rochas e recifes, com 12 milhas de comprimento, com a sua extremidade Oeste a 13,5 milhas de *Pointe du Raz*, definido pelos pontos: $(48°04,0'N; 005°08,0'W)$, $(48°04,5'N; 004°47,5'W)$ e $(48°01,5'N; 004°49,0'W)$;

- o Esquema de Separação de Tráfego - sentido Norte e sentido Sul - representados nos diagramas das Figuras 3.16 e 3.17, respetivamente;

- o posicionamento da Carta náutica $n°3427$.

Nesta representação, a grelha deverá ser definida em intervalos de $1'$, quer em latitude, quer em longitude, o que permitirá:

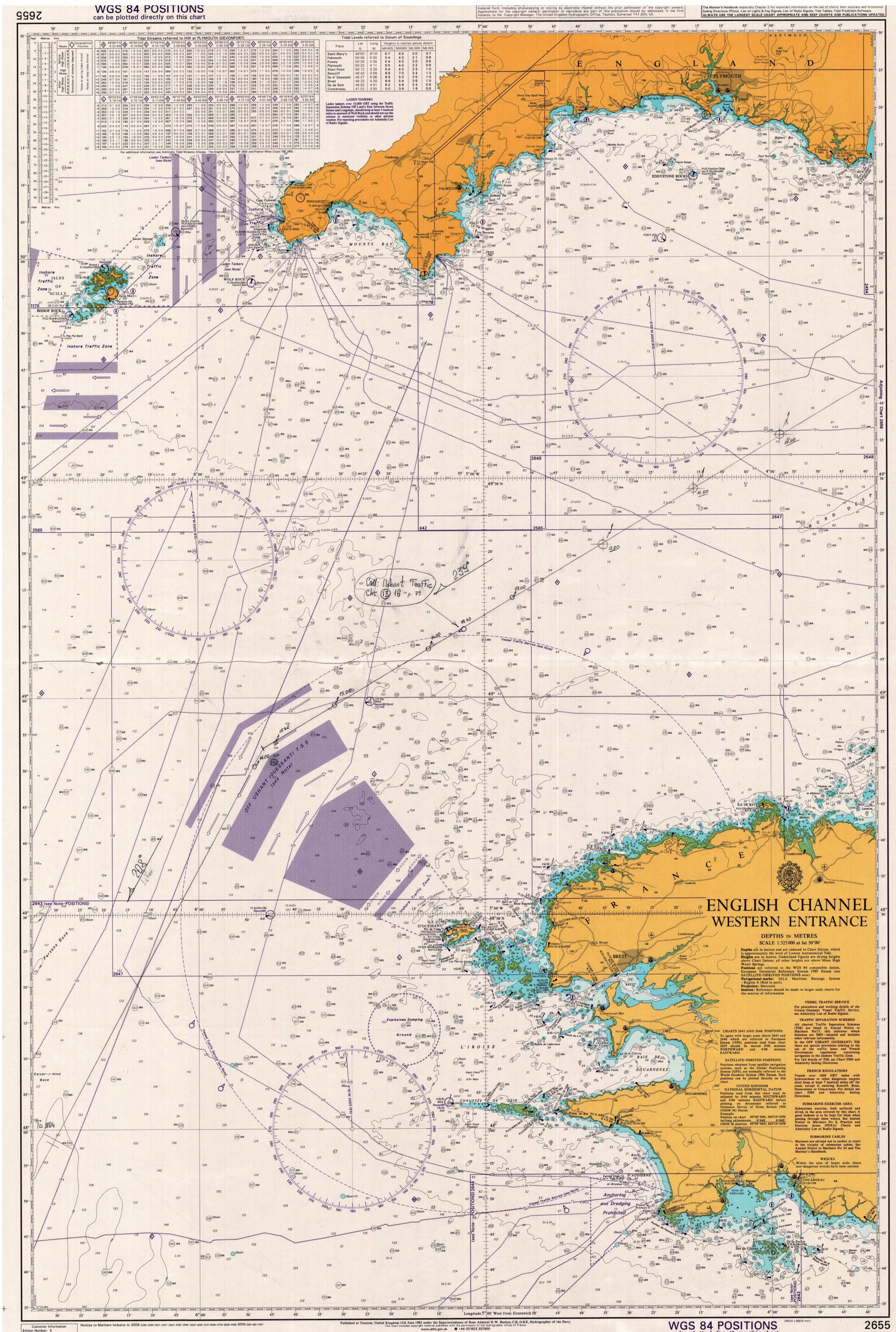

Figura 3.14: Carta náutica $n°2655$ (Fonte: Arquivo Escola Superior Náutica Infante D. Henrique).

Figura 3.15: Simplificação da Carta náutica $n°2655$.

Tabela 3.1: Pontos de linha de costa associados à Carta náutica $n°2655$.

Nome	Latitude	Longitude	Nome	Latitude	Longitude
Lervily	$48°00,0'N$	$004°34,0'W$	Pointe du Raz	$48°02,5'N$	$004°45,5'W$
	$48°04,0'N$	$004°43,0'W$		$48°06,0'N$	$004°28,0'W$
	$48°06,0'N$	$004°20,0'W$		$48°12,5'N$	$004°20,0'W$
Cap Chèvre	$48°10,0'N$	$004°33,0'W$		$48°12,0'N$	$004°33,0'W$
	$48°12,0'N$	$004°31,0'W$		$48°14,0'N$	$004°34,5'W$
	$48°14,0'N$	$004°29,5'W$		$48°16,0'N$	$004°37,0'W$
Camaret-Sur-Mer	$48°18,0'N$	$004°34,0'W$		$48°18,0'N$	$004°33,5'W$
	$48°20,0'N$	$004°33,5'W$		$48°20,0'N$	$004°32,0'W$
Le Conquete	$48°19,5'N$	$004°46,0'W$		$48°20,0'N$	$004°37,0'W$
	$48°21,5'N$	$004°32,0'W$		$48°22,0'N$	$004°47,5'W$
	$48°24,0'N$	$004°46,5'W$		$48°26,0'N$	$004°47,0'W$
	$48°28,0'N$	$004°45,5'W$	L'Aber Ildut	$48°30,0'N$	$004°46,5'W$
Argenton	$48°32,0'N$	$004°45,5'W$	Portsall	$48°34,0'N$	$004°42,5'W$
	$48°36,0'N$	$004°36,5'W$		$48°38,0'N$	$004°32,5'W$
Pontusval	$48°40,0'N$	$004°22,5'W$		$48°41,0'N$	$004°20,0'W$

- que o conjunto dos pontos definidos como pontos navegáveis nesta representação e as respetivas arestas (segmentos de paralelos e meridianos que não atravessam nenhum obstáculo), definam o grafo sobre o qual o algoritmo de Dijkstra será aplicado, de forma a ser possível obter uma rota ótima. A solução ótima obtida por este algoritmo depende, em grande parte, da resolução da grelha. Acreditamos que intervalos de $1'$ seja um bom equilíbrio entre a descrição do espaço, tendo uma relação direta com a precisão da solução, e o custo computacional (um aumento de pontos e arestas implicará um aumento do esforço computacional).

Uma grelha com intervalos de $1'$, tanto de latitude como de longitude, resulta em:

$$\begin{aligned} \Delta\lambda &= \lambda_{esq} - \lambda_{dir} = 006°10,0' - 004°20,0' = 1°50,00' = 110' \\ \Delta\varphi &= \varphi_{sup} - \varphi_{inf} = 49°20,0' - 48°00,0' = 1°20,0' = 80' \end{aligned} \tag{3.5}$$

A área em estudo está dividida em 110 meridianos e 80 paralelos, totalizando 8800 vértices resultantes da interseção dos meridianos com os paralelos.

Os pontos classificados como não navegáveis são todos aqueles que se encontram a Leste e Sul da linha de costa e os pertencentes aos arredores de *Île D'Ouessant*, *Î de Molene*, *La Parquette* e *Chaussée de Sein*. Os pontos pertencentes à área do Esquema de Separação de Tráfego poderão ser definidos com navegáveis ou não, dependendo se a navegação é feita de Norte para Sul ou de Sul para Norte.

O Esquema de Separação de Tráfego é constituído por três corredores de tráfego,

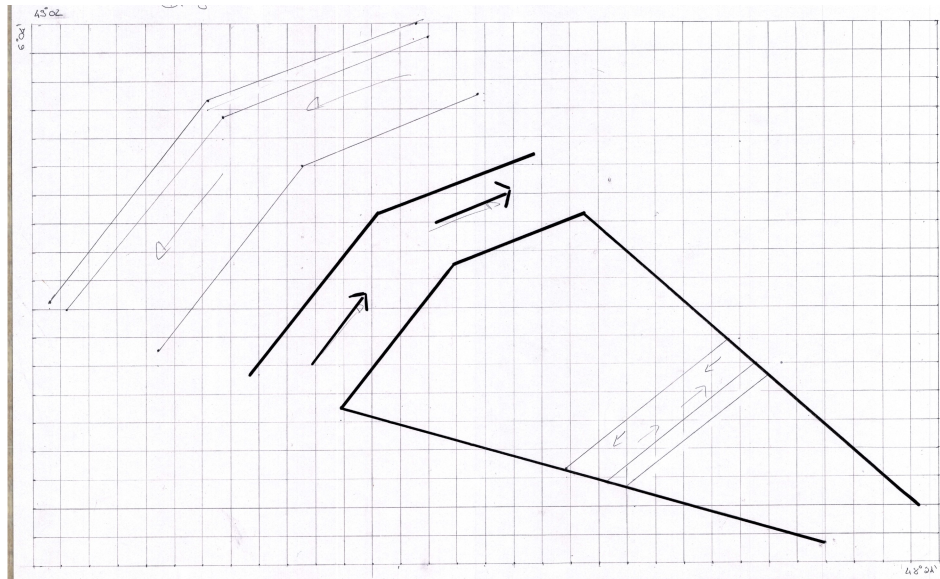

Figura 3.16: Esquema de Separação de Tráfego: Sentido Norte.

quatro corredores de separação do tráfego e uma zona de tráfego interno (do inglês *inshore traffic zone*). Esta zona só pode ser navegada por navios de comprimento inferior a 20m, veleiros e navios em faina de pesca.

Os três corredores de tráfego, de Leste para Oeste, são formados por [38]:

- um corredor de 2 milhas de largura, com navegação a decorrer nos dois sentidos, podendo ser usado por navios de cruzeiro e por navios com menos de 6000 de tonelagem bruta que naveguem de/para portos situados entre o Cap Finisterre e Cap de la Hague, desde que não transportem cargas perigosas;
- um corredor de tráfego para navios que entrem no canal da Mancha (sentido Norte);
- um corredor de tráfego para navios que saem do canal da Mancha (sentido Sul).

Brest é um porto de carga contentorizada, que se situa entre o Cap Finisterre e o Cap de La Hague. Assim, o tráfego de e para Brest navega, na sua grande maioria, no Esquema de Separação de Tráfego mais interior. No entanto, e tendo como objetivo uma aplicação mais generalizada do método proposto, optou-se por estudar e testar o comportamento do mesmo em todos os corredores do Esquema de Separação de Tráfego.

Assim, se o navio estiver a navegar de Norte para Sul usando o corredor mais externo, o corredor mais interno e toda a área a Leste desse corredor são definidos como não navegável (ver Figura 3.17). Se o navio estiver a navegar de Sul para Norte, a área não navegável é definida pela zona interior e pelo corredor externo (ver Figura 3.16). Se o navio estiver a navegar no corredor mais interno, independentemente de ser no sentido Norte ou Sul, a área não navegável é definida como a zona de tráfego interno e toda a

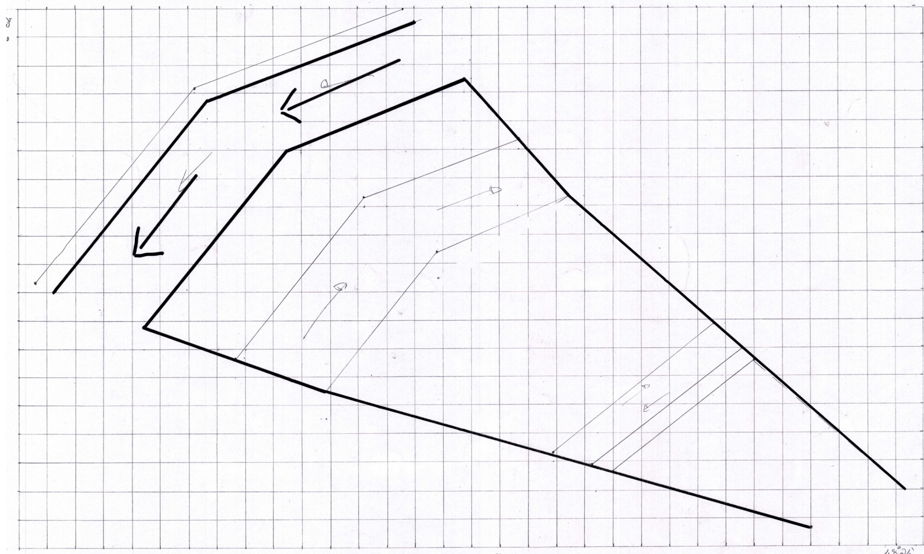

Figura 3.17: Esquema de Separação de Tráfego: Sentido Sul.

área a Oeste desse corredor.

Aplicação

Tendo como base a Figura 3.15, é possível proceder ao contorno dos obstáculos criando polígonos. Com esses polígonos, é criada uma matriz de pontos navegáveis e não navegáveis, onde cada ponto navegável é categorizado com 1 e cada ponto não navegável com 0.

Para fins meramente ilustrativos, consideremos a área de navegação, com definição de obstáculos, representada na Figura 3.18, onde os mesmos estão exagerados de forma a testar o algoritmo que passaremos a descrever. O ponto verde corresponde ao ponto de partida e o ponto azul ao ponto de chegada.

Todos os pontos da área de navegação que não estão incluídos em nenhum obstáculo criado constituem os vértices da rede onde o algoritmo de Dijkstra será aplicado. Este constitui o primeiro passo do algoritmo.

Durante a aplicação do algoritmo de Dijkstra, são analisados os sucessores do vértice que, em cada momento, é considerado em cada iteração.

Seja s o vértice corrente. Os sucessores de s são definidos de acordo com os vértices que são seus vizinhos mais próximos em direção de qualquer um dos pontos cardeais (norte, sul, este e oeste). Alternativamente, podem ser considerados também os vértices que estão nas direções dos pontos colaterais (noroeste, nordeste, sudeste e sudoeste). Portanto, cada vértice poderá ter até 4 ou 8 sucessores, a depender se também são considerados ou não os pontos colaterais, atendendo se os mesmos correspondem a pontos navegáveis ou não.

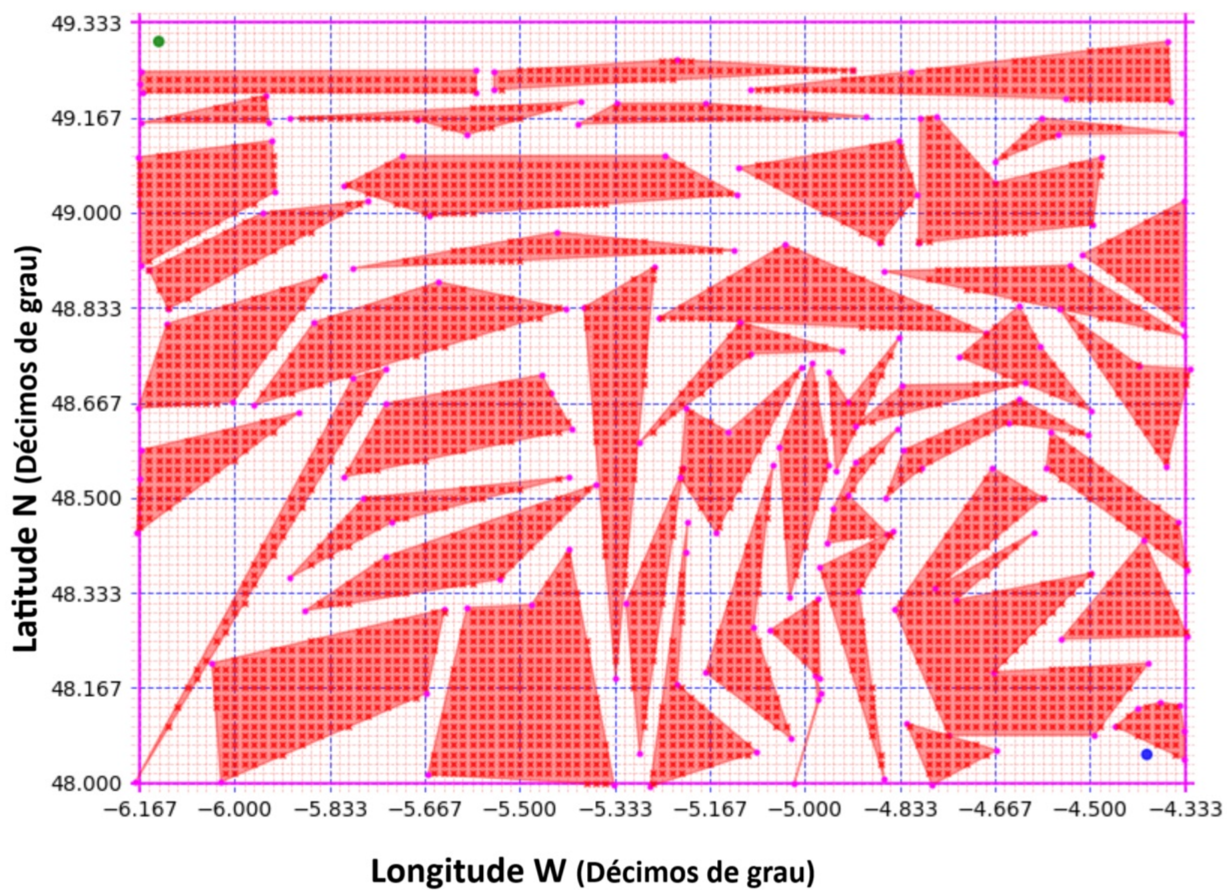

Figura 3.18: Ilustração de uma área de navegação, com obstáculos predefinidos.

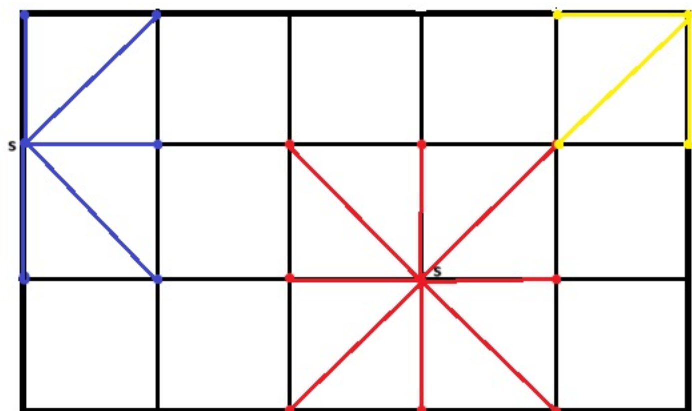

Figura 3.19: Vértices sucessores de s, de acordo se são seus vizinhos mais próximos em direção aos pontos cardeais e colaterais.

A Figura 3.19 ilustra esta situação.

Durante ainda o algoritmo de Dijkstra, é também verificado se o percurso entre dois vértices adjacentes pode ser realizado, isto porque, apesar dos mesmos poderem ser navegáveis, o percurso entre eles poderá intersetar um polígono definido como não navegável.

Na verdade, a primeira fase do método aqui proposto não é mais do que uma adaptação do algoritmo de Dijkstra. De seguida, descrevemos a primeira fase do método, em pseudo-código.

....

```
Fase 1 (adaptação do algoritmo de Dijkstra):

Parâmetros de entrada:
d(P,Q): função que determina a distância entre dois pontos P(x0,y0) e Q(x1,y1)
X: lista dos vértices navegáveis da rede
s: vértice inicial
t: vértice final

Inicialização:
p = s: vértice corrente
l(p) = 0
l(q) = +inf, para todo o q em X deferente de p
L = {p}: lista dos vértices já analisados

Procedimento principal:
enquanto p diferente de t:
    para todo o q sucessor de p não em L:
        se o percurso entre p e q intersetar algum polígono:
            ignorar q e passar ao próximo
        caso contrário:
            l(q) = min {l(q), l(p) + d(p,q)}
    seja r tal que l(r) = min {l(q) : q vértice de X não em L}
    p = r
    colocar p em L, isto é, L U {p}
Saída:
U: lista ordenada dos vértices que constituem o caminho mais curto entre s e t
```

....

A Figura 3.20 ilustra a solução ótima obtida no final da primeira fase do método proposto, aplicada à rede da Figura 3.19, com a opção de incluir ambos os pontos cardeais e os colaterais para o cálculo dos vizinhos de cada vértice.

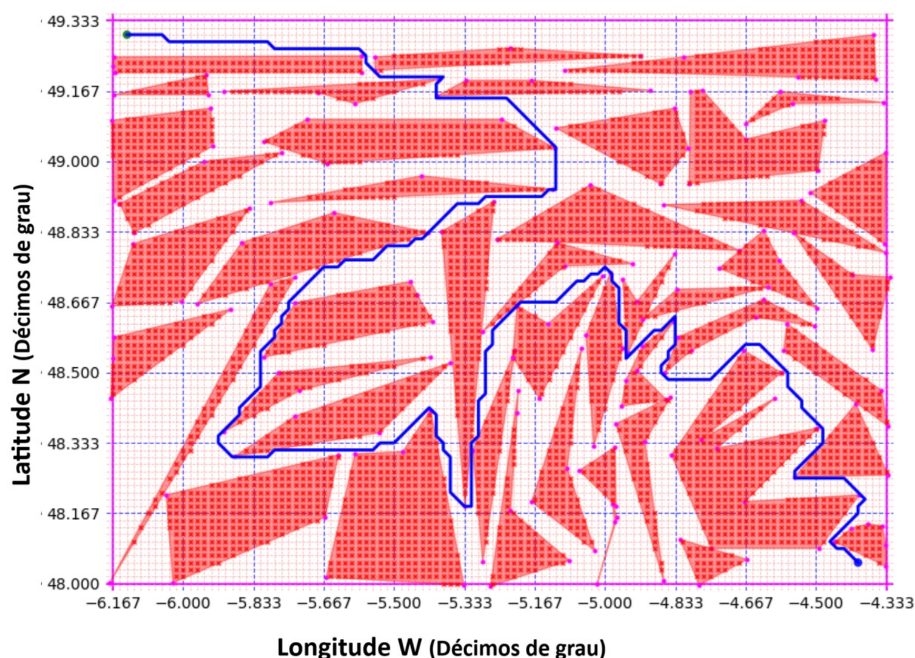

Figura 3.20: Rota ótima obtida no final da primeira fase do método proposto usando ambos os pontos cardeais e os colaterais.

Atendendo às distâncias reais consideradas, a derrota ótima obtida tem uma distância total de aproximadamente 341 milhas.

Para efeitos de comparação, a primeira fase do método foi também aplicada à mesma rede, com a opção de incluir apenas os pontos cardeais para o cálculo dos vizinhos dos vértices, esperando-se que a mesma não seja tão boa como a solução anterior em termos de distância total.

A Figura 3.21, ilustra a solução ótima obtida para esta segunda alternativa, onde a distância total obtida foi de 423 milhas.

Após a determinação do caminho mais curto entre o ponto de partida e o ponto de chegada, inicia-se uma segunda fase do método.

Esta fase subsequente tem por base a ideia subjacente ao método do polígono convexo, com o objetivo primordial de melhorar a solução obtida na primeira fase. Sobre esta segunda fase, foram elaboradas três variantes alternativas.

As Figuras 3.22) - 3.24 ilustram a primeira variante da segunda fase do método proposto. Sejam A, B e C três vértices consecutivos que fazem parte do caminho obtido na primeira fase.

Se A, B e C forem colineares, o B e o C avançam um vértice (Figura 3.22).

Caso contrário, averigua-se se o percurso AC interseta algum polígono. Se o percurso for livre, o mesmo é considerado em detrimento do percurso ABC. Isto é realizado fazendo B = C e avançando C para o próximo vértice.

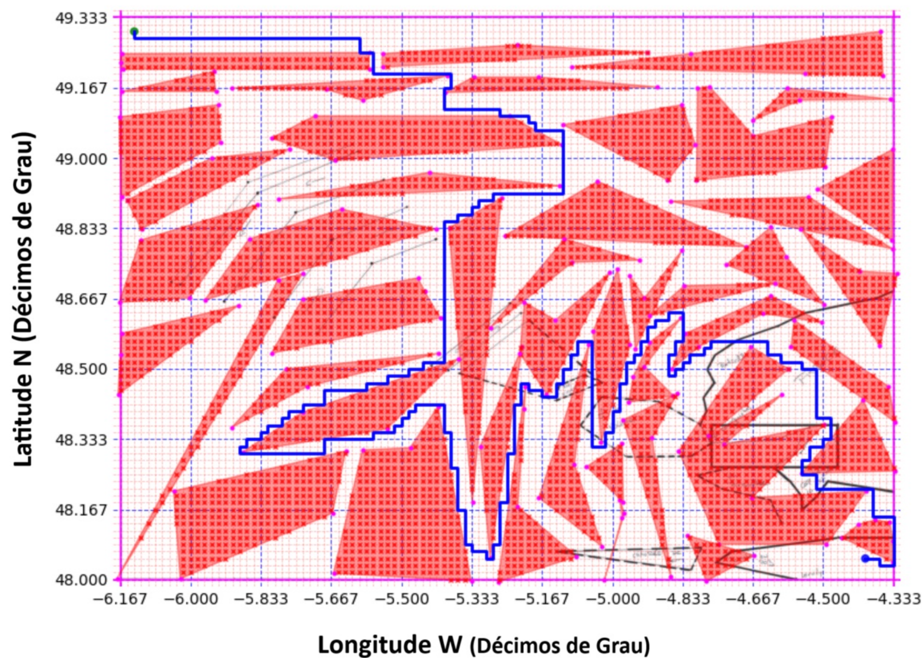

Figura 3.21: Rota ótima obtida no final da primeira fase do método proposto usando apenas os pontos cardeais.

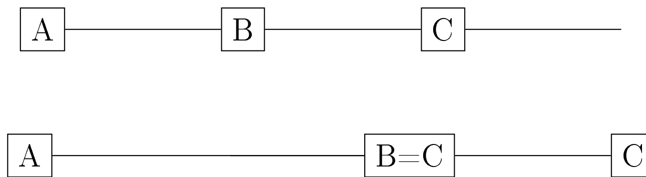

Figura 3.22: Segunda fase do método proposto: A, B e C colineares.

Pela desigualdade triangular, esta atualização garante que o caminho resultante tem distância total inferior à do caminho anterior (Figura 3.23).

Se o percurso AC não for livre de obstáculos (Figura 3.24), todos os três vértices avançam para os seus sucessores, isto é, A = B, B = C e C avança para o próximo vértice do caminho.

Quando implementado computacionalmente, o caminho obtido no final da segunda fase conterá exatamente o mesmo número de vértices que o caminho original. De seguida, apresenta-se a primeira variante da segunda fase do método em pseudo-código.

. . . .

Fase 2 (variante 1 - rubber band):

Parâmetros de entrada:

d(P,Q): função que determina a distância entre dois pontos P(x_0,y_0) e Q(x_1,y_1)

U: lista ordenada dos vértices que constituem o caminho obtido na primeira fase

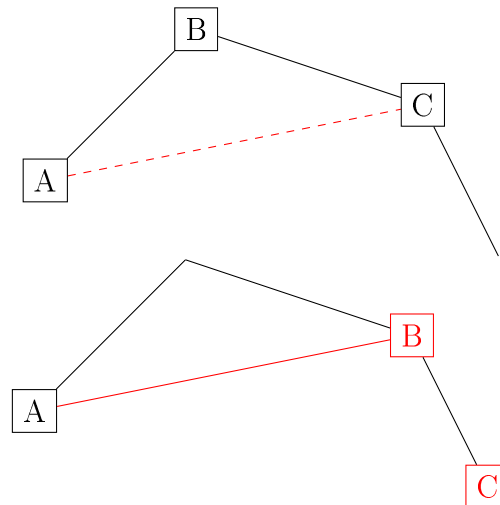

Figura 3.23: Segunda fase do método proposto: A, B e C não colineares, com AC livre de obstáculos.

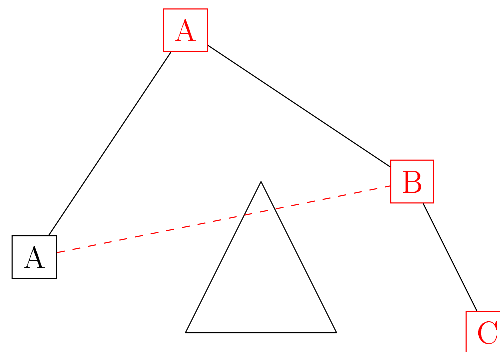

Figura 3.24: Segunda fase do método proposto: A, B e C não colineares, com o segmento entre A e o novo B a intersetar obstáculos.

```
N: número de vértices em U

Inicialização:
i=1, j=2, k=3: índices dos três primeiros vértices da lista U
A = U(i),
B = U(j): os dois primeiros vértices da lista U
pending=False: se existe alguma operação ainda pendente para ser realizada

Procedimento principal:
repetir:
    se k igual a N:
        se pending for True:
            ligar (A, B, i, j, U)
```

```
        STOP
    C = U(k)
    se A, B e C são não colineares:
        se AC não intersetar nenhum polígono:
            pending = True
        caso contrário:
            ligar (A, B, i, j, U)
            pending = False
            i = j
            A = B
    j = k
    k = k + 1
    B = C

Saída:
U: lista atualizada dos vértices que constituem o caminho obtido na segunda fase
Função ligar (A, B, i, j, U):
    ”’
    permite ligar A a B, reajustando as posições dos vértices entre A e B;
    A lista U é atualizada
    ”’
    n = j-1
    dx = (B(0) - A(0))/n
    dy = (B(1) - A(1))/n
    para k = 1 até n-1:
        Q: ponto de coordenadas (A(0) +k*dx, A(1) +k*dy)
        U(i+k) = Q
```

....

Relativamente à segunda variante da segunda fase do método proposto, grande parte do procedimento é semelhante ao da primeira variante, exceto na atualização do caminho quando o percurso AC não é livre de obstáculos.

Neste caso, quando os vértices B e C avançam para os seus sucessores, o vértice A é redefinido num ponto intermédio entre A e o novo B, em vez de avançar para o antigo B, como se ilustra na Figura 3.25. A divisão do segmento entre A e o novo B em segmentos mais pequenos permite encontrar caminhos com um maior grau tangencial aos obstáculos, potenciando caminhos finais mais curtos.

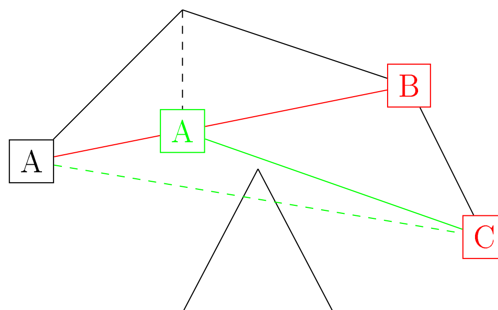

Figura 3.25: Segunda variante da segunda fase do método proposto: quando o segmento entre A e o novo C interseta obstáculos.

De seguida, apresenta-se o algoritmo da segunda variante da segunda fase do método proposto em pseudo-código.

. . . .

Fase 2 (variante 2):

Parâmetros de entrada:

d(P,Q): função que determina a distância entre dois pontos P(x_0,y_0) e Q(x_1,y_1)

U: lista ordenada dos vértices que constituem o caminho obtido na primeira fase

N: número de vértices em U

Inicialização:

i=1, j=2, k=3: índices dos três primeiros vértices da lista U

A = U(i),

B = U(j): os dois primeiros vértices da lista U

pending=False: se existe alguma operação ainda pendente para ser realizada

Procedimento principal:

```
repetir:
    se k igual a N:
        se pending for True:
            ligar (A, B, i, j, U)
        STOP
    C = U(k)
    se A, B e C são não colineares:
        se AC não intersetar nenhum polígono:
            pending = True
        caso contrário:
            ligar (A, B, i, j, U)
```

```
            pending = False
            repetir:
                i = i+1
                A = U(i)
                se i igual a j ou AC não interstar nenhum polígono:
                    sair do ciclo anterior
            se i < j:
                ligar (A, C, i, k, U)
    j = k
    k = k + 1
    B = C
```

Saída:
U: lista atualizada dos vértices que constituem o caminho obtido na segunda fase

....

A terceira variante da segunda fase do método proposto não é mais do que a composição das duas primeiras variantes, uma após a outra, considerando-se o caminho obtido pela segunda variante como ponto de partida para a aplicação da primeira variante.

Na Figura 3.26 podemos visualizar as melhorias sucessivas de cada variante da segunda fase do método proposto.

A linha vermelha corresponde ao caminho obtido pela primeira variante com uma distância total de 324 milhas naúticas.

A linha verde, apesar de muito ténue e obscurecida por outras linhas, corresponde ao caminho obtido pela segunda variante, com uma distância total de 319.40 milhas naúticas. Esta variante corresponde àquela onde a melhoria foi a mais acentuada relativamente ao caminho usado como ponto de partida.

Finalmente, a linha amarela resulta da aplicação da terceira variante, correspondendo ao caminho com a distância mais curta obtida, ou seja, de 319.36 milhas naúticas, apesar de ser uma diferença marginal comparativamente ao da segunda variante.

3.6 Dados AIS

A área primeiramente selecionada para o desenvolvimento do projeto foi a entrada do porto de Lisboa e o Esquema de Separação de Tráfego do Cabo da Roca. Esta área tem uma grande densidade de tráfego pois toda a navegação entre o norte da Europa e o Mar Mediterrâneo ou a costa Africana passam pela costa portuguesa. O porto de Lisboa é

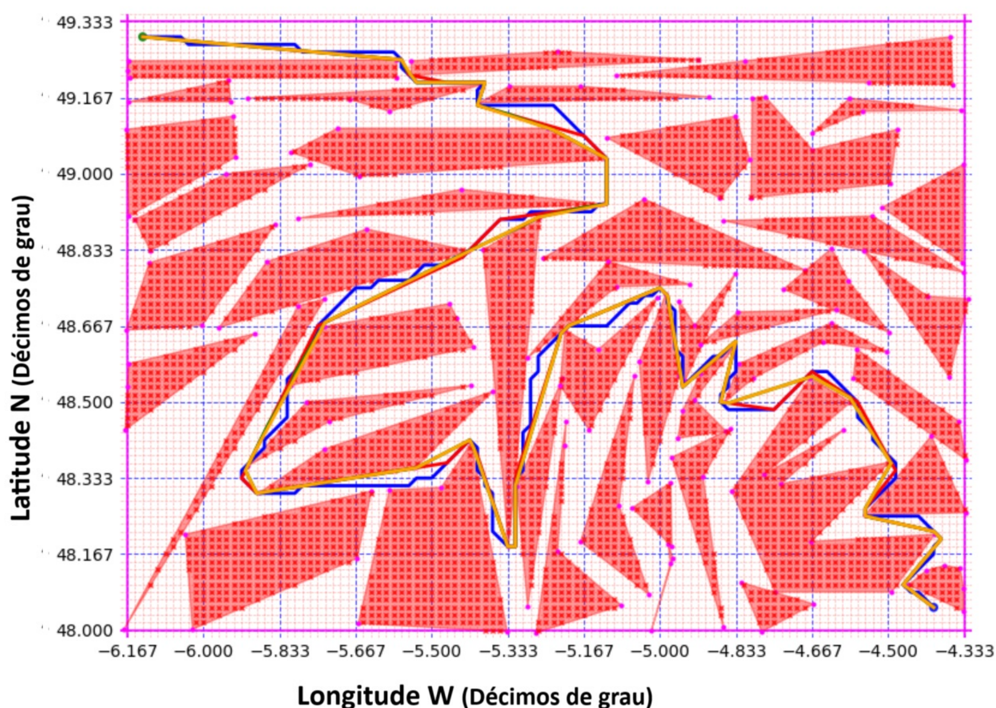

Figura 3.26: Comparação entre os vários caminhos obtidos pelas três variantes da segunda fase do método proposto, assim como o caminho obtido na primeira fase.

um porto com algum movimento condicionado devido à grande quantidade de baixios, proporcionando um conjunto de obstáculos naturais que o algoritmo teria de contornar.

Além disso, foi um porto amplamente estudado durante a minha formação/trabalho inicial, pelo que já existia algum conhecimento prévio sobre a navegação e os perigos desta área.

No entanto, não foi possível obter dados de navegação com os quais se pudesse tirar alguma conclusão sobre a viabilidade deste estudo. Contactou-se a EMSA (*European Maritime Safety Agency*), a DGRM (*Direção Geral dos Recursos Marítimos*) e o VTS (*Vessel Traffic Services*) sem sucesso; a *Spire Maritime* e a *Marine Traffic* (Bases de dados AIS privados) também sem sucesso de forma gratuita.

Considerou-se o uso de dados obtidos nas próprias companhias de navegação, através dos diários de bordo onde se registam as posições em intervalos mínimos de quatro horas. Contactou-se a *Empresa de Navegação Madeirense*, a *Transinsular*, a *Mutualista*, a *Portline* e a Escola Naval[6] todas sem sucesso.

Por fim optou-se por procurar na base de dados aberta da Google[7] um conjunto de dados com os quais pudéssemos trabalhar. Não tendo sido encontrados nenhum conjunto

[6] A tese [12] foi elaborada com recurso a dados de corvetas.
[7] https://datasetsearch. research.google.com/

de dados que pertencessem à costa portuguesa, selecionou-se a área descrita nos capítulos anteriores do Noroeste de França.

Primeiramente, procedeu-se à seleção dos dados e posteriormente ao tratamento dos mesmos.

Seleção:

O conjunto de dados escolhido (que contivesse a área em estudo) já estava separado em 4 ficheiros agrupando os diferentes dados pelo tipo de mensagem obtido pelo AIS.

O formato destas mensagens é especificado pela ITU (*International Telecomunications Union*). Existem 27 tipos de mensagens de AIS [41], no entanto, na Tabela 3.2, apenas listaremos as que constam no conjunto de dados disponível.

Tabela 3.2: Tipos de mensagens AIS presentes nos dados recolhidos.

Código	Significado	Ficheiro
ITU 1	mensagens automáticas em intervalos de acordo com a sua velocidade (Classe A)	*nari_dinamic*
ITU 2	mensagens em intervalos designados pela autoridade competente (Classe A)	*nari_dinamic*
ITU 3	mensagens em resposta por um pedido especifico da entidade competente (Classe A)	*nari_dinamic*
ITU 5	mensagens de navios estáticos (Classe A)	*nari_ais_static*
ITU 9	mensagens de socorro	*nari_dinamic_sar*
ITU 18	mensagens automáticas em intervalos de acordo com a sua velocidade (Classe B)	*nari_dinamic*
ITU 19	mensagens em intervalos designados pela autoridade competente(Classe B)	*nari_dinamic* e *nari_ais_static*
ITU 21	mensagens ajudas à navegação	*nari_dinamic_aton*
ITU 24	mensagens de navios estáticos (Classe B)	*nari_ais_static*

Os tipos de mensagens com relevância para o presente estudo são os do tipo ITU 1, ITU 2 e ITU 3; sendo então selecionado o ficheiro *nari_dinamic* com os parâmetros descritos na Tabela 3.3.

Cada linha de dados contém nove atributos, descritos na Tabela 3.4.

Procedeu-se então à filtração destes dados seguindo os passos:

Passo 1: (rateofturn)

A coluna rateofturn foi eliminada por se considerar a informação nela contida de baixa relevância e redundante para o estudo atual.

Tabela 3.3: Parâmetros selecionados (Fonte: *AIS_Data*).

Parâmetro	Detalhe
Objeto:	AIS Dynamic Data (version 2)
Nome:	*nari_dynamic.csv*
Fonte:	Naval Academy, France (ecole-navale.fr)
Versão:	v2 (31/05/2017)
SRID:	WGS84
Cobertura:	Longitude entre −10.00 e 0.00 e Latitude entre 45.00 e 51.00
Volume:	18 648 556 mensagens
Período:	6 meses (01/10/2015 00:00:00 UTC a 31/03/2016 23:59:59 UTC)
Licença:	CC-BY-NC-SA-4.0

Tabela 3.4: Atributos dos dados (Fonte: *AIS_Data*)

Atributo	Descrição
MMSI:	identificador do navio
status:	estado da navegação
rateofturn:	guinada
speed:	velocidade
course:	rumo
heading:	proa
lon:	longitude
lat:	latitude
t:	tempo em UNIX epochs

Passo 2: (lat e lon)

A área em estudo está compreendida entre:

$$48^\circ\ 00,0'\ N \leq \varphi \leq 49^\circ\ 20,0'\ N; 004^\circ\ 20,0'\ W \leq \lambda \leq 006^\circ\ 10,0'\ W$$

procedendo-se então à seleção dos dados contidos nesta área.

Passo 3:(MMSI)

Seguidamente selecionou-se os dados removendo MMSI (*Maritime Mobile Service Identity*) anómalos.

O MMSI deve ser um número de 9 dígitos entre 201.000.000 e 775.999.999 em que os 3 primeiros números representam o código do país. Contudo, por vezes, o MMSI não satisfaz as especificações exigidas pela ITU ou IMO; seja por falhas do emissor, por erros do operador ao inserir a informação ou porque a autoridade regional não adere completamente à regulamentação. Nestes casos, existe um legado de 6 dígitos regionais e

deve-se adicionar 3 zeros antes do número) [41].

Procedeu-se então à eliminação de todos os dados associados a um $MMSI < 100.000.000$ e $MMSI > 775.999.999$.

Passo 4:(status)

Por fim, procedeu-se ao estudo da coluna status. O estado da navegação é um código numérico definido pela ITU, que tem como objetivo fornecer a informação respetiva. A Tabela 3.5 descreve esta variável [18].

Tabela 3.5: Estados de navegação [18].

Código	Estado
0	navegação a motor
1	ancorado
2	desgovernado
3	manobra restrita
4	manobra condicionada pelo calado
5	atracado a bóias
6	encalhado
7	em faina de pesca
8	à vela
9 e 10	reservado para cargas perigosas, substâncias nocivas, poluentes marinhos
11 e 12	a reboque
14	SART, EPIRB

Para efeitos deste estudo, apenas se considerou os navios em navegação usando o motor (0), procedendo-se à eliminação dos dados associados aos outros estados de navegação e, posteriormente, à eliminação desta coluna (após a seleção, a variável torna apenas um único valor).

Tratamento:

Na fase seguinte, os dados foram trabalhados de forma a conseguirmos obter informações relativas ao percurso executado pelos navios selecionados resultantes da fase anterior.

Primeiramente, agrupou-se os dados por navio. Cada navio tem um MMSI único.

Seguidamente, os dados de cada navio foram agrupados por viagem. O tempo em Unixtimea[8] foi convertido em UTC[9]. Como uma viagem dentro da área considerada não ultrapassaria as 24 horas de navegação, os pontos foram agrupados considerando este intervalo de tempo, isto é, se entre dois pontos tivessem passados mais de 24 horas, os

[8]Quantidade de segundos que passaram desde 00:00:00 de 1 de Janeiro de 1970.

[9]*Coordinated Universal Time.*

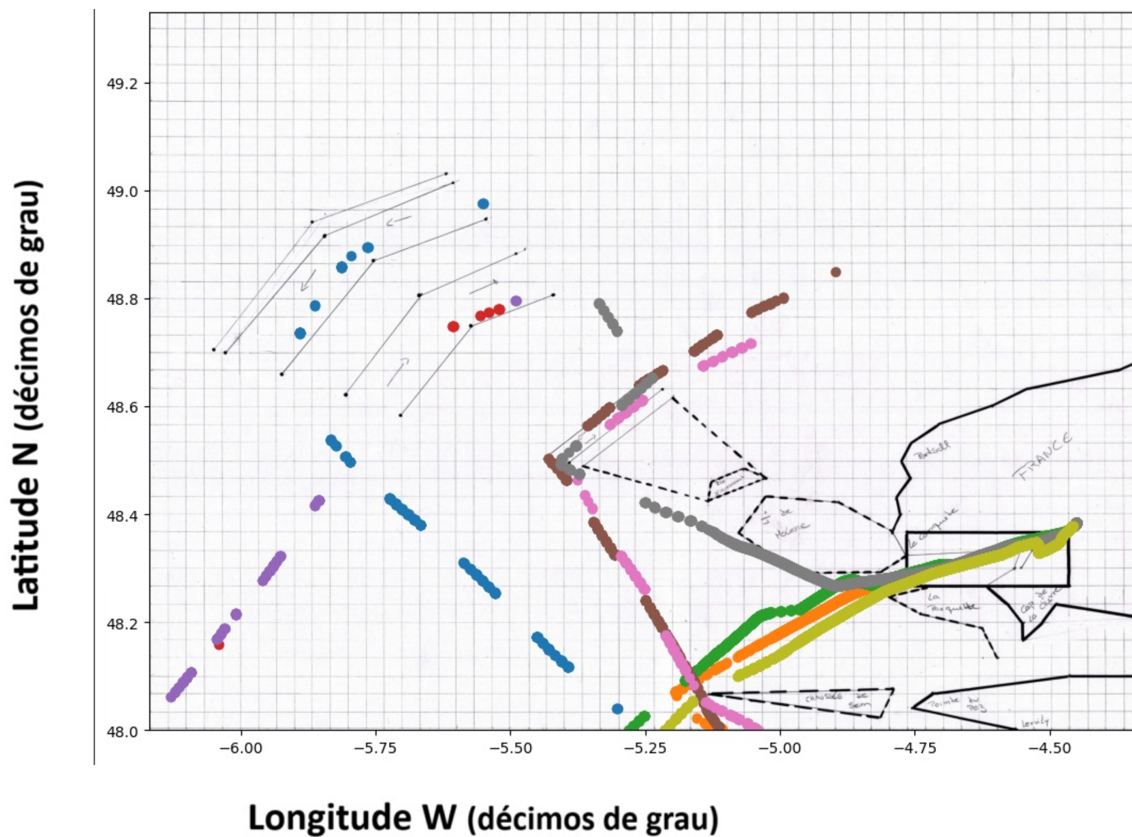

Figura 3.27: Dados de AIS referentes ao navio *Patrick*.

pontos pertenceriam a viagens diferentes. Assim, torna-se mais fácil visualizar as viagens respetivas.

Tenhamos como exemplo a Figura 3.27, correspondente às viagens do navio *Patrick*, de carga geral. Cada conjunto de posições da mesma cor corresponde a uma viagem feita por este navio.

4 Experiências computacionais

Como já foi referido anteriormente, a viagem na zona selecionada pode ser feita usando diferentes corredores do esquema de separação de tráfego, sendo que a escolha depende do tipo de navio, do tipo de carga que o navio transporta (perigosa ou não) e dos portos de partida e/ou destino. Assim, as experiências computacionais aqui apresentadas consistem na comparação de resultados caso a caso, isto é, corredor a corredor, com vista à validação do método proposto neste trabalho. Para que fosse possível realizar estas experiências, foi desenvolvido uma pequena aplicação usando a linguagem de programação Python, versão 3.12, estando o script completo em anexo. Em todas as derrotas obtidas pela aplicação os pontos colaterias foram considerados aquando da determinação de nodos vizinhos.

4.1 Esquema de separação de tráfego exterior sul

Para começar, consideremos as viagens realizadas por 5 navios diferentes, obtidas a partir dos dados AIS, ao longo do corredor externo no sentido sul, ilustradas na Figura 4.1. Não existem dados suficientes para aferir sobre os pontos de partida e chegada das respetivas viagens, mas é possível saber qual o tipo de navio de acordo com o respetivo MMSI. Assim, dois deles são navios químicos petroleiros, de transporte de carga perigosa, e os outros três navios de carga geral, que podem transportar ou não carga perigosa.

Usando a aplicação, uma viagem semelhante seria, por exemplo, a ilustrada na Figura 4.2, onde estão representadas as derrotas obtidas durante as duas fases do método proposto: a azul, a derrota obtida na primeira fase após a aplicação do algoritmo Dijkstra e, a amarelo, a derrota final melhorada com a segunda fase, variante 3. Apesar de terem sido também determinadas as putras duas derrotas com as variantes 1 e 2 da segunda fase, as mesmas não são bem visíveis na figura. As distâncias das derrotas visíveis correspondentes foram de, respetivamente, 104 e 98 milhas náuticas, observando-se uma melhoria de 6% da derrota final quando comparada com a da primeira fase.

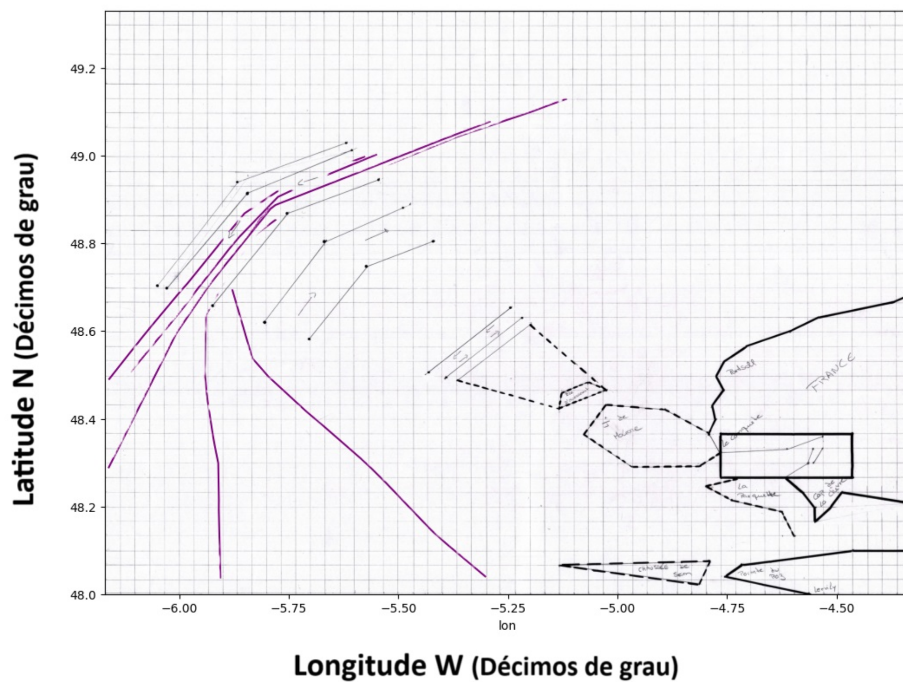

Figura 4.1: Esquema de separação de tráfego exterior sul: Dados de AIS referentes a viagens de 5 navios diferentes.

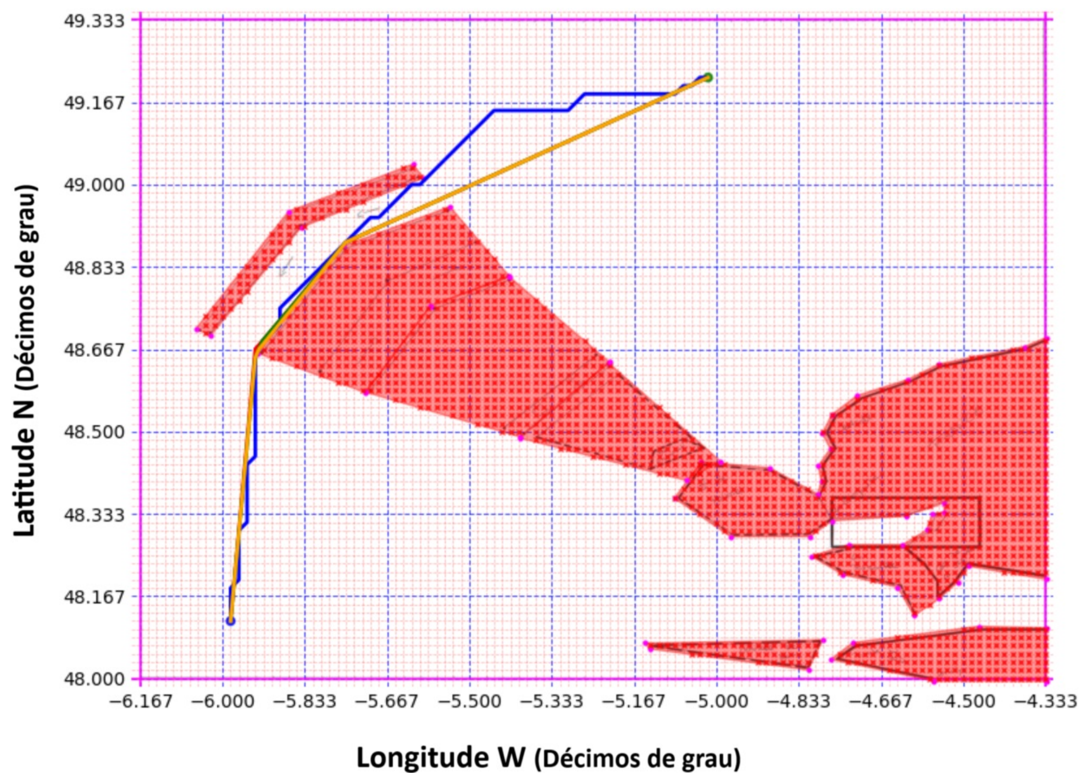

Figura 4.2: Esquema de separação de tráfego sul: Derrotas obtidas pelo método proposto.

4.2 Esquema de separação de tráfego exterior sul, com entrada no porto

Consideremos, agora, as viagens de 3 navios diferentes, obtidas a partir dos dados AIS, ao longo do corredor externo no sentido sul com entrada num porto nas proximidades de Brest ilustradas na Figura 4.3. De igual modo, os dados são omissos quanto às localizações dos pontos de partida e de chegada, apesar do ponto de chegada acontecer numa vizinhança de Brest. Um dos navios pertence à frota militar alemã, não se sabendo concretamente o objetivo da sua viagem, o outro é de carga geral, suspeitando-se transportar carga perigosa atendendo ao corredor usado pelo navio, e o terceiro é um navio químico petroleiro, de transporte de carga perigosa.

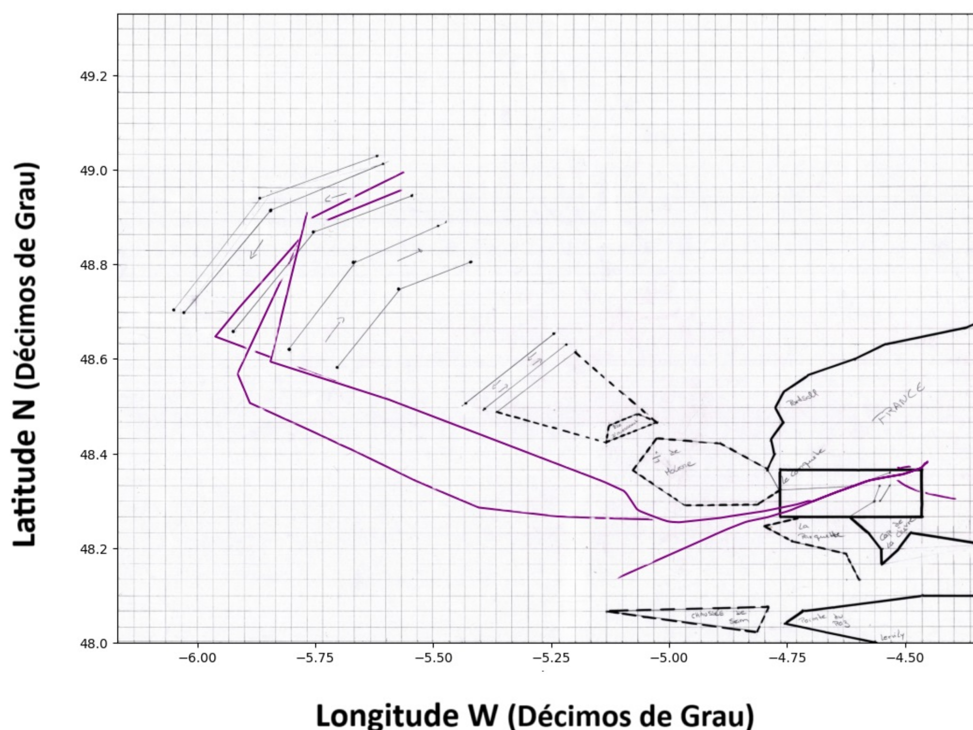

Figura 4.3: Esquema de separação de tráfego exterior sul, com entrada no porto: Dados de AIS referentes a viagens de 3 navios diferentes.

Para fazermos um elo de comparação com os dados de AIS usando a aplicação, podemos considerar uma viagem semelhante à da ilustrada na Figura 4.4, onde são bem visíveis, a azul, a derrota obtida na primeira fase do método proposto e, a amarelo, a derrota final obtida na segunda fase, variante 3. Menos visíveis são as duas derrotas, a vermelho e a verde, obtidas na segunda fase, variantes 1 e 2, respetivamente. Os comprimentos das derrotas mais visíveis são de 140 e 132 milhas náuticas, respetivamente, havendo uma

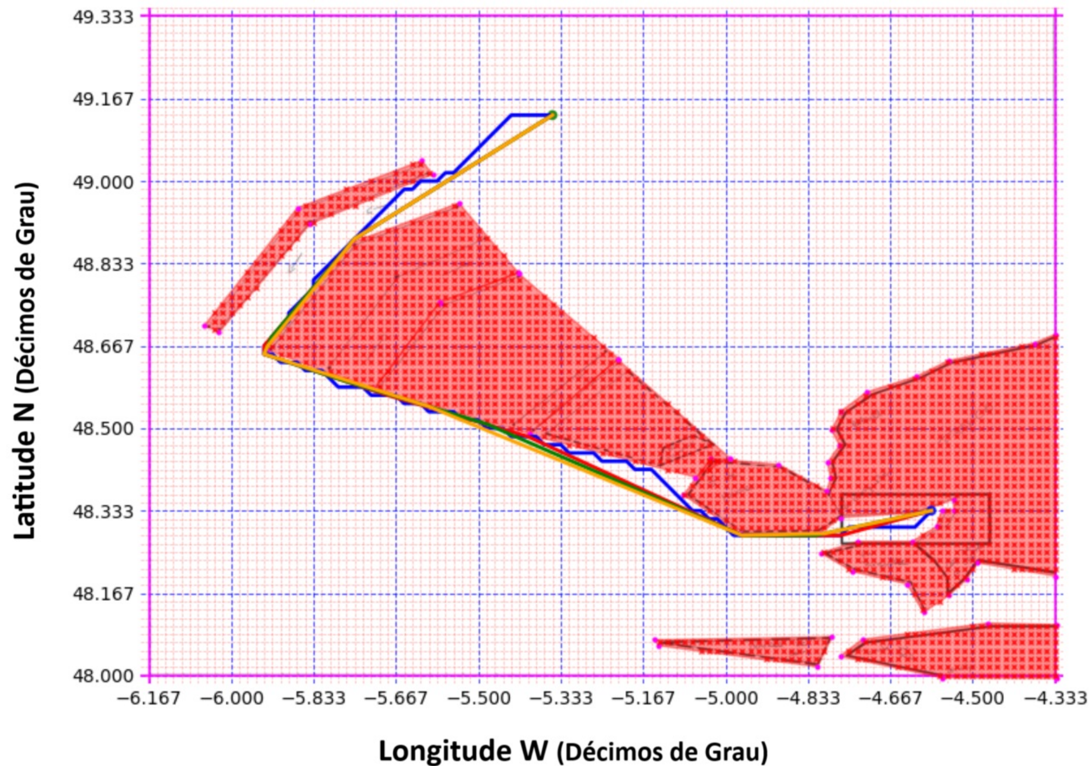

Figura 4.4: Esquema de separação de tráfego exterior sul, com entrada no porto: Derrotas obtidas pelo método proposto.

melhoria de 6% entre a derrota obtida na primeira fase e a derrota melhorada na segunda fase, variante 3.

4.3 Esquema de separação de tráfego exterior norte

Tenhamos agora, como exemplo, 9 viagens de 8 navios diferentes, obtidas a partir dos dados AIS, ao longo do corredor externo no sentido norte, como se ilustra na Figura 4.5. As localizações dos pontos de partida e de chegada não são conhecidas. Três dos navios são quimicos petroleiros, de transporte de carga perigosa, quatro são navios de carga geral, que podem transportar ou não carga perigosa, e dois são navios de carga contentorizada, também podendo transportar ou não carga perigosa.

Uma viagem semelhante seria, por exemplo, a da representada na Figura 4.6, obtida por aplicação do método proposto. As derrotas apresentadas são resultado da aplicação da primeira fase (linha azul) e da segunda fase, variantes 1-3 (linhas vermelha, verde e amarela, respetivamente).

O comprimento da derrota obtida na primeira fase é de 118 milhas náuticas, enquanto

Figura 4.5: Esquema de separação de tráfego exterior norte: Dados de AIS relativos a 9 viagens de 8 navios diferentes.

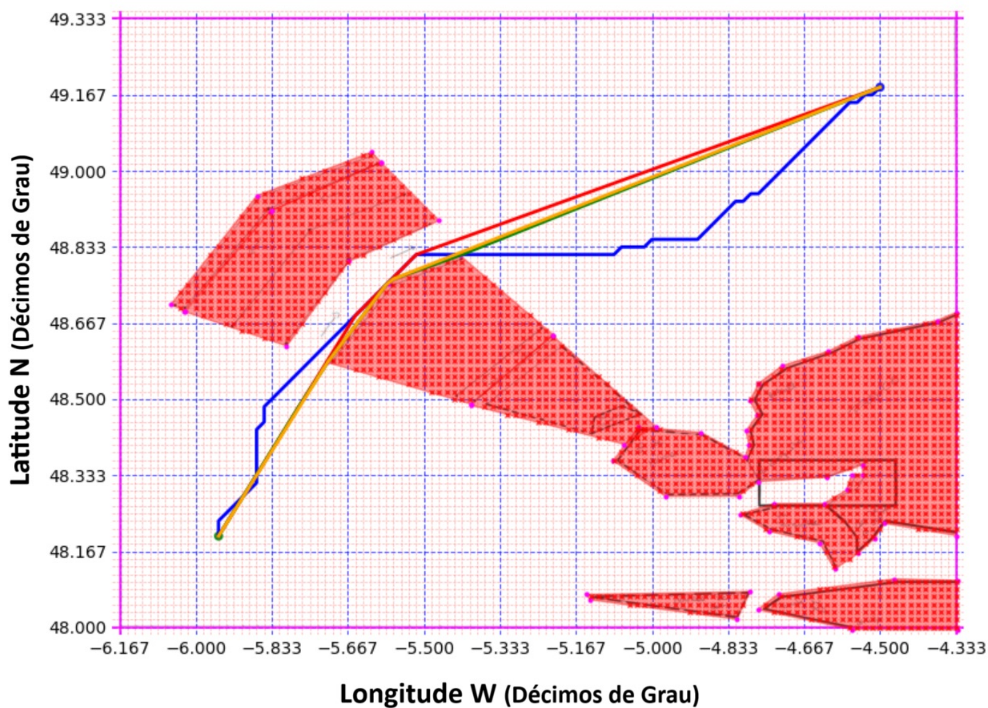

Figura 4.6: Esquema de separaçao de tráfego exterior norte: Derrotas obtidas pelo método proposto.

que o da derrota melhorada obtida na segunda fase, variante 3, é de 110 milhas náuticas. Neste caso houve uma melhoria na ordem dos 7%.

4.4 Esquema de separação de tráfego exterior norte, com saída do porto

Para ilustração de viagens ao longo do corredor externo no sentido norte, com saída das proximidades de Brest, consideremos as viagens de 5 navios diferentes, obtidas a partir dos dados AIS (ver Figura 4.7). Analogamente aos casos anteriores, os dados disponíveis não contêm informação suficiente sobre as localizações dos pontos de partida e de chegada, apesar da saída acontecer nas proximidades de Brest. Um dos navios é de carga geral, um é de carga contentorizada, sendo muito provável transportar carga perigosa atendendo ao corredor de tráfego por onde navega, e outros três são químicos petroleiros, de transporte de carga perigosa.

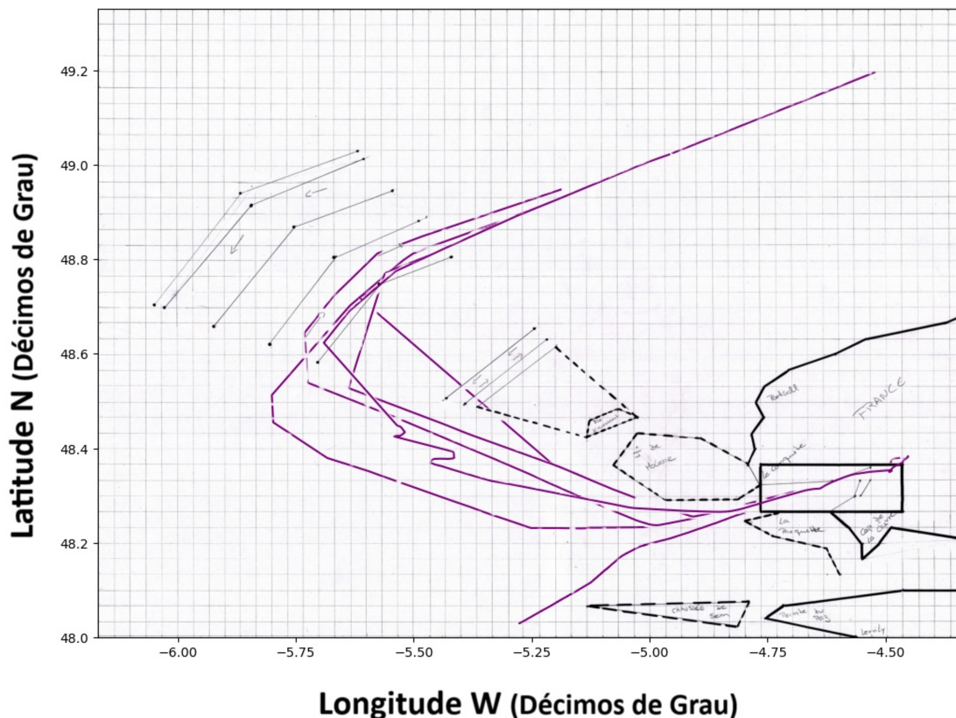

Figura 4.7: Esquema de separação de tráfego exterior norte, com saída do porto: Dados de AIS relativos a viagens de 5 navios diferentes.

Usando a aplicação, poderemos definir uma viagem semelhante, por exemplo, à da representada na Figura 4.8. As derrotas apresentadas foram obtidas utilizando o método

proposto: a azul, por aplicação da primeira fase e, a vermelho , a verde e a amarelo, por aplicação da segunda fase, variantes 1-3, respetivamente. As distâncias das derrotas primeira e última são de 164 e 153 milhas náuticas, respetivamente, obtendo-se uma melhoria, entre as duas, de aproximadamente 7%.

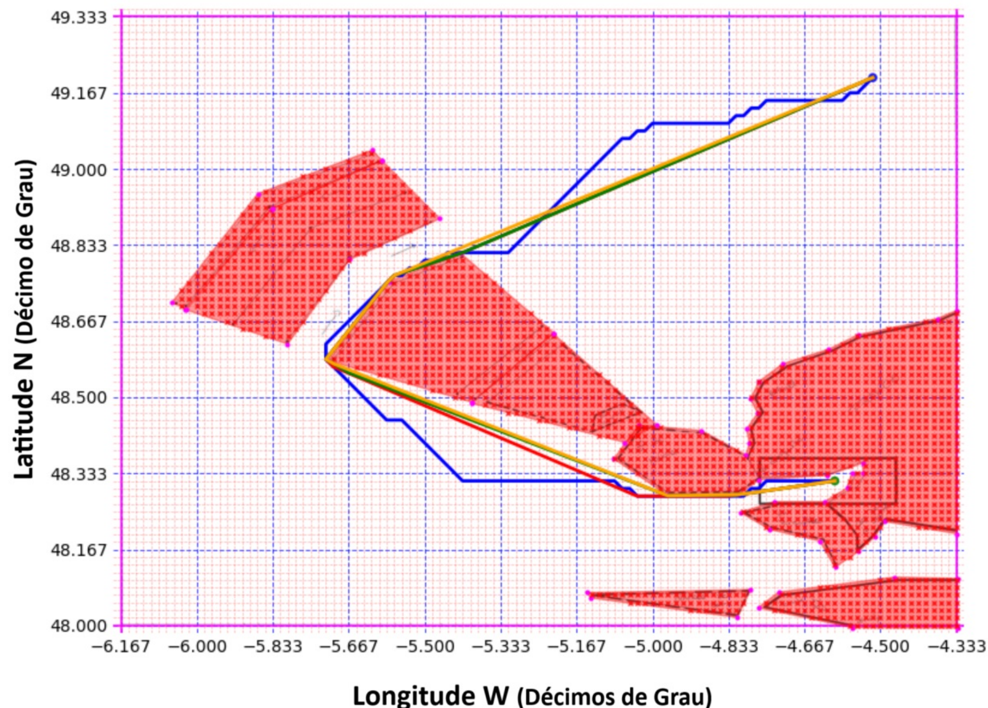

Figura 4.8: Esquema de separação de tráfego exterior norte, com saída do porto: Derrotas obtidas pelo método proposto.

4.5 Esquema de separação de tráfego interior

Como já foi referido no capítulo anterior, o esquema de separação de tráfego interior é usado em ambos os sentidos. Os navios que podem navegar ao longo deste corredor são navios que transportam carga não perigosa, navegando entre o Cabo Finisterre e o Cabo de La Hague, e navios de cruzeiro.

A Figura 4.9 ilustra uma sobreposição de 70 viagens realizadas por 16 navios diferentes que utilizam o corredor interno, sem entrar nem sair das proximidades de Brest. Destes, oito são de cruzeiro, cinco são de carga geral, um de carga contentorizada, um de carga extremamente pesada e um veleiro navio-escola. Em nenhum destes casos são conhecidas as localizações exatas de partida e de chegada.

A Figura 4.10 mostra uma sobreposição de quinze viagens realizadas por 11 navios

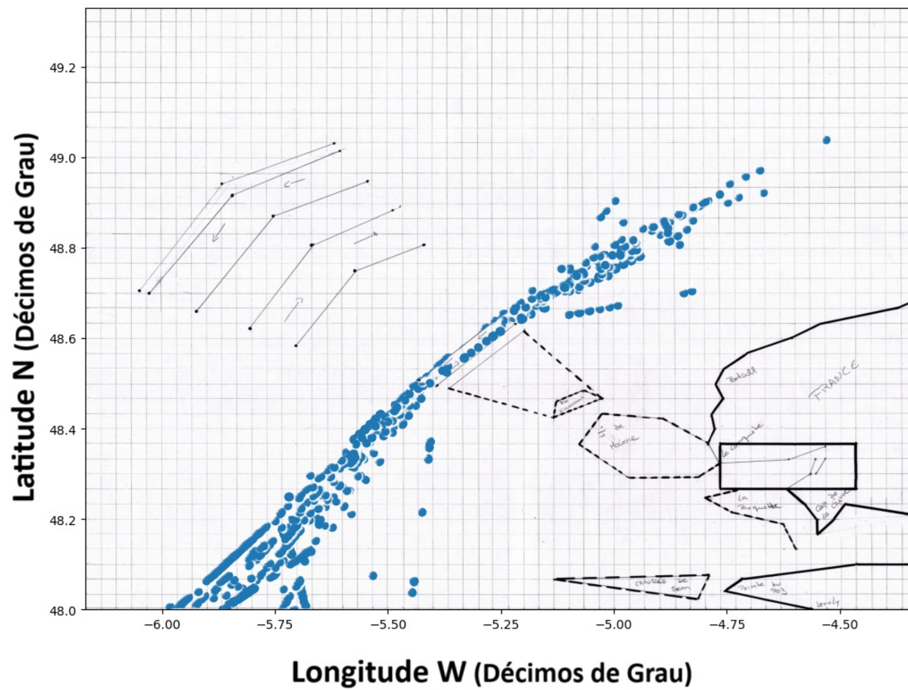

Figura 4.9: Esquema de separação de tráfego interior: Sobreposição da dados AIS relativos a 70 viagens de 16 navios diferentes.

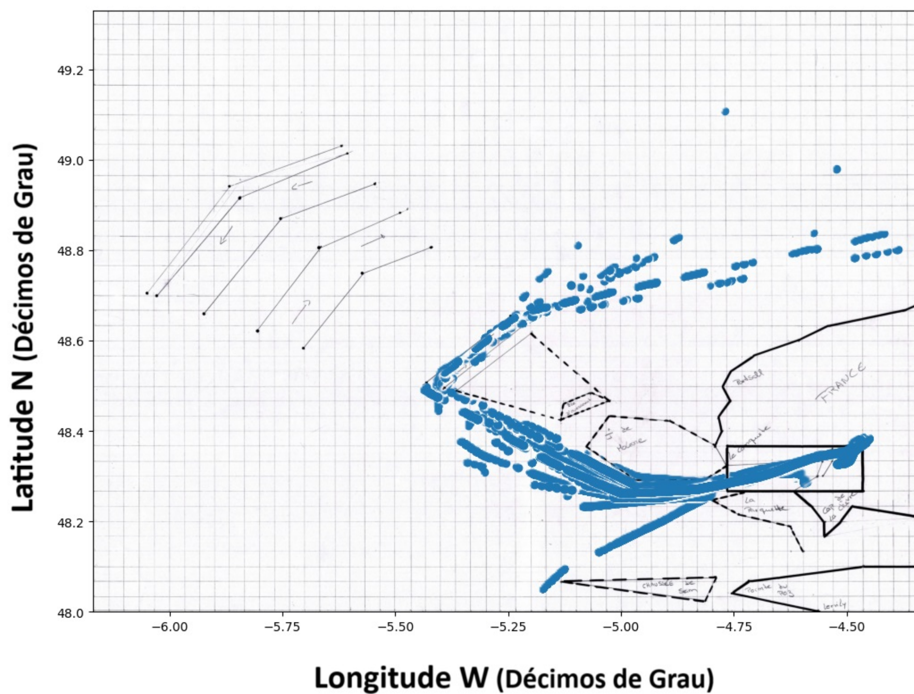

Figura 4.10: Esquema separação de tráfego interior, com entrada ou saída do porto: Sobreposição de dados AIS relativos a 15 viagens de 11 navios diferentes.

diferentes, havendo entradas e saídas das proximidades de Brest. Destes, um navio é um petroleiro, muito provávelmente vazio, oito são de carga geral, um de carga extremamente pesada e um veleiro navio-escola.

Usando uma abordagem análoga às dos outros esquemas de separação de tráfego, foram obtidas derrotas para ambos estes dois casos, utilizando o método proposto. As Figuras 4.11 e 4.12 ilustram os resultados obtidos, correspondentemente.

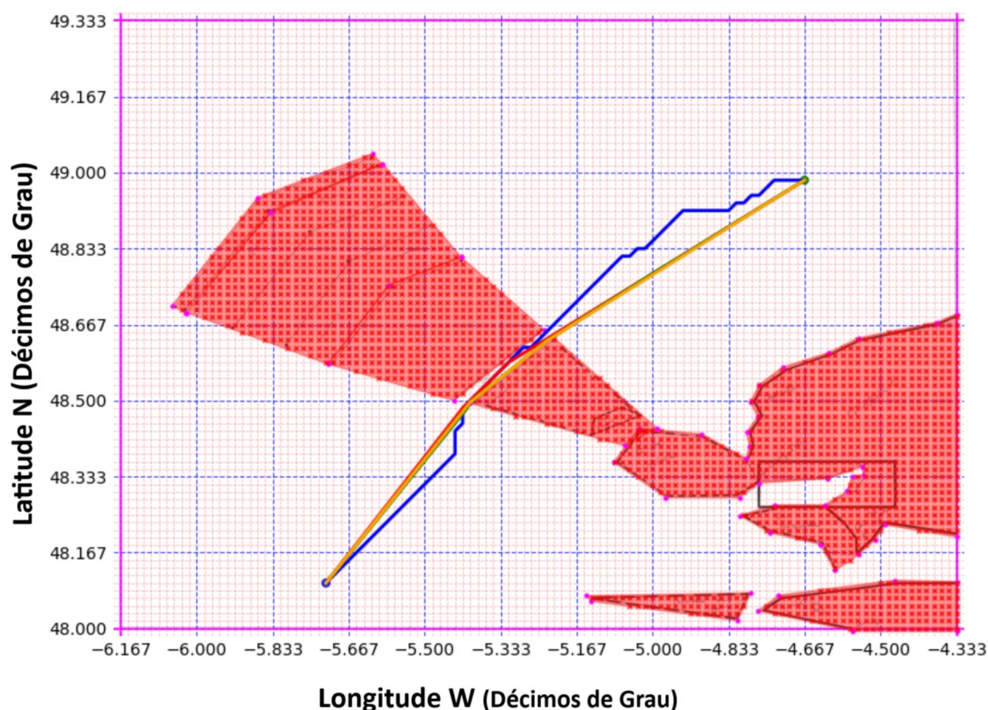

Figura 4.11: Esquema de separação de tráfego interior, de norte para sul: Derrotas obtidas pelo método proposto.

No primeiro caso, a distância total da derrota obtida na primeira fase do método é 88 milhas náuticas, enquanto que a da derrota melhorada, obtida na segunda fase, variante 3 é de 83 milhas náuticas, obtendo-se uma melhoria de aproximadamente 6%.

Já no segundo caso, a distância da derrota obtida na primeira fase é de 110 milhas náuticas, sendo a da derrota melhorada de 103 milhas náuticas, com uma melhoria de aproximadamente 6%.

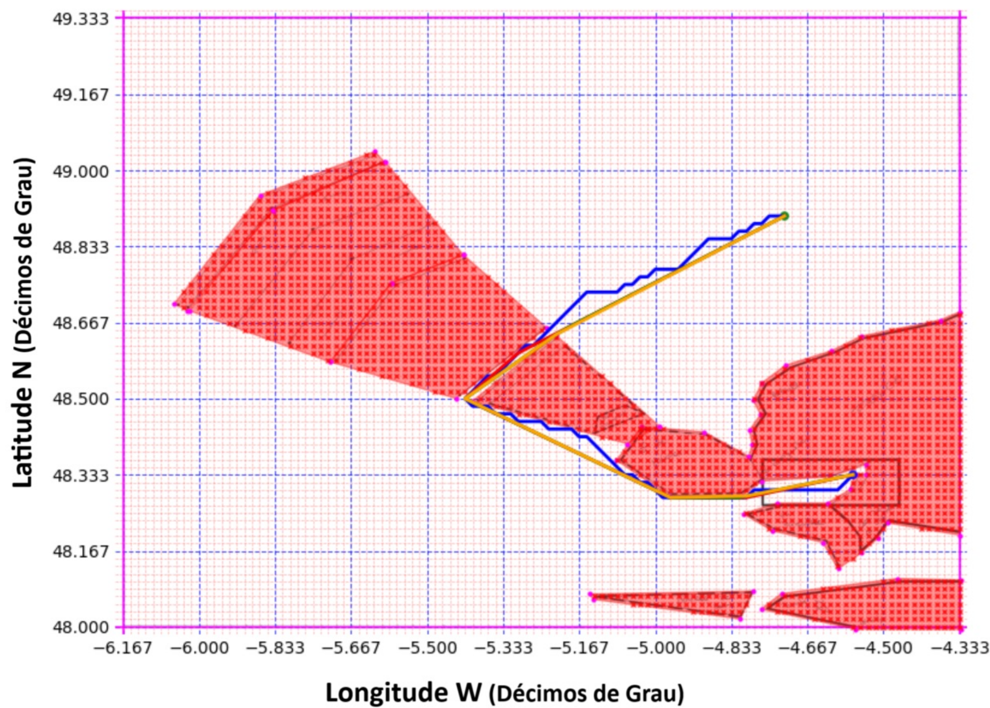

Figura 4.12: Esquema de separação de tráfego interior, com estrada no porto: Derrotas obtidas pelo método proposto.

4.6 Resultados

Para uma consulta mais fácil e melhor visualização dos resultados obtidos ao longo das experiências computacionais, os mesmos foram compilados e sumarizados na tabela 4.1 seguinte.

Tabela 4.1: Sumário dos resultados das experiências computacionais.

	Comprimento da Derrota	(milhas)	
EST	1ªFase	2ªFase, variante 3	Melhoria (%)
Exterior-S	104	98	5.8
Exterior-S-porto	140	132	5.7
Exterior-N	118	110	6.8
Exterior-N-porto	164	153	6.7
Interior	88	83	5.7
Interior-porto	110	103	6.4

Em todos os cenários abordados nas experiências computacionais, o método proposto foi sempre capaz de encontrar uma derrota correspondente a um cenário real, com base nos dados AIS, mesmo apenas aplicando a primeira fase, que serve de ponto de partida para a obtenção de uma derrota melhorada. A segunda fase, como tivemos oportunidade de observar, mostrou capacidade não só de melhorar a derrota obtida na primeira fase,

em média de 6%, como também de minimizar o número de mudanças de rumo, aplicando um procedimento baseado no método do polígono convexo.

5 Notas finais e trabalhos futuros

Como já tivemos a oportunidade de referir inicialmente, o tema deste trabalho está em rápida expansão e ainda com muitos desafios pela frente. Os navios autónomos já são uma realidade, o que aumenta a urgência em desenvolver algoritmos cada vez mais refinados, de forma a melhorar a precisão e eficiência da viagem sujeita a múltiplos fatores, tanto endógenos como exógenos.

Neste documento, procedeu-se a uma simplificação ampla da realidade de forma a diminuir a quantidade e complexidade de variáveis e torná-lo exequível, enquanto nosso primeiro estudo na área. Ainda há muito a desenvolver no setor, e muito mais ainda para aprender e estudar.

Assim, cremos que o objetivo inicialmente proposto foi parcialmente atingido, atendendo ao quão ramificado o tema seria, desde as diferentes fases de navegação, o tipo de navegação e as variáveis a considerar, tendo sido necessário escolher um pequeno ramo dessa árvore para estudar e desenvolver. Tal como descrito no primeiro capítulo, optou-se por nos focarmos na fase do planeamento da viagem. Escolhida a fase, selecionou-se a navegação costeira, tendo como objetivo primordial a minimização da distância.

Dentro deste contexto, acreditamos que o objetivo tenha sido atingido, visto que o método proposto tem o potencial de minimizar não só a distância percorrida entre dois pontos como também a quantidade de pontos de mudança de rumo, aproximando-se, tanto quanto foi possível averiguar empiricamente, das derrotas realizadas pelos navios na área em estudo.

Tendo em conta o funcionamento dos algoritmos escolhidos, a viagem pode ser otimizada não apenas em função da distância mas também em função do tempo de viagem (se considerarmos as condições meteorológicas e os seus efeitos na velocidade); do consumo de combustível (acrescentando informações de propulsão da máquina do navio); e da segurança (se tivermos em conta zonas de pirataria).

Numa fase posterior, reconhecemos que seria necessário aprimorar o método proposto de modo a acomodar mudanças de cartas náuticas e cobrir zonas de navegação mais restritas (navegação portuária e cabotagem), tal como descrito na secção 3.5 com o método

das isolinhas[1], já por si, uma adaptação de métodos semelhantes presentes na literatura.

Ainda neste tipo de navegação, há espaço para melhoramento, como sair da fase do planeamento e considerar objetos também dinâmicos, em que o algoritmo tivesse a capacidade de analisar e adaptar, em cada momento, a melhor derrota mediante as alterações no meio circundante.

Seria interessante, em trabalhos futuros, testar este algoritmo, com eventuais adaptações, na navegação oceânica, onde os obstáculos a evitar seriam essencialmente as condições meteorológicas. Outro aspeto que achamos que merecia uma maior atenção no futuro seria um estudo mais aprofundado de diferentes algoritmos e possíveis sinergias entre eles.

Não podemos deixar de referir alguns aspetos que achamos interessantes sobre o algoritmo A*, descrito na Secção 3.3, enquanto melhoria do algoritmo de Dijkstra, sobre os métodos das isopones e das isócronas, descritos na secção 3.4, e sobre outros métodos de outra natureza como a utilização do conceito de linhas de campo de Energia Potencial e os diagramas de Voronoi.

Por fim, acreditamos que este estudo aqui apresentado tem a inovação, criatividade e qualidade para ser divulgado na comunidade científica, enquanto junção de duas abordagens que, ao trabalharem em conjunto, complementam-se, minimizando os pontos fracos de cada uma isoladamente, junção essa não encontrada em outra literatura estudada.

[1] Método pensado por mim, baseado em métodos semelhantes presentes na literatura. Neste método as linhas de igual comprimento são definidas usando o conceito de pontos de não retorno.

Referências

[1] Unmanned ships: Navigation and more. *Gyroscopy and Navigation 12* (2021), 96–108.

[2] Alexander, L., Ryan, J., and Casey, M. (2004) Integrated navigation system: Not a sum of its parts. Canadian Hidrographic Association.

[3] Avgouleas, K., and Sclavounos, P. (2019) Fuel-efficient ship routing.

[4] Bolbot, V., Theotokatos, G., Boulougouris, E., Wennersberg, L. A., Nordahl, H., Rødseth, , Faivre, J., and Colella, M. (2020) Paving the way towards autonomous shipping development for european waters – the AUTOSHIP project.

[5] Bowditch, N. (2019) *American pratical navigator: An epitome of navigation*, vol. 1. National Geospatial Intelligence Agency.

[6] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. (2009) *Introduction to algorithms*, 3rd ed. The MIT Press.

[7] Costa, E., and Simões, A. (2008) *Inteligência artificial: fundamentos e aplicações*, 3rd ed. FCA.

[8] Dijkstra, E. W. (1959) A note on two problems in connexion with graphs. *Numerische Mathematik 1*, 1 (1959), 269–271.

[9] Emiris, I. Z. (2021)Autonomous and optimized ship routing. In *SNAME 2021 - 7th International Symposium on Ship Operations, Management and Economics* (Athens, Greece), no. SNAME-SOME-2021-017.

[10] Fan, Y., and Nie, Y. (2006) Optimal routing for maximizing the travel time reliability. *Networks and Spatial Economics 6*, 333–344.

[11] Felski, A., and Zwolak, K. (2020) The ocean-going autonomous ship—challenges and threats. *Journal of Marine Science and Engineering 8*, 1.

[12] Freitas, H. M. G. (2013) Sistemas de apoio à decisão: optimização das derrotas dos navios do tipo corveta. Master's thesis, Escola Naval.

[13] Grifoll, M., Martorell, L., Castells, M., and de Osés, F. M. (2018) Ship weather routing using pathfinding algorithms: the case of Barcelona – Palma de Mallorca. *Transportation Research Procedia 33*, 299–306. XIII Conference on Transport Engineering, CIT2018.

[14] Group Technology R. (2018) Remote-controlled and autonomous ships.

[15] Han, P., and Yang, X. (2020) Big data-driven automatic generation of ship route planning in complex maritime environments. *Acta Oceanologica Sinica 39*, AOS-39-08-hanpeng, 113.

[16] Hong, I., Murray, A. T., and Wolf, L. J. (2016) Spatial filtering for identifying a shortest path around obstacles. *Geographical Analysis 48*, 2, 176–190.

[17] IMO. (2023) https://www.imo.org/en/ourwork/safety/pages/electroniccharts.aspx.

[18] Internacional Telecommunication Union - Radiocommunication Sector. *Technical characteristics for an automatic identification system using time division multiple access in the VHF maritime mobile frequency band (02/2014).*

[19] Jia, S., Dai, Z., and Zhang, L. (2018) Automatic ship routing with high reliability and efficiency between two arbitrary points at sea. *Journal of Navigation 72*, 1–17.

[20] Jie, J., and Miao, C. (2021) Analysis and selection of shipping route in ocean. *Journal of Physics: Conference Series 2029*, 1, 012151.

[21] Jørgensen, U., Belingmo, P. R., Murray, B., Berge, S. P., and Pobitzer, A. (2022) Ship route optimization using hybrid physics-guided machine learning. *Journal of Physics: Conference Series 2311*, 1, 012037.

[22] Kongsberg. (2023) https://www.kongsberg.com.

[23] Kuhlemann, S., and Tierney, K. (2020) A genetic algorithm for finding realistic sea routes considering the weather. *Journal of Heuristics*.

[24] Lee, W., Choi, G.-H., and wan Kim, T. (2021) Visibility graph-based path-planning algorithm with quadtree representation. *Applied Ocean Research 117*, 102887.

[25] Li, W., and Xie, P. (2020) Automatic pilot ship route planning based on a rrt guided genetic algorithm. *Journal of Physics: Conference Series 1550*, 3, 032085.

[26] Li, X. (2021) A research on the design and optimization of shipping routes in the Arctic. *Journal of Physics: Conference Series 1848*, 1, 012138.

[27] Liu, Y., Wang, T., and Xu, H. (2022) PE-A* algorithm for ship route planning based on field theory. *IEEE Access 10*, 36490–36504.

[28] Miguens, A. P. *(2019) Navegação: a ciência e a Arte - Navegação astronómica e derrotas.*

[29] Monirisk. (2023) monirisk.tecnico.ulisboa.pt.

[30] Monteiro, M. Slides de mestrado de navegação e geomática. Escola Naval.

[31] Novac, V., and Rusu, E. (2022) Ship routing using A* algorithm – a Black Sea case study. *Journal of Environmental Protection and Ecology 23*, 586–594.

[32] Novac, V., Rusu, E., Gasparotti, C., and Stăvărache, G. (2020) Ship routing in the Black Sea based on Dijkstra algorithm. pp. 301–308.

[33] Projeto MUNIN. (2023) www.unmanned-ship.org/munin.

[34] Rolls Royce. (2016) Remote and autonomous ships: The next steps. AAWA Project.

[35] Rolls Royce and FinFerries. (2018) SVAN Project. slides.

[36] Rong, H., Teixeira, A., and Guedes Soares, C. (2019) Ship trajectory uncertainty prediction based on a gaussian process model. *Ocean Engineering 182*, 499–511.

[37] Rong, H., Teixeira, A., and Guedes Soares, C. (2020) Data mining approach to shipping route characterization and anomaly detection based on AIS data. *Ocean Engineering 198*, 106936.

[38] Silveira, P., Teixeira, A., and Guedes Soares, C. (2011) Analysis of maritime traffic off the coast of Portugal.

[39] Silveira, P., Teixeira, P., and Guedes Soares, C. (2019) AIS based shipping routes using the Dijkstra algorithm. *TransNav, the International Journal on Marine Navigation and Safety of Sea Transportation 13*, 3, 565–571.

[40] Tsou, M.-C., and Cheng, H.-C. (2013) An ant colony algorithm for efficient ship routing. *Polish Maritime Research 20*, 3, 28–38.

[41] Tunaley, J. K. (2013) Utility of various ais messages for maritime awareness.

[42] Walther, L., Rizvanolli, A., Wendebourg, M., and Jahn, C. (2016) Modeling and optimization algorithms in ship weather routing. *International Journal of e-Navigation and Maritime Economy 4*, 31–45.

[43] Wang, H. (2018) Voyage optimization algorithms for ship safety and energy-efficiency.

[44] Wang, Q., Zhong, M., Shi, G., Zhao, J., and Bai, C. (2021) Route planning and tracking for ships based on the ecdis platform. *IEEE Access 9*, 71754–71762.

[45] Wen, Y., Sui, Z., Zhou, C., Xiao, C., Chen, Q., Han, D., and Zhang, Y. (2020) Automatic ship route design between two ports: A data-driven method. *Applied Ocean Research 96*, 102049.

[46] Xu, J., Wang, Z., and Song, L. (2011) Study on optimal algorithm for shipping route automatic-generation based on electronic chart. *Applied Mechanics and Materials 105-107*.

[47] Zhao, W., Wang, Y., Zhang, Z., and Wang, H. (2021) Multicriteria ship route planning method based on improved particle swarm optimization–genetic algorithm. *Journal of Marine Science and Engineering 9*, 357.

[48] Zhengping, T., Chao, J., and Xiaohai, W. (2021) Design of ship route through waterway based on dynamic programming. *Journal of Physics: Conference Series 1906*, 1, 012001.

ANEXO

```
1    import math
2
3    import numpy as np
4    import matplotlib.pyplot as plt
5    import tkinter as tk
6
7    from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
8    from matplotlib.backends.backend_tkagg import NavigationToolbar2Tk
9    from matplotlib.backend_bases import MouseButton
10   from matplotlib.figure import Figure
11   from matplotlib.lines import Line2D
12   from matplotlib.patches import Circle, Polygon
13   from matplotlib.path import Path
14
15   from shapely.geometry import LineString, Polygon as Polygon_
16
17   from timeit import default_timer as timer
18
19   from tkinter.filedialog import asksaveasfile, askopenfile
20
21   #=============================================================================
22   # MODULO NODOS EXTREMOS
23   #-----------------------------------------------------------------------------
24
25   START_COLOR = "green"
26   START_ALPHA = 0.8
27   START_SELECTED_COLOR = "limegreen"
28
29   END_COLOR = "blue"
30   END_ALPHA = 0.8
31   END_SELECTED_COLOR = "deepskyblue"
32
33   def eucdist(P, Q):
34       '''
35       dist = eucdist(P, Q)
36
37       Distância Euclideana entre os pontos P(x,y) e Q(x,y).
38       '''
39       return math.sqrt((Q[0]-P[0])**2 + (Q[1]-P[1])**2)
40
41   class NodosExtremos:
42       '''
43       NodosExtremos(navegacao)
44
45       Cria os vértices inicial e final da rota a encontrar.
46       '''
47       def __init__(self, navegacao):
48           self.navegacao = navegacao
49           nrows, ncols = navegacao.nrows, navegacao.ncols
50           radius = navegacao.radius
51           # posições em coordenadas geográficas
52           x0, y0 = navegacao.xy_from_ij(0, 0)
53           x1, y1 = navegacao.xy_from_ij(nrows-1, ncols-1)
54           start = Circle(
55               xy=(x0, y0),
56               radius=radius, color=START_COLOR, alpha=START_ALPHA, animated=True,
57           )
58           end = Circle(
59               xy=(x1, y1),
60               radius=radius, color=END_COLOR, alpha=END_ALPHA, animated=True,
61           )
62           navegacao.ax.add_artist(start)
63           navegacao.ax.add_artist(end)
64           self._S = (start, START_COLOR, START_SELECTED_COLOR)
65           self._T = (end, END_COLOR, END_SELECTED_COLOR)
66           self._initialise_variables()
67
68       def _initialise_variables(self):
```

```
69            # indica se algum dos nodos está em movimento seguindo o rato
70            self._is_active = False
71            # nodo em movimento
72            self._active_node = None
73
74        def active(self):
75            return self._is_active
76
77        def activate(self, node):
78            '''
79            activate(node)
80
81            Activa o nodo dado para movimento.
82            '''
83            self._is_active = True
84            self._active_node = node
85            node[0].set_color(node[2])
86
87        def _check_changes(self):
88            '''
89            Averigua qualquer alteração na configuração da área de navegação,
90            alterando as posições dos nodos inicial e final, caso necessário.
91            '''
92            nrows, ncols = self.navegacao.nrows, self.navegacao.ncols
93            start, end = self._S[0], self._T[0]
94            # posições no formato (i, j)
95            i0, j0 = self.navegacao.ij_from_xy(*start.center)
96            i1, j1 = self.navegacao.ij_from_xy(*end.center)
97            # ajusta posições
98            if i0 < 0: i0 = 0
99            if j0 < 0: j0 = 0
100           if i0 >= nrows: i0 = nrows-1
101           if j0 >= ncols: j0 = ncols-1
102           if i1 < 0: i1 = 0
103           if j1 < 0: j1 = 0
104           if i1 >= nrows: i1 = nrows-1
105           if j1 >= ncols: j1 = ncols-1
106           # posições em coordenadas geográficas (x, y)
107           x0, y0 = self.navegacao.xy_from_ij(i0, j0)
108           x1, y1 = self.navegacao.xy_from_ij(i1, j1)
109           start.center = x0, y0
110           end.center = x1, y1
111
112       def deactivate(self):
113           '''
114           deactivate()
115
116           Desactiva o nodo activo corrente.
117           '''
118           self._active_node[0].set_color(self._active_node[1])
119           self._active_node = None
120           self._is_active = False
121
122       def draw(self):
123           self._check_changes()
124           self.navegacao.ax.draw_artist(self._S[0])
125           self.navegacao.ax.draw_artist(self._T[0])
126
127       def move_active_node_to(self, P):
128           '''
129           move_selected_node_to(P)
130
131           Muda o nodo activo para a posição P(x,y).
132           '''
133           self._active_node[0].center = P
134
135       def swap(self):
136           '''
```

```
137             swap()
138 
139             Troca de posições os nodos inicial e final.
140             '''
141             center = self._S[0].center
142             self._S[0].center = self._T[0].center
143             self._T[0].center = center
144 
145         def which_selected(self, P):
146             '''
147             node = which_selected(P)
148 
149             Retorna o nodo que está na vizinhança do ponto P(x,y).
150             '''
151             if eucdist(P, self._S[0].center) < self.navegacao.radius:
152                 return self._S
153             elif eucdist(P, self._T[0].center) < self.navegacao.radius:
154                 return self._T
155             else: return None
156 
157     #================================================================================
158     # MODULO POLIGONO
159     #--------------------------------------------------------------------------------
160 
161     UNFINALISED_POLY_FACECOLOR = "orange"
162     UNFINALISED_POLY_EDGECOLOR = "red"
163     UNFINALISED_POLY_ALPHA = 0.4
164     UNFINALISED_VERTEX_POINTS_COLOR = "red"
165     UNFINALISED_VERTEX_POINTS_ALPHA = 0.8
166 
167     FINALISED_POLY_FACECOLOR = "red"
168     FINALISED_POLY_EDGECOLOR = "red"
169     FINALISED_POLY_ALPHA = 0.4
170     FINALISED_VERTEX_POINTS_COLOR = "magenta"
171     FINALISED_VERTEX_POINTS_ALPHA = 1.0
172 
173     VERTEX_POINTS_SIZE = 2
174 
175     SELECTED_POLY_FACECOLOR = "orangered"
176     SELECTED_POLY_ALPHA = 0.6
177 
178     INTERIOR_POINTS_COLOR = "red"
179     INTERIOR_POINTS_ALPHA = 0.8
180     INTERIOR_POINTS_SIZE = 3
181 
182     C_MINIMUM_VERTICES = 3
183 
184     E_VERTEX_APPENDED = 1
185     E_VERTEX_IGNORED = 2
186     E_POLY_FINALISED = 3
187 
188     class Poligono:
189         '''
190         Poligono(navegacao)
191 
192         Cria um polígono na área de navegação, definindo uma zona não navegável.
193         '''
194         def __init__(self, navegacao):
195             self.navegacao = navegacao
196             # vértices que definem o polígono, em coordenadas geográficas
197             self._vertices = []
198             # pontos interiores ao polígono, em coordenadas geográficas
199             self._interior_points = []
200             # objectos associados ao polígono inseridos no gráfico
201             self._poly_obj = None
202             self._vertex_points_obj = None
203             self._interior_points_obj = None
204             self._initialise_variables()
```

```
205
206         def _initialise_variables(self):
207             # indica se o polígono está em construção
208             self._under_construction = False
209             # indica se o polígono está seleccionado
210             self._is_selected = False
211             # indica qual dos vértices está sob selecção do rato
212             self._i_select = -1
213
214         def finalised(self):
215             return self._STATE == E_POLY_FINALISED
216
217         def selected(self):
218             return self._is_selected
219
220         def under_construction(self):
221             return self._under_construction
222
223         def valid(self):
224             return len(self._vertices) >= C_MINIMUM_VERTICES
225
226         def vertex_appended(self):
227             return self._STATE == E_VERTEX_APPENDED
228
229         def vertex_ignored(self):
230             return self._STATE == E_VERTEX_IGNORED
231
232         def append_vertex(self, P):
233             '''
234             append_vertex(P)
235
236             Acrescenta o ponto P(x,y) ao polígono sob construção.
237
238             STATE: { vertex_appended, vertex_ignored, finalised }
239
240             Nota: É suposto o ponto P(x,y) corresponder à posição actual do rato,
241             com coordenadas (event.xdata, event.ydata). Quando os dois últimos
242             vértices da lista estão suficientemente próximos um do outro, o polígono
243             é finalisado, sendo a lista reduzida em uma unidade (o último vértice é
244             retirado para evitar duplicação de vértices). Se P coincidir com algum
245             dos outros vértices já existentes, é ignorado. Caso contrário, P é
246             acrescentado à lista de vértices do polígono.
247             '''
248             if self._poly_obj is None:
249                 # a lista é inicializada com duas entradas; a segunda tem apenas a
250                 # finalidade de memorizar a posição do rato enquanto em movimento
251                 self._vertices.append(P)
252                 self._vertices.append(P)
253                 self.set_artists()
254                 self.set_finalised(False)
255                 self._STATE = E_VERTEX_APPENDED
256             else:
257                 radius = self.navegacao.radius
258                 i_before_last = len(self._vertices)-2
259                 Q = self._vertices[i_before_last]
260                 # se os dois últimos vértices correspondem ao mesmo, finaliza
261                 if eucdist(P, Q) < radius:
262                     self._vertices.pop()
263                     self._update_artists()
264                     self.set_finalised(True)
265                     self._STATE = E_POLY_FINALISED
266                 else:
267                     # se o vértice P já está na lista, é ignorado
268                     for Q in self._vertices[:i_before_last]:
269                         if eucdist(P, Q) < radius:
270                             self._STATE = E_VERTEX_IGNORED
271                             return
272                     # P é adicionado à lista de vértices
```

```
273                 self._vertices.append(P)
274                 self._update_artists()
275                 self._STATE = E_VERTEX_APPENDED
276 
277     def attach_interior_points(self):
278         '''
279         attach_interior_points()
280 
281         Anexa os pontos interiores do polígono à àrea de navegação.
282         '''
283         navegacao = self.navegacao
284         nrows, ncols = navegacao.nrows, navegacao.ncols
285         step = navegacao.fstep
286         data = self._poly_obj.get_xy()
287         min_ = np.min(data, axis=0)
288         max_ = np.max(data, axis=0)
289         i1, j0 = navegacao.ij_from_xy(*min_)
290         i0, j1 = navegacao.ij_from_xy(*max_)
291         i0 = min(max(0, i0), nrows-1)
292         i1 = min(max(0, i1), nrows-1)
293         j0 = min(max(0, j0), ncols-1)
294         j1 = min(max(0, j1), ncols-1)
295         path = Path(data)
296         interior_points = self._interior_points
297         for i in range(i0, i1+1):
298             for j in range(j0, j1+1):
299                 x, y = navegacao.xy_from_ij(i, j)
300                 if path.contains_point((x, y)):
301                     interior_points.append((x, y))
302         # se existirem pontos interiores, os mesmos são mostrados
303         if interior_points:
304             data = np.array(interior_points)
305             if self._interior_points_obj is None:
306                 interior_points_obj = Line2D(
307                     xdata=data[:,0],
308                     ydata=data[:,1],
309                     markersize = INTERIOR_POINTS_SIZE,
310                     marker="x",
311                     linestyle="None",
312                     color=INTERIOR_POINTS_COLOR,
313                     alpha=INTERIOR_POINTS_ALPHA, animated=True,
314                 )
315                 navegacao.ax.add_artist(interior_points_obj)
316                 self._interior_points_obj = interior_points_obj
317             else:
318                 self._interior_points_obj.set_data(data.T)
319         # delimita a área de navegação onde está inserido o polígono
320         self.xmin, self.ymin = min_ - step
321         self.xmax, self.ymax = max_ + step
322 
323     def contains_point(self, P):
324         '''
325         bool = contains_point(P)
326 
327         Retorna True se o polígono contém o ponto P.
328         '''
329         data = self._poly_obj.get_xy()
330         path = Path(data)
331         return path.contains_point(P)
332 
333     def contains_vertex(self, P):
334         '''
335         bool = contains_vertex(P)
336 
337         Retorna True se o ponto P é vértice do polígono.
338         '''
339         radius = self.navegacao.radius
340         for i, Q in enumerate(self._vertices):
```

```
341                 if eucdist(P, Q) < radius:
342                     self._i_select = i
343                     return True
344             return False
345 
346         def draw(self):
347             if self._vertices:
348                 self.navegacao.ax.draw_artist(self._poly_obj)
349                 self.navegacao.ax.draw_artist(self._vertex_points_obj)
350                 if self._interior_points:
351                     self.navegacao.ax.draw_artist(self._interior_points_obj)
352 
353         def inarea(self, P):
354             '''
355             inarea(P)
356 
357             Retorna True se o ponto P(x,y) está dentro da área que delimita o
358             polígono. Isto permite averiguar se, durante a navegação, o ponto P pode
359             ser atingido a partir de outro ponto Q(x,y) sem que o segmento de recta
360             QP intersecte o polígono. Caso contrário, a rota de Q para P é excluída.
361             '''
362             x, y = P
363             return x >= self.xmin and \
364                 x <= self.xmax and y >= self.ymin and y <= self.ymax
365 
366         def move_selected_vertex_to(self, P):
367             '''
368             move_selected_vertex_to(P)
369 
370             Move a posição do vértice seleccionado para P.
371             '''
372             self._vertices[self._i_select] = P
373             self._update_artists()
374 
375         def remove_interior_points(self):
376             '''
377             remove_interior_points()
378 
379             Remove os pontos interiores do polígono da área de navegação.
380             '''
381             if self._interior_points:
382                 self._interior_points.clear()
383                 self._interior_points_obj.set_xdata([])
384                 self._interior_points_obj.set_ydata([])
385 
386         def reset(self):
387             '''
388             reset()
389 
390             Reinicia o polígono, no seu todo.
391             '''
392             if self._vertices:
393                 self.remove_interior_points()
394                 self._vertices.clear()
395                 self._vertex_points_obj.set_xdata([])
396                 self._vertex_points_obj.set_ydata([])
397                 self._poly_obj.remove()
398                 self._poly_obj = None
399             self._initialise_variables()
400 
401         def set_artists(self):
402             '''
403             set_artists()
404             '''
405             data = np.array(self._vertices)
406             self._poly_obj = Polygon(xy=data, animated=True)
407             self.navegacao.ax.add_artist(self._poly_obj)
408             if self._vertex_points_obj is None:
```

```
409                 self._vertex_points_obj = Line2D(
410                     xdata=data[:,0],
411                     ydata=data[:,1],
412                     markersize = VERTEX_POINTS_SIZE,
413                     marker="o", linestyle="None", animated=True,
414                 )
415                 self.navegacao.ax.add_artist(self._vertex_points_obj)
416             else:
417                 self._vertex_points_obj.set_data(data.T)
418 
419         def set_finalised(self, finalised):
420             '''
421             set_finalised(finalised)
422             '''
423             self._under_construction = not finalised
424             if finalised:
425                 self._poly_obj.set_facecolor(FINALISED_POLY_FACECOLOR)
426                 self._poly_obj.set_edgecolor(FINALISED_POLY_EDGECOLOR)
427                 self._poly_obj.set_alpha(FINALISED_POLY_ALPHA)
428                 self._vertex_points_obj.set_color(FINALISED_VERTEX_POINTS_COLOR)
429                 self._vertex_points_obj.set_alpha(FINALISED_VERTEX_POINTS_ALPHA)
430             else:
431                 self._poly_obj.set_facecolor(UNFINALISED_POLY_FACECOLOR)
432                 self._poly_obj.set_edgecolor(UNFINALISED_POLY_EDGECOLOR)
433                 self._poly_obj.set_alpha(UNFINALISED_POLY_ALPHA)
434                 self._vertex_points_obj.set_color(UNFINALISED_VERTEX_POINTS_COLOR)
435                 self._vertex_points_obj.set_alpha(UNFINALISED_VERTEX_POINTS_ALPHA)
436 
437         def set_selected(self, selected):
438             '''
439             set_selected(selected)
440             '''
441             self._is_selected = selected
442             if selected:
443                 self._poly_obj.set_facecolor(SELECTED_POLY_FACECOLOR)
444                 self._poly_obj.set_alpha(SELECTED_POLY_ALPHA)
445             else:
446                 self._poly_obj.set_facecolor(FINALISED_POLY_FACECOLOR)
447                 self._poly_obj.set_alpha(FINALISED_POLY_ALPHA)
448 
449         def shrink(self):
450             '''
451             shrink()
452 
453             Encolhe o polígono em um vértice (o penúltimo vértice é removido).
454             '''
455             if len(self._vertices) > 2:
456                 i_before_last = len(self._vertices)-2
457                 self._vertices.pop(i_before_last)
458                 self._update_artists()
459             else:
460                 self.reset()
461 
462         def track_mouse_position(self, P):
463             '''
464             track_mouse_position(P)
465 
466             O vértice P, de coordenadas (event.xdata, event.ydata), é colocado na
467             última entrada da lista de vértices.
468             '''
469             i_last = len(self._vertices)-1
470             self._vertices[i_last] = P
471             self._update_artists()
472 
473         def _update_artists(self):
474             '''
475             Actualiza os objectos associados ao polígono inseridos no gráfico.
476             '''
```

```
477             data = np.array(self._vertices)
478             self._poly_obj.set_xy(data)
479             self._vertex_points_obj.set_data(data.T)
480
481     #================================================================================
482     # MODULO LISTA de POLIGONOS
483     #--------------------------------------------------------------------------------
484
485     E_POLY_APPENDED = 10
486     E_POLY_IGNORED = 11
487
488     class ListaPoligonos:
489         '''
490         ListaPoligonos(navegacao)
491
492         Gere a lista de polígonos definidos na área de navegação.
493         '''
494         def __init__(self, navegacao):
495             self.navegacao = navegacao
496             self._list = []
497             self._initialise_variables()
498
499         def _initialise_variables(self):
500             # indica se algum dos polígonos está em alteração
501             self._is_changing = False
502             # polígono em alteração
503             self._active_poly = None
504
505         def changing(self):
506             return self._is_changing
507
508         def empty(self):
509             return not self._list
510
511         def poly_appended(self):
512             return self._STATE == E_POLY_APPENDED
513
514         def poly_ignored(self):
515             return self._STATE == E_POLY_IGNORED
516
517         def append(self, poly):
518             '''
519             bool = append(poly)
520
521             O polígono é acrescentado à lista de polígonos. Retorna False se o
522             polígono não é válido, sendo este reinicializado.
523             '''
524             if poly.valid():
525                 poly.attach_interior_points()
526                 self._list.append(poly)
527                 return True
528             else:
529                 poly.reset()
530                 return False
531
532         def deselect(self):
533             '''
534             bool = deselect()
535
536             Retorna True se algum polígono estava seleccionado.
537             '''
538             selected = False
539             for poly in self._list:
540                 if poly.selected():
541                     poly.set_selected(False)
542                     selected = True
543             return selected
544
```

```
545         def draw(self):
546             for poly in self._list:
547                 poly.draw()
548 
549         def move_changing_vertex_to(self, P):
550             '''
551             move_changing_vertex_to(P)
552             '''
553             self._active_poly.move_selected_vertex_to(P)
554 
555         def remove_selected(self):
556             '''
557             bool = remove_selected()
558 
559             Remove os polígonos seleccionados. Retorna True se algum foi removido.
560             '''
561             polygons = self._list
562             k, removed = 0, False
563             while k < len(polygons):
564                 poly = polygons[k]
565                 if poly.selected():
566                     poly.reset()
567                     polygons.pop(k)
568                     removed = True
569                 else:
570                     k += 1
571             return removed
572 
573         def reset(self):
574             '''
575             reset()
576 
577             Os polígonos são removidos da lista.
578             '''
579             for poly in self._list:
580                 poly.reset()
581             self._list.clear()
582             self._initialise_variables()
583 
584         def start_changing(self, poly):
585             '''
586             start_changing(poly)
587             '''
588             self._is_changing = True
589             self._active_poly = poly
590             poly.set_selected(True)
591             poly.remove_interior_points()
592 
593         def stop_changing(self):
594             '''
595             stop_changing()
596             '''
597             self._active_poly.attach_interior_points()
598             self._active_poly = None
599             self._is_changing = False
600 
601         def update_interior_points(self):
602             '''
603             update_interior_points(self):
604 
605             Actualiza os pontos interiores dos polígonos na área de navegação.
606             '''
607             for poly in self._list:
608                 poly.remove_interior_points()
609                 poly.attach_interior_points()
610 
611         def which_contains_point(self, P):
612             '''
```

```
613             poly = which_contains_point(P)
614
615             Retorna o polinómio cujo P é ponto interior. Caso não exista, retorna
616             None.
617             '''
618             for poly in self._list:
619                 if poly.contains_point(P):
620                     return poly
621             return None
622
623         def which_contains_vertex(self, P):
624             '''
625             poly = which_contains_vertex(P)
626
627             Retorna o polinómio cujo P é vértice. Caso não exista, retorna None.
628             '''
629             for poly in self._list:
630                 if poly.contains_vertex(P):
631                     return poly
632             return None
633
634     #===============================================================================
635     # MODULO ROTA OPTIMA
636     #-------------------------------------------------------------------------------
637
638     PATH_BEST_LINE_COLOR = "blue"
639     PATH_BEST_LINE_ALPHA = 1.0
640     PATH_BEST_LINE_WIDTH = 2
641
642     PATH_IMPROVED_LINE_COLOR = "red"
643     PATH_IMPROVED_LINE_COLOR_2 = "green"
644     PATH_IMPROVED_LINE_COLOR_3 = "orange"
645
646     def collinear(A, B, C):
647         '''
648         collinear(A, B, C)
649
650         Averigua se as coordenadas geográficas A, B e C são colineares.
651         '''
652         a = A[0] * (B[1] - C[1])
653         b = B[0] * (C[1] - A[1])
654         c = C[0] * (A[1] - B[1])
655         return abs(a + b + c) <= 1e-16
656
657     def geodist(P, Q):
658         '''
659         geodist(P, Q)
660
661         Calcula a distância geodésica (em milhas náuticas) entre P e Q.
662         '''
663         dx = (Q[0] - P[0])
664         dy = (Q[1] - P[1])
665         return 60.0 * math.sqrt(dx**2 + dy**2)
666
667     class RotaOptima:
668         '''
669         RotaOptima(navegacao)
670         '''
671         def __init__(self, navegacao):
672             self.navegacao = navegacao
673             self._best_path = []
674             self._improved_path = []
675             self._improved_path2 = []
676             self._improved_path3 = []
677             self._best_path_obj = Line2D(
678                 xdata=[],
679                 ydata=[],
680                 color=PATH_BEST_LINE_COLOR,
```

```
681                 linewidth=PATH_BEST_LINE_WIDTH,
682                 alpha=PATH_BEST_LINE_ALPHA, animated=True,
683             )
684             self._improved_path_obj = Line2D(
685                 xdata=[],
686                 ydata=[],
687                 color=PATH_IMPROVED_LINE_COLOR,
688                 linewidth=PATH_BEST_LINE_WIDTH,
689                 alpha=PATH_BEST_LINE_ALPHA, animated=True,
690             )
691             self._improved_path2_obj = Line2D(
692                 xdata=[],
693                 ydata=[],
694                 color=PATH_IMPROVED_LINE_COLOR_2,
695                 linewidth=PATH_BEST_LINE_WIDTH,
696                 alpha=PATH_BEST_LINE_ALPHA, animated=True,
697             )
698             self._improved_path3_obj = Line2D(
699                 xdata=[],
700                 ydata=[],
701                 color=PATH_IMPROVED_LINE_COLOR_3,
702                 linewidth=PATH_BEST_LINE_WIDTH,
703                 alpha=PATH_BEST_LINE_ALPHA, animated=True,
704             )
705             navegacao.ax.add_artist(self._best_path_obj)
706             navegacao.ax.add_artist(self._improved_path_obj)
707             navegacao.ax.add_artist(self._improved_path2_obj)
708             navegacao.ax.add_artist(self._improved_path3_obj)
709 
710         def ij_from_v(self, v):
711             '''
712             ij_from_v(v)
713 
714             Converte índices v em entradas (i, j).
715             '''
716             i = v // self.navegacao.ncols
717             j = v % self.navegacao.ncols
718             return i, j
719 
720         def v_from_ij(self, i, j):
721             '''
722             v_from_ij(i, j)
723 
724             Converte entradas (i, j) em índices v.
725             '''
726             return j + i*self.navegacao.ncols
727 
728         def v_from_xy(self, x, y):
729             '''
730             v_from_xy(x, y)
731 
732             Converte coordenadas geográficas (x, y) em índices v.
733             '''
734             navegacao = self.navegacao
735             step = navegacao.fstep
736             ri = round((y-navegacao.ymin-step)/step)
737             rj = round((x-navegacao.xmin-step)/step)
738             v = (rj+1) + (navegacao.nrows-ri-2)*navegacao.ncols
739             return v
740 
741         def xy_from_v(self, v):
742             '''
743             xy_from_v(v)
744 
745             Converte índices v em coordenadas geográficas (x, y).
746             '''
747             navegacao = self.navegacao
748             step = navegacao.fstep
```

```
749             i = v // navegacao.ncols
750             j = v % navegacao.ncols
751             x = navegacao.xmin + j*step
752             y = navegacao.ymin + (navegacao.nrows-1-i)*step
753             return x, y
754
755         def compute_best_route(self):
756             '''
757             etime, ok = compute_best_route()
758
759             Determina a rota óptima usando o método de Dijkstra.
760             '''
761             navegacao = self.navegacao
762             full_compass_rose = navegacao.full_compass_rose
763             nrows, ncols = navegacao.nrows, navegacao.ncols
764
765             # define a matriz de pontos navegáveis e não navegáveis
766             t0 = timer()
767             A_ = np.ones((nrows, ncols), dtype=np.uint8)
768             for poly in navegacao._polygons._list:
769                 for P in poly._interior_points:
770                     i, j = navegacao.ij_from_xy(*P)
771                     A_[i,j] = 0
772             etime = timer() - t0
773             A = A_.ravel()
774             N = len(A)
775             self._A = A
776
777             # valida os vértices inicial e final para o cálculo da rota óptima
778             start = self.v_from_xy(*navegacao._nodes._S[0].center)
779             end = self.v_from_xy(*navegacao._nodes._T[0].center)
780             if not A[start]:
781                 etime = round(etime, 4)
782                 navegacao.showmessage(f"Elapsed time: {etime} sec", "blue")
783                 err = "O vértice inicial é não navegável."
784                 tk.messagebox.showwarning(title="Aviso", message=err)
785                 return etime, False
786             if not A[end]:
787                 etime = round(etime, 4)
788                 navegacao.showmessage(f"Elapsed time: {etime} sec", "blue")
789                 err = "O vértice final é não navegável."
790                 tk.messagebox.showwarning(title="Aviso", message=err)
791                 return etime, False
792
793             self._start, self._end = start, end
794
795             if full_compass_rose:
796                 rota_i = (-1, -1, -1, 0, 0, 1, 1, 1)
797                 rota_j = (-1, 0, 1, -1, 1, -1, 0, 1)
798             else:
799                 rota_i = (-1, 0, 0, 1)
800                 rota_j = (0, -1, 1, 0)
801             # necessário para testar a intersecção entre a rota e os polígonos
802             shapely_polygons = self._shapely_polygons()
803
804             t0 = timer()
805
806             # INICIALIZAÇÃO
807
808             k = 0
809             # vértices que têm etiqueta no estado temporário
810             T = np.full(N, True, dtype=bool)
811             # valores das etiquetas
812             labels = np.full(N, np.inf, dtype=float)
813             # antecessores nas rotas mais curtas desde o vértice inicial
814             paths = np.arange(0, N, dtype=int)
815             # vértice com etiqueta no estado permanente
816             p = start
```

```
817            T[p] = False
818            labels[p] = 0.0
819
820            # CICLO PRINCIPAL
821
822            STOP = False
823            while not STOP:
824                k += 1
825                i0, j0 = self.ij_from_v(p)
826                P = navegacao.xy_from_ij(i0, j0)
827
828                # ACTUALIZAÇÃO DAS ETIQUETAS
829
830                L_p = labels[p]
831                # para cada sucessor de p...
832                for i, j in zip(rota_i, rota_j):
833                    i1, j1 = i0+i, j0+j
834                    # posição (i1, j1) do sucessor fora da grelha
835                    if i1 < 0 or i1 >= nrows or j1 < 0 or j1 >= ncols:
836                        continue
837                    v = self.v_from_ij(i1, j1)
838                    # sucessor v, no formato índice, é não navegável
839                    if not A[v]:
840                        continue
841                    # etiqueta do vértice v está no estado permanente
842                    if not T[v]:
843                        continue
844                    # ... v em coordenadas geográficas
845                    Q = navegacao.xy_from_ij(i1, j1)
846                    # segmento de recta PQ intersecta um polígono não navegável
847                    if self._intersect(P, Q, shapely_polygons, improving=False):
848                        continue
849                    L_new = L_p + geodist(P, Q)
850                    # actualiza a etiqueta do vértice v
851                    if L_new < labels[v]:
852                        labels[v] = L_new
853                        paths[v] = p
854
855                # PROCURA DO PRÓXIMO VÉRTICE COM ETIQUETA PERMANENTE
856
857                while True:
858                    I = np.where(T)[0]
859                    # não existem mais vértices com etiqueta no estado temporário
860                    if len(I) == 0:
861                        STOP = True
862                        break
863                    idx = np.argmin(labels[I])
864                    p = I[idx]
865                    T[p] = False
866                    # se o vértice p encontrado for navegável, aceita-se; caso
867                    # contrário, a procura continua
868                    if A[p]:
869                        break
870                # o vértice p encontrado corresponde ao vértice final
871                if p == end:
872                    STOP = True
873
874            etime += timer() - t0
875
876            # DEFINE A ROTA ÓPTIMA A PARTIR DOS CÁLCULOS REALIZADOS ANTERIORMENTE
877
878            path = [end]
879            v = end
880            if paths[v] == end:
881                etime = round(etime, 4)
882                navegacao.showmessage(f"Elapsed time: {etime} sec", "blue")
883                err = "Não existe rota entre os pontos indicados."
884                tk.messagebox.showinfo(title="Informação", message=err)
```

```
885                 return etime, False
886
887             t0 = timer()
888
889             while True:
890                 v = paths[v]
891                 path.append(v)
892                 if v == start:
893                     break
894             path.reverse()
895             for v in path:
896                 P = self.xy_from_v(v)
897                 self._best_path.append(P)
898             self._best_value = labels[end]
899
900             etime += timer() - t0
901
902             data = np.array(self._best_path)
903             self._best_path_obj.set_data(data.T)
904             return round(etime, 4), True
905
906         def _connect(self, A, B, i_A, i_B, path):
907             count = i_B - i_A
908             dx = (B[0]-A[0]) / count
909             dy = (B[1]-A[1]) / count
910             for i in range(1, count):
911                 path[i_A+i] = (A[0]+i*dx, A[1]+i*dy)
912
913         def draw(self):
914             self.navegacao.ax.draw_artist(self._best_path_obj)
915             self.navegacao.ax.draw_artist(self._improved_path_obj)
916             self.navegacao.ax.draw_artist(self._improved_path2_obj)
917             self.navegacao.ax.draw_artist(self._improved_path3_obj)
918
919         def _improve_route_version_1(self, path, shapely_polygons):
920             '''
921             Primeira versão da rota melhorada.
922             '''
923             N = len(path)
924             pending = False
925             i_A, i_B, i_C = 0, 1, 2
926             A, B = path[:2]
927             while True:
928                 if i_C >= N:
929                     if pending:
930                         self._connect(A, B, i_A, i_B, path)
931                     break
932                 C = path[i_C]
933                 if collinear(A, B, C):
934                     pass
935                 elif not self._intersect(A, C, shapely_polygons, improving=True):
936                     pending = True
937                 else:
938                     self._connect(A, B, i_A, i_B, path)
939                     pending = False
940                     i_A = i_B
941                     A = B
942                 i_B = i_C
943                 i_C += 1
944                 B = C
945
946         def _improve_route_version_2(self, path, shapely_polygons):
947             '''
948             Segunda versão da rota melhorada.
949             '''
950             N = len(path)
951             pending = False
952             i_A, i_B, i_C = 0, 1, 2
```

```
953             A, B = path[:2]
954             while True:
955                 if i_C >= N:
956                     if pending:
957                         self._connect(A, B, i_A, i_B, path)
958                     break
959                 C = path[i_C]
960                 if collinear(A, B, C):
961                     pass
962                 elif not self._intersect(A, C, shapely_polygons, improving=True):
963                     pending = True
964                 else:
965                     self._connect(A, B, i_A, i_B, path)
966                     pending = False
967                     while True:
968                         i_A += 1
969                         A = path[i_A]
970                         if i_A == i_B or not self._intersect(A, C, shapely_polygons,
971     improving=True):
972                             break
973                     if i_A < i_B:
974                         self._connect(A, C, i_A, i_C, path)
975                 i_B = i_C
976                 i_C += 1
977                 B = C
978
979         def improve_route(self):
980             '''
981             improve_route()
982
983             Melhora a rota usando o rubberband.
984             '''
985             shapely_polygons = self._shapely_polygons()
986
987             t0 = timer()
988
989             improved_path = self._best_path.copy()
990             self._improve_route_version_1(improved_path, shapely_polygons)
991             self._improved_path = improved_path
992
993             improved_path2 = self._best_path.copy()
994             self._improve_route_version_2(improved_path2, shapely_polygons)
995             self._improved_path2 = improved_path2
996
997             # terceira versão da rota melhorada (segunda + primeira)
998             improved_path3 = improved_path2.copy()
999             self._improve_route_version_1(improved_path3, shapely_polygons)
1000            self._improved_path3 = improved_path3
1001
1002            improved_value = 0.0
1003            improved_value2 = 0.0
1004            improved_value3 = 0.0
1005            P = improved_path[0]
1006            P2 = improved_path2[0]
1007            P3 = improved_path3[0]
1008            N = len(improved_path)
1009            for i in range(1, N):
1010                Q = improved_path[i]
1011                Q2 = improved_path2[i]
1012                Q3 = improved_path3[i]
1013                improved_value += geodist(P, Q)
1014                improved_value2 += geodist(P2, Q2)
1015                improved_value3 += geodist(P3, Q3)
1016                P = Q
1017                P2 = Q2
1018                P3 = Q3
1019            self._improved_value = improved_value
1020            self._improved_value2 = improved_value2
```

```
1021            self._improved_value3 = improved_value3
1022
1023            etime = timer() - t0
1024
1025            data = np.array(self._improved_path)
1026            data2 = np.array(self._improved_path2)
1027            data3 = np.array(self._improved_path3)
1028            self._improved_path_obj.set_data(data.T)
1029            self._improved_path2_obj.set_data(data2.T)
1030            self._improved_path3_obj.set_data(data3.T)
1031            return round(etime, 4)
1032
1033        def _intersect(self, P, Q, shapely_polygons, improving):
1034            '''
1035            Averigua se o rumo PQ intersecta algum dos polígonos.
1036            '''
1037            matplotlib_polygons = self.navegacao._polygons._list
1038            line = LineString((P, Q))
1039            for poly, shapely_poly in zip(matplotlib_polygons, shapely_polygons):
1040                if not improving and not poly.inarea(Q):
1041                    continue
1042                if not line.intersection(shapely_poly).is_empty:
1043                    return True
1044            return False
1045
1046        def reset(self, all=True):
1047            if all:
1048                self._best_path.clear()
1049                self._best_path_obj.set_xdata([])
1050                self._best_path_obj.set_ydata([])
1051            self._improved_path.clear()
1052            self._improved_path_obj.set_xdata([])
1053            self._improved_path_obj.set_ydata([])
1054            self._improved_path2.clear()
1055            self._improved_path2_obj.set_xdata([])
1056            self._improved_path2_obj.set_ydata([])
1057            self._improved_path3.clear()
1058            self._improved_path3_obj.set_xdata([])
1059            self._improved_path3_obj.set_ydata([])
1060
1061        def _shapely_polygons(self):
1062            '''
1063            Converte a lista matplotlib.Polygon numa lista shapely.Polygon.
1064            '''
1065            matplotlib_polygons = self.navegacao._polygons._list
1066            shapely_polygons = []
1067            for poly in matplotlib_polygons:
1068                shapely_polygons.append(Polygon_(poly._vertices))
1069            return shapely_polygons
1070
1071    #==========================================================================
1072    # MODULO PRINCIPAL DE NAVEGACAO
1073    #--------------------------------------------------------------------------
1074
1075    AREA_BORDER_COLOR = "magenta"
1076    AREA_BORDER_ALPHA = 0.8
1077
1078    MINOR_TICKS_COLOR = "red"
1079    MINOR_TICKS_ALPHA = 0.3
1080
1081    MAJOR_TICKS_COLOR = "blue"
1082    MAJOR_TICKS_ALPHA = 0.7
1083
1084    def default_values():
1085        lat = (48, 0, 49, 20)
1086        lon = (-6, 10, -4, 20)
1087        step = 1
1088        nticks = 10
```

```
1089        full_compass_rose = True
1090        return *lat, *lon, step, nticks, full_compass_rose
1091
1092    class SistemaNavegacao:
1093        '''
1094        SistemaNavegacao(root, tight_layout=True)
1095        '''
1096        def __init__(self, root, tight_layout=True):
1097            self.root = root
1098            self._bg = None
1099            self._map_obj = None
1100            self._area_border = None
1101
1102            self._area_panel(tight_layout)
1103            self._actions_panel()
1104
1105            values = default_values()
1106            self._put_values(values)
1107            self._set_values(values)
1108
1109            self._axes_configuration()
1110            # self._load_map(filename="mapa.jpg", draw_now=False)
1111
1112            self._nodes = NodosExtremos(self)
1113            self._current = Poligono(self)
1114            self._polygons = ListaPoligonos(self)
1115            self._route = RotaOptima(self)
1116
1117        def _area_panel(self, tight_layout):
1118            '''
1119            Define o painel que contém a área de navegação. Inclui um espaço para
1120            curtas mensagens (statusline) sobre acções realizadas em cada momento.
1121            '''
1122            area_frame = tk.Frame(self.root, border=1, relief=tk.SUNKEN)
1123            area_frame.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)
1124
1125            fig = Figure(tight_layout=tight_layout)
1126            ax = fig.add_subplot()
1127
1128            canvas = FigureCanvasTkAgg(fig, area_frame)
1129            canvas.get_tk_widget().pack(fill=tk.BOTH, expand=True)
1130
1131            canvas.mpl_connect("draw_event", self.on_draw)
1132
1133            canvas.mpl_connect('button_press_event', self.on_mouse_press)
1134            canvas.mpl_connect('button_release_event', self.on_mouse_release)
1135            canvas.mpl_connect('motion_notify_event', self.on_mouse_move)
1136
1137            canvas.mpl_connect("key_press_event", self.on_key_press)
1138
1139            toolbar = NavigationToolbar2Tk(canvas, area_frame)
1140            toolbar.update()
1141            toolbar.pack(side=tk.BOTTOM, fill=tk.X)
1142
1143            statusline = tk.Label(toolbar, text="")
1144            statusline.pack(anchor=tk.E)
1145
1146            self.ax = ax
1147            self.canvas = canvas
1148            self.statusline = statusline
1149
1150        def _actions_panel(self):
1151            '''
1152            Define o painel que permite interagir com a área de navegação e a
1153            aplicação em geral.
1154            '''
1155            actions_frame = tk.Frame(self.root)
1156            actions_frame.pack(padx=5, pady=20, side=tk.LEFT, fill=tk.Y)
```

```
1157
1158            ysmall = 5
1159            ybig = 4*ysmall
1160
1161            tk.Label(actions_frame, text="Limites Geográficos:").pack(anchor=tk.W)
1162
1163            limits_frame = tk.Frame(actions_frame)
1164            limits_frame.pack(anchor=tk.W)
1165
1166            lon_frame = tk.Frame(limits_frame)
1167            lon_frame.pack(anchor=tk.W)
1168            c = 0
1169            tk.Label(lon_frame, text="O-X: Lon").grid(row=0, column=c)
1170            c += 1
1171            lon_min_g = tk.Entry(lon_frame, width=4, justify=tk.RIGHT)
1172            lon_min_g.bind("<Return>", self.update_area_callback)
1173            lon_min_g.grid(row=0, column=c)
1174            c += 1
1175            tk.Label(lon_frame, text="°").grid(row=0, column=c)
1176            c += 1
1177            lon_min_m = tk.Entry(lon_frame, width=2, justify=tk.RIGHT)
1178            lon_min_m.bind("<Return>", self.update_area_callback)
1179            lon_min_m.grid(row=0, column=c)
1180            c += 1
1181            tk.Label(lon_frame, text="' to").grid(row=0, column=c)
1182            c += 1
1183            lon_max_g = tk.Entry(lon_frame, width=4, justify=tk.RIGHT)
1184            lon_max_g.bind("<Return>", self.update_area_callback)
1185            lon_max_g.grid(row=0, column=c)
1186            c += 1
1187            tk.Label(lon_frame, text="°").grid(row=0, column=c)
1188            c += 1
1189            lon_max_m = tk.Entry(lon_frame, width=2, justify=tk.RIGHT)
1190            lon_max_m.bind("<Return>", self.update_area_callback)
1191            lon_max_m.grid(row=0, column=c)
1192            c += 1
1193            tk.Label(lon_frame, text="'").grid(row=0, column=c)
1194
1195            lat_frame = tk.Frame(limits_frame)
1196            lat_frame.pack(anchor=tk.W)
1197            c = 0
1198            tk.Label(lat_frame, text="O-Y: Lat").grid(row=0, column=c)
1199            c += 1
1200            lat_min_g = tk.Entry(lat_frame, width=4, justify=tk.RIGHT)
1201            lat_min_g.bind("<Return>", self.update_area_callback)
1202            lat_min_g.grid(row=0, column=c)
1203            c += 1
1204            tk.Label(lat_frame, text="°").grid(row=0, column=c)
1205            c += 1
1206            lat_min_m = tk.Entry(lat_frame, width=2, justify=tk.RIGHT)
1207            lat_min_m.bind("<Return>", self.update_area_callback)
1208            lat_min_m.grid(row=0, column=c)
1209            c += 1
1210            tk.Label(lat_frame, text="' to").grid(row=0, column=c)
1211            c += 1
1212            lat_max_g = tk.Entry(lat_frame, width=4, justify=tk.RIGHT)
1213            lat_max_g.bind("<Return>", self.update_area_callback)
1214            lat_max_g.grid(row=0, column=c)
1215            c += 1
1216            tk.Label(lat_frame, text="°").grid(row=0, column=c)
1217            c += 1
1218            lat_max_m = tk.Entry(lat_frame, width=2, justify=tk.RIGHT)
1219            lat_max_m.bind("<Return>", self.update_area_callback)
1220            lat_max_m.grid(row=0, column=c)
1221            c += 1
1222            tk.Label(lat_frame, text="'").grid(row=0, column=c)
1223
1224            label = tk.Label(actions_frame, text="Discretização da Área:")
```

```
1225            label.pack(pady=(ybig,0), anchor=tk.W)
1226
1227            discretise_frame = tk.Frame(actions_frame)
1228            discretise_frame.pack(anchor=tk.W)
1229
1230            steps_frame = tk.Frame(discretise_frame)
1231            steps_frame.pack(anchor=tk.W)
1232            c = 0
1233            tk.Label(steps_frame, text="Step").grid(row=0, column=c)
1234            c += 1
1235            step = tk.Entry(steps_frame, width=4, justify=tk.RIGHT)
1236            step.bind("<Return>", self.update_area_callback)
1237            step.grid(row=0, column=c)
1238            c += 1
1239            tk.Label(steps_frame, text="'  Major nTicks").grid(row=0, column=c)
1240            c += 1
1241            nticks = tk.Entry(steps_frame, width=2, justify=tk.RIGHT)
1242            nticks.bind("<Return>", self.update_area_callback)
1243            nticks.grid(row=0, column=c)
1244
1245            var_full_compass_rose = tk.IntVar(actions_frame)
1246
1247            full_compass_rose = tk.Checkbutton(
1248                actions_frame,
1249                text="Incluir os Pontos Colaterais",
1250                variable=var_full_compass_rose, onvalue=1, offvalue=0,
1251            )
1252            full_compass_rose.pack(pady=(ybig,0))
1253
1254            insert_map = tk.Button(actions_frame, text="Inserir Mapa...")
1255            insert_map.config(command=self.insert_map_callback)
1256            insert_map.pack(pady=(ybig,0))
1257
1258            remove_map = tk.Button(actions_frame, text="Remover Mapa")
1259            remove_map.config(command=self.remove_map_callback)
1260            remove_map.pack(pady=(ysmall,0))
1261
1262            load_config = tk.Button(
1263                actions_frame, text="Ler Zonas Não Navegáveis...",
1264            )
1265            load_config.config(command=self.load_config_callback)
1266            load_config.pack(pady=(ybig,0))
1267
1268            save_config = tk.Button(actions_frame, text="Gravar Como...")
1269            save_config.config(command=self.save_config_callback)
1270            save_config.pack(pady=(ysmall,0))
1271
1272            compute_route = tk.Button(actions_frame, text="Determinar a Rota Ótima")
1273            compute_route.config(command=self.compute_route_callback)
1274            compute_route.pack(pady=(ybig,0))
1275
1276            improve_route = tk.Button(actions_frame, text="Aplicar Rubberband")
1277            improve_route.config(command=self.improve_route_callback)
1278            improve_route.pack(pady=(ysmall,0))
1279
1280            clear_area = tk.Button(actions_frame, text="Apagar Tudo")
1281            clear_area.config(command=self.clear_area_callback)
1282            clear_area.pack(pady=(ybig,0))
1283
1284            quit = tk.Button(actions_frame, text="Sair", command=self.quit_callback)
1285            quit.pack(pady=(ybig,0), side=tk.BOTTOM)
1286
1287            self.widget_values = (
1288                lat_min_g, lat_min_m, lat_max_g, lat_max_m,
1289                lon_min_g, lon_min_m, lon_max_g, lon_max_m,
1290                step,
1291                nticks,
1292                var_full_compass_rose,
```

```
1293            )
1294            self.full_compass_rose_check = full_compass_rose
1295
1296        def _axes_configuration(self):
1297            '''
1298            Configura o sistema de eixos.
1299            '''
1300            # valores de configuração, em minutos
1301            xmin_ = 60*self.lon[0] + (-1 if self.lon[0] < 0 else 1) * self.lon[1]
1302            xmax_ = 60*self.lon[2] + (-1 if self.lon[2] < 0 else 1) * self.lon[3]
1303            ymin_ = 60*self.lat[0] + (-1 if self.lat[0] < 0 else 1) * self.lat[1]
1304            ymax_ = 60*self.lat[2] + (-1 if self.lat[2] < 0 else 1) * self.lat[3]
1305            step = self.step
1306            nticks = self.nticks
1307            # ... em graus decimais
1308            fstep = step/60.0
1309            fstep1 = 1.0/60.0
1310            radius = 0.5/60.0
1311            fnsteps = step*nticks/60.0
1312
1313            # grelha de pontos
1314            xrange = xmax_ - xmin_
1315            yrange = ymax_ - ymin_
1316            xq = math.trunc((xrange+1) / step)
1317            yq = math.trunc((yrange+1) / step)
1318            xr = (xrange+1) - step*xq
1319            yr = (yrange+1) - step*yq
1320            nrows = yq + int(yr > 1e-16)
1321            ncols = xq + int(xr > 1e-16)
1322            # ... em graus decimais
1323            xoffset = 0.5 * (xrange - step*(ncols-1)) / 60.0
1324            yoffset = 0.5 * (yrange - step*(nrows-1)) / 60.0
1325            xmin_ /= 60.0
1326            xmax_ /= 60.0
1327            ymin_ /= 60.0
1328            ymax_ /= 60.0
1329            minor_xticks = np.arange(xmin_+xoffset, xmax_+xoffset+fstep, fstep)
1330            minor_yticks = np.arange(ymin_+yoffset, ymax_+yoffset+fstep, fstep)
1331            major_xticks = np.arange(xmin_+xoffset, xmax_+xoffset+fstep, fnsteps)
1332            major_yticks = np.arange(ymin_+yoffset, ymax_+yoffset+fstep, fnsteps)
1333
1334            # limites da área de navegação ajustados
1335            xmin = minor_xticks[0]
1336            ymin = minor_yticks[0]
1337            xmax = minor_xticks[ncols-1]
1338            ymax = minor_yticks[nrows-1]
1339
1340            # limites dos eixos (inclui margem)
1341            xlim = [xmin_-fstep1, xmax_+fstep1]
1342            ylim = [ymin_-fstep1, ymax_+fstep1]
1343
1344            # configuração dos eixos e definição da grelha
1345            self.ax.set_aspect('equal')
1346            self.ax.set_xlim(xlim)
1347            self.ax.set_ylim(ylim)
1348            self.ax.set_xticks(major_xticks)
1349            self.ax.set_xticks(minor_xticks, minor=True)
1350            self.ax.set_yticks(major_yticks)
1351            self.ax.set_yticks(minor_yticks, minor=True)
1352            self.ax.grid(
1353                which="minor",
1354                linestyle="--",
1355                linewidth=0.5, color=MINOR_TICKS_COLOR, alpha=MINOR_TICKS_ALPHA,
1356            )
1357            self.ax.grid(
1358                which="major",
1359                linestyle="--",
1360                linewidth=0.75, color=MAJOR_TICKS_COLOR, alpha=MAJOR_TICKS_ALPHA,
```

```
1361            )
1362
1363            # omite ticks em ambos os eixos
1364            for pos in ["top", "bottom", "left", "right"]:
1365                self.ax.spines[pos].set_visible(False)
1366            self.ax.tick_params(axis="both", which="both", length=0)
1367
1368            # borda da área de navegação
1369            if self._area_border is None:
1370                top = Line2D(
1371                    xlim, (ymax, ymax),
1372                    linewidth=1.5, color=AREA_BORDER_COLOR, alpha=AREA_BORDER_ALPHA,
1373                )
1374                bottom = Line2D(
1375                    xlim, (ymin, ymin),
1376                    linewidth=1.5, color=AREA_BORDER_COLOR, alpha=AREA_BORDER_ALPHA,
1377                )
1378                left = Line2D(
1379                    (xmin, xmin), ylim,
1380                    linewidth=1.5, color=AREA_BORDER_COLOR, alpha=AREA_BORDER_ALPHA,
1381                )
1382                right = Line2D(
1383                    (xmax, xmax), ylim,
1384                    linewidth=1.5, color=AREA_BORDER_COLOR, alpha=AREA_BORDER_ALPHA,
1385                )
1386                self.ax.add_line(top)
1387                self.ax.add_line(bottom)
1388                self.ax.add_line(left)
1389                self.ax.add_line(right)
1390                self._area_border = (top, bottom, left, right)
1391            else:
1392                top, bottom, left, right = self._area_border
1393                top.set_xdata(xlim)
1394                top.set_ydata((ymax, ymax))
1395                bottom.set_xdata(xlim)
1396                bottom.set_ydata((ymin, ymin))
1397                left.set_xdata((xmin, xmin))
1398                left.set_ydata(ylim)
1399                right.set_xdata((xmax, xmax))
1400                right.set_ydata(ylim)
1401
1402            self.xmin, self.xmax = xmin, xmax
1403            self.ymin, self.ymax = ymin, ymax
1404            self.nrows, self.ncols = nrows, ncols
1405            self.fstep = fstep
1406            self.radius = radius
1407            self.extent = [xmin_, xmax_, ymin_, ymax_]
1408
1409        def showmessage(self, text="", color="black"):
1410            '''
1411            showmessage(text="", color="black")
1412
1413            Mostra mensagem na statusline.
1414            '''
1415            self.statusline.config(text=text.rjust(40), fg=color)
1416            self.statusline.update_idletasks()
1417
1418        def ij_from_xy(self, x, y):
1419            '''
1420            i, j = ij_from_xy(x, y)
1421
1422            Converte coordenadas geográficas (x, y) em entradas (i, j).
1423            '''
1424            step, ny = self.fstep, self.nrows-1
1425            i = ny - (round((y-self.ymin-step)/step) + 1)
1426            j = round((x-self.xmin-step)/step) + 1
1427            return i, j
1428
```

```
1429        def xy_from_ij(self, i, j):
1430            '''
1431            x, y = xy_from_ij(i, j)
1432
1433            Converte entradas (i, j) em coordenadas geográficas (x, y).
1434            '''
1435            step, ny = self.fstep, self.nrows-1
1436            x = self.xmin + j*step
1437            y = self.ymin + (ny-i)*step
1438            return x, y
1439
1440        def _get_values(self):
1441            '''
1442            Retorna os valores de configuração contidos nos widgets.
1443            '''
1444            widget_values = self.widget_values
1445            try:
1446                widget_values_ = widget_values[:-1]
1447                values = list(int(widget.get()) for widget in widget_values_[:8])
1448                values.append(float(widget_values_[8].get()))
1449                values.append(float(widget_values_[9].get()))
1450                # values = list(int(widget.get()) for widget in widget_values_)
1451            except Exception as err:
1452                tk.messagebox.showerror(title="Erro", message=err)
1453                return None
1454            var_full_compass_rose = widget_values[len(widget_values_)]
1455            values.append(var_full_compass_rose.get() == 1)
1456            return tuple(values)
1457
1458        def _put_values(self, values):
1459            '''
1460            Coloca os valores de configuração nos widgets.
1461            '''
1462            values_ = values[:-1]
1463            last_value = values[len(values_)]
1464            for value, widget in zip(values_, self.widget_values):
1465                widget.insert(0, str(value))
1466            if last_value:
1467                self.full_compass_rose_check.select()
1468            else:
1469                self.full_compass_rose_check.deselect()
1470
1471        def _set_values(self, values, changed=True):
1472            '''
1473            Atribui os valores de configuração dados às variáveis correspondentes.
1474            '''
1475            if changed:
1476                self.lat = values[:4]
1477                self.lon = values[4:8]
1478                self.step = values[8]
1479                self.nticks = values[9]
1480            self.full_compass_rose = values[10]
1481
1482        def _draw_animated(self):
1483            self._polygons.draw()
1484            self._current.draw()
1485            self._nodes.draw()
1486            self._route.draw()
1487
1488        def _load_map(self, filename, draw_now):
1489            '''
1490            load_map(filename, draw_now)
1491
1492            Lê mapa a partir de um ficheiro.
1493            '''
1494            self.showmessage("Lendo mapa...", "red")
1495            try:
1496                map_data = plt.imread(filename)
```

```
1497            except Exception as err:
1498                tk.messagebox.showerror(title="Erro", message=err)
1499                self.showmessage()
1500                return
1501            if self._map_obj is not None:
1502                self._map_obj.remove()
1503            map_obj = self.ax.imshow(map_data, extent=self.extent, alpha=0.8)
1504            self._map_obj = map_obj
1505            if draw_now:
1506                self.canvas.draw()
1507                self.canvas.flush_events()
1508            self.showmessage()
1509
1510        def redraw_canvas(self):
1511            self.canvas.restore_region(self._bg)
1512            self._draw_animated()
1513            self.canvas.blit(self.canvas.figure.bbox)
1514            self.canvas.flush_events()
1515
1516        def clear_area_callback(self):
1517            '''
1518            clear_area_callback()
1519
1520            Apagar tudo da área de navegação.
1521            '''
1522            self.showmessage()
1523            self._route.reset()
1524            self._polygons.reset()
1525            self._current.reset()
1526            self.redraw_canvas()
1527
1528        def compute_route_callback(self):
1529            '''
1530            compute_route_callback()
1531
1532            Determinar a rota óptima.
1533            '''
1534            self._route.reset()
1535            self.redraw_canvas()
1536            # permite ter os valores mais actuais antes de calcular a rota
1537            if not self.update_area_callback():
1538                return
1539            self.showmessage("Calculando a rota ótima...", "red")
1540            etime, ok = self._route.compute_best_route()
1541            if ok:
1542                self.redraw_canvas()
1543                best_value = round(self._route._best_value, 3)
1544                self.showmessage(
1545                    f"Rota ótima: {best_value} milhas, em {etime} sec", "blue",
1546                )
1547
1548        def improve_route_callback(self):
1549            '''
1550            improve_route_callback()
1551
1552            Aplicar o rubber band para melhorar a solução.
1553            '''
1554            self.showmessage()
1555            self._route.reset(all=False)
1556            self.redraw_canvas()
1557            self.showmessage("Aplicando rubberband...", "red")
1558            etime = self._route.improve_route()
1559            self.redraw_canvas()
1560            bvalue = round(self._route._best_value, 3)
1561            ivalue = round(self._route._improved_value, 3)
1562            ivalue2 = round(self._route._improved_value2, 3)
1563            ivalue3 = round(self._route._improved_value3, 3)
1564            s = f"{bvalue}->{ivalue}->{ivalue2}->{ivalue3}"
```

```
1565            self.showmessage(f"Melhoria: {s} milhas, em {etime} sec", "blue")
1566
1567        def insert_map_callback(self):
1568            '''
1569            insert_map_callback()
1570
1571            Inserir mapa no sistema de eixos.
1572            '''
1573            self.showmessage()
1574            files = [
1575                ("JPG file", "*.jpg"),
1576                ("PNG file", "*.pnj"),
1577                ("All Files", "*.*"),
1578                ]
1579            file = askopenfile(
1580                filetypes=files, defaultextension=files, initialfile="mapa.jpg",
1581            )
1582            if file is not None:
1583                self._load_map(filename=file.name, draw_now=True)
1584
1585        def load_config_callback(self):
1586            '''
1587            load_config_callback()
1588
1589            Ler zonas não navegáveis a partir de um ficheiro.
1590            '''
1591            self.showmessage()
1592            files = [
1593                ("Polygons file", "*.pol"),
1594                ("All Files", "*.*"),
1595                ]
1596            file = askopenfile(
1597                filetypes=files, defaultextension=files, initialdir="config",
1598            )
1599            if file is None:
1600                return
1601            self.clear_area_callback()
1602            self.showmessage("Lendo ficheiro...", "red")
1603            with open(file.name, "r") as f:
1604                lines = f.readlines()
1605            f.close()
1606            v = lines[0].split()
1607            self._nodes._S[0].center = (float(v[0]), float(v[1]))
1608            self._nodes._T[0].center = (float(v[2]), float(v[3]))
1609            for line in lines[1:]:
1610                line = line.split()
1611                poly = Poligono(self)
1612                for i in range(0, len(line), 2):
1613                    v = (float(line[i]), float(line[i+1]))
1614                    poly._vertices.append(v)
1615                poly.set_artists()
1616                poly.set_finalised(True)
1617                self._polygons.append(poly)
1618            self.redraw_canvas()
1619            self.showmessage()
1620
1621        def quit_callback(self):
1622            '''
1623            quit_callback()
1624
1625            Sair da aplicação.
1626            '''
1627            self.root.destroy()
1628
1629        def remove_map_callback(self):
1630            '''
1631            remove_map_callback()
1632
```

```
1633            Remove mapa do sistema de eixos.
1634            '''
1635            self.showmessage()
1636            if self._map_obj is not None:
1637                self._map_obj.remove()
1638                self._map_obj = None
1639                self.canvas.draw()
1640                self.canvas.flush_events()
1641
1642        def save_config_callback(self):
1643            '''
1644            save_config_callback()
1645
1646            Grava as zonas não navegáveis num ficheiro.
1647            '''
1648            self.showmessage()
1649            if self._polygons.empty():
1650                msg = "A área de navegação não contém nenhuma zona excluída."
1651                tk.messagebox.showinfo(title="Informação", message=msg)
1652                return
1653            files = [
1654                ("Polygons file", "*.pol"),
1655                ("All Files", "*.*"),
1656                ]
1657            file = asksaveasfile(
1658                filetypes=files, defaultextension=files, initialdir="config",
1659            )
1660            if file is None:
1661                return
1662            self.showmessage("Gravando ficheiro...", "red")
1663            s = self._nodes._S[0].center
1664            t = self._nodes._T[0].center
1665            with open(file.name, "w") as f:
1666                f.write(f"{s[0]} {s[1]} {t[0]} {t[1]}\n")
1667                for poly in self._polygons._list:
1668                    V = list(f"{v[0]} {v[1]}" for v in poly._vertices)
1669                    line = ' '.join(v for v in V)
1670                    f.write(f"{line}\n")
1671            f.close()
1672            self.showmessage()
1673
1674        def update_area_callback(self, *args):
1675            '''
1676            bool = update_area_callback(*args)
1677
1678            Actualiza a área de navegação, com todos os objectos lá incluídos.
1679            Retorna True se não houver erros. Caso contrário, False.
1680            '''
1681            self.showmessage()
1682            new_values = self._get_values()
1683            if new_values is None:
1684                return False
1685            old_values_ = (*self.lat, *self.lon, self.step, self.nticks)
1686            changed = any(new != old for new, old in zip(new_values, old_values_))
1687            self._set_values(new_values, changed=changed)
1688            if changed:
1689                self._axes_configuration()
1690                self._polygons.update_interior_points()
1691                self.canvas.draw()
1692                self.canvas.flush_events()
1693            else:
1694                self.showmessage("Nenhuma alteração efectuada!", "green")
1695            return True
1696
1697        def on_draw(self, event):
1698            if event is not None:
1699                if event.canvas != self.canvas:
1700                    raise RuntimeError
```

```
1701            self._bg = self.canvas.copy_from_bbox(self.canvas.figure.bbox)
1702            self._draw_animated()
1703
1704        def on_key_press(self, event):
1705            if event.key == "escape":
1706                if self._current.under_construction():
1707                    self._current.shrink()
1708                elif not self._polygons.deselect():
1709                    return
1710                self.redraw_canvas()
1711            elif event.key == "delete":
1712                if self._current.under_construction():
1713                    self._current.reset()
1714                elif not self._polygons.remove_selected():
1715                    return
1716                self.redraw_canvas()
1717
1718        def on_mouse_move(self, event):
1719            if event.inaxes != self.ax:
1720                return
1721            P = (event.xdata, event.ydata)
1722            if self._nodes.active():
1723                self._nodes.move_active_node_to(P)
1724            elif self._polygons.changing():
1725                self._polygons.move_changing_vertex_to(P)
1726            elif self._current.under_construction():
1727                self._current.track_mouse_position(P)
1728            else: return
1729            self.redraw_canvas()
1730
1731        def on_mouse_press(self, event):
1732            if event.inaxes != self.ax:
1733                return
1734            P = (event.xdata, event.ydata)
1735            if event.button is MouseButton.LEFT:
1736                if self._current.under_construction():
1737                    self._current.append_vertex(P)
1738                    if self._current.finalised():
1739                        if self._polygons.append(self._current):
1740                            self._current = Poligono(self)
1741                    elif self._current.vertex_ignored():
1742                        return
1743                    self.redraw_canvas()
1744                    return
1745                node = self._nodes.which_selected(P)
1746                if node is not None:
1747                    self._nodes.activate(node)
1748                    self.redraw_canvas()
1749                    return
1750                poly = self._polygons.which_contains_vertex(P)
1751                if poly is not None:
1752                    self._polygons.start_changing(poly)
1753                    self.redraw_canvas()
1754                    return
1755                poly = self._polygons.which_contains_point(P)
1756                if poly is not None:
1757                    poly.set_selected(not poly.selected())
1758                    self.redraw_canvas()
1759                    return
1760                self._current.append_vertex(P)
1761                if self._current.finalised():
1762                    if self._polygons.append(self._current):
1763                        self._current = Poligono(self)
1764                elif self._current.vertex_ignored():
1765                    return
1766                self.redraw_canvas()
1767            elif event.button is MouseButton.RIGHT:
1768                if self._nodes.which_selected(P) is not None:
```

```
1769                    self._nodes.swap()
1770                    self.redraw_canvas()
1771
1772
1773        def on_mouse_release(self, event):
1774            if event.button is MouseButton.LEFT:
1775                if self._nodes.active():
1776                    self._nodes.deactivate()
1777                elif self._polygons.changing():
1778                    self._polygons.stop_changing()
1779                else: return
1780                self.redraw_canvas()
1781
1782    #================================================================================
1783    # MAIN
1784    #--------------------------------------------------------------------------------
1785
1786    # cria a janela principal
1787    root = tk.Tk()
1788    # esconde a janela principal para a sua configuração
1789    root.withdraw()
1790    # root.iconbitmap("icon.ico")
1791    root.title("Automatização de um Sistema de Navegação Marítima")
1792    root.option_add("*Font", "courier 10")
1793
1794    # dimensiona e posiciona a janela principal no ecran
1795    screen_width = root.winfo_screenwidth()
1796    screen_height = root.winfo_screenheight()
1797    scale = 0.85
1798    width = int(scale*screen_width)
1799    height = int(scale*screen_height)
1800    xpos = min(50, screen_width//2 - width//2)
1801    ypos = min(5, screen_height//2 - height//2)
1802    root.geometry(f"{width}x{height}+{xpos}+{ypos}")
1803
1804    app = SistemaNavegacao(root, tight_layout=True)
1805    root.protocol("WM_DELETE_WINDOW", app.quit_callback)
1806
1807    # mostra a janela principal
1808    root.deiconify()
1809    root.mainloop()
```