O Problema de Visita de Polígonos*

Gabriel F. Ushijima† Ernesto G. Birgin†

15 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 visita 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 visita 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 pensar nesse 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 visita um conjunto de regiões arbitrárias no plano, na medida que 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. Finalmente, mostramos que a primeira abordagem tem complexidade $O(n^2)$ e as outras duas $O(nk \log(n/k))$, onde n é o 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 na sua versão mais simples, no qual os polígonos são convexos e não se sobrepõem, chamado de TPP Irrestrito. Usamos tanto a formulação, quanto 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 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 tem como base as definições e propriedades apresentadas por Dror et al. (2003) [1], assumimos que o leitor tem familiaridade com o conteúdo desse artigo. A seguir, retomamos as definições principais 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, que é 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 é uma partição do plano em um número finito de regiões tais que, dois pontos em $\mathbb{R}^2$ pertence 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 temos $S_1, \ldots, S_i$, 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'. Assim, o i-path até p será exatamente o $(i-1)$-path até p', com excessão de que devemos refletir seu último segmento a partir do ponto que ele passa pela 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 quadro ilustra o caso em que o ponto está em uma região de aresta, assim como o processo de reflexão e construção do caminho, e o terceiro quadro apresenta o caso em que o ponto está na região de atravessar o polígono.

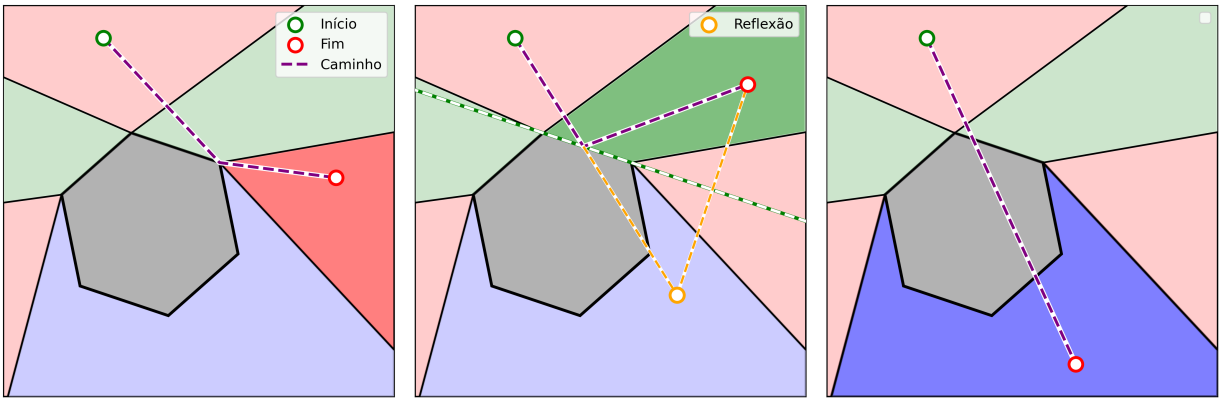


Figura 1: Três casos possíveis para o último segmento de um 1-path até um ponto p.

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 $\mathbb{R}^2$ excluindo o último ponto do caminho p, que é conhecido. Também assumimos que temos 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 o tipo de região (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) // Localiza o ponto p no mapa de último passo 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 que calculemos o i-path até qualquer ponto p de forma eficiente,

podendo ser utilizado para calcular o k-path até t, que é exatamente a solução do problema. No entanto, frequentemente, nos interessamos apenas no último segmento do i-path até p. Ademais, o caso em que um ponto está em uma região associada a um vértice de P_i permite que determinemos o último segmento diretamente, sem precisar calcular o restante do caminho. Assim, 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 que discutiremos 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, vamos calcular 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, apenas precisamos 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 nessa 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.

A Figura 2 ilustra como podemos usar essa ideia para determinar a região de primeiro contato de um polígono, realizando o teste para ambos vértices de cada aresta do polígono. Implementamos essa ideia no Algoritmo 3, usando o vértice v^1 como o ponto p para determinar se a aresta e pertence a T_i.

Ademais, uma vez que temos $T_0, \ldots, T_i$, podemos calcular o mapa de último passo S_i de P_i. Para tal, notamos que S_i tem exatamente três tipos de regiões: regiões associadas a vértices de P_i em T_i, regiões associadas a arestas de P_i em T_i e uma única região de atravessar P_i, que coincide com as arestas que não estão em T_i. Ademais, uma região associada à aresta e é delimitada pelas regiões associadas às pontas de e e pela aresta e em si. Além disso, a região de atravessar P_i é definida como o complemento das regiões associadas a vértices e arestas de P_i. Assim, para construir S_i, precisamos apenas determinar as regiões associadas a vértices de P_i em T_i.

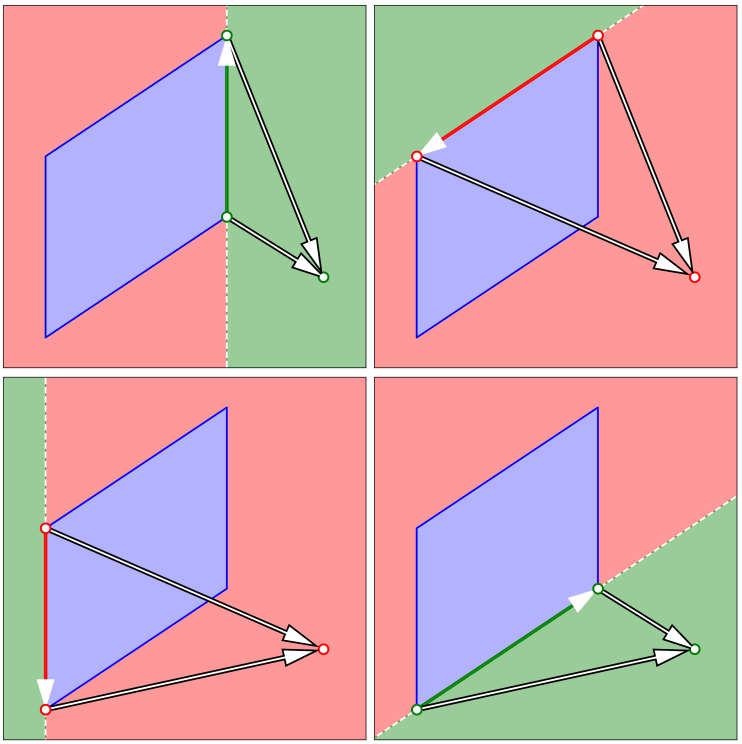


Figura 2: Determinando região de primeiro contato de um polígono.

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 :
8             T_i ← T_i ∪ e
9     retorna T_i
```

Pelas condições de otimalidade discutidas em [1, **Local Optimality Conditions.**], sabemos que para cada vértice v de P_i que está em T_i, a região associada a v em S_i é simplesmente um cone definido pelo ponto v e um par ordenado de raios que partem de v. Podemos calcular esses raios a partir do último segmento do $(i-1)$-path até v e sua reflexão em relação às arestas de P_i que contêm v. Entrando em mais detalhe, 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 faça parte de 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 faça parte de T_i, ou simplesmente o vetor $\overrightarrow{qv}$ caso contrário. Repetimos esse processo para todos os vértices de P_i e assim obtemos todas as regiões associadas a vértices de P_i em S_i. Implementamos essa ideia no Algoritmo 4.

Uma vez que temos T_i e S_i, desejamos implementar o algoritmo `LocalizaPonto` que localiza

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.

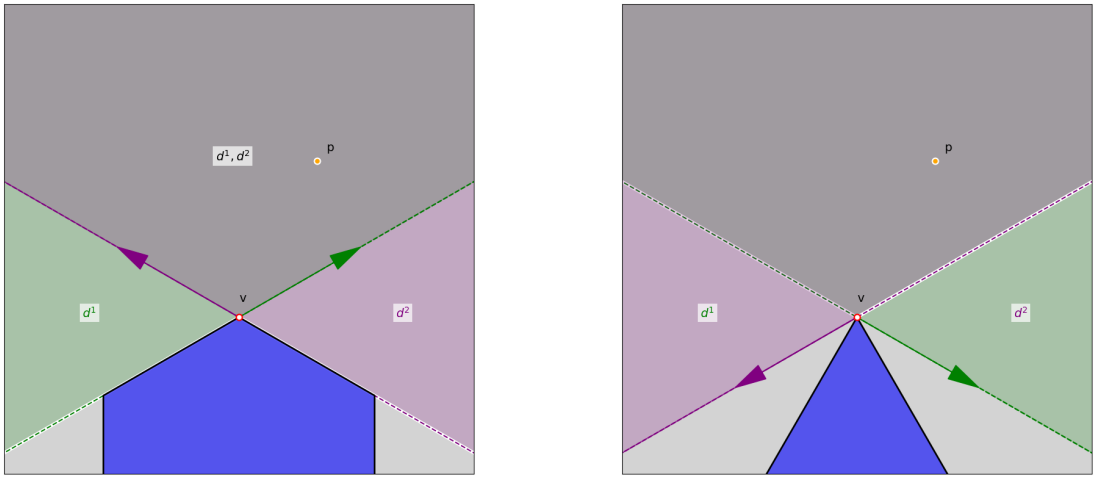
```
1  define MapaDeÚltimoPasso(i, s, (P_1, ..., P_i), (T_1, ..., T_i), (S_1, ..., S_{i-1})):
2      S_i ← ∅
3      para cada vértice v de P_i tal que existe e em T_i incidente a v :
4          e^1 ← aresta anterior de v em P_i
5          e^2 ← aresta posterior de v em P_i
6          q ← ConsultaÚltimoPasso(v, 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
```

um ponto p no mapa de último passo S_i. Na primeira abordagem, implementamos esse procedimento de forma direta, verificando cada região de vértice, então cada região de aresta e caso o ponto não pertença à nenhuma dessas retornamos 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 precisamos determinar 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 ideias são ilustradas na Figura 3.

Ademais, 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 consideramos os raios r^1 como o segundo raio do vértice v^1 e r^2 como o primeiro raio do vértice v^2 em S_i. Então, p pertence à região associada a e ele 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. Fazemos dois loops: o primeiro verifica todas as regiões associadas a vértices de P_i em T_i e o segundo verifica todas as regiões associadas a arestas de P_i em T_i. Caso o ponto p não pertença a nenhuma dessas regiões, retornamos a região de atravessar P_i.

Finalmente, uma vez que sabemos como calcular T_i, S_i e localizar um ponto p em S_i, podemos implementar o algoritmo completo para resolver o problema TPP, conforme mostrado no Algoritmo 6. Esse algoritmo itera sobre todos os polígonos $P_1, \ldots, P_k$, calculando T_i e S_i para cada polígono usando os 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, onde apenas o algoritmo



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

Figura 3: Verificando se um ponto pertence a uma região associada a um vértice em um mapa de último passo.

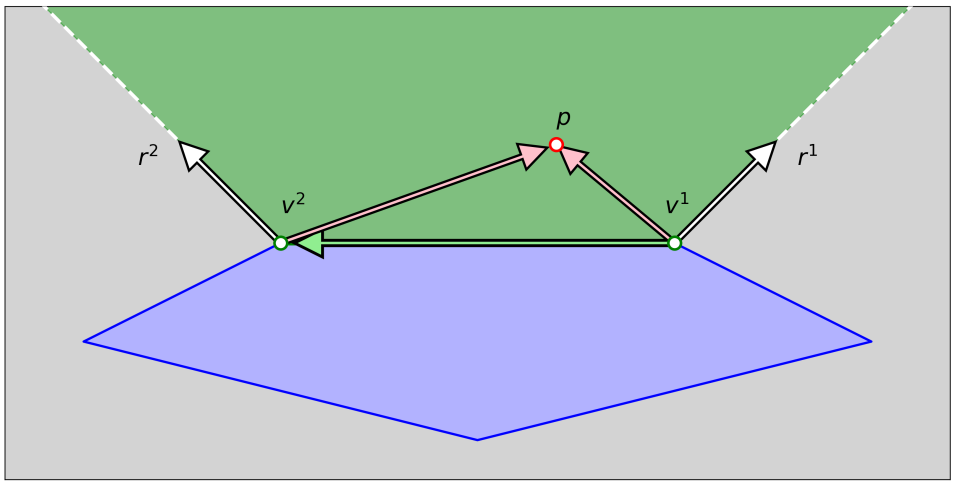


Figura 4: Verificando se um ponto pertence a uma região de aresta.

`LocalizaPonto` será modificado.

2.2 Segunda Abordagem

Na segunda abordagem, vamos alterar 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 desse algoritmo é usar "arestas fictícias"que conectam dois vértices u^i e u^j formando uma nova região R. Essa região tem a propriedade que podemos verificar se p está em R em tempo $O(1)$ e p está em R se e somente se p está em alguma região de vértice ou de aresta entre u^i e u^j. Assim, podemos usar uma estratégia de busca binária para localizar p em S_i.

Devemos mencionar que estamos assumindo que o ponto p não está no interior do polígono P_i. Esse fato é garantido pela suposição de que os polígonos são disjuntos, incluindo s e t. No entanto, caso desejemos lidar com o caso em que p pode estar no interior de P_i, podemos iniciar o algoritmo verificando se p está no interior de P_i em tempo $O(\log(|P_i|))$, embora não discutiremos essa verificação aqui, pois não é o foco deste trabalho.

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 e em T_i incidente a v **:**
    $(r^1, r^2) \leftarrow$ raios associados a v em S_i
    se $r^1 \times r^2 \geq 0$ **:**
      retorna $(r^1 \times (p - v) > 0) \wedge (r^2 \times (p - v) < 0)$
    senão
      retorna $(r^1 \times (p - v) > 0) \vee (r^2 \times (p - v) < 0)$
  para cada aresta $e = \overline{v^1 v^2}$ de P_i em 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$(t, k, s, (P_1, \ldots, P_k), (T_1, \ldots, T_k), (S_1, \ldots, S_k))$

A Figura 5 será usada para descrever cada etapa do algoritmo a seguir. O primeiro quadro mostra um polígono P_i de 8 vértices, dos quais 5 pertencem a T_i, marcados como $u^1, \ldots, u^5$, também desenhamos as regiões de vértices em vermelho, regiões de aresta em verde a região de travessia em azul, vamos localizar um ponto nesse mapa de último passo. O algoritmo em si tem 2 passos iniciais e então inicia a busca binária, da maneira que descrevemos 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. Podemos verificar se p está em R usando a mesma estratégia que usamos no 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. Se p estiver em 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): Usamos a

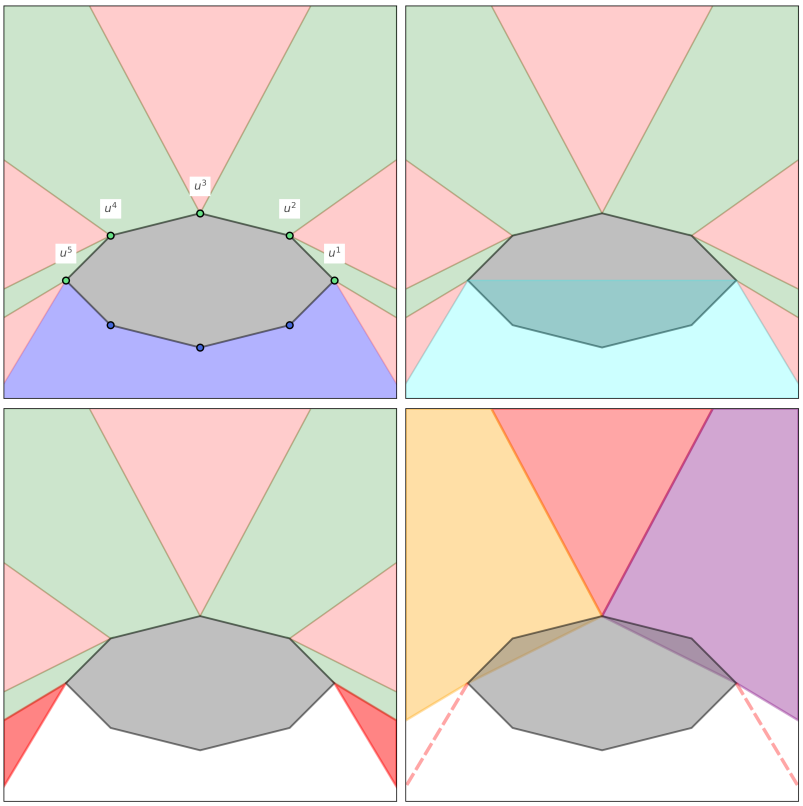


Figura 5: Estratégia de busca binária para localizar um ponto em um mapa de último passo.

mesma estratégia do Algoritmo 5 para verificar se p está em alguma das regiões de vértice de u^1 ou u^5, caso esteja, retornamos a região correspondente.

Passo 3. Busca Binária (Quadro 4): Agora sabemos que p não está na região de travessia nem nas regiões de vértice de u^1 ou u^5, assim, podemos iniciar a busca binária entre os vértices u^1 e u^5. Para tal, selecionamos u^3 como um vértice intermediário e então verificamos se p está na região de vértice de u^3, se sim, 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 está em R_1, se sim, repetimos o processo recursivamente considerando apenas os vértices entre u^1 e u^3. Caso contrário, sabemos que p está em R_2 e repetimos o processo recursivamente considerando apenas os vértices entre u^3 e u^5. Repetimos esse passo até que p esteja em uma região de vértice ou reste apenas dois vértices, a esse ponto, sabemos que p deve estar na região de aresta entre esses vértices.

A maioria desse algoritmo é direta e fácil de ser implementada, exceto pela parte de verificar se p está na região R_1, uma vez que ela pode ter uma forma que não é coberta pela nossa verificação original do Algoritmo 5, pois quando criamos uma aresta fictícia entre dois vértices v^1 e v^2 com raios r^1 e r^2, criamos a possibilidade de que r^1 e r^2 formem um ângulo maior que 180°, ou estejam em lados diferentes da aresta fictícia, como ilustrado na Figura 6.

Vamos dizer que queremos determinar se um ponto p pertence à região de aresta fictícia entre os vértices v^1 e v^2 delimitados pelos raios r^1 partindo de v^1 e r^2 partindo de v^2. Os casos são determinados pelo "lado"que r^1 e r^2 se encontram em relação à direção $d = v^2 - v^1$, tratamos cada caso separadamente:

- $d \times r^1 < 0 \wedge d \times r^2 < 0$: Esse caso é o ilustrado no primeiro quadro da Figura 6 e é o caso que estamos acostumados a lidar. Nesse caso, simplesmente verificamos se p está entre os raios

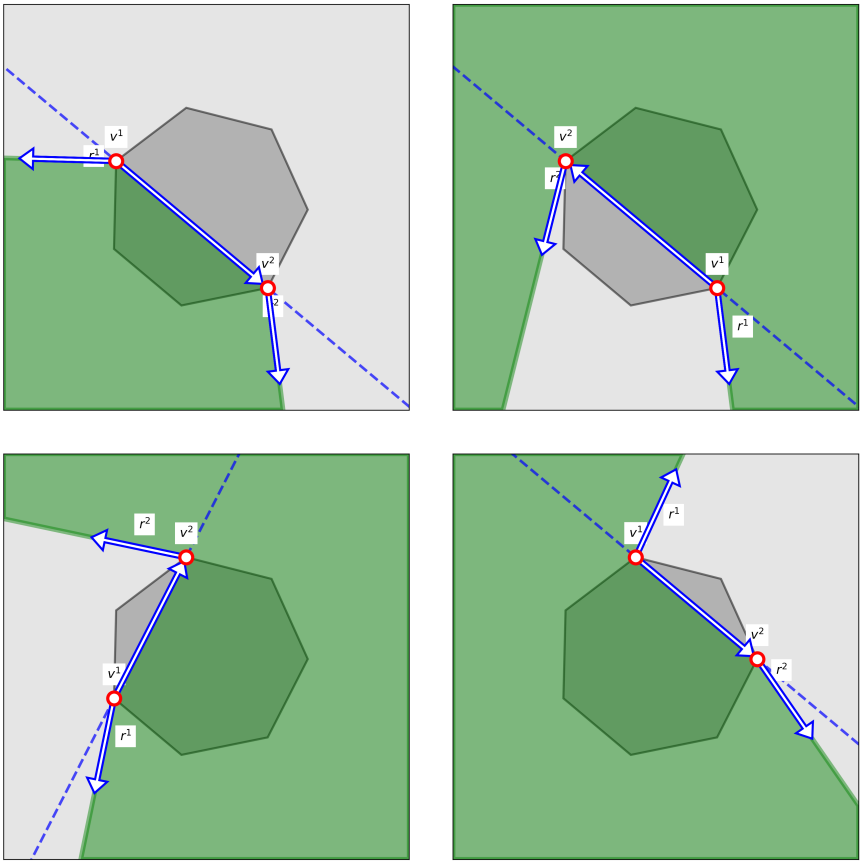


Figura 6: Quatro casos possíveis ao criar uma região de aresta fictícia.

r^1 e r^2 e do lado externo da aresta $\overline{v^1v^2}$.

- $d \times r^1 > 0 \wedge d \times r^2 > 0$: Esse caso é o ilustrado no segundo quadro da Figura 6 e é o complemento do caso anterior. Nesse caso, verificamos se p não está entre os raios r^1 e r^2 **e** está do lado interno da aresta $\overline{v^1v^2}$.

- $d \times r^1 < 0 \wedge d \times r^2 > 0$: Esse caso é o ilustrado no terceiro quadro da Figura 6. Para esse caso, precisamos primeiro verificar de qual lado da reta que contém $\overline{v^1v^2}$ o ponto p está. Se p estiver do lado externo, então verificamos se p está do lado negativo de r^2, caso contrário, verificamos se p está do lado positivo de r^1.

- $d \times r^1 > 0 \wedge d \times r^2 < 0$: Esse caso é o ilustrado no quarto quadro da Figura 6. Para esse caso, precisamos primeiro verificar de qual lado da reta que contém $\overline{v^1v^2}$ o ponto p está. Se p estiver do lado externo, então verificamos se p está do lado positivo de r^1, caso contrário, verificamos se p está do lado negativo de r^2.

Com essas verificações em mente, podemos implementar o algoritmo `LocalizaPonto` usando a estratégia de busca binária, conforme mostrado no Algoritmo 7. Nele, implementamos a função auxiliar `VerificaAresta` que verifica se p está na região de aresta fictícia entre dois vértices v^1 e v^2 usando as verificações discutidas anteriormente. Também abstraimos a verificação se p está na região de vértice de um dado vértice u^{mid} em uma verificação direta, uma vez que essa verificação é idêntica àquela usada no Algoritmo 5.

A vantagem dessa abordagem é que a busca binária reduz o número de regiões de vértice que precisamos verificar para localizar p em S_i de $O(|T_i|)$ para $O(\log(|T_i|))$, o que pode ser uma melhoria significativa quando P_i tem muitos vértices em T_i. Vamos analizar o efeito dessa mudança adiante 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, buscamos trazer uma motivação para essa nova implementação do algoritmo `LocalizaPonto`. Considere o caso de um único polígono P_1 com muitos

vértices, um milhão, por exemplo, nossa segunda abordagem calcula T_1, então S_1 e finalmente faz uma única consulta ao mapa de último passo S_1 para localizar o ponto t. No entanto, essa consulta usa no máximo 20 regiões de vértice de P_1 para localizar t em S_1, assim, calculamos 999980 regiões de vértice de P_1 que nunca são utilizadas. Portanto, na terceira abordagem, buscamos utilizar uma estratégia de memoização para calcular apenas as regiões de vértice de P_i que são necessárias para localizar os pontos consultados no mapa de último passo S_i.

O primeiro problema que precisamos resolver é como determinar se uma dada aresta $e = \overline{v^1v^2}$ de P_i está T_i fazendo apenas uma única chamada ao algoritmo ConsultaÚltimoPasso. Para isso, podemos simplesmente reutilizar a ideia do Algoritmo 3 que calculava T_i por completo, mas agora, ao invés de verificar todas as arestas de P_i, verificamos apenas a aresta e. Assim, calculamos o último passo q do i-path até o ponto médio v^1 da aresta e e verificamos se $(q - v^1) \times (v^2 - v^1)$ é negativo, se for, então e pertence a T_i, caso contrário, não pertence.

A seguir, precisamos calcular os raios associados a um vértice v de P_i em S_i, novamente fazendo apenas uma única chamada ao algoritmo ConsultaÚltimoPasso. Para isso, reutilizamos a ideia do Algoritmo 4 que calculava S_i por completo, mas agora, ao invés de calcular os raios para todos os vértices de P_i, calculamos apenas os raios para o vértice v. No entanto, esse algoritmo assumia que sabemos quais arestas de P_i pertencem a T_i, então precisamos saber se as arestas incidentes em v pertencem a T_i ou não sem fazer chamadas adicionais ao algoritmo ConsultaÚltimoPasso.

Para isso, retomamos a ideia de se e é uma aresta de P_i, p um ponto qualquer em e e q o último passo do $(i-1)$-path até p, então $e \in T_i$ se e somente se $(q - p)$ está do lado externo de e. Assim, seja v um vértice de P_i e e^1 e e^2 as arestas incidentes em v, podemos calcular o último passo q do $(i-1)$-path até v e então verificar se $(q - v)$ está do lado externo de e^1 e e^2 para determinar se essas arestas pertencem a T_i. Com essa informação, podemos facilmente adaptar o Algoritmo 4 para calcular apenas os raios associados a v em S_i.

Uma última observação importante é que como não sabemos T_i, pode ser que tentemos calcular os raios associados a um vértice v de P_i em S_i, mas nenhuma das arestas incidentes em v pertença a T_i. Nesse caso, não precisamos alterar o algoritmo, uma vez que seu comportamente natural será retornar dois raios cuja direção é igual ao raio que chega em v no $(i-1)$-path, o que representa uma região vazia em S_i, o que é condizente com um vértice que não pertence a T_i.

Finalmente, podemos descrever como implementamos o algoritmo `LocalizaPonto` usando essa estratégia de memoização. Vamos reutilizar a ideia de busca binária da segunda abordagem, mas agora nosso passos iniciais serão simplificados, uma vez que delegamos a parte de verificar a região de travessia para depois da busca binária. O algoritmo 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. Calcule os raios associados a v^1 em S_i e verifique se p está na região de vértice de v^1, se sim, retorne essa região.

Passo 2. Seja $v' = v^{\lfloor m/2 \rfloor}$, calcule os raios associados a v' em S_i e verifique se p está na região de vértice de v', se sim, retorne essa região.

Caso contrário, crie as arestas fictícias entre v^1 e v' e entre v' e v^m, gerando as regiões R_1 e R_2. Verifique se p está em R_1 usando o mesmo procedimento do Algoritmo 7, se sim, repita esse passo considerando apenas os vértices entre v^1 e v', caso contrário, repita esse passo considerando apenas os vértices entre v' e v^m.

Esse passo é repetido até que encontremos um vértice v^k tal que p está na região de vértice de v^k ou restem apenas dois vértices v^l e v^r, nesse ponto, 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, concluimos que ele deve

> estar na região de aresta entre os dois últimos vértices v^l e v^r considerados. Para finalizar, usamos as ideia discutidas acima para determinar se essa aresta está em T_i fazendo apenas uma consulta e então retornamos a região de aresta correspondente, caso contrário, retornamos a região de travessia de P_i.

Dessa forma, concluimos a descrição da terceira abordagem para implementar o algoritmo `LocalizaPonto` usando memoização. Decidimos não incluir o pseudocódigo completo dessa abordagem, uma vez que ele seria bastante semelhante ao Algoritmo 7, com as modificações discutidas acima.

2.4 Análise de Complexidade

Primeiramente, vamos analizar a complexidade da primeira abordagem. Vamos dizer que $|P_i|$ é o número de vértices do polígono P_i e $|T_i|$ é o número de arestas em T_i. Também vamos dizer que k é o número total de polígonos na sequência $P_1, \ldots, P_k$ e n é o número total de vértices em todos os polígonos, ou seja, $n = \sum_{i=1}^{k} |P_i|$.

- LocalizaPonto 5: A complexidade desse algoritmo é $O(|T_i|)$, pois no pior caso, precisamos verificar todas as regiões de vértice e de aresta em T_i para localizar o ponto p.
- ConsultaÚltimoPasso e ConsultaCaminho 2 e 1: Ambos os algoritmos começam fazendo uma chamada ao algoritmo `LocalizaPonto`, o pior caso ocorre quando o ponto p está na região de atravessar P_i ou em uma região de aresta, o que exige outra consulta ao mapa de último passo S_{i-1}, resultando em um total de i chamadas ao algoritmo `LocalizaPonto`. Portanto, a complexidade desses algoritmos é $O(\sum_{j=1}^{i} |T_j|)$.
- RegiãoDePrimeiroContato e MapaDeÚltimoPasso 3 e 4: Ambos os algoritmos fazem uma única passagem pelos vértices de P_i, fazendo uma consulta para cada vértice. Portanto, a complexidade desses algoritmos é $O(|P_i| \sum_{j=1}^{i} |T_j|)$.
- TPP completo 6: O algoritmo faz uma chamada para calcular T_i e S_i para cada i de 1 a k, e finalmente faz uma chamada para consultar o caminho até t. Portanto, a complexidade total do algoritmo TPP:

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

Dessa forma, concluimos que a complexidade total do algoritmo TPP usando a primeira abordagem é $O(n^2)$. A seguir, vamos analizar a complexidade da segunda abordagem.

- LocalizaPonto 5: A complexidade desse algoritmo é $O(\log(|T_i|))$, que é o pior caso da busca binária que fazemos para localizar o ponto p em S_i.

- ConsultaÚltimoPasso e ConsultaCaminho 2 e 1: Ambos os algoritmos começam fazendo uma chamada ao algoritmo `LocalizaPonto`, o pior caso ocorre quando o ponto p está na região de atravessar P_i ou em uma região de aresta, o que exige outra consulta ao mapa de último passo S_{i-1}, resultando em um total de i chamadas ao algoritmo `LocalizaPonto`. Portanto, a complexidade desses algoritmos é $O(\sum_{j=1}^{i} \log(|T_j|))$.

- RegiãoDePrimeiroContato e MapaDeÚltimoPasso 3 e 4: Ambos os algoritmos fazem uma única passagem pelos vértices de P_i, fazendo uma consulta para cada vértice. Portanto, a complexidade desses algoritmos é $O(|P_i| \sum_{j=1}^{i} \log(|T_j|))$.

- TPP completo 6: O algoritmo faz uma chamada para calcular T_i e S_i para cada i de 1 a k, e finalmente faz uma chamada para consultar o caminho até t. Portanto, a complexidade total do algoritmo TPP:

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

  Considerando que a soma $\sum_{i=1}^{k} \log(|P_j|)$ é maximizada quando todos os polígonos têm o mesmo número de vértices, ou seja, $|P_j| = n/k$ para todo j, temos que a complexidade final do algoritmo TPP usando a segunda abordagem é $O(nk \log(n/k))$.

Dessa forma, temos que a complexidade final do algoritmo TPP usando a segunda abordagem é $O(nk \log(n/k))$. Essa complexidade é exatamente a indicada por Dror et al. (2003) [1], assim, concluímos que nossa implementação é tão eficiente quanto a proposta original.

Finalmente, resta analizar a complexidade da terceira abordagem. No entanto, a análise dessa abordagem é mais complexa, uma vez que a complexidade do algoritmo `LocalizaPonto` depende muito da entrada específica, ou seja, da distribuição dos vértices nos polígonos e das consultas feitas ao mapa de último passo. No entanto, é fácil notar que a terceira abordagem nunca será pior do que a segunda abordagem, uma vez que no pior caso, ela calculará todas as regiões de vértice em T_i, resultando em uma complexidade igual à da segunda abordagem. Portanto, podemos concluir que a complexidade da terceira abordagem é $O(nk \log(n/k))$ no pior caso, mas pode ser melhor em casos específicos dependendo das consultas feitas.

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.