UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

O Problema da Visita de Polígonos

Gabriel Freire Ushijima

São Paulo, SP
5 de novembro de 2025

Sumário

1 Introdução

Este relatório descreve a implementação e os resultados obtidos na resolução do problema da visita de polígonos, usando como base o paper de Dror et al. (2003) [1] que descreve algoritmos para o caso sem e com restrições. Buscamos apresentar uma abordagem mais prática e detalhada para o problema, sem um foco tão grande na análise teórica.

2 O Problema de Visita de Polígonos Irrestrito

Vamos tomar como base a implementação do algoritmo de Mitchell para o problema irrestrito que fizemos em *Python*. O código pode ser encontrado no arquivo `TouringPolygons/problem1.py`.

2.1 Definições e Notação

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

A figura ao lado 1 ilustra um exemplo de entrada e a solução ótima para o problema para um caso com 3 polígonos. Temos s como o ponto verde, t como o ponto vermelho e os polígonos P_1, P_2 e P_3 como o triângulo azul, o trapézio laranja e o pentágono verde, respectivamente. O caminho mínimo é representado pela linha roxa.

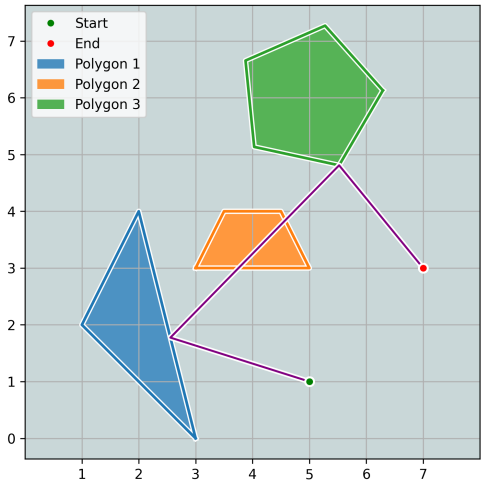

Figura 1: Caminho mínimo para um caso de 3 polígonos.

Retomando o paper, definimos um i-path até p como um caminho mínimo que começa em s, encosta em cada um dos polígonos $P_1, P_2, \ldots, P_i$ e termina em p. Note que nosso objetivo é encontrar um k-path até t. Enquanto não vamos entrar em detalhes, todo i-path é único.

A função central desse algoritmo será a função Query(p, i), que recebe um ponto p e um índice i e retorna o penúltimo ponto q do i-path até p (ou seja, o ponto imediatamente anterior à p nesse caminho).

Primeiramente, vamos descrever procedimentos auxiliares que serão úteis para a implementação da função Query. Finalmente, vamos descrever como respoder consultas usando esses procedimentos. Por enquanto, vamos assumir que sabemos como respoder as consultas.

2.2 Representação dos Objetos

Antes de descrever os procedimentos auxiliares, é importante definir como representamos os objetos geométricos envolvidos no problema. Utilizamos as seguintes representações:

- **Pontos** e **Vetores**: Representados por uma classe `Vector2` que guarda um par de números reais (x, y).
- **Polígonos**: Representados por uma classe `Polygon2` que guarda uma lista ordenada de seus vértices (representados por `Vector2`) no sentido anti-horário. Um vértice arbitrário é escolhido para ser indexado como o primeiro.
- **Caminho Final**: Representado como uma lista de pontos (representados por `Vector2`) na ordem em que aparecem no caminho.

2.3 Particionando o Plano

É possível mostrar [1] que para cada polígono P_i podemos criar uma partição S_i do plano tal que o comportamento da função Query(p, i) depende exclusivamente de qual região da partição S_i o ponto p pertence, sendo possível localizar cada ponto de maneira eficiente. Por falta de um nome melhor, chamamos cada parte de S_i de *região*.

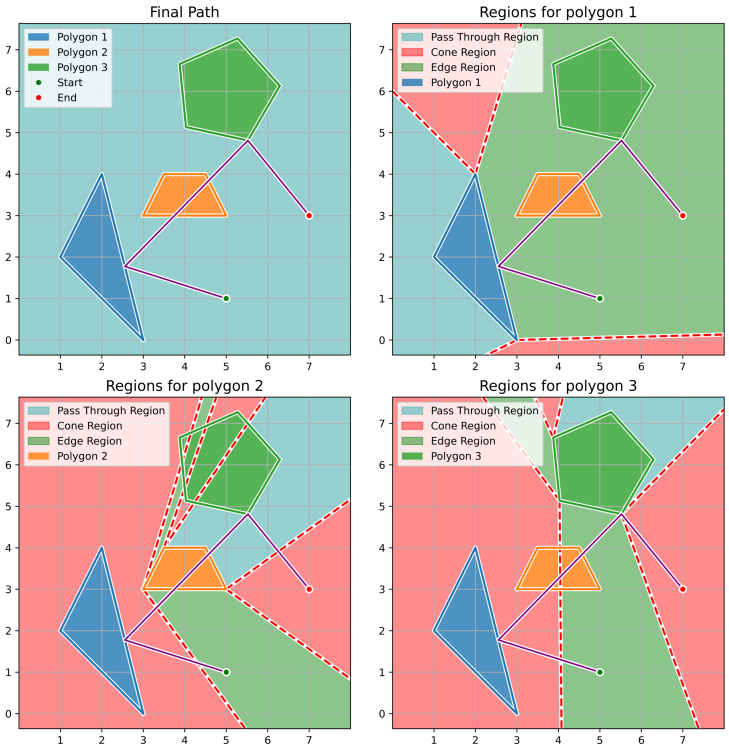

Figura 2: Partição do plano para cada polígono do exemplo anterior.

Enquanto a quantidade de regiões de S_i pode ser numerosa a depender do número de vértices de P_i, cada região pode ser de exatamente 3 tipos diferentes:, Regiões de Vértice, Regiões de Aresta e Regiões de Atravessar.

Nas seções a seguir vamos descrever como usar essas partições para responder consultas, mas por enquanto vamos descrever como representar e construir essas partições. Por enquanto, apenas mantenha em mente que se conseguirmos representar e construir essas partições, então podemos usá-las para responder consultas e resolver o problema de maneira eficiente.

A partir da ilustração 2, note que cada região de vértice (*Cone Region*) está associada a um vértice de P_i, cada região de aresta (*Edge Region*) está associada a uma aresta de P_i e há exatamente uma região de atravessar (*Pass Through Region*). Ademais, essas regiões se complementam, ou seja, o conjunto de todas as regiões de vértice, aresta e atravessar é o plano todo e nenhuma região se sobrepõe a outra. Para que isso seja válido, devemos considerar pontos dentro dos polígonos P_i como pertencentes à região de atravessar.

Ademais, note que cada região de vértice é delimitada por duas semi-retas que partem de um vértice de P_i, incluindo todos os pontos que estão entre essas semi-retas. Similarmente, cada região de aresta é delimitada por duas semi-retas que partem dos extremos de uma aresta de P_i e a prórpia aresta, incluindo todos os pontos que estão entre essas semi-retas e a aresta. Finalmente, a região de atravessar é o complemento das regiões de vértice e aresta.

2.3.1 Representado as Partições

Como mencionado, as regiões se complementam, assim, do ponto de vista de implementação, decidimos armazenar apenas as regiões de vértice, uma vez que as regiões de aresta podem ser construídas a partir delas, simplesmente verificando se um ponto está entre as semi-retas de duas regiões de vértice adjacentes e a aresta que as separa.

Para representar as regiões de vértice, armazenamos uma lista de pares de direções (d_1, d_2) para cada vértice v de P_i, onde d_1 e d_2 são vetores unitários que representam as direções das semi-retas que delimitam a região de vértice associada a v. Nem todo vértice de P_i gera uma região de vértice, assim, se um vértice v não gerar uma região, então armazenamos um par (d, d) qualquer na posição correspondente da lista, representando uma região válida, mas vazia. Isso será mais relevante adiante.

Essa representação já permite que verifiquemos se um ponto p pertence a uma região de vértice e qual. No entanto, não conseguimos distinguir entre regiões de aresta e a região de atravessar. Para resolver esse problema, armazenamos uma lista de booleanas que indica se uma aresta é visível ou não. Assim, para determinar se um ponto p pertence a uma região de aresta ou à região de atravessar, basta verificar se p está entre as semi-retas de duas regiões de vértice adjacentes e, em caso positivo, verificar se a aresta que as separa é visível ou não.

2.3.2 Detectando Arestas Visíveis

Seja e um aresta de P_i e u, v seus vértices na ordem que aparecem em P_i (antihorário). Dizemos que e é visível se e somente se para todo ponto interno p de e o i-path até p não passa pela parte interna de P_i. Como a borda de P_i e sua parte interna são contínuas, podemos fazer esse teste para um único ponto interno de e e teremos o mesmo resultado.

Assim, temos a intuição de pegar o ponto médio m de e ou qualquer outro ponto interno de e, calcular o último passo q do i-path até m e verificar se o segmento $\overline{qm}$ passa por P_i. No entanto, uma abordagem ingênua, testando cada aresta de P_i individualmente, seria muito lenta, tomando tempo $O(|P_i|^2)$.

Felizmente, não é necessário implementar um teste eficiente de interseção de segmento e polígono, uma vez que temos a informação adicional que m pertence à borda de P_i, mais especificamente na aresta e. Por esse motivo, como P_i é convexo e suas arestas estão ordenadas em sentido antihorário, sabemos que se o produto interno de $\overrightarrow{uv}$ com $\overrightarrow{mq}$ for positivo, então $\overrightarrow{mq}$ chega em m pela parte interna de P_i.

Algorithm 1: ArestasVisíveis(P_i, i)
Determina as arestas visíveis de P_i

```
n ← |P_i| // Número de vértices de P_i
visível ← [False] * n // Inicializa todas as arestas como não visíveis
for j ← 0 até n − 1 :
    u ← P_i[j] // j-ésimo vértice de P_i
    v ← P_i[j + 1] // vértice que segue u, considerando P_i[n] = P_i[0]
    m ← (u + v)/2 // ponto médio
    q ← Query(m, i − 1) // penúltimo ponto do (i − 1)-path até m

    // Marca a aresta j como visível se o produto vetorial for negativo
    visível[j] ← ((v − u) × (q − u) < 0)
return visível
```

2.3.3 Construindo Regiões de Vértice

Uma vez que sabemos quais arestas são visíveis, podemos construir as regiões de vértice. Vamos considerar o vértice v de P_i e calculamos $p = \text{Query}(v, i-1)$. Vamos explicar posteriormente como calcular p, mas uma intuição inicial é pensar que $\text{Query}(q, 0) = s$ para qualquer q, uma vez que um 0-path não precisa tocar em nenhum polígono, assim, o menor caminho é uma linha reta de s até q.

Agora consideramos a direção de $d = v - p$ e a maneira que esse raio chega em v. Sejam v_1 o vértice que precede e v_2 o que segue v em P_i (no sentido anti-horário). Se a aresta (v_1, v) é visível, então d_1 será a reflexão de d em relação à aresta (v_1, v), caso contrário, d_1 será a mesma direção de d. Similarmente, se a aresta (v, v_2) é visível, então d_2 será a reflexão de d em relação à aresta (v, v_2), caso contrário, d_2 será a mesma direção de d. Assim, criamos a região de vértice (d_1, d_2) associada à v.

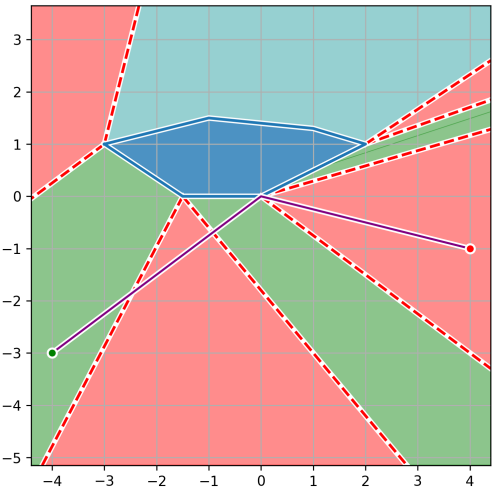

Figura 3: Partição de um polígono com múltiplas arestas não visíveis.

Encorajamos o leitor a verificar que essa lógica se aplica na imagem ao lado 3, usando o ponto inicial s como resposta de $\text{Query}(v, 0)$ para qualquer vértice v. Os vértices (-2, 0) e (0, 0) terão ambas as arestas visíveis, assim, suas regiões de vértice serão delimitadas pelas reflexões do raio que parte de s. O vértice (-3, 1) terá apenas a aresta posterior visível, assim, sua região de vértice será delimitada pela reflexão do raio em relação à aresta posterior e a própria direção do raio. Similarmente, o vértice (2, 1) tem apenas a aresta anterior visível. Finalmente, os vértices (1, 1.25) e (-1, 1.5) não terão nenhuma aresta visível, assim, suas regiões de vértice serão simplesmente a direção do raio em ambas as semi-retas.

Algorithm 2: RefleteDireção(d, v_1, v_2)
Reflete a direção d em relação à aresta (v_1, v_2)

```
e ← v_2 − v_1
N ← Vector2(−e.y, e.x)/|e| // Vetor normal à aresta
return d − 2 · (d^T · N)N // Fórmula de reflexão
```

Algorithm 3: Particiona($P_1, \ldots, P_k$)
Constroi as partições dos polígonos

```
1  regiõesDeVértice ← [] // Lista vazia
2  for i ← 1 até k :
3      visível ← ArestasVisíveis(P_i, i)
4      regiõesDeVértice.append([]) // Adiciona uma nova lista para as regiões de vértice de P_i
5      for j ← 0 até |P_i| − 1 :
6          v ← P_i[j] // j-ésimo vértice de P_i
7          v_1 ← P_i[j − 1] // vértice que precede v, considerando P_i[−1] = P_i[|P_i| − 1]
8          v_2 ← P_i[j + 1] // vértice que segue v, considerando P_i[|P_i|] = P_i[0]
9
10         p ← Query(v, i − 1) // penúltimo ponto do (i − 1)-path até v
11         d ← v − p // direção do raio de p até v
12
13         if visível[j − 1] :
14             d_1 ← Reflete(d, v_1, v) // reflexão de d em relação à aresta (v_1, v)
15         else
16             d_1 ← d
17         if visível[j] :
18             d_2 ← Reflete(d, v, v_2) // reflexão de d em relação à aresta (v, v_2)
19         else
20             d_2 ← d
21         // Armazena a região de vértice associada a v como o par (d_1, d_2)
22         regiõesDeVértice[i - 1].append((d_1, d_2))
```

2.4 Localizando Pontos na Partição

Gostariamos de responder consultas do tipo Query(p, i) de maneira eficiente. Para isso, precisamos ser capazes de localizar o ponto p na partição S_i de maneira eficiente, ou seja, determinar se p pertence a uma região de vértice, região de aresta ou região de atravessar e, em caso positivo, qual região.

Esse processo não é trivial, principalmente se desejamos fazer isso de maneira eficiente. Para tal, primeiro vamos considerar dois problemas mais simples: determinar se p pertence a uma região de vértice e determinar se p pertence a uma região de aresta.

2.4.1 Verificando Regiões de Vértice

Primeiramente vamos implementar um procedimento auxiliar que verifica se um ponto p está em uma região de vértice associada a um vértice v delimitada por duas direções d_1 e d_2. Basicamente, se o ângulo entre d_1 e d_2 é menor ou igual a 180°, então verificamos se p está entre as semi-retas usando produtos vetoriais. Caso contrário, verificamos se p não está no complemento da região.

Algorithm 4: PontoEmVértice(p, v, d_1, d_2)
Verifica se o ponto p está na região de vértice associada ao vértice v delimitada pelas direções d_1 e d_2

```
1  // Se o ângulo da região de vértice é maior que 180° // verificamos se p não pertence ao
       complemento da região.
2  if d_1 × d_2 < 0 :
3      return ¬ PontoEmVértice(p, v, d_2, d_1) // Troca as direções e inverte o resultado
4  // Se o ângulo da região de vértice é menor ou igual a 180°
5  // Verificamos se p está entre as semi-retas
6  return (d_1 × (p − v) ≥ 0) ∧ ((p − v) × d_2 ≥ 0)
```

2.4.2 Verificando Regiões de Aresta

Uma vez que conseguimos verificar se um ponto pertence a uma região de vértice, podemos usar essa funcionalidade para verificar se um ponto pertence a uma região de aresta.

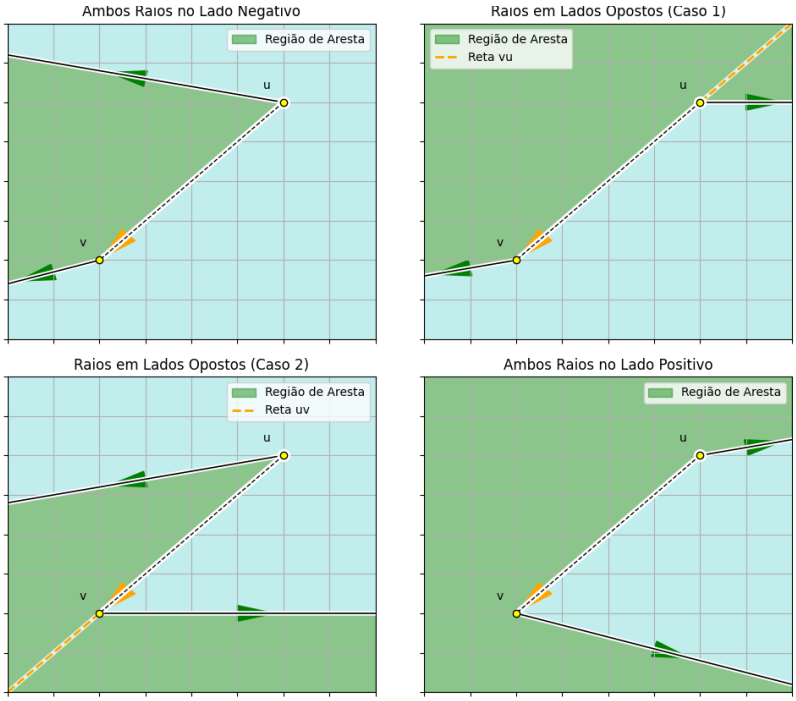

Figura 4: Casos possíveis para região de aresta.

Esse problema é mais complexo, uma vez que existem diferentes casos possíveis para uma região de aresta, dependendo do ângulo entre as direções d_1 e d_2. Esses casos não aparecem explicitamente na partição S_i, na verdade apenas o primeiro caso aparece. No entanto, vamos usar esses outros casos adicionais para otimizar a localização de pontos na partição posteriormente, permitindo complexidade $\log(|P_i|)$ para detecção.

Essencialmente existem 4 casos possíveis para uma região de aresta, dependendo do ângulo entre as direções d_1 e d_2. A figura ao lado 4 ilustra esses casos. Para sabermos com qual caso estamos lidando, definimos $e = v - u$ e calculamos os produtos vetoriais $c_1 = e \times d_1$ e $c_2 = e \times d_2$. Agora, temos os seguintes casos:

- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **ambas arestas no lado negativo**, que é o caso tradicional que aparece na partição S_i. Nesse caso, verificamos se p está entre as semi-retas e a aresta e usando produtos vetoriais.
- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **aresta anterior no lado negativo e aresta posterior no lado positivo**, o *Caso 1* na figura. Nesse caso, dividimos a região de aresta em duas regiões de vértice, definidas por $(u, d_1, u - v)$ e $(v, u - v, d_2)$, então verificamos se o ponto está em qualquer uma das duas.
- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **aresta anterior no lado positivo e aresta posterior no lado negativo**, o *Caso 2* na figura. Nesse caso, dividimos a região de aresta em duas regiões de vértice, definidas por $(u, d_1, v - u)$ e $(v, v - u, d_2)$, então verificamos se o ponto está em qualquer uma das duas.
- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **ambas arestas no lado positivo**. Nesse caso, simplesmente verificamos se o ponto não pertence ao complemento da região, ou seja, verificamos se p não pertence à região de aresta definida por (v, u, d_2, d_1).

Dessa forma, conseguimos verificar se um ponto pertence a uma região de aresta usando o procedimento de verificação de regiões de vértice. Agora podemos implementar o procedimento completo na forma de um algoritmo.

2.4.3 Localização Eficiente de Pontos na Partição

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

Por esse motivo, desejamos explorar a ideia de busca binária para resolver o problema. Primeiramente, vamos assumir que o ponto não pertence ao polígono P_i, isso vale como suposição inicial dos polígonos serem

Algorithm 5: PontoEmAresta(p, u, v, d_1, d_2)
Verifica se o ponto p está na região de aresta associada à aresta $e = (u, v)$ delimitada pelas direções d_1 e d_2

```
// Se vértices são muito próximos, tratamos a região de aresta
// como uma região de vértice.
if |u − v| < 10^−8 :
    return PontoEmVértice(p, v, d_2, d_1)
e ← v − u
c_1 ← d_1 × e
c_2 ← e × d_2
if c_1 ≤ 0 ∧ c_2 ≤ 0 :
    return (d_1 × (p − u) ≥ 0) ∧ ((p − u) × d_2 ≤ 0) ∧ ((v − u) × (p − u)
else if c_1 ≤ 0 ∧ c_2 > 0 :
    return PontoEmVértice(p, u, d_1, u − v) ∨ PontoEmVértice(p, v, u − v, d_2)
else if c_1 > 0 ∧ c_2 ≤ 0 :
    return PontoEmVértice(p, u, d_1, v − u) ∨ PontoEmVértice(p, v, v − u, d_2)
else
    return ¬ PontoEmAresta(p, v, u, d_2, d_1)
```

disjuntos e os pontos s e t estarem fora dos polígonos. Ainda assim, se desejássemos evitar essa suposição, poderíamos simplesmente verificar se o ponto está dentro do polígono P_i usando um teste de ponto em polígono antes de localizar o ponto na partição, o é um problema clássico que leva tempo $O(\log(|P_i|))$.

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

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

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

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

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

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

2.5 Respondendo Consultas

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

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

Algorithm 6: LocalizaRegião(p, P, regiõesDeVértice)
Localiza o ponto p na partição do polígono P

```
// Criamos uma função auxiliar usando fechamento
// para facilitar a implementação da busca binária
def pertence(i, j):
    v_1 ← P[⌊i/2⌋] // Vértice i do polígono P
    v_2 ← P[⌊j/2⌋] // Vértice j do polígono P
    // Vamos lembrar que regiõesDeVértice é uma lista de pares (d_1, d_2)
    // Também vamos tratar i (mod 2) como uma operação tradicional de módulo
    d_1 ← regiõesDeVértice[⌊i/2⌋][i (mod 2)]
    d_2 ← regiõesDeVértice[⌊j/2⌋][j (mod 2)]
    return PontoEmAresta(p, v_1, v_2, d_1, d_2)
l ← 0 r ← 2 · |P| − 1
if pertence(r, 0) :
    return r // Ponto pertence à região entre o último e o primeiro vértice
while l + 1 ≠ r do
    m ← ⌊(l + r)/2⌋
    if pertence(l, m) :
        r ← m
    else
        l ← m
return l // Ponto pertence à região entre l e r
```

- Região de Vértice: Seja $R = (v, d_1, d_2)$. Nesse caso, o i-path até p deve passar pelo vértice v, tocando o polígono P_i. Assim, temos que $\text{Query}(p, i) = v$.
- Região de Aresta: Seja $R = (u, v, d_1, d_2)$. Nesse caso, o i-path até p deve passar por algum ponto q da aresta $e = (u, v)$, tocando o polígono P_i. Para calcular q, primeiro determinamos $q' = \text{Query}(p', i-1)$, onde p' é a reflexão de p em relação à aresta e. Agora, dizemos que q é a interseção entre $\overline{q'p'}$ e a aresta e. Finalmente, respondemos $\text{Query}(p, i) = q$.
- Região de Atravessa: Seja R a região de atravessa. Nesse caso, o i-path até p automaticamente atravessa o polígono P_i em algum ponto. Portanto, podemos simplesmente responder $\text{Query}(p, i) = \text{Query}(p, i-1)$.

Agora vamos implementar esse procedimento como um algoritmo, usando as contruções que fizemos anteriormente, uma vez que é um pouco mais complexo do que a descrição acima.

Vamos usar também uma função auxiliar InterseçãoDeSegmento que calcula a interseção entre dois segmentos dados seus extremos. Essa não é descrita aqui, mas é uma implementação padrão de geometria computacional.

2.6 Calculando o Caminho Mínimo

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

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

Usando essa lógica e juntando as implementações anteriores em um algoritmo completo, temos:

2.7 Análise de Complexidade

Nessa seção, vamos analisar a complexidade do algoritmo completo. Primeiramente, a função LocalizaRegião leva tempo $O(\log(|P_i|))$ para localizar um ponto na partição S_i, uma vez que usa busca binária.

Algorithm 7: Query(p, i)
Responde a consulta Query(p, i)

```
1  if i = 0 :
2      return s // Caso base
3  R ← LocalizaRegião(p, P_i, regiõesDeVértice[i − 1])
4  j ← ⌊R/2⌋
5  if R (mod 2) = 0 :
6      return v ← P_i[j] // Vértice associado à região de vértice
7  if ¬ visível[j] :
8      return Query(p, i − 1) // Região de atravessa
9  // Região de aresta
10 u ← P_i[j] // Vértice inicial da aresta
11 v ← P_i[j + 1] // Vértice final da aresta
12 p′ ← u + RefleteDireção((p − u), u, v)
13 q′ ← Query(p′, i − 1)
14 m ← InterseçãoDeSegmento(p′, q′, u, v)
15 return m // Ponto de interseção
```

Algorithm 8: CaminhoMínimo$(s, t, P_1, \ldots, P_k)$
Calcula o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$

```
1  if k = 0 :
2      return [s, t] // Caso base: caminho direto de s até t
3  regiãoDeVértice ← Particiona(P_1, ..., P_k)
4  caminho ← [] // Lista vazia para armazenar o caminho
5  p ← t // Começamos do ponto final t
6  for i ← k até 1 :
7      q ← Query(p, k) // Penúltimo ponto do k-path até p
8      // Verificamos se os pontos p e q são distintos devido à regiões de atravessar
9      if |q − p| > 10^−8 :
10         caminho.append(p) // Adiciona p ao caminho
11     p ← q // Atualiza p para o próximo ponto
12     k ← k − 1 // Decrementa k
13 caminho.append(s) // Adiciona o ponto inicial s
14 caminho.reverse() // Inverte a lista para obter o caminho correto
15 return caminho
```

Ademais, a função Query faz uma chamada recursiva para $i-1$ em cada nível de recursão, resultando em i níveis de recursão. Assim, o tempo total para responder uma consulta Query(p, i) é $O(i \log(|P_i|))$.

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

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

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

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

- Particiona($P_1, \dots, P_n$): Chamamos ArestasVisíveis para cada polígono P_i, assim, temos:

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

- CaminhoMínimo: Chamamos Particiona uma única vez e então calculamos o caminho final cuja complexidade é insignificante próximo de Particiona, então a complexidade final é a mesma:

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

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

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

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

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

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

3 O Problema de Visita de Polígonos Geral

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

3.1 Definições e Notação

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

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

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

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

3.2 Caminhos Restritos

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

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.