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

Reporte de Proyecto Terminal

Algoritmo y heurística para el problema de recolección de un banco de alimentos

Emmanuel Valenzuela Jaramillo
2132003253

Trimestre 2017-Primavera

Asesor:

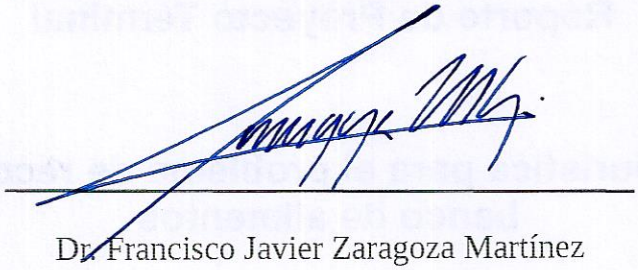
Dr. Francisco Javier Zaragoza Martínez
Profesor Investigador Titular C
Departamento de Sistemas

Coasesor:

Dr. Marco Antonio Heredia Velasco
Profesor Investigador Asociado D
Departamento de Sistemas

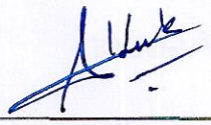
Declaratoria

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.



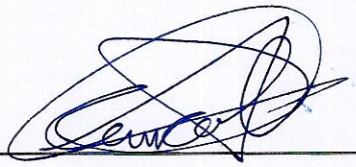
Dr. Francisco Javier Zaragoza Martínez

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

Yo, Emmanuel Valenzuela Jaramillo, 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, así como en el Repositorio Institucional de la UAM Azcapotzalco.



Emmanuel Valenzuela Jaramillo

Tabla de Contenido

Resumen

Un banco de alimentos es una organización sin fines de lucro que se basa en el voluntariado. Su principal objetivo es evitar el desperdicio o mal uso de los alimentos, recuperando, especialmente los no perecederos, de la sociedad y redistribuyéndolos entre las personas que lo necesiten.

Un ejemplo es BAMX, una organización conformada por más de 50 bancos de alimentos distribuidos por todo el territorio mexicano. Actualmente benefician a 1,137,000 personas en pobreza alimentaria de los que 32% son niños, 19% adolescentes, 39% adultos y 10% adultos mayores. De los alimentos que recolectan el 56% son perecederos (frutas y verduras) y el otro 44% son no perecederos (abarrotes, cereales, etc.). [7]

El problema de recolección del banco de alimentos se plantea así. Se tiene un banco de alimentos que quiere recolectar los alimentos sobrantes de diferentes puntos (comercios, empresas o personas). Cada punto tiene sus coordenadas y se comunica con el banco cada día para informarle si tiene o no alimentos sobrantes por lo que no todos los días se tienen que recorrer los mismos puntos. Ir a cada punto a recolectar los alimentos que se tienen disponibles tiene un cierto costo. Cada alimento que se recolecta representa una ganancia. El banco tiene un camión recolector que puede decidir si visitar un punto o no, cada día, dependiendo de la relación que exista entre el costo de ir y la ganancia.

En el proyecto se diseñó e implementó un algoritmo exacto y una heurística para resolver el problema de recolección de un banco de alimentos.

Ejecutamos nuestra heurística con diversos casos de prueba así como con el algoritmo exacto y observamos que en el caso de la heurística el tiempo parecía ser constante mientras que en el algoritmo exacto parecía crecer exponencialmente. Las soluciones que generó la heurística se acercaban bastante a la solución óptima encontrada observando que superaban el 80% del optimo y alcanzando casi el 90%.

Por lo anterior consideramos que nuestra heurística podría dar buenos resultados en la práctica, en unos pocos minutos incluso para casos con 200 vértices.

Introducción

Un banco de alimentos es una organización sin fines de lucro que se basa en el voluntariado. Su principal objetivo es evitar el desperdicio o mal uso de los alimentos, recuperando, especialmente los no perecederos, de la sociedad y redistribuyéndolos entre las personas que lo necesiten.

Un ejemplo es BAMX, una organización conformada por más de 50 bancos de alimentos distribuidos por todo el territorio mexicano. Actualmente benefician a 1,137,000 personas en pobreza alimentaria de los que 32% son niños, 19% adolescentes, 39% adultos y 10% adultos mayores. De los alimentos que recolectan el 56% son perecederos (frutas y verduras) y el otro 44% son no perecederos (abarrotes, cereales, etc.). [7]

El problema de recolección del banco de alimentos se plantea así. Se tiene un banco de alimentos que quiere recolectar los alimentos sobrantes de diferentes puntos (comercios, empresas o personas). Cada punto tiene sus coordenadas y se comunica con el banco cada día para informarle si tiene o no alimentos sobrantes por lo que no todos los días se tienen que recorrer los mismos puntos. Ir a cada punto a recolectar los alimentos que se tienen disponibles tiene un cierto costo. Cada alimento que se recolecta representa una ganancia. El banco tiene un camión recolector que puede decidir si visitar un punto o no, cada día, dependiendo de la relación que exista entre el costo de ir y la ganancia.

En el proyecto se diseñó e implementó un algoritmo exacto y una heurística para resolver el problema de recolección de un banco de alimentos. Éste de ninguna manera está relacionado con BAMX o cualquier otro banco de alimentos existente.

Justificación

Debido a que los bancos de alimentos llevan a cabo la recolección con vehículos motorizados como camiones o camionetas, se tiene que gastar combustible para llegar a los diferentes puntos, esto representa un gasto económico que se quiere minimizar. Además al tratarse de alimentos, especialmente los perecederos, el tiempo de transporte es crucial por lo que entre más rápido se logre recolectarlos mejor. Teniendo una buena ruta se logra recolectar la mayor cantidad de alimentos minimizando el tiempo, evitando el desperdicio, y el gasto de combustible.

Objetivos

Objetivo general: Diseñar, implementar y evaluar un algoritmo exacto y una heurística para resolver el problema de recolección de un banco de alimentos.

Objetivos específicos:

- Diseñar e implementar un algoritmo generador de instancias del problema de recolección.
- Diseñar e implementar un algoritmo exacto para solucionar el problema.
- Diseñar e implementar una heurística para solucionar el problema.
- Comparar los resultados de la heurística con los del algoritmo exacto.

Antecedentes

Proyectos terminales

1) Implementación de un algoritmo de aproximación para el problema del cartero con restricciones en los arcos [1].

El problema se pretende resolver es una variación del problema del cartero chino, en el cual se ve obligado a pasar por todos los puntos y regresar al inicio, como la presente propuesta. A diferencia de este proyecto que tiene que recorrer los arcos, en nuestro proyecto recorremos los vértices.

2) Tres algoritmos para desplazar un robot en el plano en presencia de obstáculos [2].

Su proyecto consiste en algoritmos que construyen una navegación libre de colisiones que lleva al robot desde un punto inicial a un punto final. Al igual que el presente proyecto se busca el camino más corto para llegar de un punto A a un punto B pudiendo elegir el más conveniente. Se diferencia por el punto final, ya que en el caso de nuestro proyecto, ambos puntos son el mismo.

3) Aplicación para dispositivos con Android que encuentre la mejor ruta entre dos estaciones del Metro [3].

El proyecto arroja como resultado la mejor ruta entre dos estaciones del Metro basándose en distintas métricas. Se diferencia del presente proyecto porque el punto de origen y destino no son el mismo pero se asemeja ya que una de sus funciones permite escoger por cuales estaciones se quiere o no pasar, forzando la elección de tomarlas o evitarlas.

Artículos

4) Approximation Algorithms for Some Routing Problems [4].

El artículo presenta varios algoritmos de aproximación para algoritmos de enrutamiento NP-completos con el valor de aproximación al óptimo con varias heurísticas, tal como se plantea en el presente proyecto. En el artículo habla de variaciones del problema del agente viajero (TSP) con k agentes.

5) The prize collecting traveling salesman problem [5].

Se habla de una variación del TSP en la cual en cada ciudad se consigue un premio pero cada ciudad que no se visita genera una penalización. Esta variación se conoce como el problema del agente viajero recolector de premios (PCTCP). Se busca minimizar el costo del viaje y las penalizaciones además de maximizar el costo del total de premios que se consiguieron. Se relaciona con el presente proyecto en la elección de visitar o no un punto ya que esto me puede generar una pérdida mayor a la ganancia.

Libros

6) An additive approach for the optimal solution of the prize-collecting traveling salesman problem [6]

El capítulo de este libro habla, al igual que [5], sobre la variante del problema del agente viajero, el PC-TCP. Se abordan métodos para la aproximación a la solución óptima del problema así como resultados computacionales de pruebas hechas y llegando a un algoritmo de ramificación y poda para la solución óptima.

Marco Teórico

En 1954, Dantzig, Fulkerson y Johnson propusieron atacar el problema del agente viajero con métodos de programación lineal como se describe a continuación:

Sea $G=(V,E)$ una gráfica y x el vector característico de un recorrido del agente viajero, es decir, $x_e=1$ si el recorrido usa la arista $e\in E$ y $x_e=0$ si no la usa. Entonces x satisface que:

$$x(\delta(v))=2 \quad , \quad \forall \ v \ \in \ V$$
$$x_e\in\{0,1\} \quad \forall e\in E$$

Lo que quiere decir que la suma de todos las aristas que llegan a cada vértice *v* debe ser igual a 2 para asegurar así que solo una llega y solo una sale por lo que se pasa por el vértice una sola vez. Las desigualdades dicen que los valores de las variables que representas las aristas son 0 si no se utiliza ó 1 si sí.

Para asegurar que no haya subrecorridos (como se muestra en la figura 1 en la que seis de sus vértices aseguran que solo llega una arista y sale otra pero no están conectadas) agregamos las siguientes restricciones:

$$x(\delta(S))\geq 2, \quad \forall S\subseteq V \ con \ \emptyset \neq S \neq V$$

En donde aseguramos que para cada subconjunto S de vértices hay al menos dos aristas que van al complemento como se muestra en la figura 2.

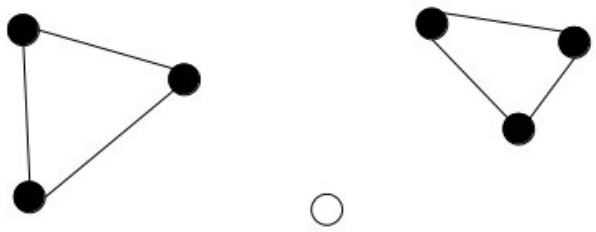


Figura 1: Ejemplo de subrecorridos que se pueden generar

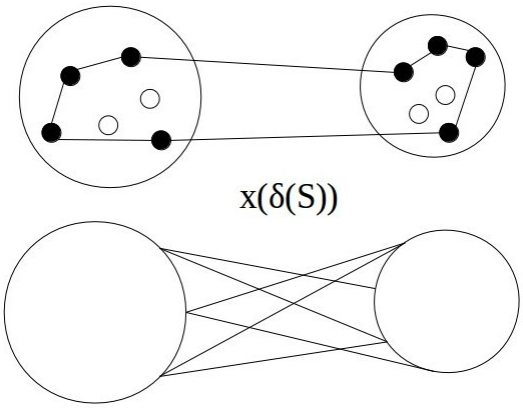


Figura 2: Suma de las aristas que cruzan entre el subconjunto y su complemento

Con esto el modelo de de programación lineal de Dantzig, Fulkerson y Johnson queda así:

$$\begin{gathered} \textit{Minimizar} \sum (c_e x_e : e \in E) \\ \text{sujeto a} \\ x(\delta(v)) = 2 \quad , \quad \forall \ v \in V \\ x(\delta(S)) \geq 2, \ \ \forall \emptyset \neq S \neq V \\ 0 \leq x_e \leq 1, \ \ \forall e \in E \end{gathered}$$

A partir del cual se va a trabajar y modificar de la siguiente manera:

Tenemos un conjunto de *n* puntos $X = \{x_1, x_2, ..., x_n\}$, $x_i \in \{0,1\}$ donde x_i es 1 cuando se pasa por él o 0 cuando no, cada uno con su respectivo valor de premio p_i que nos genera una ganancia, además de un conjunto de *n(n-1)/2* aristas $Y = \{y_1, y_2, ..., y_{n(n-1)/2}\}$, $y_i \in \{0,1\}$ donde y_i es 1 cuando utilizamos esa arista o 0 cuando no, cada una con un costo c_i que nos genera un costo por lo que la función objetivo es:

$$\textit{Maximizar} \sum_{i=1}^{n} p_i x_i - \sum_{i=0}^{n(n-1)/2} c_i y_i$$

Si la suma de todas las aristas que llegan y todas las que salen al vértice i es igual a 2 aseguramos que solo una vez se entra y solo una vez se sale. El conjunto E_i es el conjunto de todas las aristas del vértice i, entonces para cada $i \in \{1,2,\ldots,n\}$

$$\sum_{e \in E_i} y_e = 2x_i$$

Adicionalmente como siempre se parte del origen y se regresa a éste $x_1 = 1$

Ahora para asegurar que no haya subrecorridos, para cada subconjunto *S* de vértices que no sea el vacío ni el conjunto completo $\emptyset \subsetneq S \subsetneq V$ definimos las variables $z_s \in \{0,1\}$ y $w_s \in \{0,1\}$ donde z_s es 1

si paso por algún vértice de ese subconjunto o 0 si no y w_s es 1 si salgo o entro al subconjunto S o 0 si no. Para asegurarnos de ello se hacen las siguientes restricciones:

$$|s|z_s \geq \sum_{i \in S} x_i \geq z_s$$

Con lo que forzamos el valor de z_s a 0 si no se pasa por ningún vértice de s o 1 si sí. Adicionalmente revisamos lo mismo para el complemento de s

$$|\bar{s}|z_{\bar{s}} \geq \sum_{i \in \bar{S}} x_i \geq z_{\bar{s}}$$

Tenemos con esto z_s y $z_{\bar{s}}$ que sólo pueden tener cuatro combinaciones de valores posibles: 01, 10 y 11, no podemos tener 00 ya que como forzamos a pasar por el origen en al menos uno de los subconjuntos está éste vértice. Las primeras combinaciones nos dicen que solo pasé por uno de los subconjuntos por lo que no salí de él. Caso contrario, en 11 ya que ésto dice que hay que ir y regresar al menos una vez y de la siguiente manera forzamos el valor de las w_s

$$2w_s + 1 \geq z_s + z_{\bar{s}} \geq 2w_s$$

Por último hay que asegurar que si se cruzan el subconjunto con su complemento el total de aristas que cruzan sea al menos 2. Sea C el conjunto de aristas que cruzan de S al complemento, entonces:

$$\sum_{c \in C} y_c \geq 2w_s$$

Así nuestro modelo de programación lineal del problema queda de la siguiente manera

$$\textit{Maximizar} \sum_{i=1}^{n} p_i x_i - \sum_{i=0}^{n(n-1)/2} c_i y_i$$

sujeto a

$$x_1 = 1$$

$$\sum_{e \in E_i} y_e = 2x_i \quad , \quad \forall i \in \{1,2,\ldots,n\}$$

$$\sum_{c \in C} y_c \geq 2w_s \quad , \quad \forall \emptyset \subsetneq S \subsetneq V$$

$$|s|z_s \geq \sum_{i \in S} x_i \geq z_s \quad , \quad \forall \emptyset \subsetneq S \subsetneq V$$

$$|\bar{s}|z_{\bar{s}} \geq \sum_{i \in \bar{S}} x_i \geq z_{\bar{s}} \quad , \quad \forall \emptyset \subsetneq S \subsetneq V$$

$$2w_s + 1 \geq z_s + z_{\bar{s}} \geq 2w_s \quad , \quad \forall \emptyset \subsetneq S \subsetneq V$$

$$0 \leq x_i \leq 1 \; \forall i \in X$$

$$0 \leq y_i \leq 1 \; \forall i \in Y$$

$$0 \leq z_s \leq 1 \; \forall \emptyset \subsetneq S \subsetneq V$$

$$0 \leq w_s \leq 1 \; \forall \emptyset \subsetneq S \subsetneq V$$

Desarrollo del Proyecto

Generar instancias

Se implementó una estructura para representar cada punto, la cual tiene dos valores enteros, que son sus coordenadas en el plano, y un valor entero que representa el valor del "premio" que recojo en ese punto.

Para asignar las coordenadas, el módulo genera números aleatorios entre 0 y 30 (se considera un plano de 30 km² ya que es el ancho aproximado de la Ciudad de México) auxiliándose de la función *rand()* de la biblioteca *stdlib.h* como se puede ver en el código del Apendice A.1.

Para la asignación del valor del premio en cada punto se generan valores aleatorios con distribución exponencial, esto para obtener con menor frecuencia valores altos. Para esto se usa la función de densidad de probabilidad $f(x)=\lambda e^{-\lambda x}$ y mediante el método de la transformación inversa obtenemos la función de distribución acumulada $F(x)=1-e^{-\lambda x}$ y despejando tenemos la función $F^{-1}(p)\frac{-\ln(1-p)}{\lambda}$ a la cual se le pasa un valor aleatorio entre 0 y 1 para generar números aleatorios con distribución exponencial con media λ la cual utilizamos como 30, como se puede apreciar en el Apéndice A.2.

En los siguientes histogramas se puede apreciar la frecuencia de los números generando 1000, 10000 y 100000 números.

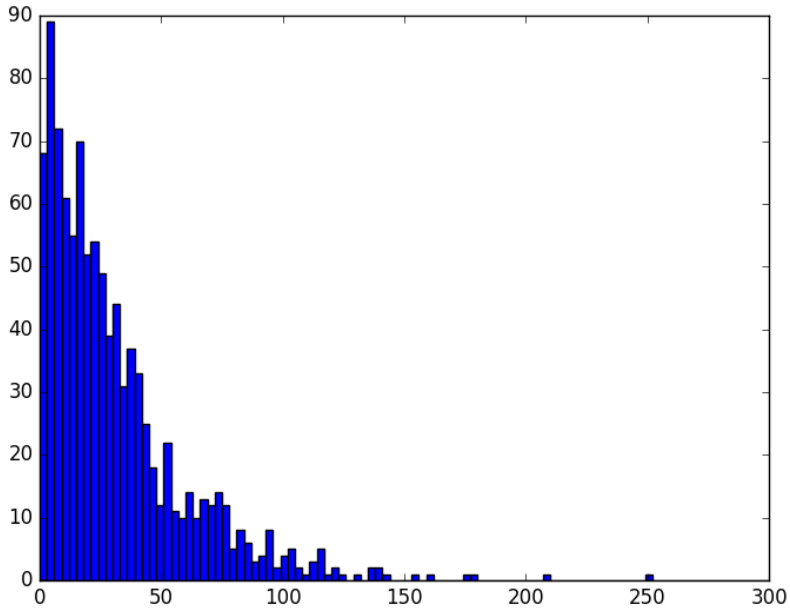


Figura 3: Histograma de 1000 números aleatorios con distribución exponencial

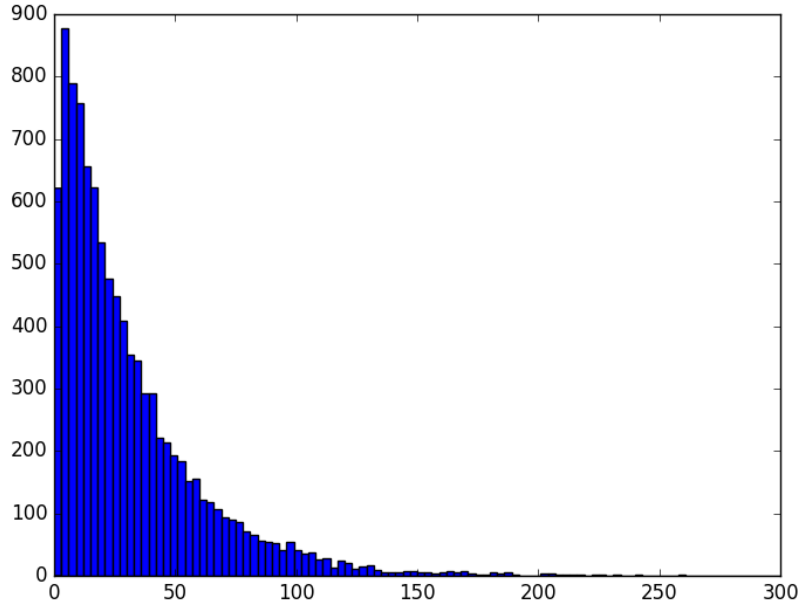


Figura 4: Histograma de 1000 números aleatorios con distribución exponencial

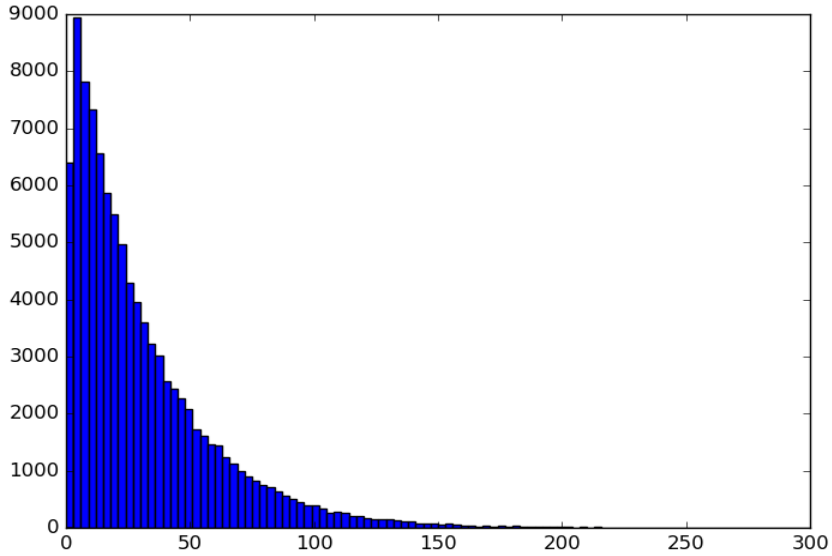


Figura 5: Histograma de 100000 números aleatorios con distribución exponencial

Por último se generan dos archivos con la instancia creada, uno para el algoritmo exacto, que es el archivo de entrada para Gurobi (que se explicará en el módulo del algoritmo exacto), y otro para la heurística. Podemos apreciar la generación de la instancia en el apéndice B.

Algoritmo Exacto

Como ya se explicó, atacamos el problema modelándolo como un problema de programación lineal y para solucionarlo se utiliza la herramienta Gurobi[8] que es una de las mejores herramientas de software para resolver modelos de programación lineal entera mixta. Hay que mandarle un archivo con la función objetivo, las restricciones y los tipos de variables. Éste archivo es el que crea el módulo generador.

Trabajamos con los siguientes elementos:

a) Función objetivo; $Maximizar \sum_{i=1}^{n} p_i x_i - \sum_{i=0}^{n(n-1)/2} c_i y_i$

b) Restricción de las aristas que llegan a un vértice; $\sum_{e \in E_i} y_e = 2x_i$

c) Restricción para asegurar al menos una ida y una vuelta; $\sum_{c \in C} y_c \geq 2w_s$

d) Restricción para verificar que se visita un punto en un subconjunto; $|s| z_s \geq \sum_{i \in S} x_i \geq z_s$

e) Restricción para verificar que se visita un punto en un el complemento; $|\bar{s}| z_{\bar{s}} \geq \sum_{i \in \bar{S}} x_i \geq z_{\bar{s}}$

f) Restricción para forzar la ida y el regreso; $2w_s + 1 \geq z_s + z_{\bar{s}} \geq 2w_s$

g) Tipos de variables; $0 \leq x_i \leq 1 \; \forall i \in X$, $0 \leq y_i \leq 1 \; \forall i \in Y$, $0 \leq z_s \leq 1 \; \forall \emptyset \subsetneq S \subsetneq V$, $0 \leq w_s \leq 1 \; \forall \emptyset \subsetneq S \subsetneq V$

El cómo el módulo generador creó los elementos a, b y g se puede apreciar en el apéndice B.4 para el resto de los elementos se implementó una función recursiva que generaba todas las posibles cadenas de 0's y 1's en un vector de tamaño n y se trabajaba con cada una de estas cadenas. Los 1's en la cadena representan el conjunto S y los 0's el complemento con lo que se podía trabajar los elementos de la c, d, e y f. El elemento d, e y f tienen dos partes cada uno ya que se tratan de dos restricciones diferentes, entonces nombraremos las restricciones así: c → Rest1, d.1 → Rest2, d.2 → Rest3, e.1 → Rest4, e.2 → Rest5, f.1 → Rest6, f.2 → Rest7.

Ya con ésto, para cada subconjunto S que no es el vacío ni el completo se generan las restricciones tal como se puede apreciar en el apéndice B.5.

Adicionalmente manejaremos la siguiente restricción para asegurar que el recorrido se haga en menos de ocho horas, lo que es una jornada de trabajo y llamaremos t a lo que cuesta entonces:

$$\sum_{i=0}^{n(n-1)/2} c_i y_i \leq t$$

Gurobi recibe el archivo que se creó, encuentra la solución optima y genera un archivo con los valores que se tienen que asignar a las variables para la solución que encontró incluyendo cada x_i, y_i, z_s y w_s . Los datos que nos importan de este archivo son las x_i y las y_i que nos dirán por cuales puntos sí hay que pasar y en qué orden.

Heurística

Algoritmo Glotón

Para empezar se lee del archivo con la instancia el valor de la *n* seguido de las coordenadas de los puntos y sus respectivos premios, después se genera una matriz cuyos elementos $A_{i,j}$ representan la distancia (manhattan) entre el punto *i* y el *j*.

Para generar una primera solución lo que se hace es, empezando por el punto 0, moverse al punto más cercano que no se haya visitado antes, para esto se utiliza un vector en el que se va registrando los puntos visitados. Se aproxima una velocidad promedio de 20 km/hr y una jornada de trabajo de 8 horas por lo que la distancia máxima a recorrer son 160 km. Cada que se mueve al siguiente punto se pregunta primero si llegar a él y volver al inicio suman menos de 160 km, de lo contrario preguntar si se logra con el siguiente punto y así hasta que no se puedan recorrer más en el limite establecido asegurando así recorrer los puntos seleccionados en menos de 8 horas.

El costo de un recorrido se obtiene suponiendo lo siguiente:

- Un rendimiento de 9 km por litro y cada litro cuesta $17, así obtenemos un valor *g* que representa el precio de recorrer 1 km.
- Un salario promedio por día *s* de $105.5
- Un mantenimiento *m* que se le da al vehículo cada 6000 km con precio de $4000 y se considera una proporción de éste por cada kilómetro.

Si *d* es la longitud del recorrido, el costo total se calcula así $c=(d*g)+m+s$.

La ganancia de un recorrido se calcula sumando el valor de los premios de los puntos que se visitan haciendo así el total igual a la ganancia menos el costo del recorrido. Se puede apreciar la ejecución de éste módulo en la figura 6. El código en el que se implementa ésto se puede apreciar en el Apéndice C.

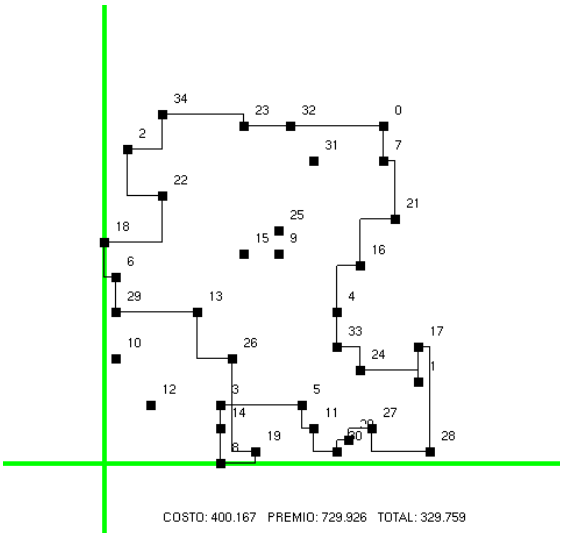


Figura 6: Ejemplo del recorrido que encuentra el algoritmo glotón

2 – Opt

Una vez que se tiene el recorrido inicial dado por el algoritmo glotón se procede a verificar cada par de aristas de la siguiente manera:

En una primera instancia se tiene un recorrido $\ldots \rightarrow a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f \rightarrow g \rightarrow \ldots$ y con el par de aristas $b \rightarrow c$ y $e \rightarrow f$ se revisa si recorrerlos de la manera $b \rightarrow e$ y $c \rightarrow f$ cuesta menos como se muestra en la figura 7. De ser así se intercambian formando un nuevo recorrido, de lo contrario se deja el recorrido como estaba. Esto se realiza para cada par de aristas hasta que ningún intercambio genere una mejora es cuando decimos que ya es 2-Opt.

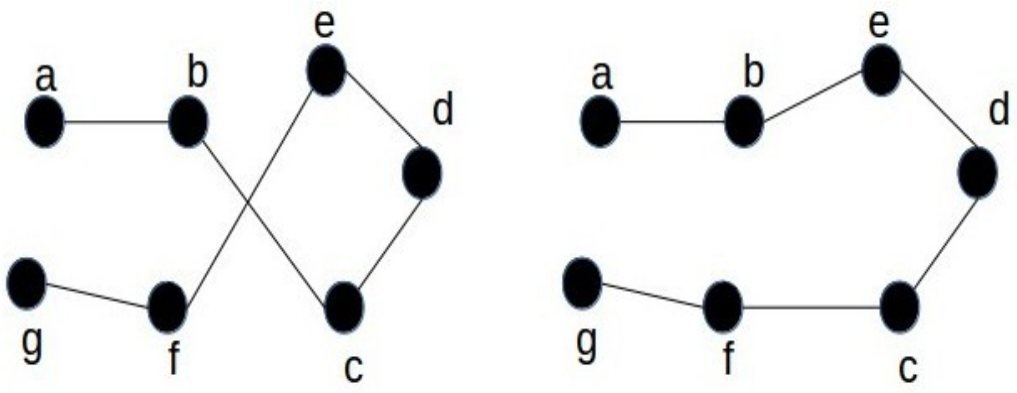


Figura 7: Ejemplo de mejora 2-Opt

El código del desarrollo de este punto se encuentra en el apéndice D.

1 – Opt

Visitar un punto genera un costo traducido a lo que cuesta ir y volver de éste así como una ganancia que es el valor del premio. El siguiente paso es verificar que realmente visitar un punto es la mejor decisión revisando si al quitarlo mejora nuestro objetivo ya que haciéndolo, aunque perdemos el premio, gastamos menos en recorrido. Así si la diferencia entre el valor del premio menos el costo de recorrerlo resulta ser negativo decidimos no visitarlo ya que esto representa que perdemos al decidir sí hacerlo.

Desde el principio puede que queden algunos puntos fuera del recorrido (por no alcanzar a recorrerlos en menos de 8 horas) y después de revisar si conviene quitarlos quedan otros tantos. Para cada uno de estos puntos sin visitar hay que decidir si poniéndolo en algún lugar del recorrido genera una mejora, ésto se hace pesado porque de decidir sí ponerlo se tiene que revisar en qué parte del recorrido se tiene que hacer.

Esto se verifica para cada punto que esté y que no hasta que ya ningún cambio represente una mejora y es cuando decimos que ya es 1-Opt. El código del desarrollo de este punto se encuentra en el apéndice E.

Resultados

Después de evaluar algunas instancias de 10,11,12,13,14,15 y 16 vértices (que están en la parte de entregables) con ambos algoritmos obtenemos los siguientes resultados:

instancia	Resultado		Tiempo		%
	Exacto	Heurística	Exacto	Heurística	
1	111.171	93.915	0.004	0.004	84.48
2	208.664	200.941	0.008	0.012	96.3
3	75.934	56.884	0.008	0.02	74.91
4	269.113	224.269	0.008	0.016	83.34
5	199.935	157.796	0.012	0.012	78.92
		Promedio:	0.008	0.0128	83.59

Tabla 1: Evaluación de las instancias de 10 vértices

instancia	Resultado		Tiempo		%
	Exacto	Heurística	Exacto	Heurística	
1	150.71	111.821	0.008	0.016	74.20
2	85.602	83.023	0.008	0.008	96.99
3	116.883	94.342	0.012	0.012	80.71
4	65.515	65.515	0.008	0.02	100.00
5	111.188	75.81	0.016	0.008	68.18
		Promedio:	0.0104	0.0128	84.02

Tabla 2: Evaluación de las instancias de 11 vértices

instancia	Resultado		Tiempo		%
	Exacto	Heurística	Exacto	Heurística	
1	219.188	174.735	0.016	0.02	79.72
2	360.551	352.243	0.04	0.024	97.70
3	126.706	99.766	0.016	0.016	78.74
4	199.915	190.608	0.004	0.008	95.34
5	202.405	180.102	0.028	0.02	88.98
		Promedio:	0.0208	0.0176	88.10

Tabla 3: Evaluación de las instancias de 12 vértices

instancia	Resultado		Tiempo		%
	Exacto	Heurística	Exacto	Heurística	
1	360.619	346.145	0.06	0.012	95.99
2	295.065	263.353	0.036	0.008	89.25
3	91.512	88.742	0.044	0.008	96.97
4	148.002	124.248	0.056	0.008	83.95
5	265.47	189.471	0.02	0.02	71.37
		Promedio:	0.0432	0.0112	87.51

Tabla 4: Evaluación de las instancias de 13 vértices

instancia	Resultado		Tiempo (s)		%
	Exacto	Heurística	Exacto	Heurística	
1	130.509	130.509	0.064	0.004	100.00
2	133.472	117.12	0.064	0.012	87.75
3	165.891	120.467	0.064	0.028	72.62
4	152.518	144.962	0.064	0.004	95.05
5	163.08	112.799	0.068	0.012	69.17
		Promedio:	0.0648	0.012	84.92

Tabla 5: Evaluación de las instancias de 14 vértices

instancia	Resultado		Tiempo (s)		%
	Exacto	Heurística	Exacto	Heurística	
1	305.646	288.855	0.084	0.024	94.51
2	306.03	306.03	0.436	0.004	100.00
3	135.491	93.656	0.14	0.004	69.12
4	155.381	128.495	0.144	0.02	82.70
5	324.321	323.256	0.124	0.016	99.67
		Promedio:	0.1856	0.0136	89.20

Tabla 6: Evaluación de las instancias de 15 vértices

instancia	Resultado		Tiempo (s)		%
	Exacto	Heurística	Exacto	Heurística	
1	264.127	254.168	13.412	0.016	96.23
2	284.113	253.101	3.996	0.008	89.08
3	309.148	297.815	6.576	0.02	96.33
4	145.451	129.345	10.62	0.02	88.93
5	338.452	221.341	6.304	0.016	65.40
		Promedio:	8.1816	0.016	87.19

Tabla 7: Evaluación de las instancias de 16 vértices

Vértices	Tiempo promedio	
	Exacto	Heurística
10	0.008	0.0128
11	0.0104	0.0128
12	0.0208	0.0176
13	0.0208	0.0176
14	0.0432	0.0112
15	0.1856	0.0136
16	8.1816	0.016

Tabla 8: Promedio del tiempo que tardaron ambos algoritmos

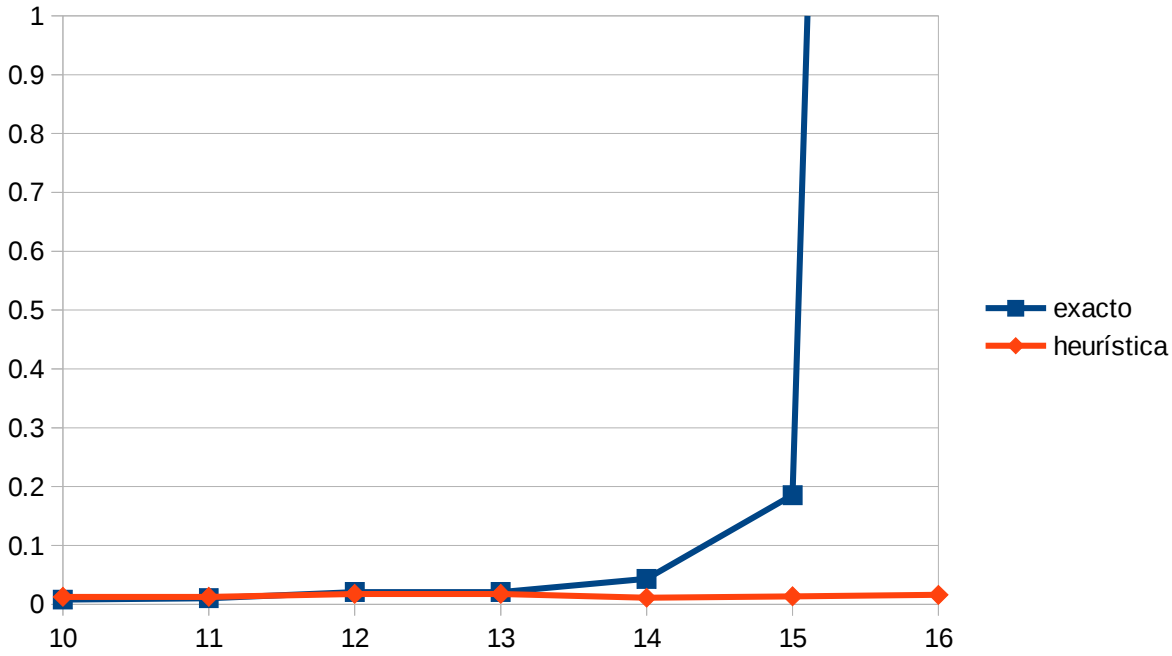


Figura 8: Gráfica del promedio de los tiempos

La salidas correspondientes a la peor instancia para nuestra heurística se pueden ver en el Apéndice F.

Conclusiones

Se diseñaron e implementaron en C el algoritmo generador de instacias, que también generaba el modelo para el algoritmo exacto y la heurística.

Para el algoritmo exacto se usó Gurobi, que es una de las mejores herramientas de software para resolver modelos de programación lineal entera mixta. Hay que mandarle un archivo con la función objetivo, las restricciones y los tipos de variables, archivo que crea el módulo generador. Esto solo nos da el valor óptimo de la función objetivo así que le pedimos guardar el resultado en otro archivo que es la solución del algoritmo exacto.

Con Gurobi se pudieron resolver casos hasta con 16 vértices en promedios de tiempo como los que se pueden observar en la Tabla 8. Esta aplicación requiere en la vida real una respuesta en menos de algunos minutos para 200 vértces.

Ejecutamos nuestra heurística con diversos casos de prueba así como con el algoritmo exacto y observamos que en el caso de la heurística el tiempo parecía ser constante mientras que en el algoritmo exacto parecía crecer exponencialmente como se muestra en la Figura 8. Las soluciones que generó la heurística se acercaban bastante a la solución óptima encontrada observando que superaban el 80% del optimo y alcanzando casi el 90% como se aprecia en las Tablas 1–7.

Por lo anterior consideramos que nuestra heurística podría dar buenos resultados en la práctica, en unos pocos minutos incluso para casos con 200 vértices.

Apéndices

A. Código para la generación de números aleatorios.

A.1 Números aleatorios entre m y n.

```
int rRand(int m, int n){
      return m + rand() % (n + 1 - m);
}
```

A.2 Números aleatorios con distribución exponencial con media de 30.

```
double randExp(){
  double l=1.0/30;
      double n=(rand()+1.0)/(RAND_MAX+2.0);
      return 1-log(n)/l;
}
```

B. Código para la generación de la instancia.

B.1 Estructura usada para representar los puntos.

```
typedef struct{
      int x,y;
      double p;
}pt;
pt pts[N];
```

B.2 Asignación de los valores de los puntos.

```
      for(int i=0;i<n;i++){
            pts[i].x=rRand(li,ls);
            pts[i].y=rRand(li,ls);
            pts[i].p=(randExp());
            v[i]=0;
      }
      pts[0].p=0;
```

B.3 Generador del archivo de la instancia para la heurrística.

```
void generadorOpt() {

      archivo2= abrirArchivo("instancia_opt.op","w");

      fprintf(archivo2, "%i\n",N);

      for(int i=0;i<N;i++){
            fprintf(archivo2, "%i\n", pts[i].x);
            fprintf(archivo2, "%i\n", pts[i].y);
            fprintf(archivo2, "%lf\n", pts[i].p);
      }
      cerrarArchivo(archivo2);
}
```

B.4 Generador del archivo de la instancia para el algoritmo exacto.

```
void generadorGurobi(){
       int s=0,u=0,i,j;

      archivo= abrirArchivo("instancia_gurobi.lp","w");

  fprintf(archivo, "max\n");

  fprintf(archivo, "obj: ");

//suma de los premios
    for ( i = 1; i < N; i++) {
      fprintf(archivo, "%lf x_%i ",pts[i].p,i);
      if(i<N-1) fprintf(archivo, "+ ");
    }

    Manh();

//suma de los costos
  for ( i = 0; i < N; i++) {
    for ( j = i+1; j < N; j++) {
      double d = A[i][j];
        fprintf(archivo, "- %lf y_%i_%i ",d,i,j);
    }
  }
  fprintf(archivo, "\n");

  fprintf(archivo, "Subject To\n");
  fprintf(archivo, "x_0 = 1\n");
```

```
//PARA LLENAR EL VECTOR DE LAS ARISTAS
  int c=(N*(N-1))/2;
  int y[c][2],cont=0;
  for (i = 0; i < N; i++) {
    for (j = i; j < N; j++) {
      if (j!=i) {
            y[cont][0]=i;
            y[cont][1]=j;
            cont ++;
      }
    }
  }
//TODOS LOS QUE LLEGAN  Y SALEN DE X_i
for (i = 0; i < N; i++) {
  cont=0;
  for (j = 0; j < c; j++) {
    if(y[j][0]==i || y[j][1]==i){
      fprintf(archivo, "y_%i_%i ",y[j][0],y[j][1]);
      // fprintf(archivo, "+ ");
      if(cont < N-2) fprintf(archivo, "+ ");
      cont ++;
    }
  }
  fprintf(archivo, "- 2 x_%i = 0\n",i );
}
fprintf(archivo, "\n");

//RESTRICCIONES VARIAS
sConj(N-1,s,u);

//TIPOS DE VARIABLES
  fprintf(archivo, "Binary\n");

//LAS X (los vertices)
  for (int i = 0; i < N; i++) {
    fprintf(archivo, "x_%i ",i);
  }
//LAS Y (los arcos)
  for ( i = 0; i < N; i++) {
    for ( j = i; j < N; j++) {
      if (i!=j)
        fprintf(archivo, "y_%i_%i ",i,j);
    }
  }
//LAS Z y las W
for (i = 1; i < pow(2,N)-1; i++) {
  fprintf(archivo, "z_%i ",i );
  fprintf(archivo, "w_%i ",i );
}
  fprintf(archivo, "\nend");
  cerrarArchivo(archivo);
}
```

B.5 Código de la función que opera con los subconjuntos para el generador.

```
void sConj(int n, int s,int u) {
  if(n>=0){
    a[n]=1;
    sConj(n-1, 2*s+1,u+1);
    a[n]=0;
    sConj(n-1, 2*s,u);
  }else{
    if(u!=0 && u!=N){
      imprimeRest1(s);
      imprimeRest2(s);
      imprimeRest3(s,u);
      imprimeRest4(s);
      imprimeRest5(s,u);
      imprimeRest6(s);
      fprintf(archivo,"\n");
    }
  }
}
```

B.6 Código de las funciones que imprimen las restricciones para cada subconjunto creado.

```
void imprimeRest1(int s) {
 int flag = 1;
  for (int i = 0; i < N; i++) {
   for (int j = i+1; j < N; j++) {
    if (a[i] + a[j] == 1) {
     if(flag==1){
      fprintf(archivo,"y_%i_%i ",i,j );
      flag = 0;
     }else {
      fprintf(archivo,"+ y_%i_%i ",i,j );
     }
    }
   }
  }
  fprintf(archivo,"- 2 w_%i >= 0\n",s);
}
```

```
void imprimeRest2(int s){
 int flag = 1;
 for (int i = 0; i < N; i++) {
  if(a[i]==1) {
   if(flag==1){
    fprintf(archivo,"x_%i ",i );
    flag = 0;
   }else
   fprintf(archivo,"+ x_%i ",i );
  }
 }
 fprintf(archivo,"- z_%i >= 0\n",s );
}

void imprimeRest3(int s, int u){
 int flag=1;
 for (int i = 0; i < N; i++) {
  if(a[i]==1) {
   if(flag==1){
    fprintf(archivo,"x_%i ",i );
    flag = 0;
   }else
   fprintf(archivo,"+ x_%i ",i );
  }
 }
 fprintf(archivo,"- %i z_%i <= 0\n",u,s );
}

void imprimeRest4(int s){
 int flag=1;
 int sc=pow(2,N)-1-s;
 for (int i = 0; i < N; i++) {
  if(a[i]==0) {
   if(flag==1){
    fprintf(archivo,"x_%i ",i );
    flag = 0;
   }else
   fprintf(archivo,"+ x_%i ",i );
  }
 }
 fprintf(archivo,"- z_%i >= 0\n",sc );
}
```

```
void imprimeRest5(int s, int u){
 int flag=1;
 int sc=pow(2,N)-1-s,
 uc= N - u;
 for (int i = 0; i < N; i++) {
  if(a[i]==0) {
   if(flag==1){
    fprintf(archivo,"x_%i ",i );
    flag = 0;
   }else
   fprintf(archivo,"+ x_%i ",i );
  }
 }
 fprintf(archivo,"- %i z_%i <= 0\n",uc,sc );
}

void imprimeRest6(int s) {
 int sc=pow(2,N)-1-s;
 fprintf(archivo,"z_%i + z_%i - 2 w_%i >= 0\n",s,sc,s);
 fprintf(archivo,"z_%i + z_%i - 2 w_%i <= 1\n",s,sc,s);
}
```

C. Código del algoritmo glotón.

C.1 Algoritmo que utiliza la matriz de distancias A para hacer el recorrido V.

```
void rec(int n, double A[][n], int V[],int p){

int i,j;
      double h=0; //se recorren aprox. 20km/hr y en un turno de 8hrs de
                                                                  recorren 160km
      for(i=0;i<n;i++) d[i]=0;
      V[0]=p; //va a iniciar en el punto p
      d[p]=1;

      for(i=1;i<n && h<160.0;i++){
            p=inMin(n,A[p]);
            V[i]=p;
            d[p]=1;
            //    ir de A a B      ir de B a 0      ir de A a 0
            h+=(A[V[i-1]][V[i]] + A[V[i]][V[0]] - A[V[i-1]][V[0]]);
      }
      if(i<n){
            V[i-1]=0;
            d[p]=0;
      }
}
```

C.2 Cálculo del costo

```
double cost(int n, double A[][n], int V[]){

	int i;
	double c=0,g,s,m=0;
	g=(1.0/9)*17; //suponiendo un rendimiento de 9km por litro y que cada
	                                                          litro cueste $17
	s=105.5; //salario del conductor p/día

	for(i=0;i<n;i++)
		c+=A[V[i]][V[(i+1)%n]];

	m=c*(2/3); //además cada 6000 km debería haber mantenimiento que
	                                                          cuesta $4000
	return (c*g)+m+s;
}
```

C.3 Cálculo de la ganancia

```
double gan(int n, pt p[],int v[]){
	double s=0;
	for (int i = 0; i < n; i++)
		s+=p[v[i]].p;
	return s;
}
```

D. Código de la heurística 2 – Opt.

D.1 Código de la función 2 – Opt

```
int opt2(int n, double A[][n]) {
            int x;
            double cost,costn;
            int flag=1,ret=0;
while (flag==1) {
flag=0;
      for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                  if(i!=j && i+1!=j && i!=j+1){
                        cost=A[v[i]][v[(i+1)%n]]+A[v[j]][v[(j+1)%n]];
                        costn=A[v[i]][v[j]]+A[v[(i+1)%n]][v[(j+1)%n]];
                        if (costn<cost) {
                              nRut(n,i,j+1);
                              flag=1;
                              ret=1;
                         }
                        }
                  }
            }
      }
      return ret;
}
```

D.2 Código de la función que genera la nueva ruta del 2 – Opt

```
int indPosCero(){
      for (int i = 0; i < n; i++) {
            if(v[i]==0 && v[(i+1)%n]!=0) return i;
      }
}

void nRut(int n,int x, int y) {
//va a intercambiar en la ruta v de entre x a y
  int a,f,g=x+1,h=1;
  if (x<y) f=(y-x)/2;
  else f=((n-1)-(x-y))/2;
  for (int z = 0; z < f; z++) {
    a=v[g%n];
            v[g%n]=v[((y+n)-h)%n];
    v[((y+n)-h)%n]=a;
    g++;
    h++;
  }
```

```
	int ce=indPosCero(),aux[n];
	for (int i = 0; i < n; i++)		aux[i]=v[i];

	for (int i = 0; i < n; i++) {
		v[i]=aux[ce%n];
		ce++;
	}
}
```

E. Código de la heurística 1 – Opt.

E.1 Código de la función 1 – Opt

```
int opt1(char c){
	int ret=0;
	double c1,c2,g=(1.0/9)*17,m=0,a=0;
	for (int i = 1; i < longitud(v); i++){
		if (d[i]==1) {
			a=A[v[(i-1)%n]][v[(i)%n]]+A[v[(i)%n]][v[(i+1)%n]];
				m=a*(2.0/3.0);
				c1=(a*g)+m - pts[v[i]].p;
				a=A[v[(i-1)%n]][v[(i+1)%n]];
				m=a*(2.0/3.0);
				c2=(a*g)+m;

				if (c2<c1) {
					d[i]=0;
					nRut1(i,v);
					ret=1;
				}
			}
	}
	for (int i = 1; i < longitud(v); i++){
		if (d[i]==0) {
					ret=o1(i,c);
			}
	}
	return ret;
}
```

E.2 Funciones auxiliares para la generación de la nueva ruta del 1 – Opt

```
void nRut1(int x, int a[]){
      int i;
      for (i = x; i < n ; i++) a[i]=a[(i+1)%n];
}

void nRut2(int x,int y, int a[]){
      int i;
      for (i = n-1; i > x ; i--) a[i]=a[(i-1)%n];
  a[(i+1)%n]=y;
}

int longitud(int v[]){
      for (int i = 0; i < n; i++)
            if(v[(i+1)%n]==0) return i+1;
}

int o1(int i,char c){
      int aux[n],j,c1,c2;
      for (j = 0; j < n; j++)    aux[j]=v[j];
      c1=gan(n,pts,v) - cost(n, A, v);
      for (j = 0; j < longitud(v); j++) {
            nRut2(j,i,aux);
            c2=gan(n,pts,aux) - cost(n, A, aux);
            if (c2>c1 && dist(n,A,aux)<=160) {
                  // for (int z = 0; z < n; z++) v[z]= aux[z];
                  d[i]=1;
                  nRut2(j,i,v);
                  return 1;
            }
            nRut1(j+1,aux);
      }
      return 0;
}
```

F. Peores casos

En las siguientes páginas observamos el peor de los casos de cada categoría de las instancias.

F.1 10 vértices

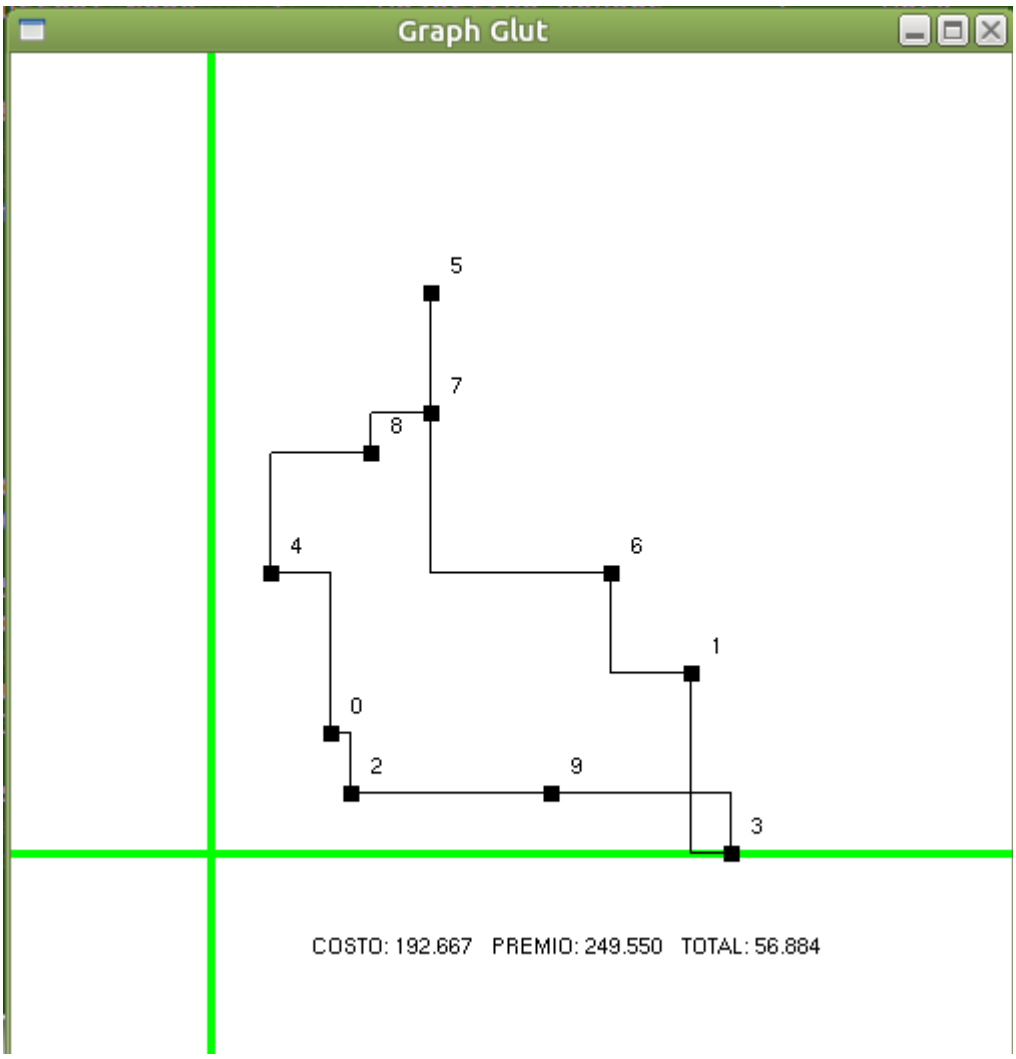


Ilustración 1: Peor caso observado en las instancias con 10 vértices

F.1 11 vértices

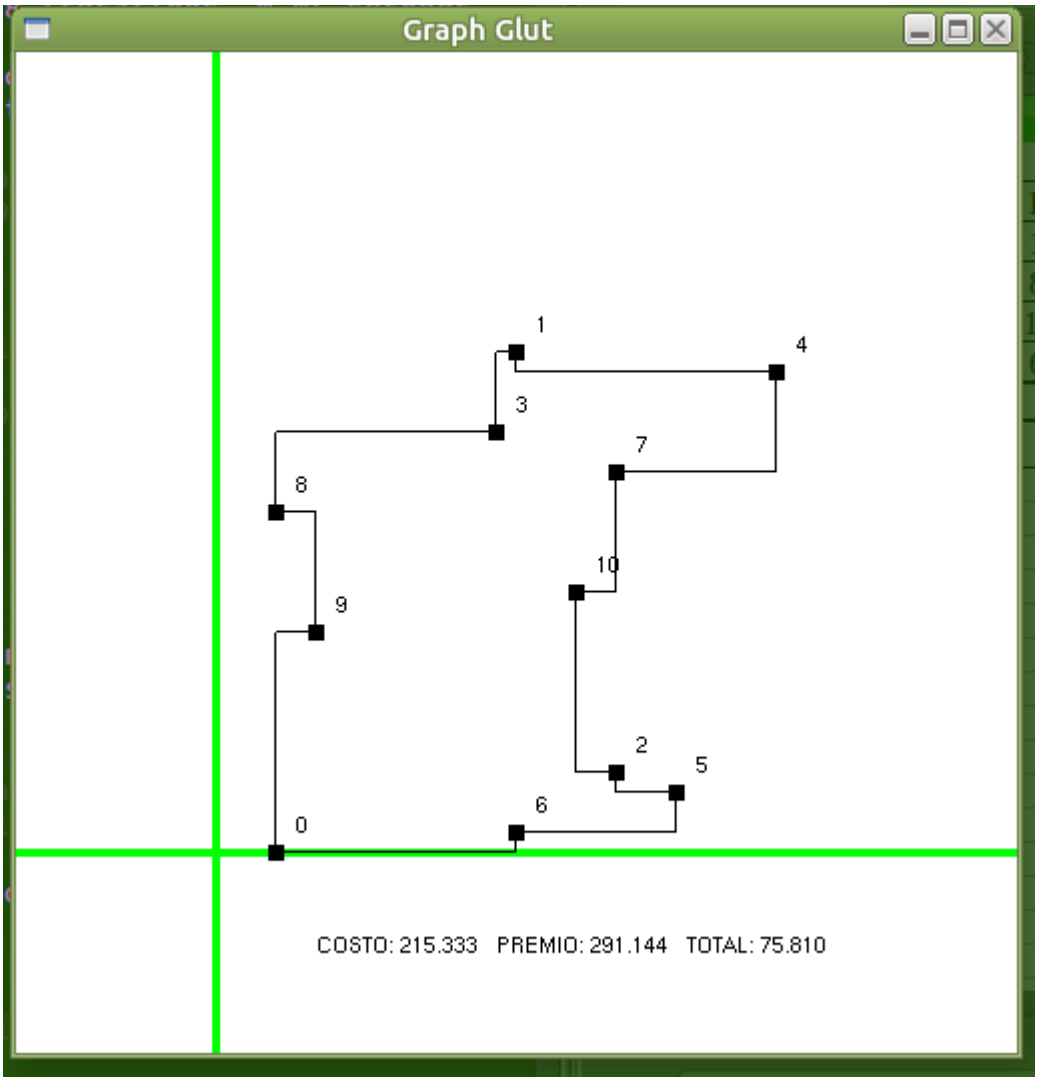


Ilustración 2: Peor caso observado en las instancias con 11 vértices

F.1 12 vértices

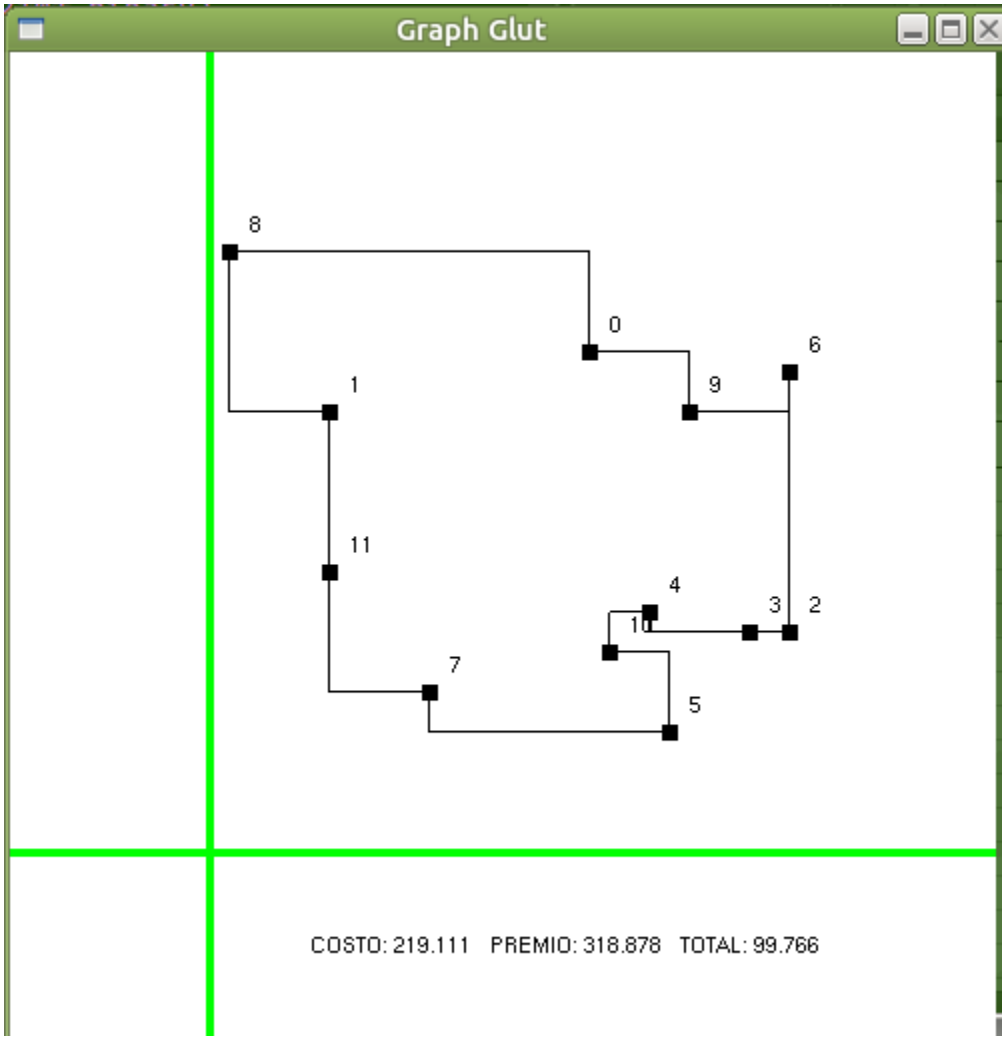


Ilustración 3: Peor caso observado en las instancias con 12 vértices

F.1 13 vértices

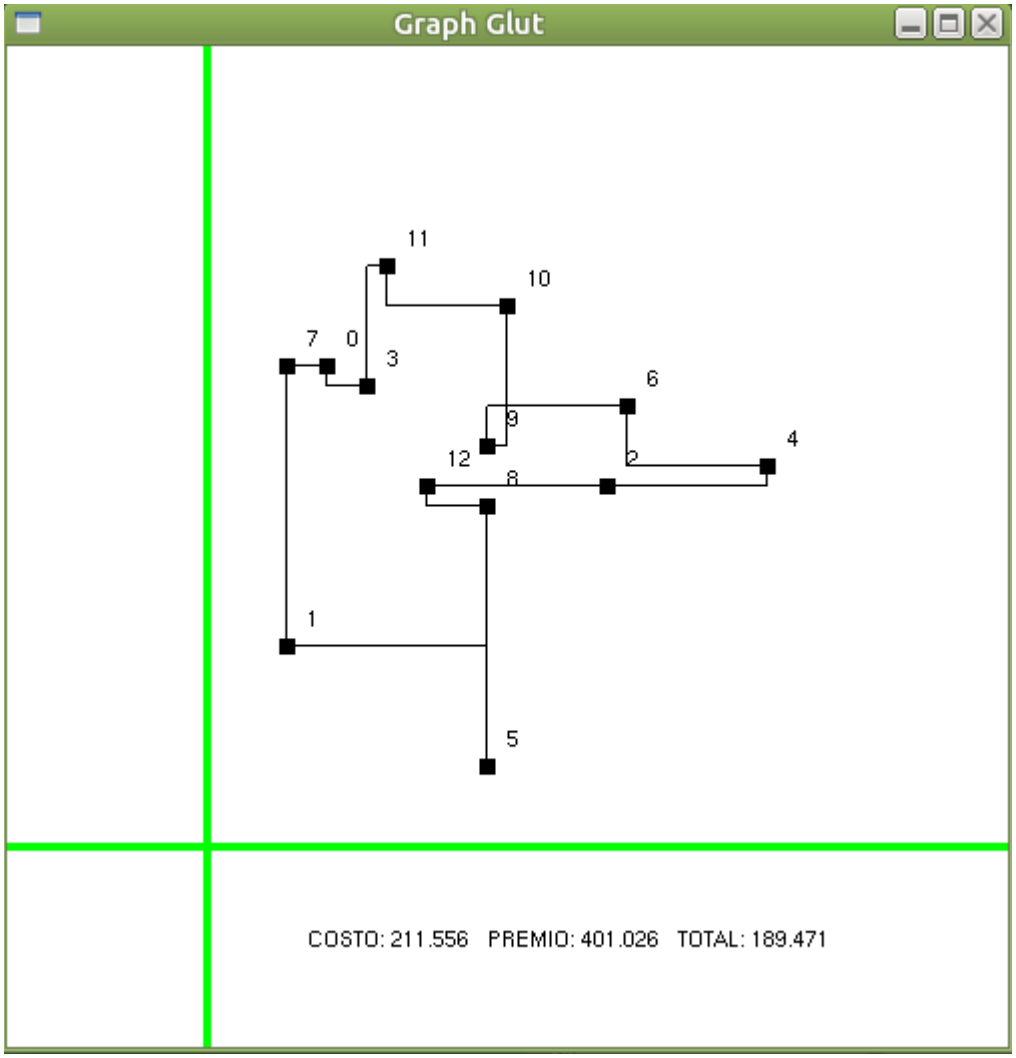


Ilustración 4: Peor caso observado en las instancias con 13 vértices

F.1 14 vértices

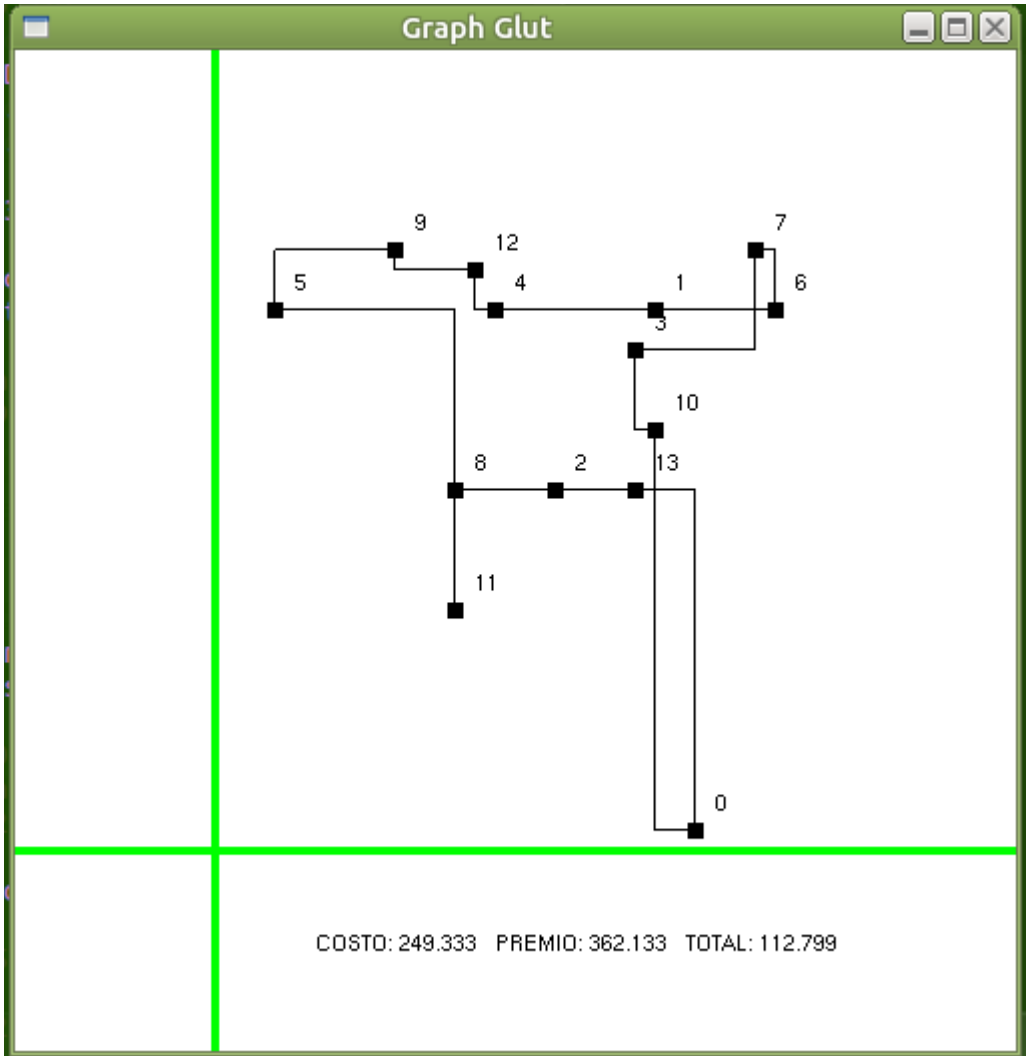


Ilustración 5: Peor caso observado en las instancias con 14 vértices

F.1 15 vértices

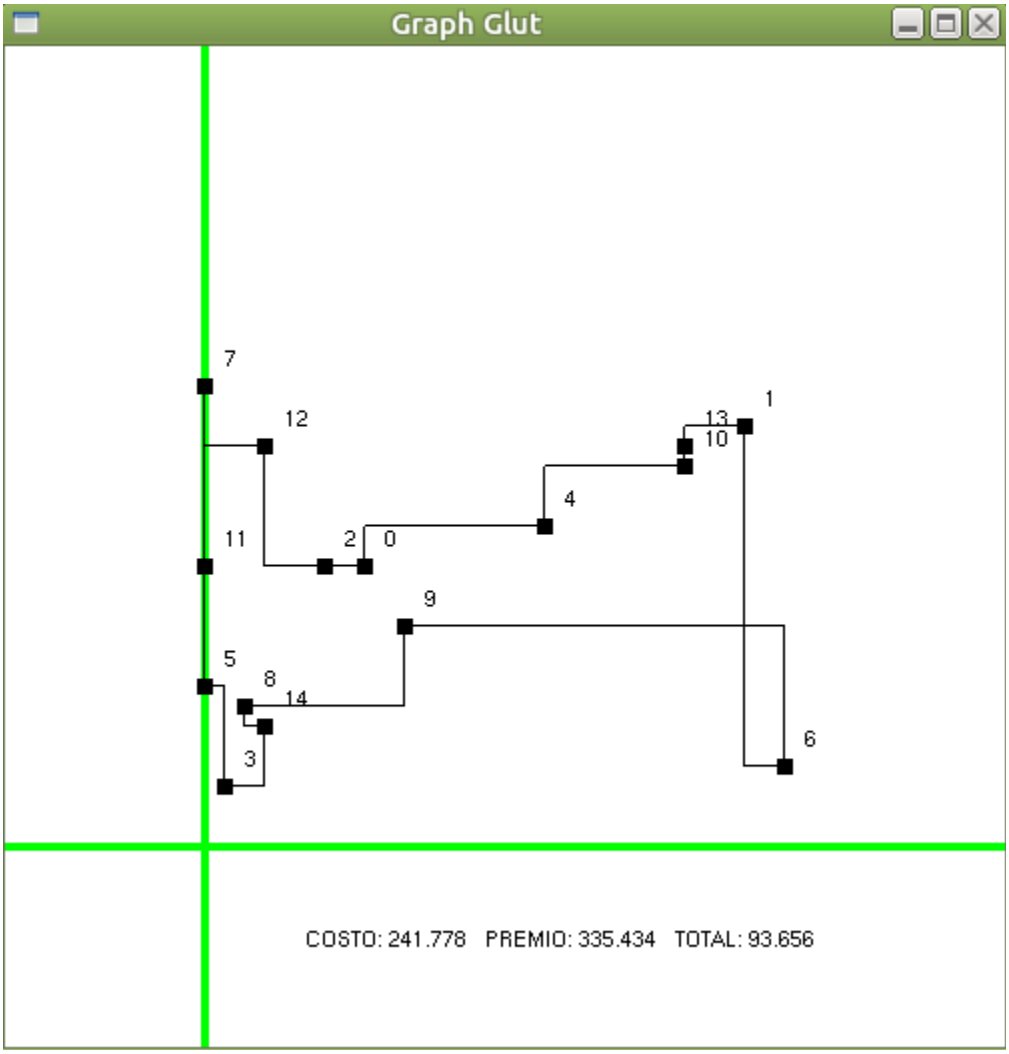


Ilustración 6: Peor caso observado en las instancias con 15 vértices

F.1 16 vértices

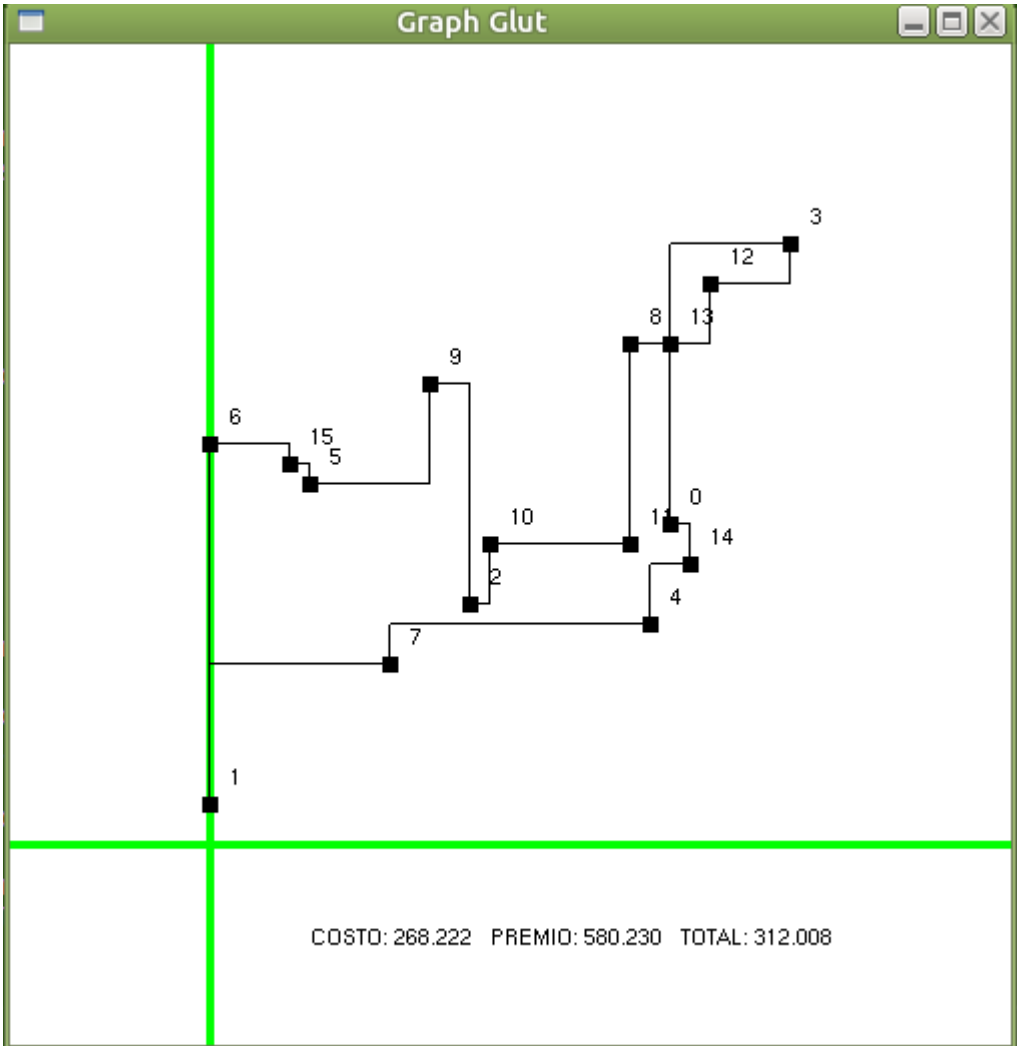


Ilustración 7: Peor caso observado en las instancias con 16 vértices

Bibliografía

[1] R. Zambrano Gerbacio, "Implementación de un algoritmo de aproximación para el problema del cartero con restricciones en los arcos". Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, 2013.

[2] A. Tapia de la Rosa, "Tres algoritmos para desplazar un robot en el plano en presencia de obstáculos". Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, 2013.

[3] V. M. Hernández Muñoz, "Aplicación para dispositivos con Android que encuentre la mejor ruta entre dos estaciones del Metro". Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, 2013.

[4] G. Frederickson, M. Hecht and C. Kim, "Approximation Algorithms for Some Routing Problems", SIAM Journal on Computing, vol. 7, no. 2, pp. 178-193, 1978.

[5] E. Balas, "The prize collecting traveling salesman problem", Networks, vol. 19, no. 6, pp. 621-636, 1989.

[6] M. Fischetti and P. Toth, "An additive approach for the optimal solution of the prize-collecting traveling salesman problem", in Vehicle routing : methods and studies, A. Assad and B. Golden, Ed. Elsevier Science Publishers, B.V., 1988, pp. 319–343.

[7] "BAMX | Bancos de Alimentos de México", Bamx.org.mx, 2017. [Online]. Available: https://bamx.org.mx/. [Accessed: 01- Mar- 2017].

[8] "Gurobi Optimization, Inc", Gurobi Optimizer Reference Manual, 2016. [Online]. Available: http://www.gurobi.com/. [Accessed: 19- Jul- 2017].