

O Problema de Visita de Polígonos*

Gabriel F. Ushijima[†]

Ernesto G. Birgin[†]

16 de fevereiro de 2026

Resumo

Este trabalho apresenta diferentes implementações de algoritmos para o Problema de Visita de Polígonos (Touring Polygons Problem - TPP), que consiste em encontrar um caminho de comprimento mínimo que visite uma sequência de polígonos no plano. Em sua forma geral, o problema é NP-difícil. Por esse motivo, inicialmente consideramos uma versão simplificada em que os polígonos são convexos e não se sobrepõem, para a qual apresentamos um algoritmo exato baseado em mapas de último passo. Em seguida, para o caso não convexo, particionamos cada polígono não convexo em um conjunto de polígonos convexos e utilizamos nossa solução anterior para resolver cada combinação possível de polígonos convexos. Como essa solução é um tanto ineficiente, discutimos estratégias de otimização inteira mista para melhorar a performance dessa estratégia. Por fim, realizamos experimentos computacionais para avaliar o desempenho dos algoritmos propostos em instâncias de diferentes tamanhos e características, comparando-os com solvers comerciais. Os resultados indicam que as abordagens desenvolvidas são eficazes em uma ampla variedade de cenários práticos.

Palavras-chave: problema de visita de polígonos, otimização geométrica, complexidade, experimentos numéricos.

1 Introdução

O Problema de Visita de Polígonos (Touring Polygons Problem - TPP), introduzido em [2], é um problema de otimização geométrica que consiste em determinar um caminho de comprimento mínimo que visite uma sequência de polígonos no plano. Esse tipo de problema surge naturalmente em aplicações de roteirização e planejamento de trajetórias, especialmente no contexto de veículos autônomos, nos quais é necessário garantir a visita eficiente a regiões específicas do espaço, como em tarefas de inspeção automatizada de armazéns [3, 4]. Podemos interpretar esse problema como um caso específico do Problema do Caixeiro Viajante com Regiões (*Traveling Salesman Problem with Neighborhoods* - TSPN) [1], no qual o objetivo é encontrar um caminho de comprimento mínimo que visite um conjunto de regiões arbitrárias no plano, sendo que, no TPP, essas regiões são representadas por polígonos e a ordem de visita é pré-definida.

Dentre as diversas variações do problema, consideramos inicialmente uma versão simplificada em que os polígonos são convexos e não se sobrepõem. Consideramos ainda que a distância é medida na norma Euclidiana. Para esse caso, apresentamos três abordagens distintas, as duas primeiras são baseadas em ideias descritas em [2], enquanto a terceira é uma extensão que utiliza técnicas de memoização para melhorar o desempenho no caso médio. Considerando que n é o número total de vértices de todos os polígonos e k é o número de polígonos, as abordagens propostas são as seguintes:

*Este trabalho foi parcialmente apoiado pela agência FAPESP (processo 2025/13861-1).

[†]Instituto de Matemática e Estatística, Universidade de São Paulo, Rua do Matão, 1010, Cidade Universitária, 05508-090, São Paulo, SP, Brazil (e-mail: gabriel_ushijima@ime.usp.br, egbirgin@ime.usp.br).

1. Implementação intuitiva das ideias descritas em [2], baseadas em mapas de último passo, resolvendo o problema em tempo $O(n^2)$. Buscamos familiarizar o leitor com os conceitos e técnicas fundamentais para a resolução do TPP, como as regiões de primeiro contato e os mapas de último passo, que serão utilizados em todas as implementações subsequentes.
2. Implementação que melhora a eficiência da primeira ao empregar uma estratégia de busca binária para a localização de pontos nos mapas de último passo, resolvendo o problema em tempo $O(nk \log(n/k))$, como descrito em [2].
3. Implementação que utiliza uma técnica de memoização, com o objetivo de evitar cálculos redundantes e reduzir o custo computacional. Mantemos a mesma complexidade de pior caso da abordagem anterior, $O(nk \log(n/k))$, mas, a depender do tamanho dos polígonos, pode apresentar um desempenho significativamente melhor na prática.

Em seguida, buscamos resolver o caso geral do TPP, no qual os polígonos podem ser não convexos. Para isso, vamos particionar cada polígono não convexo em um conjunto de polígonos convexos e aplicar as soluções desenvolvidas para o caso convexo sobre cada combinação possível de polígonos convexos resultantes. Essa abordagem é, a priori, ineficiente, assim buscamos otimizar o processo por meio de técnicas de otimização combinatória.

Além disso, outra forma de resolver o caso geral do TPP é modelá-lo como um problema de programação inteira mista (Mixed Integer Linear Programming - MILP) e utilizar solvers comerciais, como o Gurobi, para obter uma solução. Essa abordagem também resolve o problema de forma exata utilizando ferramentas bem conhecidas, assim, buscamos mostrar que nossa implementação é competitiva mesmo quando comparada a solvers comerciais de alto desempenho.

2 O Problema de Visita de Polígonos Convexos

Inicialmente, consideramos o TPP em sua versão mais simples, na qual os polígonos são convexos e não se sobrepõem, denominada TPP Irrestrito. Utilizamos tanto a formulação quanto a solução apresentadas em [2] como base para nossas implementações. Para nossos propósitos, definimos o problema da seguinte forma:

Problema 1: TPP Convexo

Dado um ponto inicial $s \in \mathbb{R}^2$, um ponto final $t \in \mathbb{R}^2$ e uma sequência de polígonos convexos e disjuntos P_1, \dots, P_k , dados por sequências de vértices ordenados em sentido anti-horário, encontre um caminho π de comprimento mínimo que se inicia em s , termina em t , e para o qual existem pontos p_1, \dots, p_k tais que $p_i \in P_i$ para todo $i = 1, 2, \dots, k$, e o caminho π passa por esses pontos nessa mesma ordem.

Todas as implementações a seguir têm como base as definições e propriedades apresentadas em [2]. A seguir, incluímos as principais definições que serão utilizadas.

Caminho Ótimo (i -path): Um i -path ($0 \leq i \leq k$) até um ponto $p \in \mathbb{R}^2$ é um caminho de comprimento mínimo que se inicia em s , termina em p e visita os polígonos P_1, P_2, \dots, P_i em ordem. É fácil notar que qualquer i -path deve ser a união de segmentos de reta; assim, tratamos caminhos como sequências ordenadas de pontos em \mathbb{R}^2 .

Veja a Figura 1(a) para um exemplo de um 2-path. Note que a ordem deve ser respeitada, pois visitar P_2 antes de P_1 seria mais eficiente nesse caso.

Região de Primeiro Contato (T_i): Denotamos por T_i a região de primeiro contato de um polígono P_i , definida como o conjunto de pontos p em seu perímetro tais que o último segmento do $(i-1)$ -path até p intersecta P_i exclusivamente nesse ponto. Em outras palavras, T_i

consiste dos pontos p no perímetro de P_i tais que o último segmento do $(i - 1)$ -path até p não atravessa o interior de P_i .

Veja a Figura 1(b) para um exemplo de região de primeiro contato de dois polígonos P_1 e P_2 . Note que uma região de primeiro contato é sempre composta por uma região contínua de arestas do polígono, como é o caso de T_1 e T_2 na figura.

Mapa de Último Passo (S_i): Denotamos por S_i o mapa de último passo de um polígono P_i , que consiste em uma partição do plano em um número finito de regiões. Na partição podem existir duas ou mais regiões do tipo “região de vértice”, uma ou mais regiões do tipo “região de aresta” e exatamente uma região do tipo “região de atravessar”. Cada região de vértice em S_i está associada a um vértice v de P_i assim como cada região de aresta está associada a uma aresta e de P_i . Isso não quer dizer que todo vértice e toda aresta de P_i tenha uma região associada em S_i .

Um ponto $p \in \mathbb{R}^2$ pertence a uma região de vértice associada ao vértice v se e somente se o i -path até p tem \overline{vp} como último segmento. Analogamente, um ponto $p \in \mathbb{R}^2$ pertence a uma região de aresta associada à aresta e se e somente se o último segmento do i -path até p inicia-se no interior de e . Por último, um ponto $p \in \mathbb{R}^2$ pertence à região de atravessar se e somente se não pertence a nenhuma região de vértice ou aresta.

Veja a Figura 1(c) para um exemplo de mapa de último passo de um polígono P_1 . As regiões de vértice são pintadas em vermelho, as regiões de aresta em verde e a região de atravessar em azul. Para cada região, incluímos um exemplo de ponto pertencente a ela, bem como o i -path até esse ponto, ilustrando a definição de cada tipo de região. Note que o mapa de último passo é uma partição do plano, ou seja, cada ponto do plano pertence a exatamente uma região do mapa, considerando que o polígono em si faz parte da região de atravessar.

Como demonstrado em [2, **Lemma 3**], se dispomos de S_1, \dots, S_i , então é possível determinar o i -path até qualquer ponto $p \in \mathbb{R}^2$ de maneira eficiente. Para tal, consideramos os três casos possíveis para o último segmento do i -path até p , que dependem da região de S_i à qual p pertence:

Região de Vértice: Se p pertence a uma região associada a um vértice v de P_i , então o último segmento do i -path até p é o segmento \overline{vp} . Assim, o i -path até p é simplesmente o $(i - 1)$ -path até v , seguido do segmento \overline{vp} . A Figura 2(a) ilustra essa situação.

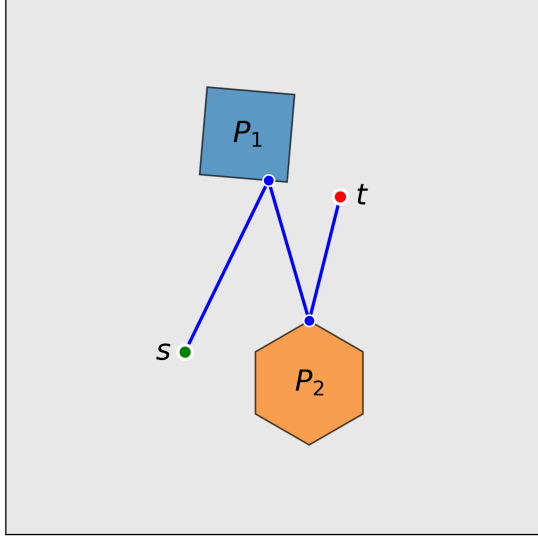
Região de Aresta: Se p pertence a uma região associada a uma aresta $e = \overline{uv}$ de P_i , então refletimos o ponto p em relação à reta que contém e , obtendo um ponto refletido p' . Calculamos o $(i - 1)$ -path até p' cujo último segmento parte de um ponto q' , então chamamos de q a interseção do segmento $\overline{q'p'}$ com a aresta e . O i -path até p é então formado pela parte do $(i - 1)$ -path até p' que vai desde o início (ponto s) até o ponto q , seguida do segmento \overline{qp} . A Figura 2(b) ilustra essa situação, mostrando como o ponto deve ser refletido e então como usamos o $(i - 1)$ -path até p' para determinar o i -path até p .

A seguir, apresentamos a fórmula para calcular o ponto refletido p' e o ponto de interseção q , onde usamos o operador \cdot para denotar o produto escalar, o operador $\|\cdot\|$ para denotar a norma euclidiana e o operador \times para denotar o produto vetorial¹, todos em \mathbb{R}^2 :

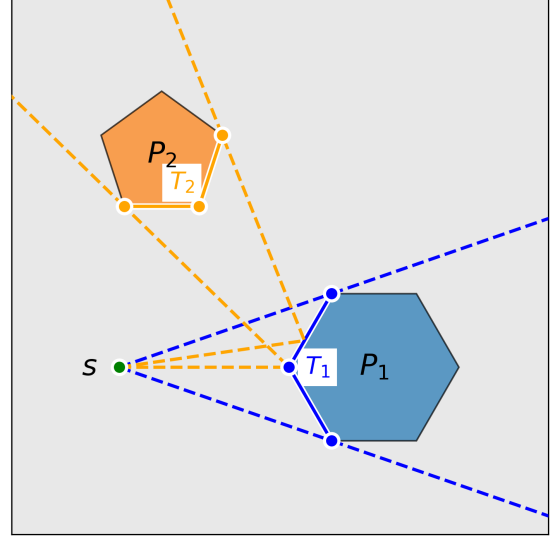
$$p' = p + 2 \frac{(u - p) \cdot (v - u)}{\|v - u\|^2} (v - u), \quad (1)$$

$$q = q' + \frac{(u - q') \times (v - u)}{(p' - q') \times (v - u)} (p' - q'). \quad (2)$$

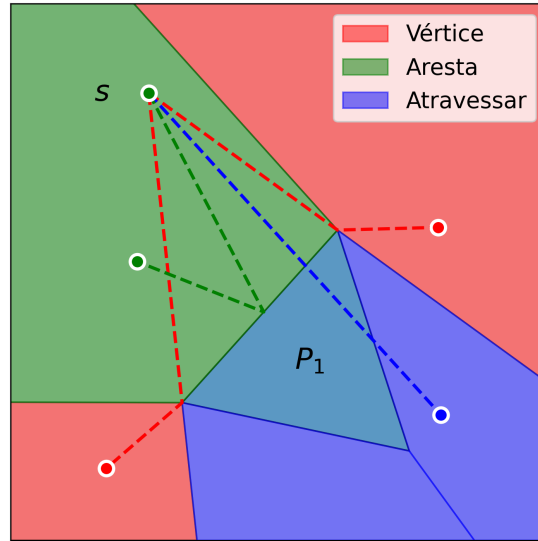
¹Para dois vetores $a = (a_x, a_y)$ e $b = (b_x, b_y)$, temos $a \times b = a_x b_y - a_y b_x$.



(a) Caminho mínimo de s até t visitando P_1 e P_2 .



(b) Região de primeiro contato de P_1 e P_2 .



(c) Mapa de último passo de P_1 .

Figura 1: Exemplos de i -path (a), região de primeiro contato (b) e mapa de último passo (c).

Região de Atravessar: Se p pertence à região de atravessar P_i , então o último segmento do i -path até p atravessa o interior de P_i . Nesse caso, o i -path até p é simplesmente o $(i-1)$ -path até p . A Figura 2(c) ilustra essa situação.

Com base nessas ideias, implementamos o Algoritmo 1, que, dado um ponto $p \in \mathbb{R}^2$ e um índice $1 \leq i \leq k$, retorna o i -path até p como uma sequência de pontos em \mathbb{R}^2 , desconsiderando o último ponto do caminho, p , que é conhecido. Assumimos também a existência de uma função **LocalizaPonto**, que, dado $p \in \mathbb{R}^2$ e $1 \leq i \leq k$, localiza o ponto p no mapa de último passo S_i de P_i e retorna a região R à qual p pertence. Essa região carrega tanto o tipo de região à qual o ponto pertence assim como o vértice v da região se for uma região de vértice e a aresta e se for uma região de aresta.

O Algoritmo 1 permite calcular o i -path até qualquer ponto $p \in \mathbb{R}^2$ de forma eficiente, podendo

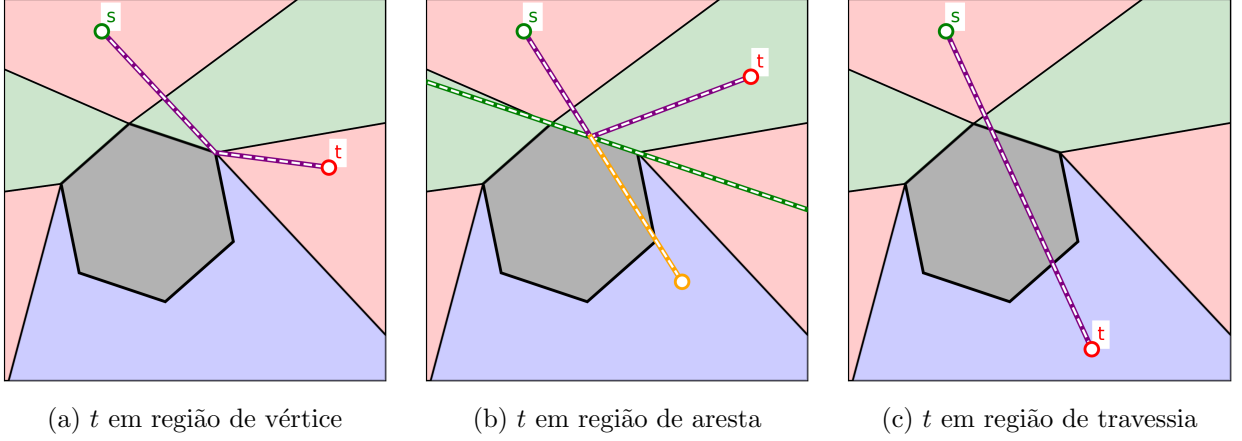


Figura 2: Três casos possíveis no caminho mínimo do ponto s (em verde) até t (em vermelho) ao visitar um polígono. Em vermelho, regiões de vértice; em verde, regiões de aresta; em azul, região de atravessar.

Algoritmo 1: Consulta Caminho

Entrada: $p, i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_i)$

Saída : i -path até p como uma sequência de pontos em \mathbb{R}^2 , desconsiderando p .

```

1 define ConsultaCaminho( $p, i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_i)$ ):
2   se  $i = 0$  :
3     retorna  $[s]$ 
4    $R \leftarrow \text{LocalizaPonto}(p, P_i, T_i, S_i)$ 
5   se  $R$  corresponde a um vértice  $v$  de  $P_i$  :
6      $\mathbb{P} \leftarrow \text{ConsultaCaminho}(v, i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1}))$ 
7     retorna  $\mathbb{P} \oplus [v]$  // Adiciona  $v$  ao final do caminho  $\mathbb{P}$ 
8   senão se  $R$  corresponde a uma aresta  $e = \overline{uv}$  de  $P_i$  :
9      $p' \leftarrow$  reflexão de  $p$  em relação à reta que contém  $e$  (Veja 1)
10     $\mathbb{P} \leftarrow \text{ConsultaCaminho}(p', i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1}))$ 
11     $q' \leftarrow$  último ponto de  $\mathbb{P}$ 
12     $q \leftarrow$  ponto de interseção entre as retas que passam por  $\overline{q'p'}$  e  $e$  (Veja 2)
13    retorna  $\mathbb{P} \oplus [q]$  // Adiciona  $q$  ao final do caminho  $\mathbb{P}$ 
14   senão se  $R$  corresponde a atravessar  $P_i$  :
15     retorna  $\text{ConsultaCaminho}(p, i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1}))$ 

```

ser utilizado para obter o k -path até t , que corresponde exatamente à solução do problema. No entanto, frequentemente nos interessamos apenas no último segmento do i -path até p . Além disso, no caso em que um ponto pertence a uma região associada a um vértice de P_i , é possível determinar o último segmento diretamente, sem a necessidade de calcular o restante do caminho. Dessa forma, implementamos o Algoritmo 2, que, dado um ponto $p \in \mathbb{R}^2$ e um índice $1 \leq i \leq k$, retorna o ponto inicial do último segmento do i -path até p .

Esses dois algoritmos serão utilizados em todas as implementações discutidas a seguir. A diferença entre elas estará na implementação do algoritmo **LocalizaPonto** e na forma como calculamos as regiões de primeiro contato T_i e os mapas de último passo S_i para cada polígono P_i .

Algoritmo 2: Consulta Último Passo

Entrada: $p, i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_i)$ **Saída** : Ponto que precede p no i -path até p .

```
1 define ConsultaÚltimoPasso( $p, i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_i)$ ):
2   se  $i = 0$  :
3     retorna  $s$ 
4    $R \leftarrow \text{LocalizaPonto}(p, P_i, T_i, S_i)$ 
5   se  $R$  corresponde a um vértice  $v$  de  $P_i$  :
6     retorna  $v$ 
7   senão se  $R$  corresponde a uma aresta  $e = \overline{uv}$  de  $P_i$  :
8      $p' \leftarrow$  reflexão de  $p$  em relação à reta que contém  $e$  (Veja 1)
9      $q' \leftarrow \text{ConsultaÚltimoPasso}(p', i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1}))$ 
10     $q \leftarrow$  ponto de interseção entre  $\overline{q'p'}$  e  $e$  (Veja 2)
11    retorna  $q$ 
12  senão se  $R$  corresponde a atravessar  $P_i$  :
13    retorna
      ConsultaÚltimoPasso( $p, i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1})$ )
```

2.1 Primeira Abordagem

A alto nível, o algoritmo para resolver o TPP Convexo é o seguinte: para cada $1 \leq i \leq k$, calculamos a região de primeiro contato T_i de P_i utilizando T_1, \dots, T_{i-1} e S_1, \dots, S_{i-1} através do Algoritmo 3, depois calculamos o mapa de último passo S_i de P_i utilizando T_1, \dots, T_i e S_1, \dots, S_{i-1} através do Algoritmo 4. Por fim, calculamos o k -path até t utilizando o Algoritmo 1. A seguir, vamos detalhar as implementações de cada um desses passos.

Primeiramente, para calcular a região de primeiro contato T_i de um polígono P_i , precisamos determinar quais arestas de P_i pertencem a T_i . Seja $e = \overline{uv}$ uma aresta de P_i , p um ponto qualquer dessa aresta e q o ponto que precede p no $(i-1)$ -path até p . Note que $e \in T_i$ se, e somente se, q está do lado externo de e . Esse teste vale para qualquer ponto na aresta e , assim, escolhemos utilizar um vértice para potencialmente reutilizar esse resultado adiante. A Figura 3 ilustra como podemos usar essa ideia para determinar a região de primeiro contato de um polígono, realizando o teste para ambos os vértices de cada aresta do polígono, note que as arestas em verde tem o ponto que precede os vértices em seu lado externo enquanto as arestas em vermelho tem o ponto que precede os vértices no lado interno. Implementamos essa ideia no Algoritmo 3, utilizando o vértice u como o ponto p para determinar se a aresta e pertence a T_i .

Uma vez que temos T_i , podemos calcular o mapa de último passo S_i de P_i , para tal, vamos primeiro retomar as condições de otimalidade local discutidas em [2, **Local Optimality Conditions.**], que nos permitem determinar o mapa de último passo S_i de um polígono P_i a partir da região de primeiro contato T_i . Seja v um vértice de um polígono qualquer P com arestas incidentes e^1 e e^2 e suponha que desejamos visitar P partindo de q e terminando em p . Para que o caminho mínimo de q até p passe por v , é necessário que p esteja dentro do cone definido por v e por um par ordenado de raios que partem de v determinados a partir de e^1 e e^2 nessa ordem.

O raio associado a uma aresta e incidente a v será $v - q$ se q está do lado interno de e ou $q' - v$ onde q' é a reflexão de q em relação à reta perpendicular a e que passa por v caso contrário. Assim, concluímos a condição de otimalidade local para um vértice v de P , no entanto, como todo i -path é único, temos que otimalidade local implica otimalidade global. Assim, para determinar o mapa de último passo S_i de P_i , basta calcular os pares ordenados de raios associados a cada vértice de P_i que tem uma aresta incidente pertencente a T_i . Implementamos essa ideia no Algoritmo 4,

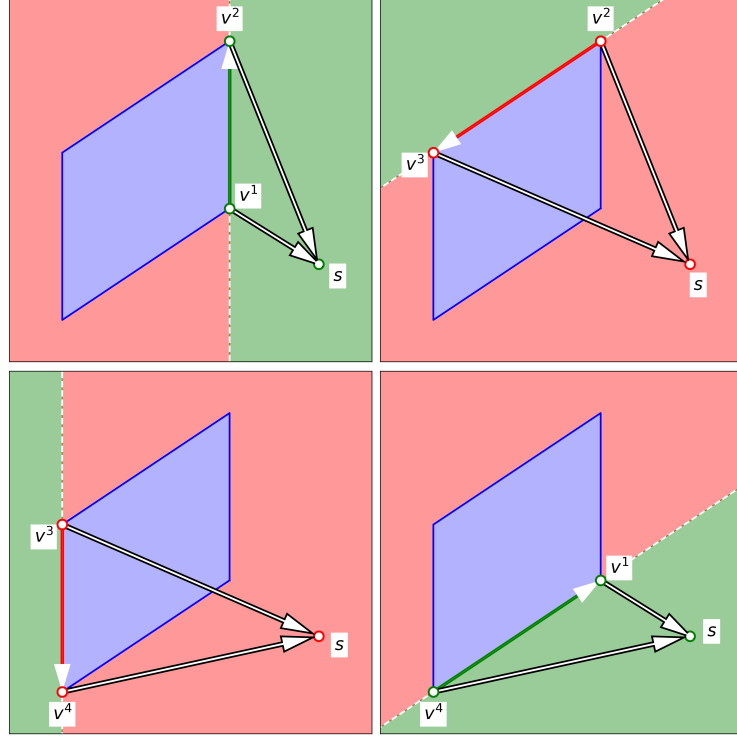


Figura 3: Usamos o Algoritmo 3 para determinar a região de primeiro contato do polígono. Pintamos em verde as arestas que pertencem a T_i e em vermelho as que não pertencem.

Algoritmo 3: Região de Primeiro Contato

Entrada: $i, s, (P_1, \dots, P_i), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1})$

Saída : Região de primeiro contato T_i de P_i como um conjunto de arestas de P_i .

```

1 define RegiãoDePrimeiroContato( $i, s, (P_1, \dots, P_i), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1})$ ):
2    $T_i \leftarrow \emptyset$ 
3   para cada aresta  $e$  de  $P_i$  :
4      $u \leftarrow$  vértice anterior de  $e$ 
5      $v \leftarrow$  vértice posterior de  $e$ 
6      $q \leftarrow$  ConsultaÚltimoPasso( $u, i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1})$ )
7     se  $(q - u) \times (v - u) < 0$  :
8        $T_i \leftarrow T_i \cup e$ 
9   retorna  $T_i$ 

```

onde, para cada vértice v de P_i que tem uma aresta incidente pertencente a T_i , calculamos os raios associados a v e associamos o par ordenado desses raios a v em S_i .

Uma vez que sabemos calcular T_i e S_i , desejamos implementar o algoritmo **LocalizaPonto**, que localiza um ponto p no mapa de último passo S_i . Na primeira abordagem, implementamos esse procedimento de forma direta, verificando inicialmente todas as regiões associadas a vértices, depois todas as regiões associadas a arestas e, caso o ponto não pertença a nenhuma dessas regiões, retornando a região de atravessar P_i . Para verificar se um ponto p pertence a uma região associada a um vértice v de P_i com raios r^1 e r^2 , primeiro determinamos se o ângulo de abertura entre r^1 e r^2 é menor ou maior que 180° . Se o ângulo for menor que 180° , então p pertence à região associada a v se o produto vetorial entre r^1 e \vec{vp} for positivo e o produto vetorial entre \vec{vp} e r^2 for negativo. Caso contrário, p pertence à região associada a v se o produto vetorial entre r^1 e \vec{vp} for positivo

Algoritmo 4: Mapa de Último Passo

Entrada: $i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_{i-1})$

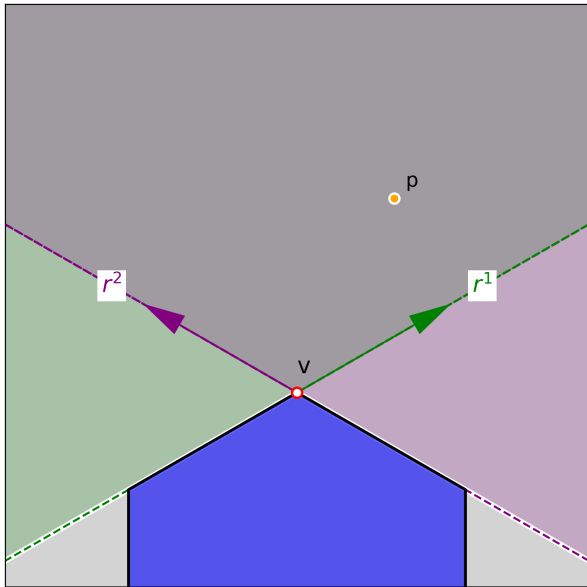
Saída : Mapa de último passo S_i de P_i como uma associação de pares ordenados de raios aos vértices das arestas de T_i .

// Função auxiliar para calcular o raio refletido na aresta $e = \overline{uv}$ incidente a v onde q é o ponto que precede v no $(i-1)$ -path até v .

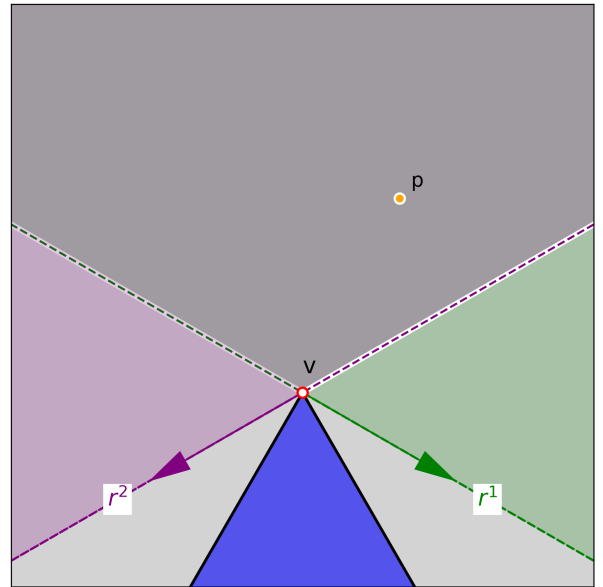
```
1 define Reflexão( $q, v, e$ ):
2   se  $(u - v) \times (q - v) \geq 0$  :
3     retorna  $v - q$ 
4   senão
5      $e_{\perp} \leftarrow$  vetor perpendicular a  $e$  que passa por  $v$ 
6      $q' \leftarrow$  reflexão de  $q$  em relação à reta que contém  $e_{\perp}$  (Veja 1)
7     retorna  $q' - v$ 

8 define MapaDeÚltimoPasso( $i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_{i-1})$ ):
9    $S_i \leftarrow \emptyset$ 
10  para cada vértice  $v$  de  $P_i$  tal que existe uma aresta  $e$  em  $T_i$  incidente a  $v$  :
11     $e^1 \leftarrow$  aresta anterior de  $v$  em  $P_i$ 
12     $e^2 \leftarrow$  aresta posterior de  $v$  em  $P_i$ 
13     $q \leftarrow$  ConsultaÚltimoPasso( $v, i-1, s, (P_1, \dots, P_{i-1}), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1})$ )
14     $r^1 \leftarrow$  Reflexão( $q, v, e^1$ )
15     $r^2 \leftarrow$  Reflexão( $q, v, e^2$ )
16    Associa  $(r^1, r^2)$  a  $v$  e adiciona à  $S_i$ 
17  retorna  $S_i$ 
```

ou o produto vetorial entre \vec{vp} e r^2 for negativo. Essas situações são ilustradas na Figura 4.



(a) Ângulo de abertura menor que 180° .



(b) Ângulo de abertura maior que 180° .

Figura 4: Verificação de pertinência de um ponto a uma região associada a um vértice em um mapa de último passo.

Além disso, desejamos verificar se um ponto p pertence a uma região associada a uma aresta e de P_i . Para isso, consideramos que u e v são os vértices que definem a aresta e , em sentido anti-horário, e tomamos r^1 como o segundo raio associado ao vértice u e r^2 como o primeiro raio associado ao vértice v em S_i . Então, p pertence à região associada a e se estiver do lado externo da aresta e e entre os raios r^1 e r^2 , como ilustrado na Figura 5.

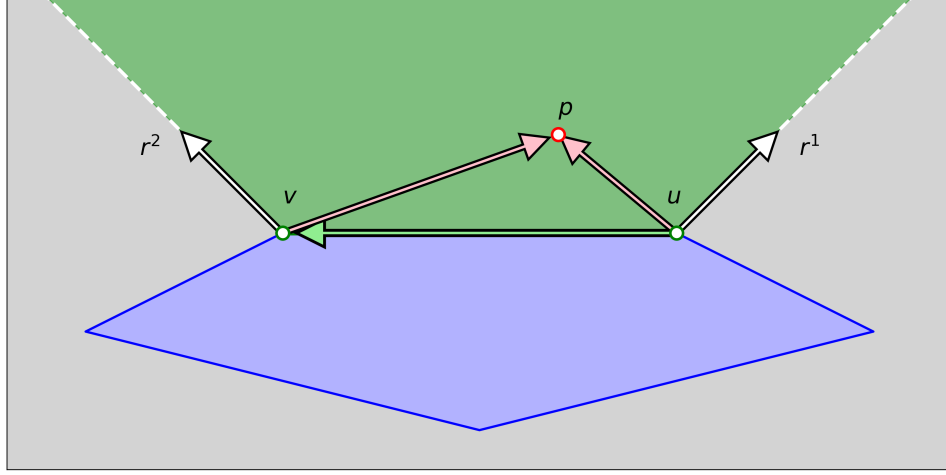


Figura 5: Verificando se um ponto pertence a uma região associada a uma aresta.

Uma vez que conseguimos verificar se um ponto pertence a uma região associada a um vértice ou a uma aresta de P_i , podemos implementar o algoritmo **LocalizaPonto** de forma direta, como mostrado no Algoritmo 5. Realizamos dois laços: o primeiro verifica todas as regiões associadas a vértices de P_i pertencentes a T_i e o segundo verifica todas as regiões associadas a arestas de P_i pertencentes a T_i . Caso o ponto p não pertença a nenhuma dessas regiões, retornamos a região de atravessar P_i .

Algoritmo 5: Localiza Ponto — Primeira Abordagem

Entrada: p, P_i, T_i, S_i

Saída : Região de S_i que contém p .

```

1 define LocalizaPonto( $p, P_i, T_i, S_i$ ):
2   para cada vértice  $v$  de  $P_i$  tal que existe uma aresta  $e$  em  $T_i$  incidente a  $v$  :
3      $(r^1, r^2) \leftarrow$  raios associados a  $v$  em  $S_i$ 
4     se  $r^1 \times r^2 \geq 0$  :
5       se  $(r^1 \times (p - v) > 0) \wedge (r^2 \times (p - v) < 0)$  :
6         retorna região associada a  $v$ 
7     senão
8       se  $(r^1 \times (p - v) > 0) \vee (r^2 \times (p - v) < 0)$  :
9         retorna região associada a  $v$ 
10  para cada aresta  $e = \overrightarrow{uv}$  de  $P_i$  pertencente a  $T_i$  :
11     $r^1 \leftarrow$  segundo raio associado a  $u$  em  $S_i$ 
12     $r^2 \leftarrow$  primeiro raio associado a  $v$  em  $S_i$ 
13    se  $(r^1 \times (p - u) > 0) \wedge (r^2 \times (p - v) < 0) \wedge ((p - u) \times (v - u) < 0)$  :
14      retorna região associada a  $e$ 
15  retorna região de atravessar  $P_i$ 

```

Finalmente, uma vez que sabemos calcular T_i , S_i e localizar um ponto p em S_i , podemos implementar o algoritmo completo para resolver o TPP, conforme mostrado no Algoritmo 6. Esse algoritmo itera sobre todos os polígonos P_1, \dots, P_k , calculando T_i e S_i para cada polígono por meio dos algoritmos discutidos anteriormente. Após calcular todas as regiões de primeiro contato e mapas de último passo, o algoritmo retorna o k -path até o ponto t utilizando o Algoritmo 1. Esse algoritmo também será utilizado na segunda abordagem, na qual apenas o algoritmo `LocalizaPonto` será modificado.

Algoritmo 6: TPP Irrestrito — Implementação Completa

Entrada: Pontos s e t , sequência de polígonos convexos disjuntos (P_1, \dots, P_k)

Saída : k -path até t .

```

1 define TPP_Irrestrito( $s, t, (P_1, \dots, P_k)$ ):
2    $(T_1, \dots, T_k) \leftarrow \emptyset$ 
3    $(S_1, \dots, S_k) \leftarrow \emptyset$ 
4   para cada  $i \leftarrow 1$  até  $k$  :
5      $T_i \leftarrow \text{RegiãoDePrimeiroContato}(i, s, (P_1, \dots, P_i), (T_1, \dots, T_{i-1}), (S_1, \dots, S_{i-1}))$ 
6      $S_i \leftarrow \text{MapaDeÚltimoPasso}(i, s, (P_1, \dots, P_i), (T_1, \dots, T_i), (S_1, \dots, S_{i-1}))$ 
7   retorna ConsultaCaminho( $t, k, s, (P_1, \dots, P_k), (T_1, \dots, T_k), (S_1, \dots, S_k)$ )

```

2.2 Segunda Abordagem

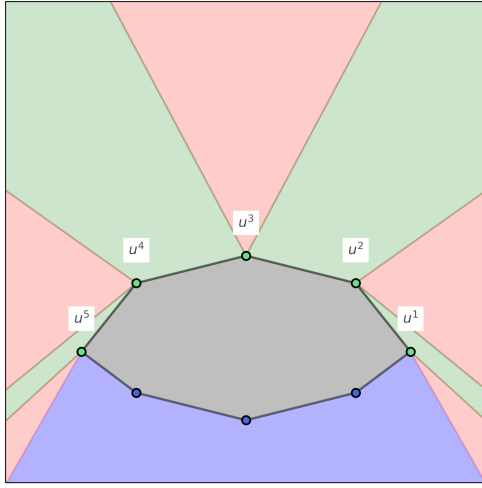
Na segunda abordagem, alteramos apenas a implementação do algoritmo `LocalizaPonto`, utilizando uma estratégia de busca binária para localizar o ponto p no mapa de último passo S_i . Definimos u^1, \dots, u^m como os vértices de P_i que pertencem a T_i , ordenados em sentido anti-horário. A ideia central do algoritmo é utilizar *arestas fictícias* que conectam dois vértices u^i e u^j , formando uma nova região R . Essa região possui a propriedade de que é possível verificar se p pertence a R em tempo $O(1)$, e p está em R se e somente se está em alguma região de vértice ou de aresta entre u^i e u^j . Dessa forma, podemos empregar uma estratégia de busca binária para localizar p em S_i .

Assumimos que o ponto p não está no interior do polígono P_i . Esse fato é garantido pela hipótese de que os polígonos são disjuntos, incluindo os pontos s e t . Caso seja necessário lidar com situações em que p possa estar no interior de P_i , o algoritmo pode ser iniciado com uma verificação de pertinência em tempo $O(\log|P_i|)$. Essa verificação, no entanto, não será discutida aqui, pois não é o foco deste trabalho.

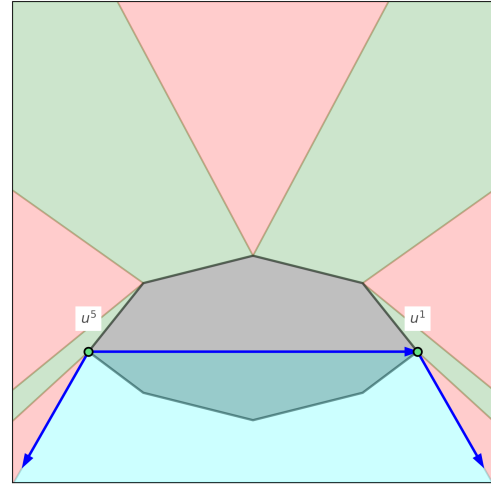
A Figura 6 será utilizada para descrever cada etapa do algoritmo. O primeiro quadro apresenta um polígono P_i com 8 vértices, dos quais 5 pertencem a T_i , denotados por u^1, \dots, u^5 . As regiões de vértice estão destacadas em vermelho, as regiões de aresta em verde e a região de travessia em azul. O objetivo é localizar um ponto nesse mapa de último passo. O algoritmo possui dois passos iniciais e, em seguida, inicia a busca binária, conforme descrito a seguir:

Algoritmo para localizar um ponto p em S_i :

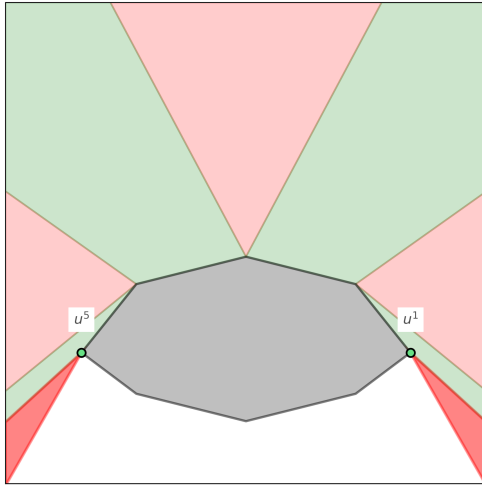
Passo 1. Verificar se p está na região de travessia de P_i (Quadro 2): criamos uma aresta fictícia entre u^5 e u^1 , gerando uma região R que contém exatamente a região de travessia e parte de P_i . A verificação se p pertence a R é feita utilizando a mesma estratégia do Algoritmo 5, considerando u^5 e u^1 como os vértices que definem a aresta fictícia e o segundo raio de u^5 e o primeiro raio de u^1 como os raios que delimitam R . Caso p pertença a R , retornamos a região de travessia.



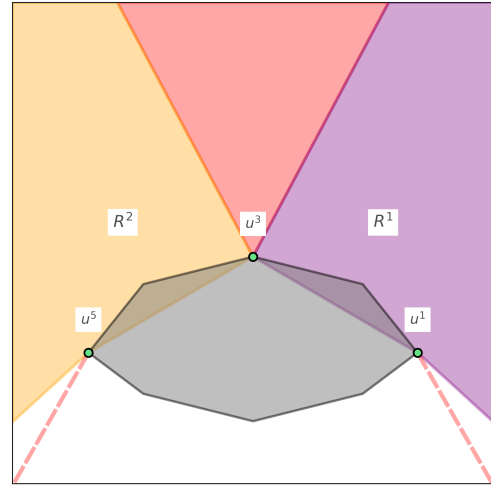
(a) Visão geral do polígono.



(b) Verificando região de travessia.



(c) Verificando regiões de vértice adjacentes à região de travessia.



(d) Particionando o restante do plano em duas regiões de aresta e uma de vértice.

Figura 6: Ilustração dos três passos do algoritmo de busca binária para localizar um ponto em um mapa de último passo.

Passo 2. Verificar se p está em alguma região de vértice de u^1 ou u^5 (Quadro 3): utilizamos a mesma verificação do Algoritmo 5. Se p pertencer a alguma dessas regiões, retornamos a região correspondente.

Passo 3. Busca binária (Quadro 4): sabendo que p não está nem na região de travessia nem nas regiões de vértice de u^1 ou u^5 , iniciamos a busca binária entre os vértices u^1 e u^5 . Seleccionamos u^3 como vértice intermediário e verificamos se p pertence à região de vértice de u^3 . Em caso afirmativo, retornamos essa região.

Caso contrário, criamos duas arestas fictícias, uma entre u^1 e u^3 e outra entre u^3 e u^5 , gerando as regiões R_1 (em roxo) e R_2 (em laranja). Verificamos se p pertence a R_1 ; se sim, repetimos esse passo recursivamente considerando apenas os vértices entre u^1 e u^3 . Caso contrário, sabemos que p pertence a R_2 e repetimos esse passo considerando apenas os vértices entre u^3 e u^5 . Esse passo é repetido até que p pertença a uma região de vértice ou restem apenas dois vértices em que concluímos que p está na região de aresta entre esses dois vértices.

A maior parte desse algoritmo é direta de implementar, exceto pela verificação se p pertence à região R_1 . Isso ocorre porque essa região pode assumir formas que não são cobertas pela verificação original do Algoritmo 5. Em particular, ao criar uma aresta fictícia entre dois vértices u e v com raios r^1 e r^2 , pode ocorrer que esses raios formem um ângulo maior que 180° ou estejam em lados distintos da aresta fictícia, como ilustrado na Figura 7.

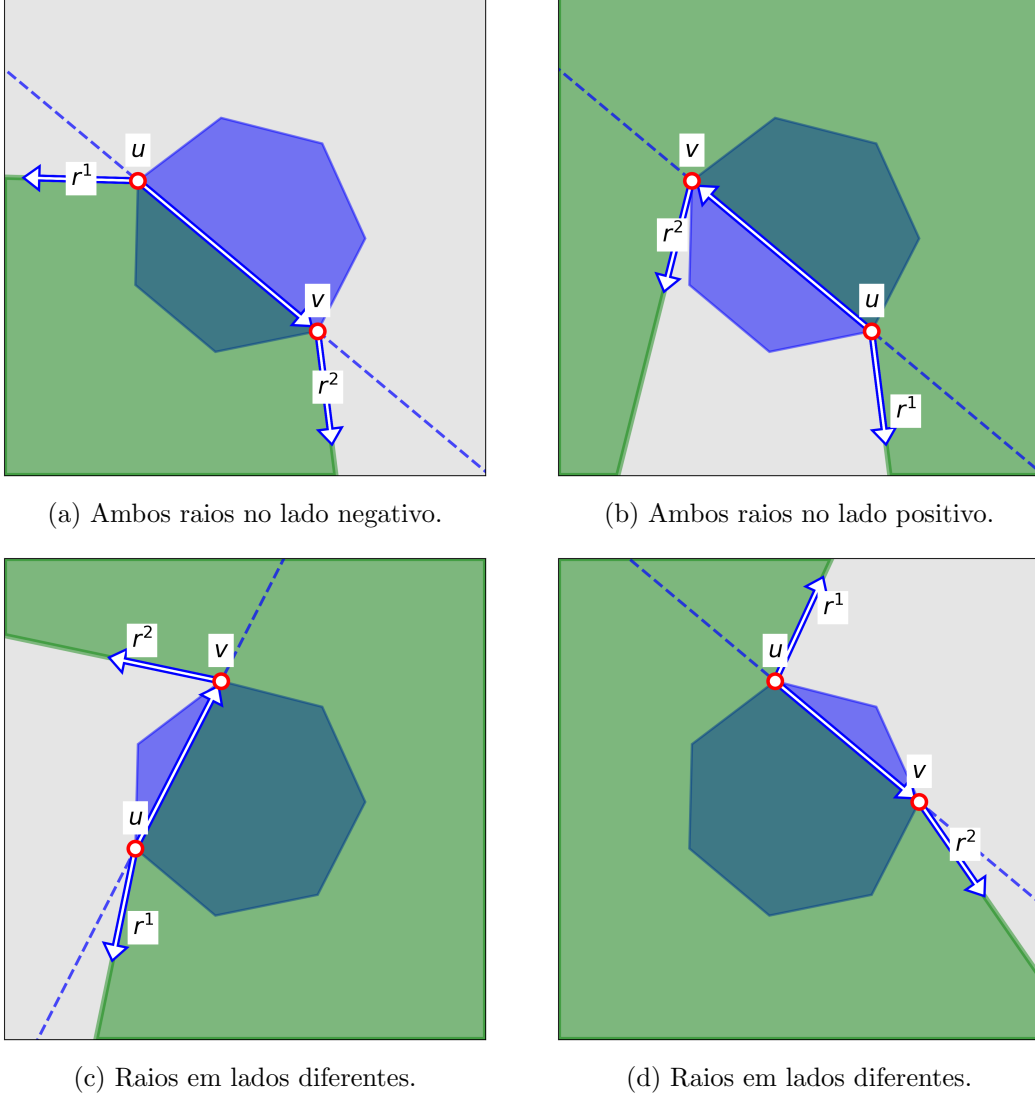


Figura 7: Ilustração dos 4 casos possíveis ao verificar se um ponto pertence à região de uma aresta fictícia entre dois vértices.

Desejamos determinar se um ponto p pertence à região de aresta fictícia entre os vértices u e v , delimitada pelos raios r^1 partindo de u e r^2 partindo de v . Os casos são definidos pelo lado em que r^1 e r^2 se encontram em relação à direção $d = v - u$, e tratamos cada situação separadamente:

- (a) $d \times r^1 < 0 \wedge d \times r^2 < 0$: Verificamos se p está entre os raios r^1 e r^2 e do lado externo da aresta \overline{uv} .
- (b) $d \times r^1 > 0 \wedge d \times r^2 > 0$: Esse caso é o complemento do anterior. Verificamos se p não está entre os raios r^1 e r^2 e se está do lado interno da aresta \overline{uv} .
- (c) $d \times r^1 < 0 \wedge d \times r^2 > 0$: Inicialmente verificamos de que lado da reta que contém \overline{uv} o ponto p se encontra. Se p estiver do lado externo, verificamos se está do lado negativo de r^2 ; caso contrário, verificamos se está do lado positivo de r^1 .

- (d) $d \times r^1 > 0 \wedge d \times r^2 < 0$: Procedemos de forma análoga ao caso anterior, invertendo os testes correspondentes aos raios.

Com essas verificações, podemos implementar o algoritmo **LocalizaPonto** utilizando a estratégia de busca binária, conforme apresentado no Algoritmo 7. Nesse algoritmo, implementamos a função auxiliar **VerificaAresta**, responsável por verificar se p pertence à região de uma aresta fictícia entre dois vértices u e v , de acordo com os casos discutidos anteriormente. A verificação se p pertence à região de vértice de um dado vértice u^{mid} é abstraída, pois é idêntica àquela utilizada no Algoritmo 5.

A principal vantagem dessa abordagem é que a busca binária reduz o número de verificações para localizar p em S_i de $O(|T_i|)$ para $O(\log|T_i|)$, o que pode representar uma melhoria significativa quando P_i possui muitos vértices em T_i . Analisaremos o impacto dessa modificação com mais detalhes na análise de complexidade.

2.3 Terceira Abordagem

Antes de discutirmos a terceira abordagem, apresentamos a motivação para essa nova implementação do algoritmo **LocalizaPonto**. Considere o caso de um único polígono P_1 com um grande número de vértices, por exemplo, um milhão. Pela segunda abordagem, calculamos T_1 , em seguida S_1 e, finalmente, realizamos uma única consulta ao mapa de último passo S_1 para localizar o ponto t . No entanto, essa consulta utiliza no máximo cerca de 20 regiões de vértice de P_1 para localizar t em S_1 . Assim, acabamos calculando aproximadamente 999 980 regiões de vértice que nunca são utilizadas. Diante disso, na terceira abordagem buscamos empregar uma estratégia de memoização, de modo a calcular apenas as regiões de vértice de P_i que são efetivamente necessárias para localizar os pontos consultados no mapa de último passo S_i .

O primeiro problema a ser resolvido é determinar se uma dada aresta $e = \overline{uv}$ de P_i pertence a T_i realizando apenas uma única chamada ao algoritmo **ConsultaÚltimoPasso**. Para isso, reutilizamos a ideia do Algoritmo 3, que originalmente calculava T_i por completo. Agora, ao invés de verificar todas as arestas de P_i , testamos apenas a aresta e , calculando o último passo q do $(i - 1)$ -path até o vértice u da aresta e e afirmamos que e está em T_i se e somente se q está do lado externo de e .

Em seguida, precisamos calcular os raios associados a um vértice v de P_i em S_i , novamente realizando apenas uma única chamada ao algoritmo **ConsultaÚltimoPasso**. Para isso, adaptamos a ideia do Algoritmo 4, que originalmente calculava S_i por completo. Agora, em vez de calcular os raios para todos os vértices de P_i , calculamos apenas os raios associados ao vértice v . Contudo, esse algoritmo pressupõe que sabemos quais arestas de P_i pertencem a T_i , de modo que precisamos determinar se as arestas incidentes em v pertencem a T_i sem realizar chamadas adicionais ao algoritmo **ConsultaÚltimoPasso**.

Para resolver isso, retomamos a caracterização de que, dada uma aresta e de P_i , um ponto p em e e o último passo q do $(i - 1)$ -path até p , temos que $e \in T_i$ se e somente se q está do lado externo de e . Assim, seja v um vértice de P_i e sejam e^1 e e^2 as arestas incidentes em v . Calculamos o último passo q do $(i - 1)$ -path até v e verificamos se o ponto q está do lado externo de e^1 e de e^2 , determinando, assim, se essas arestas pertencem a T_i respectivamente. Com essa informação, podemos adaptar diretamente o Algoritmo 4 para calcular apenas os raios associados a v em S_i .

Uma observação importante é que, como não conhecemos previamente T_i , pode ocorrer de tentarmos calcular os raios associados a um vértice v de P_i em S_i sem que nenhuma das arestas incidentes em v pertença a T_i . Nesse caso, o algoritmo não precisa de qualquer modificação, pois seu comportamento natural é retornar dois raios com a mesma direção do último segmento do $(i - 1)$ -path que chega a v . Isso corresponde a uma região vazia em S_i , o que é consistente com o fato de que v não pertence a T_i .

Com essas ideias, podemos finalmente descrever a implementação do algoritmo **LocalizaPonto** utilizando a estratégia de memoização. Reutilizamos a busca binária da segunda abordagem, mas

Algoritmo 7: Localiza Ponto — Segunda Abordagem

Entrada: p, P_i, T_i, S_i **Saída** : Região de S_i que contém p .

```
1 define LocalizaPonto( $p, P_i, T_i, S_i$ ):
2   define VerificaAresta( $u, v$ ):
3      $r^1 \leftarrow$  segundo raio associado a  $u$  em  $S_i$ 
4      $r^2 \leftarrow$  primeiro raio associado a  $v$  em  $S_i$ 
5      $d \leftarrow v - u$ 
6     se  $d \times r^1 < 0 \wedge d \times r^2 < 0$  :
7        $\lfloor$  retorna  $r^1 \times (p - u) > 0 \wedge r^2 \times (p - v) < 0 \wedge (p - u) \times (v - u) < 0$ 
8     senão se  $d \times r^1 > 0 \wedge d \times r^2 > 0$  :
9        $\lfloor$  retorna  $r^1 \times (p - u) \leq 0 \wedge r^2 \times (p - v) \geq 0 \wedge (p - u) \times (v - u) \geq 0$ 
10    senão se  $d \times r^1 < 0 \wedge d \times r^2 > 0$  :
11      se  $(p - u) \times (v - u) < 0$  :
12         $\lfloor$  retorna  $r^2 \times (p - v) < 0$ 
13      senão
14         $\lfloor$  retorna  $r^1 \times (p - u) > 0$ 
15    senão
16      se  $(p - u) \times (v - u) < 0$  :
17         $\lfloor$  retorna  $r^1 \times (p - u) > 0$ 
18      senão
19         $\lfloor$  retorna  $r^2 \times (p - v) < 0$ 
20   $u^1, \dots, u^m \leftarrow$  vértices de  $P_i$  em  $T_i$ , ordenados em sentido anti-horário
21  se VerificaAresta( $u^m, u^1$ ) :
22     $\lfloor$  retorna região de atravessar  $P_i$ 
23  se  $p$  está na região de vértice de  $u^1$  :
24     $\lfloor$  retorna região associada a  $u^1$ 
25  se  $p$  está na região de vértice de  $u^m$  :
26     $\lfloor$  retorna região associada a  $u^m$ 
27   $l \leftarrow 1$ 
28   $r \leftarrow m$ 
29  while  $l + 1 \neq r$  do
30     $mid \leftarrow \lfloor (l + r) / 2 \rfloor$ 
31    se  $p$  está na região de vértice de  $u^{mid}$  :
32       $\lfloor$  retorna região associada a  $u^{mid}$ 
33    se VerificaAresta( $u^l, u^{mid}$ ) :
34       $\lfloor$   $r \leftarrow mid$ 
35    senão
36       $\lfloor$   $l \leftarrow mid$ 
37  retorna região associada a aresta  $\overline{u^l u^r}$ 
```

agora os passos iniciais são simplificados, pois a verificação da região de travessia é postergada para o final do algoritmo. O procedimento segue da seguinte forma:

Algoritmo para localizar um ponto p em S_i :

Passo 1. Sejam v^1, \dots, v^m os vértices de P_i ordenados em sentido anti-horário. Calculamos os raios associados a v^1 em S_i e verificamos se p pertence à região de vértice de v^1 . Em caso afirmativo, retornamos essa região.

Passo 2. Seja $v' = v^{\lfloor m/2 \rfloor}$. Calculamos os raios associados a v' em S_i e verificamos se p pertence à sua região de vértice. Se isso ocorrer, retornamos essa região.

Caso contrário, criamos as arestas fictícias entre u e v' e entre v' e v^m , gerando as regiões R_1 e R_2 . Verificamos se p pertence a R_1 utilizando o mesmo procedimento descrito no Algoritmo 7. Se sim, repetimos esse passo considerando apenas os vértices entre u e v' ; caso contrário, repetimos esse passo considerando apenas os vértices entre v' e v^m .

Assim seguimos até que encontremos um vértice v^k tal que p pertença à sua região de vértice ou restem apenas dois vértices v^l e v^r . Nesse último caso, sabemos que p deve estar na região de aresta entre esses vértices.

Passo 3. Caso p não tenha sido localizado em uma região de vértice, concluímos que ele está na região de aresta entre os dois últimos vértices v^l e v^r . Para finalizar, com uma única consulta, determinamos se essa aresta pertence a T_i . Se pertencer, retornamos a região de aresta correspondente; caso contrário, retornamos a região de travessia de P_i .

Dessa forma, concluímos a descrição da terceira abordagem para a implementação do algoritmo **LocalizaPonto** utilizando memoização. Optamos por não apresentar o pseudocódigo completo dessa abordagem, pois ele seria muito semelhante ao Algoritmo 7, diferindo apenas pelas modificações discutidas acima.

2.4 Análise de Complexidade

Começamos analisando a complexidade da primeira abordagem. Seja $|P_i|$ o número de vértices do polígono P_i e $|T_i|$ o número de arestas em T_i . Seja ainda k o número total de polígonos na sequência P_1, \dots, P_k e n o número total de vértices em todos os polígonos, isto é,

$$n = \sum_{i=1}^k |P_i|.$$

LocalizaPonto 5: A complexidade desse algoritmo é $O(|T_i|)$, pois, no pior caso, é necessário verificar todas as regiões de vértice e de aresta associadas a T_i para localizar o ponto p .

ConsultaÚltimoPasso 2 e ConsultaCaminho 1: Ambos os algoritmos iniciam com uma chamada ao algoritmo **LocalizaPonto**. O pior caso ocorre quando o ponto p pertence à região de travessia de P_i ou a uma região de aresta, o que exige uma nova consulta ao mapa de último passo S_{i-1} . Assim, no pior caso, são feitas i chamadas ao algoritmo **LocalizaPonto**, resultando em uma complexidade total de:

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

RegiãoDePrimeiroContato 3 e MapaDeÚltimoPasso 4: Ambos os algoritmos realizam uma única passagem pelos vértices de P_i , fazendo uma consulta para cada vértice. Dessa forma, a complexidade desses algoritmos é:

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

TPP completo 6: O algoritmo executa uma chamada para calcular T_i e S_i para cada $i = 1, \dots, k$, e ao final realiza uma chamada para consultar o caminho até o ponto t . Assim, a complexidade total do algoritmo TPP é dada por:

$$\begin{aligned} O\left(\sum_{i=1}^k |P_i| \sum_{j=1}^i |T_j| + \sum_{j=1}^k |T_j|\right) &= O\left(\sum_{i=1}^k |P_i| \sum_{j=1}^i |P_j| + \sum_{j=1}^k |P_j|\right) \\ &= O\left(\sum_{i=1}^k |P_i| \sum_{j=1}^i |P_j|\right) \\ &= O\left(\sum_{i=1}^k |P_i| \sum_{j=1}^k |P_j|\right) \\ &= O\left(\sum_{i=1}^k |P_i| \cdot n\right) \\ &= O(n^2). \end{aligned}$$

Assim, concluímos que a complexidade total do algoritmo TPP utilizando a primeira abordagem é $O(n^2)$. A seguir, analisamos a complexidade da segunda abordagem.

LocalizaPonto 7: A complexidade desse algoritmo é $O(\log|T_i|)$, correspondente ao pior caso da busca binária utilizada para localizar o ponto p em S_i .

ConsultaÚltimoPasso 2 e ConsultaCaminho 1: De forma análoga ao caso anterior, o pior caso envolve i chamadas ao algoritmo **LocalizaPonto**, resultando na complexidade:

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

RegiãoDePrimeiroContato 3 e MapaDeÚltimoPasso 4: Cada um desses algoritmos itera sobre os vértices de P_i e realiza uma consulta para cada vértice. Portanto, a complexidade é:

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

TPP completo 6: O custo total do algoritmo é:

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

Observando que a soma $\sum_{j=1}^k \log|P_j|$ é maximizada quando todos os polígonos possuem o mesmo número de vértices, isto é, quando $|P_j| = n/k$ para todo j , obtemos que a complexidade final do algoritmo TPP usando a segunda abordagem é:

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

Dessa forma, concluímos que a complexidade final do algoritmo TPP utilizando a segunda abordagem é $O(nk \log(n/k))$. Essa complexidade coincide exatamente com a apresentada por Dror et al. (2003) [2], o que indica que nossa implementação atinge a mesma eficiência assintótica da proposta original.

Por fim, resta analisar a complexidade da terceira abordagem. Essa análise é mais delicada, pois a complexidade do algoritmo **LocalizaPonto** passa a depender fortemente da instância específica, em particular da distribuição dos vértices nos polígonos e das consultas efetivamente realizadas ao mapa de último passo. Ainda assim, é imediato observar que a terceira abordagem nunca será pior do que a segunda, uma vez que, no pior caso, todas as regiões de vértice em T_i serão calculadas, recuperando exatamente o comportamento da segunda abordagem. Assim, concluímos que a complexidade da terceira abordagem é $O(nk \log(n/k))$ no pior caso, podendo ser significativamente melhor em instâncias favoráveis, dependendo do padrão das consultas realizadas.

Referências

- [1] Esther M. Arkin, Sándor P. Fekete, and Joseph S. B. Mitchell. The traveling salesman problem with neighborhoods: A survey. In Gregory Gutin and Abraham P. Punnen, editors, *The Traveling Salesman Problem and Its Variations*, pages 377–443. Springer, Boston, MA, 2005.
- [2] Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph S. B. Mitchell. Touring a sequence of polygons. In *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '03, pages 473 – 482, New York, NY, USA, 2003. Association for Computing Machinery.
- [3] Jianxian Liu and Hongyuan Liu. Research on path optimization method for warehouse inspection robot. *Applied Artificial Intelligence*, 37(1), 2023.
- [4] Karl J. Obermeyer. Path planning for a uav performing reconnaissance of static ground targets in terrain. *GNC-31: Intelligent Control in Aerospace Applications*, 2009.