Universidad Nacional Autónoma de México

Facultad de Estudios Superiores Acatlán

UN ALGORITMO EXPONENCIAL PARA ACOPLAR UN ÁRBOL DE STEINER AL INTERIOR DE UN POLÍGONO SIMPLE

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
LICENCIADO EN MATEMÁTICAS APLICADAS Y COMPUTACIÓN

PRESENTA:
MARCOS CORTÉS VALADEZ

DIRECTOR DE TESIS:
JOSÉ SEBASTIÁN BEJOS MENDOZA

Enero 2014

Para las mujeres más importantes en mi vida:
Tere, Arely y Evelin

Agradecimientos

En primer lugar a mi mamá Teresa, quien además de regalarme la vida misma, ella ha sido el apoyo más grande que he tenido.

Un sincero agradecimiento a mi director de tesis y gran amigo Sebastián, quien me abrió la puerta hacia un mundo fascinante conocido como geometría computacional, mundo del cual no pretendo salir. Debo admirar su paciencia y su actitud, las cuales han sido un aporte invaluable para todo el desarrollo de esta tesis.

Creo que siempre es necesario tener un lugar de trabajo, por lo que agradezco también el espacio que se me ha brindado en el Laboratorio de Algoritmos para la Robótica, sitio en el cual pude expresar mis ideas, pulirlas y finalmente presentarlas en este escrito.

A Raúl, una excelente persona, mi amigo del alma y colega en muchos proyectos, por comprender todas mis ideas en el mismo instante en que se las comunico (o incluso antes). Por ayudarme siempre a desenredar la enmarañada bola de pensamientos en mi cabeza.

Finalmente, a todos mis compañeros y amigos que directa o indirectamente contribuyeron con un granito de arena durante este entretenido y arduo camino que todo el mundo llama cómodamente: escribir una tesis.

Índice

Introducción

Supongamos un robot que se mueve de un punto a otro dentro de una región poligonal P. El robot avanza en línea recta hacia adelante y cada vez que desea cambiar de dirección se debe detener para rotar. Se considera que el movimiento recto hacia adelate es "barato", mientras que un giro es "costoso", en téminos de la energía consumida y la perdida de precisión en la trayectoria del robot. Es interesante entonces encontrar una trayetoria con el menor número de giros para un robot móvil. En la geometría computacional este problema es conocido como: el problema de encontrar una trayectoria mínima en aristas.

Los primeros en estudiar este problema fueron ElGindy en [8] y Suri en [17]. Usando técnicas de visibilidad, Suri en [17, 18] y Gosh en [6] desarrollaron algunos algoritmos para calcular una trayectoria mínima en aristas. Ambos algoritmos tienen complejidad en tiempo de $O(n)$ en donde n es el número de vértices de la región poligonal.

Sebastian Bejos (coordinador del Laboratorio de algoritmos para la robótica -LAR- de la FES Acatlán) planteó una generalización del problema. En esta generalización se desea encontrar un árbol con el menor número de aristas o segmentos que conecte a un conjunto M de k puntos dentro de una región poligonal P.

Siendo estudiante de 7mo semestre de la Licenciatura en Matemáticas Aplicadas y Computación (MAC), junto con Raul Martínez comencé a trabajar junto con Sebastián Bejos en una variación un poco más simple del problema. En lugar de construir un árbol óptimo dentro de la región poligonal, deseabamos saber si, dado un árbol de Steiner T con k nodos terminales, éste podría ser usado como un árbol de trayectorias, de tal manera que todos los vértices terminales sean incrustados como puntos de M, y donde los vértices Steiner de T tuvieran la libertad de colocarse en partes específicas del polígono. Tiempo después obtuvimos un par de resultados para esta variante del problema, los cuales aprovechamos como temas de tesis. El primer resultado que se encontró fue que este problema pertenece a la clase de problemas NP. Raúl explica con detalle este hecho en la tesis [13]. Nuestro segundo resultado, fue un algoritmo exponencial que valida si un árbol de Steiner dado puede ser acoplado dentro de un polígono simple. En este trabajo se da a conocer este algoritmo.

De manera simple, nuestro algoritmo toma como entrada un árbol de Steiner T, un polígono simple P y un conjunto M de k puntos que se encuentran en el interior de P. El algoritmo asigna todos los nodos terminales de T a puntos de M, luego comprueba si es posible acomodar los vértices Steiner de T dentro del póligono sin que las aristas de T se intersecten con las aristas de P, si esto es posible entonces el árbol puede ser acoplado.

En el capítulo 1, se mencionan algunos conceptos básicos, como por ejemplo algunas

definiciones de teoría de grafos, los tipos de visibilidad en el plano, tipos de polígonos, etc. En el capítulo 2 se explican a detalle cuatro algoritmos de geometría computacional usados como subrutinas en nuestro algoritmo. En el capítulo 3 veremos como se calcula la trayectoria mínima en aristas entre dos puntos dentro de un polígono simple con el algoritmo de Gosh [6, 7]. Para terminar, en el capítulo 4 se describe a detalle nuestro algoritmo que verifica si un árbol de Steiner puede acoplarse al interior de un polígono simple.

Capítulo 1

Conceptos básicos

En este capítulo se definen todos los conceptos que usaremos a lo largo de todo el trabajo. Comenzaremos con definiciones de gráficas y gráficas no dirigidas. Definimos también lo que es un árbol y un árbol de Steiner. Por último y más importante se encuentran los conceptos de visibilidad los cuales son una parte importante de la investigación. Vease [15, 5] para familiarizarse con más de estos conceptos.

1.1. Árboles de Steiner

Se define una **gráfica** G como un par de conjuntos disjuntos (V, E) donde $E \subseteq V \times V$, es decir, cada elemento de E esta formado por un par de elementos de V, por ejemplo, sean u y v dos elementos de V entonces (u, v) o uv denotan un elemento de E, también puede ser denotado por una sola letra, por ejemplo: e. Cabe mencionar que no todas las parejas entre los elementos de V podrían estan en E. Los elementos de V son llamados *vértices* (o nodos) de G y los elementos de E son llamados *aristas* de G.

La forma usual de representar una gráfica es dibujando un punto por cada vértice y unir dos de estos puntos con un trazo solo si los dos vértices correspondientes forman una arista de E. La localización de los puntos en la representación es irrelevante (ver figura 1.1).

El conjunto de vértices V de una gráfica G es denotado como $V(G)$ y el conjunto de aristas es denotado como $E(G)$. Una arista e es *incidente* a un vértice v si $v \in e$. Los dos vértices que forman una arista e son llamados *vértices finales* de e. Dos aristas $e_1 = (x_1, y_1)$ y $e_2 = (x_2, y_2)$ son *adyacentes* si tienen un vértice final en común. El grado (o valencia) $\delta(v)$ de un vértice v es el número de aristas incidentes a v.

Una gráfica no vacía G es conexa (o gráfica *conectada*) si para cada par de sus vértices existe un camino en G (ver figura 1.1 como ejemplo de una gráfica no conectada).

Un **camino** (o recorrido) es una gráfica $R = (V, E)$ donde $V = \{v_0, v_1, \ldots, v_k\}$ y $E = \{v_0v_1, v_1v_2, \ldots, v_{k-1}v_k\}$ tal que $v_i \neq v_j$ si $i \neq j$. La *longitud de un camino* R esta dada por el número de aristas, es decir, la cardinalidad de su conjunto E. Un camino es denotado como una secuencia de sus vértices, es decir, $R = v_0v_1 \ldots v_k$ y se dice que R es un camino de v_0 a v_k.

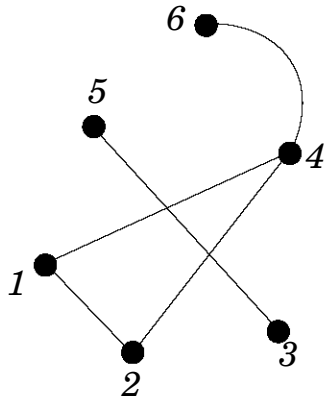


Figura 1.1: Ejemplo de una gráfica $G = (V, E)$ donde $V = \{1, 2, 3, 4, 5, 6\}$ y $E = \{(1, 2), (1, 4), (2, 4), (5, 3), (4, 6)\}$

Si $R = v_0 \dots v_{k-1}$ es un camino y $k \geq 3$, entonces la gráfica $C = R + \{v_{k-1}v_0\}$ es llamada un ciclo. De la misma forma que un camino, un ciclo es denotado como una secuencia de sus vértices, es decir, $C = v_0v_1 \dots v_{k-1}v_0$. La *longitud de un ciclo* C esta dada por el número de aristas de C.

Una **gráfica dirigida** es un par (V, E) de conjuntos disjuntos donde V es el conjunto de vértices y E es el conjunto de aristas. Además, una gráfica dirijida tiene dos funciones:

i) origen: $E \to V$
ii) destino: $E \to V$

que son utilizadas para asignar a cada arista e un vértice inicial *origen(e)* y un vértice final *destino(e)*. Se dice que la arista dirigida e va del vértice *origen*(e) al vértice *destino*(e) (ver figura 1.2). En los dibujos una arista dirigida es representada por una flecha.

Un **bosque** es una gráfica G que no contiene ciclos, la cual también es llamada gráfica acíclica. Un bosque conectado es llamado un *árbol*, por lo que, un bosque es una gráfica que está compuesta por árboles. Los vértices de grado igual a 1 en un árbol son llamados *hojas*. Muchas veces es conveniente considerar un vértice de un árbol como principal, llamandolo *raíz* del árbol. Un árbol con una raíz fija r es llamado *árbol con raíz* y se denota como T_r. Si la raíz r de un árbol T_r tiene grado igual a 1 ($\delta(r) = 1$), r no será llamada una hoja de T.

El *diámetro* de un árbol T es el camino más largo entre cada par de nodos de T.

Un **árbol de Steiner** $T = (V, E)$ es un árbol tal que el conjunto de nodos V está dividido en dos subconjuntos K y $S = V \setminus K$. Los nodos que se encuentran en el subconjunto K son llamados nodos *terminales* y los nodos de S son llamados *Steiner*. La única propiedad de los nodos Steiner es que no deben de ser hojas de T. Un nodo terminal no necesariamente es una hoja de T. Normalmente los nodos Steiner son usados como vértices con

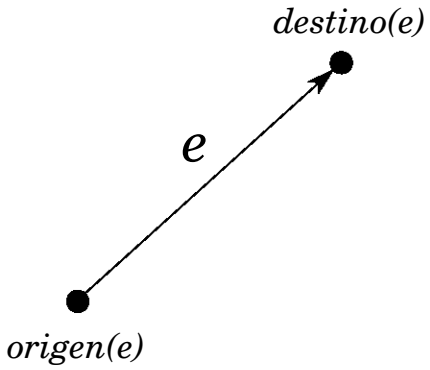


Figura 1.2: Vértices $origen(e)$ y $destino(e)$ de la arista dirigida e.

características distintas a los términales, estas características dependen del problema. En nuestro ejemplo del robot, usamos los nodos terminales como los puntos de la habitación a los que el robot debe llegar, mientras que los nodos Steiner los usamos como puntos sin posición previa en los cuales el robot deberá hacer un giro.

Normalmente usaremos la notación $V_s(T)$ y $V_t(T)$ para hacer referencia a los conjuntos de nodos Steiner y terminales de T respectivamente. En los dibujos los nodos terminales de T estarán representados por un punto totalmente negro mientras que los nodos Steiner serán circulos blancos con circunferencia negra, ver figura 1.3.

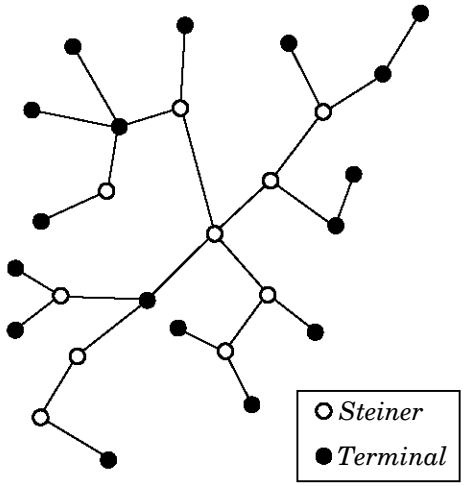


Figura 1.3: Un árbol de Steiner.

1.2. Visibilidad en el plano

Un **polígono** P es una región R en el plano delimitada por un conjunto finito de segmentos de línea (llamados también aristas de P) de tal manera que dentro de R es posible crear un camino entre cualesquiera dos puntos de R sin que alguna arista del camino se intersecte con algun segmento de P, es decir, ningún par de aristas de P debe intersectarse excepto en sus vértices finales. Observe que la frontera de P es un ciclo. Si P contiene ciclos en su interior (desconectados de P), entonces P es llamado un *polígono con agujeros*, de otra forma, P es llamado *polígono simple* o *polígono sin agujeros*. La región R es llamada el *interior* de P. Usualmente se denota a esta región con la misma etiqueta P y la fontera de P como $bd(P)$. La parte de la frontera o secuencia seleccionada por los vértices v_j, v_{j+1}, ..., v_m es denotada como $bd(v_j, v_m)$. La parte del plano que está fuera de P es llamado el *exterior* de P.

Un polígono simple P es *convexo* si el ángulo interno de cada vértice es a lo más π.

En los capítulos siguientes usamos una técnica para calcular regiones dentro de un polígono. Esta técnica es un recorrido sobre los vértices de la frontera del polígono. Usamos las palabras dextrógiro y levógiro para referirnos al orden o dirección que habrá que seguir sobre los vértices. Se dice que un recorrido es *dextrógiro* cuando gira en el mismo sentido de las manecillas del reloj $\circlearrowright$. En contrasentido, decimos que es *levógiro* cuando gira al contrario de las manecillas del reloj $\circlearrowleft$.

Se dice que dos puntos p y q en P son visibles si el segmento de linea que une a p y q está totalmente contenido dentro de P. Esta definición permite que el segmento pq pase a través de los vértices de P. Se dice que q es visible por p si p y q son visibles en P. Es obvio que, si q es visible por p entonces p es visible por q, solo cuando p y q son puntos.

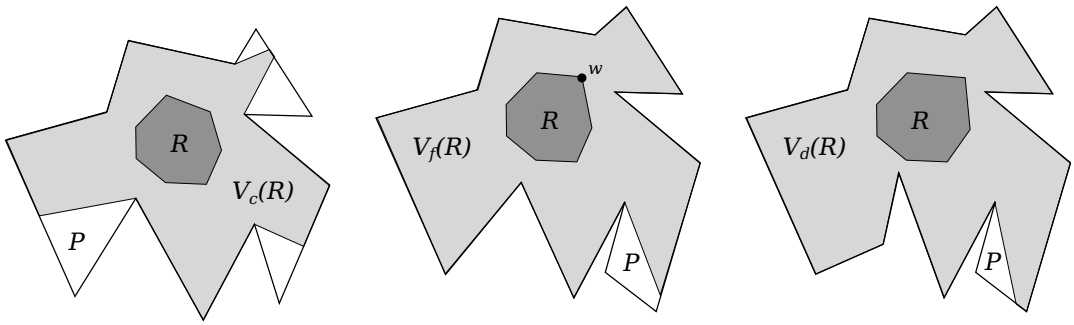
Sea q un punto dentro del polígono simple P y sea $V(q)$ el *polígono de visibilidad* o región visible de q dentro de P tal que cada punto de $V(q)$ es visible por q.

Generalizando la noción de visibilidad de un punto es posible obtener la visibilidad un segmento y de una región. En este trabajo solo se necesita el algoritmo para obtener la visibilidad de una región, por lo que presentamos los tres diferentes tipos de visibilidad de una región convexa R propuestos por Ghosh en [6] los cuales no son más que una generalización de los propuestos por Avis y Toussaint [4] para un segmento.

- Se dice que $V_c(R)$ es la región de *visibilidad completa* para una región R si cualquier punto $w \in R$ y cualquier punto $z \in V_c(R)$, w y z son visibles (ver figura 1.4(a)).
- Se dice que $V_f(R)$ es la región de *visibilidad fuerte* para R si existe un punto $w \in R$ tal que para todo punto $z \in V_f(R)$, w y z son visibles (ver figura 1.4(b)).
- Se dice que $V_d(R)$ es la región de *visibilidad débil* para R si para cada punto $z \in V_d(R)$ existe un punto $w \in R$ (dependiendo de z) tal que w y z son visibles (ver figura 1.4(c)).

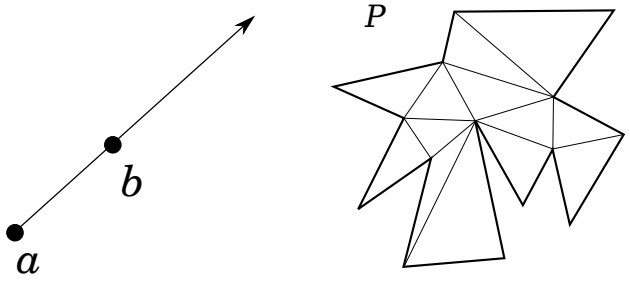
Un *rayo* es definido como una semilínea dibujada desde un punto a pasando a través de un punto b y es denotado como $\overrightarrow{ab}$ (ver figura 1.5(a)).

El *cierre convexo* de un conjunto de puntos C en el plano es la frontera del polígono convexo de menor área que contiene a C.



(a) (b) (c)

Figura 1.4: (a) Visibilidad completa de R. (b) Visibilidad fuerte de R. (c) Visibilidad débil de R.



(a) (b)

Figura 1.5: (a) Un rayo de a pasando por b. (b) Una triangulación de P usando diagonales sin intersecciones.

Una *diagonal* en un polígono P es un segmento de línea que conecta dos vértices de P y está totalmente contenida en P.

Dado un polígono P, una *triangulación* de P es una división de todo su interior en triángulos usando diagonales que no se intersectan más que en los vértices de P (ver figura 1.5(b)).

Capítulo 2

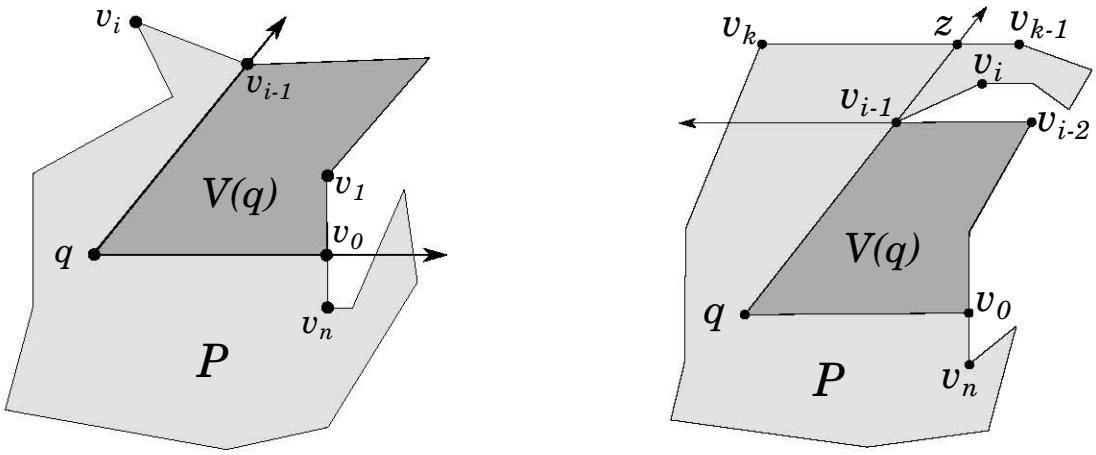
Algoritmos geométricos

En este capítulo mostramos cuatro algoritmos que usamos como subrutinas de nuestro algoritmo principal. El primero obtiene la región que un punto puede ver al estar dentro de un polígono simple. El segundo y el tercero obtienen la región que un conjunto de puntos pueden ver dentro de un polígono simple de dos maneras distintas. Por último se presenta un algoritmo que calcula la región donde se intersectan dos polígonos simples.

2.1. Polígono de visibilidad de un punto dentro de un polígono simple

En esta sección, se presenta el algoritmo de Lee publicado en [10] para calcular el polígono de visibilidad $V(q)$ de un punto q dentro de un polígono simple P de n vértices en tiempo $O(n)$.

En resumen, para calcular $V(q)$ el algoritmo trabaja de la siguiente forma: tomando $bd(P)$ como una secuencia de vértices en forma levógira, se traza un rayo a partir de q hacia el vértice inicial v_0 de P. Imagine que este rayo simula una manecilla de reloj la cual podemos mover a nuestro antojo de forma dextrógira o levógira con centro en q. Movemos esta manecilla colocandola sobre cada vértice v_i siguiendo la secuencia de $bd(P)$ para formar cada vez un rayo $\overrightarrow{qv_i}$; suponga que los vértices v_0 hasta v_{i-1} en la secuencia son visibles por q, entonces la manecilla pasará por cada uno abriendose como si se tratara de un abanico, este movimiento irá creando el polígono de visibilidad $V(q)$ hasta encontrarse con algún vértice no visible por q. Para saber cuando un vértice v_i no es visibile por q existen varios casos que veremos a detalle más adelante, pero en escencia se toma la manecilla y se coloca sobre la punta del pico que bloquea la visibilidad de q a v_i y se prolonga hasta que se intersecte con alguna arista de P, tomamos el segmento formado a partir del punto en la punta del pico hasta el punto donde se intersecta el rayo. Este segmento será el límite entre la parte visible y la no visible por q dentro de P. Luego continuamos abriendo el abanico apartir de este límite y hacemos lo mismo cada vez que encontremos un vértice no visible por q. Cuando lleguemos de nuevo al vértice v_0 habremos dado una vuelta completa y tendremos el polígono de visibilidad $V(q)$ totalmente calculado.



(a) (b)

Figura 2.1: (a) El vértice v_i es insertado en la pila. (b) Los vértices de $bd(v_i, v_{k-1})$ no son visibles por q.

A continuación se ve a detalle la construcción de $V(q)$, escrita por Ghosh en [7].

Sea v_0 el punto más cercano a q entre los puntos de intersección de $bd(P)$ con la linea horizontal dibujada a partir de q hacia la derecha (ver figura 2.1(a)). Se asume que los vértices de P estan etiquetados con $v_1, v_2, \ldots, v_n$ en orden levógiro, donde v_1 es el siguiente vértice de v_0. v_1 y v_n se encuentran a la izquierda y derecha de $\overrightarrow{qv_0}$ respectivamente.

El procedimiento para calcular $V(q)$ recorre $bd(P)$ de forma levógira a partir de v_0. Se utiliza una estructura de pila, para guardar cada vértice o punto de $bd(P)$ visible por q. Puede verse que al finalizar el procedimiento la pila contendrá todos los vértices de $V(q)$ y que cada par de vértices adyacentes dentro de la pila generarán una arista para el polígono $V(q)$, agregando tambien una arista para unir el vértice que se encuentra en el tope de la pila con el del fondo.

Ahora veremos como el algoritmo verifica la visibilidad de q a cada vértice v_i que es tomado en el recorrido. Se tienen los siguientes casos:

- **Caso 1.** v_i se encuentra a la izquierda de $\overrightarrow{qv_{i-1}}$ (ver figura 2.1(a)). Insertar v_i en la pila, ya que, por lo que se sabe hasta ahora, v_i es visible por q.

- **Caso 2.** v_i se encuentra a la derecha de $\overrightarrow{qv_{i-1}}$. Ya que v_{i-1} y v_i no pueden ser ambos visibles por q, se tienen dos subcasos:

  - **Caso 2a.** v_i se encuentra a la derecha de $\overrightarrow{v_{i-2}v_{i-1}}$ (ver figura 2.1(b)) o $bd(v_0, v_{i-1})$ es intersectado por qv_i. Entonces, el vértice v_i y tal vez algunos de los vértices siguientes a v_i, en orden levógiro, no son visibles por q. Sea $v_{k-1}v_k$ la primer arista a partir de v_i en $bd(v_{i+1}, v_n)$ en orden levógiro tal que $v_{k-1}v_k$ es intersectado por $\overrightarrow{qv_{i-1}}$. Sea z el punto de intersección. Ya que ningún vértice de $bd(v_i, v_{k-1})$ es visible por q y, por lo tanto, z es el siguiente punto

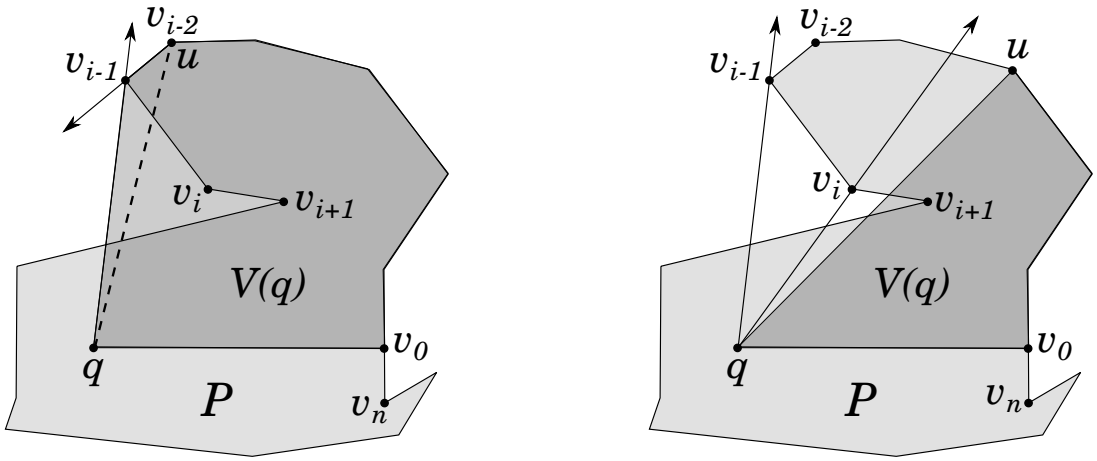
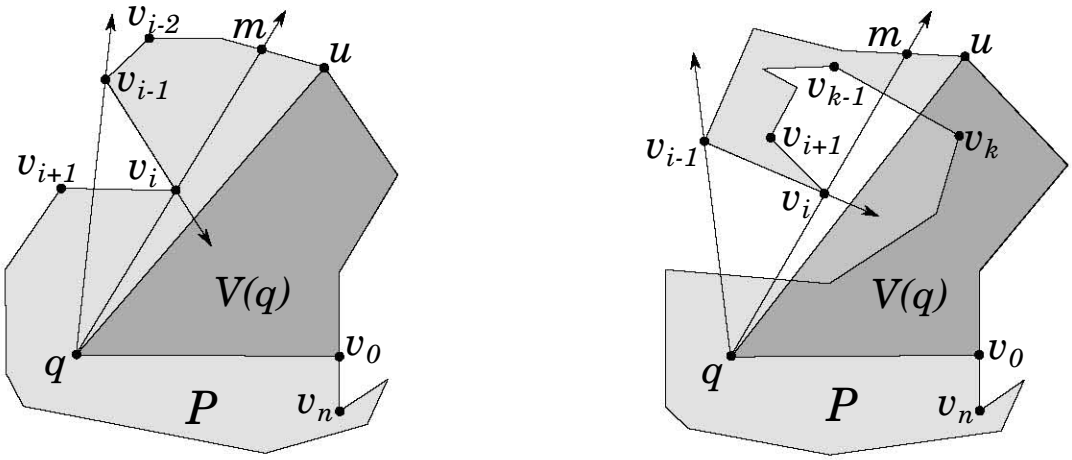


Figura 2.2: (a) La arista $v_{i-1}v_i$ se intersecta con uq. (b) La arista $v_{i-1}v_i$ no se intersecta con uq.

a partir de v_{i-1} en $bd(v_{i-1}, v_n)$ que es visible por q, entonces v_iz es una arista construida de $V(q)$ donde q, v_{i-1} y z son puntos colineales. Insertar z y v_k en la pila y hacer a v_{k+1} el nuevo v_i.

- **Caso 2b.** v_i se encuentra a la izquierda de $\overrightarrow{v_{i-2}v_{i-1}}$ (ver figura 2.2(a)) o qv_i es intersectado por $bd(v_{i+1}, v_n)$. Entonces, el vértice v_i y tal vez algunos vértices anteriores a v_i, en orden levógiro, que estan en la pila no son visibles por q. Sacar el elemento v_i de la pila insertado en el caso 1. Sea u el vértice en la cabeza de la pila. La arista $v_{i-1}v_i$ es llamada una *arista de avance*. Mientras $v_{i-1}v_i$ se intersecte con uq y u sea un vértice de P, eliminar un elemento en la pila, este ciclo es llamado *retroceso*. Note que los vértices eliminados no son visibles por q, ya que su visibilidad es bloqueada por la arista de avance. Después de ejecutar el retroceso anterior, pueden ocurrir dos situaciones:
  - **(i)** $v_{i-1}v_i$ no es intersectado por uq (ver figura 2.2(b))
  - **(ii)** $v_{i-1}v_i$ es intersectado por uq (ver figura 2.4)

  Para la primera situación (i), el procedimiento decide si es necesario hacer otro retroceso (ver figuras 2.2(b) y 2.3) tomando en cuenta las siguientes condiciones: si v_{i+1} se encuentra a la derecha de $\overrightarrow{qv_i}$ (ver figura 2.2(b)), el retroceso toma v_iv_{i+1} como arista de avance, si v_{i+1} se encuentra a la izquierda de $\overrightarrow{qv_i}$ (ver figura 2.3), sea m el punto de intersección de $\overrightarrow{qv_i}$ con la arista de $bd(P)$ conteniendo a u, si v_{i+1} se encuentra a la derecha de $\overrightarrow{v_{i-1}v_i}$ entonces el retroceso termina (ver figura 2.3(a)). Insertar m y v_i en la pila y hacer v_{i+1} el nuevo v_i. Si v_{i+1} se encuentra a la izquierda de $\overrightarrow{v_{i-1}v_i}$ (ver figura 2.3(b)), recorrer $bd(v_{i+1}, v_n)$ a partir de v_{i+1} hasta que un vértice v_k sea encontrado, tal que la arista $v_{k-1}v_k$ se intersecte con mv_i. Continuar el retroceso con $v_{k-1}v_k$ como la arista de avance.



(a) (b)

Figura 2.3: (a) El retroceso termina insertando m y v_i en la pila. (b) El retroceso continúa con $v_{k-1}v_k$ como la arista de avance.

Para la segunda situación (ii), u no es un vértice de P (ver figura 2.4), entonces sea w el vértice inmediato debajo de u en la pila (observe que uw es una arista 'construida' calculada en el caso 2a), sea p el punto de intersección de uq y $v_{i-1}v_i$, si $p \in qw$ (ver figura 2.4(a)), q no puede ver a u ni a w ya que $v_{i-1}v_i$ bloquea la visibilidad, cuando esto ocurra eliminamos un elemento de la pila, continuamos el retroceso y hacemos $v_{i-1}v_i$ la arista de avance; en otro caso, $v_{i-1}v_i$ es intersectado por uw ya que p pertenece a uw (ver figura 2.4(b)). Recorrer $bd(v_{i+1,v_n})$ a partir de v_{i+1} hasta que un vértice v_k sea encontrado tal que la arista $v_{k-1}v_k$ se intersecte con wp en algún punto z, así que, $bd(w, z)$, excepto w y z, no son visibles por q. Eliminamos un elemento de la pila e insertarmos z y v_k en la pila, luego hacemos v_{k+1} el nuevo v_i.

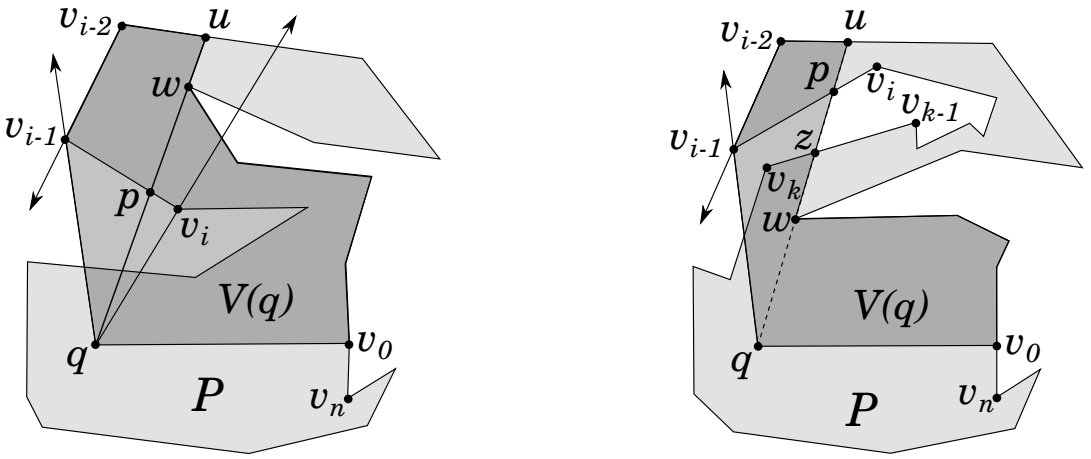
Se puede ver que la complejidad del algoritmo es $O(n)$ ya que el rayo que tomamos como manecilla solo pasa una vez por cada vértice de P, cuando ejecuta el retroceso pasa de nuevo por cada vértice y en el peor caso pasa por todos una segunda vez lo cual no afecta la complejidad.

El algoritmo 2.1 y 2.2 presenta los pasos para calcular $V(q)$. Como antes, se asume que v_0 es el punto más cercano en $bd(P)$ a la derecha de q y el vértice v_1 es el siguiente vértice de v_0 en orden levógiro. Para iniciar insertamos v_0 en la pila y hacemos $i = 1$.

Algorithm 2.1 *Visibilidad de un punto - Parte 1/2*

Entrada: Un polígono P y un punto q dentro del polígono
Salida: Polígono de visibilidad de q en P

1: Insertar v_i en la pila
2: Hacer $i \leftarrow i+1$
3: **if** $i = |V(P)| + 1$ **then**
4: **goto** línea 44
5: **end if**
6: **if** v_i se encuentra a la izquierda de $\overrightarrow{qv_{i-1}}$ **then**
7: **goto** línea 1
8: **end if**
9: **if** v_i se encuentra a la derecha de $\overrightarrow{qv_{i-1}}$ **y** a la derecha de $\overrightarrow{v_{i-2}v_{i-1}}$ **then**
10: Buscar desde v_{i+1} en orden de levógiro hasta encontrar un vértice v_k tal que $v_{k-1}v_k$ intersecte con qv_{i-1}. Hacer z el punto de intersección.
11: Insertar z en la pila
12: $i \leftarrow k$
13: **goto** línea 1
14: **end if**



(a) (b)

Figura 2.4: (a) La arista $v_{i-1}v_i$ no se intersecta con la arista construida uw. (b) La arista $v_{i-1}v_i$ se intersecta con uw en el punto p y la arista $v_{k-1}v_k$ se intersecta con pw en el punto z.

Algorithm 2.2 *Visibilidad de un punto - Parte 2/2*

15: **if** v_i se encuentra a la derecha de $\overrightarrow{qv_{i-1}}$ **y** a la izquierda de $\overrightarrow{v_{i-2}v_{i-1}}$ **then**
16: Sea u que denota el elemento en el tope de la pila.
17: Sacar un elemento de la pila.
18: **while** $u \in V(P)$ **and** $v_{i-1}v_i$ intersecte con uq **do**
19: Sacar un elemento de la pila.
20: **end while**
21: **end if**
22: **if** $v_{i-1}v_i$ no se intersecte con uq **then**
23: **if** v_{i+1} se encuentra a la derecha de $\overrightarrow{qv_i}$ **then**
24: Hacer $i \leftarrow i+1$
25: **goto** línea 17
26: **end if**
27: Sea m el punto de intersección de $\overrightarrow{qv_i}$ y la arista que contiene a u
28: **if** v_{i+1} se encuentra a la derecha de $\overrightarrow{v_{i-1}v_i}$ **then**
29: Insertar m en la pila
30: **goto** línea 1
31: **end if**
32: Buscar desde v_{i+1} en orden levógiro hasta encontrar un vértice v_k tal que $v_{k-1}v_k$ intersecte con mv_i.
33: $i \leftarrow k$
34: **goto** línea 17
35: **end if**
36: Sea w el vértice debajo de u en la pila.
37: Sea p el punto de intersección entre $v_{i-1}v_i$ y uq.
38: **if** $p \in qw$ **or** q, w, u son no colineales **then**
39: Sacar un elemento de la pila.
40: **goto** línea 17
41: **end if**
42: Buscar desde v_{i+1} en orden levógiro hasta encontrar un vértice v_k tal que $v_{k-1}v_k$ se intersecte con wp.
43: Insertar el punto de intersección en la pila.
44: $i \leftarrow k$
45: **goto** línea 1
46: **return** los elementos de la pila como vértices del polígono de visibilidad.

2.2. Polígono de visibilidad completa de una región convexa en un polígono simple

En esta sección se muestra el algoritmo dado por *Gosh* en [6] para calcular el *polígono de visibilidad completa* $V_c(C)$ de una región convexa C que se encuentra dentro de un polígono simple P en tiempo $O(n+k)$ donde n es el número de vértices de P y k el número de vértices del cierre convexo de C.

Asumimos que C es dado como una secuencia circular de vértices, por lo que a partir de aquí se verá a C como un polígono convexo donde sus vértices $c_1, c_2, \ldots, c_k$ son dados de forma dextrógira. También asumimos que los vértices de P están dados de forma dextrógira.

Para cada punto $u \in P - C$ y un punto $v \in bd(C)$, se dice que el segmento uv es una **tangente** de C si el rayo $\overrightarrow{uv}$ se intersecta con $bd(C)$ solo en el punto v.

Sea S el conjunto que contiene a todos los vértices de $bd(C)$. Para calcular $V_c(C)$, mostramos con el siguiente lema que es suficiente encontrar el polígono de visibilidad completa de S dentro de P.

Lema 2.2.1. Sea S el conjunto de vértices de C. Si $V_c(C)$ y $V_c(S)$ son los polígonos de visibilidad completa de C y S respectivamente dentro de P entonces $V_c(C) = V_c(S)$.

Demostración. Para probar la iguldad, debemos mostrar que $V_c(C) \subseteq V_c(S)$ y $V_c(S) \subseteq V_c(C)$. La primera proposición se demuestra con la siguiente observación. Es fácil ver que los vértices de S son los puntos más alejados que están dentro de C, entonces algún vértice en S puede alcanzar a ver la misma área que cualquier punto que está dentro de C, aún más, podemos afirmar que estos puntos tienen mucha más visibilidad que los puntos que están dentro del aréa C y por lo tanto $V_c(C) \subseteq V_c(S)$. Ahora probaremos que $V_c(S) \subseteq V_c(C)$. Sea u cualquier punto dentro de $V_c(S) - C$ (ver figura 2.5), mostraremos que u está dentro de $V_c(C)$. Sea uc_i y uc_j las dos tangentes de u a C donde c_i y c_j pertenecen a S; mostrando que estas tangentes están dentro de $V_c(C)$ afirmamos que el interior del pico formado por estas tangentes también se encuentra dentro de $V_c(C)$ y por lo tanto u será visible por C o dicho en otras palabras $u \in V_c(C)$. Ahora solo queda confirmar que las tangentes uc_i y uc_j también pertenecen a $V_c(C)$, esto se comprueba fácilmente argumentando lo siguiente: debido a que los vértices c_i y c_j se encuentran en C y sabemos que ambos vértices ven al punto u entonces las tangentes uc_i y uc_j pertenecen a $V_c(C)$. □

Con fines prácticos usaremos $V_c(C)$ para hacer referencia al polígono de visibilidad completa de C pero en realidad estaremos calculando $V_c(S)$ el polígono de visibilidad completa de sus vértices. Además, para evitar confusión dejaremos de usar S y usaremos C, diremos que los vértices $c_1, c_2, \ldots, c_k$ están en C.

Un algoritmo relativamente trivial para calcular $V_c(C)$ en tiempo $O(kn)$ es el siguiente: calcular el polígono de visibilidad $V(c_1)$ dentro de P, luego dentro de $V(c_1)$ calculamos $V(c_2)$,hacemos lo mismo con $V(c_3)$ dentro de $V(c_2)$ y seguimos sucesivamente hasta terminar con $V(c_k)$ dentro de $V(c_{k-1})$; cuando terminemos, $V(c_k)$ será el polígono de visibilidad completa $V_c(C)$. Ya que calcular un polígono de visibilidad para cada vértice de C toma

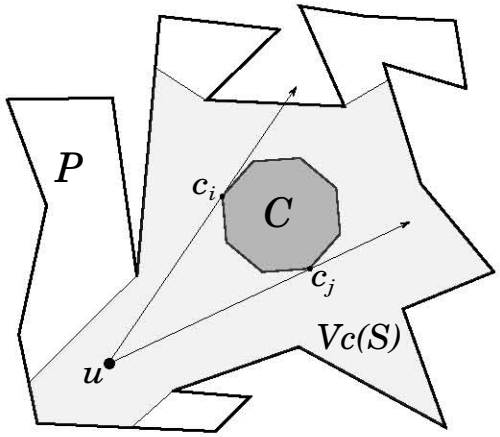


Figura 2.5: Las dos tangentes uc_i y ucj estan totalmente contenidas en P.

tiempo $O(n)$, este algoritmo toma $O(kn)$. Es posible mejorar este tiempo a $O(n+k)$ con el siguiente algoritmo.

En resumen, el algoritmo genera $V_c(C)$ con ayuda de rayos (formados a partir de un vértice $c_j \in C$ pasando por $v_i \in P$ de tal forma que no se intersecten con alguna arista de C). Estos rayos se van creando ordenadamente de forma dextrógira por cada vértice c_j de C y cada vértice v_i de P incrementando i o j. Cuando se termina de hacer el recorrido dextrógiro con los rayos, se hace un nuevo recorrido de manera similar pero ahora de forma levógira. Antes que nada se inicializa $V_c(C) = P$. Luego, cada vez que el rayo encuentra un pico en P, donde existe una parte de P que no es visible por C, simplemente se elimina esta parte en $V_c(C)$ y se coloca una arista construida que va del vértice que está en la punta del pico hasta el primer punto de $bd(P)$ que es intersectado por el rayo. Haciendo ambos recorridos, se garantiza que $V_c(C)$ es el polígono de visibilidad completa de C dentro de P.

Ahora explicamos detalladamente los pasos del algoritmo. Sean v_ic_i y v_ic_j las tangentes de un vértice $v_i \in P$ al conjunto convexo C donde c_i y c_j son vértices de C. El segmento v_ic_i es llamado *tangente izquierda* de v_i si todos los puntos de C estan a la izquierda de $\overrightarrow{v_ic_i}$. De manera similar, el segmento v_ic_j es llamado *tangente derecha* de v_i si todos los puntos de C estan a la derecha de $\overrightarrow{v_ic_j}$. Sea v_c (o v_{cc}) el punto en $bd(P)$ más cercano a c_1 entre los puntos de intersección de $\overrightarrow{c_kc_1}$ (o $\overrightarrow{c_2c_1}$ respectivamente) con $bd(P)$ (ver figura 2.6).

En el siguiente lema se presenta la idea principal del algoritmo. Ya que para este algoritmo es necesario comenzar con $V(c_1)$. Por simplicidad en la notación, calculamos $V(c_1)$ y lo tomaremos como si fuera P, es decir, $V(c_1)$ será denotado por P, solo para este algoritmo.

Lema 2.2.2. Un vértice v_i de P no es visible para todos los vértices de C si y solo si la tangente izquierda de v_i se intersecta con $bd(v_c, v_{i-1})$ o si la tangente derecha de v_i se

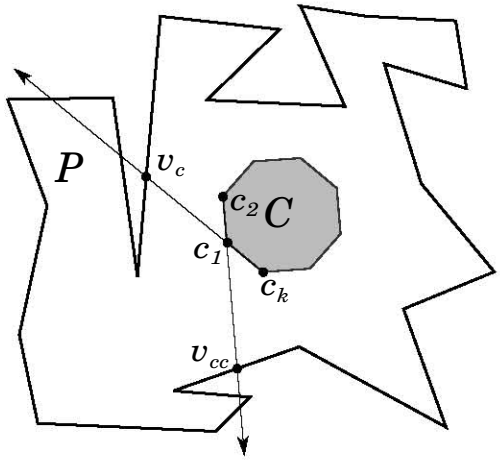


Figura 2.6: Los puntos v_c y v_{cc} son los más cercanos a c_1 de los puntos de intersección entre los rayos y bdP.

intersecta con $bd(v_{i+1}, v_{cc})$.

Demostración. Sean v_ic_i y v_ic_j las tangentes izquierda y derecha de p_i respectivamente a C. Si $bd(v_c, v_{i-1})$ (o $bd(v_{i+1}, v_{cc})$) intersecta con la tangente izquierda (o derecha respectivamente) de v_i, entonces v_i no es visible desde v_i (o v_j, respectivamente) y por lo tanto v_i no es visible para todos los vértices de C (ver figura 2.7).

Recíprocamente se muestra que si $bd(v_c, v_{i-1})$ o $bd(v_{i+1}, v_{cc})$ no son intersectados por las tangentes izquierda o derecha respectivamente de v_i, entonces v_i es visible para todos los vértices de C. Observe que si $v_i \in bd(v_c, v_{cc})$, entonces $bd(v_c, v_{i-1})$ (o $bd(v_{i+1}, v_{cc})$) no puede ser intersectado por la tangente derecha (o izquierda respectivamente) de v_i, ya que v_i es visible desde v_1. De manera similar, si $v_i \in bd(v_{cc}, v_c)$, entonces $bd(v_{i+1}, v_c)$ (o $bd(v_{cc}, v_{i-1})$) no puede ser intersectado por la tangente izquierda (o derecha, respectivamente) de v_i, ya que v_i es visible desde c_1 (ver figura 2.7). Esto implica que si $bd(v_c, v_{i-1})$ (o $bd(v_{i+1}, v_{cc})$) no es intersectado por v_ic_i (o v_ic_j, respectivamente), entonces las dos tangentes de v_i estan dentro de P. Por lo tanto, v_i es visible por todos los vértices de C. □

Ahora se presenta el algoritmo *Comp-Visibility* para calcular el polígono de visibilidad completa de la región C dentro de P, (ver algoritmo 2.3). Primero se calcula el polígono de visibilidad de c_1 en P: $V(c_1)$, por el algoritmo de Lee en [10] o en el capítulo 2.1. Como $V_c(C) \subseteq V(c_1)$, el resto del algoritmo trabajará solamente sobre el polígono $V(c_1)$. Asumimos nuevamente que P es completamente visible desde c_1 en el algoritmo. Una lista doblemente ligada VP guarda la frontera de $V_c(C)$ que es inicializada con $bd(P)$. El algoritmo recorre $bd(P)$ y C de forma dextrógira empezando en v_{c+1} (el vértice siguiente de v_c) en $bd(P)$ y c_1 en C. Sean v_i y c_j los vértices que son tomados en cada iteración.

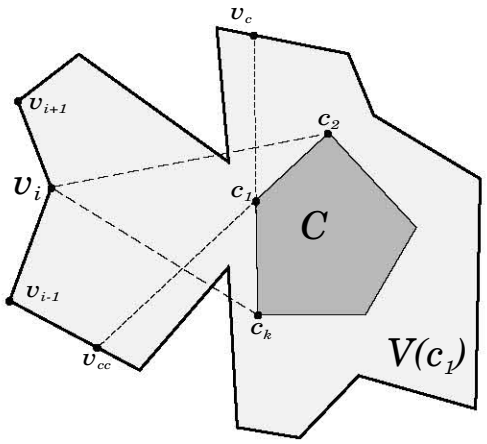


Figura 2.7: Las tangentes v_ic_2 y v_ic_k se intersectan con $bd(v_{i+1}, v_{cc})$ y $bd(v_c, v_{i-1})$, respectivamente, por lo tanto v_i no es visible para todos los vértices de C.

Algorithm 2.3 *Comp-Visibility*

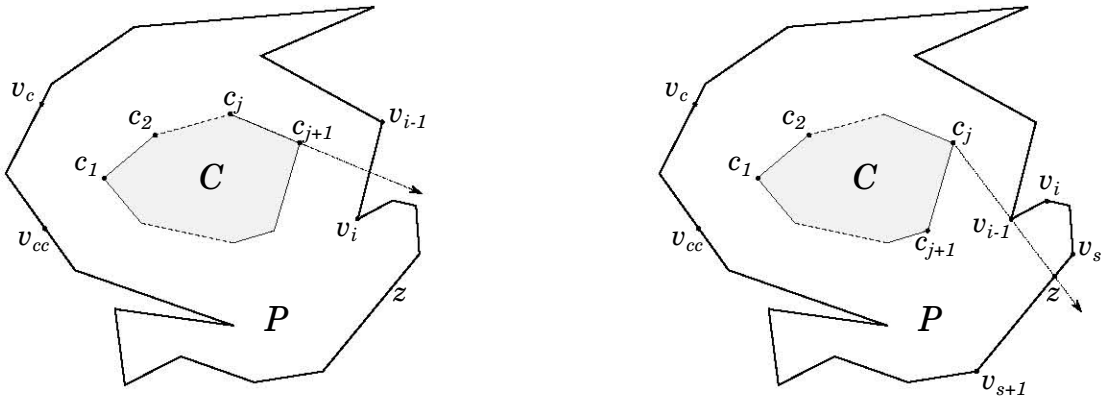
Entrada: Un polígono P y una región C dentro de P.
Salida: Una secuencia de aristas que forman el polígono de visibilidad completa de C.

1: Si $v_i = v_c$, entonces terminar.
2: Mientras v_i esté a la derecha de $\overrightarrow{c_jc_{j+1}}$ hacer $j \leftarrow j + 1$ (ver figura 2.8(a)).
3: Si v_i esta a la izquierda de $\overrightarrow{c_jv_{i-1}}$ entonces asignar i a s e incrementar s hasta encontrar una arista v_sv_{s+1} que sea intersectada por $\overrightarrow{c_jv_{i-1}}$ en algún punto z (ver figura 2.8(b)). Reemplazar $(v_i, v_{i+1}, \ldots, v_s)$ por z en VP. Luego hacer $v_i \leftarrow z$.
4: Si v_i esta a la derecha o sobre el rayo $\overrightarrow{c_jv_{i-1}}$, entonces $i \leftarrow i + 1$.

Una vez que el recorrido dextrógiro ha terminado, el algoritmo explora los vértices restantes en VP comenzando en v_{cc} y los vértices en C comenzando en c_1, ahora de forma levógira, haciendo el mismo procedimiento. Después de estos dos recorridos, VP contiene la frontera del polígono de visibilidad completa de C en P.

Teorema 2.2.1. El algoritmo *Comp-visibility* calcula correctamente el polígono de visibilidad completa de C en P en tiempo $O(n + k)$

Demostración. Para probar que el algoritmo es correcto es suficiente mostrar que, por el lema 2.2.2, *Comp-visibility* elimina un vértice v_i de VP, si y solo si, la tangente izquierda de v_i intersecta $bd(v_c, v_{i-1})$ o si la tangente derecha de v_i intersecta $bd(v_{i+1}, v_{cc})$. Para cada vértice p_i de VP, el recorrido dextrógiro de Comp-visibility verifica la intersección entre $bd(v_c, v_{i-1})$ y la tangente izquierda de v_i. Sea v_i y c_j los vértices actualmente bajo consideración por el recorrido dextrógiro. Se asume que para cada vértice v_s de $bd(v_c, v_{i-1})$ en VP, $bd(v_c, v_{s-1})$ no es intersectado por la tangente izquierda de v_s y $v_{i-1}c_j$ es la tangente izquierda de v_{i-1}. En el paso 2, j se incrementa en uno hasta que v_ic_j llegue a ser la tangente



(a) (b)

Figura 2.8: Recorrido del algoritmo Comp-Visibility

izquierda de v_i (ver figura 2.8(a)). Si la tangente izquierda de v_i hace intersección con $bd(v_c, v_{i-1})$, en el paso 3 se encuentra un punto z, siguiendo un recorrido dextrógiro sobre $bd(P)$ (ver figura 2.8(b)), para el cual la tangente izquierda no intersecte con $bd(v_c, z)$. Para el caso contrario en el paso 4 se toma el siguiente punto o vértice. Una vez que el recorrido dextrógiro ha terminado, la tangente izquierda de cada vértice o punto de VP se encuentra dentro de P. Análogamente al terminar el recorrido levógiro, la tangente derecha de cada vértice o punto de VP se encuentra dentro de P. Por lo tanto Comp-visibility calcula correctamente el polígono de visibilidad completa de C en P.

Ahora se analiza la complejidad en tiempo del algoritmo. Calcular el polígono de visibilidad de c_1 en P toma $O(n)$ [10]. Los puntos v_c y v_{cc} pueden ser calculados en $O(n)$, ya que tenemos que probar los n segmentos de P para saber cuales se intersectan con los rayos. Luego, el algoritmo pasa por los n vértices de P y los k vértices de C una vez de forma dextrógira y una vez de forma levógira. Por lo tanto la complejidad total en tiempo del algoritmo es $O(n + k)$. □

2.3. Polígono de visibilidad débil de una región en un polígono simple.

En esta sección se muestra el algoritmo dado por *Gosh* en [6] para calcular el *polígono de visibilidad débil* $V_d(C)$ de una región convexa C que se encuentra dentro de un polígono simple P en tiempo $O(n+k)$ donde n es el número de vértices de P y k el número de vértices del cierre convexo de C.

Como es mencionado por *Gosh* en [6] este algoritmo también calcula $V_d(C)$ donde C es una región no convexa. Simplemente transformamos a C en una región convexa tomando su cierre convexo.

Asumimos que C es dado como una secuencia circular de vértices, por lo que a partir de aquí se verá a C como un polígono convexo donde sus vértices c_1, c_2, ..., c_k son dados de forma dextrógira. También asumimos que los vértices de P están dados de forma dextrógira.

Con el fin de encontrar el polígono de visibilidad débil de C dentro de P, basta calcular el polígono de visibilidad débil de $bd(C)$ (la frontera de C) dentro de P, lo cual es demostrado con el siguiente lema. Por simplicidad denotados a $bd(C)$ como F, observe que F contiene las aristas y los vértices de C.

Lema 2.3.1. Sean $V_d(C)$ y $V_d(F)$ los polígonos de visibilidad débil de C y F en P respectivamente, entonces $V_d(C) = V_d(F)$.

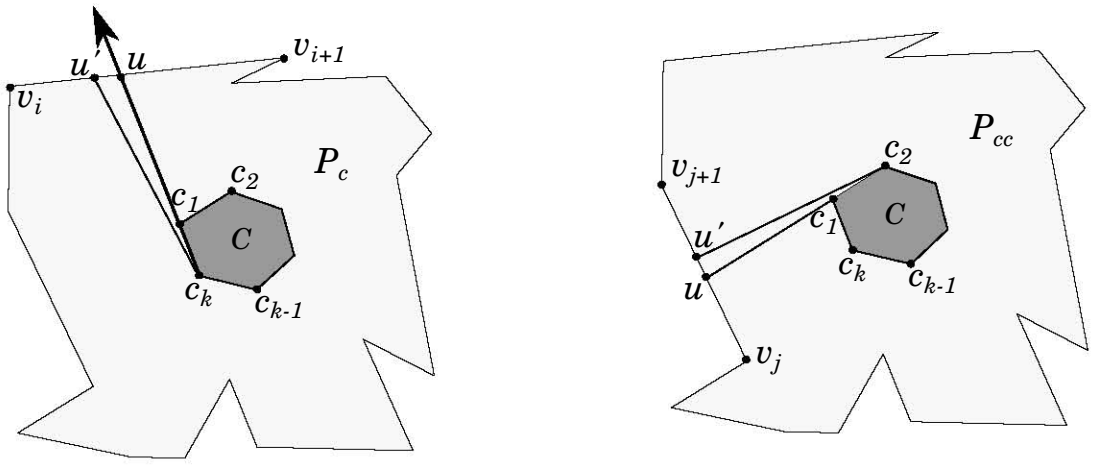
Demostración. Para probar la equivalencia, debemos mostrar que $V_d(C) \subseteq V_d(F)$ y también que $V_d(F) \subseteq V_d(C)$. Para la primera proposición, es fácil ver que todos los puntos que están en F también se encuentran en C por lo tanto $V_d(F) \subseteq V_d(C)$. Ahora probamos que $V_d(C) \subseteq V_d(F)$. Sea u un punto de $V_d(C) - C$. Se mostrará que $u \in V_d(F)$. Como $u \in (V_d(C) - C)$, existe un punto $z \in C$ tal que u es visible desde z. Si z pertenece a F, la proposición se cumple. De otro modo, si z está en C entonces uz se intersectará con F en algún punto z' por lo que u será visible desde z'. Por lo tanto $u \in V_d(F)$. □

Para más fácil usaremos $V_d(C)$ haciendo referencia al polígono de visibilidad débil de C pero en realidad estaremos calculando $V_d(F)$ el polígono de visibilidad débil de su frontera.

El lema anterior sugiere un algoritmo que calcula $V_d(C)$ en tiempo $O(kn)$. Simplemente se calcula el polígono de visibilidad débil de cada arista de F y se toma la unión de estos polígonos. Calcular el polígono de visibilidad débil de una arista toma tiempo $O(n)$ con el algoritmo de Guibas [12] (una vez que $P - C$ es triangulado) y como se tienen k aristas en C este algoritmo toma $O(kn)$.

A continuación se describe el algoritmo *Weak-Visibility* que toma tiempo $O(n+k)$ en el peor caso, sin contar el tiempo que toma calcular la triangulación de $P - C$.

Asumimos que C es un agujero convexo dentro de P. Se construyen dos nuevos polígonos P_c y P_{cc} a partir de P (los cuales son polígonos sin agujeros) de la siguiente forma. Sea u el punto más cercano a c_1 de los puntos de intersección entre el rayo $\overrightarrow{c_k c_1}$ y $bd(P)$, se denota la arista de $bd(P)$ que es intersectada por el rayo como $v_i v_{i+1}$ (ver figura 2.9(a)). Crear un vértice u' sobre $v_i v_{i+1}$ arbitrariamente cercano a u en sentido levógiro. Cortamos



(a) (b)

Figura 2.9: Los polígonos P_c y P_{cc} creados a partir del polígono P con el agujero convexo C.

la región $P - C$ por uc_1 y $u'c_1$ para obtener P_c, haciendo una analogía tomamos la región $P-C$ como si se tratara de un trozo de papel y con unas tijeras hacemos un corte siguiendo el segmento uc_1 y otro corte siguiendo el segmento $u'c_1$, con esto creamos el polígono P_c = (c_1, u, p_{i+1}, p_{i+2}, ..., p_i, u', c_k, c_{k-1}, ..., c_1). Note que el *camino más corto* de c_1 a u' en P_c es (c_1, c_2, ..., c_k, u'). Luego, obtenemos P_{cc} cortando la región $P - C$ de manera similar. Sea v el punto más cercano a c_1 de los puntos de intersección entre el rayo $\overrightarrow{c_2c_1}$ y $bd(P)$, se denota la arista de $bd(P)$ que es intersectada por el rayo como v_jv_{j+1} (ver figura 2.9(b)). Crear un punto v' sobre v_jv_{j+1} arbitrariamente cercano a v en sentido dextrógiro. Cortar la región $P - C$ por vc_1 y $v'c_1$ para obtener P_{cc}, donde P_{cc} = (c_2, v', v_{j+1}, v_{j+2}, ..., v_j, v, c_1, c_k, c_{k-1}, ..., c_2). Note que el *camino más corto* de c_1 a v' es (c_1, c_k, ..., c_2, v').

Notación:

El *árbol de caminos más cortos* de un polígono, con raíz en un vértice u, es la unión de todos los caminos más cortos de u a todos los vértices del polígono.

Sean SPT_c y SPT_{cc} los árboles de caminos más cortos de P_c y P_{cc} con c_1 como la raíz. El algoritmo triangula P_c y P_{cc} por el algoritmo de Tarjan y Van Wyk en [16] y luego calcula SPT_c y SPT_{cc} por el algoritmo de Guibas en [12]. Debido a que el algoritmo se basa en la dirección en que un vértice se encuentra a partir de otro anterior en algún camino del árbol, se menciona lo siguiente para conseguir que siempre se tenga una dirección izquierda o derecha. Si tres vértices consecutivos a, b y c en cualquier camino de SPT_c o SPT_{cc} son colineales, es decir que, b es un hijo de a y c es un hijo de b, el árbol es modificado haciendo a b y a c como hijos de a. Por lo tanto se asume que no hay tres vértices consecutivos en un camino de SPT_c o SPT_{cc} que sean colineales. Se denota el camino más corto de un vértice a a uno b en P_c y P_{cc} por $SP_c(a, b)$ y $SP_{cc}(a, b)$, respectivamente. En los siguientes lemas, se presenta la idea principal usada en el algoritmo.

Lema 2.3.2. Si un vértice v_s de P es visible por algún punto de F, entonces $SP_c(c_1, v_s)$

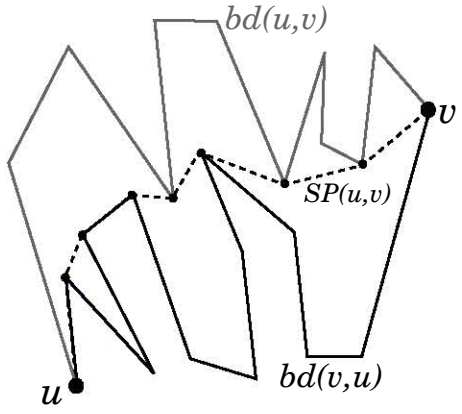


Figura 2.10: El $SP(u, v)$ solamente hace giros a la izquierda en los vértices de $bd(u, v)$ y giros a la derecha en los vértices de $bd(v, u)$.

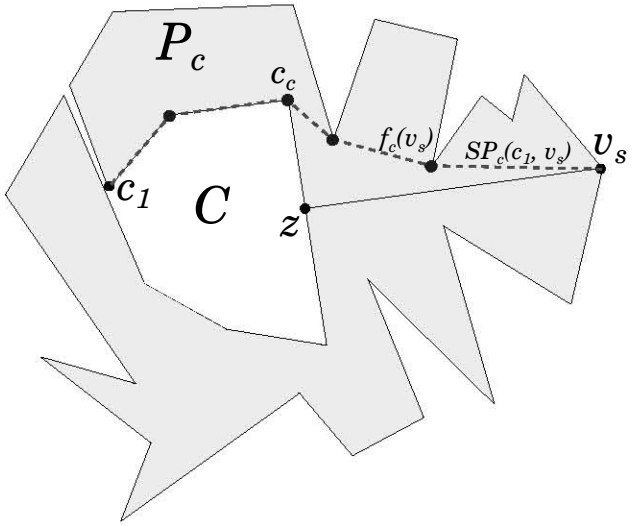
hace un giro a la izquierda en cada vértice de P y $SP_{cc}(c_1, v_s)$ hace un giro a la derecha en cada vértice de P.

Demostración. Observe que el camino más corto entre dos vértices u y v en un polígono solamente hace giros a la izquerda en los vértices de $bd(u, v)$ y giros a la derecha solo en los vértices de $bd(v, u)$ (ver figura 2.10).

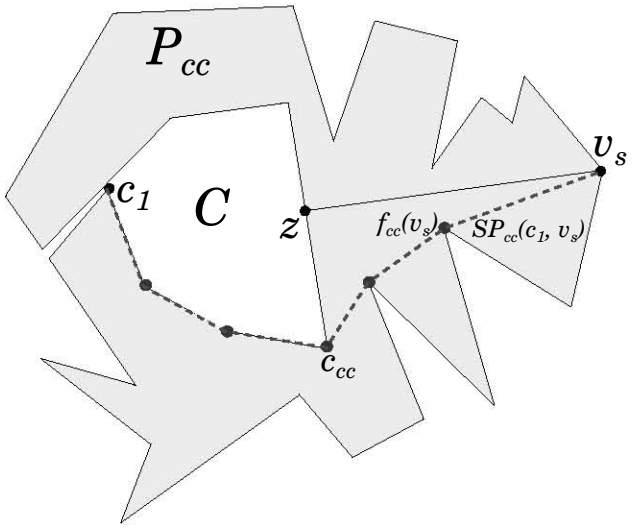
Si v_s es visible desde un punto z en F, entonces $SP_c(c_1, v_s)$ no puede intersectar al segmento zv_s (ver figura 2.11(a)). Por lo tanto, los vértices de P en $SP_c(c_1, v_s)$ se encuentran en $bd(u, v_s)$ y la primera proposición se demuestra por la observación de antes. Analogamente, $SP_{cc}(c_1, v_s)$ siempre hace un giro a la derecha en cada vértice de P que esté en el camino (ver figura 2.11(b)) por lo que la segunda proposición se demuestra de la misma forma. □

Lema 2.3.3. Si $SP_c(c_1, v_s)$ hace un giro a la izquierda en cada vértice de P y $SP_{cc}(c_1, v_s)$ hace un giro a la derecha en cada vértice de P, entonces v_s es visible desde algún punto de F.

Demostración. Sean $f_c(v_s)$ y $f_{cc}(v_s)$ los padres de v_s en $SP_c(c_1, v_s)$ y $SP_{cc}(c_1, v_s)$ respectivamente. Sean c_c y c_{cc} los últimos vértices de F que están en $SP_c(c_1, v_s)$ y $SP_{cc}(c_1, v_s)$ respectivamente, atravesando desde c_1 hasta v_s. Ya que $SP_c(c_1, v_s)$ hace un giro a la derecha en c_c y más tarde hace un giro a la izquierda en cada vértice de P, c_c se encuentra a la derecha del rayo $\overrightarrow{v_s f_c(v_s)}$ (ver figura 2.11(a)). Analogamente, ya que $SP_{cc}(c_1, v_s)$ hace un giro a la derecha en cada vértice de P, c_{cc} se encuentra a la izquierda del rayo $\overrightarrow{p_s f_{cc}(v_s)}$. Así que, c_c y c_{cc} se encuentran en lados opuestos del pico formado por los rayos $\overrightarrow{v_s f_c(v_s)}$



(a)



(b)

Figura 2.11: Los caminos más cortos $SP_c(c_1, v_s)$ y $SP_{cc}(c_1, v_s)$ no pueden intersectar con el segmento zv_s.

y $\overrightarrow{v_s f_{cc}(v_s)}$. Por lo que el pico se intersecta con F. Por lo tanto v_s es visible desde algún punto de F dentro del pico. □

En el algoitmo 2.4 se muestran los pasos del algoritmo *Weak-Visibility* para calcular el polígono de visibilidad débil de C en P.

Algorithm 2.4 *Weak-Visibility*

Entrada: Un polígono P y una región C dentro de P.
Salida: Polígono de visibilidad débil de C dentro de P.

1: Obtener los polígonos sin agujeros P_c y P_{cc} cortando la región $P - C$.
2: Triangular P_c y P_{cc} por el algoritmo de Tarjan y Van Wyk en [16].
3: Calcular SPT_c y SPT_{cc} ambos con raíz en c_1 por el algoritmo de Guibas en [12].
4: Sea VP una lista doblemente ligada la cual contendrá los vértices de P débilmente visibles desde F.
5: Inicializar $VP \leftarrow bd(P)$.
6: Hacer un recorrido primero en profundidad en SPT_c (o SPT_{cc}) para obtener la dirección del giro en cada vértice v_i de P en el camino.
7: Si algún camino hace un giro a la derecha (o izquierda respectivamente) en v_i, eliminar los vértices de VP que son descendientes de v_i en SPT_c (o SPT_{cc} respectivamente).
8: Sea v_j el descendiente de v_i en SPT_c (o SPT_{cc}) con el subíndice j más pequeño (o más grande respectivamente) (ver figura 2.12).
9: Sea z el punto de intersección entre $v_{j-1}v_j$ (o v_jv_{j+1}) y el rayo trazado desde v_i' el padre de v_i atravezando v_i, es decir $\rightarrow v_i'v_i$.
10: Insertar z en VP en medio de v_j y v_{j-1} (o v_{j+1} respectivamente).

Ahora se analizará la complejidad en tiempo del algoritmo. El paso 1 requiere tiempo $O(n)$, ya que el tamaño de P_c y P_{cc} es $n + k + 2$. En el paso 3, SPT_c y SPT_{cc} pueden ser calculados en tiempo $O(n + k)$. En los pasos 4 al 10, cada vértice de SPT_c y SPT_{cc} es analizado una sola vez y las operaciones realizadas toman tiempo constante. Así que, la complejidad total en tiempo del algoritmo es $O(n + k)$ sin tomar en cuenta el tiempo requerido para la triangulación de $P - C$ del paso 2.

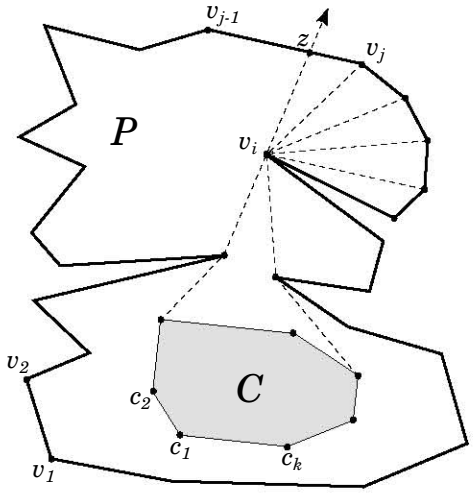


Figura 2.12: El vértice v_j es el descendiente de v_i en SPT_c con el subíndice j más pequeño.

2.4. Algoritmo para calcular la intersección de dos polígonos

Sean P y Q dos polígonos simples con n_1 y n_2 vértices respectivamente. En esta sección se presenta un algoritmo para calcular la intersección ($P \cap Q$) de P y Q. La complejidad en tiempo del algoritmo es $O(n \log n + k \log n)$ que corresponde al algoritmo de Map Overlay descrito en [2], donde n es el número total de vértices de P y Q, es decir, $n{=}n_1 + n_2$ y k es el número de vértices creados por la intersección de las aristas de P y Q.

La entrada del algoritmo son dos polígonos P' y Q', los cuales son copias de P y Q respectivamente, con algunas modificaciones. Estas modificaciones se explican a continuación solo para el polígono P, por lo que, deberán ser agregadas en Q de la misma forma.

Se asume que los vértices de P están etiquetados con v_1, v_2, ..., v_{n_1} en orden levógiro y las aristas están etiquetadas con e_1, e_2, ..., e_{n_1} en el mismo orden, de tal manera que $e_i = v_i v_{i+1}$.

Creamos una copia exacta P' del polígono P y transformamos cada arista $e_i \in P'$ en una arista con dirección, de tal manera que todas las aristas del polígono sigan un recorrido en orden levógiro. Esto es, sea $e_i = v_i v_{i+1}$ entonces $destino(e_i) = v_{i+1}$ y $origen(e_i) = v_i$ (ver figura 2.13(a)). Para la arista $e_{n_1}{=}v_{n_1} v_1$, $origen(e_{n_1}) = v_{n_1}$ y $destino(e_{n_1}) = v_1$.

Sean e_j y e_i dos aristas dirijidas. Se dice que e_j es la arista *siguiente* de e_i si $origen(e_j)$ = $destino(e_i)$, es decir, el vértice origen de e_j es el vértice destino de e_i. Recíprocamente, se dice que e_i es la arista *anterior* de e_j.

Entonces, sea $siguiente(e_i)$ la arista siguiente de e_i, es decir, $siguiente(e_i) = e_{i+1}$. Se le llama *cara* a una porción del plano que está delimitada por aristas. Observe que un polígono simple P es una cara en el plano, por lo que de aquí en adelante cada polígono es visto como una cara. El exterior de P también es una cara. Sea $cara(e_i)$ la cara que se encuentra del lado izquierdo de la arista e_i.

Si C es una cara y e una arista dirigida tal que $cara(e) = C$ se dice que la cara de e

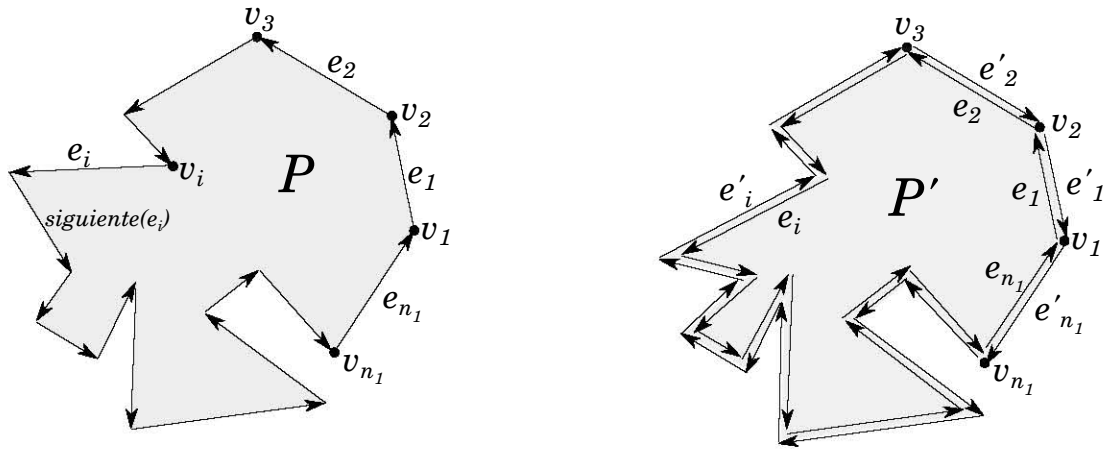


Figura 2.13: Construcción del polígono P'.

es C. Como todas las aristas de P' rodean la cara P' y llevan un orden levógiro, $cara(e_i) = P'$ para cada $e_i \in P'$.

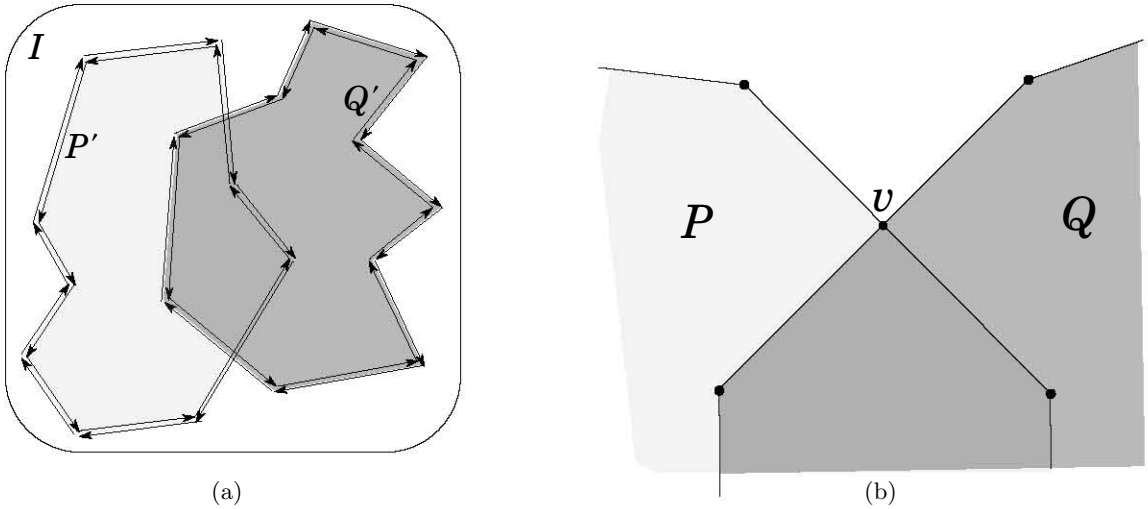
Hasta aquí se ha transformado P en un polígono P' tal que todas sus aristas son dirigidas y llevan un orden levógiro, ahora se crearán nuevas aristas dirigidas en P', las cuales llevarán un orden dextrógiro. Para cada arista $e_i \in P'$ crear una nueva arista e'_i copia de e_i y hacer $origen(e'_i) = destino(e_i)$, $destino(e'_i) = origen(e_i)$ y $siguiente(e'_i) = e'_{i-1}$ (ver imagen 2.13(b)). Ya que, todas las nuevas aristas de P' llevan un orden dextrógiro, la cara que se encuentra del lado izquierdo de cada e'_i es la cara exterior. Entonces, $cara(e'_i) = exterior$ para cada $e_i \in P'$. Se dice que e'_i es la inversa de e_i y se usa $inv(e_i)$ para referirse a la arista dirigida e'_i.

Después de haber agregado todas las modificaciones de P y obtenido el polígono modificado P' es necesario hacer lo mismo para Q creando un polígono modificado Q'.

Ahora se explican los pasos del algoritmo. Observe que los polígonos P' y Q' son conjuntos de aristas con vértices en el plano. Usaremos un conjunto I que contendrá todas las aristas de P' y Q' (ver figura 2.14(a)). Luego, usamos el algoritmo de *Bentley-Ottmann* publicado en [9] para encontrar todas las intersecciones entre los segmentos de P y Q. Asumimos que los vértices de P no son tomados en cuenta como puntos de intersección entre las aristas de P, análogo para Q. Note que aplicamos el algoritmo sobre la unión de las aristas de P y Q, no sobre las aristas de P' y Q'. Es necesario actualizar I cada vez que el algoritmo de *Bentley-Ottmann* encuentre un nuevo punto de intersección v. Se tienen los siguientes tres casos dependiendo del tipo de intersección que genera al punto v.

- **Caso 1.** v no es vértice de P ni de Q (ver figura 2.14(b)).

  Crear el vértice v en I. Sean e_P y e_Q las aristas de P' y Q' respectivamente, tales que $v \in e_P$ y $v \in e_Q$. Siendo $inv(e_P)$ y $inv(e_Q)$ las aristas inversas de e_P y e_Q



(a) (b)

Figura 2.14: (a) El conjunto I que contiene todas las aristas de P' y Q'. (b) El punto v no es un vértice de P ni de Q.

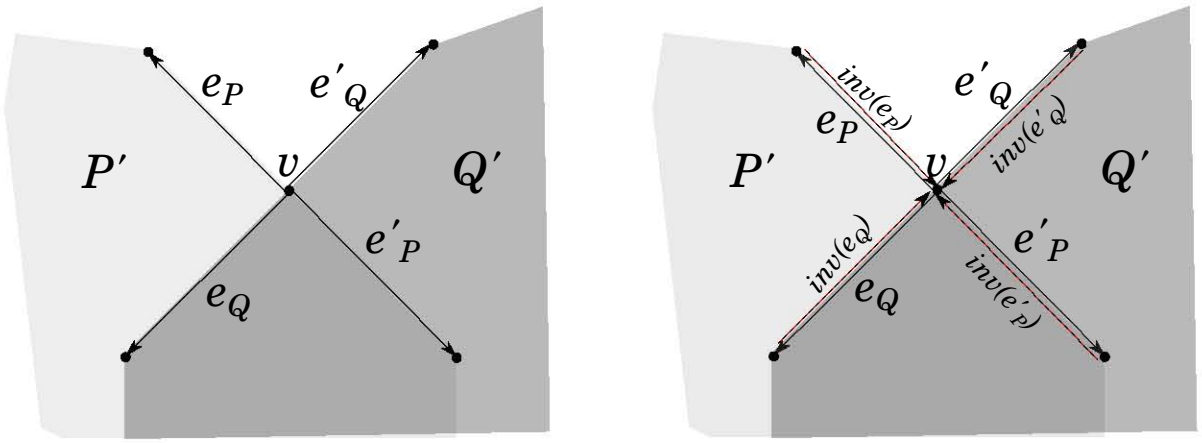
respectivamente. Observe que $inv(e_P)$ y $inv(e_Q)$ también contienen al punto v.

Actualizamos I de la siguiente forma. Modificamos el origen de e_P, e_Q, $inv(e_P)$ e $inv(e_Q)$ por v (ver figura 2.15(a)). Para evitar confución y simplificar la notación, las aristas $inv(e_P)$ e $inv(e_Q)$ ahora las llamaremos e'_P y e'_Q respectivamente. Posteriormente, creamos las aristas inversas de e_P, e_Q, e'_P y e'_Q de tal forma que el origen de $inv(e_P)$, $inv(e_Q)$, $inv(e'_P)$ e $inv(e'_Q)$ sea el destino de e_P, e_Q, e'_P y e'_Q; y el destino de cada arista $inv(e_P)$, $inv(e_Q)$, $inv(e'_P)$ e $inv(e'_Q)$ sea v (ver figura 2.15(b)). Para terminar actualizamos la arista siguiente de las antiguas aristas anteriores de e_P, e_Q, e'_P y e'_Q por $inv(e_P)$, $inv(e_Q)$, $inv(e'_P)$ e $inv(e'_Q)$ respectivamente.

Por consiguiente, también es necesario actualizar las aristas siguientes y las caras de las cuatro aristas nuevas $inv(e_P)$, $inv(e_Q)$, $inv(e'_P)$ e $inv(e'_Q)$. Ya que cada una de las cuatro nuevas aristas podría tener más de una arista siguiente tendremos que seleccionar solo una haciendo el procedimiento siguiente. Sea e la arista dirigida con múltiples aristas siguientes y sea c el destino de e y el origen de las aristas siguientes de e. Tomando como centro al punto c, rotar e como si fuera una manecilla de reloj, en sentido dextrógiro (ver figura 2.16(a)) y seleccionar como arista siguiente de e a la primera arista con la que se empalme e. Para actualizar la cara de cada una de las nuevas aristas, basta con copiar la cara de su arista siguiente. Terminando la actualización de I para este caso.

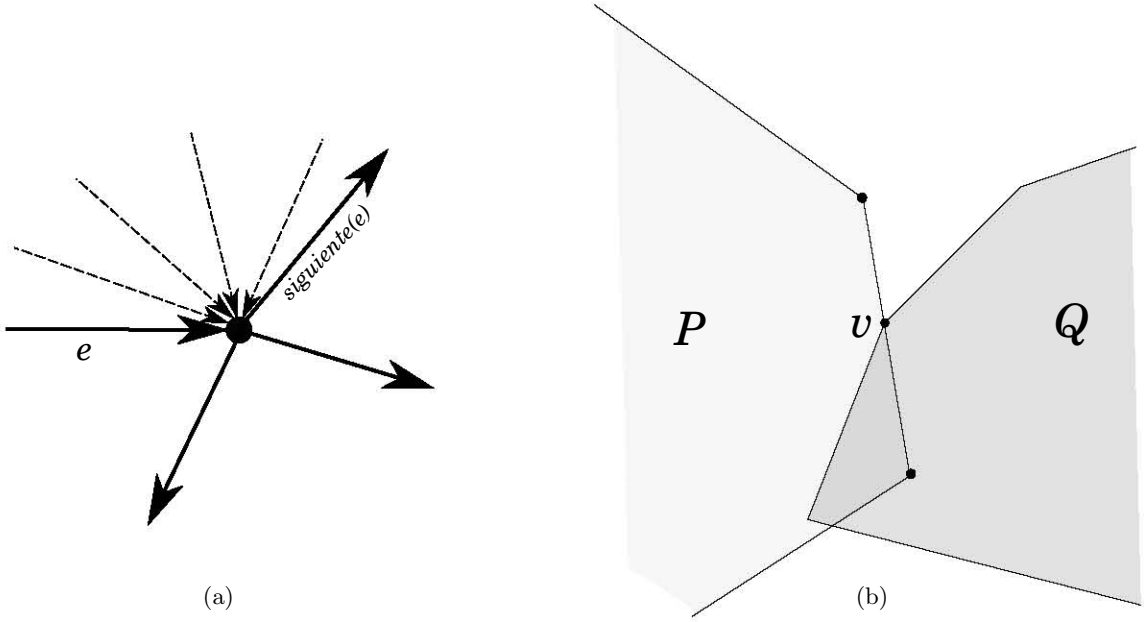
- **Caso 2.** v es vértice de P o de Q (ver figura 2.16(b)).

  Para este caso se toma a v como un vértice P que se intersecta con una arista de Q. Para el caso en que v sea un vértice de Q intersectandose sobre una arista de P, la actualización se realiza de forma análoga.



(a) (b)

Figura 2.15: Actualización del conjunto I para el caso 1.



(a) (b)

Figura 2.16: (a)La arista siguiente de e es encontrada simulando una rotación de e. (b) El punto v es un vértice de Q pero no de P.

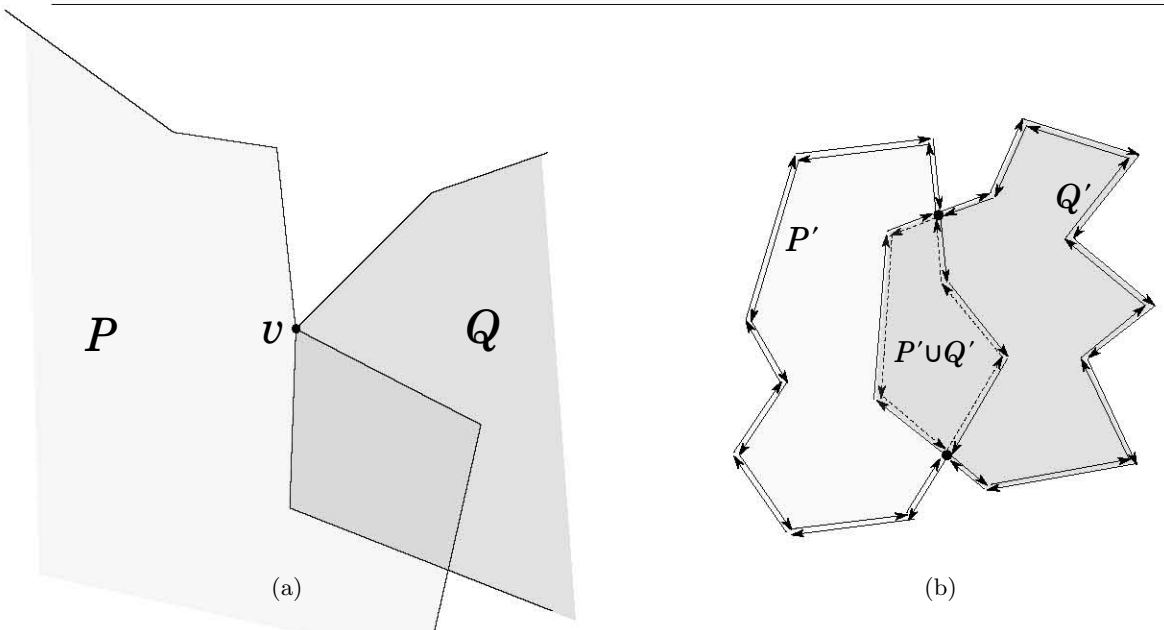


Figura 2.17: (a) El punto v es un vértice de P y también de Q. (b) El polígono generado por la intersección de P y Q puede encontrarse siguiendo el ciclo de las flechas punteadas.

Crear el vértice v en I. Sea e_Q la arista de Q' tal que $v \in e_Q$. Cambiar el origen de e_Q e $inv(e_Q)$ por v. Crear las aristas inversas de e_Q e $inv(e_Q)$ de tal manera que el origen de $inv(e_Q)$ e $inv(inv(e_Q))$ sea el destino de e_Q e $inv(e_Q)$ respectivamente; y el vértice destino de $inv(e_Q)$ e $inv(inv(e_Q))$ sea el origen de e_Q e $inv(e_Q)$. Cambiar la artista siguiente y la cara de todas las aristas tales que su vértice destino sea v, de la misma manera que en el caso 1. Terminando la actualización de I para este caso.

- **Caso 3.** v es vértice de P y de Q (ver figura 2.17(a)). En este caso v es un vértice se P y también de Q. Crear el vértice v en I. Para actualizar I, basta con cambiar la artista siguiente y la cara de todas las aristas tales que su vértice destino sea v usando los métodos vistos en el caso 1. Terminando la actualización de I para este caso.

Habiendo encontrado todas las intersecciónes entre las aristas de P y Q, y actualizado I para cada uno de los casos, resta obtener los polígonos formados por la intersección de P y Q.

Observe que, si se toma una arista al azar de I y se sigue un recorrido tomando su arista siguiente sucesivamente, se formará un ciclo que es visto como un polígono. Para saber si este polígono es un polígono de intersección entre P y Q se analiza la cara de cada arista del ciclo, si existen aristas con cara P y tambien aristas con cara Q dentro de este ciclo entonces se trata de un polígono de intersección (ver figura 2.17(b)).

El algoritmo 2.5 presenta los pasos de manera formal para obtener los polígonos de intersección de dos polígonos:

Si la unión de los polígonos P y Q es requerida, ésta se calcula de manera similar. Basta con tomar el ciclo de todas la aristas tales que tengan como cara el exterior. Observe que

Algorithm 2.5 Intersección de polígonos

Entrada: Dos polígonos P y Q.
Salida: Los polígonos generados por la intersección entre los polígonos P y Q.

1: Obtener los polígonos P' y Q'.
2: Copiar los conjuntos P' y Q' en uno nuevo llamado I.
3: Usando el algoritmo de Bentley-Ottmann, obtener los puntos de intersección v entre las aristas de los polígonos P y Q.
4: Aplicar alguno de los procedimientos para actualizar I dependiendo del caso al que pertenece v.
5: Tomar los ciclos en I tales que las aristas dirigidas en estos ciclos tengan como cara ambos polígonos. Guardar estos polígonos en un conjunto *Intersección*.
6: **return** Intersección.

puede existir más de un ciclo con esta característica, por lo que el polígono que representa la unión entre P y Q será una región o polígono con agujeros.

Capítulo 3

Algoritmo polinomial para calcular trayectorias mínimas en el número de aristas

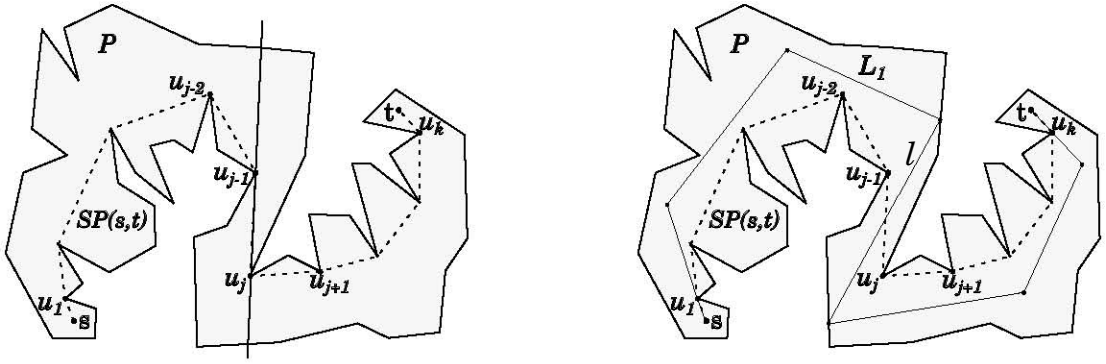
En este capítulo mostramos un caso particular del problema principal de este trabajo. Supongamos que se tiene un polígono simple P con n vértices y dos puntos s y t dentro de P, el objetivo es encontrar una trayectoria con segmentos de línea que conecte a estos dos puntos y que esté totalmente contenido dentro de P, además se desea que esta trayectoria tenga el menor número de segmentos. Observe que es un caso particular ya que en nuestro problema en lugar de tener dos puntos dentro de P se generaliza a k puntos. A continuación se muestra un algoritmo que da solución al caso particular en tiempo $O(n)$.

3.1. Trayectoria mínima en aristas mediante visibilidad completa

En esta sección se describe el algoritmo dado por Gosh en [6] para calcular una *trayectoria mínima en aristas* (llamada MLP) entre dos puntos s y t dentro de un polígono simple P con n vértices en tiempo $O(n)$.

La idea principal del algoritmo es dividir P en subpolígonos, luego calcular una MLP en cada uno y posteriormente unir las MLP vecinas por medio de aristas especiales llamadas *aristas salientes*.

Sea $SP(s,t)$ el *camino más corto* (s, u_1, u_2, ..., u_k, t) del punto s al punto t dentro de P donde u_1, u_2, ..., u_k son vértices de P. Se sabe que $SP(s,t)$ puede ser calculado en tiempo $O(n)$ por el algoritmo de Lee y Preparata en [11] una vez que el polígono P ha sido triangulado en tiempo $O(n)$ por el algoritmo de Chazelle [1]. Apartir de aquí se asume que el polígono P está triangulado. Para más fácil denotamos como $MLP(s,t)$ (o MLP de forma general) a la trayectoria mínima en aristas del punto s al punto t. Cuando decimos *trayectoria* haremos referencia a alguna trayectoria de aristas que pudiera no ser la mínima.



(a) (b)

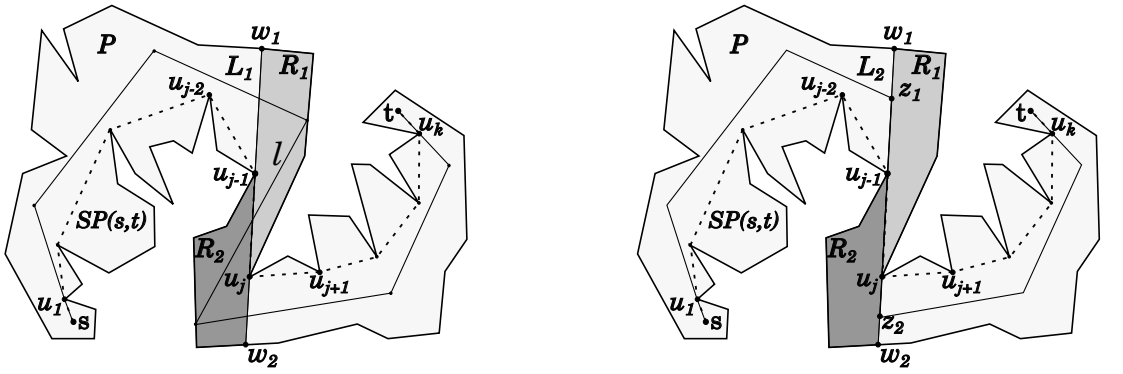
Figura 3.1: (a) La arista $u_j u_{j-1}$ es una saliente de $SP(s,t)$. (b) Una trayectoria mínima en aristas que no contiene ninguna arista saliente de $SP(s,t)$.

Una diferencia entre el SP en distancia euclidiana y la MLP entre dos puntos está en que el primero es el camino entre dos puntos tal que la suma de la longitud de sus aristas es la menor en distancia euclidiana; mientras que el MLP busca ser un camino con el menor número de aristas no importando la longitud de éstas. Otra diferencia es que los vértices del SP son solo vértices de P, mientras que los vértices de la MLP pueden encontrarse siempre en el interior de P. Es importante saber que la MLP no es única debido a que los vértices de ésta pueden colocarse en distintas partes de P. Un ejemplo práctico para describir la utilidad de la MLP es el siguiente: supongamos que se tiene un robot el cual quiere desplazarse del punto s al punto t dentro de un cuarto con la forma del polígono P (visto desde arriba) en el menor tiempo posible, a primera vista podriamos pensar que si el robot toma el camino con menor distancia llegará más rápido al punto t que si tomara a la MLP, pero que pasaría si cada vez que el robot llega a un vértice del camino se detuviera y girara con un proceso lento, la respuesta es que tardaría bastante tiempo en llegar a su destino, en cambio, si tomará a la MLP nuestro robot no tendría que detenerse ni girar demasiadas veces, aprovechando este tiempo en acelerar para llegar a cada vértice de manera más rápida.

Una arista $u_j u_{j-1} \in SP(s,t)$ es llamada *arista saliente* si los puntos u_{j-2}, $u_{j+1} \in SP(s,t)$ se encuentran en lados opuestos de la línea que pasa a través de los puntos u_j y u_{j-1} (ver figura 3.1(a)).

Lema 3.1.1. Existe una $MLP(s,t)$ que contiene todas las aristas *salientes* del $SP(s,t)$.

Demostración. Considere a L_1 como una $MLP(s,t)$ tal que no contiene a la arista saliente $u_{j-1}u_j$ de $SP(s,t)$ (ver figura 3.1(b)). Sea w_1 y w_2 los puntos de intersección entre $bd(P)$ (frontera de P) con $\overrightarrow{u_j u_{j-1}}$ y $\overrightarrow{u_{j-1} u_j}$ respectivamente. Es fácil ver que, el segmento $w_1 w_2$ particiona al polígono P en cuatro regiones disjuntas de las cuales dos, R_1 y R_2, no contienen a ninguno de los puntos s o t. Puesto que P es una región cerrada y acotada, existe una arista l en L_1 tal que un punto extremo se encuentra en R_1 y otro se encuentra



(a) Una MLP que no contiene ninguna arista saliente de $SP(s,t)$.

(b) Una MLP llamada L_2 que contiene a la arista saliente de $SP(s,t)$.

Figura 3.2: Trayectorias mínimas en aristas conectando s a t.

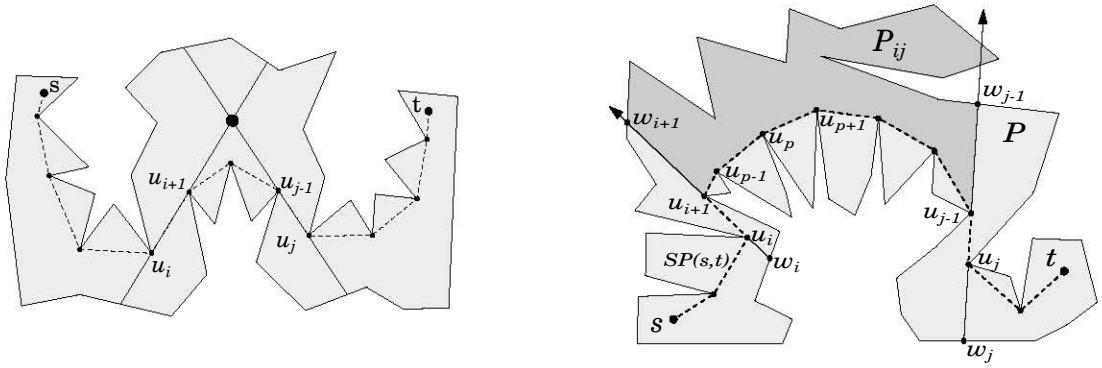
en R_2, ya que cualquier trayectoria desde s debe intersectar a la arista saliente $u_{j-1}u_j$ para poder alcanzar al punto t (ver figura 3.2(a)).

Sean z_1 y z_2 los puntos de intersección de L_1 con $u_{j-1}w_1$ y u_jw_2, respectivamente. Una nueva trayectoria L_2 puede ser construida usando a L_1, removiendo la porción de L_1 que está entre los puntos z_1 y z_2, y agregando el segmento z_1z_2. Esta modificación ha eliminado totalmente a la arista l y, por lo tanto, L_2 es una MLP entre s y t conteniendo la arista saliente $u_{j-1}u_j$ (ver figura 3.2(b)). Si existe otra saliente $u_{i-1}u_i$ que no este contenida en L_2, eliminar la porción de L_2 como antes y construir otra MLP L_3 entre s y t conteniendo ambas aristas salientes $u_{j-1}u_j$ y $u_{i-1}u_i$. Repitiendo este proceso para cada arista saliente de $SP(s,t)$, una MLP entre s y t puede ser construida conteniendo todas las aristas salientes del $SP(s,t)$. □

El lema anterior propone el siguiente procedimiento para construir una $MLP(s,t)$.

- Separar P en subpolígonos extendiendo cada arista saliente desde sus puntos extremos hasta $bd(P)$.
- Conectar las extensiones de cada par de aristas salientes consecutivas en el $SP(s,t)$ con MLP's, para formar una $MLP(s,t)$. Se puede dar un caso particular en el que dos extensiones de aristas salientes consecutivas se intersectan en un punto z, por lo que z puede ser tomado como un *punto de giro* de la $MLP(s,t)$ (ver figura 3.3(a)).

Ahora veremos a detalle como divimos el polígono y calculamos las MLP en cada subpolígono. Sean w_{i+1} y w_{j-1} los puntos de intersección más cercanos entre las extensiones $\overrightarrow{u_iu_{i+1}}$ y $\overrightarrow{u_ju_{j-1}}$ con $bd(P)$ respectivamente, a lo que llamaremos *puntos de extensión* (ver figura 3.3(b)). Ya que w_{i+1} y w_{j-1} pertenecen a la frontera de P etiquetada de forma levógira desde u_j hasta u_i (i. e., $bd(u_j,u_i)$), estos pueden ser calculados comprobando la intersección de $\overrightarrow{u_iu_{i+1}}$ y $\overrightarrow{u_ju_{j-1}}$ con las aristas de $db(u_j,u_i)$. Esto significa que los puntos de extensión de todas las aristas salientes de $SP(s,t)$ pueden ser calculados en tiempo $O(n)$. P_{ij} denota el subpolígono acotado por $bd(w_{j-1},w_{i+1})$, el segmento $w_{i+1}u_{i+1}$, el



(a) (b)

Figura 3.3: (a)Las extensiones de las aristas salientes u_iu_{i+1} y $u_{j-1}u_j$ se intersectan formando un *punto de giro*. (b) Subpolígono P_{ij} formado por aristas saliente no intersectadas.

$SP(u_{i+1}, u_{j-1})$ y el segmento $u_{j-1}w_{j-1}$. Notese que, dado que $SP(u_{i+1}, u_{j-1})$ no contiene ninguna saliente, ningún vértice del $bd(w_{j-1}, w_{i+1})$ pertenece al $SP(u_{i+1}, u_{j-1})$ y, por lo tanto, P_{ij} es un subpolígono simple.

Sea L_{ij} una MLP desde un punto en $u_{i+1}w_{i+1}$ a algún punto de $u_{j-1}w_{j-1}$. Una trayectoria es *convexa* si todos sus puntos de giro dan vuelta hacia el mismo lado.

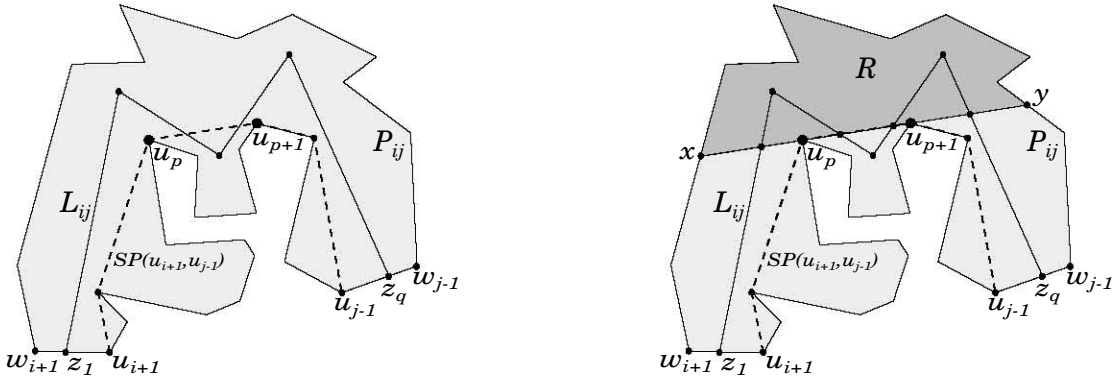
Lema 3.1.2. Toda MLP L_{ij} que está dentro de P_{ij} es *convexa*.

Demostración. Sin perdida de generalidad, se asume que $SP(u_{i+1}, u_{j-1})$ hace un giro a la derecha en cada vértice de la trayectoria. Sea $L_{ij} = (z_1z_2, \ldots, z_{q-1}z_q)$, donde $z_1 \in u_{i+1}w_{i+1}$ y $z_q \in u_{j-1}w_{j-1}$ (ver figura 3.4(a)). Para probar el lema basta con mostrar que L_{ij} hace un giro a la derecha en cada punto de giro $z_2, z_3, \ldots, z_{q-1}$ mientras atraviesa P_{ij} desde z_1 hasta z_q.

Sea u_pu_{p+1} una arista de $SP(u_{i+1}, u_{j-1})$ y sean x y y los puntos de intersección más cercanos de $bd(w_{j-1}, w_{i+1})$ con $\overrightarrow{u_{p+1}u_p}$ y $\overrightarrow{u_pu_{p+1}}$ respectivamente. Sea también R la región de P_{ij} acotada por el segmento xy y la frontera de P_{ij} $bd(y, x)$ (ver figura 3.4(b)). Si L_{ij} intersecta a xy en tres o más puntos o R contiene tres o más puntos de giro de L_{ij}, entonces el número de aristas en L_{ij} puede ser reducido usando el segmento xy, lo cual contradice la minimalidad de L_{ij}. Ahora, asumase que L_{ij} intersecta con xy en dos puntos y R contiene a lo más dos puntos de giro (z_r y z_{r+1}) de L_{ij} (ver figura 3.5(a)).

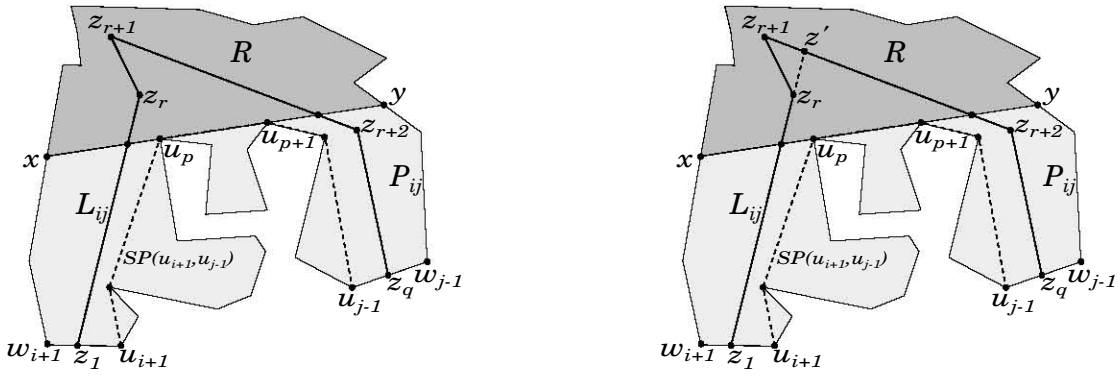
Si L_{ij} hace un giro a la izquierda en z_r, entonces se extiende la arista $z_{r-1}z_r$ desde z_r atravezando la arista $z_{r+1}z_{r+2}$ en un punto z' (ver figura 3.5(b)). Esto significa que existe otro camino de aristas $z_1z_2, \ldots, z_{r-1}z'$,$z'z_{r+2}, \ldots, z_{q-1}z_q$, que tiene una arista menos que L_{ij}, por lo que estariamos hablando de una contradicción, por lo tanto, L_{ij} hace un giro a la derecha en z_r. Argumentos análogos muestran que L_{ij} hace un giro a la derecha en cada uno de sus puntos de giro.

Los argumentos anteriores también muestran que L_{ij} no intersecta ninguna arista de $SP(u_{i+1}, u_{j-1})$ y, por lo tanto, L_{ij} se encuentra totalmente contenida en P_{ij}.



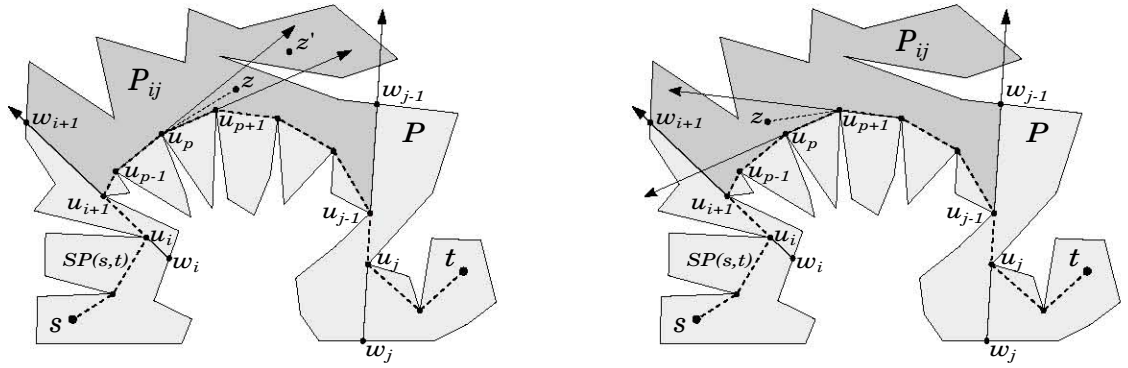
(a) (b)

Figura 3.4: (a)$SP(u_{i+1}, u_{j-1})$ y una MLP L_{ij} de z_1 a z_q. (b)L_{ij} intersecta a xy en más de tres puntos.



(a) (b)

Figura 3.5: L_{ij} intersecta a xy en dos puntos y R contiene a lo más dos puntos de giro de L_{ij}



(a) El segmento zu_p es la tangente izquierda de z en el vértice u_p

(b) El segmento zu_{p+1} es la tangente derecha de z en el vértice u_{p+1}

Figura 3.6: Tangente izquierda y derecha de z

□

Con el fin de calcular una MLP convexa L_{ij}, es necesario conocer las definiciones de tangente izquierda y derecha de un punto $z \in P_{ij}$ a $SP(u_{i+1}, u_{j-1})$. El segmento zu_p es llamado *tangente izquierda* (o *tangente derecha*) de z en el vértice $u_p \in SP(u_{i+1}, u_{j-1})$ (ver figura 3.6(a)) si zu_p está totalmente contenido en P_{ij} y z está a la derecha de $\overrightarrow{u_{p-1}u_p}$ ($\overrightarrow{u_pu_{p-1}}$ respectivamente) y a la izquierda de $\overrightarrow{u_pu_{p+1}}$ ($\overrightarrow{u_{p+1}u_p}$ respectivamente). Nótese que para todos los puntos $z' \in P_{ij}$ que están a la derecha de $\overrightarrow{u_{p-1}u_p}$ y a la izquierda de $\overrightarrow{u_pu_{p+1}}$, $z'u_p$ podría no ser la tangente izquierda de z' si el segmento $z'u_p$ no está totalmente contenido en P_{ij}. Así que, algunos puntos de P_{ij} pueden no tener tangente izquierda o derecha a $SP(u_{i+1}, u_{j-1})$.

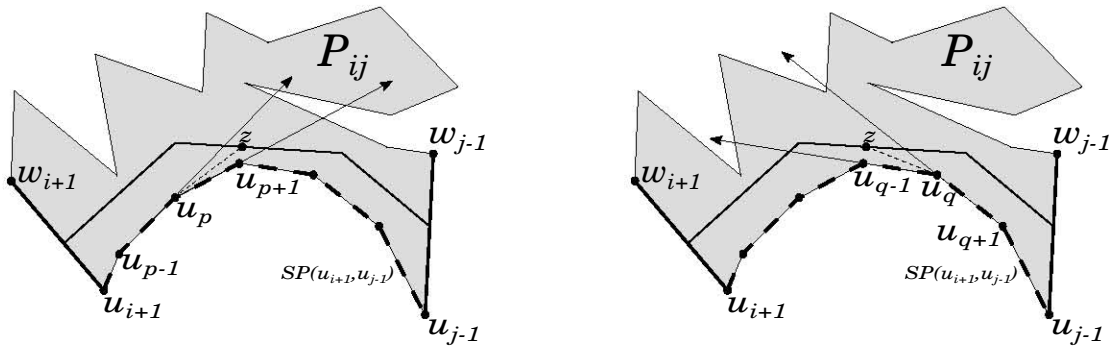
Ya con las definiciones de tangente izquierda y derecha presentamos el siguiente lema.

Lema 3.1.3. Si un punto $z \in P_{ij}$ se encuentra dentro de una trayectoria convexa entre $u_{i+1}w_{i+1}$ y $u_{j-1}w_{j-1}$ dentro de P_{ij}, entonces z tiene ambas tangentes, izquierda y derecha.

Demostración. La prueba del lema es solamente para la tangente izquierda de z, ya que la prueba para la tangente derecha es análoga. Sea u_p el vértice de $SP(u_{i+1}, u_{j-1})$ tal que z se encuentra a la derecha de $\overrightarrow{u_{p-1}u_p}$ y a la izquierda de $\overrightarrow{u_pu_{p+1}}$. Si el segmento zu_p se encuentra dentro de P_{ij}, entonces zu_p es la tangente izquierda de z. Si zu_p no se encuentra dentro de P_{ij}, esto significa que $bd(w_{j-1}, w_{i+1})$ se intersecta con zu_p. Ya que z pertenece a la trayectoria convexa entre $u_{i+1}w_{i+1}$ y $u_{j-1}w_{j-1}$ dentro de P_{ij}, por presunción, $bd(w_{j-1}, w_{i+1})$ ha atravezado zu_p intersectando también a la trayectoria convexa, lo cual es una contradicción. Por lo tanto, zu_p es la tangente izquierda de z.

□

Sea R_{ij} el conjunto de todos los puntos en P_{ij} tal que cada punto de R_{ij} tiene tangente izquierda y tangente derecha a $SP(u_i, u_j)$. El procedimiento para calcular R_{ij} se explica más adelante. Por ahora se asume que R_{ij} ha sido calculado. L_{ij} puede ser construido



(a) (b)

Figura 3.7: Como el punto z se encuentra sobre la trayectoria convexa, z tiene tangente izquierda (a) y tangente derecha (b)

en R_{ij} como sigue (ver figura 3.8). Sea $z_1 \in u_{i+1}w_{i+1}$ el primer punto de giro de L_{ij}. Si w_{i+1} pertenece a R_{ij} entonces $z_1 = w_{i+1}$. En otro caso, z_1 es el punto más alejado de u_{i+1} en el segmento $u_{i+1}w_{i+1}$ que pertenece a R_{ij} (i. e., el siguiente vértice después de u_{i+1} en sentido dextrógiro en R_{ij}). Trazar la tangente derecha de z_1 a $SP(u_{i+1}, u_{j-1})$ y extender la tangente hasta que se encuentre con la frontera de R_{ij} en algún punto z_2. De nuevo, trazar la tangente derecha desde z_2 hasta $SP(u_{i+1}, u_{j-1})$ y extender la tangente hasta que se encuentre con la frontera de R_{ij} en algún punto z_3. Repetir este procedimiento hasta que un punto z_q sea encontrado sobre $u_{j-1}w_{j-1}$. Así, la trayectoria $z_1z_2, z_2z_3, \ldots, z_{q-1}z_q$ es construida entre $u_{i+1}w_{i+1}$ y $u_{j-1}w_{j-1}$. Por lo tanto proponemos el siguiente lema.

Lema 3.1.4. La trayectoria L_{ij}=$z_1z_2, z_2z_3, \ldots, z_{q-1}z_q$ es una trayectoria mínima en aristas dentro de P_{ij}, donde $z_1 \in u_{i+1}w_{i+1}$ y $z_p \in u_{j-1}w_{j-1}$.

Demostración. Ya que ninguna arista de cualquier trayectoria que conecte $u_{i+1}w_{i+1}$ con $u_{j-1}w_{j-1}$ puede intersectar más de una tangente izquierda de los puntos de giro de L_{ij} al $SP(u_{i+1}, u_{j-1})$, la trayectoria L_{ij} es una trayectoria mínima en aristas. □

Considerando la primer arista y la última del $SP(s,t)$ como aristas salientes en $SP(s,t)$, los caminos greedy entre las extensiones de cada par de aristas salientes consecutivas en $SP(s,t)$ son calculadas y luego son conectadas como se mencionó anteriormente para formar una trayectoria mínima en aristas entre s y t.

Ahora se menciona el procedimiento para calcular R_{ij}. Para calcularlo, el algoritmo de Gosh en [6], particiona P_{ij} en subpolígonos extendiendo algunas de las aristas de $SP(u_{i+1}, u_{j-1})$ hasta la frontera de P_{ij} tal que:

- Dos extensiones no pasen a través del mismo triángulo en la triangulación de P, y
- La porción $SP(u_p, u_q)$ del $SP(u_{i+1}, u_{j-1})$, en cada subpolígono, no de un giro mayor a 2π (ver figura 3.9(a)).

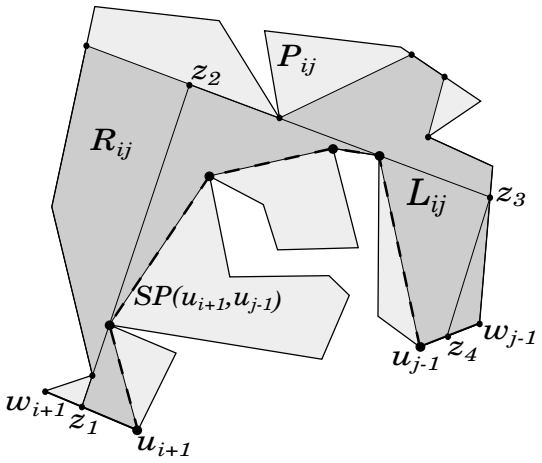


Figura 3.8: Construyendo una trayectoria mínima en aristas L_{ij} de $w_{i+1}u_{i+1}$ a $u_{j-1}w_{j-1}$ dentro de R_{ij}

Tratar la región delimitada por $SP(u_p, u_q)$ y el segmento u_pu_q como un conjunto convexo C_p. Calcular $Vc(C_p)$ dentro del subpolígono de P_{ij} acotado por u_qw_q, $bd(w_q, w_p)$, w_pu_p y $SP(u_p, u_q)$ (ver figura 3.9(b)). Observese que la unión de los subpolígonos de visibilidad completa de los conjuntos convexos C_p es R_{ij} y el tiempo tomado para calcular R_{ij} en P_{ij} es proporcional al tamaño de P_{ij}.

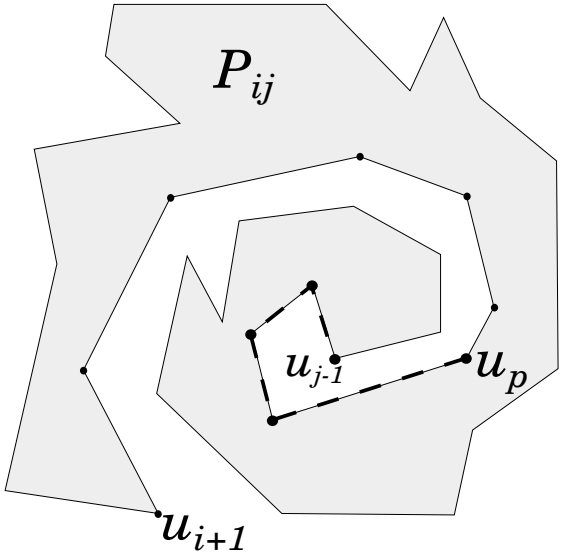
Ahora, se presentan los pasos principales del algoritmo para calcular una trayectoria mínima en aristas entre dos puntos s y t dentro de un polígono simple P.

Algorithm 3.1 *Minimum-Link-Path*

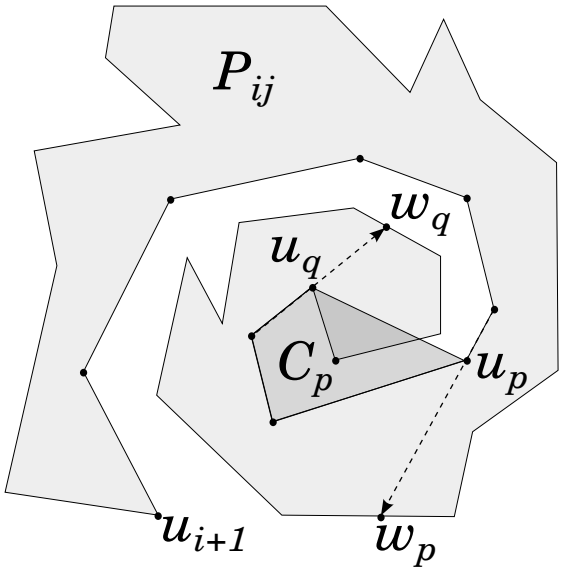
Entrada: Un polígono P y dos puntos s y t dentro de P.
Salida: Una secuencia de aristas que forman una trayectoria mínima en aristas de s a t.

1: Calcular $SP(s,t)$ usando el algoritmo de Lee y Preparata.
2: Dividir P en subpolígonos extendiendo cada arista saliente de $SP(s,t)$ en ambos extremos hasta $bd(P)$. También extender las aristas primera y última de $SP(s,t)$ a $bd(P)$.
3: En cada subpolígono de P, construir la trayectoria entre las extensiones de las aristas salientes.
4: Conectar las trayectorias usando las aristas salientes y sus extensiones para formar una trayectoria mínima en aristas entre s y t.

Cada uno de los pasos se puede calcular en tiempo $O(n)$. Por lo tanto, la complejidad total en tiempo del algoritmo es $O(n)$.



(a) $SP(u_p, u_{j-1})$ da un giro mayor a 2π



(b) $SP(u_p, u_q)$ y el segmento $u_p u_q$ acotan la región convexa C_p

Figura 3.9

Capítulo 4

El problema de acoplar un árbol de Steiner al interior de un polígono simple

En este capítulo definimos el problema principal de este trabajo y damos un algoritmo que lo resuelve. También mostramos que la complejidad de este algoritmo es $O(k!)$ en el peor caso.

4.1. Definición del problema

Se tiene un polígono simple P con n vértices, un árbol de Steiner T con k terminales y un conjunto M de k puntos dentro de P. El problema consiste en verificar si es posible dibujar a T usando únicamente segmentos rectilíneos dentro de P, colocando cada uno de los nodos terminales de T en un punto de M, sin que ninguna de las aristas de T se intersecte con alguna de las aristas de P; es decir, que T esté totalmente contenido dentro de P, ver figura 4.1.

Visto de otra forma, la tarea está en asignar coordenadas a los nodos de T de tal forma que las aristas de T, puestas como segmentos rectilíneos, estén totalmente contenidas en P.

La complejidad de solucionar este problema radica en que cada uno de los vértices terminales de T debe ser asignado a un punto del conjunto $M \subset P$, además esta asignación debe ser biyectiva.

4.2. Esbozo del algoritmo

El algoritmo que se presenta para solucionar el problema de acoplar un árbol de Steiner al interior de un polígono simple, establece en primer lugar una asignación de cada terminal $t_i \in V_t(T)$ a cada punto de M y después verifica si esta asignación es factible para poder dibujar o acoplar T. En caso de que T no pueda ser acoplado dentro de P con esta asignación el algoritmo se repite con una nueva asignación.

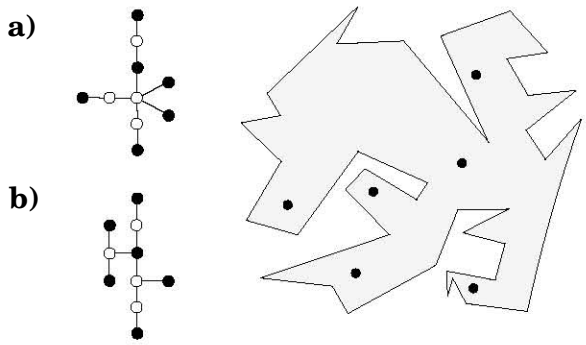


Figura 4.1: Dos árboles de Steiner y un polígono con puntos en su interior. Observe que solo el árbol del inciso b) puede ser acoplado.

La asignación es una biyección $f_l : V_t(T) \to M$ que asocia a cada $t_i \in V_t(T)$ un punto M, donde $l \in \{1, \ldots, k!\}$. Para verificar si f_l es una asginación factible para acoplar T en P, debemos verificar que cada arista en T pueda ser dibujada como un segmento rectilíneo dentro de P, respentando las adyacencias en T. Existen tres casos diferentes para verificar que una arista pueda ser dibujada en P. El caso más sencillo es cuando una arista de T esta conformada por dos vértices terminales. Si t_i y t_j son los vértices terminales de la arista, entonces simplemente verificamos que $f_l(t_i) \in V(f_l(t_j))$. El segundo caso es cuando la arista está conformada por un vértice terminal t_i y un vértice Steiner s. En este caso debemos tomar todos los vértices terminales t_i, $\ldots$, t_j adyacentes a s y verificar si $\bigcap_{h\in\{i\ldots j\}} V(f_l(t_h)) \neq \varnothing$. Si esta intersección es vacía entonces al menos una de las aristas st_h no puede ser dibujada en P, pero si la intersección existe entonces esta intersección es la región R en la que podemos colocar a s y así dibujar las aristas dentro de P. El último caso, cuando ambos vértices de la arista en T son vértices Steiner, no es sencillo y se explicará más adelante cuando se desarrolle más la intuición sobre el funcionamiento del algoritmo.

El algoritmo sigue un proceso recursivo tomando un vértice $r \in T$ como raíz, y recorre a T con un recorrido primero en profundidad a partir de r hasta las hojas. Comienza por verificar que las aristas incidentes a las hojas pueden ser dibujadas como segmentos rectilíneos dentro de P. Si esto es posible entonces el algoritmo continua con las aristas que están a un nivel superior, de tal forma que el último nivel que verifique sea el de las aristas que son incidentes a r. Ahora veremos con más detalle como trabaja el algoritmo.

4.3. El algoritmo exponencial y su complejidad

En esta parte analizaremos el algoritmo con más detalle verificando también que trabaja correctamente. El algoritmo trabaja recursivamente tomando subárboles de T. Cada subárbol estará conformado por un vértice raíz (o padre) x y vértices hijos que son los vértices adyacentes inmediatos a x, de esta forma el algoritmo solo verificará las aristas

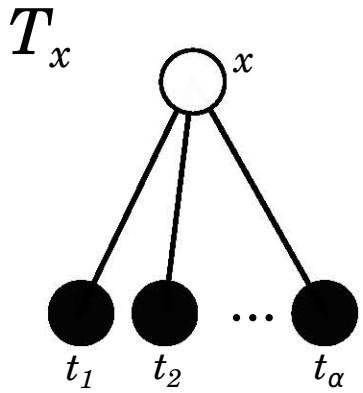


Figura 4.2: Subárbol de un solo nivel.

del subárbol. Al verificar que todas las aristas pueden ser dibujadas como segmentos rectilíneos dentro de P el algoritmo regresará un valor verdadero y se dice que este subárbol puede ser acoplado dentro de P. En caso de que alguna arista no pueda ser dibujada el algoritmo regresa un valor falso y termina, para comenzar de nuevo con una asginación diferente. Definimos varios tipos de subárboles así como la forma en que el algoritmo verifica si puede ser acoplado.

- **Subárboles base** Cuando el subárbol está conformado por un solo vértice.

  Para este caso el algoritmo regresa verdadero ya que un árbol con un solo vértice obviamente puede ser acoplado dentro de P.

Sea T_x un tipo de subárbol de T con raíz en el vértice x e hijos terminales t_1, t_2, ..., t_α todos hojas de T (ver figura 4.2). Decimos que T_x es un árbol de un solo nivel.

Cuando un subárbol del tipo T_x entra en el algoritmo, éste hace una llamada así mismo por cada subárbol de T_x. Como todos los subárboles de T_x son casos base el algoritmo regresa un valor verdadero por cada uno. Después de verificar todos todos los subárboles de T_x, el algoritmo verifica que cada una de las aristas puede ser dibujada como segmento rectilíneo dentro de P. La verificación para las aristas de T_x toma en cuenta dos posibles casos: cuando x es un terminal y cuando es un Steiner.

Subárboles de un solo nivel

- **caso 1:** Cuando la raíz es un terminal con todos sus hijos terminales.

  Para comenzar la verificación el algoritmo calcula $V(f_l(x))$, luego comprueba que los puntos $f_l(t_j) \in V(f_l(x))$ para $j = \{1, \ldots, \alpha_i\}$. Si todos los puntos se encuentran dentro de $V(f_l(x))$ el procedimiento regresa un valor verdadero debido a que las aristas de T_x pueden ser dibujadas como segmentos rectilíneos dentro de P, lo que quiere decir que T_x puede ser acoplado dentro de P. Si algún punto $f_l(t_j)$ se encuentra fuera de $V(f_l(x))$ el procedimiento regresa un valor falso ya que $f_l(x)$ no puede ver a $f_l(t_j)$ y por lo tanto la arista xt_j de T_x se intersecta con las aristas de P.

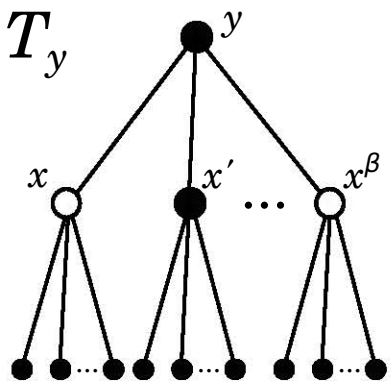


Figura 4.3

- **caso 2:** Cuando la raíz es un Steiner con todos sus hijos terminales. Debido a que la raíz es un Steiner, podría ser posible colocarlo en cualquier parte de P ya que los vértices Steiner de T no están ligados con algún punto de P como en el caso de los vértices terminales. Se denota como $R(x)$ a la región dentro de P donde es posible colocar al Steiner x. Para encontrar $R(x)$ calculamos $\bigcap_{i=1}^{\alpha} V(f_l(t_i)) \neq \varnothing$. Si esta región existe el algoritmo regresa un valor verdadero, ya que la visibilidad que hay entre cualquier punto de $R(x)$ con $f_l(t_1)$, $f_l(t_2)$, ..., $f_l(t_\alpha)$ nos permite colocar a x dentro de $R(x)$ sin que las aristas de T_x salgan de P. En caso que la región no exista o algún polígono $V(f_l(t_j))$ para $j = \{1, \ldots, \alpha_i\}$ no se intersecte con $R(x)$ el algoritmo regresa un valor falso.

Hasta aquí se ha analizado como el procedimiento verifica el acoplamiento para subárboles de T que están formados por un solo vértice (subárboles base) y también para los que están formados por un vértice padre (Steiner o terminal) e hijos terminales todos hojas de T (subárboles de un nivel). Ahora se analizará la verificación para el acoplamiento de subárboles de T que están conformados por un nodo padre (Steiner o terminal) con hijos que también son padres de subárboles de un solo nivel.

Sea T_y un subárbol de T con raíz en el vértice y, tal que los hijos de y son raíces de los subárboles T_x, $T_{x'}$, ..., T_{x^β} (ver figura 4.3), se dice que T_y es un subárbol de dos niveles.

Cuando T_y entra en el procedimiento, éste hace una llamada así mismo por cada subárbol de T_y. Suponiendo que el procedimiento regresa verdadero para cada subárbol T_x, $T_{x'}$, ..., T_{x^β} (utilizando la verificación para los subárboles de un solo nivel) esto significa que es posible acoplar todos dentro de P, entonces queda saber si el nodo padre y puede ser acoplado también sin que las aristas que conectan a éste con los subárboles se intersectan con las aristas de P, es decir que estas aristas puedan ser dibujadas como segmentos rectilíneos dentro de P. En caso de que el procedimiento regrese un valor falso para alguno de los subárboles el algoritmo termina y vuelve a generar una nueva asignación f_l.

La verificación para T_y se hace de la misma forma que en el árbol de un solo nivel con la diferencia de que ahora y puede tener hijos de tipo Steiner además de los terminales,

por lo que para hacer la verificación de T_y se toman en cuenta los siguientes dos tipos de subárboles:

Subárboles de dos niveles

- **caso 3:** Cuando la raíz es un terminal con hijos Steiner y también terminales.

  El algoritmo verifica primero el acoplamiento para los subárboles que tienen padres terminales. Si esta verificación se cumple toca verificar los subárboles con padres Steiner. Sea s_j un vértice hijo de y de tipo Steiner. Se sabe que $R(s_j)$ es la región donde es posible colocar a s_j para que el subárbol T_{s_j} pueda ser acoplado en P (utilizando la verificación para subárboles de un solo nivel con raíz Steiner), por lo que si $f_l(y)$ puede ver a la región $R(s_j)$ o parte de ella en P, para cada hijo s_j de y, entonces y puede ser acoplado dentro de P y el procedimiento regresa un valor verdadero. En caso de que y no pueda ver $R(s_j)$ para algún s_j, o algún subárbol de y con padre terminal no pueda ser acoplado, el procedimiento regresa un valor falso.

- **caso 4:** Cuando la raíz es un Steiner con hijos Steiner y también terminales.

  Primero, el procedimiento hace una verificación para todos los hijos terminales utilizando la verifición del subárbol de dos niveles con raíz Steiner, suponiendo que el procedimiento regresa verdadero para todos, se obtiene una región $Rt(y)$ donde es posible colocar al Steiner y de tal forma que sea visible por todos sus hijos terminales. Luego, el procedimiento continua con una verificación para sus hijos Steiner. Sean s_1, s_2, ..., s_π los hijos Steiner de y, observe que las regiones $R(s_1)$, $R(s_2)$, ..., $R(s_\pi)$ ya han sido obtenidas gracias a la verificación previa de cada subárbol. Ahora, sea $Rs(y)$ la región en P tal que es visible por al menos un punto de cada una de las regiones $R(s_1)$, $R(s_2)$, ..., $R(s_\pi)$, es decir, $Rs(y)$=$V_d(R(s_1)) \cap V_d(R(s_2)) \cap \ldots \cap V_d(R(s_\pi))$. Si $Rs(y) \neq \varnothing$ entonces queda obtener la región tal que y pueda ser colocado, esta región es $R(y)$ que es calculada con la intersección de $Rt(y)$ y $Rs(y)$, es decir, $R(y)$=$Rt(y) \cap Rs(y)$. Para terminar se regresa verdadero si $R(y) \neq \varnothing$, y falso si no se cumple.

Observe que es posible verificar otro tipo de árbol T'_y con subárboles T_{x^j} tales que esten conformados por un sólo vértice (ver figura 4.4) o por subárboles mucho más grandes de la misma forma que se verifica el acoplamiento de T_y ya que el procedimiento se invoca dando como entrada cualquier tipo de subárbol.

La verificación para todos los posibles tipos de subárboles de T ha sido analizada en los casos anteriores por lo que ahora veremos la verificación para el árbol completo T. El procedimiento verifica T comenzando por r usando la verificación para los subárboles de dos niveles como si fuera uno de ellos. Cuando se hace la llamada al procedimiento por cada subárbol de T, cada uno se verifica como un árbol de dos niveles. Se utiliza este método sucesivamente hasta encontrar subárboles de un solo nivel para los cuales se utiliza la verificación para los subárboles de un solo nivel, luego el caso base para las hojas de T. Por lo anterior, confirmamos que se puede verificar el acoplamiento de T usando los casos anteriores.

Ahora presentamos el pseudocódigo del algoritmo para verificar si un árbol de Steiner T con raíz r es acoplable dentro de P (algoritmo 4.1), con el fin de comprenderlo fácilmente

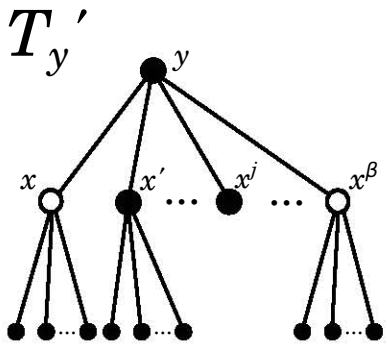


Figura 4.4

describimos el funcionamiento de algunas variables. Para un nodo x de T denotamos a $x.type$ como un atributo el cual guarda una etiqueta que indica en que conjunto de nodos de T se encuentra x ($Steiner$ si x pertenece al conjunto de vértices Steiner $V_s(T)$ y $Terminal$ si x pertenece al conjunto de terminales $V_t(T)$). Si x es un Steiner, usamos $x.label$ para etiquetar a x con un entero. El variable global g funciona como un contador y ayuda a asignar un número a cada nodo Steiner de T. La tarea del arreglo R es contener cada una de las regiones donde es posible colocar los vértices Steiner, ya que cada vértice Steiner será etiquetado con $x.label$, $R[x.label]$ será la región donde x puede ser colocado. Las listas H_s y H_t simplemente se utilizan para guardar y accesar fácilmente a los hijos de x que son Steiner y terminales respectivamente. Suponemos que la biyección f_l ya es dada antes de comenzar el algoritmo. Observe que el algoritmo tendrá que ejecutarse una vez por cada asignamiento f_l, donde l va de 1 hasta $k!$.

El algoritmo 4.1 recibe como entrada un árbol de Steiner con raíz T_r, para calcular T_r a partir de T simplemente tomamos como raíz r al vértice que se encuentra justo a la mitad de la trayectoria más larga entre cualesquiera dos vértices de T.

El algoritmo 4.2 es una extensión del algoritmo 4.1 que obtiene la posición exacta de cada uno de los vértices Steiner dentro del polígono. Éste consiste básicamente en hacer un recorrido primero en anchura sobre T a partir de su nodo raíz r. Antes de comenzar con la recursión es necesario obtener la posición de r en caso de que éste sea un vértice Steiner. Podemos tomar un punto arbitrariamente en $R[r.label]$ como la posición de r. Observe que la región $R[r.label]$ pertenece a $V_d(R[s.label])$ donde s es cualquier hijo Steiner de r, entonces cualquier punto de $R[r.label]$ es visible por algún punto de $R[s.label]$. Luego, dentro del recorrido recursivo cuando un subárbol con raíz terminal e hijos Steiner s_1, $\ldots$, s_β es tomado, se calcula $V(t)$ y se toma un punto s_{p_i} arbitrariamente dentro de la región $V(t) \cap R[s_i.label]$ como la posición de s_i para $i \in \{1, \ldots, \beta\}$. Si el subárbol tomado tiene raíz Steiner s con hijos Steiner s_1, $\ldots$, s_β el algoritmo calcula $V(s_p)$, donde s_p es la posición de s, y computa $R2_{s_i} = V(s_p) \cap R_{s_i}$ para $i \in \{1, \ldots, \beta\}$, luego toma un punto arbitrariamente de $R2_{s_i}$ como la posición de s_i para $i \in \{1, \ldots, \alpha\}$. Este proceso se repite con cada nodo Steiner de T.

Algorithm 4.1 *Steiner-Tree-Embedded(T_r)*

Entrada: Un árbol de Steiner con raíz T_r.

Salida: Verdadero si T_r puede ser acoplado en P con la biyección f_l, falso en caso contrario.

```
1:
2: if δ(r) > 1 then
3:     for each nodo i hijo de r do
4:         if i.type == Steiner then
5:             Insertar i en H_s(r)
6:         end if
7:         if i.type == Terminal then
8:             Insertar i en H_t(r)
9:         end if
10:        if Steiner-Tree-Embedded(T_i) == False then
11:            return False
12:        end if
13:    end for
14:
15:    if r.type == Steiner then
16:        g++
17:        r.label=g
18:        R[r.label]= ⋂_{i∈H_t(r)} V(f_l(i)) ∩ ⋂_{i∈H_s(r)} Vr(R[i.label])
19:        if R[r.label] = ∅ then
20:            return False
21:        end if
22:    end if
23:
24:    if r.type == Terminal then
25:        for each i ∈ H_t(r) do
26:            if i ∉ V(r) then
27:                return False
28:            end if
29:        end for
30:        for each i ∈ H_s(r) do
31:            if V(r) ∩ R[i.label] = ∅ then
32:                return False
33:            end if
34:        end for
35:    end if
36: end if
37:
38: return True
```

Si r es un Steiner tomar un punto arbitrario de $R[r.label]$ como $r.position$

Algorithm 4.2 *Position(T_r)*

Entrada: Un árbol de Steiner T_r con raíz r.
Salida: Un árbol de Steiner T con posiciones en sus nodos Steiner.

```
1:
2: if δ(r) > 1 then
3:     for each nodo i hijo de r do
4:         if i.type == Steiner then
5:             if r.type == terminal then
6:                 R2[i.label] ← V(r) ∩ R[i.label]
7:                 i.position ← punto arbitrario de R2[i.label]
8:             else
9:                 R2[i.label] ← V(r.position) ∩ R[i.label]
10:                i.position ← punto arbitrario de R2[i.label]
11:            end if
12:        end if
13:        Position(T_i)
14:    end for
15: end if
16: return
```

Analizaremos la complejidad del algoritmo 4.1 dividiendolo en tres partes: la primera parte esta conformada por el ciclo for de la línea 3 a la 13; la segunda es la parte del algoritmo para procesar los subárboles con raíz Steiner de la línea 15 a la 22; y la tercera que procesa los subárboles con raíz terminal de la línea 24 a la 36.

La primer parte puede verse como un recorrido primero en profundidad que recorre todos los vértices del árbol de manera recursiva y su complejidad es $O(m)$ donde m es el número total de nodos en el árbol.

Cuando la segunda parte se ejecuta se calcula la intersección de todos los polígonos de visibilidad de los puntos $f_l(t)$, para cada nodo terminal t hijo de r_i con $i \in \{1, \ldots, m\}$ y se calculan los polígonos de visibilidad débil de las regiones $R[s.label]$, para cada nodo Steiner s hijo de r_i. Como hicimos ver en el capitulo 2, calcular $V(f_l(t))$ toma tiempo $O(n)$, y el polígono de visibilidad débil de una región toma tiempo $O(n)$. Por lo que la complejidad de hacer esto es $O((\alpha_i + \beta_i)n)$ donde α_i y β_i son el número de hijos de r_i terminales y Steiner, respectivamente. La intersección de todos estos polígonos calculados puede computarse en tiempo $O((\alpha_i+\beta_i)n \log n)$. Por lo tanto la segunda parte toma tiempo $O((\alpha_i + \beta_i)n \log n)$.

En la tercera parte se calcula el polígono de visibilidad de $f_l(r_i)$ en tiempo $O(n)$. Se verifica que cada punto $f_l(t)$, para cada nodo t hijo de r_i, pertenezca a $V(f_l(r_i))$ en tiempo $O(\alpha_i n)$, y se calcula la intersección entre cada una de las regiones $R[s.label]$, para cada hijo Steiner s de r_i, en tiempo $O(\beta_i n \log n)$. Por lo tanto la complejidad en tiempo de esta tercera parte es $O((\alpha_i + \beta_i)n \log n)$.

En base a lo anterior podemos ver que el algoritmo en total tiene complejidad

$$\sum_{i=1}^{m} O((\alpha_i + \beta_i) n \log n) = O(mn \log n)$$

Hasta ahora solo hemos calculado la complejidad del algoritmo 4.1 el cual trabaja sobre solo un asignamiendo f_1. La complejidad de nuestro algoritmo completo esta dada por las veces que debemos ejecutar el algoritmo 4.1, que es el número de asignaciones o biyecciones posibles entre $V_t(T)$ y M, lo cual es igual a $k!$. Por lo tanto, concluimos con el siguiente teorema.

Teorema 4.3.1. *Sea P un polígono simple con n vértices, M un conjunto de k puntos dentro de P y T un árbol de Steiner con k terminales. El algoritmo Steiner-Tree-Embedded verifica si T puede ser acoplado al interior de P en tiempo $O(k!)$.*

Conclusiones

El problema de encontrar una trayectoria mínima en aristas (MLP) es un problema muy estudiado en geometría computacional. Consiste en encontrar una trayectoria que conecte a dos puntos s y t que se encuentran en el interior de una región polígonal P. Esta trayectoria debe estar compuesta únicamente con segmentos rectilíneos y debe estar totalmente contenida en el interior de P. Así mismo, se desea que el número de segmentos rectilíneos sea el mínimo.

En este trabajo se avanza en el desarrollo de algoritmos para una generalización del problema de encontrar trayectorias mínimas en aristas dentro de polígonos simples. En esta generalización se desea encontrar un árbol que conecte un conjunto M de k puntos dentro de P, de manera que este árbol tenga el menor número de aristas. Este problema fue planteado por el asesor de esta tesis y no se ha estudiado antes.

En esta tesis se describe un algoritmo que verifica si un árbol de Steiner con k terminales puede ser utilizado como un árbol de trayectorias mínimas en aristas para conectar los k puntos en el interior de P. El algoritmo toma como entrada un árbol de Steiner T, un polígono simple P con n vértices y un conjunto M de k puntos que se encuentran en el interior de P. Primero, el algoritmo crea una biyección entre los vértices terminales de T y el conjunto M, es decir, $f_l : V_t(T) \to M$. Luego, haciendo un recorrido recursivo sobre T, el algoritmo verifica que éste pueda ser acoplado en el interior de P, comprobando que cada una de sus aristas puedan ser dibujadas como un segmento rectilíneo dentro de P. Si el algoritmo detecta que esto no es posible, crea una nueva biyección $f_{l+1} : V_t(T) \to M$ y verifica de nuevo. Analizando la complejidad, el procedimiento que verifica si T puede ser acoplado dentro de P con una biyección es de orden $O(mn \log n)$, pero este procedimiento se ejecuta en el peor caso $k!$ veces, por lo que la complejidad total del algoritmo es $O(k!)$.

En [13], la cual es una tesis que se desarrolló de manera paralela a este trabajo, se demuestra que el problema del árbol de trayectorias mínimas en aristas pertenece a la clase de problemas NP-completos, por lo que no se espera encontar un algoritmo polinomial que solucione este problema. Sin embargo, pensamos que es posible reducir la complejidad con técnicas de programación dinámica. Como trabajo futuro es interesante investigar la posibilidad de crear algoritmos de aproximación para esta generalización. Además, falta explorar el problema sobre polígonos ortogonales o polígonos con agujeros.

Bibliografía

[1] B. Chazelle. *Triangulating a simple polygon in linear time.* Discrete & Computational Geometry, 6:485-524, 1991.

[2] Berg, Mark de. Cheong, Otfried. Kreveld, Marc van. *Computational Geometry: Algorithms and Aplications.* Springer, Heidelberg, 3rd Edition, 38-39, 2008.

[3] Cormen Tomas H., Leiserson Charles E., Rivest Ronald L., and Stein Cliford. *Introduction to Algorithms*, Third Edition, Mc Graw-Hill, 2009.

[4] D. Avis y G. T. Toussaint. *An optimal algorithm for determining the visibility of a polygon from an edge.* IEEE Trans. Comput. C-30 (1981), 910-914.

[5] Frank Harary. *Graph Theory.* Addison-Wesley, Philippines, 1969.

[6] Ghosh Subir, Kumar. *Computing the visibility polygon from a convex set and related problems.* Journal of Algorithms, Volume 12:75-95, 1991.

[7] Ghosh, Subir Kumar *Visibility Algorithms in the Plane.* Cambridge, New York, 2007.

[8] H. ElGindy. *Hierarchical decomposition of polygons with applications, PhD thesis.* School of Computer Science, McGill Univ., Montreal, Canada, (1985).

[9] J. L. Bentley y T. A. Ottmann. *Algorithms for reporting and counting geometric intersections.* IEEE Transactions on Computers 28, 643-647 (1979).

[10] Lee, D. T. *Visibility of a simple polygon.* Computer Vision, Graphics, and Image Processing, 22: 207-221, 1983.

[11] Lee, D. T. y Preparata, F. P. *Euclidean shortest paths in the presence of rectilinear barriers.* Networks, 14:393-415, 1984.

[12] L. Guibas, J. Hershberger, D. Leven, M. Sharir y R. Tarjan. *Linear time algorithms for visibility and shortest path problems inside trianguled simple polygons.* Algorithmica 2 (1987), 209-233.

[13] Martínez Chávez, Raúl Eduardo. *Sobre la complejidad de incrustar árboles de Steiner al interior de polígonos simples, Tesis de Licenciatura.*, FES Acatlán, UNAM, México, México, por publicarse.

[14] Preparata Franco P., Shamos Michael Ian. *Computational Geometry an Introduction.* Springer, 1985.

[15] Reinhard Diestel. *Graph Theory.* Springer, Heidelberg, 3rd Edition, 2005.

[16] R. E. Tarjan y C. Van Wyk. *An $O(nloglogn)$-time algorithm for triangulating a simple polygon.* SIAM J. Comput. 17 (1988), 143-178

[17] S. Suri. *A linear time algorithm for minimum link paths inside a simple polygon.* Comput. Vision Graph. Image Process. 35 (1986), 99-110.

[18] S. Suri. *Minimum link paths in polygons and related problems.* (Ph. D. Thesis)-Johns Hopkins University, 1987.