Instituto de Matemática e Estatística
Universidade de São Paulo

Problemas de deslocamento no plano em geometria computacional

Aluno: Natan Costa Lima
Orientador: Professor Carlos Eduardo Ferreira

Sumário

Parte I
Técnica

1 Introdução

Neste projeto estudamos algoritmos e estruturas de dados para problemas de deslocamento no plano.

Em diversas aplicaçõs práticas surgem problemas deste tipo. Por exemplo, considere um robô que deve se deslocar entre vários objetos de uma sala para realizar suas tarefas sem colidir com obstáculos. Ou considere um museu onde o guarda noturno precisa vigiar todo o museu, neste caso seria importante ele andar em um caminho onde a distância percorrida fosse pequena e mesmo assim ele conseguisse avistar todo o museu.

Há diversas variantes interessantes destes problemas, no caso do robô, quando colocamos restrições que este deve satisfazer no caminho (ele pode rodar em qualquer ângulo?) e no que desejamos otimizar (encontrar o caminho mais curto? caminho mais rápido?) obtemos novas variantes.

Nosso objetivo neste trabalho de conclusão de curso foi estudar diversos problemas como esses e implementar algoritmos para resolvê-los.

2 Conceitos e tecnologias estudadas

2.1 Deslocamento em regiões com obstáculos

Um problema que foi estudado é o problema de deslocamento no plano bidimensional, onde queremos determinar um caminho no plano em que um certo robô deve se locomover sem colidir com obstáculos.

Chamemos nosso ambiente, onde estão os obstáculos (em forma de polígono), de R. Considere dois pontos do ambiente s e t, para os quais queremos traçar um caminho com início em s e término em t. Queremos responder às seguintes questões:

1. Existe um caminho de s a t, livre de obstáculos em R?
2. Encontre um caminho de s a t em R, livre de obstáculos.
3. Encontre um caminho de menor distância de s a t, livre de obstáculos.

É fácil perceber que uma resposta para questão (3) satisfaz à (2), assim como uma resposta para (2) resolve a (1). Logo, a questão de decisão é a mais fácil das três.

Quanto à questão (3), sua definição não é tão clara. Se considerarmos um robô como sendo um ponto e ainda que ele possa se mover para todos os lados livremente, então bastaria pegarmos o caminho com a menor distância euclidiana. Suponha agora que seja um carro que precisaria acelerar e desacelerar para fazer curvas, ou ainda pensemos em um robô que demore muito para virar. Nestes casos, a menor distância euclidiana pode não ser a melhor escolha.

Aqui iremos aplicar algumas restrições. Talvez a mais drástica à primeira vista, seja considerarmos o ambiente como um plano em duas dimensões. Mas, se pensarmos em um mapa ou uma planta do local, isto não se parece mais tão restritivo. Consideraremos também que o ambiente seja estático, ou seja, não levamos em conta pessoas ou outros objetos se movendo pelo cenário.

Primeiro, consideremos o nosso robô como sendo um ponto e ainda que ele possa andar em todas as direções livremente. Uma forma de acharmos um caminho livre de obstáculos foi proposta por Kedem e outros [6][7]. A ideia consiste em dividir a área livre em trapézios, o que chamaremos de mapa trapezoidal, para depois calcularmos um caminho livre de obstáculos.

2.1.1 Mapa trapezoidal

Para alcançarmos nosso objetivo temos que olhar para um problema mais simples. Vamos discutir aqui sobre localização de pontos no plano. Podemos enunciar da seguinte maneira: Dado um mapa e um ponto q (definido pelas suas coordenadas) que queremos localizar, ache a região do mapa que contenha q.

No nosso caso o mapa P será um conjunto de polígonos no plano, e S o conjunto de retas que formam os polígonos de P. Consideraremos n como sendo o número de vértices em P.

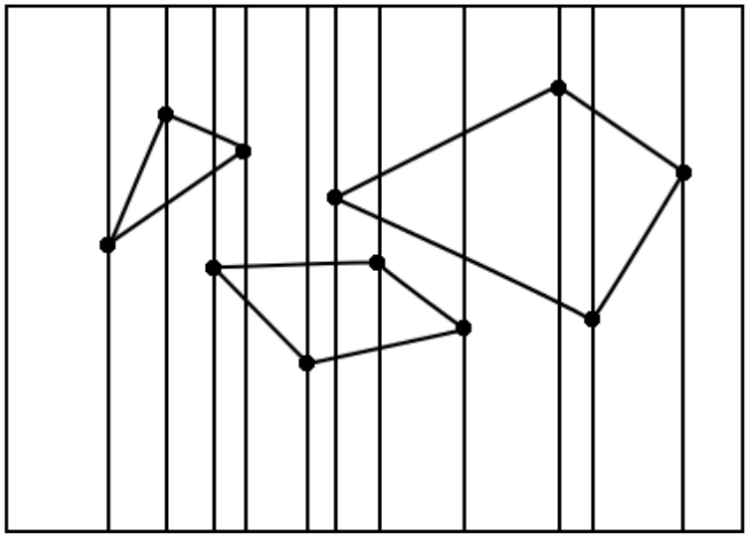

Figura 1: Partição em fatias

Para nos ajudar a visualizar o problema, vamos apresentar uma estrutura mais simples que satisfaça nossos pedidos de localização. Traçamos linhas verticais através dos vértices dos polígonos, como na figura 1. Isto particionará o plano em fatias verticais. Como iremos guardar as coordenadas x dos vértices em um vetor ordenado, isto torna possível determinar em qual fatia vertical se encontra q em tempo $O(\log n)$. Como não temos segmentos que se cruzam, então dentro das nossas fatias verticais não haverá vértices, apenas segmentos que não se cruzam. Isto significa que podemos ordenar de cima para baixo os segmentos contidos em cada fatia vertical.

Agora podemos enunciar o algoritmo de busca completo da sequinte maneira: Primeiro executamos uma busca binária pela coordenada x para encontrarmos a fatia vertical em que o ponto q se encontra e em seguida podemos fazer uma busca binária para saber onde o ponto se encontra dentro daquela fatia. A operação que precisamos para isto é: Dado um ponto q e um segmento s, determinar se q está acima ou abaixo de s.

O tempo de localização é bom, nós apenas fazemos duas buscas binárias. O primeiro vetor (de fatias verticais) tem no máximo tamanho n no caso em que todos os vértices têm coordenada x diferente. O segundo vetor também tem tamanho $O(n)$ pois cada fatia pode ser cruzada por no máximo $n/2$ segmentos.

Quanto à memória, podemos observar que o vetor com informações sobre as fatias verticais usa espaço $O(n)$, mas temos que considerar também que usamos um vetor de segmentos para cada fatia, que pode ter até $n/2$ segmentos cruzando cada

fatia vertical, o que nos leva a $O(n^2)$ de espaço total.

Usando uma ideia parecida podemos definir o que chamaremos de mapa trapezoidal. Os segmentos na periferia do mapa irão nos atrapalhar agora, então por questão de conveniência incluiremos um grande quadrado que chamaremos de R cobrindo todos os elementos de P. Isto não é um problema quando queremos procurar por pontos fora de R pois estes pontos sempre estarão fora dos polígonos.

Assumiremos também que não há vértices que compartilhem a mesma coordenada x, esta restrição não é muito realista pois em situações reais, é comum termos pontos na mesma coordenada x, ou até mesmo segmentos verticais. Mais tarde iremos mostrar como retirar essa restrição.

Então temos: S que é o conjunto de segmentos que formam os polígonos de P, onde $|S| = n$, envoltos por um quadrado R com a propriedade que não há dois vértices que compartilhem a mesma coordenada x, traçaremos duas linhas verticais de todos os pontos extremos dos segmentos (vértices), uma para cima e outra para baixo, parando em outra reta de S ou quando tocarmos uma das arestas de R.

O mapa trapezoidal de S é simplesmente a subdivisão induzida por S, o quadrado R e as linhas verticais citadas acima. Veja figura 2.

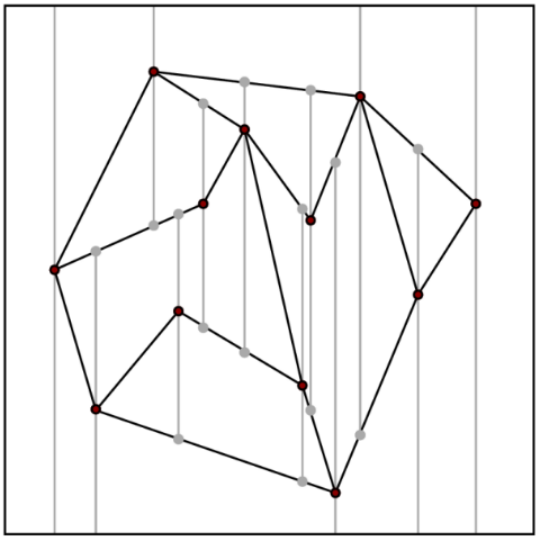

Figura 2: Um exemplo de mapa trapezoidal

2.1.2 Computando o mapa trapezoidal

Descreveremos aqui um algoritmo para computar o mapa trapezoidal $T(S)$ de um conjunto de segmentos S com n elementos em posição geral. Para discutir o algoritmo temos que definir uma estrutura de busca D, que auxiliará na busca de pontos dentro do mapa. Esta estrutura é um grafo dirigido e acíclico, que terá, quando concluído uma única origem (onde começamos a busca) e um único nó para

cada trapézio definido pelo mapa trapezoidal de S. Há dois tipos de nós em D, os $nós_x$ representam os vértices dos polígonos ou o ponto final das retas e os $nós_y$ representam os próprios segmentos.

Quando fizermos uma consulta por um ponto q, começamos a busca pela origem e prosseguimos até uma das folhas, onde estão os trapézios. Em cada um dos nós deste caminho, se estivermos em um $nó_y$ a pergunta é: o ponto q está acima ou abaixo da reta representada por este nó? Baseado nestas perguntas decidimos se andamos pela aresta da esquerda ou pela aresta da direita a partir do nó atual.

O algoritmo que apresentaremos é incremental, isto é, ele adiciona os segmentos um a um no mapa trapezoidal, atualizando a estrutura da mesma maneira. A ordem em que os segmentos são inseridos no mapa influencia a estrutura de busca, fazendo com que o tempo de busca não seja bom em todos os casos. Provaremos mais adiante que o tempo de busca quando a estrutura é construída de maneira aleatória é boa. Comecemos com o esqueleto do algoritmo:

algoritmo MAPA-TRAPEZOIDAL(S)
entrada. Um conjunto S de segmentos que não se cruzam.
saida. O mapa trapezoidal $T(S)$ e a estrutura D para $T(S)$.

1. Determinar a estrutura R que contenha os elementos de S.
2. Inicializar T e D.
3. Compute uma permutação aleatória dos elementos de S.
4. **para** $i \leftarrow 1$ até n **faça**
5. Ache o conjunto $\Delta_0, \Delta_1, ..., \Delta_k$ de trapézios que s_i intersecta.
6. Remova $\Delta_0, \Delta_1, ..., \Delta_k$, criando folhas para os novos trapézios formados.
7. Adicione essas novas folhas na estrutura.
8. Devolva T e D

Para acharmos $\Delta_0, \Delta_1, \ldots, \Delta_k$ usaremos a seguinte rotina.

algoritmo SIGA-SEGMENTO(T, D, s_i)
entrada. O mapa T, sua estrutura de busca D e o novo segmento s_i.
saida. A sequência $\Delta_0, \Delta_1, \ldots, \Delta_k$ de trapézios que é intersectada por s_i.

1. Sejam p e q os dois pontos das extremidades esquerda e direita de s_i.
2. Localizar p na estrutura D para achar Δ_0.
3. $j \leftarrow 0$.
4. **enquanto** q está à direita do ponto que define Δ_j pela direita **faça**
5. **se** o ponto da direita de Δ_j está abaixo de s_i
6. **então** Δ_j recebe o vizinho abaixo de Δ_j
7. **senão** Δ_j recebe o vizinho acima de Δ_j
8. $j \leftarrow j + 1$
9. devolva $\Delta_0, \Delta_1, \ldots, \Delta_k$

2.1.3 Complexidade

Teorema. O algoritmo MAPA-TRAPEZOIDAL computa o mapa trapezoidal $T(S)$ de um conjunto S de n segmentos em posição geral e a estrutura de busca D em tempo esperado $O(n \log n)$, o tempo esperado de busca de um ponto q é $O(\log n)$ e o tamanho esperado de D é $O(n)$.

Prova. Sabemos que o tempo de busca é linear no tamanho do caminho a ser percorrido na estrutura, então basta inferirmos o tamanho esperado do maior caminho. É fácil ver que um caminho cresce de no máximo 3 a cada iteração, logo, $3n$ é um limitante superior para o tempo de busca. Este é o pior caso possível na escolha da ordem em S. Como inserimos os elementos em ordem aleatória, a chance de ocorrer o pior caso deve ser pequena. Vamos investigar o tamanho esperado do caminho neste caso.

Considere um caminho correspondente à busca de um ponto q em D. Todos os nós nesse caminho foram criados em alguma iteração do algoritmo. Seja X_i, com $1 \leq i \leq n$, o número de nós criados no caminho na iteração i. Se considerarmos S e q fixos, então X_i é uma variável aleatória que depende apenas da permutação dos segmentos. Logo podemos expressar o tamanho esperado do caminho como:

$E[\sum_{i=1}^{n} X_i] = \sum_{i=1}^{n} E[X_i]$

Como observamos anteriomente, cada iteração adiciona no máximo 3 novos nós nos caminhos, então $X_i \leq 3$. Isto quer dizer que se P_i representa a probabilidade de um nó ser criado no caminho da busca de q na iteração i, teremos:

$E[X_i] \leq 3P_i$

Considere S_i, o subconjunto de S que contém os segmentos processados até a iteração i do algoritmo e o mapa trapezoidal $T(S_i)$. Note que $\Delta_q(S_i)$ é unicamente definido como função de S_i e não depende da ordem em que os segmentos de S_i foram inseridos. Para inferirmos P_i temos que pensar ao contrário, consideraremos $T(S_i)$ e olharemos para a probabilidade de $\Delta_q(S_i)$ desaparecer do mapa trapezoidal quando removermos o segmento s_i. O trapézio $\Delta_q(S_i)$ desaparecerá se acontecer um dos seguintes casos:

1. o segmento de cima de $\Delta_q(S_i)$ for removido.
2. o segmento de baixo de $\Delta_q(S_i)$ for removido.
3. o segmento da esquerda de $\Delta_q(S_i)$ for removido.
4. o segmento da direita de $\Delta_q(S_i)$ for removido.

Todos eles têm chance de $1/i$ de acontecer, pois na iteração i há exatamente i segmentos no mapa trapezoidal. Assim teremos:
$\sum_{i=1}^{n} E[X_i] \leq \sum_{i=1}^{n} 3P_i \leq \sum_{i=1}^{n} 12/i = 12 \sum_{i=1}^{n} 1/i = 12H_n$
onde $H_n := 1/1 + 1/2 + 1/3 + ... + 1/n$
Sabemos que para $n > 1$, $\ln(n) < H_n < \ln(n) + 1$. Assim concluímos que o tempo esperado de busca é $O(\log n)$.

Para estimarmos o tamanho de D basta verificarmos quantos nós há em D. As folhas equivalem ao número de trapézios do mapa trapezoidal, logo sabemos que há $O(n)$ nós que são folhas. Seja k_i o número de novos trapézios na iteração i, sabemos que o número de novos nós internos é exatamente $k_i - 1$. Logo o número de nós em D é limitado por:
$O(n) + E[\sum_{i=1}^{n}(k_i - 1)] = O(n) + \sum_{i=1}^{n} E[k_i]$.
Agora falta estimarmos um limite para k_i. Considere $S_i \subseteq S$.
Então:

$$\delta(\Delta, s) := \begin{cases} 1 & \text{se } \Delta \text{ desaparece de } T(S_i) \text{ quando } s \text{ for removido de } S_i \\ 0 & \text{caso contrário} \end{cases}$$

Na análise de tempo de busca vimos que até 4 segmentos podem fazer um trapézio desaparecer, logo:
$\sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \leq 4|T(S_i)| = O(i)$.
Como k_i é o número de trapézios criados na intersecção de s_i, ou equivalentemente o número de trapézios em $T(S_i)$ que aparece quando s_i é removido e s_i é um elemento aleatório de S_i, nós podemos estimar o valor esperado de k_i tomando a média sobre todos os segmentos de S:
$E[k_i] = 1/i \sum_{s \in S_i} \sum_{\Delta \in T(S_i)} \delta(\Delta, s) \leq O(i)/i = O(1)$.
Concluímos assim que o número de nós criados é $O(1)$ a cada iteração do algoritmo e portanto $O(n)$ é um limitante para o consumo de memória.

Nos falta agora provar o consumo de tempo para construção do mapa. Devemos observar que o tempo para inserir o segmento s_i é $O(k_i)$ mais o tempo para localizar

o ponto da extremidade esquerda do segmento s_i em $T(S_{i-1})$, usando as análises anteriores podemos deduzir que o consumo de tempo esperado do algoritmo é dado por:

$O(1) + \sum_{i=1}^{n} O(\log i) + O(E[k_i]) = O(n \log n).\square$

2.1.4 Grafo de caminhos livres

Com o mapa trapezoidal em mãos, temos que representar agora a região livre de obstáculos. Para isto, basta retirarmos do mapa os trapézios que se encontram dentro dos polígonos.

Vejamos como isso nos ajuda a calcular o caminho livre de obstáculos entre dois pontos, digamos s e t, em P. Se os dois pontos então contidos no mesmo trapézio, então é fácil: os trapézios representam regiões livres, logo, basta traçarmos uma linha reta entre os dois. Agora, caso eles estejam em trapézios diferentes, não basta traçarmos uma linha reta, pois algumas vezes precisamos fazer curvas para entrar em determinados trapézios. Para nos ajudar, construiremos o que vamos chamar aqui de grafo de caminhos livres.

Neste grafo os centros dos trapézios e o centro de cada linha vertical serão os vértices e haverá arcos ligando os centros dos trapézios aos centros das linhas verticais que são adjacentes a estes trapézios. Note que estes arcos estão na região onde o robô pode andar, pois tanto os trapézios como as linhas verticais estão na região livre do mapa.

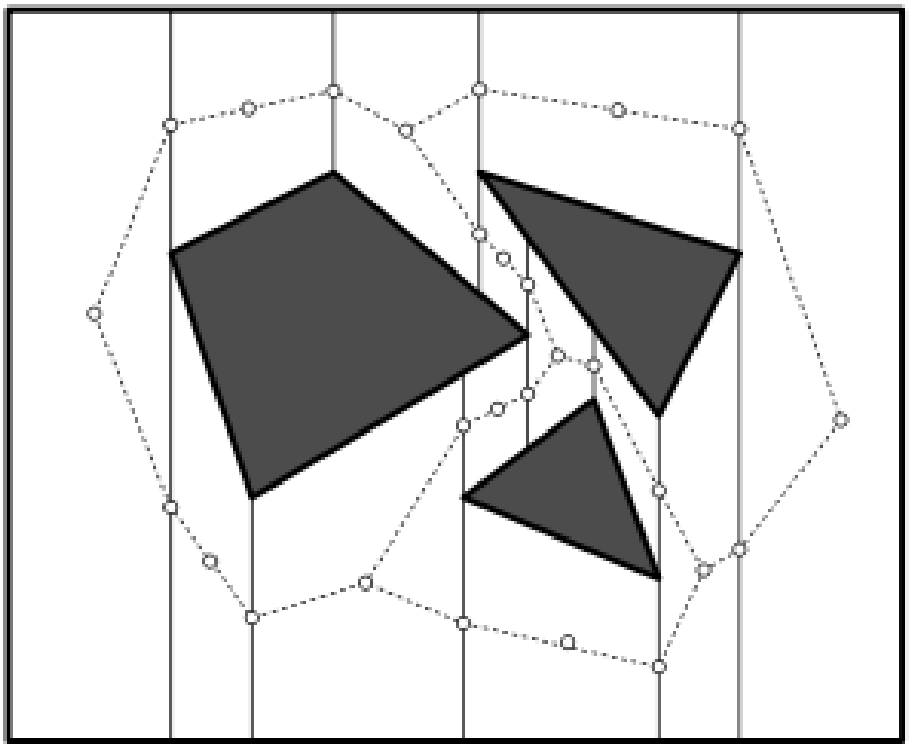

Figura 3: Um exemplo de grafo de caminhos livres.(Fonte: Computacional Geometry Algorithms and Applications,[1])

Com o grafo montado basta adicionarmos uma aresta de s ao vértice que se encontra no meio do trapézio onde s está contido, e adicionamos também uma aresta de t ao vértice do seu trapézio, depois disto usamos busca em largura, busca em profundidade ou até mesmo o algoritmo de Dijkstra para acharmos um caminho possível de s a t.

2.2 Caminhos mínimos

Vamos considerar agora o problema de encontrarmos um caminho de menor distância euclideana. A figura 4 ilustra a diferença entre o método anterior e o melhor caminho.

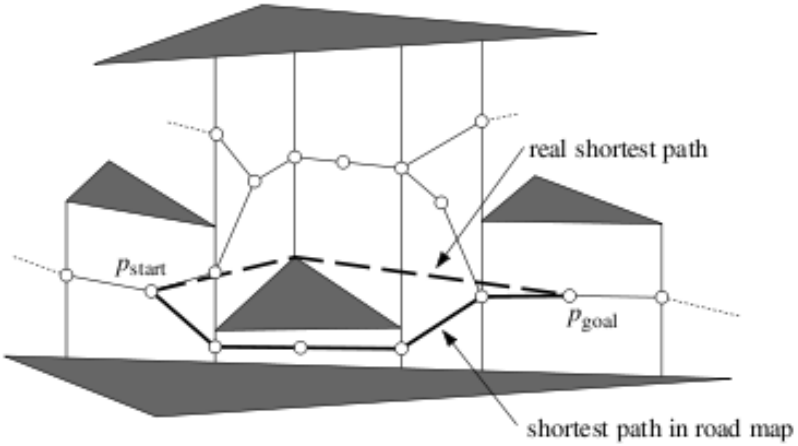

Figura 4: O caminho mínimo não está no grafo.(Fonte: Computacional Geometry Algorithms and Applications,[1]

O melhor caminho nem sempre é o mesmo, depende do robô que estamos considerando, como já discutimos no início.

Alguns robôs demoram para virar, então seria bom um caminho que além de ser curto tenha poucas curvas. Iremos considerar aqui um robô pontual que se move através de um conjunto disjunto de polígonos S. Como sempre esses polígonos são chamados de obstáculos e o número de arestas dos polígonos de S é denotado por n. Temos também dois pontos s e t que serão os pontos inicial e final do caminho respectivamente, o que estamos interessados é um caminho com menor distância euclideana entre s e t. Se considerarmos os polígonos como conjuntos fechados, ou seja, se não pudermos fazer nosso robô andar pelas bordas dos obstáculos, então não existe caminho mínimo pois sempre seria possível chegar mais perto dos obstáculos. Logo consideramos que os obstáculos são polígonos abertos e podemos caminhar sobre sua borda.

Não faz muito sentido considerarmos fazer curvas durante o caminho, pois podemos cortar caminho por uma linha reta e fazer com que o caminho seja menor. O lema seguinte formaliza esse conceito.

Lema - caminhos poligonais Qualquer caminho mínimo entre dois pontos s e t entre os polígonos em S é um caminho poligonal onde seus vértices são vértices dos polígonos em S.

Prova Suponha por absurdo que o caminho C não seja poligonal e seja mínimo. Como os obstáculos são poligonais, isto significa que existe um vértice p em C que está contido no espaço livre, ou seja, existe um ϵ tal que o disco D com centro em p e raio ϵ está totalmente contido em um espaço livre, logo, pegamos os dois pontos onde o caminho se cruza com D e conectamos eles com uma linha reta ao invés de

passarmos por p. Isto é um atalho digamos assim, logo, encontramos um caminho de comprimento menor que C, pois qualquer caminho de comprimento mínimo tem que ser mínimo localmente também. Desta forma podemos concluir que o caminho tem que ser poligonal e excluímos a possibilidade de haver pontos contidos no espaço livre. De maneira similar conseguimos excluir os vértices encontrados nas arestas dos polígonos pois podemos usar um argumento semelhante, agora ao invés de ter um disco, temos a metade de um, e podemos usar o mesmo argumento para diminuir o caminho. Assim só nos resta os vértices dos polígonos para p.□

2.2.1 Grafo de Visibilidade

Podemos definir agora como $Gvis(P^*)$ o grafo de visibilidade de P onde seus vértices são os vértices contidos em $P \bigcup \{s, t\} = P^*$, onde s e t são os pontos para os quais queremos o menor caminho. Por definição os arcos em $Gvis(P^*)$ são entre vértices, que agora incluem s e t, onde há caminho em linha reta entre eles, e que não colida com obstáculos.

Sabemos assim que o menor caminho entre s e t consiste de arcos neste grafo, logo, podemos usar o seguinte algoritmo para achar o menor caminho:

algoritmo CAMINHO-MINIMO(P, s, t)
entrada. Um conjunto P de polígonos e dois pontos s e t localizados em algum espaço livre.
saida. O menor caminho sem colisão entre s e t.

1. $Gvis(P^*) \leftarrow$ GRAFO-DE-VISIBILIDADE(P^*)
2. $\forall i, j \in Gvis(P^*), custo(i, j) \leftarrow$ a distância euclideana do segmento $\bar{ij}$.
3. Use o algoritmo de Dijkstra para calcular o caminho mais curto entre s e t.
4. Devolva o caminho.

2.2.2 Construindo o grafo de visibilidade

Seja S o conjunto de obstáculos em forma de polígonos disjuntos no plano com n arestas no total. Nós temos que decidir quais duplas de vértices se "enxergam", ou seja, têm um trajeto em linha reta livre de obstáculos entre os dois (uma aresta no grafo de visibilidade). Um algoritmo $O(n^3)$ é imediato, basta para cada segmento $\overline{ij}$ com $i, j \in P$ testar se $\overline{ij}$ cruza com alguma aresta de polígonos em P.

Se executarmos os testes, não em ordem arbitrária, mas considerando uma ordem especial, podemos desenvolver um algoritmo mais eficiente que o algoritmo ingênuo.

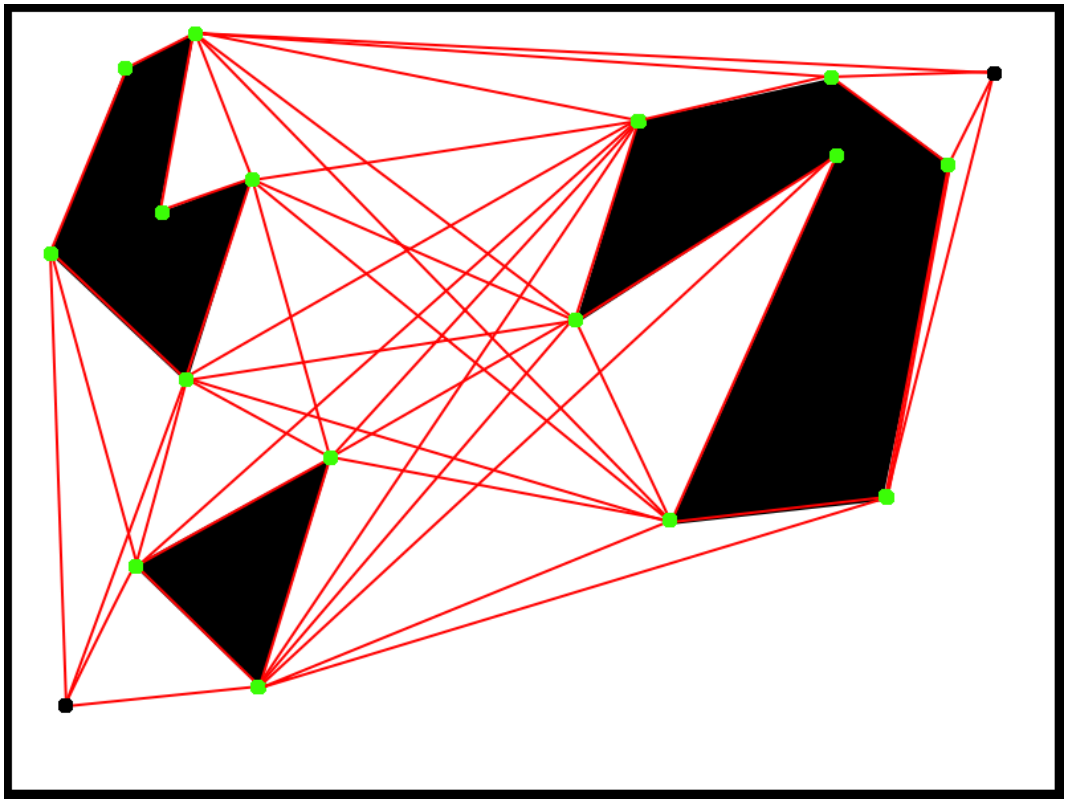

Figura 5: Em vermelho as arestas do grafo, em verde os vértices dos polígonos e em preto, s, t e a região interior dos polígonos

Para cada vértice p, decidiremos quais outros vértices são visíveis a partir de p. Quando analisamos se um vértice w é visível a partir de p temos que consultar se o segmento $\overline{pw}$ intersecta algum dos obstáculos. Agora considerando todos os vértices do nosso conjunto e um p fixo, uma ordem intuitiva para tratar os outros pontos é a ordem horária ao redor de p tentando armazenar alguma informação sobre os pontos que já foram tratados previamente.

Um vértice w é visível a partir de p se o segmento $p\bar{w}$ não intersecta o interior dos obstáculos, se considerarmos ρ um semi-reta que começa em p e passa por w, bastaria ver se ρ não cruza com alguma aresta pertencente a um obstáculo antes que atinja w, ou seja, podemos executar uma busca binária por w no conjunto de arestas que intersectam com ρ. Caso p seja um vértice de algum dos obstáculos, então há o caso em que w pertença ao mesmo polígono e não há segmentos entre p e w, mas w não é visível a partir de p (o segmento está dentro do polígono).

Conforme tratamos os vértices em ordem horária, mantemos uma árvore balanceada para guardar os segmentos que cruzam com ρ, ou seja, nós rodamos a semi-reta ρ ao redor de p. Algoritmos deste tipo são chamados de algoritmos de linha de varredura.

O que será guardado na linha de varredura são os segmentos (as paredes dos

obstáculos) na ordem em que são intersectados por ρ. Os eventos que atualizam a linha são os vértices de S. Para decidir se um vértice w é visível a partir de p, temos que buscar por p na estrutura de varredura, que chamamos de T e depois atualizamos a estrutura inserindo ou removendo arestas incidentes em w.

Vamos definir então o algoritmo.

algoritmo GRAFO-DE-VISIBILIDADE(P, s, t)
entrada. Um conjunto P de polígonos que não se intersectam, dois pontos s e t fora dos polígonos.
saida. O grafo de visibilidade $Gvis(P^*)$.

1. $V \leftarrow P \bigcup \{s, t\}$
2. **para cada** $p \in V$ **faça**
3. $\quad O \leftarrow$ ORDENA($p, V - \{p\}$)
4. $\quad T \leftarrow$ INICIALIZA(p)
5. $\quad$ **para cada** $q \in O$ **faça**
6. $\quad\quad$ **se** VISIVEL(T, p, q)**faça**
7. $\quad\quad\quad$ Adicione a aresta pq em $GVis(P)$
8. $\quad\quad$ ADICIONA(T, p, q)
9. $\quad\quad$ REMOVA(T, p, q)
10. devolva $GVis(P)$

O algoritmo usa algumas subrotinas que detalharemos a seguir:

- ORDENA(p, S) recebe um ponto p e um conjunto de pontos S e ordena os pontos em S no sentido anti-horário começando do eixo x em relação a p. Esta rotina tem complexidade $O(n \log n)$, para executarmos as comparações basta usarmos funções trigonometricas que geralmente já estão implementadas nas linguagens de programação mais conhecidas. A função devolve um conjunto de pontos ordenado que será a ordem que trataremos os pontos eventos.
- INICIALIZA(p) inicializa a estrutura da linha de varredura com os segmentos que cruzam a semi-reta paralela ao eixo x e que tem como início p e sentido positivo. Esta função também gasta tempo $O(n \log n)$ no pior caso, quando $O(n)$ segmentos cruzam a semi-reta citada.

- VISIVEL(T, p, q) decide se q é visível a partir de p vendo se o segmento $\overline{pq}$ cruza com o segmento mais perto de p na linha de varredura. Além desta verificação também precisamos tratar alguns casos especiais, como o caso em que o segmento $\overline{pq}$ está inteiramente contido em algum polígono de S, neste caso precisaríamos fazer uma consulta para verificar se um ponto da reta está dentro do polígono ao qual p e q pertencem. Esta consulta demoraria $O(n)$ se feita de qualquer maneira pois não fazemos suposições sobre os polígonos serem ou não convexos fazendo com que nosso algoritmo ficasse ineficiente, mas graças ao mapa trapezoidal, descrito anteriormente, podemos executar esta consulta em tempo $O(\log n)$, que não nos atrapalha.

- ADICIONA(T, p, q), insere os segmentos que começam em q e têm como outra ponta um ponto no sentido anti-horário em relação a linha de varredura, na estrutura balanceada T que representa a linha de varredura. A cada ponto evento, nosso critério de comparação muda, seguindo a direção da linha. A ordem é imposta pela ordem em que a linha de varredura cruza os segmentos contidos em T. Como os segmentos não se cruzam, a ordem de comparação não mudará entre um ponto evento e outro, garantindo que a estrutura continue consistente em todo o algoritmo.

- REMOVA(T, p, q), remove da estrutura balanceada T os segmentos que começam em q e têm como outra ponta um ponto no sentido anti-horário em relação a linha de varredura.

2.2.3 Complexidade

Podemos analisar o consumo de tempo da construção do Grafo de visibilidade: Temos para cada ponto uma ordenação por ângulo que custa $O(n \log n)$ nos levando a $O(n^2 \log n)$, após a ordenação, para cada outro ponto verificamos a linha de varredura, inserimos, removemos e muitas vezes consultamos o mapa trapezoidal, cada uma destas operações tem consumo de tempo $O(\log n)$ levando a um total de $O(n^2 \log n)$. Vale ressaltar que se o número de vértices dos polígonos é n então o número de segmentos com que estamos trabalhando também é n, portanto faz sentido falar em função de n quando nos referimos a segmentos ou vértices.

2.3 Tratamento de casos degenerados

Nas seções anteriores colocamos algumas restrições. Em primeiro lugar assumimos que os pontos estavam em posição geral, ou seja, não haveria dois pontos com mesma coordenada x. Para este caso podemos notar que se fizermos uma pequena rotação nos pontos, tal que os pontos não troquem de ordem resolveríamos o problema, mas transformações levam a problemas numéricos de precisão. Uma abordagem melhor é usar uma pequena perturbação proporcional ao y de cada ponto,

isto nos leva a um mapeamento dos pontos chamado transformação de Shear. Segue um exemplo ilustrativo:

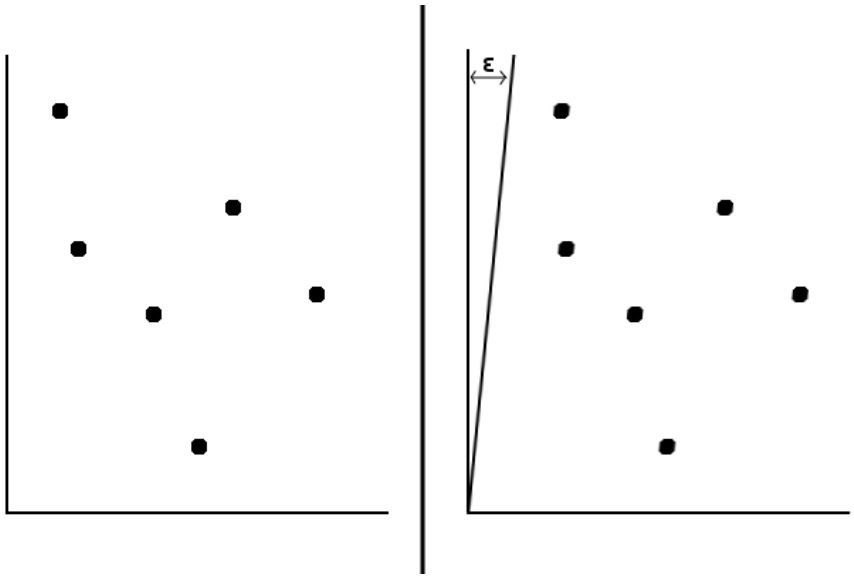

Figura 6: Exemplo da transformação de Shear

No desenho estamos usando uma transformação no eixo x por algum valor ϵ:

$$\varphi := \begin{pmatrix} x \\ y \end{pmatrix} \rightarrow \begin{pmatrix} x + \epsilon y \\ y \end{pmatrix}$$

Se $\epsilon > 0$ for pequeno suficiente, a transformação φ preserva a ordem dos pontos.

Sabendo disto, quando recebemos como entrada um conjunto de pontos S, podemos passar para os algoritmos vistos um conjunto novo $\varphi S := \{\varphi s | s \in S\}$. Lidar com transformações nos leva a problemas de aproximação e arredondamento por causa do ponto flutuante como foi dito antes. Mas aqui há um truque. No caso do mapa trapezoidal por exemplo queremos fazer dois tipos de comparação com os pontos. Na primeira temos dois pontos q e s e queremos saber se p está abaixo ou acima de s, para a outra operação, tínhamos um segmento s e um ponto p e queríamos saber se p estava a direita, esquerda ou sobre s.

Nossa transformação preserva a relação entre pontos, ou seja se tivermos dois pontos $p = (x_p, y_p)$ e $q = (x_q, y_q)$ e $x_p \leq y_p$ então $x_p + \epsilon y_p \leq x_q + \epsilon y_q$, assim como preserva a relação entre segmentos e pontos ou seja se p está à direita, esquerda ou sobre um segmento s, então φp estará à direita, esquerda ou sobre φs respectivamente.

Note agora que para fazer as comparações não precisamos efetivamente de ϵ, podemos simplesmente comparar os pontos lexicograficamente para impor uma ordenação, resolvendo assim os problemas numéricos.

2.4 O problema da rota do vigia

Iremos apresentar aqui outro problema de visibilidade, conhecido como problema da rota do vigia (*Watchman Route*). O problema consiste em acharmos uma rota interna a um polígono tal que todos os pontos do polígono são visíveis de algum ponto da rota. Uma possível aplicação seria pensar como a rota de um vigia noturno que deseja cuidar de uma galeria, e gostaríamos que além de vigiar toda a galeria, ainda o fizesse em pouco tempo, para isto temos o objetivo de minimizar o tamanho da rota.

Vamos mostrar aqui que este problema é NP-difícil para polígonos arbitrários.

2.4.1 Definindo o problema

Problema do vigia
Entrada: Polígono P com buracos, inteiro k
Questão: Existe uma rota de tamanho menor ou igual a k, tal que esta não intersecte o exterior de P ou o interior de algum dos buracos de P e todos os pontos do polígono são visíveis de algum ponto da rota?

Mostraremos que o problema do vigia é NP-Difícil através de uma redução ao problema do caixeiro viajante geométrico retilinear (**geometric traveling salesman**).

Eis a definição:
Caixeiro viajante geométrico retilinear
Entrada: Conjunto S com n pontos no plano, inteiro b
Questão: Existe um caminho de comprimento menor ou igual a b que visita todos os pontos de S através de um caminho retilinear?

2.4.2 Dificuldade do problema

Esta variante do problema do caixeiro viajante geométrico retilinear continua NP-difícil [10].Dizemos que o problema é retilinear se os caminhos só podem ser feitos com linhas horizontais ou verticais.

A trasformação envolve a construção de uma galeria com $O(n)$ corredores retilineares envoltos em um retângulo como na figura 10. As paredes dos corredores definem $O(n^2)$ buracos convexos no interior do polígono. Por fim inserimos, no lugar dos pontos, estruturas como as mostradas abaixo. Claramente a construção pode ser feita em tempo polinomial. Temos assim o seguinte resultado:

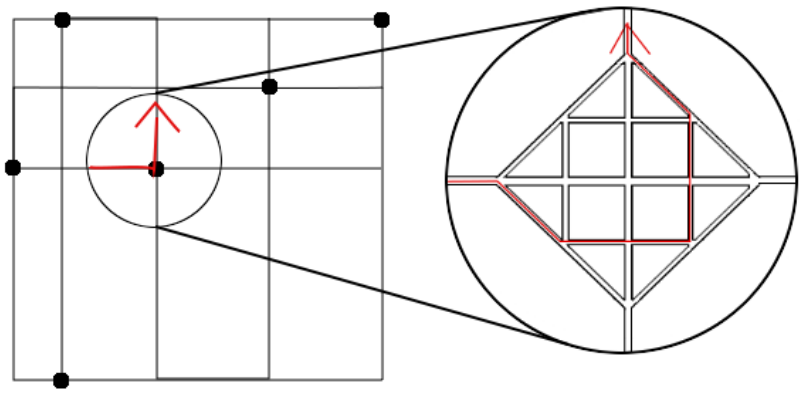

Figura 7: Método de construção para representar o problema do caixeiro como problema do vigia

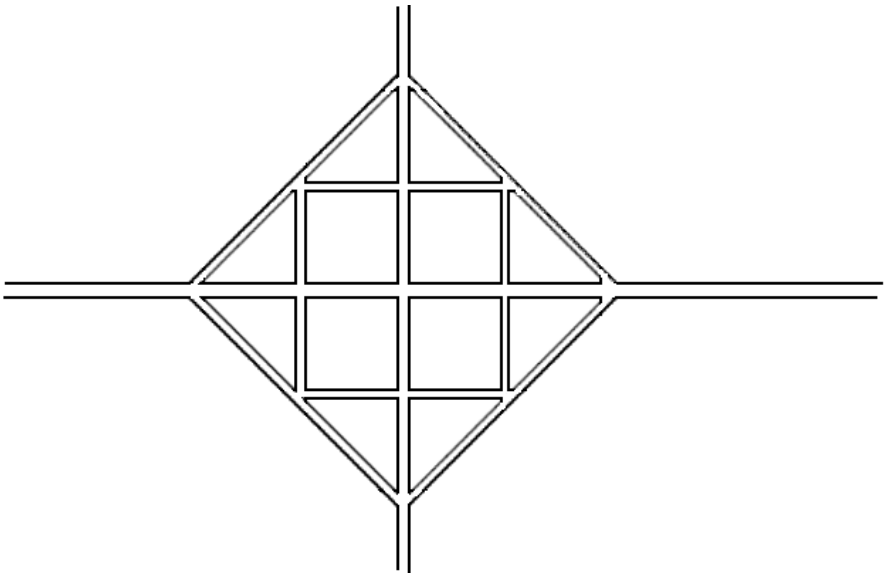

Figura 8: Estrutura inserida para moldar o problema do caixeiro no problema do vigia.

Teorema O problema do vigia é NP-difícil até mesmo com polígonos convexos e buracos convexos.

Prova Dada uma entrada para o problema do caixeiro viajante geométrico retilinear, construímos uma estrada para o problema do vigia como descrito acima.

Vamos chamar de R o retângulo que engloba todos os pontos de S. Devemos notar que a estrutura força as rotas produzidas pelo algoritmo a visitarem todas as intersecções que representam pontos de S e ainda, notemos que todos os corredores são visíveis a partir de alguma dessas intersecções. Assim teremos que o caminho que visita as intersecções correspondentes aos pontos de S será a rota do vigia, ou seja, todos os pontos do polígono serão visíveis pelo menos por um ponto na rota. Haverá uma solução do tamanho B para o problema do caixeiro se e somente se existir uma rota do vigia de comprimento $B + (2\sqrt{2}nx + 4nx) - (2b + 4c)x$, onde x é o tamanho do lado dos quadrados dentro da estrutura, b é o número de pontos em S que ficam nas arestas mais externas do retângulo R e c é o número de pontos de S que são pontas do retângulo externo R. □

Para explicar este número que relaciona os dois problemas usarei das próximas imagens que a menos de rotações e inversões, tentam representar todos os tipos de passagem pelos pontos na entrada do problema do vigia pelas estruturas que foram citadas acima e também deixa claro os casos dos cantos e das bordas.

Como estamos considerando que os quadrados têm lado x, para passar pelos cantos temos que percorrer uma rota com comprimento $2\sqrt{2}x$ para conseguirmos cobrir visualmente toda a estrutura.

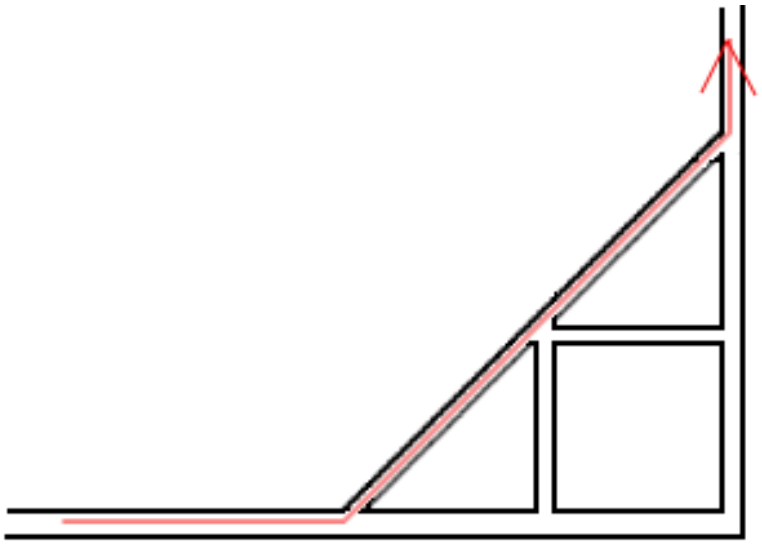

Figura 9: Aqui o comprimento da menor rota é $2\sqrt{2}x$

Para os pontos localizados na margem de R a menor rota tem comprimento $2\sqrt{2}x + 2x$, em todas as seis formas mínimas possíveis de se passar pela estrutura.

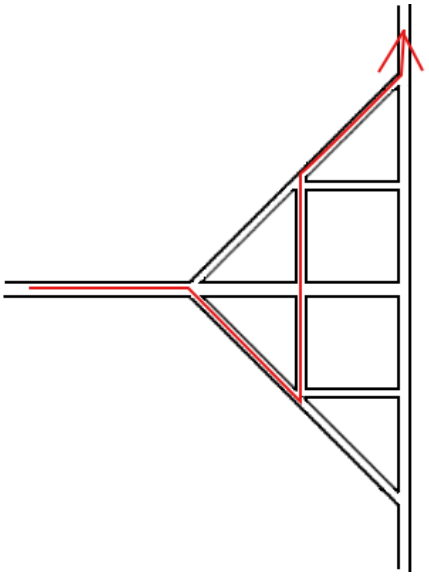

Figura 10: Aqui o comprimento da menor rota é $2\sqrt{2}x + 2x$

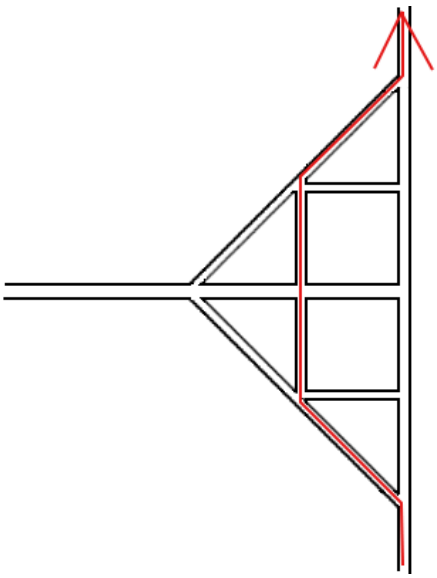

Figura 11: Aqui o comprimento da menor rota é $2\sqrt{2}x + 2x$

Já para os pontos centrais, a menor rota no problema do vigia tem comprimento $2\sqrt{2}x + 4x$ nas doze formas mínimas possíveis de se passar pela estrutura.

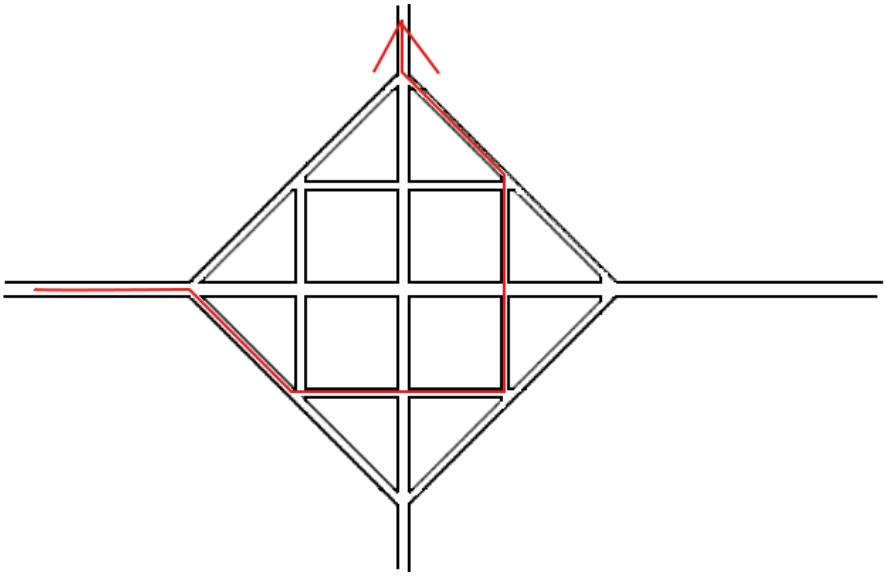

Figura 12: Aqui o comprimento da menor rota é $2\sqrt{2}x + 4x$

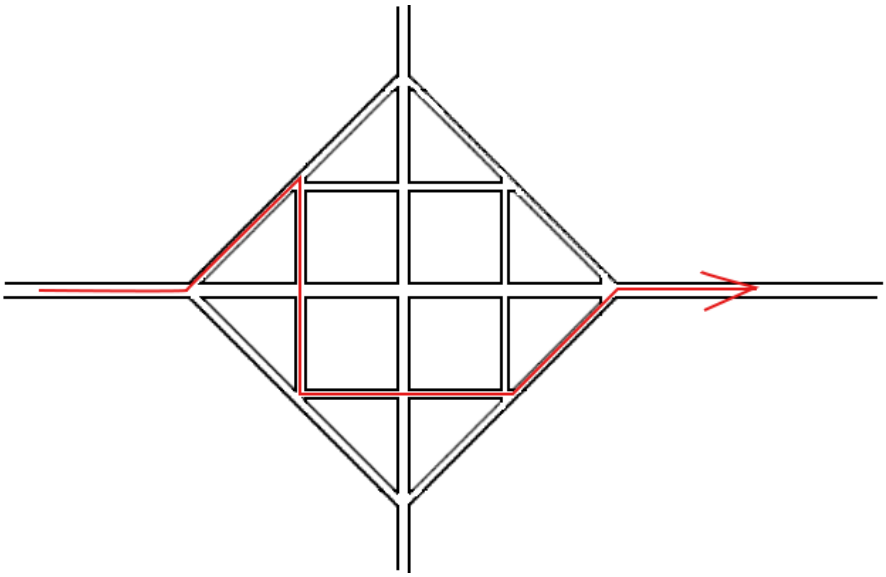

Figura 13: Aqui o comprimento da menor rota é $2\sqrt{2}x + 4x$

3 Resultados obtidos

O algoritmo implementado no programa resolve o problema do menor caminho euclidiano entre dois pontos sem colidir com o conjunto de obstáculos. O programa aceita tanto entradas feitas a mão quanto geradas aleatoriamente, facilitando a criação de casos de teste.

A seguir mostramos alguns screenshots de testes realizados com nosso programa:

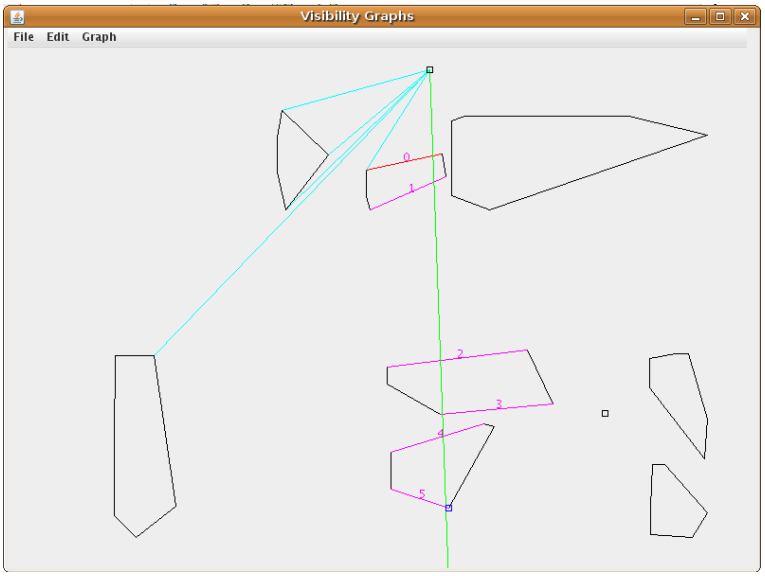

Figura 14: A linha de varredura e algumas arestas já inseridas.

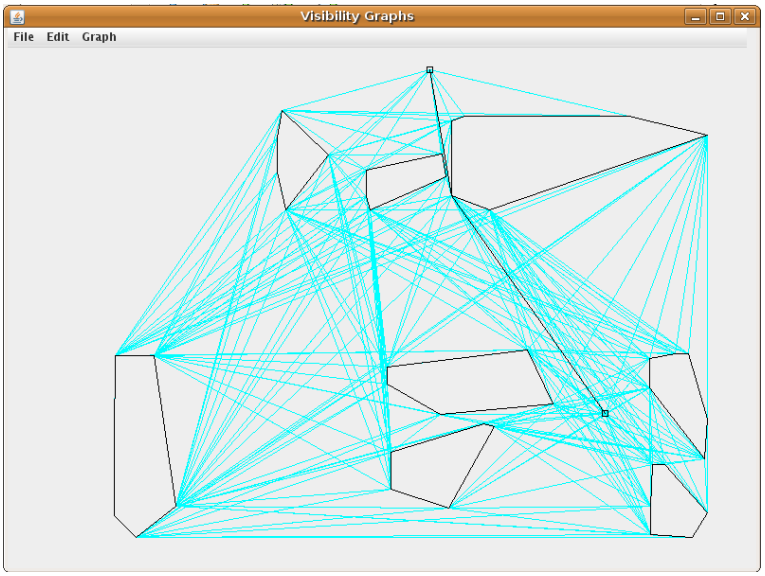

Figura 15: O grafo completo e o menor caminho

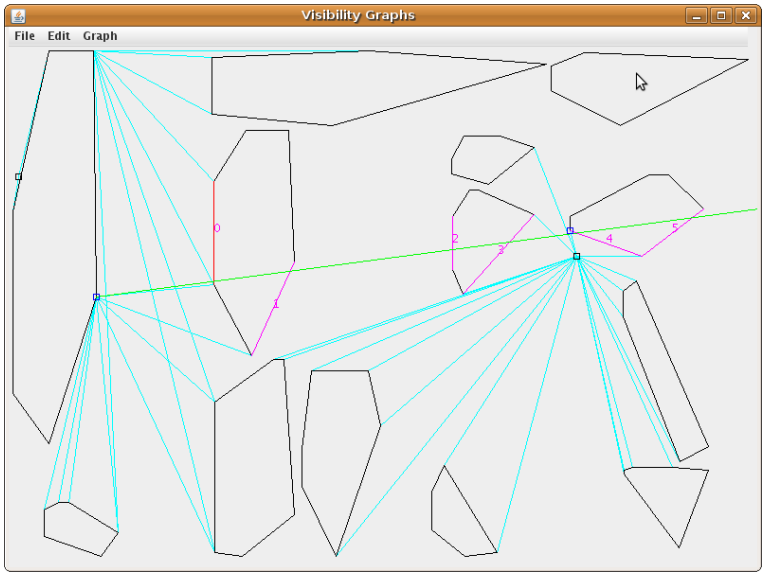

Figura 16: A linha de varredura e algumas arestas

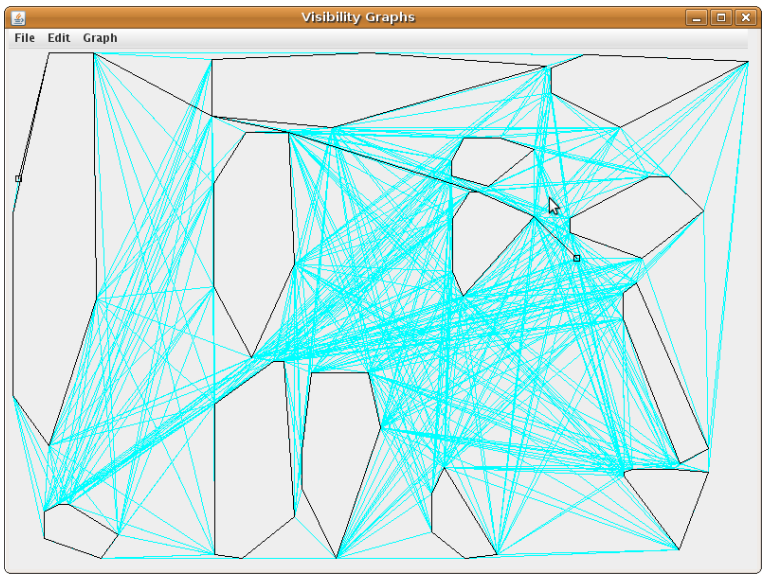

Figura 17: O grafo completo e o menor caminho

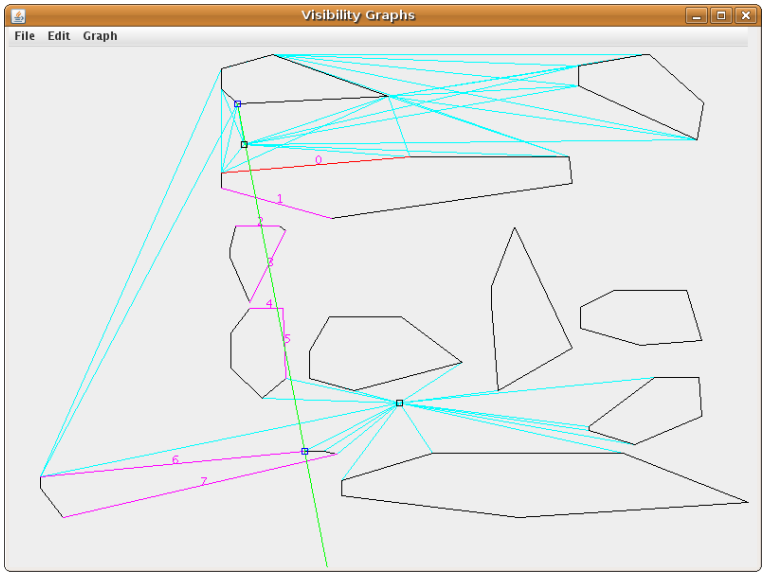

Figura 18: A linha de varredura e algumas arestas

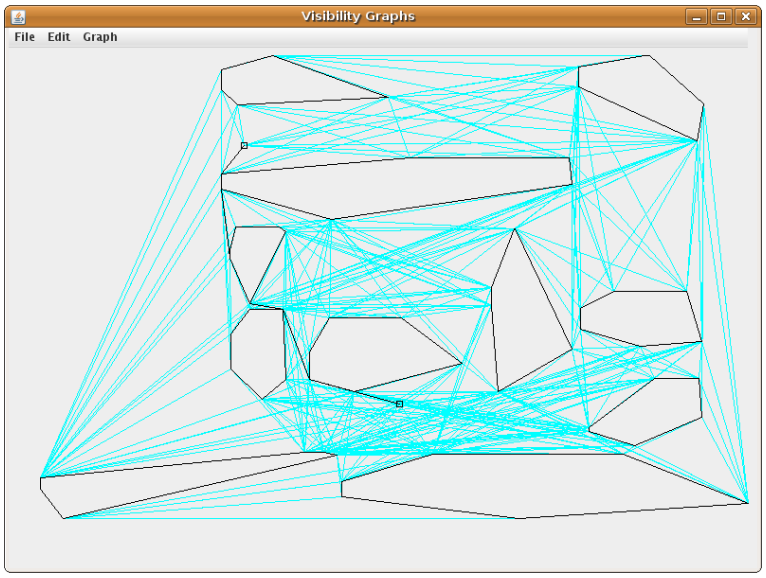

Figura 19: O grafo completo e o menor caminho

Foram implementados dois algoritmos para a solução do problema. Um mais ingênuo com complexidade $O(n^3)$, o outro com complexidade $O(n^2 \log n)$ descrito neste trabalho.

Implementamos também uma funcionalidade que permite fazer o *upload* de um programa que faz um robô do modelo NXT da Lego andar pelo caminho calculado pelo algoritmo. Exemplos desta funcionalidade podem ser encontrados no site YouTube através dos seguintes links:

- Vídeo 1 (http://www.youtube.com/watch?v=5ZWD7QSaZMI)
- Vídeo 2 (http://www.youtube.com/watch?v=DJHMvWKYw1g)
- Vídeo 3 (http://www.youtube.com/watch?v=gybuA2J36q0)
- Vídeo 4 (http://www.youtube.com/watch?v=m7NwY3XflEA)

E vídeos do robô andando através dos caminhos gerados acima podem ser encontrados nos seguintes links:

- Vídeo 1 (http://www.youtube.com/watch?v=iFtGrxy0zPQ)
- Vídeo 2 (http://www.youtube.com/watch?v=UfQpMN9QMPk)
- Vídeo 3 (http://www.youtube.com/watch?v=i5dMrIz_uQw)
- Vídeo 4 (http://www.youtube.com/watch?v=XOTtHYVE0KY)

4 Conclusão

Para a construção do grafo de visibilidade, existe um algoritmo melhor que o apresentado encontrado em [8]. Quanto ao problema do vigia, há algoritmos para a resolução do problema quando, ao invés de considerar polígonos arbitrários, consideramos alguns polígonos específicos, como polígonos retilineares sem buracos, ou polígonos simples. Nos problemas apresentados percebemos a dificuldade de problemas de geometria computacional e apesar de ser uma área antiga, há muitos estudos a serem feitos sobre o assunto.

Como citado anteriormente problemas de geometria computacional podem ter diversas aplicações práticas, como na área de robótica. Sendo assim, como motivação prática para a monografia de um modo geral, apresentamos o robô programável NXT da Lego.

Referências

[1] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computacional Geometry: Algorithms and applications (second edition)*, Springer, New York, 2005.

[2] Joseph O'Rourke, *Computacional Geometry in C (second edition)*, Cambridge University Press, United Kingdom, 1998.

[3] K. Mulmuley, *Computacional Geometry:an Introduction through Ramdomized Algorithms.*, Prentice Hall, Chigago, 1998.

[4] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[5] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press, 2005.

[6] K. Kedem, R. Livne, J.Pach, and M.Sharir, *On the union of Jordan regions and collision-free translation motion amdst polygonal obstacles*, Discrete Comput. Geom. **1** (1986), 59–71.

[7] K. Kedem and M.Sharir, *An efficient algorithm for planning collision-free translation motion of convex polygon object in 2-dimensional space amidst polygonal obstacles.*, Proceedings of the 1st Annual Symp. Comp. Geom. (1985), 75–80.

[8] S. K. Ghosh and D. M. Mount, *An output-sensitive algorithm for computing visibility graphs.*, SIAM J. Comput **20** (1991), 888–910.

[9] Wei-pang Chin, *Optimum watchman routes.*, Information processing letters **28 issue 1** (1988), 39-44.

[10] M. R. Garey, R. L. Graham, and D. S. Johnson, *Some NP-complete geometric problems.*, Proceedings of the eight annual ACM symposium on Theory of computing (1976), 10-22.

Parte II
Subjetiva

1 Desafios e frustrações

Os estudos na área de geometria computacional começaram quando me interessei pela área de algoritmos e procurei o Professor Carlos Eduardo, no primeiro semestre de 2007, quando ele lecionava a disciplina de Estrutura de dados. A partir de 2008 comecei a fazer algumas optativas eletivas tais como otimização combinatória e desafios de programação que só me fizeram gostar mais da área de combinatória. Em 2008 submetemos uma proposta de bolsa CNPq para estudar problemas e algoritmos de deslocamento no plano e o projeto foi aceito. Apresentei o projeto de iniciação científica no IV Simpósio de Iniciação Científica e Pós-Graduação do IME-USP e no Simpósio Internacional de Iniciação Científica da Universidade de São Paulo (SIICUSP).

O maior desafio foi conciliar as matérias, o estágio e o trabalho de conclusão. Com os encontros semanais com o orientador consegui entender melhor todos os algoritmos e me convencer de fato que as provas funcionavam. Gostaria de ter estudado alguns algoritmos para a resolução do problema do vigia mas, devido a complexidade dos artigos relacionados, não consegui entender direito e acabei desistindo de escrever sobre eles.

Outro desafio muito grande foi a parte do robô. O IME-USP possui um robô do modelo RCX da Lego. Além do modelo ser mais antigo e conectar-se ao computador através de portas pouco usadas atualmente, tivemos que limpá-lo pois o esquecimento de uma bateria fez com que esta vazasse dentro do aparelho. Depois de conseguir ligar e enviar o programa ao robô, percebemos que ele é instável e

possui configuração complicada. Apesar de seguirmos as instruções da biblioteca escolhida, o robô não andava como esperávamos e, por isso, não terminamos em tempo hábil para a apresentação.

Posteriormente, pedimos à POLI-USP o robô do modelo NXT que comportou-se de maneira esperada. Com isso conseguimos dar prosseguimento e concluir a parte do robô, que nos ajuda a visualizar na prática os algoritmos estudados.

2 Matérias relevantes ao trabalho

Dentre as matérias que me ajudaram a realizar este trabalho de formatura, as mais importantes são:

- **MAC0110 - Introdução à computação** Antes desta matéria eu não sabia nada de computação e foi nela que dei meus primeiros passos.
- **MAC0122 - Princípios de desenvolvimento de algoritmos** Vimos algumas estruturas de dados mais simples e alguns algoritmos não triviais, o que fez crescer a vontade de estudar mais a fundo combinatória e complexidade.
- **MAC0211 - Laboratório de programação** Foi nesta matéria que tive meus primeiros contatos com interface gráfica, o que ajudou na hora de construir um programa visual que simula o algoritmo. Além disso, aprendi ferramentas essenciais como LaTeX, a qual utilizei para escrever esta monografia.
- **MAC0323 - Estrutura de dados** Nesta matéria aprendi estruturas de dados mais complexas como árvores balanceadas que são usadas várias vezes nos algorimos apresentados nesta monografia.
- **MAC0327 - Desafios de programação** Nesta matéria meus conhecimentos em grafos, análise de algoritmos, programação dinâmica e muitos outros assuntos relevantes se solidificaram.
- **MAC0328 - Algoritmos em grafos** Nesta monografia, grafos foram intensamente usados e esta matéria como o próprio nome já diz foi um dos meus primeiros contatos com grafos.
- **MAC0338 - Análise de algoritmos** Me ajudou muito a discernir e visualizar a complexidade dos algoritmos.
- **MAC0331 - Geometria computacional** Esta é bem auto-explicativa.

Algumas outras matérias não menos importantes são MAT0139 Álgebra linear para computação, MAC0450 Algoritmos de Aproximação e introdução a teoria dos grafos.

3 Conceitos relevantes

Os conceitos mais relevantes são grafos, que, como já foi dito, foi usado intensamente em todos os assuntos apresentados na monografia, árvores binárias balanceadas, pois geralmente linhas de varredura são representadas computacionalmente por elas, e conceitos básicos de geometria analítica que facilitaram muito a compreensão das primitivas usadas na confecção dos algoritmos de geometria computacional.

4 Futuro

Uma possível continuação para este trabalho seria o estudo aprofundado do problema do vigia para um melhor entendimento dos algoritmos utilizados para os casos em que o polígono é simples ou retilinear sem buracos. Outra possibilidade seria estudar algoritmos mais eficientes para construção do grafo de visibilidade.

5 Agradecimentos

Agradeço à CNPq pelo apoio no desenvolvimento desta iniciação científica, ao orientador pela paciência e discussões que me fizeram aprender muito e à POLI-USP, em especial ao Giuliano Salcas Olguin, pelo empréstimo do robô. Agradeço também aos meus familiares e à Marcela pelo apoio durante todo o processo de confecção do TCC. Este trabalho de conclusão me ajudou a entender melhor o mundo de geometria computacional, assim como me incentivou a fazer matérias bem interessantes na área de otimização combinatória.