CENTRO UNIVERSITÁRIO FEI

LEONARDO DA SILVA COSTA

IMPLEMENTAÇÃO E COMPARAÇÃO DE ALGORITMOS DE *PATH PLANNING* PARA FUTEBOL DE ROBÔS

São Bernardo do Campo

2020

LEONARDO DA SILVA COSTA

IMPLEMENTAÇÃO E COMPARAÇÃO DE ALGORITMOS DE *PATH PLANNING* PARA FUTEBOL DE ROBÔS

Relatório final de Iniciação Científica apresentado ao Centro Universitário FEI, como parte dos requisitos do Programa PIBIC-FEI. Orientado pelo Prof. Dr. Flavio Tonidandel

São Bernardo do Campo

2020

RESUMO

Este projeto propõe a implementação e comparação de dois algoritmos de *path planning* afim de melhorar o cálculo de trajetória dos robôs da RoboCup na categoria *Small Size League* (SSL) de futebol de robôs. Foram estudados os algoritmos A Estrela (A*) e *Rapidly Random exploring Trees* (RRT), pois são os mais comuns dentre as equipes participantes desta competição. A utilização de um bom algoritmo de cálculo de trajetória é de extrema importância para a equipe, já que a colisão entre dois robôs durante o jogo pode gerar penas graves para o robô infrator. Além disso, também é necessário que o algoritmo utilizado seja rápido o suficiente, devido à alta dinâmica do jogo, para encontrar uma solução aceitável. O algoritmo escolhido foi o A*, pois ele é capaz de encontrar caminhos em tempos inferiores à $3ms$ na maioria dos casos, os caminhos encontrados também serão sempre os menores possíveis e devido à sua simplicidade ele pode ser alterado facilmente, por exemplo, caso seja necessário é possível fazer com que o algoritmo encontre somente uma parte do caminho total até o destino, isso pode ser útil para algum caso em que o destino está sendo totalmente bloqueado por algum obstáculo.

Palavras-chave: *Path planning*. Futebol de robôs. *A star*. *Rapidly Random exploring Trees*. RRT. A*.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

A categoria *Small Size League* (SSL) do futebol de robôs da *RoboCup* [1]é uma das mais dinâmicas de toda a competição devido aos passes e chutes rápidos. A velocidade da bola pode chegar a até $6{,}5m/s$, além disso, também existem os robôs do time adversário, que atualmente são 6 robôs, sendo um deles o goleiro, que podem movimentar-se livremente pelo campo com uma velocidade de aproximadamente $2m/s$. Para conseguir manter toda essa velocidade de jogo, os robôs também devem conseguir movimentar-se rapidamente pelo campo desviando de obstáculos visando o menor número possível de colisões. De acordo com uma mudança recente nas regras qualquer colisão cuja a diferença de velocidade dos robôs seja maior que $1{,}5m/s$, o robô com a maior velocidade receberá uma falta, que, de acordo com o site das regras da SSL (RULES, 2011), gera a cobrança de um *Direct Free Kick*, que é onde um robô oponente pode chutar diretamente para o gol, o que é algo perigoso e deve ser evitado ao máximo.

Para isso, além de ser necessário um bom controle do robô, também é preciso um bom método para calcular sua trajetória. Existem diversos algoritmos que podem ser empregados, e alguns deles serão abordados neste estudo.

1.1 OBJETIVO

Este estudo tem como objetivo implementar e analisar dois algoritmos de cálculo de trajetória, sendo eles o A Estrela (A*), cuja primeira aparição pode ser vista em Hart, Nilsson e Raphael (1968), e o *Rapidly Random exploring Trees* (RRT), que foi descrito pela primeira vez em LaValle (1998), para que seja escolhido o melhor em termos de velocidade de processamento, uso de memória e segurança do trajeto, que por fim será utilizado pela equipe ROBOFEI da categoria *RoboCup Small-Size*.

1.2 JUSTIFICATIVA

O algoritmo atual utilizado pela equipe é de difícil compreensão e ao realizar testes com os robôs fica nítido que ele também não é eficaz como deveria ser. Outro aspecto a ser analisado é que possuir um bom método de cálculo de trajetórias influencia em diversas habilidades que o robô possui como interceptar bolas, posicionar-se para fazer um gol e também conseguir voltar rapidamente para a defesa. Além disso, auxilia na possibilidade de jogadas novas e com certo nível de complexidade que exijam um posicionamento rápido e preciso dos robôs.

O novo algoritmo também será implementado de forma completamente modular, ou seja, para trocar para algum outro algoritmo ou então realizar manutenções, não será necessário fazer mudanças drásticas no código de estratégia.

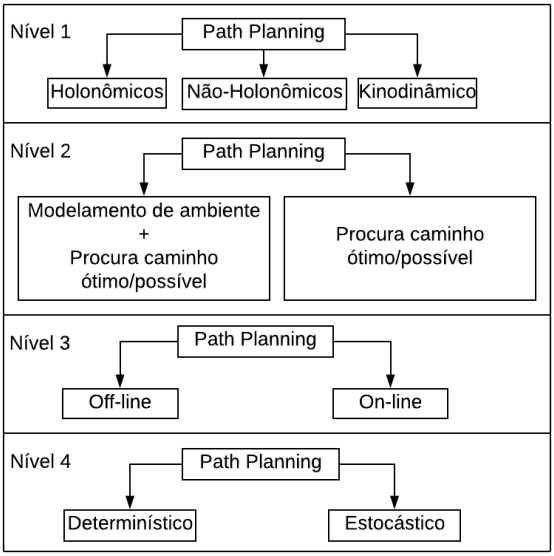
[1]Competição em nível mundial que se desenrola todos os anos. Visa o estudo e desenvolvimento da Inteligência Artificial (IA) e da Robótica

2 REVISÃO BIBLIOGRÁFICA

2.1 PATH PLANNING – PLANEJAMENTO DE TRAJETÓRIA

O termo *path planning* foi desenvolvido em diversas áreas, como na robótica, inteligência artificial e teoria de controle. Na robótica, *path planning* tem a ver com problemas onde é necessário achar um caminho otimizado para sair de um ponto inicial e chegar a um ponto final, esse caminho otimizado pode ser interpretado de diversas maneiras, pode ser que ele possua o menor comprimento do ponto inicial até o final, ou então evitar qualquer tipo de colisão, evitar paradas constantes ou que o robô tenha que corrigir sua direção a todo momento. Desde os anos 60 foram desenvolvidos diversos tipos diferentes de algoritmos para cálculo de trajetória, cada um deles com características especiais que ajudam a distinguir uns dos outros e também a decidir qual algoritmo utilizar baseado no tipo de problema enfrentado, é possível dividir essas características em 4 níveis como mostra a Figura 1 (SOUISSI et al., 2013).

Figura 1 – Classificação dos tipos de path-planners.



Fonte: Adaptado de Souissi et al. (2013).

No primeiro nível têm-se três categorias:

- Holonômicos: Em robótica, um sistema é considerado como sendo holonômico quando há controle de todos seus graus de liberdade, como num braço robótico, ou então os robôs utilizados na categoria *Small Size League*(SSL) que utilizam rodas omnidirecionais.
- Não-Holonômicos: É um sistema onde não há controle de todos os graus de liberdade, como num automóvel.
- Kinodinãmico ou *Kinodynamic*: De acordo com Souissi et al. (2013), é um termo criado para designar dois tipos de restrições: cinemáticas e dinâmicas. Desviar

de obstáculos e limitações de velocidade são, respectivamente, exemplos de restrições cinemáticas e dinâmicas.

No segundo nível têm-se as diferentes maneiras que cada algoritmo utiliza para chegar a um resultado. No caso do A* (HART; NILSSON; RAPHAEL, 1968) e suas variações é sempre necessário possuir uma representação do ambiente, já no caso do RRT (LAVALLE, 1998) isto já não é necessário.

No terceiro nível são separados os algoritmos que podem ser utilizados *on-line* e outros que somente podem ser utilizados *off-line*. Para este trabalho deve-se utilizar um algoritmo cuja execução possa ser feita de forma *on-line*, embora o A* necessite de uma certa modelagem do ambiente ele ainda pode encaixar-se nessa categoria.

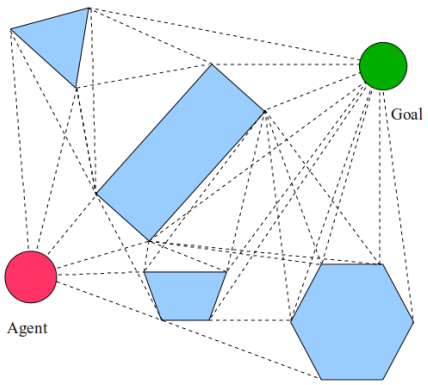
E no último nível estão os algoritmos determinísticos, são aqueles que calculam o caminho de forma matemática, como é o caso do A*. Estes algoritmos sempre irão calcular o mesmo caminho dadas as mesmas condições iniciais e finais, e existem os estocásticos ou probabilísticos, que são aqueles que descobrem o caminho de forma aleatória, como é o caso do RRT ou então algoritmos baseados em *Particle Swarm Optimization*(PSO) (SASKA et al., 2006).

2.1.1 Modelagem de ambiente

Como será utilizado o A*, que é um algoritmo que necessita de uma representação do ambiente, aqui serão apresentadas duas formas de representar um ambiente, essa representação também é chamada de espaço de estados, que é onde têm-se todas as posições e orientações possíveis do robô.

O primeiro método são os grafos de visibilidade, de acordo com Souissi et al. (2013), o princípio do método de grafo de visibilidade é modelar o ambiente ligando os vértices de todos os obstáculos entre si próprios e também com o ponto final e inicial, utilizando este método em conjunto com um algoritmo da família A* sempre obtém-se o caminho mais curto possível. É possível ver como é construído um grafo de visibilidade na Figura 2.

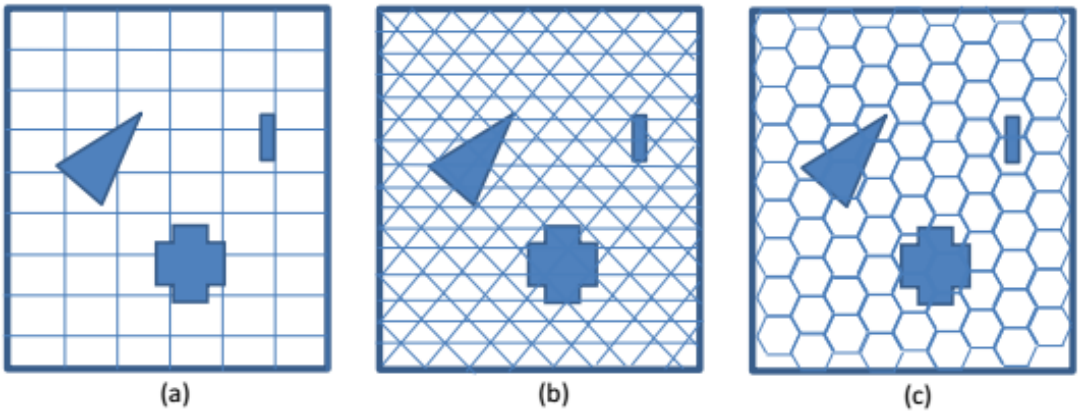
Figura 2 – Exemplo de grafo de visibilidade.



Fonte: Site da internet[2].

O segundo método são as representações em grade regular. Neste tipo de representação o ambiente é dividido em células que podem ser quadradas, Fig. 3-a, triangulares, Fig. 3-b ou hexagonais, Fig. 3-c. É o método mais simples e também muito utilizado em jogos.

Figura 3 – Tipos de grades regulares.



Fonte:Souissi et al. (2013).

Uma desvantagem em relação aos grafos de visibilidade é que ao aplicar um algoritmo da família A* que não faça busca em ângulo como o *Theta Star*(Theta*) não será encontrado o caminho mais curto. Em Nash e Koenig (2013) há um estudo que mostra as diferenças no comprimento do caminho mais curto calculado em uma grade e o caminho mais curto real.

Neste trabalho foi utilizada uma representação em grade regular com células quadradas com 8 vizinhos cada. Para fazer isto, foi usada uma matriz de duas dimensões cujo tamanho dependerá das dimensões do campo. Como o campo a ser usado neste trabalho possui $9000x6000mm$, fica inviável criar uma matriz com grandes dimensões utilizando uma resolução de $1mm$, isso faria com que fossem usados cerca de $0{,}5GB$ de armazenamento. Para contornar este problema a matriz foi criada numa escala reduzida do campo de forma que não haja um impacto significativo na precisão dos caminhos gerados. Essa redução foi feita da seguinte maneira, as dimensões do campo e de todos os objetos, como os robôs e a bola, serão divididos por um certo fator de escala que será determinado pelo software. Por exemplo, para um campo de $9000x6000mm$ e um fator de escala de 50 têm-se uma matriz de $180x120$ células e os robôs que possuem $180mm$ de diâmetro irão ocupar 4 células de diâmetro, ou seja, 16 células. Na construção da matriz também foi adicionada uma distância de *offset* em ambos os lados do campo, isso fará com que mesmo que o robô saia dos limites do campo ainda seja possível atribuir a ele uma posição na matriz e, portanto, calcular um caminho.

[2]Disponível em: <https://ubm.io/2G54Bup> Acesso em: 4 fev. 2018 as 17:00.

2.1.2 A* - A Estrela

Dentre os algoritmos de *path planning*, o A Estrela (A*) é o mais conhecido. Ele foi descrito pela primeira vez em 1968 no Instituto de pesquisa de Stanford por Peter Hart, Nils Nilsson e Bertram Raphael (GUPTA; UMRAO; KUMAR, s.d.), e desde então é um dos mais utilizados.

O que o torna uma boa escolha para certas áreas, como a indústria aviónica, é o fato dele ser probabilisticamente completo ou admissível, de acordo com Hart, Nilsson e Raphael (1968) isso significa que é garantido que o algoritmo irá encontrar um caminho ótimo. No caso do A* um caminho é considerado ótimo quando ele possui o menor comprimento possível. Outra vantagem é devido ao fato dele ser determinístico.

Ele é a base de muitos outros algoritmos utilizados atualmente, como o D* (STENTZ, 1994) e também existem diversos artigos que propõem otimizações utilizando outros métodos como o *Jump Point Search* (JPS) (DUCHOŇ et al., 2014).

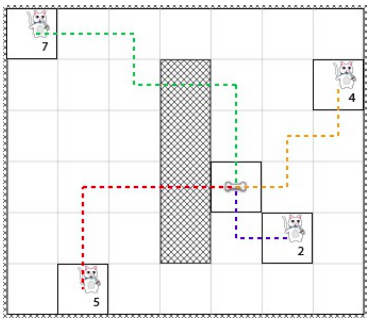
No A* todas as células são analisadas a partir da Equação 1.

$$f(v) = h(v) + g(v) \tag{1}$$

Onde $h(v)$ é a heurística, que pode ser a distância Euclidiana ou Manhattan da célula inicial até o destino, $g(v)$ é a somatória dos custos do caminho do estado inicial até o estado atual através da sequência selecionada de células. A célula com o menor valor de $f(v)$ será selecionada como sucessora na sequência. Uma boa característica deste algoritmo é que outras funções podem ser adicionadas para o cálculo de $f(v)$, por exemplo, tempo ou chance de colisão, também é preciso guardar em cada célula quem foi que a gerou, ou seja, a célula antecessora à ela, para que seja possível extrair o caminho final ao término do algoritmo, no Apêndice A têm-se um fluxograma que representa a execução do A*.

A seguir têm-se um exemplo ilustrado de como funciona o A*. Nesse exemplo a heurística $h(v)$ utilizada foi a distância Manhattan, para calculá-la é simples, basta contar o número de células entre o ponto inicial e o final ignorando os obstáculos, é possível ver isso na Figura 4 onde a distância de cada trajeto está escrita no canto direito inferior de cada célula inicial.

Figura 4 – Exemplo de cálculo da heurística.

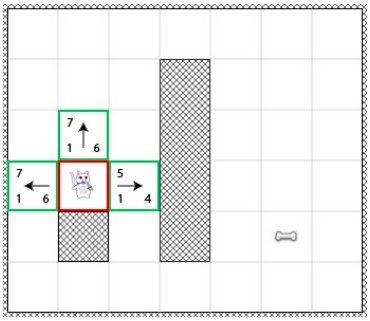


Fonte: Site da Internet [3].

Nas imagens a seguir o valor de $f(v)$ estará no canto esquerdo superior da célula, $h(v)$ estará no canto direito inferior e $g(v)$ no canto esquerdo inferior.

No primeiro passo (Figura 5) , as células adjacentes do ponto inicial são colocadas na *open list* e seus respectivos valores de $f(v)$ são calculados, a *open list* é basicamente uma lista onde estão as células que podem ser utilizadas para construir um trajeto, essa lista deve ser implementada, preferencialmente, como uma fila de prioridade ou *priority queue*, e também existe a *closed list* onde fica guardado quais células já foram avaliadas, as células que estão na *open list* estão com a borda verde e as que estão na *closed list* estão com a borda vermelha.

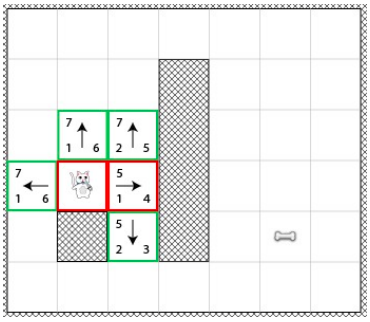
Figura 5 – Primeiro passo do algoritmo.



Fonte: Site da Internet [3].

No segundo passo (Figura 6), é escolhida a célula com menor valor de $f(v)$ e esta será considerada como sucessora na sequência, isso se repetirá até que seja encontrado um trajeto até a célula final.

Figura 6 – Segundo passo do algoritmo.



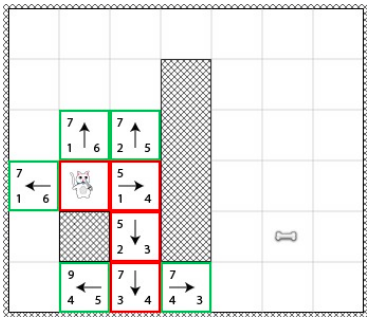
Fonte: Site da Internet [3].

Na Figura 7 ocorre um empate entre os valores de $f(v)$ de algumas células, neste caso uma solução simples é que o algoritmo escolha a célula cujo valor de $f(v)$ é o mais recente.

E então finalmente o algoritmo chega à célula final e a coloca na *open list* (Figura 8), quando isso ocorre o trajeto foi encontrado e basta seguir o caminho inverso para montá-lo, o trajeto final encontrado está ilustrado na Figura 9.

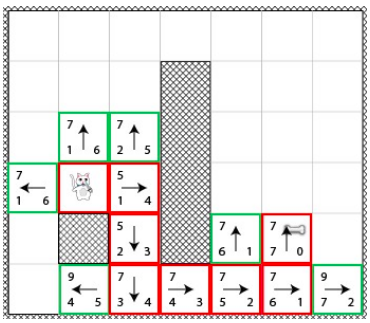
[3]Disponível em: <https://bit.ly/2Ludcx6> Acesso em: 8 ago. 2017 as 18:23.
[4]Disponível em: <https://bit.ly/2Ludcx6> Acesso em: 8 ago. 2017 as 18:23.

Figura 7 – Exemplo de empate nos valores de $f(v)$.



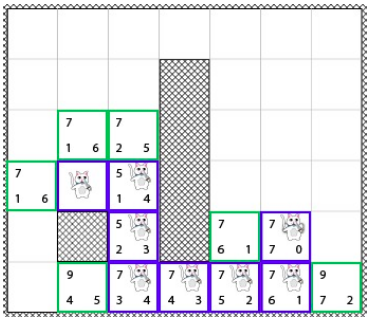
Fonte: Site da Internet [4].

Figura 8 – Exemplo da célula final sendo encontrada.



Fonte: Site da Internet [4].

Figura 9 – Trajeto final estabelecido.



Fonte: Site da Internet [4].

2.1.3 D*-Lite – D Star Lite

Um grande problema do A* são os obstáculos, ao encontrar algum isso pode fazer com que o algoritmo volte para o ponto inicial para recalcular o trajeto, o que leva a um tempo de processamento maior.

Para ambientes dinâmicos onde os obstáculos aparecem e se movimentam rapidamente o A* não é um algoritmo tão otimizado, o D*-*Lite* descrito em Koenig e Likhachev (2002), que é uma modificação do A*, seria uma proposta mais eficiente para este caso.

O D*-*Lite* funciona de forma parecida com o A*, nele têm-se a Equação 2 que funciona como uma função de avaliação.

$$rhs(s) = g(s') + c(s,s') \tag{2}$$

Onde $g(s)$ é a função objetivo e é calculada em função de rhs, e $c(s,s')$ é o custo para ir de s a s', por exemplo, se o deslocamento for para cima/baixo ou direita/esquerda o custo será igual a 1, se ele for diagonal o custo será igual a $1{,}4$. Basicamente $rhs(s)$ é o custo, assim como $g(v)$ do A*, o rhs da posição sucessora s' sera calculado como sendo o valor mínimo do rhs de todos seus 8 vizinhos possíveis e então a posição que possuir menor valor de rhs será a sucessora. Vale lembrar que é necessário sempre guardar a informação sobre a partir de qual posição s que foi gerado o rhs de s'.

Ao contrário do A*, ao encontrar um obstáculo no momento de seguir o caminho o D*-*Lite* não irá simplesmente voltar à posição com o menor valor de rhs. O que acontece é que será recalculado o valor do rhs de todos os vizinhos do obstáculo e então será feito um novo trajeto a partir do local em que o obstáculo foi detectado.

Embora o D*-*Lite* possua uma boa performance ao ser aplicado em ambientes dinâmicos, devido à sua capacidade de reajustar somente uma parte do caminho gerado anteriormente, nos resultados apresentados em Sun, Koenig e Yeoh (2008) é possível ver claramente que o D*-*Lite* só é mais rápido que o A* quando se trata de um alvo estacionário, e nos casos onde têm-se um ambiente dinâmico e um alvo em movimento, que é o caso do SSL, o tempo de processamento do D*-Lite chega a ser pelo menos 20 vezes mais lento que do A*, por esse motivo ele não foi implementado.

2.1.4 RRT - Rapidly Random exploring Trees

O RRT (LAVALLE, 1998) é um algoritmo muito utilizado pelas equipes da categoria *RoboCup Small-Size*. De acordo com Bruce e Veloso (2002) um algoritmo simples de cálculo de trajetória utilizando RRT funciona da seguinte forma: começa com uma simples árvore com somente a configuração inicial, e então, é gerado um ponto aleatoriamente dentro dos limites do espaço utilizado, que para este trabalho será um ponto que esteja dentro do campo de futebol. Após isso procura-se o vértice da árvore mais próximo desse ponto aleatório e então estende-se esse vértice em direção ao vértice aleatório utilizando uma distância base pré-definida. Assim, a árvore é construída aleatoriamente, é possível ver o seu pseudo algoritmo na Figura 10 e a forma como a árvore é construída na Figura 11.

Como visto na Figura 10, o RRT possui cinco funções características, sendo elas:

- BUILD RRT: Esta função seria o *main* de todo o algoritmo, é nela que é feita a criação do primeiro vértice da árvore, conhecido como vértice raiz ou *root*, na Figura 11 este seria o vértice q_{init}, e também e onde fica o laço de repetição principal que varia de 1 até K, sendo K o número máximo de iterações desejadas, este valor

Figura 10 – Pseudo Algoritmo RRT.

```
BUILD_RRT(q_init)
1  T.init(q_init);
2  for k = 1 to K do
3      q_rand ← RANDOM_CONFIG();
4      EXTEND(T, q_rand);
5  Return T
```

```
EXTEND(T, q)
1  q_near ← NEAREST_NEIGHBOR(q, T);
2  if NEW_CONFIG(q, q_near, q_new) then
3      T.add_vertex(q_new);
4      T.add_edge(q_near, q_new);
5      if q_new = q then
6          Return Reached;
7      else
8          Return Advanced;
9  Return Trapped;
```

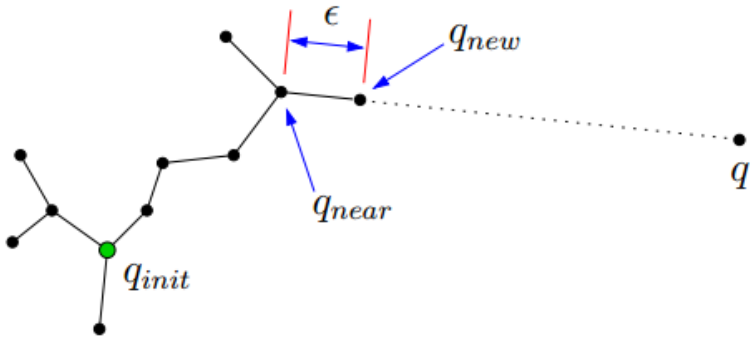
Fonte: Kuffner e LaValle (2000).

é definido a fim de evitar que o algoritmo gaste um grande tempo realizando uma busca.

- RANDOM CONFIG: Aqui é onde é criado um vértice aleatório no espaço de busca para que a árvore seja estendida em direção a ele. Na Figura 11 este seria o vértice q.
- EXTEND: É nesta função em que a árvore será expandida em direção a um vértice gerado aleatoriamente. Isso é feito da seguinte maneira: primeiro é preciso encontrar um vértice da árvore que seja mais próximo da árvore já existente, isso é feito através da função NEAREST NEIGHBOR, após isso será criado um novo vértice, na Figura 11 este vértice será o q_{new}, de modo que este novo vértice esteja na mesma direção do vértice aleatório e esteja à uma distância padrão, no caso da Figura 11 seria a distância ϵ, e então, caso não exista colisão com algum outro objeto este novo vértice é adicionado à árvore.
- NEAREST NEIGHBOR: É onde é realizada a busca para encontrar o vértice da árvore mais próximo do ponto aleatório gerado, na Figura 11 este seria o vértice q_{new}, essa busca pode ser feita de diversas formas, porém, em Bruce e Veloso (2002) é sugerido que seja utilizada uma K-d *Tree*, que é uma estrutura de dados que realiza este tipo de busca de uma forma mais eficaz, para que esta busca seja feita de forma mais rápida.
- NEW CONFIG: É onde é verificado se é possível estender a árvore em direção ao vértice q_{new} a partir do vértice q_{near}, basicamente é nesta função em que é analisado se o novo vértice gerado pode causar uma colisão ou não.

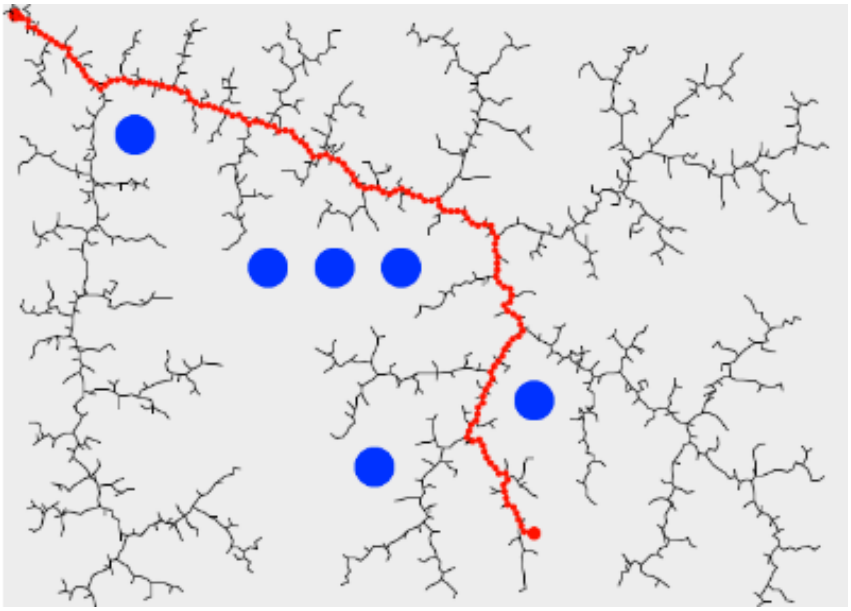
Na Figura 12 é possível ver uma ilustração de uma execução completa do RRT.

Figura 11 – Exemplo de construção da árvore.



Fonte: Kuffner e LaValle (2000).

Figura 12 – Um exemplo ilustrado de como funciona o algoritmo RRT.



Fonte: Página da internet [5].

Legenda: Os pontos azuis representam obstáculos, o caminho em vermelho é o trajeto final e os caminhos em preto são trajetos explorados pelo algoritmo.

2.1.5 Outras equipes

O *Team Description Paper* (TDP) e *Extended Team Description Paper* (ETDP) de outras equipes possuem informações úteis, por exemplo, em Zickler et al. (2010) nota-se que a equipe CMDragons utiliza um tipo modificado do algoritmo RRT que se chama ERRT, nesta modificação do RRT é feito o uso de *waypoints*, que são simplesmente os pontos que definem o caminho encontrado anteriormente, o algoritmo funciona de forma bem simples, com uma probabilidade p estende-se a árvore em direção ao destino, após isso, com uma probabilidade

[5]Disponível em: <https://bit.ly/2Nwc7lI> Acesso em: 31 jul. 2017 as 22:33.

r estende-se a árvore em direção à um *waypoint* aleatório, e por fim, com uma probabilidade $1 - p - r$ estende-se a árvore em direção à um ponto gerado aleatoriamente no espaço, mais detalhes podem ser vistos em Bruce e Veloso (2002).

De acordo com Poudeh et al. (2016), a equipe MRL utiliza um método um pouco diferente dos tradicionais como o A* ou RRT. O algoritmo utilizado por eles consiste em uma combinação de curvas simples como polinômios simples, senoidais ou linhas retas, onde é calculado se há algum obstáculo no caminho e também qual o melhor tipo de trajetória de acordo com alguns parâmetros.

Em Rodríguez et al. (2014) há um outro tipo de algoritmo de cálculo de trajetória proposto pela equipe STOx's, o algoritmo funciona de forma bem simples, nele, tenta-se recursivamente ligar o ponto inicial ao destino, caso exista algum obstáculo no caminho é adicionado um ponto intermediário para evitar este obstáculo.

2.1.6 grSim

Um dos métodos que foi utilizado para realizar os testes neste trabalho é o simulador grSim (MONAJJEMI; KOOCHAKZADEH; GHIDARY, 2011). O grSim é um simulador 3D desenvolvido pela equipe *Parsian Robotics Team*, nele é possível simular de forma integral um jogo oficial. O simulador funciona de forma bem simples, basta conectar-se ao endereço de *multicast* configurado e na respectiva porta, após feito isso já é possível receber os dados dele como se fosse um jogo real, ele também possui um outro endereço e porta onde é possível enviar comandos, é dessa forma que consegue-se movimentar os robôs, chutar e ligar o mecanismo de drible.

O simulador também conta com uma aba que possui diversas configurações, nela é possível ajustar os endereços de *multicast* e portas utilizadas, as dimensões do campo e da bola, é possível configurar até mesmo um ruído que faz com que os robôs sumam da visão, o que é algo comum nos jogos reais.

3 METODOLOGIA

Foi feita uma pesquisa sobre cada método de cálculo de trajetória e então a devida adaptação de cada um deles que estarão, na maioria dos casos, escritos em inglês e em forma de algoritmo, para a linguagem utilizada no software de estratégia, que no nosso caso é C++ na IDE do *QtCreator*.

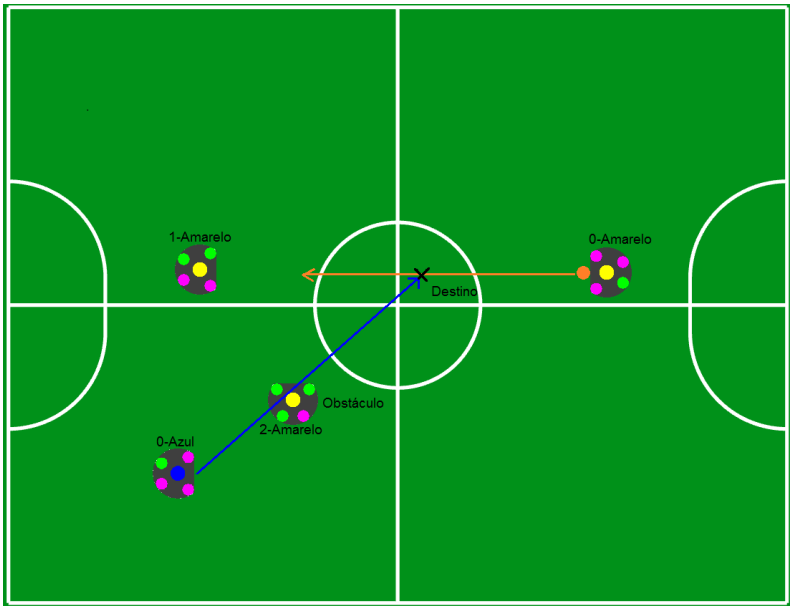
Após feita a implementação dos dois métodos, serão feitos diversos testes, por exemplo, simular situações onde um bom trajeto é crucial para uma boa execução da jogada, como, interceptar uma bola (ilustrado na Figura 13), posicionar-se para receber um passe (ilustrado na Figura 14) ou então voltar para a área de defesa (ilustrado na Figura 15).

Características importantes para serem avaliadas nos testes são: tempo de processamento, uso de memória, segurança e comprimento do trajeto.

Os testes serão feitos no grSim, e após isso serão realizados testes em campo utilizando os robôs reais no laboratório da RoboFEI.

Nos testes que serão realizados foi utilizado um notebook cujas especificações são: Intel® Core™ i7 com quatro núcleos de $2.8GHz$ e $16GB$ de RAM. O espaço de estados foi de $126x86$ células nos testes feitos no grSim, e $62x48$ células nos testes realizados em campo no laboratório. Para o A* pode-se utilizar a heurística Euclidiana e a de Manhattan, porém, nos testes feitos não foram percebidas mudanças significativas ao utilizar uma ou outra, por este motivo nos testes definitivos foi utilizada a heurística de Manhattan por ser mais simples.

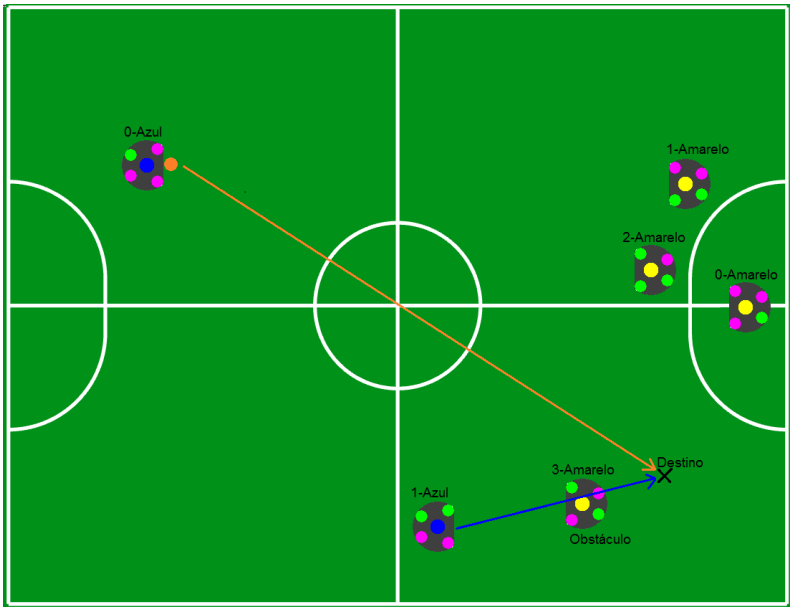
Figura 13 – Situação 1, interceptar bola.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura é possível observar uma situação onde o robô 0-Azul deve ir ao ponto destino, marcado com um X, a fim de interceptar um possível passe que irá ocorrer entre os robôs 0-Amarelo e 1-Amarelo, porém, ao seguir uma linha reta existe o robô 2-Amarelo como obstáculo, por isso, é necessário calcular uma trajetória para evitar uma colisão.

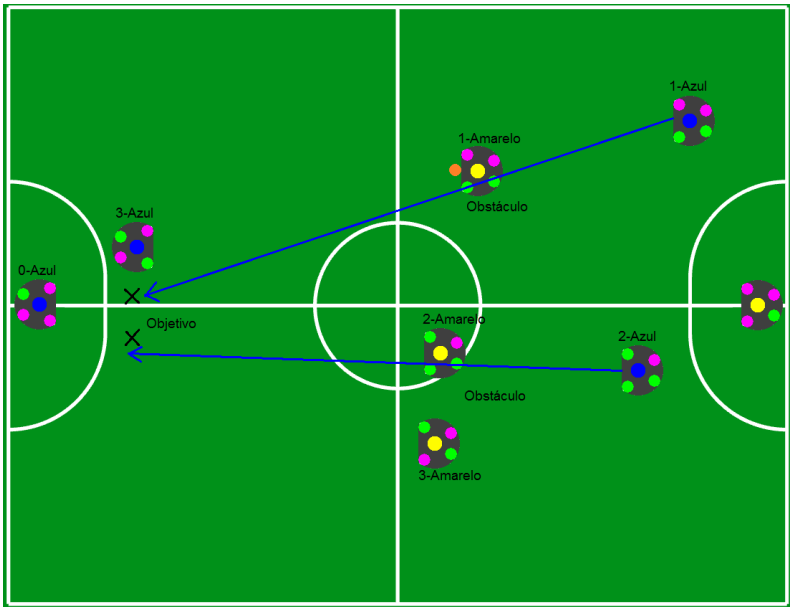
Figura 14 – Situação 2, receber passe.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura está ilustrada uma situação onde o robô 1-Azul deve receber um passe vindo do 0-Azul que deverá passar pelo ponto destino demarcado pelo X, mas, ao seguir diretamente para este ponto ocorreria uma colisão com o robô 3-Amarelo, assim, é preciso gerar uma trajetória que evite isto.

Figura 15 – Situação 3, voltar para área de defesa.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se a representação de uma situação onde os robôs 1-Azul e 2-Azul devem retornar para suas respectivas posições na área de defesa, demarcadas pelo X, entretanto, ao seguir diretamente para seus destinos haverão colisões com os robôs 1-Amarelo e 2-Amarelo, por isso, é preciso que seja calculada uma trajetória que evite isto.

4 RESULTADOS FINAIS

Ambos algoritmos foram implementados com sucesso, nesta seção serão apresentados os resultados obtidos a fim de compará-los e decidir qual o melhor para o *software* de estratégia da equipe RoboFEI.

4.1 A*

O algoritmo A* foi o mais simples de ser implementado e otimizado, seu único problema é que para que o mesmo funcione é necessário possuir algum tipo de mapa, o que acaba aumentando o consumo de memória. Porém, como visto na seção de modelagem de ambiente foi possível criar um mapa que represente da melhor maneira possível o campo real e ainda assim reduzir o consumo de memória. Outro detalhe que é preciso prestar atenção no algoritmo é que é necessário realizar uma ordenação na *open list* à todo momento a fim de sempre saber qual célula possui o menor valor de $f(v)$. Como esta operação ocorre diversas vezes para encontrar um caminho e o tamanho da *open list* tende a não ser pequeno, não pode-se utilizar métodos triviais para ordenação de dados. Pensando nisso está sendo utilizado uma *priority queue* como estrutura de dados para a *open list*. Uma *priority queue* ou lista de prioridade é uma estrutura de dados cuja função é ordenar uma determinada lista a partir de um certo valor atribuído à cada dado que será inserido na lista, no caso da *open list* os dados serão ordenados de acordo com o valor de $f(v)$.

Nas Tabelas 1 e 2 é possível ver como foi o desempenho do A* para resolver as situações propostas nas Figuras 13, 14, 15.

Tabela 1 – Resultados obtidos para o tempo de cálculo para as três situações teste propostas.

Situação	Tempo de cálculo médio (ms)	Desvio padrão do tempo de cálculo (ms)	Máximo	Mínimo
1	0,404	0,182	1,558	0,289
2	0,365	0,139	1,404	0,281
3	26,088	7,365	46,511	3,691

Fonte: Elaborada pelo autor.

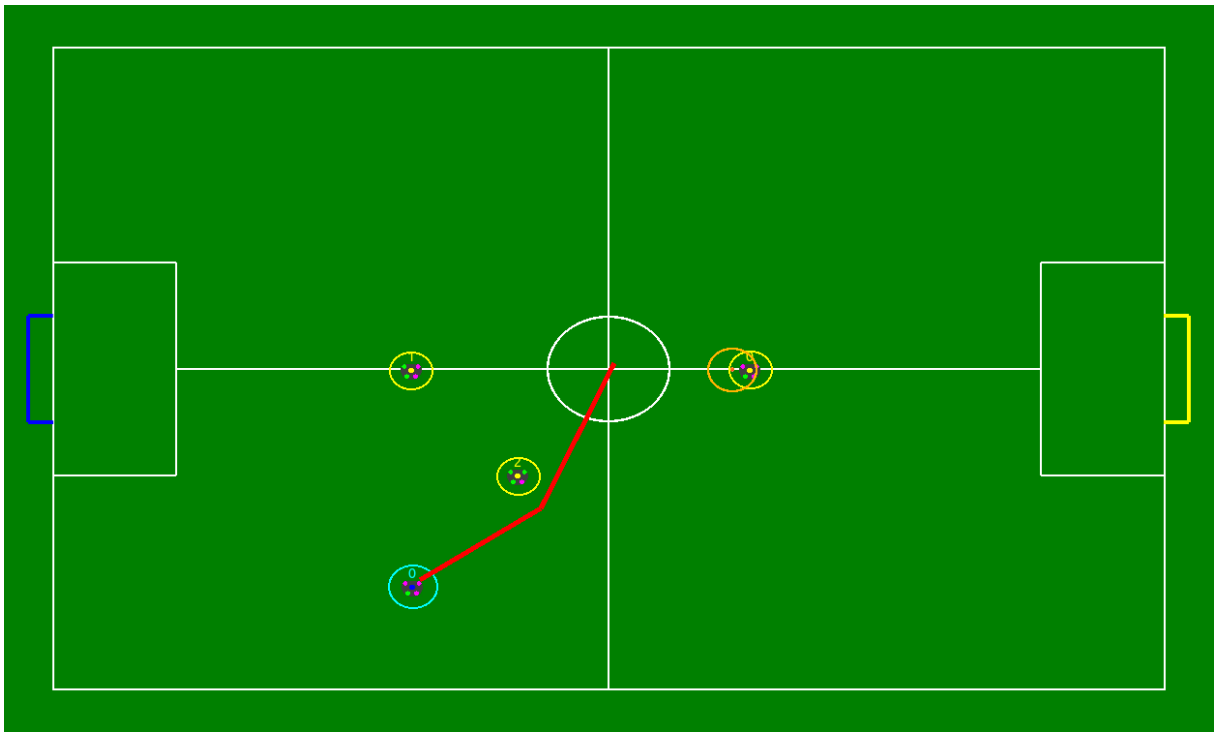
Tabela 2 – Resultados obtidos para o comprimento do caminho gerado para as três situações teste propostas.

Situação	Comprimento do caminho(mm)	Desvio padrão do comprimento (mm)
1	2694,992	9,422
2	2372,960	0,000
3	13244,600	0,000

Fonte: Elaborada pelo autor.

Nas Figuras 16, 17 e 18 têm-se os resultados obtidos utilizando o A*

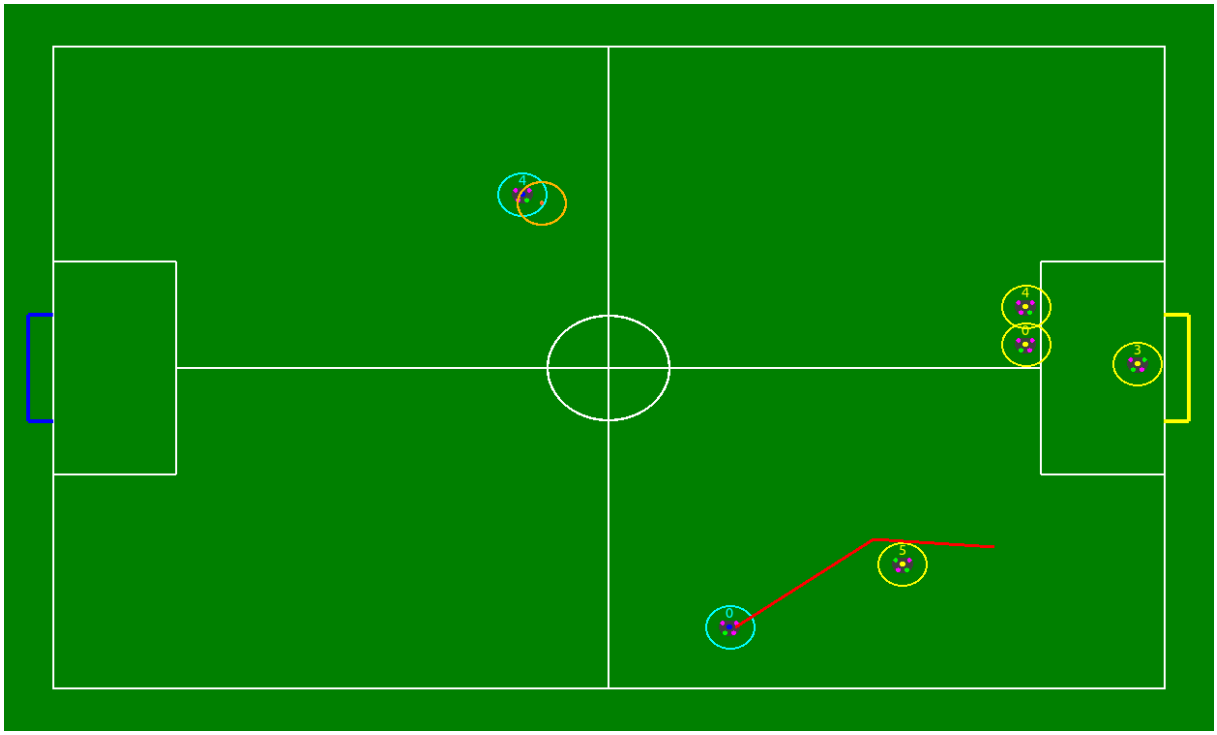
Figura 16 – Teste situação 1 A*.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se o caminho encontrado pelo A* em vermelho e o robô amarelo-2 como obstáculo.

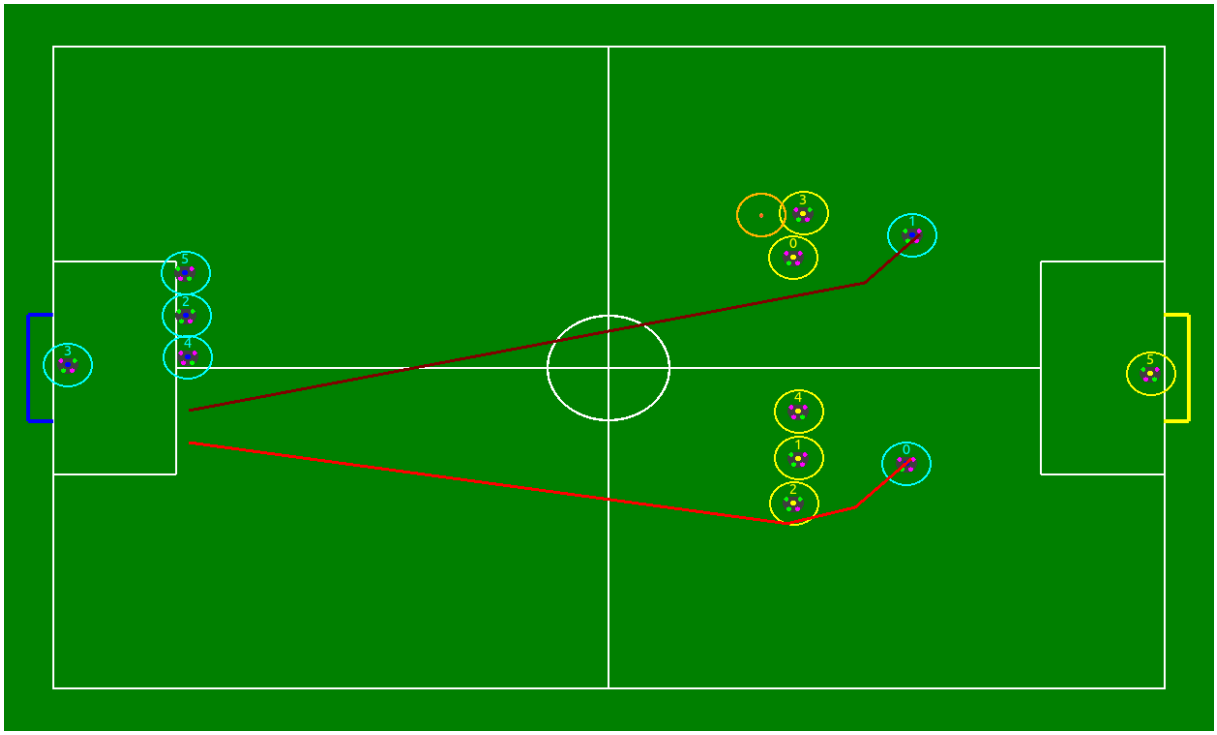
Figura 17 – Teste situação 2 A*.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se o caminho encontrado pelo A* em vermelho e o robô amarelo-5 como obstáculo.

Figura 18 – Teste situação 3 A*.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se os caminhos encontrados pelo A* em vermelho e vermelho escuro e os robôs amarelos 0, 1, 2, 3 e 4 como obstáculo.

4.2 RRT

O algoritmo RRT foi implementado com sucesso, embora ele não necessite de um mapa do ambiente como o A*, em um dos passos do algoritmo existe a necessidade de realizar uma busca para encontrar o vértice da árvore mais próximo da vértice aleatório gerado, esse tipo de problema é conhecido como *nearest neighbor search* (NNS). Ao realizar os primeiros testes com o algoritmo notou-se que a resolução deste tipo de problema não é algo trivial a ponto de ser resolvido com uma simples busca linear. É necessário tratar este problema com cuidado, pelos testes realizados o desempenho do algoritmo pode cair drasticamente ao utilizar uma solução não apropriada. Vários fatores influenciam na complexidade da resolução, como, o tamanho da árvore e o número de vezes que é necessário realizar uma busca. No caso do RRT ambos fatores citados tendem a ser importantes, já que para cada novo vértice a ser inserido na árvore é preciso realizar uma NNS. Em Bruce e Veloso (2002) é sugerido que seja utilizada uma K-d *Tree* para solucionar este problema. Uma K-d *Tree* é uma estrutura de dados própria para realizar NNS, para isso foi decidido utilizar a biblioteca *nanoflann*[6], assim não foi necessário se preocupar com o desempenho da K-d *Tree*, esta biblioteca foi escolhida pois ao realizar algumas pesquisas ela foi a que apresentava os melhores resultados. Ao utilizar a K-d *Tree* o desempenho do RRT melhorou consideravelmente, o tempo médio de cálculo inicial era cerca

de $64{,}4ms$ para encontrar um caminho em condições onde não existem muitos obstáculos, e então num teste realizado recentemente foram obtidos os resultados apresentados na Tabela 3.

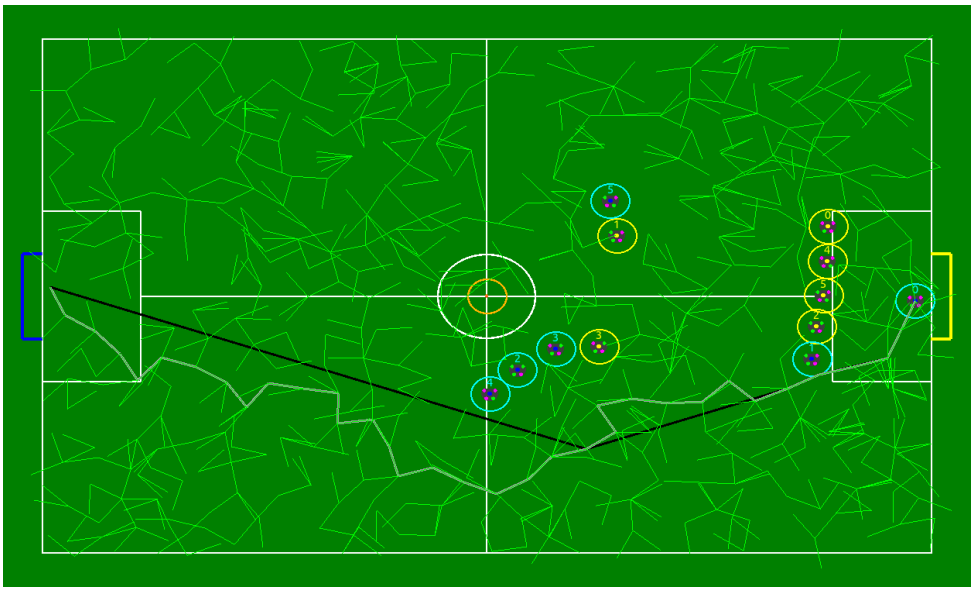
Tabela 3 – Resultados obtidos após implementação da K-d *Tree*.

Tempo de cálculo médio (ms)	Desvio padrão do tempo de cálculo (ms)
44,322	23,922
Comprimento do caminho(mm)	Desvio padrão do comprimento (mm)
9618,894	362,434

Fonte: Elaborada pelo autor.

Na Figura 19 é possível ver em que tipo de cenário foi realizado o teste, embora o tempo médio obtido não seja baixo é preciso lembrar que tal situação dificilmente aconteceria num jogo real.

Figura 19 – Cenário de teste utilizado.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se a árvore gerada pelo RRT em verde, o caminho sem tratamento algum em cinza e o caminho final após tratamento em preto, aqui o robô azul-0 deve movimentar-se de sua posição inicial até o outro gol.

Nas Tabelas 4 e 5 é possível ver como foi o desempenho do RRT para resolver as situações propostas nas Figuras 13, 14, 15.

Tabela 4 – Resultados obtidos para o tempo de cálculo para as três situações teste propostas.

Situação	Tempo de cálculo médio (ms)	Desvio padrão do tempo de cálculo (ms)	Máximo	Mínimo
1	2,736	2,225	16,903	0,057
2	2,883	1,901	15,144	0,090
3	25,232	7,365	45,526	2,907

Fonte: Elaborada pelo autor.

[6]Disponível em: <https://github.com/jlblancoc/nanoflann> Acesso em: 7 fev. 2018 as 10:00.

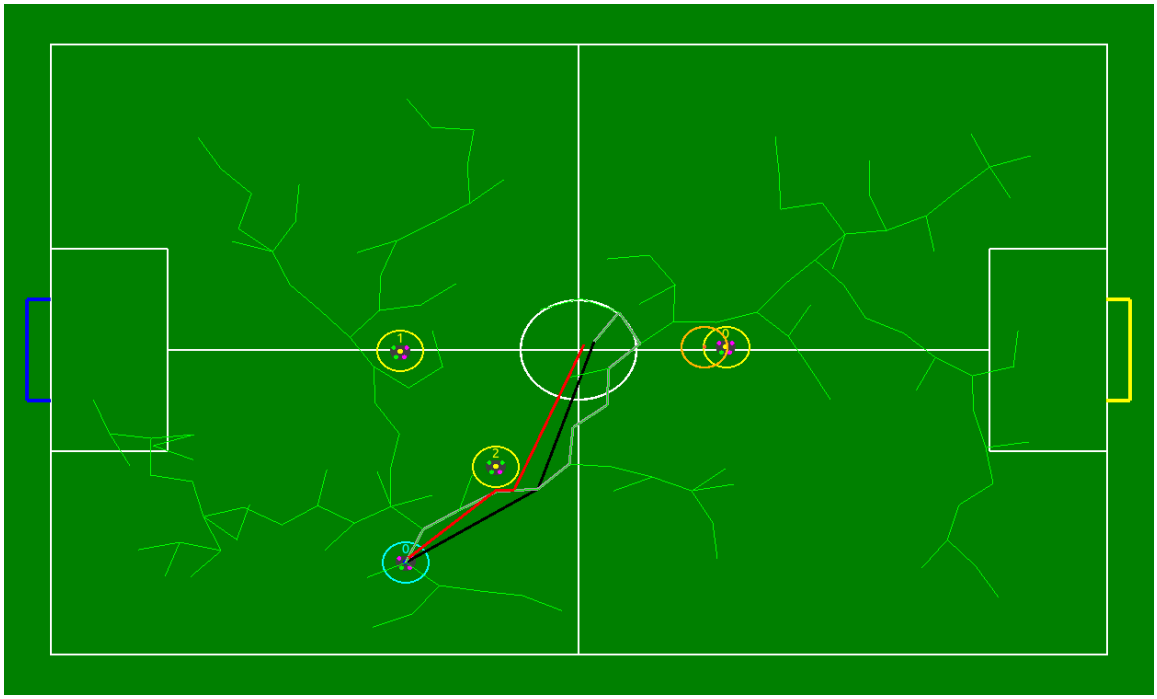
Tabela 5 – Resultados obtidos para o comprimento do caminho gerado para as três situações teste propostas.

Situação	Comprimento do caminho(mm)	Desvio padrão do comprimento (mm)
1	2786,238	161,267
2	2678,987	433,461
3	7427,869	2830,928

Fonte: Elaborada pelo autor.

Nas Figuras 20 e 21 fica evidente um defeito que o RRT possui. Embora o destino esteja bem próximo do ponto inicial o algoritmo acaba explorando uma grande área antes de encontrar um caminho.

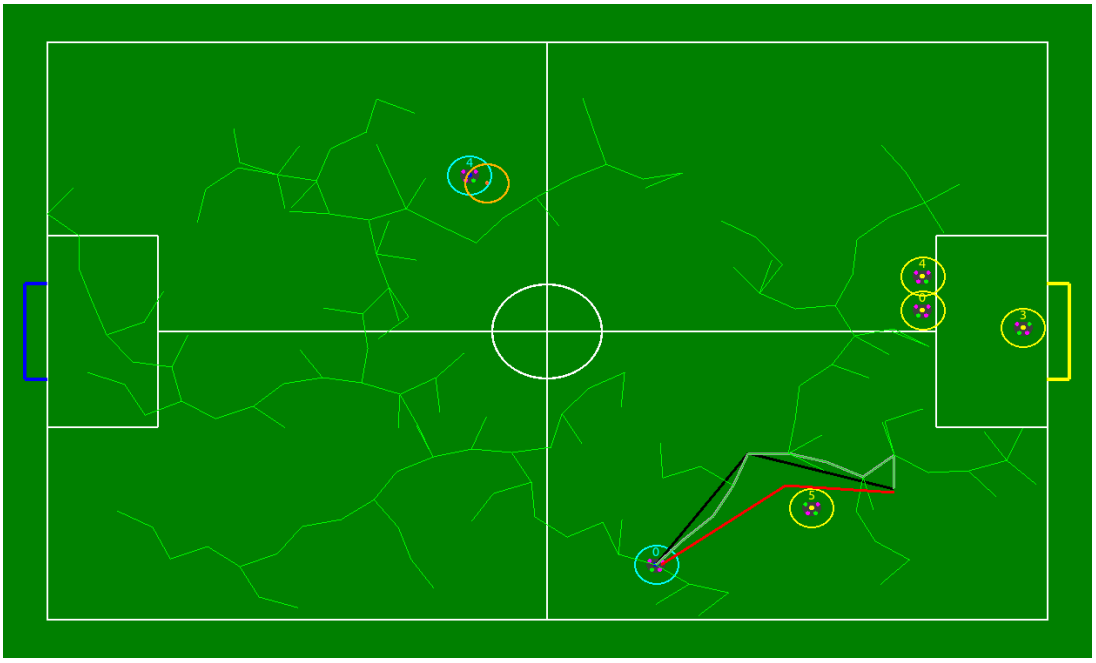
Figura 20 – Teste situação 1 RRT.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se o caminho encontrado pelo A* em vermelho, a árvore gerada pelo RRT em verde, o caminho sem tratamento em cinza, o caminho após tratamento em preto e o robô amarelo-2 como obstáculo.

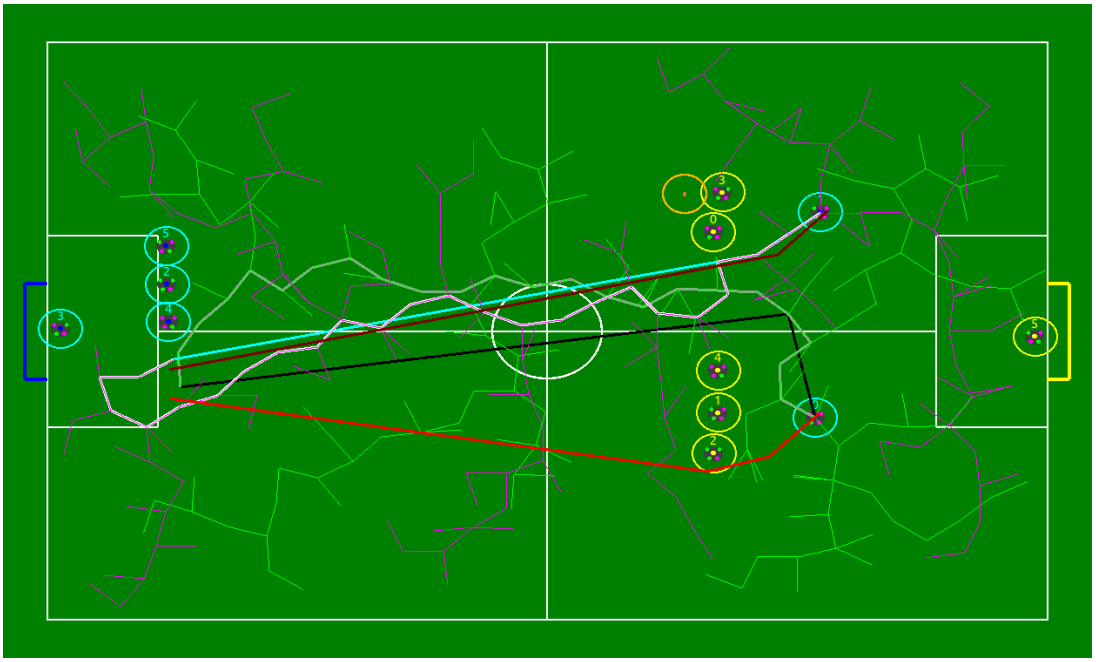
Figura 21 – Teste situação 2 RRT.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se o caminho encontrado pelo A* em vermelho, a árvore gerada pelo RRT em verde, o caminho sem tratamento em cinza, o caminho após tratamento em preto e o robô amarelo-5 como obstáculo.

Figura 22 – Teste situação 3 RRT.



Fonte: Elaborada pelo autor.

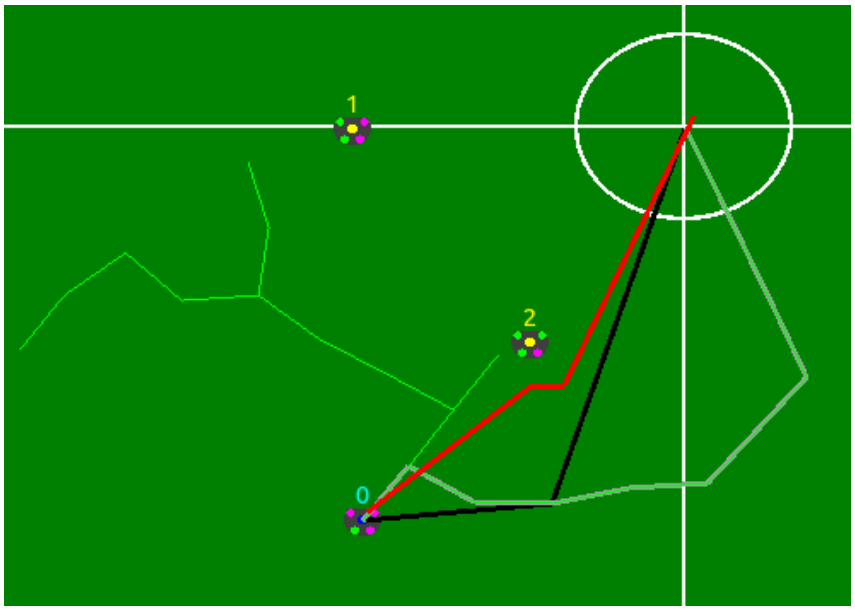
Legenda: Nesta figura tem-se o caminho encontrado pelo A* em vermelho, as árvores geradas pelo RRT em verde e roxo, o caminho sem tratamento em cinza e branco, o caminho após tratamento em preto e azul claro e os robôs amarelos 0, 1, 2, 3 e 4 como obstáculo.

4.3 ALGORITMO DE OTIMIZAÇÃO DE CAMINHOS

Como os algoritmos de *path-planning* discutidos nesta pesquisa não levam em conta as características físicas do mundo real, muitas vezes o caminho encontrado não possui certas características desejadas, por exemplo, a suavidade. Este problema é ainda mais crítico para o RRT, onde é possível ver na Figura 19 que existem muitos zigue-zagues no caminho encontrado pelo algoritmo. Mesmo que os robôs da categoria SSL sejam omni-direcionais, não é desejado que eles se movimentem dessa maneira. Pensando nisso foi desenvolvido um pequeno algoritmo cujo objetivo é linearizar o caminho encontrado por ambos *path-planners*, no Algoritmo 1 é possível ver seu pseudo-código.

Na Figura 23 é possível ver o algoritmo funcionando.

Figura 23 – Exemplo do Algoritmo 1 funcionando.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se a árvore gerada pelo RRT em verde, o caminho sem tratamento algum em cinza e o caminho final após tratamento em preto.

Algoritmo 1 – Linearizar caminho.

```
1  Entrada: CaminhoInicial;
2  Saída: CaminhoReduzido;
3  Dados: n ← CaminhoInicial.size-1;  /* Índice da última posição */
4  Dados: i ← 0 ;                     /* Variável auxiliar */
5  início
6      enquanto n > 0 faça
7          enquanto ExisteObstaculoEntre(CaminhoInicial[n],CaminhoInicial[i])
            faça
8              i++;
9          fim
10         CaminhoReduzido.prepend(CaminhoInicial[i]);
11         se n < i então
12             n ← i;
13         fim
14         senão
15             n ← i − 1;
16         fim
17         i ← 0;
18         se CaminhoReduzido.size >= CaminhoInicial.size então
19             break;
20         fim
21     fim
22     retorna CaminhoReduzido;
23 fim
```

4.4 MODIFICAÇÃO NO RRT

Devido aos problemas ilustrados nas Figuras 20 e 21 onde o RRT explora uma grande área do mapa antes de encontrar um caminho foi feita uma pequena modificação no algoritmo a fim de amenizar este problema. O método RANDOM CONFIG do pseudo algoritmo do RRT (Figura 10) foi substituído pelo método descrito no Algoritmo 2.

Embora seja um pequena modificação os resultados foram expressivos, para a mesma situação apresentada na Figura 19 é possível ver os resultados na Tabela 6 e o trajeto gerado pode ser visto na Figura 24. Ao comparar os dados da Tabela 6 com a Tabela 3 nota-se uma redução de $96{,}65\%$ no tempo de cálculo médio.

Algoritmo 2 – Modificação RRT.

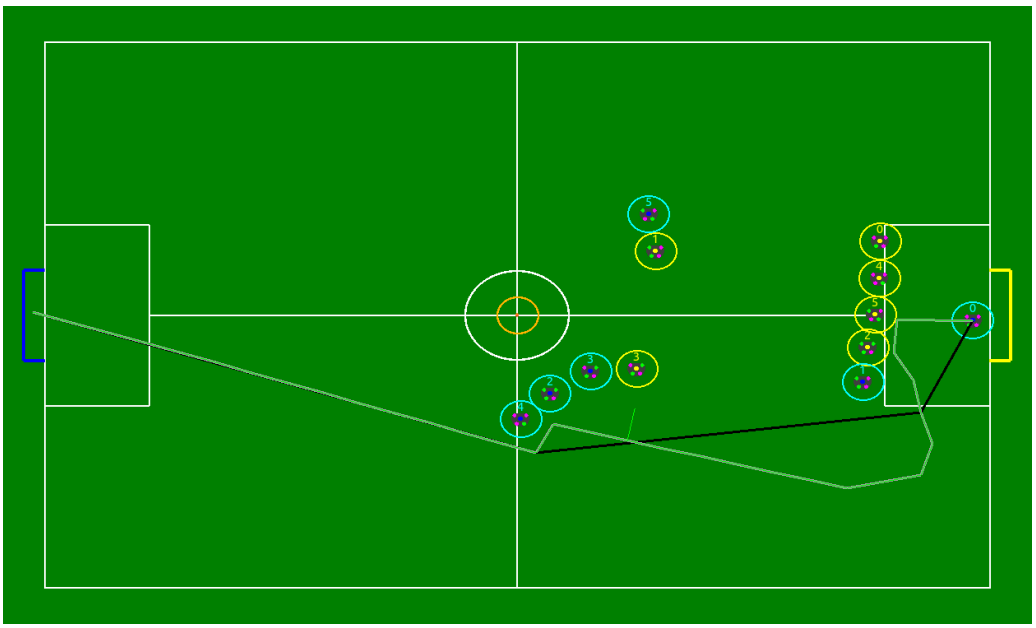
```
1  Saída: Ponto; /* Direção em que a árvore irá se expandir */
2  Dados: Destino;
3  Dados: Aux ;   /* Vértice da árvore mais próximo do destino
       */
4  início
5  |  Aux ← VerticeMaisProximoDe(Destino);
6  |  se NaoExisteObstaculoEntre(Aux, Destino) então
7  |  |  Ponto ← Destino;
8  |  fim
9  |  senão
10 |  |  Ponto ← PontoAleatorio();
11 |  fim
12 |  retorna Ponto;
13 fim
```

Tabela 6 – Resultados obtidos para a modificação feita no RRT.

Tempo de cálculo médio (ms)	Desvio padrão do tempo de cálculo (ms)	Máximo	Mínimo
1,484	1,141	26,099	0,308

Fonte: Elaborada pelo autor.

Figura 24 – Trajeto encontrado após modificação no RRT.



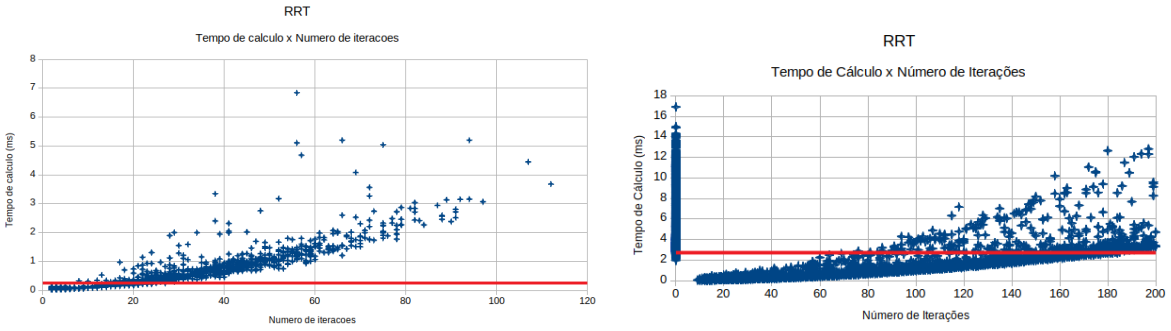
Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se a árvore gerada pelo RRT em verde, o caminho sem tratamento algum em cinza e o caminho final após tratamento em preto.

Na Figura 25 é possível ver a melhora no desempenho do RRT no tempo de cálculo em relação ao número de iterações do algoritmo. Deve-se observar nos gráficos que os pontos

acumulados em $x = 0$ representam as vezes em que o algoritmo não foi capaz de encontrar um caminho.

Figura 25 – Gráficos do desempenho do RRT sem modificações e com modificações para a mesma situação.



Fonte: Elaborado pelo autor.

Legenda: Nas figuras a linha vermelha representa o valor médio.

4.5 TESTES EM CAMPO REAL

Também foram feitos testes com os robôs reais a fim de verificar se os trajetos gerados pelos algoritmos de fato não geram colisões. Já que a movimentação dos robôs não dependem somente do algoritmo de *path-planning* utilizado e sim de um conjunto de fatores como o controle do robô, por exemplo, a velocidade do robô foi reduzida para abstrair alguma imperfeição no sistema de controle, assim é possível verificar com grande certeza de forma empírica se determinada colisão ocorreu devido à algum problema no trajeto gerado.

Nos testes feitos em campo foi feito com que o robô se movimentasse de um lado para outro de forma a desviar dos obstáculos que foram colocados no campo, à todo momento o destino do robô e a posição dos obstáculos eram modificados para que fossem testadas diversas condições.

O algoritmo RRT utilizado no teste já estava com as alterações mostradas na seção 4.4.

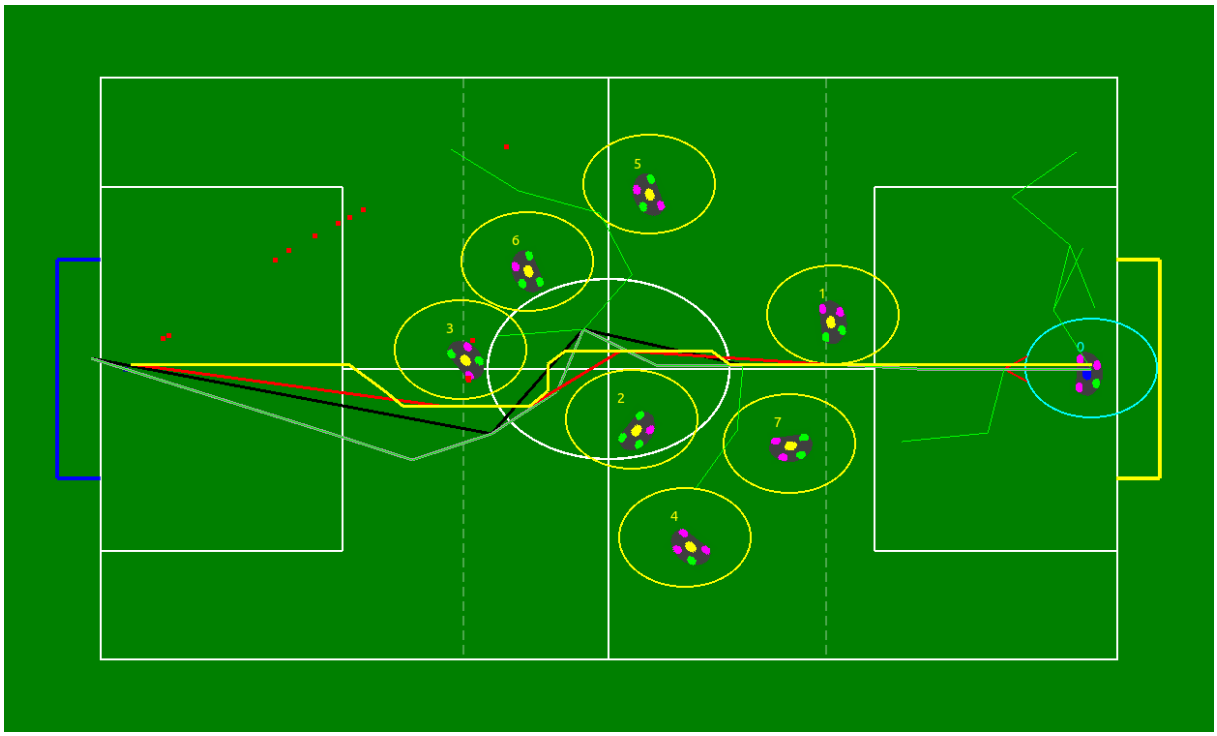
Na Figura 26 é possível ver uma das situações de teste feita em campo. Na Tabela 7 têm-se os resultados obtidos neste teste.

Tabela 7 – Resultados obtidos nos testes em campo com o RRT modificado.

Algoritmo	Tempo médio (ms)	Máximo	Mínimo	Caminhos com tempo abaixo da média	Tempos acima do máximo do outro algoritmo
A*	0,213	1,796	0,064	68,60%	0%
RRT	0,241	6,841	0,025	76,29%	2,03%

Fonte: Elaborada pelo autor.

Figura 26 – Uma das situações de teste feita em campo.



Fonte: Elaborada pelo autor.

Legenda: Nesta figura tem-se a árvore gerada pelo RRT em verde, o caminho sem tratamento algum em cinza e o caminho final após tratamento em preto, o caminho sem tratamento gerado pelo A* em amarelo e o caminho após tratamento em vermelho.

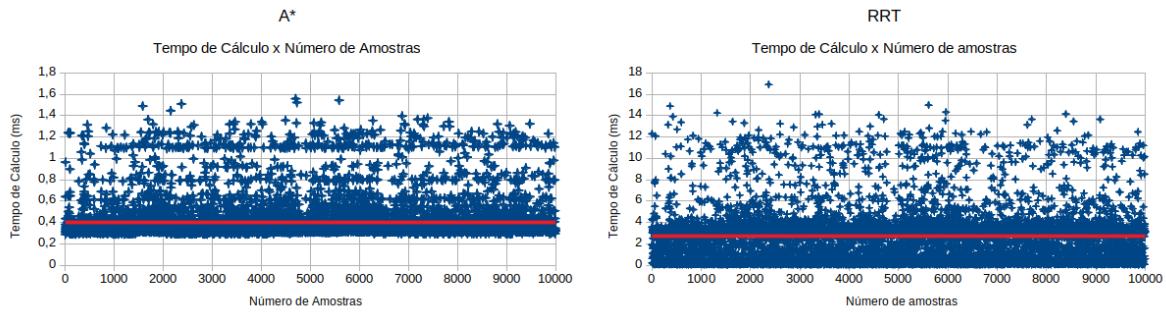
5 ANÁLISE DOS RESULTADOS

Como o objetivo deste estudo é somente comparar o A* e o RRT em suas formas primitivas, nesta conclusão não será levada em conta a performance do RRT com a mudança descrita na seção 4.4 para decidir qual o melhor algoritmo entre os dois, porém, após feita esta mudança o desempenho do RRT se equiparou ao do A* e em alguns casos ele chegou a ser um pouco melhor. Nos testes feitos somente cerca de 6% das vezes o RRT demorou mais que o tempo máximo do A* e o tempo médio do RRT foi cerca de 11% maior que o tempo médio do A*. Embora pareça ser uma pequena diferença de um algoritmo para outro deve-se lembrar que o cálculo de trajetória é feito para todos os robôs em campo, ou seja, no mínimo seis trajetos deverão ser calculados à cada iteração do *loop* do *software* de estratégia, portanto é crucial que seja utilizado o algoritmo mais rápido.

Já que uma pequena mudança no algoritmo gerou resultados consideravelmente bons, em um próximo trabalho seria interessante analisar algumas variações do RRT a fim de verificar se seu desempenho aumenta ainda mais.

Nas Figuras 27, 28 e 29 é possível ver o desempenho de cada algoritmo nas três situações. Nesses gráficos é possível ver que o A* não varia muito em torno do valor médio ao contrário do que acontece com o RRT. Para o A* a amplitude total dos dados coletados é de $1{,}269ms$, $1{,}124ms$ e $42{,}820ms$ para as situações 1, 2 e 3 respectivamente, para o RRT esses valores são cerca de $16{,}846ms$, $15{,}053ms$ e $42{,}620ms$ para as situações 1, 2 e 3 respectivamente, esses dados mostram que para o RRT pode ocorrer uma grande variação no tempo de cálculo dado um mesmo ponto inicial e destino. O principal motivo para a queda de performance do RRT parece ser o que foi citado como seu segundo ponto fraco, talvez alguma variação do algoritmo como o ERRT (BRUCE; VELOSO, 2002) possa obter uma performance superior ao A*.

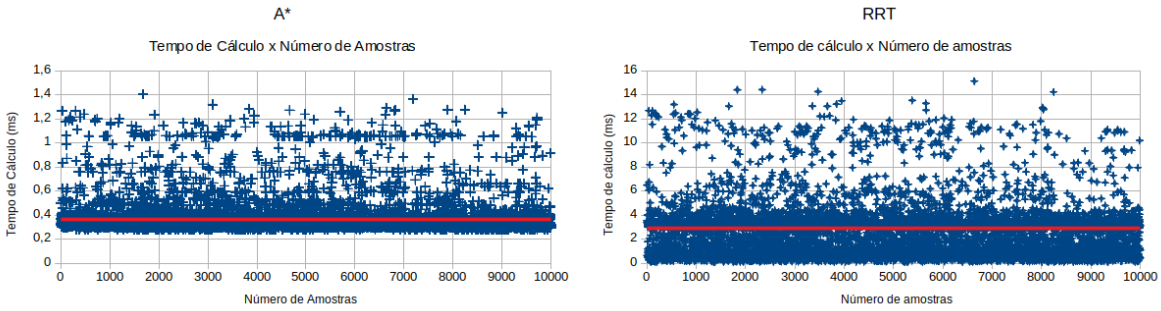
Figura 27 – Gráficos do desempenho do A* e RRT na situação 1.



Fonte: Elaborado pelo autor.

Legenda: Nas figuras a linha vermelha representa o valor médio.

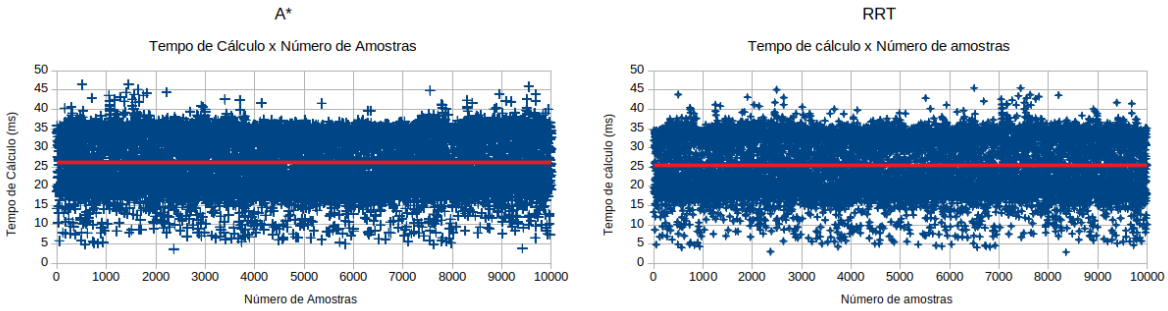
Figura 28 – Gráficos do desempenho do A* e RRT na situação 2.



Fonte: Elaborado pelo autor.

Legenda: Nas figuras a linha vermelha representa o valor médio.

Figura 29 – Gráficos do desempenho do A* e RRT na situação 3.



Fonte: Elaborado pelo autor.

Legenda: Nas figuras a linha vermelha representa o valor médio.

Nos testes utilizando os algoritmos em sua forma primitiva foram coletadas cerca de 10.000 amostras para cada situação. Após analisar esses dados é possível tomar uma decisão sobre qual algoritmo é o melhor para a equipe RoboFEI-SSL, ambos algoritmos possuem pontos fortes e fracos que serão explicados neste capítulo.

5.1 A*

O A* mostrou-se eficiente ao encontrar os trajetos em um tempo baixo. De acordo com os testes feitos, para a situação 1 cerca de 9% do total de amostras possui tempo de cálculo superior à média somada com um desvio padrão, para a situação 2 este valor é cerca de $7{,}44\%$ e para a situação 3 o valor sobe para cerca de $24{,}49\%$. Também vale a pena notar que sempre que existir um caminho possível ele irá encontrá-lo, esta é uma característica dos algoritmos determinísticos. Por ser bastante simples ele possibilita que a partir de modificações na Equação 1 sejam encontrados caminhos com novas características.

Apesar da simplicidade para implementar o algoritmo existe um passo anterior que deve ser feito que é a modelagem do ambiente, para que o algoritmo funcione é obrigatório existir uma representação do campo real e isso acaba aumentando o custo de memória e de processamento do algoritmo como um todo, pois é necessário estar sempre atualizando este mapa com as novas informações que chegam à cada aproximadamente $16ms$. Para que o consumo de memória seja reduzido e não existam muitas células para o algoritmo percorrer, o mapa foi construído utilizando uma escala de $1 : 75mm$.

Pelo fato de o algoritmo procurar um caminho até que seja constatado que não exista nenhum, nos casos em que o destino está completamente bloqueado o A* pode levar um grande tempo até explorar todas as células do mapa e então constatar que não existe caminho nenhum. Portanto é preciso sempre evitar esta situação, a solução utilizada neste trabalho foi de calcular apenas metade do caminho caso isso ocorra, assim evita-se um tempo de cálculo desnecessário e o robô não irá ficar parado esperando o destino ficar desbloqueado.

Outra característica que deve ser tratada é que o algoritmo sempre irá gerar caminhos tangentes aos obstáculos, e isso não é algo desejado que o robô faça. Assim, para que isto não seja um problema é necessário adicionar uma área de segurança em torno de todos os objetos para que ao encontrar um caminho o robô no máximo passe perto dessas áreas.

5.2 RRT

Devido à não ser necessário uma representação do ambiente real a implementação do RRT é muito simples e o custo de memória é inferior ao do A*. Ao analisar as Tabelas 1 e 4 é possível notar que o RRT possui um tempo mínimo de cálculo menor do que o A*, embora ele só encontre um caminho tão rápido quanto o A* em cerca de $15{,}97\%$, $2{,}84\%$ e $0{,}02\%$ do total de amostras para as situações 1, 2 e 3 respectivamente.

Uma grande desvantagem é que por ser um algoritmo do tipo estocástico nem sempre ele irá encontrar um caminho, então podem ocorrer casos em que é gasto um certo tempo procurando um caminho porém não é possível encontrá-lo, isto pode ser diminuído aumentando o número máximo de interações do algoritmo, porém, isso faz com o que o tempo médio de cálculo aumente, nos testes feitos este número foi definido como 200 para as situações 1 e 2, cerca de $56{,}43\%$ e $60{,}45\%$,respectivamente, do total de amostras não foi encontrado um caminho, para a situação 3 o número máximo de interações foi 500 e em $29{,}66\%$ do total de amostras não foi encontrado um caminho.

Algo que também leva à um alto tempo de cálculo é a total aleatoriedade do algoritmo, muitas vezes mesmo com o destino bem próximo do ponto inicial o algoritmo chega a explorar grande parte do mapa antes de encontrar um caminho. Nas Figuras 20 e 23 é possível ver como em um caso o algoritmo expandiu poucos vértices e no outro uma grande área do campo foi explorada. Em Bruce e Veloso (2002) é descrita uma variação do RRT que não é restrita à se

expandir somente para pontos aleatórios, isto pode diminuir consideravelmente este efeito que o RRT original possui.

Um problema que afeta ambos os algoritmos é que a geometria do trajeto gerado não é a melhor para que um robô a siga, isso ocorre devido à alguns zigue-zagues que aparecem no trajeto. Este efeito é ainda mais acentuado no RRT devido ao caminho ser gerado de forma aleatória. Portanto é necessário o uso de algum tipo de tratamento para melhorar a geometria do caminho gerado, na seção 4.3 foi apresentada a solução utilizada para este problema, porém existem outros métodos mais sofisticados, como o uso de *splines* (ELBANHAWI; SIMIC; JAZAR, 2015) ou o que é apresentado em Yang e Sukkarieh (2010).

6 CONCLUSÃO

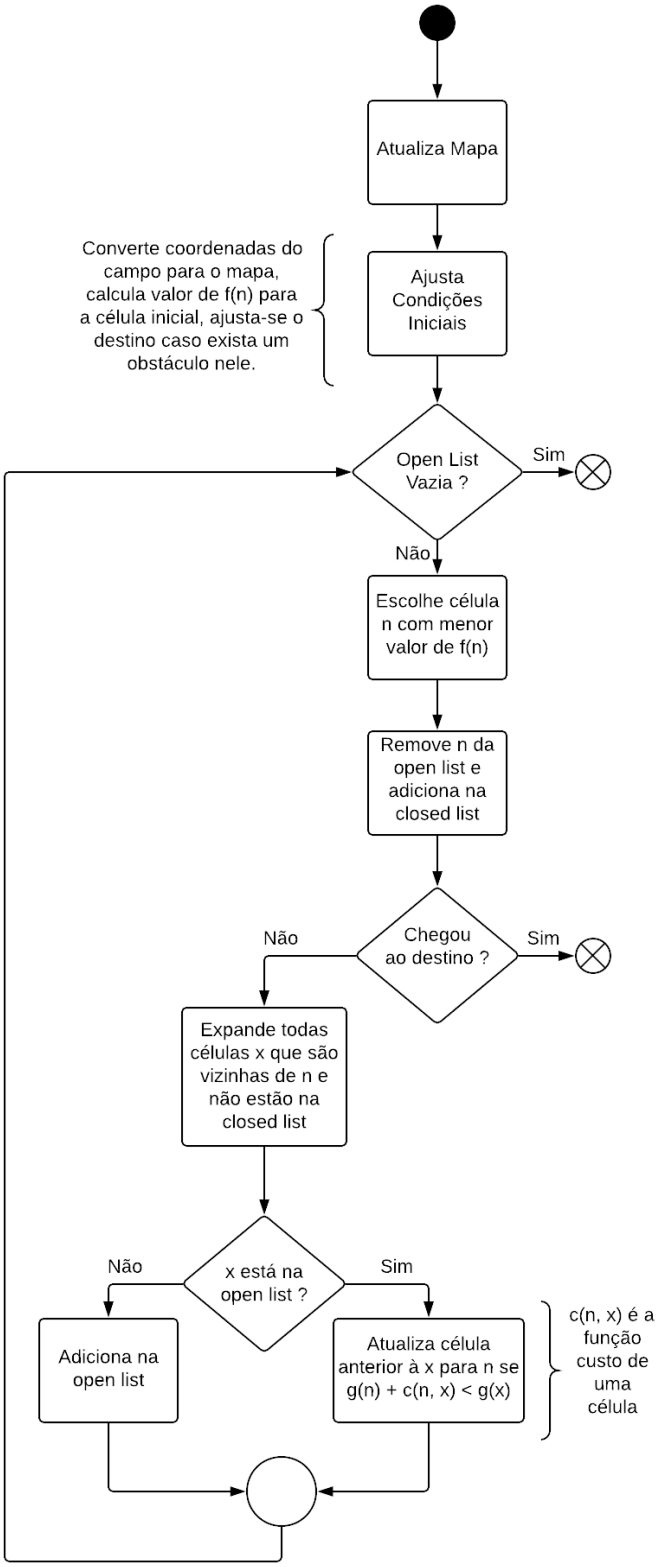
O objetivo deste trabalho foi comparar os algoritmos A* e RRT a fim de definir qual o melhor para as necessidades da equipe RoboFEI-SSL. Foram definidos os conceitos necessários para implementar de forma eficiente ambos algoritmos e também apresentadas certas características de cada um deles que devem ser levadas em conta ao utilizá-los.

A partir da análise dos dados foi possível comprovar que o A* é o algoritmo mais eficiente em comparação ao RRT, devido ao seu baixo tempo de cálculo e caminhos curtos, que são variáveis importantes no futebol de robôs.

Para trabalhos futuros seria interessante comparar algumas variações de cada uma dessas famílias de algoritmos como o *Jump Point Search* (DUCHOŇ et al., 2014) da família A* e o *Extended RRT* (BRUCE; VELOSO, 2002) da família RRT.

APÊNDICE A – FLUXOGRAMA A*

Figura 30 – Fluxograma A*.



Fonte:Adaptado de Choset (2007).

REFERÊNCIAS

BRUCE, James; VELOSO, Manuela M. Real-time randomized path planning for robot navigation. In: SPRINGER. ROBOT Soccer World Cup. [S.l.: s.n.], 2002. P. 288–295.

CHOSET, Howie. Robotic motion planning: A* and D* search. **Robotics Institute**, p. 16–735, 2007.

DUCHOŇ, František et al. Path planning with modified a star algorithm for a mobile robot. **Procedia Engineering**, Elsevier, v. 96, p. 59–69, 2014.

ELBANHAWI, Mohamed; SIMIC, Milan; JAZAR, Reza N. Continuous path smoothing for car-like robots using B-spline curves. **Journal of Intelligent & Robotic Systems**, Springer, v. 80, n. 1, p. 23–56, 2015.

GUPTA, Aakash; UMRAO, Sachin; KUMAR, Sumit. Optimal Path Planning in a Dynamic Environment.

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**, IEEE, v. 4, n. 2, p. 100–107, 1968.

KOENIG, Sven; LIKHACHEV, Maxim. Improved fast replanning for robot navigation in unknown terrain. In: IEEE. ROBOTICS and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on. [S.l.: s.n.], 2002. v. 1, p. 968–975.

KUFFNER, James J; LAVALLE, Steven M. RRT-connect: An efficient approach to single-query path planning. In: IEEE. ROBOTICS and Automation, 2000. Proceedings. ICRA'00. IEEE International Conference on. [S.l.: s.n.], 2000. v. 2, p. 995–1001.

LAVALLE, Steven M. Rapidly-exploring random trees: A new tool for path planning. Citeseer, 1998.

MONAJJEMI, Valiallah; KOOCHAKZADEH, Ali; GHIDARY, Saeed Shiry. grsim–robocup small size robot soccer simulator. In: SPRINGER. ROBOT Soccer World Cup. [S.l.: s.n.], 2011. P. 450–460.

NASH, Alex; KOENIG, Sven. Any-angle path planning. **AI Magazine**, v. 34, n. 4, p. 85–107, 2013.

POUDEH, Amin Ganjali et al. MRL extended team description 2016. In: PROCEEDINGS of the 19th International RoboCup Symposium. [S.l.: s.n.], 2016.

RODRÍGUEZ, Saith et al. Fast path planning algorithm for the robocup small size league. In: SPRINGER. ROBOT Soccer World Cup. [S.l.: s.n.], 2014. P. 407–418.

RULES, Small Size League. **Página das regras da SSL**. Acesso em 17 ago. 2017 as 10:25. Set. 2011. Disponível em: <http://wiki.robocup.org/Small_Size_League/Rules>.

SASKA, Martin et al. Robot path planning using particle swarm optimization of Ferguson splines. In: IEEE. EMERGING Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on. [S.l.: s.n.], 2006. P. 833–839.

SOUISSI, Omar et al. Path planning: A 2013 survey. In: IEEE. INDUSTRIAL Engineering and Systems Management (IESM), Proceedings of 2013 International Conference on. [S.l.: s.n.], 2013. P. 1–8.

STENTZ, Anthony. Optimal and efficient path planning for partially-known environments. In: ICRA. [S.l.: s.n.], 1994. v. 94, p. 3310–3317.

SUN, Xiaoxun; KOENIG, Sven; YEOH, William. Generalized adaptive A. In: INTERNATIONAL FOUNDATION FOR AUTONOMOUS AGENTS e MULTIAGENT SYSTEMS. PROCEEDINGS of the 7th international joint conference on Autonomous agents and multiagent systems-Volume 1. [S.l.: s.n.], 2008. P. 469–476.

YANG, Kwangjin; SUKKARIEH, Salah. An analytical continuous-curvature path-smoothing algorithm. **IEEE Transactions on Robotics**, IEEE, v. 26, n. 3, p. 561–568, 2010.

ZICKLER, Stefan et al. CMDragons 2009 extended team description. In: PROC. 14th International RoboCup Symposium, Singapore. [S.l.: s.n.], 2010.