O Problema de Visita de Polígonos*

Gabriel F. Ushijima† Ernesto G. Birgin†

19 de janeiro de 2026

Resumo

Este artigo 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, discutimos heurísticas para o caso geral, utilizando técnicas de programação inteira mista. 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 por Dror et al. (2003) [1], 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 [2], [3]. Podemos interpretar esse problema como um caso específico do Problema do Caixeiro Viajante com Regiões (Traveling Salesman Problem with Neighborhoods - TSPN) [4], 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. Para esse caso, apresentamos três abordagens distintas. A primeira é uma implementação direta das ideias descritas por Dror et al. (2003) [1], baseada em mapas de último passo. A segunda abordagem melhora a eficiência da primeira ao empregar uma estratégia de busca binária para a localização de pontos nesses mapas. A terceira abordagem utiliza uma técnica de memoização, com o objetivo de evitar cálculos redundantes e reduzir o custo computacional. Por fim, mostramos que a primeira abordagem possui complexidade $O(n^2)$, enquanto as outras duas apresentam complexidade $O(nk \log(n/k))$, onde n é o número total de vértices de todos os polígonos e k é o número de polígonos.

*Esse 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).

Em seguida, discutimos heurísticas para o caso geral do TPP, no qual os polígonos podem ser côncavos. Baseamo-nos principalmente na ideia de particionar polígonos côncavos em subconjuntos convexos e aplicar, sobre eles, as soluções desenvolvidas para o caso convexo. Além disso, exploramos técnicas de programação inteira mista para modelar o problema e empregar solvers comerciais na obtenção de soluções aproximadas, como o Gurobi. Por fim, comparamos os resultados obtidos por essas abordagens com aqueles produzidos pelas implementações propostas neste trabalho.

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 por Dror et al. (2003) [1] como base para nossas implementações. Para nossos propósitos, definimos o problema da seguinte forma:

> **Problema 1: TPP Irrestrito**
> 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 $P_1, \ldots, P_k$, 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 $p_1, \ldots, p_k$ tais que $p_i \in P_i$ para todo $i = 1, 2, \ldots, k$, e o caminho π passa por esses pontos nessa mesma ordem.

Todas as implementações a seguir têm como base as definições e propriedades apresentadas por Dror et al. (2003) [1]. Assumimos que o leitor possui familiaridade com o conteúdo desse artigo. A seguir, retomamos as principais definições que serão utilizadas.

- **Caminho Ótimo** (i-path): Um i-path até um ponto p é um caminho de comprimento mínimo que se inicia em s, termina em p e visita os polígonos $P_1, P_2, \ldots, P_i$ 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 $\mathbb{R}^2$.
- **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. Intuitivamente, T_i representa os pontos no perímetro de P_i onde o último segmento do $(i-1)$-path não atravessa o interior de P_i.
- **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 tais que dois pontos em $\mathbb{R}^2$ pertencem à mesma região se, e somente se, o último segmento do i-path até esses pontos possui a mesma origem: um vértice de P_i, uma aresta de P_i, ou atravessa o interior de P_i.

Como demonstrado em [1, **Lemma 3**], se dispomos de $S_1, \ldots, S_i$, então é possível determinar o i-path até qualquer ponto p 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 $\overline{vp}$.
- **Região de Aresta**: Se p pertence a uma região associada a uma aresta $e = \overline{v^1 v^2}$ de P_i, então refletimos o ponto p em relação à reta que contém e, obtendo um ponto refletido p'. O i-path até p será exatamente o $(i-1)$-path até p', com exceção de que devemos refletir seu último segmento a partir do ponto em que ele cruza a aresta e.

- **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 1 ilustra esses três casos. O primeiro quadro mostra o caso em que o ponto está em uma região de vértice, o segundo ilustra o caso em que o ponto está em uma região de aresta, bem como o processo de reflexão e construção do caminho, e o terceiro apresenta o caso em que o ponto está na região de atravessar o polígono.

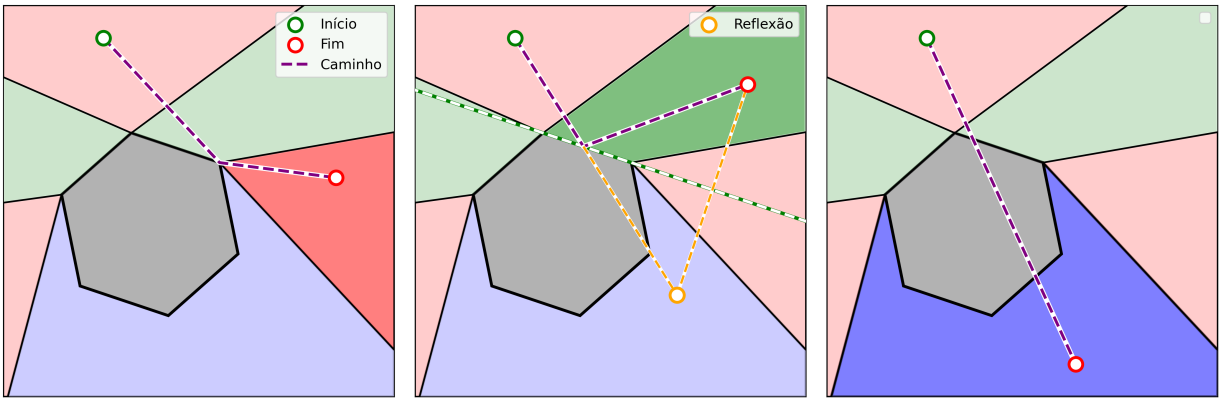


Figura 1: Três casos possíveis para o último segmento de um 1-path até um ponto p.

Com base nessas ideias, implementamos o Algoritmo 1, que, dado um ponto p e um índice i, 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 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 informações suficientes para determinar seu tipo (vértice, aresta ou atravessar) e os dados necessários para reconstruir o caminho conforme descrito anteriormente.

Algoritmo 1: Consulta Caminho

Entrada: $p, i, s, (P_1, \ldots, P_i), (T_1, \ldots, T_i), (S_1, \ldots, S_i)$
Saída : i-path até p como uma sequência de pontos em $\mathbb{R}^2$, desconsiderando p.

```
1  define ConsultaCaminho(p, i, s, (P_1, ..., P_i), (T_1, ..., T_i), (S_1, ..., S_i)):
2      se i = 0 :
3          retorna [s]
4      R ← LocalizaPonto(p, P_i, T_i, S_i)
5      se R corresponde a um vértice v de P_i :
6          ℙ ← ConsultaCaminho(v, i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
7          retorna ℙ ⊕ [v] // Adiciona v ao final do caminho ℙ
8      senão se R corresponde a uma aresta e = v¹v² de P_i :
9          p' ← reflexão de p em relação à reta que contém e
10         ℙ ← ConsultaCaminho(p', i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
11         q' ← último ponto de ℙ
12         q ← ponto de interseção entre as retas que passam por q'p' e e
13         retorna ℙ ⊕ [q] // Adiciona q ao final do caminho ℙ
14     senão se R corresponde a atravessar P_i :
15         retorna ConsultaCaminho(p, i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
```

O Algoritmo 1 permite calcular o i-path até qualquer ponto p de forma eficiente, podendo

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 e um índice i, retorna o ponto inicial do último segmento do i-path até p.

Algoritmo 2: Consulta Último Passo

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

```
1  define ConsultaÚltimoPasso(p, i, s, (P_1, ..., P_i), (T_1, ..., T_i), (S_1, ..., S_i)):
2      se i = 0 :
3          retorna s
4      R ← 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 de P_i :
8          p' ← reflexão de p em relação à reta que contém e
9          q' ← ConsultaÚltimoPasso(p', i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
10         q ← ponto de interseção entre q'p' e e
11         retorna q
12     senão se R corresponde a atravessar P_i :
13         retorna
               ConsultaÚltimoPasso(p, i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
```

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.

2.1 Primeira Abordagem

Primeiramente, calculamos a região de primeiro contato T_i de um polígono P_i. Sabemos que essa região é a união de arestas de P_i, sendo delimitada por vértices de P_i [1, **Lemma 2.**]. Assim, precisamos apenas determinar quais arestas de P_i pertencem a T_i. Seja $e = \overline{v^1 v^2}$ 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, a direção $(q - p)$ está do lado externo de P_i. Como isso vale para qualquer ponto p em e, o critério pode ser testado nos vértices da aresta.

A Figura 2 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. Implementamos essa ideia no Algoritmo 3, utilizando o vértice v^1 como o ponto p para determinar se a aresta e pertence a T_i.

Uma vez que temos $T_0, \ldots, T_i$, podemos calcular o mapa de último passo S_i de P_i. Para isso, observamos que S_i possui exatamente três tipos de regiões: regiões associadas a vértices de P_i pertencentes a T_i, regiões associadas a arestas de P_i pertencentes a T_i e uma única região de atravessar P_i, que corresponde às arestas que não estão em T_i. Além disso, uma região associada a uma aresta e é delimitada pelas regiões associadas aos vértices extremos de e e pela própria aresta

[1] Definimos o operador $\times$ como o produto vetorial em $\mathbb{R}^2$. Ou seja, para dois vetores $a = (a_x, a_y)$ e $b = (b_x, b_y)$, temos $a \times b = a_x b_y - a_y b_x$.

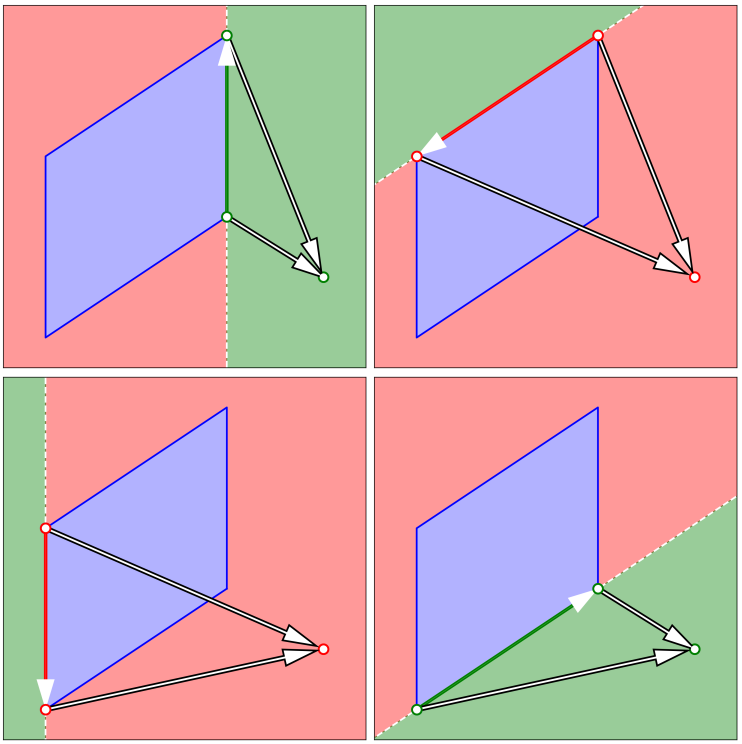


Figura 2: Determinando a região de primeiro contato de um polígono.

Algoritmo 3: Região de Primeiro Contato

Entrada: $i, s, (P_1, \ldots, P_i), (T_1, \ldots, T_{i-1}), (S_1, \ldots, S_{i-1})$
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, ..., P_i), (T_1, ..., T_{i-1}), (S_1, ..., S_{i-1})):
2     T_i ← ∅
3     para cada cada aresta e de P_i :
4         v^1 ← vértice anterior de e
5         v^2 ← vértice posterior de e
6         q ← ConsultaÚltimoPasso(v^1, i − 1, s, (P_1, ..., P_{i-1}), (T_1, ..., T_{i-1}), (S_1, ..., S_{i-1}))
7         se (q − v^1) × (v^2 − v^1) < 0^1 :
8             T_i ← T_i ∪ e
9     retorna T_i
```

e. A região de atravessar P_i é definida como o complemento das regiões associadas a vértices e arestas de P_i. Dessa forma, para construir S_i, precisamos apenas determinar as regiões associadas aos vértices de P_i pertencentes a T_i.

Pelas condições de otimalidade discutidas em [1, **Local Optimality Conditions.**], sabemos que, para cada vértice v de P_i que pertence a T_i, a região associada a v em S_i é um cone definido pelo ponto v e por um par ordenado de raios que partem de v. Esses raios podem ser calculados a partir do último segmento do $(i-1)$-path até v e de sua reflexão em relação às arestas de P_i incidentes em v. Mais especificamente, se e^1 e e^2 são as arestas de P_i, em sentido anti-horário, que contêm v, e q é o ponto que precede v no $(i-1)$-path até v, então o primeiro raio é a reflexão do vetor $\overrightarrow{qv}$ em relação à reta que contém e^1, caso e^1 pertença a T_i, ou simplesmente o vetor $\overrightarrow{qv}$, caso contrário. De forma análoga, o segundo raio é a reflexão do vetor $\overrightarrow{qv}$ em relação à reta que contém e^2, caso e^2 pertença a T_i, ou simplesmente o vetor $\overrightarrow{qv}$, caso contrário. Repetimos esse processo

para todos os vértices de P_i, obtendo assim todas as regiões associadas a vértices de P_i em S_i. Essa ideia é implementada no Algoritmo 4.

Algoritmo 4: Mapa de Último Passo

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

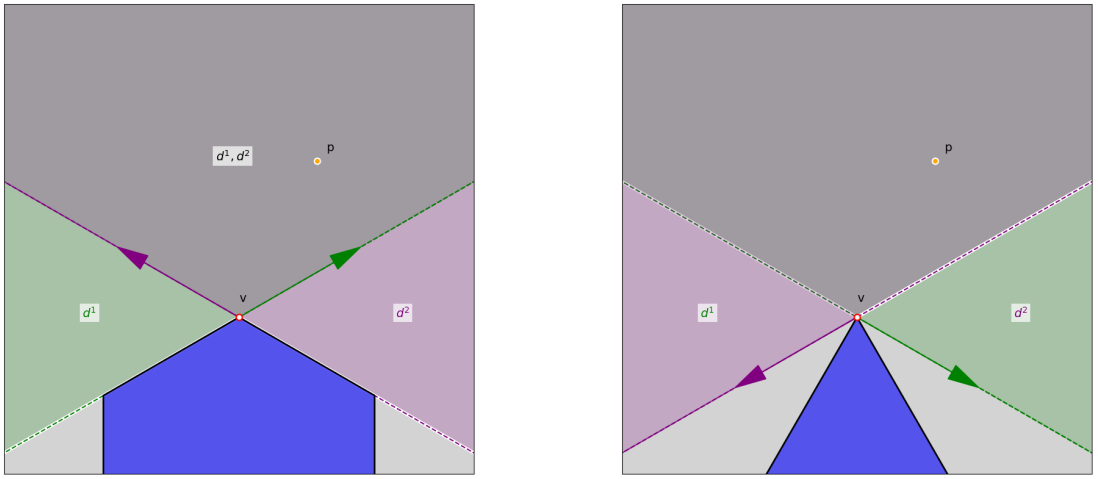
```
1  define MapaDeÚltimoPasso(i, s, (P_1, ..., P_i), (T_1, ..., T_i), (S_1, ..., S_{i-1})):
2      S_i ← ∅
3      para cada vértice v de P_i tal que existe uma aresta e em T_i incidente a v :
4          e^1 ← aresta anterior de v em P_i
5          e^2 ← aresta posterior de v em P_i
6          q ← ConsultaÚltimoPasso(v, i − 1, s, (P_1, ..., P_{i−1}), (T_1, ..., T_{i−1}), (S_1, ..., S_{i−1}))
7          d ← v − q
8          se e^1 pertence a T_i :
9              r^1 ← reflexão de d em relação à reta que contém e^1
10         senão
11             r^1 ← d
12         se e^2 pertence a T_i :
13             r^2 ← reflexão de d em relação à reta que contém e^2
14         senão
15             r^2 ← d
16         Associa (r^1, r^2) a v e adiciona à S_i
17     retorna 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 $\overrightarrow{vp}$ for positivo **e** o produto vetorial entre $\overrightarrow{vp}$ e r^2 for negativo. Caso contrário, p pertence à região associada a v se o produto vetorial entre r^1 e $\overrightarrow{vp}$ for positivo **ou** o produto vetorial entre $\overrightarrow{vp}$ e r^2 for negativo. Essas situações são ilustradas na Figura 3.

Além disso, desejamos verificar se um ponto p pertence a uma região associada a uma aresta e de P_i. Para isso, consideramos que v^1 e v^2 são os vértices que definem a aresta e, em sentido anti-horário, e tomamos r^1 como o segundo raio associado ao vértice v^1 e r^2 como o primeiro raio associado ao vértice v^2 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 4.

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.

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



(a) Ângulo de abertura menor que 180°. (b) Ângulo de abertura maior que 180°.

Figura 3: Verificação de pertinência de um ponto a uma região associada a um vértice em um mapa de último passo.

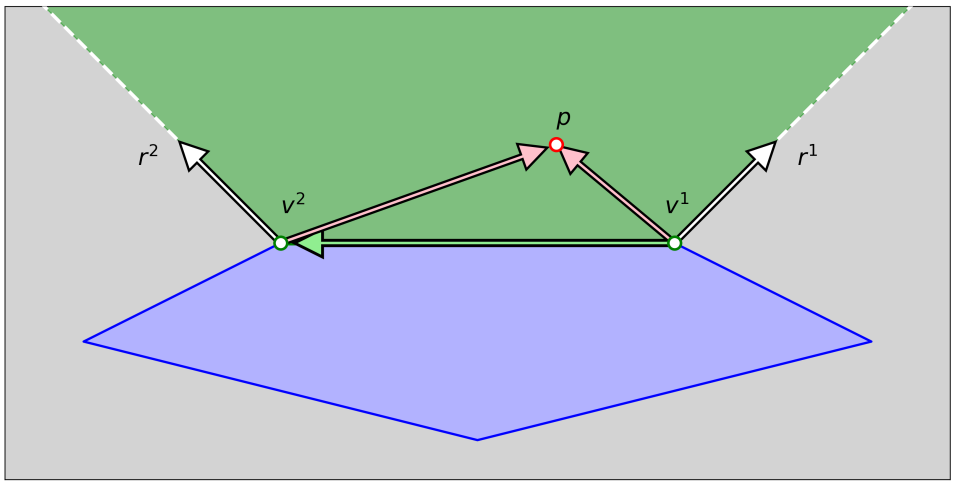


Figura 4: Verificando se um ponto pertence a uma região associada a uma aresta.

algoritmo itera sobre todos os polígonos $P_1, \ldots, P_k$, 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.

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 $u^1, \ldots, u^m$ 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

Algoritmo 5: Localiza Ponto — Primeira Abordagem

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

define LocalizaPonto(p, P_i, T_i, S_i)**:**
  para cada vértice v de P_i tal que existe uma aresta e em T_i incidente a v **:**
    $(r^1, r^2) \leftarrow$ raios associados a v em S_i
    se $r^1 \times r^2 \geq 0$ **:**
      se $(r^1 \times (p - v) > 0) \wedge (r^2 \times (p - v) < 0)$ **:**
        retorna região associada a v
    senão
      se $(r^1 \times (p - v) > 0) \vee (r^2 \times (p - v) < 0)$ **:**
        retorna região associada a v
  para cada aresta $e = \overline{v^1 v^2}$ de P_i pertencente a T_i **:**
    $r^1 \leftarrow$ segundo raio associado a v^1 em S_i
    $r^2 \leftarrow$ primeiro raio associado a v^2 em S_i
    se $(r^1 \times (p - v^1) > 0) \wedge (r^2 \times (p - v^2) < 0) \wedge ((p - v^1) \times (v^2 - v^1) < 0)$ **:**
      retorna região associada a e
  retorna região de atravessar P_i

Algoritmo 6: TPP Irrestrito — Implementação Completa (Primeira e Segunda Abordagens)

Entrada: Pontos s e t, sequência de polígonos convexos disjuntos $(P_1, \ldots, P_k)$
Saída : k-path até t.

define TPP_Irrestrito$(s, t, (P_1, \ldots, P_k))$**:**
  $(T_1, \ldots, T_k) \leftarrow \emptyset$
  $(S_1, \ldots, S_k) \leftarrow \emptyset$
  para cada $i \leftarrow 1$ até k **:**
    $T_i \leftarrow$ RegiãoDePrimeiroContato$(i, s, (P_1, \ldots, P_i), (T_1, \ldots, T_{i-1}), (S_1, \ldots, S_{i-1}))$
    $S_i \leftarrow$ MapaDeÚltimoPasso$(i, s, (P_1, \ldots, P_i), (T_1, \ldots, T_i), (S_1, \ldots, S_{i-1}))$
  retorna ConsultaCaminho$(t, k, s, (P_1, \ldots, P_k), (T_1, \ldots, T_k), (S_1, \ldots, S_k))$

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 5 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 $u^1, \ldots, u^5$. 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

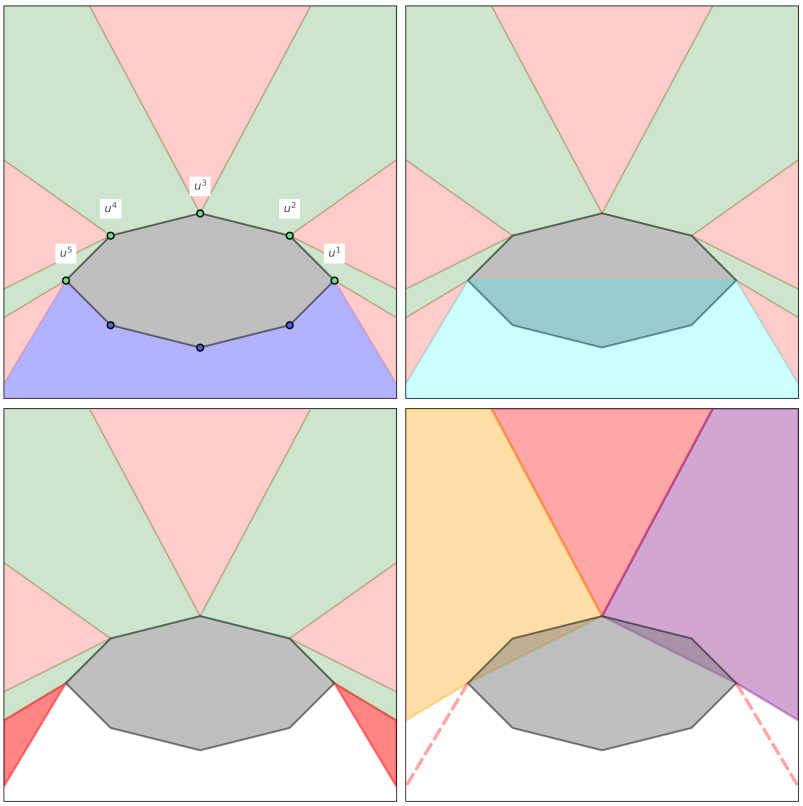


Figura 5: Estratégia de busca binária para localizar um ponto em um mapa de último passo.

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.

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. Selecionamos 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 o processo recursivamente considerando apenas os vértices entre u^1 e u^3. Caso contrário, sabemos que p pertence a R_2 e repetimos o processo considerando apenas os vértices entre u^3 e u^5. Esse procedimento é repetido até que p pertença a uma região de vértice ou restem apenas dois vértices. Nesse último caso, concluímos que p está na região de aresta entre esses 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 v^1 e v^2 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 6.

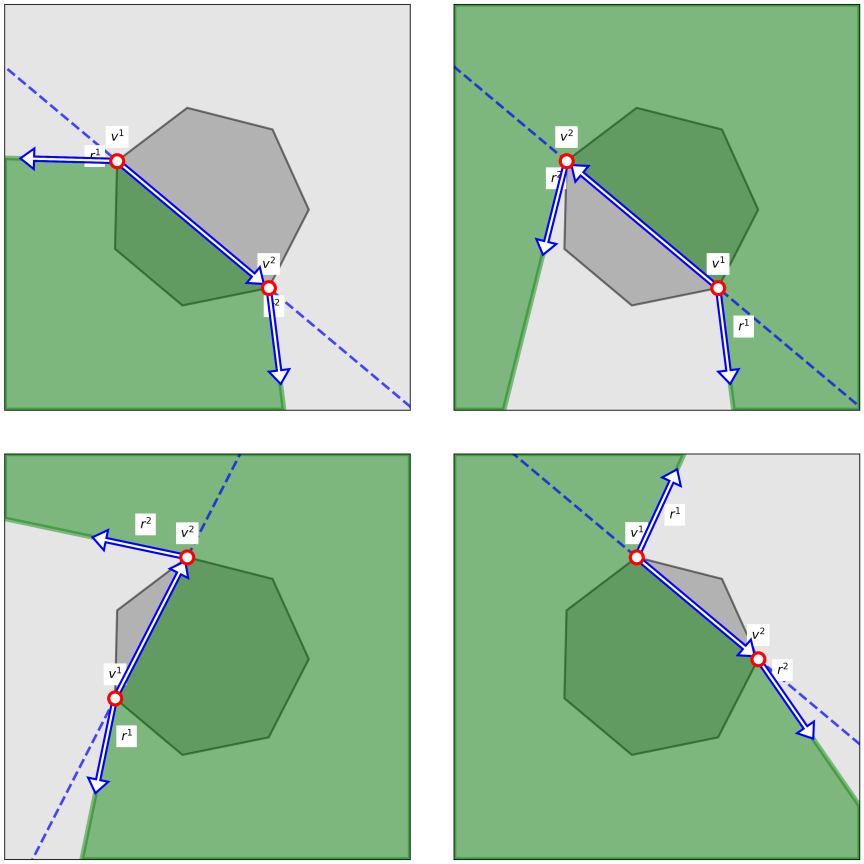


Figura 6: Quatro casos possíveis ao criar uma região de aresta fictícia.

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

- $d \times r^1 < 0 \wedge d \times r^2 < 0$: caso ilustrado no primeiro quadro da Figura 6. Verificamos se p está entre os raios r^1 e r^2 e do lado externo da aresta $\overline{v^1 v^2}$.
- $d \times r^1 > 0 \wedge d \times r^2 > 0$: ilustrado no segundo quadro da Figura 6. 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 $\overline{v^1 v^2}$.
- $d \times r^1 < 0 \wedge d \times r^2 > 0$: ilustrado no terceiro quadro da Figura 6. Inicialmente verificamos de que lado da reta que contém $\overline{v^1 v^2}$ 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.
- $d \times r^1 > 0 \wedge d \times r^2 < 0$: ilustrado no quarto quadro da Figura 6. 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 v^1 e v^2, 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 regiões de vértice verificadas 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.

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(v^1, v^2):
3          r^1 ← segundo raio associado a v^1 em S_i
4          r^2 ← primeiro raio associado a v^2 em S_i
5          d ← v^2 − v^1
6          se d × r^1 < 0 ∧ d × r^2 < 0 :
7              retorna r^1 × (p − v^1) > 0 ∧ r^2 × (p − v^2) < 0 ∧ (p − v^1) × (v^2 − v^1) < 0
8          senão se d × r^1 > 0 ∧ d × r^2 > 0 :
9              retorna r^1 × (p − v^1) ≤ 0 ∧ r^2 × (p − v^2) ≥ 0 ∧ (p − v^1) × (v^2 − v^1) ≥ 0
10         senão se d × r^1 < 0 ∧ d × r^2 > 0 :
11             se (p − v^1) × (v^2 − v^1) < 0 :
12                 retorna r^2 × (p − v^2) < 0
13             senão
14                 retorna r^1 × (p − v^1) > 0
15         senão
16             se (p − v^1) × (v^2 − v^1) < 0 :
17                 retorna r^1 × (p − v^1) > 0
18             senão
19                 retorna r^2 × (p − v^2) < 0
20     u_1, ..., u_m ← vértices de P_i em T_i, ordenados em sentido anti-horário
21     se VerificaAresta(u^m, u^1) :
22         retorna região de atravessar P_i
23     se p está na região de vértice de u^1 :
24         retorna região associada a u^1
25     se p está na região de vértice de u^m :
26         retorna região associada a u^m
27     l ← 1
28     r ← m
29     while l + 1 ≠ r do
30         mid ← ⌊(l + r)/2⌋
31         se p está na região de vértice de u^mid :
32             retorna região associada a u^mid
33         se VerificaAresta(u^l, u^mid) :
34             r ← mid
35         senão
36             l ← mid
37     retorna região associada a aresta u^l u^r
```

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 $e = \overline{v^1 v^2}$ 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, verificamos apenas a aresta e. Calculamos o último passo q do $(i-1)$-path até o vértice v^1 da aresta e e verificamos o sinal de $(q - v^1) \times (v^2 - v^1)$. Se esse valor for negativo, então e pertence a T_i; caso contrário, não pertence.

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 $(q - p)$ 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 vetor $(q - v)$ está do lado externo de e^1 e de e^2, determinando, assim, se essas arestas pertencem a T_i. 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 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 $v^1, \ldots, v^m$ 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 v^1 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 v^1 e v'; caso contrário, repetimos o procedimento considerando apenas os vértices entre v' e v^m.

Esse passo é repetido 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, utilizamos as ideias discutidas anteriormente para determinar, com uma única consulta, 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 $P_1, \ldots, P_k$ 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** e **ConsultaCaminho** (2 e 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** e **MapaDeÚltimoPasso** (3 e 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 $i = 1, \ldots, k$, 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** e **ConsultaCaminho**: De forma análoga ao caso anterior, o pior caso envolve i chamadas ao algoritmo `LocalizaPonto`, resultando em uma complexidade

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

- **RegiãoDePrimeiroContato** e **MapaDeÚltimoPasso**: Cada um desses algoritmos percorre 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**: 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

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

Portanto, 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) [1], 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] M. Dror, A. Efrat, A. Lubiw e J. S. B. Mitchell, "Touring a sequence of polygons," em *Proceedings of the Thirty-Fifth Annual ACM Symposium on Theory of Computing*, sér. STOC '03, San Diego, CA, USA: Association for Computing Machinery, 2003, pp. 473–482, ISBN: 1581136749. DOI: 10.1145/780542.780612. endereço: https://doi.org/10.1145/780542.780612.

[2] J. Liu e H. Liu, "Research on Path Optimization Method for Warehouse Inspection Robot," *Applied Artificial Intelligence*, v. 37, n. 1, 2023. endereço: https://www.tandfonline.com/doi/full/10.1080/08839514.2023.2254048#abstract.

[3] K. J. Obermeyer, "Path Planning for a UAV Performing Reconnaissance of Static Ground Targets in Terrain," *GNC-31: Intelligent Control in Aerospace Applications*, 2009. endereço: https://doi.org/10.2514/6.2009-5888.

[4] E. M. Arkin, S. P. Fekete e J. S. B. Mitchell, "The Traveling Salesman Problem with Neighborhoods: A Survey," em *The Traveling Salesman Problem and Its Variations*, G. Gutin e A. P. Punnen, ed., Boston, MA: Springer, 2005, pp. 377–443.