UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CIÊNCIA DA COMPUTAÇÃO

Nícolas Goeldner

Da segmentação semântica ao planejamento de movimentação em veículo autônomo

Florianópolis
2022

Nícolas Goeldner

Da segmentação semântica ao planejamento de movimentação em veículo autônomo

Este Trabalho de Conclusão de Curso foi julgado adequado para obtenção do Título de Bacharel em Ciência da Computação e aprovado em sua forma final pelo curso de Graduação em Ciência da Computação.

Florianópolis, 17 de Março de 2022.

Prof. Jean Everson Martina, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Aldo von Wangenheim, Dr. rer.nat.
Orientador
Universidade Federal de Santa Catarina

Prof. Roberto Simoni, Dr.
Coorientador
Universidade Federal de Santa Catarina

Prof. Mateus Grellert, Dr.
Avaliador
Universidade Federal de Santa Catarina

Prof. Jônata Tyska Carvalho, Dr.
Avaliador
Universidade Federal de Santa Catarina

Nícolas Goeldner

Da segmentação semântica ao planejamento de movimentação em veículo autônomo

Trabalho de Conclusão de Curso submetido ao Curso de Graduação em Ciência da Computação do Centro Tecnológico da Universidade Federal de Santa Catarina como requisito para obtenção do título de Bacharel em Ciência da Computação.
Orientador: Prof. Aldo von Wangenheim, Dr. rer.nat.
Coorientador: Prof. Roberto Simoni, Dr.

Florianópolis
2022

AGRADECIMENTOS

Muito obrigado.
Espero que todas as pessoas que me ajudaram saibam o quão especiais são para mim.

RESUMO

Veículos autônomos têm recebido muita atenção ultimamente. Esses veículos são compostos por um sistema de navegação autônoma, o qual possui, em diversas ocasiões, um componente específico de planejamento de movimentação. Assim, neste trabalho, propomos um estudo de sistemas de navegação autônoma e de métodos de planejamento de movimentação. Como parte do estudo, desenvolvemos a arquitetura do sistema de navegação autônoma e alguns módulos de um veículo autônomo chamado de Crawler. Apresentamos, detalhadamente, parte do módulo de Percepção, o módulo de Decisão de Comportamento e o módulo de Planejamento de Movimentação. Tentamos trazer ao máximo as motivações para cada decisão tomada durante o desenvolvimento dos módulos. O Capítulo 1 é uma breve introdução ao trabalho. O Capítulo 2 discorre sobre alguns conceitos básicos para o entendimento do que foi desenvolvido. O Capítulo 3 apresenta o Crawler. O Capítulo 4 explica a definição de goal states para o Crawler. O Capítulo 5 aborda o planejamento de movimentação do Crawler. Por fim, o Capítulo 6 discute alguns últimos detalhes e encerra o trabalho.

Palavras-chave: Veículo Autônomo. Sistema de Navegação Autônoma. Planejamento de Movimentação.

ABSTRACT

Autonomous vehicles have been receiving a lot of attention for the past years. These vehicles are composed of an autonomous navigation system, which has, on several occasions, a specific motion planning component. So, in this work, we propose a study on autonomous navigation systems and motion planning methods. As part of this study, we have developed the autonomous navigation system architecture and some modules of an autonomous vehicle called Crawler. We present in details parts of the Perception module, the Behavioural Decision module and the Motion Planning module. We try as much as possible to discuss the reasons why each decision was taken during the development of the modules. Chapter 1 is a brief introduction to the work. Chapter 2 discusses some basic concepts for understanding what was developed. Chapter 3 introduces the Crawler. Chapter 4 explains the definition of goal states for the Crawler. Chapter 5 covers the motion planning for the Crawler. Finally, Chapter 6 discusses some final details and concludes the work.

Key-words: Autonomous Vehicles. Autonomous Navigation System. Motion Planning.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS E SIGLAS

SNA	Sistema de Navegação Autônoma
PM	Planejamento de Movimentação
DC	Decisão de Comportamento
ODD	Operational Design Domain
OG	Occupancy Grid
PMP	Piano Mover's Problem
BVP	Boundary Value Problem
MPUDCP	Motion Planning Under Differential Constraints Problem
ROS	Robot Operating System
ZS	Método de esqueletonização de (ZHANG; SUEN, 1984)
GH	Método de esqueletonização de (GUO; HALL, 1989)
MA	Medial Axis - método de esqueletonização implementado em scikit-image
BFS	Breadth-First Search
DFS	Depth-First Search
OMPL	Open Motion Planning Library
RRT	Rapidly-Exploring Random Trees
RRTg	RRT geométrico
RRTc	RRT baseado em controle
SST	Stable Sparse RRT

SUMÁRIO

1 INTRODUÇÃO

A navegação autônoma é uma área que vem ganhando muita atenção nos últimos anos. Um dos maiores eventos que impulsionou o desenvolvimento de veículos autônomos foi uma competição chamada DARPA Urban Challenge (DUC), realizada em 2007. Nessa competição, os veículos deveriam trafegar por 96km sem nenhuma intervenção humana. Apenas seis times conseguiram desenvolver sistemas de navegação autônoma (SNA) que guiaram seus respectivos automóveis até a linha de chegada sem descumprir nenhuma das regras da competição. Outros grandes expoentes para a navegação autônoma são as empresas Tesla, Waymo, Apple, Baidu e várias outras que possuem projetos de veículos autônomos, sendo que muitas dessas empresas já possuem veículos trafegando por algumas cidades para realizar testes. Há inclusive projetos grandes, como o Apollo[1] da Baidu, que disponibilizam grande parte do que desenvolvem publicamente. A navegação autônoma pode nos trazer diversas vantagens, redução dos números de acidentes e congestionamentos de trânsito, melhora na qualidade de vida e diminuição da emissão poluentes são apenas algumas. Porém, ainda há vários problemas a serem resolvidos se quisermos desfrutar dessas vantagens totalmente. Vamos discutir um pouco mais sobre SNAs para que possamos perceber os diversos problemas inerentes à navegação autônoma voltada a automóveis.

Uma organização comum para os componentes de um SNA é a divisão em módulos sensoriais e módulos de planejamento e controle. Os módulos sensoriais são responsáveis pela percepção do ambiente e, também, pela identificação do estado do veículo em relação ao ambiente em que ele está inserido. Já os módulos de planejamento e controle, são responsáveis por, a partir das informações obtidas pelos módulos sensoriais, planejar a movimentação e executá-la para que o veículo chegue ao seu destino. De acordo com (LIU et al., 2017), podemos dividir essas últimas responsabilidades em cinco módulos: (i) Predição do Tráfego, (ii) Roteamento, (iii) Decisão de Comportamento, (iv) Planejamento de Movimentação e (v) Controle. A Predição do Tráfego é responsável por analisar as percepções do ambiente para prever os caminhos pelos quais outros veículos, pedestres, ciclistas e vários outros objetos móveis provavelmente irão passar e quando passarão por determinado ponto. O Roteamento cuida da rota que um veículo deve fazer para chegar ao seu destino final, por exemplo, por quais ruas deve passar. Já a Decisão de Comportamento é responsável por determinar como o veículo deve se comportar para que siga a rota dada pelo módulo anterior, por exemplo, trocar de faixa, continuar na mesma faixa, esperar o semáforo abrir, etc. O Planejamento de Movimentação determina, dentre algumas características, por quais pontos da pista o veículo deve passar e uma velocidade, aceleração e orientação para cada ponto de forma que respeite o comportamento escolhido. Por último, o Controle é responsável por traduzir as informações de velocidade, aceleração e orientação em comandos que devem ser dados aos atuadores do veículo.

Todas essas responsabilidades, que foram divididas e atribuídas a módulos de um

[1] Link para o website da Apollo: https://apollo.auto

sistema, são vitais para a navegação autônoma. Entretanto, neste trabalho, focaremos em um módulo específico: Planejamento de Movimentação (PM). O principal desafio que deve ser resolvido pelo PM é a construção de uma trajetória sem colisões (contra objetos fixos e dinâmicos), fisicamente realizável, confortável, segura, dentre várias outras qualidades. Além disso, o PM deve considerar questões temporais, visto que um veículo necessita constantemente de uma trajetória durante a navegação. Abordaremos questões dos outros módulos também, em especial, do módulo de Decisão de Comportamento, pois possui uma interface mais forte com o Planejamento de Movimentação e conforme desenvolvemos este trabalho, nos deparamos com um problema muito interessante relacionado ao módulo de Decisão de Comportamento. Assim, apresentamos este trabalho como o resultado de parte de um estudo de SNAs com foco no módulo de Planejamento de Movimentação.

Para que possamos compreender e avaliar módulos de PM e SNAs minuciosamente, precisamos entender algumas decisões que foram tomadas durante o desenvolvimento do sistema. Portanto, ter alguma experiência em modelagem e implementação desses sistemas certamente pode ajudar a tornar visíveis algumas dificuldades e questões levantadas por outras pessoas ao desenvolverem seus sistemas. Por isso, decidimos desenvolver nosso próprio veículo autônomo, o Crawler - projeto o qual possui múltiplos integrantes. Como parte deste trabalho, desenvolvemos vários componentes do Crawler (falaremos, exatamente, quais mais adiante), porém vale salientar que escolhemos detalhar bem mais o processo a partir da obtenção da imagem segmentada (resultado da segmentação semântica) até a geração de uma trajetória pelo módulo de Planejamento de Movimentação. Tivemos que cuidar de outros módulos, apesar de o nosso foco inicial ser no módulo de PM, pois para que pudéssemos abordar devidamente o módulo de PM, precisávamos de resultados de outras partes do SNA.

No próximo capítulo, discutimos uma arquitetura usual de SNAs com detalhes de cada módulo e introduzimos o problema de planejamento de movimentação. Em seguida, abordamos o Crawler trazendo detalhes de hardware, especificidades da organização de seu SNA e alguns detalhes de componentes essenciais para a navegação que, entretanto, não possuem muito espaço no restante deste trabalho. No quarto capítulo, apresentamos o problema da definição de goal states considerando algumas especificidades do Crawler; trazemos, também, a solução proposta e experimentos realizados. A solução compõe o módulo de Decisão de Comportamento e parte do módulo de Percepção. No quinto capítulo, voltamos ao planejamento de movimentação, porém, agora, no contexto do Crawler. No sexto e último capítulo, apresentamos alguns detalhes e resultados da integração do que foi desenvolvido ao SNA; esclarecemos o que já foi feito para o projeto do Crawler e abordamos diversas melhorias que podem ser feitas; por fim, concluímos. O código desenvolvido para o trabalho pode ser encontrado em https://codigos.ufsc.br/nicolas.goeldner/Crawler-SNA ou em https://github.com/ngoeldner/Crawler-SNA. Haverá um aviso indicando qual repositório está sendo mantido.

2 CONCEITOS BÁSICOS

Começamos apresentando alguns conceitos gerais relacionados com a navegação autônoma.

Níveis de Automação de Direção A Society of Automotive Engineers (SAE) publicou um documento (SAE J3016) definindo seis níveis de automação de direção para automóveis. No nível 0, há apenas avisos ao motorista, o veículo não toma nenhuma ação. No nível 1, o veículo pode auxiliar o motorista com a direção, por exemplo, mantendo o veículo centralizado na sua faixa ou pode ajudar com a aceleração/frenagem. Se houver essas duas características concomitantes, passamos para o nível 2. No nível 3, o veículo pode navegar sem alguém dirigir em algumas situações, porém deve haver alguém pronto para dirigir quando requisitado pelo veículo. Já no nível 4 não precisa haver um motorista, porém as condições de navegação do veículo são limitadas; por exemplo, o veículo pode estar limitado a não navegar com neve ou limitado a navegar apenas em certas áreas. Por último, no nível 5 o veículo não possui limites de funcionamento, como haver condições climáticas desfavoráveis ou estar em ambiente não urbano. A Figura 1 apresenta e diferencia os níveis em detalhes.

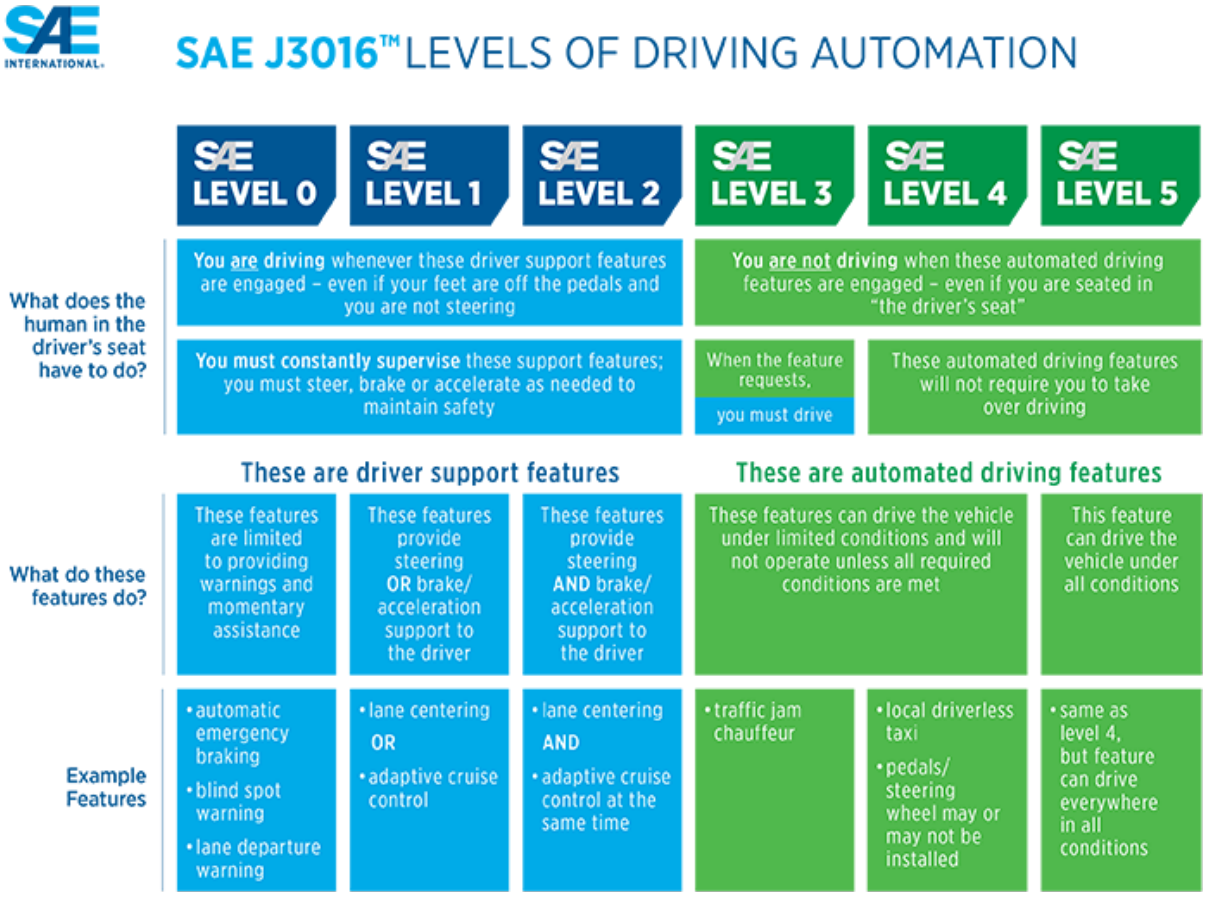


Figura 1 – Níveis de Automação de Direção. Fonte: sae.org

Operational Design Domain (ODD) De acordo com (TRANSPORTATION; ADMINISTRATION, 2016), em tradução livre, um Operational Design Domain é uma descrição de um

domínio de operação específico, para o qual uma função ou sistema é projetado para operar apropriadamente. Nessa descrição pode-se incluir tipos de estradas, limites de velocidade, condições ambientais (tempo, dia/noite, etc), dentre várias outras restrições de domínio.

Veículo Autônomo Ao longo deste trabalho, ao nos referirmos a veículos autônomos, estamos trazendo o termo mais em direção a automóveis que podem navegar sem intervenção humana em determinado ODD, ou seja, estamos nos referindo a automóveis com nível 4 de autonomia.

Sistema de Navegação Autônomo (SNA) Um sistema de navegação autônoma nesse contexto, passa a ser o hardware e o software responsáveis por tornar a navegação automatizada o suficiente a ponto de termos um veículo autônomo.

Via, pista e faixa de trânsito No contexto deste trabalho, abordaremos vias como toda parte da rua dedicada ao deslocamento de veículos, pessoas, bicicletas, etc. Já pista está relacionada a parte da via dedicada ao deslocamento de apenas veículos como carros e motos. Por último, as faixas são subdivisões das pistas com largura suficiente para o deslocamento dos veículos. A Figura 2 demonstra esses três conceitos. Vale notar que consideremos que há mais de uma pista quando houver uma bifurcação, entroncamento, cruzamento, divisão física da área navegável ou qualquer situação em que há mais de uma rota para o veículo.

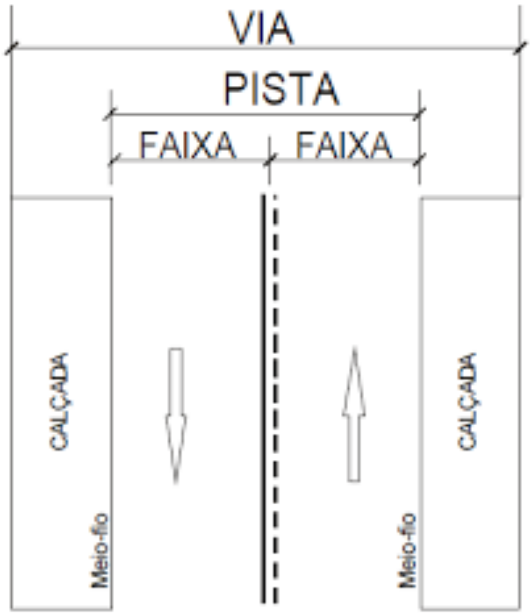


Figura 2 – Via, pista e faixa de trânsito. Fonte: Fiscalização de Trânsito.

Roll, Pitch e Yaw Usamos esses termos para indicar qual eixo do veículo ou da câmera, por exemplo, estamos abordando. A Figura 3 ajuda a indicar com qual eixo cada um desses termos está relacionado.

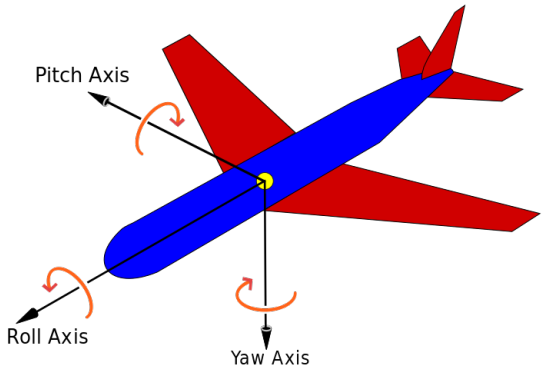


Figura 3 – Roll, pitch e yaw de um avião. Fonte: Wikipedia.

2.1 ARQUITETURA USUAL DE SNAS

Os veículos que competiram no DARPA Urban Challenge influenciaram o desenvolvimento de diversos outros veículos autônomos. Uma das principais influências é a arquitetura do SNA. A arquitetura utilizada por vários dos veículos desenvolvidos para o DARPA Urban Challenge ((MONTEMERLO et al., 2009) e (REINHOLTZ et al., 2009), por exemplo) pode ser representada pela Figura 4. Um dos veículos que foi influenciado pelos veículos do DARPA Urban Challenge, com reflexos inclusive na sua arquitetura, é o Bertha da Mercedes Benz (ZIEGLER et al., 2014). Muitas vantagens advêm da arquitetura da Figura 4, dentre elas estão: (i) Modularidade: a divisão do problema da navegação autônoma em diversos módulos facilita o desenvolvimento de soluções, pois quebra o problema em subproblemas menos complexos; acelera o desenvolvimento, pois permite uma paralelização do trabalho; incentiva que testes sejam feitos em módulos individuais, por exemplo, podemos testar diversos módulos para resolver o problema de planejamento de movimentação apenas substituindo um módulo por outro (desde que as entradas e as saídas sejam as mesmas). (ii) A quebra do problema de navegação autônoma em subproblemas já explorados em outras áreas, como exemplo podemos citar a detecção de objetos (placas, semáforos, carros, etc), que é um grande segmento da área de visão computacional; outro exemplo é a necessidade do veículo de ter uma boa capacidade de localizar-se, segmento muito explorado pela robótica. Podemos listar diversas outras vantagens ligadas à modularidade e ao fato de tentarmos resolver o problema da navegação autônoma com subproblemas já conhecidos, entretanto faremos uma breve analogia com a maneira com que as pessoas dirigem e, então, entraremos em detalhes mais técnicos de soluções para cada um dos módulos.

Em geral, quando uma pessoa quer ir a algum lugar, ela pensa nas ruas pelas quais vai trafegar e já imagina na sua mente o caminho em um nível não tão detalhado (Roteamento). Quando uma pessoa está dirigindo, ela precisa ter uma noção de localização para saber por qual rua ela deve prosseguir (Localização) e enquanto isso ela deve se atentar a alguns componentes das vias, como pedestres que tentam atravessar a pista, semáforos e placas de pare (Percepção). Com essas informações do estado da via, ela deve tomar uma decisão do que fa-

zer, como trocar de faixa, esperar o pedestre atravessar, cuidar com o veículo à frente (Decisão de comportamento), em seguida ela deve pensar como executar determinado comportamento (Planejamento de Movimentação), ela deve reduzir a velocidade para virar à direita? Em que ponto deve começar a virar o volante? Será que deve frear mais bruscamente para evitar um acidente ou uma frenagem mais suave é suficiente para parar o veículo em segurança? Com essa ideia do que fazer, basta executar a movimentação planejada (Controle). Essa arquitetura tenta refletir como tratamos o paralelismo e concorrência de nossas percepções e pensamentos; em qual nível na direção tratamos situações emergenciais, dependendo da gravidade da situação, não pensamos se devemos frear ou não quando um outro veículo cruza a nossa frente de repente, simplesmente freamos.

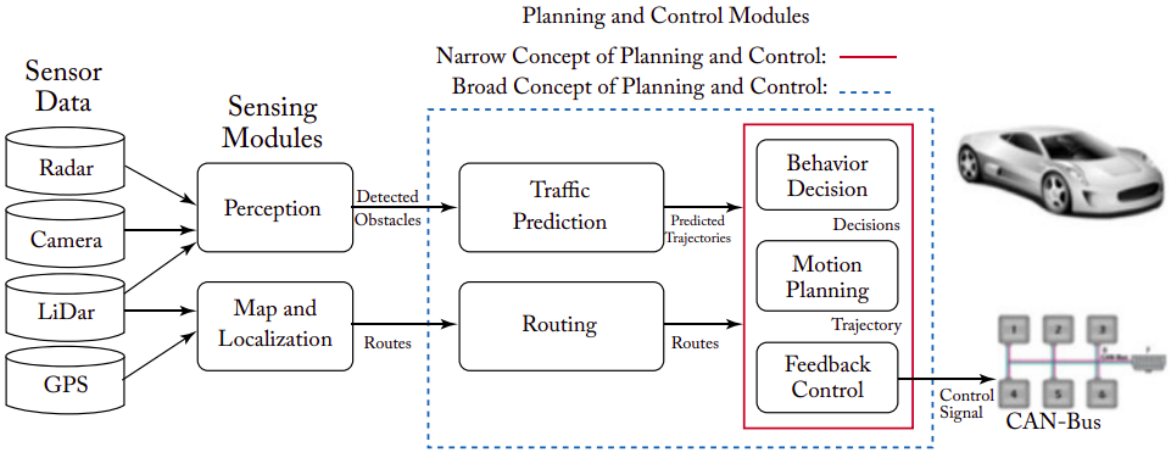


Figura 4 – Arquitetura usual de SNA. Fonte: (LIU et al., 2017)

Comecemos a apresentação de cada módulo pelos módulos sensoriais, em específico, o módulo de Localização. Esse módulo é responsável por localizar o veículo de alguma forma que o ajude com a identificação de onde ele está nos mapas, como por exemplo, em coordenadas de latitude e longitude. Sensores muito utilizados neste módulo são GPS, IMU e wheel encoder. Alguns SNAs utilizam câmeras (odometria visual), LiDARs ou até RADARs para ajudar a obter uma localização mais precisa. Usamos vários sensores para obter a localização, pois assim, podemos obter vantagens específicas de cada tipo de sensor, por exemplo, GPS não acumula erros com as suas estimativas, mas não é tão preciso quanto e possui uma frequência menor que as estimativas feitas a partir de dados da IMU e dos wheel encoders, que por outro lado se usados sozinhos para estimar a localização, podem acumular imprecisões de localização. Fusão de sensores é o nome que se dá ao processo de utilizar informação proveniente de vários sensores para obter uma informação mais robusta e confiável. Um dos conjuntos de técnicas mais conhecidos de fusão de sensores são os filtros de Kalman. Um exemplo que podemos dar de veículo autônomo que utiliza filtros de Kalman com odometria visual é a Bertha da Mercedes Benz (ZIEGLER et al., 2014).

O outro módulo sensorial essencial para SNAs é a Percepção. Esse módulo provê informações do ambiente em que o veículo está inserido, como por exemplo, por onde ele pode

trafegar; indicações e obrigações de como dirigir dadas por sinalizações de trânsito; posição e orientação dos outros usuários das vias; e informações até mesmo que podem ajudar o veículo a se localizar. Alguns veículos autônomos - (MONTEMERLO et al., 2009) (URMSON et al., 2009) - usam LiDAR e câmera monocular para obter essas informações, porém tecnologias de RADARs e câmeras estereoscópicas também são utilizadas em alguns veículos (ZIEGLER et al., 2014). O LiDAR emite feixes de luz em várias direções e mede o tempo que leva para o feixe retornar, com esse tempo é possível inferir a distância até o objeto que refletiu o feixe; se o feixe não retornar, então infere-se que não há nenhum objeto naquela direção. Com isso, temos um mapa 3D com nuvens de pontos que representam objetos no ambiente, como ilustrado pela Figura 5. Já com a câmera monocular, temos informações visuais (cores) de uma projeção do ambiente 3D em uma imagem. Assim, com o LiDAR e a câmera monocular conseguimos a estrutura geométrica espacial e as cores dos objetos. Porém, como essas informações são exploradas? Para responder a essa pergunta, vamos abordar, a seguir, alguns conceitos da área de Visão Computacional e explicaremos as interseções com as necessidades de um SNA.

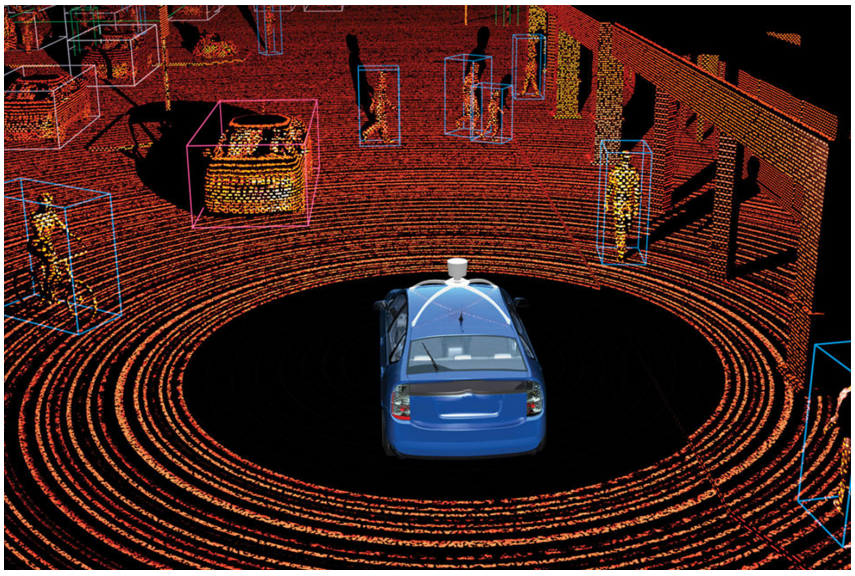
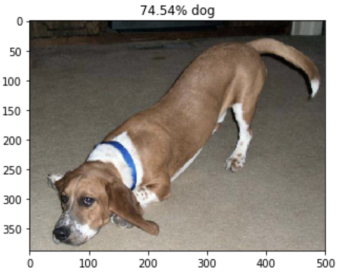
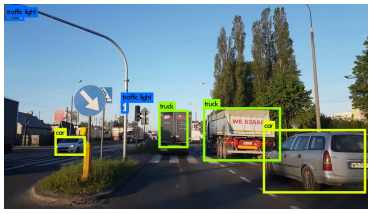


Figura 5 – Visualização de nuvens de pontos obtidas por um LiDAR. Os paralelepípedos, em volta de alguns participantes do ambiente, não fazem parte dos dados obtidos pelo LiDAR, foram adicionados por um componente do módulo de Percepção ligado a detecção de objetos. Fonte: University of Michigan - Stories. Imagem de Graham Murdock.

Alguns dos problemas abordados pela Visão Computacional são: Classificação de Imagem, Detecção de Objetos e Segmentação Semântica. Um exemplo de um problema de Classificação de Imagem é diferenciar imagens de gatos de imagens de cachorros. Soluções para esse problema devem receber como entrada uma imagem (com um gato ou um cachorro) e como saída devemos ter a sinalização de a qual dos dois animais a imagem corresponde - Figura 6a. Com a vitória da AlexNet (KRIZHEVSKY; SUTSKEVER; HINTON, 2012), uma Rede Neural Convolucional (CNN) no Imagenet Challenge (RUSSAKOVSKY et al., 2014), as CNNs passaram a ser bastante estudadas e se tornaram a principal solução para o problema de Classificação de Imagem, inclusive várias soluções para Classificação de Imagem são usadas como backbones para Detecção de Objetos e Segmentação Semântica. No problema de



(a) Classificação de imagem. Fonte: Medium - Image Classifier: Cats vs Dogs.



(b) Detecção de objetos. Fonte: Analytics Vidhya.

(c) Segmentação semântica. Fonte: CityScapes examples.

Figura 6 – Três problemas de Visão Computacional.

Detecção de Objetos, o foco está em achar bounding boxes que delimitam objetos de interesse e em classificá-los corretamente - Figura 6b. Já para a Segmentação Semântica, queremos classificar cada pixel como pertencente a uma classe de interesse - Figura 6c.

Uma abordagem comum para SNAs é usar uma rede neural de segmentação semântica e uma de detecção de objetos. A principal função da rede de segmentação semântica, nesse contexto, é achar o caminho navegável (as faixas de trânsito) para que com mais alguns processamentos, saibamos o centro das faixas de trânsito. Assim, o veículo pode ter alguma noção de um caminho aconselhado a se fazer. Outra função é detectar buracos, rachaduras e outras falhas que podem haver na pista. Em alguns casos, a rede de segmentação semântica pode ajudar a rede de detecção de objetos a localizar e classificar um objeto. Já a rede de detecção de objetos, busca por cada instância de cada classe de interesse. As bounding boxes encontradas com suas respectivas classes podem ser usadas em conjunto com os dados do LiDAR para obter uma estimativa em 3D da posição, orientação e tamanho do objeto. Outra questão que devemos levar em consideração são as diferentes estimativas obtidas em tempos relativamente próximos, pois as estimativas passadas podem ajudar com a inferência da posição e orientação, visto que os objetos dependendo da classe possuem algumas restrições de movimentação (um carro não consegue trocar de sentido tão rápido, uma pessoa não consegue andar a 30m/s e uma placa não troca de posição, por exemplo) e vários objetos não mudam de tamanho com o passar do tempo. Uma boa estimativa dessas características tanto no presente quanto em alguns instantes no passado são fundamentais, principalmente para o módulo de Predição de Tráfego.

Com a área navegável encontrada pela segmentação semântica e as informações obtidas a partir da rede de detecção de objetos em conjunto com os dados do LiDAR, podemos construir um mapa (chamado de occupancy grid) que nos diz por onde podemos navegar. O occupancy grid pode ser pensado como uma matriz 2D cujas células apresentam valores de 0 a 1, sendo que 0 significa que o local está ocupado e 1 livre. Há diversas versões de occupancy grids, sendo que vamos apresentar mais detalhes neste parágrafo de uma versão probabilística, apresentada em (WASLANDER; KELLY, 2019), em que cada célula pode apresentar valores quaisquer entre 0 e 1. Vale ressaltar três das principais diferenças entre a área navegável da segmentação semântica (ANSS) e essa versão de occupancy grid (OG). A primeira é que com

a ANSS temos informações 3D projetadas sobre uma imagem 2D que não nos traz diretamente se uma posição da via está ocupada ou não; já com o OG, temos a projeção inversa da imagem, assim podemos saber diretamente se, por exemplo, a posição a um metro à frente do veículo está ocupada. A segunda é que na ANSS se houver um veículo que está se movendo em cima da pista, essa área não aparecerá como navegável, porém ela deve aparecer como livre no OG (ressaltamos que essa característica está presente nesta versão de OG - em outras versões isso pode não valer). A terceira diferença é que o OG leva em consideração o seu estado no instante passado, enquanto que as redes de segmentação semântica normalmente não consideram questões temporais. Um dos motivos para usarmos OGs probabilísticos com essas características é que, por levarem em conta estimativas passadas, esses OGs tendem a ser mais robustos. Ao iniciar, um OG começa com todas as células incertas (0.5). Para atualizar o estado do OG levamos em consideração os dados do LiDAR, a ANSS, o estado anterior e as características dos objetos que estão em movimento. Em geral, podemos fazer três operações com a crença de cada posição estar livre: aumentar, diminuir ou manter (caso não tenhamos informações). A ideia é que com a movimentação do veículo mais informação será obtida e com isso os ruídos (tanto dos sensores quanto pela existência de objetos móveis) não terão muito impacto. Outro ponto que vale a pena levantarmos é que o OG não é um mapa que vai possuir informações de uma rua inteira, na verdade ele representa uma região próxima do veículo apenas e seu tamanho pode variar conforme a velocidade do veículo.

Outro mapa que pode ser usado para auxiliar algumas etapas da navegação autônoma é o mapa HD (High Definition map). Esse mapa é bem diferente do occupancy grid, ele assemelha-se bem mais aos mapas que usamos para saber como chegar em determinado lugar, porém, como o próprio nome diz, com um nível de detalhamento bem maior. Os mapas HD são muito usados (LIU et al., 2017) para o módulo de Roteamento, podem auxiliar na identificação de alguns objetos de interesse na via, como placas e buracos e também na identificação do centro das faixas. Algumas informações úteis que podem estar em mapas HD são: referências às bordas ou centro das faixas de trânsito, sinalizações (placas, semáforos, faixa de pedestres, etc), informações de falhas na pista e de obras. Por essas informações mudarem apenas em questões de dias ou meses, os mapas HD podem ser atualizados pelos SNAs quando o veículo não estiver em uso. Com os mapas HD, o SNA pode tirar vantagem de saber onde estão alguns objetos na via antes de vê-los, o que pode ser muito útil em rodovias em dias com pouca visibilidade do ambiente por causa de chuva, neblina ou falta de iluminação adequada por exemplo. Porém, os mapas HD também trazem consigo alguns desafios, como a conectividade segura do veículo com o exterior, pois o veículo precisa ser capaz de carregar novos mapas e também de atualizá-los.

Com as informações que obtemos dos diversos sensores e dos mapas, podemos fazer previsões da movimentação dos diversos componentes dinâmicos que compõem o trânsito. Esse módulo se chama Predição do Tráfego (PT). Neste módulo precisamos de soluções que diferenciam o comportamento dos diversos componentes do trânsito, visto que o perfil de movimentação de, por exemplo, ciclistas, pedestres e carros, são completamente diferentes.

Para que possamos tratar cada componente de forma adequada, precisamos juntar as informações de sensores, como LiDARs e câmeras, obtidas no módulo de Percepção para que possamos associar estimativas melhores para a classificação e para o histórico de características (posição, orientação, velocidade) do componente. Além dessas informações, o comportamento que se quer prever depende também de outros componentes dinâmicos, o que aumenta ainda mais a complexidade do problema. Algumas soluções buscam separar o problema de prever as ações futuras de outros componentes em duas partes: prever o comportamento em alto nível (parar, acelerar, trocar de faixa, etc) e prever a movimentação. Assim, pode-se aproveitar algumas técnicas do módulo de Decisão de Comportamento e PM. Porém, apostar tudo em um comportamento pode não ser uma boa ideia, principalmente por causa da grande incerteza relacionada com o problema. Por isso, muitas soluções para este módulo são abordagens que sustentam simultaneamente diversas hipóteses e tentam associar cada hipótese a uma probabilidade.

Agora, vamos passar para o módulo de roteamento, também chamado de planejamento de missão. As duas principais funções deste módulo são apresentar um caminho da origem até o destino e caso o veículo tenha que sair da rota por algum motivo (manobra de emergência, obra na pista, etc), gerar um novo caminho. Vale ressaltar que utilizar a mesma solução desenvolvida para o roteamento de automóveis dirigidos por pessoas não é adequado, pois os mapas para esses problemas não apresentam o mesmo nível de detalhamento dos mapas HD. Além disso, o melhor caminho para um veículo dirigido por uma pessoa pode não ser o mesmo que o melhor caminho para um veículo autônomo, visto que as dificuldades na navegação são diferentes. As soluções para este módulo, em geral, modelam os mapas HD como grafos dirigidos ponderados e realizam uma busca pelo caminho com menor custo usando algoritmos com heurísticas para acelerar a busca. Um algoritmo muito conhecido que identifica o caminho de custo mínimo em grafos dirigidos ou não dirigidos com pesos não negativos é o algoritmo de Dijkstra (DIJKSTRA, 1959). A* (HART; NILSSON; RAPHAEL, 1968) é outro algoritmo também muito conhecido que pode ser visto como uma extensão do algoritmo de Dijkstra. O A* adiciona ao algoritmo de Dijkstra a noção de heurística para agilizar a busca e mantém a otimalidade desde que a função heurística nunca superestime o custo de uma aresta. Muitos dos módulos usados em SNAs possuem um componente baseado no algoritmo de Dijkstra ou no A*. Porém, os módulos de roteamento possuem outros problemas, como a definição do custo das arestas, definição das partes do HD que devem ser utilizados para achar um caminho e a comunicação com o módulo de Decisão de Comportamento e o módulo de PM. Observe a Figura 7. Quem mora em Florianópolis sabe que as duas faixas mais à direita (caminho para o norte da ilha), nas horas de rush, possuem muito mais movimento que as três faixas mais à esquerda, sendo que seguindo pelas faixas à esquerda há outro caminho que pode levar ao mesmo lugar das duas faixas à direita. Quando o veículo percebe que a rota selecionada (faixas à direita) está com tráfego muito mais intenso, será que vale a pena trocar de rota? Quem vai responder esse pergunta é o módulo de Decisão de Comportamento, porém ele necessita de informações do custo de caminhos alternativos próximos de onde estamos para tomar decisões desse tipo. Além disso, manter caminhos alternativos também ajuda a não ter que recomputar a

rota inteira até o destino final caso a rota principal não seja seguida.

Figura 7 – Situação de possível troca de rota. Fonte: Google Maps.

O módulo de Decisão de Comportamento (DC), ou planejamento de comportamento, é o módulo que mais sofre por causa da alta complexidade e diversidade dos diferentes cenários nos quais um veículo pode se encontrar. Normalmente, o módulo de DC é o que possui mais entradas, visto que depende da rota traçada; de informações atuais e passadas do veículo (posição, velocidade, orientação, etc); informações de objetos dinâmicos (classe, velocidade, orientação, posição, etc) e estáticos que podem ocupar a pista reduzindo as opções de movimentação do veículo; percepções e informações fornecidas pelo mapa HD das sinalizações de trânsito que podem ser sinalizações verticais (placas), sinalizações horizontais (marcações de divisão de faixas, faixa de pedestres, etc), gestos de um guarda de trânsito, semáforos, dispositivos auxiliares (cones) e até sinais sonoros feitos por guardas de trânsito; occupancy grid; histórico de comportamentos; regras de trânsito; perfil de navegação de preferência de passageiro (caso o passageiro possa pedir para o veículo ir mais devagar por não estar com pressa, por exemplo), dentre várias outras especificidades de diferentes soluções. Como saída deste módulo, podemos ter o comportamento específico que deve ser seguido impreterivelmente ou um conjunto de possíveis comportamentos com a probabilidade ou com valores de "vontade" que o veículo tem para executar cada um, sendo que o comportamento realmente executado depende do comportamento futuro dos objetos dinâmicos no trânsito.

Uma solução simples para esse módulo é a modelagem dos diversos cenários e suas transições como uma Máquina de Estados Finita (MEF). A principal ideia para soluções baseadas em MEFs é que os diferentes comportamentos (parar, seguir veículo, manter velocidade, desacelerar) devem ser representados pelos estados e as transições seriam regras de trânsito, por exemplo, avançar apenas quando o pedestre terminar de atravessar a faixa, que seria traduzida para a MEF como uma transição de parar para pedestre para acelerar cuja condição é o pedestre ter terminado de atravessar. Na Figura 8 podemos observar um exemplo extremamente simplificado de uma MEF para veículos autônomos. Soluções nesse estilo são ótimas (princi-

palmente, por causa da simplicidade) para tratar ODDs com poucos e simples cenários, porém são inviáveis para atingir nível 4 de autonomia, visto que cenários não previstos podem existir. Outro ponto negativo dessas soluções é que só há um comportamento selecionado, o que não reflete a incerteza frente os diversos objetos dinâmicos presentes em situações reais de trânsito.

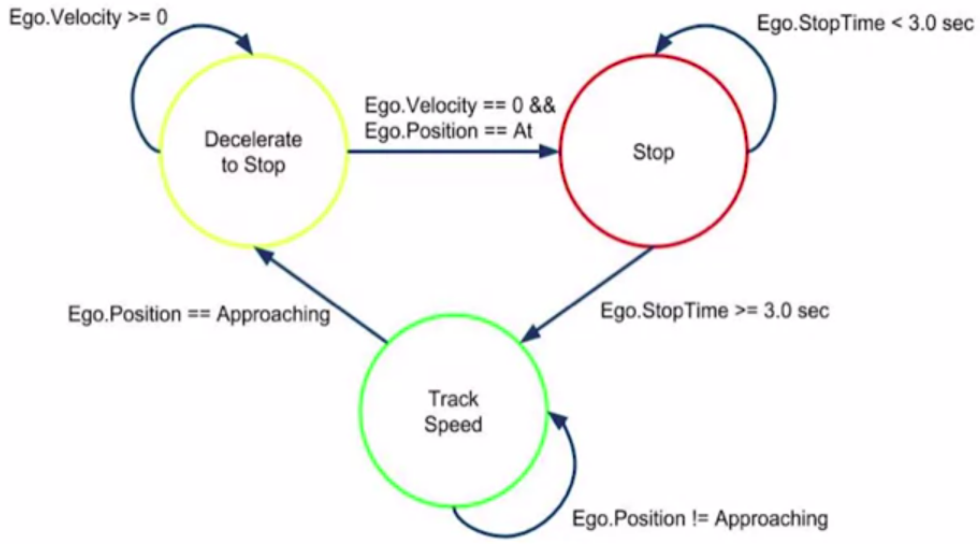


Figura 8 – Máquina de estados finita com três estados: Stop, Track Speed e Decelerate to Stop. Fonte: (WASLANDER; KELLY, 2019).

A partir do comportamento selecionado pelo módulo de DC (o qual pode ser representado por algum objetivo de estado final do veículo - chamado de goal state), o estado atual, o occupancy grid e as previsões de movimentação dos componentes dinâmicos do ambiente, o módulo de Planejamento de Movimentação deve traçar uma trajetória de estados que seja ao menos segura (sem colisão) e factível (levando em conta as restrições físicas do veículo). Mais algumas condições podem ser adicionadas também, como por exemplo, a suavidade (sem curvas e acelerações repentinas) da trajetória e a sua distância em relação ao centro da faixa. Explicações mais detalhadas do problema de planejamento de movimentação para veículos autônomos e possíveis soluções serão abordadas nas seções seguintes.

Por último, temos o módulo de Controle. Vale trazer uma divisão entre soluções de controle: sistemas de malha aberta e malha fechada. Sistemas de malha aberta não levam em conta a saída do sistema para ajustar as próximas saídas. Já um sistema de malha fechada vai considerar essas possíveis diferenças entre o valor desejado e o valor obtido e tentará adaptar-se para atingir o valor desejado. Imagine um veículo autônomo que quer passar por certo ponto com determinada velocidade, um sistema de malha aberta apenas calcularia quanto que deveria ser acionado do acelerador para chegar naquela velocidade. Por outro lado, um sistema de malha fechada analisaria se o quanto deveria ter sido acionado do acelerador em instantes passados, foi adequado para chegar nas velocidades desejadas anteriormente e com essa análise um sistema de malha fechada adaptaria o valor calculado do acionamento do acelerador. De

acordo com (SAMAK; SAMAK; KANDHASAMY, 2020), sistemas de malha aberta não são uma opção para veículos autônomos, visto que o veículo está sujeito a condições de operação que mudam constantemente. Em geral, o módulo de Controle recebe a trajetória de estados do módulo de PM e ações tomadas em cada parte da trajetória e tenta inferir os comandos que devem ser passados aos atuadores. Além disso, o módulo de Controle pode usar informações adicionais de sensores como IMU e GPS para obter feedbacks.

2.2 PLANEJAMENTO DE MOVIMENTAÇÃO - MOTION PLANNING

Como vimos na seção anterior, o módulo de planejamento de movimentação, enquanto busca uma trajetória de um estado inicial a um estado final, tenta tratar três pontos: restrições físicas do veículo, obstáculos fixos e obstáculos móveis. Para que haja um bom entendimento das consequências de cada uma dessas dificuldades no módulo de PM, vamos analisar problemas mais simples. Esses problemas são: Piano Mover's Problem (PMP), two-point Boundary Value Problem (BVP) e planejamento de movimentação com restrições diferenciais em ambiente com obstáculos (Motion Planning Under Differential Constraints Problem - MPUDCP).

2.2.1 Piano Mover's Problem

Antes de trazermos uma definição formal do problema, vamos pensar em um exemplo. Observe a Figura 9. Imagine que queremos transportar um piano, representado como um retângulo, a partir da sua configuração inicial dada por $q_0 = (x_0, y_0, \theta_0)$ até um configuração final $q_f = (x_f, y_f, \theta_f)$. Ademais, devemos nos atentar que há obstáculos no ambiente em que o piano está, paredes por exemplo. Podemos rotacionar e transladar o piano à vontade, desde que não haja uma colisão com os obstáculos e toda mudança da configuração do piano deve ser contínua (não podemos simplesmente teletransportar o piano para a configuração final). Com isso, surgem duas perguntas: é possível chegar à configuração final? Qual sequência de configurações representa um caminho viável até a configuração final? Na Figura 10, podemos observar uma simplificação de uma sequência de configurações que consegue chegar a configuração final sem colidir com nenhum obstáculo.

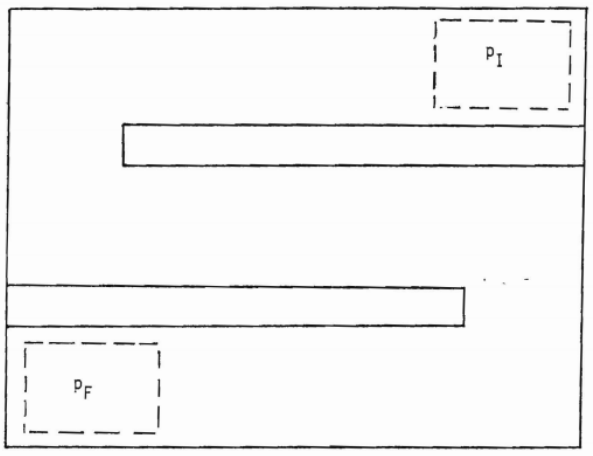


Figura 9 – Piano Mover's Problem. Fonte: (REIF, 1979).

A seguir, abordamos alguns conceitos importantes para o planejamento de movimentação e, então, apresentamos a definição formal do PMP dada por (LAVALLE, 2006). Porém, antes, devemos fazer algumas pequenas observações: planejamos a movimentação para um objeto específico, esse objeto será chamado de robô. Os robôs podem ser constituídos de um único

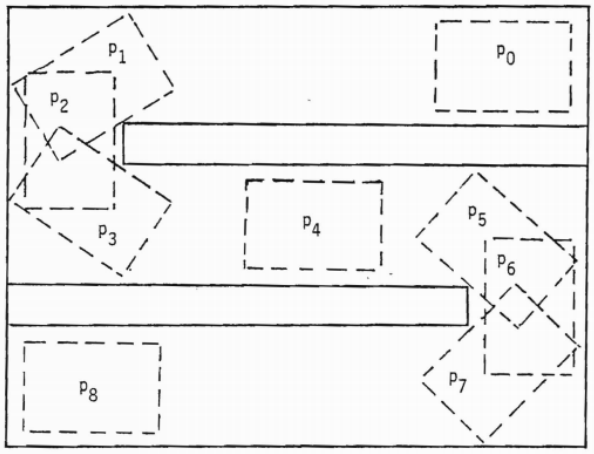


Figura 10 – Piano Mover's Problem - solução. Fonte: (REIF, 1979).

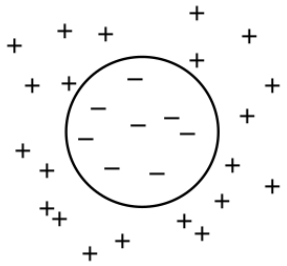


Figura 11 – Conjunto algébrico representado pela região com os símbolo de "+". Fonte: (LAVALLE, 2006).

corpo rígido ou de mais vários componentes conectados através de estruturas que permitem os componentes se moverem com uma certa independência entre si.

Semi-algébrico Um modelo/região/conjunto semi-algébrico é um conjunto (neste caso representa uma região de um espaço W) derivado a partir da união e intersecção de conjuntos algébricos, que por sua vez são derivados a partir de uma única primitiva polinomial. Essas primitivas polinomiais são funções polinomiais. Um exemplo de um conjunto algébrico dado por (LAVALLE, 2006) é: $H = \{(x,y) \in W | f(x,y) \leq 0\}$. Cuja primitiva polinomial usada é: $f = x^2 + y^2 - 4$. Esse conjunto algébrico resulta na região representada pelos símbolos de "+"na Figura 11.

Espaço de Configuração (C-space) Voltando ao exemplo do piano, podemos pensar no C-space como o espaço dos estados do piano, sendo que o estado do piano é representado por (x,y,θ). Porém vale lembrar que o piano, nesse exemplo, está em um mundo $W = \mathbb{R}^2$. A busca que devemos fazer para encontrar uma solução para o problema é feita sobre o C-space. Outro exemplo que podemos dar é o de um drone navegando por uma cidade levando em conta uma representação em $W = \mathbb{R}^3$ da cidade e do drone. Um estado do drone é dado por $q = (x,y,z,\theta_1,\theta_2,\theta_3)$, onde (x,y,z) representa a posição do drone e $(\theta_1,\theta_2,\theta_3)$ representa a orientação do drone. O C-space neste caso é o espaço de todos os estados q.

PMP 1. Há um mundo W em que o robô e os obstáculos estão, sendo $W = \mathbb{R}^2$ ou $W = \mathbb{R}^3$.

2. Os obstáculos são definidos como uma região semi-algébrica $O \subset W$.
3. Há um robô semi-algébrico definido em W. Pode ser um robô de corpo rígido A ou uma coleção de m componentes $A_1, A_2, ..., A_m$, sendo que esses componentes possuem descrições de suas possíveis movimentações.
4. Há um espaço de configuração C especificado a partir de todas as transformações que podem ser aplicadas no robô. A partir de C, derivamos C_{obs} e C_{free} - espaço com colisão e livre de colisão, respectivamente.
5. Há uma configuração inicial dada por $q_0 \in C_{free}$.
6. Há uma configuração final dada por $q_f \in C_{free}$.
7. Caso haja um caminho contínuo $\tau : [0,1] \rightarrow C_{free}$ tal que $\tau(0) = q_0$ e $\tau(1) = q_f$, este deve ser apresentado. Caso contrário, deve-se avisar que não há um caminho.

Foi demonstrado por (REIF, 1979) que o Piano Mover's Problem é PSPACE-hard e se $P \neq NP$, temos que o PMP não pode ser resolvido em tempo polinomial. Depois, apresentaremos o Motion Planning Under Differential Constraints Problem (MPUDCP) como uma generalização do PMP.

2.2.2 Two-point Boundary Value Problem

De acordo com (GLADWELL, 2008), podemos definir o Boundary Value Problem como um sistema de equações diferenciais ordinárias com valores especificados para a solução e as derivadas em mais de um ponto. Já o Two-point Boundary Value Problem (que será chamado de BVP para simplificar, apesar de BVP representar uma classe maior de problemas) possui exatamente dois pontos especificados para a solução e as derivadas. Um dos métodos existentes para resolver BVPs é o método shooting. Podemos pensar no MPUDCP, que veremos a seguir, como um BVP com obstáculos, pois os dois pontos do BVP podem ser pensados como os pontos de origem e destino no C-space. Um ponto negativo de pensar no MPUDCP como uma extensão do BVP é que, de acordo com (LAVALLE, 2006), em geral, os métodos para resolver BVPs não podem ser bem adaptados para lidar com obstáculos.

2.2.3 Motion Planning Under Differential Constraints Problem

Agora que já foram abordados e diferenciados o PMP e o BVP, podemos discutir em detalhes o MPUDCP. Primeiramente, vamos apresentar alguns conceitos e representações relacionadas com as restrições diferenciais.

Voltando ao exemplo do piano, porém agora consideramos que não é possível rotacionar o piano, ou seja, o C-space é $\mathbb{R}^2$. Além disso, adicionamos ao problema a velocidade, representada por $\dot{q} = (\dot{x}, \dot{y})$. Gostaríamos de ter algumas restrições na velocidade, por exemplo, a

velocidade deve ser positiva no eixo x ou a velocidade no eixo x não pode passar de determinado valor. Existem duas formas de representar as restrições: implicitamente e parametricamente.

A representação implícita é qualquer restrição na forma:

$$g(q,\dot{q}) \bowtie 0$$

Já a representação paramétrica é dada por n equações da forma:

$$\dot{q_1} = f_1(q,u)$$

$$\dot{q_2} = f_2(q,u)$$

$$\dots$$

$$\dot{q_n} = f_n(q,u)$$

mais as inequações com termos de u e q se existirem.

Acima, q é um vetor do C-space de dimensão n; q_i representa o valor da iésima dimensão de q; $\bowtie$ representa $\{=,<,\leq,>,\geq\}$; u representa uma ação. Na representação paramétrica temos a noção de ação que podemos tomar em cada estado q. De uma maneira mais formal, para o exemplo do piano, temos $T_q(\mathbb{R}^2)$ como um espaço vetorial tangente a q e que possui todos $\dot{q}$. Ao restringirmos as velocidades, temos um espaço $U(q) \subset T_q(\mathbb{R}^2)$ que representa essas velocidades permitidas em q. Quando adicionamos a noção de ação, as possíveis ações são elementos de $U(q)$, assim elas não representam mais uma velocidade diretamente. Para obter a velocidade precisamos de uma função f que mapeia q e u em $\dot{q}$. Isso é útil, pois podemos usar essa noção de ação para representar ações físicas reais, como por exemplo, o acelerador de um carro ou o comando que define o ângulo de rolagem de um drone.

Para ilustrar um pouco melhor a diferença entre as duas representações, vamos dar dois exemplos (ainda usando o piano com $q = (x,y)$) em que começamos com a forma implícita e vamos gerar a forma paramétrica. Para o primeiro exemplo, a restrição implícita será da forma:

$$\dot{y} - \dot{x} + 2 + x = 0$$

Com isso, podemos definir a restrição paramétrica como:

$$\dot{y} = u_1$$

$$\dot{x} = u_1 + 2 + x$$

No segundo exemplo, a restrição implícita é dada por:

$$\dot{y}^2 + \dot{x}^2 \geq 2$$

$$\dot{x} = 1$$

Com isso, a restrição paramétrica pode ser dada por:

$$\dot{x} = 1$$

$$\dot{y} = u_1$$

$$u_1^2 \geq 1$$

A seguir, apresentamos a definição (com algumas simplificações) dada por (LAVALLE, 2006) ao MPUDCP. Essa definição é dada como uma extensão da definição do PMP e considera as restrições na forma paramétrica.

1. Há um mundo, um robô, uma região de obstáculos e um C-space definidos da mesma forma em que foram definidos no PMP.

2. Há um intervalo de tempo $T = [0, \infty)$.

3. Há um manifold diferenciável X, chamado de espaço de estados (state space). X pode ser igual a C ou pode ser espaço de fase (phase space) derivado de C. A partir de X e da região de obstáculos são derivados X_{obs} e X_{free} - state space com colisão e sem respectivamente.

4. Cada $x \in X$ possui um conjunto restrito de ações $U(x) \subseteq \mathbb{R}^m \cup \{u_t\}$, onde u_t representa a ação final e m é o número de variáveis de ação.

5. As restrições estão representadas na forma paramétrica. Com isso, temos todos $\dot{x_i}$ representados na forma de uma de uma função $f_i(x, u)$.

6. Há um estado inicial $x_0 \in X_{free}$.

7. Há um conjunto de estados finais $X_f \in X_{free}$.

8. Caso haja uma trajetória de estados $\tilde{x}(t) = \tilde{x}(0) + \int_0^t f(\tilde{x}(t'), u(t'))dt'$, sendo $\tilde{u} : T \to U$ uma trajetória de ações, tal que (i) $\tilde{x}(0) = x_0$; (ii) existe algum $t_f > 0$ para qual $\tilde{u}(t_f) = u_t$, $\tilde{x}(t_f) \in X_f$ e $\forall_{0 \leq t \leq t_f}\ \tilde{x}(t) \in X_{free}$, esta trajetória deve ser apresentada. Caso contrário, deve-se avisar que não há uma trajetória.

Podemos estender o MPUDCP para possuir objetos dinâmicos com o seguinte adendo à descrição do problema: há um conjunto de objetos móveis D definidos da mesma forma que o nosso robô, porém para cada $d \in D$ temos uma trajetória $\tilde{x} : T \to X$. Além disso, X_{free} passa a depender do instante t e com isso, mais algumas pequenas adaptações da definição são necessárias.

Com o problema formalmente definido, podemos discutir em mais detalhes algumas ideias. Comecemos pelas diferenças entre o problema formalizado e o problema que veículos autônomos devem resolver não apenas no módulo de planejamento de movimentação, mas em todos os componentes relacionados mais diretamente com a movimentação do veículo (Roteamento, Decisão de Comportamento, Planejamento de Movimentação e Controle). Duas grandes diferenças que podemos notar é o desconhecimento e as incertezas do veículo frente ao ambiente (inclusive aos componentes dinâmicos presentes no ambiente). Na formalização, os obstáculos do ambiente são totalmente descritos por modelos semi-algébricos, porém o veículo modela o

occupancy grid apenas para uma região em sua volta relativamente pequena (o occupancy grid é baseado nos módulos sensoriais, se o veículo não passou ainda por uma região, ele não tem como modelá-la) quando comparada com trajetos que realizamos diariamente e essa modelagem do occupancy grid possui incerteza, visto que os módulos sensoriais não são perfeitos. Outro ponto é que, na formalização, há o conhecimento do estado de todos os componentes dinâmicos em todos os instantes de tempo, o que é bem diferente das previsões que o módulo de Previsão do Tráfego faz. Ademais, as movimentações dos componentes dinâmicos, como outros veículos, podem depender das movimentações do veículo. Todos esses pontos esclarecem porque não podemos tratar a movimentação de um veículo autônomo como uma única instância do problema formalizado. Porém, com a arquitetura usual de SNA, tenta-se preparar o sistema, aproximando-o ao máximo ao problema formalizado, para que possamos usar soluções ao menos similares.

Apesar de todas as dificuldades adicionais que a navegação autônoma de veículos possui em relação ao problema formalizado, há algumas facilidades. Uma delas é o fato de que veículos tendem a seguir o centro da faixa de trânsito em que estão e situações em que os veículos realizam curvas acentuadas saindo da faixa são bem raras, com exceção de alguns momentos, como por exemplo, entrar e sair de algum lugar onde o veículo estacionou ou parou. Por causa disso, podemos focar a busca dos pontos da trajetória em algumas áreas específicas do C-space relativas ao veículo, como áreas mais a frente na mesma faixa de trânsito usando um ângulo de abertura não tão grande.

Falamos diversas vezes de planejamento de trajetórias e de caminhos, mas ainda não esclarecemos a diferença desses dois conceitos no contexto de planejamento de movimentação. O planejamento de trajetória se refere à busca de uma sequência de pontos no C-space que possuem uma noção de tempo ou que ainda não possuem, mas será atribuído a cada ponto um instante de tempo. Já o planejamento de caminho se restringe a mesma busca por essa sequência de pontos, mas sem a noção de tempo. Para resolver o PMP, por exemplo, não é necessária nenhuma noção de tempo, por isso podemos chamar o PMP como um problema de planejamento de caminho. Uma forma de tentar simplificar a tarefa realizada pelo módulo de PM é quebrar o planejamento da trajetória em duas partes: planejamento de caminho desconsiderando os objetos móveis (com isso reduzimos o problema ao MPUDCP) e depois é feita a atribuição de velocidade ou instantes de tempo a cada um dos pontos considerando, agora, a predição da trajetória dos objetos dinâmicos do tráfego.

Algumas restrições simples que podem ser consideradas no PM são restrições de aceleração e curvatura. Restrições de aceleração estão relacionadas a quanto um veículo pode acelerar dependendo da potência do motor ou de questões de derrapagem. Além disso, é importante que sejam consideradas questões de conforto dos passageiros e de segurança, pois acelerações bruscas são mais imprevisíveis do ponto de vista de outros integrantes do trânsito. Mas idealmente, não devemos limitar rigidamente a aceleração máxima ao nível de conforto do passageiro, pois talvez uma aceleração mais brusca precise ser usada para evitar um acidente. Uma forma de modelar esse comportamento é por meio do uso de funções de custo de uma

trajetória para manter a aceleração em nível confortável e o uso de restrições representadas por inequações na forma implícita ou paramétrica para manter a aceleração sempre em níveis fisicamente possíveis de serem alcançados pelo veículo. Há também as restrições de curvatura, visto que há um ângulo máximo de rotação das rodas.

3 CRAWLER

Crawler é o nome do veículo autônomo que estamos desenvolvendo - salientamos, novamente, que diversas pessoas já contribuíram para este projeto. Podemos observar o veículo na Figura 12. Por causa de seu tamanho e peso reduzido e por não atingir velocidades altas, o Crawler é praticamente inofensivo e essa sua característica contribui muito para a realização dos mais diversos testes ligados a navegação autônoma. Neste capítulo, apresentaremos o ODD do Crawler e a sua estrutura (com estrutura do Crawler estamos nos referindo à tudo que compõe o Crawler, desde às rodas até o SNA), começando com o hardware e a infraestrutura de software e, em seguida, o software desenvolvido que compõe os módulos do SNA do Crawler. Daremos ênfase para o motivo de algumas decisões tomadas que influenciaram a organização da sua estrutura.

3.1 ODD DO CRAWLER

Antes de entrarmos em detalhes da estrutura do Crawler, é importante explicarmos o ODD para qual o Crawler foi projetado, assim trazemos mais motivações para a sua estrutura ter sido desenvolvida da forma como foi. Apresentaremos, agora, o ODD do Crawler e conforme formos apresentando os diversos componentes do Crawler, traremos algumas flexibilizações e limitações que admitimos durante o desenvolvimento. Além disso, abordaremos algumas das consequências do ODD escolhido para a estrutura do Crawler.

O Crawler foi desenvolvido para se locomover em estradas de terra, pavimentadas ou asfaltadas que podem possuir buracos ou outros obstáculos (por exemplo, carros estacionados), os quais o Crawler, se possível, deve contornar. Não pode haver qualquer componente dinâmico na estrada, como outros veículos ou pedestres. O Crawler deve a partir de um ponto inicial chegar ao seu destino, caso seja possível. O ponto inicial, o destino e a rota do ponto inicial ao destino são definidos por alguma pessoa, sendo que o Crawler deve começar, claramente, no ponto inicial. Se houver alguma bifurcação no caminho, a rota deve indicar por onde o Crawler deve seguir. O Crawler deve percorrer o seu caminho sem nenhuma intervenção humana. As regras de trânsito (exemplos: parar em semáforos; seguir o sentido das faixas de trânsito; seguir indicações de velocidade) são completamente ignoradas pelo Crawler. O Crawler não possui nenhum mapa construído previamente.

A ausência de componentes dinâmicos na estrada e a ignorância frente às regras de trânsito são dois fatores que ajudaram a simplificar extremamente o SNA do Crawler. O Módulo de Predição de Tráfego torna-se desnecessário, assim como todas as informações que seriam utilizadas apenas por esse módulo, como por exemplo, não é necessário identificar e diferenciar veículos e pedestres para pensar em suas possíveis trajetórias. Além disso, o Módulo de Decisão de Comportamento não depende das regras de trânsito para a escolha do goal state para o Módulo de Planejamento de Movimentação, por isso não necessita da identificação das sinalizações de trânsito e nem do conhecimento de algumas das regras que são aplicadas em

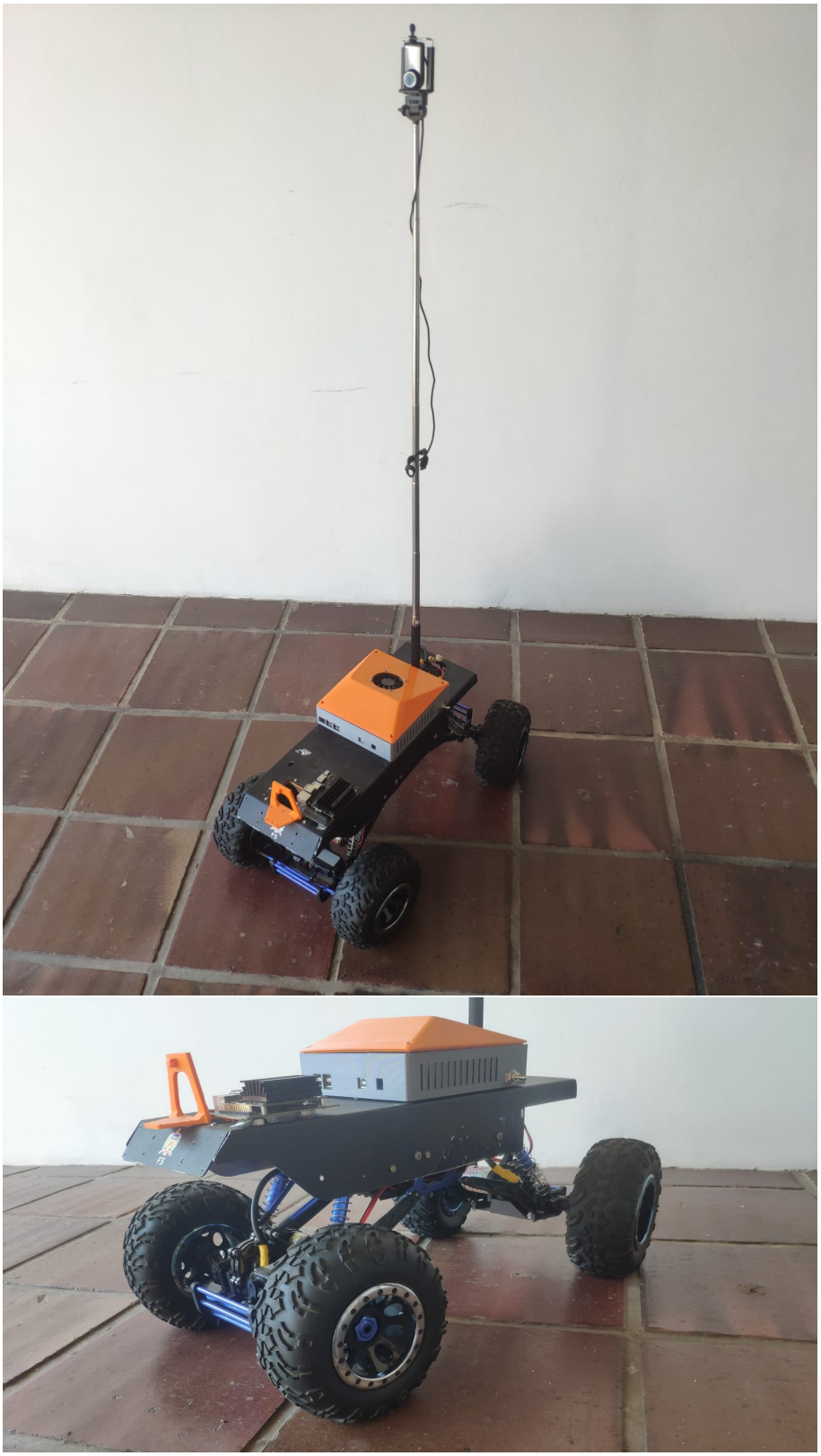


Figura 12 – Fotografias do Crawler.

determinado local.

3.2 HARDWARE E INFRAESTRUTURA

O Crawler possui cerca de 60 cm de comprimento; 47,5 cm de largura; 24 cm de altura; considerando a câmera e seu suporte, passa para 135 cm de altura, sendo que a câmera fica 5 cm abaixo (130 cm); a distância entre os eixos é cerca de 40 cm e a distância entre a câmera e a frente do veículo é 45 cm. No decorrer desta seção, focamos em apresentar os dispositivos (sensores, atuadores e computadores) presentes no Crawler, o que eles usam para se comunicar e apresentamos, brevemente, a infraestrutura de software. Não abordamos detalhes da mecânica nem da eletrônica do Crawler, como alimentação, drivers PWM (os quais geraram vários problemas), PCB utilizada, restrições de tensão e corrente, dentre vários outros detalhes.

3.2.1 Dispositivos

Estamos utilizando quatro sensores: uma IMU GY-521/MPU6050 que possui giroscópio e acelerômetro; um GPS ublox NEO-6M-0-001; uma câmera monocular Logitech Carl Zeiss Tessar 2.0/3.7 2MP Autofocus; e, por último, um odômetro AS5600. A IMU consegue nos fornecer também sua orientação que é inferida a partir dos dados de aceleração linear e velocidade angular. Vale notar que não utilizamos nenhum sensor que nos fornece distância em relação aos obstáculos no ambiente, como LiDAR, RADAR e câmera estéreo; isso faz com que tivéssemos que fazer algumas suposições (explicadas mais à frente) durante a geração do occupancy grid.

Há quatro atuadores no Crawler: dois motores e dois servos. Um dos motores tem a função de girar as duas rodas dianteiras e o outro as traseiras. Há um servo que direciona as rodas dianteiras e outro as rodas traseiras. Esses atuadores são comandados por sinais PWM (Pulse-width modulation).

Os três principais computadores que estão no Crawler são: um Arduino Mega 2560, uma Raspberry Pi 3 B v1.2 e uma Jetson Nano 2GB Developer Kit. A seguir, apresentaremos a divisão de tarefas para cada um dos computadores, vale notar que a divisão que apresentaremos aqui reflete, diretamente, o código implementado e não a organização lógica do SNA. Cada um desses componentes será explicado em detalhes na seção 3.3.

1. Arduino
    - Comandos dos Atuadores
2. Raspberry
    - Interface do Odômetro
    - Interface do GPS
    - Interface da IMU

- Módulo de Localização
- Módulo de Decisão de Comportamento
- Módulo de Planejamento de Movimentação
- Módulo de Controle

3. Nano

- Módulo de Percepção e Mapeamento

O Arduino é responsável por mandar sinais PWM para os atuadores. A escolha de mandar os sinais PWM pelo Arduino e não pela Raspberry foi feita, pois acreditávamos que na Raspberry não havia hardware PWM; posteriormente, percebemos que havíamos nos enganado. Assim, seguimos com o desenvolvimento do SNA e depois que descobrimos que na Raspberry há hardware PWM, tivemos outras prioridades relacionadas ao SNA. Todavia, acreditamos que seria interessante fazer alguns testes sem o Arduino e comparar se há alguma consequência para a navegação, em geral. A Nano é responsável apenas pela percepção e pelo mapeamento porque ela possui uma GPU que acelera o processamento dos dados visuais. Os demais componentes ficaram sob responsabilidade da Raspberry para reduzir o atraso com a troca de mensagens entre os componentes e para aliviar os custos de processamento dos outros computadores.

3.2.2 Conexões

O Arduino e todos os sensores se comunicam com a Raspberry através de I2C, com exceção do GPS que se comunica com a Raspberry por meio de UART. Para a comunicação entre a Raspberry e a Nano, utilizamos TCP/IP por Ethernet, pois necessitamos de uma taxa de transferência de dados maior entre esses dois dispositivos. A conexão da câmera com a Nano é feita por USB. A imagem 13 ilustra os dispositivos e suas conexões.

3.2.3 Infraestrutura de Software

Os sistemas operacionais que utilizamos para a Raspberry é o Ubuntu 20.04, para a Nano é o Ubuntu 18.04 e para o Arduino apenas carregamos o código no firmware usando a Arduino IDE. Uma alternativa seria usarmos um sistema operacional de tempo real para a Raspberry e para a Nano para que tivéssemos mais controle sobre a frequência de execução de cada componente do sistema, porém decidimos em usar o Ubuntu por causa de compatibilidade com bibliotecas da Nvidia e compatibilidade com ROS. Além disso, por nós (desenvolvedores do Crawler) estarmos mais acostumados com o Ubuntu, podemos dedicar mais do nosso tempo para tratar dos problemas intrínsecos à navegação autônoma e não problemas relacionados com especificidades de SOs. Ademais, reconhecemos que podemos ter problemas de atrasos com alguns processos, porém, se isso ocorrer tentaremos tratar desses problemas no próprio Ubuntu antes de trocar de SO.

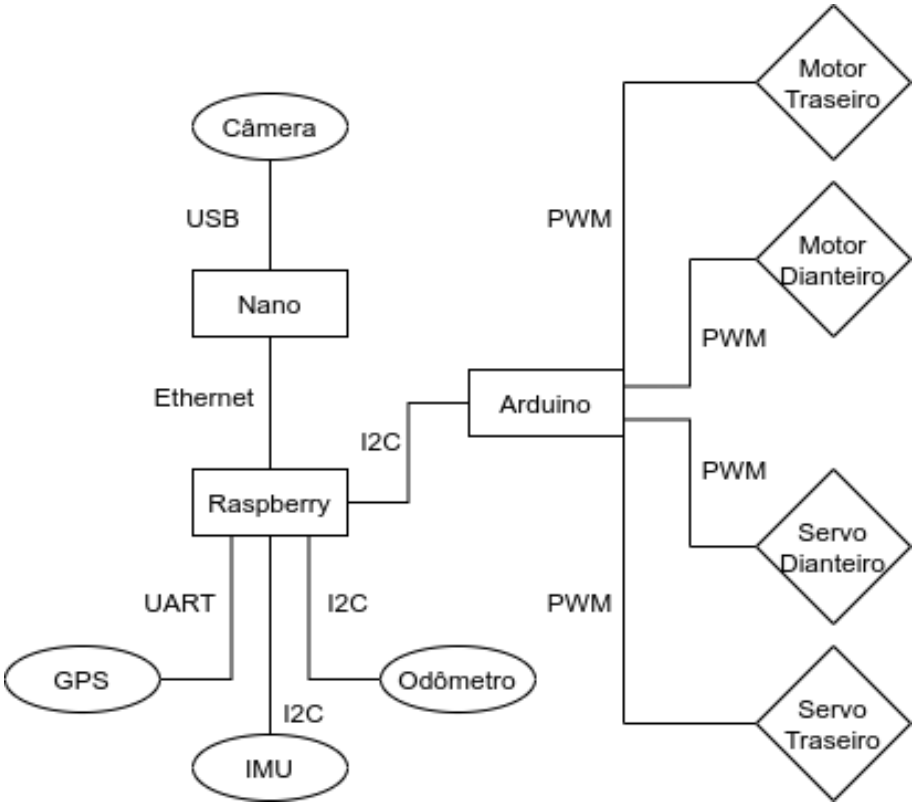


Figura 13 – Dispositivos presentes no Crawler e suas conexões.

De acordo com docs.ros.org, ROS (Robot Operating System) é um conjunto de bibliotecas e ferramentas para desenvolver aplicações de robótica. Para o Crawler, optamos pelo ROS2 para a Raspberry e para a Nano. O principal motivo para usarmos ROS é para realizar a comunicação entre os diversos componentes. Visto que o ROS abstrai certos aspectos da comunicação entre componentes facilitando e acelerando o desenvolvimento dessa parte, pudemos focar em outros aspectos do SNA. Outro ponto positivo do ROS são os seus pacotes e a forma de organizar esses pacotes para que troquem mensagens entre si e, assim, constituam um sistema maior, o que permite uma maior modularização para o código.

3.3 SOFTWARE

Organizamos o software de forma que pudéssemos ao máximo refletir a arquitetura usual de SNAs, porém principalmente por questões de desempenho computacional tivemos que fazer algumas adaptações, as quais serão explicadas no decorrer desta seção. Por levarmos em conta a modularização do código com o objetivo de substituirmos os componentes de seus respectivos módulos, decidimos usar pacotes ROS para representar alguns componentes e buscamos utilizar ao máximo as mensagens ROS para a troca de informação.

Nas subseções a seguir, apresentaremos em detalhes cada um dos componentes (com exceção das partes de percepção das pistas e decisão de comportamento que serão apresentadas no Capítulo 4 e do planejamento de movimentação que será apresentado no Capítulo 5) do SNA do Crawler e ainda abordaremos mais alguns detalhes do software. Porém antes, devemos es-

clarecer alguns pontos. A Figura 14 mostra quais informações são passadas de um componente para outro. A câmera captura uma imagem e passa essa imagem para o componente que realiza a segmentação semântica, o qual por sua vez manda a imagem segmentada para o componente de mapeamento que por sua vez gera o occupancy grid. Em seguida, o OG é enviado para o componente de percepção das pistas, o qual envia as informações geradas para o componente de decisão de comportamento que por sua vez gera um goal state e o manda para o componente de planejamento de movimentação. O PM usa o OG e o goal state para gerar uma trajetória, a qual é enviada para o componente de controle que junto com as informações processadas da odometria, do GPS e da IMU envia comandos para os motores e os servos via Arduino. Os dois retângulos maiores simbolizam que os componentes dentro fazem parte do mesmo programa/pacote ROS. O estimador de estado do controle fornece ao componente de decisão de comportamento e ao componente de planejamento de movimentação uma estimativa do estado do veículo. Por fim, os checkpoints são fornecidos pelo componente de roteamento que está representado de forma diferente pois durante a navegação não existe realmente um componente de roteamento executando, há apenas os checkpoints gerados por esse componente.

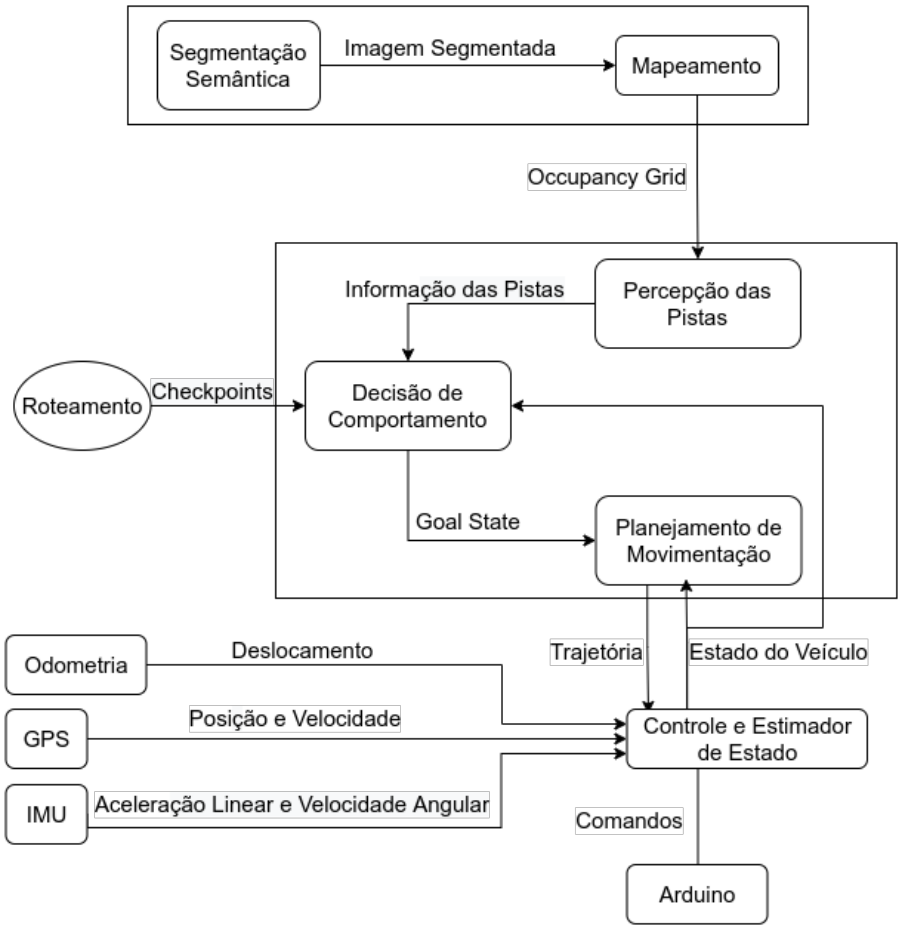


Figura 14 – Fluxo de informação e componentes do SNA do Crawler.

3.3.1 Arduino

O código presente no Arduino serve, apenas, como uma interface entre os atuadores e os comandados enviados pela Raspberry. O Arduino recebe por i2c a intensidade desejada para ser passada para os motores e o ângulo de rotação das rodas dianteiras (deixamos as rodas traseiras alinhadas com o Crawler). Esses sinais são passados por PWM. No Arduino, há um detalhe a mais que vale apresentar, a conexão entre a Raspberry e o Arduino pode falhar (o que infelizmente já aconteceu e causou um acidente). Portanto, para que o Arduino não fique enviando sinais espúrios para os atuadores, por motivos de segurança, depois de determinado tempo sem receber os comandos por i2c, o Arduino começa a parar a veículo até que um comando seja recebido da Raspberry novamente.

3.3.2 Mensagens ROS

Há seis tipos de mensagens ROS publicadas em todo o SNA e todas possuem carimbo de tempo e um identificador o que ajuda a analisar as informações após a navegação. Uma das mensagens é a intensidade da aceleração passada para o motor, cujos valores variam de 0 a 255. Outra mensagem é o ângulo das rodas ligado à direção do Crawler, cujos valores variam de 0 a 90. Uma das mensagens possui as informações obtidas da IMU (aceleração linear, velocidade angular, orientação dado por quatérnios e por roll, pitch e yaw). Uma das mensagens corresponde ao deslocamento obtido pelas medições do odômetro. Outra corresponde às informações obtidas pelo GPS. A última mensagem é a trajetória gerada pelo componente de planejamento de movimentação. Todas essas mensagens são publicadas na rede por meio de ROS publishers. Vale notar que todas as outras informações que transitam pelos componentes do SNA não utilizam ROS publishers.

3.3.3 Interface com controle remoto e Raspberry

Para poder realizar testes com o Crawler desconsiderando o componente de controle escolhemos usar um controle remoto para comandar a aceleração e a direção do Crawler, usamos um controle de Xbox One ligado, por USB, a um computador que através do pacote ROS joy_linux publica mensagens com os comandos dados pelo controle remoto, as quais são recebidas pela Raspberry e repassadas para o Arduino em seguida. Aqui, há uma medida de segurança similar à medida presente no Arduino; caso a conexão com o controle seja perdida, a Raspberry vai parando o veículo.

3.3.4 Interface do Encoder e IMU

Cada uma dessas interfaces é um pacote ROS. A interface com o encoder recebe por i2c o ângulo de rotação de uma peça acoplada ao eixo traseiro. A interface converte esse ângulo

em deslocamento levando em conta as rotações das rodas traseiras. Já a IMU possui dois pacotes ROS, um para inicializar a IMU e outro para receber as informações. Durante a inicialização precisamos configurar algumas características do relógio da IMU e da extensão dos valores que serão medidos pela IMU, dentre outras configurações. A IMU possui um giroscópio e um acelerômetro, usamos as informações fornecidas por esses sensores para calcular a aceleração linear e a velocidade angular do Crawler, além disso podemos obter estimativas do yaw, pitch e roll. Os dados do encoder são amostrados com 100Hz, já os dados da IMU são amostrados com 50Hz. Precisamos deixar a frequência de amostragem dos dados do encoder maior, pois, às vezes, dependendo da velocidade de rotação das rodas, não é possível saber se houve uma rotação completa entre uma amostra e a seguinte. Devemos notar que decidimos deixar as interfaces com os sensores na Raspberry para não ter um atraso a mais quando formos publicar os dados dos sensores, o que aconteceria se deixássemos a interface no Arduino ou até mesmo na Nano (lembremos que a versão do ROS na Nano é a Eloquent e não a Foxy - versão presente na Raspberry).

3.3.5 Controle

O módulo de Controle e de Localização (também chamado de estimador de estados) foi desenvolvido por Matheus Wagner, um integrante do projeto do Crawler. Por os meus conhecimentos teóricos de controle serem ínfimos e por ter sido ele quem desenvolveu todo o módulo de Controle, os parágrafos a seguir desta seção foram escritos por ele.

O objetivo do sistema de controle é utilizar as medidas dos sensores para determinar os comandos que serão enviados aos atuadores. Este sistema pode ser dividido em um conjunto de componentes, um estimador de estado, um controlador da dinâmica lateral do veículo e um controlador da dinâmica longitudinal do veículo. O controlador recebe como entrada não só os sinais medidos pelos sensores, mas também uma trajetória expressada em termos de orientação, velocidade longitudinal e posição, lateral e longitudinal, no sistema de referências global. Como deseja-se que o veículo passe por todos os pontos da trajetória, o sinal de referência só é atualizado quando o erro de rastreamento é menor do que um determinado valor.

Para cada um dos componentes utiliza-se um modelo físico do veículo diferente, um modelo da dinâmica lateral, um modelo da dinâmica longitudinal, desacoplados um do outro, utilizados no projeto dos seus respectivos controladores, e um modelo que combina dinâmica lateral e longitudinal que é utilizado pelo estimador de estados. Os modelos utilizados foram baseados em (RAJAMANI, 2012). A determinação dos parâmetros modelo é realizada através de um procedimento de identificação de sistemas offline utilizando o método dos mínimos quadrados baseado em dados experimentais.

Para realizar a estimação de estados utiliza-se o filtro de Kalman de Cubatura (ARASARATNAM; HAYKIN, 2009), uma vez que o modelo que combina as dinâmicas lateral e longitudinal é não-linear, logo um filtro de Kalman convencional não deve ser utilizado, e o filtro de Kalman de Cubatura não exige a linearização do modelo, prevenindo instabilidades

numéricas.

O controlador da dinâmica longitudinal do sistema é um controlador PID com feed-forward dos sinais estimados de yaw rate e de velocidade lateral para cancelar os efeitos de perturbação que essas grandezas podem gerar na dinâmica longitudinal do sistema quando o yaw rate é diferente de zero.

O controlador da dinâmica lateral exige um tratamento mais complexo uma vez que a dinâmica lateral varia em função da velocidade longitudinal. Dessa forma, optou-se por lidar com este sistema linear variante no tempo através de um controlador que realiza o posicionamento dos polos do sistema em malha fechada antes de cada atualização dos sinais de controle, determinando a matriz de ganhos do controlador por realimentação de estados. Note que para que esse procedimento seja realizado, assume-se que a velocidade longitudinal permanece constante durante um período de amostragem. Logo é necessário garantir que a taxa de atuação do controlador longitudinal seja maior do que a taxa de atuação do controlador lateral, de forma que na maior parte do período de amostragem a velocidade longitudinal do veículo esteja próxima da velocidade desejada, velocidade esta utilizada para determinar os ganhos do controlador da dinâmica lateral.

3.3.6 Segmentação Semântica e Mapeamento

Nesta seção, apresentamos a rede utilizada para segmentação semântica e como foi realizado o seu treinamento. Em seguida, discutimos como integramos a captura das imagens e a segmentação semântica. Depois, abordamos como geramos um mapa do ambiente com a imagem segmentada e, por fim, o envio das informações obtidas para o módulo seguinte.

A rede neural convolucional usada para realizar a segmentação semântica é uma HRNetV2 (WANG et al., 2019). O treinamento da rede foi feito utilizando o método fit_one_cycle da biblioteca FastAI. O dataset utilizado foi o RTK Dataset (RATEKE; WANGENHEIM, 2021). Nesse dataset a segmentação semântica possui 12 classes, sendo estrada asfaltada, pavimentada e de terra, e falhas na pista algumas dessas classes. Realizamos alguns testes com a Lite-HRNet e com a U-Net também.

Para integrar a captura das imagens com o processamento feito pela HRNet, utilizamos uma biblioteca da Nvidia chamada jetson-inference. Essa biblioteca faz com que a rede neural seja processada pela GPU da Nano acelerando o processamento das imagens e fazendo com que cada imagem leve entre 95 e 160 milissegundos (a maioria dos valores ficam bem próximos de 160 e uma outra grande parte dos valores ficam próximos de 95) para obtermos a saída da HRNet (que ainda não é a imagem segmentada e sim a ”probabilidade”de cada classe para cada pixel). Porém, infelizmente, a geração da imagem segmentada é feita fora da GPU causando um acréscimo de aproximadamente 195 milissegundos. Esse pós-processamento poderia ser acelerado pela GPU, entretanto teríamos que fazer algumas modificações na biblioteca jetson-inference. Em cada iteração fazemos mais do que apenas obter a imagem segmentada, salvamos a imagem original e a segmentada; geramos, enviamos e salvamos o occupancy grid;

além de mais alguns pequenos detalhes. Em geral essas operações a mais demandam por volta de 100 milissegundos, totalizando, em geral, aproximadamente 450 milissegundos por iteração. É extremamente raro passar de 500 milissegundos. Houve diversos experimentos em que centenas de imagens foram processadas e nenhuma iteração passou de 500 milissegundos. Devemos notar que esses valores são para imagens RGB 800x600. Outro detalhe é que a imagem segmentada gerada possui 400x300 pixels.

Como só estamos usando uma câmera monocular para obter informações do ambiente, o próximo passo relacionado aos dados visuais é gerar uma representação do ambiente. No caso do Crawler, optamos por usar a imagem segmentada para gerar um occupancy grid. Porém, como uma imagem não nos fornece informações suficientes para representarmos um ambiente 3D, decidimos representar o ambiente como um occupancy grid 2D e assumimos que tudo visto na imagem está no nível da pista e a altura de tudo na imagem é zero. Essas suposições trazem algumas consequências, como alguns objetos na pista com certa altura ocuparão mais do occupancy grid (OG) quando comparados com quanto ocupam no mundo real e objetos que estão posicionados a uma certa altura da pista aparecerão estarem bem mais à frente quando comparados com suas reais posições. Porém, desconsiderando objetos acima da pista que seriam obstáculos que realmente limitariam a movimentação do Crawler, as suposições feitas não geram áreas representadas como livres no OG que estão, na verdade, ocupadas; as suposições, apenas fazem com que haja mais áreas ocupadas que existem realmente.

Explicamos para que a imagem segmentada é utilizada e as consequências das suposições feitas, mas resta abordar como o occupancy grid representa o ambiente e como criamos o OG a partir da imagem. Com as suposições apresentadas anteriormente, o OG representa uma visão de cima da pista e perpendicular ao plano da pista. A área que decidimos representar no OG é dada por um retângulo com 10 metros de comprimento, 8 metros de largura, alinhado com o yaw do Crawler e inicia a 1 metro a partir da frente do Crawler. Certamente, esses valores não são os melhores, porém permitem a obtenção de bons resultados para o restante do SNA. Ademais, devemos notar que esses valores dependem muito da velocidade do Crawler, da câmera e da sua posição e orientação, das consequências ao tempo de processamento e à qualidade dos resultados de outras partes do SNA que dependem do OG, dentre diversos outros fatores. Fizemos diversos testes para chegarmos a esses valores e, frequentemente, fazemos modificações. Cada célula do OG possui 5 cm de comprimento e largura no mundo real, portanto o OG é uma matriz 200x160. Voltando o foco à criação do OG, para popularmos cada célula do OG com a informação de que se essa área é não navegável ou navegável, fazemos o seguinte: obtemos o centro de cada célula do OG em coordenadas 3D. Depois, projetamos cada um dos centros na imagem segmentada e obtemos coordenadas em pixels. Se o pixel está fora da imagem ou se a classe do pixel na imagem segmentada representa uma área não navegável, a respectiva célula do OG é considerada não navegável; caso contrário, a célula do OG é considerada navegável. Para fazer a projeção dos pontos, usamos o modelo de câmera com distorção de Brown-Conrady; sendo que os parâmetros intrínsecos da câmera foram estimados (esse processo é conhecido por calibração da câmera) a partir de fotos tiradas da Figura

15 impressa e os parâmetros extrínsecos foram obtidos da seguinte forma: medimos a altura da câmera no Crawler; admitimos que o yaw da câmera é o mesmo que o do Crawler, que o roll e o pitch não são alterados e que o roll é zero; o pitch estimamos usando marcações no piso e comparando a posição da marcação no mundo real com a marcação na projeção. O modelo de Brown-Conrady considera as distorções que uma imagem capturada pode apresentar. Devemos notar que as câmeras reais não são equivalentes ao modelo de câmera de pinhole, elas geram algumas diferenças na imagem que nesse caso consideramos distorções. Dependendo da intensidade dessas distorções, o occupancy grid pode ficar bem distante da realidade. Decidimos, inicialmente, não utilizar um OG probabilístico que levasse em conta informações de iterações passadas porque não queríamos depender da localização do Crawler que ainda não estava madura o suficiente para utilizarmos.

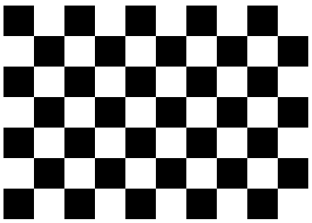


Figura 15 – Imagem usada para calibrar a câmera.

Com o OG gerado, o enviamos por TCP via Ethernet para a Raspberry para que a percepção das pistas seja feita. Decidimos não usar ROS para enviar o OG, pois o ROS presente na Nano é o Eloquent e o presente na Raspberry é o Foxy (não pudemos usar a mesma versão do ROS por causa de limitações ligadas à versão do sistema operacional e algumas bibliotecas utilizadas) e não há garantias de que a comunicação entre nós ROS de versões diferentes funcione.

3.3.6.1 Alguns resultados e considerações

Na Figura 16 exibimos alguns exemplos da segmentação semântica e do occupancy grid. Podemos perceber que a segmentação semântica está longe de ser perfeita, partes mais distantes e muito próximas da via são classificadas como background, às vezes buracos/falhas são adicionados em locais que não existem, muitas vezes há uma confusão entre via asfaltada com via pavimentada não asfaltada e sombras estão prejudicando bastante a qualidade da segmentação semântica. Há realmente muito a melhorar na segmentação semântica e ressaltamos que há várias questões relacionadas com a segmentação semântica que decidimos não abordar por estarem fora do escopo do trabalho.

Passando para a geração do OG, um ponto muito importante que talvez possa parecer que não impacta tanto é o pitch da câmera. Mudanças de apenas 5 graus nesse ângulo impactam muito no quão distante achamos que partes da via estão. Ilustramos esse impacto na Figura 17. Podemos perceber diferenças de aproximadamente 5 metros na localização de certas regiões

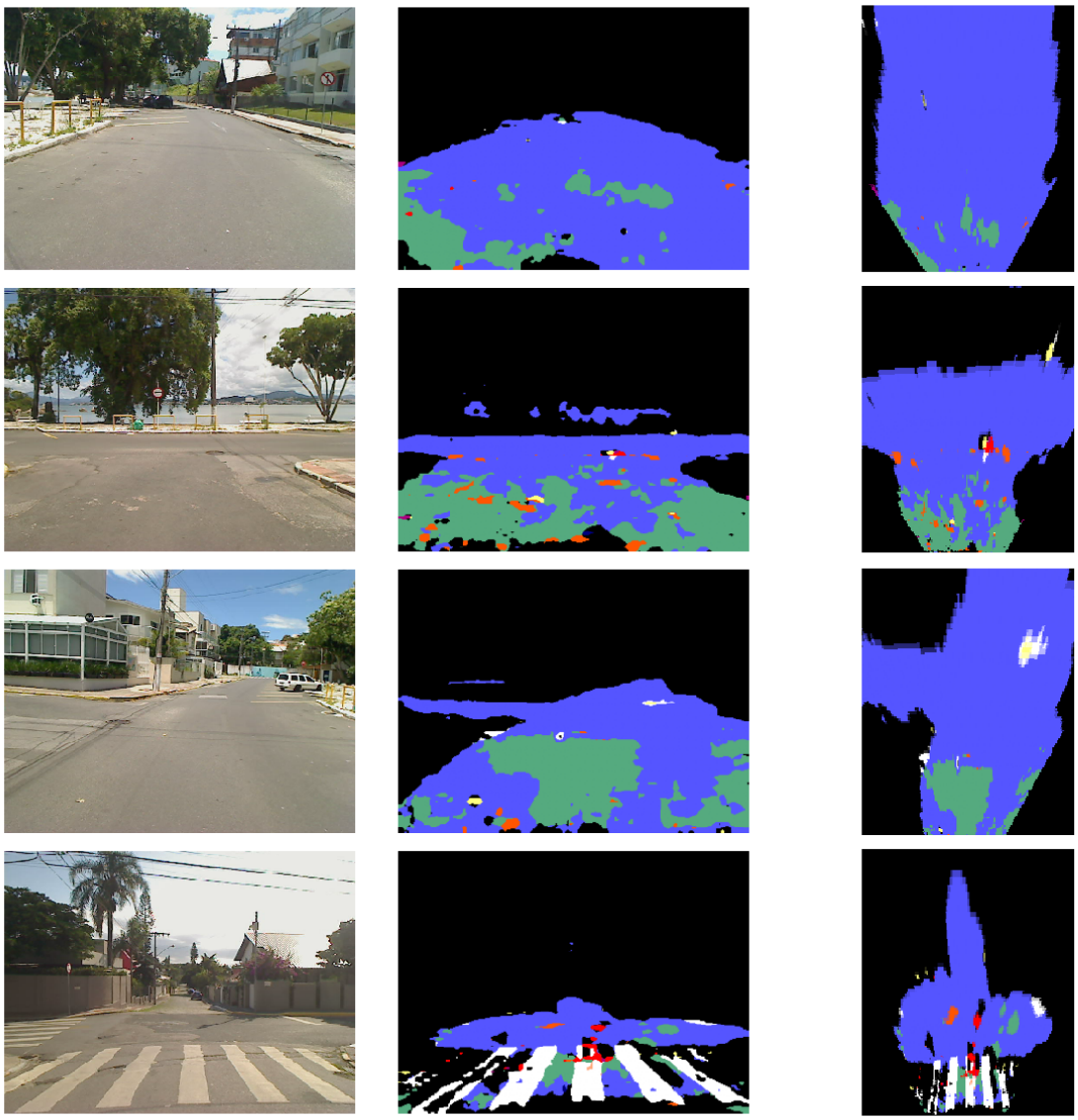


Figura 16 – Exemplos da segmentação semântica e geração do occupancy grid. Na esquerda estão as imagens originais. No centro, estão as imagens segmentadas (máscaras). Na direita, estão os OGs. A cor mais para um azul/roxo representa uma via asfaltada; o verde representa uma via pavimentada não asfaltada; o branco representa sinalizações na horizontais na via; o vermelho representa buracos/falhas na via e o preto representa qualquer coisa que não é de interesse (background). Há ao todo 12 classes, porém as mencionadas são as que mais aparecem nas imagens de exemplo.

no OG, apesar de estarmos considerando apenas 5 graus de diferença. Portanto, uma boa estimativa do pitch da câmera é fundamental para a geração de um OG que represente de forma fidedigna o ambiente. Todavia, o ajuste do ângulo da câmera é feito manualmente o que dificulta esse processo de ajuste. Ademais, por causa da suspensão do Crawler e também por a via apresentar imperfeições o pitch da câmera sofre alterações durante a navegação, trazendo ainda mais dificuldades para a estimativa do pitch. Ideias que pensamos para mitigar esses problemas são: ajustar o pitch da câmera antes de iniciar a navegação usando marcações no chão com uma certa distância conhecida para auxiliar a estimativa inicial do pitch e usar dados da IMU para saber o pitch do Crawler quando uma imagem for capturada.

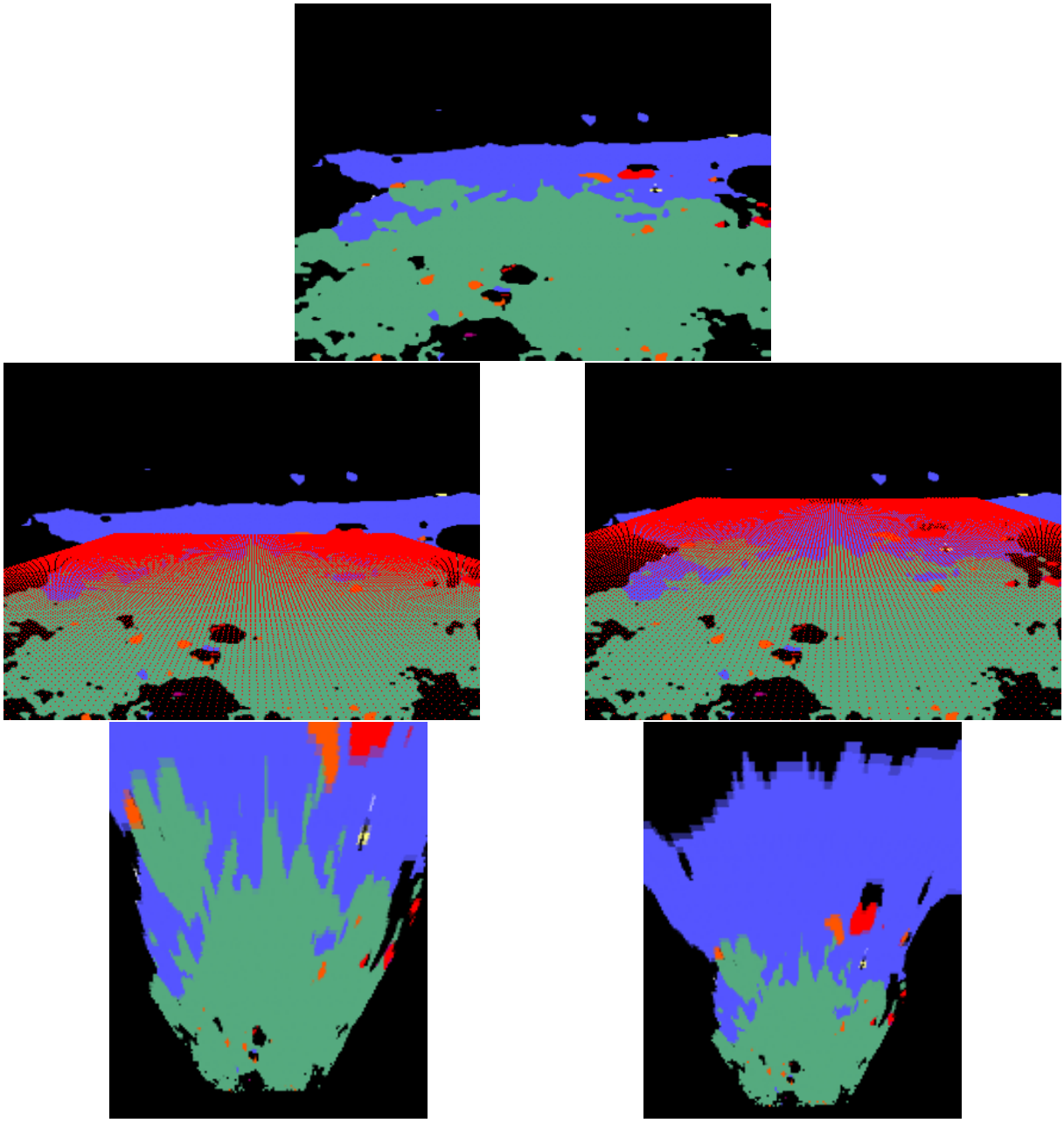


Figura 17 – Exemplo da influência do pitch da câmera na geração do occupancy grid. Na primeira linha está a máscara. Na segunda linha estão, da direita para esquerda respectivamente, a máscara com o OG projetado sobre a imagem considerando 5 e 10 graus de pitch para baixo da câmera (cada ponto vermelho representa a localização de sua respectiva célula na imagem segmentada - atenção para não confundir os buracos/falhas com os pontos vermelhos). Na última linha estão os OGs gerados.

3.3.7 Roteamento

O roteamento é feito manualmente por meio da escolha de coordenadas geográficas que representam checkpoints. Esses checkpoints devem ser escolhidos de forma que indiquem sempre um caminho para o Crawler, por exemplo: se houver uma bifurcação ou um entroncamento logo a frente do Crawler, deve haver um checkpoint que permita o Crawler saber por qual pista ele deve seguir; se houver uma interseção em T, também deve haver um checkpoint indicando para qual lado o Crawler deve seguir. Desenvolvemos uma aplicação simples que usa a API do Google Maps para facilitar a escolha dos checkpoints.

4 DETECÇÃO DE PISTAS DE TRÂNSITO E ESCOLHA DE GOAL STATE SEM MAPA HD

Neste capítulo, abordaremos detalhadamente uma parte do módulo de Percepção e o módulo de DC do Crawler. Mas por que vamos dar tanto enfoque para esses dois módulos, visto que o foco inicial deste trabalho era em SNAs em geral e em planejamento de movimentação? O principal motivo é que nos deparamos com um problema interessante para o qual precisávamos de uma solução para que pudéssemos seguir com o desenvolvimento do SNA, porém não encontramos nenhuma solução que se encaixasse bem com o ODD do Crawler - dentre algumas das principais características que inabilitaram diversas soluções existentes estão: o Crawler deve poder se locomover em ambientes não urbanos (sem demarcações na via), o Crawler não possui nenhum mapa prévio e a única fonte de informação do estado do ambiente onde o Crawler se encontra é a sua câmera monocular. Assim, neste capítulo apresentamos como moldamos parte do módulo de Percepção (responsável pela detecção/representação das pistas de trânsito) e o módulo de DC (responsável pela escolha do goal state) para solucionar o nosso problema. Começamos apresentando o problema, em seguida, descrevemos algumas ideias de soluções. Escolhemos uma das ideias e vamos discutindo como tornar essa ideia em uma solução e, conforme avançamos, apresentamos resultados de experimentos realizados. Seguimos aos poucos esculpindo e dando forma a solução. Por fim, comentamos em detalhes as motivações frente algumas escolhas tomadas para desenvolver a solução e exibimos exemplos de aplicação da solução.

4.1 O PROBLEMA

O problema que estamos tentando resolver é como encontrar um goal state que respeite as condições impostas pelo pelo módulo de roteamento (precisamos fornecer uma sequência de goal states que permita ao crawler seguir o direcionamento dado pelo roteamento) e que seja adequado para o motion planner (o goal state precisa pertencer ao free state space e deve ser alcançável a partir do seu respectivo start state). Agora, a seleção dos checkpoints do roteamento é feita manualmente a partir da seleção de coordenadas usando o Google Maps, o que nos permite escolher os pontos de forma que facilite o direcionamento do Crawler para as ruas que desejamos, porém não temos garantia que haverá algum ponto selecionado que pertencerá ao occupancy grid map após a segmentação semântica e nem garantia que o Crawler esteja realmente no local em que ele pensa que está, o que pode ”deslocar”o checkpoint, no ponto de vista do Crawler, da sua real posição.

Agora, vamos abordar em mais detalhes porque não podemos garantir que cada checkpoint esteja sempre no free state space do OG gerado em cada iteração. Quando marcamos um checkpoint usando o Google Maps, não temos como saber se há algum obstáculo naquele exato local, como um carro estacionado. Além disso, o próximo checkpoint não, necessariamente, estará na área confinada pelo OG, o Crawler pode precisar desviar de um obstáculo no meio

da pista e durante essa manobra ele perde de vista o checkpoint. Mas por que queremos tanto que os checkpoints estejam no free state space do OG? Porque, assim, a escolha do goal state que será passado para o planejamento de movimentação é, simplesmente, o próprio checkpoint. Como, então, escolheremos um goal state alcançável e pertencente ao free state space para o planejamento de movimentação?

Vamos voltar nossa atenção a problemas gerados por uma outra falta de garantia; a falta de garantia de uma excelente localização do Crawler. Imagine uma situação em que o Crawler está ante uma bifurcação e precisa decidir por qual das pistas ele vai seguir, ele acredita (pela sua estimativa de localização) que está em frente à pista que bifurca à esquerda, quando na verdade está em frente à pista que bifurca à direita. Por isso, o Crawler vai planejar a sua movimentação para a via errada e a navegação estará totalmente comprometida. Várias perguntas surgem com este problema. Talvez as primeiras a surgirem sejam perguntas como: O que podemos fazer para amenizar o erro de localização que paulatinamente cresce? Como podemos limitar o erro? Será que algum tipo de odometria visual deve ser usada junto a mapas HD? Perguntas assim, infelizmente, fogem do escopo deste trabalho e encontram-se mais voltadas ao módulo de localização. Para planejarmos a movimentação devemos saber que erros de localização existem e devemos tentar contorná-los para que a navegação não seja comprometida e, assim, surgem perguntas como: Sabendo que o próximo checkpoint está provavelmente em tal região, como escolher um goal state que guie o Crawler para o caminho correto?

4.2 SOLUÇÕES

Dentre as duas perguntas dos dois parágrafos anteriores, decidimos focar em responder, a princípio, a última pergunta. Assim, fizemos buscas por diversas fontes, mas não achamos respostas que caibam, perfeitamente, para o Crawler. Por isso, desenvolvemos nossas próprias ideias para solucionar o problema.

Encontramos duas opções, para ajudar a resolver nosso problema: (i) passamos a diferenciar pistas e escolhemos goal states que nos permitam seguir pela pista escolhida pelo componente de roteamento, sendo que os checkpoints não definirão diretamente os goal states e sim as vias. (ii) Aumentamos a quantidade de checkpoints e tentamos marcá-los em locais que possam ser identificados durante a navegação para que o Crawler tenha mais indicativos de por onde seguir. Ainda discutindo a ideia (ii), diminuir a distância entre os checkpoints faz com que precisemos fazer ainda mais ajustes da posição dos checkpoints, sendo que talvez uma das formas menos complexas seja tentar marcar os checkpoints (durante o roteamento) sempre no centro das pistas e depender dessa informação para fazer ajustes de posição dos checkpoints (e de localização do Crawler). Para isso dependeríamos, ainda mais, da visão do Crawler para detectarmos centros da pista, por conseguinte precisaríamos diferenciar pistas. Tanto para ideia (i) quanto para a ideia (ii) mapas HD do ambiente facilitam muito a detecção de diferentes faixas e pistas, porém como não temos um mapa HD dependemos de todo o segmento de visão do veículo para detectar corretamente diferentes pistas, portanto a qualidade da segmentação

semântica é um limitante dessas duas ideias para o Crawler. Decidimos seguir pela ideia (i) por não depender da seleção manual do roteamento para selecionar os centros das pistas, porém dependemos agora de uma localização do Crawler com erros não tão grandes durante toda a navegação.

4.2.1 Detecção de pistas

Vamos discutir agora como que fizemos para detectar as pistas de trânsito. Vale lembrar que como entrada para o detector de pistas temos o occupancy grid. Fizemos várias buscas em artigos para encontrar uma forma de detectar vias que fosse viável e condizente com toda a estrutura e o ODD do Crawler, porém não encontramos nada. Muito do que encontramos focava em detectar faixas em ambientes urbanos com boas sinalizações, já outros dependiam de informações advindas de mapas HD e alguns usavam LiDAR para detectar diferenças de altura entre a área navegável e a não navegável. Depois de várias buscas decidimos testar algumas técnicas de visão computacional para ver se poderíamos obter resultados satisfatórios e a técnica que se sobressaiu foi a esqueletonização. Podemos pensar na esqueletonização como um processo que recebe uma representação de algo e tenta gerar uma representação mais simplificada. A execução desse processo está ligada, muitas vezes, a uma ideia de afinamento sucessivo da representação até obtermos uma representação mínima que mantenha algumas propriedades da representação original como conectividade. Com o afinamento espera-se, muitas vezes também, que a representação simplificada seja dada pelos ”centros”da representação original. Essa definição se encaixa muito bem com o problema de detectar pistas e seus centros, pois um bom esqueleto para uma pista é o seu centro e se houver alguma bifurcação na pista, por exemplo, o esqueleto deve bifurcar também. Testamos três diferentes técnicas de esqueletonização: (ZHANG; SUEN, 1984), (GUO; HALL, 1989) e uma técnica criada originalmente por Lee Kamentsky como parte da biblioteca CellProfiler - as implementações usadas estão disponíveis na biblioteca scikit-image (os créditos dados a Lee Kamentsky pela terceira técnica foram dados pelo scikit-image). Antes de entramos em detalhes a respeito de cada técnica, vamos falar sobre algumas etapas antes da esqueletonização.

Lembremos que a Raspberry acabou de receber o occupancy grid da Nano e neste momento o OG é uma imagem binária 200x160, cada pixel representa uma área de 25 cm^2 (5x5 cm) e as linhas representam o ambiente a frente do Crawler e as colunas do OG o ambiente aos lados. Outro detalhe é que o OG começa 100 cm à frente do Crawler e se estende por uma área de 800000 cm^2 (200 vezes 5 vezes 160 vezes 5) ou 80 m^2. Primeiramente usamos a operação morfológica de abertura sendo o kernel a matriz quadrada 5x5 composta de apenas 1s. Vale notar que a operação de abertura é usada para remover pequenas sobras de áreas navegáveis que provavelmente são falhas da segmentação semântica. Em seguida aplicamos a operação morfológica de fechamento com o kernel em formato de círculo com raio 12. Essa operação de fechamento serve para tapar alguns buracos que provavelmente são falhas advindas da segmentação semântica. Além disso, o fato de o kernel ser em formato de círculo ajuda a evi-

tar que algumas bordas do OG fiquem muito retas. Essas duas operações servem apenas como um pré-processamento para ajudar a tornar a detecção das pistas de trânsito menos ruidosa. Na Figura 18 podemos ver alguns exemplos da aplicação dessas duas etapas. Vale notar que estamos utilizando valores bem grandes para os tamanhos dos kernels e que seria muito mais adequado reduzir os kernels e melhorar a qualidade da segmentação semântica. Ademais, frequentemente mudamos os tamanhos dos kernels em nossos experimentos, assim, não devemos nos apegar muito a esses valores específicos.

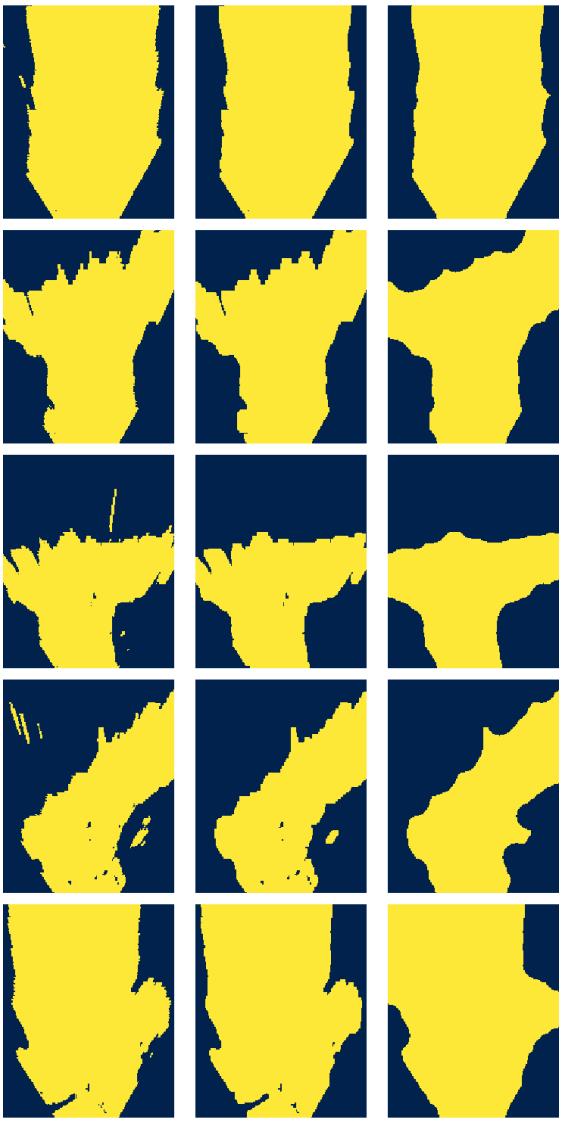


Figura 18 – Exemplos da aplicação das operações de morfologia matemática de abertura e fechamento para occupancy grids. Na primeira coluna temos o OG sem modificação. Na segundo coluna podemos ver o OG depois de aplicarmos a operação de abertura. Na terceira coluna podemos ver o resultado de aplicarmos a operação de fechamento no OG da segunda coluna.

4.2.1.1 Esqueletonização

Agora podemos voltar para os três métodos de esqueletonização que testamos. Comecemos pelo método de (ZHANG; SUEN, 1984) (ZS). Esse método possui duas fases, na pri-

meira ele passa por todos os pixels da imagem que estão ocupados e confere quatro condições, se todas as condições forem respeitadas o pixel deixa de estar ocupado ao fim da fase. Se pelo menos um pixel respeitou as quatro condições, a segunda fase é realizada da mesma forma como a primeira, com a diferença de que duas condições são trocadas. Todas as condições são simples e dependem apenas de alguns ou todos os oito vizinhos do pixel em análise. O método fica realizando as duas fases e alternando entre elas até que não haja pixel que respeite as quatro condições da respectiva fase, nesse momento o algoritmo para. Os pixels que sobraram são o esqueleto da imagem original.

Em (GUO; HALL, 1989), os autores criam dois novos métodos de esqueletonização A1 e A2 e comparam-os com mais métodos, inclusive com método descrito anteriormente, no qual o método A1 é baseado. A partir de agora, quando nos referirmos ao algoritmo desenvolvido por (GUO; HALL, 1989) (GH), estamos nos referindo ao A1, pois esse é o método que utilizamos em nossos testes. O GH é bem similar ao ZS, é, também, um método de esqueletonização em duas fases e depende dos oito vizinhos para um pixel ser eliminado, porém as condições do GH são diferentes e o algoritmo para apenas quando as duas fases não possuem mais pixels para eliminar. Nos testes expostos no próprio artigo, os autores apresentam dados que mostram que o A1 precisa de mais iterações que o ZS para terminar de executar, mas consegue eliminar mais pixels que não são estritamente necessários para a formação de um bom esqueleto dada por eles.

O terceiro método não possui um artigo (chamaremos esse método de MA - medial axis) e esse método é bem diferente dos outros dois. O MA não é baseado em sucessivas iterações por todos os pixels para gerar o esqueleto, na verdade esse método passa por cada pixel apenas uma vez para decidir se ele vai ser eliminado ou não. Por esse motivo, esse método não pode ser paralelizado como os dois métodos anteriores. Nesse método é calculada a distância euclidiana de cada pixel ocupado até a área não ocupada mais próxima. Além disso, é calculado quanto dos oito vizinhos de cada pixel estão ocupados. Em seguida, os pixels são ordenados com base nesses dois valores, o pixel estará mais no início da lista quanto mais próximo de uma área não ocupada ele estiver e quanto mais vizinhos ocupados ele possuir (a distância é prioritária no ordenamento em relação a quantidade de vizinhos ocupados). Posteriormente, cada um dos pixels dessa lista ordenada são analisados se serão eliminados ou não. Vale salientar que cada pixel será analisado apenas uma vez e a ordem da análise de cada pixel importa (o que não é verdade para os outros dois métodos), por isso esse método é considerado serial. O pixel é mantido ou não com base em duas condições apenas: (a) se remover o pixel muda o número de Euler da imagem, o pixel é mantido; (b) se o pixel possui dois ou menos vizinhos ocupados (dos oito), ele é mantido. Caso não satisfaça (a) nem (b), o pixel é eliminado. Na condição (a), falamos em número de Euler de uma imagem, esse conceito se refere a quantidade de objetos menos a quantidade de buracos na imagem; consideramos um objeto o conjunto de pixels ocupados conectados entre si (levando em conta os oito vizinhos) e o conceito de buraco é o equivalente só que para pixels não ocupados.

Antes de discutirmos os resultados da aplicação de cada um dos métodos, apresentare-

mos uma imagem da aplicação do ZS para ilustrar a esqueletonização. Na Figura 19, podemos observar um cavalo sendo representado por uma imagem binária à esquerda e à direita temos o resultado da esqueletonização.

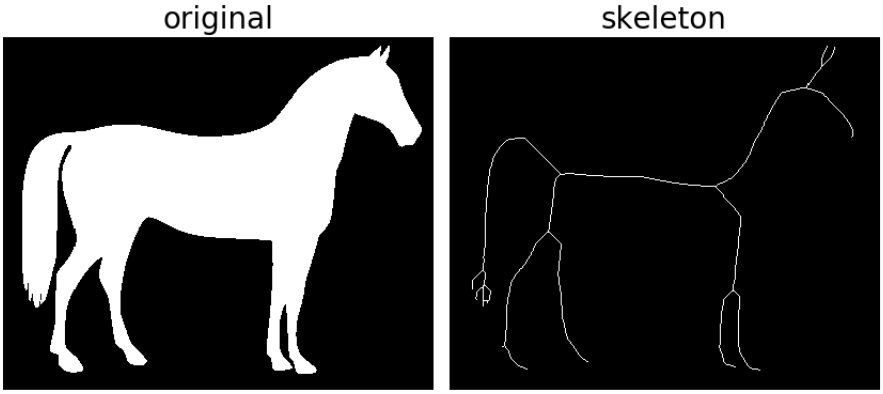


Figura 19 – Exemplo de esqueletonização. Fonte: scikit-image.

4.2.1.2 Resultados

Decidimos avaliar, principalmente, tempo de execução e representatividade da via. Para analisar a representatividade da via, não utilizamos nenhuma métrica, achamos que análises visuais dos resultados aliadas às análises temporais foram suficiente para tomarmos uma decisão em relação aos três métodos. Para comparar o tempo de execução dos três métodos, separamos imagens segmentadas com resultados variados. Já para a análise da representatividade, procuramos testar cada método com vias de diferentes formatos.

Vamos começar os apresentando alguns resultados de como o MA, o ZS e o GH se comportaram com vias (apesar de o certo ser ”com a parte visível das vias”, chamaremos apenas de via por simplicidade) sem qualquer entroncamento (chamaremos de vias simples), com bifurcação, com cruzamento em T e com cruzamento em X. Fizemos questão de obter várias imagens com tipos de vias além de vias simples, pois representam situações comuns com as quais o Crawler deve conseguir lidar e são situações nas quais é importante termos uma boa representatividade da via, visto que são nessas situações que o Crawler precisa tomar uma decisão de por qual pista seguir. Notemos, ainda, que as imagens tivemos que obter por conta própria, pois precisamos saber e definir uma altura e orientação para a câmera para a projeção sobre o OG, além de necessitarmos dos parâmetros da câmera utilizada por causa do mesmo motivo. Ademais, câmeras diferentes poderiam possuir diferentes campos de visão, o que poderia impactar na representatividade do OG e por conseguinte na esqueletonização. Por fim, buscamos trazer imagens que representam o comportamento geral de cada método de esqueletonização nos vários experimentos que realizamos (não estamos trazendo nenhuma exceção para representar algum método). Com isso tudo em mente, observemos as Figuras 20 e 21.

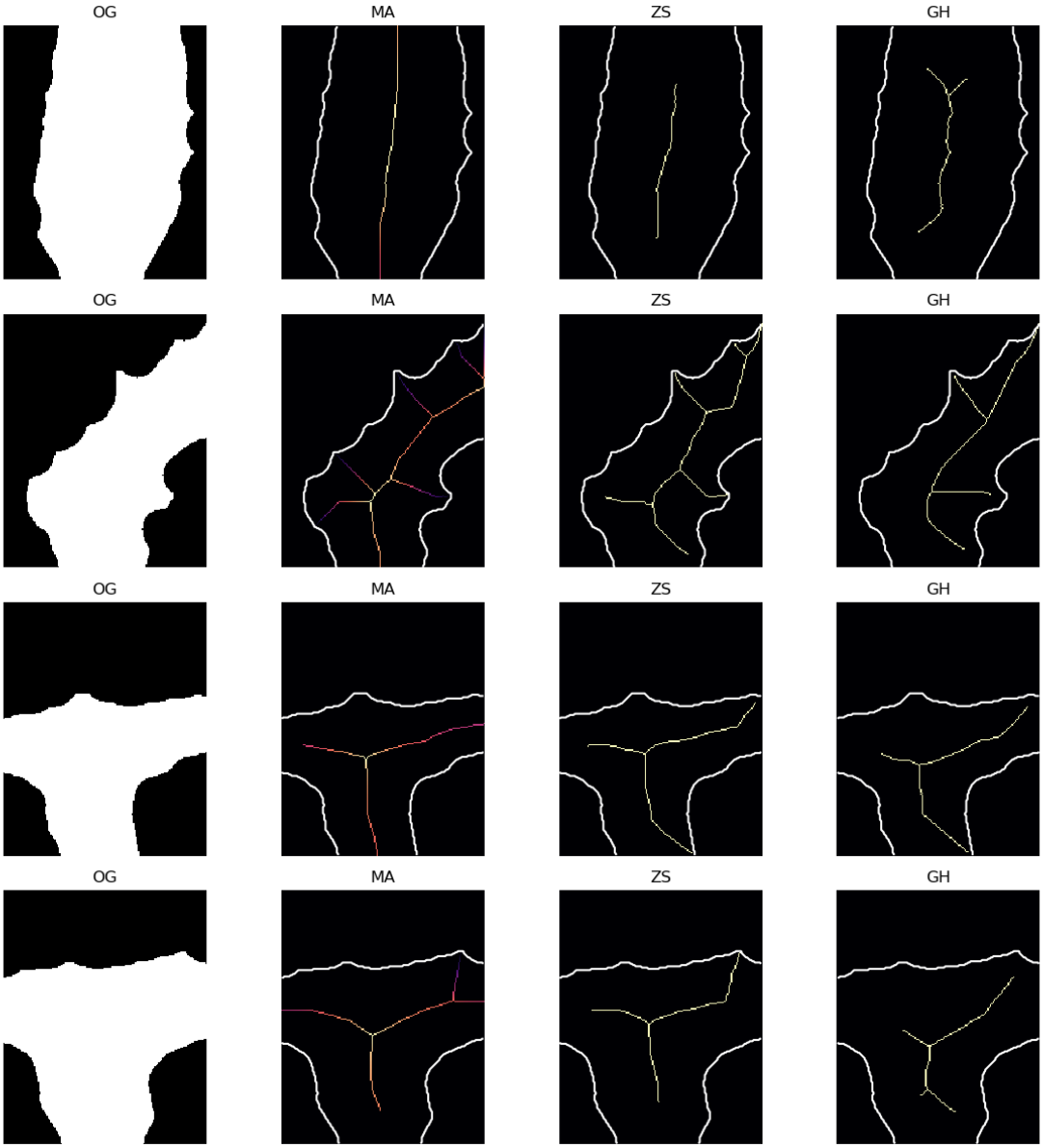


Figura 20 – Comparação entre os três métodos de esqueletonização abordados. As duas primeiras linhas são vias simples e as duas últimas são vias com cruzamento em T. Na primeira coluna apresentamos o OG pós operações de morfologia matemática e nas outras três apresentamos o resultado da aplicação de cada método. Decidimos adicionar adicionar as bordas da área navegável apenas para que pudéssemos ter uma referência melhor para analisarmos o esqueleto - as bordas brancas nas colunas dos métodos não fazem parte do esqueleto. Por fim, na coluna do MA, a cor de cada pixel do esqueleto é escolhida com base na distância das bordas - quão mais longe da borda mais claro o pixel.

Para discutir os resultados da aplicação dos métodos de esqueletonização selecionamos as Figuras 20 e 21 como exemplos. Podemos notar que dos três métodos, o que conseguiu traçar caminhos que mais se aproximam da representação simplificada que desejamos de cada pista foi o MA. Porém, o MA tende a criar alguns entroncamento a mais do que realmente existem. Podemos perceber, na verdade, que os três métodos tendem a ramificar demais o esqueleto e o método que exibe mais esse comportamento errôneo é o MA. Em geral, os esqueletos gerados pelo ZS e pelo GH são bem parecidos, com exceção de alguns casos em que o esqueleto do GH fica um pouco mais curto do que deveria em algumas ramificações e em casos mais raros

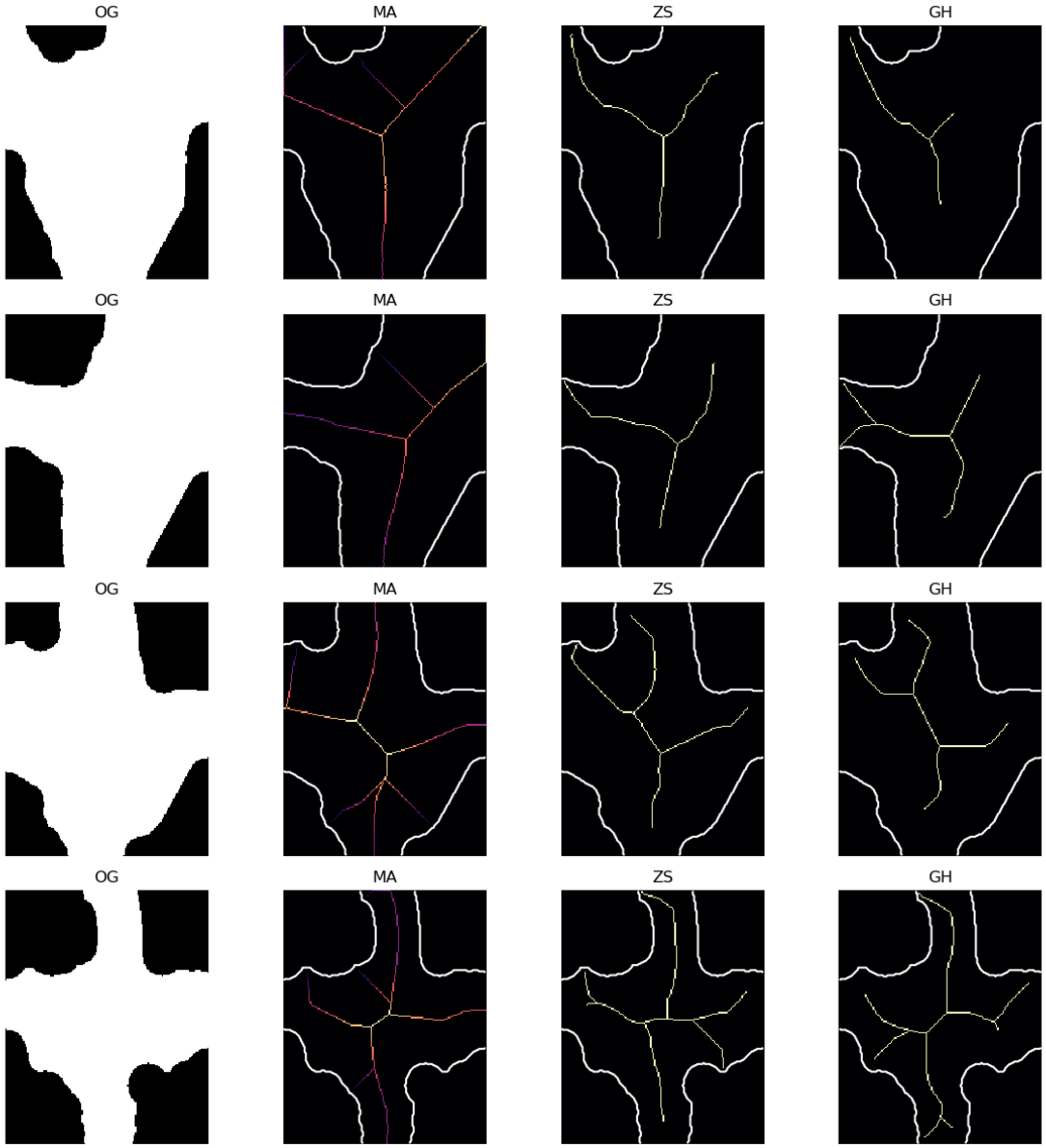


Figura 21 – Comparação entre os três métodos de esqueletonização abordados. As duas primeiras linhas são vias com bifurcação e as duas últimas são vias com cruzamento em X.

ainda, (não demonstrados nas Figuras 20 e 21) o GH não cria uma ramificação para algumas pistas. Ainda em relação ao ZS e o GH, em vários casos, conforme uma ramificação adentra uma pista, em geral próximo de bifurcações, o esqueleto desvia um pouco do meio da pista e vai se aproximando de uma das bordas. Ademais, podemos notar que o quão prejudicial falhas da segmentação semântica são para os três métodos de esqueletonização; diversas ”pontas”que acabam por sobrar mesmo após o pré-processamento com morfologia matemática fazem com que ramificações equivocadas do esqueleto sejam criadas ou, em alguns casos, ”puxam”para si partes do esqueleto. Na Figura 22 trazemos alguns exemplos de como o pré-processamento com morfologia matemática é importante para uma boa esqueletonização frente imagens segmentadas com tantas falhas.

Realizamos um teste na Raspberry para medir o tempo de execução dos três métodos.

Usamos 70 OGs nesse teste, sendo que cada OG era uma imagem de 200x160 pixels. Os OGs selecionados variaram bastante em formato e em quantidade de pixels ocupados. As médias e o desvio padrão do tempo de execução, assim como uma comparação com a média do método mais rápido estão na Tabela 2. Em geral, para cada um dos três métodos não houve uma notável diferença entre o tempo de execução para diferentes OGs, apenas para o ZS alguns OGs demoraram um pouco mais do que dobro da sua média. Vale a pena ressaltarmos alguns motivos de tamanha diferença entre o tempo de execução dos métodos, o ZS foi implementado praticamente todo em Cython que é usado para gerar código em C para aumentar a eficiência do código. Já o GH e o MA foram implementados em Python usando numpy e scipy, sendo que parte do MA está implementada em Cython também. Apesar do ZS e do GH serem métodos paralelizáveis, nenhuma das implementações usufruiu desse característica. Embora o MA analise apenas uma vez cada pixel, os pixels são analisados fora da ordem de organização da memória, o que pode ser uma das causas do seu atraso. Vale lembrar que, em (GUO; HALL, 1989), os testes feitos pelos autores mostram que a quantidade de iterações do GH não foi tão grande, quando comparada com o ZS, para demorar 30 vezes o tempo de ZS, o que indica que a forma como foram implementados os dois métodos influenciou no tempo de execução dos testes com os OGs. O mesmo teste foi executado em meu notebook e as proporções entre os tempos de execução de cada método foram diferentes. O MA levou 19 vezes o tempo do ZS e o GH levou 23 vezes o tempo do ZS. Além disso, os métodos ZS, GH e MA executaram, aproximadamente, 12, 16 e 15 vezes mais rápido respectivamente. Outro detalhe que vale ressaltar é que o tempo para calcular a distância de cada pixel até a borda (técnica usada no MA, cujo resultado pode auxiliar na escolha do goal state) foi, em média, 0.02026 na Raspberry, cerca de metade do tempo do ZS, portanto podemos obter essa informação sem grandes atrasos.

Tabela 2 – Média e desvio padrão em segundos dos tempos de execução dos três métodos de esqueletonização testados. Também mostramos uma comparação da média dos métodos com a média do método mais rápido, que neste caso é o ZS.

	Média	Desvio Padrão	Média/Média_ZS
ZS	0.04569	0.01778	1
GH	1.40689	0.27759	30.79208
MA	1.03322	0.14314	22.61370

Com base nos resultados obtidos, tanto das medições do tempo de execução quanto das análises visuais dos esqueletos, decidimos usar o ZS. Como observamos nos exemplos, os esqueletos do ZS e do GH não ficaram muito diferentes, com exceção de algumas ocasiões quando o GH acabou encurtando demasiadamente algumas ramificações. Portanto, a enorme diferença entre o tempo de execução foi o fator decisivo entre esses dois métodos. Poderíamos ter implementado o GH usando Cython para que pudéssemos ter uma comparação mais justa do tempo de execução, porém decidimos que não valeria a pena, pois nos testes executados pelos autores o GH apresentou mais iterações do que o ZS e achamos os resultados do ZS levemente

melhores. Agora, comparando o MA e o ZS, o fato de o MA já ter uma boa parte do seu código implementado em Cython, além de não ser paralelizável, acaba diminuindo a margem de tempo que podemos obter com otimizações do código. Isso aliado ao tempo de execução de mais de meio segundo desfavoreceu em muito o MA. Portanto, mesmo com os bons resultados dos esqueletos gerados pelo MA, optamos pelo ZS. Devemos salientar, novamente, que o tempo médio do ZS foi muito bom, sendo que nem desfrutamos da sua capacidade de paralelização o que pode acelerar em muito o método. A partir de agora, sempre que comentarmos a respeito de algum esqueleto, estaremos falando de algum esqueleto gerado pelo ZS.

4.2.2 Escolha do goal state

A Raspberry recebe o OG, retira um pouco do ruído com morfologia matemática e aplica a técnica de esqueletonização para possuir uma representação mais útil para selecionar um goal state. Porém, ainda não temos o goal state. Lembremos que os checkpoints, agora, apenas avisam por quais caminhos o Crawler deve seguir (eles não indicam por qual ponto específico da via o Crawler deve passar). Além disso, a esqueletonização nos dá uma boa ideia de onde o meio da área navegável é e os caminhos que existem. Podemos, também, usar a informação da distância de cada pixel à borda. Então, o que decidimos de heurística para guiar o Crawler em direção ao checkpoint é um método que depende da proximidade ao próximo checkpoint. Se o Crawler está longe de um checkpoint, ele, simplesmente, escolhe como goal state o ponto (pertencente ao esqueleto, com uma certa distância das bordas, alcançável e não tão próximo da borda quanto outros pontos) que está mais longe dele. Agora, se o Crawler está perto de um checkpoint, ele seleciona um ponto (pertencente ao esqueleto, com uma certa distância das bordas e alcançável) que está mais perto do checkpoint e, em seguida, traça um caminho desse ponto ao checkpoint (estendendo o esqueleto) e seleciona como goal state o ponto mais próximo do checkpoint - esse ponto deve estar a uma certa distância da borda e deve ser um ponto antes de, eventualmente, cruzar a borda. Vamos lembrar que o goal state pode não ser apenas uma posição, a orientação e a velocidade, por exemplo, podem compor o goal state também, o espaço do goal state depende do espaço do planejamento de movimentação. Explicações mais detalhadas da inclusão desses e outros componentes no goal state serão apresentadas no capítulo de PM. Por enquanto, imaginemos que o goal state é composto pela posição e orientação do veículo. Com isso em mente, a orientação é definida por alguns pixels do caminho do pixel do goal state até alguns pixels retrocedendo no caminho (do esqueleto e sua extensão) do pixel mais próximo do Crawler até o goal state.

Para ilustrar todo esse processo de obtenção do goal state, apresentamos a Figura 23. Como podemos perceber na primeira coluna (1) da imagem e pelas discussões anteriores, o resultado da esqueletonização não é perfeito, há ramificações a mais, algumas partes do esqueleto se aproximam demais das bordas, já outras são demasiadamente curtas apresentando um péssima representação da pista. Vamos retomar os passos, primeiramente, para caso haja checkpoint nas proximidades e, em seguida, sem checkpoint. A primeira coisa a se fazer é cortar o

pontos próximos demais das bordas e identificar o ponto do esqueleto mais próximo do Crawler, o chamamos de ponto inicial (2). Depois, percorremos os pixels do esqueleto a partir do ponto inicial e, assim, obtemos caminhos no esqueleto. Procuramos pelo ponto mais próximo ao checkpoint que pertence a um caminho e selecionamos o seu caminho (3), chamamos esse ponto de ponto próximo. Estendemos o caminho a partir do ponto próximo até o checkpoint ou até atingirmos uma distância em relação às bordas. O último ponto dessa extensão é a posição do goal state. Por último, obtemos a orientação do goal state (4). Agora, para caso estejamos longe de um checkpoint. Não há diferença alguma até obtermos os caminhos. O que fazemos, em seguida, é selecionar um ponto do esqueleto para ser o goal state; essa escolha é feita com base na distância em relação ao Crawler e às bordas. Selecionamos o caminho do ponto selecionado (3). Por fim, obtemos a orientação da mesma forma como se houvesse checkpoint (4). Para auxiliar o entendimento de todo o processo, fizemos dois algoritmos em alto nível que podem ser observados em Algorithm 1 e 2 (em ambos os algoritmos admitimos que o OG já foi pré-processado).

Algorithm 1 definição_do_goal_state_com_checkpoint(OG, estado_crawler, checkpoint)

$esqueleto \leftarrow esqueletonização(OG)$
$esqueleto \leftarrow cortar\_pixels\_próximos\_borda(esqueleto)$
$ponto\_inicial \leftarrow ponto\_mais\_próximo\_crawler(esqueleto, estado\_crawler)$
$esqueleto\_percorrido \leftarrow percorrer\_esqueleto(esqueleto, ponto\_inicial)$
$ponto\_próximo \leftarrow ponto\_mais\_próximo\_checkpoint(esqueleto\_percorrido, checkpoint)$
$caminho\_selecionado \leftarrow encontrar\_caminho(esqueleto\_percorrido, pronto\_próximo)$
$caminho\_estendido \leftarrow estender\_caminho\_selecionado(caminho\_selecionado, checkpoint, OG)$
$goal\_state.posição \leftarrow caminho\_estendido[caminho\_estendido.size() - 1]$
$goal\_state.orientação \leftarrow gerar\_orientação(caminho\_estendido)$

Algorithm 2 definição_do_goal_state_sem_checkpoint(OG, estado_crawler)

$esqueleto \leftarrow esqueletonização(OG)$
$esqueleto \leftarrow cortar\_pixels\_próximos\_borda(esqueleto)$
$ponto\_inicial \leftarrow ponto\_mais\_próximo\_crawler(esqueleto, estado\_crawler)$
$esqueleto\_percorrido \leftarrow percorrer\_esqueleto(esqueleto, ponto\_inicial)$
$ponto\_longe \leftarrow ponto\_longe\_crawler(esqueleto\_percorrido, estado\_crawler)$
$goal\_state.posição \leftarrow ponto\_longe$
$caminho\_selecionado \leftarrow encontrar\_caminho(esqueleto\_percorrido, pronto\_longe)$
$goal\_state.orientação \leftarrow gerar\_orientação(caminho\_selecionado)$

Agora, explicaremos alguns detalhes de implementação da escolha do goal state. Depois que obtemos o esqueleto, descartamos os pixels que ficam perto demais das bordas (pixels com distância da borda menor que 10 pixels (50 cm) são eliminados - essa distância depende da qualidade da segmentação semântica e se o Crawler vai navegar por caminhos muito estreitos), as distâncias de todos os pixels à borda são obtidas da mesma forma como o MA obtém essa informação. A partir de agora, quando falamos em pixels do esqueleto estamos nos referindo aos pixels não tão próximos da borda. Em seguida, para sabermos quais pixels do esqueleto

são alcançáveis, criamos um grafo não dirigido onde os pixels do esqueleto são os vértices e as arestas são dadas pela existência de pixels vizinhos (vizinhança de oito) pertencentes ao esqueleto. Como ponto de partida da busca no grafo escolhemos o pixel do esqueleto mais próximo ao Crawler (considerando distância euclidiana) e o marcamos como o primeiro pixel alcançável. Com isso, fazemos uma BFS (Breadth-First Search) pelo grafo, e assim, obtemos os pixels alcançáveis do esqueleto.

Dos pixels alcançáveis do esqueleto, selecionamos o mais longe do Crawler cuja distância das bordas é maior que a mediana das distâncias das bordas dos pixels alcançáveis para ser a posição do goal state caso o Crawler esteja distante do checkpoint (checkpoint fora do OG) e o mais próximo do checkpoint para auxiliar na escolha do goal state caso contrário (distância euclidiana para ambos os casos). Em seguida, extraímos o caminho do ponto inicial até o ponto final selecionado. Esse caminho é o caminho existente que possui menos pixels e podemos utilizá-lo para auxiliar, dependendo do método, o planejamento de movimentação. Apenas se o checkpoint estiver próximo, estendemos o caminho do ponto auxiliar até o mais próximo possível do checkpoint, a extensão é feita selecionando sempre o vizinho do ponto final mais próximo do checkpoint e que não esteja próximo demais da borda (10 pixels (50 cm)). Se não houver vizinho que respeita a distância em relação à borda e que seja mais próximo do checkpoint do que o último ponto, a extensão para e o último ponto é a posição do goal state. Por último, a orientação é dada pelo vetor que sai da posição do ponto com índice i (sendo $i = (\mathit{quantidade\ de\ pixels\ no\ caminho} * 0.75) - 1$ e o índice do pixel inicial igual a zero) e vai até a posição do goal state.

Por fim, apresentaremos as nossas motivações frente alguns detalhes da nossa ideia de definição do goal state. Ao eliminarmos os pixels próximos demais da borda diminuímos o risco de o Crawler passar perto demais de alguma borda (o mantendo próximo aos centros da pista, lugar onde, em geral, queremos que ele fique) ou de acabar sendo direcionado para um caminho que não existe, pois é apenas um erro da segmentação semântica. Em relação ao ponto de partida da BFS, usamos o ponto mais próximo ao Crawler, pois supomos que haverá na área logo à frente do Crawler uma região navegável representada, por mais que minimamente, por parte do esqueleto. O motivo para descobrirmos quais pixels são alcançáveis é para o Crawler não selecionar como posição do goal state um ponto que está em outra pista (por erro de localização o checkpoint pode estar mais próxima dessa outra pista) a qual não se pode alcançar (pode haver uma separação entre a pista atual e a outra ou por essa outra pista talvez ser um erro da segmentação semântica). Para a busca no grafo utilizamos BFS e não DFS, pois não gostaríamos de caminhos desnecessariamente maiores neste momento. Quando o Crawler está longe de um checkpoint, consideramos apenas os pixels com distância das bordas maior que a mediana, pois assim evitamos desviar dos centros da pista, permitindo ainda uma distância considerável entre o goal state e o Crawler. Fazemos a extensão até o checkpoint, pois em alguns casos a esqueletonização não adentra tanto quanto deveria algumas pistas. Com todas essas motivações em mente, observemos atentamente os exemplos da Figura 24 e reflitamos o que poderia ocorrer caso alguma das condições/passos do processo de obtenção do goal state

não existisse. Há diversos outros detalhes que poderíamos abordar, porém voltaremos, a seguir, para o motivo não de pequenos detalhes da solução e sim o motivo para a construção da solução.

4.2.3 Conclusões

Nesta seção discutiremos algumas das consequências da solução proposta para os problemas apontados na Seção 4.1. É importante que, neste momento, o leitor relembre dos problemas que tínhamos para que possa compreender melhor os motivos de todas as escolhas da solução proposta. Com a esqueletonização seguida da definição do goal state (baseada, dentre outros motivos, na distância em relação ao próximo checkpoint, visto que o checkpoint define locais onde uma decisão de qual caminho seguir deve ser tomada) conseguimos uma forma de guiar o veículo pela sua rota. Porém, devemos notar que passamos a ter que confiar na segmentação semântica, na esqueletonização e no módulo de localização para guiar o Crawler. De qualquer forma, já sabíamos que teríamos que confiar na segmentação semântica para detectar os obstáculos/área não navegável para o módulo de planejamento de movimentação e teríamos que confiar também no módulo de localização, pois o Crawler possui uma rota que é definida com coordenadas geográficas. Portanto, adicionamos apenas uma dependência, a esqueletonização, todavia, a solução como um todo (a esqueletonização inclusa) faz com que não precisemos de uma confiança extrema na segmentação semântica, na localização e na esqueletonização. Vamos continuar a discutir um pouco mais sobre a confiança nesses segmentos. Dependemos do módulo de localização apenas quando o Crawler está perto do checkpoint para saber qual caminho seguir e não precisamos de uma precisão na casa dos decímetros, pois as vias possuem muito mais do que isso e são separadas, normalmente, por muito mais e o Crawler enxerga o suficiente à frente e aos lados para que possamos passar a considerar o checkpoint com distância suficiente para decidir o caminho a tempo. Em relação à segmentação semântica, a própria esqueletonização já tenta extrair uma representação simplificada da imagem segmentada transformada no occupancy grid e nesse processo acaba por desconsiderar algumas falhas da segmentação semântica, além do mais, usamos morfologia matemática para proteger bastante a esqueletonização dessas falhas e, ainda, não consideramos pixels próximos demais da borda para nos precavermos de extensões indevidas da área navegável geradas pela segmentação semântica. Outro detalhe que nem começamos a discutir são as questões temporais, notemos que o occupancy grid inicia 1 metro à frente do Crawler quando a imagem do ambiente foi capturada, e que o OG possui 10 metros de extensão à frente, o que dá ao Crawler bastante tempo para ele se preparar para a próxima iteração. As discussões temporais serão abordadas apenas depois de apresentarmos mais partes do SNA do Crawler. Enfim, com o goal state definido, estamos prontos para passar para a etapa seguinte: o planejamento de movimentação.

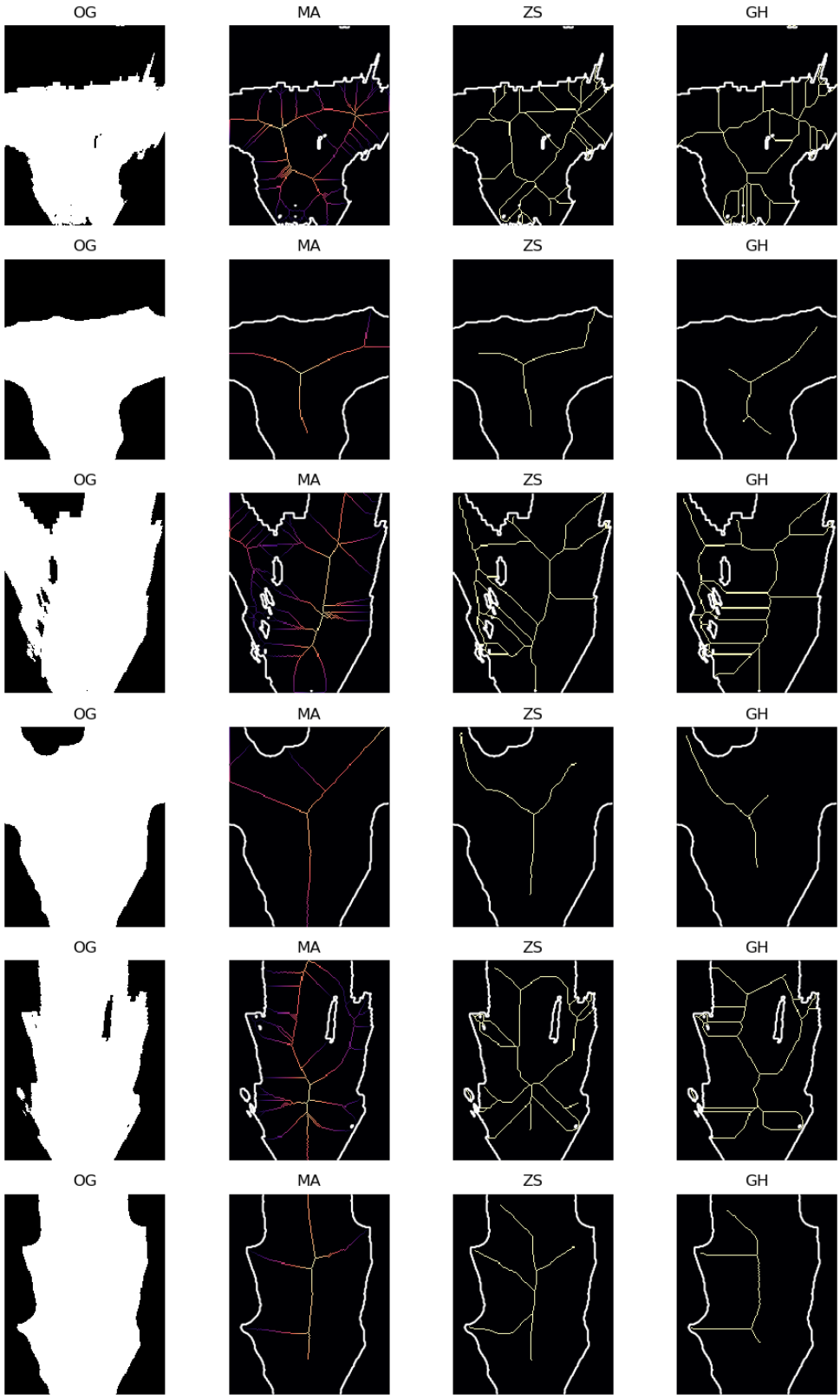


Figura 22 – Comparação entre aplicar esqueletonização com e sem pré-processamento do OG com morfologia matemática para cada um dos três métodos de esqueletonização abordados. A cada par de linhas apresentamos a esqueletonização no OG não pré-processado na primeira linha e na segunda com o OG pré-processado.

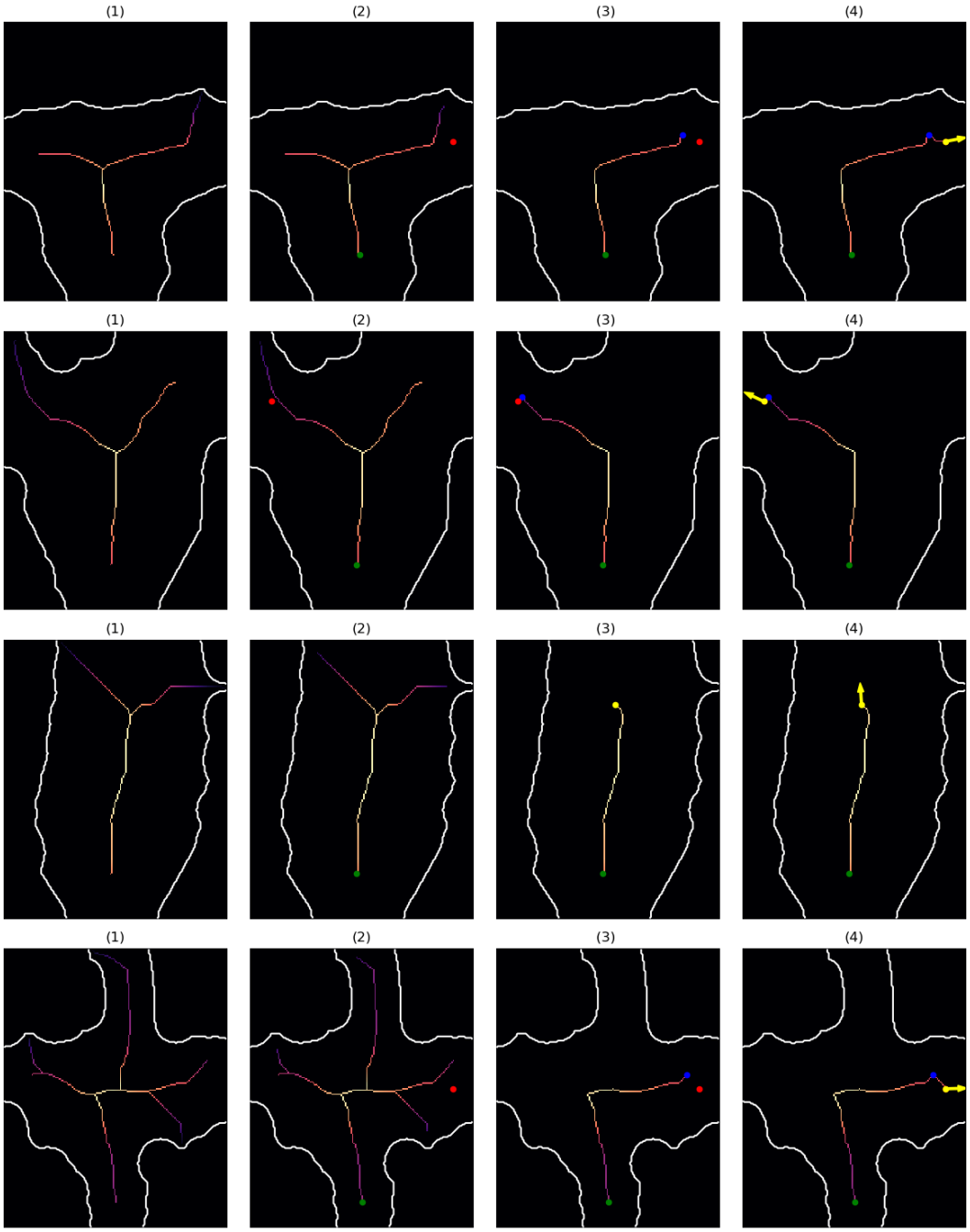


Figura 23 – Separamos em quatro etapas a obtenção do goal state. Em (1), mostramos o OG pré-processado com o esqueleto. Em (2), mostramos o ponto mais próximo do Crawler em verde e o checkpoint, se existir, em vermelho. Em (3), apresentamos o caminho selecionado e o ponto pertencente ao esqueleto que nos fez escolhê-lo. Quando não temos checkpoint o ponto estará em amarelo (representando a posição do goal state) e quando temos checkpoint o ponto estará em azul. Em (4), quando temos checkpoint estendemos o caminho a partir do ponto azul e, assim, obtemos o ponto amarelo representando o goal state; se o ponto vermelho não estiver visível em (3) ou em (4) é porque o ponto azul ou o amarelo podem estar por cima. Além disso, em (4), mostramos a orientação do goal state. A terceira linha representa um caso em que não há checkpoint próximo; para as demais há.

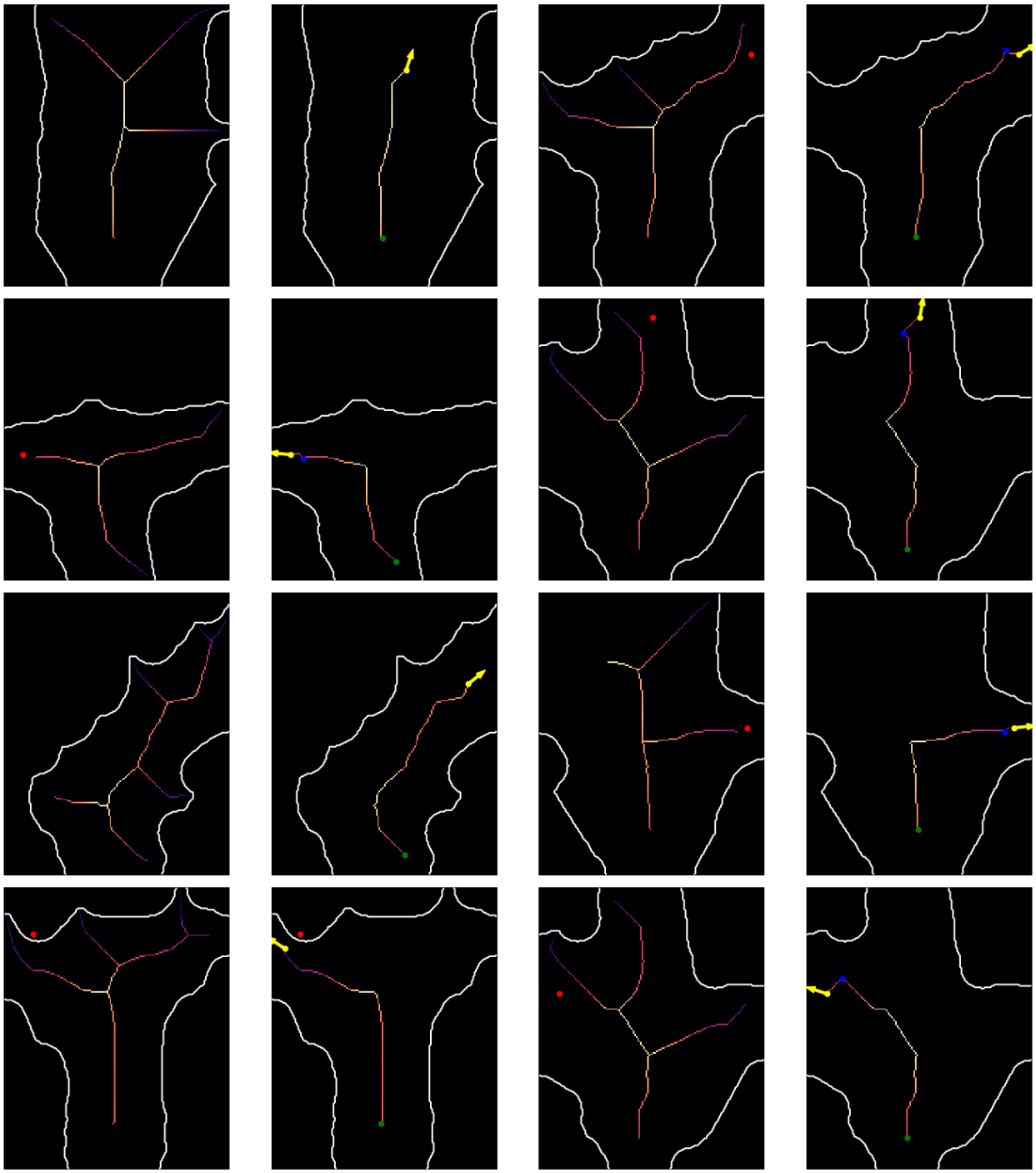


Figura 24 – Exemplos de definição de goal state a partir do esqueleto.

5 PLANEJAMENTO DE MOVIMENTAÇÃO PARA O CRAWLER

Vamos retomar o papel do módulo de Planejamento de Movimentação (PM) na organização do sistema de navegação autônoma apresentado no capítulo de conceitos básicos. Devemos saber que o módulo anterior ao PM é o módulo de Decisão de Comportamento (DC); o qual determina, como o próprio nome diz, um comportamento, que pode ser seguir em frente, parar, etc. A forma com que expressamos esse comportamento no Crawler é através de um goal state pertencente ao state space sobre o qual será feito o PM. O PM usa o goal state fornecido pelo módulo de Decisão de Comportamento como objetivo de estado final para a trajetória que será gerada. O Módulo de PM pode usar informações do ambiente obtidas por sensores próprios ou informações obtidas previamente como mapas para auxiliar no planejamento. Para o Crawler, utilizamos o OG como única fonte de informação do ambiente. Além do OG, possuímos informações de localização mais refinadas e dados mais brutos de sensores (odômetro, IMU e GPS). Essas informações podem auxiliar o PM a tornar a trajetória gerada mais próxima da real trajetória que será executada. A trajetória é enviada para o Módulo de Controle, o qual tentará ao máximo segui-la. O que acabamos de discutir é uma arquitetura que reflete a arquitetura do Crawler, porém devemos ressaltar que há diversas formas com que os três módulos citados podem interagir para alcançarem uma forma de navegação autônoma.

Agora que o papel do PM foi rapidamente relembrado, podemos abordar alguns detalhes mais específicos do Crawler. Com a detecção das pistas de trânsito e com a definição do goal state, acabamos por gerar, além do goal state, um ponto inicial do esqueleto, um caminho desse ponto inicial até o goal state dado pelo esqueleto ou pelo esqueleto e sua extensão e o OG pós operações de morfologia matemática. Agora, talvez surja uma pergunta, por que não podemos simplesmente utilizar o caminho do esqueleto? Dentre vários motivos, há três que devem esclarecer essa dúvida e relembrar porque usamos métodos de PM: (i) o caminho pode não ser realizável por questões físicas do veículo (a curvatura em algum ponto pode ser grande demais); (ii) não há garantia que o ponto inicial do caminho será a posição do veículo quando ele começar a executar a trajetória planejada; (iii) o caminho pode ser mais longo do que o necessário e errático (o caminho dado pelo esqueleto pode ser um zigue-zague). Um detalhe que vale a pena trazermos é que o OG que devemos usar para o Crawler é o OG pós operações de morfologia matemática, pois o goal state poderia estar em uma área considerada não navegável antes das operações, o que pode atrapalhar ou até inviabilizar vários métodos de PM.

Para diversos métodos de PM bastam o goal state, uma maneira de ver se um estado é válido (OG), um estado inicial e uma definição do espaço ao qual pertencem os estados. Alguns métodos podem necessitar de alguns componentes a mais, porém o cerne da maioria dos métodos é esse.

O PM quando comparado com o DC, pelo menos para o projeto do Crawler (em grande parte porque a sua arquitetura foi projetada para isso), nos direciona bem mais a usarmos uma solução praticamente já pronta. O que queremos dizer com isso é que há diversos algoritmos de PM que podemos utilizar e que resolveriam em grande parte a tarefa de PM no Crawler. Por

isso, ao invés de tentarmos desenvolver o nosso próprio método de PM, decidimos utilizar a OMPL (Open Motion Planning Library) (ŞUCAN; MOLL; KAVRAKI, 2012) - uma biblioteca de PM que possui diversos métodos implementados.

A OMPL divide os planners em dois grandes grupos chamados de planners geométricos e planners baseados em controle. Se o planejamento possui qualquer tipo de restrição diferencial, então os planners baseados em controle devem ser usados. Caso contrário, os planners geométricos podem ser usados. A OMPL consiste de métodos baseados em amostragem do state space, o que contribui para a separação dos métodos de planejamento e da checagem de colisão, a qual fica completamente a cargo do usuário da biblioteca, assim como a representação do ambiente por consequência. A Figura 25 exibe essa separação e ainda mostra uma separação entre as técnicas de amostragem e as técnicas de busca, o que também há na OMPL.

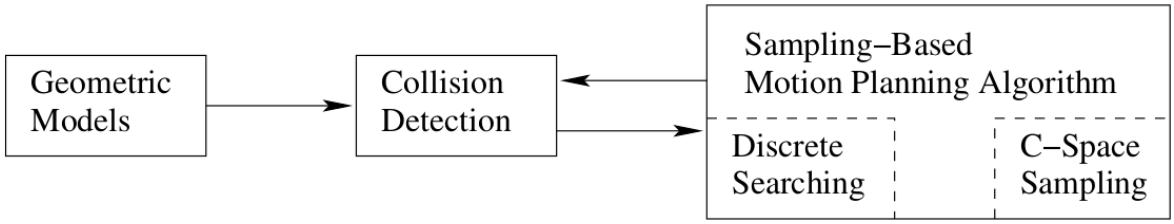


Figura 25 – Organização de planners baseados em amostragem. Fonte: (LAVALLE, 2006).

É importante abordarmos mais alguns aspectos da diferença entre a classe dos métodos geométricos e classe dos métodos baseados em controle. Os métodos geométricos conseguem resolver o Piano Mover's Problem, porém não servem para resolver o MPUDCP, o qual necessita de métodos baseados em controle. Alguns planners, como o RRT, podem pertencer às duas classes de métodos, desde que sejam feitas algumas adaptações. Já outros, foram desenvolvidos especificamente para uma das classes ou algumas suposições tomadas durante seu desenvolvimento, para alguns casos, inviabilizam o seu uso como método geométrico ou baseado em controle.

Cabe, aqui, discutirmos dois novos conceitos: funções de propagação e de steering. As funções de propagação estão relacionadas à propagação de uma ação/controle u ou uma função $\tilde{u} : T \to U$ por um intervalo de tempo t a partir de um estado inicial x_0 (talvez valha a pena retornar à definição do MPUDCP, em especial à equação com integral do oitavo item). Um exemplo de função de propagação é uma função que nos dará o estado final de um veículo, atualmente em x_0, se o acelerarmos com uma intensidade u em linha reta por t segundos. Portanto, a função de propagação pode ser pensada como uma forma de simular um movimento para tentar prever o seu estado final. Já a função de steering, de acordo com (LI; LITTLEFIELD; BEKRIS, 2014), é uma função capaz de achar o caminho ótimo (caso haja uma noção de custo) entre dois pontos na ausência de obstáculos. Se houver restrições diferenciais, a função de steering é equivalente a um solucionador de BVP. Porém, se não houver restrições, a função de steering pode ser simplesmente uma função que gera um segmento de reta conectando os dois pontos. De acordo com (LI; LITTLEFIELD; BEKRIS, 2014), não é fácil produzir uma

função de steering para muitos sistemas dinâmicos. Às vezes há funções de steering, mas elas são muito caras computacionalmente. Por isso, em diversas ocasiões não haverá uma função de steering para ser utilizada pelo planner, o que é uma desvantagem para métodos que dependem de funções de steering para resolver MPUDCPs.

Precisamos discutir mais detalhes da OMPL para que não pairem dúvidas mais à frente. Na OMPL, há alguns componentes que devemos destacar, dentre eles estão StateSpace, SpaceInformation (SI), ProblemDefinition e planner. O StateSpace define o espaço no qual o planejamento vai ser feito ($\mathbb{R}^2$ com yaw e $\mathbb{R}^3$ com roll, pitch e yaw são dois exemplos simples). É no StateSpace que definimos também os limites do espaço; não faz sentido planejarmos para uma área muito maior do que a área do OG. O SI é quem possui, dentre diversas informações do espaço, uma forma de validar um movimento e uma forma de checar se um estado é válido. Vale ressaltar que a forma de checar se um estado é válido deve ser implementada pelo usuário da biblioteca e é nesse método, normalmente, que as informações do ambiente de área navegável são incorporadas. No ProblemDefinition, há os conceitos de estado inicial e goal state os quais devem ser definidos pelo usuário. Em relação ao planner, para vários, basta escolhê-lo e ajustar os seus parâmetros. Evidenciamos que as técnicas de amostragem não precisam ser escolhidas e nem configuradas (apesar de ser possível), pois a OMPL já cuida dessa parte. Na OMPL, há um framework (MOLL; SUCAN; KAVRAKI, 2015) que nos permite fazer benchmark de planners e diversas configurações para o planejamento de movimentação. Usamos esse framework para selecionar bons parâmetros para cada planner e para obter diversos outros resultados, por exemplo, tempo de execução e diminuição do custo da melhor trajetória com o passar do tempo.

Para que pudéssemos escolher um planner adequado para o problema do planejamento de movimentação no contexto do Crawler, pensamos em realizar experimentos com alguns planners da OMPL. Com os experimentos podemos obter diversas informações, como tempo de execução, qualidade dos caminhos gerados, porcentagem de vezes que o planejamento não é completado em determinado limite de tempo, dentre várias outras. Entretanto, devemos notar que alguns planners possuem diversos parâmetros que influenciam extremamente o seu desempenho. Inclusive há parâmetros que não relacionam-se diretamente com um planner específico e sim com o planejamento em geral, os quais também podem mudar bastante o desempenho dos planners. Por isso, decidimos, como parte dos experimentos, fazer uma busca por bons parâmetros para os planners e para o planejamento em geral. Escolhemos testar os planners RRT, RRT* e SST, sendo que consideramos duas implementações do RRT, o RRT geométrico e o RRT baseado em controle - em seguida, já apresentaremos cada planner. Escolhemos esses planners por vários motivos, dentre eles estão: são planners com estrutura e ideias similares; da maneira com que foram utilizados, dois desses planners usam funções de propagação e dois usam funções de steering; dois desses planners otimizam um custo e os outros dois não; todos os quatro estão presentes na OMPL; dois desses planners são baseados em controle e dois são geométricos. Outro ponto que vale mencionar é que utilizamos a versão 1.5.2 da OMPL.

No restante do capítulo apresentamos como cada planner utilizado funciona, algumas de suas características e os state spaces utilizados. Em seguida, apresentamos alguns testes ini-

ciais realizados com cada planner para selecionarmos bons parâmetros. Então, comparamos os planners e discutimos alguns de seus resultados. Por fim, escolhemos um planner e mostramos o que precisamos fazer para melhorar o seu desempenho para embarcá-lo propriamente.

5.1 PLANNERS E ESPAÇOS TESTADOS

5.1.1 RRT - Rapidly-Exploring Random Trees

O RRT (LAVALLE, 1998) é um planner de busca incremental que serviu de base para diversos outros planners como o RRT* que vamos abordar em seguida. No método do RRT, primeiramente, adicionamos o estado inicial à árvore. Depois, iteramos por um número fixo de vezes (como no artigo) ou até atingirmos alguma condição de parada (como na OMPL). De acordo com o artigo, nessa iteração, amostramos um estado aleatório x_{rand} considerando todo o state space, selecionamos o vértice x_{near} mais próximo desse estado e encontramos uma ação u e um Δt que nos direcionam de x_{near} à x_{rand}. Como não há garantias que u nos guiará exatamente à x_{rand}, simulamos aplicar u por Δt a partir de x_{near} e, assim, obtemos x_{new}. Adicionamos x_{new} ao conjunto de vértice, adicionamos a aresta (x_{near}, x_{new}) ao conjunto de arestas e voltamos ao início da iteração. Deve-se checar por colisões para o movimento de aplicar u por Δt a partir de x_{near}.

Para o RRT geométrico (RRTg) da OMPL, não procuramos por uma ação u, apenas interpolamos x_{near} e x_{rand} e checamos se a interpolação não possui colisões, se for válido o movimento de x_{near} a x_{rand}, adicionamos o vértice e a aresta nova à árvore. Além disso, no RRT geométrico há alguns parâmetros que podemos configurar, sendo um deles a distância máxima de x_{near} até x_{new} e o fato de adicionarmos vértices no caminho da interpolação de x_{near} à x_{new}. Já o RRT baseado em controle (RRTc) da OMPL, segue mais as diretivas do artigo. Se houver uma função de steering, essa é usada para procurar por um par u e Δt para que sejam usados na função de propagação para encontrar o x_{new}; caso contrário, o par $(u, \Delta t)$ é amostrado aleatoriamente e o estado final gerado a partir da função de propagação é o x_{new}. No RRT baseado em controle também há os parâmetros mencionados do RRT geométrico.

5.1.2 RRT*

O RRT* (KARAMAN; FRAZZOLI, 2011) é bem similar ao RRT, as diferenças estão nos ajustes das arestas do grafo e na adição de uma noção de custo para cada vértice - o vértice inicial tem custo zero e o custo para todos os outros vértices é baseado no custo do seu respectivo vértice pai e da conexão entre o vértice pai e o vértice atual. No RRT*, após gerar o x_{new}, não é gerada necessariamente uma aresta de x_{near} para x_{new}; são considerados mais vétices próximos ao x_{new} além do x_{near} (a partir de agora, chamaremos o conjunto de vértices próximos selecionados de X_{near} e x_{near} será chamado de $x_{nearest}$). Os autores citam duas estratégias para obter outros vértices para checar a conexão, uma das estratégias é pegar os k vizinhos mais

próximos e outra estratégia é pegar os vértices com distância de no máximo r - tanto k quanto r possuem valores mínimos para que o RRT* seja ótimo assintoticamente. No RRT*, após obter X_{near}, uma aresta é criada entre x_{new} e o vértice pertencente a X_{near} que gera menor custo para x_{new} e cuja aresta gerada seja válida (sem colisões). Um exemplo para as consequências de não adicionar diretamente uma aresta de $x_{nearest}$ a x_{new} pode ser visualizado na Figura 26. Em seguida, checa-se todos os vértice de X_{near}, com exceção do conectado a x_{new}, para saber se o vértice continua com o seu caminho de ancestrais até a raiz ou se x_{new} vira o seu novo pai; essa mudança é feita se a aresta gerada for válida e se o custo do vértice diminuir. Com isso, um grafo acíclico é gerado e o caminho final até o goal state (se houver um) será o de menor custo. Um exemplo de noção de custo que pode ser utilizada é a distância euclidiana entre os dois vértices. Essas diferenças entre o RRT e o RRT* fazem com que o RRT* possa obter caminhos com custos cada vez menores, se existirem, quanto maior for o seu tempo de execução. Vale notar que o RRT* necessita de uma função de steering para criar as arestas. Caso estejamos trabalhando com um problema com restrições diferenciais, essa característica do RRT* pode inviabilizar o seu uso caso não tenhamos uma função de steering adequada. Sem restrições diferenciais podemos, simplesmente, gerar como aresta uma linha reta entre o par de vértices.

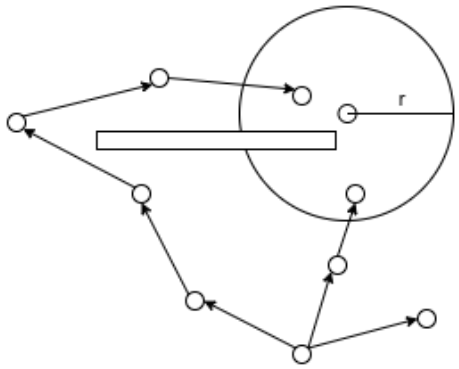


Figura 26 – Exemplo da otimização do RRT*. O vértice sem nenhuma aresta representa o x_{new} e o retângulo representa um obstáculo. Podemos ver que caso não fossem considerados todos os vértices a um raio r de distância (como no RRT), o x_{new} se conectaria a um vértice cujo custo (comprimento do caminho) é muito maior do que o custo de outro vértice próximo.

5.1.3 SST - Stable Sparse RRT

O SST (LI; LITTLEFIELD; BEKRIS, 2014) é um planner que consegue lidar com restrições diferenciais desde que haja uma função de propagação. Além disso, ele é um planner baseado em amostragem e é quase ótimo assintoticamente. Podemos dividir o SST em três partes: seleção do vértice a partir do qual haverá uma propagação, checagem do melhor vértice em uma vizinhança e corte feito no grafo. Algumas observações: começamos com a posição inicial como vértice raiz do grafo; os vértices são classificados em ativos ou inativos; existe uma noção de custo para ser minimizado (estamos falando de um planner quase ótimo). O SST, após fazer a inicialização (adicionar a raiz ao grafo, atribuir custo zero à raiz, dentre outros pequenos

detalhes), já começa a iterar da mesma forma como o RRT e o RRT* (há uma condição de parada que pode ser quantidade de iterações, tempo, etc). A partir de agora estaremos abordando o que acontece em cada iteração. Primeiramente, é amostrado um estado aleatório x_{rand} do espaço, procura-se pelos vértices X_{near} à uma distância δ_{BF} de x_{rand}; considerando todos os vértices em X_{near}, é selecionado o com menor custo, caso X_{near} seja vazio, seleciona-se o vértice mais próximo. O vértice selecionado é chamado de $x_{selected}$. A partir de $x_{selected}$ é usada uma função de propagação com controle e duração escolhidos aleatoriamente, gerando x_{new}. Deve-se, então, conferir se a aresta gerada pela propagação é livre de colisão, caso seja, é atribuído o custo de x_{new}. Agora, deve-se checar se x_{new} é o melhor vértice da sua vizinhança com base no seu custo. Encontra-se o vértice ativo mais próximo a x_{new}, o qual será chamado de s_{new} se a distância entre s_{new} e x_{new} for maior que δ_s, x_{new} é adicionado ao conjunto dos vértices ativos e a aresta $(x_{selected}, x_{new})$ é adicionada ao conjunto de arestas. Deve-se fazer o mesmo se a distância de s_{new} e x_{new} for menor do que δ_s e se o custo de x_{new} for menor do que o custo de s_{new}; nesse caso deve-se ainda passar s_{new} para o conjunto dos vértices inativos e realizar os cortes no grafo, se houver algum para ser feito. Se a distância de s_{new} e x_{new} for menor do que δ_s e se o custo de x_{new} for maior do que o custo de s_{new}, apenas passa-se para a próxima iteração. Agora, vamos falar de como funcionam os cortes no grafo. Essa parte é relativamente simples, apenas deve-se retirar os vértices folhas inativos recursivamente até que todos os vértices folhas inativos tenham sido eliminados.

Resumidamente, os motivos para o SST não ser ótimo assintoticamente e sim quase ótimo são o δ_{BF} e o δ_s que favorecem, respectivamente, uma exploração mais voltada para uma rápida otimização do caminho e os cortes no grafo. Ainda em (LI; LITTLEFIELD; BEKRIS, 2014), é apresentado um outro planner, chamado SST*. Esse planner itera fazendo chamadas sucessivas do SST sobre o mesmo grafo e, conforme itera, diminui δ_{BF} e δ_s utilizando um fator ξ que é passado como parâmetro, junto com os δ_{BF} e δ_s iniciais. Relacionando com o SST, por causa dessa constante diminuição de δ_{BF} e δ_s, o SST* é ótimo assintoticamente. Não utilizamos o SST* em nenhum experimento, o apresentamos apenas para explicar o porquê de o SST ser assintoticamente quase ótimo. O principal motivo de não termos realizado experimentos com o SST* foi por termos outras questões mais prioritárias relacionadas a este trabalho.

5.1.4 Espaços

Os espaços que estamos nos referindo aqui são os espaços nos quais o planejamento é feito - o state space. Usamos dois state spaces diferentes para os experimentos:

1. R2: (x, y)
2. SE2: (x, y, yaw)

onde R2 e SE2 são os nomes que demos aos espaços; x e y representam a posição no plano do OG e yaw é o yaw do veículo. Estamos usando o R2 para os planners geométricos (RRTg

e RRT*), pois adicionar o yaw não geraria o efeito que desejamos. Se estivéssemos usando o SE2 nos planners geométricos o veículo poderia mudar sua orientação sem causar consequência alguma para a sua posição ou posição futura e, além disso, a orientação entraria para os cálculos de distância entre dois estados influenciando os planners desnecessariamente. Utilizamos o SE2 para os planners baseados em controle (RRTc e SST).

Precisamos abordar o modelo físico do veículo utilizado para cada um dos espaços, pois cada espaço possui uma capacidade de representação diferente e isso reflete nas suposições de cada modelo e por consequência, na sua complexidade. No modelo do R2, desconsideramos qualquer noção de tempo/velocidade e o yaw do veículo também não é considerado. Assim, podemos admitir que um caminho válido (sem considerar colisões) entre dois pontos é uma reta e a inclinação da reta anterior do caminho em nada influencia a próxima. Já no SE2, o modelo do veículo considera velocidade, rotação das rodas dianteiras e orientação. A Figura 27 mostra o veículo que consideramos para esse modelo. Vale notar que apenas as rodas dianteiras rotacionam (para mudar a orientação do veículo) e vamos chamar o ângulo dessa rotação de ângulo de steering. Além disso, consideramos que as ações U são feitas sobre a velocidade e o ângulo de steering $(u = (u_s, u_\phi))$ e que ambos podem mudar de valor instantaneamente, por exemplo, a u_s nesse modelo pode ir de 0 m/s para 100 m/s em um instante. Ainda para esse modelo, consideramos as seguintes equações:

1. $x_f = x_0 + (u_s * \Delta t) * \cos(\theta_0)$
2. $y_f = y_0 + (u_s * \Delta t) * \sin(\theta_0)$
3. $\theta_f = \theta_0 + ((u_s * \Delta t)/L) * \tan(u_\phi)$

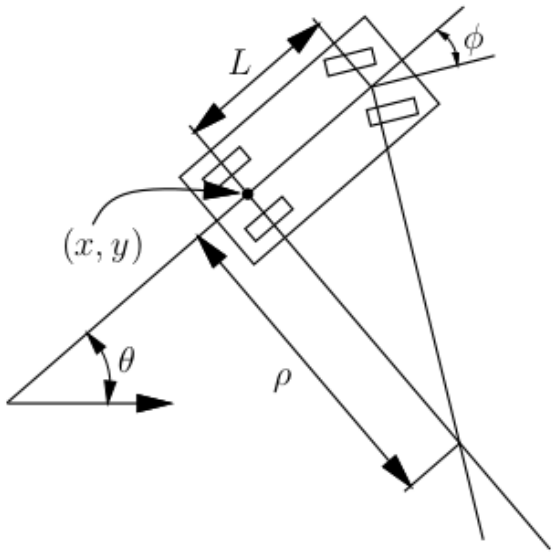


Figura 27 – Modelo do veículo utilizado para o state space SE2. Fonte: (LAVALLE, 2006).

Para que pudéssemos obter as equações tivemos que admitir que Δt tende a zero, porém passos de tempo menores tendem a causar atrasos para os planners (cada movimento será menor). Vale notar que as equações anteriores são as equações usadas para a função de propagação

dos planners no SE2, onde propagamos q_0 por Δt segundos usando a ação $u = (u_s, u_\phi)$. Uma última observação para esse modelo, tivemos que admitir que Δt tende a zero, pois consideramos que o veículo vai andar em direção a sua orientação inicial e não em direção a todas as orientações intermediárias geradas pela aplicação do ângulo de steering.

Vale ressaltar que apesar de o SE2 conseguir uma representação um pouco mais próxima do veículo real, esse espaço possui uma dimensão a mais do que o R2. Precisamos notar que representações mais simplificadas do veículo, apesar de simplificarem o planejamento, podem prejudicar o módulo de controle e a confiança na trajetória traçada pelo PM, visto que a trajetória pode estar bem distante de uma trajetória fisicamente factível de ser executada pelo veículo. Ademais, a maior dimensionalidade de alguns espaços quando comparados com outros traz detrimentos no tempo de execução, principalmente, pelo aumento do tamanho do espaço que precisa ser explorado. Mas será que esse detrimento no tempo de execução será tão alto para o SE2 que teremos que usar a representação mais simplificada do sistema do veículo? Ou a qualidade dos caminhos gerados no R2 será tão deplorável que teremos que utilizar o SE2 mesmo com o maiores tempos de execução? Quem sabe a qualidade dos caminhos gerados em ambos os espaços não será suficiente para uma boa navegação? Pretendemos responder essas perguntas e diversas outras relacionadas com os planners escolhidos com os experimentos que apresentamos a seguir.

5.2 EXPERIMENTOS DOS PLANNERS

Decidimos dividir os experimentos com os planners em duas etapas, para ambas usamos o framework de benchmark da OMPL. Na primeira, o objetivo principal é procurar por bons parâmetros para os planners, de forma que ao fim tenhamos selecionado uma configuração boa de parâmetros de cada planner. Aproveitamos para fazer uma análise inicial de cada planner e, também, para obter bons parâmetros para todo o planejamento. Ademais, vale notar que realizamos os experimentos da primeira etapa no meu notebook. Já, na segunda etapa, realizamos os experimentos de cada configuração selecionada na Raspberry Pi e o objetivo desses experimentos é obter mais informações do desempenho de cada planner, de forma que ao fim da segunda etapa possamos escolher um planner com sua respectiva boa configuração de parâmetros. Devemos salientar que apesar de termos separado todo esse processo de escolha do planner em duas etapas, às vezes tivemos que nos adiantar para a segunda etapa para realizar alguns testes na Raspberry ou voltamos para a primeira etapa para ajustar alguns detalhes. Outro ponto importante é o motivo de não termos executado a primeira etapa na Raspberry; por testarmos diversas combinações de parâmetros, a quantidade de configurações às vezes era relativamente grande e por a Raspberry ser mais bem mais lenta do que meu notebook, levaríamos muito mais tempo para executar essa etapa e não possuiríamos o dinamismo necessário para comparar algumas mudanças de parâmetros de todo o planejamento (não de um planner específico). Por fim, vale notar que realizamos alguns testes para perceber se havia uma correspondência entre os desempenhos obtidos no meu notebook e na Raspberry, o que foi de fato observado.

Antes de apresentarmos os resultados dos experimentos, há alguns conceitos e detalhes da OMPL e do benchmark para abordarmos. Comecemos pela região do goal state. Alguns planners não possuem uma forma de forçar a conexão de um estado específico (goal state, por exemplo) com o resto do grafo, pode ser por não possuir uma função de steering ou por o próprio planner não usar funções de steering. Assim, definimos uma região em volta do goal state em que consideramos qualquer estado dentro dessa região como o goal state. Quanto maior essa região, mais podemos facilitar o planejamento. Agora, discutiremos, brevemente, as otimizações feitas pelo RRT* e pelo SST. Esses planners utilizam alguma noção de custo que pode ser escolhida pelo usuário do planner; tamanho do caminho, curvatura máxima e clearance (distância a obstáculos) são exemplos de custos comuns considerados para a navegação veicular, porém devemos ressaltar que alguns planners possuem restrições quanto ao custo utilizado. Passamos, agora, às condições de término dos planners. Na OMPL, os planners sem otimização terminam se o planner gerar um caminho que chega à região do goal state. Ademais, o usuário pode especificar e até criar uma condição de parada a mais que pode ser, por exemplo, quantidade de iterações ou tempo de execução. Para os planners com otimização de um custo, o planner irá parar caso seja gerado um caminho à região do goal state com um custo abaixo de um valor que deve ser especificado pelo usuário. A condição de término extra definida pelo usuário também pode ser utilizada para os planner com otimização. Ao utilizarmos o benchmark da OMPL, precisamos especificar como condição de término extra, que é o limite de tempo para cada planejamento. Outro conceito importante é o de movimento, relacionamos esse conceito com as arestas do grafo dos planners. Considerando os planners abordados, os movimentos podem ser frutos de uma função de steering ou uma função de propagação (depende do planner). Adiante com mais um detalhe, cada espaço necessita de limites para que estados possam ser amostrados dentro desses limites. Para o R2 e para o SE2 uma de suas dimensões está limitada em [-400, 400) e a outra em [-50, 1000). O SE2 possui outra dimensão cujos limites são [-π, π). Esses valores foram escolhidos, pois o OG representa 800 cm de largura e 1000 cm de comprimento. Para explicar os 50 cm precisamos comentar, antes, sobre o start state do planejamento. Como estamos realizando esses experimentos de forma desacoplada da localização do Crawler, precisamos de algum estado do state space para ser o start state do planejamento. Assim, escolhemos o ponto (0, -50) e para o SE2 com yaw na mesma direção em que o OG foi construído. Escolhemos esses valores pois imaginamos que esse estado seria próximo do estado o qual o Crawler normalmente poderia estar. Por isso, consideramos 50 cm fora do OG para o planejamento e, ainda, consideramos esses 50 cm como navegáveis, o que não é uma suposição muito ruim, visto que da câmera até a frente do Crawler são 45 cm. Nas imagens que apresentam planejamentos neste capítulo, haverá uma região amarela na parte inferior da imagem, essa área representa os 50 cm adicionados ao OG. Com todos esses novos conceitos e observações em mente, podemos discutir a seleção de parâmetros.

5.2.1 Seleção de parâmetros

Para realizar os testes e selecionar os parâmetros do RRTg, RRT*, RRTc e SST, escolhemos oito saídas do módulo de DC, quatro dessas oito estão ilustradas na última coluna da Figura 23 e para as outras quatro também escolhemos uma de cada tipo de via. Um ponto importante que precisamos abordar é a noção de custo que usamos para o RRT* e para o SST, para ambos os planners utilizamos o tamanho do caminho gerado como custo e consideramos, inicialmente, o valor mínimo de custo, relacionado com a condição de término do planner, para o caminho como a distância L1 entre o estado inicial e o goal state. A escolha desse valor foi apenas uma estimativa. Mais à frente discutiremos mais detalhes da otimização e algumas consequências da otimização em si e de como a otimização foi configurada (custo, valor mínimo, etc).

5.2.1.1 R2

Executamos cada configuração de planner por 5 vezes para cada saída do módulo de DC, com tempo limite de 1 segundo. Escolhemos 5 cm de raio para a região do goal state. Outro parâmetro que modificamos do planejamento foi o longestValidSegment. Esse parâmetro indica o comprimento máximo que cada segmento de movimento pode ter. Uma das consequências desse parâmetro é a influência na checagem da validade de um movimento (checar se há colisões), o movimento é quebrado em segmentos e um ponto de cada segmento é utilizado para checar por colisão. Lembremos que a definição da função de validação de um estado é responsabilidade do usuário, visto que essa função depende do ambiente para o qual o planejamento está sendo feito - a OMPL não tem como saber a priori quais regiões do state space estão ocupadas. Voltando ao parâmetro, decidimos usar longestValidSegment=5.0. Em seguida, vamos abordar em detalhes os resultados desse experimento para o RRTg e, mais à frente, para o RRT*.

O RRTg possui dois parâmetros que decidimos ajustar: range e addIntermediateStates. Já havíamos comentado um pouco sobre eles na explicação do RRTg, agora, retomaremos esses conceitos e adicionaremos alguns detalhes. Range é o tamanho máximo que um movimento pode ter. Se um movimento foi gerado cujo tamanho é maior que range, encurta-se o movimento até ele ficar com tamanho range - neste caso o x_{new} será um x_{new} mais próximo de x_{near}. Comentamos, anteriormente, que no RRTg interpolamos x_{near} até x_{new}, para interpolar juntamos os dois pontos por um segmento de reta. Já o addIntermediateStates controla se quando adicionamos um x_{new} ao grafo, adicionamos também estados intermediários do movimento (x_{near}, x_{new}). A quantidade de estados intermediários que são adicionados depende do parâmetro longestValidSegment de forma que a quantidade seja $n = ceil((tamanho\,do\,movimento)/longestValidSegment) - 1$. Os valores que decidimos testar para os dois parâmetros do RRTg, assim como o tempo médio considerando todas as execuções de cada configuração de RRTg estão apresentados na Tabela 3. Na Figura 28, apresentamos alguns exemplos para ilustrar a diferença que a mudança de parâmetros pode fazer nos caminhos

gerados pelo RRTg.

Tabela 3 – Configurações do RRTg e seus respectivos tempos médios de execução. O menor tempo está em negrito e o segundo menor está em itálico.

	addIntermediateStates=False	addIntermediateStates=True
range=50.00	0.001610	0.009105
range=100.00	*0.001226*	0.007564
range=250.00	0.002092	0.004207
range=500.00	0.001274	0.003052
range=1320.04	**0.001222**	0.003802

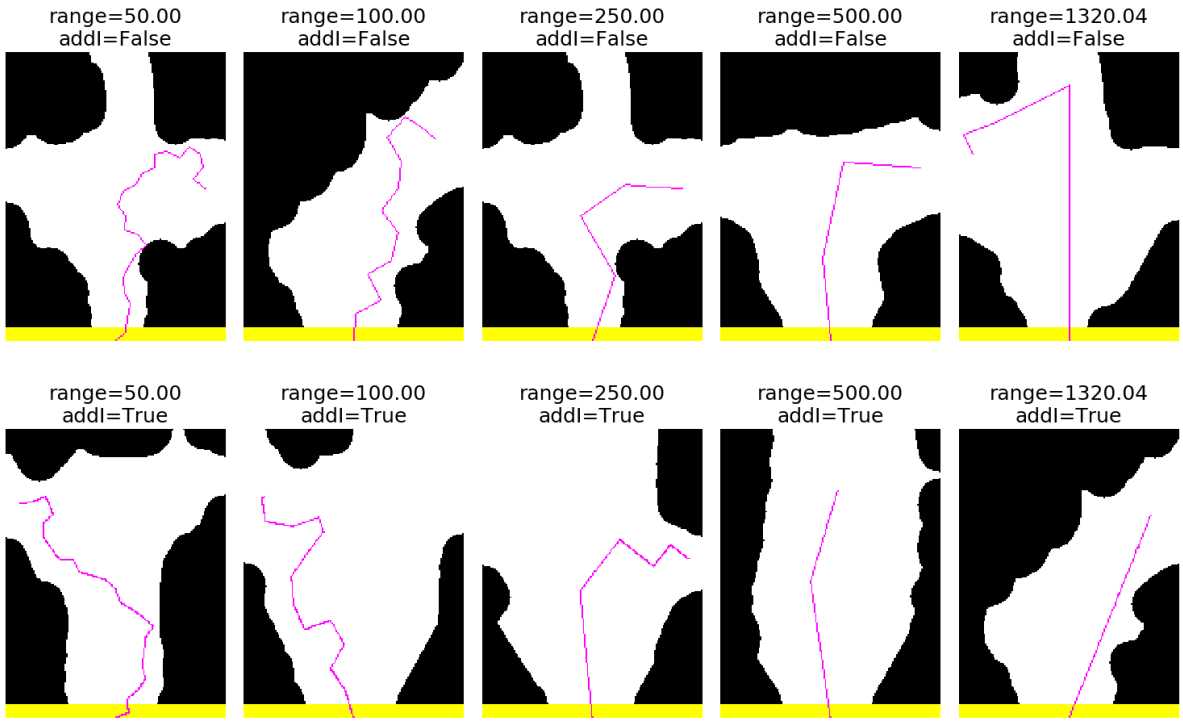


Figura 28 – Comparação entre as diferentes configurações de RRTg. Há diferentes OGs e goal states. Abreviamos o parâmetro addIntermediateStates para addI.

Como podemos perceber pela tabela e pela figura, adicionar os estados intermediários tende a aumentar bastante o tempo e, pelo o que observamos, os caminhos gerados não possuem uma qualidade melhor. Ademais, o tempo de execução tende a diminuir se usarmos valores maiores para o range, o que faz sentido, pois assim o planner precisa de menos estados no caminho até o goal state. Além disso, a qualidade para ranges menores fica comprometida por causa dos zigue-zagues excessivos, porém mesmo com ranges maiores não temos garantia que isso não ocorrerá (apenas diminuímos a probabilidade). Com isso em mente, os caminhos com ranges grandes são muitas vezes uma reta (quando possível) ou caminhos que tendem a ir quase que diretamente para o goal state (não precisam de muitas movimentações para chegar no goal state). Resumidamente, por tender a não possuir tantos zigue-zagues e por ter tender a ter um tempo

de execução melhor, decidimos seguir para a próxima etapa com a configuração com menor tempo de execução que é (range=1320.04, addIntermediateStates=False). Há três observações que queremos apresentar: (i) os valores de teste para os ranges foram escolhidos com base em alguns experimentos prévios; (ii) um dos valores é 1320.04 pois essa é, aproximadamente, a distância máxima possível entre dois estados no state space; (iii) como já esperávamos, os caminhos gerados pelo RRTg não representam bons caminhos para veículos executarem, alguns dos motivos são que eles não seguem de forma alguma a estrutura da via e possuem curvas bruscas (a mudança de nenhum parâmetro fará com que essas péssimas características sejam corrigidas).

Agora, passamos para o RRT*. O RRT* implementado na OMPL pode possuir algumas características a mais (como poda do grafo e algumas técnicas específicas de amostragem) que o algoritmo apresentado anteriormente, porém escolhemos alguns parâmetros (os quais não detalharemos) para permanecermos com uma implementação do método original. Com uma exceção, decidimos testar uma dessas pequenas modificações: a rejeição do x_{new}, cujo parâmetro relacionado é o newStateRejection que é uma variável booleana. Se estivermos usando newStateRejection e não houver como gerar um caminho até o goal state usando x_{new} melhor do que o caminho atual já encontrado (se houver um, caso contrário newStateRejection não influencia em nada no funcionamento do RRT*), então x_{new} não é adicionado ao grafo. Se newStateRejection for falso, o comportamento do algoritmo não é modificado. O RRT* possui um parâmetro range que é equivalente ao parâmetro range do RRTg. Lembremos que o RRT* possui duas formas de obter o conjunto X_{near}, usando os k vizinhos mais próximos ou os vizinhos a uma distância de no máximo r e existem valores mínimos tanto para k quanto para r. Para os nossos testes, obtemos X_{near} a partir dos k vizinhos mais próximos. Um dos parâmetros que influencia o valor de k é o rewireFactor, esse parâmetro é usado no cálculo para definir k (multiplica-se rewireFactor por uma variável que define um k mínimo). Assim como foi feito para o RRTg, apresentamos a Tabela 3 com o tempo médio considerando todas as execuções de cada configuração do RRT* e na Figura 29, apresentamos alguns exemplos.

Comecemos analisando o impacto do newStateRejection. Podemos perceber que nas configurações com os piores tempos médio de execução o impacto do newStateRejection foi bem alto, reduzindo em alguns casos por volta de 40% do tempo médio (range=50.00, rewireFactor=1.50). Porém, nos casos com tempo médio melhor, não ajudou tanto, sendo que em alguns deles o tempo médio de execução piorou. Analisando visualmente os caminhos, não percebemos diferenças de qualidade quando comparamos execuções com e sem newStateRejection; o mesmo podemos dizer quando comparamos execuções com diferentes valores de rewireFactor. A partir das informações da Tabela 4, não conseguimos inferir com grandes certezas que algum valor do rewireFactor é muito melhor do que outro. Porém, como 6 das 10 combinações range/newStateRejection apresentaram menor tempo para rewireFactor igual a 1.10 ou 1.50, tendemos a escolher um desses dois. A escolha dos valores para serem testados de rewireFactor e range foi feita com base em alguns testes prévios. Por último, o impacto do parâmetro range no RRT* foi extremamente similar ao seu impacto no RRTg, o tempo médio

Tabela 4 – Configurações do RRT* e seus respectivos tempos médios de execução. O menor tempo está em negrito e o segundo menor está em itálico.

NSR=False	RF=1.05	RF=1.10	RF=1.50	RF=2.00	RF=5.00
range=50.00	0.033593	0.029215	0.036038	0.036635	0.032288
range=100.00	0.004746	0.005635	0.004870	0.004439	0.005805
range=250.00	0.001787	0.001749	0.001641	0.001708	0.002007
range=500.00	0.001624	**0.001404**	0.001890	0.001695	0.001566
range=1320.04	0.001697	0.001616	0.001494	0.001779	0.001606
NSR=True	RF=1.05	RF=1.10	RF=1.50	RF=2.00	RF=5.00
range=50.00	0.027469	0.027751	0.021763	0.027961	0.027572
range=100.00	0.004821	0.003805	0.004785	0.004688	0.005025
range=250.00	0.001598	0.001729	0.001505	0.001449	0.001789
range=500.00	0.001769	0.001507	0.001621	0.001619	*0.001479*
range=1320.04	0.001522	0.001794	0.001810	0.001627	0.001631

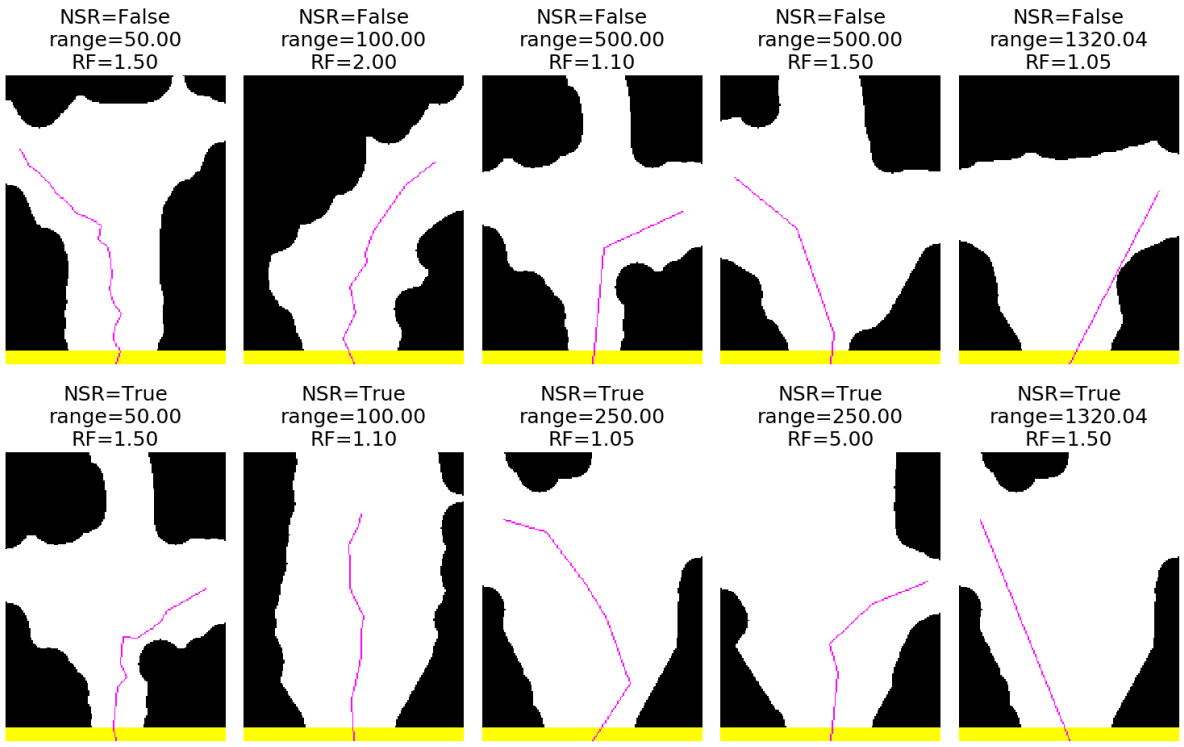


Figura 29 – Comparação entre as diferentes configurações de RRT*. Há diferentes OGs e goal states. Abreviamos o parâmetro newStateRejection para NSR e o parâmetro rewireFactor para RF.

e os zigue-zagues tendem a diminuir conforme aumentamos o range. A combinação com menor tempo foi (range=500.00, rewireFactor=1.10, newStateRejection=False), porém como em diversos casos newStateRejection ajudou a diminuir bastante o tempo médio de execução e quando newStateRejection aumentou o tempo de execução o seu impacto foi bem menor. Por todos esse fatores, escolhemos a configuração (range=500.00, rewireFactor=1.10, newStateRejection=True).

5.2.1.2 SE2

Assim como nos testes feitos no R2, realizamos 5 execuções para configuração de planner, mas reduzimos o limite de tempo de cada execução para 0.1 segundo. Essa redução foi feita para que tenhamos alguma trajetória, por mais que ela não atinja o goal state dentro dos limites que desejamos, pois o RRTc e o SST com as configurações selecionadas às vezes levam um tempo inviável para o SNA. Ainda aumentamos o raio da região do goal state para 20 (não é bem cm pois nessa medida consideramos o yaw do veículo também, de forma que 1 cm de distância para a posição (x, y) seja equivalente a 1 grau de distância para o yaw - lembremos que no SE2 o goal state possui orientação). Para os planners baseados em controle temos dois parâmetros que chamaremos de quantidade mínima e máxima de passos do controle que, especificamente, demarcam o intervalo da quantidade de vezes que a função de propagação vai ser chamada para um determinado controle. Escolhemos os valores 5 e 15 para esses parâmetros - não deixamos valores mais baixos para não termos muitas pequenas curvas em S e valores muito altos podem fazer com que o Crawler se desloque muito em direção de partes não muito interessantes do state space. Consideramos como velocidades possíveis para o Crawler valores no intervalo [0, 50] cm/s. Ademais, consideramos $\Delta t = 0.2$ segundo. Dessa forma, o deslocamento máximo do Crawler por movimento é 10 cm. Em relação ao ângulo de steering, decidimos amostrar valores no intervalo [-12, 12] graus. Há três principais motivos para termos considerado esse intervalo com valores reduzidos: (i) podemos evitar a formação de pequenas curvas em s ao longo do caminho. (ii) Dessa forma, no modelo não é possível o ângulo de steering fazer grandes mudanças repentinas. Por exemplo, ir -45 graus para 45 graus em um intervalo de 0.2 segundo, uma mudança tão grande assim do ângulo de steering necessitaria no Crawler de, aproximadamente, 3-4 vezes esse tempo. (iii) Por estarmos restringindo os valores de ângulo de steering para os valores que normalmente seriam utilizados (curvas que necessitam 45 graus para o ângulo de steering são bem raras, ainda mais levando em consideração a distância entre os eixos do Crawler), podemos, em alguns casos, facilitar e acelerar o planejamento. Entretanto, por não considerarmos o real intervalo de ângulo de steering, limitamos de certa forma o planejamento; todavia, essa limitação, para os experimentos realizados, nunca inviabilizou nenhum planejamento. Um pequeno detalhe é que a para validar um movimento no SE2, validamos o estado final de cada passo dado pelo movimento - não há um parâmetro equivalente ao longestValidSegment para os métodos baseados em controle. Assim, podemos dar um passo de 10 cm e realizar apenas uma validação de estado.

Para o RRTc há um parâmetro que decidimos testar, o addIntermediateStates. A ideia é que se addIntermediateStates=True, o planner adiciona o estado de cada passo do movimento ao grafo; caso contrário, apenas o estado final do movimento é adicionado ao grafo. Os tempos médios de execução do RRTc podem ser vistos em 5. Esses tempos nos indicam que mesmo considerando uma região para o goal state relativamente grande, ficou bem próximo do limite de 0.1 segundo. Ademais, observamos que uma grande parte das execuções, principalmente para addIntermediateState=True, não conseguiram traçar uma trajetória que chegou à região do

goal state, porém diversos desses resultados ficaram próximos. Na Figura 30, podemos observar trajetórias geradas pelo RRTc. Novamente, em geral, o addIntermediateStates não trouxe nenhuma nítida diferença de qualidade para trajetórias que conseguiram chegar à região do goal state. Por esses motivos, daremos preferência para o RRTc com addIntermediateState=False.

Tabela 5 – Configurações do RRTc e seus respectivos tempos médios de execução.

addI=False	addI=True
0.077159	0.098224

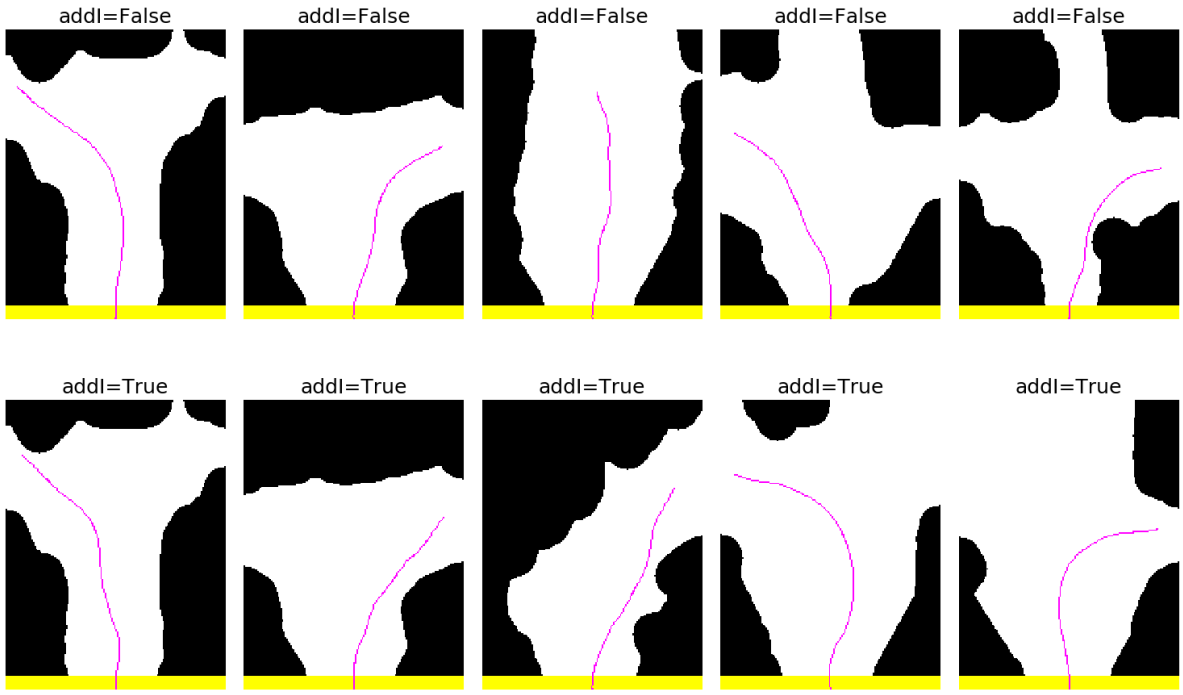


Figura 30 – Exemplos de trajetórias geradas pelo RRTc. Abreviamos addIntermediateStates para addI.

O SST possui dois parâmetros com os quais testamos diferentes valores: selectionRadius (δ_{BF}) e pruningRadius (δ_s). Os autores do SST discutem diversos detalhes desses dois parâmetros, mas resumidamente, quanto maior o pruningRadius maiores são as chances de um caminho não ser descoberto se houver passagens pequenas, entretanto há mais cortes sendo feitos o que tende a acelerar o planner. Já um selectionRadius maior favorece a busca por trajetórias com custo menor em detrimento à exploração do state space. Pelo o que podemos observar na Tabela 6, o selectionRadius que gerou os menores tempos médios de execução foi 25.0 e os melhores pruningRadius foram 2.5 e 5.0. Optamos seguir com o pruningRadius=5.0, pois apresentou menor variância. Ao analisarmos visualmente as trajetórias geradas, não notamos nenhuma diferença significativa, em geral, com a mudança dos parâmetros, alguns exemplos estão apresentados na Figura 31.

Tabela 6 – Configurações do SST e seus respectivos tempos médios de execução. O menor tempo está em negrito e o segundo menor está em itálico.

	SR=0.2	SR=1.0	SR=5.0	SR=25.0	SR=50.0
PR=1.0	0.098964	0.088882	0.080939	0.099222	0.077317
PR=2.5	0.096567	0.094433	0.097121	**0.072391**	*0.073864*
PR=5.0	0.093508	0.088136	0.082610	0.077468	0.090433
PR=10.0	0.096082	0.093102	0.080892	0.097221	0.102288
PR=30.0	0.103492	0.105936	0.095719	0.082188	0.092518

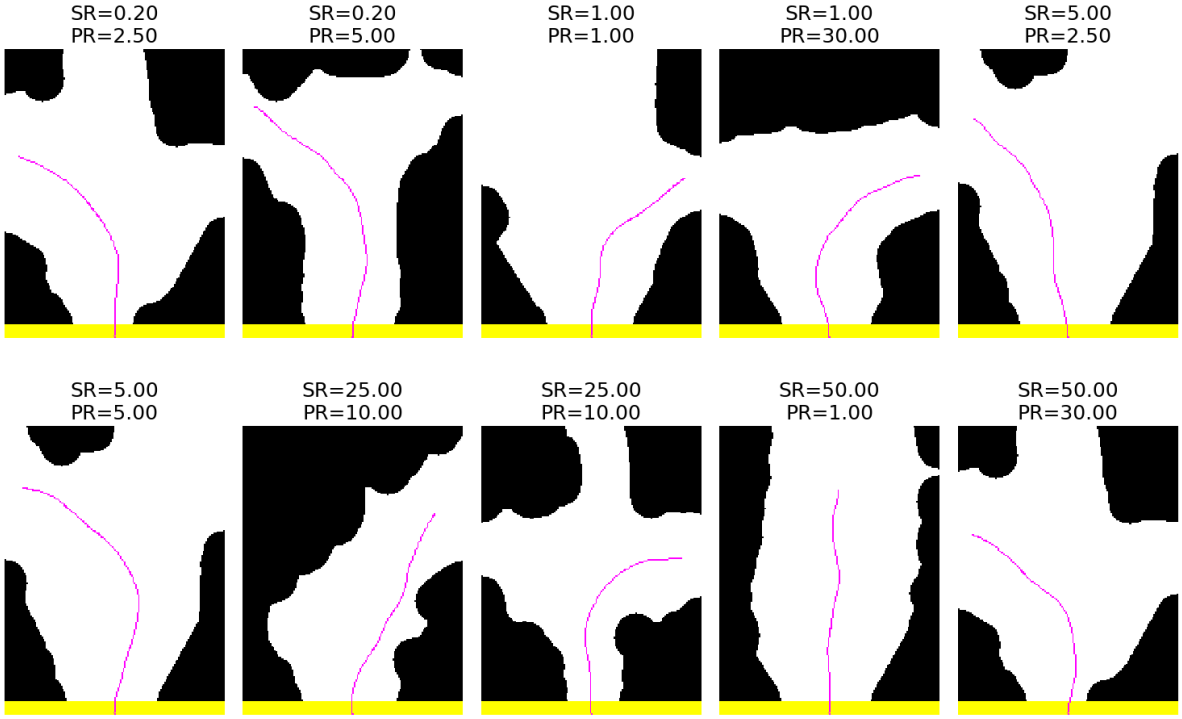


Figura 31 – Comparação entre as diferentes configurações de SST. Há diferentes OGs e goal states. Abreviamos o parâmetro selectionRadius para SR e o parâmetro pruningRadius para PR.

5.2.2 Comparação entre os diferentes planners

Primeiramente, compararemos os planners com base nas informações obtidas da primeira etapa dos experimentos e com base no próprio algoritmo de cada planner. Em seguida, apresentaremos as informações da segunda etapa dos experimentos. Comecemos comparando a qualidade dos caminhos/trajetórias geradas. É clara a superior qualidade das trajetórias geradas pelos métodos baseados em controle. Os métodos geométricos geram caminhos com curvas bruscas, pois os diversos movimentos que compõem os caminhos são sempre gerados por uma interpolação no $\mathbb{R}^2$ (segmento de reta); muitas dessas curvas são bem difíceis de serem seguidas pelo veículo e algumas são até mesmo impossíveis (por o veículo não dar ré, por exemplo). Pela ausência de uma função de steering que represente minimamente o comportamento do veículo

(não consideramos gerar segmentos de reta entre estados como uma função de steering aceitável para o veículo), os métodos RRTg e RRT* são inerentemente métodos sem garantia alguma de boa qualidade para o caminho gerado. Ademais, a orientação do estado inicial e do goal state não influenciam em nada no caminho, o que piora ainda mais a sua qualidade. Entretanto, podemos amenizar essa péssima qualidade com uma boa escolha do parâmetro range para ambos os métodos e para o RRT* a sua capacidade de otimização pode nos ajuda a evitar pequenas curvas ou até mesmo voltas desnecessárias, porém (considerando o tamanho do caminho como custo) favorecemos caminhos demasiadamente retos. Resumidamente, para o RRTc e SST, por termos funções de propagação que consideram algumas (apesar de simples) restrições físicas do veículo e por gerarmos os segmentos das trajetórias com base nessas funções de propagação, obtemos em geral uma qualidade bem superior de trajetórias. Com o SST, temos a possibilidade ainda de melhorarmos a qualidade da trajetória ao usufruirmos da sua capacidade de otimização.

Um ponto importante que devemos discutir é a otimização no RRT* e no SST e algumas consequências da otimização. A princípio, devemos nos lembrar que a otimização é inerente a esses dois planners, de forma que a construção do grafo já é feita pensando no custo de cada movimento. Além disso, essa otimização é feita com base em uma função de custo ligada a uma distância - com custo podemos nos referir ao custo de um movimento entre dois estados ou ao custo de um estado que é dado por uma função que combina os custos (soma, por exemplo) do seu vértice pai com o custo do movimento/aresta (vértice pai, estado atual). Há algumas restrições para as funções de custo e de combinar custos, as quais devem ser respeitadas para garantir a otimalidade/quase otimalidade. Ao compararmos o RRT* ao RRTg e o SST ao RRTc, as suas otimizações fazem com que cada uma de suas iterações seja mais custosa computacionalmente. Porém, esse custo a mais, pode ajudar o planner a achar um caminho ao goal state mais rapidamente, especialmente para o SST cujos parâmetros possuem impacto direto na exploração do state space. As distâncias que utilizamos nos experimentos com RRT* e SST foram, respectivamente, a distância euclidiana e a distância euclidiana com peso; o custo que utilizamos para um movimento foi a distância entre os dois vértices e a combinação do custo foi dada pela soma dos custos. Decidimos utilizar a distância euclidiana com peso para o SST (e para o RRTc, também - para checar se determinado estado pertence à região do goal state) para que pudéssemos balancear melhor, de acordo com a nossa noção de qualidade de caminho, o impacto da orientação do veículo na trajetória gerada. Se não tivéssemos usado pesos, o veículo estar a π cm da posição do goal state seria equivalente (para a função de custo) ao veículo estar na posição do goal state com orientação oposta, o que é muito pior para a navegação. Outro ponto importante para a otimização no RRT* e no SST é decidir quando a trajetória encontrada é boa o suficiente. Nos experimentos da primeira etapa, usamos como condição de término a distância L1 entre o estado inicial e o goal state. Acreditamos que para o SST talvez seja melhor otimizarmos o máximo possível (desde que escolhamos melhor o que otimizar para não termos trajetórias excessivamente retas), pois assim teremos menos incertezas frente o tempo de execução do planner e também não desperdiçaríamos poder computacional (se houver de sobra).

Um grande diferencial entre os resultados entre os métodos baseados em controle e os métodos geométricos é o tempo de execução. Ressaltamos, ainda, que para os métodos geométricos a região do goal state é bem menor quando comparada com a região do goal state dos métodos baseados em controle, com 5 cm e 20 ”cm”(peso para orientação) de raio respectivamente. Observamos que as quantidades de estados pertencentes ao grafo necessários para alcançar o goal state para o RRTg e RRT* foram bem mais baixas em alguns casos na faixa de 10-50 estados, especialmente com range alto e sem adicionar estados intermediários para o RRTg, quando comparadas aos resultados do RRTc e SST (alguns casos na faixa de 1500-3000 estados). Porém, em alguns casos adicionando estados intermediários para o RRTg, a quantidade de estados necessários para chegar à região do goal state foi próxima a 4000 estados e, mesmo assim, em geral, o tempo se manteve abaixo dos tempos dos métodos baseados em controle. A quantidade de estados a mais gerados pelo RRTc e SST, deve-se principalmente a dois motivos: a dimensão maior do state space e por termos que considerar intervalos pequenos de tempo para cada movimento (para o modelo do veículo se aproximar mais da realidade), o que gera partes menores de trajetória.

Em suma, o RRTg e o RRT* geram, normalmente, caminhos deploráveis apesar de serem rápidos. Já as trajetórias do RRTc e SST são, em geral, boas, porém o tempo necessário para gerá-las é alto. Ademais, o RRT* e SST podem ser melhorados ainda mais se utilizarmos uma boa otimização. Por enquanto, o SST, claramente, se apresenta como a melhor opção para o Crawler. Entretanto, o tempo de execução do SST na Raspberry pode ser um entrave para a utilização desse planner. Assim, por todos esses motivos, decidimos mudar o foco da segunda etapa dos experimentos para não mais testar as quatro configurações selecionadas dos planners na Raspberry e, sim, para testar, principalmente, o SST com o objetivo de descobrir seu desempenho na Raspberry e realizar otimizações caso seja necessário e caso a escolha do SST aparente ser, ainda, o melhor caminho a se seguir. Agora, apresentaremos a segunda etapa dos experimentos.

Apesar de estarmos focando no SST, gostaríamos de obter uma ideia do desempenho dos outros planners também na Raspberry. Assim, realizamos alguns experimentos iniciais na Raspberry com os quatro planners e com várias combinações de configuração. Percebemos que, em geral, as execuções levam cerca de 6 a 10 vezes mais tempo para serem executadas na Raspberry. Considerando o limite de tempo de 1 segundo para os métodos geométricos, não houve nenhuma nítida diminuição da qualidade dos seus caminhos tanto para se aproximarem do goal state quanto para a qualidade do caminho inteiro, visto que mesmo com a menor velocidade de execução na Raspberry o planner não foi interrompido por causa do limite de tempo e, sim, por terminar a geração do caminho. O que não foi o caso dos métodos baseados em controle, para estes planners houve uma clara diminuição de qualidade (demonstrada principalmente pelo distanciamento do estado final da trajetória em relação ao goal state). A Figura 32 dá alguns exemplos de trajetórias geradas pelo SST. Para os planners baseados em controle continuamos considerando 0.1 segundo como tempo limite de execução, tempo que como demonstrado nos experimentos na primeira etapa não era suficiente, em geral, para o planner parar antes de atin-

gir o limite de tempo, porém, mesmo assim, conseguíamos uma boa qualidade de trajetória. Portanto, para que possamos utilizar o SST devemos aumentar o limite de tempo e, talvez, fazer mais alguns ajustes que acelerem o planner.

Figura 32 – Exemplos de trajetórias geradas usando SST com tempo limite de execução de 0.1 segundo, selectionRadius=25.0 e pruningRadius=5.0.

5.2.3 SST na Raspberry

Tivemos que mudar diversos parâmetros para melhorar a performance do SST na Raspberry. Conforme fomos executando os experimentos, fomos percebendo uma certa convergência de bons resultados para os valores que acabamos por escolher. Vale notar que há tantos parâmetros do planejamento os quais decidimos mudar que se fôssemos testar cada combinação dos valores que testamos, a gigantesca quantidade de combinações inviabilizaria o experimento. Dessa forma, fomos testando combinações de parâmetros que fazem sentido com base nas suas consequências para o planejamento. Assim, decidimos não apresentar extensas comparações de diversas combinações de parâmetros. O que faremos é apresentar os parâmetros escolhidos e discutir um pouco a respeito das consequências que mudar certos parâmetros podem acarretar.

Antes de discutirmos os parâmetros, devemos ressaltar algumas características que desejamos para o planejamento. Um ponto muito importante é que quanto mais confiança tivermos que o planejamento consegue ser executado sem ultrapassar certo limite de tempo mais confiança teremos na navegação. Outro característica que necessitamos é que apesar de, em alguns casos, o planejamento ter que ser abortado por atingir o limite de tempo, o ponto mais próximo da região do goal state deve estar realmente próximo. Por fim, o planejamento, idealmente, não deve diminuir a quantidade de iterações completas de todo o SNA (todos os módulos serem executados pelo menos uma vez), porém se isso ocorrer, seu impacto deve ser mínimo. Podemos perceber que a última característica desejada desfavorece as outras duas, entretanto com as nossas escolhas de parâmetros buscamos por um equilíbrio adequado ao Crawler.

Comecemos por uma radical mudança de parâmetro, aumentamos o raio da região do goal state de 20 para 60. Esse é um grande fator para o aumento de performance que obtivemos. Como, em geral, as pistas para onde o Crawler foi projetado para andar são bem maiores que 60 cm e estão separadas por bem mais, não há grandes prejuízos ao aumentarmos a região; com exceção da orientação que também é considerada no cálculo da distância. Por isso, aumentamos o peso do subespaço da orientação, de forma que cada ângulo da orientação de diferença equivalha a 2 cm de distância do subespaço de coordenadas (x, y). Assim, consideramos pertencentes à região do goal state estados com no máximo 30 graus de diferença (dependendo, claramente, da distância (x, y) ao goal state). Essas duas mudanças ajudaram muito o planner achar uma trajetória até a região do goal state nos intervalos de tempo testados. Todavia, passamos a confiar um pouco mais na localização do Crawler, especialmente quando o Crawler estiver próximo de um checkpoint - lembremos que se houver um estado pertencente à região do goal state que pertença a outra pista que não é a que está na rota do Crawler e uma trajetória for gerada para lá, a navegação fica totalmente comprometida.

Consideramos a velocidade constante em 50 cm/s. Isso nos ajuda a evitar a geração de pequenos movimentos que velocidades próximas de 0 geravam. A ideia é que com passos maiores podemos chegar mais rapidamente à região do goal state. Ademais, se possível, gostaríamos de manter sempre uma velocidade fixa para que a escolha de uma velocidade não seja uma tarefa a mais necessária. Entretanto, devemos notar que velocidades maiores aumentam os impactos gerados de não termos Δt tendendo a zero. Ademais, ainda há os impactos para o módulo de controle que devemos considerar com essa nossa suposição. Aproveitando que comentamos sobre o Δt, aumentamos o seu valor de 0.2 para 0.3 segundo. O objetivo de aumentar o Δt é o mesmo de aumentar a velocidade mínima, aumentar o tamanho dos movimentos. Como aumentamos o tamanho de cada movimento, tivemos que diminuir a quantidade de passos que damos um controle amostrado, assim, reduzimos a quantidade mínima para 2 e a máxima para 5.

Mudamos o custo mínimo (relacionado a otimização) que uma trajetória gerada deve ter para parar o planejamento. Deixamos o custo mínimo como o valor predefinido da OMPL, que é 0. Assim, o planner utilizará todo o seu tempo limite em todas as execuções. Não nos demos o trabalho de procurar por um valor mínimo adequado, pois, em todos os casos que observamos, sempre há algo para melhorar na trajetória. Mantemos o custo como o tamanho da trajetória. Acreditamos, entretanto, que teremos mais informações para mudar a otimização conforme formos realizando experimentos com o Crawler navegando autonomamente.

Por fim, o tempo limite de execução. A princípio, decidimos testar de forma mais minuciosa três valores 0.8, 1.0 e 1.2 segundo. Para esses testes, estávamos com muita variação de resultados por causa da aleatoriedade no planejamento, por isso aumentamos para 30 execuções por saída do módulo de DC (usamos as mesmas dos experimentos da primeira etapa), totalizando 240 execuções por tempo limite de execução. Na Tabela 7, podemos observar a quantidade de execuções que não chegaram à região do goal state para os três limites de tempo e a distância do estado mais próximo do goal state para cada um desses casos. Assim, percebemos

que com um aumento de 50% (0.8 para 1.2) no tempo de execução, reduzimos pela metade a quantidade de não chegadas ao goal state. Devemos notar que com mais tempo, além de mais execuções atingindo o objetivo do planejamento, conseguimos otimizar mais os caminhos que terminaram o planejamento com alguma folga. Entretanto, devemos admitir que, em geral, não houve nenhuma nítida melhoria da qualidade das trajetórias. Para os três casos, as trajetórias possuem uma qualidade muito boa para a grande maioria dos casos. Os resultados com esses três tempos limite de execução são excelentes quando comparados com o que tínhamos antes. Na Figura 33, podemos observar exemplos de trajetórias geradas com limite de 1.0 segundo. Quando comparamos com a Figura 32, podemos observar o aumento na qualidade das trajetórias. Por exemplo, para o segundo e o quarto OG, a orientação final da trajetória ficou muito melhor. Outra questão é uma maior qualidade considerando a trajetória inteira, isso pode ser observado principalmente, com curvas não tão abertas para o primeiro OG e sem curvas desnecessárias para o segundo OG.

Tabela 7 – Distância do estado mais próximos ao goal state para os planejamentos que não chegaram à região do goal state considerando os diferentes limites de tempo em segundo.

	[60, 70)	[70, 80)	[80, 90)	[90, 100)	[100, 110)	Total
0.8	11	5	5	3	2	26
1.0	13	7	0	0	0	20
1.2	8	4	0	1	0	13

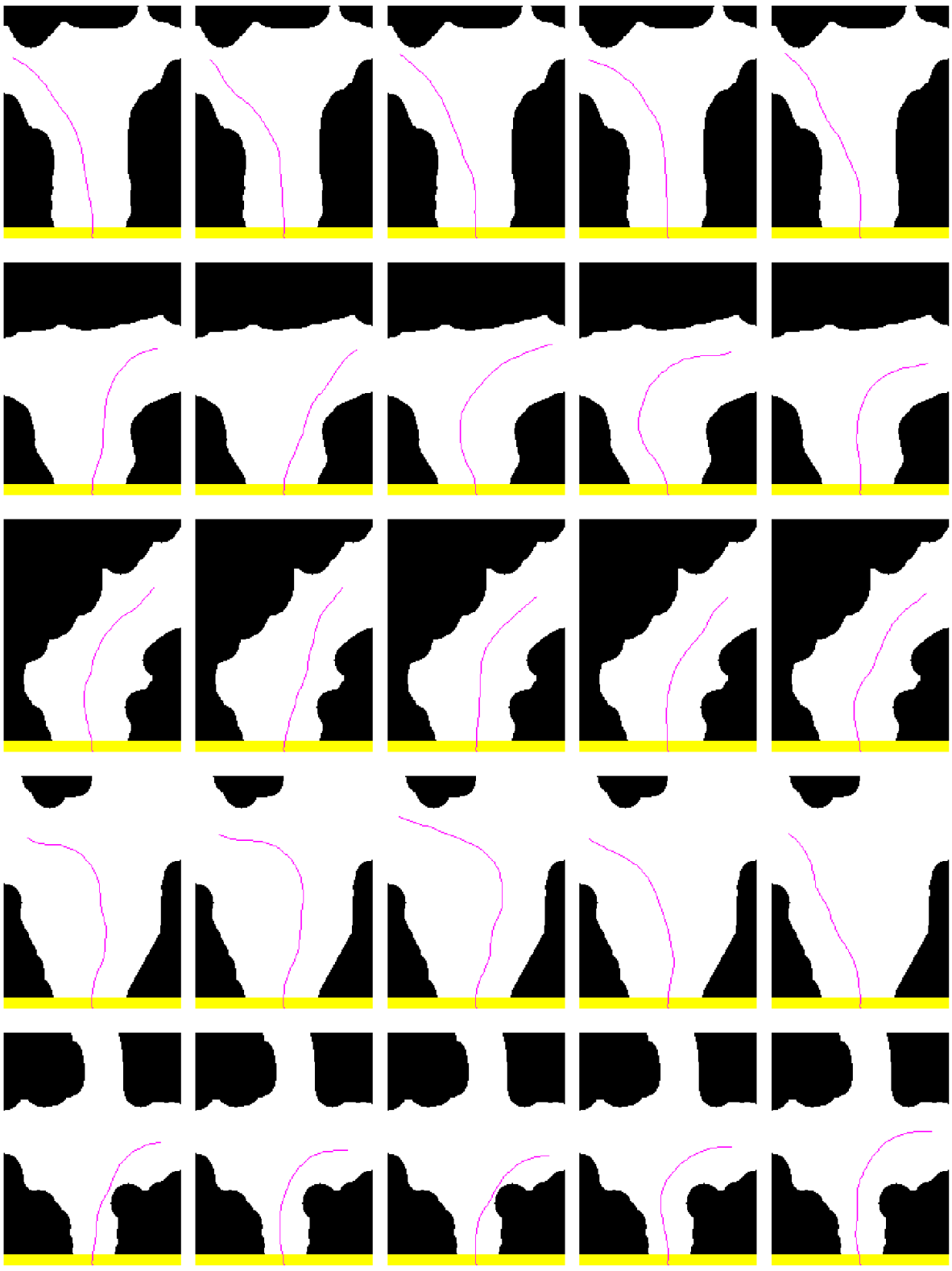


Figura 33 – Exemplos de trajetórias geradas usando SST na Raspberry Pi com tempo limite de execução de 1.0 segundo, selectionRadius=25.0 e pruningRadius=5.0.

6 DISCUSSÕES FINAIS

6.1 A PARTIR DA SEGMENTAÇÃO SEMÂNTICA ATÉ A GERAÇÃO DE UMA TRAJETÓRIA

Já discutimos separadamente vários aspectos e as motivações relacionadas ao módulo de Percepção, DC e PM. Porém, chegou o momento de nos dirigirmos diretamente a uma questão que afeta todos os módulos e cujas discussões bem mais aprofundadas guardamos para o final, o tempo. Há, ainda, vários pontos que não abordamos relacionados ao tempo. O nosso foco até agora foi em equilibrar qualidade dos resultados com o tempo de execução. Mas será que o tempo final da execução de todos módulos que desenvolvemos do SNA permite o Crawler navegar ininterruptamente? Ademais, como fazemos para sincronizar todos os módulos? A sincronização é fundamental para o funcionamento do SNA, não queremos que módulos usem informações muito antigas como se fossem atuais, com a desculpa de que foi o que conseguiram receber a tempo. Sem mais delongas, antes de apresentarmos os experimentos, abordaremos algumas questões relacionadas com o tempo, começando por uma grande correção que tivemos que fazer.

6.1.1 Correção para a detecção de pistas e escolha do goal state

A primeira implementação que fizemos de tudo apresentado no Capítulo 4 foi em Python 3. Escolhemos essa linguagem por causa das opções de esqueletonização presentes na biblioteca scikit-image. Ademais, apresentamos os testes realizados para medir o tempo da esqueletonização que chamamos de ZS e do cálculo de distância às bordas, os quais mostraram-se muito bons para a Raspberry (ambos ficavam abaixo de 100 milissegundos na maior parte dos casos). Entretanto, outras partes do nosso método para escolher o goal state demandaram bastante tempo, as iterações levavam em média 1,25 segundo. As piores partes foram: o fechamento morfológico (465 milissegundos); a criação de algumas listas (como pixels pertencentes ao esqueleto, pixels pertencentes ao esqueleto distantes das bordas e algumas necessárias para a construção do grafo) (740 milissegundos) e a construção do grafo (150 milissegundos). Inicialmente, havíamos aceitado esses valores, pois poderíamos otimizar, posteriormente, as partes que degradaram completamente o desempenho do método. Porém, assim que descobrimos o tempo que o planejamento de movimentação demanda, pensamos em duas alternativas: (i) deixar apenas o pré-processamento com as operações de morfologia matemática e a esqueletonização em Python e o restante faríamos em C++. Para essa ideia, precisaríamos de uma comunicação eficiente entre os dois processos - o processo desenvolvido em Python precisaria enviar o OG modificado pelas operações de morfologia matemática para o processo desenvolvido em C++. (ii) Desenvolver tudo em C++, inclusive a esqueletonização. Escolhemos a segunda opção, pois já estávamos usando a OMPL em C++, dessa forma poderíamos ter apenas um processo cuidando da detecção de pistas, escolha do goal state e do plane-

jamento de movimentação e, assim, não precisaríamos nos preocupar com uma entrega de informação eficiente entre os dois processos. Ademais, com a segunda opção, por estarmos usando C++ e estruturas de dados mais adequadas, poderíamos otimizar praticamente todo o processo. O único ponto negativo dessa opção é que teríamos que implementar ou procurar por uma implementação da esqueletonização, da obtenção da distância às bordas e das operações de morfologia matemática. Para o cálculo da distância às bordas e para as operações de morfologia matemática usamos OpenCV. Já a esqueletonização, decidimos implementar, pois não achamos com tanta facilidade nenhuma implementação confiável do método ZS. Escolhemos manter o ZS, principalmente, por causa do seu bom tempo de execução em Cython.

Não houve problemas para implementar o ZS, o método descrito no artigo é, extremamente, simples e ainda pudemos nos basear no código em Cython da scikit-image. Ao fim conseguimos reduzir o tempo médio de todo o processo para cerca de 550 milissegundos. Entretanto, percebemos que desses 550 milissegundos, 480 eram usados para realizar a esqueletonização. Infelizmente, tivemos outras prioridade relacionadas ao SNA e não pudemos investigar o motivo pelo qual o método que implementamos levou muito mais tempo que a implementação do scikit-image em Cython. Devemos salientar que esse foi o nosso primeiro contato com Cython e talvez tenhamos deixado passar despercebido algum detalhe/otimização feita na implementação do scikit-image. Assim, acreditamos que ainda haja um grande potencial de otimização para o processo de detecção de pistas e definição do goal state. Contudo, a redução para 550 milissegundos já viabiliza muito mais a utilização de todo esse processo no SNA.

6.1.2 Comunicação entre a Nano e a Raspberry

Sabemos que o tempo médio para a Nano gerar a imagem segmentada e o OG é cerca de 450 milissegundos. Já o tempo médio para a Raspberry gerar o goal state é 550 milissegundos, porém o tempo varia bastante, chegando a 700 milissegundos algumas vezes. Por fim, das três opções de tempo limite que testamos para o planner, escolhemos a princípio o tempo de 1 segundo (1000 milissegundos). Como todo o processo de geração do goal state (incluindo a esqueletonização) está na Raspberry, assim como o planejamento de movimentação e ambos estão em C++, decidimos juntar esses dois processos para evitar mais atrasos ocasionados pela troca de informação entre essas duas partes. Poderíamos ter usado algum mecanismo eficiente de compartilhamento de memória para manter dois processos, porém não vimos muitas vantagens em fazer isso. Portanto, ficamos com um processo na Raspberry que leva em média 1550 milissegundos, cerca de 3,5 vezes mais que o processo na Nano.

Vamos discutir então como fizemos a comunicação entre os dois processos. Usamos socket com TCP. Lembremos que a Nano e a Raspberry estão conectadas por um cabo Ethernet. Deixamos o processo da Raspberry como servidor e o da Nano como cliente. O processo na Nano não foi muito modificado, apenas adicionamos a configuração inicial do cliente socket e assim que o OG é gerado o processo da Nano espera o processo da Raspberry avisar que está

pronto para receber o OG; com a confirmação recebida, o OG é enviado. Já o processo da Raspberry teve mais modificações. A primeira foi a adição da configuração inicial do servidor socket. Além disso, criamos mais 3 threads para o processo: uma para receber o OG (receiver), outra para fazer tudo desde o pré-processamento com morfologia matemática até gerar a trajetória (processor) e outra para copiar o OG da receiver para a processor (copier). Vamos começar discutindo as razões de usarmos mais de uma thread. Não gostaríamos que a Nano tivesse que esperar pela Raspberry para fazer as suas tarefas. A Nano poderia ir enviando todas os OGs gerados sem se preocupar com o processo da Raspberry, porém teríamos que fazer o processo da Raspberry cuidar para usar sempre o último OG disponível, o que achamos que poderia nos expor a muito mais detalhes relacionados a sockets que teríamos que cuidar. Por esses motivos, e por termos mais facilidade em trabalhar com threads do que com sockets, resolvemos usar mais de uma thread. Agora, vamos discutir o motivo de usarmos mais de duas threads. Gostaríamos que a Raspberry pudesse receber um novo OG ao mesmo tempo que estivesse fazendo o planejamento de movimentação, por exemplo, e gostaríamos que a Raspberry lesse o OG assim que o recebesse. Se estivéssemos usando apenas duas threads, a receiver poderia modificar o OG enquanto a processor estivesse o usando, o que tornaria o OG inválido. Há mais diversos casos que poderíamos considerar, mas todos nos trouxeram motivos para usar três threads. Vamos agora, considerar a solução com as três threads. Mas antes uma pequena observação: para mantermos a consistência do OG, a copier deve copiar o OG apenas quando as outras duas threads não estão executando. Apenas a receiver começa executando, assim que ela termina de ler um OG, ela dá uma oportunidade para a copier executar e espera receber uma oportunidade de executar. Se a copier puder copiar (nem a processor nem a receiver podem estar executando), ela copia o OG e dá uma oportunidade para a receiver e para a processor executarem e passa a esperar pela sua próxima oportunidade de executar. Se a processor estiver executando e a receiver não, a copier apenas dá uma oportunidade para a receiver executar e passa a esperar pela sua próxima oportunidade de executar. Caso contrário, a copier apenas espera pela sua próxima oportunidade de executar. Assim que a processor puder executar, ela realiza tudo desde o pré-processamento do OG com morfologia matemática até o planejamento de movimentação e, ao fim, dá mais uma oportunidade para a copier executar. Essa oportunidade dada pela processor à copier, faz com que o processo não precise esperar para processar o próximo OG caso algum já esteja disponível. Usamos threads, mutexes e semáforos POSIX para implementar esse comportamento das threads.

6.1.3 Validação de um estado no planejamento de movimentação

Um pequeno detalhe que deixamos de comentar anteriormente é como é feita a checagem se um estado é válido durante o planejamento de movimentação. Para os experimentos específicos dos planners, simplesmente, checamos se a posição do estado está como navegável ou não navegável no OG - podemos perceber que em diversas das imagens apresentadas o caminho passa demasiadamente próximo das bordas. Lembremos que o Crawler não é um ponto,

ele possui largura e comprimento e que consideramos para o planejamento que a região do Crawler que o representa é a região abaixo da câmera, com isso, a parte mais distante dessa região é a sua frente (por volta de 45 cm). Como queríamos não trazer ainda mais atrasos para o planejamento, decidimos manter a simplicidade da validação de um estado. Assim, para os experimentos de todo processo, além de checarmos se a posição do estado está em uma área navegável, checamos também se a distância daquela célula do OG é maior ou igual a 50 cm das bordas. Poderíamos ter usado alguma solução que levasse em conta a orientação do Crawler, entretanto isso poderia trazer mais atrasos (por ter que checar mais células do OG, por exemplo) ou teríamos que desenvolver uma solução mais complexa, o que não achamos necessário. Algumas das razões para acharmos isso é que não há um desperdício grande de espaço navegável, ademais podemos estar excluindo espaços que foram indevidamente percebidos como navegáveis pela segmentação semântica e dessa forma ajudamos a manter o Crawler longe das bordas. Essas razões também se encaixam muito bem para outros ajustes que podem ser feitos a parâmetros relacionados com a distância às bordas, como a distância mínima para não excluir um ponto do esqueleto.

6.1.4 Experimentos de todo o processo

Realizamos alguns experimentos com os módulos de Percepção, DC e PM embarcados no Crawler. Gostaríamos de ter alguns aspectos relevantes para comentar aqui, mas tudo ocorreu conforme esperávamos e conforme todos os outros experimentos apresentados no decorrer do texto. Em geral, a segmentação semântica foi a etapa que mais limitou a qualidade da trajetória gerada. Tudo relacionado com a esqueletonização, definição do goal state e planejamento de movimentação ocorreu normalmente. Algo que seria extremamente relevante testar, mas que ainda não é possível, é se o Crawler consegue seguir a trajetória gerada e se as várias trajetórias conseguem fornecer uma boa navegação. Para isso, precisaríamos do módulo de Controle e de Localização. Há alguns pontos que com o acréscimo desses módulo precisam ser ajustados, como o start state do planejamento de movimentação. Para o start state precisamos saber o estado do Crawler quando a imagem do OG atual foi capturada pela câmera e ao iniciarmos o planejamento precisamos prever o estado do Crawler quando o controle começará a executar o planejamento que será feito. Para saber o estado do Crawler no momento em que capturamos a imagem, podemos simplesmente manter alguns estados passados. Já para prever o estado futuro, podemos olhar na trajetória planejada anteriormente, onde o Crawler deverá estar no instante que acabar o planejamento atual. Lembremos que o planner possui um tempo limite de execução. Com o estado antigo e o estudo futuro, podemos ”posicionar”corretamente o start state no OG atual. Para que essa solução funcione, o caminho seguido pelo Crawler não pode variar muito da trajetória gerada, mas isso só saberemos com os módulos de Controle e Localização prontos.

6.2 ALGUMAS CONSIDERAÇÕES

Discutimos tanto tudo o que estudamos, desenvolvemos e testamos, mas ainda não esclarecemos todos os pontos que ficaram em aberto, nem destacamos as diversas melhorias que podem ser feitas, nem as partes que não foram feitas ainda para o Crawler e nem o que gostaríamos de ter estudado e abordado aqui. Comecemos por esse último ponto, em algumas ocasiões no texto afirmamos que o foco deste trabalho, a princípio, era em SNAs e no planejamento de movimentação. Tentamos ao máximo manter o foco original, porém tivemos que desenvolver tantos outros aspectos do Crawler para poder ter uma discussão concreta do planejamento de movimentação que, infelizmente, não pudemos dedicar tanto tempo nem avançar o quanto desejávamos nessa área. Por exemplo, gostaríamos de ter apresentado diversos planners que foram usados em veículos autônomos reais (como alguns veículos desenvolvidos para o DARPA Urban Challenge) que foram postos a provas e que tiveram um excelente desempenho. Planejávamos, inclusive, implementar alguns desses planners e testá-los no Crawler. Além disso, gostaríamos de ter abordado diversos SNAs já desenvolvidos e compará-los.

Vamos discutir, agora, o que ainda não foi desenvolvido para o Crawler. O módulo de Controle foi desenvolvido em parte. Já há uma primeira versão do módulo, mas ainda nenhum teste foi realizado com os dados obtidos pelos sensores do Crawler e nem foi feita a identificação dos parâmetros. O módulo de Localização está associado ao estimador de estados do Controle, por isso não o temos. Como ainda não temos uma maneira decente de estimar o estado do Crawler, não temos como testar se os diversos parâmetros escolhidos para a percepção das pistas e definição do goal state estão adequados para fornecer a robustez necessária diante dos erros de localização. A interface com o GPS está em desenvolvimento, não há nenhum problema específico com esta interface, apenas tivemos outras prioridades.

Há tanto que pode ser melhorado no Crawler. Esperávamos um desempenho muito melhor da Raspberry Pi, tanto que no início achávamos que o que atrasaria uma iteração completa do SNA seria a segmentação semântica, mas como vimos a segmentação semântica levou muito menos tempo que as tarefas da Raspberry e ainda, praticamente, metade do seu tempo pode ser bem otimizado. Acreditamos que vale a pena realizar alguns testes com alguns modelos mais novos da Raspberry, pois acelerar as suas tarefas é algo que pode melhorar bastante a navegação. Mas, talvez, a melhoria mais necessária para o SNA atual esteja relacionada com a qualidade da segmentação semântica. Diversas vezes, a segmentação semântica deixou a desejar. Notamos dois principais problemas: sombras e sinalizações horizontais prejudicaram muito a segmentação semântica e partes mais distantes da via não eram identificadas muitas vezes. Acreditamos que adicionar algumas imagens ao dataset de treinamento com a câmera na sua altura e pitch atual ajude a melhorar a segmentação semântica. Entretanto, com tempo, podemos testar outros modelos de HRNets ou ainda outras redes neurais. Outra questão que nos incomoda um pouco é o ângulo de abertura da câmera. Não ressaltamos muito esse ponto ao apresentarmos os OGs gerados, mas uma parte considerável do OG (a parte mais próxima do Crawler) fica como não navegável porque a câmera não enxerga aquela região. Esse problema

fica bem evidente na primeira linha da Figura 16. Resolver esse problema é simples, apenas precisamos de uma câmera com um ângulo de abertura maior.

Continuamos com as possíveis melhorias do Crawler. Há a questão da esqueletonização que desenvolvemos ter ficado com um desempenho bem pior que a esqueletonização do scikit-image. Estudar um pouco mais de Cython pode ajudar a entender os motivos pelos quais houve uma diferença de desempenho tão grande. Isso pode ajudar aliviar bastante o tempo de execução desse processo na Raspberry. Um ponto que não comentamos praticamente nada está relacionado com a execução dos testes com o Crawler na rua. Preparar toda a estrutura necessária leva bastante tempo e, atualmente, não há muito dinamismo para visualizar os resultados. Automatizar certos aspectos para obter os resultados dos testes e para realizar essas configurações iniciais pode facilitar e acelerar muito a realização de experimentos.

Falamos diversas vezes nós decidimos, nós escolhemos, nós desenvolvemos ... e consideramos nós, pois com certeza este trabalho não é só meu, diversas pessoas contribuíram para este trabalho e para o projeto do Crawler diretamente e indiretamente. Mas acredito que neste momento, valha a pena destacar o que não fui eu quem fez e o que eu, com todo o suporte que recebi, fiz para este projeto. Como eu já havia dito, o que foi feito do módulo de Controle e de Localização foi o Matheus Wagner que desenvolveu. Não tive nenhuma participação na interface do GPS, nem na montagem da estrutura física do Crawler (com exceção do suporte da câmera, anteriormente a câmera ficava muito próxima do chão, o que deixava a segmentação semântica pior ainda). Não tive participação no desenvolvimento da parte eletrônica do Crawler. Não desenvolvi os primeiros protótipos da interface com o Arduino e com o wheel encoder. Podemos passar agora para o que eu realmente fiz. Fiz correções, modificações e melhorias na interface com o Arduino e com o wheel encoder. Desenvolvi a interface com a IMU. Incluí as medidas de segurança para caso as comunicações entre o Arduino e a Raspberry e entre o controle remoto e a Raspberry caiam - são medidas simples porém se elas existissem antes, alguns componentes do Crawler não teriam queimado em um acidente. Desenvolvi o módulo de Percepção, incluindo todo o processamento feito na Nano e na Raspberry. Treinei diversas HRNets, porém decidi usar a rede treinada por outro contribuidor do Crawler. Desenvolvi, também, os módulos de DC e de PM. Ressalto que com ”desenvolvi”não digo apenas fazer o código que está no Crawler, incluo também buscar garantias de o que foi feito realmente funciona, como realizar testes, desenvolver maneiras de visualizar o que foi feito, adquirir bases teóricas, dentre várias outras formas. Além disso, escolhi partes do ODD do Crawler e estruturei todo o seu SNA. Por fim, talvez as questões mais importantes para todo o trabalho tenham sido tentar ao máximo fornecer uma firme base a todos nossos passos e raciocinar sobre os caminhos que deveríamos seguir.

REFERÊNCIAS

ARASARATNAM, I.; HAYKIN, S. Cubature kalman filters. **IEEE Transactions on Automatic Control**, Institute of Electrical and Electronics Engineers (IEEE), v. 54, n. 6, p. 1254–1269, jun. 2009. Disponível em: https://doi.org/10.1109/tac.2009.2019800.

DIJKSTRA, E. W. A note on two problems in connexion with graphs. **Numerische Mathematik**, Springer Science and Business Media LLC, v. 1, n. 1, p. 269–271, dez. 1959. Disponível em: https://doi.org/10.1007/bf01386390.

GLADWELL, I. Boundary value problem. **Scholarpedia**, Scholarpedia, v. 3, n. 1, p. 2853, 2008. Disponível em: https://doi.org/10.4249/scholarpedia.2853.

GUO, Z.; HALL, R. W. Parallel thinning with two-subiteration algorithms. **Communications of the ACM**, Association for Computing Machinery (ACM), v. 32, n. 3, p. 359–373, mar. 1989. Disponível em: https://doi.org/10.1145/62065.62074.

HART, P.; NILSSON, N.; RAPHAEL, B. A formal basis for the heuristic determination of minimum cost paths. **IEEE Transactions on Systems Science and Cybernetics**, Institute of Electrical and Electronics Engineers (IEEE), v. 4, n. 2, p. 100–107, 1968. Disponível em: https://doi.org/10.1109/tssc.1968.300136.

KARAMAN, S.; FRAZZOLI, E. **Sampling-based Algorithms for Optimal Motion Planning**. 2011.

KRIZHEVSKY, A.; SUTSKEVER, I.; HINTON, G. E. Imagenet classification with deep convolutional neural networks. In: PEREIRA, F. et al. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2012. v. 25. Disponível em: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

LAVALLE, S. Rapidly-exploring random trees : a new tool for path planning. **The annual research report**, 1998.

LAVALLE, S. **Planning algorithms**. Cambridge New York: Cambridge University Press, 2006. ISBN 0521862051.

LI, Y.; LITTLEFIELD, Z.; BEKRIS, K. E. **Asymptotically Optimal Sampling-based Kinodynamic Planning**. 2014.

LIU, S. et al. Creating autonomous vehicle systems. **Synthesis Lectures on Computer Science**, Morgan & Claypool Publishers LLC, v. 6, n. 1, p. i–186, out. 2017. Disponível em: https://doi.org/10.2200/s00787ed1v01y201707csl009.

MOLL, M.; SUCAN, I. A.; KAVRAKI, L. E. Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. **IEEE Robotics & Automation Magazine**, Institute of Electrical and Electronics Engineers (IEEE), v. 22, n. 3, p. 96–102, set. 2015. Disponível em: https://doi.org/10.1109/mra.2015.2448276.

MONTEMERLO, M. et al. Junior: The stanford entry in the urban challenge. In: . [S.l.: s.n.], 2009. p. 91–123.

RAJAMANI, R. **Vehicle Dynamics and Control**. Springer US, 2012. Disponível em: https://doi.org/10.1007/978-1-4614-1433-9.

RATEKE, T.; WANGENHEIM, A. von. **Road surface detection and differentiation considering surface damages**. 2021. Disponível em: https://doi.org/10.1007/s10514-020-09964-3.

REIF, J. H. Complexity of the mover's problem and generalizations. In: **20th Annual Symposium on Foundations of Computer Science (sfcs 1979)**. IEEE, 1979. Disponível em: https://doi.org/10.1109/sfcs.1979.10.

REINHOLTZ, C. et al. Odin: Team VictorTango's entry in the DARPA urban challenge. In: **Springer Tracts in Advanced Robotics**. Springer Berlin Heidelberg, 2009. p. 125–162. Disponível em: https://doi.org/10.1007/978-3-642-03991-1_4.

RUSSAKOVSKY, O. et al. **ImageNet Large Scale Visual Recognition Challenge**. 2014.

SAMAK, C. V.; SAMAK, T. V.; KANDHASAMY, S. **Control Strategies for Autonomous Vehicles**. 2020.

ŞUCAN, I. A.; MOLL, M.; KAVRAKI, L. E. The Open Motion Planning Library. **IEEE Robotics & Automation Magazine**, v. 19, n. 4, p. 72–82, December 2012. https://ompl.kavrakilab.org.

TRANSPORTATION, U. D. of; ADMINISTRATION, N. H. T. S. **Federal Automated Vehicles Policy Accelerating the Next Revolution In Roadway Safety**. 2016. Disponível em: https://www.transportation.gov/AV/federal-automated-vehicles-policy-september-2016.

URMSON, C. et al. Autonomous driving in urban environments: Boss and the urban challenge. In: **Springer Tracts in Advanced Robotics**. Springer Berlin Heidelberg, 2009. p. 1–59. Disponível em: https://doi.org/10.1007/978-3-642-03991-1_1.

WANG, J. et al. **Deep High-Resolution Representation Learning for Visual Recognition**. 2019.

WASLANDER, S.; KELLY, J. **Motion Planning for Self-Driving Cars**. 2019. Disponível em: https://www.coursera.org/learn/motion-planning- self-driving-cars?specialization=self-driving-cars.

ZHANG, T. Y.; SUEN, C. Y. A fast parallel algorithm for thinning digital patterns. **Communications of the ACM**, Association for Computing Machinery (ACM), v. 27, n. 3, p. 236–239, mar. 1984. Disponível em: https://doi.org/10.1145/357994.358023.

ZIEGLER, J. et al. Making bertha drive—an autonomous journey on a historic route. **IEEE Intelligent Transportation Systems Magazine**, Institute of Electrical and Electronics Engineers (IEEE), v. 6, n. 2, p. 8–20, 2014. Disponível em: https://doi.org/10.1109/mits.2014.2306552.

Da segmentação semântica ao planejamento de movimentação em veículo autônomo

Nícolas Goeldner[1]

[1]Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Florianópolis – SC – Brasil

nicolas.goeldner@grad.ufsc.br

Abstract. *Autonomous vehicles have been receiving a lot of attention for the past years. These vehicles are composed of an autonomous navigation system, which has, on several occasions, a specific motion planning component. So, in this work, we propose a study on autonomous navigation systems and motion planning methods. As part of this study, we have developed the autonomous navigation system architecture and some modules of an autonomous vehicle called Crawler. We present in details parts of the Perception module, the Behavioural Decision module and the Motion Planning module. We try as much as possible to discuss the reasons why each decision was taken during the development of the modules. Chapter 1 is a brief introduction to the work. Chapter 2 discusses some basic concepts for understanding what was developed. Chapter 3 introduces the Crawler. Chapter 4 explains the definition of goal states for the Crawler. Chapter 5 covers the motion planning for the Crawler. Finally, Chapter 6 discusses some final details and concludes the work.*

Resumo. *Veículos autônomos têm recebido muita atenção ultimamente. Esses veículos são compostos por um sistema de navegação autônoma, o qual possui, em diversas ocasiões, um componente específico de planejamento de movimentação. Assim, neste trabalho, propomos um estudo de sistemas de navegação autônoma e de métodos de planejamento de movimentação. Como parte do estudo, desenvolvemos a arquitetura do sistema de navegação autônoma e alguns módulos de um veículo autônomo chamado de Crawler. Apresentamos, detalhadamente, parte do módulo de Percepção, o módulo de Decisão de Comportamento e o módulo de Planejamento de Movimentação. Tentamos trazer ao máximo as motivações para cada decisão tomada durante o desenvolvimento dos módulos. O Capítulo 1 é uma breve introdução ao trabalho. O Capítulo 2 discorre sobre alguns conceitos básicos para o entendimento do que foi desenvolvido. O Capítulo 3 apresenta o Crawler. O Capítulo 4 explica a definição de goal states para o Crawler. O Capítulo 5 aborda o planejamento de movimentação do Crawler. Por fim, o Capítulo 6 discute alguns últimos detalhes e encerra o trabalho.*

1. Introdução

A navegação autônoma é uma área que vem ganhando muita atenção nos últimos anos. Um dos maiores eventos que impulsionou o desenvolvimento de veículos autônomos foi uma competição chamada DARPA Urban Challenge (DUC), realizada em 2007. Nessa

competição, os veículos deveriam trafegar por 96km sem nenhuma intervenção humana. Apenas seis times conseguiram desenvolver sistemas de navegação autônoma (SNA) que guiaram seus respectivos automóveis até a linha de chegada sem descumprir nenhuma das regras da competição. Outros grandes expoentes para a navegação autônoma são as empresas Tesla, Waymo, Apple, Baidu e várias outras que possuem projetos de veículos autônomos, sendo que muitas dessas empresas já possuem veículos trafegando por algumas cidades para realizar testes. Há inclusive projetos grandes, como o Apollo da Baidu, que disponibilizam grande parte do que desenvolvem publicamente. A navegação autônoma pode nos trazer diversas vantagens, redução dos números de acidentes e congestionamentos de trânsito, melhora na qualidade de vida e diminuição da emissão poluentes são apenas algumas. Porém, ainda há vários problemas a serem resolvidos se quisermos desfrutar dessas vantagens totalmente. Apresentamos este trabalho como o resultado de parte de um estudo de SNAs com foco no módulo de Planejamento de Movimentação - um dos módulos de um SNA.

Na próxima seção, discutimos uma arquitetura usual de SNAs com detalhes de cada módulo e introduzimos o problema de planejamento de movimentação. Em seguida, abordamos o Crawler trazendo detalhes de hardware, especificidades da organização de seu SNA e alguns detalhes de componentes essenciais para a navegação que, entretanto, não possuem muito espaço no restante deste trabalho. Na quarta seção, apresentamos o problema da definição de goal states considerando algumas especificidades do Crawler; trazemos, também, a solução proposta e experimentos realizados. A solução compõe o módulo de Decisão de Comportamento e parte do módulo de Percepção. Na quinta seção, voltamos ao planejamento de movimentação, porém, agora, no contexto do Crawler. Na sexta e última seção, apresentamos esclarecemos o que já foi feito para o projeto do Crawler e por fim, concluímos. O código desenvolvido para o trabalho pode ser encontrado em https://codigos.ufsc.br/nicolas.goeldner/Crawler-SNA ou em https://github.com/ngoeldner/Crawler-SNA. Haverá um aviso indicando qual repositório está sendo mantido.

2. Conceitos Básicos

2.1. SNAs

De acordo com [Liu et al. 2017], uma organização comum para os componentes de um SNA é a divisão em módulos sensoriais e módulos de planejamento e controle. Como podemos observar na Figura 1, os módulos sensoriais são responsáveis pela percepção do ambiente e, também, pela identificação do estado do veículo em relação ao ambiente. A partir das informações obtidas pelos módulos sensoriais, os módulos de planejamento e controle, são responsáveis por planejar e executar a movimentação para que o veículo chegue ao destino final. Podemos dividir essas responsabilidades de planejamento e execução em cinco módulos: (i) Predição do Tráfego, (ii) Roteamento, (iii) Decisão de Comportamento, (iv) Planejamento de Movimentação e (v) Controle. A Predição do Tráfego é responsável por analisar as percepções do ambiente para prever os caminhos pelos quais outros veículos, pedestres, ciclistas e vários outros objetos móveis provavelmente irão passar e quando passarão por determinado ponto. O Roteamento cuida da rota que um veículo deve fazer para chegar ao seu destino final, por exemplo, por quais ruas deve passar. Já a Decisão de Comportamento é responsável por determinar como o veículo deve se comportar para que siga a rota dada pelo módulo anterior, por exemplo, trocar

de faixa, continuar na mesma faixa, esperar o semáforo abrir, etc. O Planejamento de Movimentação determina, dentre algumas características, por quais pontos da pista, com qual velocidade, aceleração e orientação o veículo deve passar por tais pontos para que respeite o comportamento escolhido. Por último, o Controle é responsável por traduzir as informações de velocidade, aceleração e orientação em comandos que devem ser dados aos atuadores do veículo.

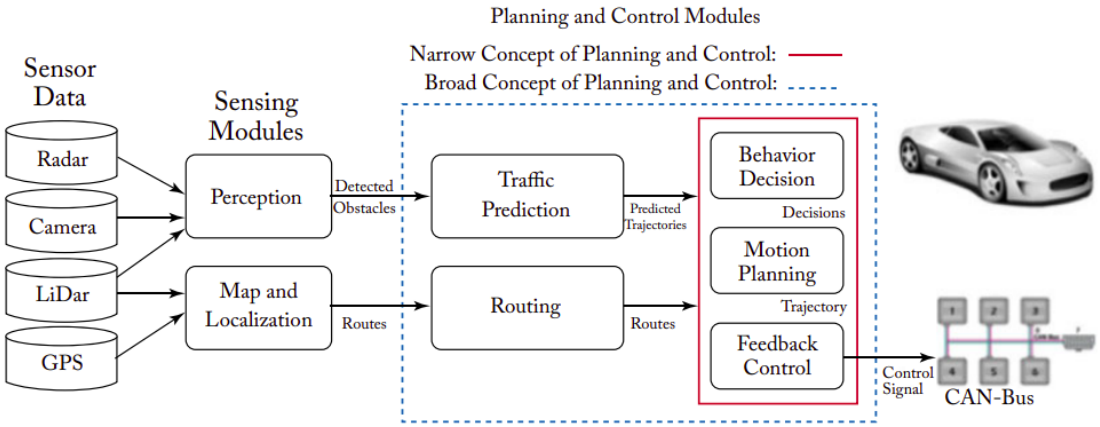


Figura 1. Arquitetura usual de SNA. Fonte: [Liu et al. 2017]

Vamos fazer, agora, uma analogia com a forma com que uma pessoa dirige para esclarecer o papel de cada um desses módulos. Em geral, quando uma pessoa quando ela vai dirigir para algum lugar, ela pensa nas ruas que vai passar e já imagina o caminho em um nível não tão detalhado (Roteamento). Quando uma pessoa está dirigindo, ela precisa ter uma noção de localização para saber por qual rua ela deve seguir e para não colidir com os outros veículos (Localização). Ela deve se atentar a outros veículos, pedestres que tentam atravessar a pista, semáforos e placas de pare (Percepção). Com essas informações do ambiente, ela deve tomar uma decisão do que fazer, como trocar de faixa, esperar o pedestre atravessar, cuidar com o veículo da frente (Decisão de Comportamento), em seguida ela deve pensar como executar determinado comportamento (Planejamento de Movimentação). Por exemplo, ela deve reduzir a velocidade para virar à direita? Ou em que ponto ela deve começar a virar o volante? Com essa ideia do que fazer, basta executar a movimentação planejada (Controle).

Em relação ao mapeamento, podemos construir um mapa chamado de occupancy grid, que diz por onde o veículo pode navegar. O occupancy grid pode ser pensado como uma matriz 2D e as células apresentam valores 0 ou 1, sendo que 0 significa que o local está ocupado e 1 livre, por exemplo. O occupancy grid representa uma região próxima do veículo e representa o ambiente como se estivesse vendo a rua de cima.

O módulo de Decisão de Comportamento pode representar o comportamento selecionado como um objetivo de estado final do veículo que se chama goal state. Esse estado pode ser composto de uma posição, orientação, velocidade e várias outras características. A partir desse goal state, do estado atual do veículo, do occupancy grid e das previsões de movimentação dos componentes dinâmicos do ambiente, o módulo de Planejamento de Movimentação deve traçar uma trajetória de estados que seja ao menos segura (sem colisão) e realizável (levando em conta as restrições físicas do veículo).

2.2. Planejamento de Movimentação

Agora, vamos abordar o problema de planejamento de movimentação - Motion Planning. Como comentamos há pouco, o módulo de planejamento de movimentação, enquanto busca uma trajetória de um estado inicial a um estado final (o goal state), tenta tratar três pontos: restrições físicas do veículo, obstáculos fixos e obstáculos móveis. Para que haja um bom entendimento das consequências de cada uma dessas dificuldades, apresentamos três problemas. Esses problemas são: Piano Mover's Problem (PMP), o planejamento de movimentação com restrições diferenciais (Motion Planning Under Differential Constraints Problem) e o two-point Boundary Value Problem (BVP).

2.2.1. Piano Mover's Problem

Começando pelo Piano Mover's Problem. Antes de apresentarmos uma definição formal do problema, apresentamos um exemplo. Imaginemos que queremos transportar um piano, representado como um retângulo na Figura 2, a partir da sua configuração inicial dada por $Pi = (x_0, y_0, \theta_0))$ que correspondem à posição do piano e a orientação do piano. Queremos levar o piano até uma configuração final $Pf = (x_f, y_f, \theta_f)$. Precisamos cuidar com os obstáculos no ambiente, que nesse exemplo são as paredes. Podemos rotacionar e transladar o piano à vontade, desde que não gere nenhuma colisão com os obstáculos. Outro ponto importante para o Piano Mover's Problem é que toda mudança da configuração do piano deve ser contínua (não podemos simplesmente teletransportar o piano para a configuração final). Podemos pensar em duas perguntas relacionadas com esse problema: é possível chegar na configuração final? E qual sequência de configurações representa um caminho viável até a configuração final? Na Figura 3, podemos observar uma simplificação de uma sequência de configurações que consegue chegar na configuração final sem colidir com nenhum obstáculo. Já foi demonstrado por [Reif 1979] que o Piano Mover's Problem é PSPACE-hard e se $P \neq NP$, sabemos que o Piano Mover's Problem não pode ser resolvido em tempo polinomial.

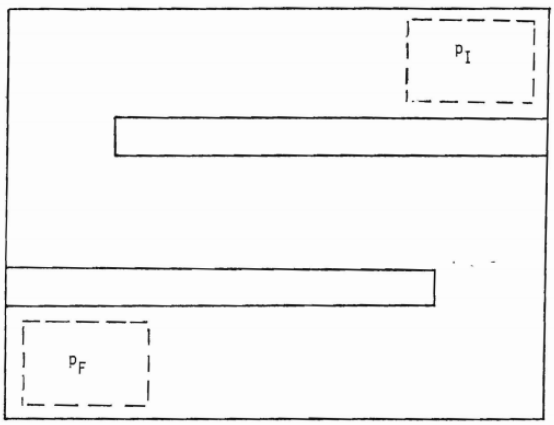


Figura 2. Piano Mover's Problem. Fonte: [Reif 1979].

Antes de trazermos uma formalização do Piano Mover's Problem, há alguns conceitos importantes que precisam ficar bem esclarecidos. Começando por um pequeno detalhe. Chamaremos de robô o objeto para qual o planejamento está sendo feito, no

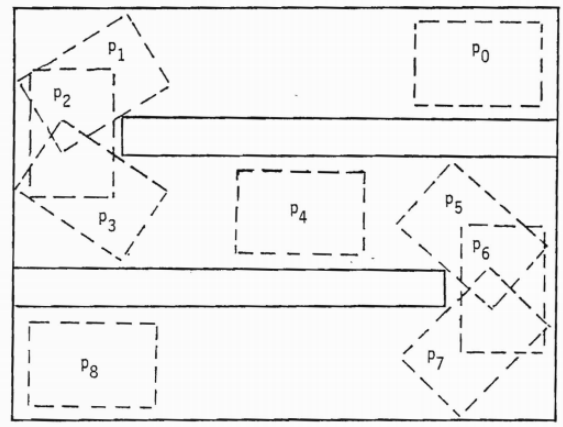


Figura 3. Piano Mover's Problem - solução. Fonte: [Reif 1979].

exemplo do piano o piano seria o robô. Próximo conceito: espaço de configurações (configuration space ou C-space). O configuration space representa o espaço de todas as configurações possíveis que o robô pode ter. Por exemplo: se só nos preocupamos com a posição (x, y) do robô e limitarmos o posicionamento do robô para o intervalo [0,1] para x e y; então o configuration space do robô são todas as posições com x entre 0 e 1 e y entre 0 e 1. O configuration space possui duas partições: uma região livre e uma região ocupada. O que determina se uma configuração é livre ou ocupada depende de algumas características do robô, como o seu tamanho por exemplo, e dos obstáculos do ambiente. Por exemplo, se imaginarmos que o P5 da Figura 3 está um pouco mais para baixo e admitindo que posição da configuração P5 fica no centro do robô; sabemos que essa configuração não será uma configuração válida para o robô, apesar da posição da configuração do robô não estar em um local ocupado por um obstáculo. É muito importante essa noção de que o configuration space é diferente do ambiente em que o robô tá localizado. No exemplo do piano, o configuration space leva em conta a orientação e por isso ele possui três dimensões e para saber se uma configuração é válida ou não precisamos levar em conta cada uma dessas dimensões, inclusive a orientação; não basta apenas olhar para posição. Assim, não é tão simples saber se uma configuração é válida ou não, ainda mais para configuration spaces mais complexos. E é mais complexo ainda modelar explicitamente o configuration space.

A seguir está uma formalização de [LaValle 2006] simplificada do Piano Mover's Problem. Ressaltamos que não há nada de muito especial na formalização. O único ponto que vale a pena chamar atenção é que não há um limite para dimensão para o configuration space e dimensões maiores do configuration space tendem a tornar o problema cada vez mais difícil de se resolver.

1. Há um mundo W em que o robô e os obstáculos estão, sendo $W = R^2$ ou $W = R^3$.
2. Os obstáculos são definidos em W.
3. Há um robô definido em W.
4. Há um espaço de configuração C especificado a partir de todas as transformações que podem ser aplicadas no robô. A partir de C, derivamos C_{obs} e C_{free} - espaço com colisão e livre de colisão, respectivamente.
5. Há uma configuração inicial dada por $q_0 \in C_{free}$.

6. Há uma configuração final dada por $q_f \in C_{free}$.
7. Caso haja um caminho contínuo $\tau : [0, 1] \rightarrow C_{free}$ tal que $\tau(0) = q_0$ e $\tau(1) = q_f$, este deve ser apresentado. Caso contrário, deve-se avisar que não há um caminho.

2.2.2. Motion Planning Under Differential Constraints Problem

O Problema de Planejamento de Movimentação com Restrições Diferenciais ou Motion Planning Under Differential Constraints Problem é de forma bem resumida o Piano Mover's Problem com restrições diferenciais. A seguir, vamos trazer alguns conceitos que auxiliam a entender soluções para esse problema. Um desses conceitos é o State Space. Vamos imaginar que estamos fazendo o planejamento para um teste de frenagem de um veículo. Queremos que o veículo acelere o máximo possível em direção a uma parede e ele deve frear antes para que não bata. A informação da velocidade do veículo é extremamente importante para saber quando frear. Essa informação pode ser inserida no configuration space desse problema e com isso o configuration space passa a se chamar state space. A ideia é que estamos representando nesse espaço a velocidade que é algo a mais do que todas as configurações possíveis do veículo - que no exemplo poderiam ser a posição e orientação dele. Assim, as configurações passam a ser chamadas de estados. E o planejamento passa a ser feito no state space que possui, nesse exemplo a posição, orientação e velocidade do veículo. Só um pequeno detalhe, todo configuration space é um state space mas nem todo space state é um configuration space.

Outro conceito importante é o conceito de ação. A ideia é que em todo estado uma ação pode ser tomada que faz com que o robô passe do estado atual para um novo estado. No exemplo anterior, a intensidade com que o veículo acelera seria uma ação.

Por fim, uma observação ligada a trajetórias geradas por um planejamento de movimentação com restrições diferenciais. Essas trajetórias podem ser dadas como um estado inicial e uma sequência de ações que levam do estado inicial até o estado final, também chamado de goal state.

2.2.3. Boundary Value Problem

De acordo com [Gladwell 2008], podemos definir o Boundary Value Problem como um sistema de equações diferenciais ordinárias com valores especificados para solução e para as derivadas em mais de um ponto. Já o Two-point Boundary Value Problem possui exatamente dois pontos especificados para solução e para as derivadas. Podemos pensar no MPUDCP, como um BVP com obstáculos, já que os dois pontos do BVP podem ser pensados como os pontos de origem e destino no configuration space. Um ponto negativo de pensar no MPUDCP como uma extensão do BVP é que, de acordo com [LaValle 2006], em geral, os métodos para resolver BVPs não podem ser bem adaptados para lidar com obstáculos.

3. Crawler

Crawler é o nome do veículo autônomo que estamos desenvolvendo - diversas pessoas já contribuíram e contribuem para esse projeto. Na Figura 4 podemos observar o Cra-

wler. Ele tem cerca de 60 cm de comprimento; 47,5 cm de largura; 25 cm de altura; considerando a câmera e o suporte, passa para 135 cm de altura.

Nessa parte da apresentação sobre o Crawler, abordaremo alguns detalhes do Operational Design Domain (ODD) do Crawler que é basicamente para que ambiente e situação o Crawler foi projetado. Depois, apresentaremos os dispositivos (sensores, atuadores e computadores) presentes no Crawler e o que eles usam para se comunicar. Por fim, apresentaremos o SNA do Crawler e alguns componentes de forma mais detalhada. Não abordaremos nenhum detalhe da mecânica nem da eletrônica do Crawler, como alimentação, drivers PWM , PCB utilizada, restrições de tensão e corrente, dentre várias outras questões afim.

3.1. ODD

O Crawler foi projetado para navegar em estradas de terra, pavimentadas e asfaltadas que podem ter buracos ou outros obstáculos como carros estacionados, por exemplo. Sempre que possível o Crawler deve contornar esses obstáculos. Não pode existir qualquer componente dinâmico na estrada, como outros veículos ou pedestres. O Crawler deve a partir de um ponto inicial chegar ao seu destino, caso seja possível. O ponto inicial, o destino e a rota do ponto inicial ao destino são definidos por alguma pessoa, sendo que o Crawler deve começar, claramente, no ponto inicial. Se existir qualquer bifurcação no caminho, a rota deve indicar por onde o Crawler deve seguir. O Crawler deve percorrer o seu caminho sem nenhuma intervenção humana. As regras de trânsito como parar no sinal vermelho; seguir o sentido das faixas de trânsito; seguir indicações de velocidade são completamente ignoradas pelo Crawler. O Crawler não possui nenhum mapa construído previamente.

A ausência de componentes dinâmicos na estrada e esse descaso com as regras de trânsito são dois fatores que ajudaram a simplificar bastante o SNA do Crawler. O Módulo de Predição de Tráfego não é mais necessário. Além disso, o Módulo de Decisão de Comportamento não depende das regras de trânsito para escolher o goal state para o Módulo de Planejamento de Movimentação. Por isso não existe a necessidade de identificar sinalizações de trânsito e nem de conhecer as regras que existem onde o Crawler vai navegar.

3.2. Hardware e Software

A Figura 5 mostra os dispositivos presentes no Crawler. O Crawler possui 4 sensores: câmera monocular; GPS; IMU e um Odômetro que é um Wheel Encoder. O Crawler tem 4 atuadores: dois motores e dois servos que direcionam as rodas. E o Crawler tem 3 "computadores": Uma Jetson Nano; uma Raspberry Pi 3 e um Arduino.

A Figura 6 mostra a organização do SNA do Crawler. A segmentação semântica e o mapeamento estão presentes na Nano. A percepção de pistas de trânsito, o módulo de Decisão de Comportamento e o de Planejamento de Movimentação estão na Raspberry como um único programa - esses três componentes representam boa parte deste trabalho e mais à frente discutiremos vários detalhes desses componentes. Voltando para imagem, o módulo de Controle e de Estimação de Estados do veículo e as interfaces com os sensores também estão na Raspberry. Por fim, no Arduino está apenas a interface com os atuadores.

Figura 4. Fotografias do Crawler.

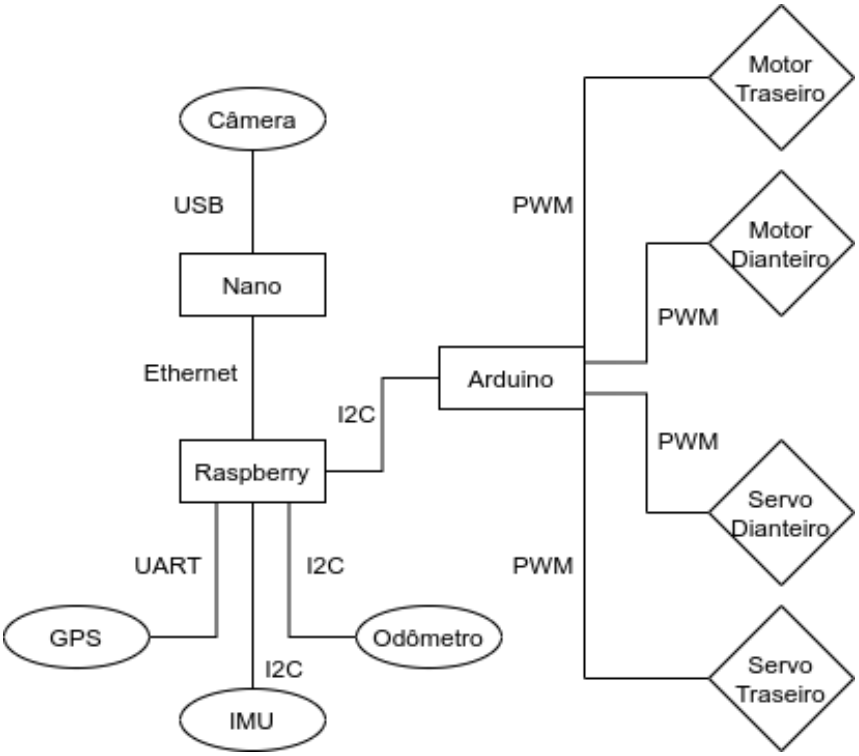


Figura 5. Dispositivos presentes no Crawler e suas conexões.

O controle e a estimação de estados que está acoplada ao controle estão sendo desenvolvidos pelo Matheus Wagner - um integrante do projeto. Em relação às interfaces com os sensores, estamos usando ROS2 para enviar os dados para os outros componentes. O Arduino apenas repassa aos atuadores os comandos que recebe do controle. Há algumas questões de segurança também, caso algum componente perca a comunicação com outro, mas não entraremos em detalhes sobre isso.

O roteamento é feito manualmente por meio da escolha de coordenadas geográficas que representam checkpoints. Esses checkpoints devem ser escolhidos de forma que indiquem sempre um caminho para o Crawler. Por exemplo: se houver uma bifurcação ou um entroncamento logo à frente do Crawler, deve existir um checkpoint que permita o Crawler saber por qual pista ele deve seguir; se houver uma interseção em T, também deve existir um checkpoint indicando para qual lado o Crawler deve seguir.

3.2.1. Segmentação Semântica e Mapeamento

A rede neural convolucional usada para realizar a segmentação semântica é uma HR-NetV2 [Wang et al. 2019]. O treinamento da rede foi feito com o método fit_one_cycle da biblioteca FastAI. O dataset utilizado foi o RTK Dataset [Rateke and von Wangenheim 2021]. Nesse dataset a segmentação semântica possui 12 classes; estrada asfaltada, pavimentada e de terra, falhas na pista e poça d'água são algumas dessas classes.

Para integrar a captura das imagens com o processamento feito pela HRNet, utilizamos uma biblioteca da Nvidia chamada jetson-inference. Essa biblioteca faz com que a

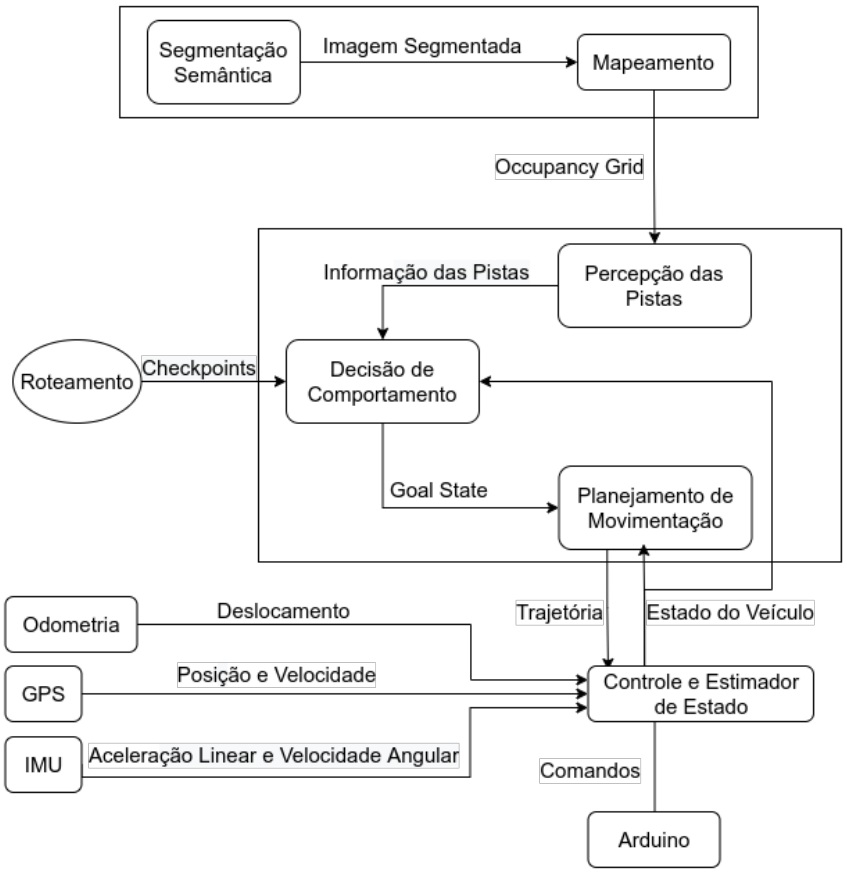


Figura 6. Fluxo de informação e componentes do SNA do Crawler.

rede neural seja processada pela GPU da Nano e assim acelera o processamento das imagens e faz com que cada imagem leve entre 95 e 160 milissegundos para ser processada pela HRNet. Mas, infelizmente, a geração da imagem segmentada é feita fora da GPU e isso causa um acréscimo de aproximadamente 200 milissegundos. Em cada iteração de processamento da imagem capturada pela câmera fazemos mais do que só obter a imagem segmentada. Salvamos a imagem original e a segmentada; geramos, enviamos e salvamos o occupancy grid; além de mais alguns pequenos detalhes. Em geral, essas operações a mais demandam por volta de 100 milissegundos. Dessa forma, o total de tempo por iteração fica próximo de 450 milissegundos. Devemos notar que esses valores são para imagens RGB 800x600.

O occupancy grid representa uma visão de cima da pista e perpendicular ao plano da pista. A área que decidimos representar no occupancy grid é dada por um retângulo com 10 metros de comprimento, 8 metros de largura, alinhado com o yaw do Crawler e inicia a 1 metro a partir da frente do Crawler. Cada célula do OG tem 5 cm de comprimento e largura no mundo real. Portanto o OG é uma matriz 200x160. Para definir cada célula do OG como navegável ou não navegável obtemos o centro de cada célula do OG em coordenadas do mundo real. Depois, projetamos cada um desses centros na imagem segmentada e obtemos coordenadas em pixels. Se o pixel está fora da imagem ou se a classe do pixel na imagem segmentada representa uma área não navegável, a respectiva célula do OG é considerada não navegável; caso contrário, a célula do OG é considerada

navegável. Para fazer a projeção inversa dos pontos, utilizamos o modelo de câmera com distorção de Brown-Conrady. Depois de fazer esse processo com todos os pontos do OG, o OG está pronto e já podemos enviar o OG por TCP via Ethernet para Raspberry para que a percepção das pistas seja feita.

Na Figura 7 podemos ver alguns exemplos da segmentação semântica e do occupancy grid. Podemos perceber que a segmentação semântica está longe de ser perfeita, as partes mais distantes e muito próximas da via são classificadas como background; às vezes buracos/falhas são adicionados em lugares que não existem; muitas vezes tem uma confusão entre via asfaltada com via pavimentada e as sombras tão prejudicando bastante a qualidade da segmentação semântica. Existe realmente muito a melhorar na segmentação semântica, mas neste trabalho nosso foco é a partir da segmentação semântica.

4. Percepção de Pistas e Definição do Goal State

Antes de começarmos a falar da definição do goal state é bom relembrar rapidamente o que é o goal state. O goal state é o estado que se quer alcançar para cada planejamento realizado; esse objetivo é fornecido pelo módulo de Decisão de Comportamento. Toda essa parte da apresentação é voltada para o problema de como definir o goal state para o planejamento de movimentação do Crawler - o que é essencialmente o módulo de Decisão de Comportamento do Crawler. A primeira observação que precisamos fazer é por que não foi usado algo já pronto? O Operational Design Domain do Crawler tem algumas especificidades que inviabilizam o uso de várias dessas soluções já prontas. Por exemplo, o Crawler deve navegar em áreas não urbanas e sem demarcações na via; o Crawler não tem nenhum mapa previamente construído do ambiente e a única fonte de informação do ambiente é uma câmera monocular.

4.1. Problema

A seguir explicaremos alguns detalhes do problema da definição do goal state. Precisamos encontrar um estado que serve como goal state para o planejamento de movimentação. Esse goal state precisa pertencer ao free state space e precisa ser alcançável pelo Crawler. Além disso, o goal state precisa permitir que o Crawler siga pela rota definida pelo módulo de roteamento.

As principais informações que temos para resolver o problema são o occupancy grid e a sequência de coordenadas do roteamento - os checkpoints. A primeira ideia que pode vir à mente é por que não usar os checkpoints como goal states? Primeiramente, não temos nenhuma garantia que o checkpoint pertencerá ao free state space. Além disso, não temos nenhuma garantia que o checkpoint estará localizado dentro do occupancy grid.

Outro problema relacionado com a definição do goal state é os erros de estimação de estado do Crawler. Vamos imaginar que o Crawler está em uma bifurcação, com um caminho para direita e outro para esquerda. O Crawler precisa seguir pelo caminho da direita e ele pensa que está na frente da entrada desse caminho. Mas ele está, na verdade, na frente da entrada do caminho da esquerda. Nesse caso o Crawler seguiria pelo caminho errado e a navegação inteira ficaria comprometida. Com isso algumas pessoas podem se perguntar como melhorar a estimação de estados do Crawler. Mas isso é um problema do módulo de Localização/Estimação de estados - isso não é responsabilidade do módulo de Decisão de Comportamento. A real preocupação do módulo de Decisão

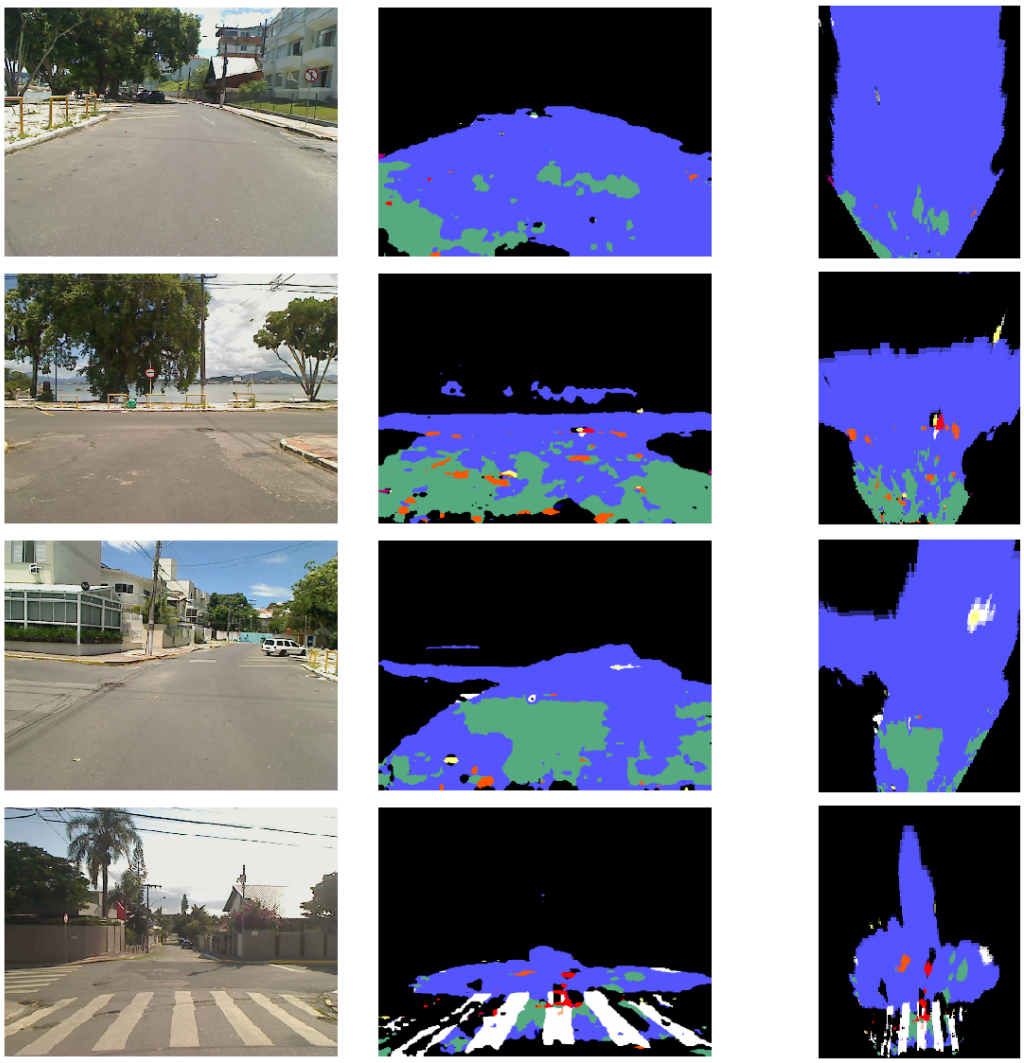


Figura 7. Exemplos da segmentação semântica e geração do occupancy grid. Na esquerda estão as imagens originais. No centro, estão as imagens segmentadas (máscaras). Na direita, estão os OGs. A cor mais para um azul/roxo representa uma via asfaltada; o verde representa uma via pavimentada não asfaltada; o branco representa sinalizações na horizontais na via; o vermelho representa buracos/falhas na via e o preto representa qualquer coisa que não é de interesse (background). Há ao todo 12 classes, porém as mencionadas são as que mais aparecem nas imagens de exemplo.

de Comportamento é como escolher um goal state que guie o Crawler para o caminho correto, sabendo que o checkpoint provavelmente está em determinada região?

Em suma, há duas questões principais relacionadas com a definição do goal state: como escolher um goal state alcançável e pertencente ao free state space para o planejamento de movimentação? E como escolher um goal state que guie o Crawler para o caminho correto admitindo possíveis erros de posicionamento do checkpoint em relação ao Crawler?

4.2. Solução

A ideia de solução que seguimos foi a seguinte: passamos a diferenciar pistas de trânsito e os checkpoints passam a indicar apenas por qual pista o Crawler deve seguir - assim, só precisamos de checkpoints em bifurcações e entroncamentos. Por exemplo, partes de vias que possuem apenas um caminho, seguindo reto ou fazendo curvas, não necessitam de checkpoints. Já para um cruzamento seria necessário um checkpoint indicando para qual pista o Crawler deve seguir.

Mas como podemos perceber e diferenciar as pistas de trânsito? Novamente, fizemos várias buscas mas não encontramos nada que se encaixasse no Crawler. Por isso, decidimos voltar nossa atenção a técnicas de visão computacional. Sendo que focamos em uma técnica chamada de esqueletonização.

4.2.1. Esqueletonização

O que é a esqueletonização? Podemos pensar na esqueletonização como uma técnica de afinamento das bordas, de forma que ao fim do processo tenhamos uma representação simplificada (um esqueleto) do que foi esqueletonizado. Testamos três técnicas de esqueletonização. Chamamos as técnicas de ZS, GH e MA que são referentes, respectivamente, às técnicas apresentadas em [Zhang and Suen 1984] e [Guo and Hall 1989] (a A1 neste caso) e a uma técnica sem artigo chamada de medial axis pela sua biblioteca. Testou as implementações desses três métodos da biblioteca scikit-image. Na Figura 8 há um de exemplo da esqueletonização de um cavalo.

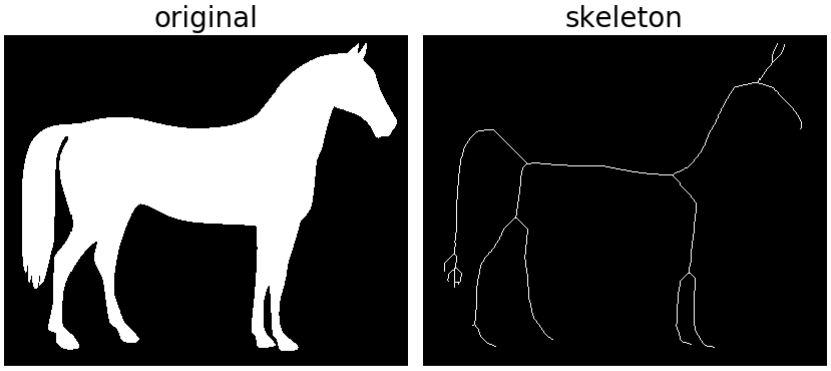


Figura 8. Exemplo de esqueletonização. Fonte: scikit-image.

Agora, vamos contextualizar o processo de esqueletonização no SNA do Crawler para depois entrar em mais detalhes das técnicas de esqueletonização. A única fonte de informação que o Crawler tem do ambiente é a câmera monocular. Ele tira uma foto do ambiente, faz a segmentação semântica e gera o occupancy grid. Uma observação é que o occupancy grid possui várias falhas da segmentação semântica que atrapalham a esqueletonização. Por isso, antes de aplicar a esqueletonização, fazemos um pré-processamento com operações de abertura e fechamento da morfologia matemática. Essas operações conseguem melhorar bastante a qualidade do occupancy grid para ser usado na

esqueletonização. Na Figura 9 temos um exemplo das consequências de aplicar essas operações em occupancy grids. Vale ressaltar que a esqueletonização no Crawler é usada para gerar um esqueleto da via, dessa forma identificando diferentes pistas se existirem. E o esqueleto é feito da área navegável do occupancy grid.

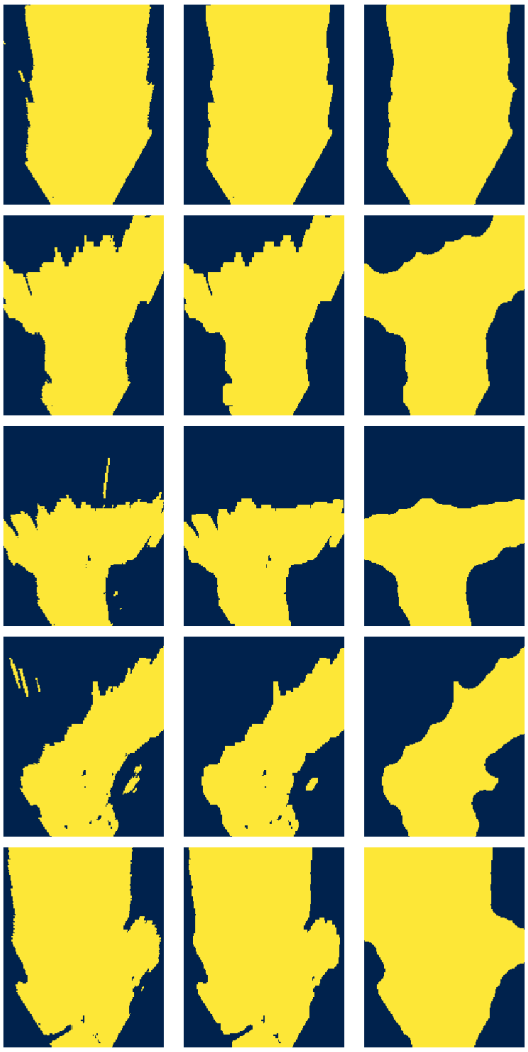


Figura 9. Exemplos da aplicação das operações de morfologia matemática de abertura e fechamento para occupancy grids. Na primeira coluna temos o OG sem modificação. Na segundo coluna podemos ver o OG depois de aplicarmos a operação de abertura. Na terceira coluna podemos ver o resultado de aplicarmos a operação de fechamento no OG da segunda coluna.

Voltando para as técnicas de esqueletonização. Os métodos ZS e GH são métodos bem similares. A ideia dos dois é a seguinte: os métodos iteram até o esqueleto estar pronto, inicialmente o esqueleto é a área navegável. Em cada uma dessas iterações têm duas sub-iterações. Em cada uma dessas sub iterações testa-se algumas condições de pixels que estão livres, se as condições forem respeitadas o pixel é marcado para ser eliminado do esqueleto. Os pixels marcados são eliminados apenas ao fim de uma iteração. Esses métodos são extremamente paralelizáveis. Já o método MA é chamado de método serial. No MA é feita apenas uma passada pelos pixels, mas os testes feitos para ver se

um pixel é eliminado ou não dependem do resultado dos pixels analisados anteriormente - o que torna o método serial.

Para decidir qual método utilizar, realizamos alguns experimentos. Consideramos principalmente o tempo de execução e a representatividade da via nos experimentos. Nas Figuras 10 e 11 estão alguns esqueletos gerados por cada um dos métodos.

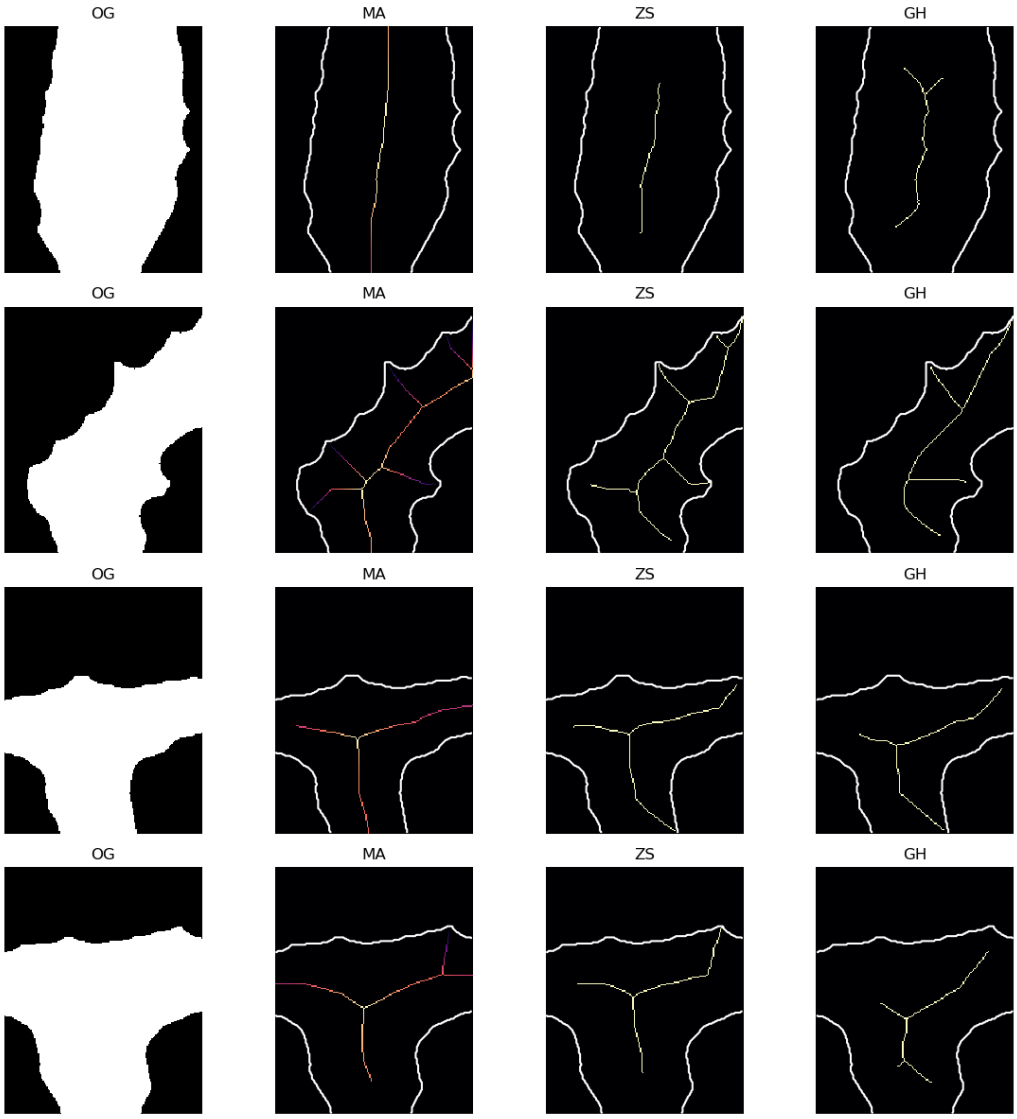


Figura 10. Comparação entre os três métodos de esqueletonização abordados. As duas primeiras linhas são vias simples e as duas últimas são vias com cruzamento em T. Na primeira coluna apresentamos o OG pós operações de morfologia matemática e nas outras três apresentamos o resultado da aplicação de cada método. Decidimos adicionar adicionar as bordas da área navegável apenas para que pudéssemos ter uma referência melhor para analisarmos o esqueleto - as bordas brancas nas colunas dos métodos não fazem parte do esqueleto. Por fim, na coluna do MA, a cor de cada pixel do esqueleto é escolhida com base na distância das bordas - quão mais longe da borda mais claro o pixel.

Podemos notar que dos três métodos o MA foi o que gerou ramos que mais se

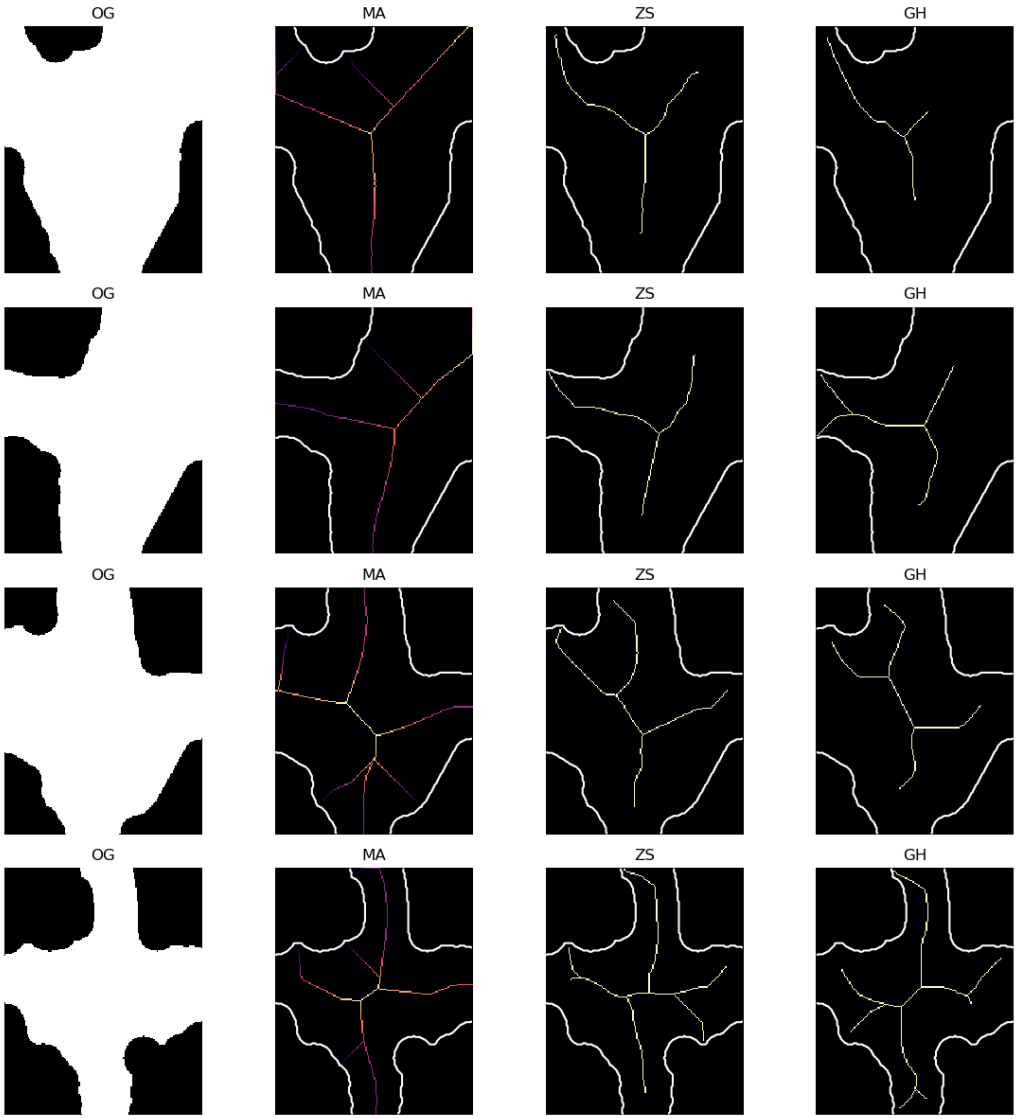


Figura 11. Comparação entre os três métodos de esqueletonização abordados. As duas primeiras linhas são vias com bifurcação e as duas últimas são vias com cruzamento em X.

aproximam dos centros das pistas. Mas o MA tende a gerar algumas ramificações a mais. Já o ZS e o GH apresentam resultados bem similares. Ambos às vezes geram ramos que se aproximam demais das bordas e às vezes alguns ramos não adentram muito uma pista. Em geral, a qualidade dos três métodos é bem próxima.

A principal conclusão que obtivemos com a análise de qualidade dos métodos é que os métodos de esqueletonização não são perfeitos, eles podem gerar ramificações próximas das bordas, ramificações que não adentram quase a sua respectiva pista e podem gerar ramificações espúrias.

Passando para análise do tempo de execução. As médias do tempo de execução na Raspberry foram: 0.02 segundos para o ZS; 1.12 segundos para o GH e 0.66 segundos para o MA. Enfim, decidimos usar o ZS como método de esqueletonização, já que a

qualidade dos métodos não variou muito e o tempo do ZS foi bem melhor, dentre mais alguns motivos.

4.3. Definição do Goal State

Assim, já conseguimos uma forma de diferenciar as pistas - a esqueletonização. Mas como que vamos definir o goal state a partir do esqueleto? O método desenvolvido depende da proximidade do checkpoint. Primeiro, se o checkpoint estiver próximo. A primeira coisa a se fazer é cortar os pontos próximos demais das bordas e identificar o ponto do esqueleto mais próximo do Crawler, chamamos esse ponto de ponto inicial. Depois, percorrer os pixels do esqueleto a partir do ponto inicial e, assim, obter caminhos no esqueleto. Em seguida, procurar pelo ponto mais próximo do checkpoint que pertence a um caminho e selecionar esse caminho. Depois, estender o caminho a partir do ponto próximo até o checkpoint ou até atingir uma distância mínima em relação às bordas. O último ponto dessa extensão é a posição do goal state. Por último, resta obter a orientação do goal state.

Para caso o Crawler esteja longe de um checkpoint. Não tem diferença alguma até obter os caminhos. Em seguida, deve-se selecionar um ponto do esqueleto para ser o goal state; essa escolha é feita com base na distância em relação ao Crawler e às bordas. Depois, deve-se selecionar o caminho do ponto selecionado. Por fim, basta obter a orientação da mesma forma como se houvesse checkpoint.

Na Figura 12 há quatro exemplos do passo a passo da execução do método de definição do goal state que acabamos de descrever. Na Figura 13 há mais alguns exemplos, porém apenas do resultado de todo o processo - não mais o passo a passo. Como podemos perceber, os resultados ficaram muito bons; até mesmo com checkpoints fora da área navegável e em casos em que a esqueletonização não representou muito bem as pistas.

Comentaremos sobre alguns benefícios que o processo como um todo traz para uma boa definição do goal state. Desconsiderar os pontos próximos demais da borda é importante porque algumas partes da borda podem ser falhas da segmentação semântica, assim, ajudamos a evitar que o Crawler saia da área navegável. A extensão do esqueleto ajuda nos casos em que o esqueleto não adentrou muito bem na pista que o Crawler deve seguir. A forma como é escolhido um pixel do esqueleto para ser o goal state quando não tem nenhum checkpoint próximo contribui bastante para manter o Crawler longe das bordas e vale notar que na maior parte do tempo o Crawler não está próximo de nenhum checkpoint. Por fim, a forma como a orientação é definida ajuda a evitar que pequenos desvios no esqueleto influenciem a orientação final.

Concluindo todo o processo de percepção das pistas de trânsito e definição do goal state. Inicialmente, tínhamos o problema de como definir o goal state para o planejamento de movimentação. O goal state precisa pertencer ao free state space, ser alcançável e precisa guiar o Crawler para sua rota pré definida - já discutimos alguns problemas relacionados com esses requisitos do goal state. Para resolver o problema da definição do goal state propomos uma solução que usa esqueletonização. Fizemos diversos testes para decidir qual técnica de esqueletonização usar. Além disso, desenvolvemos uma forma de definir o goal state a partir de um esqueleto.

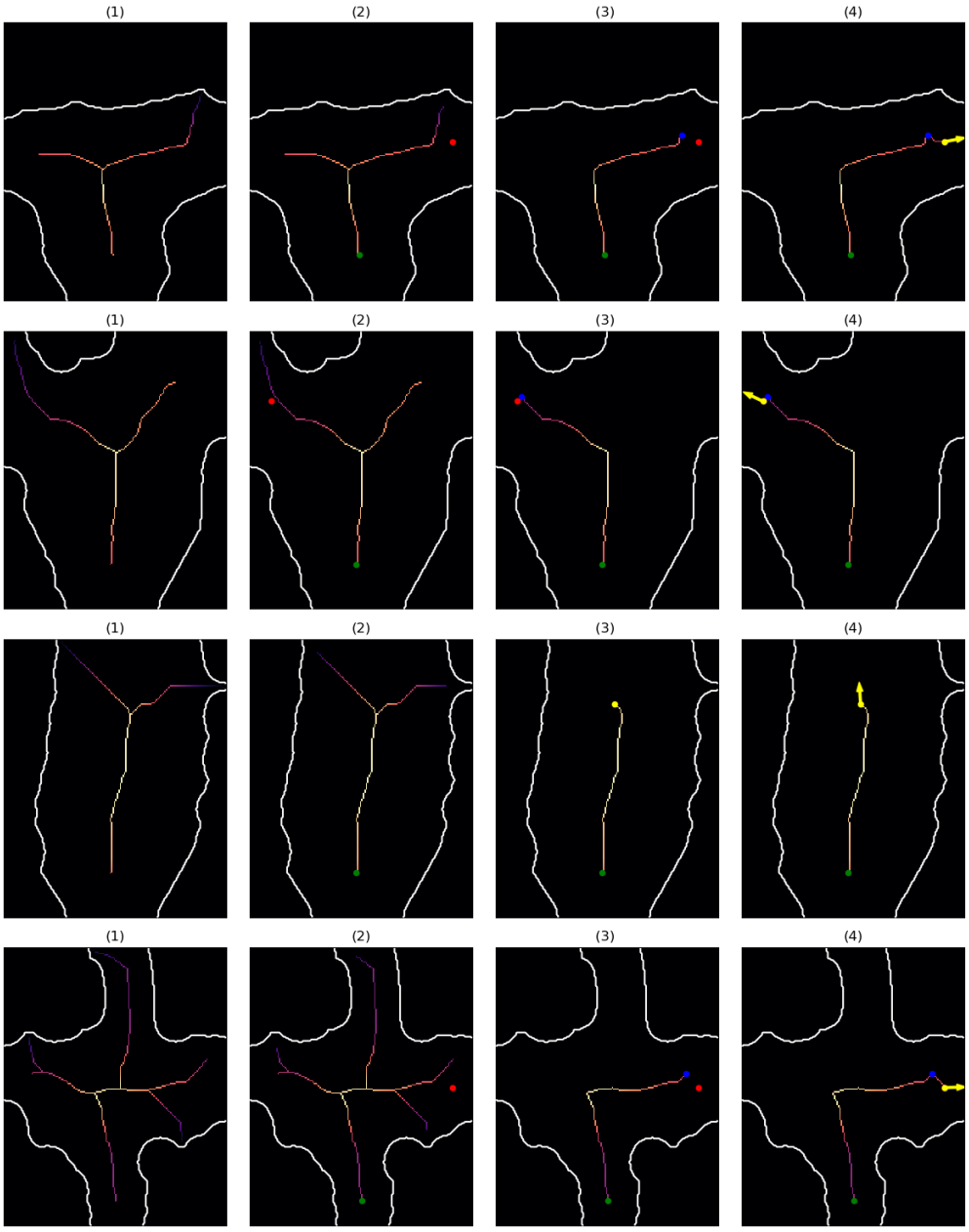


Figura 12. Separamos em quatro etapas a obtenção do goal state. Em (1), mostramos o OG pré-processado com o esqueleto. Em (2), mostramos o ponto mais próximo do Crawler em verde e o checkpoint, se existir, em vermelho. Em (3), apresentamos o caminho selecionado e o ponto pertencente ao esqueleto que nos fez escolhê-lo. Quando não temos checkpoint o ponto estará em amarelo (representando a posição do goal state) e quando temos checkpoint o ponto estará em azul. Em (4), quando temos checkpoint estendemos o caminho a partir do ponto azul e, assim, obtemos o ponto amarelo representando o goal state; se o ponto vermelho não estiver visível em (3) ou em (4) é porque o ponto azul ou o amarelo podem estar por cima. Além disso, em (4), mostramos a orientação do goal state. A terceira linha representa um caso em que não há checkpoint próximo; para as demais há.

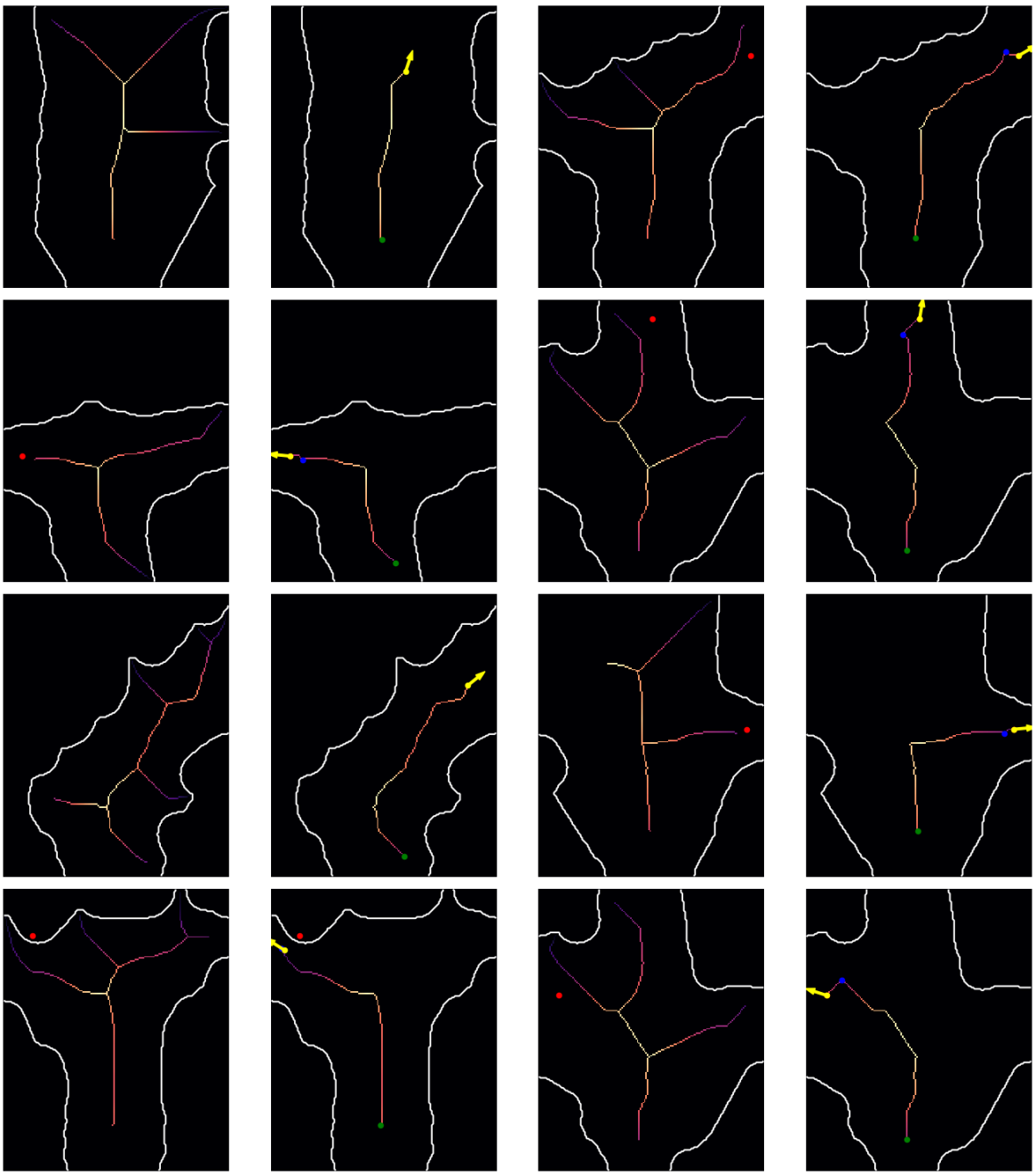


Figura 13. Exemplos de definição de goal state a partir do esqueleto.

5. Planejamento de Movimentação para o Crawler

Há alguns pontos que vale a pena revisitarmos. O papel do módulo de Planejamento de Movimentação é gerar uma trajetória livre de colisões e que possa ser realizada pelo veículo considerando suas limitações físicas. Conseguimos com o módulo de Percepção uma representação simplificada das pistas (o esqueleto) e com o módulo de Decisão de Comportamento conseguimos um objetivo final para o planejamento (o goal state) que pertence ao esqueleto.

Diferentemente do módulo de Decisão de Comportamento, o planejamento de

movimentação do Crawler está bem mais direcionado para usar uma solução praticamente já pronta. Isso deve-se principalmente pela forma como desenvolvemos a arquitetura do SNA do Crawler. Assim, decidimos usar para o planejamento de movimentação uma biblioteca chamada OMPL - Open Motion Planning Library [Şucan et al. 2012].

A OMPL consiste de métodos baseados em amostragem do state space, o que contribui para separação dos métodos de planejamento (os planners) e da checagem de colisão de um estado - como demonstrado na Figura 14. Essa validação ou checagem de colisão de um estado fica completamente a cargo do usuário da biblioteca e a representação do ambiente também por consequência. A OMPL divide os planners em dois grandes grupos chamados de planners geométricos e planners baseados em controle. Se o planejamento possui qualquer tipo de restrição diferencial, ou seja, é um problema de Motion Planner Under Differential Constraints, então os planners baseados em controle devem ser usados. Caso contrário, se é um problema similar ao Piano Mover's Problem, os planners geométricos podem ser usados.

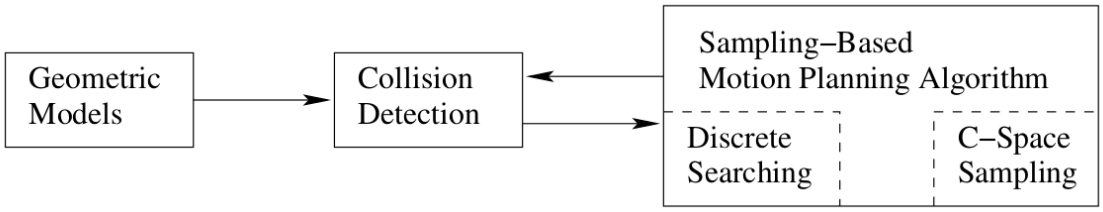


Figura 14. Organização de planners baseados em amostragem. Fonte: [LaValle 2006].

Agora discutiremos dois novos conceitos: funções de propagação e de steering. Alguns planners dependem dessas funções para realizar o planejamento. As funções de propagação estão relacionadas com a propagação de uma ação/controle u por um intervalo de tempo t a partir de um estado inicial x_0. Um exemplo de função de propagação é uma função que dá o estado final de um veículo, inicialmente em $x_0 = (0, 0)$, se for acelerado com uma intensidade $u = 10m/s^2$ em linha reta por $t = 1$ segundo. Dessa forma, a função de propagação pode ser pensada como uma forma de simular um movimento para tentar prever o estado final.

Já a função de steering, de acordo com [Li et al. 2014], é uma função capaz de, dados dois pontos, achar um caminho entre esses dois pontos na ausência de obstáculos. Se existir restrições diferenciais, a função de steering é equivalente a uma solução de um two point Bondary Value Problem. Mas, se não existir restrições, a função de steering pode ser simplesmente uma função que gera um segmento de reta que conecta os dois pontos. De acordo com [Li et al. 2014], para muitos sistemas dinâmicos não é fácil produzir uma função de steering. Às vezes podemos até ter funções de steering, mas elas podem ser muito caras computacionalmente. Por isso, em diversas ocasiões não teremos uma função de steering adequada para ser utilizada pelo planner, o que é uma desvantagem para os métodos que dependem de funções de steering para planejar com restrições diferenciais.

Voltando para OMPL, há um framework [Moll et al. 2015] que permite fazer benchmark de planners e diversas configurações para o planejamento de movimentação. Uti-

lizamos esse framework para selecionar bons parâmetros para cada planner e para obter diversos outros resultados, como por exemplo, tempo de execução e quantidade de planejamentos que conseguiram chegar ao goal state.

Para que pudéssemos escolher um planner adequado para o problema do planejamento de movimentação no contexto do Crawler, decidimos testar alguns planners da OMPL e dividimos os experimentos com os planners em duas etapas. Antes de falar de cada etapa, um ponto importante para comentar é que alguns planners têm vários parâmetros que influenciam extremamente o seu desempenho. Inclusive existem parâmetros que não estão relacionados diretamente com um planner específico e sim com o planejamento em geral. Esses parâmetros também podem mudar bastante o desempenho dos planners. Por isso, decidimos, como primeira parte dos experimentos, fazer uma busca por bons parâmetros para os planners e para o planejamento em geral. Escolhemos testar os planners RRT, RRT* e SST, sendo que consideramos duas implementações do RRT, o RRT geométrico e o RRT baseado em controle. Já na segunda etapa dos experimentos como já tínhamos vários indicativos de qual planner usar, foco foi mais voltado para otimizar o planner escolhido. Uma observação importante é que os experimentos da primeira etapa foram realizados em um notebook. Já, na segunda etapa, os experimentos foram feitos na Raspberry Pi.

No restante dessa parte da apresentação sobre o planejamento de movimentação no Crawler, abordaremos rapidamente como cada planner utilizado funciona. Em seguida, apresentaremos a primeira etapa dos experimentos com os planners. E então, discutiremos alguns resultados interessantes desses experimentos. E por fim, comentaremos um pouco sobre a segunda etapa dos experimentos.

5.1. Planners

As explicações que daremos agora sobre os funcionamento dos planner fornecem uma ideia geral sobre planners, existem vários detalhes que não abordaremos. Começando pelo RRT (Rapidly-Exploring Random Trees) geométrico [LaValle 1998]. A ideia desse planner é que ele vai iterar sobre 4 passos até atingir uma condição de parada - que pode ser um número de iterações, um tempo limite, até chegar em determinado local do state space, etc. Esse método inicializa um grafo contendo apenas um estado/vértice inicial. Em cada iteração (i) um estado aleatório do state space é amostrado. Em seguida, (ii) procura-se pelo estado pertencente ao grafo que está mais próximo do estado amostrado. Depois, (iii) uma função de steering é usada para conectar os dois estados. Por último, (iv) se a aresta for válida (sem colisão) o estado amostrado e a nova aresta são adicionados ao grafo.

Passamos para o RRT* [Karaman and Frazzoli 2011]. Esse é um método ótimo assintoticamente; isso significa que ele pode otimizar o custo do caminho até o goal state com o passar do tempo. O RRT* é bem similar ao RRT geométrico. Uma das diferenças é que considera-se um conjunto de estados próximos do estado amostrado e o estado com menor custo é selecionado. A outra diferença é que o método cogita refazer algumas ligações dos estados próximos se essas novas arestas gerarem um menor custo para os estados.

O RRT baseado em controle também é bem similar ao RRT geométrico. A diferença principal é que uma função de propagação é usada ao invés de uma função

de steering. Por isso, é preciso amostrar uma ação e um intervalo de tempo também.

Já o SST (Stable Sparse RRT) [Li et al. 2014] é o método que mais distoa dos demais. Esse planner é um planner quase ótimo assintoticamente; o que quer dizer que ele pode otimizar o custo das trajetórias até o goal state até um certo limite. No SST também é usada uma função de propagação, mas a escolha do estado inicial da propagação depende do custo dos estados próximos. Nesse método há momentos em que alguns cortes no grafo podem são feitos.

5.2. Experimentos

Agora já podemos analisar os resultados da primeira etapa dos experimentos que estão apresentados nas Figuras 15, 16, 17 e 18. Primeiramente, uma comparação da qualidade das trajetórias geradas. Os métodos baseados em controle apresentam uma qualidade das trajetórias geradas muito superior. Os métodos geométricos geram caminhos com curvas completamente bruscas, porque os diversos movimentos que compõem os caminhos são sempre gerados por uma interpolação no R2; muitas dessas curvas são bem difíceis de serem seguidas pelo veículo e algumas são até mesmo impossíveis (por o veículo não dar ré, por exemplo). Por causa da ausência de uma função de steering que represente minimamente o comportamento do veículo (não consideramos gerar segmentos de reta entre estados como uma função de steering aceitável para o Crawler), os métodos RRTg e RRT* são inerentemente métodos sem garantia alguma de uma boa qualidade para o caminho gerado. Além disso, a orientação do estado inicial e do goal state não influenciam em nada no caminho, o que piora ainda mais a qualidade.

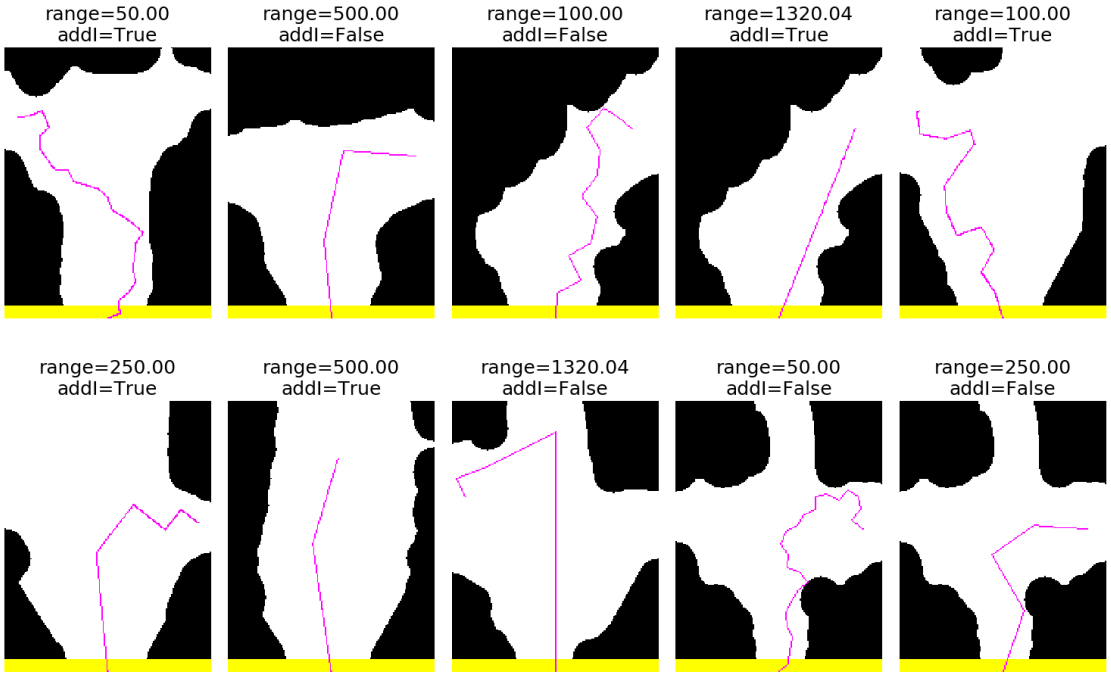


Figura 15. RRTg na primeira etapa dos experimentos.

Resumidamente, para o RRTc e SST, por terem funções de propagação que consideram algumas (apesar de simples) restrições física do veículo e por gerarem os seg-

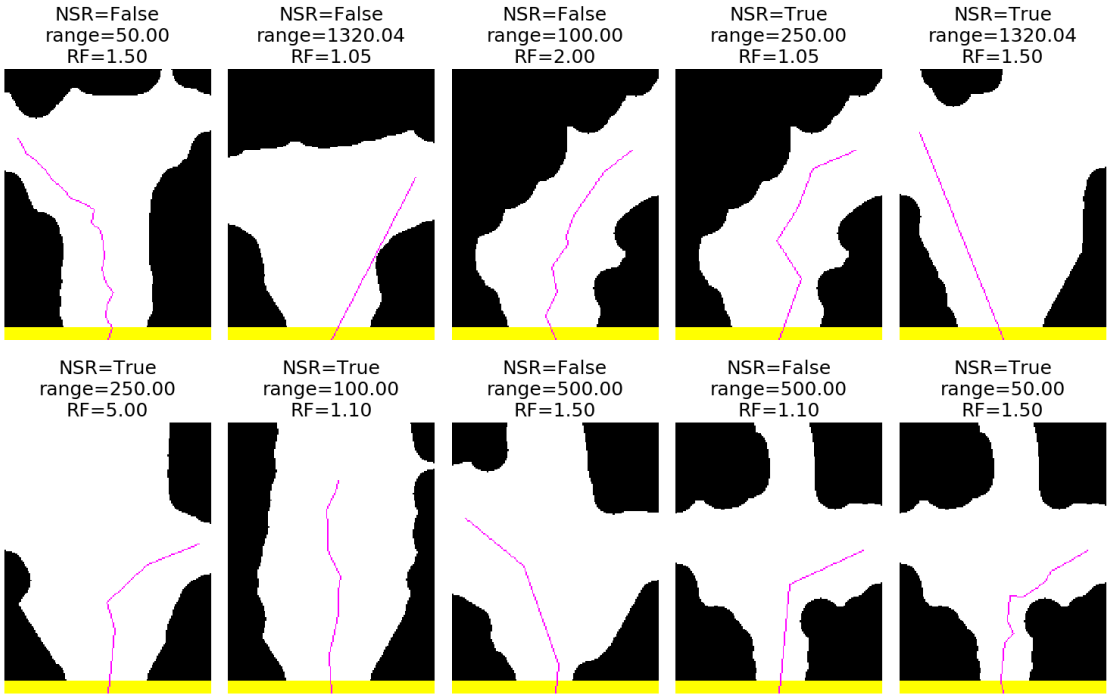


Figura 16. RRT* na primeira etapa dos experimentos.

mentos das trajetórias com base nessas funções de propagação, eles conseguem obter em geral uma qualidade bem superior de trajetórias. Com o SST, temos a possibilidade ainda de melhorar a qualidade da trajetória com o tempo por causa da sua capacidade de otimização.

Precisamos discutir ainda alguns detalhes da otimização no RRT* e no SST e algumas consequências dessa otimização. A princípio, ressaltamos que a otimização é inerente a esses dois planners, de forma que a construção do grafo já é feita pensando no custo de cada movimento. Comparando o RRT* ao RRTg e o SST ao RRTc, as suas otimizações fazem com que cada uma de suas iterações seja mais custosa computacionalmente. Mas, esse custo a mais, pode ajudar o planner a achar um caminho até goal state mais rapidamente. Assim que o RRT* e o SST acham um caminho até o goal state, eles podem continuar otimizando esse caminho com base na noção de custo utilizada. Podemos deixar esses dois planners otimizando o caminho até chegar em um custo mínimo ou podemos deixar eles otimizarem o máximo que conseguirem em certo intervalo de tempo. Uma última observação sobre a otimização desses dois planners é que o custo usado depende do tamanho da trajetória.

Agora, comentaremos um pouco sobre os tempos de cada um dos planners, especialmente com a configuração de parâmetros escolhida. O RRTg ficou com uma média de 0.0012 segundo e o RRT* ficou bem próximo com uma média de 0.0015 segundo. O RRTc e o SST ficaram com um tempo médio bem maior: 0.0772 e 0.075 segundo respectivamente. Esses tempos foram medidos com os métodos executando em um notebook e não na Raspberry Pi. Em geral, o RRTg e o RRT* necessitam de um número bem menor de iterações para chegar ao goal state - um dos vários motivos é a dimensão menor do

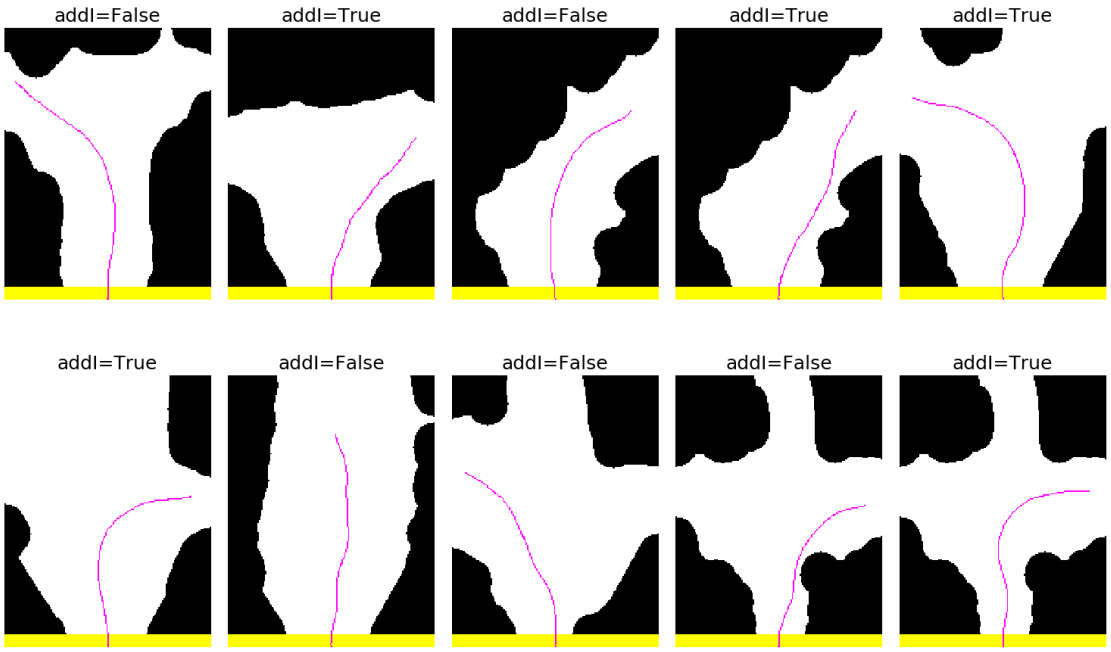


Figura 17. RRTc na primeira etapa dos experimentos.

espaço utilizado para esses métodos. Vale notar que existem diversos outros fatores que influenciam no tempo de execução.

Em suma, o RRTg e o RRT* geram, normalmente, caminhos deploráveis apesar de serem rápidos. Já as trajetórias do RRTc e SST são, em geral, boas, porém o tempo necessário para gerá-las é alto. Ademais, o RRT* e SST podem ser melhorados ainda mais se utilizarmos uma boa otimização. Por enquanto, o SST, claramente, se apresenta como a melhor opção para o Crawler. Entretanto, o tempo de execução do SST na Raspberry pode ser um entrave para a utilização desse planner. Assim, por todos esses motivos, focamos no SST durante a segunda etapa dos experimentos com o objetivo de descobrir seu desempenho na Raspberry e realizar otimizações caso fosse necessário. Agora, apresentaremos a segunda etapa dos experimentos.

Na segunda etapa, testamos cada um dos planners na Raspberry Pi e os resultados foram bem similares com exceção das questões relacionadas com o tempo de execução. Percebeu que todos os planners demoraram bem mais na Raspberry Pi, inclusive o SST. Se tivéssemos deixado as configurações do SST do jeito que estavam, o tempo de execução influenciaria muito na qualidade, dentro de um limite de tempo estipulado para o Crawler. Por isso, focamos em fazer uma seleção ainda melhor de parâmetros do planner e do planejamento em geral. Novamente, não entraremos em detalhes dos parâmetros testados. Apenas comentaremos um pouco a respeito dos resultados finais. Ao fim do testes, ficamos com três limites de tempo para o SST: 0.8; 1.0 e 1.2 segundo. Para o tempo de 0.8 segundo, 11% dos planejamentos não conseguiram chegar ao goal state. Para o tempo de 1.0 segundo, 8% dos planejamentos não foram completados e para o tempo de 1.2 segundo apenas 5%. É importante perceber que muitos desses planejamentos que não

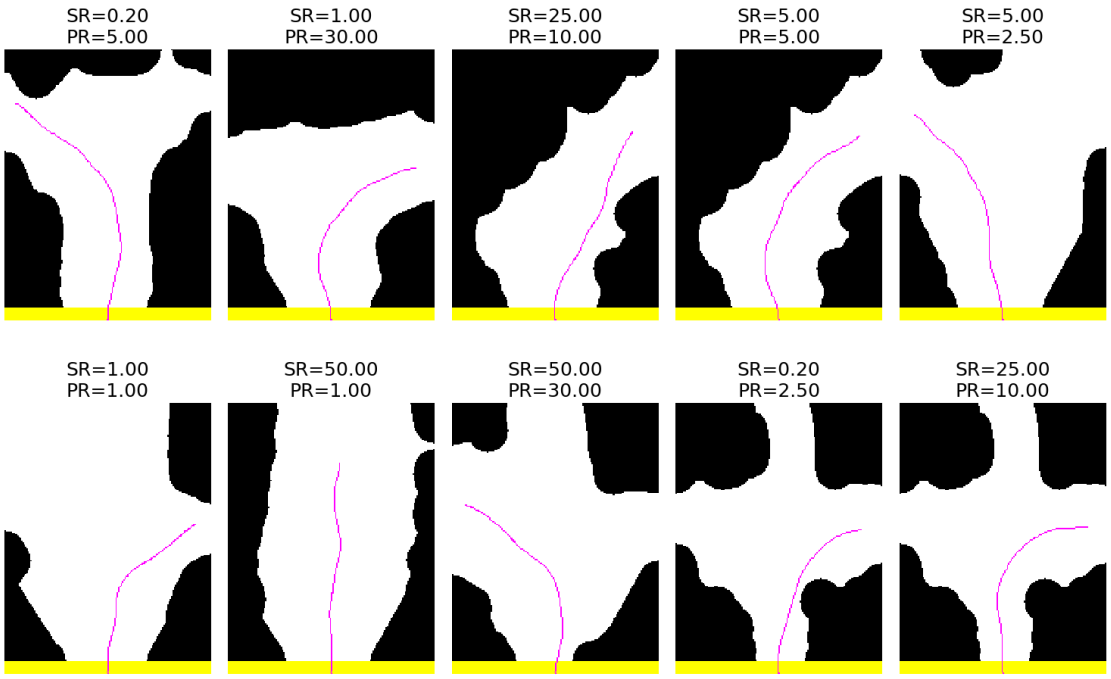


Figura 18. SST na primeira etapa dos experimentos.

foram concluídos, chegaram bem próximos do goal state; só uma ínfima minoria acabava se distanciando bastante ou apresentando uma orientação bem distinta da orientação do goal state. Acabamos escolhendo o tempo de 1 segundo de tempo limite. Na Figura 19 há alguns exemplos do planejamento final selecionado. Podemos perceber que apesar de às vezes ele não chegar muito próximo do goal state, a qualidade do caminho continua muito boa.

6. Discussões Finais

Há várias questões que não comentamos. Por exemplo, a parte da percepção de pistas de trânsito e definição do goal state foi implementada em Python3 e decidimos por motivos de eficiência, implementar tudo em C++; incluindo o método de esqueletonização. Outro ponto muito importante é a comunicação e a sincronização entre a Nano e a Raspberry. Além disso há os experimentos de todo o processo embarcado no Crawler, que de forma extremamente resumida se comportou da mesma forma como se estivéssemos executando os módulos separadamente. Mas acreditamos que nesse momento final, é realmente importante destacar tudo que foi feito para o Crawler como parte deste trabalho. Fizemos correções, modificações e melhorias na interface com o Arduino e com o wheel encoder. Desenvolvemos a interface com a IMU. Incluímos as medidas de segurança no SNA do Crawler - são medidas simples mas se elas existissem antes, alguns componentes do Crawler não teriam queimado em um acidente. Desenvolvemos o módulo de Percepção, incluindo todo o processamento feito na Nano e na Raspberry. Treinamos diversas HR-Nets, mas decidimos usar a rede treinada por outro integrante do projeto do Crawler. Desenvolvemos, também, os módulos de DC e de PM. Ressaltamos que com ”desenvolvemos”não falamos apenas fazer o código que está no Crawler, eu incluímos também

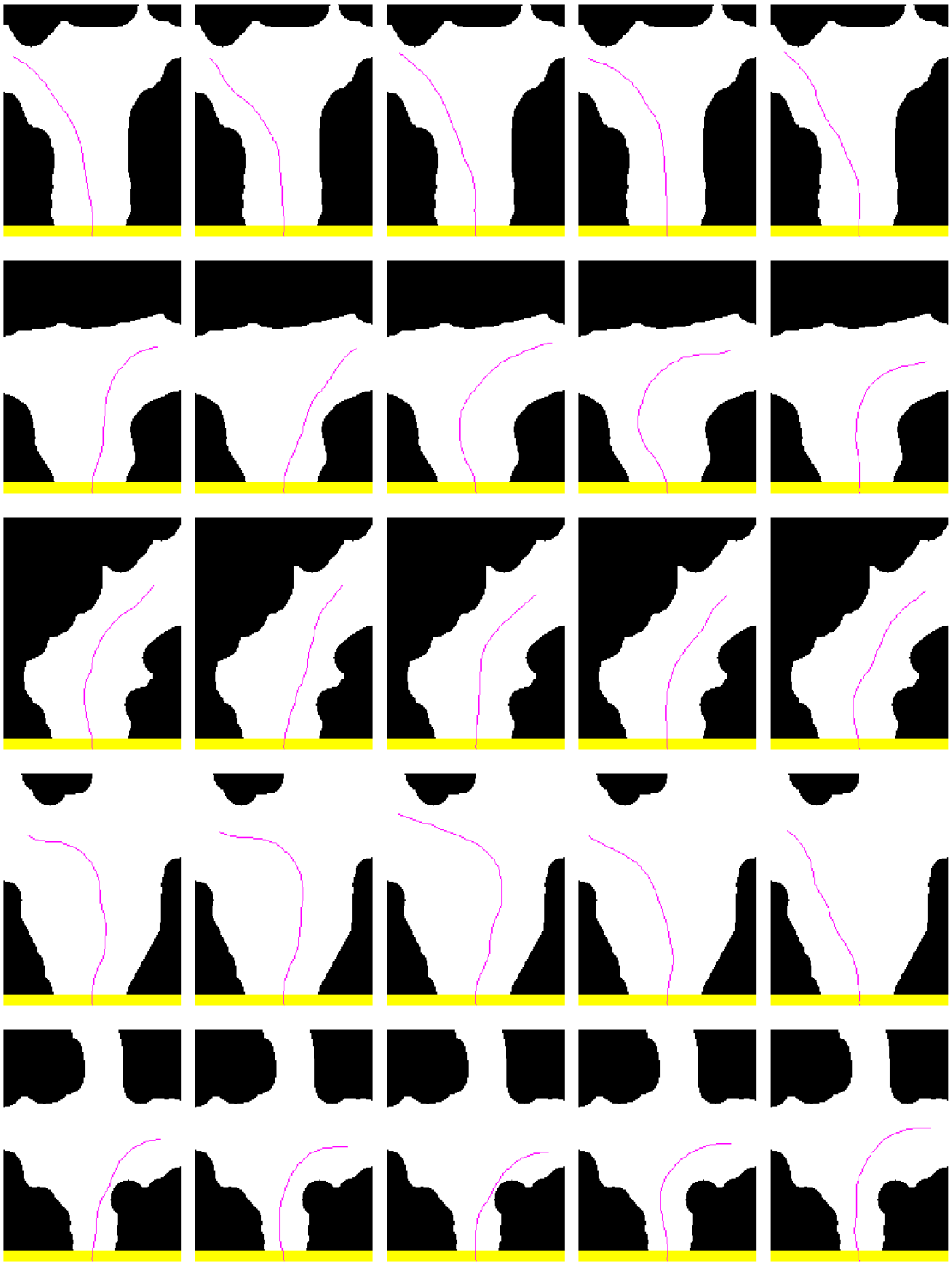


Figura 19. Exemplos de trajetórias geradas usando SST na Raspberry Pi com tempo limite de execução de 1.0 segundo, selectionRadius=25.0 e pruningRadius=5.0.

buscar garantias de o que foi feito realmente funciona, como realizar testes, desenvolver maneiras de visualizar o que foi feito, adquirir bases teóricas, dentre várias outras formas.

Além disso, escolhemos partes do ODD do Crawler e estruturamos todo o SNA. Sendo que neste artigo, focamos em apresentar o módulo de DC e o módulo de PM do Crawler.

Referências

Gladwell, I. (2008). Boundary value problem. *Scholarpedia*, 3(1):2853.

Guo, Z. and Hall, R. W. (1989). Parallel thinning with two-subiteration algorithms. *Communications of the ACM*, 32(3):359–373.

Karaman, S. and Frazzoli, E. (2011). Sampling-based algorithms for optimal motion planning.

LaValle, S. (1998). Rapidly-exploring random trees : a new tool for path planning. *The annual research report*.

LaValle, S. (2006). *Planning algorithms*. Cambridge University Press, Cambridge New York.

Li, Y., Littlefield, Z., and Bekris, K. E. (2014). Asymptotically optimal sampling-based kinodynamic planning.

Liu, S., Li, L., Tang, J., Wu, S., and Gaudiot, J.-L. (2017). Creating autonomous vehicle systems. *Synthesis Lectures on Computer Science*, 6(1):i–186.

Moll, M., Sucan, I. A., and Kavraki, L. E. (2015). Benchmarking motion planning algorithms: An extensible infrastructure for analysis and visualization. *IEEE Robotics & Automation Magazine*, 22(3):96–102.

Rateke, T. and von Wangenheim, A. (2021). Road surface detection and differentiation considering surface damages.

Reif, J. H. (1979). Complexity of the mover's problem and generalizations. In *20th Annual Symposium on Foundations of Computer Science (sfcs 1979)*. IEEE.

Şucan, I. A., Moll, M., and Kavraki, L. E. (2012). The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82. `https://ompl.kavrakilab.org`.

Wang, J., Sun, K., Cheng, T., Jiang, B., Deng, C., Zhao, Y., Liu, D., Mu, Y., Tan, M., Wang, X., Liu, W., and Xiao, B. (2019). Deep high-resolution representation learning for visual recognition.

Zhang, T. Y. and Suen, C. Y. (1984). A fast parallel algorithm for thinning digital patterns. *Communications of the ACM*, 27(3):236–239.