UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

O Problema da Visita de Polígonos

Gabriel Freire Ushijima

São Paulo, SP
2 de janeiro de 2026

Resumo

Em diferentes contextos, como robótica, planejamento de rotas e gráficos computacionais, surge a necessidade de encontrar caminhos eficientes que visitem uma série de regiões ou objetos no espaço. O Problema da Visita de Polígonos (TPP) é um problema clássico de otimização geométrica que busca determinar o caminho mais curto que visita uma sequência de polígonos convexos disjuntos em um plano. No presente trabalho se apresentam algoritmos para resolver o TPP em dois cenários distintos: o TPP Irrestrito, onde o caminho pode atravessar os polígonos, e o TPP Restrito, onde o caminho deve contornar os polígonos sem atravessá-los. Baseando-se no trabalho seminal de Dror et al. (2003), este relatório detalha a implementação prática dos algoritmos propostos, destacando suas complexidades computacionais e estratégias de otimização. Além disso, são apresentados resultados experimentais que ilustram a eficácia dos algoritmos em diferentes configurações de entrada. O código-fonte desenvolvido está disponível em um repositório público, facilitando a reprodução dos resultados e a exploração adicional do problema.

Sumário

Notações

$v = (v_1, v_2)$	Representação de vetores em $\mathbb{R}^2$ como tupla de coordenadas reais.
$\|v\| = \sqrt{v_1^2 + v_2^2}$	Norma Euclidiana de um vetor.
$\langle u, v \rangle = u_1 v_1 + u_2 v_2$	Produto interno definido como produto escalar de vetores.
$u \times v = u_1 v_2 - u_2 v_1$	Produto vetorial definido como determinante 2D, retornando um escalar. Essa definição tem a seguinte propriedade: $\text{sign}(u \times v) = \begin{cases} +1, & \text{se o ângulo no sentido } \textbf{anti-horário} \text{ entre } u \text{ e } v \text{ é menor que } 180° \\ -1, & \text{se o ângulo no sentido } \textbf{horário} \text{ entre } u \text{ e } v \text{ é menor que } 180° \\ 0, & u \text{ e } v \text{ colineares} \end{cases}$
$\overline{uv}$	Segmento de reta entre os pontos u e v.
$\overrightarrow{uv} = v - u$	Vetor direcionado do ponto u até o ponto v.
$\|P\|$	Se P é um polígono, então $\|P\|$ é o número de vértices de P.
∂P	Fronteira (perímetro) do polígono P.

1 Introdução

O Problema de Visita de Polígonos (TPP - *Touring Polygons Problem*) é um problema clássico de otimização geométrica que envolve encontrar o caminho mais curto que visita em ordem uma sequência de polígonos em um plano. O problema é ilustrado pela Figura 1.

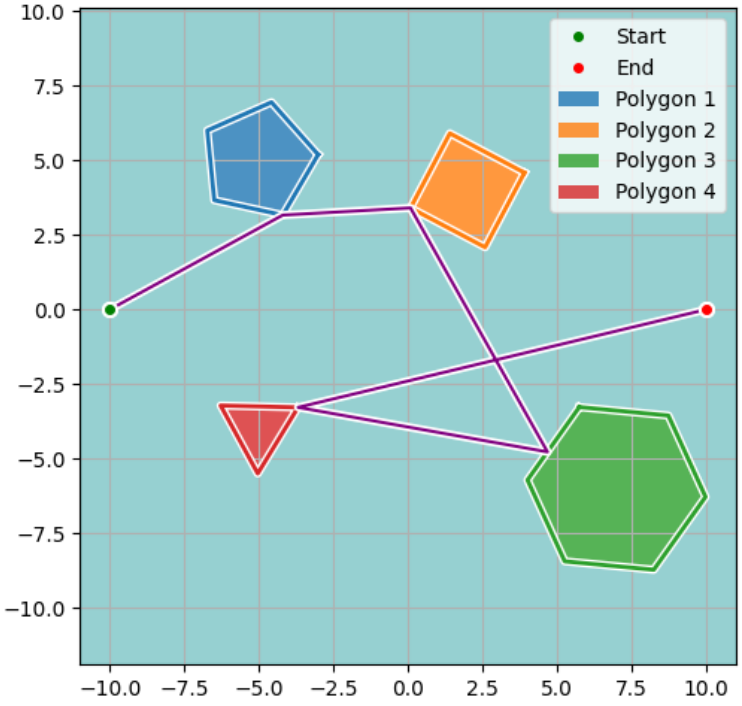


Figura 1: Exemplo do Problema de Visita de Polígonos Irrestrito com 4 polígonos.

Esse é um caso específico do problema do Caxeiro Viajante com Vizinhanças (TSPN - *Traveling Salesman Problem with Neighborhoods*) **tspn-iacopo-2011**, que é por sua vez uma generalização do famoso Problema do Caxeiro Viajante (TSP - *Traveling Salesman Problem*) **tsp-applegate2006**. Esse problema tem aplicações em diversas áreas, principalmente em robótica no contexto de planejamento de rotas, onde um robô precisa visitar uma série de regiões de interesse, representadas por polígonos, de maneira eficiente.

Enquanto os problemas de TSP e TSPN são conhecidos por serem NP-difíceis **tsp-applegate2006**, **tspn-iacopo-2011**, o TPP exige que a ordem de visita dos polígonos seja pré-definida, o que permite o desenvolvimento de algoritmos polinomiais para resolver o problema em certos casos. Em particular, Dror et al. (2003) **tpp-dror2003** propuseram algoritmos eficientes para resolver o TPP em dois cenários distintos: o TPP Irrestrito, onde o caminho pode atravessar os polígonos, e o TPP Restrito, onde o caminho deve se manter dentro de "cercas"que delimitam os polígonos, como se a visita dos polígonos fosse feita por um robô que não pode sair de uma área segura que engloba os polígonos.

Nesse relatório, buscamos discutir e implementar os algoritmos propostos por Dror et al. (2003) **tpp-dror2003**, apresentando uma visão prática do problema e suas soluções. Enquanto o artigo original foca bastante na análise teórica dos algoritmos, nosso objetivo é fornecer uma abordagem mais acessível e detalhada, enfatizando a implementação e os resultados experimentais. Vamos introduzir os conceitos fundamentais lentamente, junto de diversas visualizações, com o objetivo de facilitar o entendimento do leitor. Para quem deseja uma compreensão mais profunda dos aspectos teóricos, recomendamos a leitura do artigo original.

Além dessas duas variações, também buscamos explorar o caso onde os polígonos não são necessariamente convexos, o que adiciona uma camada extra de complexidade ao problema. Podemos mostrar que nesse caso, mesmo quando não temos "cercas", o problema é NP-difícil **tpp-dror2003**, no entanto, exploramos diferentes estratégias para lidar com essa complexidade adicional, partindo de uma enumeração completa e então removendo diversas soluções inviáveis através de uma estratégia de Branch and Bound, usando os algoritmos para o TPP Irrestrito como cálculo de limites inferiores. Dessa forma, conseguimos resolver instâncias razoavelmente grandes do problema com polígonos não convexos em um tempo aceitável.

Esse caso não foi abordado no artigo original, na verdade, essa variação não parece ter sido estudada na literatura no momento da escrita deste relatório, por esse motivo consideramos que essa é uma contribuição original deste trabalho.

Uma parte central desse trabalho é a implementação prática dos algoritmos propostos, uma vez que isso distingue esse relatório do artigo original, que não inclui detalhes de implementação. Por esse motivo, disponibilizamos todo o código-fonte desenvolvido em um repositório público no GitHub, permitindo que outros pesquisadores e entusiastas possam reproduzir os resultados apresentados aqui, bem como explorar e expandir o trabalho realizado. As implementações dos algoritmos foram realizadas tanto em *Python* quanto em *C++*, enquanto as visualizações utilizaram a biblioteca *matplotlib* para *Python* e a biblioteca *Pygame* para criar um editor gráfico interativo. Além do código em si, também incluímos instruções detalhadas sobre como executar os algoritmos e reproduzir os experimentos apresentados neste relatório.

2 O Problema de Visita de Polígonos Irrestrito

Primeiramente vamos considerar o caso do Problema de Visita de Polígonos Irrestrito (TPP Irrestrito), onde não vamos impor restrições sobre o caminho que visita os polígonos. Esse problema foi originalmente descrito por Dror et al. (2003) **tpp-dror2003**, que propuseram um algoritmo com complexidade $O(nk \log(n/k))$ para resolver o problema, onde n é o número total de vértices dos polígonos e k é o número de polígonos. Buscamos apresentar uma versão prática desse algoritmo, focando no entendimento e implementação. O problema em si pode ser enunciado da seguinte maneira:

> Dados dois pontos $s, t \in \mathbb{R}^2$ e uma sequência de polígonos convexos disjuntos $P_1, P_2, \ldots, P_k$, encontrar o caminho de menor comprimento que se inicia em s, termina em t e toca cada polígono P_i em ordem em pelo menos um ponto, podendo atravessá-los.

A Figura 2 ilustra um exemplo de entrada e a solução ótima para o problema para um caso com 3 polígonos. Temos s como o ponto verde, t como o ponto vermelho e os polígonos P_1, P_2 e P_3 como o triângulo azul, o trapézio laranja e o pentágono verde, respectivamente. O caminho mínimo é representado pela linha roxa.

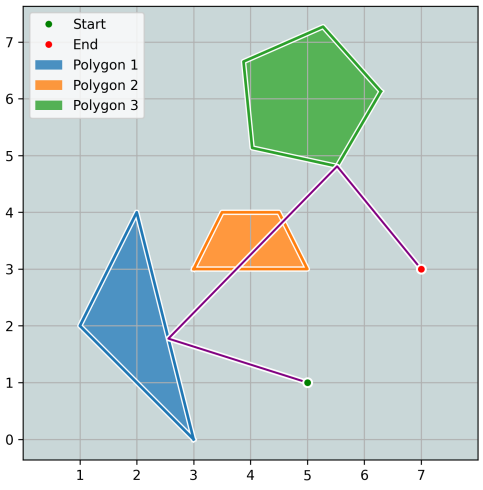


Figura 2: Caminho mínimo para um caso de 3 polígonos.

No contexto de otimização geométrica, muitos problemas similares podem ser resolvidos usando uma técnica geral de computar um *mapa de caminho mínimo*, que particiona o plano em regiões onde o comportamento do caminho mínimo é o mesmo. Para o nosso problema, qualquer solução deve tocar cada polígono P_i pela primeira vez em algum vértice ou aresta, assim, poderíamos agrupar os pontos de acordo com a sequência de vértices e arestas onde o caminho mínimo toca os polígonos. De fato, se calculássemos esse mapa, poderíamos responder consultas em tempo $O(k+\log(n))$ **tpp-dror2003**, no entanto, no pior caso, esse mapa tem complexidade $\Theta((n-k)2^k)$, ou seja, exponencial no número de polígonos k **tpp-dror2003**, tornando essa abordagem inviável.

Por esse motivo, usamos uma nova técnica baseada em subdividir o plano de acordo com *mapas de último passo mínimo*, que para cada polígono P_i, agrupa pontos p tais que o ponto q que antecede p do caminho mínimo partindo de s, tocando $P_1, \ldots, P_i$ e chegando em p é um mesmo vértice ou uma mesma aresta. O conjunto dessas partições pode ser contruído em tempo $O(nk \log(n/k))$, permitindo responder consultas em tempo $O(k \log(n/k))$ **tpp-dror2003**. Dessa forma, podemos encontrar o caminho mínimo que visita todos os polígonos em tempo Polinomial.

2.1 Condições de Otimalidade Local

Primeiramente, definimos um i-path até p como um caminho mínimo que se inicia em s e toca os polígonos $P_1, \ldots, P_i$ em ordem, terminando em um ponto p. Podemos mostrar que tal i-path é **único** e assim otimalidade **local** implica otimalidade **global tpp-dror2003**. Note que um k-path até t é exatamente o caminho mínimo que buscamos.

Claramente, apenas precisamos considerar caminhos formados pela união de segmentos de reta entre s e t. Ademais, tal caminho é *localmente mínimo* se dobras ocorrem apenas no perímetro de cada polígono e, para cada dobra $b \in \partial P_i$, mover b ligeiramente ao longo de qualquer direção na fronteira de P_i enquanto mantemos os outros pontos do caminho fixos aumenta o comprimento total do caminho. Isso implica que para pontos de dobra b no interior de uma aresta de P_i, o ângulo de incidência do segmento que chega em b é igual ao ângulo de reflexão do segmento que sai de b (Lei da Reflexão).

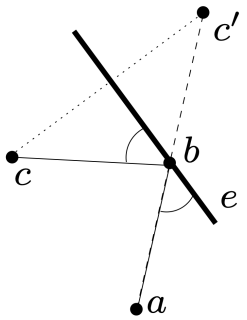


Figura 3: Condições de otimalidade em arestas.

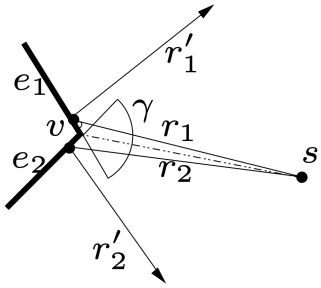


Figura 4: Condições de otimalidade em vértices.

Já para pontos de dobra que são vértices $v = e^1 \cap e^2$ de P_i, as condições de otimalidade dizem que, assumindo que u é o ponto imediatamente antes de v no caminho, um segmento saindo de v deve estar contido no cone γ delimitado pelas reflexões r^1 e r^2 da direção $\overrightarrow{uv}$ em relação às duas arestas e^1 e e^2. Intuitivamente, isso ocorre porque qualquer segmento saindo de v fora desse cone poderia ser substituído por um segmento refletido em uma das arestas, reduzindo o comprimento total do caminho.

A Figura 3 ilustra a condição de otimalidade para dobras em arestas, na qual temos um caminho com os pontos a, b e c, onde b é um ponto de dobra em um parte interior da aresta e. Observamos que o segmento $\overline{ab}$ faz o mesmo ângulo com a aresta e que o segmento $\overline{bc}$, assim, as condições de otimalidade estão satisfeitas. Já a Figura 4 ilustra a condição de otimalidade para dobras em vértices, na qual temos um caminho partindo de s que chega em um vértice v de um polígono. Observamos que segmentos que saem de v devem estar dentro do cone γ delimitado pela reflexão do segmento $\overline{sv}$ em relação às duas arestas e^1 e e^2 que se encontram em v.

Adicionalmente, vale mencionar que se um caminho chega em um ponto p atravessando o interior de um polígono P_i, então o último segmento desse caminho é localmente mínimo, pois a menor distância entre dois pontos é uma linha reta.

2.2 Região de Primeiro Contato

Para cada $i \in \{1, \ldots, k\}$, definimos a região de primeiro contato T_i de P_i como o conjunto de pontos $p \in \partial P_i$ tais que o $(i-1)$-path até p toca P_i pela primeira vez em p após visitar os polígonos $P_1, \ldots, P_{i-1}$. Podemos mostrar que essa região é contínua e é delimitada por vértices de P_i **tpp-dror2003**.

Sabemos que i-paths localmente mínimos só podem sair de T_i de maneiras específicas, continuando em linha reta ou refletindo de maneira apropriada, adicionalmente, como otimalidade local implica otimalidade global, se existe um caminho localmente ótimo até p, então esse é o caminho mínimo até p.

Assim, seja $p \in \mathbb{R}^2$, v um vértice de P_i tal que $v \in T_i$ e u o ponto que precede v no $(i-1)$-path até v. Pelas condições de otimalidade local, sabemos que o segmento $\overline{vp}$ só pode ser parte de um i-path até p se o o segmento $\overline{vp}$ está contido dentro do cone γ formado pela reflexão do vetor $\overrightarrow{uv}$ em relação às duas retas que contém cada uma das duas arestas que se encontram em v. Como as condições de otimalidade são tanto necessárias quanto suficientes, podemos afirmar que se p pertence ao cone γ, então o caminho formado pelo $(i-1)$-path até v seguido do segmento $\overline{vp}$ é o i-path até p.

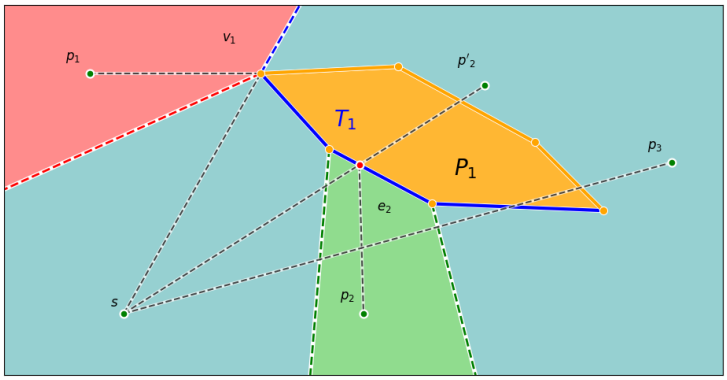


Figura 5: Caminhos mínimos visitando um polígono.

Ademais, seja $p \in \mathbb{R}^2$ e vamos assumir que o i-path até p tem uma dobra b na parte interior de uma aresta e de P_i limitada pelos vértices v^1 e v^2 tais que $v^1, v^2 \in T_i$. Seja u o ponto

que precede b no $(i-1)$-path até b. Pelas condições de otimalidade local, sabemos que o segmento $\overline{bp}$ só pode ser parte de um i-path até p se o ângulo de incidência do segmento $\overline{ub}$ com a aresta e for igual ao ângulo de reflexão do segmento $\overline{bp}$ com a aresta e. Isso é possível se e somente se p está entre as reflexões dos raios que chegam em v^1 e v^2 em relação à aresta e e se p está do lado "externo"da aresta e (ou seja, o segmento $\overline{bp}$ não atravessa o interior de P_i). Como as condições de otimalidade são tanto necessárias quanto suficientes, podemos afirmar que se p satisfaz essas condições, então o caminho formado pelo $(i-1)$-path até b seguido do segmento $\overline{bp}$ é o i-path até p.

Finalmente, se p não satisfaz nenhuma das condições anteriores, então o i-path até p deve atravessar o interior de P_i para chegar em p. Nesse caso, o i-path até p é exatamente o $(i-1)$-path p, pois esse caminho automaticamente visita P_i ao atravessá-lo. A Figura 5.

A Figura 5 ilustra essas três possibilidades um caso de um único polígono. Temos que o ponto p^1 está no cone do vértice v^1, portanto, o 1-path até p^1 parte de s, chega em v^1 e segue em linha reta até p^1. Já o ponto p^2 está entre as reflexões dos raios que chegam em v^2 e v^3 em relação à aresta e^2, assim, o 1-path até p^2 parte de s, chega em um ponto b na aresta e^2 e reflete até p^2. Finalmente, o ponto p^3 não satisfaz nenhuma das condições anteriores, portanto, o 1-path até p^3 parte de s e chega diretamente em p^3, atravessando o interior do polígono.

2.3 O Mapa de Último Passo Mínimo

As ideias das duas últimas seções sugerem uma maneira de particionar o plano de acordo com o último passo do i-path até cada ponto p. Para cada polígono P_i, podemos particionar o plano em regiões onde o último passo do i-path até qualquer ponto p dentro dessa região é um mesmo vértice de P_i, uma mesma aresta de P_i ou atravessa o interior de P_i. Denotamos essa partição por S_i e chamamos de **regiões de vértice**, **regiões de aresta** e **região de travessia** as regiões onde o último passo do i-path é um vértice, uma aresta ou atravessa o interior de P_i, respectivamente.

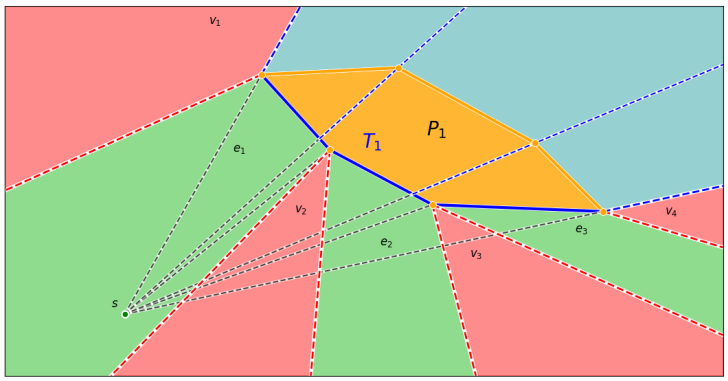


Figura 6: Mapa de último passo mínimo para um polígono.

Ilustramos essa ideia na Figura 6, onde temos o polígono P_1 em laranja, as regiões de vértice em vermelho, aresta em verde e travessia em ciano, também colorimos T_1 em azul. Note que para cada vértice $v \in T_1$ de P_1, temos que sua região de vértice é o cone formado pelas reflexões do vetor $\overrightarrow{sv}$ em relação às duas arestas que se encontram em v. Já para cada aresta $e \subseteq T_1$ de P_1, temos que sua região de aresta é o conjunto de pontos entre as reflexões dos raios que chegam nos dois vértices que limitam e em relação à aresta e e do lado externo de e. Finalmente, a região de travessia é o restante do plano.

Para construir S_i, começamos determinando T_i, detalhamos mais a frente como fazer isso de maneira eficiente. Em seguida, para cada vértice $v \in T_i$ de P_i, construímos sua região de vértice calculando o ponto u que precede v no $(i-1)$-path até v e então calculando as reflexões do vetor $\overrightarrow{uv}$ em relação às duas arestas que se encontram em v. Uma vez que temos as regiões de vértice, não é necessário construir as regiões de aresta explicitamente, pois elas podem ser derivadas das regiões de vértice adjacentes e da aresta correspondente. Finalmente, a região de travessia é simplesmente o complemento das regiões de vértice e aresta.

Uma vez que definimos a partição S_i do plano, conseguimos determinar o i-path até qualquer ponto p de maneira eficiente. Basta determinar em qual região de S_i o ponto p está e então aplicar as condições de otimalidade local para construir o i-path até p de acordo com o último passo do caminho. Ademais, vamos expor adiante como localizar um ponto p em S_i em tempo $O(\log(n))$, permitindo responder consultas de i-paths em tempo $O(i\log(n))$.

A partir dessas ideias, note que primeiro contruímos a partição S_1 usando apenas o ponto s. Em seguida, podemos usar s e S_1 para construir S_2, depois usar s, S_1 e S_2 para construir S_3 e assim por diante, até construirmos S_k. Finalmente, podemos usar s, $S_1, S_2, \ldots, S_k$ para determinar o k-path até o ponto t, que é exatamente o caminho mínimo que buscamos. A Figura 7 ilustra as partições S_1, S_2 e S_3 para o exemplo da Figura 2.

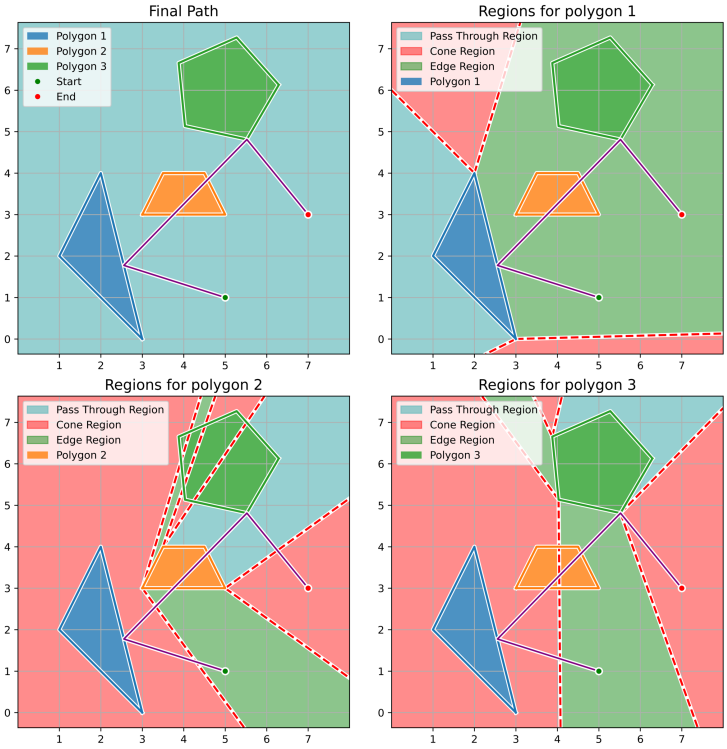


Figura 7: Partição do plano para cada polígono do exemplo anterior.

Dessa forma, concluímos o embasamento teórico necessário para implementar o algoritmo proposto por Dror et al. (2003) **tpp-dror2003** para resolver o TPP Irrestrito. Nas próximas seções, vamos discutir em detalhe como implementar cada parte do algoritmo e as subrotinas necessárias para construir as partições S_i de maneira eficiente.

2.4 Determinando a Região de Primeiro Contato

Como discutido acima, a região de primeiro contato T_i de um polígono P_i é o conjunto de pontos $p \in \partial P_i$ tais que o $(i-1)$-path até p toca P_i pela primeira vez em p. Adicionalmente, sabemoos que T_i é contínua, delimitada por vértices de P_i e sempre contém pelo menos uma aresta de P_i e nunca contém o polígono inteiro. Tentar provar isso...

É importante ressaltar que apenas determinar quais vértices de P_i pertencem a T_i não é suficiente, pois T_i pode conter todos os vértices de P_i e ainda assim ter uma aresta que não pertence a T_i. Por esse motivo, buscamos determinar o primeiro e o último vértice de P_i que pertencem a T_i e então considerar apenas as arestas entre esses dois vértices.

Primeiramente, sejam b, v, a vértices consecutivos de P_i em sentido anti-horário (b para *before* e a para *after*) e seja u o ponto que precede v no $(i-1)$-path até v. Usando as condições de otimalidade de primeira ordem, podems concluir que se as direções $\overrightarrow{bv}$ e $\overrightarrow{va}$ estão em lados opostos do vetor $\overrightarrow{uv}$, então v é um dos extremos de T_i, mais especificamente, se $\overrightarrow{bv}$ está no lado anti-horário de $\overrightarrow{uv}$, então v é o primeiro vértice de T_i, caso contrário, é o último vértice de T_i.

Enquanto a maioria dessas operações são simples verificações geométricas baseadas em produtos vetoriais, precisamos de uma maneira eficiente de determinar o ponto u que precede v no $(i-1)$-path até v. No momento, só sabemos responder essa consulta para um 0-path, que é simplesmente o ponto s. No entanto, como discutimos acima, isso é suficiente para construir T_1 e S_1, os quais nos permitem responder consultas de 1-paths. Assim, podemos usar s, T_1 e S_1 para construir T_2 e S_2, os quais nos permitem responder consultas de 2-paths. Repetindo esse processo, conseguimos construir T_i e S_i para qualquer $i \in \{1, \ldots, k\}$.

Por esse motivo, vamos criar uma função Consulta$(p, s, (P_1, \ldots, P_i), (T_1, \ldots, T_i), (S_1, \ldots, S_i))$ que dado um ponto p, o ponto inicial s, os polígonos $P_1, \ldots, P_i$, as regiões de primeiro contato $T_1, \ldots, T_i$ e as partições $S_1, \ldots, S_i$, retorna o i-path até p. Usando essa função, podemos determinar o ponto u que precede um vértice v no $(i-1)$-path até v simplesmente chamando Consulta$(v, s, P_1, \ldots, P_{i-1}, T_1, \ldots, T_{i-1}, S_1, \ldots, S_{i-1})$ e então retornando o penúltimo ponto do caminho retornado. Vamos detalhar a implementação dessa função mais adiante, mas por enquanto, assumimos que essa função está disponível e funciona corretamente.

Utilizando essa função e as ideias acima, implementamos o Algoritmo 1 para determinar a região de primeiro contato T_i, que retorna uma lista de vértices consecutivos de P_i que pertencem a T_i e tais que o primero e o último vértice dessa lista são exatamente os extremos de T_i.

Algoritmo 1: Região de Primeiro Contato

Entrada: s, $(P_1, \ldots, P_i)$, $(T_1, \ldots, T_{i-1})$, $(S_1, \ldots, S_{i-1})$
Saída : T_i na forma de uma lista de vértices consecutivos de P_i.

$i \leftarrow -1$ // Índice do primeiro vértice de T_i
$j \leftarrow -1$ // Índice do último vértice de T_i
for v vértice de P_i **:**
  $b \leftarrow$ vértice antes de v em P_i
  $a \leftarrow$ vértice depois de v em P_i
  $path \leftarrow \text{Consulta}(v, s, (P_1, \ldots, P_{i-1}), (T_1, \ldots, T_{i-1}), (S_1, \ldots, S_{i-1}))$
  $u \leftarrow$ penúltimo ponto de $path$
  $d^1 \leftarrow v - b$
  $d^2 \leftarrow a - v$
  $d^u \leftarrow u - v$
  if $d^u \times d^1 \geq 0 \wedge d^u \times d^2 \leq 0$ **:**
    $i \leftarrow$ índice de v em P_i
  if $d^u \times d^1 \leq 0 \wedge d^u \times d^2 \geq 0$ **:**
    $j \leftarrow$ índice de v em P_i
return vértices de P_i entre os índices i e j (inclusive)

2.5 Construindo as Partições

Uma vez que determinamos T_i, o próximo passo é contruir a partição S_i do plano. Conforme discutido acima, para representar S_i, precisamos apenas contruir as regiões de vértice, pois as regiões de aresta podem ser derivadas das regiões de vértice adjacentes e a região de travessia é simplesmente o complemento das regiões de vértice e aresta.

Pelas condições de otimalidade local, sabemos que um ponto p pertence à região de vértice associada a um vértice v de P_i se $v \in T_i$ e se o segmento $\overline{vp}$ está contido no cone formado pelas reflexões do vetor $\overrightarrow{uv}$ em relação às retas contendo as cada uma das duas arestas que se encontram em v, onde u é o ponto que precede v no $(i-1)$-path até v. Assim, para construir as regiões de vértice, precisamos apenas calcular essas reflexões para cada vértice $v \in T_i$. Também devemos notar que uma das arestas incidentes a v pode não estar na região de primeiro contato, nesse caso, seria impossível refletir o vetor em relação a reta que contem essa aresta, assim, consideramos o raio seguindo a direção original de $\overrightarrow{uv}$.

Para refletir um vetor d em relação a uma aresta (v^1, v^2), podemos usar o Algoritmo 2, que calcula a direção refletida de d em relação à reta que contém a aresta (v^1, v^2).

Juntado as ideias acima, implementamos o Algoritmo 3 para construir as regiões de vértice S_i de P_i, que retorna S_i como uma sequência de pares ordenados de direções associadas aos vértices de T_i.

Algoritmo 2: RefleteDireção

Entrada: d, v^1, v^2
Saída : Direção refletida de d em relação à reta que contém a aresta (v^1, v^2).

$\omega \leftarrow v^2 - v^1$
$\omega \leftarrow \omega / ||\omega||$ // Normalizar ω
$d_\perp \leftarrow (-\omega_2, \omega_1)$ // Vetor perpendicular a ω
$proj_\perp \leftarrow \langle d, d_\perp \rangle$
return $d - 2\langle proj_\perp, d_\perp \rangle$ // Refletir d em relação a reta

Algoritmo 3: Região de Vértice

Entrada: s, $(P_1, \ldots, P_i)$, $(T_1, \ldots, T_i)$, $(S_1, \ldots, S_{i-1})$
Saída : S_i como uma sequência de pares ordenados de direções associadas aos vértices de T_i.

$S_i \leftarrow \emptyset$
for v vértice de T_i **:**
  path $\leftarrow$ Consulta$(v, s, (P_1, \ldots, P_{i-1}), (T_1, \ldots, T_{i-1}), (S_1, \ldots, S_{i-1}))$
  $u \leftarrow$ penúltimo ponto de path
  $d \leftarrow v - u$
  $v^1 \leftarrow$ vértice antes de v em P_i
  $v^2 \leftarrow$ vértice depois de v em P_i
  if $(v^1, v) \in T_i$ **:**
    $d^1 \leftarrow$ RefleteDireção(d, v^1, v)
  else
    $d^1 \leftarrow d$
  if $(v, v^2) \in T_i$ **:**
    $d^2 \leftarrow$ RefleteDireção(d, v, v^2)
  else
    $d^2 \leftarrow d$
  // Adicionar região de vértice associada a v em S_i
  $S_i \leftarrow S_i \oplus (d^1, d^2)$
return S_i

2.6 Localizando Pontos na Partição

Uma vez que construímos T_i e S_i, o próximo passo é implementar a função Consulta que retorna o i-path até um ponto p. No entanto, isso requer localizar o ponto p na partição S_i, ou seja, determinar qual vértice, aresta ou região de travessia contém o ponto p. Primeiramente, vamos descrever uma solução em tempo $O(|S_i|)$, que itera sobre todas as regiões de vértice e aresta de S_i e verifica se o ponto p pertence a alguma dessas regiões. Em seguida, vamos descrever como otimizar essa solução para tempo $O(\log(|P_i|))$ usando uma estratéfia apropriada baseada em busca binária.

Para ambas soluções, precisamos conseguir verificar se um ponto p pertence a uma região de vértice ou a uma região de aresta específicos. Para uma região de vértice associada a um vértice v de T_i, precisamos apenas verificar se o segmento $\overline{vp}$ está contido no cone formado pelos vetores d^1 e d^2, calculados no Algoritmo 3. Para tal, devemos considerar dois casos, dependendo do ângulo entre d^1 e d^2. Se o ângulo for menor ou igual a 180°, então p pertence à região de vértice se e somente se p está à esquerda de d^1 **e** à direita de d^2. Caso contrário, p pertence à região de vértice se e somente se p está à esquerda de d^1 **ou** à direita de d^2, as Figuras 8 e 9 ilustram esses dois casos. Usando essas ideias, implementamos o procedimento **??** para verificar se um ponto p pertence a uma região de vértice específica.

Algoritmo 4: PontoEmVértice

Entrada: p, v, d^1, d^2
Saída : `true` se p pertence à região de vértice associada a v delimitada por d^1 e d^2, `falso` caso contrário.

if $d^1 \times d^2 \geq 0$ **:**
  return $d^1 \times (p - v) \geq 0 \wedge d^2 \times (p - v) \leq 0$
else
  return $d^1 \times (p - v) \geq 0 \vee d^2 \times (p - v) \leq 0$

Ademais, também precisamos verificar se um ponto p pertence a uma região de aresta associada a uma aresta (u, v) de T_i com direções d^1 partindo de u e d^2 partindo de v. A princípio, temos apenas um único caso, pois pela maneira que determinamos as direções d^1 e d^2, é impossível que $d^1 \times d^2 < 0$, dessa forma, temos um único caso para considerar, no qual, p pertence à região de aresta se e somente se p está entre as semi-retas definidas por d^1 e d^2 **e** p está do lado externo da aresta (u, v). Essencialmente, estamos lidando

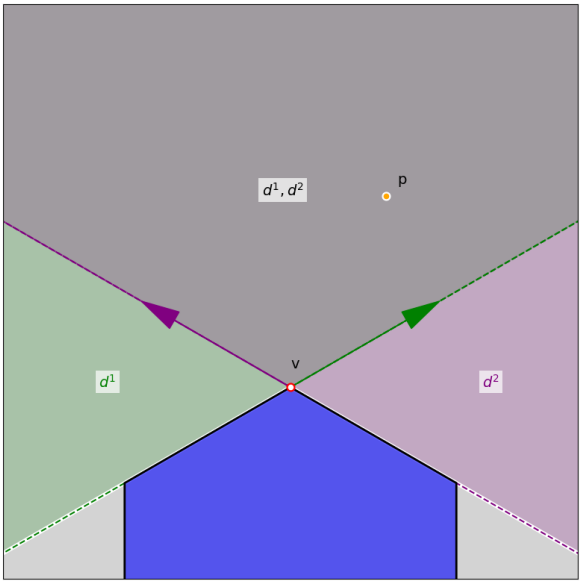


Figura 8: Região de vértice onde $d^1 \times d^2 > 0$.

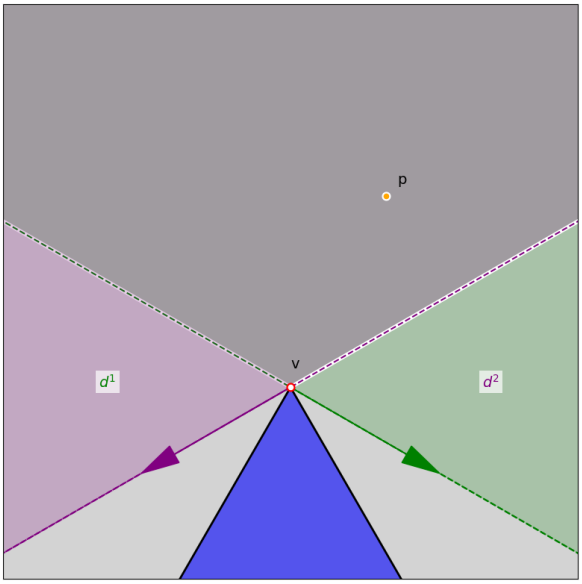


Figura 9: Região de vértice onde $d^1 \times d^2 < 0$.

com o caso da Figura 8 com a restrição adicional de que o ponto deve estar do lado correto da aresta. Usando essas ideias, implementamos o Algoritmo 5 para verificar se um ponto p pertence a uma região de aresta específica.

Algoritmo 5: PontoEmAresta

Entrada: p, u, v, d^1, d^2
Saída : `true` se p pertence à região de aresta associada a (u, v) delimitada por d^1 e d^2, `falso` caso contrário.

1 **return** $d^1 \times (p - u) \geq 0 \wedge d^2 \times (p - v) \leq 0 \wedge ((v - u) \times (p - u) \leq 0)$

Finalmente, podemos usar os Algoritmos 4 e 5 para localizar um ponto p na partição S_i iterando sobre todas as regiões de vértice e aresta de S_i e verificando se p pertence a alguma dessas regiões. Se p pertencer a uma região de vértice associada a um vértice v, retornamos esse vértice. Se p pertencer a uma região de aresta associada a uma aresta (u, v), retornamos essa aresta. Caso contrário, retornamos que p pertence à região de travessia. Implementamos esse procedimento no Algoritmo 6. Vale notar que não detalhamos o que é retornado exatamente, mas isso pode ser facilmente implementado retornando o índice do vértice ou da aresta na estrutura de dados que representa S_i.

2.7 Localizando Pontos na Partição Eficientemente

Na seção anterior, descrevemos uma maneira de localizar um ponto p na partição S_i em tempo $O(|S_i|)$ iterando sobre todas as regiões de vértice e aresta de S_i. No entanto, podemos otimizar essa solução para tempo $O(\log(|P_i|))$ usando uma estratégia baseada em busca binária. Para tal, não precisamos de nenhuma estrutura de dados adicional, apenas perceber que as regiões de vértice e aresta de S_i podem ser unidas em uma única região de aresta, permitindo que verifiquemos se p pertence à essa nova região em tempo $O(1)$, permitindo assim o uso de busca binária.

Essencialmente, sejam $v^1, \ldots, v^n$ vértices consecutivos de T_i em sentido anti-horário. Para determinar se um ponto p pertence à **alguma** região de vértice desses vértices ou à alguma região de aresta entre esses vértices, poderíamos verificar cada região de vértice e aresta individualmente, como fizemos na seção anterior. No entanto, também podemos considerar a região de aresta definida pela semi-reta partindo de v^1 até v^n, com raios d^1 e d^2 onde d^1 é o primeiro raio de v^1 e d^2 é o segundo raio de v^n. Note que essa região de aresta contém todas as regiões de vértice e aresta entre v^1 e v^n e não contém nenhuma outra região de vértice ou aresta. Assim, podemos verificar se p pertence a alguma região de vértice ou aresta entre v^1 e v^n simplesmente verificando se p pertence a essa nova região de aresta. Usando essa ideia, podemos implementar uma busca binária para localizar p na partição S_i.

Algoritmo 6: LocalizaPonto

Entrada: p, T_i, S_i
Saída : Região de S_i que contém p (vértice, aresta ou travessia).

```
for v vértice de T_i :
    (d^1, d^2) ← direções associadas a v em S_i
    if PontoEmVértice(p, v, d^1, d^2) :
        return v
for (u, v) aresta de T_i :
    d^1 ← segunda direção associada a u em S_i
    d^2 ← primeira direção associada a v em S_i
    if PontoEmAresta(p, u, v, d^1, d^2) :
        return (u, v)
return Região de travessia
```

A Figura **??** ilustra essa ideia, onde temos quatro regiões de vértice e três regiões de aresta unidas em uma única região de aresta maior. Note que essa nova região de aresta pode ser verificada usando o Algoritmo 5, permitindo assim a implementação da busca binária.

Infelizmente, o Algoritmo 5 não é suficiente para verificar se um ponto p pertence a uma região de aresta arbitrária, como a que acabamos de definir, pois a união de regiões de vértice e aresta pode gerar diferentes casos para a região de aresta resultante, dependendo do ângulo entre d^1 e d^2. Por esse motivo, precisamos primeiro estender o Algoritmo 5 para lidar com esses casos adicionais antes de podermos implementar a busca binária.

Essencialmente existem 4 casos possíveis para uma região de aresta, dependendo do ângulo entre as direções d^1 e d^2. A Figura 11 ilustra esses casos. Para sabermos com qual caso estamos lidando, definimos $w = v - u$ e calculamos os produtos vetoriais $c_1 = w \times d^1$ e $c_2 = w \times d^2$. Agora, temos os seguintes casos:

- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **ambas arestas no lado negativo**, que é o caso tradicional que aparece na partição S_i. Nesse caso, verificamos se p está entre as semi-retas e a aresta (u, v) usando produtos vetoriais.

- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **aresta anterior no lado negativo e aresta posterior no lado positivo**, o *Caso 1* na figura. Nesse caso, dividimos a região de aresta em duas regiões de vértice, definidas por $(u, d^1, u - v)$ e $(v, u - v, d^2)$, então verificamos se o ponto está em qualquer uma das duas.

- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **aresta anterior no lado positivo e aresta posterior no lado negativo**, o *Caso 2* na figura. Nesse caso, dividimos a região de aresta em duas regiões de vértice, definidas por $(u, d^1, v - u)$ e $(v, v - u, d^2)$, então verificamos se o ponto está em qualquer uma das duas.

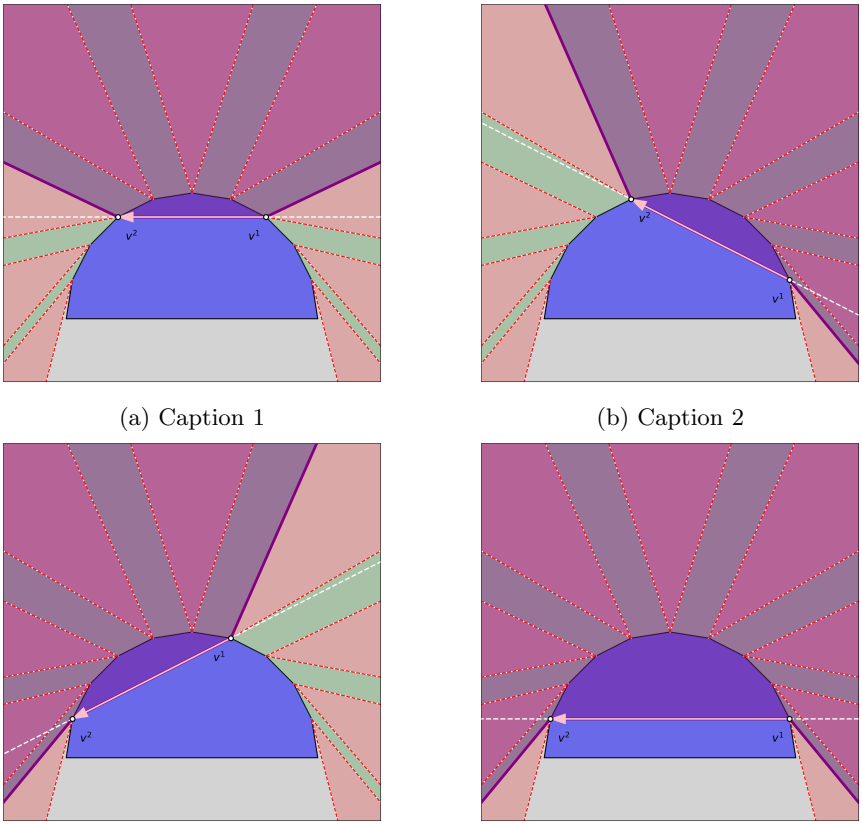
- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **ambas arestas no lado positivo**. Nesse caso, simplesmente verificamos se o ponto não pertence ao complemento da região, ou seja, verificamos se p não pertence à região de aresta definida por (v, u, d^2, d^1).

Dessa forma, conseguimos verificar se um ponto pertence a uma região de aresta usando o procedimento de verificação de regiões de vértice. Agora podemos implementar o procedimento completo no algoritmo 7.

2.7.1 Localização Eficiente de Pontos na Partição

Agora que sabemos como verificar se um ponto pertence a uma região de vértice ou de aresta, podemos usar essas funcionalidades para localizar um ponto p na partição S_i. Uma abordagem ingênua seria iterar sobre todas as regiões de vértice e aresta, verificando se p pertence a alguma delas. Isso funcionaria, no entanto, essa abordagem tomaria tempo $O(|P_i|)$, o que é muito lento.

Por esse motivo, desejamos explorar a ideia de busca binária para resolver o problema. Primeiramente, vamos assumir que o ponto não pertence ao polígono P_i, isso vale como suposição inicial dos polígonos serem disjuntos e os pontos s e t estarem fora dos polígonos. Ainda assim, se desejássemos evitar essa suposição,



(a) Caption 1

(b) Caption 2

(c) Caption 3

(d) Caption 4

Figura 10: Overall caption for the 2×2 grid.

poderíamos simplesmente verificar se o ponto está dentro do polígono P_i usando um teste de ponto em polígono antes de localizar o ponto na partição, o é um problema clássico que leva tempo $O(\log(|P_i|))$.

Uma questão que diferencia nosso problema de uma busca binária tradicional é que as regiões são circulares. Assim, o primeiro passo é verificar se o ponto pertence à região entre o último e o primeiro vértice de P_i. Se sim, retornamos essa região. Caso contrário, reduzimos o problema para uma lista linear de regiões.

Outra diferença que precisamos considerar é que nossa entrada contém dois tipos de regiões: regiões de vértice e regiões de aresta. Para lidar com isso, podemos tratar regiões de vértice como regiões de aresta degeneradas, onde a aresta tem comprimento zero. Dessa forma, podemos aplicar o mesmo raciocínio para ambos os tipos de regiões, lidando com uma lista de $2|P_i| - 1$ regiões de aresta, uma vez que já eliminamos a última. Note que a implementação de PontoEmAresta já lida com esse caso especial.

Finalmente, podemos implementar a busca binária propriamente dita. A ideia é manter dois índices l e r que representam o intervalo atual de regiões que estamos considerando. Inicialmente, definimos $n = |P_i|$, $l = 0$ e $r = 2n - 1$. Nos baseamos na ideia de que apenas as regiões após l e antes de r podem conter o ponto p, ambos limites inclusivos.

Enquanto $l + 1 \neq r$, calculamos o índice médio $m = \lfloor (l + r)/2 \rfloor$ e verificamos se o ponto p pertence à região de aresta entre l e m. Se estiver, atualizamos $r = m$, caso contrário, atualizamos $l = m$, isso vale pois P_i é convexo e a região de aresta entre l e m cobre todas as regiões entre l e m e nenhuma região entre m e r.

Repetindo isso até que $l + 1 = r$, sabemos que o ponto p pertence à região de aresta entre l e r. Finalmente, retornamos essa região, vamos codificar essa região como um inteiro que representa o índice da região.

Essa codificação segue a seguinte lógica: regiões de vértice são representadas por índices pares, onde o índice $2j$ representa a região de vértice associada ao j-ésimo vértice de P_i. Regiões de aresta são representadas por índices ímpares, onde o índice $2j + 1$ representa a região de aresta entre o j-ésimo e o $(j + 1)$-ésimo vértice de P_i. Note que a região entre o último e o primeiro vértice é representada pelo índice $2|P_i| - 1$.

Implementamos essa ideia no algoritmo 8, que recebe um ponto p e um polígono P_i e retorna o índice

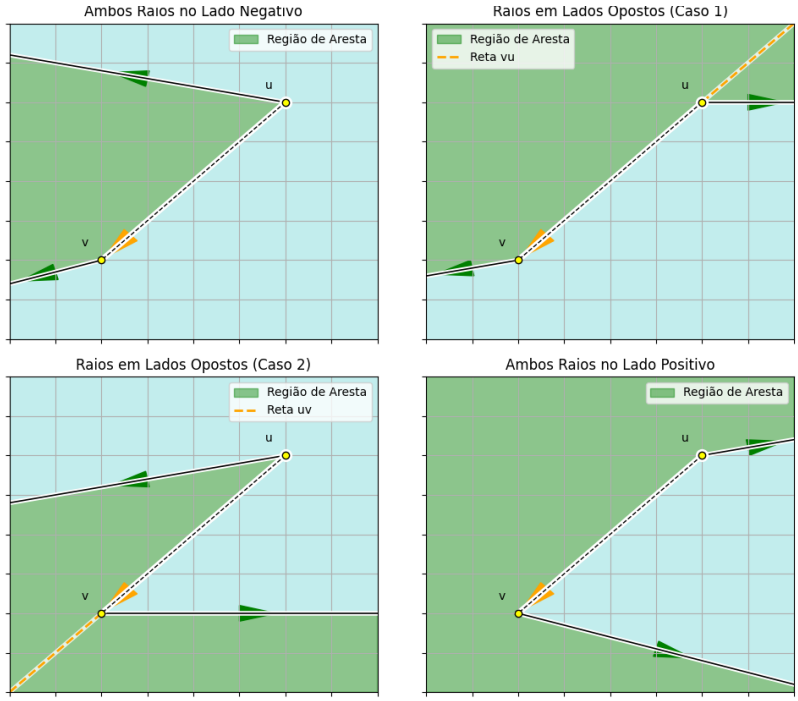


Figura 11: Casos possíveis para região de aresta.

Algoritmo 7: PontoEmAresta(p, u, v, d^1, d^2)
Verifica se o ponto p está na região de aresta associada à aresta $e = (u, v)$ delimitada pelas direções d^1 e d^2

// Se vértices são muito próximos, tratamos a região de aresta
// como uma região de vértice.
if $|u - v| < 10^{-8}$ **:**
    return PontoEmVértice(p, v, d^2, d^1)
$w \leftarrow v - u$
$c_1 \leftarrow d^1 \times w$
$c_2 \leftarrow w \times d^2$
if $c_1 \leq 0 \wedge c_2 \leq 0$ **:**
    return $(d^1 \times (p - u) \geq 0) \wedge ((p - u) \times d^2 \leq 0) \wedge ((v - u) \times (p - u) \leq 0)$
else if $c_1 \leq 0 \wedge c_2 > 0$ **:**
    return PontoEmVértice(p, u, d^1, $u - v$) $\vee$ PontoEmVértice(p, v, $u - v$, d^2)
else if $c_1 > 0 \wedge c_2 \leq 0$ **:**
    return PontoEmVértice(p, u, d^1, $v - u$) $\vee$ PontoEmVértice(p, v, $v - u$, d^2)
else
    return $\neg$ PontoEmAresta(p, v, u, d^2, d^1)

da região de S_i que contém p.

Algoritmo 8: LocalizaRegião(p, P_i)
Localiza o ponto p na partição do polígono P_i

```
1  def getDireção(i):
2      if i (mod 2) = 0 :
3          return direção anterior associada ao vértice ⌊i/2⌋ de P_i
4      else
5          return direção anterior associada ao vértice ⌊i/2⌋ de P_i
6  def pertence(i, j):
7      v_1 ← Vértice ⌊i/2⌋ do polígono P_i
8      v_2 ← Vértice ⌊j/2⌋ do polígono P_i
9      d^1 ← getDireção(i)
10     d^2 ← getDireção(j)
11     return PontoEmAresta(p, v_1, v_2, d^1, d^2)
12 l ← 0
13 r ← 2|P| − 1
14 if pertence(r, 0) :
15     return r // Ponto pertence à região entre o último e o primeiro vértice
16 while l + 1 ≠ r do
17     m ← ⌊(l + r)/2⌋
18     if pertence(l, m) :
19         r ← m
20     else
21         l ← m
22 return l // Ponto pertence à região entre l e r
```

2.8 Respondendo Consultas

Finalmente, podemos descrever como responder consultas do tipo Query(p, i) usando as partições S_i que construímos anteriormente. O procedimento utilizado é recursivo, assim, primeiro definimos o caso base, que é Query$(p, 0) = s$, uma vez que um 0-path não precisa tocar em nenhum polígono, assim, o menor caminho é uma linha reta de s até p.

Para $i > 0$, primeiramente determinamos a região R de S_i que contém p. Uma vez que sabemos qual região contém p, podemos responder a consulta dependendo do tipo de região:

- Região de Vértice: Seja $R = (v, d^1, d^2)$. Nesse caso, o i-path até p deve passar pelo vértice v, tocando o polígono P_i. Assim, temos que Query$(p, i) = v$.
- Região de Aresta: Seja $R = (u, v, d^1, d^2)$. Nesse caso, o i-path até p deve passar por algum ponto q da aresta $e = (u, v)$, tocando o polígono P_i. Para calcular q, primeiro determinamos $q' =$ Query$(p', i-1)$, onde p' é a reflexão de p em relação à aresta e. Agora, dizemos que q é a interseção entre $\overline{q'p'}$ e a aresta e. Finalmente, respondemos Query$(p, i) = q$.
- Região de Atravessa: Seja R a região de atravessa. Nesse caso, o i-path até p automaticamente atravessa o polígono P_i em algum ponto. Portanto, podemos simplesmente responder Query$(p, i) =$ Query$(p, i-1)$.

Para determinar o ponto de interseção entre o segmento $\overline{q'p'}$ e a aresta e, implementamos o procedimento 9, que calcula a interseção entre duas retas dadas por dois pontos e suas direções. A demonstração desse procedimento é simples e pode ser feita algebricamente, mas não é o foco desse relatório. Também estamos assumindo que as retas não são paralelas, o que é garantido pela construção do algoritmo.

Agora podemos usar essas ideias para implementar o procedimento completo de resposta a consultas na forma do algoritmo 10.

Algoritmo 9: InterseçãoRetas(s, d, s', d')
Calcula a interseção entre as retas definidas por pontos $s + td$ e $s' + t'd'$, onde $t, t' \in \mathbb{R}$

```
1 Δs ← s − s'
2 r ← (Δs × d')/(d × d')
3 return s + rd // Ponto de interseção
```

Algoritmo 10: Query(p, i)
Responde a consulta Query(p, i)

```
1 if i = 0 :
2     return s // Caso base
3 R ← LocalizaRegião(p, P_i)
4 j ← ⌊R/2⌋
5
6 // Região de vértice
7 if R (mod 2) = 0 :
8     return Vértice j do polígono P_i[j]
9
10 // Região de atravessa
11 if Aresta j de P_i não pertence à T_i :
12     return Query(p, i − 1) // Região de atravessa
13
14 // Região de aresta
15 u ← Vértice j de P_i
16 v ← Vértice que segue u em P_i
17 p' ← u + RefleteDireção((p − u), u, v)
18 q' ← Query(p', i − 1)
19 m ← InterseçãoDeRetas(p', q' − p', u, v − u)
20 return m // Ponto de interseção
```

2.9 Calculando o Caminho Mínimo

Agora que sabemos como responder consultas do tipo Query(p, i), calcular o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$ é trivial.

Pela definição de Query, sabemos que o menor caminho que parte de s, toca todos os polígonos $P_1, \ldots, P_k$ e termina em t deve ter como penúltimo ponto $q_k = \text{Query}(t, k)$. Ademais, o caminho que toca os polígonos $P_1, \ldots, P_{k-1}$ e termina em q_k deve ter como penúltimo ponto $q_{k-1} = \text{Query}(q_k, k-1)$. Repetindo esse raciocínio, chegamos até o ponto inicial s.

Usando essa lógica e juntando as implementações anteriores no algoritmo 11, podemos calcular o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$.

Algoritmo 11: CaminhoMínimo$(s, t, P_1, \ldots, P_k)$
Calcula o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$

```
1  if k = 0 :
2      return [s, t] // Caso base: caminho direto de s até t
3  for i ← 1 até k :
4      T_i ← ArestasVisíveis(P_i)
5      S_i ← ConstróiRegiõesDeVértice(P_i)
6  caminho ← [] // Lista vazia para armazenar o caminho
7  p ← t // Começamos do ponto final t
8  for i ← k até 1 :
9      q ← Query(p, k) // Penúltimo ponto do k-path até p
10     // Verificamos se os pontos p e q são distintos devido à regiões de atravessar
11     if |q − p| > 10^−8 :
12         caminho.append(p) // Adiciona p ao caminho
13     p ← q // Atualiza p para o próximo ponto
14     k ← k − 1 // Decrementa k
15 caminho.append(s) // Adiciona o ponto inicial s
16 caminho.reverse() // Inverte a lista para obter o caminho correto
17 return caminho
```

2.10 Análise de Complexidade

Nessa seção, vamos analisar a complexidade do algoritmo completo. Primeiramente, a função LocalizaRegião leva tempo $O(\log(|P_i|))$ para localizar um ponto na partição S_i, uma vez que usa busca binária. Ademais, a função Query faz uma chamada recursiva para $i-1$ em cada nível de recursão, resultando em i níveis de recursão. Assim, o tempo total para responder uma consulta Query(p, i) é $O(i \log(|P_i|))$.

- RefleteDireção(...): $O(1)$, pois faz um número constante de operações.
- PontoEmVértice(...): $O(1)$, pois faz um número constante de operações.
- PontoEmAresta(...): $O(1)$, pois faz um número constante de operações e chamadas para PontoEmVértice.
- LocalizaRegião(P_i, ...): $O(\log|P_i|)$, pois usa busca binária em $2|P_i| - 1$ regiões.
- Query$(i, \ldots)$: Fazemos no máximo i chamadas recursivas, cada uma chama LocalizaRegião(P_i, ...) uma vez. Assim, temos a complexidade:

$$O\left(\sum_{j=1}^{i} \log|P_j|\right)$$

- ArestasVisíveis(P_i, ...): Chamamos Query para os $|P_i|$ vértices de P_i, assim, temos uma complexidade final de:

$$O\left(|P_i| \sum_{j=1}^{i} \log|P_j|\right)$$

- ConstróiRegiõesDeVértice(P_i, ...): Chamamos Query para os $|P_i|$ vértices de P_i, assim, temos uma complexidade final de:

$$O\left(|P_i| \sum_{j=1}^{i} \log|P_j|\right)$$

- CaminhoMínimo: Chamamos ArestasVisíveis e ConstróiRegiõesDeVértice para cada polígono $P_1, \ldots, P_k$e então calculamos o caminho final cuja complexidade é insignificante próximo do primeiro passo, então a complexidade final é:

$$O\left(\sum_{i=1}^{k}|P_i| \sum_{j=1}^{i} \log|P_j|\right)$$

Enquanto podemos dizer que determinamos a complexidade final do problema, gotaríamos de ter uma forma mais simples e de acordo com o artigo original. Assim, definimos $n = \sum_{i=1}^{k}|P_i|$ e note que:

$$\begin{aligned} O\left(\sum_{i=1}^{k}|P_i| \sum_{j=1}^{i} \log|P_j|\right) &= O\left(\sum_{i=1}^{k}|P_i| \sum_{j=1}^{k} \log|P_j|\right) \\ &= O\left(\left(\sum_{i=1}^{k}|P_i|\right) \cdot \left(\sum_{i=1}^{k} \log|P_i|\right)\right) \\ &= O\left(n \sum_{i=1}^{k} \log|P_i|\right) \end{aligned}$$

Ademais, se fixarmos o valor de k, temos que o valor de $\sum_{i=1}^{k} \log|P_i|$ é máximo quando $|P_1| = \cdots = |P_k| = n/k$. Dessa forma, concluimos que em um pior caso a complexidade é:

$$O(nk \log(n/k))$$

Dessa forma, nosso algoritmo está de acordo com o artigo original.

3 O Problema de Visita de Polígonos Geral

Vamos tomar como base a implementação do algoritmo de Mitchell para o problema restrito que fizemos em *Python*. O código pode ser encontrado no arquivo `TouringPolygons/problem2.py`.

3.1 Definições e Notação

O problema segue de forma similar ao anterior, mas agora também recebemos como entrada 'cercas' $F_0, \ldots, F_k$ tais que para todo $0 \leq i \leq k$ vale que o polígono P_i e P_{i+1} estão contidos em F_i, para tal, consideramos $P_0 = \{s\}$ e $P_{k+1} = \{t\}$. Nosso objetivo é encontrar o caminho de menor comprimento que se inicia em s, termina em t, toca cada polígono P_i em pelo menos um ponto e nunca sai da cerca F_i no seu caminho entre P_i e P_{i+1}.

Dizemos que um caminho π de a até b respeita as cercas $F_i, \ldots, F_j$ se π toca todos os polígonos $P_{i+1}, \ldots, P_j$ e para cada $i \leq l < j$, o trecho de π entre P_l e P_{l+1} está contido em F_l. Ademais, definimos um i-path até p como um caminho mínimo de s até p que respeita as cercas $F_0, \ldots, F_i$. Note que nosso objetivo é encontrar um k-path até t.

A função central desse algoritmo continua sendo Query, no entanto, dessa vez vamos adicionar um novo parâmetro, assim, a função Query(p, i, j) recebe um ponto p e dois índices i e j e retorna o penúltimo ponto q do menor caminho até p que parte de s, toca todos os polígonos $P_1, \ldots, P_i$ e respeita as cercas $F_0, \ldots, F_j$.

Primeiramente, é essencial que $i \leq j$, ademais, se $i = j$ então Query(p, i, j) é simplesmente o penúltimo ponto do i-path até p. Adicionamos o parâmetro j para o caso em que queremos um i-path, mas p está fora da cerca F_i um caso que aparece naturalmente na recursão do algoritmo. Por enquanto, vamos assumir que sabemos como respoder as consultas.

3.2 Caminhos Restritos

Outra função extremamente importante para a implementação desse algoritmo é a função Fenced(p_1, p_2, i, j) que retorna o menor caminho de p_1 até p_2 que respeita as cercas $F_i, \ldots, F_j$. Note que se $i = j$, então Fenced(p_1, p_2, i, j) é simplesmente o segmento $\overline{p_1 p_2}$ se ele estiver contido em F_i e não existe caso contrário. Assim, vamos assumir que $i < j$.