



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
4 de novembro de 2025

Sumário

1	Introdução	2
2	O Problema de Visita de Polígonos Irrestrito	2
2.1	Definições e Notação	2
2.2	Representação dos Objetos	2
2.3	Particionando o Plano	2
2.3.1	Representado as Partições	3
2.3.2	Detectando Arestas Visíveis	3
2.3.3	Construindo Regiões de Vértice	4
2.4	Respondendo Consultas	5
2.5	Calculando o Caminho Mínimo	5
3	O Problema de Visita de Polígonos Geral	5
3.1	Definições e Notação	5
3.2	Caminhos Restritos	6

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, \dots, 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**.

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, \dots, 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 $\text{Query}(p, i)$, 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.

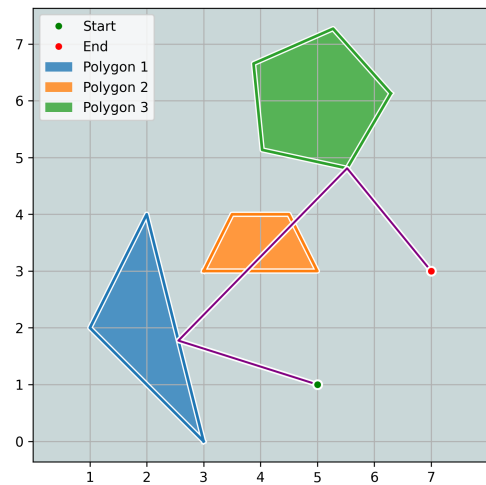


Figura 1: Caminho mínimo para um caso de 3 polígonos.

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 $\text{Query}(p, i)$ 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*.

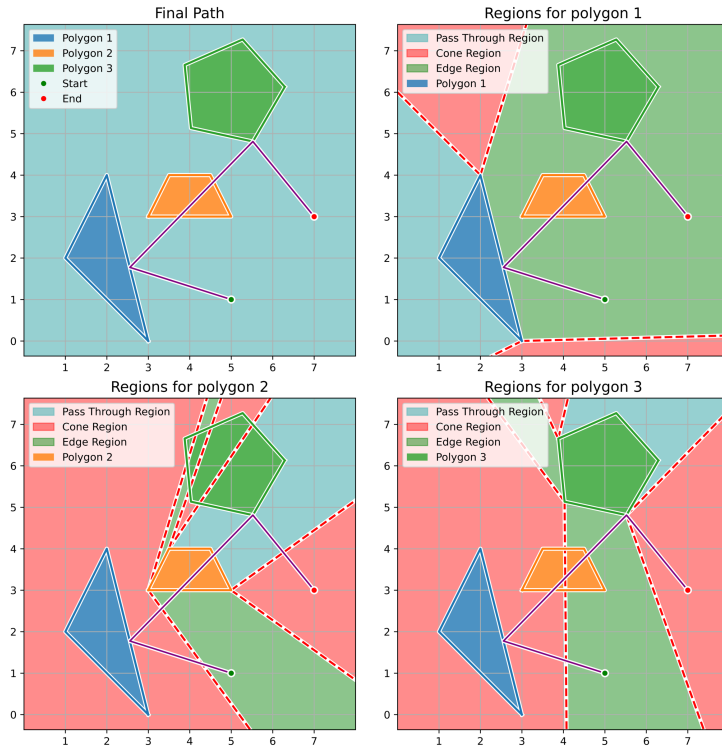


Figura 2: Partição do plano para cada polígono do exemplo anterior.

mos 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 própria 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.

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, deve-

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 anti-horário, 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: Determina as arestas visíveis de P_i

```

1  $n \leftarrow |P_i|$  // Número de vértices de  $P_i$ 
2 for  $j \leftarrow 0$  até  $n - 1$  :
3    $u \leftarrow e[0]$ 
4    $v \leftarrow e[1]$ 
5    $m \leftarrow (u + v)/2$ 
6    $q = \text{Query}(m, i)$ 
```

2.3.3 Construindo Regiões de Vértice

Primeiramente, a partir da figura, note que nem todo vértice e nem toda aresta de P_i geram uma região na partição S_i . Dizemos que uma aresta que gera uma região é uma aresta visível, assim, nosso primeiro passo é encontrar todas as arestas visíveis de P_i .

Uma maneira intuitiva de se uma aresta e é visível é calcular $p = \text{Query}(m, i - 1)$ onde m é o ponto médio de e e verificamos se \overline{pm} atravessa P_i . Se o segmento atravessa P_i , então e não é visível, caso contrário, e é visível. No entanto, essa abordagem é ineficiente, uma vez que determinar se \overline{pm} atravessa P_i custa $O(|P_i|)$ com uma abordagem ingênua e uma abordagem mais eficiente seria complexa de implementar. Por esse motivo vamos calcular essas arestas posteriormente.

Agora vamos calcular as regiões de vértice de um polígono P_i . Vamos considerar o vértice v de P_i e calculamos $p = \text{Query}(v, i - 1)$, a seguir explicamos 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 .

O cálculo de si da reflexão um vetor d em relação à uma aresta $e = (u, v)$, é uma simples conta com base em projeções de vetores e pode ser implementada como a função abaixo:

```

def reflect(d: Vector2, e: Edge2) -> Vector2:
    dir = (e.v - e.u)
    normal = Vector2(-dir.y, dir.x)
    proj_len = d.dot(normal) / normal.dot(normal)
    projection = normal * proj_len
    return d - 2 * projection
```

Usando essa função, definimos $d_1 = \text{reflect}(d, (v_1, v))$ se a aresta (v_1, v) for visível e $d_1 = d$ caso contrário. A mesma lógica segue para d_2 .

Encorajamos o leitor a verificar que essa lógica se aplica nas duas imagens ao lado 3 ??, usando o ponto inicial s como resposta de $\text{Query}(v, 0)$ para qualquer vértice v . Para vértices com ambas arestas visíveis,

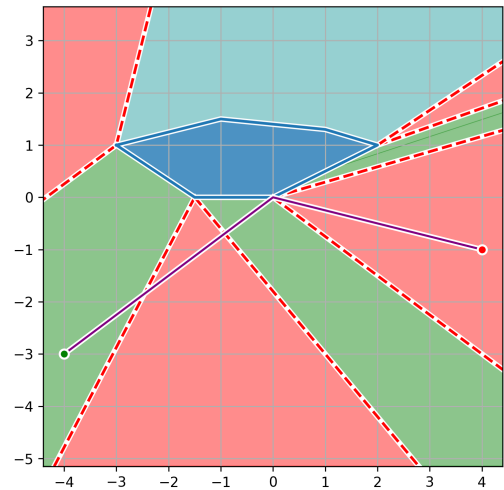


Figura 3: Partição de um polígono com múltiplas arestas não visíveis.

ambas as direções são reflexões, para vértices com uma aresta visível, uma direção é reflexão e a outra é a mesma direção do raio, e para vértices com ambas as arestas não visíveis, ambas as direções são iguais à direção do raio, assim não são desenhada.

Isso completa a construção das regiões de vértice. No entanto, ainda não esclarecemos como determinar se uma aresta é visível ou não de maneira eficiente.

2.4 Respondendo Consultas

Agora que sabemos como construir as partições S_i , podemos descrever como usá-las para responder consultas $\text{Query}(p, i)$. O procedimento utilizado é recursivo, assim, primeiro definimos o caso base, que é $\text{Query}(p, 0) = s$, 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 . Descrevemos acima como representar cada região e a partir dela é fácil determinar à qual um ponto p pertence. Também é importante ressaltar que p sempre pertence a exatamente uma região. Agora vamos analisar os 3 casos possíveis para 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 $\text{Query}(p, i) = \text{Query}(p, i - 1)$.

2.5 Calculando o Caminho Mínimo

Uma vez que sabemos como responder consultas, calcular o caminho final é muito simples. Basta calcular $q_k = \text{Query}(t, k)$, então $q_{k-1} = \text{Query}(q_k, k - 1)$ e assim por diante, até $q_0 = s$. Assim, o caminho mínimo é simplesmente a sequência s, q_1, \dots, q_k, t .

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, \dots, 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, \dots, F_j se π toca todos os polígonos P_{i+1}, \dots, 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, \dots, 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 $\text{Query}(p, i, j)$ 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, \dots, P_i e respeita as cercas F_0, \dots, F_j .

Primeiramente, é essencial que $i \leq j$, ademais, se $i = j$ então $\text{Query}(p, i, j)$ 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 responder as consultas.

3.2 Caminhos Restritos

Outra função extremamente importante para a implementação desse algoritmo é a função $\text{Fenced}(p_1, p_2, i, j)$ que retorna o menor caminho de p_1 até p_2 que respeita as cercas F_i, \dots, F_j . Note que se $i = j$, então $\text{Fenced}(p_1, p_2, i, j)$ 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>.