Universidad Autónoma Metropolitana Unidad Azcapotzalco
División de Ciencias Básicas e Ingeniería
Licenciatura en Ingeniería en Computación

Heurísticas para la cobertura mutua bipartita

Proyecto de investigación

Trimestre 2018 Invierno

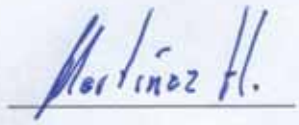
Miguel Ángel Martínez Huitrón
2123068255

Asesor
Dr. Francisco Javier Zaragoza Martínez
Profesor Titular
Departamento de Sistemas

Coasesor
Dr. Marco Antonio Heredia Velasco
Profesor Asociado
Departamento de Sistemas

9 de abril del 2018

Yo, Miguel Ángel Martinez Huitrón, doy mi autorización a la Coordinación de servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, asícomo en el Repositorio Institucional de la UAM Azcapotzalco.



Miguel Ángel Martínez Huitrón

Yo, Francisco Javier Zaragoza Martínez, declaro que aprobé el contenido del presente Reporte de proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de la UAM Azcapotzalco.

Asesor
Dr. Francisco Javier Zaragoza Martínez

Yo, Marco Antonio Heredia Velasco, declaro que aprobé el contenido del presente Reporte de proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de la UAM Azcapotzalco.

Coasesor
Dr. Marco Antonio Heredia Velasco

Resumen

Dado dos conjuntos de puntos blancos y negros sobre un plano cartesiano con distribución uniforme, queremos que cada punto negro sea cubierto con algún punto blanco y, a su vez, que cada punto blanco sea cubierto con algún punto negro. Diremos que un conjunto de coberturas mutuas es óptimo si se cumple que la suma de las distancias entre los puntos de cada coberturas es mínima. Este proyecto propone alternativas de menor costo sobre el costo computacional que se obtiene al resolver el problema con la solución óptima, donde la solución óptima se consigue planteando y resolviendo su modelo lineal entero. Teniendo la implementación de la solución óptima, se desarrollaron tres heurísticas llamadas: Dos veces, Izquierda y Derecha, con las cuales se comprueba un costo computacional inferior y en el peor de los casos valores próximos a un medio del óptimo.

Índice

1. Introducción

Cuando hablamos de un hospital y una clínica de especialidades, decimos que existe una cobertura mutua, si cada una atiende las necesidades de la otra. Cada hospital necesita por lo menos de una clínica y cada clínica por lo menos de un hospital. En la figura 1 se muestra, para un grupo de clínicas y un grupo de hospitales, un ejemplo de coberturas mutuas factibles, las clínicas se encuentran coloreadas de negro y los hospitales de blanco, cada segmento de recta representa una cobertura entre dos unidades.

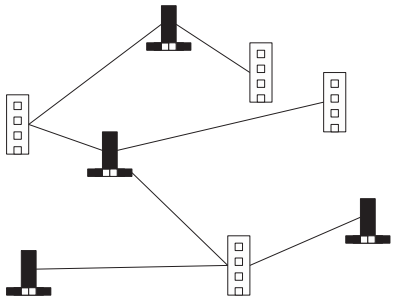


Figura 1: Coberturas factibles entre hospitales y clínicas

Podemos hablar de un segundo ejemplo de coberturas mutuas. Si alguna marca de supermercados decide ofrecer el servicio de entrega a domicilio, donde un cliente hace la petición vía telefónica y una sucursal responde a la misma, el cliente puede ir a la sucursal por su producto o recibir el producto en su casa. Aquí también, cada cliente necesita al menos una sucursal asignada y cada sucursal al menos un cliente asignado. En la figura 2 se muestra para un grupo de clientes y un grupo de sucursales, un ejemplo de coberturas factibles.

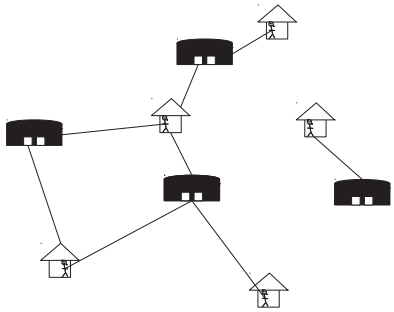
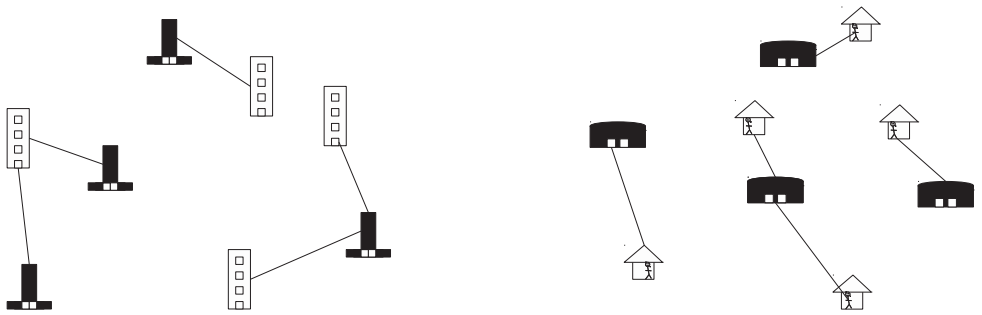


Figura 2: Coberturas factibles en clientes y sucursales.

Para el caso de un hospital y una clínica, cada cobertura tiene asociada una distancia, misma que quizá ha de recorrer un paciente entre hospital y clínica. Para el caso de los supermercados también cada cobertura tiene asociada una distancia. Tomando lo anterior en consideración, no siempre es fácil ofrecer coberturas mutuas entre los grupos involucrados, de tal forma que se puedan obtener

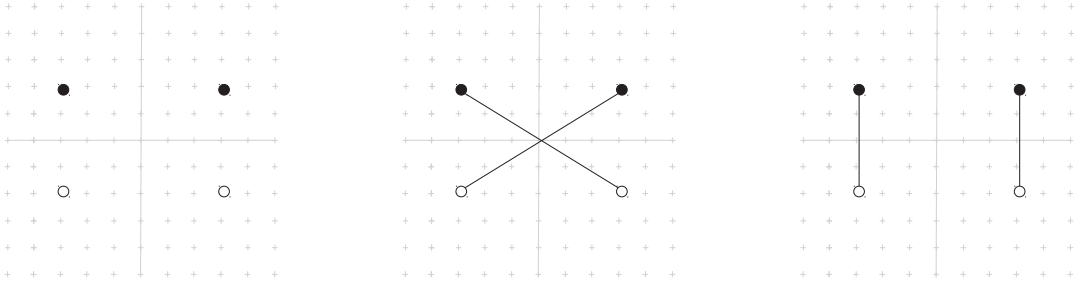
los mejores beneficios en distancias. Diremos que un conjunto de coberturas mutuas es óptima, si se cumple que los individuos de cada grupo están cubiertos y la suma de las distancias de todas las coberturas es mínima. En la figura 3 podemos ver coberturas propuestas con mayor beneficio para los grupos mostrados anteriormente.



(a) (b)

Figura 3: Coberturas con mayor beneficio.

Hoy en día, diversos algoritmos de asignación ofrecen encontrar beneficios óptimos sobre el problema en general, esto a cambio de altos tiempos de ejecución. Así entonces, se quiere proponer heurísticas que no precisamente nos lleven a una solución óptima, pero sí a una solución aceptable en tiempos menores. Con esto en mente, trabajaremos sobre un modelo matemático que nos ayudará a formalizar situaciones como las ejemplicadas, donde el grupo de clínicas (o sucursales) podrán ser representados por un conjunto finito de puntos negros en el plano, mientras que el grupo de hospitales (o clientes) por un conjunto de puntos blancos en el plano. Una cobertura mutua será entonces un segmento de recta que unirá puntos de diferente color. El costo de cada cobertura sera su longitud euclidiana, y el costo de un conjunto de coberturas la suma de cada una de ellas. Un ejemplo del modelo se muestra en la figura 4.



(a) Puntos en un plano (b) Coberturas factibles (c) Coberturas óptimas

Figura 4: Modelo matemático de coberturas mutuas

2. Justificación

De manera general, nuestro problema plantea encontrar una forma rápida de asociar dos grupos de entidades por medio de coberturas mutuas. Cada entidad de un grupo tiene la propiedad de depender de la cobertura de al menos una entidad del otro grupo, sin límite por entidad de cubrir una o varias entidades. Este problema puede envolver a grupos con cualidades de dependencia semejante, por ejemplo: grupos de máquinas, computadoras, centros de trabajo, centros de servicio, entidades orgánicas, sistemas embebidos, moléculas, partículas en física, etc. Finalmente los resultados de este proyecto, pretenden conocer más acerca de la complejidad del problema y contribuir a su estudio.

3. Objetivos

3.1. Objetivo general

- Diseñar, implementar y evaluar un algoritmo exacto y heurísticas para el problema de la cobertura mutua bipartita.

3.2. Objetivos específicos

1. Diseñar e implementar un algoritmo que genere instancias del problema utilizando distribuciones de probabilidad.
2. Diseñar e implementar un algoritmo exacto que nos entregue la solución óptima.
3. Diseñar e implementar heurísticas que obtengan rápidamente soluciones al problema.
4. Comparar los resultados del algoritmo exacto con cada heurística.

4. Antecedentes

4.1. Externos

J. Akiyama y Urrutia en su artículo [4], comienzan con una entrada A que contiene $2n$ puntos en posición general donde n se encuentran coloreados de color azul y los demás de color rojo, definen una trayectoria alterna P de A como una secuencia $p_1, p_2, ..., p_{2n}$ de puntos de A tal que p_{2i} es azul y p_{2i+1} es roja, además P es simple si no se cruza a si misma. En su trabajo implementa y diseña un algoritmo de tiempo $O(n^2)$ que encuentra una trayectoria alterna P (si existe), de un conjunto de puntos A, donde P es simple y cuyos vértices son aristas de un polígono convexo. Su trabajo, al igual que este, toma como entrada un

conjunto bicoloreado de puntos en el plano, pero con la diferencia de que busca una trayectoria alterna, no una cobertura.

S. Cabello, J. M. Díaz-Báñez y P. Pérez-Lantero en su artículo [5], estudian el problema de encontrar dos discos disjuntos sobre puntos coloreados de rojo y azul, tal que uno cubre sólo puntos rojos y el otro puntos azules. Este problema busca maximizar los puntos contenidos en los discos. Estos autores prueban que el problema se puede resolver en tiempo $O(n^4)$. También presentan un algoritmo aleatorio con alta probabilidad de devolver una aproximación a la solución óptima. Su trabajo, al igual que este, toma como entrada un conjunto de puntos bicoloreado en el plano, pero con la diferencia de que busca discos disjuntos, no una cobertura entre ellos.

S. Bereg en su artículo [6] trabaja con una entrada $S = R \cup B$ de puntos situado en el plano en posición general, donde R y B denotan puntos de color rojo y puntos de color azul, respectivamente. Define un 4-hoyo como un cuadrilatero con vertices en S, que en su interior no contiene puntos de S. Un 4-hoyo esta balanceado si tiene 2 vértices rojos y 2 azules. En este trabajo se prueba que si R y B contienen n puntos cada uno, entonces S tiene al menos $\frac{n^2-4n}{12}$ $4 - hoyo$ balanceados, esta cota es justa hasta un factor constante. Su trabajo, al igual que este, toma como entrada un conjunto de puntos bicoloreados en el plano, pero con la diferencia de que los unen formando polígonos, no buscando coberturas entre ellos.

4.2. Internos

Zelzin Marcela Márquez Navarrete [3], en su proyecto terminal con título "Heurísticas para acoplamientos euclidianos sin cruces", desarrolló dos métodos llamados veleta y cerradura convexa para encontrar sobre un conjunto de puntos, acoplamientos de costo máximo sin cruces. Su trabajo, al igual que este, toma como entrada un conjunto de puntos en el plano, pero con la diferencia de que los puntos no están coloreados y se busca acoplamientos sin cruces, no una cobertura.

José Daniel Faustinos Vargas [2], en su proyecto terminal con título "Identificación de una configuración en un conjunto de puntos en el plano", diseñó e implementó algoritmos capaces de decidir si cierta configuración de puntos estaba presente en un conjunto grande de puntos del plano. Su trabajo, al igual que este, toma como entrada un conjunto de puntos en el plano, pero sin colores.

Saúl Martínez Juarez [1], en su proyecto terminal con título "Algoritmo y heurística para incrustar métricas en una línea", trabaja con una matriz de distancias, cuyos elementos contienen las distancias entre los puntos de un plano euclidiano, esta matriz que representa un métrica, tiene la característica de poder ser incrustada en una línea recta sin que se pierda información. Este problema

es de optimización, el cual busca encontrar una incrustación sobre una línea recta de menor longitud. En nuestro trabajo al igual que en el suyo, se obtendrá una matriz de distancias el cual, contendrá las distancias, pero para construir.

5. Marco teorico

En matemáticas, un diagrama de Voronoi es una partición de un plano en regiones basadas en la distancia a puntos en un subconjunto específico del plano. Ese conjunto de puntos (llamados semillas, sitios o generadores) se especifica de antemano, y para cada semilla hay una región correspondiente que consiste en todos los puntos más cercanos a esa semilla que a cualquier otro. Estas regiones se llaman células Voronoi. El diagrama de Voronoi de un conjunto de puntos es dual a su triangulación de Delaunay.

6. Desarrollo del proyecto

Para el primer objetivo se comenzó con el desarrollo de los módulos *genRect* y *genPolar*, que nos permite generar cualquier instancia del problema, si queremos encontrar la solución exacta de nuestro problema es necesario calcular una matriz de distancias, el cual se logra con el modulo *calcDistancias*, esta matriz contiene todas las distancias euclídeas que se pueden dar entre puntos de un conjunto y el otro. El modulo *modeloLinealGurobi* hace uso de la matriz de distancias y construye un archivo plano de extensión .lp que contiene el formato de nuestro modelo lineal de entrada para Gurobi, el cual después de ejecutar Gurobi se obtiene un archivo plano de extensión .sol que contiene la solución exacta, todo lo anterior para el calculo de la solución óptima se implementan en un único modulo llamado *optimobipartitaRec*, finalmente se desarrollo el modulo llamado *masCercano* que se describe mas adelante para la construcción e implementación de las tres heurísticas: *dosVeces*, *izquierda* y *derecha*.

6.1. Generadores de coordenadas

6.1.1. Distribución uniforme rectangular

El modulo $genRect(n, s, t)$ recibirá como parámetros tres enteros n, s y t, donde n indicará el numero de puntos que se deben generar, s el límite de la coordenada $x \in [0, s]$, y t el límite de la coordenada $y \in [0, t]$ y regresará un arreglo de n puntos distintos dado por sus coordenadas $(x_1, y_1), (x_2, y_2), ...(x_n, y_n)$. Los puntos tendrán distribución uniforme en x y y.

6.1.2. Distribución uniforme polar

El módulo $genPolar(n, s)$ recibirá como parámetro un entero n y un número real s, donde n indicará el numero de puntos que se deben generar y s el límite de la coordenada $r \in [0, s]$ y regresará un arreglo de n puntos distintos dado por sus coordenadas polares $(r_1, \theta_1), (r_2, \theta_2), ...(r_n, \theta_n)$. Los puntos tendrán distribución uniforme en r y $\theta \in [0, 2\pi]$.

6.2. Solución óptima

Empezaremos con el desarrollo y la definición de nuestro modelo lineal entero.

6.2.1. Modelo lineal entero de la cobertura mutua bipartita

Al tener una instancia del problema con n puntos blancos en **A** y m puntos negros en **B**, queremos saber cual es la solución óptima a una cobertura mutua bipartita, por lo cual se construye una matriz que contiene la distancia de cada cobertura posible de tamaño $n \times m$ con valores $d_{i,k}$ tal que $i \in \{1, 2, .., n\}$ y $k \in$

$\{1, 2, .., m\}$, cuyos elementos representan las distancias entre los puntos $\mathbf{a}_i \in \mathbf{A}$ y $\mathbf{b}_k \in \mathbf{B}$, con esto se obtiene el modelo lineal entero que debe ser resuelto para obtener el valor óptimo de la cobertura mutua bipartita.

Minimizar:

$$\sum_{i=1}^{n}\sum_{k=1}^{m} d_{i,k}x_{i,k}$$

sujeto a:

$$\sum_{i=1}^{n} x_{i,k} \geq 1 \text{ para todo valor } k$$

$$\sum_{k=1}^{m} x_{i,k} \geq 1 \text{ para todo valor } i$$

$$x_{i,k} \in \{0, 1\}$$

La solución del modelo nos probé los valores $x_{i,k}$ que representan la cobertura entre dos puntos $\mathbf{a}_i$ y $\mathbf{b}_k$, el valor $x_{i,k}$ es 1 si hay cobertura y 0 si no lo hay.

La función a minimizar de nuestro modelo

$$\sum_{i=1}^{n}\sum_{k=1}^{m} d_{i,k}x_{i,k}$$

que también puede escribirse como:

$$d_{1,1}x_{1,1} + d_{1,2}x_{1,2} + d_{1,3}x_{1,3} + ... + d_{1,m}x_{1,m}+$$

$$d_{2,1}x_{2,1} + d_{2,2}x_{2,2} + d_{2,3}x_{2,3} + ... + d_{2,m}x_{2,m}+$$

$$d_{3,1}x_{3,1} + d_{3,2}x_{3,2} + d_{3,3}x_{3,3} + ... + d_{3,m}x_{3,m}+$$

$$...+$$

$$...+$$

$$d_{n,1}x_{n,1} + d_{n,2}x_{n,2} + d_{n,3}x_{n,3} + ... + d_{n,m}x_{n,m}$$

representa la suma de todas las coberturas que existen para ambos conjuntos de puntos. por otro lado hay dos grupos de restricciones principales, por ejemplo, la primera que se presenta

$$\sum_{i=1}^{n} x_{i,k} \geq 1 \text{ para todo valor } k$$

que también puede escribirse como:

$$x_{1,1} + x_{2,1} + x_{3,1} + ... + x_{n,1} \geq 1$$

$$x_{1,2} + x_{2,2} + x_{3,2} + ... + x_{n,2} \geq 1$$

$$x_{1,3} + x_{2,3} + x_{3,3} + ... + x_{n,3} \geq 1$$

$$...$$

$$...$$

$$x_{1,m} + x_{2,m} + x_{3,m} + ... + x_{n,m} \geq 1$$

Asegura que todo punto $\mathbf{b}_k$ por lo menos debe de cubrir un punto $\mathbf{a}_i$. Y el segundo grupo de restricciones

$$\sum_{k=1}^{m} x_{i,k} \geq 1 \text{ para todo valor } i$$

que también puede escribirse como:

$$x_{1,1} + x_{1,2} + x_{1,3} + ... + x_{1,m} \geq 1$$

$$x_{2,1} + x_{2,2} + x_{2,3} + ... + x_{2,m} \geq 1$$

$$x_{3,1} + x_{3,2} + x_{3,3} + ... + x_{3,m} \geq 1$$

$$...$$

$$...$$

$$x_{n,1} + x_{n,2} + x_{n,3} + ... + x_{n,m} \geq 1$$

Asegura que todo punto $\mathbf{a}_i$ por lo menos debe de cubrir un punto $\mathbf{b}_k$.
Por ultimo $x_{i,k}$ deberá tomar únicamente valores de 0 y 1

$$x_{i,k} \in \{0, 1\}$$

6.2.2. Script para Gurobi

Por ahora tenemos una función lineal con variables binarias que se deberá minimizar sujeto a ciertas restricciones. Gurobi es un paquete de software destinado a resolver a gran escala problemas de programación lineal (LP), programación de enteros mixtos (MIP), y otros problemas relacionados. A Gurobi se le debe pasar un archivo plano de extensión .lp que deberá contener un modelo lineal como el que se muestra en la Figura 5, para ello es necesario realizar un programa que pueda acoplar el modelo al formato con elementos que se obtienen de la matriz de distancias y sus respectivas restricciones.

```
Minimize
17.246011 x_0_0 + 19.284339 x_0_1 + 13.701721 x_0_2 +
30.056979 x_1_0 + 32.092339 x_1_1 + 26.228821 x_1_2 +
29.206772 x_2_0 + 31.194536 x_2_1 + 26.141737 x_2_2

Subject To
x_0_0 + x_0_1 + x_0_2 >= 1
x_1_0 + x_1_1 + x_1_2 >= 1
x_2_0 + x_2_1 + x_2_2 >= 1
x_0_0 + x_1_0 + x_2_0 >= 1
x_0_1 + x_1_1 + x_2_1 >= 1
x_0_2 + x_1_2 + x_2_2 >= 1

Binary
x_0_0 x_0_1 x_0_2
x_1_0 x_1_1 x_1_2
x_2_0 x_2_1 x_2_2

End
```

Figura 5: Ejemplo sencillo del contenido del archivo que debe ser leído por Gurobi para obtener la solución óptima.

6.2.3. Matriz distancia

Se usaran dos módulos para coordenadas polares y rectangulares, el primer módulo $matrizDistanciasRec(n, m, \mathbf{A}, \mathbf{B})$ recibirá como parámetros dos enteros n, m, y dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y regresará un arreglo D de $n \times m$ números con valores $d_{i,k}$ tal que $i \in \{1, 2, .., n\}$ y $k \in \{1, 2, .., m\}$, que serán las distancias entre los puntos $\mathbf{a}_i \in \mathbf{A}$ y $\mathbf{b}_k \in \mathbf{B}$.

El segundo módulo $matrizDistanciasPolar(n, m, \mathbf{A}, \mathbf{B})$ nos dará el mismo arreglo D, solo que $\mathbf{A}$ y $\mathbf{B}$ son arreglos de puntos en coordenadas polares.

6.2.4. Generador del archivo de entrada para Gurobi

El modulo $modeloLinealGurobi(n, m, D)$ recibirá como parámetro dos enteros n, m y un arreglo D de $m \times n$ números con valores $d_{i,k} \in D$ tal que $i \in \{1, 2, .., n\}$ y $k \in \{1, 2, .., m\}$, que son las distancias entre los puntos $\mathbf{a}_i \in \mathbf{A}$ y $\mathbf{b}_k \in \mathbf{B}$. Este modulo nos genera un archivo con extensión .lp con el formato del modelo lineal para Gurobi.

6.2.5. Modulo Óptimo

Se implemento un modulo el cual implementa todos los pasos necesario para la obtención de la solución óptima de cualquier instancia, desde las creación y lectura de archivos de Gurobi, hasta la interpretación de la solución en un formato adecuado. El modulo llamado $optimobipartitaRec(n, m, \mathbf{A}, \mathbf{B})$ recibe dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y nos devuelve una estructura que contiene los arreglos de enteros Z y Z' de tamaño n y m, con $z_i \in Z$ y $z'_k \in Z'$ tal que

$i \in \{1, 2, .., n\}$ y $k \in \{1, 2, .., m\}$, así cada cobertura esta dada entre los puntos $\mathbf{a}_i$ y $\mathbf{b}_{z_i}$, más las coberturas entre los puntos $\mathbf{b}_k$ y $\mathbf{a}_{z'_k}$. Originalmente Gurobi nos entrega una matriz de ceros y unos, de tamaño $n \times m$ que es la solución de nuestro modelo lineal entero, sin embargo por naturaleza del problema la información de la solución se puede comprimir quitando redundancias y simplificando, para obtener los arreglos Z y Z'.

6.3. Algoritmos heurísticos

6.3.1. Más cercano

Para las heurísticas propuestas necesitamos un primer módulo al que llamaremos $masCercano(n, m, \mathbf{A}, \mathbf{B})$ que recibe principalmente dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno, y nos entrega un arreglo Z de n enteros con $z_i \in Z$ tal que $i \in \{1, 2, .., n\}$, donde $\mathbf{b}_{z_i} \in \mathbf{B}$ es más cercano a $\mathbf{a}_i \in \mathbf{A}$, finalmente los puntos $\mathbf{a}_i$ y $\mathbf{b}_{z_i}$ que se obtienen del índice y el valor apuntado por el índice en el arreglo Z, formaran las primeras coberturas que unirán cada punto del conjunto $\mathbf{A}$, al punto mas cercano en $\mathbf{B}$

Para la implementación de este modulo se usaron las librerías de OpenCV y se describe a continuación los principales pasos en su implementación:

- Se define un objeto con la clase Subdiv2D que nos permite representa una subdivisión de Delaunay vacía, donde se pueden agregar un conjunto de puntos 2D.
- Se define y se le pasa un objeto tipo *Rect* que se usa para acotar los puntos que se agregaran.
- Con el método $insert()$ agregamos los m puntos del conjunto $\mathbf{B}$.
- Por medio del método $findNearest()$ podemos encontrar el índice del punto perteneciente al conjunto B más cercano a un punto de entrada, el punto de entrada no necesariamente es un punto del conjunto, usando lo anterior, se consiguen los n valores $z_i \in Z$, con lo cual decimos que el índice i del punto de entrada $\mathbf{a}_i \in \mathbf{A}$ se le asigna un índice zi del punto $\mathbf{b}_{z_i} \in \mathbf{B}$, el cual se encuentra mas cercano al punto de entrada.
- Teniendo los n valores del arreglo Z, lo devolvemos desde la función.

6.3.2. Heurística dos veces

La primera heurística llamada $dosVeces(n, m, \mathbf{A}, \mathbf{B})$ recibe dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y llama a $masCercano(n, m, \mathbf{A}, \mathbf{B})$ y a $masCercano(m, n, \mathbf{B}, \mathbf{A})$ para obtener los arreglos Z y Z' de tamaño n y m, con $z_i \in Z$ tal que $z'_k \in Z'$, $i \in \{1, 2, .., n\}$ y $k \in \{1, 2, .., m\}$, con esto se tiene una cobertura factible, donde

cada cobertura esta dada entre los puntos $\mathbf{a}_i$ y $\mathbf{b}_{z_i}$, más las coberturas entre los puntos $\mathbf{b}_k$ y $\mathbf{a}_{z'_k}$, por otro lado se deben de quitar las coberturas redundantes de Z en Z', para esto se da el pseudocódigo siguiente, que muestra como quitar y marcar la ausencia de una cobertura en Z' con un valor entero -1.

```
1: for i=1:n do
2:   if Z'[Z[i]] == i then
3:     Z'[Z[i]] = -1
4:   end if
5: end for
```

finalmente este modulo nos devuelve una estructura que contiene los arreglos Z y Z' que representan las coberturas de nuestra heurística.

6.3.3. Heurística izquierda

La segunda heurística llamada $izquierda(n, m, \mathbf{A}, \mathbf{B})$ recibe dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y llama a $masCercano(n, m, \mathbf{A}, \mathbf{B})$ y a $masCercano(m, n, \mathbf{B}, \mathbf{A})$ para obtener los arreglos Z y Z', por otro lado el objetivo de este modulo es completar una cobertura mutua que unirá cada punto del conjunto $\mathbf{A}$ al punto mas cercano en $\mathbf{B}$ y asignar a cada punto no usado de $\mathbf{B}$ su punto más cercano de $\mathbf{A}$, por lo cual se resuelve quitando las coberturas con los índices usados de $\mathbf{B}$ en Z sobre Z', para esto se da el pseudocódigo siguiente, que muestra como quitar y marcar la ausencia de una cobertura en Z' con un valor entero -1.

```
1: for i=1:n do
2:   Z'[Z[i]]= -1
3: end for
```

finalmente este modulo nos devuelve una estructura que contiene los arreglos Z y Z' que representan las coberturas de nuestra heurística.

6.3.4. Heurística derecha

La tercera heurística llamada $derecha(n, m, \mathbf{A}, \mathbf{B})$ recibe dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y llama a $masCercano(n, m, \mathbf{A}, \mathbf{B})$ y a $masCercano(m, n, \mathbf{B}, \mathbf{A})$ para obtener los arreglos Z y Z', por otro lado el objetivo de este modulo es completar una cobertura mutua que unirá cada punto del conjunto $\mathbf{B}$ al punto mas cercano en $\mathbf{A}$ y asignar a cada punto no usado de $\mathbf{A}$ su punto más cercano de $\mathbf{B}$, por lo cual se resuelve quitando las coberturas con los índices usados de $\mathbf{A}$ en Z' sobre Z, para esto se da el pseudocódigo siguiente, que muestra como quitar y marcar la ausencia de una cobertura en Z con un valor entero -1.

```
1: for k=1:m do
2:   Z[Z'[k]]= -1
3: end for
```

finalmente este modulo nos devuelve una estructura que contiene los arreglos Z y Z' que representan las coberturas de nuestra heurística.

6.3.5. Heurística mejor

Nos ofrece el mejor resultado entre la heurística izquierda y derecha.

6.3.6. Costo de la heurística

El modulo $costografobipartita(n, m, \mathbf{A}, \mathbf{B}, Z, Z')$ recibe dos arreglos $\mathbf{A}$, $\mathbf{B}$ de n y m puntos cada uno y los arreglos enteros Z, Z' (obtenidos por cualquier modulo anterior descrito) de n y m valores enteros cada uno, con $z_i \in Z$ y $z'_k \in Z'$ tal que $i \in \{1, 2, .., n\}$ y $k \in \{1, 2, .., m\}$. El costo de la heurística se calcula sumando las distancias entre los puntos $\mathbf{a}_i$ y $\mathbf{b}_{z_i}$, con $i \in \{1, 2, .., n\}$, solo para $z_i \neq -1$, más las distancias entre los puntos $\mathbf{b}_k$ y $\mathbf{a}_{z'_k}$ con $k \in \{1, 2, .., m\}$, solo para $z'_k \neq -1$.

6.4. Voronoi en la sudivicion de Delaunay

Con respecto a nuestra subdirección de Delaunay que se utilizo por medio de objeto Subdiv2D de OpenCV, si conectamos los centros de las circunferencias circunscritas en los triángulos de la subdivisión es posible construir el diagrama de Voronoi de este conjunto, esto muestra la dualidad de ambas subdivisiones, sin embargo en el objeto ya se encuentra implementado con el método $getVoronoiFacetList()$.

6.5. Evaluación

Usaremos los dos generadores de puntos para obtener varias instancias, donde n sera igual a m. Cada una de estas instancias será resuelta óptimamente y además se le aplicarán las tres heurísticas mencionadas. los valores de costo obtenidos por las heurísticas se comparan con el valor óptimo mediante una variación porcentual de la siguiente forma:

$$vp = (\frac{heuristica}{optimo} - 1) * 100 \tag{1}$$

Donde se muestra qué tan pequeña es la diferencia entre las 2 soluciones comparada con la solución óptima (que equivaldría a una unidad). Para la obtención de los resultados se entrega un solo programa que se encuentra en el apéndice, que construye y ejecuta todo lo necesario, donde lo único que se necesita hacer para

probarlo es el compilador de C++11, instalar el paquete de openCV e instalar el software Gurobi.

6.6. Especificación técnica

El valor óptimo se obtuvo mediante el uso de Gurobi, el cual se obtuvo de forma gratuita desde el sitio web del software, las heurísticas fueron implementadas en el lenguaje C++11 y se usaron las librerías de desarollo OpenCV en Ubuntu, el paquete que se descargo y se instalo se llama: libopencv-dev(3.1.0+dfsg1-1 exp1ubuntu3) la ultima versión en su momento y pueden ser fácilmente instalado con: sudo apt-get install libopencv-dev. los archivos de salida se encuentran en formato CSV.

7. Resultados

7.0.1. Distribución rectangular

n	optimo		heuristica:		heuristica		heuristica:		heuristica:	
	gurobi costo:	tiempo en s.	izquierda costo:	tiempo en s.	derecha costo:	tiempo en s.	mejor costo:	tiempo en s.	dosveces costo:	tiempo en s.
50	175.377	0.1175	185.6163	0.0005	185.6163	0.0004	185.6163	0.0004	185.6163	0.0004
100	196.1369	0.3728	222.0246	0.0021	221.1566	0.0021	221.1566	0.0021	232.874	0.0021
150	253.9587	0.615	281.2632	0.0037	287.7838	0.0036	281.2632	0.0036	299.2708	0.0034
200	258.7667	0.9988	298.6565	0.0052	291.7459	0.005	291.7459	0.005	313.625	0.0049
250	345.9729	1.5203	398.9287	0.0068	394.9867	0.0065	394.9867	0.0065	421.6389	0.0065
300	352.0746	1.938	403.8057	0.0084	407.0393	0.0081	403.8057	0.0081	427.9066	0.0081
350	397.5611	2.5735	455.4046	0.01	446.7614	0.0098	446.7614	0.0098	477.6724	0.0098
400	400.0233	3.1683	455.2707	0.0054	464.6675	0.0053	455.2707	0.0053	488.9623	0.0053
450	444.097	3.8825	511.3939	0.0138	514.9498	0.0136	511.3939	0.0136	541.0477	0.0136
500	457.9992	4.5813	528.756	0.016	522.1016	0.0157	522.1016	0.0157	557.8905	0.0157
550	468.5705	5.6982	530.4892	0.018	534.9069	0.0177	530.4892	0.0177	561.7817	0.0177
600	495.6508	6.4357	561.1938	0.0147	569.3044	0.0145	561.1938	0.0145	598.1949	0.0145
650	546.547	7.8996	617.9368	0.0115	619.7599	0.008	617.9368	0.008	652.6134	0.008
700	576.5704	8.7922	655.6871	0.0252	649.8008	0.0247	649.8008	0.0247	689.3081	0.0248
750	560.0569	9.1935	649.6939	0.0099	646.801	0.0097	646.801	0.0097	687.6685	0.0098
800	591.8088	10.2127	677.3679	0.0106	671.9619	0.0104	671.9619	0.0104	706.7517	0.0105
850	589.2086	11.4988	682.8268	0.0117	679.3301	0.0115	679.3301	0.0115	718.3861	0.0115
900	625.6592	12.9941	717.038	0.0343	717.4211	0.0338	717.038	0.0338	761.3812	0.0338
950	639.9567	14.4201	729.6016	0.0133	724.5468	0.0132	724.5468	0.0132	760.7538	0.0132
1000	657.9188	15.7542	744.066	0.0145	747.7194	0.0143	744.066	0.0143	787.8614	0.0144
1050	655.0854	17.6456	762.9216	0.0156	762.2925	0.0154	762.2925	0.0154	808.0082	0.0154
1100	691.6908	19.3396	784.2738	0.0164	795.6359	0.0175	784.2738	0.0175	829.7764	0.0165
1150	713.2803	20.9388	818.3768	0.0177	813.0165	0.0175	813.0165	0.0175	859.3478	0.0175
1200	752.14	22.8196	859.1639	0.0183	862.3166	0.0182	859.1639	0.0182	902.2443	0.0182
1250	724.5991	24.7143	830.0264	0.0197	834.1116	0.0195	830.0264	0.0195	875.6286	0.0196
1300	751.5787	26.9748	859.4243	0.0208	862.0522	0.0206	859.4243	0.0206	906.8635	0.0206
1350	778.0234	28.4578	889.4092	0.022	891.1321	0.0218	889.4092	0.0218	931.7782	0.022
1400	788.9734	31.7282	898.2122	0.0236	893.9526	0.0229	893.9526	0.0229	936.3423	0.023
1450	797.19	34.0381	913.0969	0.0239	908.1612	0.0237	908.1612	0.0237	963.1416	0.0239
1500	780.7471	37.1589	895.9403	0.0254	903.2665	0.0251	895.9403	0.0251	949.7593	0.0251
1550	838.3221	40.9868	957.0159	0.0267	954.3297	0.0262	954.3297	0.0262	1014.4525	0.0262
1600	840.6284	43.6954	955.7943	0.0277	955.9293	0.0274	955.7943	0.0274	1009.9967	0.0275
1650	856.7407	45.5023	980.2035	0.0291	980.8304	0.0286	980.2035	0.0286	1032.7225	0.0286
1700	850.8253	48.2256	983.704	0.0301	985.0448	0.0298	983.704	0.0298	1039.6785	0.0298
1750	875.5455	50.784	1005.559	0.0312	1001.7432	0.031	1001.7432	0.031	1058.7552	0.0313
1800	879.3514	55.3718	1002.5688	0.0327	993.368	0.0324	993.368	0.0324	1051.718	0.0324
1850	898.0571	57.052	1025.066	0.0338	1032.0194	0.0334	1025.066	0.0334	1085.531	0.0335
1900	923.1836	61.4282	1058.5359	0.0352	1056.4778	0.0349	1056.4778	0.0349	1121.538	0.035
1950	904.481	63.2692	1029.7469	0.0363	1033.7028	0.0361	1029.7469	0.0361	1092.2354	0.0363
2000	930.5914	68.4347	1068.3745	0.0381	1074.0643	0.0376	1068.3745	0.0376	1127.5309	0.0376
2050	914.4233	71.6717	1050.1661	0.0395	1046.4639	0.0391	1046.4639	0.0391	1106.0616	0.0391
2100	949.4417	75.756	1081.6112	0.0405	1081.807	0.0403	1081.6112	0.0403	1142.2792	0.0403
2150	947.6027	79.4612	1081.6628	0.0416	1080.6437	0.0414	1080.6437	0.0414	1136.6035	0.0412
2200	962.2249	83.9206	1109.8022	0.0435	1110.7776	0.043	1109.8022	0.043	1177.4727	0.043
2250	987.7874	89.6931	1134.4143	0.0451	1132.2784	0.0447	1132.2784	0.0447	1198.1133	0.0448
2300	993.8156	91.6936	1131.7316	0.046	1133.9639	0.0458	1131.7316	0.0458	1192.8531	0.0459
2350	985.004	94.9722	1136.4106	0.0476	1128.3478	0.0471	1128.3478	0.0471	1195.9832	0.0472
2400	1008.0361	99.4163	1150.1849	0.0495	1152.8591	0.0492	1150.1849	0.0492	1211.9956	0.0491
2450	1029.4519	103.6371	1178.9841	0.0504	1174.1517	0.0499	1174.1517	0.0499	1242.0598	0.0499
2500	1037.2715	109.4969	1185.6044	0.052	1183.9529	0.052	1183.9529	0.052	1256.3905	0.0519

Tabla 1: Resultados en distribución rectangular, 50 a 2500 puntos.

n	optimo		heuristica:		heuristica		heuristica:		heuristica:	
	gurobi costo:	tiempo en s.	izquierda costo:	tiempo en s.	derecha costo:	tiempo en s.	mejor costo:	tiempo en s.	dosveces costo:	tiempo en s.
2550	1059.3246	114.1252	1218.5625	0.1524	1220.9401	0.1349	1218.5625	0.1349	1286.3699	0.0585
2600	1057.7117	119.8433	1209.0098	0.0552	1202.4298	0.0548	1202.4298	0.0548	1273.7891	0.0547
2650	1075.5079	125.6093	1229.0228	0.0566	1219.6329	0.0565	1219.6329	0.0565	1288.6562	0.0563
2700	1093.9622	131.9298	1241.5725	0.0671	1238.6362	0.057	1238.6362	0.057	1298.6829	0.0572
2750	1077.4768	136.6776	1233.8684	0.0595	1233.1304	0.0592	1233.1304	0.0592	1307.3202	0.0593
2800	1097.3246	139.6889	1244.7173	0.0618	1251.2783	0.0612	1244.7173	0.0612	1318.9081	0.0612
2850	1121.5155	144.4894	1277.7897	0.0635	1268.9199	0.0629	1268.9199	0.0629	1343.1395	0.063
2900	1128.2574	152.6343	1295.0245	0.0646	1284.9615	0.0639	1284.9615	0.0639	1357.197	0.064
2950	1112.9271	157.6154	1283.0121	0.0704	1271.0151	0.0663	1271.0151	0.0663	1337.6707	0.0667
3000	1140.4376	163.9131	1312.5256	0.0684	1305.2471	0.0674	1305.2471	0.0674	1382.8922	0.0677
3050	1171.4087	170.6296	1335.4766	0.07	1328.4792	0.0695	1328.4792	0.0695	1401.3033	0.0695
3100	1168.2048	174.9865	1326.5282	0.0713	1331.3879	0.0707	1326.5282	0.0707	1402.9641	0.0711
3150	1166.7719	182.6178	1337.1005	0.0723	1347.0643	0.0722	1337.1005	0.0722	1413.8914	0.0719
3200	1166.179	195.6089	1332.0522	0.0745	1339.0125	0.0882	1332.0522	0.0882	1411.3563	0.0741
3250	1166.0233	192.6895	1330.5453	0.0754	1338.2749	0.0752	1330.5453	0.0752	1409.4417	0.0751
3300	1180.3419	201.1183	1351.7556	0.0772	1353.4921	0.0766	1351.7556	0.0766	1434.1748	0.0769
3350	1188.7638	206.6532	1363.8872	0.2422	1359.3234	0.1994	1359.3234	0.1994	1436.4388	0.0797
3400	1201.4072	214.8286	1376.9211	0.0806	1371.476	0.0796	1371.476	0.0796	1454.2694	0.0796
3450	1259.6115	224.9281	1442.0847	0.0832	1439.9801	0.0826	1439.9801	0.0826	1520.5446	0.0828
3500	1211.1927	225.6684	1383.8539	0.0844	1381.3453	0.084	1381.3453	0.084	1458.2666	0.0838
3550	1214.6436	235.8999	1390.7783	0.0852	1389.0271	0.0851	1389.0271	0.0851	1474.1755	0.0846
3600	1251.5607	238.544	1434.1689	0.0876	1425.7323	0.0876	1425.7323	0.0876	1503.5299	0.0875
3650	1238.3751	245.1704	1419.7108	0.0897	1424.1638	0.0895	1419.7108	0.0895	1501.7441	0.1041
3700	1234.4358	255.9268	1412.007	0.0908	1409.6598	0.0901	1409.6598	0.0901	1489.9641	0.0904
3750	1259.6528	257.9435	1442.7526	0.0928	1449.3693	0.0922	1442.7526	0.0922	1529.7163	0.0922
3800	1290.48	270.1906	1471.059	0.0947	1478.6122	0.0941	1471.059	0.0941	1554.951	0.0941
3850	1271.9612	279.92	1454.6825	0.0968	1453.9103	0.0964	1453.9103	0.0964	1540.1224	0.0961
3900	1282.6766	281.0118	1462.6835	0.0985	1459.9644	0.0983	1459.9644	0.0983	1542.0491	0.0983
3950	1286.1935	288.6353	1484.1407	0.1009	1477.8538	0.1003	1477.8538	0.1003	1567.1848	0.1008
4000	1304.7916	300.3863	1493.2948	0.1023	1494.5162	0.1015	1493.2948	0.1015	1575.8781	0.1019
4050	1303.6467	310.6259	1488.7521	0.1042	1493.4681	0.1039	1488.7521	0.1039	1569.2235	0.104
4100	1312.4113	319.787	1505.3351	0.1055	1496.6218	0.1046	1496.6218	0.1046	1582.3955	0.1046
4150	1333.0226	322.0666	1526.4774	0.1093	1518.7738	0.1086	1518.7738	0.1086	1598.8595	0.1088
4200	1324.4172	333.9944	1510.5308	0.1111	1517.037	0.1106	1510.5308	0.1106	1600.4215	0.1103
4250	1343.3745	342.9394	1534.1533	0.1134	1537.3381	0.1129	1534.1533	0.1129	1622.2922	0.1129
4300	1337.5697	345.0276	1536.045	0.1148	1534.3533	0.114	1534.3533	0.114	1620.5154	0.1143
4350	1374.9657	366.2045	1571.8877	0.1163	1579.771	0.1156	1571.8877	0.1156	1664.7622	0.1161
4400	1367.106	364.9019	1560.9359	0.1189	1567.7548	0.1179	1560.9359	0.1179	1649.371	0.118
4450	1375.3795	372.2027	1580.8969	0.1204	1571.3676	0.1196	1571.3676	0.1196	1663.4402	0.1195
4500	1382.8375	385.0817	1581.6874	0.1219	1580.9723	0.1216	1580.9723	0.1216	1672.0511	0.1215
4550	1398.0231	392.0211	1598.3087	0.1247	1603.3975	0.1237	1598.3087	0.1237	1693.1664	0.1237
4600	1405.7091	396.6644	1608.5707	0.1265	1602.8658	0.1257	1602.8658	0.1257	1696.5815	0.1258
4650	1402.3234	411.9519	1608.8439	0.1283	1608.7678	0.128	1608.7678	0.128	1701.7297	0.1276
4700	1411.8344	421.9272	1618.2318	0.1296	1615.9583	0.1295	1615.9583	0.1295	1701.8035	0.1292
4750	1428.924	434.0813	1619.5693	0.1436	1634.5989	0.1325	1619.5693	0.1325	1716.0901	0.1326
4800	1425.7272	463.9285	1638.6915	0.1354	1626.1613	0.1346	1626.1613	0.1346	1722.4215	0.1345
4850	1442.9768	449.1975	1653.4408	0.1474	1646.0546	0.1351	1646.0546	0.1351	1740.358	0.1363
4900	1446.4813	460.6017	1645.766	0.1388	1649.0321	0.1385	1645.766	0.1385	1737.9059	0.139
4950	1479.4672	469.0049	1690.8101	0.1449	1693.8137	0.1549	1690.8101	0.1549	1783.0703	0.1544
5000	1485.5719	648.9288	1688.2185	0.1445	1688.9728	0.1434	1688.2185	0.1434	1780.4011	0.1437

Tabla 2: Resultados en distribución rectangular, 2550 a 5000 puntos.

7.0.2. Variación porcentual en distribución rectangular

n	vp izquierda	vp derecha	vp mejor	vp dosveces
50	5.83845088	5.83845088	5.83845088	5.83845088
100	13.19879125	12.75624322	12.75624322	18.7303358
150	10.75155133	13.31913417	10.75155133	17.84231058
200	15.4153529	12.74476198	12.74476198	21.1999071
250	15.30634336	14.16694776	14.16694776	21.87049911
300	14.69322127	15.61166298	14.69322127	21.5386171
350	14.54958747	12.37553171	12.37553171	20.15068879
400	13.81104551	16.16010867	13.81104551	22.2334549
450	15.15364887	15.95435231	15.15364887	21.83097386
500	15.44910995	13.99618165	13.99618165	21.81036561
550	13.21438289	14.15718659	13.21438289	19.89267357
600	13.22362437	14.85997803	13.22362437	20.68877928
650	13.06196905	13.39553597	13.06196905	19.4066384
700	13.72194965	12.70103356	12.70103356	19.55315431
750	16.00498092	15.48844412	15.48844412	22.78547055
800	14.45721997	13.54374926	13.54374926	19.42230328
850	15.88880407	15.29534701	15.29534701	21.92389928
900	14.6052036	14.66643502	14.6052036	21.69264034
950	14.00796335	13.21809741	13.21809741	18.87582394
1000	13.09389548	13.64919197	13.09389548	19.7505528
1050	16.46139572	16.36536244	16.36536244	23.34394874
1100	13.38502695	15.02768289	13.38502695	19.96348658
1150	14.73424964	13.98274984	13.98274984	20.47827481
1200	14.22925253	14.64841652	14.22925253	19.9569628
1250	14.54974206	15.11352967	14.54974206	20.84318073
1300	14.3492092	14.69885988	14.3492092	20.66114966
1350	14.31651027	14.53795606	14.31651027	19.76223337
1400	13.84568859	13.30579713	13.30579713	18.67856381
1450	14.53943226	13.92029504	13.92029504	20.81706996
1500	14.75422707	15.69258471	14.75422707	21.64749635
1550	14.15849588	13.83807012	13.83807012	21.00987198
1600	13.6999773	13.71603672	13.6999773	20.14782037
1650	14.41075462	14.48392728	14.41075462	20.54084742
1700	15.61762444	15.77521261	15.61762444	22.19647206
1750	14.84942816	14.41360843	14.41360843	20.92520606
1800	14.01230498	12.96598834	12.96598834	19.60156088
1850	14.1426308	14.91690228	14.1426308	20.8754989
1900	14.66147146	14.43853639	14.43853639	21.48591028
1950	13.84947832	14.28684516	13.84947832	20.75824699
2000	14.80597177	15.41738941	14.80597177	21.16283258
2050	14.84463486	14.43976767	14.43976767	20.95728532
2100	13.92075996	13.9413826	13.92075996	20.31062044
2150	14.14728979	14.03974472	14.03974472	19.94515212
2200	15.33709011	15.43845935	15.33709011	22.36980149
2250	14.84397351	14.62774277	14.62774277	21.29262835
2300	13.87742354	14.10204267	13.87742354	20.02760874
2350	15.37116601	14.55261095	14.55261095	21.41912114
2400	14.10155847	14.36684658	14.10155847	20.23335275
2450	14.52541882	14.05600398	14.05600398	20.65253364
2500	14.3002965	14.14108071	14.14108071	21.12455611

Tabla 3: Variación porcentual en distribución rectangular de los 50 a 2500 puntos.

n	vp izquierda	vp derecha	vp mejor	vp dosveces
2550	15.03202135	15.25646624	15.03202135	21.4330244
2600	14.30428537	13.68218769	13.68218769	20.42876145
2650	14.27371198	13.40064541	13.40064541	19.81838534
2700	13.49318103	13.22477139	13.22477139	18.71369047
2750	14.51461414	14.44612079	14.44612079	21.33163331
2800	13.43200544	14.02991421	13.43200544	20.19306776
2850	13.93419886	13.14332258	13.14332258	19.76111788
2900	14.78094449	13.88903809	13.88903809	20.2914335
2950	15.28267215	14.20470397	14.20470397	20.19391926
3000	15.08964629	14.45142636	14.45142636	21.25978659
3050	14.00603393	13.40868477	13.40868477	19.6254817
3100	13.55270925	13.96870651	13.55270925	20.0957315
3150	14.59827752	15.45224049	14.59827752	21.17976101
3200	14.22364834	14.82049497	14.22364834	21.02398517
3250	14.10966659	14.7725693	14.10966659	20.87594648
3300	14.52237695	14.66949534	14.52237695	21.50503172
3350	14.73155559	14.34764417	14.34764417	20.83466875
3400	14.60902681	14.15579997	14.15579997	21.04716869
3450	14.48646666	14.3193834	14.3193834	20.71536343
3500	14.25546901	14.04835085	14.04835085	20.39922301
3550	14.50093674	14.35676276	14.35676276	21.36691783
3600	14.59043896	13.9163526	13.9163526	20.13239949
3650	14.64303505	15.00261916	14.64303505	21.26730423
3700	14.3848064	14.19466286	14.19466286	20.70000724
3750	14.53573556	15.06101523	14.53573556	21.43951889
3800	13.99316533	14.57846693	13.99316533	20.49400223
3850	14.36532026	14.30461086	14.30461086	21.08249843
3900	14.03369329	13.82170689	13.82170689	20.22119215
3950	15.39015708	14.90135815	14.90135815	21.84673612
4000	14.44699675	14.54060556	14.44699675	20.77622971
4050	14.19904641	14.56080087	14.19904641	20.37183847
4100	14.69994963	14.03603428	14.03603428	20.57161501
4150	14.51249214	13.93458746	13.93458746	19.94241508
4200	14.05249041	14.54374045	14.05249041	20.83967952
4250	14.20146058	14.4385352	14.20146058	20.76246795
4300	14.8385015	14.71202585	14.71202585	21.15371633
4350	14.32195727	14.89530248	14.32195727	21.07663486
4400	14.17811786	14.67690143	14.17811786	20.64689936
4450	14.94259584	14.24974707	14.24974707	20.94408852
4500	14.37984579	14.32813328	14.32813328	20.91450369
4550	14.32634411	14.69034381	14.32634411	21.1114752
4600	14.43126462	14.02542674	14.02542674	20.69221861
4650	14.72702374	14.72159703	14.72159703	21.35073122
4700	14.61909414	14.4580625	14.4580625	20.538464
4750	13.34187822	14.39369064	13.34187822	20.09666714
4800	14.93724045	14.05837667	14.05837667	20.81003294
4850	14.58540428	14.07353188	14.07353188	20.60886911
4900	13.77720542	14.00300163	13.77720542	20.14713913
4950	14.2850683	14.48808733	14.2850683	20.52111057
5000	13.6409823	13.69175736	13.6409823	19.84617507

Tabla 4: Variación porcentual en distribución rectangular de los 2550 a 5000 puntos.

7.0.3. Distribución polar

n	optimo		heuristica:		heuristica		heuristica:		heuristica:	
	gurobi costo:	tiempo en s.	izquierda costo:	tiempo en s.	derecha costo:	tiempo en s.	mejor costo:	tiempo en s.	dosveces costo:	tiempo en s.
50	118.9292	0.6982	138.862	0.001	134.5042	0.0009	134.5042	0.0009	145.049	0.0009
100	202.8143	0.9635	223.209	0.0022	228.6103	0.0021	223.209	0.0021	237.9746	0.0021
150	210.8817	1.3065	242.1757	0.0035	246.5346	0.0034	242.1757	0.0034	263.8434	0.0034
200	253.2649	1.5687	293.1618	0.005	294.6502	0.0049	293.1618	0.0049	311.7387	0.0049
250	281.0357	1.8847	313.6545	0.0064	315.5529	0.0062	313.6545	0.0062	334.3522	0.0062
300	301.1091	2.4398	346.2554	0.0266	351.2304	0.0081	346.2554	0.0081	371.6643	0.0081
350	317.2581	3.1207	358.7491	0.0099	356.6795	0.0096	356.6795	0.0096	376.118	0.0096
400	351.4168	3.8341	405.2996	0.0118	403.0919	0.0115	403.0919	0.0115	426.024	0.0115
450	370.7473	4.5189	424.5534	0.0138	421.2229	0.0135	421.2229	0.0135	454.61	0.0135
500	374.4601	5.3981	424.1011	0.0158	424.0131	0.0156	424.0131	0.0156	447.89	0.0156
550	403.6194	6.1903	461.5424	0.0176	463.9752	0.0174	461.5424	0.0174	486.7502	0.0174
600	417.6624	7.2469	478.147	0.0196	475.7382	0.0193	475.7382	0.0193	500.1741	0.0192
650	450.1426	8.46	514.5682	0.025	518.774	0.0217	514.5682	0.0217	547.8802	0.0219
700	446.3359	9.5473	511.4966	0.0088	514.5743	0.0086	511.4966	0.0086	545.9748	0.0086
750	458.606	9.4715	515.8112	0.0095	519.9885	0.0094	515.8112	0.0094	552.235	0.0094
800	488.9313	11.1972	556.9646	0.0105	559.0884	0.0103	556.9646	0.0103	589.0278	0.0104
850	532.5532	12.0565	613.4645	0.0114	611.4838	0.0112	611.4838	0.0112	647.3641	0.0112
900	515.1877	13.4352	587.467	0.0122	581.162	0.0121	581.162	0.0121	614.0541	0.0121
950	531.7165	15.0271	606.8818	0.0131	605.2092	0.013	605.2092	0.013	636.9782	0.023
1000	533.8798	16.7043	614.716	0.0142	609.2062	0.014	609.2062	0.014	643.5428	0.0141
1050	559.5916	18.2754	639.4927	0.0153	641.8157	0.0151	639.4927	0.0151	676.1409	0.015
1100	587.1528	19.9565	672.4646	0.016	680.1079	0.016	672.4646	0.016	716.0695	0.0158
1150	584.8829	21.7165	667.8538	0.0173	668.4625	0.017	667.8538	0.017	705.6803	0.017
1200	575.7003	23.5492	658.7444	0.018	658.8076	0.0179	658.7444	0.0179	700.6279	0.0179
1250	584.9997	25.7456	663.7076	0.0189	669.5734	0.0187	663.7076	0.0187	703.8273	0.0187
1300	642.2386	28.2351	744.2226	0.0204	736.2394	0.0201	736.2394	0.0201	781.0571	0.0201
1350	632.9423	30.4557	721.8651	0.0212	714.6099	0.021	714.6099	0.021	758.6042	0.0212
1400	636.3094	32.5612	724.4187	0.0226	726.8614	0.0223	724.4187	0.0223	765.8796	0.0223
1450	673.435	36.3135	751.7203	0.0235	770.7205	0.0234	751.7203	0.0234	801.6346	0.0233
1500	682.5369	38.1098	779.8602	0.0244	773.645	0.0242	773.645	0.0242	817.4496	0.0243
1550	702.3478	40.7014	802.1831	0.026	808.6252	0.0257	802.1831	0.0257	850.2863	0.0257
1600	689.0397	45.7435	785.3351	0.0273	793.787	0.0267	785.3351	0.0267	833.8801	0.0268
1650	698.4359	46.4352	795.9487	0.0283	793.2391	0.0279	793.2391	0.0279	839.6939	0.0279
1700	717.5116	50.5214	826.5541	0.0294	826.7104	0.0291	826.5541	0.0291	872.8112	0.0425
1750	723.5231	53.4395	827.4005	0.0306	824.9939	0.0304	824.9939	0.0304	867.2019	0.0304
1800	734.0022	57.2652	832.9077	0.0318	840.9999	0.0314	832.9077	0.0314	881.9987	0.0314
1850	721.3284	58.4527	828.6688	0.033	828.0547	0.0327	828.0547	0.0327	870.7974	0.0327
1900	734.9575	62.5451	842.6569	0.0342	845.3297	0.0397	842.6569	0.0397	893.6653	0.0342
1950	743.7089	67.0724	850.2586	0.0354	849.2132	0.0348	849.2132	0.0348	894.2835	0.0348
2000	784.3618	70.0333	895.9578	0.0374	891.7087	0.0366	891.7087	0.0366	944.1161	0.0365
2050	781.4857	74.8813	891.6616	0.0382	898.8997	0.0378	891.6616	0.0378	951.5975	0.0377
2100	793.5156	77.5004	898.3131	0.0399	907.1971	0.0396	898.3131	0.0396	951.0474	0.0396
2150	821.8798	82.1476	945.9395	0.0403	942.4234	0.04	942.4234	0.04	991.5594	0.0401
2200	800.843	87.7429	920.0972	0.042	929.954	0.0419	920.0972	0.0419	980.142	0.0418
2250	818.4113	89.6485	932.212	0.0439	941.2078	0.0434	932.212	0.0434	990.1407	0.0434
2300	820.9719	94.5788	946.6616	0.0448	945.7622	0.0446	945.7622	0.0446	1000.8575	0.0444
2350	838.964	99.6037	959.6848	0.0463	956.8986	0.0461	956.8986	0.0461	1006.4336	0.0462
2400	854.2463	103.9381	978.374	0.1322	990.396	0.1312	978.374	0.1312	1044.1688	0.054
2450	859.5474	109.0233	981.8298	0.1359	980.9275	0.1354	980.9275	0.1354	1039.4089	0.1353
2500	877.2108	112.9867	1002.6116	0.0508	1008.0046	0.0503	1002.6116	0.0503	1060.2489	0.06

Tabla 5: Resultados en distribución polar, 50 a 2500 puntos.

n	optimo		heuristica:		heuristica		heuristica:		heuristica:	
	gurobi costo:	tiempo en s.	izquierda costo:	tiempo en s.	derecha costo:	tiempo en s.	mejor costo:	tiempo en s.	dosveces costo:	tiempo en s.
2550	863.2684	115.6724	986.307	0.0526	976.5164	0.0523	976.5164	0.0523	1029.1838	0.0524
2600	869.4802	125.4782	990.9464	0.0529	984.0861	0.0527	984.0861	0.0527	1038.3966	0.0626
2650	878.5625	128.8241	1008.0249	0.0546	1003.5643	0.0545	1003.5643	0.0545	1058.1165	0.0544
2700	884.0259	134.1485	1011.9844	0.0561	1015.3335	0.0558	1011.9844	0.0558	1068.9182	0.0559
2750	919.3461	138.2837	1045.6752	0.0578	1042.572	0.0575	1042.572	0.0575	1103.5656	0.0574
2800	902.4148	141.8409	1035.2799	0.0592	1039.5751	0.0589	1035.2799	0.0589	1096.9031	0.0588
2850	922.5098	148.7133	1050.9019	0.0607	1054.0769	0.0603	1050.9019	0.0603	1111.7651	0.0604
2900	939.9679	156.7126	1078.1938	0.0626	1078.6058	0.0623	1078.1938	0.0623	1132.7759	0.0622
2950	940.0976	161.667	1072.1794	0.0644	1073.6	0.064	1072.1794	0.064	1128.8846	0.0642
3000	963.5358	166.1479	1103.5608	0.0656	1103.3397	0.0654	1103.3397	0.0654	1159.9834	0.0655
3050	961.4044	174.222	1096.1521	0.0673	1092.979	0.0667	1092.979	0.0667	1154.7893	0.0667
3100	973.3102	176.3985	1123.0387	0.0692	1115.3516	0.0688	1115.3516	0.0688	1179.4779	0.0689
3150	972.9473	183.8949	1108.688	0.0706	1110.3246	0.0703	1108.688	0.0703	1172.1598	0.0702
3200	970.009	202.2751	1103.734	0.0719	1105.0874	0.0714	1103.734	0.0714	1164.2958	0.0715
3250	976.3711	200.1206	1118.4277	0.0731	1116.929	0.0723	1116.929	0.0723	1177.7638	0.0724
3300	960.3395	205.6044	1101.7432	0.0755	1100.8433	0.0911	1100.8433	0.0911	1159.821	0.0756
3350	992.824	211.6762	1128.7645	0.0777	1127.2128	0.0771	1127.2128	0.0771	1190.5425	0.0931
3400	1007.8066	218.5471	1150.3828	0.0785	1156.2441	0.0779	1150.3828	0.0779	1217.796	0.078
3450	1007.0089	231.6598	1151.0095	0.0803	1177.3226	0.0799	1151.0095	0.0799	1241.4059	0.0796
3500	1022.4538	228.8597	1181.7231	0.0819	1177.5458	0.0814	1177.5458	0.0814	1244.9229	0.0813
3550	1039.4427	239.3406	1182.8716	0.0833	1183.5104	0.0828	1182.8716	0.0828	1246.549	0.0829
3600	1018.5237	243.9378	1163.7371	0.2442	1163.1917	0.1417	1163.1917	0.1417	1226.0439	0.0857
3650	1034.199	251.4626	1187.1104	0.0865	1184.1517	0.0861	1184.1517	0.0861	1247.3579	0.0859
3700	1066.2218	261.4979	1216.3876	0.0882	1221.4094	0.0878	1216.3876	0.0878	1287.3389	0.0877
3750	1054.3619	268.8639	1213.9501	0.0904	1204.2573	0.1021	1204.2573	0.1021	1278.6396	0.09
3800	1063.3186	273.7296	1226.3105	0.0922	1217.7798	0.0914	1217.7798	0.0914	1286.7577	0.0916
3850	1064.2217	283.0701	1213.1851	0.0935	1213.2628	0.0929	1213.1851	0.0929	1279.481	0.0934
3900	1068.5398	296.9753	1221.7896	0.109	1235.9462	0.0969	1221.7896	0.0969	1301.9648	0.0971
3950	1088.652	295.7242	1250.077	0.0984	1252.819	0.0976	1250.077	0.0976	1326.0439	0.1089
4000	1103.2703	303.0806	1253.2395	0.1002	1253.8661	0.0994	1253.2395	0.0994	1323.8387	0.0994
4050	1094.0869	317.1839	1248.0549	0.1009	1240.3591	0.1006	1240.3591	0.1006	1308.7529	0.1005
4100	1092.3577	327.6962	1251.0686	0.1034	1249.3115	0.103	1249.3115	0.103	1320.8185	0.103
4150	1108.4386	334.2388	1277.1703	0.1059	1268.6356	0.1054	1268.6356	0.1054	1349.296	0.1053
4200	1133.1366	346.3306	1288.9707	0.1073	1289.8328	0.1067	1288.9707	0.1067	1355.3855	0.1067
4250	1131.7089	353.2896	1295.4402	0.1085	1294.4308	0.1082	1294.4308	0.1082	1370.0861	0.108
4300	1113.5375	355.5816	1277.9364	0.1114	1284.0315	0.1109	1277.9364	0.1109	1356.2122	0.1112
4350	1126.8514	369.9886	1288.4131	0.113	1284.6373	0.1125	1284.6373	0.1125	1359.0334	0.1128
4400	1155.5281	373.6524	1312.3485	0.1144	1317.6215	0.1139	1312.3485	0.1139	1385.8198	0.1141
4450	1151.4596	383.7051	1311.2723	0.1182	1318.5511	0.1172	1311.2723	0.1172	1385.7148	0.118
4500	1179.3541	394.1113	1348.3356	0.1192	1347.2423	0.1184	1347.2423	0.1184	1418.7245	0.1307
4550	1172.8096	394.6935	1342.5316	0.1203	1341.2191	0.1197	1341.2191	0.1197	1418.1598	0.1196
4600	1176.5283	405.8842	1344.5485	0.1224	1346.3438	0.1217	1344.5485	0.1217	1417.7311	0.122
4650	1148.4218	424.7019	1321.5969	0.1259	1338.7317	0.125	1321.5969	0.125	1416.6036	0.1359
4700	1169.8611	433.3833	1343.5878	0.1279	1348.2412	0.1269	1343.5878	0.1269	1417.7909	0.1271
4750	1200.8496	447.0143	1372.7281	0.128	1372.6533	0.1275	1372.6533	0.1275	1447.5203	0.1391
4800	1208.4326	469.3814	1379.4404	0.142	1377.8483	0.1308	1377.8483	0.1308	1451.9098	0.1306
4850	1192.1304	460.566	1361.4907	0.132	1355.2808	0.1313	1355.2808	0.1313	1429.4149	0.1316
4900	1219.0513	473.756	1414.3871	0.1365	1416.8962	0.1361	1414.3871	0.1361	1496.3926	0.1358
4950	1194.0189	479.0688	1366.0082	0.1378	1360.6941	0.1373	1360.6941	0.1373	1441.7853	0.1372
5000	1213.2085	648.1588	1391.4585	0.1396	1384.8643	0.1393	1384.8643	0.1393	1470.5061	0.1392

Tabla 6: Resultados en distribución polar, 2550 a 5000 puntos.

7.0.4. Variación porcentual en distribución polar

n	vp izquierda	vp derecha	vp mejor	vp dosveces
50	16.76022373	13.09602688	13.09602688	21.96247852
100	10.05584912	12.71902425	10.05584912	17.33620361
150	14.83959964	16.90658791	14.83959964	25.11441249
200	15.75303171	16.34071678	15.75303171	23.08799996
250	11.6066393	12.28214067	11.6066393	18.97143317
300	14.9933363	16.64556136	14.9933363	23.43177274
350	13.07799549	12.42565596	12.42565596	18.55268628
400	15.33301766	14.7047893	14.7047893	21.23040219
450	14.5128771	13.61455633	13.61455633	22.61990849
500	13.25668609	13.23318559	13.23318559	19.60953917
550	14.35089592	14.95364197	14.35089592	20.59633407
600	14.48169622	13.90496248	13.90496248	19.75559686
650	14.3122646	15.24659075	14.3122646	21.71258619
700	14.59902732	15.28857526	14.59902732	22.32374765
750	12.47371382	13.38458284	12.47371382	20.41599979
800	13.91469517	14.34907113	13.91469517	20.4725081
850	15.19309245	14.82116716	14.82116716	21.55857856
900	14.02970218	12.80587638	12.80587638	19.19036499
950	14.13634898	13.82178285	13.82178285	19.79658333
1000	15.14127337	14.10924332	14.10924332	20.54076592
1050	14.27846665	14.69359083	14.27846665	20.82756425
1100	14.52974422	15.83150076	14.52974422	21.95624376
1150	14.18589943	14.28997155	14.18589943	20.65326239
1200	14.42488392	14.43586185	14.42488392	21.70011028
1250	13.45434878	14.45705015	13.45434878	20.31242067
1300	15.87945664	14.63642951	14.63642951	21.61478616
1350	14.04911633	12.9028507	12.9028507	19.85361067
1400	13.8469273	14.23081287	13.8469273	20.36276692
1450	11.62477448	14.44616036	11.62477448	19.03667021
1500	14.25905325	13.34845046	13.34845046	19.76635988
1550	14.21451025	15.13173388	14.21451025	21.0634247
1600	13.97530505	15.20192523	13.97530505	21.02061753
1650	13.96159619	13.57364362	13.57364362	20.22490539
1700	15.19731528	15.2190989	15.19731528	21.64419363
1750	14.35716427	14.02454186	14.02454186	19.85821876
1800	13.47482337	14.57729963	13.47482337	20.16295047
1850	14.88093357	14.79579897	14.79579897	20.72135244
1900	14.65382692	15.01749421	14.65382692	21.5941466
1950	14.32680179	14.18623604	14.18623604	20.2464432
2000	14.22761792	13.68589087	13.68589087	20.36742483
2050	14.09826181	15.02445918	14.09826181	21.76774316
2100	13.20673469	14.3263094	13.20673469	19.85238854
2150	15.09462819	14.66681624	14.66681624	20.64530604
2200	14.89108352	16.12188656	14.89108352	22.38878282
2250	13.90507438	15.00425275	13.90507438	20.98326355
2300	15.30986627	15.20031319	15.20031319	21.91129806
2350	14.38927058	14.05717051	14.05717051	19.9614763
2400	14.53066873	15.93799119	14.53066873	22.23275652
2450	14.22637076	14.12139691	14.12139691	20.92514037
2500	14.29540083	14.91019034	14.29540083	20.86591957

Tabla 7: Variación porcentual en distribución polar de los 50 a 2500 puntos.

n	vp	vp	vp	vp
	izquierda	derecha	mejor	dosveces
2550	14.25264726	13.11851563	13.11851563	19.21944554
2600	13.96997884	13.1809672	13.1809672	19.42728541
2650	14.73570748	14.22799175	14.22799175	20.43724835
2700	14.47451936	14.85336572	14.47451936	20.91480578
2750	13.74119061	13.40364635	13.40364635	20.03810099
2800	14.72328468	15.19925205	14.72328468	21.55198474
2850	13.91769497	14.26186475	13.91769497	20.51526173
2900	14.70538515	14.74921644	14.70538515	20.51218983
2950	14.04979653	14.2009085	14.04979653	20.08163833
3000	14.53241281	14.50946607	14.50946607	20.38819938
3050	14.01571493	13.68566651	13.68566651	20.11483409
3100	15.38343069	14.59364137	14.59364137	21.18211645
3150	13.95149563	14.11970618	13.95149563	20.47515832
3200	13.78595456	13.92547904	13.78595456	20.02938117
3250	14.54944744	14.39595047	14.39595047	20.62665517
3300	14.72434488	14.63063844	14.63063844	20.77197699
3350	13.69230599	13.53601444	13.53601444	19.91475831
3400	14.14717864	14.7287684	14.14717864	20.8362795
3450	14.29983389	16.91282967	14.29983389	23.27655694
3500	15.57716349	15.16860713	15.16860713	21.75835231
3550	13.7986346	13.8600906	13.7986346	19.92474429
3600	14.25724311	14.20369501	14.20369501	20.37460689
3650	14.78549099	14.49940485	14.49940485	20.61101393
3700	14.08391762	14.55490781	14.08391762	20.73837732
3750	15.13599837	14.21669353	14.21669353	21.27141544
3800	15.32860424	14.52633294	14.52633294	21.01337266
3850	13.99740298	14.0047041	13.99740298	20.22692264
3900	14.34198333	15.66683805	14.34198333	21.84523216
3950	14.82797074	15.07984186	14.82797074	21.80604087
4000	13.5931512	13.64994598	13.5931512	19.99223581
4050	14.07273956	13.36934022	13.36934022	19.62056213
4100	14.52920596	14.36835205	14.36835205	20.91446785
4150	15.22246699	14.452492	14.452492	21.72943093
4200	13.75245491	13.82853577	13.75245491	19.61360175
4250	14.46761619	14.37842364	14.37842364	21.06347312
4300	14.76366086	15.31102455	14.76366086	21.79313225
4350	14.33744503	14.00236979	14.00236979	20.60449142
4400	13.5713186	14.02764675	13.5713186	19.92956294
4450	13.87914087	14.51127769	13.87914087	20.34419618
4500	14.32830903	14.23560574	14.23560574	20.296737
4550	14.47140269	14.35949194	14.35949194	20.91986628
4600	14.28101644	14.43360946	14.28101644	20.50123231
4650	15.07939853	16.57142872	15.07939853	23.35220387
4700	14.85019888	15.2479726	14.85019888	21.1930972
4750	14.31307468	14.30684575	14.30684575	20.54134839
4800	14.15120711	14.01945793	14.01945793	20.14818203
4850	14.20652472	13.68561694	13.68561694	19.90424034
4900	16.02359146	16.22941545	16.02359146	22.75058482
4950	14.40423598	13.95917602	13.95917602	20.75062631
5000	14.69244569	14.14891175	14.14891175	21.20802813

Tabla 8: Variación porcentual en distribución polar de los 2550 a 5000 puntos.

7.0.5. Voronoi, 50 vs 50

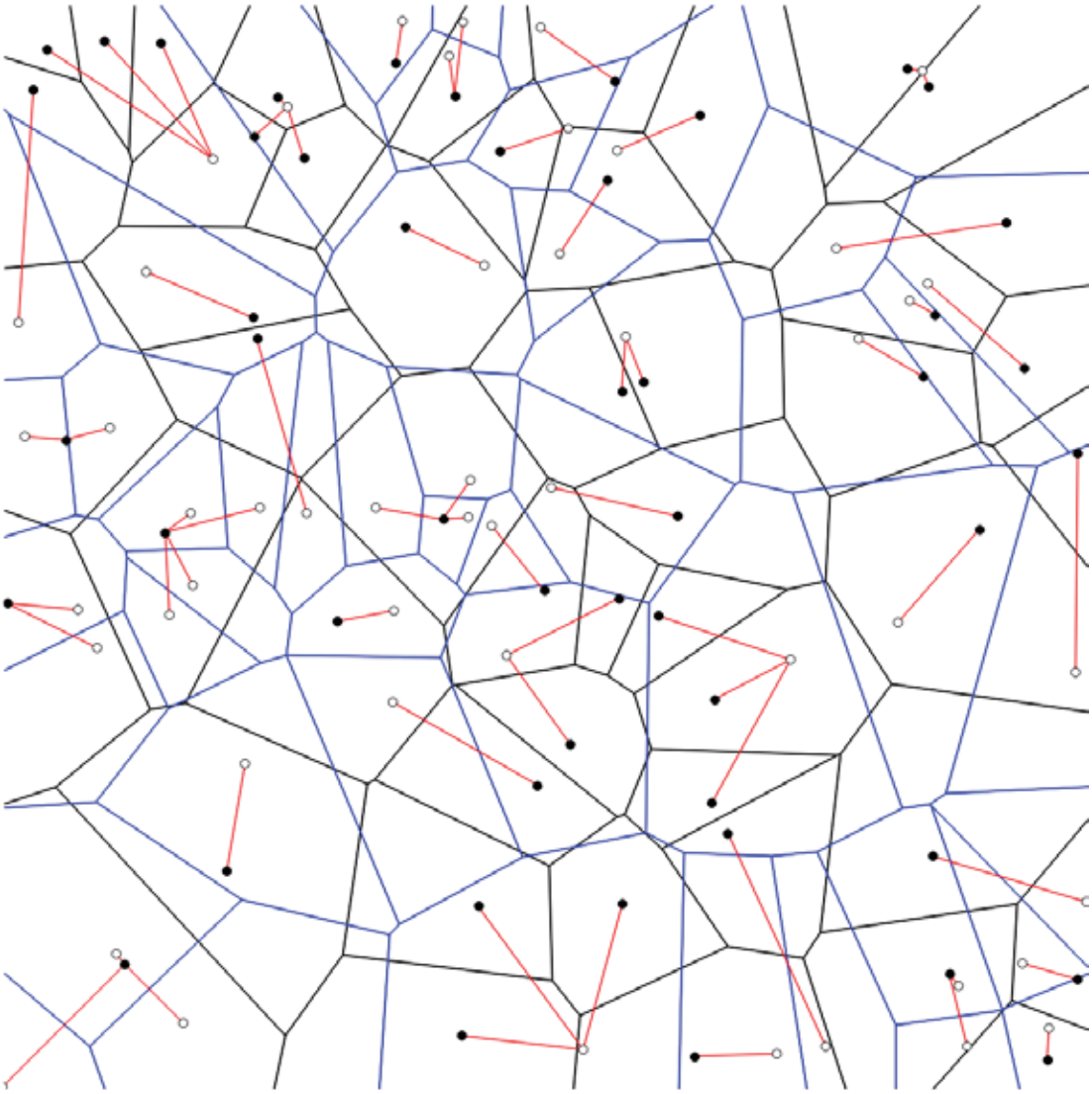


Figura 6: Óptimo, distribución rectangular, 50 vs 50

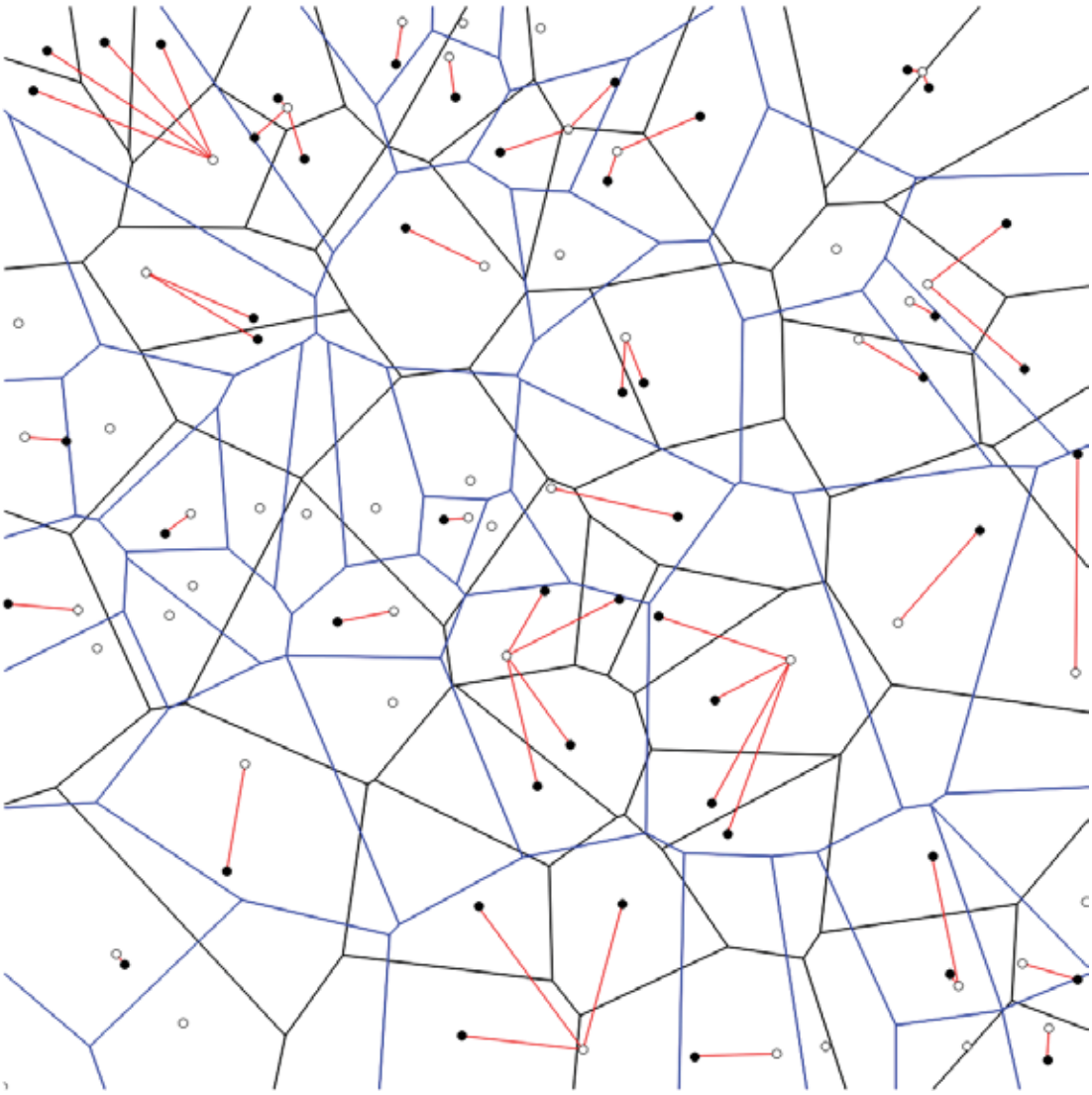


Figura 7: Unión parcial a la izquierda, distribución rectangular, 50 vs 50

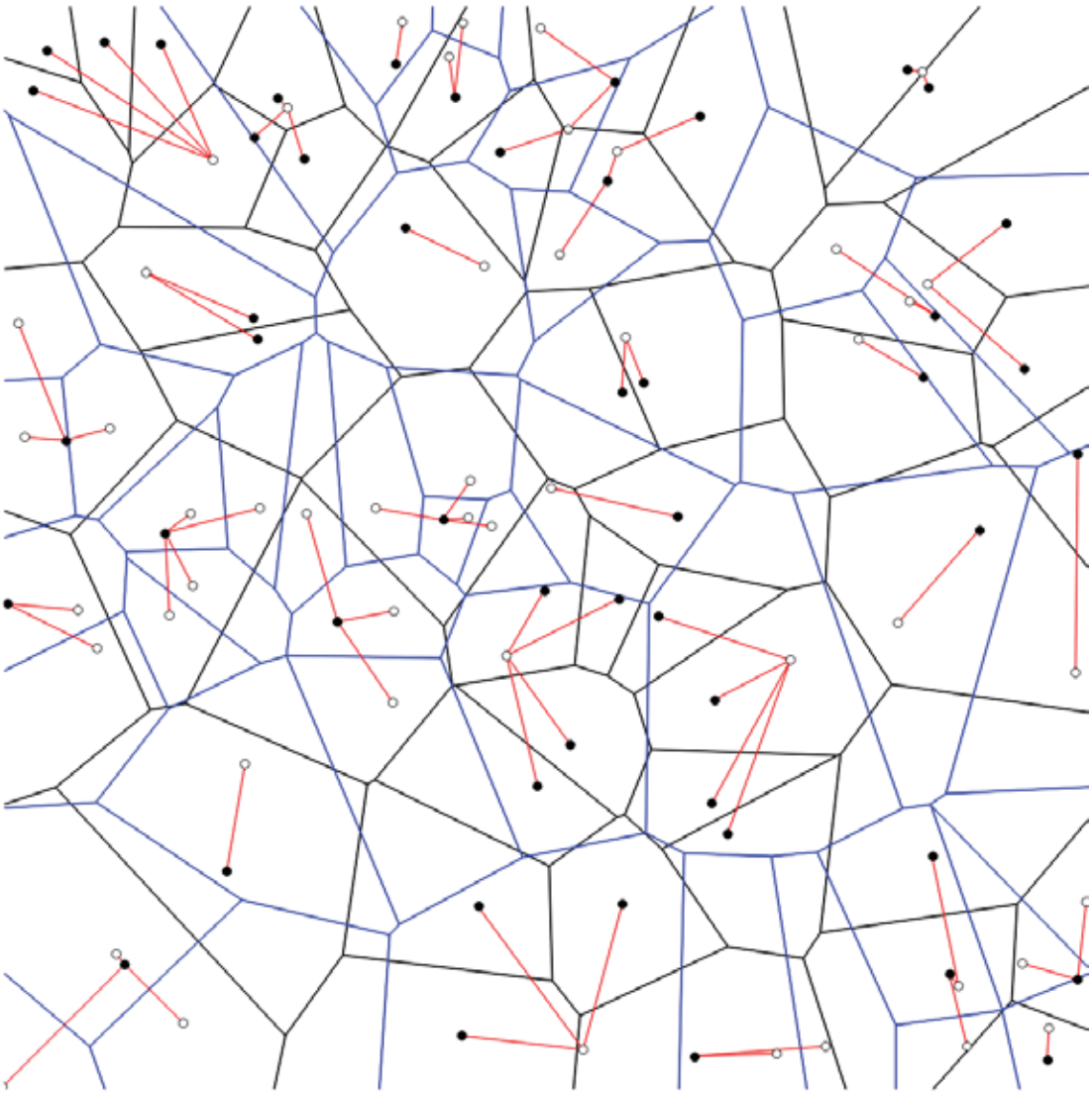


Figura 8: Izquierda, distribución rectangular, 50 vs 50

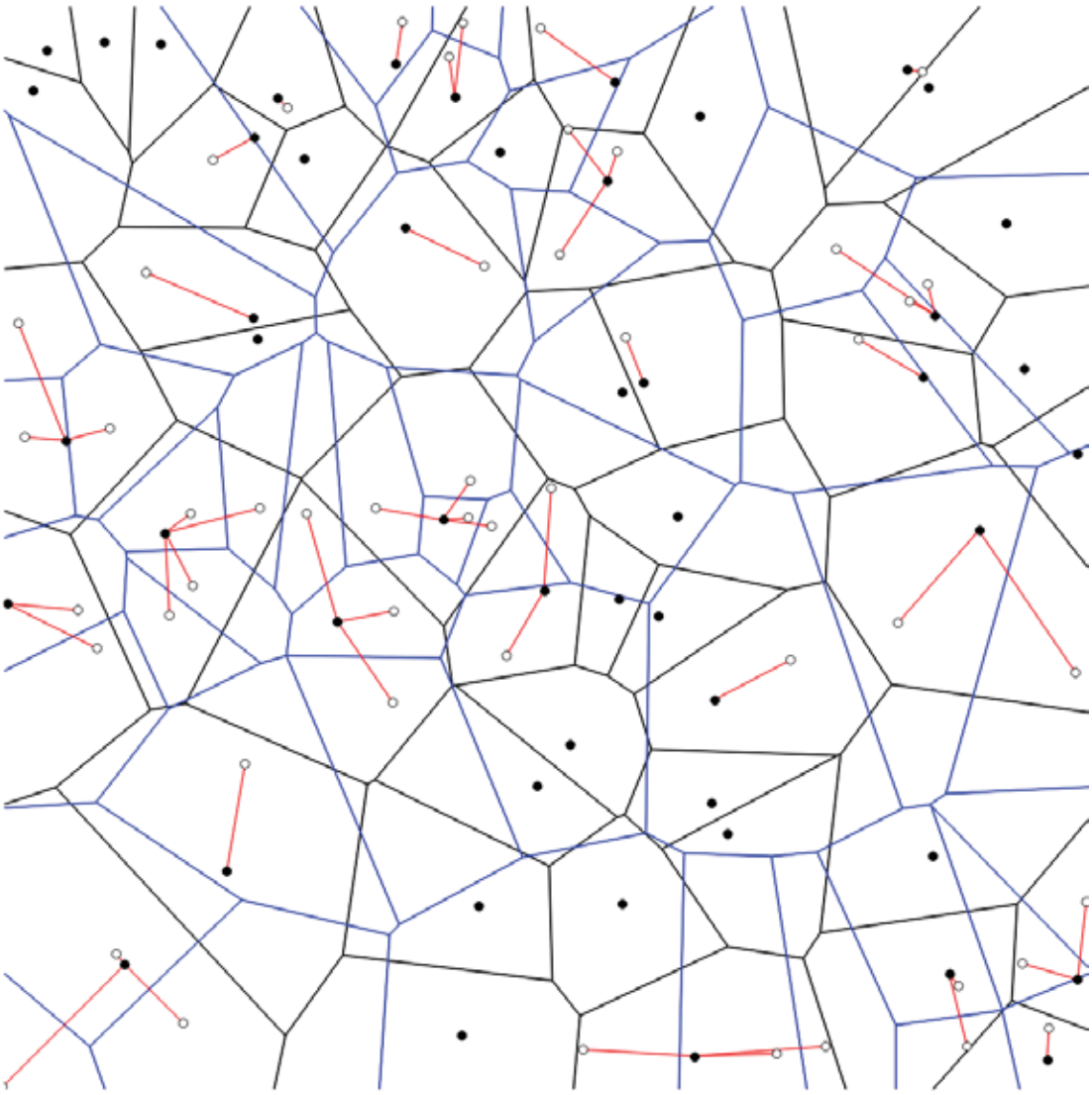


Figura 9: Unión parcial a la derecha, distribución rectangular, 50 vs 50

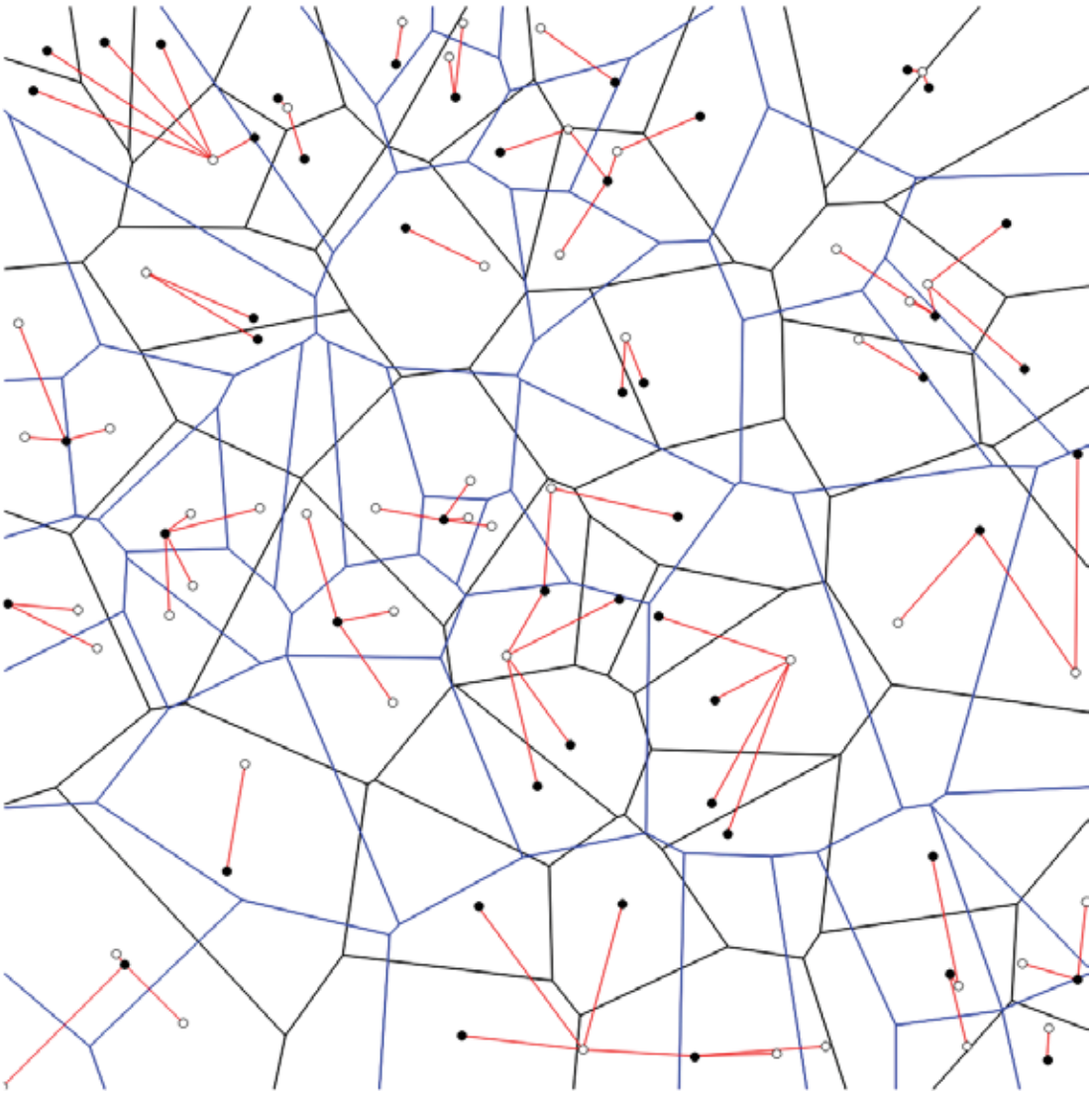


Figura 10: Derecha, distribución rectangular, 50 vs 50

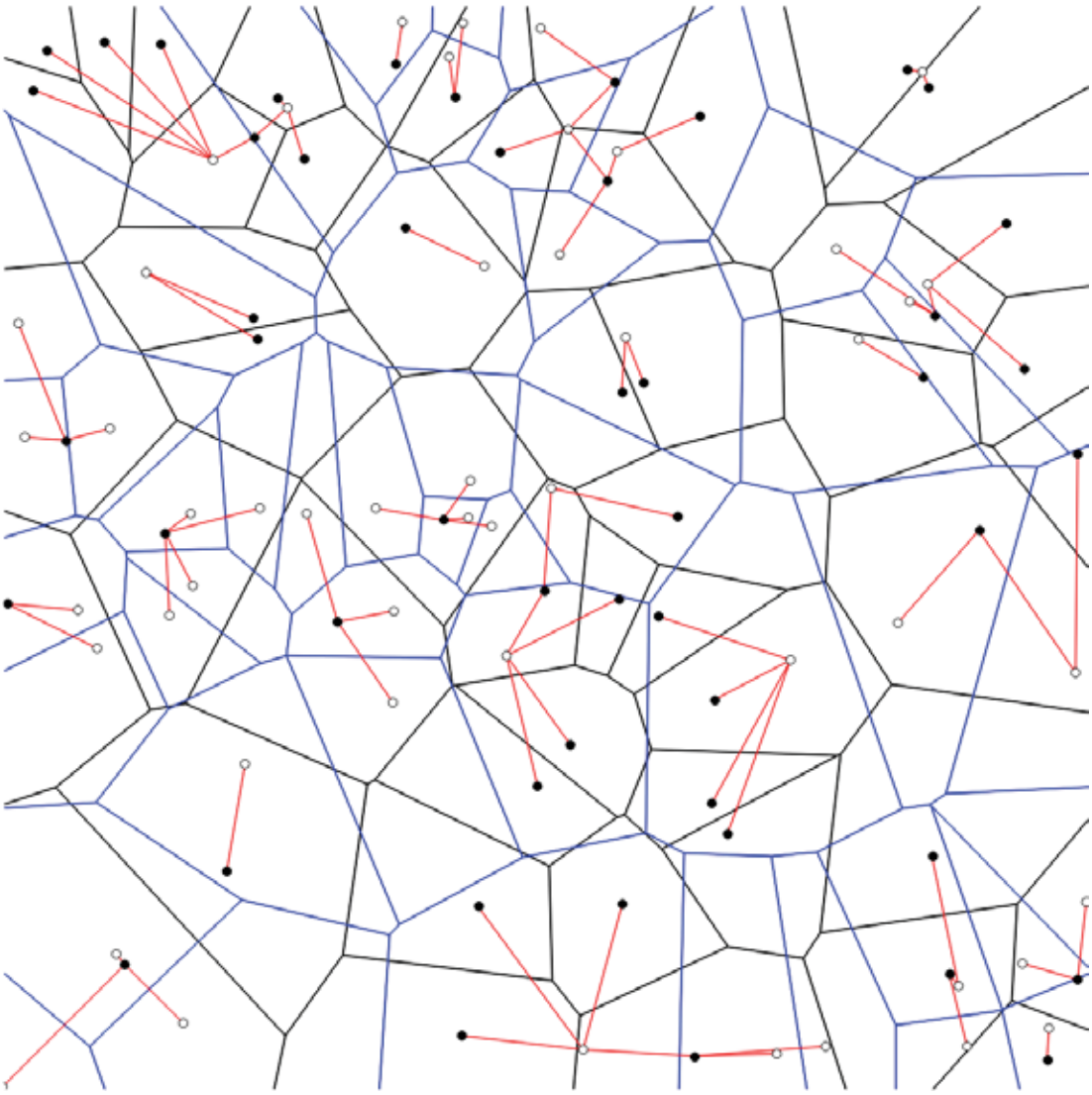


Figura 11: Dos veces, distribución rectangular, 50 vs 50

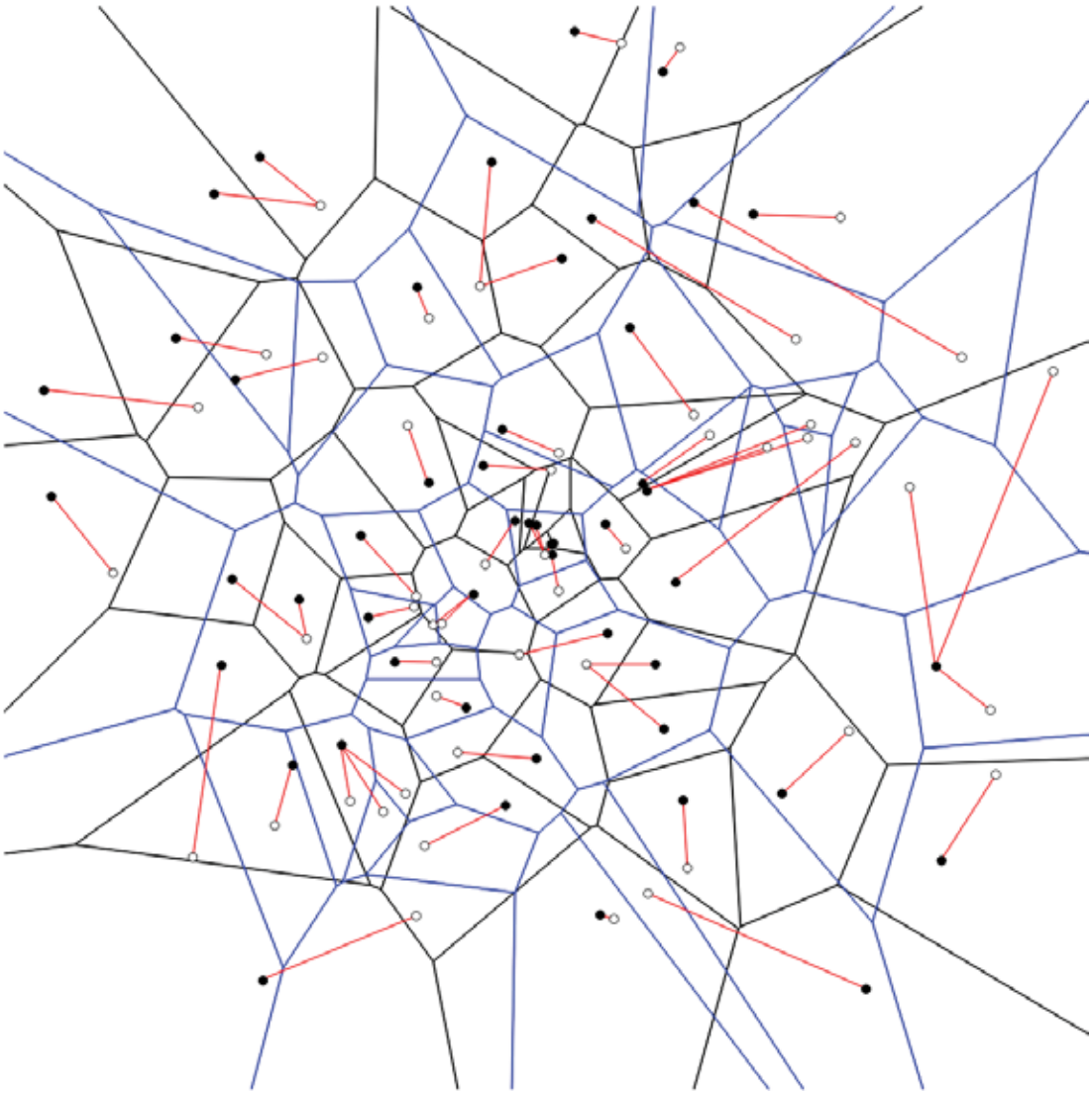


Figura 12: Óptimo, distribución polar, 50 vs 50

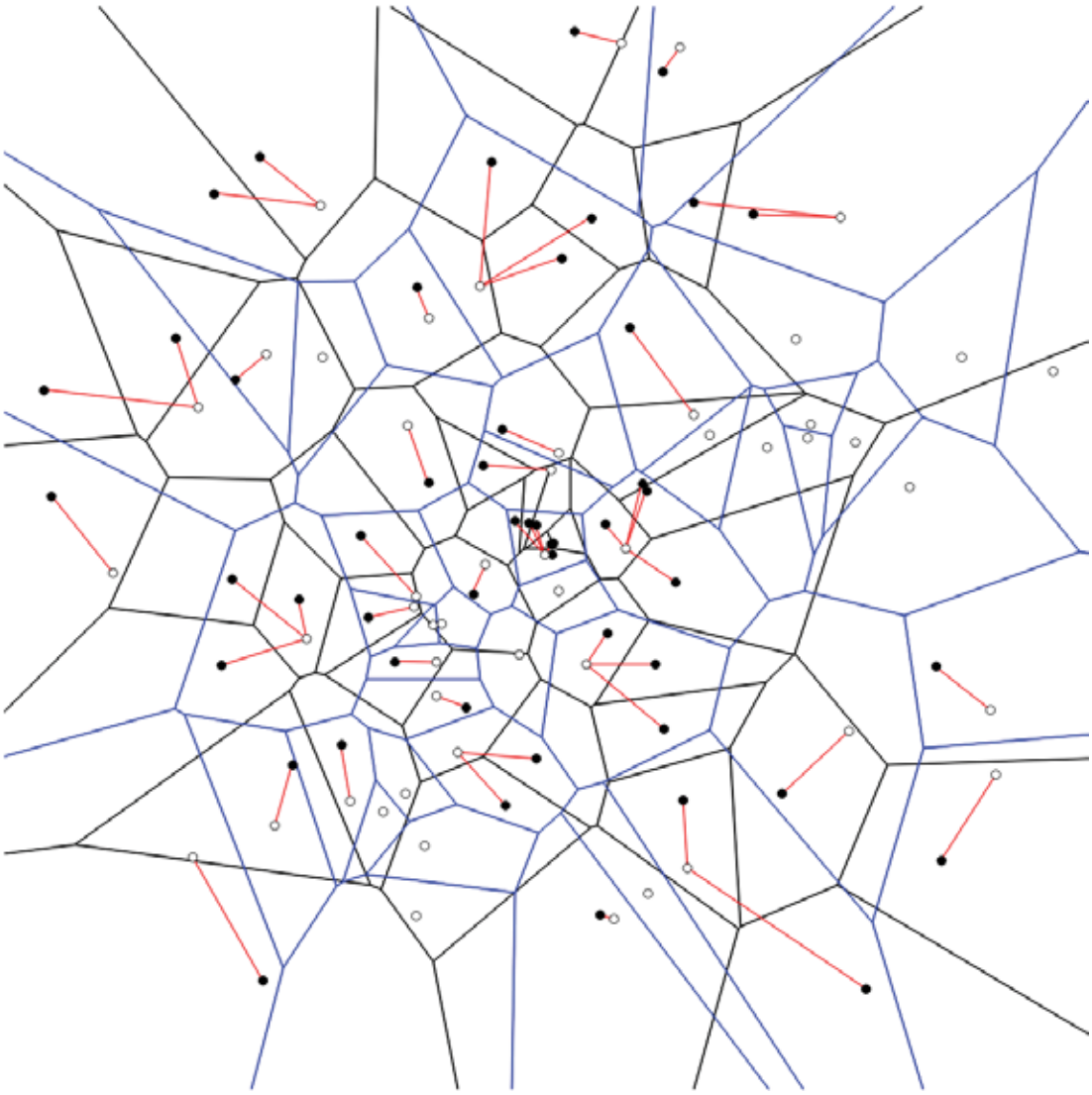


Figura 13: Unión parcial a la izquierda, distribución polar, 50 vs 50

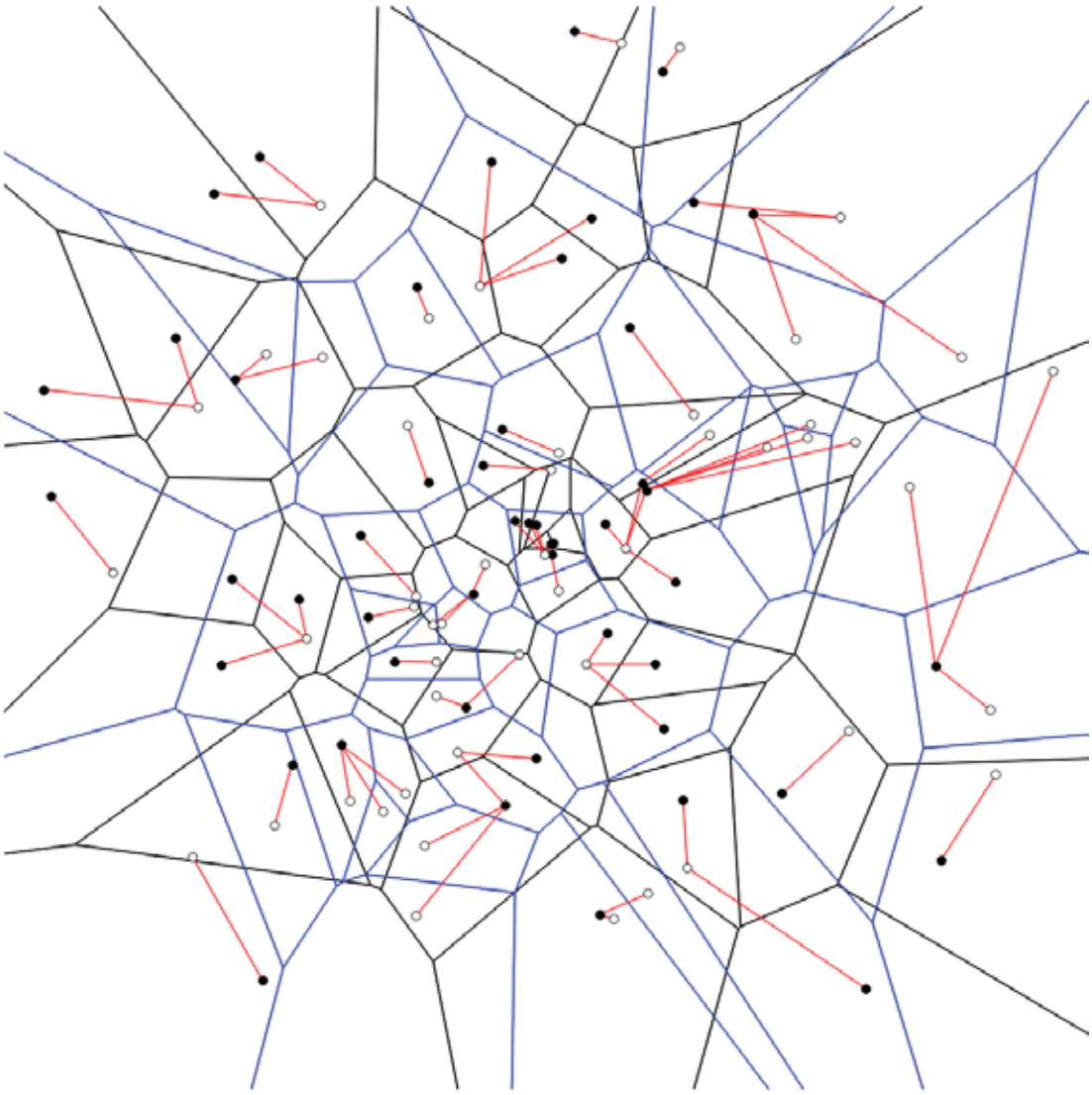


Figura 14: Izquierda, distribución polar, 50 vs 50

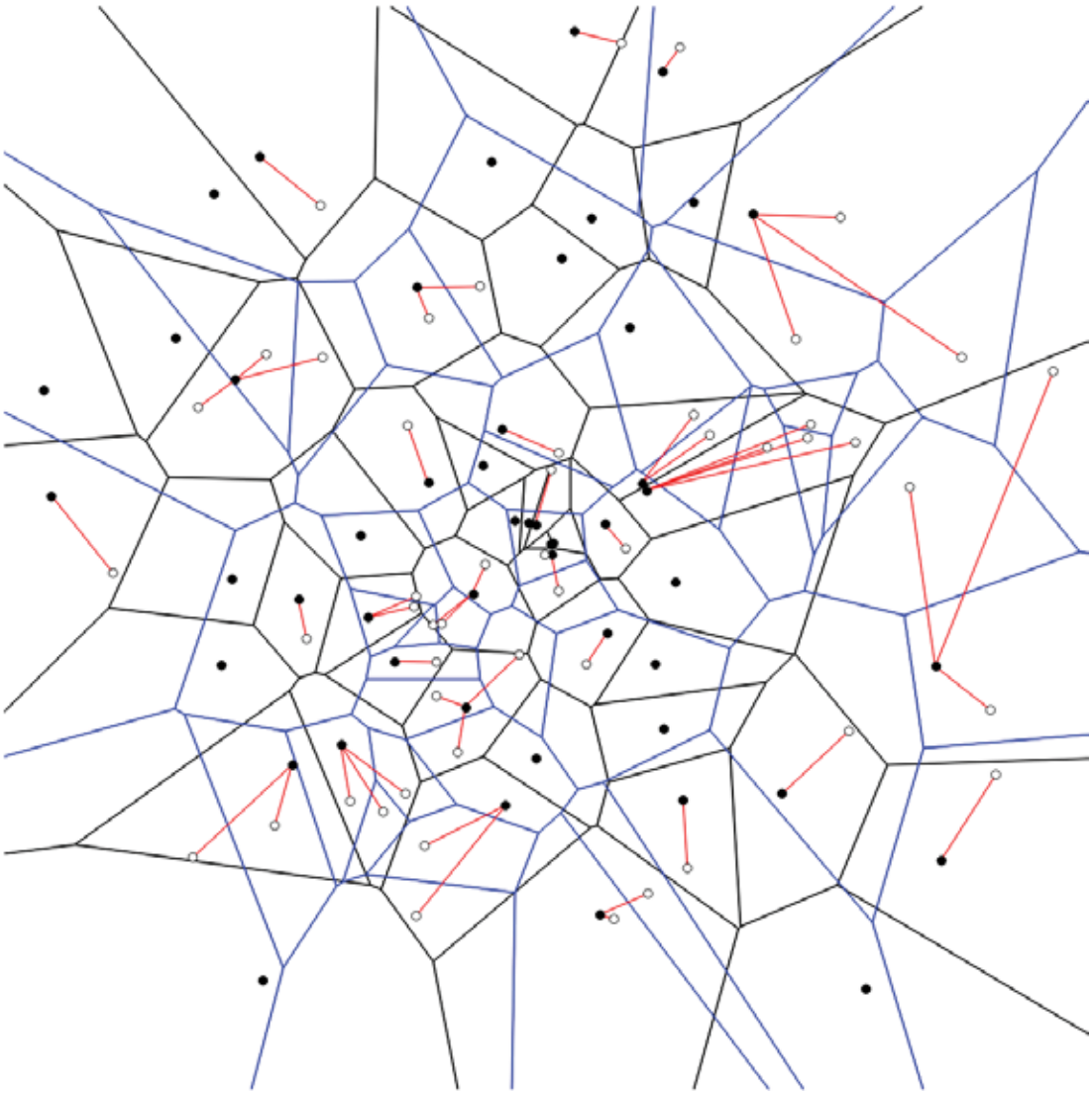


Figura 15: Unión parcial a la derecha, distribución polar, 50 vs 50

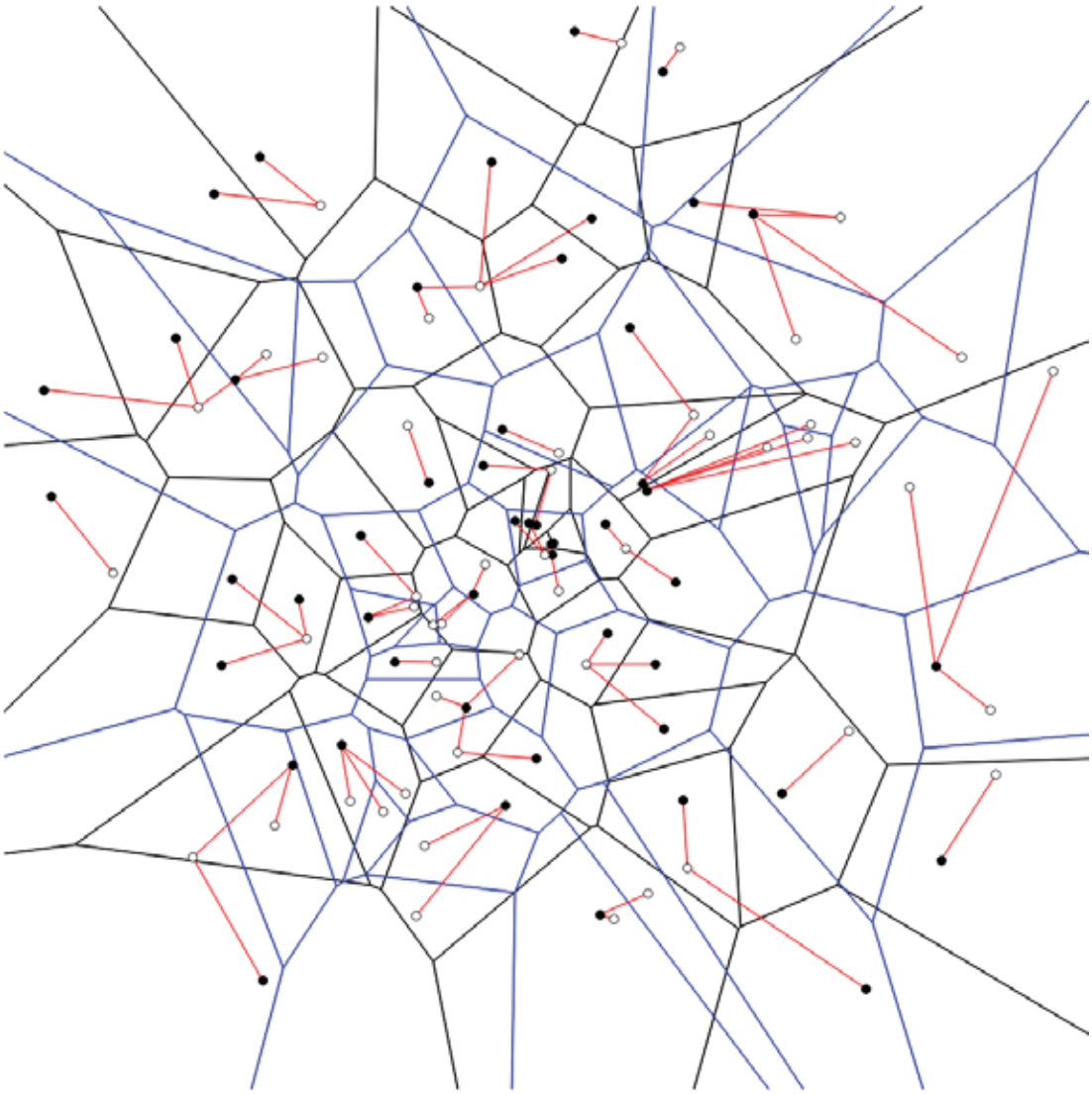


Figura 16: Derecha, distribución polar, 50 vs 50

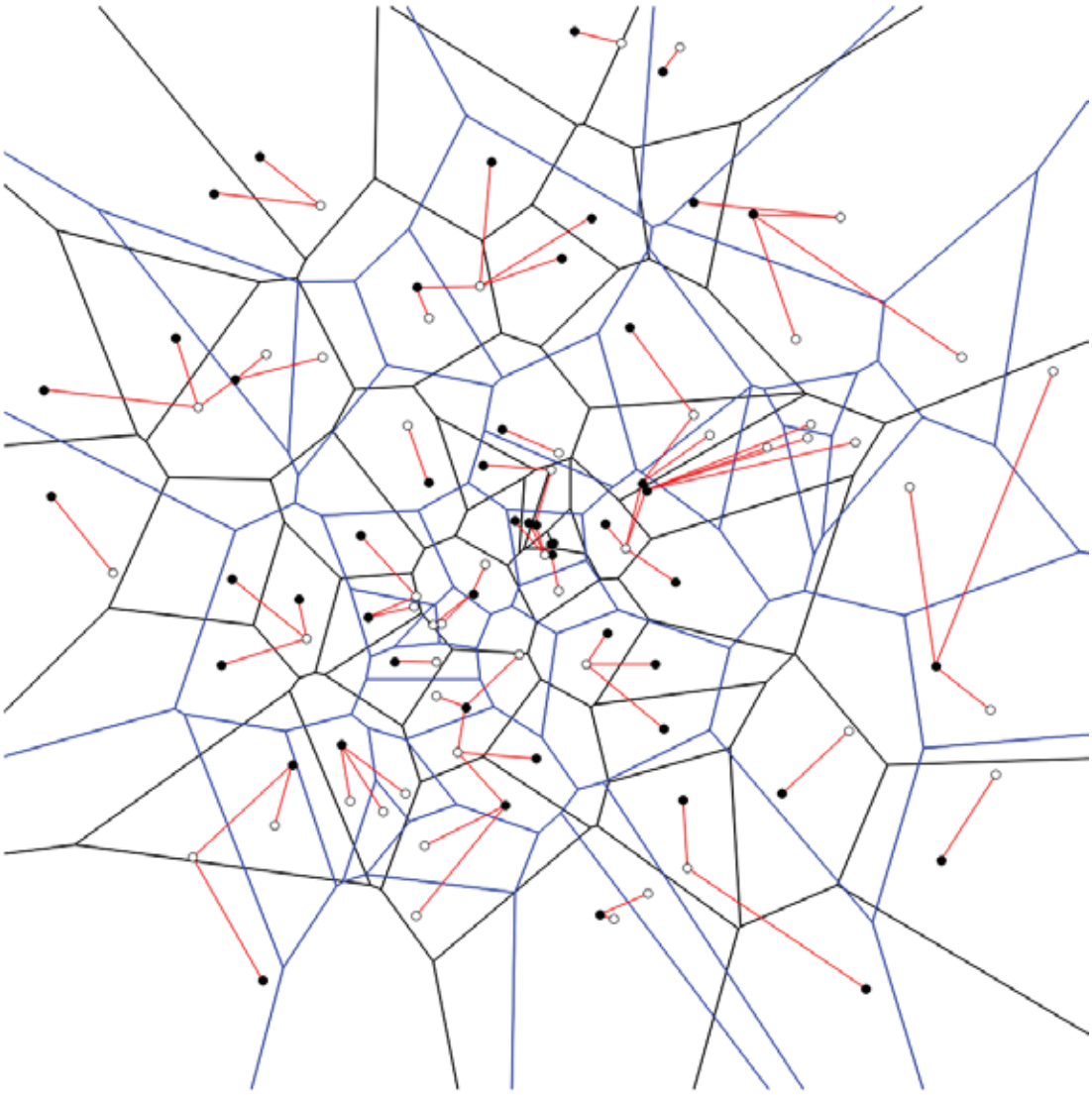


Figura 17: Dos veces , distribución polar, 50 vs 50

8. Conclusiones

En este proyecto atacamos el problema de cobertura mutua bipartita, el cual consiste en tener dos conjuntos no vacíos de puntos blancos y negros y hacer una asignación de al menos un punto blanco a cada negro y al menos un punto negro a cada blanco, de modo que la suma de las longitudes de las asignaciones sea la mínima posible. Para este problema se conoce un algoritmo exacto de tiempo $O(n^3)$ en el número total n de puntos, lo cual se considera demasiado lento.

De esta manera, propusimos algunas heurísticas que encuentran asignaciones factibles (no necesariamente óptimas) en tiempo $O(n \log n)$. Estas heurísticas fueron implementadas y ejecutadas contra un banco de instancias. Se verificó experimentalmente que el tiempo de ejecución fuera el planteado. También se verificó experimentalmente que nuestras heurísticas nunca producen una solución cuyo costo sea peor que el doble del costo óptimo, aunque a veces el costo sí es muy cercano al doble. Esto cierra una pequeña conjetura acerca de qué tan mala podía ser la primera heurística. Para la segunda heurística sigue abierta la conjetura de que nunca produce soluciones con valor mayor a 1.5 veces el óptimo. Esto podría ser el tema de una tesis de posgrado.

9. Apéndice

```
1
2  #include <opencv2/imgproc/imgproc.hpp>
3  #include <opencv2/highgui/highgui.hpp>
4  #include <stdlib.h>
5  #include <stdio.h>
6  #include <time.h>
7  #include <unistd.h>
8  #include <vector>
9  #include <iostream>
10 #include <sys/time.h>
11 using namespace cv;
12 using namespace std;
13
14 #define M_PI 3.14159265358979323846
15 #define DELTA 1000000
16 #define SIZE_WINDOW 1024
17
18 struct cordrec{
19   double x;
20   double y;
21 };
22
23 struct cordpolar{
24   double r;
25   double theta;
26 };
27
28 struct bipartita{
29   int *A;
30   int *B;
31 };
32
33 struct cordrec * genRec(int n,int s,int t);
34 struct cordpolar * genPolar(int n,int s);
35 double * matrizDistanciasPolar(int n, int m,struct cordpolar *
     arrayrthetaA,struct cordpolar *arrayrthetaB);////<-
36 double * matrizDistanciasRec(int n, int m,struct cordrec *arrayxyA,
     struct cordrec *arrayxyB);
37 void modeloLinealGurobi(const char* nombre,int n,int m, double *D);
38 struct cordrec *  polarToRect(int n,int s,struct cordpolar *
     arrayrtheta);
39 void drawbipartirta(const char *str,int n,int m,int s,int t,struct
     cordrec * cordA,struct cordrec * cordB,int *optxy,int *optyx);
40 void drawbipartitaPolar(const char *str,int n,int m,int s,struct
     cordpolar * cordA,struct cordpolar * cordB ,int *optrt,int *opttr
       );////<-
41 void drawbipartitaVoronoiColores(const char *str,int n,int m,int s,
     int t,struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *
     optxy,int  *optyx);
```

```
42 void drawbipartitaVoronoi(const char *str,int n,int m,int s,int t,
       struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *optxy,int
         *optyx);
43 void printMatrizDistancias(int n,int m,double *D);
44 struct bipartita tomarMatrizBinariaSol(const char * nombre,int n,int
       m);
45 int * bipartitaCercanoIzquierda(int n,int m,struct cordrec *arrayxyA
       ,struct cordrec *arrayxyB);
46 int * bipartitaCercanoDerecha(int n,int m,struct cordrec *arrayxyA,
       struct cordrec *arrayxyB);
47 int * bipartitaCercano(int n,int m,struct cordrec *arrayxyA,struct
       cordrec *arrayxyB);
48
49 float costografobipartitaRec(int n,int m,struct cordrec *arrayxyA,
       struct cordrec *arrayxyB,int * xy,int * yx);
50 float costografobipartitaPolar(int n,int m,struct cordpolar *
       arrayrthetaA,struct cordpolar *arrayrthetaB,int * rt ,int * tr);
51
52 struct bipartita optimobipartitaRec(int n,int m,struct cordrec *
       arrayxyA,struct cordrec *arrayxyB);
53 struct bipartita optimobipartitaPolar(int n,int m,struct cordpolar *
       arrayrthetaA,struct cordpolar *arrayrthetaB);
54 struct bipartita heuristicaDosVeces(int n,int m,struct cordrec *
       arrayxyA,struct cordrec *arrayxyB);
55 struct bipartita heuristicaIzquierda(int n,int m,struct cordrec *
       arrayxyA,struct cordrec *arrayxyB);
56 struct bipartita heuristicaDerecha(int n,int m,struct cordrec *
       arrayxyA,struct cordrec *arrayxyB);
57
58 void simulacionRec(int M,int N,int s,int t);
59 void simulacionPol(int M,int N,int s);
60
61 void guardarInstancia(const char * nombre,int indice,int n,int m,
       struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *optxy,int
         *optyx);
62
63
64 int main(){
65   ///comenzamos sembrando la semilla para la funcion rand()
66   srand(time(NULL));
67
68   int n=50,m=50,s=30,t=30;
69
70   struct cordrec *arrayxyA = genRec(n,s,t);///genera n codenadas
       rectangulares donde s y t son sus limites de x,y, respectivamente
71   struct cordrec *arrayxyB = genRec(m,s,t);///genera m codenadas
       rectangulares donde s y t son sus limites de x,y, respectivamente
72   struct cordrec *arrayxyZ;///genera n codenadas rectangulares donde
         s y t son sus limites de x,y respectivamente
73   struct cordpolar *arrayrthetaA = genPolar(n,s);///genera n
       codenadas polares donde s es el limite del angulo theta
74   struct cordpolar *arrayrthetaB = genPolar(m,s);///genera m
       codenadas polares donde s es el limite del angulo theta
```

```
75
76  struct bipartita bip_0=optimobipartitaRec(n,m,arrayxyA,arrayxyB);
77  drawbipartitaVoronoi("rectangular/optimo.png",n,m,s,t,arrayxyA,
      arrayxyB,bip_0.A,bip_0.B);
78
79  int* bip_1= bipartitaCercanoIzquierda(n,m,arrayxyA,arrayxyB);
80  drawbipartitaVoronoi("rectangular/parcial_izquierda.png",n,m,s,t,
      arrayxyA,arrayxyB,bip_1,NULL);
81
82  struct bipartita bip_2= heuristicaIzquierda(n,m,arrayxyA,arrayxyB)
      ;
83  drawbipartitaVoronoi("rectangular/izquierda.png",n,m,s,t,arrayxyA,
      arrayxyB,bip_2.A,bip_2.B);
84
85  int* bip_3= bipartitaCercanoDerecha(n,m,arrayxyA,arrayxyB);
86  drawbipartitaVoronoi("rectangular/parcial_derecha.png",n,m, s,t,
      arrayxyA,arrayxyB,NULL,bip_3);
87
88  struct bipartita bip_4= heuristicaDerecha(n,m,arrayxyA,arrayxyB);
89  drawbipartitaVoronoi("rectangular/derecha.png",n,m,s,t,arrayxyA,
      arrayxyB,bip_4.A,bip_4.B);
90
91  struct bipartita  bip_5= heuristicaDosVeces(n,m,arrayxyA,arrayxyB)
      ;
92  drawbipartitaVoronoi("rectangular/dosVeces.png",n,m,s,t,arrayxyA,
      arrayxyB,bip_5.A,bip_5.B);
93
94  struct cordrec *DistribucionPolarA = polarToRect(n,s,arrayrthetaA)
      ;///hacemos una conversi n , n puntos
95  struct cordrec *DistribucionPolarB = polarToRect(m,s,arrayrthetaB)
      ;///hacemos una conversi n , m puntos
96
97  struct bipartita bip_0_1=optimobipartitaRec(n,m,DistribucionPolarA
      ,DistribucionPolarB);
98  drawbipartitaVoronoi("polar/optima.png",n,m,s,s,DistribucionPolarA
      ,DistribucionPolarB,bip_0_1.A,bip_0_1.B);
99
100 int* bip_0_2= bipartitaCercanoIzquierda(n,m,DistribucionPolarA,
      DistribucionPolarB);
101 drawbipartitaVoronoi("polar/parcial_izquierda.png",n,m, s,s,
      DistribucionPolarA,DistribucionPolarB,bip_0_2,NULL);
102
103 struct bipartita bip_0_3= heuristicaIzquierda(n,m,
      DistribucionPolarA,DistribucionPolarB);
104 drawbipartitaVoronoi("polar/izquierda.png",n,m,s,s,
      DistribucionPolarA,DistribucionPolarB,bip_0_3.A,bip_0_3.B);
105
106 int* bip_0_4= bipartitaCercanoDerecha(n,m,DistribucionPolarA,
      DistribucionPolarB);
107 drawbipartitaVoronoi("polar/parcial_derecha.png",n,m,s,s,
      DistribucionPolarA,DistribucionPolarB,NULL,bip_0_4);
108
```

```
109   struct bipartita bip_0_5= heuristicaDerecha(n,m,DistribucionPolarA
        ,DistribucionPolarB);
110   drawbipartitaVoronoi("polar/derecha.png",n,m,s,s,
        DistribucionPolarA,DistribucionPolarB,bip_0_5.A,bip_0_5.B);
111
112   struct bipartita bip_0_6= heuristicaDosVeces(n,m,
        DistribucionPolarA,DistribucionPolarB);
113   drawbipartitaVoronoi("polar/dosVeces.png",n,m,s,s,
        DistribucionPolarA,DistribucionPolarB,bip_0_6.A,bip_0_6.B);
114
115   struct timeval ti, tf;
116   double costo,tiempo;
117   gettimeofday(&ti, NULL);    // Instante inicial
118
119   simulacionRec(100,100,s,t);
120   simulacionPol(100,100,s);
121
122   gettimeofday(&tf, NULL);    // Instante final
123   tiempo = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.tv_usec)
        /1000000.0;
124   printf("tiempo total de ejecucion : %f\n",tiempo);
125
126   return 0;
127 }
128
129
130 void simulacionRec(int M,int N,int s,int t){
131
132   struct cordrec *arrayxyA;
133   struct cordrec *arrayxyB;
134
135   FILE *file;
136   file = fopen ( "simulacionRec.csv", "w" );
137
138   fprintf(file, "N,Optimo:,,Heuristica:,,Heuristica:,,Heuristica:,,
        Heuristica:\n");
139   fprintf(file, ",Gurobi(),,Izquierda(),,Derecha(),,Mejor(),,
        DosVeces()\n");
140   fprintf(file, "  ,costo,tiempo,costo,tiempo,costo,tiempo,costo,
        tiempo,costo,tiempo\n");
141
142
143   struct timeval ti, tf;
144
145   double costo,tiempo;
146   double costo_I, costo_D,tiempo_I,tiempo_D;
147
148   struct bipartita bip;
149   int* v;
150
151   for(int n=0;n<=N || n<=M; ){
152
153     printf("################## Rec: %d \n",n);
```

```
154
155      arrayxyA = genRec(n,s,t);///genera n codenadas rectangulares
         donde s y t son sus limites de x,y, respectivamente
156      arrayxyB = genRec(n,s,t);///genera n codenadas rectangulares
         donde s y t son sus limites de x,y, respectivamente
157      fprintf(file, "%d",n);
158
159    if(n<=M){
160      gettimeofday(&ti, NULL);    // Instante inicial
161      bip= optimobipartitaRec(n,n,arrayxyA,arrayxyB);
162      gettimeofday(&tf, NULL);    // Instante final
163      tiempo = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.tv_usec)
         /1000000.0;
164      costo= costografobipartitaRec(n,n,arrayxyA,arrayxyB,bip.A,bip.B)
         ;
165      //guardarInstancia("InstaciasRec/optima/optima",n,n,n,arrayxyA,
         arrayxyB,bip.A,bip.B);
166      fprintf(file, ",%.4f,%.4f",costo,tiempo);
167    }else{
168      fprintf(file, ",,");
169    }
170    if(n<=N){
171
172
173
174      gettimeofday(&ti, NULL);    // Instante inicial
175      bip= heuristicaIzquierda(n,n,arrayxyA,arrayxyB);
176      gettimeofday(&tf, NULL);    // Instante final
177      tiempo_I =  (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.
         tv_usec)/1000000.0;
178      costo_I= costografobipartitaRec(n,n,arrayxyA,arrayxyB,bip.A,bip.
         B);
179      //guardarInstancia("InstaciasRec/izquierda/izquierda",n,n,n,
         arrayxyA,arrayxyB,bip.A,bip.B);
180      fprintf(file, ",%.4f,%.4f",costo_I,tiempo_I);
181
182
183      gettimeofday(&ti, NULL);    // Instante inicial
184      bip= heuristicaDerecha(n,n,arrayxyA,arrayxyB);
185      gettimeofday(&tf, NULL);    // Instante final
186      tiempo_D = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.
         tv_usec)/1000000.0;
187      costo_D= costografobipartitaRec(n,n,arrayxyA,arrayxyB,bip.A,bip.
         B);
188      //guardarInstancia("InstaciasRec/derecha/derecha",n,n,n,arrayxyA
         ,arrayxyB,bip.A,bip.B);
189      fprintf(file, ",%.4f,%.4f",costo_D,tiempo_D);
190
191      if(costo_D<costo_I)
192        fprintf(file, ",%.4f,%.4f",costo_D,tiempo_D);
193      else
194        fprintf(file, ",%.4f,%.4f",costo_I,tiempo_D);
195
```

```
196       gettimeofday(&ti, NULL);    // Instante inicial
197       bip= heuristicaDosVeces(n,n,arrayxyA,arrayxyB);
198       gettimeofday(&tf, NULL);    // Instante final
199       tiempo = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.tv_usec)
          /1000000.0;
200       costo= costografobipartitaRec(n,n,arrayxyA,arrayxyB,bip.A,bip.B)
          ;
201       //guardarInstancia("InstaciasRec/dosVeces/dosVeces",n,n,n,
          arrayxyA,arrayxyB,bip.A,bip.B);
202       fprintf(file, ",%.4f,%.4f",costo,tiempo);
203
204     }
205       fprintf(file, "\n");
206
207       if(n>=0) n+=50;
208       else n++;
209
210     }
211     fprintf(file, "#Tiempo en segundos,\n");
212     fclose ( file );
213 }
214
215
216 void simulacionPol(int M,int N,int s){
217
218     struct cordpolar *arrayrthetaA;
219     struct cordpolar *arrayrthetaB;
220
221     struct cordrec *DistribucionPolarA;
222     struct cordrec *DistribucionPolarB;
223
224     FILE *file;
225     file = fopen ( "simulacionPol.csv", "w" );
226
227     fprintf(file, "N,Optimo:,,Heuristica:,,Heuristica:,,Heuristica:,,
          Heuristica:\n");
228     fprintf(file, ",Gurobi(),,Izquierda(),,Derecha(),,Mejor(),,
          DosVeces()\n");
229     fprintf(file, "  ,costo,tiempo,costo,tiempo,costo,tiempo,costo,
          tiempo,costo,tiempo\n");
230
231     struct timeval ti, tf;
232
233     double costo,tiempo;
234     double costo_I, costo_D,tiempo_I,tiempo_D;
235
236     struct bipartita bip;
237     int* v;
238
239     for(int n=0;n<=N || n<=M ; ){
240
241       printf("################## Pol: %d \n",n);
242
```

```
243     arrayrthetaA = genPolar(n,s);///genera n codenadas polares donde
        s es el limite del angulo theta
244     arrayrthetaB = genPolar(n,s);///genera n codenadas polares donde
        s es el limite del angulo theta
245     DistribucionPolarA = polarToRect(n,s,arrayrthetaA);///hacemos
        una convercion
246     DistribucionPolarB = polarToRect(n,s,arrayrthetaB);///hacemos
        una convercion
247
248     fprintf(file , "%d",n);
249
250     if(n<=M){
251
252     gettimeofday(&ti, NULL);    // Instante inicial
253     bip= optimobipartitaRec(n,n,DistribucionPolarA ,
        DistribucionPolarB);
254     gettimeofday(&tf, NULL);    // Instante inicial
255     tiempo = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.tv_usec)
        /1000000.0;
256     costo= costografobipartitaRec(n,n,DistribucionPolarA ,
        DistribucionPolarB ,bip.A,bip.B);
257     //guardarInstancia("InstaciasPol/optima/optima",n,n,n,
        DistribucionPolarA ,DistribucionPolarB ,bip.A,bip.B);
258     fprintf(file , ",%.4f,%.4f",costo ,tiempo);
259
260     }else {
261
262       fprintf(file , ",,");
263
264     }
265
266     if(n<=N){
267
268
269     gettimeofday(&ti, NULL);    // Instante inicial
270     bip= heuristicaIzquierda(n,n,DistribucionPolarA ,
        DistribucionPolarB);
271     gettimeofday(&tf, NULL);    // Instante final
272     tiempo_I = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.
        tv_usec)/1000000.0;
273     costo_I= costografobipartitaRec(n,n,DistribucionPolarA ,
        DistribucionPolarB ,bip.A,bip.B);
274     //guardarInstancia("InstaciasPol/izquierda/izquierda",n,n,n,
        DistribucionPolarA ,DistribucionPolarB ,bip.A,bip.B);
275     fprintf(file , ",%.4f,%.4f",costo_I ,tiempo_I);
276
277
278     gettimeofday(&ti, NULL);    // Instante inicial
279     bip= heuristicaDerecha(n,n,DistribucionPolarA ,DistribucionPolarB
        );
280     gettimeofday(&tf, NULL);    // Instante final
281     tiempo_D = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.
        tv_usec)/1000000.0;
```

```
282     costo_D= costografobipartitaRec(n,n,DistribucionPolarA,
        DistribucionPolarB,bip.A,bip.B);
283     //guardarInstancia("InstaciasPol/derecha/derecha",n,n,n,
        DistribucionPolarA,DistribucionPolarB,bip.A,bip.B);
284     fprintf(file, ",%.4f,%.4f",costo_D,tiempo_D);
285
286     if(costo_D<costo_I)
287       fprintf(file, ",%.4f,%.4f",costo_D,tiempo_D);
288     else
289       fprintf(file, ",%.4f,%.4f",costo_I,tiempo_D);
290
291     gettimeofday(&ti, NULL);    // Instante inicial
292     bip= heuristicaDosVeces(n,n,DistribucionPolarA,
        DistribucionPolarB);
293     gettimeofday(&tf, NULL);    // Instante final
294     tiempo = (tf.tv_sec - ti.tv_sec)*1.0 + (tf.tv_usec - ti.tv_usec)
        /1000000.0;
295     costo= costografobipartitaRec(n,n,DistribucionPolarA,
        DistribucionPolarB,bip.A,bip.B);
296     //guardarInstancia("InstaciasPol/dosVeces/dosVeces",n,n,n,
        DistribucionPolarA,DistribucionPolarB,bip.A,bip.B);
297     fprintf(file, ",%.4f,%.4f",costo,tiempo);
298
299     }
300
301     fprintf(file, "\n");
302
303     if(n>=0) n+=50;
304     else n++;
305   }
306   fprintf(file, "#Tiempo en segundos,\n");
307   fclose(file);
308 }
309
310
311
312 void guardarInstancia(const char * nombre,int indice,int n,int m,
      struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *optxy,int
        *optyx){
313
314   FILE *file;
315   char s[100];
316   sprintf(s,"%s_%d.csv",nombre,indice);
317   file = fopen ( s, "w" );
318
319   fprintf(file, "Indice,A,,b,B',,,Indice,B,,a,A',,\n");
320
321   for(int i=0;i<n || i<m;i++){
322
323       fprintf(file, "%d,",i);
324
325 if(i<n){
326 fprintf(file, "%.4f,%.4f,",arrayxyA[i].x,arrayxyA[i].y);
```

```
327 if(optxy[i]!=-1)
328 fprintf(file, "%d,%.4f,%.4f,",optxy[i],arrayxyB[optxy[i]].x,arrayxyB
    [optxy[i]].y);
329 else fprintf(file, "%d,*,*,",optxy[i]);
330 }
331
332 if(i>=n)fprintf(file, ",,,,,");
333
334 if(i<m){
335 fprintf(file, ",%d,",i);
336 fprintf(file, "%.4f,%.4f,",arrayxyB[i].x,arrayxyB[i].y);
337 if(optyx[i]!=-1)
338 fprintf(file, "%d,%.4f,%.4f,",optyx[i],arrayxyA[optyx[i]].x,arrayxyA
    [optyx[i]].y);
339 else fprintf(file, "%d,*,*,",optyx[i]);
340 }
341
342   fprintf(file, "\n");
343
344   }
345
346   double costo= costografobipartitaRec(n,m,arrayxyA,arrayxyB,optxy,
      optyx);
347   fprintf(file, "Costo: %f\n",costo);
348
349   fclose ( file );
350
351 }
352
353
354
355 struct bipartita optimobipartitaPolar(int n,int m,struct cordpolar *
    arrayrthetaA,struct cordpolar *arrayrthetaB){
356
357   ///en esta funcion tomamos el tama o y lo puntos en coordenadas
      polares de los conjuntos A y B y retornamos una matriz de
      distancias
358   double *rthetamatrizD = matrizDistanciasPolar(n,m,arrayrthetaA,
      arrayrthetaB);
359   ///contruimos dentro de un archivo el modelo lineal para gurobi,
      partiendo de una matriz de distancia
360   modeloLinealGurobi("modpolar.lp",n,m,rthetamatrizD);
361
362   //contruimos el archivo nombre y contenido
363   system("gurobi_cl ResultFile=modpolar.sol modpolar.lp");
364   //
      ////////////////////////////////////////////////////////////////////////

365   struct bipartita optrtheta =tomarMatrizBinariaSol("modpolar.sol",n
      ,m);
366   return optrtheta;
367 }
368
```

```
369
370
371 float costografobipartitaRec(int n,int m,struct cordrec *arrayxyA,
      struct cordrec *arrayxyB,int * xy,int * yx){
372   float costo=0;
373
374   if(xy!=NULL)
375   for(int i=0;i<n;i++){
376       if(xy[i]!=-1){
377         costo+= sqrt(pow(arrayxyA[i].x-arrayxyB[xy[i]].x,2) + pow(
      arrayxyA[i].y-arrayxyB[xy[i]].y,2));
378       }
379   }
380
381   if(yx!=NULL)
382   for(int i=0;i<m;i++){
383     if(yx[i]!=-1){
384       costo+= sqrt(pow(arrayxyA[yx[i]].x-arrayxyB[i].x,2) + pow(
      arrayxyA[yx[i]].y-arrayxyB[i].y,2));
385     }
386   }
387   return costo;
388 }
389
390 float costografobipartitaPolar(int n,int m,struct cordpolar *
      arrayrthetaA,struct cordpolar *arrayrthetaB,int * rt ,int * tr){
391   float costo=0;
392
393   if(rt!=NULL)
394     for(int i=0;i<n;i++){
395       if(rt[i]!=-1){
396         costo+= sqrt(pow(arrayrthetaA[i].r,2) + pow(arrayrthetaB[rt[
      i]].r,2) - 2*arrayrthetaA[i].r*arrayrthetaB[rt[i]].r*cos(
      arrayrthetaA[i].theta-arrayrthetaB[rt[i]].theta));
397       }
398     }
399   if(tr!=NULL)
400     for(int i=0;i<m;i++){
401       if(tr[i]!=-1){
402         costo+= sqrt(pow(arrayrthetaA[tr[i]].r,2) + pow(arrayrthetaB
      [i].r,2) - 2*arrayrthetaA[tr[i]].r*arrayrthetaB[i].r*cos(
      arrayrthetaA[tr[i]].theta-arrayrthetaB[i].theta));
403       }
404     }
405
406   return costo;
407 }
408
409 struct bipartita optimobipartitaRec(int n,int m,struct cordrec *
      arrayxyA,struct cordrec *arrayxyB){
410
411   ///en esta funcion tomamos el tama o y lo puntos en coordenadas
       rectangulares del conjunto A y B y devolvemos una matriz de
```

```
    distancias
412   double *xymatrizD = matrizDistanciasRec(n,m,arrayxyA,arrayxyB);
413   ///contruimos dentro de un archivo el modelo lineal para gurobi,
      partiendo de una matriz de distancia
414   modeloLinealGurobi("modrec.lp",n,m,xymatrizD);
415   ////////////ejecutar el modelos con gurobi "modrec.mod"
416
417   system("gurobi_cl ResultFile=modrec.sol modrec.lp");
418   ///obtener la matriz binaria con nuestra solucion
419     struct bipartita bip=tomarMatrizBinariaSol("modrec.sol",n,m);
420   return bip;
421 }
422
423 struct cordrec * genRec(int n,int s,int t){
424   struct cordrec * cord= (struct cordrec*)malloc(n*sizeof(struct
      cordrec));
425     for(int i=0;i<n;i++){
426       cord[i].x=(double(s)/DELTA)*(rand()%(DELTA+1));
427       cord[i].y=(double(t)/DELTA)*(rand()%(DELTA+1));
428       }
429   return cord;
430 }
431
432 struct cordpolar * genPolar(int n,int s){
433   struct cordpolar * cord = (struct cordpolar*)malloc(n*sizeof(
      struct cordpolar));
434   for(int i=0;i<n;i++){
435     cord[i].r=(double(s)/DELTA)*(rand()%(DELTA+1));
436     cord[i].theta=((2.0*M_PI)/DELTA)*(rand()%(DELTA+1));
437     }
438   return cord;
439 }
440
441 double * matrizDistanciasRec(int n, int m,struct cordrec *arrayxyA,
      struct cordrec *arrayxyB){
442   double *D = (double *)malloc(sizeof(double)*n*m);
443   for(int i=0;i<n;i++){
444     for(int j=0;j<m;j++){
445       D[i*m+j]= sqrt(pow(arrayxyA[i].x-arrayxyB[j].x,2) + pow(
      arrayxyA[i].y-arrayxyB[j].y,2));
446     }
447   }
448   return D;
449 }
450
451 double * matrizDistanciasPolar(int n, int m,struct cordpolar *
      arrayrthetaA,struct cordpolar *arrayrthetaB){
452   double *D = (double *)malloc(sizeof(double)*n*m);
453   for(int i=0;i<n;i++){
454     for(int j=0;j<m;j++){
455       D[i*m+j]= sqrt(pow(arrayrthetaA[i].r,2) + pow(arrayrthetaB[j].
      r,2) - 2*arrayrthetaA[i].r*arrayrthetaB[j].r*cos(arrayrthetaA[i].
      theta-arrayrthetaB[j].theta));
```

```
456       }
457     }
458     return D;
459 }
460
461 struct cordrec * polarToRect(int n,int s,struct cordpolar *
      arrayrtheta){
462     struct cordrec *arrayrthetaRct= (cordrec *)malloc(sizeof(cordrec)*
      n); ///prepararemos nuestro arreglo Z
463     for(int i=0;i<n;i++){ ////lo convertimos de un vector de punto a
      un arreglo de puntos
464       arrayrthetaRct[i].x=double((0.5)*(arrayrtheta[i].r)*cos(
      arrayrtheta[i].theta)+s/2.0);
465       arrayrthetaRct[i].y=double((0.5)*(arrayrtheta[i].r)*sin(
      arrayrtheta[i].theta)+s/2.0);
466     }
467     return arrayrthetaRct;
468 }
469
470 void modeloLinealGurobi(const char* nombre,int n,int m, double *D){
471
472
473     FILE *file;
474     file = fopen ( nombre, "w" );
475
476
477     fprintf(file,"Minimize\n");
478
479     for(int i=0;i<n;i++){
480       for(int j=0;j<m;j++){
481         fprintf(file,"%f x_%d_%d",D[i*m+j], i,j);
482         if(i==n-1 && j==m-1)
483           fprintf(file,"\n");
484         else
485           fprintf(file," + ");
486         }
487       fprintf(file,"\n");
488     }
489
490     int k=0;
491     fprintf(file,"Subject To\n");
492     for(int i=0;i<n;i++){
493       for(int j=0;j<m;j++){
494         fprintf(file,"x_%d_%d", i,j);
495         if( j==m-1)
496           fprintf(file," >= 1");
497         else
498         fprintf(file," + ");
499       }
500       fprintf(file,"\n");
501     }
502
503     for(int j=0;j<m;j++){
```

```
504         for(int i=0;i<n;i++){
505           fprintf(file,"x_%d_%d", i,j);
506           if(i==n-1)
507             fprintf(file," >= 1");
508         else
509         fprintf(file," + ");
510         }
511       fprintf(file,"\n");
512     }
513
514     fprintf(file,"\n");
515     fprintf(file,"Binary\n");
516     for(int i=0;i<n;i++){
517       for(int j=0;j<m;j++){
518         fprintf(file,"x_%d_%d ",i,j);
519       }
520       fprintf(file,"\n");
521     }
522
523     fprintf(file,"\n");
524
525     fprintf(file,"End");
526     fclose ( file );
527 }
528
529 struct bipartita tomarMatrizBinariaSol(const char * nombre,int n,int
        m){
530
531     char *a= (char*)malloc(sizeof(char)*1000);
532       FILE *file;
533     file = fopen ( nombre, "r" );
534
535     int *binaryA = (int *)malloc(sizeof(int)*n);
536     int *binaryB = (int *)malloc(sizeof(int)*m);
537
538     for(int i=0;i<n;i++)
539     binaryA[i]=-1;
540
541     for(int i=0;i<m;i++)
542     binaryB[i]=-1;
543
544     int w;
545
546     int i=0;
547     int j=0;
548     while(fscanf(file, "%s",a)!=EOF){
549       if(a[0]=='x'){
550         if(j>=m){
551           j=0;
552           i++;
553         }
554         fscanf(file, "%d", &w );
555         if(w==1){
```

```
556         binaryA[i]=j;
557         binaryB[j]=i;
558         }
559         j++;
560       }
561     }
562     fclose (file);
563 
564 struct bipartita bip;
565 
566 //////////////quitamos vertices que se repiten//////////////////////
567 for(int i=0;i<n;i++){
568   if(binaryB[binaryA[i]]==i)
569   binaryB[binaryA[i]]=-1;
570 }
571 ////////////////////////////////////////
572   bip.A=binaryA;
573   bip.B=binaryB;
574 
575   return bip;
576 }
577 
578 void drawbipartitaVoronoiColores(const char *str,int n,int m,int s,
      int t,struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *
      optxy,int  *optyx){
579 
580   cv::Mat img(SIZE_WINDOW,SIZE_WINDOW, CV_8UC3, CV_RGB(255,255,255))
      ;
581   double rx=double(SIZE_WINDOW)/double(s);
582   double ry=double(SIZE_WINDOW)/double(t);
583 
584   Rect rect(0, 0, SIZE_WINDOW, SIZE_WINDOW); //preparamos las
      medidas que tendra el rectangulo para el diagrama de voronoi
585 
586   Subdiv2D subdiv(rect);   //creamos un objeto  Subdiv2D , el objeto
       nos proveer  todo lo nesesario para el diagrama de voronoi
587 
588   vector<Point2f> PointsxyA;
589 
590   for(int i=0;i<n;i++)
591   {
592     PointsxyA.push_back(Point2f(arrayxyA[i].x*rx,arrayxyA[i].y*ry));
593   }
594   ////////////////////////////////////////7
595   vector<Point2f> PointsxyB;
596 
597   for(int l=0;l<m;l++)
598   {
599     PointsxyB.push_back(Point2f(arrayxyB[l].x*rx,arrayxyB[l].y*ry));
600   }
601 
602   ////////////////////////////////////////////////7
603   /////insertamos los puntos del conjunto B a nuestro diagrama
```

```
604   for( vector<Point2f>::iterator it = PointsxyB.begin(); it !=
       PointsxyB.end(); it++)
605   {
606     subdiv.insert(Point(it->x,it->y));
607   }
608
609   /////preparamos las variables para obtener el diagrama de voronoi
610   vector<vector<Point2f> > facets;
611   vector<Point2f> centers;
612
613   subdiv.getVoronoiFacetList(vector<int>(), facets, centers);
614
615   vector<Point> ifacet;
616
617   for( size_t i = 0; i < facets.size(); i++ ){
618     ifacet.resize(facets[i].size());
619     for( size_t j = 0; j < facets[i].size(); j++ )///numero de
       aristas por poligono
620       ifacet[j] = facets[i][j];     ////traslado de un conjunto de
       puntos que representan un poligono
621
622     Scalar color;
623     color[0] = rand() & 255;
624     color[1] = rand() & 255;
625     color[2] = rand() & 255;
626
627     fillConvexPoly(img, ifacet, color, 8, 0);  //dibujar poligono
       convexo
628     polylines(img, ifacet, true, Scalar(), 1, CV_AA, 0);  ///dibujar
       lineas para un solo poligono
629
630   }
631
632   if(optxy!=NULL)
633   for(int i=0;i<n;i++){
634       if(optxy[i]!=-1) {
635         line(img,  Point(int(rx*arrayxyA[i].x),int(ry*arrayxyA[i].y)
       ), Point(int(rx*arrayxyB[optxy[i]].x),int(ry*arrayxyB[optxy[i]].y
       )), CV_RGB(255,0,0), 1, CV_AA);
636       }
637   }
638
639   if(optyx!=NULL)
640   for(int i=0;i<m;i++){
641       if(optyx[i]!=-1) {
642         line(img,  Point(int(rx*arrayxyB[i].x),int(ry*arrayxyB[i].y)
       ), Point(int(rx*arrayxyA[optyx[i]].x),int(ry*arrayxyA[optyx[i]].y
       )), CV_RGB(255,0,0), 1, CV_AA);
643       }
644   }
645
646   for( int i = 0; i < n; i++ ) /////numero de puntos
647   {
```

```
648       circle(img, PointsxyA[i], 4, CV_RGB(0,0,0), -1,CV_AA); ///puntos
          A
649
650     }
651
652     for( int i = 0; i < m; i++ ) /////numero de puntos
653     {
654       circle(img, centers[i], 4, CV_RGB(0,0,0),  1,CV_AA);  ///puntos
         B
655       circle(img, centers[i], 3, CV_RGB(255,255,255), -1,CV_AA);
656
657     }
658
659     vector<int> compression_params;
660     imwrite(str, img, compression_params);
661
662
663 }
664
665 void drawbipartitaVoronoi(const char *str,int n,int m,int s,int t,
       struct cordrec *arrayxyA,struct cordrec *arrayxyB,int  *optxy,int
         *optyx){
666
667     cv::Mat img(SIZE_WINDOW,SIZE_WINDOW, CV_8UC3, CV_RGB(255,255,255))
        ;
668     double rx=double(SIZE_WINDOW)/double(s);
669     double ry=double(SIZE_WINDOW)/double(t);
670
671     Rect rect(0, 0, SIZE_WINDOW, SIZE_WINDOW); //preparamos las
        medidas que tendra el rectangulo para el diagrama de voronoi
672     Rect rect_2(0, 0, SIZE_WINDOW, SIZE_WINDOW); //preparamos las
        medidas que tendra el rectangulo para el diagrama de voronoi
673
674     Subdiv2D subdiv(rect);   //creamos un objeto Subdiv2D , el objeto
        nos proveer  todo lo nesesario para el diagrama de voronoi
675     Subdiv2D subdiv_2(rect_2);
676 ////////////////////////////////////////////
677     vector<Point2f> PointsxyA;
678
679     for(int i=0;i<n;i++)
680     {
681       PointsxyA.push_back(Point2f(arrayxyA[i].x*rx,arrayxyA[i].y*ry));
682     }
683     ////////////////////////////////////////////7
684     vector<Point2f> PointsxyB;
685
686
687     for(int l=0;l<m;l++){
688       PointsxyB.push_back(Point2f(arrayxyB[l].x*rx,arrayxyB[l].y*ry));
689     }
690
691     ////////////////////////////////////////////////////7
692     /////insertamos los puntos del conjunto B a nuestro diagrama
```

```
693
694   for( vector<Point2f>::iterator it = PointsxyB.begin(); it !=
       PointsxyB.end(); it++){
695     subdiv.insert(Point(it->x,it->y));
696   }
697
698   for( vector<Point2f>::iterator it = PointsxyA.begin(); it !=
       PointsxyA.end(); it++){
699     subdiv_2.insert(Point(it->x,it->y));
700   }
701
702   /////preparamos las variables para obtener el diagrama de voronoi
703   vector<vector<Point2f> > facets;
704   vector<Point2f> centers;
705
706   vector<vector<Point2f> > facets_2;
707   vector<Point2f> centers_2;
708
709   subdiv.getVoronoiFacetList(vector<int>(), facets, centers); ///
       tomamos los poligonos del diagrama
710   subdiv_2.getVoronoiFacetList(vector<int>(), facets_2, centers_2);
       ///tomamos los poligonos del diagrama
711
712   vector<Point> ifacet;
713   vector<Point> ifacet_2;
714
715
716   for( size_t i = 0; i < facets_2.size(); i++ ){
717     ifacet_2.resize(facets_2[i].size());
718     for( size_t j = 0; j < facets_2[i].size(); j++ )///numero de
       aristas por poligono
719       ifacet_2[j] = facets_2[i][j];     ////traslado de un conjunto
       de puntos que representan un poligono
720
721     Scalar color_2;
722     color_2[0] =  0;
723     color_2[1] = 0;
724     color_2[2] =  0;
725     //fillConvexPoly(img, ifacet, color, 8, 0);  //dibujar poligono
       convexo
726     ////////////////////////////////////////////////
727
728     polylines(img, ifacet_2, true,color_2, 1, CV_AA, 0);  ///dibujar
       lineas para un solo poligono
729
730   }
731
732
733   for( size_t i = 0; i < facets.size(); i++ ) /////numero de
       poligonos
734   {
735     ifacet.resize(facets[i].size());
```

```
736     for( size_t j = 0; j < facets[i].size(); j++ )///numero de
        aristas por poligono
737       ifacet[j] = facets[i][j];     ////traslado de un conjunto de
        puntos que representan un poligono
738
739     Scalar color;
740     color[0] = 255;
741     color[1] = 0;
742     color[2] = 0;
743
744     //fillConvexPoly(img, ifacet, color, 8, 0);  //dibujar poligono
        convexo
745
746     polylines(img, ifacet, true, color, 1, CV_AA, 0);  ///dibujar
        lineas para un solo poligono
747
748   }
749
750   if(optxy!=NULL)
751   for(int i=0;i<n;i++){
752       if(optxy[i]!=-1) {
753         line(img,  Point(int(rx*arrayxyA[i].x),int(ry*arrayxyA[i].y)
        ), Point(int(rx*arrayxyB[optxy[i]].x),int(ry*arrayxyB[optxy[i]].y
        )), CV_RGB(255,0,0), 1, CV_AA);
754       }
755   }
756
757   if(optyx!=NULL)
758   for(int i=0;i<m;i++){
759       if(optyx[i]!=-1) {
760         line(img,  Point(int(rx*arrayxyB[i].x),int(ry*arrayxyB[i].y)
        ), Point(int(rx*arrayxyA[optyx[i]].x),int(ry*arrayxyA[optyx[i]].y
        )), CV_RGB(255,0,0), 1, CV_AA);
761       }
762   }
763
764   for( int i = 0; i < n; i++ ) /////numero de puntos
765   {
766     circle(img, PointsxyA[i], 4, CV_RGB(0,0,0), -1,CV_AA); ///puntos
        A
767
768   }
769
770   for( int i = 0; i < m; i++ ) /////numero de puntos
771   {
772     circle(img, centers[i], 4, CV_RGB(0,0,0),  1,CV_AA);  ///puntos
        B
773     circle(img, centers[i], 3, CV_RGB(255,255,255), -1,CV_AA);
774
775   }
776
777   vector<int> compression_params;
778   imwrite(str, img, compression_params);
```

```
779
780 }
781
782 void drawbipartirta(const char *str,int n,int m,int s,int t,struct
        cordrec * cordA,struct cordrec * cordB,int *optxy,int *optyx ){
783
784   double rx=double(SIZE_WINDOW)/double(s);
785   double ry=double(SIZE_WINDOW)/double(t);
786
787   cv::Mat img(SIZE_WINDOW,SIZE_WINDOW, CV_8UC3, CV_RGB(255,255,255))
        ;
788
789
790
791   if(optxy!=NULL)
792   for(int i=0;i<n;i++){
793       if(optxy[i]!=-1) {
794         line(img, Point(int(rx*cordA[i].x),int(ry*cordA[i].y)),
        Point(int(rx*cordB[optxy[i]].x),int(ry*cordB[optxy[i]].y)),
        CV_RGB(255,0,0), 1, CV_AA);
795       }
796   }
797
798   if(optyx!=NULL)
799   for(int i=0;i<m;i++){
800       if(optyx[i]!=-1) {
801         line(img, Point(int(rx*cordB[i].x),int(ry*cordB[i].y)),
        Point(int(rx*cordA[optyx[i]].x),int(ry*cordA[optyx[i]].y)),
        CV_RGB(255,0,0), 1, CV_AA);
802       }
803   }
804
805   for(int i=0;i<n;i++){
806   circle(img, Point(int(rx*cordA[i].x), int(ry*cordA[i].y)), 4,
        CV_RGB(0,0,0), -1,CV_AA);
807
808
809   }
810
811   for(int i=0;i<m;i++){
812     circle(img, Point(int(rx*cordB[i].x), int(ry*cordB[i].y)), 4,
        CV_RGB(0,0,0), 1,CV_AA);
813     circle(img, Point(int(rx*cordB[i].x), int(ry*cordB[i].y)), 3,
        CV_RGB(255,255,255), -1,CV_AA);
814   }
815
816   vector<int> compression_params;
817   imwrite(str, img, compression_params);
818
819 }
820
821
```

```
822 void drawbipartitaPolar(const char *str,int n,int m,int s,struct
      cordpolar * cordA,struct cordpolar * cordB,int *optrt,int *opttr
      ){
823
824   double rz=double(SIZE_WINDOW)/((2.0)*s);
825   cv::Mat img(SIZE_WINDOW, SIZE_WINDOW, CV_8UC3, CV_RGB(255,255,255)
      );
826
827   if(optrt!=NULL)
828   for(int i=0;i<n;i++){
829       if(optrt[i]!=-1) {
830         line(img,  Point(int(rz*(cordA[i].r)*cos(cordA[i].theta)+
      SIZE_WINDOW/2),int(rz*(cordA[i].r)*sin(cordA[i].theta)+
      SIZE_WINDOW/2)), Point(int(rz*(cordB[optrt[i]].r)*cos(cordB[optrt
      [i]].theta)+SIZE_WINDOW/2),int(rz*(cordB[optrt[i]].r)*sin(cordB[
      optrt[i]].theta))+SIZE_WINDOW/2), CV_RGB(255,0,0), 1, CV_AA);
831       }
832   }
833
834   if(opttr!=NULL)
835   for(int i=0;i<m;i++){
836       if(opttr[i]!=-1) {
837         line(img,  Point(int(rz*(cordB[i].r)*cos(cordB[i].theta)+
      SIZE_WINDOW/2),int(rz*(cordB[i].r)*sin(cordB[i].theta)+
      SIZE_WINDOW/2)), Point(int(rz*(cordA[opttr[i]].r)*cos(cordA[opttr
      [i]].theta)+SIZE_WINDOW/2),int(rz*(cordA[opttr[i]].r)*sin(cordA[
      opttr[i]].theta))+SIZE_WINDOW/2), CV_RGB(255,0,0), 1, CV_AA);
838       }
839   }
840
841   for(int i=0;i<n;i++){
842     circle(img, Point(int(rz*(cordA[i].r)*cos(cordA[i].theta)+
      SIZE_WINDOW/2),int(rz*(cordA[i].r)*sin(cordA[i].theta)+
      SIZE_WINDOW/2)), 4, CV_RGB(0,0,0),-1,CV_AA);
843
844
845   }
846
847   for(int i=0;i<m;i++){
848     circle(img, Point(int(rz*(cordB[i].r)*cos(cordB[i].theta)+
      SIZE_WINDOW/2),int(rz*(cordB[i].r)*sin(cordB[i].theta)+
      SIZE_WINDOW/2)), 4, CV_RGB(0,0,0),1,CV_AA);
849     circle(img, Point(int(rz*(cordB[i].r)*cos(cordB[i].theta)+
      SIZE_WINDOW/2),int(rz*(cordB[i].r)*sin(cordB[i].theta)+
      SIZE_WINDOW/2)), 3, CV_RGB(255,255,255),-1,CV_AA);
850   }
851
852
853   vector<int> compression_params;
854   imwrite(str, img, compression_params);
855
856
857 }
```

```
858
859 void printMatrizDistancias(int n,int m,double *D){
860
861   for(int i=0;i<n;i++){
862     for(int j=0;j<m;j++){
863       printf("%.2f ",D[i*m+j]);
864     }
865     printf("\n");
866   }
867 }
868
869 int *  bipartitaCercano(int n,int m,struct cordrec *arrayxyA,struct
      cordrec *arrayxyB){
870
871   Rect rect(0, 0, SIZE_WINDOW, SIZE_WINDOW); //preparamos las
       medidas que tendra el rectangulo para el diagrama de voronoi
872   Subdiv2D subdiv(rect);   //creamos una instancia a  Subdiv2D , el
       objeto nos proveer  todo lo nesesario para el diagrama de
       voronoi
873   vector<Point2f> PointsxyA;
874
875   for(int i=0;i<n;i++)
876   {
877     PointsxyA.push_back(Point2f(arrayxyA[i].x,arrayxyA[i].y));
878   }
879
880   vector<Point2f> PointsxyB;
881
882   for(int i=0;i<m;i++)
883   {
884     PointsxyB.push_back(Point2f(arrayxyB[i].x,arrayxyB[i].y));
885   }
886
887   /////insertamos los puntos del conjunto B a nuestro diagrama
888
889   for( vector<Point2f>::iterator it = PointsxyB.begin(); it !=
       PointsxyB.end(); it++)
890   {
891     subdiv.insert(Point2f(it->x,it->y));
892   }
893
894
895   int *birary= (int *)malloc(sizeof(int)*n); ///prepararemos nuestro
       arreglo Z
896
897   int ver;
898
899   for( int i= 0; i<n; i++)///dado un punto de A, este nos dara un
       pundo de B que se encuentre mas cercano
900   {
901     ver=subdiv.findNearest( PointsxyA[i],NULL);
902     birary[i]=ver-4;
903   }
```

```
904
905   return birary;
906 }
907
908 int *  bipartitaCercanoIzquierda(int n,int m,struct cordrec *
      arrayxyA,struct cordrec *arrayxyB){
909
910   return bipartitaCercano(n,m,arrayxyA,arrayxyB);
911
912 }
913
914
915 int *  bipartitaCercanoDerecha(int n,int m,struct cordrec *arrayxyA,
      struct cordrec *arrayxyB){
916
917   return bipartitaCercano(m,n,arrayxyB,arrayxyA);
918
919
920 }
921
922
923  struct bipartita heuristicaDosVeces(int n,int m,struct cordrec *
      arrayxyA,struct cordrec *arrayxyB){
924
925   int * bip_1= bipartitaCercano(n,m,arrayxyA,arrayxyB);
926   int * bip_2= bipartitaCercano(m,n,arrayxyB,arrayxyA);
927
928
929   for(int i=0;i<n;i++){
930     if(bip_2[bip_1[i]]==i)
931     bip_2[bip_1[i]]=-1;
932   }
933
934   struct bipartita bip;
935
936   bip.A=bip_1;
937   bip.B=bip_2;
938
939   return bip;
940
941 }
942
943 struct bipartita  heuristicaDerecha(int n,int m,struct cordrec *
      arrayxyA,struct cordrec *arrayxyB){
944
945   int * bip_1 = bipartitaCercano(n,m,arrayxyA,arrayxyB);
946   int * bip_2 = bipartitaCercano(m,n,arrayxyB,arrayxyA);
947
948   for(int i=0;i<m;i++){
949     bip_1[bip_2[i]]=-1;
950   }
951
952   struct bipartita bip;
```

```
953
954   bip.A=bip_1;
955   bip.B=bip_2;
956
957     return bip;
958   }
959
960 struct bipartita heuristicaIzquierda(int n,int m,struct cordrec *
      arrayxyA,struct cordrec *arrayxyB){
961   int * bip_1 = bipartitaCercano(n,m,arrayxyA,arrayxyB);
962   int * bip_2 = bipartitaCercano(m,n,arrayxyB,arrayxyA);
963
964   for(int i=0;i<n;i++){
965     bip_2[bip_1[i]]=-1;
966   }
967
968   struct bipartita bip;
969
970   bip.A=bip_1;
971   bip.B=bip_2;
972
973     return bip;
974   }
```

Referencias

[1] Saúl Martínez Juarez, *"Algoritmo y heurística para incrustar métricas en una linea"*, proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2017.

[2] José Daniel Faustinos Vargas, *"Identificación de una configuración en un conjunto de puntos en el plano"*, proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2017.

[3] Zelzin Marcela Márquez Navarrete, *"Heurísticas para acoplamientos euclidianos sin cruces"*, proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2017.

[4] J. Akiyama, Jorge Urrutia *"Simple alternating path problem"*, Article, Department of Mathematics, Tokai University, Hiratsuka, Japan, 2 February 1988.

[5] S.Cabello, J.M.Díaz-Báñez, P.Pérez-Lantero *"Covering a bichromatic point set with two disjoint monochromatic disks"*, Computational Geometry 46 (2013) 203–212.

[6] S. Bereg, J. M. Díaz-Báñez, R. Fabila-Monroy, P. Pérez-Lantero, A. Ramírez-Vigueras, T. Sakai, J. Urrutia, I. Ventura *"On balanced 4-holes in bichromatic point sets'*, Computational Geometry 48 (2015) 169–179.