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
11 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.

Recomendamos a leitura do paper original para uma compreensão mais profunda dos conceitos e das provas por trás dos algoritmos apresentados aqui. Também é importante notar que este relatório não cobre todos os detalhes do paper, mas sim uma visão geral e a implementação prática dos algoritmos.

O código implementado pode ser encontrado no repositório GitHub, contendo scripts em *Python* para resolver ambos os casos do problema.

Finalmente, recomendamos a leitura do Apêndice A para entender as notações usadas ao longo do relatório.

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 Definindo o Problema

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 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.

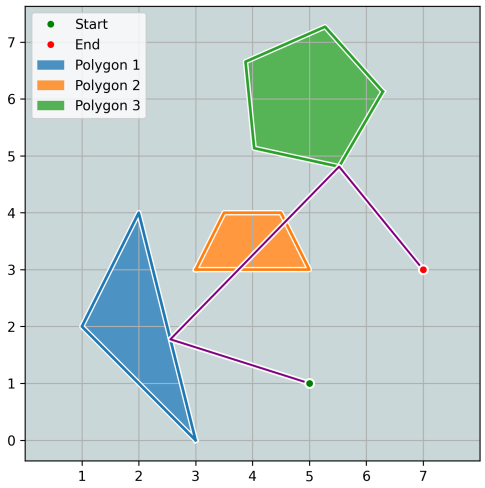

Figura 1: Caminho mínimo para um caso de 3 polígonos.

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 responder consultas usando esses procedimentos. Por enquanto, vamos assumir que sabemos como responder as consultas. Para as seções a seguir, vamos assumir que os polígonos $P_1, \ldots, P_k$ são representados por listas de vértices ordenados em sentido anti-horário, com um vértice abitrário como o primeiro.

2.2 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) dependa exclusivamente de a 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*.

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.

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.

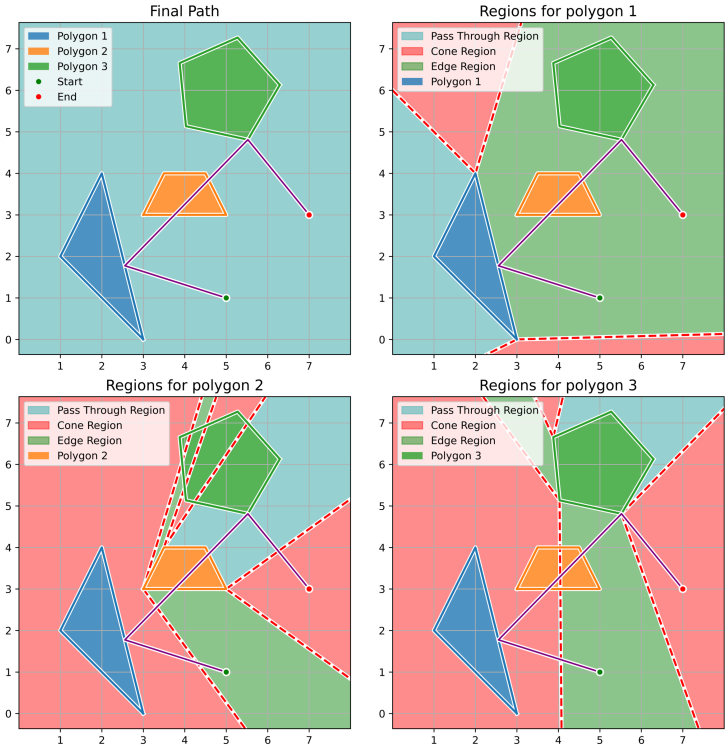

Figura 2: Partição do plano para cada polígono do exemplo anterior.

A partir da Figura 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. As fronteiras entre as regiões podem ser consideradas como pertencentes a qualquer uma das regiões que as delimitam, produzindo o mesmo resultado, mas para simplicidade, vamos considerar que pertencem às regiões de vértice.

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.

2.2.1 Região de Primeiro Contato

Para cada polígono P_i, definimos T_i como a *região de primeiro contato* de P_i, ou seja, o conjunto de pontos p no perímetro de P_i tais que o $(i-1)$-path até p toca em P_i pela primeira vez em p. Podemos provar [1] que T_i é uma região contínua do perímetro de P_i e que T_i é delimitada por vértices de P_i, ou seja, T_i é a união de uma sequência de arestas consecutivas de P_i. Por esse motivo, chamamos arestas de P_i que pertencem à T_i de **arestas visíveis** e as que não pertencem de **arestas bloqueadas**.

Essa região é importante, pois apenas as arestas visíveis de P_i que pertencem à T_i geram regiões de aresta em S_i, além disso, se um vértice está entre duas arestas bloqueadas, então ele não gera uma região de vértice em S_i.

Como T_i é a união de arestas, para determinar quais arestas de P_i pertencem à T_i, podemos simplesmente iterar sobre as arestas de P_i e verificar se o $(i-1)$-path até um ponto interno qualquer da aresta, por exemplo o ponto médio, chega em P_i pela primeira vez nesse ponto. Se isso for verdade, então a aresta pertence à T_i, caso contrário, não pertence.

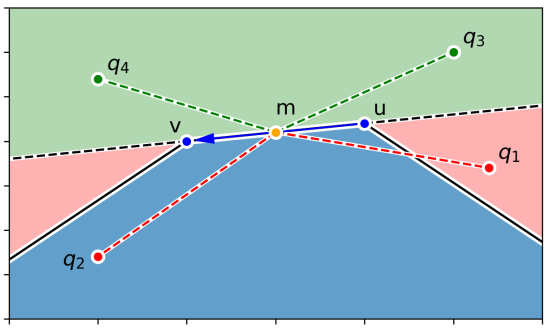

Figura 3: Testando aresta para região de primeiro contato.

No entanto, precisamos de uma maneira eficiente de fazer isso para todas as arestas de P_i. Se simplesmente fizermos isso de maneira ingênua, verificando se o segmento entre o último ponto q do $(i-1)$-path até o ponto médio m da aresta intersecta P_i, teríamos uma complexidade de $O(|P_i|)$ por aresta, resultando em $O(|P_i|^2)$

para todas as arestas de P_i.

Felizmente, como P_i é convexo e suas arestas estão ordenadas em sentido anti-horário, sabemos que o segmento $\overline{qm}$ toca P_i pela primeira vez em m se e somente se o produto vetorial de $\overrightarrow{mv}$ com $\overrightarrow{mq}$ for positivo. Usando essa ideia, podemos determinar todas as arestas visíveis de P_i em tempo $O(1)$ por aresta, totalizando $O(|P_i|)$.

A Figura 3 ilustra essa ideia. Temos o polígono P_i em azul, os vértices u e v em azul, o ponto médio m da aresta em laranja e 4 pontos q_1, q_2, q_3 e q_4 representando possíveis últimos pontos do $(i-1)$-path até m. Note que os segmentos $\overline{q_1m}$ e $\overline{q_2m}$ passam pela parte interna de P_i, assim, o produto vetorial de $\overrightarrow{uv}$ com $\overrightarrow{mq_1}$ e $\overrightarrow{mq_2}$ é positivo. Já os segmentos $\overline{q_3m}$ e $\overline{q_4m}$ não passam pela parte interna de P_i, assim, o produto vetorial de $\overrightarrow{uv}$ com $\overrightarrow{mq_3}$ e $\overrightarrow{mq_4}$ é negativo.

Implementamos essa ideia no algoritmo 1, que recebe um polígono P_i e seu índice i e retorna uma lista booleana indicando quais arestas de P_i são visíveis. Para os próximos algoritmos, vamos referenciar os objetos $T_1, \ldots, T_k$ como listas booleanas indicando quais arestas de cada polígono $P_1, \ldots, P_k$ são visíveis.

Algorithm 1: ArestasVisíveis(P_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 ← j-ésimo vértice de P_i
    v ← vértice que segue u em P_i
    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 − m) × (q − m) < 0)
return visível
```

2.2.2 Construindo Regiões de Vértice

Uma vez que sabemos quais arestas são visíveis, podemos construir as regiões de vértice. Observamos anteriormente que essas são delimitadas por duas semi-retas que partem de um vértice de P_i. Por esse motivo, nessa etapa, à cada vértice v vamos associar um par ordenado de direções $(d_{\text{prev}}, d_{\text{next}})$ tais que a região de vértice associada a v são os pontos entre as semi-retas que partem de v nas direções d_{prev} e d_{next}, no sentido anti-horário.

Para calcular as direções da regiões de vértice de um vértice v de P_i, devemos considerar o ponto $p = \text{Query}(v, i-1)$ e definimos $d = v - p$. Sejam v_{prev} o vértice que precede e v_{next} o que segue v em P_i (no sentido anti-horário). Se a aresta (v_{prev}, v) é visível, então d_{prev} será a reflexão de d em relação à aresta (v_{prev}, v), caso contrário, d_{prev} será a mesma direção de d. Similarmente, se a aresta (v, v_{next}) é visível, então d_{next} será a reflexão de d em relação à aresta (v, v_{next}), caso contrário, d_{next} será a mesma direção de d. Assim, criamos a região de vértice $(d_{\text{next}}, d_{\text{next}})$ associada à v.

Vale notar que se ambas as arestas incidentes a v estão bloqueadas, então $d_{\text{prev}} = d_{\text{next}} = d$, assim, a região de vértice associada a v é válida, mas vazia.

Encorajamos o leitor a verificar que essa lógica se aplica à Figura 4, usando o ponto inicial s como resposta de $\text{Query}(v, 0)$ para qualquer vértice v. Os vértices (-2, 0) e (0, 0) terão as suas duas arestas incidentes 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. Si-

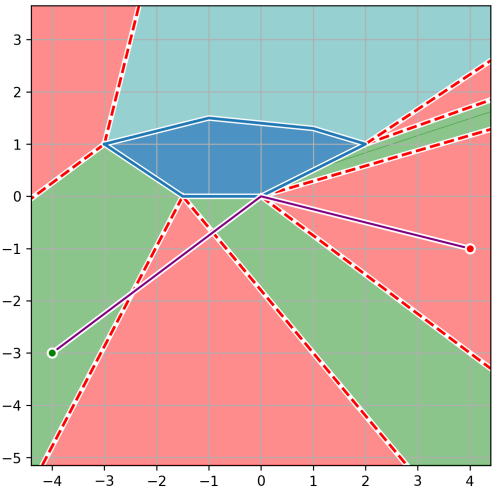

Figura 4: Partição de um polígono com múltiplas arestas não visíveis.

milarmente, 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 vazias.

Implementamos essas ideias nos algoritmos 2 e 3. O primeiro reflete uma direção em relação a uma aresta, enquanto o segundo cria a partição S_i de um polígono P_i, retornando uma lista de pares ordenados de direções representando as regiões de vértice associadas a cada vértice de P_i. Para os próximos algoritmos, vamos referenciar os objetos $S_1, \ldots, S_k$ como listas de pares ordenados de direções representando as regiões de vértice de cada polígono $P_1, \ldots, P_k$.

Algorithm 2: RefleteDireção($d, v_{\text{prev}}, v_{\text{next}}$)
Reflete a direção d em relação à aresta $(v_{\text{prev}}, v_{\text{next}})$

```
u ← v_prev − v_next
w ← Vetor normal à u, tal que |w| = 1
return d − 2⟨d, w⟩w // Fórmula de reflexão
```

Algorithm 3: ConstróiRegiõesDeVértice(P_i)
Constroi as partições do polígono P_i

```
regiões ← [] // Lista vazia
for j ← 0 até |P_i| − 1 :
    v ← j-ésimo vértice de P_i
    v_prev ← vértice que precede v em P_i
    v_next ← vértice que segue v em P_i

    p ← Query(v, i − 1) // penúltimo ponto do (i − 1)-path até v
    d ← v − p // direção do raio de p até v

    if Aresta j − 1 de P_i pertence à T_i :
        d_prev ← Reflete(d, v_prev, v) // reflexão de d em relação à aresta (v_prev, v)
    else
        d_prev ← d
    if Aresta j de P_i pertence à T_i :
        d_next ← Reflete(d, v, v_next) // reflexão de d em relação à aresta (v, v_next)
    else
        d_next ← d
    // Armazena a região de vértice associada a v como o par (d_prev, d_next)
    regiões.append((d_prev, d_next))
return regiões
```

2.3 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.

2.3.1 Verificando Regiões de Vértice

Primeiramente vamos implementar um procedimento auxiliar 4 que verifica se um ponto p está em uma região de vértice associada a um vértice v delimitada por duas direções d_{prev} e d_{next}. Basicamente, temos dois casos, dependendo do ângulo entre as direções d_{prev} e d_{next} (considerando o sentido anti-horário):

- Ângulo $\leq 180°$: Nesse caso, p deve estar à esquerda de d_{prev} e à direita de d_{next}.

- Ângulo $> 180°$: Nesse caso, p deve estar à direita de d_{prev} ou à esquerda de d_{next}.

Podemos usar produtos vetoriais para determinar a posição relativa de p em relação às direções d_{prev} e d_{next}, assim como o ângulo entre essas direções. Usando essas ideias, implementamos o procedimento no algoritmo 4.

Algorithm 4: PontoEmVértice(p, v, d_{prev}, d_{next})
Verifica se o ponto p está na região de vértice associada ao vértice v delimitada pelas direções d_{prev} e d_{next}

```
c1 ← (d_prev × (p − v) ≥ 0)
c2 ← (d_next × (p − v) ≤ 0)
if d_prev × d_next ≥ 0 :
    return c1 ∧ c2
else
    return c1 ∨ c2
```

2.3.2 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.

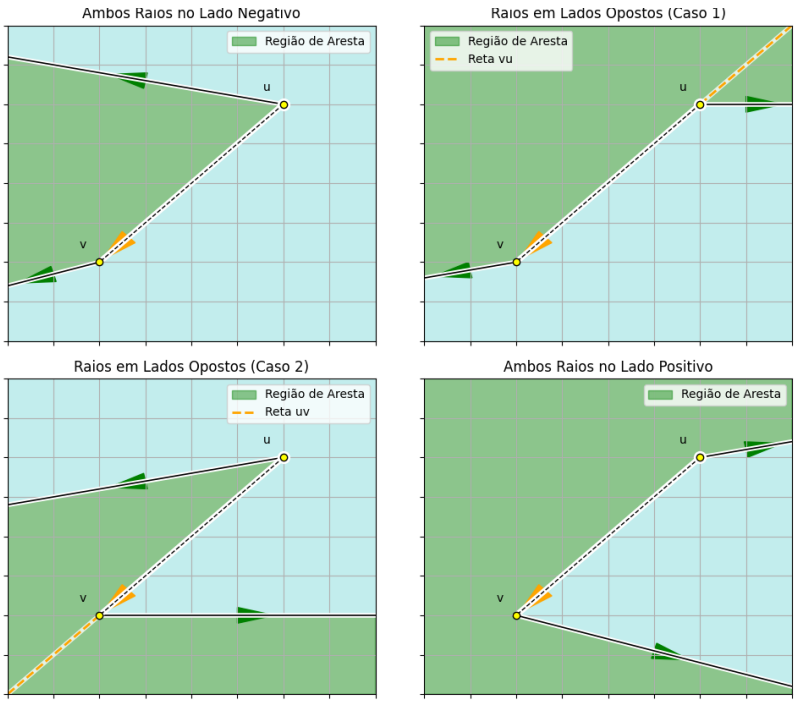

Figura 5: Casos possíveis para região de aresta.

Esse problema é mais complexo, uma vez que existem diferentes casos possíveis para uma região de aresta, dependendo do ângulo entre as direções d_{prev} e d_{next}. 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_{prev} e d_{next}. A Figura 5 ilustra esses casos. Para sabermos com qual caso estamos lidando, definimos $w = v - u$ e calculamos os produtos vetoriais $c_1 = w \times d_{\text{prev}}$ e $c_2 = w \times d_{\text{next}}$. Agora, temos os seguintes casos:

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

- $c_1 \leq 0 \wedge c_2 \leq 0$: Esse é o caso em que temos **aresta anterior no lado negativo e aresta posterior no lado positivo**, o *Caso 1* na figura. Nesse caso, dividimos a região de aresta em duas regiões de vértice, definidas por $(u, d_{\text{prev}}, u - v)$ e $(v, u - v, d_{\text{next}})$, 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 $(u, d_{\text{prev}}, v-u)$ e $(v, v-u, d_{\text{next}})$, 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 $(v, u, d_{\text{next}}, d_{\text{prev}})$.

Dessa forma, conseguimos verificar se um ponto pertence a uma região de aresta usando o procedimento de verificação de regiões de vértice. Agora podemos implementar o procedimento completo no algoritmo 5.

Algorithm 5: PontoEmAresta(p, u, v, $d_{\text{prev}}, d_{\text{next}}$)
Verifica se o ponto p está na região de aresta associada à aresta $e = (u, v)$ delimitada pelas direções d_{prev} e d_{next}

```
// 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_next, d_prev)
w ← v − u
c_1 ← d_prev × w
c_2 ← w × d_next
if c_1 ≤ 0 ∧ c_2 ≤ 0 :
    return (d_prev × (p − u) ≥ 0) ∧ (d_next × (p − u) ≤ 0) ∧ ((v − u) × (p − u) ≤ 0)
else if c_1 ≤ 0 ∧ c_2 > 0 :
    return PontoEmVértice(p, u, d_prev, u − v) ∨ PontoEmVértice(p, v, u − v, d_next)
else if c_1 > 0 ∧ c_2 ≤ 0 :
    return PontoEmVértice(p, u, d_prev, v − u) ∨ PontoEmVértice(p, v, v − u, d_next)
else
    return ¬ PontoEmAresta(p, v, u, d_next, d_prev)
```

2.3.3 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 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 $2|P_i| - 1$.

Implementamos essa ideia no algoritmo 6, que recebe um ponto p e um polígono P_i e retorna o índice da região de S_i que contém p.

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

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

2.4 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:

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

- Região de Atravessa: Seja R a região de atravessa. Nesse caso, o i-path até p automaticamente atravessa o polígono P_i em algum ponto. Portanto, podemos simplesmente responder Query$(p, i) =$ Query$(p, i-1)$.

Para determinar o ponto de interseção entre o segmento $\overline{q'p'}$ e a aresta e, implementamos o procedimento 7, que calcula a interseção entre duas retas dadas por dois pontos e suas direções. A demonstração desse procedimento é simples e pode ser feita algebricamente, mas não é o foco desse relatório. Também estamos assumindo que as retas não são paralelas, o que é garantido pela construção do algoritmo.

Agora podemos usar essas ideias para implementar o procedimento completo de resposta a consultas na forma do algoritmo 8.

Algorithm 7: InterseçãoRetas(s, d, s', d')
Calcula a interseção entre as retas definidas por pontos $s + td$ e $s' + t'd'$, onde $t, t' \in \mathbb{R}$

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

Algorithm 8: Query(p, i)
Responde a consulta Query(p, i)

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

2.5 Calculando o Caminho Mínimo

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

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

Usando essa lógica e juntando as implementações anteriores no algoritmo 9, podemos calcular o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$.

Algorithm 9: CaminhoMínimo$(s, t, P_1, \ldots, P_k)$
Calcula o caminho mínimo de s até t que toca todos os polígonos $P_1, \ldots, P_k$

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

2.6 Análise de Complexidade

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

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

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

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

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

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

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

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

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

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:

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

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 $\overline{p_1p_2}$ se ele estiver contido em F_i e não existe caso contrário. Assim, vamos assumir que $i < j$.

A Notações

1. $v = (v_1, v_2), v \in \mathbb{R}^2$. Vetores são representados como tuplas de coordenadas.
2. $|v| = \sqrt{v_1^2 + v_2^2}$. Norma Euclidiana de um vetor.
3. $\langle u, v \rangle = u_1 v_1 + u_2 v_2$. Produto interno definido como produto escalar de vetores.
4. $u \times v = u_1 v_2 - u_2 v_1$. Produto vetorial definido como determinante 2D, retornando um escalar. Essa definição tem a seguinte propriedade:

$$\text{sign}(u \times v) = \begin{cases} +1, & \text{se ângulo no sentido anti-horário entre } u \text{ e } v \text{ é menor que } 180° \\ -1, & \text{se ângulo no sentido horário entre } u \text{ e } v \text{ é menor que } 180° \\ 0, & \text{se } u \text{ e } v \text{ são colineares} \end{cases}$$

   Pode-se interpretar $u \times v$ como a área do paralelogramo formado por u e v, onde o sinal indica a orientação.

5. Dizemos que um ponto p está do **lado esquerdo** de uma direção d se $d \times p \geq 0$ e do **lado direito** se $d \times p \leq 0$.
6. $\overline{uv}$: Segmento de reta entre os pontos u e v.
7. $\overrightarrow{uv} = v - u$. Vetor direcionado do ponto u até o ponto v.
8. $|P|$. Se P é um polígono, então $|P|$ é o número de vértices de P.

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.