Davi Braga Tolentino Veloso

Projeto Orientado em Computação

Coordenação de Sistemas Multirrobôs para Representações Visuais

Belo Horizonte, Minas Gerais
2022

Davi Braga Tolentino Veloso

Projeto Orientado em Computação

Coordenação de Sistemas Multirrobôs para Representações Visuais

Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Orientador: Douglas Guimarães Macharet

Belo Horizonte, Minas Gerais
2022

Sumário

1 Introdução

Em Robótica Móvel, sistemas multiagentes são compostos por múltiplos robôs que podem cooperar e comunicar entre si para a realização de tarefas. Esses sistemas estão, paulatinamente, atraindo atenção da indústria, pois podem promover funcionalidades mais diversas e eficientes que sistemas com um único robô. Suas diversas aplicações incluem sincronização de veículos autônomos, apresentação de luzes com drones, movimentação de materiais em armazéns e entregas por meio de drones.

Figura 1 – Exemplo de apresentação de luzes usando drones: "Intel Drone Light Show at The Olympics" [1].

Nesse contexto, é ubíquo o problema de calcular trajetórias eficientes para cada um deles, de forma que não haja colisões. Essa tarefa já tem sido estudada há décadas e apresenta diversas variações, podendo-se considerar robôs de diferentes tamanhos, formas, restrições de movimento, inseridos em ambientes estáticos ou dinâmicos. Pode-se dificultar o problema, arbitrariamente, considerando-se restrições adicionais como a distância mínima ou máxima entre alguns deles, durante partes específicas do percurso, por exemplo.

Conceitualmente, dadas as configurações iniciais e finais dos robôs no espaço e as restrições de movimento, como velocidade e aceleração limitadas, objetiva-se encontrar trajetórias sem colisões para cada um deles, possivelmente otimizando uma ou mais métricas predeterminadas. As duas mais comuns são: minimizar o *makespan*, que refere-se ao tempo gasto pelo agente mais lento, ou seja, o gargalo; e minimizar a soma dos custos individuais dos mesmos.

1.1 Objetivos Gerais

Em linhas gerais, o trabalho propõe estudar o uso de múltiplos robôs em performances visuais, como as apresentações de luzes com drones. Essa tarefa intersecta a vasta área de planejamento de trajetórias em sistemas multiagentes.

Ao final do POC II, espera-se ter um software que tenha como entrada uma sequência de imagens e compute trajetórias eficientes para representá-las com uma quantidade limitada de robôs.

1.2 Objetivos Específicos

- Analisar como algoritmos de outras áreas da computação, como a Teoria dos Grafos, podem ser usados para resolver problemas de alocação de robôs;
- Comparar a eficiência de diferentes algoritmos e heurísticas sob métricas como distância total percorrida e tempo gasto na navegação dos robôs;
- Estudar métodos de alocação que permitam, dada uma imagem, distribuir eficientemente os robôs no espaço de modo a representá-la;
- Criar um simulador com interface gráfica para testar os métodos propostos em diferentes cenários.

2 Referencial Teórico

O problema do cálculo de trajetórias em sistemas multiagentes já foi abordado de várias formas. Cada modelagem requer presunções diferentes, como a holonomicidade dos robôs.

Em Robótica, um robô é dito holonômico quando o número de velocidades controláveis é igual aos seus graus de liberdade, como em robôs construídos sobre *omni-wheels*, capazes de se moverem em qualquer direção em um plano.

Por outro lado, um carro que se move em um plano tem três graus de liberdade: uma posição em cada um dos dois eixos e a orientação; mas possui apenas dois graus controláveis: aceleração/desaceleração e mudança da angulação pelo volante, então não é holonômico.

Do ponto de vista teórico, o planejamento de caminhos em sistemas multiagentes geralmente é computacionalmente intratável [2], mesmo em enunciações simplificadas e discretizadas em grafos. Em particular, a complexidade computacional para otimizar métricas, como as apresentadas na introdução, é *NP-difícil* em várias formulações, como na *pebble motion on graphs*. Esta é uma das mais simples e originou-se do estudo do famoso *15-puzzle* [3], um jogo que se desenvolve em um tabuleiro 4×4. Nela, cada robô ocupa um vértice, e em cada passo, exatamente um deles se move para um vértice vizinho desocupado. O objetivo é que todos os robôs cheguem a seus alvos. Sabe-se que minimizar o número de passos é *NP-difícil* para a generalização natural deste problema, onde utiliza-se um tabuleiro $n \times n$ [4].

A dificuldade em encontrar algoritmos exatos para esses problemas incentivou a criação de inúmeras heurísticas, e algumas serão apresentadas a seguir.

Uma técnica comum é o uso de Campos Potenciais [5], um método baseado em modelar o robô como um ponto que sofre uma força de atração em direção ao seu alvo e forças de repulsão dos obstáculos à sua volta. Esse método é classificado como reativo, pois não há um planejamento *a priori* nem um modelo do mundo, mas sim uma ligação direta entre a percepção do robô do ambiente ao seu redor e de sua ação. Embora os resultados não sejam garantidamente ótimos, esse arquétipo pode: ser paralelizado, implementado com baixo custo de memória, funcionar em situações parcialmente observáveis e simular bem comportamentos emergentes na natureza.

Outro modelo baseia-se na ideia de dividir o espaço em regiões chamadas células, de modo que seja fácil encontrar trajetórias dentro de cada uma delas. Posteriormente, uma busca seria realizada para encontrar uma sequência de células que possuíssem um caminho contínuo entre os pontos de interesse.

Também há técnicas baseadas em amostragem, como o mapa de estrada probabilístico, que seleciona pontos aleatórios no espaço para construir mapas. Usualmente, os mapas

são discretizados e a eficiência das implementações dependem dos algoritmos de busca em grafos e dos detectores de colisões usados [6].

Já foram propostos planejadores de movimento baseados em otimizadores não lineares [7, 8] e, até mesmo, redes neurais artificiais [8].

Voltando ao foco das representações visuais, o problema já foi estudado pela *Disney Research*, em um projeto chamado *Pixelbots* [9, 10], como mostra a Figura 2. Este sistema recebia como entrada uma imagem ou animação, atribuía posições e cores para os robôs disponíveis que melhor representassem a figura, e executava a navegação usando técnicas para prevenir colisões.

Figura 2 – Pixelbots: um display formado por robôs móveis [9].

Outra abordagem, feita pela *Dynamic Systems Lab*, consistiu na criação de um planejador de coreografias para quadrotores [11], baseado no projeto de primitivas de movimento, transição suave entre elas e sincronização dos drones.

Em termos práticos, existem ferramentas de prevenção de colisões desenvolvidas por Jur van den Berg, Stephen J. Guy, Jamie Snape, Ming C. Lin, e Dinesh Manocha [12] que operam em passos de tempo discretos e suportam robôs holonômicos. No caso 2D, os robôs são circulares [12], e no 3D [13], esféricos. Em 2D, também são suportados obstáculos estáticos. Essas ferramentas são populares, pois possuem uma API (*Application Programming Interface*) de fácil integração, consequentemente, já foram usadas até em jogos digitais [14]. Posteriormente, os autores estenderam o método em 2D para robôs diferenciais [15], mas não há implementação aberta como os anteriores [16, 17]. Uma visão geral da técnica usada por esses algoritmos é dada na Seção 3.2.1.

3 Representações com Robôs Homogêneos

Inicialmente, a entrada do sistema consistirá em n robôs iguais circulares e uma imagem a ser representada. Considera-se que cada robô possui um LED capaz de assumir qualquer cor RGB. Cada pixel da imagem possui 4 componentes: RGBA. As componentes RGB representam a cor, e a componente A representa o quanto o pixel deve ser representado. O pixel faz parte do desenho a ser representado se e somente se for diferente de 0. Convencionalmente, cada componente possui 256 valores possíveis. Alguns métodos neste trabalho usam o valor de A binarizado, ou seja, consideram apenas 0 ou diferente de 0, enquanto outros usam o valor original para ponderar valores de recompensa por cobrir cada pixel.

O trabalho lida com a tarefa de coordenar os robôs para representar imagens estudando, separadamente, dois problemas: alocação e navegação. As Figuras 3 a 6 mostram o esquema geral da maneira pela qual a tarefa foi abordada.

(a) Posição atual dos robôs representadas de cinza. (b) Imagem alvo a ser representada.

Figura 3 – A entrada consiste em uma imagem alvo e as posições atuais dos robôs, representados de cinza.

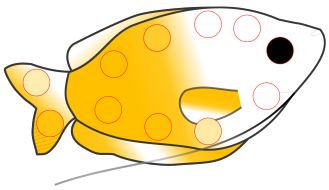


Figura 4 – Passo 1: alocação das posições e cores dos pontos objetivos, representados com círculos de perímetro vermelho.

Inicialmente, o trabalhou focou em robôs de mesmo raio, chamados homogêneos. Posteriormente, na Seção 4, a modelagem é estendida para raios variados — heterogêneos.

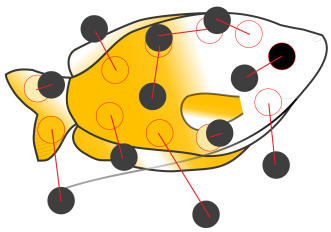


Figura 5 – Passo 2: alocação dos pontos objetivos para os robôs, ou seja, define-se para onde vai cada robô.

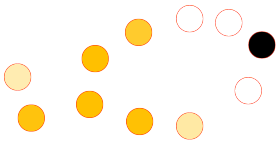


Figura 6 – Passo 3: executa-se a navegação dos robôs sem colisões.

3.1 Alocação

A tarefa da alocação consiste em escolher a posição desejada de cada robô na cena, sem sobreposições, ou seja, para cada par de robôs, a distância entre eles deve ser maior ou igual à soma de seus raios. Essa tarefa foi subdividida em duas. A primeira, descrita na Seção 3.1.1, consiste em definir as coordenadas finais dos robôs. A segunda, descrita na Seção 3.1.2, é responsável pela alocação dos robôs a elas, ou seja, decidir quem vai para cada alvo.

3.1.1 Determinação dos Alvos

Neste passo, a meta é escolher p coordenadas na imagem, $p \leq n$, além das cores que devem ser assumidas ali. Por enquanto, ignora-se qual robô vai para cada lugar.

Idealmente, os pontos devem ser bem representativos da imagem, o que é um conceito subjetivo. Em linhas gerais, dois fatores principais foram considerados para guiar a elaboração das heurísticas: espaçamento entre os pontos (cobrir a figura de forma homogênea, evitando áreas muito vazias ou cheias) e cobertura total de pixels da figura atingida por eles (ou seja, minimizar a cobertura de pixels de valor $A = 0$, similar a problemas de empacotamento de círculos em polígonos diversos).

3.1.1.1 Cores

Há várias maneiras de decidir a cor que deve ser usada pelos robôs em cada coordenada da imagem, como usar a cor do pixel no centro do círculo posicionado naquela coordenada ou a média aritmética das cores dos pixels cobertos por ele. Todavia, optou-se por fazer uma média ponderada das cores dos pixels cobertos por ele, onde os mais próximos do centro têm mais peso. Isso foi computado por meio da aplicação de um filtro gaussiano quadrado de lado $\lceil 2r \rceil$, como ilustrado na Figura 7. Um exemplo das cores decididas após essa convolução[1] é mostrado na Figura 8.

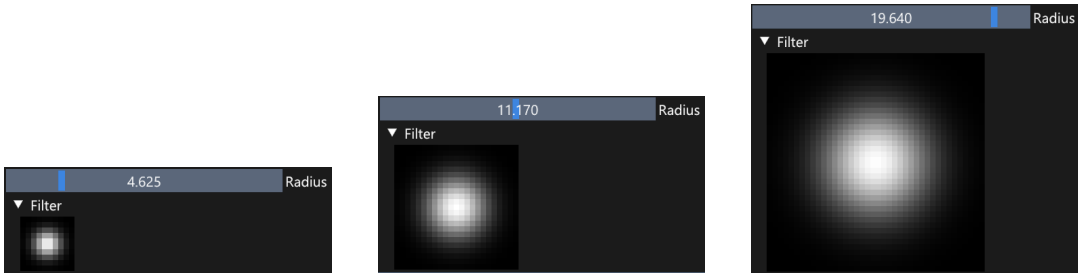
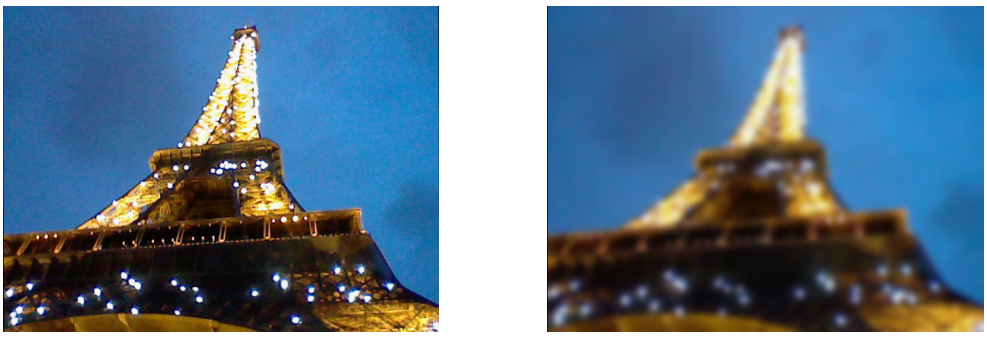


Figura 7 – Exemplos de filtros gaussianos criados para diferentes raios r. O filtro é convoluído na imagem para decidir a cor que deve ser assumida em cada pixel por um robô de lado r. Poderia-se, ainda, interpolar a cor de pixels vizinhos para obter transições ainda mais suaves.



(a) Imagem original. (b) Imagem filtrada.

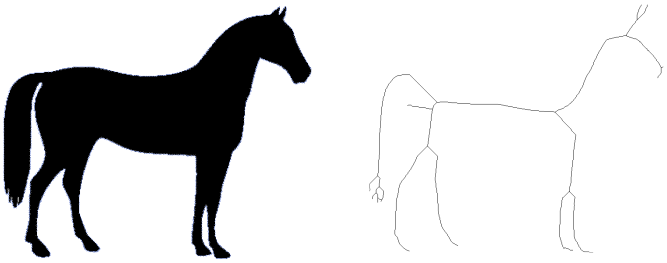
Figura 8 – Exemplo da aplicação do filtro da Figura 7 em uma imagem para calcular a cor que os robôs de determinado raio devem assumir em cada coordenada.

3.1.1.2 Coordenadas

Alguns métodos foram considerados para a escolha das coordenadas dos alvos, como usar bordas e quinas, ou usar pontos no esqueleto morfológico da imagem, como ilustrado na Figura 9.

No final, inspirado na ideia de que o empacotamento de círculos em um plano euclidiano 2D possui a maior densidade possível arranjados em uma grade hexagonal [18], foi usada

[1] Como o filtro é simétrico, correlação e convolução são a mesma operação.



(a) Imagem original. (b) Esqueleto morfológico.

Figura 9 – Exemplo de esqueleto morfológico de uma imagem, que é uma construção geométrica fora do escopo deste trabalho, apenas citada pois foi considerada.

uma heurística baseada em sobrepor uma grade dessas na imagem, como ilustrado na Figura 10, em identificar os hexágonos que possuem pixels que devem ser representados e em alocar um robô para cada um deles, consecutivamente. Na implementação feita, considera-se que cada pixel pertence a exatamente um hexágono, e usa-se a coordenada do centro do pixel para determinar o hexágono. Por fim, dentro de cada hexágono H que possui pelo menos um pixel, a coordenada do alvo é a média aritmética das coordenadas dos centros dos pixels que pertencem a H.

Para garantir que os robôs não se intersectem, é alocado no máximo um robô por hexágono H, de forma que esteja completamente contido dentro dele, então o apótema a de H deve ser maior ou igual a r. Se $a = r$, o robô deve ficar no centroide de H, e se $a > r$, o espaço geométrico que o robô pode assumir é um hexágono h de mesmo centro e apótema $a - r$, referido como região válida.

Como a média aritmética dos centros dos pixels p pode cair fora de h, é necessário limitar o quanto o alvo de cada hexágono pode se deslocar em relação a c, o centroide de h (que é o mesmo de H), clipando o vetor $p - c$. Essa operação é ilustrada na Figura 11. Se apenas os centros dos pixels dentro de h são considerados, a média aritmética cai dentro de h, pois h é convexo.

Se o número de robôs disponível R fosse ilimitado, bastaria usar $a = r$ e obter uma excelente representação da imagem, como ilustrado na Figura 12.

Entretanto, quando R é limitado, o tamanho do hexágono, que pode ser representado pelo apótema a, é escolhido adequadamente. Isso é feito por uma busca binária em a. Em cada passo da busca binária, alocam-se os robôs na grade hexagonal de apótema a. Se após a alocação, ainda restam robôs, pode-se diminuir a para obter uma representação com mais detalhes. Caso contrário, não há robôs suficientes para uma representação com aquela granularidade, então aumenta-se a, espaçando-se as coordenadas dos alvos.

Vale ressaltar que essa busca binária é aproximada, pois a função do número de robôs alocados em relação a a não é necessariamente monótona. Um exemplo disso é ilustrado na Figura 13.

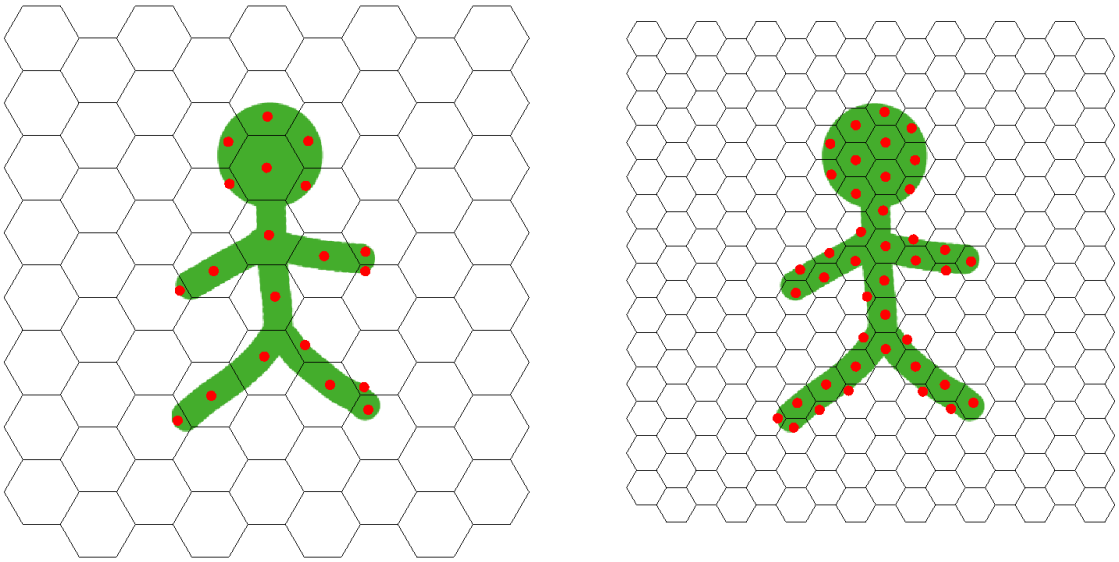
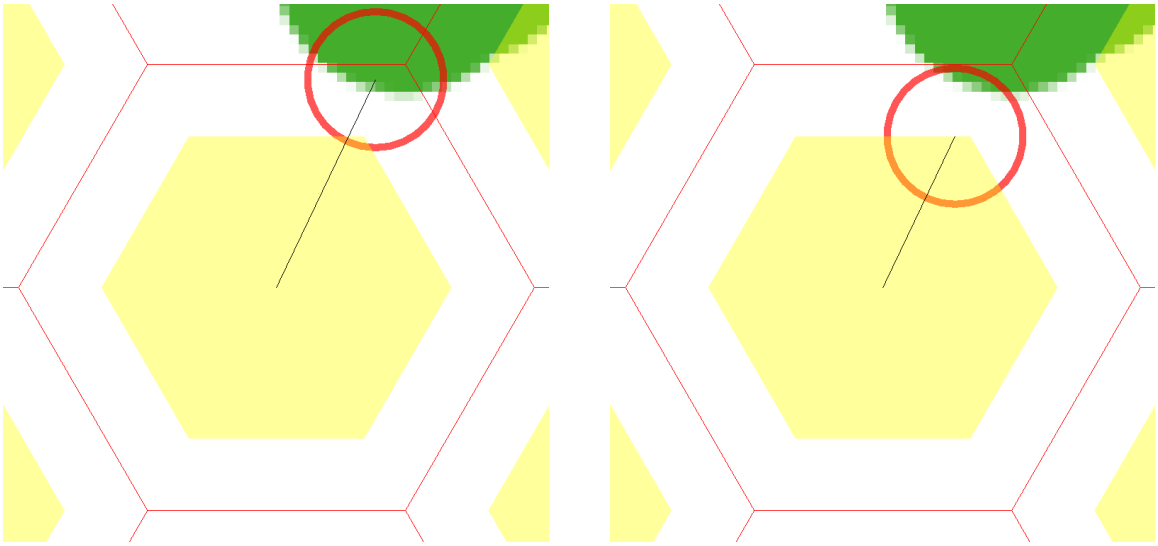


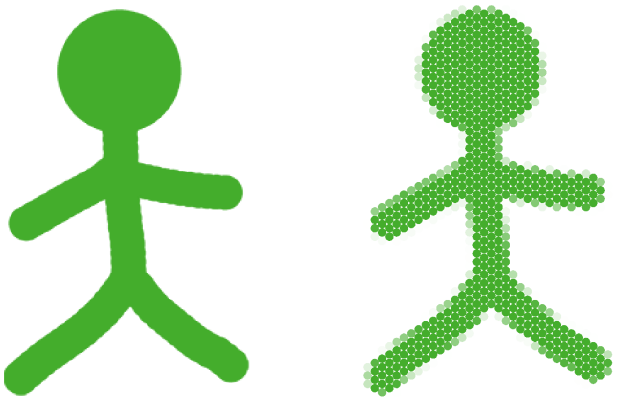
Figura 10 – Grade hexagonal adaptável ao número de robôs. Em geral, quanto mais robôs, mais detalhes podemos representar, então diminuímos o lado do hexágono.



(a) Exemplo em que a média aritmética dos centros dos pixels de um hexágono cai fora do hexágono interior h de apótema $a - r$, representado de amarelo.

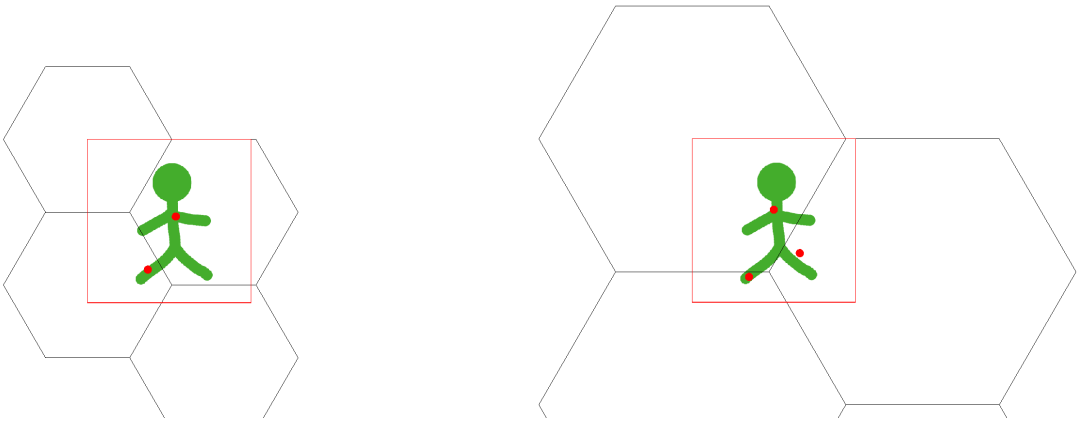
(b) Vetor após ser clipado. Sua direção se mantém, e seu comprimento é diminuido o mínimo possível para ficar dentro de h.

Figura 11 – Exemplo mostrando como a média aritmética, p, dos centros dos pixels de um hexágono H pode cair fora de sua região válida, então o vetor $p - c$, onde c é o centro de H, deve ser clipado adequadamente.



(a) Imagem original. (b) Alocação

Figura 12 – Quando o número de robôs é ilimitado (ou grande suficiente), a distribuição deles em uma grade hexagonal possibilita a melhor cobertura de um plano euclidiano 2D.



(a) Hexágono menor. (b) Hexágono maior.

Figura 13 – Exemplo incomum de como o tamanho do hexágono pode aumentar e o número de robôs alocados também. Note que, na implementação feita, o centro do hexágono mais em cima e à esquerda é coincidente com a quina de cima à esquerda da imagem, cuja borda está representada de vermelho.

Uma vantagem dessa heurística é a garantia da disjunção dos discos na alocação, enquanto outros métodos podem precisar de tratamento adicional para respeitar essa restrição. Por exemplo, no *Pixelbots* [10], é necessário redimensionar as posições alocadas para lidar com essas interseções, como descrito na Seção *Resizing of generated goal positions*.

3.1.2 Alocação dos Robôs aos Alvos

Uma vez definidos p pontos objetivos, $p \leq n$, decide-se o alvo de cada robô. Se $p < n$, $n - p$ robôs não são usados e assumem a cor do plano de fundo para não impactar a imagem final. Em um show de luzes de drones, isso seria equivalente àqueles robôs terem o LED desligado.

Nesse passo, o método desenvolvido modela os robôs como pontos, ou seja, desconsidera o espaço físico ocupado por eles. Assim, a alocação dos robôs pode ser estudada modelando o problema como um grafo bipartido, ou seja, um grafo que possui dois conjuntos independentes U e V, tais que toda aresta conecta um vértice em U a um vértice em V. Cada robô é representado por um vértice u do conjunto U, e cada ponto objetivo por um vértice v do conjunto V. Cada aresta $u_i v_j$ corresponde ao custo de atribuir o i-ésimo robô ao j-ésimo alvo.

3.1.2.1 Métricas

A seguir, são apresentadas as métricas usadas para avaliar a qualidade de uma alocação.

1. Minimizar a Soma das Arestas

   Um emparelhamento é um conjunto de arestas disjuntas, ou seja, nenhuma aresta tem vértice em comum com nenhuma outra. O problema de minimizar a soma das arestas de um emparelhamento máximo em um grafo bipartido pode ser resolvido em tempo polinomial $\mathcal{O}(n^3)$ [19, 20], usando o Algoritmo Húngaro [21]. O peso de cada aresta $u_i v_j$ corresponde à distância euclidiana entre o i-ésimo robô e o j-ésimo alvo.

   Pode-se mostrar, usando desigualdade triangular, que, em uma atribuição que minimiza a soma das arestas, duas arestas nunca se cruzam (mas podem ser coincidentes ou ter um ponto em comum se dois vértices puderem ter a mesma coordenada).

   Suponha que a atribuição gerou duas arestas R_1O_1 e R_2O_2 se cruzando em um ponto M, onde R_1 e R_2 são os robôs e O_1 e O_2 são os pontos objetivos. Primeiro, note que as arestas R_1O_2 e R_2O_1 não se cruzam, pois R_2 e O_2 estão em semiespaços diferentes em relação à reta R_1O_1. Então, R_1O_2 e O_1R_2 não possuem interseção. Resta mostrar que essa nova atribuição tem soma total menor que a atribuição inicial, que era $R_1O_1 + R_2O_2 = S$. Pela desigualdade triangular:

   $$R_1O_2 < R_1M + MO_2,$$
   $$R_2O_1 < R_2M + MO_1.$$

   Somando-se essas desigualdades, tem-se:

   $$R_1O_2 + R_2O_1 < R_1M + MO_2 + R_2M + MO_1 = S.$$

   Assim, trocar os alvos de R_1 e R_2 gerou uma atribuição ainda melhor, contradizendo a hipótese inicial. Logo, duas arestas nunca podem se cruzar em uma atribuição que otimiza essa métrica.

2. Minimizar a Maior Aresta

   Essa métrica é baseada na ideia de que, se não houvesse restrições geométricas no problema (robôs têm área, logo, se simplesmente seguissem reto, poderia haver colisão), o gargalo na geração das imagens estaria no tempo gasto pelo robô mais distante do seu alvo. O algoritmo desenvolvido para computar a alocação otimizando essa métrica usa como sub-rotina um algoritmo que calcula o emparelhamento máximo de um grafo bipartido.

   Como entrada, temos a robôs e b alvos, $b \leq a$. O número de vértices n do grafo associado é $a + b$ e o número de arestas m é ab. O emparelhamento máximo de um grafo bipartido é limitado pelo menor dos dois lados, neste caso b.

   O algoritmo ordena as m arestas e faz uma busca binária nas mesmas para encontrar o peso P, tal que exista um emparelhamento que sature[2] os alvos usando apenas as arestas de peso $\leq P$ e, se todas as arestas de peso $\geq P$ fossem eliminadas, tal emparelhamento não existiria.

   Em cada passo da busca binária, constrói-se um grafo bipartido usando-se apenas as arestas de peso menor ou igual a w, o peso da aresta sendo testada. Se esse grafo ainda tiver um emparelhamento máximo com b arestas, é possível encontrar uma alocação usando apenas as arestas de peso $\leq w$, então descartamos as arestas de peso $> w$. Caso contrário, temos que aumentar o peso de w, pois não foi possível alocar cada ponto objetivo a um robô diferente usando apenas as arestas de peso $\leq w$.

   Considere que a complexidade de construir o grafo bipartido em cada iteração é $M = \mathcal{O}(n+m)$. O número de iterações da busca binária é $B = \mathcal{O}(\log(m))$, e o custo inicial do algoritmo de ordenar as arestas é $I = \mathcal{O}(m \log(m))$.

   Seja C a complexidade do algoritmo que encontra o emparelhamento máximo em um grafo bipartido, o custo final do algoritmo é $\mathcal{O}(I + B(M + C))$. O algoritmo de Hopcroft-Karp [22] encontra o emparelhamento máximo de um grafo bipartido em tempo $\mathcal{O}(\sqrt{n}m)$. Se usarmos esse algoritmo em C, o custo final é $\mathcal{O}(m \log(m) + log(m)(n + m + \sqrt{n}m)) = \mathcal{O}(m \log(m) + log(m)n + log(m)m + log(m)\sqrt{n}m)$. Como $n = \mathcal{O}(m)$, a complexidade final é $\mathcal{O}(\sqrt{n}m \log(m))$.

   No Pseudocódigo 1, usamos a notação 0 indexada para vetores. A função "getGraphWithEdgesUpToWeight(w)"constrói o grafo usando apenas as arestas de peso $\leq w$, e "T.getMaxMatchingSize()"retorna o número de arestas no emparelhamento máximo de T.

[2] Em um emparelhamento, diz-se que um lado está saturado se todos os seus vértices estão em alguma aresta do emparelhamento.

Pseudocódigo 1 Minimizar a maior aresta

Input: Grafo bipartido completo G dos agentes e alvos, | agentes |$\geq$| alvos |
Output: Emparelhamento saturando o lado dos alvos minimizando a maior aresta

```
1: E ← sorted(G.edges())      ▷ Vetor com arestas ordenadas em ordem não decrescente
2: l ← 0
3: r ← E.size()
4: while l ≠ r do
5:     m ← (l + r)/2
6:     T ← G.getGraphWithEdgesUpToWeight(E[m])
7:     if T.getMaxMatchingSize() == b then
8:         r ← m
9:     else
10:        l ← m + 1
11:    end if
12: end while
13: return T
```

3. Minimizar a Soma das Arestas Usando Apenas as de Peso Menor ou Igual ao da Maior Aresta Encontrada na Métrica Anterior

   Por fim, foi proposta uma métrica que junta as duas anteriores. Primeiro, usa-se o algoritmo da Métrica 2 para encontrar o menor peso possível da maior aresta P. Depois, usando-se apenas as arestas de peso menor ou igual a P, aplica-se a Métrica 1. Nesse caso, perde-se a propriedade de que duas arestas nunca se cruzam, pois foi mostrado que, se isso acontecesse, seria possível gerar uma solução melhor, trocando os alvos dos robôs. Porém, a Métrica 2, aplicada inicialmente, elimina algumas arestas, logo não é sempre possível realizar essa troca.

3.1.2.1.1 Sobre a Variância das Distâncias nas Métricas

As três métricas apresentadas também foram avaliadas usando as distâncias ao quadrado. Intuitivamente, o efeito disso é penalizar mais distâncias longas, gerando alocações com distâncias mais parecidas, ou seja, variância menor. A seguir, é provado que a variância encontrada na alocação que minimiza a soma de cada aresta ao quadrado é menor ou igual àquela encontrada minimizando a soma das mesmas.

Seja n o número de alvos, ou seja, arestas/distâncias na alocação final, x_i a i-ésima distância, $s = \sum_i^n x_i$ a soma das distâncias, $S = \sum_i^n x_i^2$ a soma das distâncias ao quadrado, $\bar{x} = \frac{\sum_i^n x_i}{n}$ a média das distâncias, $V = \frac{\sum_i^n (x_i - \bar{x})^2}{n}$ a variância, tem-se:

$$
\begin{aligned}
V &= \frac{\sum_i^n (x_i - \bar{x})^2}{n} \\
&= \frac{1}{n}\sum_i^n \left(x_i - \frac{x_1 + ... + x_n}{n}\right)^2 \\
&= \frac{1}{n}\sum_i^n \left(\frac{nx_i}{n} - \frac{x_1 + ... + x_n}{n}\right)^2 \\
&= \frac{1}{n}\sum_i^n \left(\frac{nx_i - s}{n}\right)^2 \\
&= \frac{1}{n^3}\sum_i^n (nx_i - s)^2 \\
&= \frac{1}{n^3}\sum_i^n \left(n^2x_i^2 - 2nx_i s + s^2\right) \\
&= \frac{1}{n^3}\left(n^2\sum_i^n x_i^2 - 2ns\sum_i^n x_i + \sum_i^n s^2\right) \\
&= \frac{1}{n^3}\left(n^2 S - 2ns^2 + ns^2\right) \\
&= \frac{S}{n} - \frac{s^2}{n^2}.
\end{aligned}
$$

Note que poderíamos ter usado direto o resultado da Estatística:

$$\mathrm{Var}(X) = \mathbb{E}[X^2] - \mathbb{E}[X]^2,$$

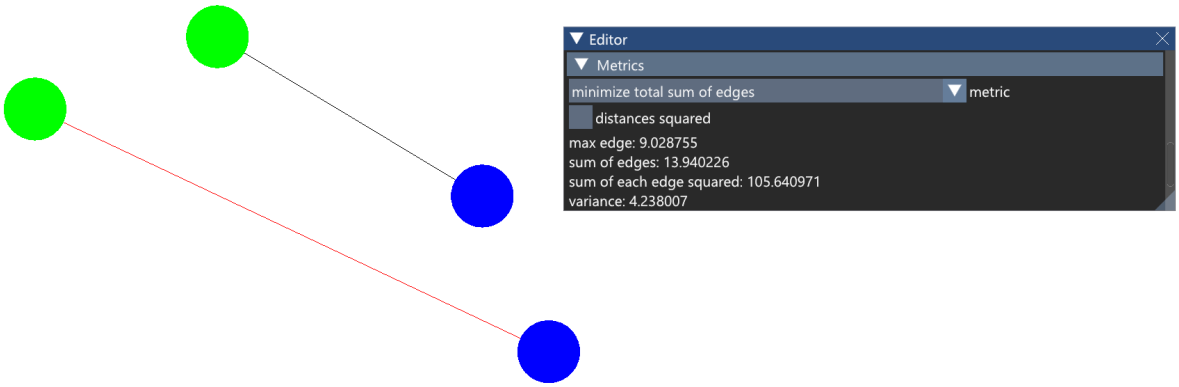
onde $\mathbb{E}[X^2] = \frac{S}{n}$ e $\mathbb{E}[X]^2 = \frac{s^2}{n^2}$.

Sejam s_1 e S_1 os valores encontrados para s e S minimizando a soma das distâncias, e s_2 e S_2 os encontrados para s e S minimizando a soma das distâncias ao quadrado, sabemos que $s_1 \leq s_2$, pois s_1 é o menor s possível, e $S_2 \leq S_1$, pois S_2 é o menor S possível. Resta mostrar que $\frac{S_2}{n} - \frac{s_2^2}{n^2} \leq \frac{S_1}{n} - \frac{s_1^2}{n^2}$:

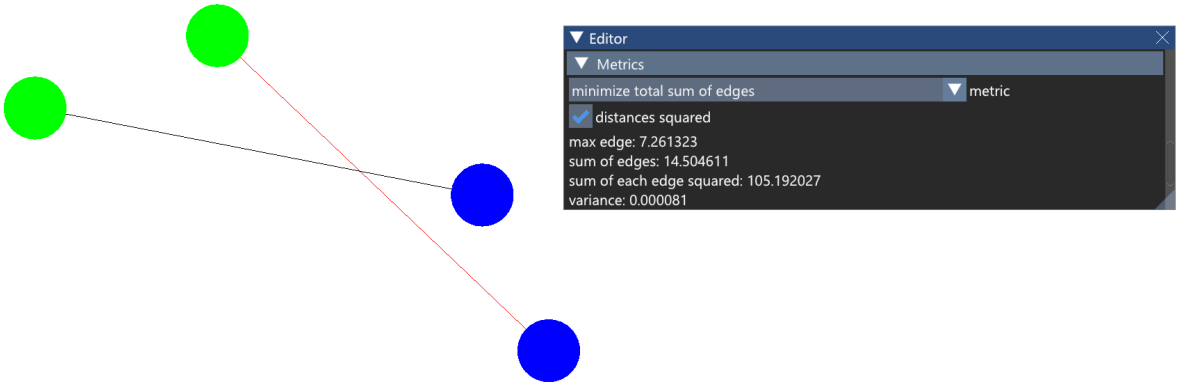
$$
\begin{aligned}
\frac{S_2}{n} - \frac{s_2^2}{n^2} &\leq \frac{S_1}{n} - \frac{s_1^2}{n^2} \\
\frac{S_2}{n} - \frac{S_1}{n} &\leq \frac{s_2^2}{n^2} - \frac{s_1^2}{n^2} \\
\frac{S_2}{n} - \frac{S_1}{n} \leq 0 &\leq \frac{s_2^2}{n^2} - \frac{s_1^2}{n^2}\ .
\end{aligned}
$$

3.1.2.1.2 Exemplos

As Figuras 14 e 15 exemplificam alocações usando as métricas. Os agentes estão de azul, os alvos de verde, as arestas, que ligam cada agente a sua alocação, de preto. A maior aresta (ou as maiores arestas, caso haja empate) da alocação é destacada de vermelho.

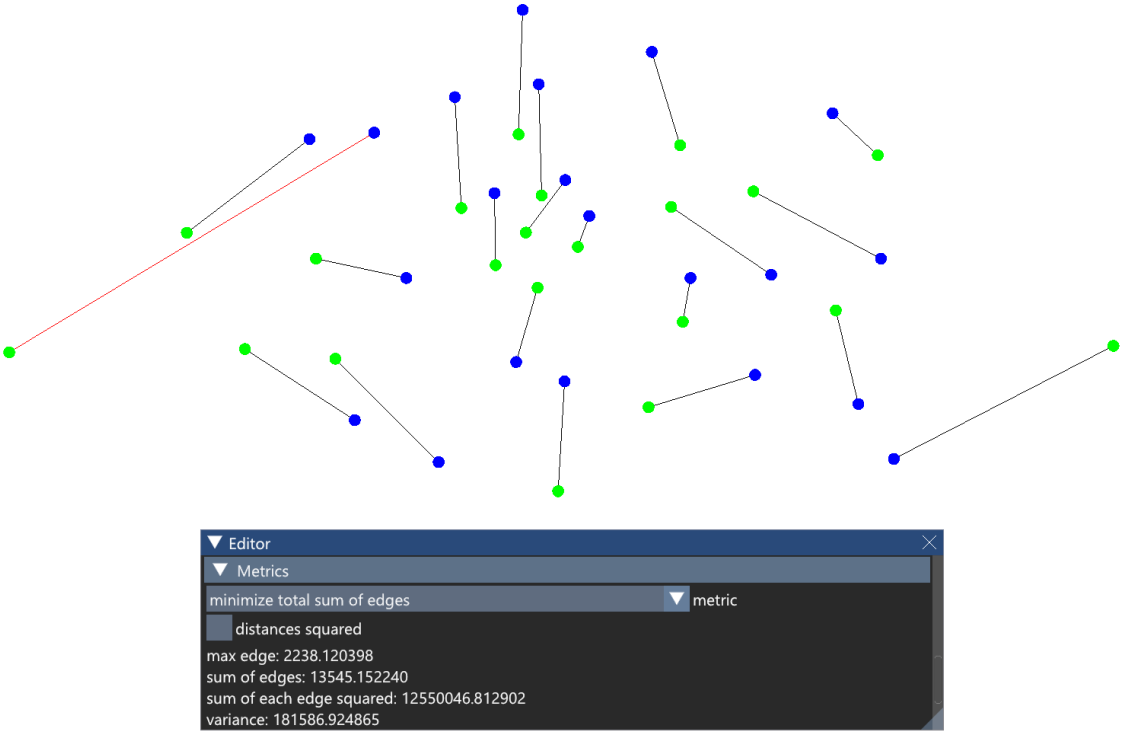


(a) Métrica 1.

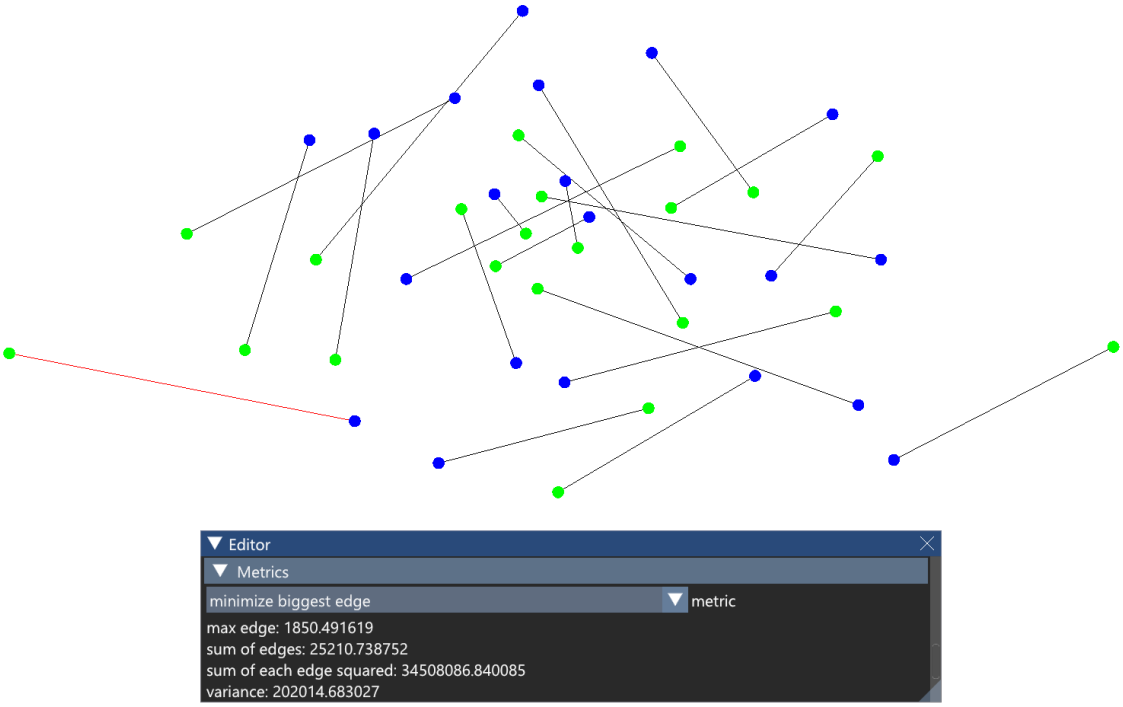


(b) Métrica 1 usando distâncias ao quadrado.

Figura 14 – Exemplo com apenas 2 agentes e 2 alvos para salientar as propriedades: minimizar a soma total das arestas resulta em uma alocação sem arestas se cruzando; a variação que usa as distâncias ao quadrado sempre tem a variância do comprimento das arestas menor ou igual à variação padrão, que usa as distâncias elevadas a 1.

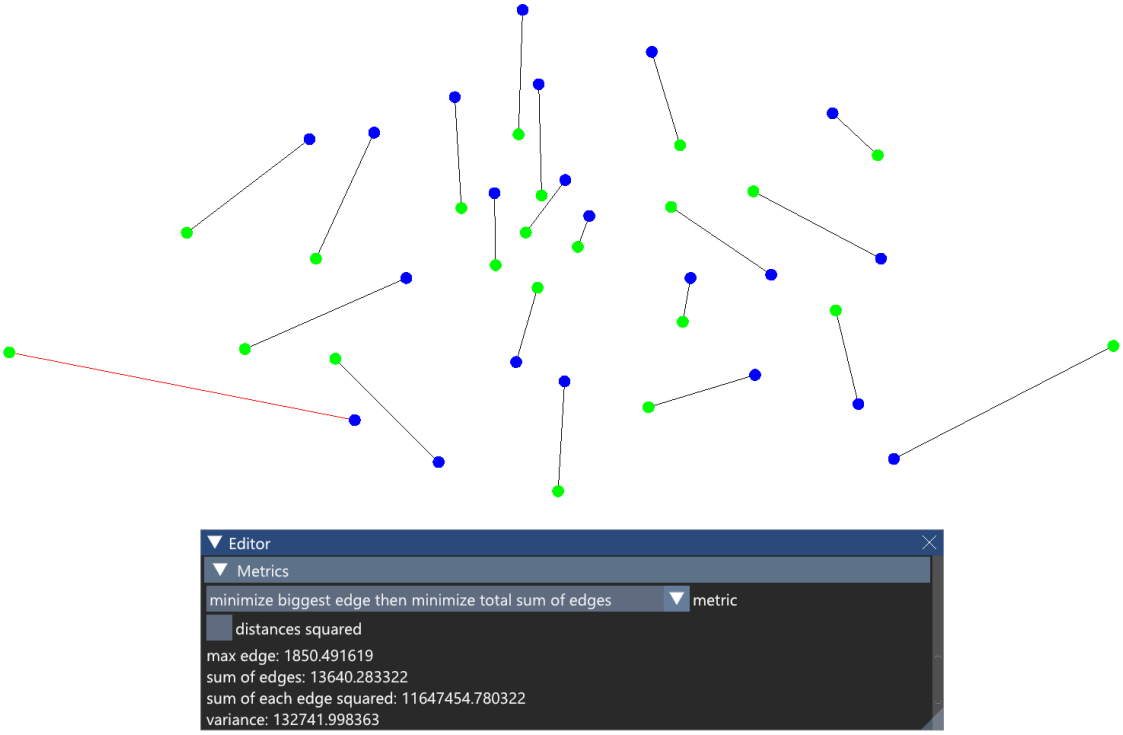


(a) Métrica 1.

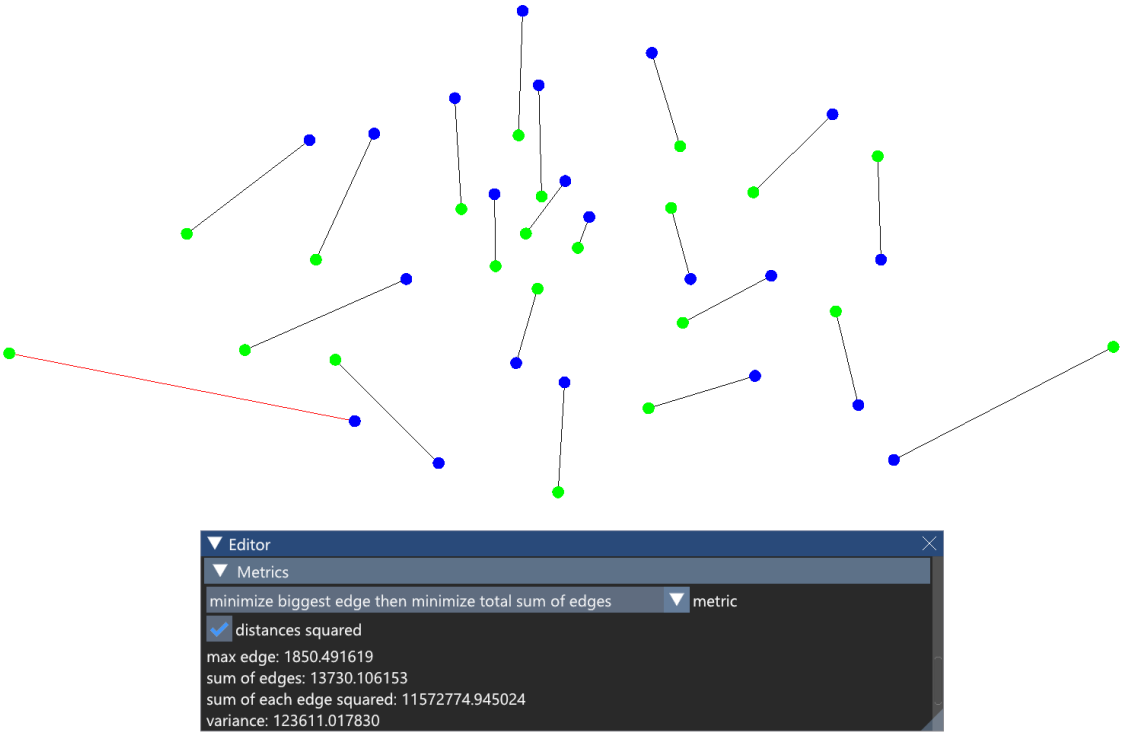


(b) Métrica 2.

Figura 15 – Exemplo das alocações em uma configuração com 21 agentes e 21 alvos.



(c) Métrica 3.



(d) Métrica 3 com distâncias ao quadrado.

Figura 15 – Exemplo das alocações em uma configuração com 21 agentes e 21 alvos.

3.2 Navegação

A navegação consiste em, dadas as posições iniciais e finais preestabelecidas de cada robô, executar a movimentação no espaço de forma que não haja colisões, respeitando as restrições cinemáticas dos agentes. Essa tarefa é extensamente estudada em robótica, mas ainda não tem uma solução perfeita.

Neste trabalho, os robôs são considerados holonômicos e possuem apenas dois graus de liberdade, referentes aos eixos de posição 2D.

Foi usada a técnica *Optimal Reciprocal Collision Avoidance* - ORCA [12] para navegar os agentes. A seguir, é dada uma visão geral.

3.2.1 Optimal Reciprocal Collision Avoidance (ORCA)

O ORCA [12] é um sistema de prevenção de colisões que é executado localmente por cada robô, em que todos devem seguir o mesmo protocolo para garantir a corretude (ausência de colisões). Assume-se que cada robô, dentro de um certo raio de observação, tem a percepção perfeita da posição, da forma e da velocidade dos obstáculos (incluindo os outros robôs) no ambiente. O ORCA é baseado no conceito de obstáculo de velocidade, apresentado a seguir.

3.2.1.1 Obstáculo de Velocidade

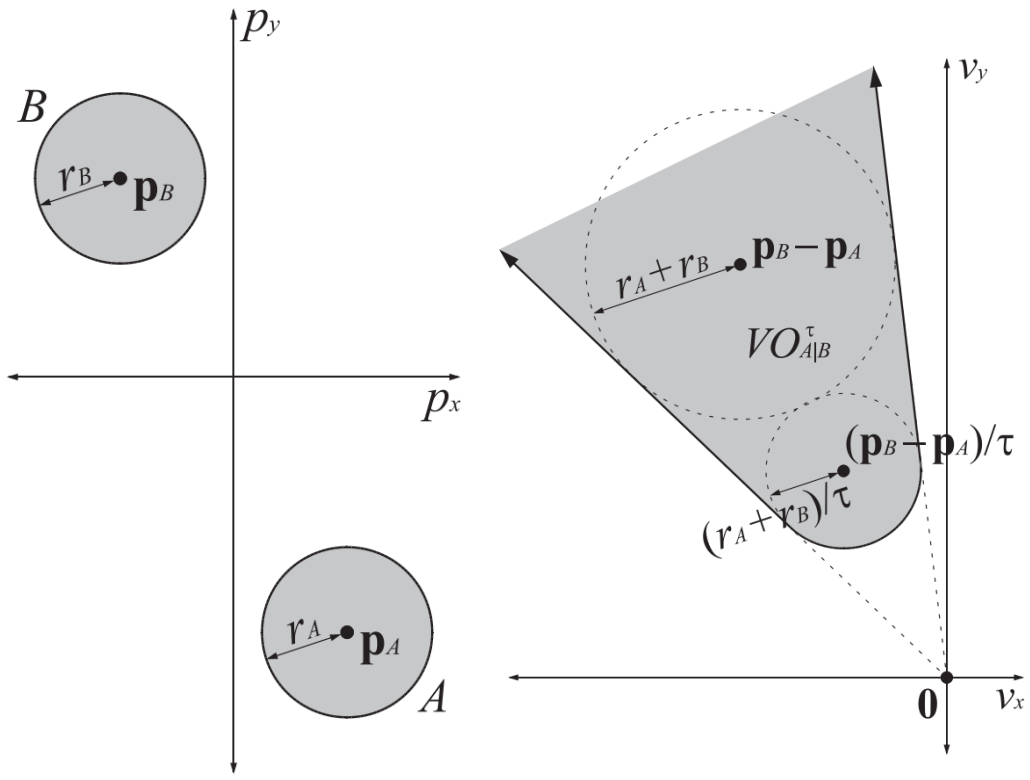
O obstáculo de velocidade $VO^{\tau}_{A|B}$ (lê-se obstáculo de velocidade de A induzido por B em uma janela de tempo τ) é o conjunto de todas as velocidades relativas de A em relação a B que resultariam em uma colisão entre A e B em algum momento antes do tempo τ. Seja $D(p, r) = \{q \mid \|q - p\| < r\}$ um disco de raio r centrado em p, define-se:

$$VO^{\tau}_{A|B} = \{v \mid \exists t \in [0, \tau] :: tv \in D(p_B - p_A, r_A + r_B)\}.$$

Geometricamente, $VO^{\tau}_{A|B}$, ilustrado na Figura 16, pode ser visto como um cone truncado, com seu ápice (único vértice de um cone) na origem (ponto de coordenada $(0, 0)$), e pernas tangentes ao disco de raio $r_A + r_B$ centrado em $p_B - p_A$. A quantidade do truncamento depende da janela de tempo τ. O cone é truncado por um arco de um disco de raio $\frac{r_A + r_B}{\tau}$ centrado em $\frac{p_B - p_A}{\tau}$. Note que quanto maior o tempo τ, menor o truncamento, então mais o obstáculo de velocidade se aproxima ao cone não truncado.

3.2.1.2 Cálculo das Velocidades que Não Resultam em Colisão

Se a velocidade v_A do robô A é inválida em relação ao robô B, ou seja, dentro de $VO^{\tau}_{A|B}$, calcula-se a menor mudança possível u que deve ser somada a v_A para evitar uma colisão com B. Cada robô é responsável por uma porcentagem do módulo de u, que deve somar 100% no total. Essa característica recíproca do algoritmo promove movimentos suaves e



(a) Posições e raios dos robôs.

(b) Espaço geométrico do obstáculo de velocidade $VO^{\tau}_{A|B}$ para $\tau = 2$.

Figura 16 – Exemplo do obstáculo de velocidade de 2 robôs.

sem oscilações. A porcentagem da responsabilidade na evitação da colisão é controlada por um parâmetro c, em que $c = \frac{1}{2}$ corresponde ao caso em que ambos são igualmente responsáveis por evitá-la. Daí vem o requerimento de seguir o mesmo protocolo.

Formalmente, seja v_A^{pref} a velocidade preferencial do robô A, $k = v_A^{\text{pref}} - v_B^{\text{pref}}$ a velocidade preferencial relativa de A em relação a B, temos:

$$u = (\arg\min_{v \in \text{VO}^{\tau}_{i|j}} \|v - k\|) - k.$$

Em uma implementação descentralizada, em que os outros robôs não conhecem a velocidade instantânea preferencial de A, usa-se a última leitura da velocidade de A.

Seja n a normal apontando para fora da borda de $\text{VO}^{\tau}_{A|B}$ no ponto $u + k$, o espaço de velocidades permitidas de A em relação a B é o semiplano

$$\text{ORCA}^{\tau}_{A|B} = \{v | (v - (v_A^{\text{pref}} + cu) \cdot n) \geq 0)\}.$$

Seja $S = D(0, v_A^{\max})$ o conjunto de velocidades possíveis de A naquele instante de tempo, em que $v_A^{\max}$ representa a velocidade máxima de A, ou seja, consideramos A holonômico, capaz de assumir qualquer velocidade de módulo menor ou igual à sua velocidade máxima a qualquer instante. O conjunto de velocidades que garantem a ausência de colisões em um horizonte de tempo τ é

$$\text{ORCA}^{\tau}_i = S \cap \bigcap_{j \neq i} \text{ORCA}^{\tau}_{i|j}.$$

Por fim, a nova velocidade ótima de A, v_A^{new}, é a velocidade v válida mais próxima de sua preferencial:

$$v_A^{\text{new}} = \arg\min_{v \in \text{ORCA}_A^\tau} \|v - v_A^{pref}\|. \tag{3.1}$$

Como ORCA_A^τ é uma região convexa limitada por restrições lineares, correspondentes aos semiplanos de velocidades permitidas em relação a cada um dos outros robôs, e a função objetivo 3.1 possui apenas um mínimo local, como salientado em [12], o problema pode ser eficientemente resolvido com técnicas de programação linear.

3.2.1.3 Considerações Finais

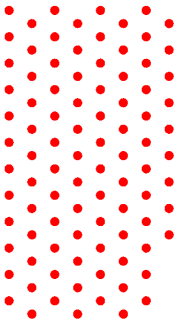
No *paper* original, também são considerados obstáculos estáticos. Nesse caso, os robôs são 100% responsáveis por evitar a colisão. Porém, não há obstáculos estáticos neste trabalho.

O algoritmo possui vários parâmetros, como o número de vizinhos máximo considerado por cada agente, o raio de observação de cada agente e número de iterações consideradas por segundo. Como este artigo não descreve todos esses detalhes de implementação, os parâmetros escolhidos nos testes apresentados na Seção 3.2.1.4 não foram reportados.

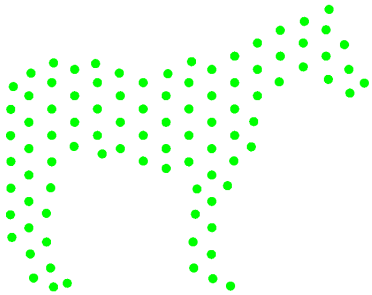
Uma limitação do ORCA é que não funciona bem em ambientes muito densos de robôs (congestionados), pois o programa linear 2D resultante pode ser insatisfazível, ou seja, a interseção dos semiplanos é vazia. Isso impossibilita os robôs de convergirem, ou seja, chegarem a seus alvos. No trabalho original [12], são discutidas algumas improvisações (gambiarras) que podem ser feitas nessas situações, mas a convergência não é garantida.

3.2.1.4 Testes

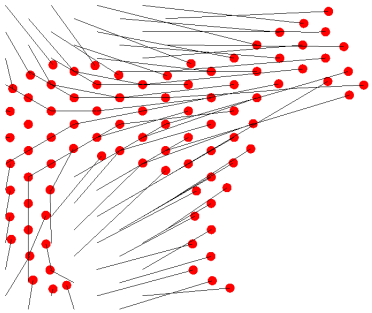
As Figuras 17 e 18 mostram, respectivamente, configurações em que o ORCA funciona bem e mal. Os robôs são representados de vermelho, os alvos de verde, e a alocação de cada um por uma aresta preta. Reconhece-se que uma imagem estática não passa bem a ideia de algo executando-se ao longo do tempo. Até considerou-se mostrar vários ângulos de gráficos 3D mostrando a evolução das posições 2D ao longo do tempo, mas por também ficar confuso, isso não foi incluído.



(a) Posições iniciais.

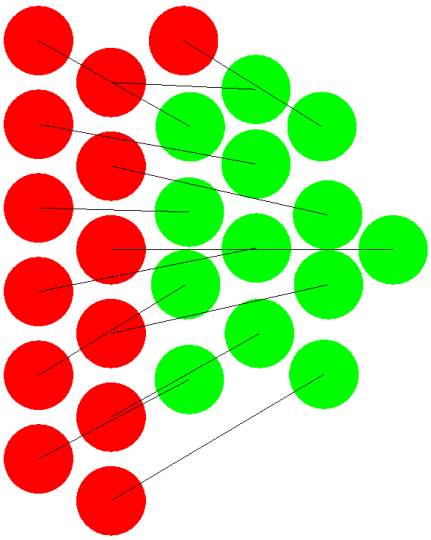


(b) Posições finais desejadas.

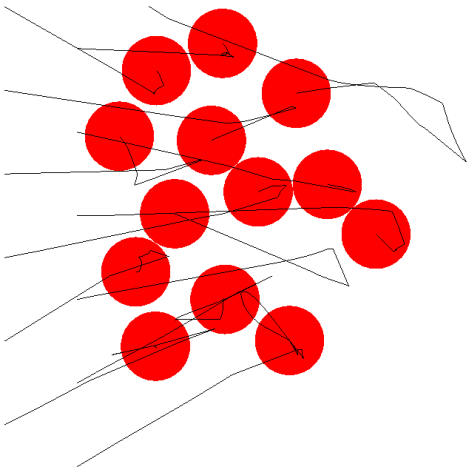


(c) Trajetórias reais executadas.

Figura 17 – Exemplo em que o ORCA funciona bem.



(a) Alocação inicial.



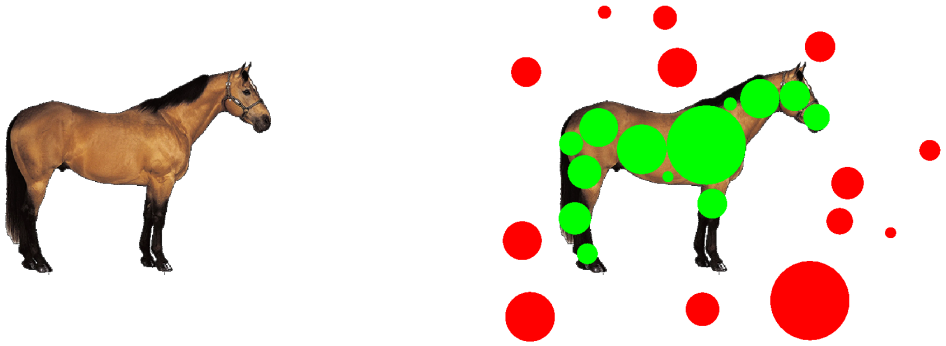
(b) Trajetórias reais executadas.

Figura 18 – Exemplo em que o ORCA funciona mal. Por ser um algoritmo reativo, não há um planejamento capaz de lidar com o congestionamento gerado em configurações apertadas, então os robôs não convergem para as suas posições desejadas.

4 Representações com Robôs Heterogêneos

Embora o navegador ORCA suporte agentes de diferentes velocidades máximas e raios, a alocação baseada na grade hexagonal da Seção 3.1.1.2 suporta apenas raios iguais. Uma forma de lidar com essa limitação é determinar que o apótema do hexágono seja maior ou igual ao maior raio de todos, mas isso seria bastante ineficiente. Por exemplo, na situação extrema em que há um robô de raio muito maior que todos os outros, os hexágonos dos robôs menores teriam muito espaço inaproveitado.

Esta seção dedica-se a relatar formas investigadas durante o POC II de estender os métodos para robôs de diferentes tamanhos (raios), ou seja, heterogêneos, como ilustrado na Figura 19.



(a) Imagem alvo. (b) Alocação.

Figura 19 – Alocação das posições finais (círculos verdes) dos robôs heterogêneos (círculos vermelhos).

Inicialmente, uma forma clássica da robótica de cobertura de área com sistemas multiagentes foi adaptada para o problema em questão. Essa técnica, descrita na Seção 4.1, baseia-se em utilizar diagramas de Voronoi para alocar e navegar os robôs simultaneamente.

Posteriormente, na Seção 4.2, foi desenvolvida uma maneira de alocar as posições alvos dos robôs disponíveis completamente *a priori*, que envolveu discretizar um problema contínuo através de amostragens, formalizar o problema de otimização associado e aproximar sua solução através de uma heurística.

4.1 Controle de Cobertura por Diagramas de Voronoi

Um Diagrama de Voronoi é uma construção geométrica que possui inúmeras generalizações. No caso em questão, usamos sua versão mais comum: dado um conjunto de n pontos em coordenadas distintas no plano 2D, o Diagrama de Voronoi é a partição do plano em n regiões, também referidas como células, tal que cada coordenada pertence à

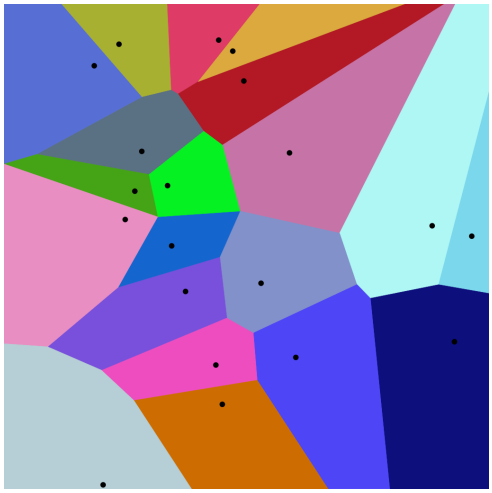


Figura 20 – Exemplo de um diagrama de Voronoi [23] em duas dimensões mantido dentro de um retângulo, formado a partir dos pontos de preto.

região do ponto mais próximo a ela. A medida mais usual de proximidade é a distância Euclidiana, então é comumente omitida. É possível notar que as células nas bordas são infinitas, então, na prática, elas são recortadas para serem mantidas dentro de regiões predefinidas, como um retângulo, como usado neste trabalho. Um exemplo é ilustrado na Figura 20.

Diagramas de voronoi podem ser computados eficientemente, em tempo $\mathcal{O}(n \log(n))$, por meio de algoritmos como o de Fortune [24].

Esses diagramas possuem diversas aplicações. Em robótica, já foram usados para realizar controle de cobertura de uma área por múltiplos agentes através do algoritmo de Lloyd, apresentado a seguir.

4.1.1 Algoritmo de Lloyd

O algoritmo de Lloyd é um algoritmo incremental: em cada passo, calcula-se o diagrama de Voronoi dos agentes e cada um deles movimenta-se um pequeno passo em direção ao centro de gravidade de sua célula, aqui chamado vetor de deslocamento. Essa ideia é ilustrada na Figura 21, e a Figura 22 mostra a evolução ao longo de algumas iterações.

Quando todos os pontos de um polígono possuem a mesma densidade, ou seja, a massa é uniformemente distribuída, o centro de gravidade é igual ao centro geométrico, chamado centroide.

Quando todos os agentes estão no centro de gravidade de sua célula, diz-se que o algoritmo convergiu. Muito já foi estudado sobre a convergência desse algoritmo [25, 26]. Usualmente, quando todos os agentes estão razoavelmente perto de seus alvos, termina-se a execução do algoritmo.

Em cada iteração, seja p_i a coordenada do i-ésimo agente, c_i a coordenada do centro de gravidade de sua célula e m um fator multiplicativo que controla o tamanho do passo,

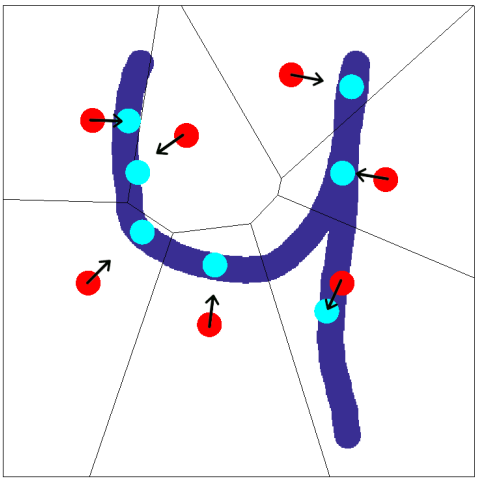
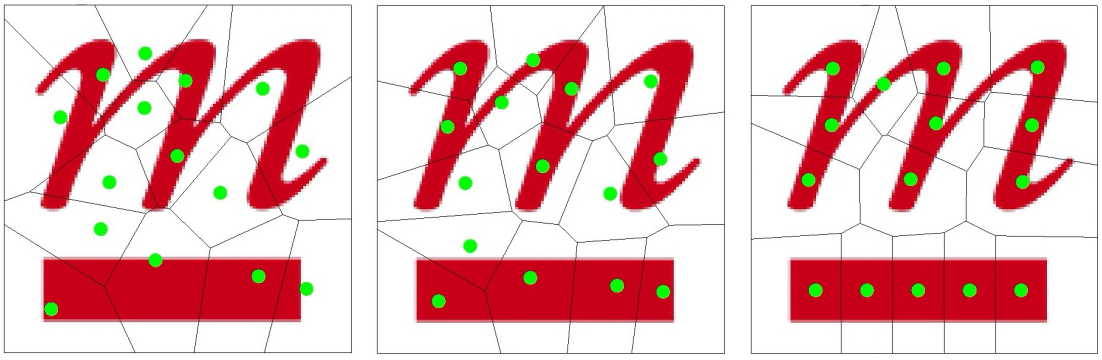


Figura 21 – De vermelho, os robôs. De preto, as arestas das células do diagrama de Voronoi gerado a partir das coordenadas dos centros dos robôs. De azul claro, os centros de gravidade de cada célula. As setas pretas mostram o vetor de deslocamento de cada robô, que é percorrido na iteração.



(a) Iteração 1. (b) Iteração 2. (c) Iteração 3.

Figura 22 – Exemplo de iterações do algoritmo de Lloyd em ordem cronológica, não necessariamente com a mesma variação de tempo entre as imagens.

o vetor de deslocamento v_i é

$$v_i = \begin{cases} \overrightarrow{0} & \text{se } p_i = c_i \\ m\frac{c_i - p_i}{\|c_i - p_i\|} & \text{caso contrário} \end{cases}. \tag{4.1}$$

Ou seja, o robô se mantém parado se já convergiu, caso contrário, anda, naquela iteração, m unidades de distância em direção ao centro de gravidade de sua célula.

Na implementação feita, no intuito de evitar colisões entre os agentes, foi usado o clássico método de campos potenciais: cada robô, dentro de um certo raio de observação, é capaz de perceber os obstáculos, no caso os outros robôs, e são modeladas forças de repulsão adequadas para garantir que sempre estejam a uma distância maior ou igual à soma de seus raios. Formalmente, o vetor de repulsão f_r que um robô de raio r_2 na

coordenada p_2 exerce em um robô de raio r_1 na coordenada p_1, $p_1 \neq p_2$, é

$$f_r = \frac{p_1 - p_2}{d} \frac{1}{c_1(d - r_2 - r_1)^{c_2}}, \tag{4.2}$$

onde $d = \|p_1 - p_2\|$, c_1, c_2 são constantes maiores que 0 que controlam a intensidade da força de repulsão entre os agentes dependendo da distância entre eles. Em teoria, essas constantes poderiam ser calibradas levando-se em conta características dos agentes, como a capacidade de desaceleração e mudança de direção, os raios de observação e as velocidades máximas, mas na prática são escolhidas empiricamente.

Essas ideias são resumidas no Pseudocódigo 2.

Pseudocódigo 2 Algoritmo de Lloyd

procedure AlgLloyd(P, F)
    Input: Posições P dos robôs, figura alvo F
    while Running **do** ▷ A variável Running é controlada externamente por outra interface. Pode-se usar outra condição de parada, como checar se todos os robôs já estão perto suficiente de seus alvos
        $P \leftarrow IteraçãoLloyd(P, F)$
    end while
end procedure
procedure IteraçãoLloyd(I, F)
    Input: Posições I dos robôs no início da iteração, figura F
    Output: Posições R dos robôs no final da iteração
    $V \leftarrow voronoi(I)$ ▷ Diagrama de Voronoi dos pontos I, onde V_i representa a célula do i-ésimo robô
    $R \leftarrow [] * |I|$ ▷ Inicializa o vetor das coordenadas finais, de tamanho $|I| = |V|$
    for $i = 1..N$ **do**
        if a região V_i em F tiver massa $\neq 0$ **then**
            $C \leftarrow CentroDeGravidade(V_i, F)$ ▷ Calcula-se o centro de gravidade da região, em F, delimitada por V_i
            $R_i \leftarrow VetorDeslocamento(i, C)$ ▷ calcula-se o vetor de deslocamento, como o descrito em 4.1
        else
            $R_i \leftarrow I_i$ ▷ Se a célula for vazia (sem massa), convencionou-se que o robô permanece parado
        end if
        for $j = 1..N, j \neq i$ **do** ▷ Forças de repulsão
            if Robô j estiver dentro do raio de observação do i **then**
                $X_{rep} \leftarrow ForçaRepulsão(i, j)$ ▷ calcula-se uma força de repulsão, como a descrita em 4.2
                $R_i \leftarrow R_i + X_{rep}$
            end if
        end for
    end for
    return F
end procedure

A seguir, é discutido como computar o centro de gravidade das células em cada iteração.

4.1.2 Cálculo do Centro de Gravidade de Cada Célula

O centro de gravidade c_g de um sistema, de massa total T, que possui P partículas, onde a i-ésima partícula possui massa m_i e está na coordenada c_i, é

$$c_g = \frac{\sum_{i=1}^{P} m_i c_i}{T} \ .$$

Note que essa expressão pode ser calculada independentemente para cada dimensão do sistema (e.g. eixo x, eixo y). No caso contínuo, como um polígono, as definições são as mesmas, mas usando-se integrais em vez de somatórios. Seja $d(c)$ a densidade de massa na coordenada c, tem-se:

$$c_g = \frac{1}{T} \int_c c d(c) \mathrm{dC} \ .$$

O centro geométrico de um sistema, também chamado centroide, é definido como o centro de gravidade quando todas as partículas do sistema possuem a mesma massa, ou seja, a densidade é uniforme. Em um polígono simples[1] de n pontos, apresentados na ordem de ocorrência ao longo do seu perímetro, onde x_n e y_n referem-se a x_0 e y_0 (ou seja, de adjacente em adjacente até voltar ao inicial), o centroide $C = (C_x, C_y)$ é [27]:

$$\begin{aligned}
A &= \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i), \\
m &= \frac{1}{6A}, \\
C_x &= m \sum_{i=0}^{n-1} (x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i), \\
C_y &= m \sum_{i=0}^{n-1} (y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i).
\end{aligned}$$

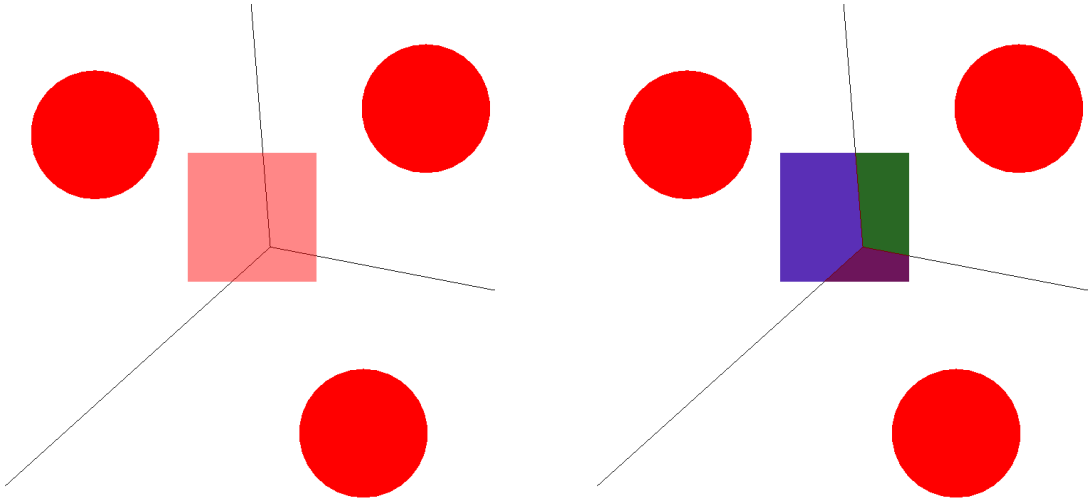
A é a área "com sinal"do polígono. Se os pontos estão na ordem anti-horária, $A > 0$, caso contrário (horária), $A < 0$. A também será usado no algoritmo.

A imagem é uma matriz M de dimensões $n \times m$. Cada entrada $M_{i,j}$ corresponde à massa de um pixel da imagem, um quadrado alinhado nos eixos da imagem de lado 1 e centro $C = (i + 0.5, j + 0.5)$. Dentro de cada pixel a massa é uniformemente distribuída.

Um quadrado $M_{i,j}$ pode intersectar várias células, como ilustrado na Figura 23, então é necessário definir sua contribuição para cada uma delas. Poderia-se considerar, por exemplo, que apenas uma dessas células intersectadas recebe totalmente a massa de M_{ij}, como a que possui a maior área, ou que engloba o centro do pixel. Porém, foi implementada a solução precisa, a qual segmenta o quadrado nos polígonos exatos que pertencem a cada célula.

A operação de unir o centro de gravidade de dois objetos será usada nas seções seguintes, então é revisada aqui: sejam c_1 e c_2 os centros de gravidade de cada um deles, e m_1 e m_2

[1] Um polígono é simples se não possui autointerseções ou buracos.



(a) Exemplo de um pixel (quadrado vermelho) intersectando várias células do diagrama de Voronoi.

(b) Segmentação do pixel nos polígonos, representados de cores diferentes, correspondentes à interseção com cada célula.

Figura 23 – Como os pixels podem intersectar várias células, são ainda segmentados para contribuírem a massa correta para cada uma delas.

suas respectivas massas. O centro de gravidade dos dois juntos é

$$\frac{c_1 m_1 + c_2 m_2}{m_1 + m_2} \ . \tag{4.3}$$

A seguir, são apresentados dois algoritmos para computar o centro de gravidade de cada célula após a construção do diagrama de Voronoi. O número de pontos/células é denotado por r.

4.1.2.1 Cálculo dos Centros de Gravidade Sem Pré-Processamento da Imagem

Para cada célula R_i, itera-se por cada pixel j e calcula-se o polígono I_{ij} formado pela interseção de R_i com o pixel j. Sejam m_{ij} e c_{ij} a massa e o centroide de I_{ij} respectivamente. O centro de gravidade C_i de cada R_i é

$$C_i = \frac{\sum_j c_{ij} m_{ij}}{\sum_j m_{ij}} \ .$$

A iteração por cada pixel e célula já é $\Theta(nmr)$. Como há a operação de interseção, não aprofundada e explicitamente contabilizada aqui, a complexidade é $\Omega(nmr)$, o que não é eficiente suficiente para executar em tempo real. Dessa forma, foi necessário desenvolver um algoritmo mais eficiente, apresentado a seguir.

4.1.2.2 Cálculo dos Centros de Gravidade Com Pré-Processamento da Imagem

Como a imagem é estática, é possível calcular algumas somas de prefixo que possibilitam encontrar o centro de gravidade de qualquer coluna na matriz em $\mathcal{O}(1)$. Assim, é possível

segmentar a célula em colunas, calcular o centro de gravidade de cada uma delas e juntá-los para encontrar o centro de gravidade da célula inteira, operações detalhadas a seguir.

Considere o indexamento das células apresentado na Figura 24. Para todo $j \in [0, m)$, compute as seguintes somas de prefixo:

$$s_{ij} = \sum_{y=0}^{j} M_{i,y} \ , \tag{4.4}$$

$$p_{ij} = \sum_{y=0}^{j} M_{i,y}(0.5 + y) \ . \tag{4.5}$$

Note que $(0.5 + y)$ corresponde à coordenada no eixo y do centro de gravidade da célula $M_{x,y} \forall x$. O centro de gravidade $C_{i,j}$ de qualquer retângulo $M_{i,0}, M_{i,1}, ...M_{i,j}$ pode ser calculado em $\mathcal{O}(1)$:

$$C_{i,j} = (i + 0.5, \frac{p_{ij}}{s_{ij}}) \ . \tag{4.6}$$

A Figura 24 será usada para explicar como essa operação pode ser estendida para calcular o centro de gravidade de uma célula inteira.

Para cada aresta AB da célula, caminha-se de A a B, detectando-se todas as interseções com os lados dos quadrados (em ambos os eixos). Convencione-se que P_X seja projeção do ponto X na última linha da imagem, como ilustrado na Figura 24. A cada interseção T encontrada até chegar em B, seja U a última interseção ao longo da caminhada, ou o vértice inicial no caso de T ser a primeira interseção, calcula-se o centro de gravidade do trapézio UP_UP_TT e junta-se, como descrito na Equação 4.3, com o centro de gravidade de todos os trapézios anteriores. No final, obtém-se o centro de gravidade do trapézio AP_AP_BB. Falta detalhar o cálculo do centro de gravidade de cada trapézio, feito a seguir.

4.1.2.3 Cálculo do Centro de Gravidade de um Trapézio Contido em Uma Mesma Coluna

Considere o trapézio AP_AP_TT, onde AT é o segmento sobreposto na aresta da célula original. Por construção, AT está completamente contido em um quadrado, então AP_AP_TT pode ser segmentado em um trapézio menor $ARST$ e um retângulo RP_AP_TS. Já foi discutido como se calcula o centro de gravidade de um retângulo contido em uma coluna. Resta explicar que, se o retângulo tem comprimento r, $r < 1$, ou seja, os pontos A e T não estão ambos nas bordas da esquerda e direita, a massa original retornada em tempo constante daquele retângulo inteiro deve ser multiplicada por r. Note que a coordenada no eixo vertical do centro de gravidade do retângulo não muda, e, a no eixo horizontal, é sempre $\frac{A.x+T.x}{2}$, onde $.x$ representa a coordenada no eixo horizontal.

O centro de gravidade do trapézio $ARST$ é simplesmente o seu centroide, pois a distribuição da massa dentro de cada pixel é homogênea.

Por fim, junta-se os centros de gravidade de $ARST$ e RP_AP_TS.

4.1.2.3.1 União dos Centros de Gravidade dos Trapézios Abaixo de Cada Aresta

Uma vez obtidos os centros de gravidade de cada trapézio abaixo de cada aresta da célula, resta juntá-los respeitando os sinais, como detalhado a seguir.

Itera-se pelos vértices da célula ou em sentido horário, ou anti-horário. Aqui, usou-se anti-horário. Para cada par de vértices adjacentes (aresta) P_1P_2, se a coordenada horizontal de P_2 for menor que a de P_1 (mais à esquerda), $P_2.x < P_1.x$, significa que é uma aresta da parte de cima da célula, caso contrário, da de baixo. Na figura de exemplo, as arestas de cima são CB, BA, e as de baixo, AD, DC. Ao juntar os centros de gravidades dos trapézios pela Equação 4.3, se o trapézio i é de uma aresta da parte de baixo, nega-se os sinais de $c_i m_i$ e m_i.

4.1.2.3.2 Análise de Complexidade

O custo de computar o trapézio abaixo de cada aresta é proporcional ao número de quadrados intersectados por ela, pois cada interseção gera uma segmentação de um trapézio contido em uma coluna. Esse número é $\mathcal{O}(n+m)$, pois no máximo um lado é intersectado por linha/coluna. Por exemplo, na figura, a aresta AB intersecta 4 quadrados, então 5 trapézios precisaram ser calculados individualmente.

Embora o perímetro e o número de lados de cada célula possam ser diferentes, ou seja, o custo de computar o centro de gravidade de cada uma delas pode variar, a computação dos trapézios de todas as arestas do diagrama é $\mathcal{O}((n+m)r)$. Isso se deve ao fato de que o número de arestas em um diagrama de Voronoi é linear em relação ao número de pontos usados para gerá-lo [28].

Dessa forma, após o pré-processamento das somas de prefixo em $\mathcal{O}(nm)$, a complexidade de tempo em cada iteração igual a $\mathcal{O}((n+m)r)$ já viabilizou a execução em tempo real no simulador desenvolvido.

4.2 Heurística da Maximização da Cobertura da Figura

A cobertura usando diagramas de Voronoi apresentada na seção anterior pode ser vista como uma heurística gulosa que depende do posicionamento inicial dos robôs e visa a maximizar a cobertura da figura. Pode ser desejado obter sempre o mesmo resultado, independentemente das condições iniciais, então essa seção formaliza o conceito de cobertura e teoriza sobre como atingir uma cobertura ótima.

Formalmente, temos n círculos de raios $r_1, r_2, ..., r_n$ e uma função não contínua $z = f(c) : \mathbb{R}^2 \to \mathbb{R}, 0 \leq z \leq 1$, que representa os valores de cobertura de cada coordenada da imagem a ser representada, ditados pelos pixels. O objetivo é escolher n coordenadas

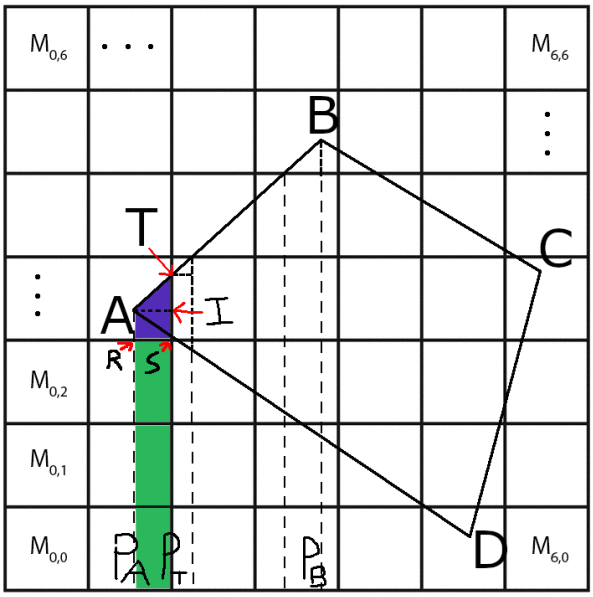


Figura 24 – Ilustração de referência para explicar o cálculo do centro de gravidade das células do diagrama de Voronoi. Para calcular o centro de gravidade do segmento AB, junta-se o centro de gravidade de cada trapézio segmentado de A a B, como o AP_AP_TT. O centro de gravidade do retângulo RP_AP_TS é calculado em $\mathcal{O}(1)$ através das somas de prefixo. O centro de gravidade da outra parte do trapézio AP_AP_TT, $ARST$, é igual ao seu centroide, por ser um polígono de densidade homogênea.

$c_1, c_2, ..., c_n$ no espaço, tal que a cobertura

$$\sum_{i}^{n} \int_{D_i} f(c)\mathrm{d}c$$

seja máxima, onde $D_i = \|c_i - c\| \leq r_i$ representa um disco de raio r_i centrado em c_i.

Objetivando simplificar este problema contínuo, é razoável usar uma técnica recorrente em robótica e Ciência da Computação como um todo: discretizar as opções. Em vez de considerarmos as infinitas coordenadas, amostramos um número finito de coordenadas que cada robô pode assumir, e tentamos resolver esse novo problema, mais restrito. Vale ressaltar que a solução do problema contínuo original ocorre quando o número de amostras tende a infinito, tal que a distância de qualquer coordenada a alguma amostra tende a zero. Assim, a precisão da solução exata depende do número de amostras.

4.2.1 Amostragem das Coordenadas Usadas na Modelagem Discreta

A tarefa da amostragem consiste em selecionar P coordenadas da imagem a serem usadas na versão discretizada do problema. A seguir são apresentadas duas heurísticas para isso.

4.2.1.1 Amostragem Pseudoaleatória

Uma forma convencional de amostrar é pseudoaleatoriamente, dando uma probabilidade igual para cada coordenada. Poder-se-ia usar apenas os centros dos pixels, por exemplo, mas decidiu-se aleatorizar as coordenadas dentro de cada pixel também, para evitar o viés de muitos pontos colineares. Optou-se por amostrar coordenadas apenas em pixels não vazios.

4.2.1.2 Amostragem Poligonal

Outra forma de amostragem consistiu em encontrar uma triangulação aproximada da figura, seguida de um passo que gulosamente amostra os centroides dos triângulos de maior área. Essa heurística é motivada pela expectativa de que as amostras serão melhores, de forma espaçada e longe de bordas. O método pode ser explicado em três passos principais, detalhados nas seções seguintes:

1. Poligonização inicial da figura, gerando linhas que delimitam as regiões que devem ser representadas e uma triangulação interna associada;
2. Simplificação da malha triangular obtida no passo anterior, mantendo a topologia original;
3. Amostragem baseada nos centroides dos triângulos das coordenadas candidatas.

4.2.1.2.1 Poligonização Inicial

O primeiro passo consiste em delimitar as regiões com polilinhas: sequências conectadas de segmentos de reta. Pode-se pensar em várias formas de fazer isso, mas optou-se por usar um algoritmo existente de geometria computacional chamado *Alpha Shape* [29], e uma visão geral sobre ele é dada a seguir.

4.2.1.2.2 Alpha Shape

O Alpha Shape pode ser visto como uma extensão do Convex Hull. Este último consiste em delimitar a menor região convexa que engloba um conjunto de pontos, e pode ser pensado como o resultado de rodar um círculo c de raio r infinito em volta dos pontos, ligando cada par de pontos tocados por c. O Alpha Shape é uma generalização dessa ideia, que parametriza r, como ilustrado na Figura 25. Assim como o Convex Hull, pode ser implementado em $\mathcal{O}(n \log(n))$ e opera em um conjunto discreto de pontos. A Figura 26 exemplifica as polilinhas resultantes para diferentes r em outra imagem.

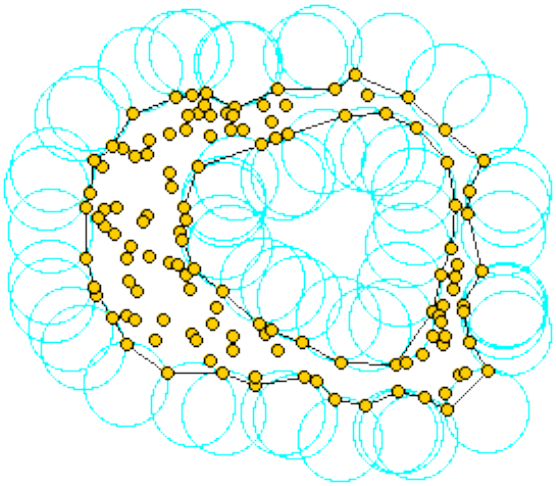
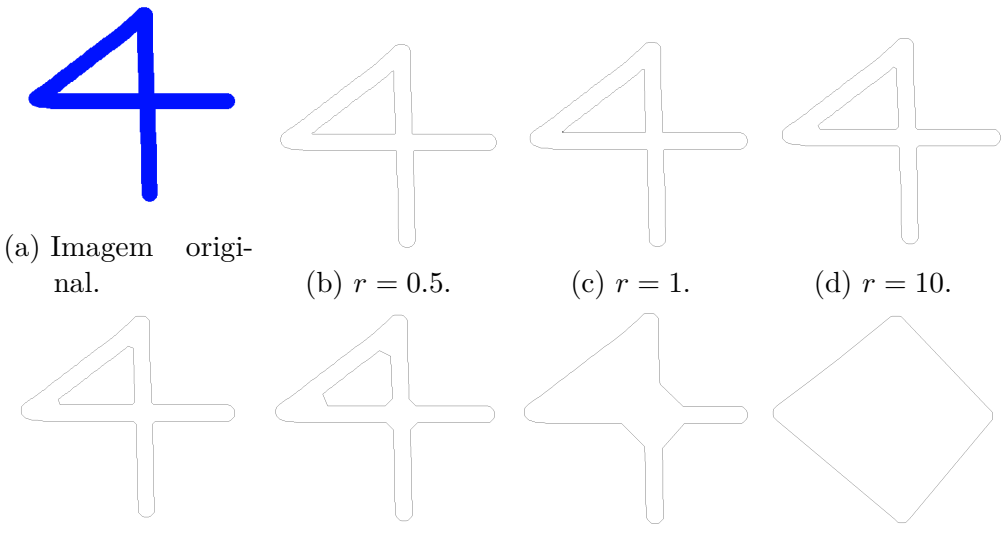


Figura 25 – Exemplo de um Alpha Shape [30].



(a) Imagem original. (b) $r = 0.5$. (c) $r = 1$. (d) $r = 10$.

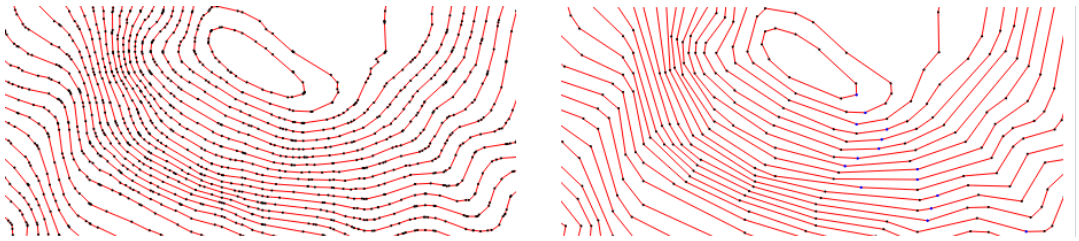
(e) $r = 20$. (f) $r = 100$. (g) $r = 1000$. (h) $r = 1000000$.

Figura 26 – Exemplo da poligonização usando o Alpha Shape para raios r diferentes. Como a distância entre cada ponto é pelo menos 1, o raio de 0.5 é o menor possível para o Alpha Shape encontrar pontos que devem ser ligados. Foram usadas as coordenadas das quinas dos pixels azuis de entrada para o algoritmo. (a) Figura a ser poligonizada. (b)-(h) Exemplos da poligonização com diferentes raios r.

4.2.1.2.3 Simplificação da Malha

As polilinhas obtidas no passo anterior poderiam ser usadas direto para construir uma triangulação de Delaunay restrita [31], ou seja, os triângulos resultantes ficam contidos dentro dos polígonos. Mas como o número de vértices nas polilinhas pode ser muito grande, é interessante aplicar um passo intermediário que diminui essa quantidade mantendo a topologia dos polígonos. A topologia refere-se aos buracos dos polígonos, ou seja, não são criados ou deletados nenhum deles.

Para isso, é usado um algoritmo de simplificação de polilinhas em duas dimensões [32] disponível no CGAL [33]. A Figura 27 mostra um exemplo de aplicação desse algoritmo.



(a) Antes. (b) Depois.

Figura 27 – Exemplo de simplificação de polilinhas [34].

Esse algoritmo usa um parâmetro r que define a razão de simplificação, que representa a porcentagem de triângulos aproximada desejada na triangulação das polilinhas simplificadas em relação ao número de triângulos na triangulação das polilinhas originais.

A Figura 28 mostra um exemplo completo da obtenção da malha triangular simplificada que é usada na próxima seção.

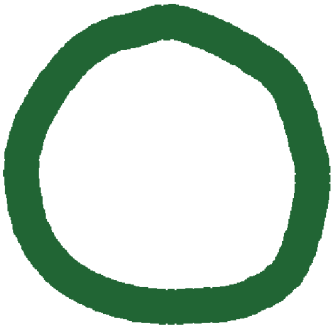
4.2.1.2.4 Amostragem Baseada na Malha Triangular

As heurísticas desenvolvidas são baseadas em amostrar, gulosamente, o centroide do triângulo de maior área, usando a estrutura de dados de uma fila de prioridade q. A cada centroide C amostrado, o seu triângulo original T é removido de q e subdividido em novos triângulos, que são inseridos em q.

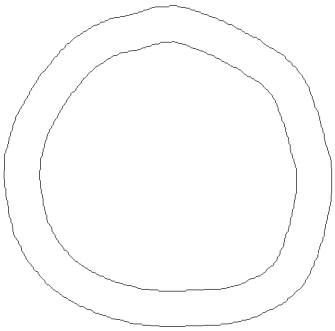
Há várias formas de subdividir um triângulo. Foram implementadas duas. Ambas começam computando as 3 medianas do triângulo, mas continuam diferentemente. Denotam-se T_1, T_2, T_3 os vértices do triângulo, e M_1, M_2, M_3 os pontos médios (medianas) de T_1T_2, T_2T_3, T_3T_1 respectivamente. A seguir, são descritas essas formas, usando a Figura 29 de exemplo.

4.2.1.2.5 Subdivisão em 6 Triângulos

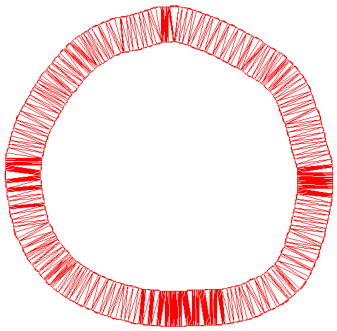
Nesse método, usa-se diretamente os 6 triângulos que são formados após traçar as medianas. São eles: CT_1M_1, CM_1T_2, CT_2M_2, CM_2T_3, CT_3M_3, CM_3T_1. Um exemplo está



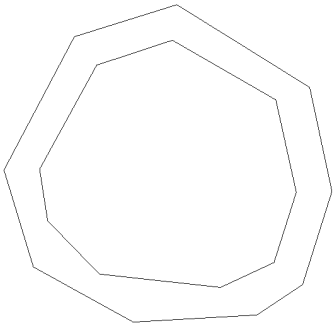
(a) Figura original.



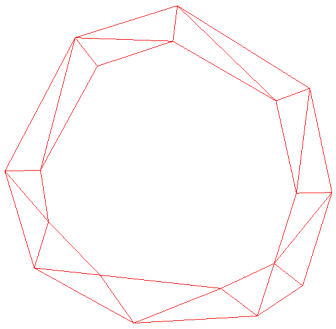
(b) Polilinhas iniciais retornas pelo Alpha Shape.



(c) Triangulação de Delaunay restrita usando as polilinhas originais.



(d) Polilinhas simplificadas com $r = 0.01$.



(e) Triangulação de Delaunay restrita usando as polilinhas simplificadas.

Figura 28 – Exemplo da simplificação da malha.

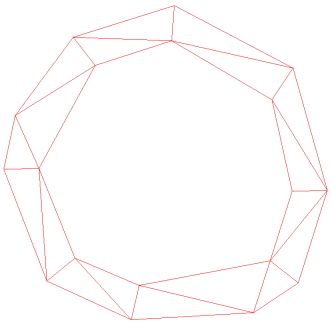
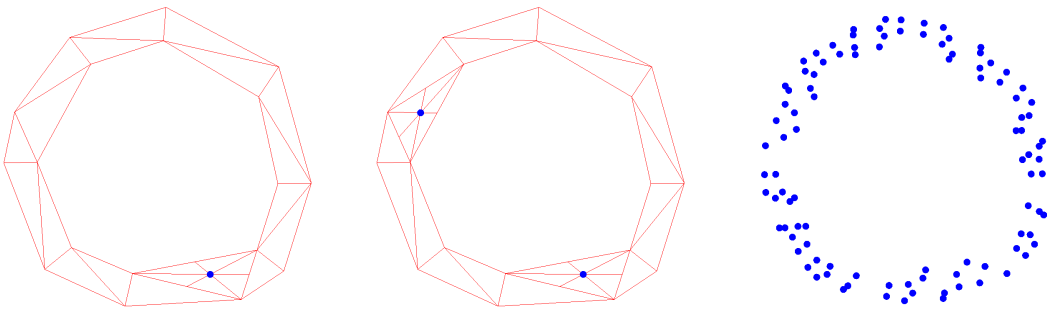


Figura 29 – Malha triangular simplificada original usada nas explicações.

na Figura 30.

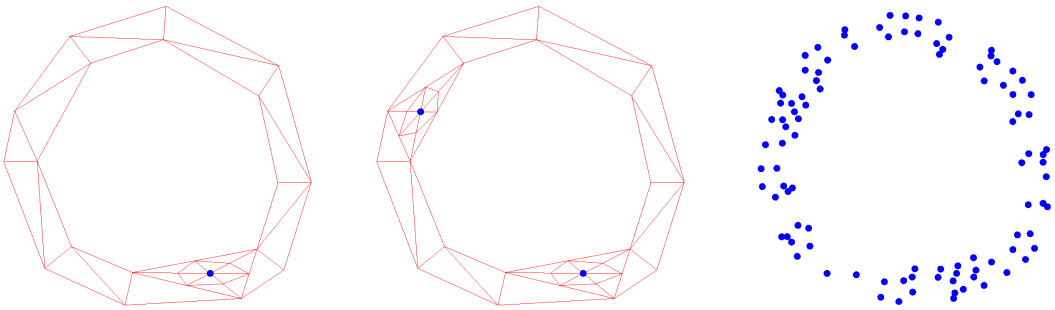


(a) 1 amostra. (b) 2 amostras. (c) 100 amostras.

Figura 30 – Exemplo da subdivisão em 6 triângulos.

4.2.1.2.6 Subdivisão em 12 Triângulos

Nesse método, consideram-se os 3 quadriláteros $CM_3T_1M_1$, $CM_1T_2M_2$, $CM_2T_3M_1$, e cada um deles é subdividido em 4 triângulos, onde cada triângulo é formado pelo centroide do quadrilátero e 2 vértices adjacentes a ele. Um exemplo está na Figura 31.



(a) 1 amostra. (b) 2 amostras. (c) 100 amostras.

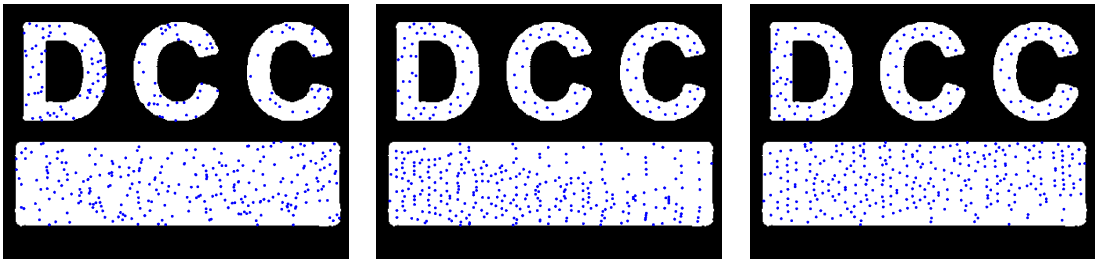
Figura 31 – Exemplo da subdivisão em 12 triângulos.

4.2.2 Comparação das Técnicas de Amostragem

A Figura 32 mostra o resultado de cada método de amostragem discutido, aplicado em uma mesma imagem.

4.2.3 Modelagem Discreta da Otimização da Cobertura da Figura pelos Robôs

Uma vez amostradas P coordenadas candidatas para posicionar os R robôs de raios $r_1, r_2, ..., r_R$, considere o grafo $G(V, E)$, onde para cada ponto P_i, são criados R vértices, ou seja, $|V| = PR$. Seja v_{pr} o vértice que representa que o ponto p foi alocado para o robô r. Cada aresta $u_{p_1r_1}v_{p_2r_2}$ existe se e somente se não for possível escolher os dois vértices u e v ao mesmo tempo, ou seja, a distância euclidiana entre as coordenadas p_1 e p_2 é



(a) Amostragem 4.2.1.1. (b) Amostragem 4.2.1.2.5. (c) Amostragem 4.2.1.2.6.

Figura 32 – Teste comparando os métodos de amostragem com 400 amostras.

menor que a soma dos raios dos robôs r_1 e r_2 (condição que garante que os círculos sejam disjuntos) ou $r_1 = r_2$ (condição que garante que cada robô só é alocado uma vez). Note que as arestas $u_{pr_i}v_{pr_j} \forall p, r_i, r_j, i \neq j$ sempre existem (assumindo $r > 0$), representando que cada coordenada pode ser escolhida no máximo uma vez.

Cada vértice v_{pr} possui um peso w representando o valor de posicionar o robô r na coordenada p. Esse valor é a cobertura obtida por r em p, correspondente à área da interseção de r com a imagem. O robô é circular, mas na implementação, foi aproximado como um polígono regular de número de lados configurável, para ser possível reaproveitar a sub-rotina já desenvolvida na Seção 4.1, que calcula a área de um polígono simples sobreposto na imagem.

Além disso, se a razão da cobertura de r em p sobre a área de r (πr_i^2) for pequena, ou seja, r cobre muito espaço "errado", pode ser desejável deletar o vértice v_{pr} do grafo, ou seja, remover essa possibilidade. Dessa forma, pode-se adicionar um parâmetro na modelagem, que indica a porcentagem mínima que cada robô deve cobrir para ser candidato para uma coordenada.

Formalmente, seja $x(v)$ uma variável binária ($x(v) \in \{0, 1\}$) que indica se o vértice v foi escolhido, objetiva-se otimizar a cobertura total

$$\max \sum_v w(v)x(v) \ ,$$

respeitando-se as restrições de que cada aresta possui no máximo um dos vértices escolhidos:

$$x(u) + x(v) \leq 1 \ \forall (u, v) \in E \ .$$

Esse problema de otimização, conhecido como conjunto independente máximo com pesos nos vértices (maximum weighted independent set), por possuir restrições de variáveis serem inteiras, é NP-difícil em grafos gerais.

Há classes de grafos em que este problema é polinomial, como grafos sem caminhos de 5 vértices [35], mas, mesmo inseridos em um contexto geométrico, não foi possível encontrar um algoritmo exato eficiente para os grafos que podem ser gerados aqui. Vale ressaltar que se objetiva a execução em tempo real, então mesmo se houvesse um algoritmo polinomial, porém muito caro, seu uso poderia ser inviável.

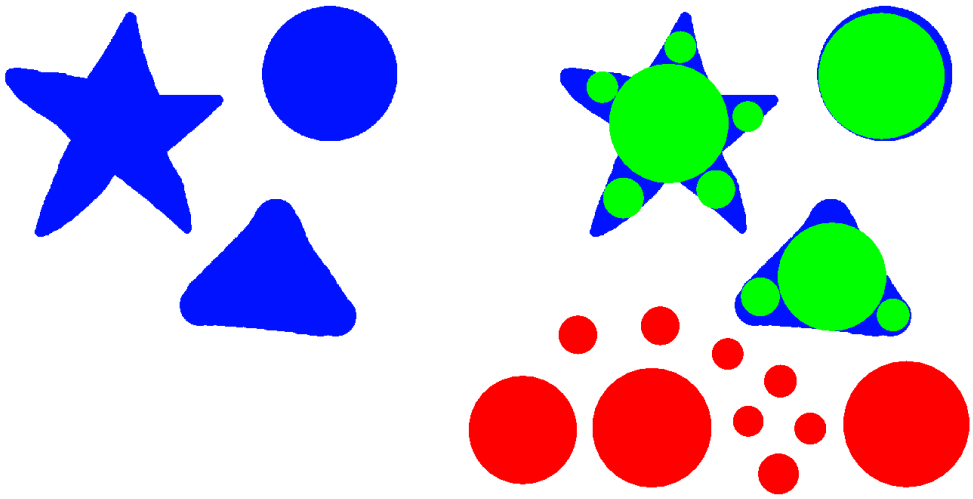
Então, foi usada uma heurística gulosa: tenta-se alocar os círculos do maior para o menor. Para cada um deles, é calculada a cobertura atingida em cada amostra disponível, e escolhe-se a melhor. Uma amostra c está disponível para o círculo r_i se ao posicionar r_i em c, r_i não colide com nenhum círculo já alocado, ou seja,

$$r_i + a_r > \|c_i - a_c\| \ \forall u \in U \ ,$$

onde A, a_r, a_c representam, respectivamente, o conjunto de círculos já alocados até aquela iteração, o raio e a coordenada de a. A complexidade de tempo dessa heurística gulosa é $\mathcal{O}(PR)$, pois, para cada robô, testa-se cada ponto.

Quanto à cor que cada agente deve assumir em cada coordenada, para aplicar o mesmo método da Seção 3.1.1.1, seria necessário computar a imagem filtrada para cada raio diferente. Para simplificar a implementação, usou-se a cor do pixel na coordenada central de cada círculo.

A Figura 33 mostra um exemplo da alocação das posições baseada na amostragem aleatória de 1000 pontos, seguida da heurística gulosa. A Figura 34 mostra exemplos com um número maior de robôs, já assumindo as cores esperadas.



(a) Figura original. (b) Alocação.

Figura 33 – Exemplo da heurística gulosa da alocação dos robôs heterogêneos. Os robôs disponíveis são os círculos vermelhos, e os círculos verdes representam a posição final alocada para eles.

4.2.4 Conclusão, Considerações Finais e Trabalhos Futuros

Nos experimentos realizados, notou-se que a heurística da alocação gulosa, em geral, dá resultados satisfatórios. Entretanto, diferentemente da heurística da grade hexagonal, não há garantia de que os robôs disponíveis fiquem bem distribuídos no espaço, o que pode resultar em áreas muito densas ou vazias se há um número insuficiente deles.

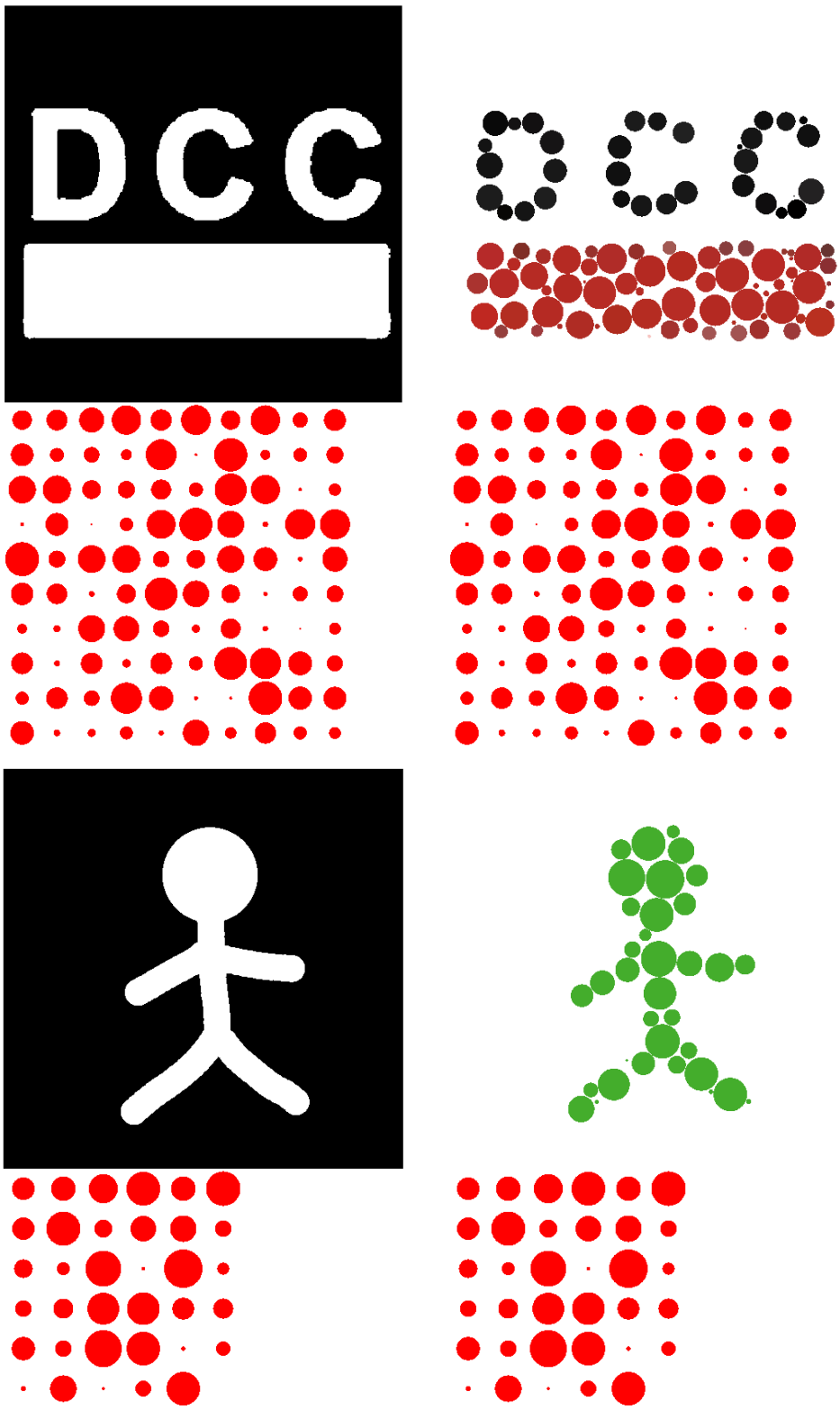


Figura 34 – Exemplo da alocação de robôs heterogêneos. Abaixo, de vermelho, os robôs disponíveis. À esquerda, de branco, a área a ser coberta, e à direita, o resultado, já assumindo as cores corretas.

A solução exata do problema de otimização em grafo chegou a ser implementada e testada em um otimizador de código aberto (GLPK), mas, como esperado, mesmo em instâncias ainda pequenas (10 robôs e 10 samples, por exemplo), a computação já era inviavelmente lenta (não terminava nem em 10 minutos, e sem previsão disponível).

Essa modelagem tem vantagens e desvantagens em relação ao controle de cobertura usando diagramas de Voronoi. Aqui, todos os robôs são usados na alocação, enquanto no Voronoi, muitos podem ser inutilizados por estarem em células sem nenhum pixel a ser representado. Além disso, essa modelagem teoriza um ótimo global, enquanto no Voronoi, os robôs convergem para um ótimo local. Por outro lado, a heurística do Voronoi é muito mais barata computacionalmente e mais robusta a inserções e deleções. Aqui, não há localidade no algoritmo: adicionar ou remover um robô pode mudar a alocação inteira anterior, enquanto no Voronoi, os robôs se ajustam rapidamente. Em robôs reais, isso seria equivalente ao Voronoi ser mais resiliente a falhas individuais.

Quanto às técnicas de amostragem, empiricamente, notou-se que quando o número de amostras é grande suficiente para a figura a ser representada, o método de amostragem usado não faz muita diferença, então é melhor usar o aleatório, por ser mais barato e menos enviesado. Entretanto, se por algum motivo o número de amostras precise ser baixo, a amostragem baseada nos centroides dos triângulos gera pontos mais garantidamente bem distribuídos, o que pode ser desejado.

Além disso, em consonância com a teoria apresentada na Seção 3.2.1.3, o navegador ORCA não funcionou bem em situações como a da Figura 33b, onde pode-se notar que seria necessária uma sincronização mais fina para o robô do centro da estrela conseguir convergir, dado que não consegue chegar em sua posição desejada unilateralmente se os robôs nos braços da estrela já estiverem posicionados, por exemplo.

Como trabalhos futuros, uma área interessante e importante a ser explorada é um planejador deliberativo de trajetórias, em vez de reativo, capaz de lidar com o congestionamento gerado nas alocações dos robôs heterogêneos.

5 Simulador

Como grande parte deste POC foi de teor prático, implementando e testando os métodos descritos, vale ressaltar o software desenvolvido, de onde foram tiradas várias das imagens deste artigo.

O código foi escrito em C++. Embora seja quase todo *cross-platform*, suportando Windows, Linux e macOS (incluindo as bibliotecas externas), só há binários pré-compilados para Windows por praticidade.

Aqui são detalhadas as principais funcionalidades.

5.1 Importação das Imagens

Como descrito em seções anteriores, o simulador usa uma imagem de 4 componentes, onde as 3 primeiras são as cores RGB, e a última especifica o quanto cada pixel faz parte da figura a ser representada. Por isso, formatos de imagem que não possuem a componente A, como JPEG, não foram suportados. Optou-se por suportar apenas PNG (Portable Network Graphics).

Há duas funcionalidades de importação. Em uma, o usuário especifica o caminho da imagem diretamente. Em outra, ele deve selecionar uma pasta com as imagens que, após carregadas, podem ser alternadas com as setas da esquerda e da direita do teclado. Nesse caso, para evitar a necessidade de programar uma funcionalidade de reordenação das imagens dentro do simulador, elas devem seguir uma convenção de nomenclatura: "1.png", "2.png", ..., "n.png". Dessa forma, são carregadas em ordem numérica, começando do "1.png"e para-se quando não existir uma imagem "x.png". Por exemplo, se há as imagens 1, 2 e 4, apenas as 1 e 2 são carregadas.

As bibliotecas nativefiledialog [36] e nativefiledialog-extended [37] foram usadas para interagir com o sistema de arquivos do computador, como mostrado na Figura 35. Vale ressaltar que atentou-se para usar as versões Unicode das funções, então é suportado caminhos de arquivos com acentos do português, por exemplo.

5.2 Interface Gráfica

A biblioteca SFML [38] é usada para abstrair a janela do sistema operacional e renderizar as coisas 2D, usando OpenGL internamente.

Os menus que possibilitam testar diferentes configurações de parâmetros foram feitos com a biblioteca ImGui [39] e sua bind [40] para SFML. Além disso, há uma opção de

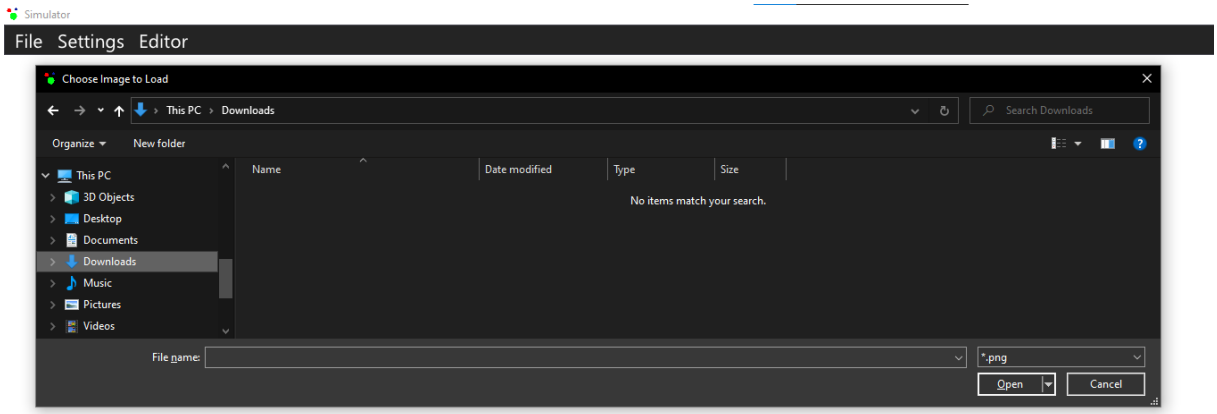


Figura 35 – Exemplo da importação de uma imagem no simulador.

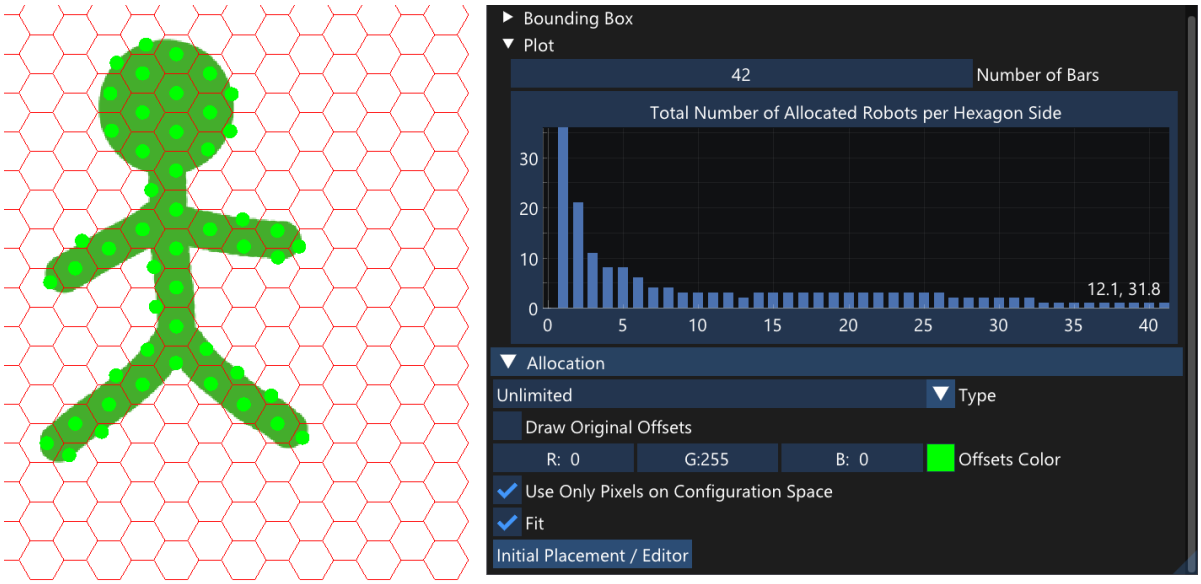


Figura 36 – Exemplo de um gráfico gerado com o implot [41] dentro de um menu gerado com o ImGui [39].

gerar um gráfico na heurística da grade hexagonal sobre o número de robôs alocados em função do lado do hexágono, que é feito com a biblioteca implot [41], como na Figura 36.

5.3 Editor

O simulador contém um editor que lida com os eventos de adicionar, mover e deletar agentes e alvos. Há a opção para recalcular a alocação baseada nas métricas da Seção 3.1.2.1 automaticamente quando algum desses eventos ocorre.

5.4 Navegador ORCA

Conforme mencionado na Seção 3.2.1.3, esse navegador possui vários parâmetros. Foi feita uma interface para permitir testá-los, mostrada na Figura 37. Se a opção "Use Fixed

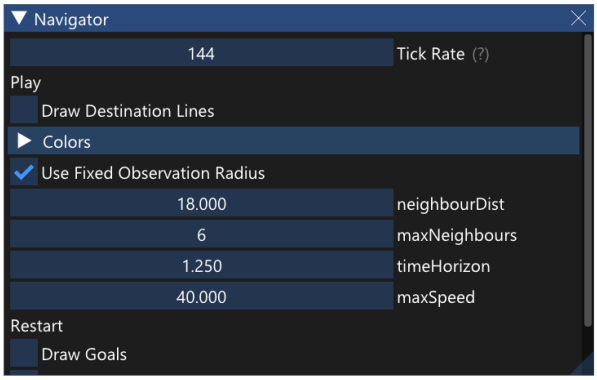


Figura 37 – Parâmetros da API do ORCA expostos no simulador.

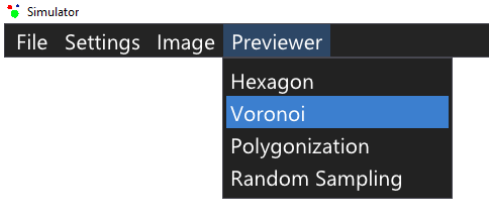


Figura 38 – Menu das heurísticas.

Observation Radius"é ligada, aparece o parâmetro "neighbourDist", que corresponde a um mesmo raio de observação usado por cada agente. Se desligada, o raio de observação de cada agente é personalizado para ser a soma de seu raio com o do maior agente no ambiente, que é previamente conhecido.

5.5 Heurísticas

Após o carregamento de uma imagem, ou pasta com imagens, fica disponível um menu com as opções das heurísticas implementadas, como na Figura 38.

Alguns detalhes de implementação de cada heurística são detalhados a seguir.

5.5.1 Grade Hexagonal

A convolução do filtro da Seção 3.1.1.1 é feita com a biblioteca OpenCV [42].

A implementação do algoritmo de emparelhamento máximo em grafo bipartido, usado na métricas 2 e 3, foi baseada em [43].

5.5.2 Voronoi

Foi usada a implementação [44] de diagrama de Voronoi. A Figura 39 mostra todas as opções disponíveis no simulador para testar a técnica da Seção 4.1, incluindo informações de depuração como realçar a célula no cursor do mouse e renderizar o centro de gravidade e o centroide das células.

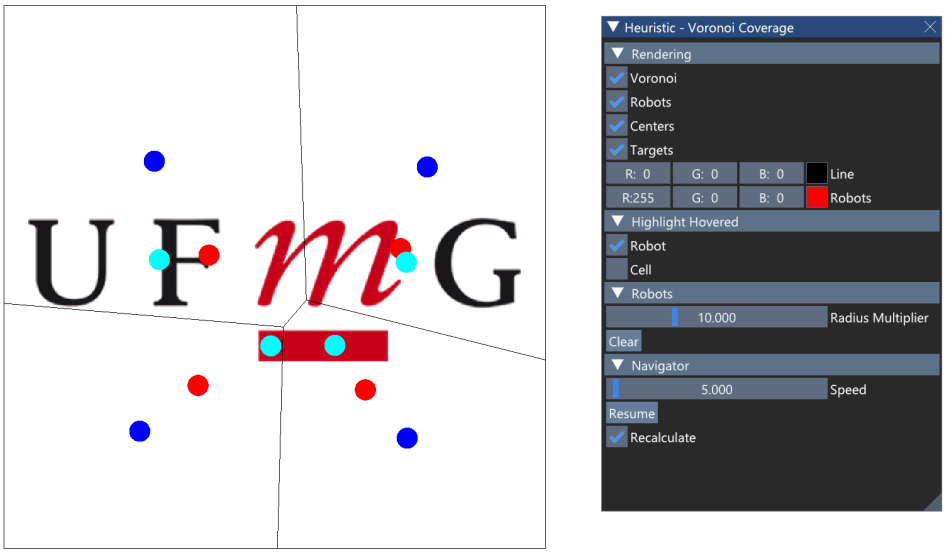


Figura 39 – Menu da heurística do Voronoi.

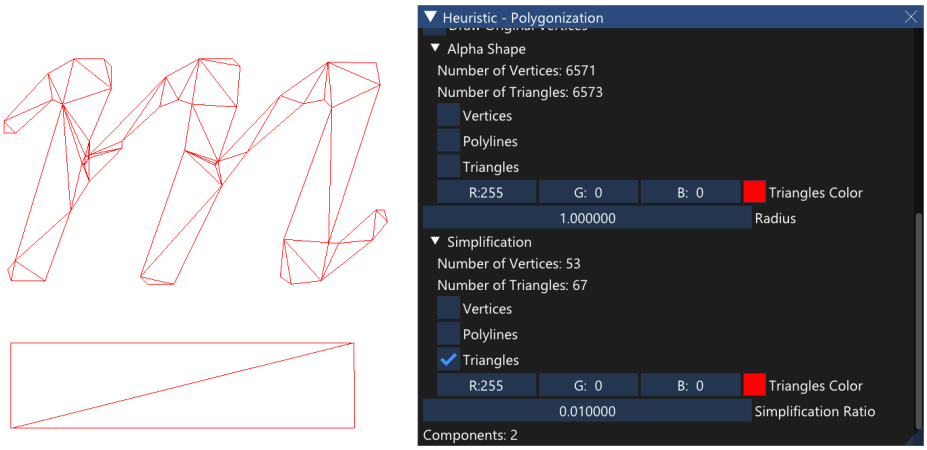


Figura 40 – Menu da heurística da poligonização.

5.5.3 Poligonização

Os algoritmos de geometria computacional usados - alpha shape, simplificação de polilinhas, triangulação de Delaunay restrita - são implementados no CGAL (Computational Geometry Algorithms Library) [33]. A Figura 40 mostra algumas opções disponíveis no menu dessa heurística.

5.5.4 Amostragem Pseudoleatória

O menu dessa heurística, mostrado na Figura 41, possui muitas coisas em comum com o da poligonização, como o parâmetro da razão mínima de cobertura dos robôs e a opção de criar uma configuração pseudoaleatória de robôs. O único parâmetro distinto é a semente do gerador de bits pseudoaleatórios.

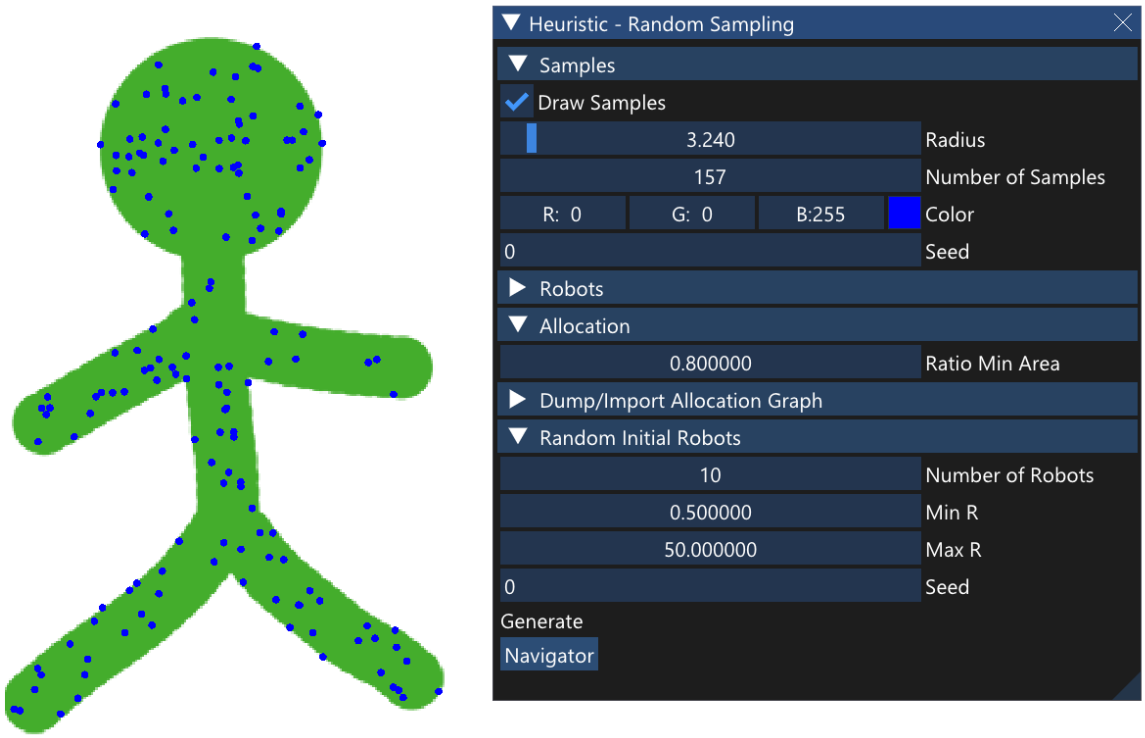


Figura 41 – Menu da heurística da amostragem pseudoaleatória.

6 Conclusão

Este POC avaliou maneiras de representar figuras em apresentações com robôs luminosos, tratando, separadamente, os passos da determinação das posições finais dos agentes e da navegação deles.

Em retrospectiva, estudou-se como alguns algoritmos clássicos, como da área de Teoria dos Grafos, Robótica e Geometria Computacional, podem ser usados nesse contexto. Além disso, foi praticada a capacidade de modularizar problemas e de formalizar funções objetivas.

Primeiro, considerou-se robôs de mesmo raio (homogêneos). Para selecionar as coordenadas dos alvos, foi desenvolvida uma heurística baseada em uma grade hexagonal, em que o tamanho do hexágono era escolhido de forma compatível com o número de robôs disponível. A alocação dos robôs aos alvos era tratada por meio do algoritmo húngaro, e diversas variações foram consideradas, como minimizar a maior distância ou a soma total delas.

Depois, a tarefa da navegação sem colisões foi discutida, apresentando um algoritmo clássico da literatura, seus usos e limitações.

Em um segundo momento, considerou-se robôs de diferentes raios (heterogêneos). Inicialmente, implementou-se um método clássico em Robótica de cobertura de área que usa diagramas de Voronoi.

Depois, foi feita uma nova modelagem, baseada na amostragem de coordenadas candidatas, que possui uma função objetivo, cuja otimalidade foi teorizada, mas foi usada uma heurística gulosa para possibilitar a execução em tempo real. Diferentes formas de amostrar as coordenadas também foram discutidas, como poligonizar a figura, triangular as polilinhas e selecionar os centroides dos triângulos.

Embora o escopo do trabalho tenha se mantido em duas dimensões, várias ideias poderiam ser estendidas para o espaço tridimensional.

Por fim, outra contribuição relevante é o simulador implementado [45], pois as visualizações gráficas são motivantes e permitem aos interessados na área aguçarem a intuição sobre os algoritmos discorridos.

Uma apresentação em vídeo do POC I está disponível em [46], e do POC II em [47].

Referências

[1] INTEL Drone Light Show. 2018. Disponível em: <https://newsroom.intel.com/news-releases/intel-drone-light-show-breaks-guinness-world-records-title-olympic-winter-games-pyeongchang-2018>.

[2] YU, J.; LAVALLE, S. M. Structure and Intractability of Optimal Multi-Robot Path Planning on Graphs. In: *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*. [S.l.]: AAAI Press, 2013. (AAAI'13), p. 1443–1449.

[3] JOHNSON, W. W.; STORY, W. E. et al. Notes on the "15" puzzle. *American Journal of Mathematics*, JSTOR, v. 2, n. 4, p. 397–404, 1879.

[4] RATNER, D.; WARRNUTH, M. Computer & information sciences university of california santa cruz santa cruz, ca 95064.

[5] KHATIB, O. Real-time obstacle avoidance for manipulators and mobile robots. In: *Autonomous Robot Vehicles*. [S.l.]: Springer, 1986. p. 396–404.

[6] GASPARETTO, A. et al. Trajectory planning in robotics. *Mathematics in Computer Science*, Springer, v. 6, n. 3, p. 269–279, 2012.

[7] KARATAS, T.; BULLO, F. Randomized searches and nonlinear programming in trajectory planning. In: IEEE. *Proceedings of the 40th IEEE Conference on Decision and Control (Cat. No. 01CH37228)*. [S.l.], 2001. v. 5, p. 5032–5037.

[8] HEGEDÜS, F. et al. Motion planning for highly automated road vehicles with a hybrid approach using nonlinear optimization and artificial neural networks. *Strojniski vestnik-Journal of Mechanical Engineering*, v. 65, n. 3, p. 148–160, 2019.

[9] PIXELBOTS. Disponível em: <https://youtu.be/4-3rkrqv014>.

[10] ALONSO-MORA, J. et al. Image and animation display with multiple mobile robots. *The International Journal of Robotics Research*, v. 31, n. 6, p. 753–773, 2012.

[11] DU, X. et al. Fast and In Sync: Periodic Swarm Patterns for Quadrotors. In: *2019 International Conference on Robotics and Automation (ICRA)*. [S.l.: s.n.], 2019. p. 9143–9149.

[12] BERG, J. v. d. et al. Reciprocal n-body collision avoidance. In: *Robotics research*. [S.l.]: Springer, 2011. p. 3–19.

[13] SNAPE, J.; MANOCHA, D. Navigating multiple simple-airplanes in 3d workspace. In: IEEE. *2010 IEEE International Conference on Robotics and Automation*. [S.l.], 2010. p. 3974–3980.

[14] RVO em Jogos. Disponível em: <`https://gamma.cs.unc.edu/RVO2/publications/AIGD12.pdf`>.

[15] SNAPE, J. et al. Smooth and collision-free navigation for multiple robots under differential-drive constraints. In: IEEE. *2010 IEEE/RSJ international conference on intelligent robots and systems*. [S.l.], 2010. p. 4584–4589.

[16] RVO 2D - Implementação. Disponível em: <`https://github.com/snape/RVO2`>.

[17] RVO 3D - Implementação. Disponível em: <`https://github.com/snape/RVO2-3D`>.

[18] CHANG, H.-C.; WANG, L.-C. A simple proof of thue's theorem on circle packing. *arXiv preprint arXiv:1009.4322*, 2010.

[19] EDMONDS, J.; KARP, R. M. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 19, n. 2, p. 248–264, apr 1972. ISSN 0004-5411.

[20] JONKER, R.; VOLGENANT, A. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, v. 38, n. 4, p. 325–340, Dec 1987. ISSN 1436-5057. Disponível em: <https://doi.org/10.1007/BF02278710>.

[21] KUHN, H. W. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, v. 2, n. 1-2, p. 83–97, 1955.

[22] HOPCROFT, J. E.; KARP, R. M. An n5/2 Algorithm for Maximum Matchings in Bipartite Graphs. *SIAM J. Comput.*, v. 2, p. 225–231, 1973.

[23] VORONOI - Exemplo. Disponível em: <`https://en.wikipedia.org/wiki/Voronoi_diagram#/media/File:Euclidean_Voronoi_diagram.svg`>.

[24] FORTUNE, S. A sweepline algorithm for voronoi diagrams. In: *Proceedings of the second annual symposium on Computational geometry*. [S.l.: s.n.], 1986. p. 313–322.

[25] DU, Q.; EMELIANENKO, M.; JU, L. Convergence of the lloyd algorithm for computing centroidal voronoi tessellations. *SIAM journal on numerical analysis*, SIAM, v. 44, n. 1, p. 102–119, 2006.

[26] LU, Y.; ZHOU, H. H. Statistical and computational guarantees of lloyd's algorithm and its variants. *arXiv preprint arXiv:1612.02099*, 2016.

[27] CENTROIDE. Disponível em: <`https://en.wikipedia.org/wiki/Centroid`>.

[28] VORONOI - Propriedades. Disponível em: <http://cs.rkmvu.ac.in/~sghosh/subhas-lecture.pdf>.

[29] EDELSBRUNNER, H.; KIRKPATRICK, D.; SEIDEL, R. On the shape of a set of points in the plane. *IEEE Transactions on information theory*, IEEE, v. 29, n. 4, p. 551–559, 1983.

[30] ALPHA Shape. Disponível em: <https://doc.cgal.org/latest/Alpha_shapes_2/index.html>.

[31] BOISSONNAT, J.-D. et al. Triangulations in cgal. In: *Proceedings of the sixteenth annual symposium on Computational geometry*. [S.l.: s.n.], 2000. p. 11–18.

[32] DYKEN, C.; DÆHLEN, M.; SEVALDRUD, T. Simultaneous curve simplification. *Journal of geographical systems*, Springer, v. 11, n. 3, p. 273–289, 2009.

[33] The CGAL Project. *CGAL User and Reference Manual*. 5.5. CGAL Editorial Board, 2022. Disponível em: <https://doc.cgal.org/5.5/Manual/packages.html>.

[34] POLILINHAS - Simplificação. Disponível em: <https://doc.cgal.org/latest/Polyline_simplification_2>.

[35] LOKSHANTOV, D.; VATSHELLE, M.; VILLANGER, Y. Independent set in p 5-free graphs in polynomial time. In: SIAM. *Proceedings of the twenty-fifth annual ACM-SIAM symposium on discrete algorithms*. [S.l.], 2014. p. 570–581.

[36] INTERFACE com o Sistema de Arquivos. Disponível em: <https://github.com/mlabbe/nativefiledialog>.

[37] INTERFACE com o Sistema de Arquivos. Disponível em: <https://github.com/btzy/nativefiledialog-extended>.

[38] SFML. Disponível em: <https://www.sfml-dev.org/>.

[39] IMGUI. Disponível em: <https://github.com/ocornut/imgui>.

[40] IMGUI SFML. Disponível em: <https://github.com/eliasdaler/imgui-sfml>.

[41] IMPLOT. Disponível em: <https://github.com/epezent/implot>.

[42] OPENCV. Disponível em: <https://github.com/opencv/opencv>.

[43] HOPCROFT Algorithm - Implementação. Disponível em: <https://judge.yosupo.jp/submission/52112>.

[44] VORONOI - Implementação. Disponível em: <https://github.com/JCash/voronoi>.

[45] SIMULADOR - Implementação. Disponível em: <https://github.com/davi-v/POC>.

[46] POC I - Apresentação. Disponível em: <https://youtu.be/CBcxTE3qJig>.

[47] POC II - Apresentação. Disponível em: <https://youtu.be/qiMKNgmetcg>.