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

Heurísticas para acoplamientos euclidianos sin cruces

Proyecto de investigación

Zelzin Marcela Márquez Navarrete
al2133033728@azc.uam.mx
2133033728

Asesores

Asesor
Dr. Francisco Javier
Zaragoza Martínez
Profesor Titular
Departamento de Sistemas
franz@azc.uam.mx

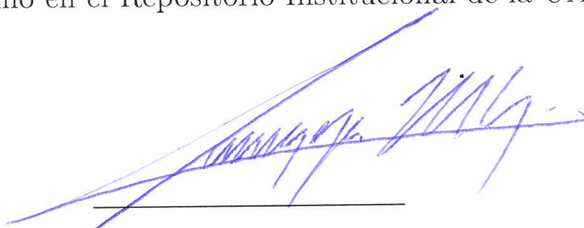
Coasesor
Dr. Marco Antonio
Heredia Velasco
Profesor Asociado
Departamento de Sistemas
hvma@azc.uam.mx

Trimestre 2017 Invierno
25 de abril de 2017

Yo, Zelzin Marcela Márquez Navarrete, doy mi autorización a la Coordinación de servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para su publicar el presentedicumento en la Biblioteca Digital, así como en el Repositorio Institucional de la UAM Azcapotzalco.

Zelzin Marcela Márquez Navarrete

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.



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.

Dr. Marco Antonio Heredia Velasco

Resumen

Dado un conjunto S de $n = 2k$ puntos en posición general (no hay tres puntos colineales) en un plano euclidiano, calcular acoplamientos perfectos de S de costo máximo.
El problema de calcular acoplamientos perfectos de costo máximo sin cruces es un problema que se cree que es NP-Duro, para solucionarlo, en este trabajo se propuso el uso de heurísticas a las que se les nombraron: veleta y cerradura convexa.
El costo máximo del acoplamiento fue evaluado con las heurísticas mencionadas anteriormente y comparado contra el valor óptimo para determinar la efectividad de cada heurística. En el peor de los casos, para cada una de las heurísticas se observa una aproximación al valor óptimo de ≈ 0.5330 para la heurística de la veleta y de ≈ 0.7298 para la heurística de la cerradura convexa.

Tabla de contenidos

1. Introducción **1**

2. Justificación **2**

3. Antecedentes **2**
- 3.1. Externos . . . 2
- 3.2. Internos . . . 4

4. Objetivos **4**
- 4.1. Objetivo general . . . 4
- 4.2. Objetivos específicos . . . 4

5. Marco teórico **5**

6. Desarrollo del proyecto **6**
- 6.1. Heurística de la veleta . . . 7
- 6.2. Heurística de cerradura convexa . . . 9
- 6.3. Algoritmo exacto . . . 11
  - 6.3.1. Programa entero . . . 11
  - 6.3.2. Script para Gurobi . . . 12
- 6.4. Evaluación . . . 12
- 6.5. Especificación técnica . . . 13

7. Resultados **14**

8. Conclusión **15**

9. Apéndice **19**
- 9.1. Código fuente . . . 19
  - 9.1.1. main.cpp . . . 19
  - 9.1.2. Coordinate.h . . . 19
  - 9.1.3. Coordinate.cpp . . . 20
  - 9.1.4. Map.h . . . 20
  - 9.1.5. Map.cpp . . . 21
  - 9.1.6. Match.h . . . 23
  - 9.1.7. Match.cpp . . . 24
  - 9.1.8. ConvexHull.h . . . 26
  - 9.1.9. ConvexHull.cpp . . . 27

9.1.10. poligono.lp 28

1. Introducción

Una *gráfica geométrica* es una gráfica $G = (V, E)$ dibujada en el plano, tal que V es un conjunto de puntos en *posición general*, –lo que significa que no existen tres puntos de V que tengan una línea recta en común– y E es el conjunto de segmentos de línea recta cuyos extremos pertenecen a V. El *costo* de una gráfica geométrica es la suma de las longitudes de sus aristas. A una gráfica geométrica se le dice *sin cruces* si no existen dos aristas que intersequen en su interior, aún así, dos aristas pueden tener un extremo en común. Dos aristas son disjuntas si no tienen puntos en común [1]. Debido a la desigualdad del triángulo, una gráfica sin cruces tiene menor costo que una con cruces.
Un vértice v es *incidente* con una arista e si $v \in e$, por lo que e es una arista de v.
Un *acoplamiento*, es un conjunto de aristas disjuntas en una gráfica G tal que no haya dos aristas que compartan un vértice en común. Un *acoplamiento perfecto* de una gráfica es un acoplamiento en el cual cada vértice de la gráfica es incidente con exactamente un arista del acoplamiento.
Dado un conjunto S de $n = 2k$ vértices en posición general, estamos interesados en calcular acoplamientos perfectos de S de costo máximo, sin cruces, en la Figura 1a. En una gráfica geométrica, un acoplamiento perfecto de costo mínimo no tiene cruces, como se muestra en la Figura 1c, debido al teorema de la desigualdad triangular, sin embargo, en un acoplamiento perfecto de costo máximo es posible que aparezcan cruces, ejemplo de ello se muestra en la Figura 1b.

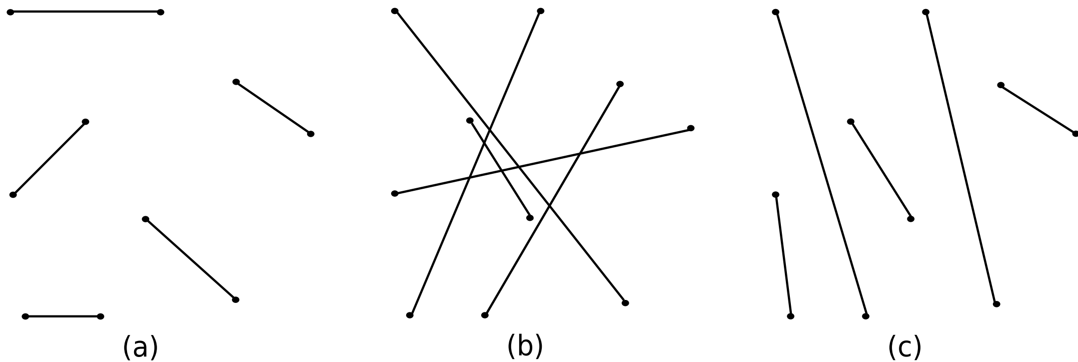


Figura 1: Acoplamientos geométricos perfectos: (a) de costo mínimo, (b) de costo máximo y (c) de costo máximo sin cruces.

Como se muestra en la Figura 1, en los acoplamientos perfectos de costo mínimo no hay cruces, no obstante, para el acoplamiento perfecto de costo máximo puede haber;

estamos interesados en el caso en el que no hay cruces. A la fecha, el problema de calcular acoplamientos perfectos de costo máximo sin cruces es un problema difícil de resolver y se cree que es NP-Duro [2]. Resolver estos problemas con un algoritmo exacto, como por ejemplo, uno de programación entera, sería muy lento. Debido a lo anterior, en este trabajo proponemos el uso de heurísticas para resolver el problema de maximización sin cruces. Los resultados obtenidos serán evaluados contra el valor óptimo para determinar la efectividad de cada heurística. Se utilizaron dos heurísticas:

- Heurística de la veleta.
- Heurística de la cerradura convexa.

La primera heurística ordena los vértices con respecto a cuatro direcciones distintas, mientras que la segunda hace uso del algoritmo de Andrew de cadena monótona para obtener la cerradura convexa [3]. Adicionalmente se tomará como referencia el valor óptimo que proporcione algoritmo exacto, para posteriormente evaluar contra ese valor el obtenido por las heurísticas.

2. Justificación

El estudio de los acoplamientos tiene una amplia variedad de aplicaciones, tal es su importancia que existen libros dedicados exclusivamente al tema [4]. Los acoplamientos de costo máximo tienen aplicación en el encaminamiento y conmutación en redes [5], cómputo numérico [6], algoritmos de gráficas multinivel, en bioinformática, para la comparación de estructuras de proteínas, procesamiento de imágenes, en la correspondencia de características de imágenes 2D [7], redes de sensores [8], algoritmos de planificación para conmutadores de entrada encolada [9], investigación de operaciones, factorización de matrices [10], específicamente para el caso de acoplamientos de costo máximo sin cruces tiene utilidad para la impresión de circuitos. La aportación principal de nuestro trabajo es la de proporcionar heurísticas que obtienen acoplamientos euclidianos perfectos de costo máximo sin cruces y la evaluación de su efectividad para resolver este problema.

3. Antecedentes

3.1. Externos

En el año 1965 fue publicado el algoritmo "*Blossom*" de Jack Edmonds en el artículo titulado "*Paths, trees, and flowers*" [11] para la construcción de acoplamientos perfectos

máximos en gráficas. Este algoritmo encuentra un acoplamiento M tal que cada vértice en V es incidente con al menos una arista en M y $|M|$ es máxima. El motivo de mayor importancia del algoritmo es que, en su artículo, Edmonds dio la primera prueba de que un acoplamiento máximo puede ser encontrado en tiempo de ejecución polinomial. Sin embargo, este algoritmo no toma en cuenta si el acoplamiento máximo geométrico tiene cruces, por lo que usar este algoritmo no sería útil para resolver nuestro problema. Para obtener acoplamientos perfectos sin cruces, Alon [2], en su artículo "*Long Non-crossing configurations in the plane*", describe el algoritmo de Hershberger y Suri en el que un conjunto de vértices se biseca mediante una línea recta y se construye un acoplamiento con un elemento de cada conjunto de tal manera que no se crucen, como se muestra en la Figura 2. Este algoritmo obtiene al menos la mitad de la distancia del acoplamiento perfecto de costo máximo (que puede contener cruces), este algoritmo es modificado de tal manera se que muestra que pueden encontrar emparejamientos perfectos sin cruces de al menos $2/\pi \approx 0.6366$, del costo máximo del acoplamiento perfecto máximo, probablemente con cruces.

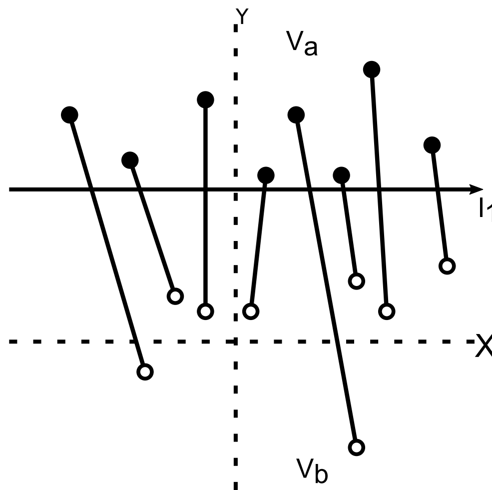


Figura 2: La línea l_1 biseca a V en dos conjuntos del mismo tamaño V_a y V_b; los puntos en V_a se muestran con círculos rellenos y los puntos en V_b con círculos vacíos. Se muestra un acoplamiento M_1 sin cruces entre V_a y V_b.

Posteriormente el trabajo de Alon fue retomado por Dumitrescu [12] con un artículo del mismo nombre, en él se revisa nuevamente cada uno de los problemas mencionados por Alon, incluyendo los acoplamientos perfectos de costo máximo, para el cual la razón de la aproximación fue también cercana a $2/\pi \approx 0.6366$.

El el artículo de Aichholzer [1] se realiza un algoritmo que permite obtener un emparejamiento perfecto sin cruces, a partir de una gráfica completa de la cual se remueve el máximo un número arbitrario de aristas de tal manera que aún exista un emparejamiento perfecto, sin embargo, para este algoritmo no se toma en cuenta la restricción de independencia de las aristas ni se da el costo máximo del emparejamieto.

3.2. Internos

Alfaro [13] en su proyecto terminal con título "*Encajes primitivos de árboles planos*", implementó cuatro algoritmos, tres para encajes primitivos grandes y otro para encontrar el encaje primitivo más pequeño para un árbol plano -sin cruces-, lo que significa que la rejilla formada por un conjunto n de puntos tiene el lado más pequeño posible.
Pérez [14] en su proyecto terminal con título "*Encajes primitivos de gráficas planares exteriores*", implementó dos algoritmos para el encaje primitivo para gráficas planares exteriores, en el cual se coloca una gráfica planar -sin cruces- en los n puntos de una rejilla, para con ellos generar un polígono convexo y otro no necesariamente convexo.
En el trabajo de Vázquez [15], con título "*A triplet integer programming model for the Euclidean 3-matching problem*", egresado de la UAM-A, se proponen las heurísticas implementadas en este proyecto para el problema de 3-acoplamientos perfectos óptimos, por lo que se realizaron modificaciones a cada uno de ellos de forma que funcionaran para sólo dos vértices.

4. Objetivos

4.1. Objetivo general

- Modificar, implementar y evaluar heurísticas para el problema de acoplamientos perfectos euclidianos máximos sin cruces.

4.2. Objetivos específicos

- Modificar e implementar la heurística de la veleta para acoplamientos perfectos.
- Modificar e implementar la heurística de cerradura convexa para acoplamientos perfectos.
- Evaluar la eficacia de cada una de las heurísticas implementadas comparando el resultado contra el valor óptimo.

5. Marco teórico

Una *gráfica* G es un par ordenado (V, E) de conjuntos tales que V es un conjunto de vértices y $E \subseteq \binom{V}{2}$ es un conjunto de aristas, así, cada uno de los elementos de E son pares de elementos de V. Asumimos que $V \cap E = \emptyset$ [16].
Las gráficas se suelen representar dibujando un punto por cada vértice y una línea que une a dos vértices por cada arista.
Dado un conjunto S de n puntos en el plano, decimos que está en *posición general* si no contiene tres puntos colineales.
Una *gráfica geométrica*, es una gráfica donde sus vértices son puntos en el plano en posición general y sus aristas son segmentos de recta. [17].
Dos aristas se *cruzan* si tienen un punto interior en común, éste es llamado *cruce*, una representación de gráfica sin cruces se muestra en la Figura 3, en la Figura 4 se muestra una con cruces.

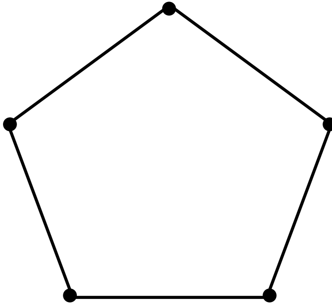


Figura 3: Gráfica geométrica sin cruces.

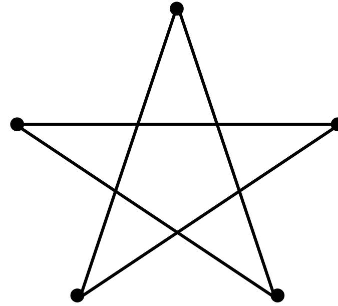


Figura 4: Gráfica geométrica con cruces.

Un cruce p es llamado *propio* si en una pequeña vecindad de p una arista atraviesa otra arista [17].
Si $V' \subseteq V$ y $E' \subseteq E$, entonces G' es una *subgráfica* de G (y G una *supergráfica* de G') denotada como $G' \subseteq G$ [18].
Un vértice v es *incidente* con una arista e si $v \in e$, por lo que e es una arista de v. Dos aristas son *independientes* si no tienen un vértice en común. A un conjunto M de aristas independientes en una gráfica $G = (V, E)$ se le llama *acoplamiento*. A los vértices de U se les denomina *acoplados* por M, los vértices no incidentes con alguna arista de M se dicen *no acoplados*. Si M es un acoplamiento de una gráfica geométrica, entonces definimos su *costo* como la suma de las longitudes de sus aristas.
Un acoplamiento es *perfecto* cuando cada vértice de la gráfica es incidente con exactamente una arista del acoplamiento, lo cual significa que un acoplamiento perfecto solo

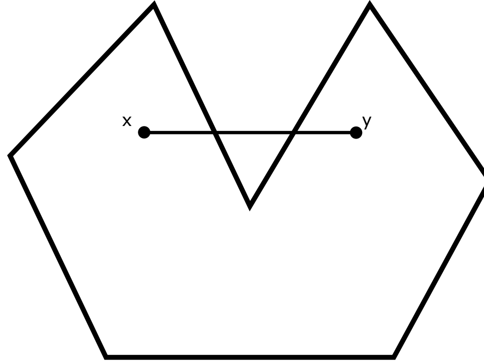


Figura 5: Si existe alguna abolladura, no es un cierre convexo.

es posible con un número par de vértices.
Un conjunto S es *convexo* si $x \in S$ y $y \in S$ implica que el segmento $xy \subseteq S$. En la Figura 5 se observa que si en alguna región hay una “abolladura” entonces no es convexo, ya que se pueden hallar dos puntos en S tales que forman un segmento de recta que contiene puntos exteriores a la región. Por lo que cualquier polígono con vértices cóncavos es no convexo.
Una *combinación convexa* de puntos $x_1, \ldots, x_k$ en un espacio vectorial real, es un punto de la forma $\alpha_1 x_1 + \ldots + \alpha_k x_k$, con $\alpha_i \geq 0$ para toda i y $\alpha_1 + \ldots + \alpha_k = 1$. Por consiguiente, un segmento de línea consiste de todas las combinaciones convexas de sus extremos, por ejemplo, un triángulo consiste de todas las combinaciones convexas de tres puntos.
Un *cierre convexo* de un conjunto S de puntos es el conjunto de todas las combinaciones convexas de puntos de S. Al cierre convexo de S le denotaremos como $H(S)$ [19].

6. Desarrollo del proyecto

Para llevar a cabo la evaluación de las heurísticas, que dado un conjunto de S puntos en posición general, obtengan el acoplamiento perfecto de costo máximo sin cruces, fue necesario obtener también el valor óptimo. En la Figura 6 se muestra el orden de dependencia en que se realizaron de los cálculos y se obtuvieron los resultados.
En lo posterior, se referirá a S, la cual será un conjunto de $n = 2k$ vértices en el plano en posición general.

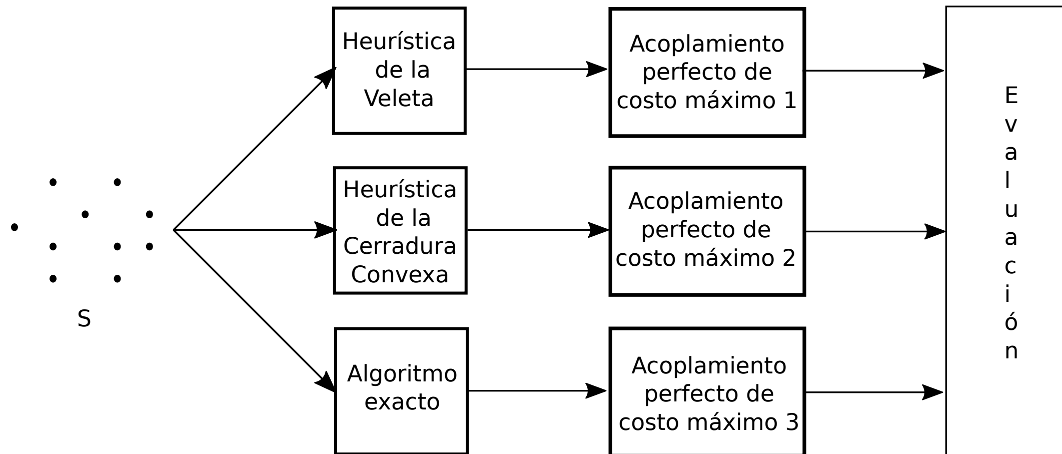


Figura 6: Los tres programas recibirán un conjunto S de puntos, del cual se obtendrán acoplamientos perfectos de costo máximo que posteriormente serán evaluadas las dos heurísticas con el valor óptimo obtenido del algoritmo exacto.

A continuación se detalla el procedimiento realizado por cada una de las heurísticas y el algoritmo exacto.

6.1. Heurística de la veleta

En esta heurística, se toma un conjunto S de puntos y se realizan cuatro ordenamientos distintos con el objetivo de seleccionar el acoplamiento de mayor costo. Primeramente se ordenan los vértices de S en dirección horizontal y se toman de dos en dos en ese orden, se realizar el mismo procedimiento para otras tres direcciones, la segunda en posición vertical, la tercera a $45°$ con respecto a la horizontal y la cuarta a $-45°$ con respecto a la horizontal, como se muestra en la Figura 7. Una vez obtenidos los cuatro acoplamientos perfectos, elegir aquella con el costo máximo.

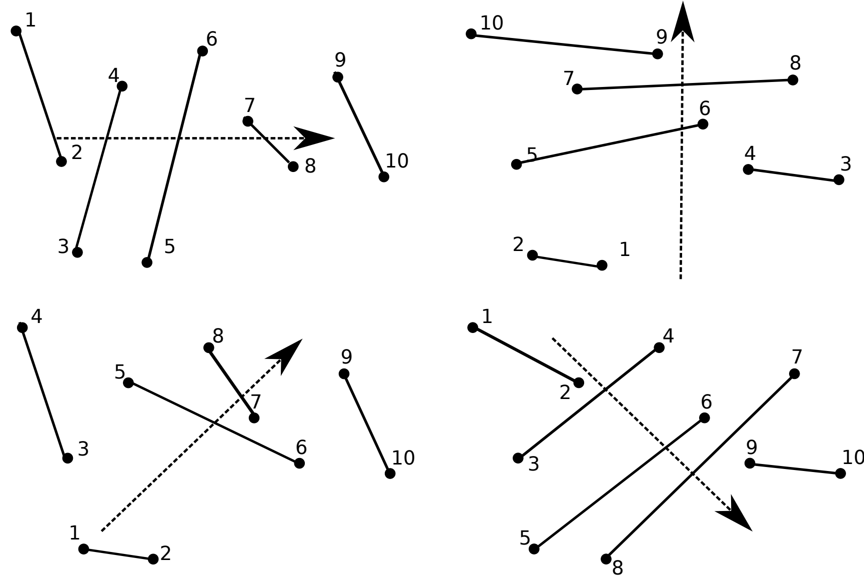


Figura 7: Se ordenan un conjunto S de puntos respecto a cuatro direcciones y se obtiene un apareamiento perfecto máximo sin cruces para cada una de ellas.

El pseudocódigo para realizar la heurística se muestra a continuación.

Para cada una de las cuatro direcciones explicadas con anterioridad, se realizan tres pasos:

- Se llama a una función que ordene todos los puntos de S en la dirección que le corresponde, en el pseudocódigo se realiza primero el ordenamiento en x y se asigna a S_X, que contiene los vértices de S en el orden que corresponda.
- Se llama a una función que realiza los acoplamientos tomando los vértices de un conjuto de dos en dos, en el primer caso se le pasa como argumento S_X y se almacenan todos los acoplamientos obtenidos en $match_X$.
- Finalmente se llama a una función que obtenga el costo total de un conjunto de aristas, para el primer caso, se pasa $match_X$ como argumento y el costo obtenido se almacena en $distance_X$.

Heurística 1 Cálculo de costo máximo de S con la heurística de la veleta

```
1: procedure VELETA(S)
2:     S_X ← SORTX(S)
3:     match_X ← MAKEMATCH(S_X)
4:     distance_X ← TOTALDISTANCE(match_X)
5:     S_Y ← SORTY(S)
6:     match_Y ← MAKEMATCH(S_Y)
7:     distance_Y ←TOTALDISTANCE(match_Y)
8:     S_XY ← SORTXY(S)
9:     match_XY ← MAKEMATCH(S_XY)
10:    distance_XY ← TOTALDISTANCE(match_XY)
11:    S_YX ← SORTYX(S)
12:    match_YX ← MAKEMATCH(S_YX)
13:    distance_YX ← TOTALDISTANCE(match_YX)
14:    maxDistance ← MAX(distance_X, distance_Y, distance_XY, distance_YX)
15:    return maxDistance
16: end procedure
```

Una vez que se realiza lo mismo para las cuatro direcciones:
X, en dirección del eje horizontal, Y, en dirección del eje vertical, XY a 45º del eje X y YX a 45º del eje Y, se llama a una función que obtiene la distancia máxima de las cuatro obtenidas con anterioridad y la almacena en *maxDistance* para ser devuelta por la heurística.

6.2. Heurística de cerradura convexa

Calcular la cerradura convexa de S, tomar la arista de la cerradura convexa de longitud máxima, quitar los dos vértices correspondientes, volver a calcular la cerradura convexa y repetir el procediemiento hasta que quede solo quede una arista, un ejemplo de ello se muestra en la Figura 8.
A continuación se encuentra el pseudicódigo correspondiente a esta heurística.

Esta heurística revisa todos los acoplamientos de S, por lo que comienza en 0 y termina en $|S|/2$.

- Se hace llamar a una función que permita crear un cierre convexo a partir de un conjunto S y se almacena en un subconjunto de S que aquí se llamó *convexHull*. En este proyecto se implementó el algoritmo de Andrew de cadena monótona, pero se puede usar cualquier algoritmo que cree un cierre convexo.

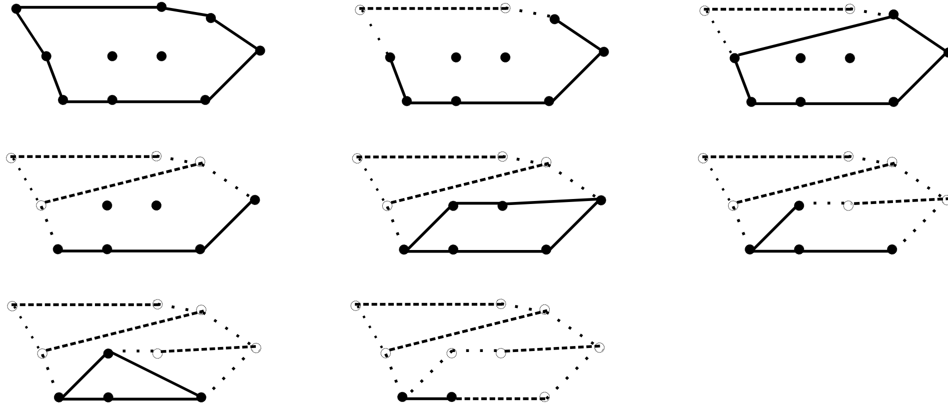


Figura 8: Se calcula la cerradura convexa de un conjunto S de puntos, se eliminan los vértices de la arista de mayor longitud y se vuelve a calcular la cerradura convexa.

Heurística 2 Cálculo de costo máximo de S con la heurística de la cerradura convexa

---

1: **procedure** CERRADURA CONVEXA(x)
2:     **for** $i \leftarrow 0, |S|/2$ **do**
3:         $convexHull \leftarrow$ MAKECONVEXHULL(S)
4:         $convexHullMatches \leftarrow$ MAKEMATCH($convexHull$)
5:         $maxMatch \leftarrow$ MAXMATCH($convexHullMatches$)
6:         $maxDistance \leftarrow maxDistance+$DISTANCE($maxMatch$)
7:         $S \leftarrow S$ - $point_a(maxMatch)$
8:         $S \leftarrow S$ - $point_b(maxMatch)$
9:     **end for**
10:     **return** $maxDistance$
11: **end procedure**

- Se llama a una función que realiza los acoplamientos tomando los vértices de un conjuto de dos en dos, en este caso se le pasa como argumento $convexHull$ y se asigna a $convexHullMatches$.
- Se llama a una función que busca el acoplamiento de mayor longitud y el resultado se le asigna a $maxMatch$.
- Se llama a una función que obtenga el costo de un emparejamiento, en este caso se le pasa como parámetro $maxMatch$ y se va acumulando en $maxDistance$ con cada iteración.
- Se eliminan los puntos a y b que corresponden a $maxMatch$, del conjunto S.
- Al finalizar todas las iteraciónes se devuelve el valor de $maxDistance$.

6.3. Algoritmo exacto

Se obtiene el valor óptimo del acoplamiento perfecto de costo máximo, resolviendo el programa entero correspondiente, para ello se hizo uso del software Gurobi.

6.3.1. Programa entero

A continuación se muestra el programa entero que debe ser resuelto para obtener el valor óptimo del costo de un acoplamiento perfecto máximos sin cruces.

$$\begin{array}{lll} \text{maximiza} & \sum_{i=1,j=1}^{n} c_{ij}x_{ij} & \\ \text{sujeto a} & \sum_{i=1}^{n}\sum_{j=1}^{m} \; x_{ij} = 1, & i = 1, ..., n \\ & \qquad x_{ik} + x_{jh} \leq 1, & j = 1, ..., m \\ & \qquad x_{ij} \in \{0, 1\} & \end{array}$$

Las variables son del tipo x_{ij}, donde x_{ij} es una variable que vale 1 si se escoge la arista ij y 0 si no se escoge. Hay dos tipos de restricciones, la primera, $x_{12}+x_{13}+x_{14}+\ldots+x_{1n} = 1$, que mantiene las aristas independientes (donde $|S| = n$ es la cantidad de vértices) y la segunda, $x_{ik} + x_{jh} \leq 1$, cada vez que h, i, j, k es un cuadrilátero convexo, que limita que un par de aristas tengan cruces.

6.3.2. Script para Gurobi

A Gurobi se le debe pasar un archivo de texto .lp con un formato como el que se muestra en la Figura 9. Para ello es necesario realizar un programa que pueda decir cuáles son todas las adyacencias existentes del conjunto S y todos lo cruces posibles, para que estos sean escritos como los subíndices correspondientes en las restricciones. En el apéndice se puede observar otro ejemplo del texto completo que se debe pasar a Gurobi para resolver un hexágono con posiciones en el plano: 2.9, 0.2, 5.9, 0.7, 7.0, 3.5, 5.1, 5.8, 2.1, 5.3, 1.0, 2.5.

```
1  Maximize
2   3.04138 edge_0_1 +  6.01664 edge_0_2 +  5.16236 edge_0_3 +  5.16236 edge_1_2 +  5.96657 edge_1_3 +  3.04138 edge_2_3
3  Subject To
4   edge_0_1  +  edge_0_2  +  edge_0_3  = 1
5   edge_0_1  +  edge_1_2  +  edge_1_3  = 1
6   edge_0_2  +  edge_1_2  +  edge_2_3  = 1
7   edge_0_3  +  edge_1_3  +  edge_2_3  = 1
8   edge_0_2 + edge_1_3 <= 1
9   edge_1_3 + edge_0_2 <= 1
10 Binary
11  edge_0_1
12  edge_0_2
13  edge_0_3
14  edge_1_2
15  edge_1_3
16  edge_2_3
17 End
```

Figura 9: Ejemplo sencillo del archivo que debe ser leído por Gurobi para obtener la solución óptima.

6.4. Evaluación

Se realizó una comparación de los valores obtenidos por las heurísticas contra el valor óptimo mediante una razón, de la siguiente forma:
$\frac{veleta}{óptimo} \leq 1$ y $\frac{cerradura}{óptimo} \leq 1$.
El programa realizado obtiene el valor óptimo de cada una de las heurísticas como se muestra en la Figura 10.

```
tsuki@KURO: /mnt/c/Users/ammy_/Desktop/Matchings
tsuki@KURO:/mnt/c/Users/ammy_/Desktop/Matchings$ ls
benchmark.tar.gz  Makefile    nbproject                  x01.rnd     x04f.rnd  x07.rnd   x11f.rnd  x14.rnd   x18f.rnd
ConvexHull.cpp    Map.cpp     poligono.rnd               x01.rnd.ch  x04.rnd   x08f.rnd  x11.rnd   x15f.rnd  x18.rnd
ConvexHull.h      Map.h       Proyecto de Integracion    x01.rnd.wr  x05f.rnd  x08.rnd   x12f.rnd  x15.rnd   x19f.rnd
Coordinate.cpp    Match.cpp   readme.txt                 x02f.rnd    x05.rnd   x09f.rnd  x12.rnd   x16f.rnd  x19.rnd
Coordinate.h      Match.h     result.dat                 x02.rnd     x06f.rnd  x09.rnd   x13f.rnd  x16.rnd   x20f.rnd
dist              matchings   run.sh                     x03f.rnd    x06.rnd   x10f.rnd  x13.rnd   x17f.rnd  x20.rnd
main.cpp          Matchings   x01f.rnd                   x03.rnd     x07f.rnd  x10.rnd   x14f.rnd  x17.rnd
tsuki@KURO:/mnt/c/Users/ammy_/Desktop/Matchings$ g++ *.cpp -o matchings -I. -std=c++11 -Wall -g
tsuki@KURO:/mnt/c/Users/ammy_/Desktop/Matchings$ ./matchings x01.rnd
x01.rnd 4890.71 8259.72
tsuki@KURO:/mnt/c/Users/ammy_/Desktop/Matchings$
```

Figura 10: Al ejecutar el programa se obtienen lo dos costos máximos del acoplamiento, a la izquiera el que corresponde a la heurística de la veleta y a la derecha el del cierre convexo.

6.5. Especificación técnica

Las heurísticas fueron implementadas en el lenguaje C++11, los archivos con puntos de prueba se encuentran en formato de texto, actualmente se cuenta con un banco de 40 archivos que contienen pares de números enteros y flotantes, dependiendo de la instancia.
Los archivos de prueba con los que se planeaba evaluar contenían originalmente $n = 900$ puntos, sin embargo, por el tiempo que tardaba en resolverse el algoritmo óptimo, se redujo a $n = 30$.
El valor óptimo se obtuvo mediante el el uso de Gurobi, el cual se obtuvo de forma gratuita desde el sitio web del software.

7. Resultados

Instancia	N	Óptimo	Veleta	RazónV	Convexa	RazónC
x01	30	8787.493	4890.71	0.5566	8259.72	0.9399
x01f	30	710.59342	591.73	0.8327	671.444	0.9449
x02	30	9660.64	5149.2	**0.5330**	9660.14	0.9999
x02f	30	784.7637	520.46	0.6632	740.126	0.9431
x03	30	8127.046	6401.53	0.7877	6759.59	0.8317
x03f	30	654.1769	494.394	0.7557	527.7	0.8067
x04	30	8086.698	5445.03	0.6733	7900.21	0.9769
x04f	30	720.0017	565.835	0.7859	689.633	0.9578
x05	30	7598.985	5228.68	0.6881	7123.52	0.9374
x05f	30	760.6452	613.718	0.8068	692.578	0.9105
x06	30	8293.852	7128.4	0.8595	6611.84	0.7972
x06f	30	867.1466	667.073	0.7693	780.002	0.8995
x07	30	7924.224	6698.61	0.8453	7872.69	0.9935
x07f	30	1026.7941	732.546	0.7134	876.156	0.8533
x08	30	8916.793	7642.04	0.8570	7540.67	0.8457
x08f	30	614.1406	411.423	0.6699	585.222	0.9529
x09	30	8638.361	5622.38	0.6509	7455.84	0.8631
x09f	30	900.1264	832.775	0.9252	834.148	0.9267
x10	30	8202.844	5760.73	0.7023	6967.64	0.8494
x10f	30	665.6899	495.96	0.7450	635.85	0.9552

Tabla 1: Costos de maximización correspondientes a la heurística de la veleta, de la cerradura convexa y del valor óptimo, junto con la razón de aproximación para las instancias de 1-20

Instancia	N	Óptimo	Veleta	RazónV	Convexa	RazónC
x11	30	9092.99	6052.5	0.6656	8723.16	0.9593
x11f	30	525.1358	328.954	0.6264	507.765	0.9669
x12	30	9163.006	7008.48	0.7649	8391.74	0.9158
x12f	30	675.94271	422.286	0.6247	665.57	0.9847
x13	30	7722.931	5338.4	0.6912	6851.03	0.8871
x13f	30	568.6533	478.085	0.8407	555.833	0.9775
x14	30	7527.493	5643.54	0.7497	7244.09	0.9624
x14f	30	701.6291	540.131	0.7698	676.53	0.9642
x15	30	8048.304	5640.54	0.7008	7077.43	0.8794
x15f	30	463.48575	354.232	0.7643	415.701	0.8969
x16	30	8532.851	5866.78	0.6876	8506.81	0.9969
x16f	30	556.7883	377.348	0.6777	523.661	0.9405
x17	30	8604.842	5219.34	0.6066	8590.79	0.9984
x17f	30	533.52294	398.478	0.7469	511.682	0.9591
x18	30	7737.8	5713.14	0.7383	7201.74	0.9307
x18f	30	425.82447	296.166	0.6955	310.767	**0.7298**
x19	30	8489.827	5916.62	0.6969	7245.94	0.8535
x19f	30	419.2628	294.303	0.7020	409.018	0.9756
x20	30	8783.991	5600.92	0.6376	8149.54	0.9278
x20f	30	364.07771	295.091	0.8105	327.784	0.9003

Tabla 2: Costos de maximización correspondientes a la heurística de la veleta, de la cerradura convexa y del valor óptimo, junto con la razón de aproximación para las instancias del 21-30

8. Conclusión

Con la evaluación realizada a cada una de las heurísticas, como era esperado, el valor de maximización óptimo es mayor que el de las heurísticas, no obstante, el tiempo que tarda en ejecutarse el algoritmo óptimo es mucho mayor. Con un banco de 30

puntos, las heurísticas obtienen el costo máximo de forma casi inmediata mientras que el algoritmo óptimo puede tardar algunos minutos.
No fue posible usar bancos de prueba de 60 puntos o más debido a que el tiempo en que tardaba el algoritmo en obtener el valor óptimo era considerablemente largo.
De la evaluación de la heurísticas, en la de la veleta, en la instancia *x02* se observa su peor acercamiento al valor óptimo, con una razón de ≈ 0.5330, ver Tabla 1. Por otro lado, para la heurística de la cerradura convexa, en la instancia *x18f* se observa su peor acercamiento al valor óptimo, con una razón de ≈ 0.7298, ver Tabla 2.
Si comparamos la razón de aproximación de cada heurística, se puede observar que el costo obtenido por la heurística de la cerradura convexa es mejor que el obtenido por la heurística de la veleta, para resolver este problema.

Referencias

[1] O. Aichholzer, S. Cabello, R. Fabila-Monroy, D. Flores-Peñaloza, T. Hackl, C. Huemer, F. Hurtado, y D. R. Wood, "Edge-removal and non-crossing configurations in geometric graphs," *Discrete Mathematics and Theoretical Computer Science*, vol. 12, num. 1, pp. 75–n/a, 2010, copyright - Copyright DMTCS 2010; Document feature - ; Last updated - 2012-03-23. [En línea]. Disponible: http://search.proquest.com/docview/925759865

[2] N. Alon, S. Rajagopalan, y S. Suri, "Long non-crossing configurations in the plane," en *Proceedings of the Ninth Annual Symposium on Computational Geometry*, ser. SCG '93. New York, NY, USA: ACM, 1993, pp. 257–263. [En línea]. Disponible: http://doi.acm.org/10.1145/160985.161145

[3] A. Andrew, "Another efficient algorithm for convex hulls in two dimensions," *Information Processing Letters*, vol. 9, num. 5, pp. 216 – 219, 1979. [En línea]. Disponible: http://www.sciencedirect.com/science/article/pii/0020019079900723

[4] M. Plummer y L. Lovász, *Matching Theory*, ser. North-Holland Mathematics Studies. Elsevier Science, 1986. [En línea]. Disponible: https://books.google.com.mx/books?id=myczZP-J344wC

[5] X. Wu y R. Srikant, "Regulated maximal matching: A distributed scheduling algorithm for multi-hop wireless networks with node-exclusive spectrum sharing," en *Proceedings of the 44th IEEE Conference on Decision and Control*, Dec 2005, pp. 5342–5347.

[6] I. S. Duff, “On algorithms for obtaining a maximum transversal,” *ACM Trans. Math. Softw.*, vol. 7, num. 3, pp. 315–330, Sep. 1981. [En línea]. Disponible: http://doi.acm.org/10.1145/355958.355963

[7] Y.-Q. Cheng, V. Wu, R. T. Collins, A. R. Hanson, y E. M. Riseman, “Maximum-weight bipartite matching technique and its application in image feature matching,” en *Proc. SPIE Visual Comm. and Image Processing*, 1996.

[8] G. Xing, C. Lu, Y. Zhang, Q. Huang, y R. Pless, “Minimum power configuration for wireless communication in sensor networks,” *ACM Trans. Sen. Netw.*, vol. 3, num. 2, Jun. 2007. [En línea]. Disponible: http://doi.acm.org/10.1145/1240226.1240231

[9] M. Fayyazi, D. Kaeli, y W. Meleis, “An adjustable linear time parallel algorithm for maximum weight bipartite matching,” *Information Processing Letters*, vol. 97, num. 5, pp. 186 – 190, 2006. [En línea]. Disponible: http://www.sciencedirect.com/science/article/pii/S0020019005003108

[10] M. Weimer, A. Karatzoglou, y A. Smola, “Improving maximum margin matrix factorization,” *Machine Learning*, vol. 72, num. 3, pp. 263–276, 2008. [En línea]. Disponible: http://dx.doi.org/10.1007/s10994-008-5073-7

[11] J. Edmonds, “Paths, trees, and flowers,” *Canad. J. Math.*, vol. 17, pp. 449–467, 1965. [En línea]. Disponible: https://cms.math.ca/openaccess/cjm/v17/cjm1965v17.0449-0467.pdf

[12] A. Dumitrescu y C. D. Tóth, “Long non-crossing configurations in the plane,” *Discrete & Computational Geometry*, vol. 44, num. 4, pp. 727–752, 2010. [En línea]. Disponible: http://dx.doi.org/10.1007/s00454-010-9277-9

[13] C. D. Alfaro Quintero, “Encajes primitivos de árboles planos,” México, D. F., 2013, proyecto terminal.

[14] J. A. Pérez Arcos, “Encajes primitivos de gráficas planares exteriores,” México, D. F., 2014, proyecto terminal.

[15] G. V. Casas, R. A. C. Campos, M. A. Heredia, y F. J. Z. Martínez, “A triplet integer programming model for the euclidean 3-matching problem,” en *Electrical Engineering, Computing Science and Automatic Control (CCE), 2015 12th International Conference on*, Oct 2015, pp. 1–4.

[16] R. Diestel, *Graph Theory*, ser. Electronic library of mathematics. Springer, 2006. [En línea]. Disponible: https://books.google.com.mx/books?id=aR2TMYQr2CMC

[17] J. Pach, “Geometric graph theory,” Cambridge University Press, Reporte Técnico., 1999.

[18] B. Xiong y Z. Zheng, *Graph Theory*, ser. Mathematical Olympiad series. East China Normal University Press, 2010. [En línea]. Disponible: https://books.google.com.mx/books?id=FHkzyUeQ1MEC

[19] J. O'Rourke, *Computational Geometry in C*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1998. [En línea]. Disponible: https://books.google.com.mx/books?id=gsv7HALW2jYC

9. Apéndice

9.1. Código fuente

9.1.1. main.cpp

Código 1: Función principal.

```
#include <vector>
#include <utility>
#include <iostream>

#include "Map.h"
#include "Match.h"

int main(int argc, char** argv) {
    Match match(argv[1]);
    double windrose_distance = match.computeWindroseMatches();
        double convexhull_distance = match.computeConvexHullMatches();
    std::cout << argv[1] << '␣'
              << windrose_distance << '␣'
              << convexhull_distance << std::endl;
    return 0;
}
```

9.1.2. Coordinate.h

Código 2: Definicion de la clase Coordinate.

```
#ifndef COORDINATE_H
#define COORDINATE_H

#include <cmath>

class Coordinate {
public:
    double x;
    double y;
    Coordinate();
    Coordinate(double a, double b):x(a),y(b){}
    double computeDistance(Coordinate);
        bool operator==(const Coordinate&);
private:
};
```

```
#endif /* COORDINATE_H */
```

9.1.3. Coordinate.cpp

Código 3: Implementación de la clase Coordinate.

```
#include <complex>

#include "Coordinate.h"

Coordinate::Coordinate() {
    x = 0.0;
    y = 0.0;
}

double Coordinate::computeDistance(Coordinate a){
    return std::sqrt(std::pow((a.x - this->x),2) + std::pow((a.y
        - this->y),2));
}

bool Coordinate::operator==(const Coordinate & rhs)
{
        return this->x == rhs.x && this ->y == rhs.y;
}
```

9.1.4. Map.h

Código 4: Definición de la clase Map.

```
#ifndef MAP_H
#define MAP_H

#include "Coordinate.h"
#include <vector>
#include <fstream>
#include <algorithm>
#include <cmath>
#include <functional>

#define PI 3.14159265358979323846

typedef std::vector<Coordinate> CoordinateSet;
typedef std::vector<Coordinate>::iterator CoordinateSetIterator;
```

```
class Map {
public:
    Map(const char* filename);

    virtual ~Map();
    void sortX();
    void sortY();
    void sortXY();
    void sortYX();
    void sortXthenY();
        void pop_back();
    typedef std::function<void(Map&)> SortAlgorithm;
    CoordinateSetIterator begin(){return coordinateSet.begin();}
    CoordinateSetIterator end(){return coordinateSet.end();}
    SortAlgorithm* getSortAlgorithm();
        const CoordinateSet& getCoordinateSet();
        CoordinateSetIterator erase(CoordinateSetIterator);

private:
    CoordinateSet coordinateSet;
    SortAlgorithm sortAlg[4];
};

#endif /* MAP_H */
```

9.1.5. Map.cpp

Código 5: Implementación de la Clase Map.

```
#include "Map.h"
#include <iostream>

Map::Map(const char* filename) {
    int n, i = 0;
    Coordinate coordinate;
    std::ifstream infile;
    infile.open (filename);
    infile >> n;
    n=n/2;

    while(infile >> coordinate.x >> coordinate.y && (i++)<n)
    {
        coordinateSet.push_back(coordinate);
        std::cout << map.back().x << "␣" << map.back().y << "␣mapa"
```

```
                << n <<std::endl;
    }
    infile.close();
    sortAlg[0] = &Map::sortX;
    sortAlg[1] = &Map::sortY;
    sortAlg[2] = &Map::sortXY;
    sortAlg[3] = &Map::sortYX;
}

Map::~Map() {
}

Map::SortAlgorithm* Map::getSortAlgorithm() {
    return sortAlg;
}

const CoordinateSet & Map::getCoordinateSet()
{
        const CoordinateSet& ref = coordinateSet;
        return ref;
}

CoordinateSetIterator Map::erase(CoordinateSetIterator eras)
{
        auto it = coordinateSet.erase(eras);
        return it;
}

void Map::sortX(){
    std::sort(coordinateSet.begin(), coordinateSet.end(),
        [](const Coordinate& a, const Coordinate& b) -> bool{
        return a.x < b.x;
    });
}

void Map::sortY(){
    std::sort(coordinateSet.begin(), coordinateSet.end(),
        [](const Coordinate& a, const Coordinate& b) -> bool{
        return a.y < b.y;
    });
}

void Map::sortXY(){
    std::sort(coordinateSet.begin(), coordinateSet.end(),
        [](const Coordinate& a, const Coordinate& b) -> bool{
        return (a.x*cos(315*PI/180) - a.y*sin(315*PI/180))
                < (b.x*cos(315*PI/180) - b.y*sin(315*PI/180));
```

```
    });
}

void Map::sortYX(){
    std::sort(coordinateSet.begin(), coordinateSet.end(),
        [](const Coordinate& a, const Coordinate& b) -> bool{
        return (a.x*sin(315*PI/180) + a.y*cos(315*PI/180))
                < (b.x*sin(315*PI/180) + b.y*cos(315*PI/180));
    });
}

void Map::sortXthenY(){
    std::sort(coordinateSet.begin(), coordinateSet.end(),
        [](const Coordinate& a, const Coordinate& b) -> bool{
        return (a.x < b.x || (a.x == b.x && a.y < b.y));
    });
}

void Map::pop_back()
{
        coordinateSet.pop_back();
}
```

9.1.6. Match.h

Código 6: Definición de la Clase Match.

```
#ifndef MATCH_H
#define MATCH_H

#include <utility>
#include <vector>
#include <string>
#include <algorithm>

#include "ConvexHull.h"
#include "Map.h"

typedef std::vector<std::pair<Coordinate, Coordinate> > MatchSet;

class Match {
public:
    Match(const char*);

    double computeWindroseMatches();
```

```
    double computeConvexHullMatches();
        void makeMatch(MatchSet&);
        void makeMatch(MatchSet&, CoordinateSet&);
private:
    Map map;
    std::string filename;

    void saveData(MatchSet&, const char* extension);
    double totalDistance(MatchSet&);
};

#endif /* MATCH_H */
```

9.1.7. Match.cpp

Código 7: Implementación de la clase Match.

```
#include "Match.h"

Match::Match(const char* filename): map(filename){
    this->filename = filename;
}

void Match::makeMatch(MatchSet& matchSet) {
    std::pair<Coordinate, Coordinate> match;
    for(auto it = map.begin(); it != map.end(); it++){
        match.first = *it;
        match.second = *(++it);
        matchSet.push_back(match);
    }
}

void Match::makeMatch(MatchSet & matchSet, CoordinateSet & convexHull)
{
        std::pair<Coordinate, Coordinate> match;
        for (auto it = convexHull.begin(); it != convexHull.end() - 1;
        it++) {
                match.first = *it;
                match.second = *(it + 1);
                matchSet.push_back(match);
        }
        match.first = convexHull.back();
        match.second = convexHull.front();
        matchSet.push_back(match);
}
```

```
double Match::totalDistance(MatchSet& matchSet) {
    double totalDist = 0.0;
    for(auto& match: matchSet){
        totalDist += match.first.computeDistance(match.second);
    }
    return totalDist;
}

void Match::saveData(MatchSet& matches, const char* extension) {
    std::string result = filename + extension;
    std::ofstream ofs(result.c_str());
    if(ofs.is_open()) {
        for(auto& match: matches)
            ofs << match.first.x << "␣" << match.first.y << "␣"
                    << match.second.x << "␣" << match.second.y
                                        << std::endl;
    }
    ofs.close();
}

double Match::computeWindroseMatches(){
    MatchSet matchSets[4];
    double distances[4];
    Map::SortAlgorithm* sort = map.getSortAlgorithm();
    for(int i = 0; i < 4 ; i++) {
        sort[i](map);
        makeMatch(matchSets[i]);
        distances[i] = totalDistance(matchSets[i]);
    }
    auto matchSetIndex = std::distance(distances, std::max_element(
        distances, distances + 3));
    saveData(matchSets[matchSetIndex], ".wr");
    return distances[matchSetIndex];
}

double Match::computeConvexHullMatches()
{
        CoordinateSet convexHull;
        ConvexHull ch;
        MatchSet convexHullMatches, maxMatches;
        int lenght = map.getCoordinateSet().size() / 2;

        for (int i = 0; i < lenght ; i++, convexHullMatches.clear())
        {
                ch(map, convexHull);
                makeMatch(convexHullMatches, convexHull);
```

```
                auto it = std::max_element(convexHullMatches.begin(),
                convexHullMatches.end(), [](std::pair<Coordinate,
                Coordinate> lhs,
                std::pair<Coordinate, Coordinate> rhs) -> bool {
                        return lhs.first.computeDistance(lhs.second)
                        < rhs.first.computeDistance(rhs.second);
                });
                maxMatches.push_back(*it);
                auto first = std::find(map.begin(), map.end(),
                it->first);
                std::iter_swap(map.end() - 1, first);
                map.pop_back();
                auto second = std::find(map.begin(), map.end(),
                it->second);
                std::iter_swap(map.end() - 1, second);
                map.pop_back();
        }
        saveData(maxMatches, ".ch");
    return totalDistance(maxMatches);
}
```

9.1.8. ConvexHull.h

Código 8: Definición de la Clase ConvexHull.

```
#ifndef CONVEXHULL_H
#define CONVEXHULL_H
#include "Map.h"
#include <vector>

class ConvexHull {
public:
    ConvexHull();
    ConvexHull(const ConvexHull& orig);
    virtual ~ConvexHull();
    void operator()(Map&, CoordinateSet&);
private:
    Coordinate anchor, tail;

        bool isCCW(Coordinate& a, Coordinate& b, Coordinate& c);
        double cross(Coordinate & o, Coordinate & a, Coordinate & b);

};

#endif /* CONVEXHULL_H */
```

9.1.9. ConvexHull.cpp

Código 9: Implementación de la Clase ConvexHull.

```
#include "ConvexHull.h"

ConvexHull::ConvexHull() {
}

ConvexHull::ConvexHull(const ConvexHull& orig) {
}

ConvexHull::~ConvexHull() {
}


void ConvexHull::operator()(Map& map, CoordinateSet& cs) {
        int n = map.getCoordinateSet().size(), k = 0;
        if (n == 1) return;
        cs.resize(2 * n);
        map.sortXthenY();
        Coordinate c;
        for (int i = 0; i < n; ++i) {
                c.x = map.getCoordinateSet()[i].x;
                c.y = map.getCoordinateSet()[i].y;
                while (k >= 2 && cross(cs[k - 2], cs[k - 1], c) <= 0)
                        k--;
                cs[k++] = map.getCoordinateSet()[i];
        }

        for (int i = n - 2, t = k + 1; i >= 0; i--) {
                c.x = map.getCoordinateSet()[i].x;
                c.y = map.getCoordinateSet()[i].y;
                while (k >= t && cross(cs[k - 2], cs[k - 1], c) <= 0)
                        k--;
                cs[k++] = map.getCoordinateSet()[i];
        }

        cs.resize(k - 1);
}

bool ConvexHull::isCCW(Coordinate& a, Coordinate& b, Coordinate& c)
{
        return a.x * (b.y - c.y) + b.x * (c.y - a.y) + c.x * (a.y - b.y)
        > 0;
}
```

```
double ConvexHull::cross(Coordinate& o, Coordinate& a, Coordinate& b)
{
        return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
}
```

9.1.10. poligono.lp

Código 10: Código muestra del programa lineal que se pasa a Gurobi para un hexágono.

```
Maximize
 3.04138 edge_0_1 +  5.26308 edge_0_2 +  6.01664 edge_0_3 +  5.16236 edge_0_4 +
2.98329 edge_0_5 +  3.00832 edge_1_2 +  5.16236 edge_1_3 +  5.96657 edge_1_4 +
5.22015 edge_1_5 +  2.98329 edge_2_3 +  5.22015 edge_2_4 +  6.08276 edge_2_5 +
3.04138 edge_3_4 +  5.26308 edge_3_5 +  3.00832 edge_4_5
Subject To
 edge_0_1  +  edge_0_2  +  edge_0_3  +  edge_0_4  +  edge_0_5  = 1
 edge_0_1  +  edge_1_2  +  edge_1_3  +  edge_1_4  +  edge_1_5  = 1
 edge_0_2  +  edge_1_2  +  edge_2_3  +  edge_2_4  +  edge_2_5  = 1
 edge_0_3  +  edge_1_3  +  edge_2_3  +  edge_3_4  +  edge_3_5  = 1
 edge_0_4  +  edge_1_4  +  edge_2_4  +  edge_3_4  +  edge_4_5  = 1
 edge_0_5  +  edge_1_5  +  edge_2_5  +  edge_3_5  +  edge_4_5  = 1
 edge_0_2 + edge_1_3 <= 1
 edge_0_2 + edge_1_4 <= 1
 edge_0_2 + edge_1_5 <= 1
 edge_0_3 + edge_1_4 <= 1
 edge_0_3 + edge_1_5 <= 1
 edge_0_3 + edge_2_4 <= 1
 edge_0_3 + edge_2_5 <= 1
 edge_0_4 + edge_1_5 <= 1
 edge_0_4 + edge_2_5 <= 1
 edge_0_4 + edge_3_5 <= 1
 edge_1_3 + edge_0_2 <= 1
 edge_1_3 + edge_2_4 <= 1
 edge_1_3 + edge_2_5 <= 1
 edge_1_4 + edge_0_2 <= 1
 edge_1_4 + edge_0_3 <= 1
 edge_1_4 + edge_2_5 <= 1
 edge_1_4 + edge_3_5 <= 1
 edge_1_5 + edge_0_2 <= 1
 edge_1_5 + edge_0_3 <= 1
 edge_1_5 + edge_0_4 <= 1
 edge_2_4 + edge_0_3 <= 1
 edge_2_4 + edge_1_3 <= 1
 edge_2_4 + edge_3_5 <= 1
 edge_2_5 + edge_0_3 <= 1
```

```
 edge_2_5 + edge_0_4 <= 1
 edge_2_5 + edge_1_3 <= 1
 edge_2_5 + edge_1_4 <= 1
 edge_3_5 + edge_0_4 <= 1
 edge_3_5 + edge_1_4 <= 1
 edge_3_5 + edge_2_4 <= 1
Binary
 edge_0_1
 edge_0_2
 edge_0_3
 edge_0_4
 edge_0_5
 edge_1_2
 edge_1_3
 edge_1_4
 edge_1_5
 edge_2_3
 edge_2_4
 edge_2_5
 edge_3_4
 edge_3_5
 edge_4_5
end
```