UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ANÁLISE DO DESEMPENHO DE TÉCNICAS DE OTIMIZAÇÃO APLICADAS AO ALGORITMO A* HÍBRIDO PARA BUSCA DE ROTAS DE VEÍCULOS

TRABALHO DE GRADUAÇÃO

Adrian Kaminski dos Santos

Santa Maria, RS, Brasil

2019

ANÁLISE DO DESEMPENHO DE TÉCNICAS DE OTIMIZAÇÃO APLICADAS AO ALGORITMO A* HÍBRIDO PARA BUSCA DE ROTAS DE VEÍCULOS

Adrian Kaminski dos Santos

Trabalho de Graduação apresentado ao Curso de Ciência da Computação da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para a obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Cesar Tadeu Pozzer

470

Santa Maria, RS, Brasil

2019

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Conclusão de Curso

ANÁLISE DO DESEMPENHO DE TÉCNICAS DE OTIMIZAÇÃO APLICADAS AO ALGORITMO A* HÍBRIDO PARA BUSCA DE ROTAS DE VEÍCULOS

elaborador por
Adrian Kaminski dos Santos

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISSÃO EXAMINADORA:

Cesar Tadeu Pozzer, Dr.
(Orientador)

Sérgio Luís Sardi Mergen, Dr. (UFSM)

Benhur de Oliveira Stein, Dr. (UFSM)

Santa Maria, 11 de Dezembro de 2019.

“We are old now, yet we were young this morning, when we carried our glass box through the streets of the City to the Home of the Scholars.”

— AYN RAND, ANTHEM, 1938

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

ANÁLISE DO DESEMPENHO DE TÉCNICAS DE OTIMIZAÇÃO APLICADAS AO ALGORITMO A* HÍBRIDO PARA BUSCA DE ROTAS DE VEÍCULOS
AUTOR: ADRIAN KAMINSKI DOS SANTOS
ORIENTADOR: CESAR TADEU POZZER
Local da Defesa e Data: Santa Maria, 11 de Dezembro de 2019

Este trabalho de graduação propõe o desenvolvimento de técnicas de otimização, aplicadas ao algoritmo A* Híbrido para busca de caminhos para veículos, baseado na implementação utilizada na navegação local de viaturas no sistema de simulação SIS-ASTROS. Durante o trabalho foi implementado, utilizando a linguagem C#, uma adaptação do algoritmo. Foram desenvolvidas duas técnicas: para otimização de cálculos de intersecção com polígonos (obstáculos para os veículos) foi implementada uma hash de aproximação de polígono, com tempo de avaliação de complexidade O(1), e para otimização da heurística do algoritmo foi utilizada o algoritmo de Dijkstra sobre um grafo de visibilidade. Ambas técnicas foram avaliadas em diferentes cenários de teste, e apresentaram desempenho superior aos algoritmos utilizados como comparação neste trabalho no que tange o tempo de execução de uma busca de caminhos.

Palavras-chave: A* Híbrido. Otimização. SIS-ASTROS. Hash. Polígono.

ABSTRACT

Undergraduate Final Work
Undergraduate Program in Computer Science
Federal University of Santa Maria

PERFORMANCE ANALYSIS OF OPTIMIZATION TECHNIQUES APPLIED TO THE HYBRID A* ALGORITHM FOR VEHICLE PATHFINDING

AUTHOR: ADRIAN KAMINSKI DOS SANTOS
ADVISOR: CESAR TADEU POZZER
Defense Place and Date: Santa Maria, December 11th, 2019

This undergraduate final work proposes the development of optimization techniques applied to the Hybrid A* pathfinding algorithm, based on the implementation currently present in the military simulation system SIS-ASTROS. An adaptation of that same algorithm was developed using C# language. Then, two different techniques were developed: a polygon approximation hash for optimization of intersection calculation between points and polygons, with O(1) complexity. It was also developed a technique for optimization of Euclidean distance heuristics using Dijkstra's algorithm over a visibility graph. Both techniques were evaluated in different test scenarios and presented superior performance when compared to other techniques used in the work, with respect to their execution time for pathfinding using the implemented Hybrid A*.

Keywords: Hybrid A*. Optimization. SIS-ASTROS. Hash. Polygon.

SUMÁRIO

1 INTRODUÇÃO

No campo de inteligência artificial, a navegação de veículos é um subconjunto de um problema maior, conhecido como problema do caminho mínimo, amplamente estudado e fruto de diversas abordagens para o mesmo problema (Algfoor et al., 2015), geralmente tendo como foco central a exploração de um espaço de busca estruturado, seja em uma grade regular, grafo, ou outras estruturas de dados. Algoritmos como Dijkstra e A* são, possivelmente, os mais conhecidos no campo de busca de caminhos.

Quase todos jogos digitais que envolvem agentes autônomos em um ambiente virtual precisam de alguma forma de algoritmo de busca de caminhos (Verth et al., 2000). O mesmo ocorre em ambientes virtuais de simulação que envolvem movimentação de entidades.

Quando estes agentes autônomos simulam veículos motorizados, como carros ou caminhões, se tornam necessários algoritmos de busca de caminhos especializados, que levam em consideração as restrições de movimento de tais veículos. Enquanto humanos podem girar sobre o próprio eixo para se deslocar na direção desejada veículos estão restritos a se deslocar sobre um círculo com raio mínimo, denominado raio mínimo de giro do veículo.

Atualmente se encontra em desenvolvimento o projeto de Sistema de Simulação SIS-ASTROS, fruto de cooperação entre a Universidade Federal de Santa Maria (UFSM) e o Exército Brasileiro (Dall'Agnol et al., 2015), que visa o adestramento de militares em instruções de emprego tático-operacional do sistema SIS-ASTROS. O sistema é composto por várias viaturas sobre rodas destinadas ao lançamento de mísseis e foguetes sobre alvos terrestres.

É parte da especificação deste sistema de simulação a navegação de viaturas em comboio dentro de um ambiente virtual 3D (Figura 1.1), construído a partir de dados geográficos de regiões brasileiras, incluindo mapa de elevação do terreno, dados de estradas, rios, lagos, e regiões de florestas (Nascimento et al., 2018).

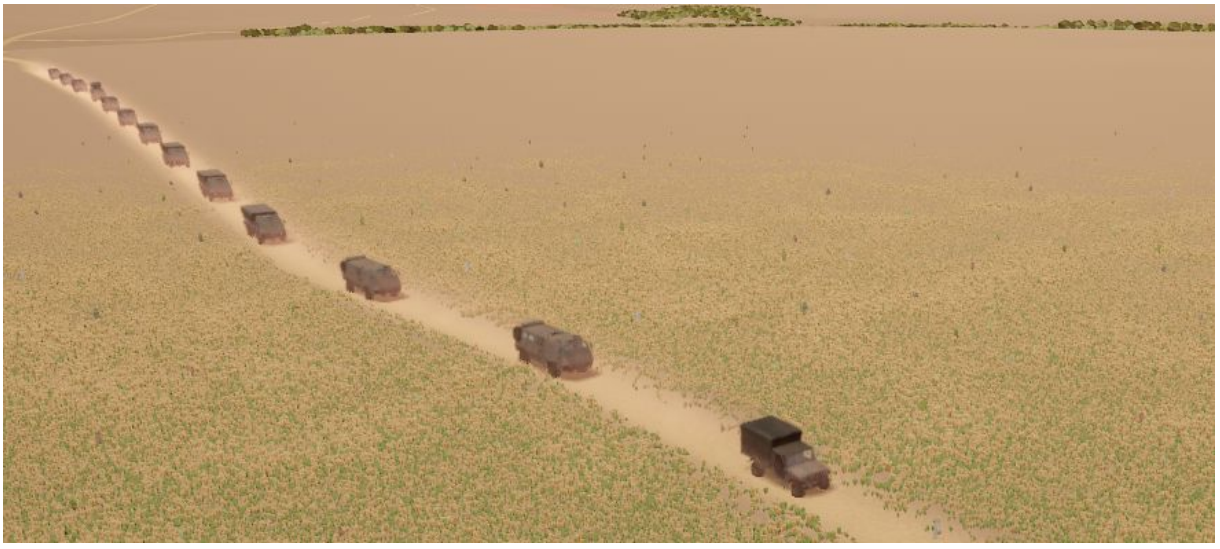


Figura 1.1 – Comboio de veículos executando navegação sobre estrada no ambiente virtual do sistema de simulação SIS-ASTROS.

No sistema de navegação em questão a busca de rotas para os veículos simulados se dá em duas partes: navegação global e local. A navegação global calcula as rotas principais, na casa dos quilômetros de distância, priorizando a navegação sobre estradas, e evitando apenas obstáculos de grande escala, como lagos e rios, enquanto a navegação local se encarrega de calcular rotas de curta distância, na casa das dezenas de metros, se mantendo fiel à rota global, ao mesmo tempo que evita obstáculos menores, como outros veículos e árvores. Por este motivo a navegação local precisa ser capaz de atualizar suas rotas rapidamente para reagir à mudanças no ambiente, especialmente em situações que envolve sincronização da movimentação de várias viaturas.

A navegação local também deve ser capaz de manobrar os veículos para sair ou entrar em comboio, o que exige que o algoritmo utilizado leve em consideração as restrições de raio de giro das viaturas e a possibilidade de andar em marcha ré. Para isso foi desenvolvido uma adaptação do algoritmo A* Híbrido (figura 1.2).

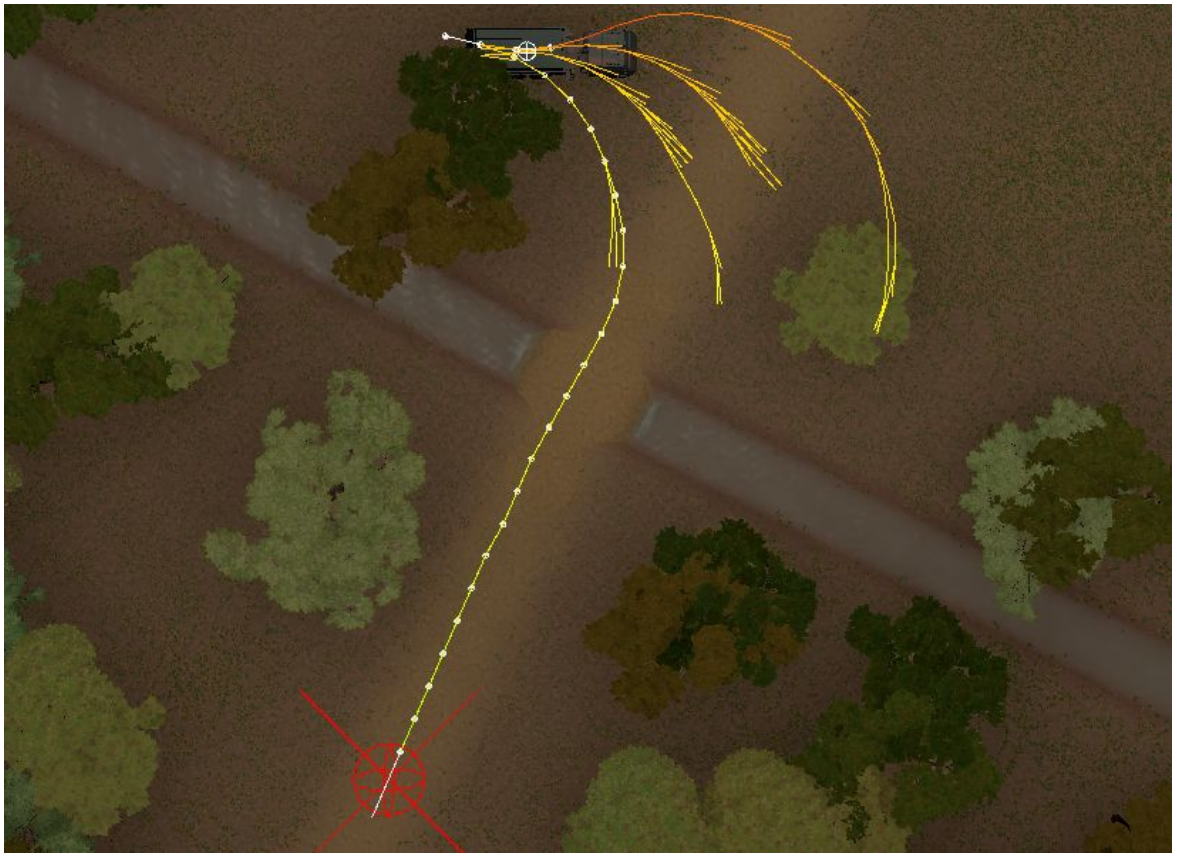


Figura 1.2 – Árvore de busca do algoritmo A* Híbrido resolvendo situação no sistema de simulação SIS-ASTROS onde o veículo deve manobrar para atravessar corretamente uma ponte, utilizando ré e evitando colisão com árvores e com o rio.

Visando acelerar o desenvolvimento tecnológico de veículos autônomos, foi organizado pela DARPA (*Defense Advanced Research Projects Agency*), em 2007, um evento intitulado "DARPA Urban Challenge". O foco deste evento foi o desenvolvimento de veículos autônomos com a capacidade de navegar em um ambiente de trânsito urbano simulado, o que levou ao desenvolvimento de um algoritmo de busca de caminhos, extensão do A* tradicional, chamado A* Híbrido. Este algoritmo se especializa em navegação de carros e considera a translação e rotação do veículo em espaço contínuo, podendo também ser aplicado a veículos virtuais em games ou simuladores, robôs autônomos de serviço em ambientes industriais, e robôs de reconhecimento em áreas de desastre (Petereit, Emter, Frey, 2012).

1.1 Motivação

A abordagem híbrida citada (algoritmo A* híbrido), para cálculo de rotas para veículos, traz um problema: dado que o algoritmo A* Híbrido age em espaço contínuo fora de um grafo conhecido, como o A* tradicional, cálculos de intersecção entre o veículo e os obstáculos do cenário devem ser utilizados para validar os nós gerados pela árvore de busca.

Estes cálculos de intersecção podem representar um grande custo computacional e, se mal otimizados, podem inviabilizar a utilização deste algoritmo para aplicações em tempo real, onde o veículo precisa reagir rapidamente a mudanças no cenário, ou em uma simulação militar de larga escala envolvendo diversos comboios com grande quantidades de viaturas autônomas e geometria complexa dos obstáculos.

Esta abordagem também depende do uso de uma heurística apropriada para alcançar a geração de um menor número de nós na árvore de busca. Quando utilizada uma heurística de distância euclidiana, abordagem utilizada atualmente na navegação local do Simulador SIS-ASTROS, a busca pode ter problemas para planejar rotas em torno de grandes obstáculos, gerando uma quantidade de nós, como ilustrado na figura 1.3.

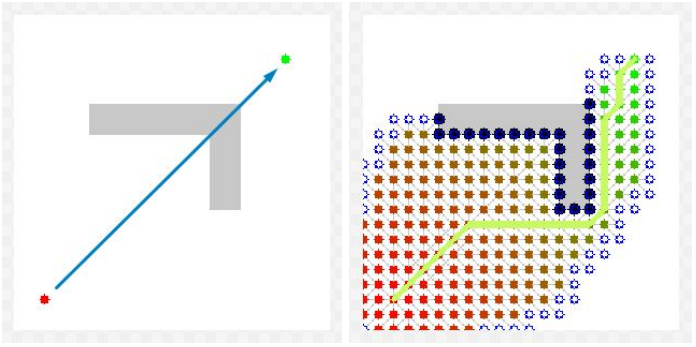


Figura 1.3 – Algoritmo A* utilizando heurística de distância euclidiana, planejando rota do ponto A (círculo vermelho) até o ponto B (círculo verde), em torno de um obstáculo (área cinza). À direita o caminho resultante (linha verde) e todos nós visitados pela busca.

O problema ilustrado na figura 1.3 também acontece na navegação local do sistema SIS-ASTROS, quando viaturas precisam navegar em torno de grandes obstáculos, como lagos, rios, ou diversas viaturas estacionadas próximas umas das outras, como ilustra a figura 1.4.

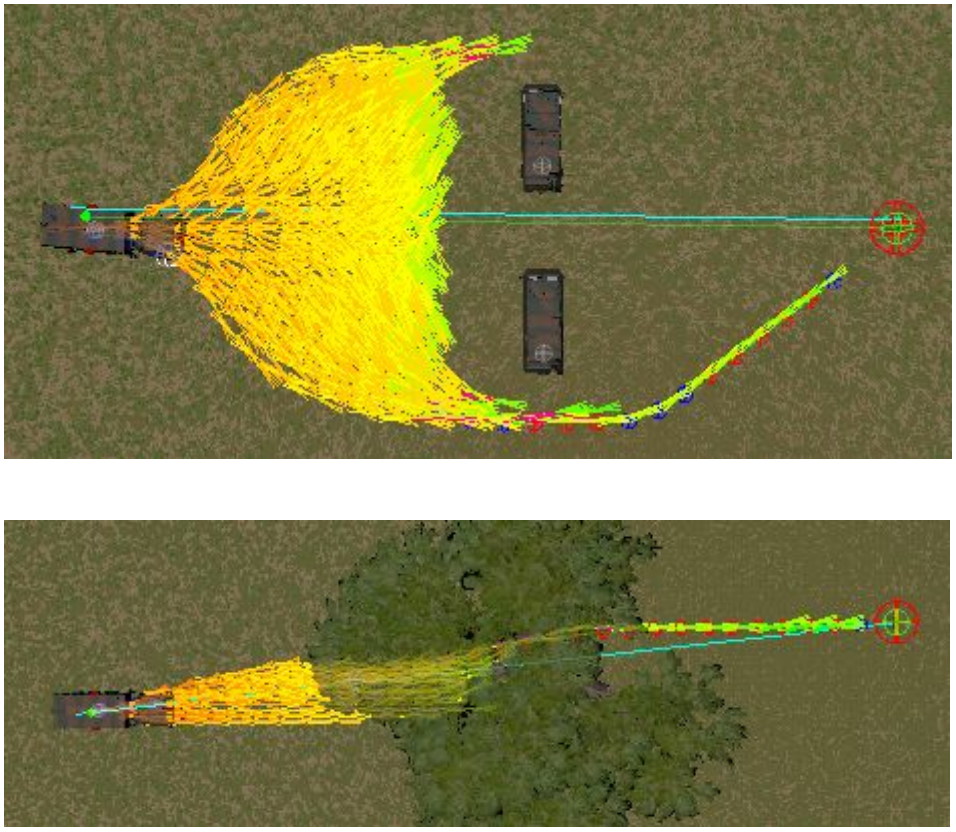


Figura 1.4 – Árvore de busca do algoritmo A* Híbrido implementado no sistema de simulação SIS-ASTROS. Acima uma situação com duas viaturas bloqueando o caminho de outra. Abaixo uma situação com apenas uma árvore bloqueando o caminho da viatura.

1.2 Objetivos

1.2.1 Objetivo geral

Este trabalho tem por objetivo o estudo e aprimoramento do algoritmo A* Híbrido para navegação de veículos em espaço contínuo com restrição de movimento, no que se refere a tempo de execução, para uso em simulações ou jogos. Para atingir tal objetivo, foi estudado o melhor uso dos parâmetros do algoritmo, o aperfeiçoamento dos cálculos de intersecção entre o veículo e os obstáculos do ambiente simulado, e a aprimoração da heurística do algoritmo aplicando técnicas já utilizadas em conjunto com o A* tradicional.

1.2.2 Objetivos específicos

- Implementação de técnicas de otimização da intersecção do veículo com os obstáculos do cenário. Isto inclui: uso de bounding box / bound circle, aproximação do volume do veículo utilizando intersecção de pontos.

- Implementação de uma hash circular para representação aproximada de polígonos, para cálculo de intersecção de ponto com polígonos de complexidade O(1). Desenvolver o algoritmo que converte um polígono qualquer para um ou mais polígonos representáveis por esta hash.

- Pré-processamento de obstáculos de grande área, para geração de um grafo de visibilidade. Sobre este grafo, calcular os menores caminhos dos pontos do grafo até o alvo desejado, e utilizar estes caminhos para calcular uma melhor heurística, e aplicá-la ao algoritmo A* Híbrido, reduzindo a árvore de busca em situações onde o caminho é bloqueado por grandes obstáculos.

- Desenvolvimento de um software que integra a implementação do algoritmo A* Híbrido às técnicas de otimização desenvolvidas neste trabalho. O algoritmo, assim como a utilização das técnicas de otimização devem ser parametrizáveis para medição e comparação do desempenho das mesmas em baterias de teste.

- Análise de resultados referentes a desempenho e qualidade dos caminhos quando utilizadas diferentes resoluções da hash discreta de busca.

- Validação das técnicas, mensurando seu desempenho em testes exaustivos, utilizando variados parâmetros, em diversos ambientes virtuais, para coleta de dados que incluem tempo de execução, tempo de pré-processamento, uso de memória, e qualidade dos caminhos gerados.

1.3 Organização do texto

O texto está organizado como segue. O capítulo 2 apresenta uma revisão da literatura, realizando uma breve introdução do leitor aos métodos utilizados para cálculo de rotas para veículo, que são objetos de estudo deste trabalho.

O capítulo 3 descreve em detalhes o software desenvolvido no trabalho, a versão de A* híbrido implementada, assim como a implementação da hash de aproximação de polígonos e o uso de grafos de visibilidade.

O capítulo 4 mostra os resultados no que tange a otimização da busca de rotas quando aplicadas as técnicas de otimização desenvolvidas ao algoritmo A* híbrido nos diversos cenários de teste.

Por fim, o capítulo 5 resume as conclusões de acordo com os resultados encontrados. Além do mais, elenca-se os possíveis trabalhos futuros a serem realizados, com o intuito de aprimorar ainda mais as técnicas desenvolvidas.

2 REVISÃO BIBLIOGRÁFICA

2.1 Algoritmo de busca A*

Amplamente utilizado para busca de caminhos sobre grafos, o algoritmo A*, inicialmente publicado em 1968 por Peter Hart, Nils Nilssone e Bertram Raphael, combina o algoritmo de Edsger Dijkstra (Dijkstra, 1959) com o uso de uma função heurística que estima a distância a ser percorrida de um ponto no espaço até o ponto alvo, visando priorizar a exploração de nós mais promissores, reduzindo a quantidade de nós gerados pela árvore de busca, sem deixar de entregar o caminho ótimo entre dois nós de um grafo finito.

Dentre um conjunto de nós de um grafo, o algoritmo armazena uma lista de nós abertos (nós que deverão ser avaliados) e uma lista de nós fechados (nós já avaliados, que não serão visitados novamente pela busca). Dado um nó inicial e um nó alvo, inicia-se o algoritmo adicionando o nó inicial do caminho à lista de nós abertos. Toda vez que um nó é aberto deve-se calcular F, que é dado por G + H. O parâmetro G representa o custo acumulado até este nó, e o parâmetro H representa a estimativa da distância que falta ser percorrida para chegar ao nó alvo.

Após a adição do primeiro nó à lista de nós abertos começa o processo iterativo do algoritmo: a cada iteração, é removido da lista de nós abertos aquele nó com menor parâmetro F, que é então adicionado à lista de nós fechados. Em seguida o algoritmo avalia os vizinhos do nó que está sendo fechado nesta iteração:

- se um vizinho do nó fechado não está na lista de nós abertos ele é adicionado à esta lista. O nó fechado é considerado “pai” do nó vizinho. Seu G é calculado somando o G do nó pai com a distância entre o vizinho e o pai.
- se um vizinho do nó fechado já está na lista de nós abertos, seu pai atual deve ser substituído pelo nó sendo fechado caso o novo G seja menor do que seu G atual (o que significa que foi encontrado um melhor caminho para se chegar neste nó vizinho, do que o que estava sendo considerado anteriormente). Tanto o G, o H, e o F deste nó devem ser recalculados.
- se o vizinho já está na lista de nós fechados, ele é ignorado.

A iteração se repete até que algum dos dois casos a seguir aconteçam:

1. O nó sendo fechado é o nó alvo do algoritmo. Neste caso o caminho foi encontrado, e pode ser reconstruído percorrendo os pais do nó alvo que acabou de ser fechado.
2. Não há mais nenhum nó para ser fechado, mas o nó final não foi encontrado. Neste caso o caminho entre o nó inicial e o final não existe.

2.2 A* Híbrido

O algoritmo A* Híbrido, objeto de estudo deste trabalho, supera algumas limitações que torna o algoritmo A* tradicional menos adequado para aplicação na navegação de veículos autônomos, os quais possuem restrições de movimento, e precisam navegar em espaços contínuos. Este algoritmo é denominado A* Híbrido pois combina o estado discreto do A* tradicional com a representação do veículo em espaço contínuo, para entregar caminhos sobre o qual é possível o veículo navegar, livre de curvas fechadas que ultrapassam os limites de raio de giro do veículo.

O algoritmo A* Híbrido foi utilizado por pesquisadores de Stanford no desafio *DARPA Urban Challenge*, garantindo o segundo lugar ao seu carro Junior neste campeonato de carros autônomos de 2007 (Montemerlo et al., 2008).

A cada etapa do algoritmo os nós expandidos na árvore de busca seguem a simulação da movimentação do veículo em N direções diferentes, incluindo ré, e seguindo as limitações de raio de giro do veículo. A figura 2.1 demonstra a diferença entre A* e A* Híbrido, no que tange o posicionamento dos nós no espaço discreto (no caso do A*) e no espaço contínuo (no caso do A* Híbrido).

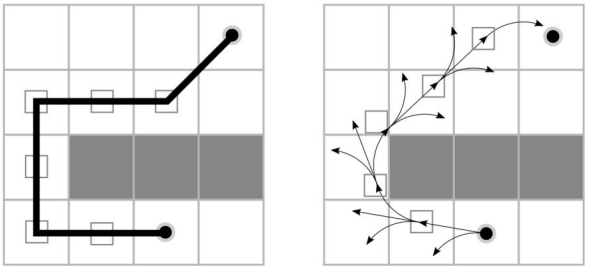


Figura 2.1 – À esquerda um exemplo de A* tradicional. À direita a diferença de posicionamento dos nós de busca quando se utiliza A* Híbrido.

Os nós gerados pelo A* Híbrido armazenam a posição *x* e *y* do veículo no espaço 2D, assim como seu ângulo de rotação θ. Estes dados de posicionamento são atribuídos a uma célula de uma grid discreta 3D, finita, de tamanho parametrizável.

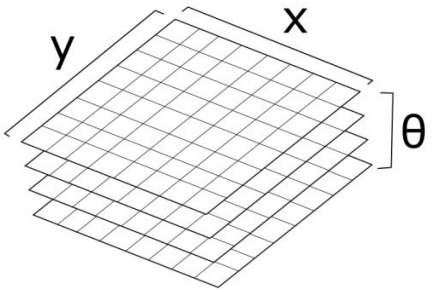


Figura 2.2 – Representação da hash espacial com uma terceira dimensão para indexação dos diferentes ângulos de rotação do veículo.

As posições do veículo geradas durante a busca do caminho pelo algoritmo A* Híbrido devem estar livres de intersecções com os obstáculos do ambiente. Para isso devem ser utilizados algoritmos de intersecção do veículo (retângulo de rotação arbitrária) com os obstáculos, que podem variar de formato e complexidade (círculos, retângulos, linhas, polígonos, entre outros). Estes cálculos de intersecção podem ser bastante custosos, tomando maior parte do tempo de execução do algoritmo.

Muitas vezes é útil circunscrever uma forma tridimensional complexa com uma forma mais simples, como para cálculos de intersecção (O'Rourke, 1985). Uma forma geométrica comumente utilizada para implementação de ray-tracing é a AABB (*axis aligned bounding box*), um retângulo alinhado aos eixos coordenados, o que facilita o cálculo de intersecção da forma com outros tipos de geometria.

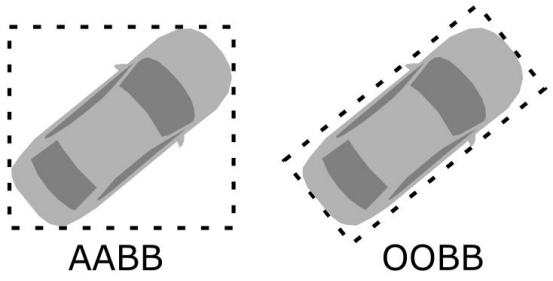


Figura 2.3 – Exemplo de bounding box alinhado aos eixos coordenados (à esquerda) e orientado ao objeto (*object oriented bounding box,* à direita), aproximando o volume de um objeto mais complexo (veículo).

Grafos de visibilidade

Em geometria computacional, um grafo de visibilidade consiste de um grafo de pontos em um plano cartesiano, onde arestas deste grafo representam uma conexão visível entre estes dois pontos, ou seja, o segmento de linha formado por estes dois pontos nao tem intersecção com um conjunto de obstáculos. Grafos de visibilidade se demonstram úteis no cálculo de caminhos ideais na presença de obstáculos complexos, podendo ser utilizados para acelerar a busca do algoritmo A* (Shah et al.,2016).

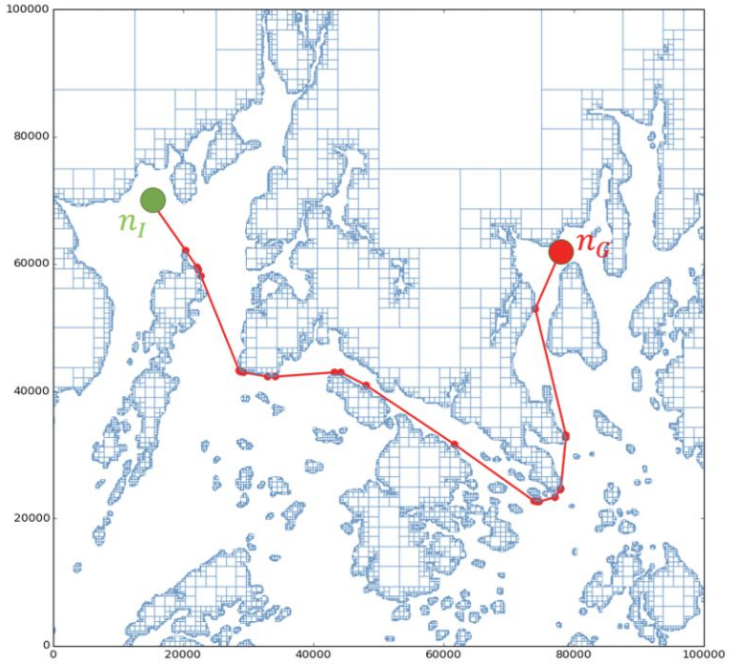


Figura 2.4 – Caminho computado em cenário real de larga escala (100km²), possibilitado pelo uso de grafos de visibilidade gerados a partir de uma quadtree.

3 DESENVOLVIMENTO

A linguagem de programação utilizada foi C#, utilizando a IDE Visual Studio Community 2019 e executado dentro do motor gráfico Unity. Os motivos da escolha deste motor gráfico são a integração com a IDE utilizada, as ferramentas que o motor oferece para implementação de interface gráfica do software, e as ferramentas de depuração do código e fácil medição de desempenho em tempo de execução. Tanto a IDE quanto o motor gráfico utilizado são gratuitos para uso não comercial.

O desenvolvimento do software, assim como os testes de desempenho foram feitos em um computador com sistema operacional Windows 10, com 8 GB de memória RAM e processador Ryzen 5 2400G com 4 núcleos / 8 threads e clock base de 3.6 Ghz.

3.1 A* Híbrido implementado

Para o fim do estudo das técnicas de otimização que são objetivo deste trabalho, foi necessário a implementação de uma adaptação do algoritmo A* Híbrido, baseado na implementação atual presente no simulador SIS-ASTROS, porém livre de especificidades de tal simulador. Os detalhes da implementação utilizada são os que seguem.

A implementação proposta do algoritmo A* híbrido tem como base o A* tradicional. Porém a diferença entre A* e o A* Híbrido implementado começa pelos nós da árvore de busca: cada nó da árvore de busca do A* Híbrido deve considerar a posição x e y do veículo no espaço contínuo 2D, assim como seu ângulo de rotação θ, como proposto no algoritmo original. Estes nós são gerados durante a execução do algoritmo, baseado no raio mínimo de giro do veículo r, uma distância d, a ser percorrida sobre o raio de giro, e um coeficiente c (entre -1 e 1), que define a direção que o veículo se moverá (esquerda, centro, ou direita).

Foram utilizados coeficientes c negativos para curvas à esquerda, e positivos para curvas à direita. O raio de giro resultante, utilizado para simular a movimentação do veículo, se dá por r / c, logo quanto menor a magnitude de c maior o raio de giro. O coeficiente 0 representa uma movimentação em linha reta e é tratado como caso especial no código implementado. Coeficientes de magnitude entre 0 e 1 representam movimentações sobre um círculo com raio de giro intermediário, entre o raio mínimo de giro do veículo e uma linha

reta, como ilustrado na figura 3.1. Coeficientes com magnitude maiores do que 1 representam um raio de giro menor do que o mínimo atingível pelo veículo.

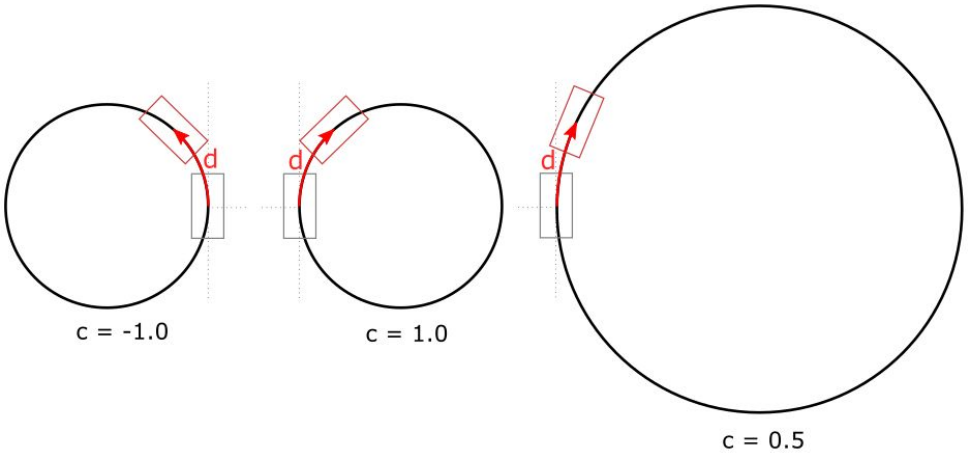


Figura 3.1 – Exemplos de movimentação utilizando diferentes coeficientes *c*.

O parâmetro *d,* que representa a distância a ser percorrida pelo veículo sobre o círculo resultante uma vez que foram definidos *r* e *c*, pode ser positivo ou negativo (no caso de movimentação em marcha ré). Estes parâmetros podem ser escolhidos com base na aplicação do algoritmo, e neste trabalho foram utilizados três valores para c: -1, 0 e 1, e dois valores para d: 2 metros e -2 metros. Logo, para cada nó gerado pelo algoritmo de busca, são gerados 6 novos nós, como ilustrado na figura 3.2.

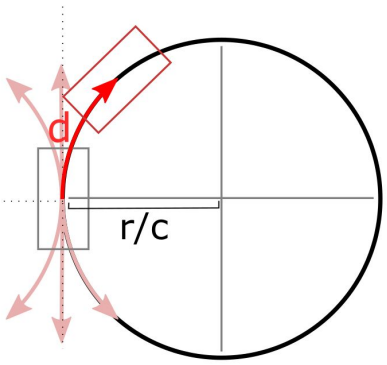


Figura 3.2 – Exemplo de movimentação gerado por uma iteração do A* Híbrido.

3.2 Grid discreta de nós

Cada nó gerado durante a busca deve ser atribuído à uma célula de uma grid discreta, para evitar a expansão de nós com posição e rotação similares, limitando o tamanho da árvore de busca, e consequentemente o tempo que o algoritmo leva para encontrar o caminho. Cada

nó gerado ocupa uma célula desta grid, e células já ocupadas não serão visitadas novamente.

A resolução desta grid irá impactar na quantidade de nós que serão expandidos na busca até encontrar o caminho desejado. Quanto maior a resolução da grid mais o algoritmo demora para encontrar o caminho, porém pode entregar melhores resultados (caminhos de menor custo). Caso a resolução da grid for muito pequena o veículo pode ignorar as melhores rotas ou até falhar na busca do caminho por ser incapaz de identificar passagens pequenas entre obstáculos. A grid deve compreender toda área de busca considerada pelo veículo.

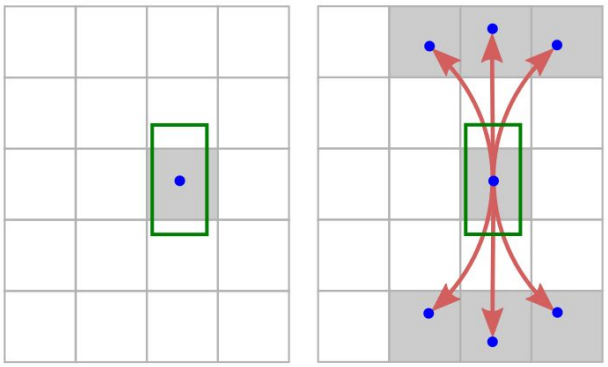


Figura 3.3 – Exemplo de grid. À esquerda o veículo (retângulo verde) em sua posição inicial. A célula em cinza está ocupada. À direita novos nós foram gerados a partir da posição inicial (pontos azuis) ocupando novas células.

Cada célula desta grid deve armazenar N subdivisões para diferenciar os ângulos de rotação do veículo. Nós de busca que estão próximos no espaço 2D, porém com ângulos de rotação do veículo diferentes, devem ser distinguidos uns dos outros. No início da busca de um caminho a grid deve ser reiniciada (marcar todas células como não ocupadas), e cada nó que for gerado pela busca deve, baseando-se na sua posição e rotação, calcular sua posição discreta nesta grid e verificar o estado de ocupação da célula correspondente. Caso já esteja ocupada o nó gerado não será adicionado à lista de nós abertos. Caso contrário o nó é aberto e a célula é marcada como ocupada.

Uma abordagem foi utilizada para limpar a hash 3D de maneira rápida, para reutilizá-lo em execuções consecutivas do algoritmo: cada célula da hash armazena um dado numérico inteiro sem sinal (por exemplo byte, ushort, ou uint). Uma outra variável '*t*', do mesmo tipo da célula, armazena o valor que é utilizado para identificar células ocupadas, e começa com o valor 1, enquanto toda a hash, por padrão da linguagem, começa com o valor 0. Durante a execução do algoritmo a avaliação de uma célula *c* se faz comparando o valor de *c*

com o valor de *t*. Se *c* = *t* então a célula é considerada ocupada. Para limpar o estado de ocupação das células da hash basta incrementar o valor de *t* em 1. Quando *t* atingir o valor máximo representável pelo seu tipo de dado é necessário a realocação da hash para zerar o estado das células.

3.3 Intersecção com obstáculos

Para cada nó gerado também deve-se calcular a intersecção do veículo com os obstáculos do cenário naquela posição e rotação. Caso exista uma intersecção o nó gerado deve ser descartado. Os obstáculos considerados neste trabalho se assemelham aos obstáculos encontrados no simulador SIS-ASTROS (veículos, pontes, árvores, e regiões de lago). Foram considerados neste trabalho obstáculos retangulares (com rotação arbitrária), circulares, e polígonos, distribuídos por um cenário de dimensões 50m x 50m. As dimensões do retângulo utilizado para representar o veículo tem largura 1,9m e comprimento 4,5m.

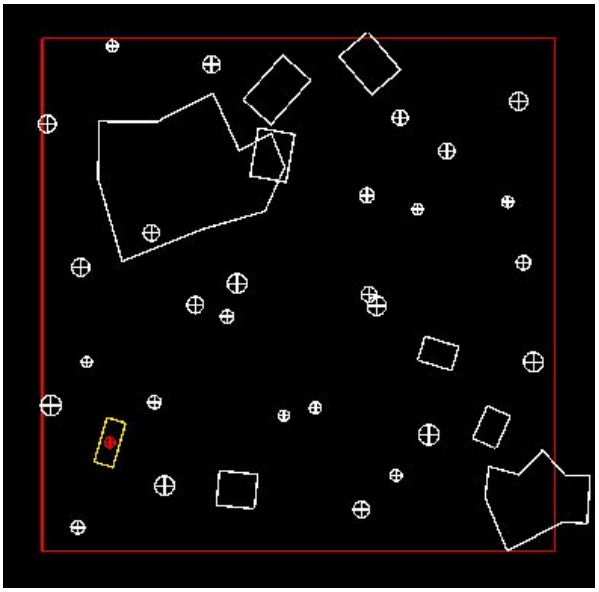


Figura 3.4 – Exemplo de cenário com obstáculos. A área navegável é indicada em vermelho, e o veículo em amarelo.

Os cálculos de intersecção utilizados aproximam a geometria do veículo utilizando dois círculos, como ilustra a figura 3.5. Desta maneira, aplicando uma expansão na geometria dos obstáculos, pode-se utilizar cálculos mais simples de intersecção de ponto com os obstáculos, ao invés de intersecção de um retângulo de rotação arbitrária com os obstáculos.

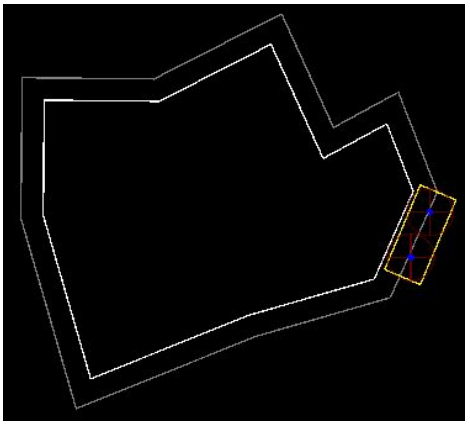


Figura 3.5 – Veículo (em amarelo) próximo de uma colisão com um polígono (branco). A linha cinza representa a geometria expandida do polígono. Os pontos azuis representam os pontos utilizados para aproximar a geometria do veículo. Para definir se o veículo intersecta o polígono se usa a intersecção dos pontos azuis com o polígono expandido. O mesmo se aplica a retângulos expandidos e círculos expandidos.

Os cálculos para definir se um ponto intersecta um polígono são mais custosos computacionalmente quando comparados com os cálculos de intersecção entre retângulos e pontos, ou intersecção entre círculos e pontos (que pode ser facilmente definida calculando a distância entre o ponto e a origem do círculo). Para o teste de intersecção entre pontos e polígonos foram utilizados duas técnicas, uma já presente na literatura, e outra é contribuição deste trabalho. São elas (respectivamente) o método de *crossing number*, e uma hash de aproximação de polígonos.

O método de *crossing number,* para definir se um ponto está dentro de um polígono, consiste em traçar uma reta entre o ponto e um ponto fora do polígono. Este ponto fora do polígono pode ser calculado utilizando a *bounding box* que o envolve. O número de intersecções entre esta reta e os segmentos que formam o polígono nos diz se o ponto está fora ou dentro do mesmo: o ponto está dentro do polígono quando o número de intersecções é ímpar, e fora quando o número de intersecções é par, como ilustra a figura 3.6.

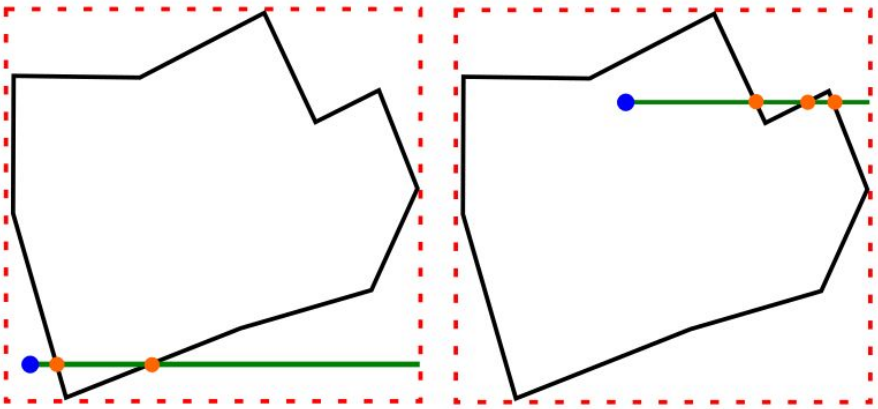


Figura 3.6 – Exemplo de intersecção de pontos (em azul) com um polígono. A linha verde é utilizada para testar intersecções com o polígono. À esquerda a reta intersecta o polígono duas vezes, indicando que o ponto está fora do polígono. À direita a reta intersecta o polígono três vezes, indicando que o ponto está dentro do polígono.

Este método depende do cálculo de intersecção entre dois segmentos de linha, e este cálculo é executado uma vez para cada segmento que forma o polígono. Isto significa que o tempo de execução escalona linearmente com a quantidade de segmentos que formam o polígono (complexidade O(n)).

3.4 Método baseado em hash para aproximação de polígonos

Foi desenvolvido neste trabalho uma técnica de representação de polígono baseado em hash espacial, para acelerar cálculos de intersecção entre pontos e polígonos. Esta técnica pode acelerar significativamente o algoritmo A* Híbrido em situações onde polígonos são os obstáculos predominantes no ambiente virtual, apenas apresentando custo extra em memória.

A representação de polígonos por esta hash consiste em definir um centro *p* para o polígono, e um array de *n* distâncias $f_1, f_2, \ldots, f_n$. Estas distâncias são a distância entre o centro do polígono e os pontos que compõem as extremidades do polígono. Estes pontos da extremidade estão agrupados em 4 diferentes quadrantes (norte, leste, sul e oeste). Cada quadrante pode armazenar uma quantidade diferente de pontos. O polígono representado em hash é a aproximação de um polígono qualquer, como exemplificado na figura 3.7, apresentando um erro de aproximação. Tal erro pode ser reduzido armazenando mais pontos na hash.

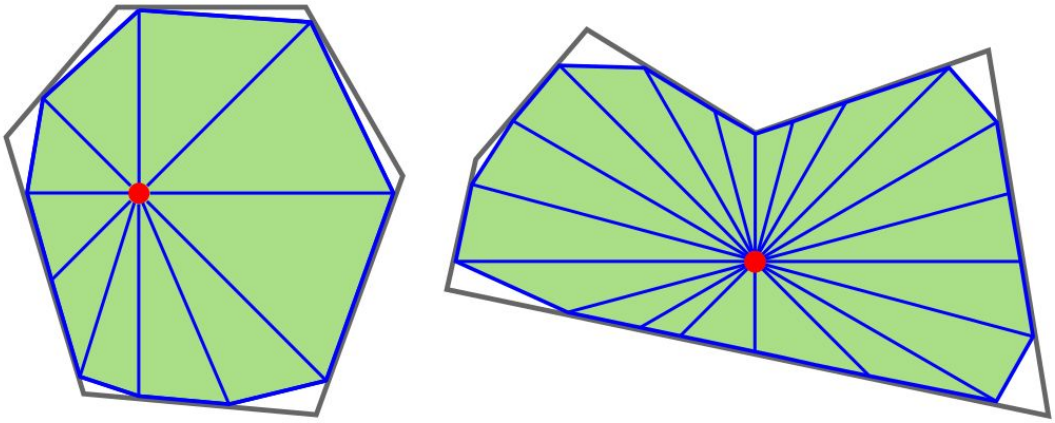


Figura 3.7 – Dois exemplos de polígonos representados pela hash de aproximação. Os polígonos em cinza são os polígonos originais, enquanto as áreas verdes são as áreas de aproximação utilizando a hash.

Em memória são armazenados: o centro do polígono no plano 2D, um array de distâncias (float), e índices que indicam o início de cada quadrante dentro do array (figura 3.8).

```
public Vector2 center;
public List<float> distances = new List<float>();

public int q1Start;
public int q2Start;
public int q3Start;
```

Figura 3.8 – Código fonte dos dados armazenados pela hash.

A posição de cada ponto da hash é definida pelo quadrante em que se encontra e vetores *v* igualmente espaçados em torno de um quadrado de lado 2. No caso do primeiro quadrante (norte) os vetores tem coordenadas (x,1), onde x varia conforme o índice do ponto dentro do seu quadrante. Por exemplo, em um caso onde há quatro pontos no quadrante norte, os vetores que definirão a posição dos pontos terão coordenadas (-1,1), (-0.5, 1), (0,1), e (0.5, 1), como ilustrado na figura 3.9.

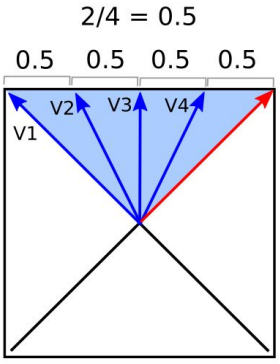


Figura 3.9 – Exemplo de quadrante com quatro pontos. Para definir a posição cada ponto do polígono pertencente a este quadrante utiliza-se os vetores em azul indicados na figura. O vetor mais à direita (em vermelho) já pertence ao próximo quadrante (leste).

Estes vetores não precisam ser calculados durante o teste de intersecção entre um ponto e o polígono representado, mas servem de base para converter um polígono qualquer em um polígono nesta representação.

Conversão de um polígono arbitrário para representação em hash

Para aplicação nos cenários de teste, é necessário converter os polígonos presentes como obstáculos do cenário para a representação utilizando o método baseado em hash desenvolvida. Esta etapa de pré-processamento só é executada uma vez por cenário. Não é necessário executá-la toda vez que um caminho é calculado.

Quando um polígono é representado utilizando o método desenvolvido os pontos deste polígono devem estar visíveis para o ponto central da hash. Logo, esta conversão depende da seleção de um "núcleo" para o polígono. Este núcleo representa a área onde, qualquer ponto da extremidade do polígono está visível para qualquer ponto dentro desta área, como ilustrado na figura 3.10. É garantido que este núcleo é um polígono convexo. Certos polígonos não apresentam um núcleo válido, e precisam ser subdivididos em polígonos menores para que o núcleo exista e a representação seja possível. Esta subdivisão será detalhada na próxima seção do trabalho.

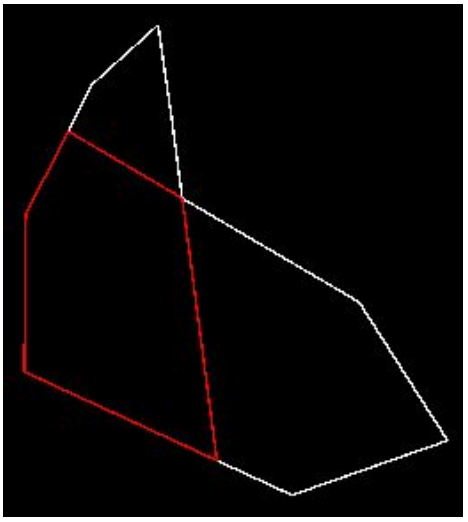


Figura 3.10 – Exemplo de polígono, e seu núcleo (em vermelho). Qualquer ponto dentro da área vermelha está visível para qualquer ponto nas extremidades do polígono.

Para encontrar este núcleo o algoritmo utiliza o *bounding box* do polígono como ponto de partida. Em seguida itera-se sobre cada segmento do polígono, criando uma reta tangente ao segmento sendo avaliado, e subtraindo do núcleo a área que está para fora desta reta, como demonstrado na figura 3.11.

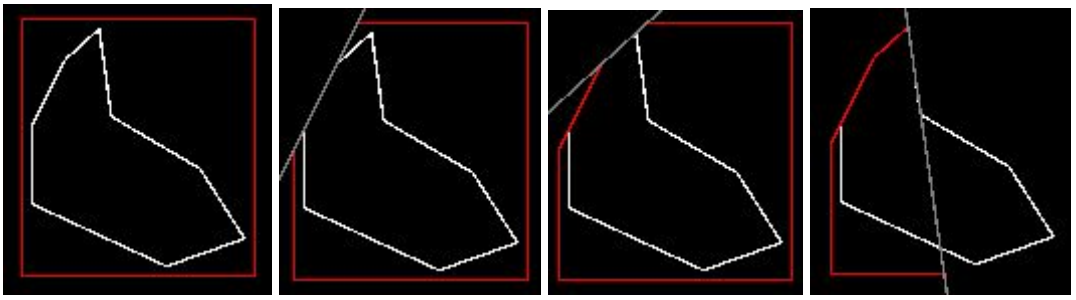


Figura 3.11 – Quatro passos de subdivisão do núcleo do polígono de exemplo.

O núcleo resultante é um polígono convexo, e a média dos pontos que formam este núcleo são utilizadas como ponto central *p* da hash que irá aproximar o polígono original. A partir deste ponto central *p,* as distâncias a serem armazenadas na hash são calculadas a partir da intersecção entre as extremidades do polígono a ser representado e as retas que partem do ponto *p* escolhido na direção dos vetores correspondentes para cada ponto da hash (como descrito na seção 3.4, e ilustrado na figura 3.12).

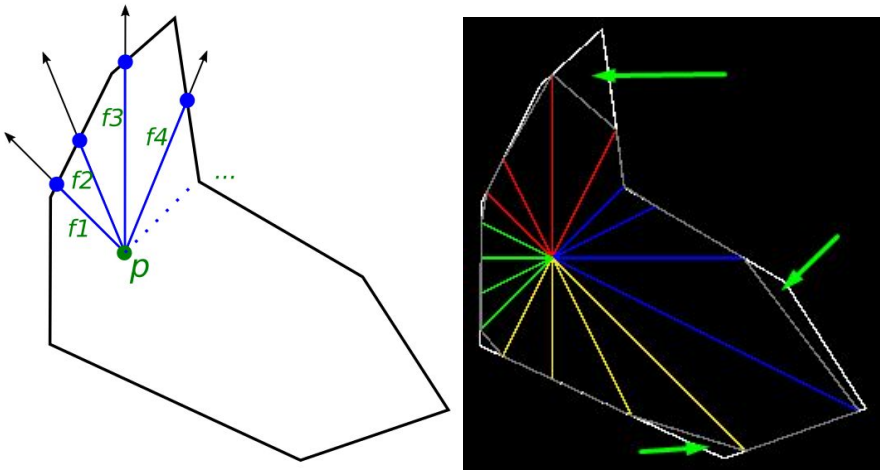


Figura 3.12 – Conversão de um polígono após a seleção de um ponto central *p*. As distâncias f_i são calculadas através das distâncias entre *p* e as extremidades do polígono. À direita todos os quadrantes do polígono resultante, apresentando um erros de aproximação (indicados pelas setas verdes) devido ao uso de poucos pontos na hash.

Alguns polígonos não podem ser convertidos, pois não satisfazem a restrição de existência de um núcleo do polígono, a partir do qual todos pontos do polígono são visíveis. A figura 3.13 mostra um exemplo de polígono que não pode ser convertido utilizando este método. Isso requer que polígonos deste tipo sejam subdivididos em dois ou mais polígonos.

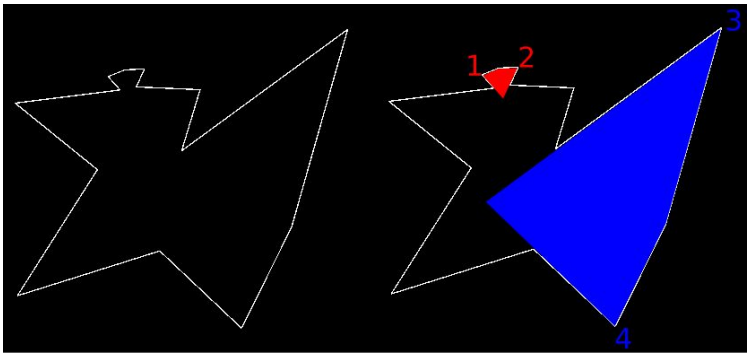


Figura 3.13 – Exemplo de polígono impossível de ser representado pela hash desenvolvida, dado que os pontos 1 e 2 só são visíveis para pontos dentro da área vermelha, e os pontos 3 e 4 só são visíveis para pontos dentro da área azul, não existe um núcleo utilizável para este polígono, que inclua todos os 4 pontos.

Subdivisão de polígonos

Durante o processo de cálculo do núcleo de um polígono, o algoritmo pode chegar em um impasse, onde uma aresta sendo avaliada irá tornar o núcleo nulo, como demonstrado nas

figuras 3.14 e 3.15. Se em algum momento da subdivisão do núcleo o mesmo estiver completamente à esquerda da reta (em direção ao centro do polígono partindo da aresta avaliada), o polígono como um todo deve ser subdividido, e o algoritmo deve recomeçar para cada polígono resultante desta subdivisão.

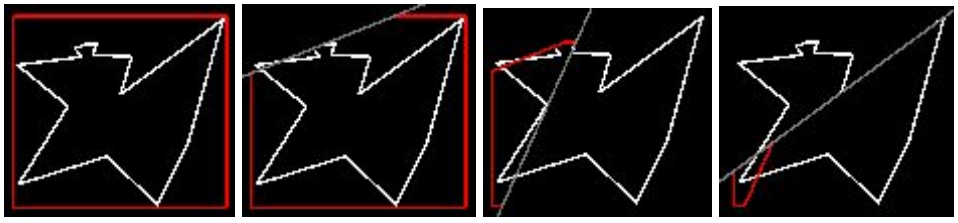


Figura 3.14 – Alguns passos da subdivisão do núcleo do polígono de exemplo, antes de chegar em um impasse.

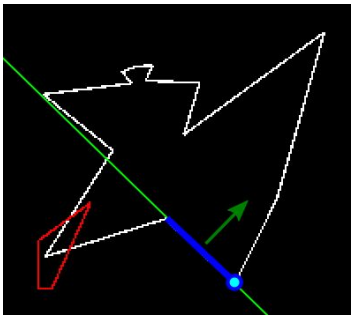


Figura 3.15 – O segmento do polígono sendo avaliado irá tornar o núcleo do polígono nulo, pois o mesmo está completamente à esquerda da reta.

Para subdividir o polígono, primeiramente são mapeados, quais pontos do polígono são côncavos e quais são convexos. Escolhemos então o agrupamento de pontos convexos mais próximo ao ponto onde ocorreu o impasse durante a subdivisão do núcleo do polígono, incluindo os pontos côncavos que os rodeiam, como ilustrado na figura 3.16. Estes pontos irão compor um novo polígono *a* e os pontos restantes irão compor um polígono *b*.

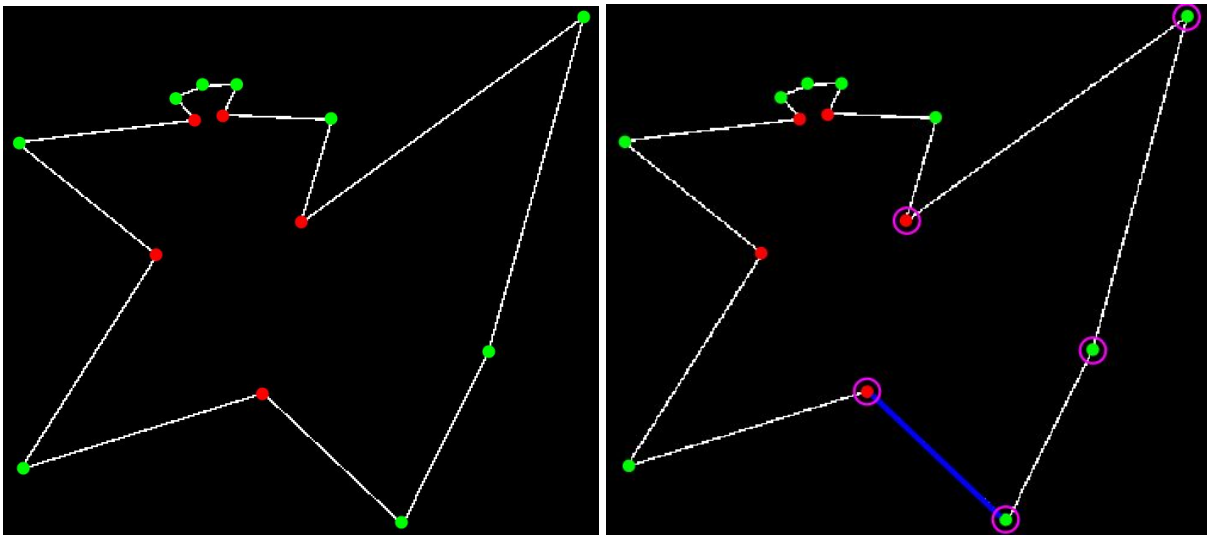


Figura 3.16 – À esquerda o mapeamento de concavidades do polígono de exemplo. Pontos em verde são convexos, e pontos em vermelho são côncavos. À direita o segmento que causou o impasse (linha azul) e os pontos escolhidos para formar um novo polígono (circulados em rosa).

O mesmo método de conversão é executado para o polígonos *a* e *b*. O polígono *a* é garantido ser um polígono convexo o que garante que ele pode ser representado pela hash sem necessidade de mais subdivisões. Já o polígono *b* pode precisar de mais subdivisões até poder ser representado pela hash, como ocorre no exemplo da figura 3.17.

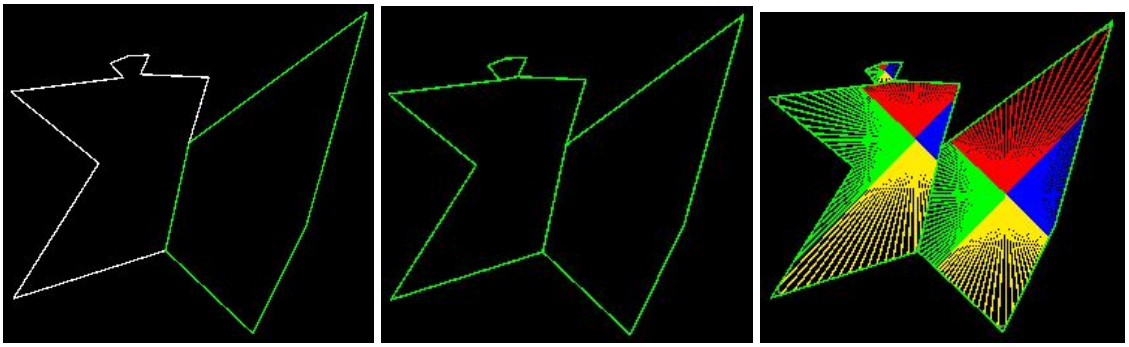


Figura 3.17 – À esquerda resultado da primeira subdivisão do polígono de exemplo. Ao centro o resultado final de todas subdivisões do polígono (totalizando 3 polígonos finais), e à direita a sua representação usando 3 hashes.

Esta escolha de pontos para subdivisão do polígono é feita de maneira gulosa, ou seja, não resulta na subdivisão ótima do polígono.

Intersecção entre ponto e polígono utilizando a hash de aproximação

Para calcular a intersecção entre um ponto *q* qualquer e o polígono representado pela hash desenvolvida se dá da seguinte maneira: primeiro é necessário definir o quadrante onde o ponto se encontra em relação ao centro *p* do polígono. Isto se faz calculando um vetor *w*, que se dá por *w* = *q* - *p*. Em seguida se compara as coordenadas *x* e *y* do vetor *w*. Os quatro casos são os seguintes, e ilustrados na figura 3.18:

1. Se |*x*| < |*y*| e y > 0 : o ponto está no quadrante norte.
2. Se |*x*| < |*y*| e y <= 0 : o ponto está no quadrante sul.
3. Se |*x*| >= |*y*| e x > 0 : o ponto está no quadrante leste.
4. Se |*x*| >= |*y*| e x <= 0 : o ponto está no quadrante oeste.

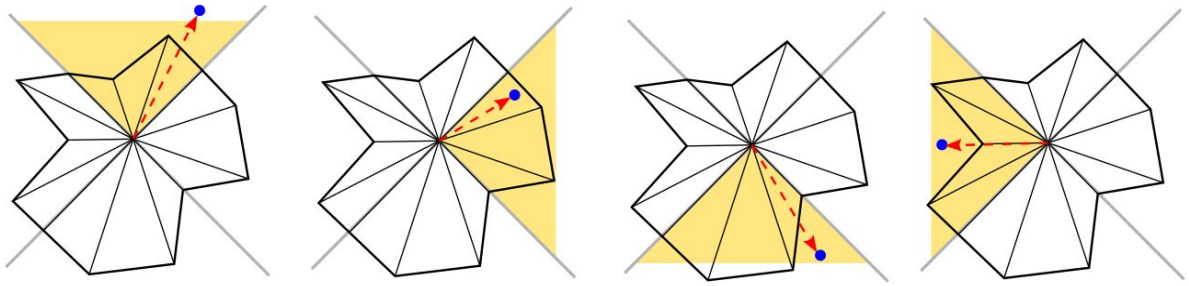


Figura 3.18 – Exemplo de quatro pontos arbitrários (em azul) em quatro quadrantes diferentes do mesmo polígono.

Definido o quadrante onde se encontra o ponto, o próximo passo é normalizar o vetor de maneira que ele se encontre na extremidade do quadrado de lado 2. Isso se dá dividindo o vetor pela sua coordenada mais significativa (*y* no caso dos quadrantes norte e sul, e *x* no caso dos quadrantes leste e oeste), resultando em um vetor *w'* (figura 3.19). A coordenada mais significativa passará a ter o valor 1, e a menos significativa irá variar entre -1 e 1.

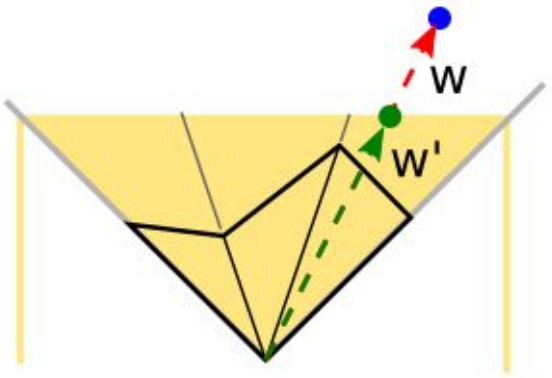


Figura 3.19 – Exemplo de um ponto no quadrante norte. O vetor *w* vai do centro do polígono até o ponto (azul), já o vetor w'(em verde), dado por w / w_y, vai do centro do polígono até a extremidade do quadrado (em amarelo).

Chamamos a coordenada menos significativa do vetor *w'* de *j*. Mapeamos *j* de um valor entre -1 e 1 para um coeficiente *k,* com valor entre 0 e 1, dado por $k = (j+1) / 2$. Com o coeficiente *k* podemos calcular um índice dentro do quadrante. Sabendo a quantidade *n* de pontos dentro deste quadrante, o índice Q resultante se dá por $\lfloor k * n \rfloor$ (onde $Q < n$).

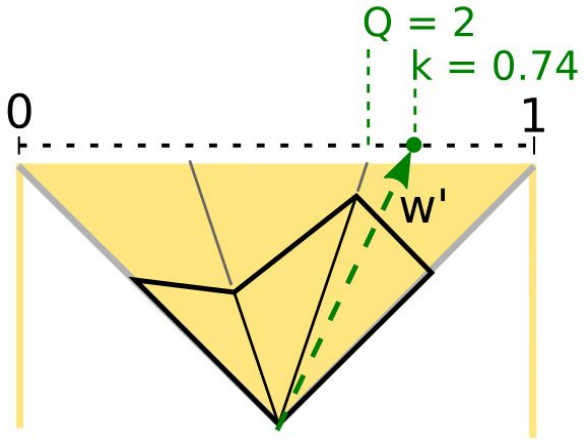


Figura 3.20 – O coeficiente *k* na extremidade do quadrado, para o ponto de exemplo (em verde), com valor 0.74. Dado que existem 3 pontos neste quadrante, o Q resultante se dá por 2.

Em seguida, calcula-se um novo coeficiente *l* de interpolação entre o ponto Q definido e o próximo ponto da hash. No exemplo da figura 3.20 o próximo ponto está em outro quadrante, o que não é problema pois os dados de distância dos pontos são armazenados em um array sequencial. Este coeficiente *l* é dado por $(k * n) - Q$, também variando entre 0 e 1 (figura 3.21).

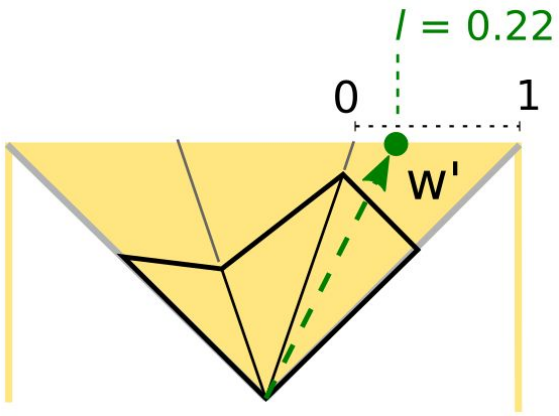


Figura 3.21 – O coeficiente *l* na extremidade do quadrado, para o ponto de exemplo (em verde).

Utilizando o coeficiente *l* aplica-se uma interpolação linear entre as distâncias f_q e f_{q+1} armazenadas nos índices Q e Q+1 da hash. Calcula-se uma distância interpolada D, que é dada por $f_q + (f_{q+1} - f_q) * l$, exemplificada na figura 3.22. Por fim, D é comparada com a distância entre o ponto original e o centro do polígono. Se a distância entre o ponto e o centro do polígono for menor que D o ponto está dentro do polígono, caso contrário o ponto está fora do polígono. Esta interpolação linear resulta em um erro de aproximação (ilustrado na figura 3.23), mas que pode ser contornado utilizando mais pontos dentro de um quadrante, caso este erro de aproximação apresente um problema para o cálculo de rotas utilizando A* Híbrido. A adição de mais pontos na hash representa um custo adicional de memória, mas não há nenhum aumento no tempo de execução do teste de intersecção.

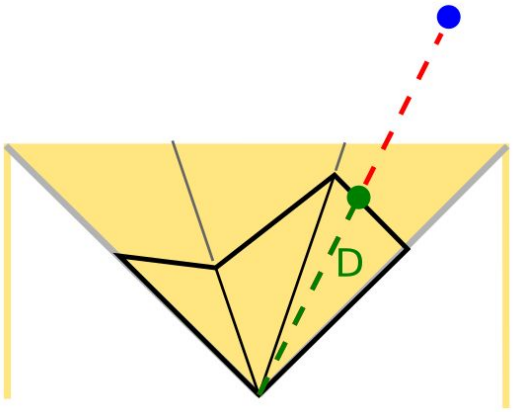


Figura 3.22 – Representação da distância D interpolada, não levando em consideração o erro de aproximação causado pela interpolação linear utilizada.

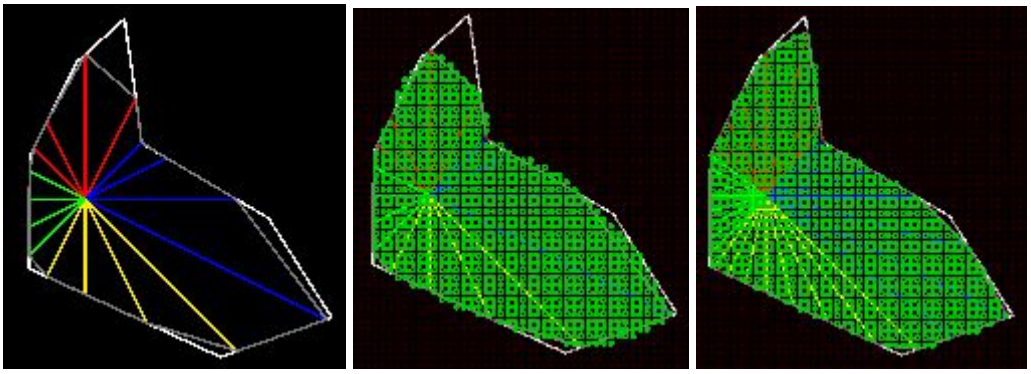


Figura 3.23 – Um exemplo de polígono aproximado usando a hash desenvolvida. Ao centro a hash contém 4 pontos por quadrante. À direita a hash contém 8 pontos por quadrante. A área verde representa a área de intersecção com o polígono. O erro de aproximação foi reduzido significativamente ao dobrar o número de pontos na hash.

3.5 Heurística utilizando grafo de visibilidade

Neste trabalho, visando a redução na quantidade de nós gerados durante a busca, foi implementado um método para aprimoramento da heurística do algoritmo A* Híbrido quando é necessário contornar grandes obstáculos presentes no cenário. O método consiste nos seguintes passos:

Passo 1: selecionar os maiores objetos do cenário. O grafo será construído em torno destes objetos, para garantir que o algoritmo não precisa gerar uma árvore de busca tão grande, e contorná-los com mais facilidade. Para isso foram utilizados os perímetros dos objetos. Perímetros maiores do que um valor *x* parametrizável são incluídos na geração do grafo. Se muitos objetos forem selecionados o grafo apresentar uma redução no desempenho do algoritmo.

Passo 2: gerar nós em torno dos objetos selecionados. Estes nós não podem conter intersecção com os próprios objetos (figura 3.24).

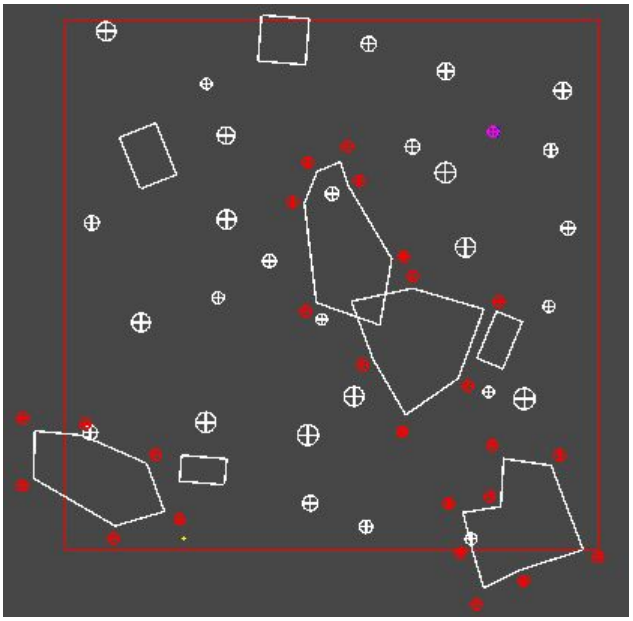


Figura 3.24 – Exemplo de cenário, onde o caminho entre o ponto amarelo e o ponto rosa estão bloqueados por dois grandes polígonos. Todos polígonos da imagem foram selecionados pelo algoritmo e os pontos vermelhos são os nós criados em torno dos obstáculos.

Passo 3: gerar o grafo de visibilidade (figura 3.25). Nesta etapa são calculadas as vizinhanças entre todos os nós do grafo, onde um nó só pode ser vizinho do outro quando a linha entre eles não intersecta nenhum dos obstáculos selecionados pelo algoritmo.

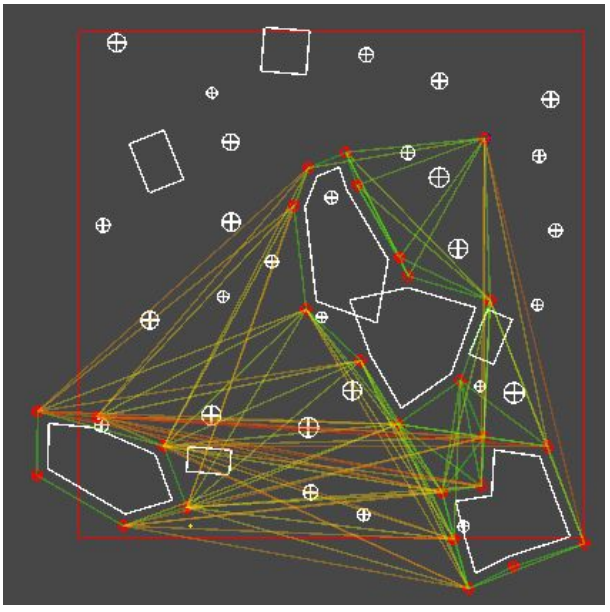


Figura 3.25 – Grafo gerado no cenário de exemplo. Linhas verdes conectam pontos mais próximos uns dos outros, e linhas vermelhas conectam pontos mais distantes.

Passo 4: para cada caminho calculado, deve-se adicionar o ponto alvo deste caminho ao grafo. Este ponto será removido do grafo após o término da execução do A* Híbrido, para reutilizar o grafo em outras execuções.

Passo 5: utilizar o algoritmo de Dijkstra para calcular os melhores caminhos sobre este grafo, partindo de todos os pontos desejados até o ponto de interesse. O algoritmo de Dijkstra, similar ao algoritmo A*, pode ser utilizado para visitar todos nós de um grafo e calcular todos os menores caminhos entre os pontos do grafo e um ponto de destino. Isto nos entrega um conjunto de arestas, que vão dos pontos do grafo até o ponto alvo e serão utilizados para calcular a heurística (figura 3.26).

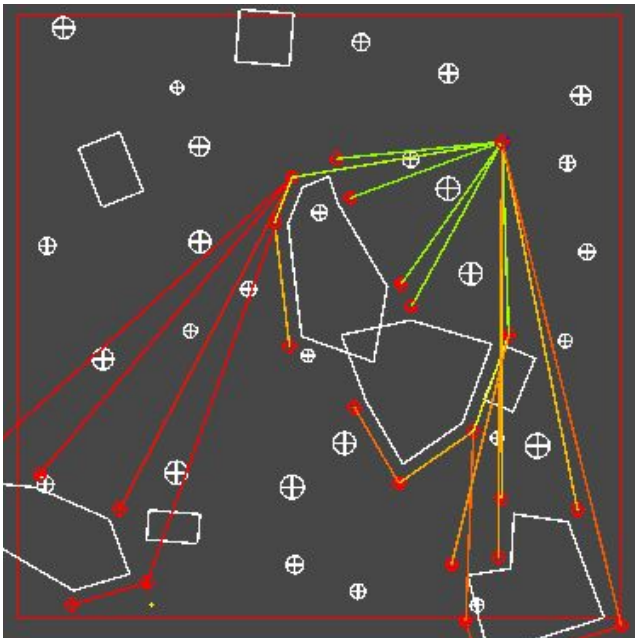


Figura 3.26 – Resultado após a aplicação do algoritmo de Dijkstra sobre o grafo construído. Um ponto de interesse foi adicionado ao grafo no canto superior direito. As linhas verdes representam arestas do grafo mais próximas do ponto alvo, e as vermelhas arestas mais distantes.

Os 3 primeiros passos são executados uma vez por cenário. Os passos 4 e 5 são executados toda vez que se deseja buscar um caminho utilizando A* Híbrido. Uma vez que o grafo está construído, durante a execução do A* Hìbrido, a heurística *h* calculada para um nó, em um ponto do espaço, se dá pela distância entre o ponto e a aresta mais próxima deste ponto (levando em consideração as arestas geradas após aplicação de Dijkstra), somada à distância daquela aresta até o ponto de interesse, sobre o grafo (figura 3.27).

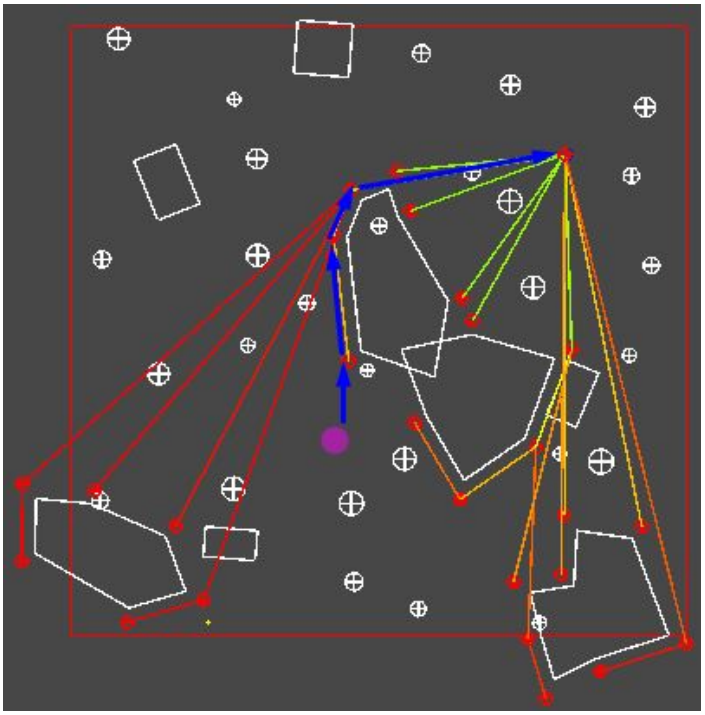


Figura 3.27 – A heurística calculada para o ponto rosa neste exemplo equivale ao tamanho de todo percurso indicado em azul.

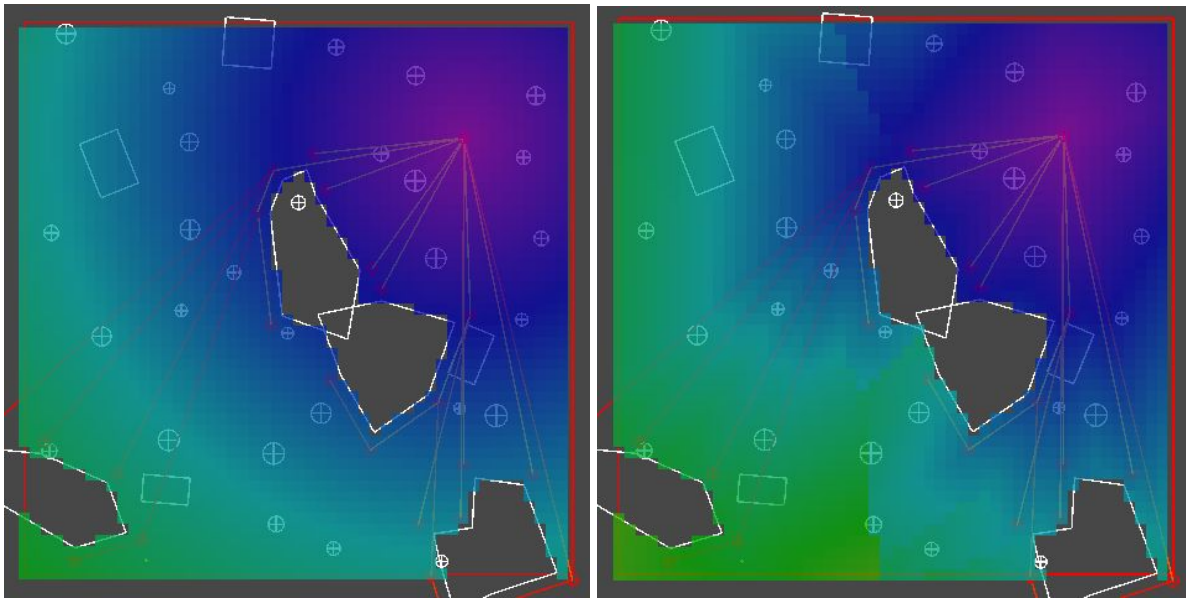


Figura 3.28 – À esquerda o resultado de uma heurística de distância euclidiana. As regiões em rosa têm o menor valor de heurística enquanto as regiões em verde têm o maior valor de heurística. À direita o resultado da heurística quando utilizado grafo de visibilidade. Veja que a região entre os dois polígonos é verde, o que significa que é menos provável que a área de busca vá explorar aquela região.

Com isso conseguimos contornar os problemas causados por grandes obstáculos, como descrito na introdução deste trabalho. O mesmo é exemplificado na figura 3.29.

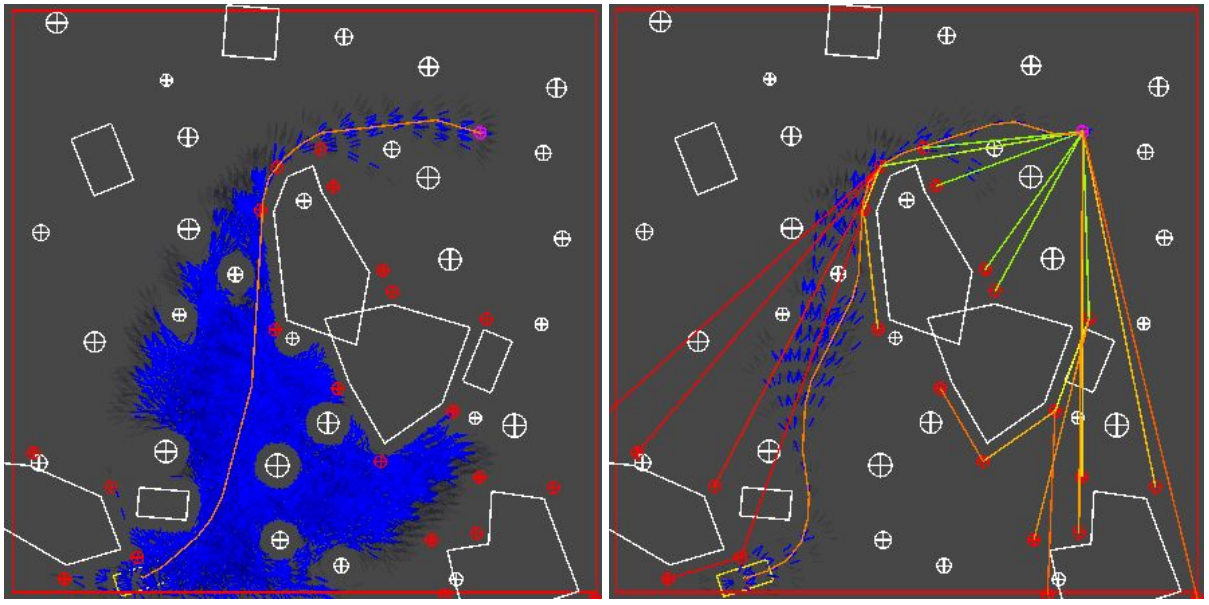


Figura 3.29 – À esquerda a expansão da árvore de busca do algoritmo A* híbrido implementado quando utilizada uma heurística de distância euclidiana (região em azul). À direita a redução significativa na árvore de busca, quando a heurística se baseia no grafo de visibilidade.

4 RESULTADOS

Todas técnicas de otimização desenvolvidas foram avaliadas em 20 diferentes cenários de teste. Seis cenários contendo uma distribuição balanceada de polígonos, retângulos e círculos. Nove cenários contendo somente um tipo de obstáculo (três por cada tipo), e cinco cenários que representam mais aproximadamente uma situação comum encontrada no simulador SIS-ASTROS, onde a região contém árvores (círculos), alguns outros veículos (retângulos), e um ou dois lagos (polígonos) de maior escala, como ilustra a figura 4.1. Para cada cenário foram calculados cinco caminhos diferentes (cinco diferentes combinações de ponto de partida do caminho e ponto de chegada), totalizando 100 situações distintas.

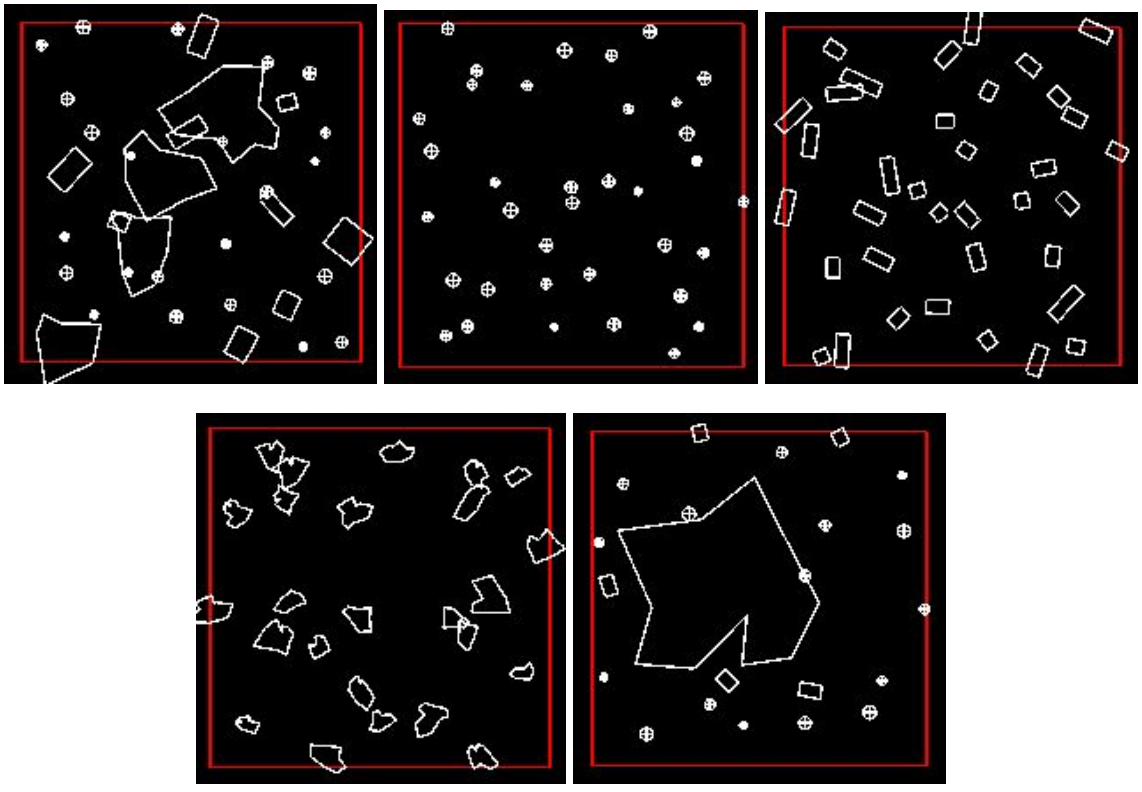


Figura 4.1 – Cinco exemplos de cenários utilizados para testar o desempenho do algoritmo implementado.

Inicialmente foram coletados dados de desempenho quando se utiliza parametrizações diferentes da grid discreta utilizada pelo algoritmo A* Híbrido. Os parâmetros testados estão presentes na tabela 4.1.

	Tamanho da célula (metros)	**Número de ângulos indexados por célula**	**Tempo de execução médio (ms)**	**Custo médio dos caminhos (metros)**
Parametrização A	0.25	8	322.5	41.95
Parametrização B	0.25	4	185.8	42
Parametrização C	0.5	16	171.94	42.14
Parametrização D	1.3	16	46.17	42.86
Parametrização E	2	24	25.76	43.62
Parametrização F	3	32	41.78	48.70

Tabela 4.1 – Resultados de desempenho das diferentes parametrizações testadas. Médias abrangem todos testes executados.

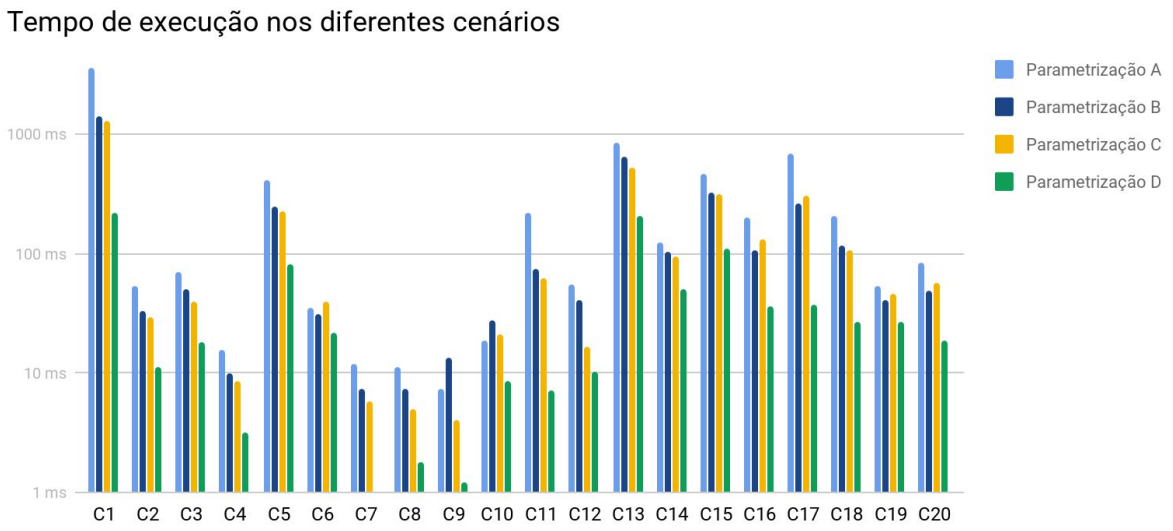
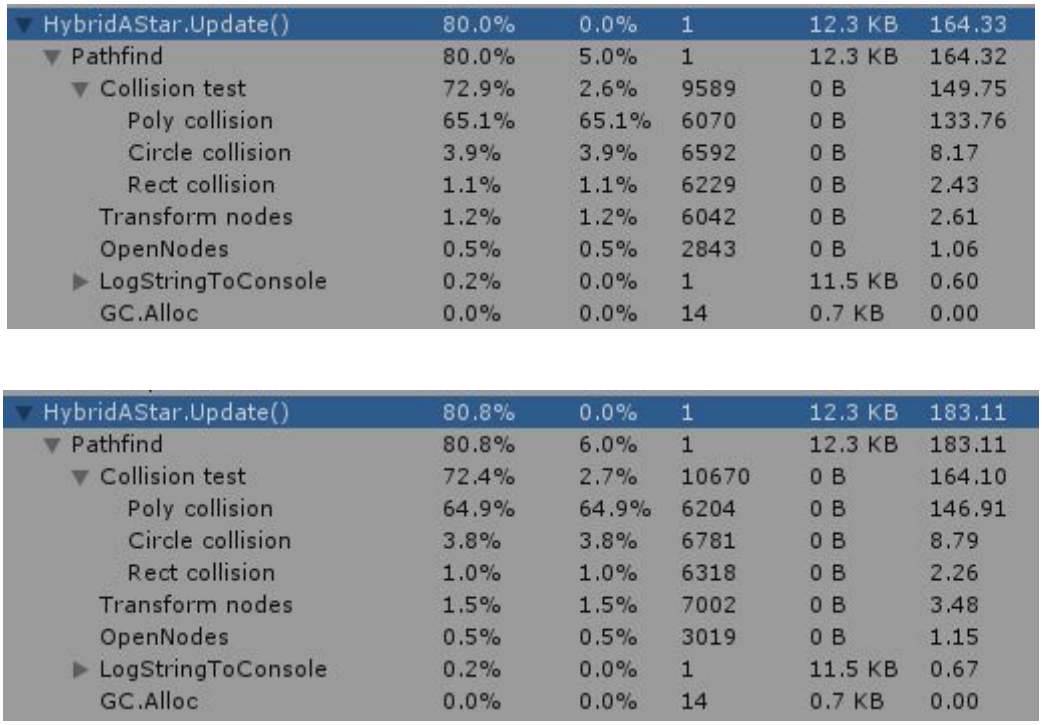


Gráfico 4.1 – Resultados de desempenho de cada cenário de teste, para cada parametrização testada. A parametrização D apresentou o menor tempo de execução em todos cenários.

Foi observado a possibilidade de redução significativa do tempo de execução do algoritmo quando escolhidos os parâmetros corretos para a grid discreta. A parametrização D teve um tempo de execução médio aproximadamente 7 vezes menor quando comparada a parametrização A, enquanto o custo do caminho (distância percorrida em metros) foi apenas 2% maior. Porém, como visto na parametrização F, quando se utiliza um tamanho de célula muito grande o algoritmo passa a perder desempenho. Além disto esta parametrização falhou na busca de caminhos em 3 dos 20 cenários de teste. Para os próximo testes foram utilizados

valores intermediários entre a parametrização C e D: tamanho de célula 0.8 metros e 16 ângulos indexados por célula.



HybridAStar.Update()	80.0%	0.0%	1	12.3 KB	164.33
Pathfind	80.0%	5.0%	1	12.3 KB	164.32
Collision test	72.9%	2.6%	9589	0 B	149.75
Poly collision	65.1%	65.1%	6070	0 B	133.76
Circle collision	3.9%	3.9%	6592	0 B	8.17
Rect collision	1.1%	1.1%	6229	0 B	2.43
Transform nodes	1.2%	1.2%	6042	0 B	2.61
OpenNodes	0.5%	0.5%	2843	0 B	1.06
LogStringToConsole	0.2%	0.0%	1	11.5 KB	0.60
GC.Alloc	0.0%	0.0%	14	0.7 KB	0.00

HybridAStar.Update()	80.8%	0.0%	1	12.3 KB	183.11
Pathfind	80.8%	6.0%	1	12.3 KB	183.11
Collision test	72.4%	2.7%	10670	0 B	164.10
Poly collision	64.9%	64.9%	6204	0 B	146.91
Circle collision	3.8%	3.8%	6781	0 B	8.79
Rect collision	1.0%	1.0%	6318	0 B	2.26
Transform nodes	1.5%	1.5%	7002	0 B	3.48
OpenNodes	0.5%	0.5%	3019	0 B	1.15
LogStringToConsole	0.2%	0.0%	1	11.5 KB	0.67
GC.Alloc	0.0%	0.0%	14	0.7 KB	0.00

Figura 4.2 – Dados de colisão em dois cenários diferentes. A coluna mais à direita são os tempos de execução de cada porção do código. Em ambos os casos os testes de colisão ponto x polígono (método *crossing number*) ocuparam em torno de 80% do tempo de execução do algoritmo de busca.

Foi observado, utilizando as ferramentas de análise de desempenho presentes no motor gráfico Unity (figura 4.2), que os cálculos de colisão, especialmente com polígonos, ocuparam a maior parte do tempo de execução do algoritmo de busca. Colisão com polígonos ocupou em torno de 80% do tempo de execução. Outros tipo de colisão ocuparam em torno de 6% do tempo de execução, sobrando 14% para as outras partes do algoritmo, como expansão de nós, transformações baseados no raio de giro do veículo e manutenção das lista de nós abertos e fechados.

Quando se compara o uso do método *crossing number* para intersecção de pontos com polígonos, e o método desenvolvido neste trabalho, obtivemos os seguintes resultados (gráfico 4.2).

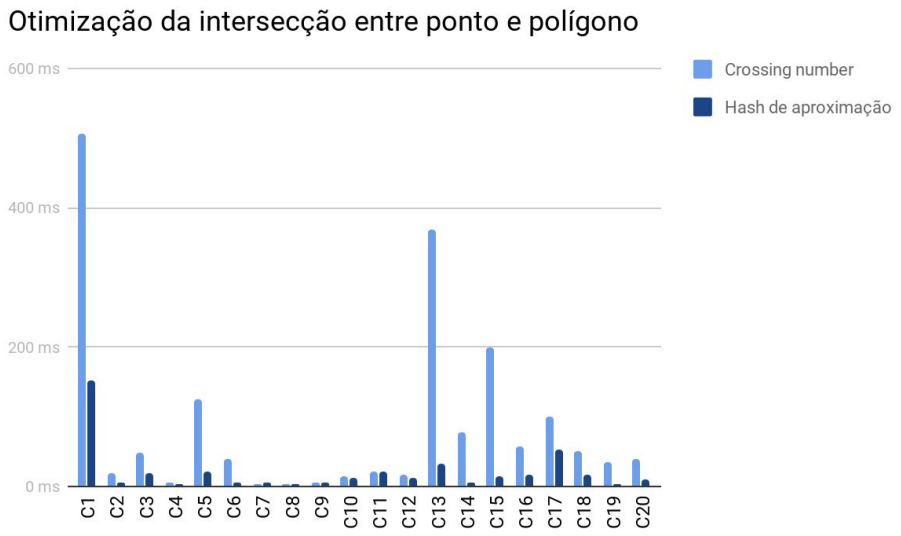


Gráfico 4.2 – Resultados de desempenho de cada cenário de teste, quando utilizado o método *crossing number* e o método baseado em hash desenvolvido no trabalho

Utilizando o método de crossing number obtivemos uma média de tempo de execução de 85.85 ms, enquanto a utilização do método baseado em hash desenvolvido neste trabalho resultou em um média de 20.97 ms, para todas situações de cálculo de caminho testados.

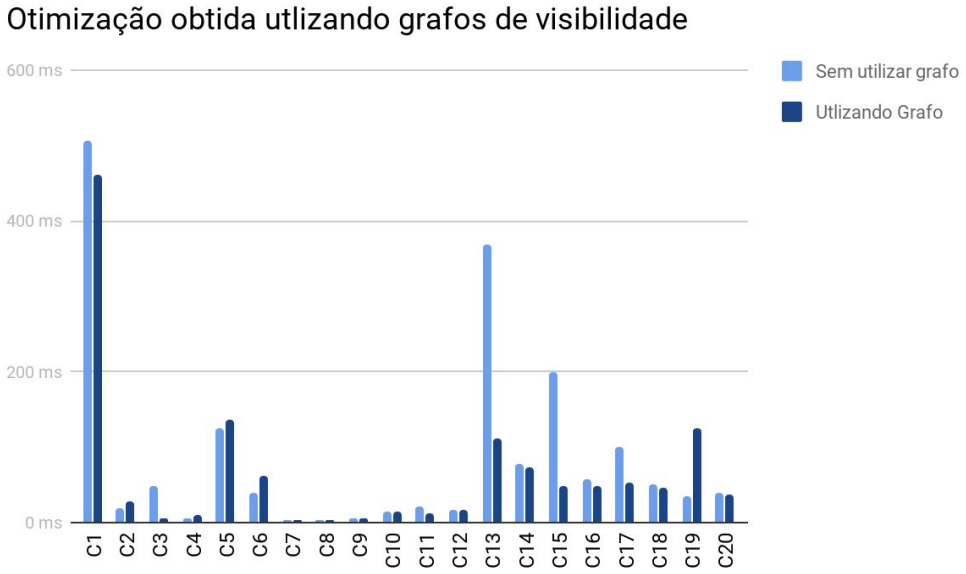


Gráfico 4.3 – Resultados de desempenho de cada cenário de teste, quando não é utilizado grafo de visibilidade para aprimorar a heurística do algoritmo A* híbrido, e quando é utilizado.

A utilização de grafo de visibilidade resultou em uma média de tempo de execução de 66 ms entre todos as situações testadas. Isto representa redução de 20% no tempo de execução necessário para executar as buscas de caminho, quando comparado com a não utilização do grafo de visibilidade. O tempo necessário para gerar estes grafos foi irrisório, tendo uma média em torno de 1 ms ou menos.

Porém em alguns cenários o desempenho piorou, o que indica a necessidade de um aprimoramento na seleção dos obstáculos que geram o grafo. Um aprimoramento possível seria a inclusão do ponto inicial do caminho como um nó no grafo. Isto pode corrigir um problema que ocorre quando as arestas do grafo de visibilidade gerado estão afastados da região onde está o caminho ótimo, fazendo com que o caminho resultante se aproxime do grafo sem necessidade (figura 4.3).

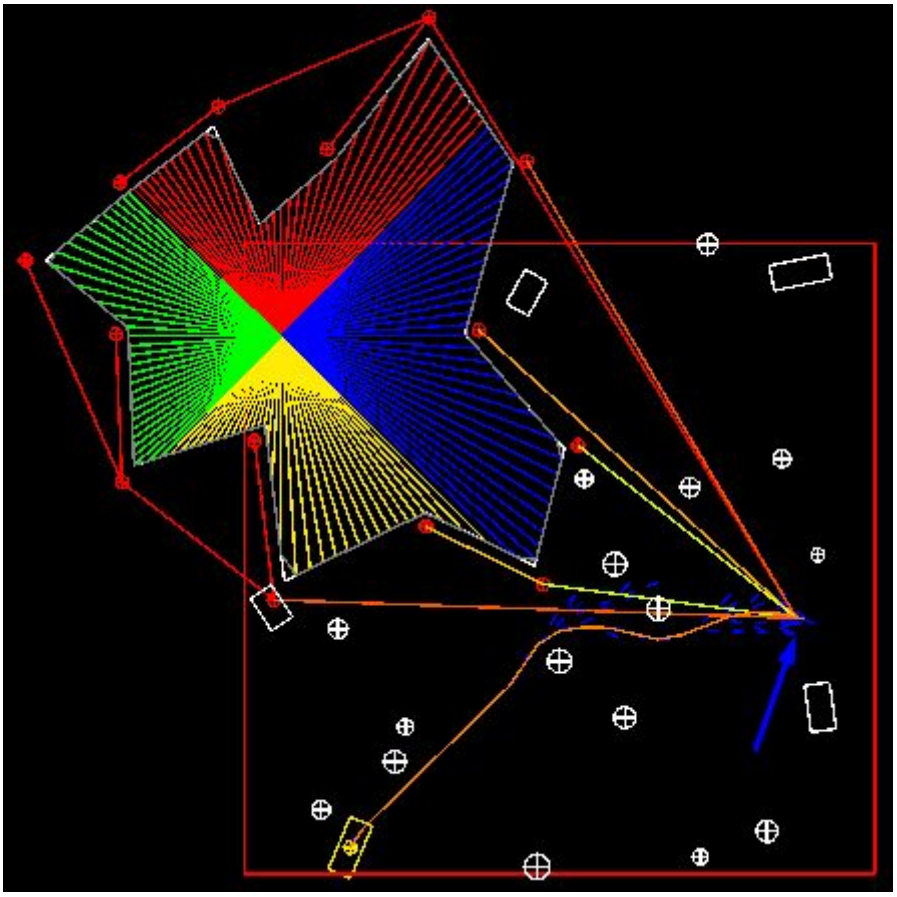


Figura 4.3 – Exemplo de caso onde um grande polígono causa a geração de um grafo de visibilidade longe do caminho ótimo entre a posição inicial (retângulo amarelo) e o ponto final (indicado pela seta azul). O caminho (em laranja, partindo do ponto inicial do veículo) não representa o caminho o ótimo.

5 CONCLUSÕES

Neste trabalho foi desenvolvida uma implementação do algoritmo A* híbrido, aplicado a busca de caminhos para veículos em ambientes virtuais, para então avaliar técnicas de otimização, visando a redução no tempo de execução do algoritmo para utilizá-lo em aplicações em tempo real, como as situações de simulação de veículos em comboio presentes no simulador SIS-ASTROS, projeto em desenvolvimento como colaboração entre o exército brasileiro e a Universidade Federal de Santa Maria.

Foi desenvolvido um software de testes implementado em C# utilizando o motor gráfico Unity. Este trabalho contribuiu com o desenvolvimento de duas técnicas de otimização que podem ser utilizadas para acelerar a execução do algoritmo A* Híbrido.

A primeira técnica desenvolvida consiste na conversão de polígonos quaisquer para uma representação baseada em hash. Utilizando esta representação em hash os cálculos de intersecção entre ponto e polígono podem ser acelerados, quando comparados com outro método, já presente na literatura, utilizado no trabalho: *crossing number. O* método *crossing number* possui complexidade O(n) enquanto a intersecção com um polígono na representação em hash possui complexidade O(1).

A segunda técnica desenvolvida consiste na construção de um grafo de visibilidade em torno dos maiores obstáculos do cenário, e a utilização do algoritmo de Dijkstra para mapear os menores caminhos entre os pontos do grafo e um ponto alvo do caminho sendo calculado. Este grafo pode então ser utilizado para calcular uma melhor heurística para o algoritmo A* Híbrido, quando comparado com o uso de distância euclidiana.

As técnicas tiveram seu desempenho avaliado em 100 situações de cálculo de caminho diferentes. Ambas as técnicas apresentaram resultados promissores, no que tange a otimização do tempo de execução do algoritmo. A qualidade do caminho resultante é afetada pelo algoritmo, mas em um nível baixo suficiente para utilização em aplicações onde o caminho ótimo não é necessário.

Para trabalhos futuros, podem ser avaliadas as situações específicas onde cada técnica obteve deterioração no tempo de execução. A hash de representação de polígonos pode ser aprimorada para selecionar uma melhor subdivisão dos polígonos originais, e economizar ainda mais tempo de execução.

REFERÊNCIAS

ALGFOOR, Zeyad Abd; SUNAR, Mohd Shahrizal; KOLIVAND, Hoshang. A comprehensive study on pathfinding techniques for robotics and video games. **International Journal of Computer Games Technology**, v. 2015, p. 7, 2015.

VAN VERTH, Jim et al. Formation-based pathfinding with real-world vehicles. In: **Game Developers Conference**. 2000.

DO NASCIMENTO, Bruno Torres; FRANZIN, Flavio Paulus; POZZER, Cesar Tadeu. GPU-Based Real-Time Procedural Distribution of Vegetation on Large-Scale Virtual Terrains. In: **2018 17th Brazilian Symposium on Computer Games and Digital Entertainment (SBGames)**. IEEE, 2018. p. 157-15709.

PETEREIT, Janko et al. Application of Hybrid A* to an autonomous mobile robot for path planning in unstructured outdoor environments. In: **ROBOTIK 2012; 7th German Conference on Robotics**. VDE, 2012. p. 1-6.

HART, Peter E.; NILSSON, Nils J.; RAPHAEL, Bertram. A formal basis for the heuristic determination of minimum cost paths. **IEEE transactions on Systems Science and Cybernetics**, v. 4, n. 2, p. 100-107, 1968.

DIJKSTRA, Edsger W. A note on two problems in connexion with graphs. **Numerische mathematik**, v. 1, n. 1, p. 269-271, 1959.

MONTEMERLO, Michael et al. Junior: The stanford entry in the urban challenge. **Journal of field Robotics**, v. 25, n. 9, p. 569-597, 2008.

O'ROURKE, Joseph. Finding minimal enclosing boxes. **International journal of computer & information sciences**, v. 14, n. 3, p. 183-199, 1985.

SHAH, Brual C.; GUPTA, Satyandra K. Speeding up A* search on visibility graphs defined over quadtrees to enable long distance path planning for unmanned surface vehicles. In: **Twenty-Sixth International Conference on Automated Planning and Scheduling**. 2016.

DOLGOV, Dmitri et al. Practical search techniques in path planning for autonomous driving. **Ann Arbor**, v. 1001, n. 48105, p. 18-80, 2008.

SAKAI, Atsushi. A path planning algorithm based on Hybrid A* for trailer truck https://atsushisakai.github.io/HybridAStarTrailer/

POZZER, Cesar Tadeu; DE LARA PAHINS, Cícero A.; HELDAL, Ilona. A hash table construction algorithm for spatial hashing based on linear memory. In: **Proceedings of the 11th Conference on Advances in Computer Entertainment Technology**. ACM, 2014. p. 35.