Memoria de divide y vencerás

Rubén Morales Pérez Francisco Javier Morales Piqueras
Bruno Santindrian Manzanedo Ignacio de Loyola Barragan Lozano
Francisco Leopoldo Gallego Salido

4 de mayo de 2016

Índice

1. Ejercicio 2

1.1. Enunciado del problema

Minimizando el número de visitas al proveedor:

Un granjero necesita disponer siempre de un determinado fertilizante. La cantidad máxima que puede almacenar la consume en r días, y antes de que eso ocurra necesita acudir a una tienda del pueblo para abastecerse. El problema es que dicha tienda tiene un horario de apertura muy irregular (solo abre determinados días). El granjero conoce los días en que abre la tienda, y desea minimizar el número de desplazamientos al pueblo para abastecerse.

- Diseñar un algoritmo greedy que determine en qué días debe acudir al pueblo a comprar fertilizante durante un periodo de tiempo determinado (por ejemplo durante el siguiente mes).

- Demostrar que el algoritmo encuentra siempre la solución óptima.

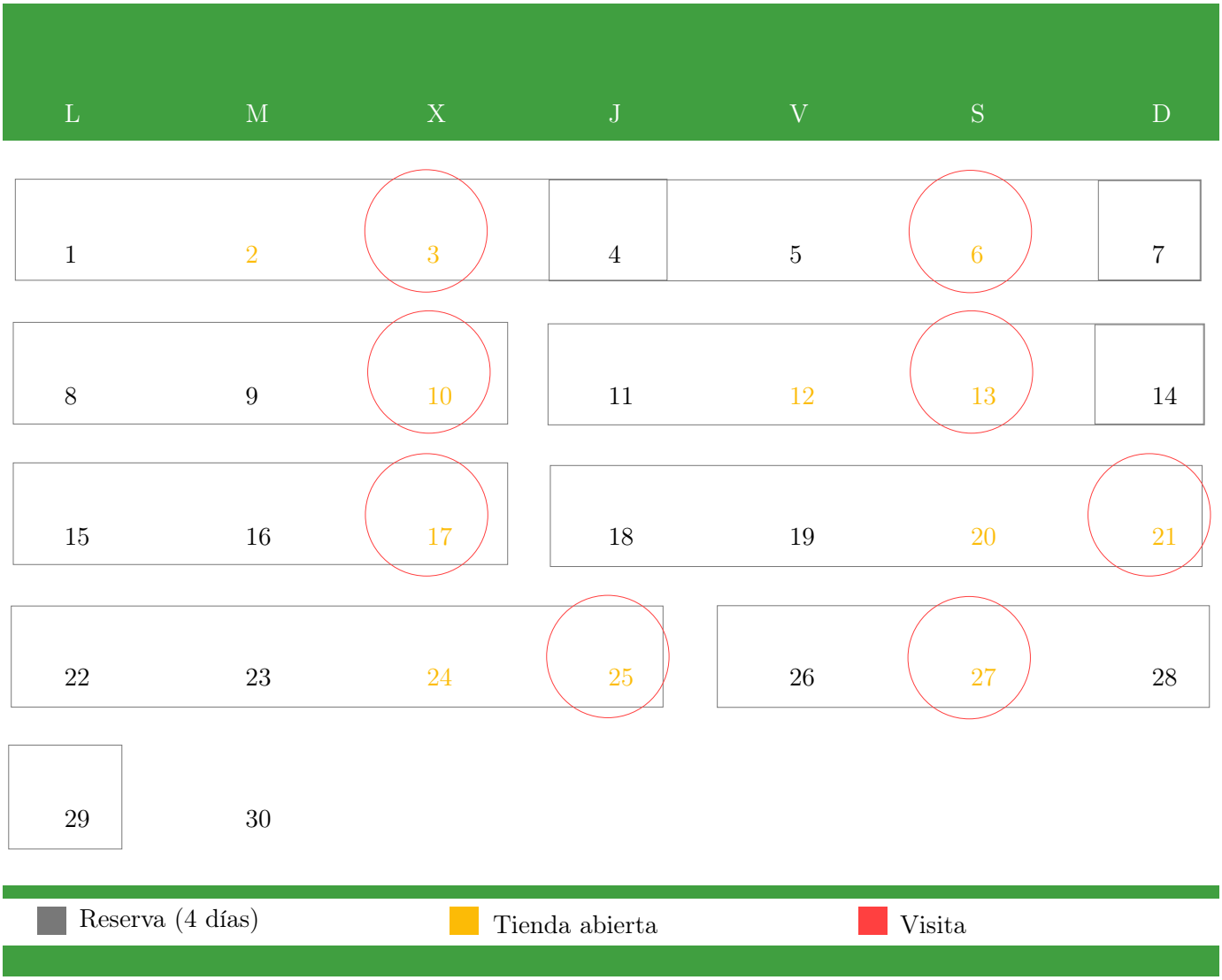
1.2. Diseño del algoritmo

El algoritmo, al basarse en la filosofía greedy, intenta escoger la mejor solución en cada fase. En nuestro caso, la mejor solución pasa por ir a comprar el día más lejano posible que cumpla estas dos condiciones.

1. Que diste menos de r días de la ultima reposición.

2. Que la tienda esté abierta.

Ejemplo:



1.2.1. Representación del algoritmo

Para programar el problema lo hemos modelado de la siguiente manera:

- Un vector de booleanos representa los días que abre la tienda.
  Por ejemplo [1, 0, 0, 1, 0, 1] significa que la tienda abre los días 1, 4 y 6.

- Una lista de enteros almacena los días que el granjero tiene que ir a comprar.

1.2.2. Implementación del algoritmo

La implementación se encuentra en *minimizar_visitas.cpp*.
La función clave es **minimize_visits()**, en la que, para buscar el día mas lejano posible, se accede

a la posición i+r del vector y se recorre hacia atrás hasta que se encuentra un día en el que la tienda esté abierta.

También hemos implementado otro algoritmo con el fin de hacer comparaciones, en el que el día de visita se elige aleatoriamente en el rango [i, i+r-1]. Esta implementación se encuentra en *visitas_ aleatorias.cpp*

1.3. Demostración de la optimalidad

Sean:

- **p** la duración del periodo (ej: 30 días)
- **r** la duración de la reserva
- **a** vector de tamaño p que representa los días que abre la tienda
  La tienda abre el día n si $a_n = 1$
  La tienda no abre el día n si $a_n = 0$
- **d** el vector que almacena los días de visita
  $d_{i+1} = d_i + k_i \ / \ k_i \in \{x \in \mathbb{N} : 0 < x < r \ y \ a_{d_i+x} = 1\}$

Formulamos dos ecuaciones:

$d^h_{i+1} = d^h_i + h_i$ donde $h_i = max\{x \in \mathbb{N} : 0 < x < r \ y \ a_{d_i+x} = 1\}$
Podemos definir $n \in \mathbb{N} \ / \ d^h_n = min\{d^h_i : d^h_i + r > p\}$ como el índice del último día que necesitamos ir a comprar.

$d^t_{i+1} = d^t_i + t_i$ donde $t_i \in \{x \in \mathbb{N} : 0 < x < r \ y \ a_{d_i+x} = 1\} \ / \ t_i < h_i$ para algún $i < n$

d^h representa la aproximación greedy, mientras que d^t representa cualquier otro algoritmo.

Por las definiciones anteriores sabemos que $\exists m \in \mathbb{N} \ / \ d^t_i \leq d^h_i \ \forall i, \ m \leq i < n \Rightarrow d^t_n \leq d^h_n$
Distinguimos dos casos:

1. Si $p - r < d^t_n \leq d^h_n$ las dos opciones son igual de buenas
2. Si $d^t_n < p - r < d^h_n$ la opción greedy es mejor

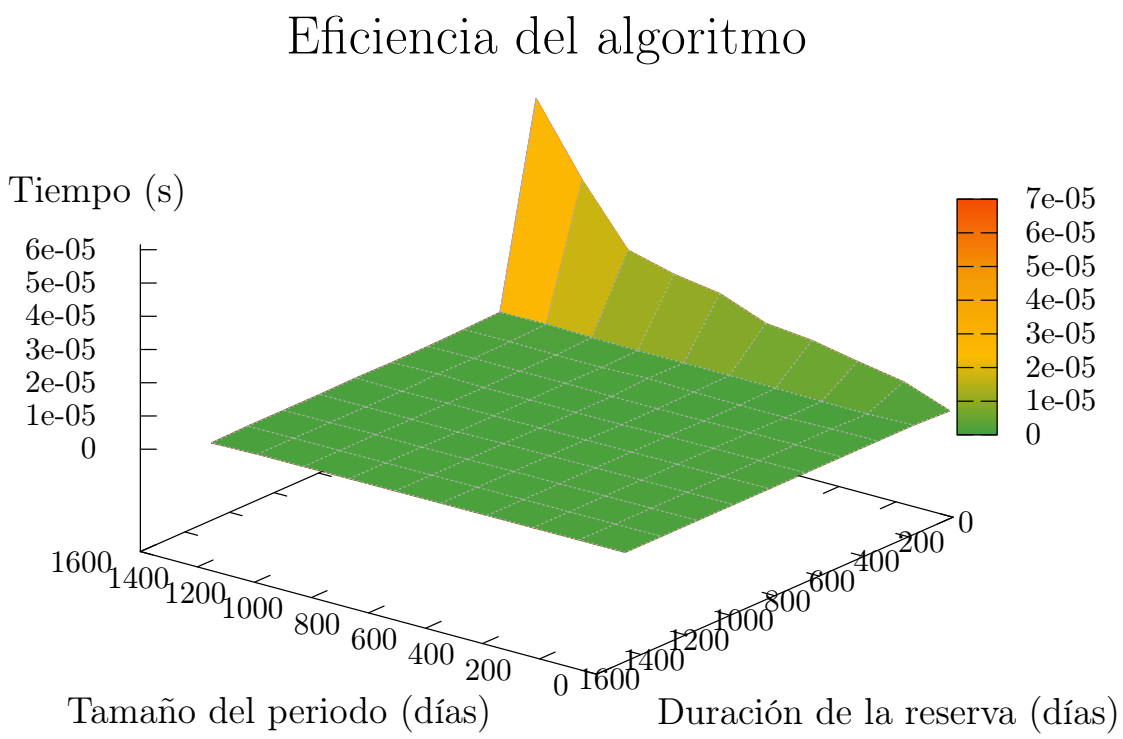
De aquí deducimos que utilizar el enfoque greedy en este caso siempre nos da el mejor resultado.

1.4. Resultados empíricos

1.4.1. Eficiencia

Aunque en esta ocasión la eficiencia no es especialmente importante, está claro que el tiempo aumenta linealmente en relación al tamaño del periodo de tiempo O(p).

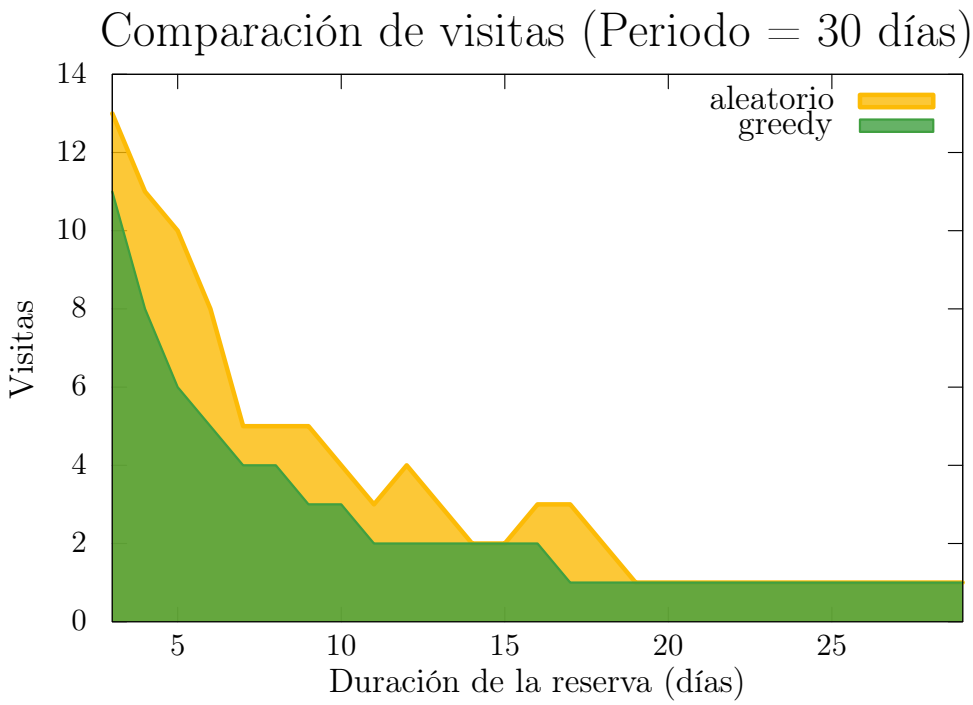
Además también aumenta al disminuir la duración de la reserva, tardando más con un periodo grande y una reserva pequeña.



1.4.2. Comparación

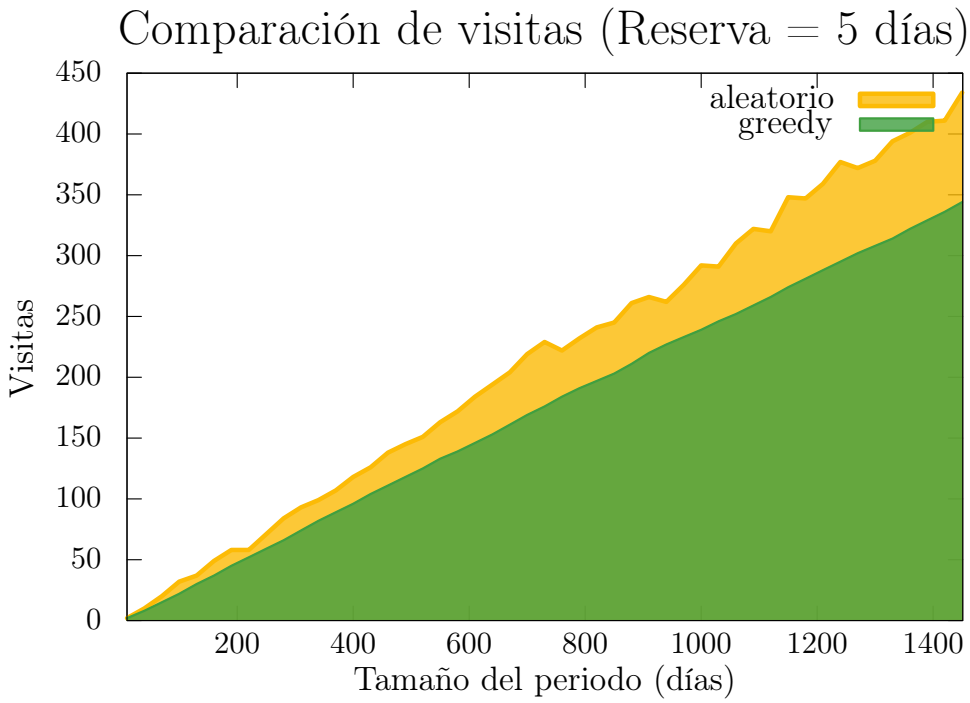
Es más interesante ver como se comporta el algoritmo respecto al número de visitas. Para esto lo compararemos con otro algoritmo que escoge los días de forma aleatoria.

Primero fijaremos el periodo de tiempo en 30 días y veremos como se comporta al variar la duración de la reserva.



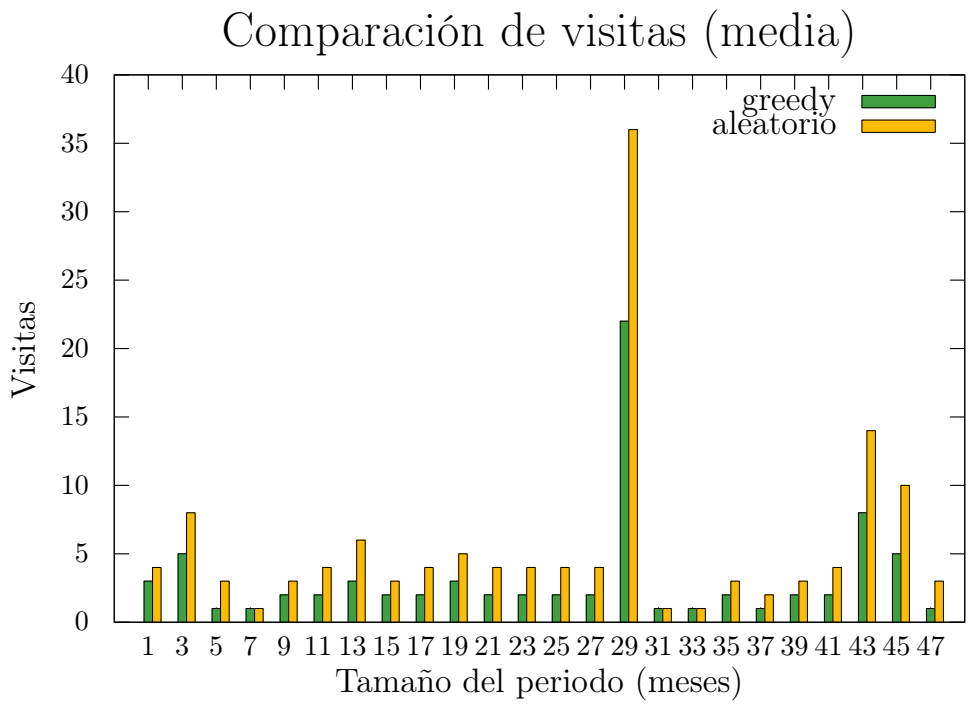
Como ya hemos probado el algoritmo greedy obtiene la mejor solución, por lo que siempre se encuentra por debajo del otro algoritmo. Además genera una gráfica descendiente, mientras que el aleatorio presenta máximos y mínimos locales.

Ahora fijamos la duración de la reserva en 5 días y variamos el tamaño del periodo. De esta manera obtenemos una gráfica creciente.



Las conclusiones son parecidas a las de la gráfica anterior. Cabe mencionar que aunque parece que las visitas aumentan de forma lineal en el algoritmo greedy, no se puede asegurar, ya que la distribución de días en los que abre la tienda es aleatoria.

También es interesante observar que ocurre de forma media. Para ello calculamos el número medio de visitas con una reserva aleatoria y en un periodo determinado.



Podemos observar dos conclusiones muy interesantes.

Primero, parece que hay una proporción entre el número de visitas medio del greedy y del aleatorio, aproximadamente $\frac{2}{3}$.

Segundo, el tamaño del periodo no influye en el número de visitas, ya que lo realmente importante es la proporción entre el periodo y la reserva y al ser esta última la media entre 1 y p las visitas se estabilizan (salvo excepciones de la probabilidad).

2. Viajante de comercio

2.1. Enunciado

En su formulación más sencilla, el problema del viajante de comercio (TSP, por Traveling Salesman Problem) se define como sigue: dado un conjunto de ciudades y una matriz con las distancias entre todas ellas, un viajante debe recorrer todas las ciudades exactamente una vez, regresando al punto de partida, de forma tal que la distancia recorrida sea mínima. Mas formalmente, dado un grafo G, conexo y ponderado, se trata de hallar el ciclo hamiltoniano de mínimo peso de ese grafo.
Una solución para TSP es una permutación del conjunto de ciudades que indica el orden en que se deben recorrer. Para el cálculo de la longitud del ciclo no debemos olvidar sumar la distancia que existe entre la última ciudad y la primera (hay que cerrar el ciclo).
Por su interés teórico y práctico, existe una variedad muy amplia de algoritmos para abordar la solución del TSP y sus variantes (siendo un problema NP-Completo, el diseño y aplicación de algoritmos exactos para su resolución no es factible en problemas de cierto tamaño). Nos centraremos en una serie de algoritmos aproximados de tipo greedy y evaluaremos su rendimiento en un conjunto de instancias del TSP. Para el diseño de estos algoritmos, utilizaremos dos enfoques diferentes: a) estrategias basadas en alguna noción de cercanía, y b) estrategias de inserción.
En el primer caso emplearemos la heurística del vecino más cercano, cuyo funcionamiento es extremadamente simple: dada una ciudad inicial v_0, se agrega como ciudad siguiente aquella v_i (no incluída en el circuito) que se encuentre más cercana a v_0. El procedimiento se repite hasta que todas las ciudades se hayan visitado.
En las estrategias de inserción, la idea es comenzar con un recorrido parcial, que incluya algunas de las ciudades, y luego extender este recorrido insertando las ciudades restantes mediante algún criterio de tipo greedy. Para poder implementar este tipo de estrategia, deben definirse tres elementos:
1. Cómo se construye el recorrido parcial inicial.
2. Cuál es el nodo siguiente a insertar en el recorrido parcial.
3. Dónde se inserta el nodo seleccionado.
El recorrido inicial se puede construir a partir de las tres ciudades que formen un triángulo lo más grande posible: por ejemplo, eligiendo la ciudad que está más al Este, la que está más al Oeste, y la que está más al norte.
Cuando se haya seleccionado una ciudad, esta se ubicará en el punto del circuito que provoque el menor incremento de su longitud total. Es decir, hemos que comprobar, para cada posible posición, la longitud del circuito resultante y quedarnos con la mejor alternativa.
Por último, para decidir cuál es la ciudad que añadiremos a nuestro circuito, podemos aplicar el siguiente criterio, denominado inserción más económica: de entre todas las ciudades no visitadas, elegimos aquella que provoque el menor incremento en la longitud total del circuito. En otras palabras, cada ciudad debemos insertarla en cada una de las soluciones posibles y quedarnos con la ciudad (y posición) que nos permita obtener un circuito de menor longitud. Seleccionaremos aquella ciudad que nos proporcione el mínimo de los mínimos calculados para cada una de las ciudades.

2.2. Vecino más cercano

2.2.1. Explicación

Esta es una de las formas más sencillas de abordar el problema del viajante.
Primero escogeremos una ciudad de inicio. Después de ello calculamos las distancias de esa ciudad al resto y escogemos la más cercana. En el programa usado de ejemplo la clase grafo almacena los

puntos (sus coordenadas) y la matriz de adyacencia del grafo, por tanto no tendremos que calcular repetidas veces las distancias. Para el algoritmo hemos usado una clase Punto y otra clase Grafo.

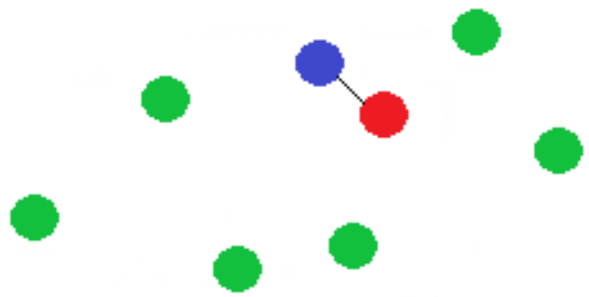


Figura 1: Eligiendo punto de inicio en azul

Estamos utilizando un enfoque greedy, calculando una solución óptima en cada paso local, con la experanza de llegar a una solución que se acerque al óptimo (o lo sea, en algunos casos). Repetimos el mismo paso con cada punto:

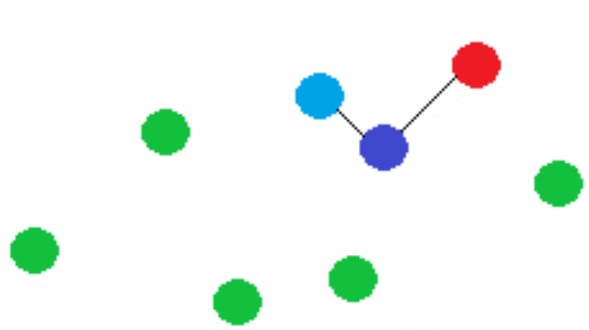


Figura 2: Tercer vértice

No debemos olvidar incluir el camino del último vértice al inicial, ya que forma parte del problema y hay que sumar también esa distancia.
El número de caminos (aristas) que debe recorrer coincide con el número de ciudades. Esta solución es muy rápida de implementar y nos da una solución cercana a la óptima, pero no tiene porque ser la óptima.

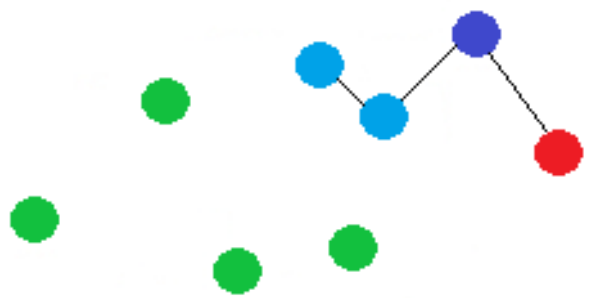


Figura 3: Cuarto vértice

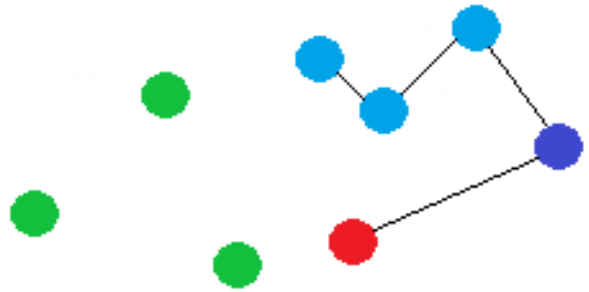


Figura 4: Quinto vértice

2.2.2. Eficiencia

Siendo n el número de ciudades por las que nuestro viajero debe pasar, denotamos cada una como $x_i = (i_a, i_b)$, $i \in \{0, 1, ..., n-1\}$ Definimos $\phi(x) : \mathbb{N} \to \mathbb{N}$ como la función que dado un vértice o ciudad x_i nos devuelve el índice de la ciudad más cercana, siendo su eficiencia lineal. Una vez iniciado el algoritmo nos interesará que solamente calcule el mínimo de las distancias con las ciudades que aún no ha recorrido.

$$\phi(x_i) = \{j \in \{0, 1, ..., n-1\} - \{i\} : min\{dist(x_i, x_j)\}\}$$

donde

$$dist(x_i, x_j) = \sqrt{(i_a - j_a)^2 + (i_b - j_b)^2}$$

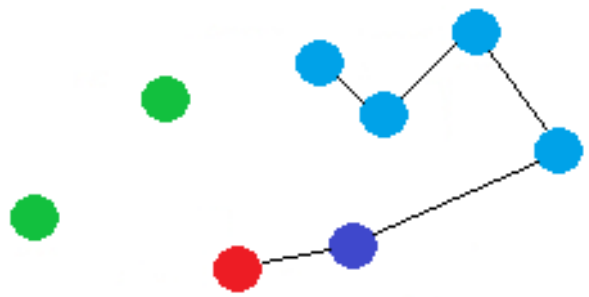


Figura 5: Sexto vértice

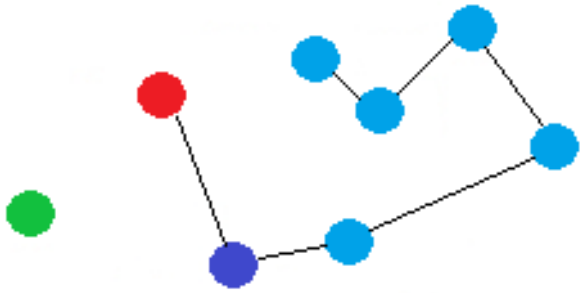


Figura 6: Séptimo vértice

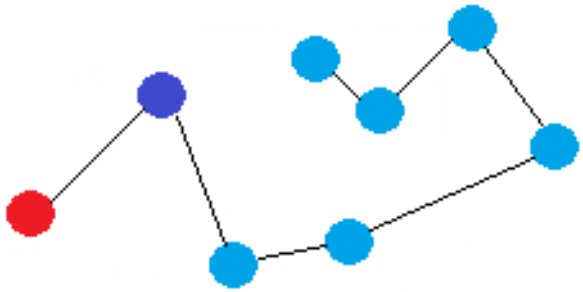


Figura 7: Octavo y último vértice

Tenemos que pasar por las n ciudades, por lo que podríamos pensar que su eficiencia es:

$$O(n) = \sum_{k=0}^{n-1} \phi(x_k)$$

Sin embargo el algoritmo que buscamos no es tan bueno como el cuadrático, ya que tenemos una restricción. En cada paso el cálculo del mínimo tiene que comprobar que la nueva ciudad no forme ciclo (pasaríamos dos veces por la misma ciudad). Por tanto definimos A como el conjunto de los índices de las ciudades que no han sido visitadas aún.

$$\phi'(x_i) = \{j \in A : min\{dist(x_i, x_j)\}\ y\ no\ forme\ ciclos\}$$

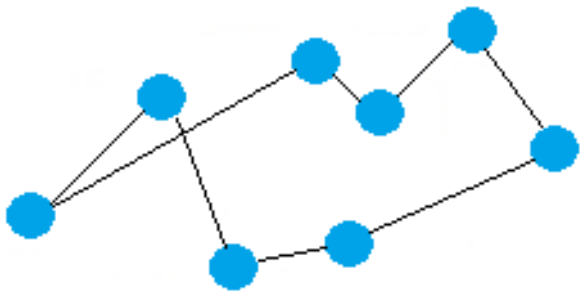


Figura 8: Recorrido vecino más cercano

Comprobar si una ciudad forma ciclos es lineal, por lo que tenemos que $\phi'(x_i)$ es cuadrático. La eficiencia del vecino más cercano queda definida como sigue:

$$\sum_{k=0}^{n-1} \phi'(x_k) \implies O(n) = n^3$$

2.3. Inserción

2.3.1. Explicación

El recorrido parcial inicial lo determinan los 3 puntos que estén más al Este, Oeste y Norte respectivamente. La única restricción es que los 3 no sean los mismos, algo que nunca pasa, pues 2 de ellos pueden ser iguales (un punto puede ser a la vez el punto más al Norte y al Este), pero 3 a la vez no.
EL nodo siguiente a insertar en el recorrido parcial será un nodo arbitrario, de lo que nos encargaremos para hacer funcionar el greedy es de insertarlo entre los dos vértices que más convengan, minimizar la distancia del circuito resultante.
Pasaremos por todos los vértices que nos quedan cada vez, y sobre cada uno calculamos la posición que más nos convenga dentro del circuito, y luego escogemos el vértice que nos da mejor resultado.

2.3.2. Eficiencia

Tenemos que recorrer n vértices, y sobre cada uno de ellos aplicaremos una función $\phi(x)$ para cada vértice.

$$\sum_{i=1}^{n} \phi(x_i)$$

Dicha función $\phi(x)$ llama a "buscar punto"que es lineal, cada vez que buscas un punto llama n veces a "buscar posicion", que a su vez llama n veces a calcular longitud.
Por tanto $\phi(x)$ es cúbica, y la eficiencia que nos queda para el algoritmo de inserción es

$$O(n) = n^4$$

2.4. Arista más corta

2.4.1. Explicación

En este caso escogeremos una técnica greedy diferente. Para intentar mejorar las estrategias del vecino más cercano y la inserción de vértices nos centraremos en una estrategia que consiste en ir seleccionando las aristas de menor longitud.
Primero crearemos una clase Arista donde guardaremos los índices de los puntos dentro del grafo (índices de ambos vértices de la arista) y la distancia entre ellos, sacada de la matriz de adyacencia del grafo.
Definimos una relación de orden entre aristas de la siguiente forma: Una arista es *menor que* otra si la distancia entre sus puntos es menor que la distancia entre los puntos de la segunda arista.
Posteriormente ordenaremos todas las aristas en un vector, e iremos seleccionando hasta pasar por todos los vértices una única vez, comprobando en cada paso que no forme nuevos ciclos.

2.4.2. Comprobación de ciclo constante

Para desarrollar una forma constante para comprobar si una arista forma ciclos o no hemos usado un vector de listas de aristas $vector < list < Edges >>$.
Cada lista definirá un camino inconexo con el resto de listas del vector, de forma que su primer elemento y último sean los extremos del camino, pasando por los puntos intermedios. Al final del algoritmo nos quedará una única lista ordenada con la solución.
Para saber si una arista forma ciclo o no asignaremos un estado a cada vértice. Inicialmente los vértices tienen estado -1, significa que no han sido usados aún por el algoritmo (no están en ninguna lista).
Si se inserta una nueva arista, su primer vértice (inicio del camino) pasará al estado [1+10*posicion], siendo *posicion* el lugar que ocupa dentro del vector de listas (es una forma de diferenciar los diferentes caminos, necesaria). De esta forma si (estado1/10 == estado2/10) sabemos que los vértices están en el mismo camino.
El segundo vértice (último elemento) tendrá el estado [2+10*posicion]
Si hubiese vértices entre el principio y el final (cuando tenemos más de una arista) los vértices intermedios tendrían estado [3+10*posicion]
Estos casos nos permiten saber directamente al insertar una arista si forma ciclo.

2.4.3. Casos posibles

Denotando como *one* y *two* los códigos de los vértices de la arista que queremos insertar.
Caso 1(bien): Ninguno está. En este caso se añadiría una nueva lista con una sola arista que nos indicaría que se ha creado un nuevo camino, en lugar de extender uno anterior.
Código:

((one == -1) & & (two == -1))

Caso 2(bien): Está uno y el otro está libre.
Código:

((one %10 == 1) & & (two == -1) || (one %10 == 2) & & (two == -1)
(two %10 == 1) & & (one == -1) || (two %10 == 2) & & (one == -1))

Caso 3(bien): La arista une caminos, están los dos vértices, pero en caminos diferentes.
Meteremos los elementos de la segunda lista de aristas en la primera
Código:

(((one %10 == 1) & & (two %10 == 1) & & (one/10 != two/10)) ||
((one %10 == 2) && (two %10 == 2) & & (one/10 != two/10)) ||
((one %10 == 1) && (two %10 == 2) && (one/10 != two/10)) ||
((one %10 == 2) && (two %10 == 1) && (one/10 != two/10)))

Caso 4(mal): Está uno en mitad y otro libre
Código:

((one == -1 & & two %10 == 3) || (two == -1 & & one %10 == 3))

Caso 5(mal): Están los dos y forman ciclo
Código:

(((one %10 == 1) || (one %10 == 2)) & & ((two %10 == 1) || (two %10 == 2)) & & (one/10 == two/10))

2.4.4. Eficiencia

Siendo n el número de puntos del grafo, para ordenar las n^2 aristas usaremos el Heapsort, por tanto obtenemos una eficiencia de $n^2 log(n^2) \implies n^2 log(n)$.

En el peor caso recorreremos las n^2 aristas hasta encontrar las n aristas deseadas. Como el algoritmo para ver si una arista forma ciclo es constante, si forma ciclo pasará a la siguiente iteración, si no forma añadirá dicha arista.

2.4.5. Explicación gráfica

Figura 9: Eligiendo punto de inicio en azul

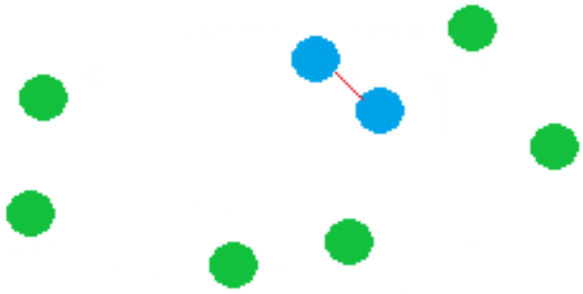


Figura 10: Tercer vértice

Debemos tener en cuenta que solamente podemos unir aristas que estén en dos caminos diferentes si sus puntos son extremos de caminos ya existentes. De esta forma aunque un vértice sea un extremo de un camino, si su otro extremo está en mitad de otro camino no nos vale, ya que llegaría un punto en el que pasaríamos dos veces por el mismo punto.

No debemos olvidar terminar el camino.

3. Bibliografia

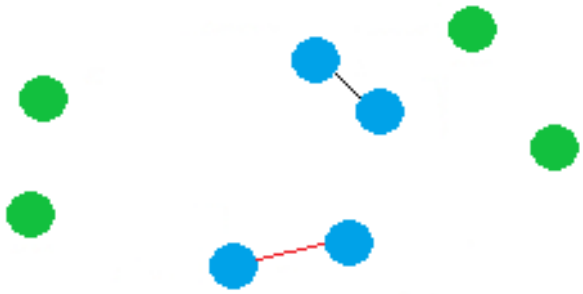


Figura 11: Cuarto vértice

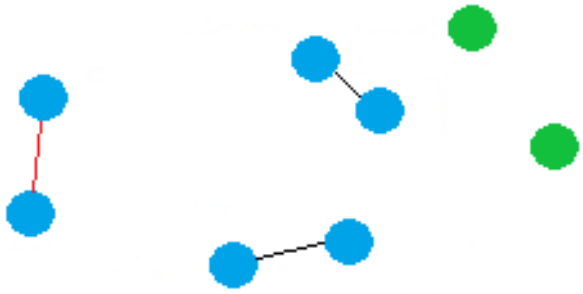


Figura 12: Quinto vértice

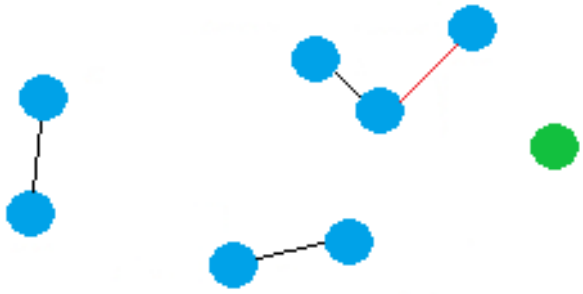


Figura 13: Sexto vértice

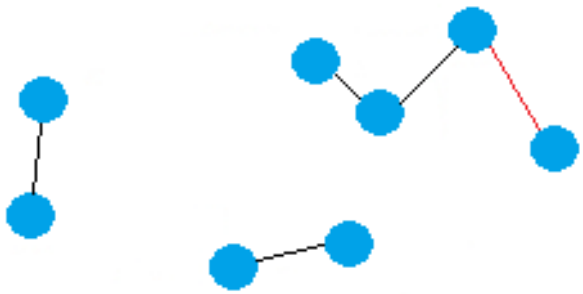


Figura 14: Séptimo vértice

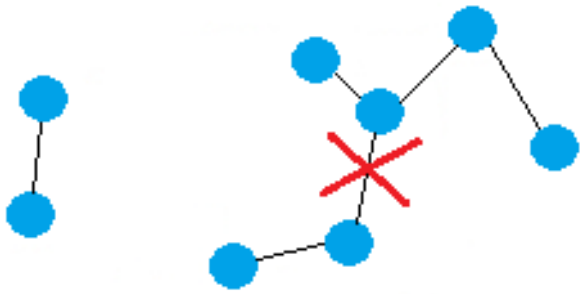


Figura 15: Octavo y último vértice

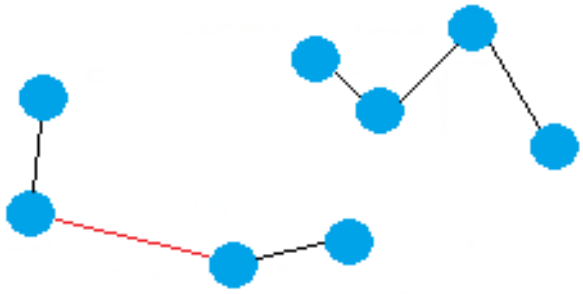


Figura 16: Octavo y último vértice

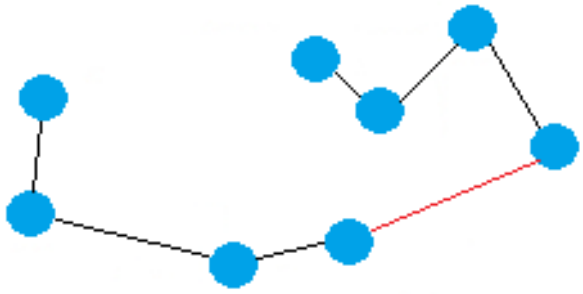


Figura 17: Octavo y último vértice

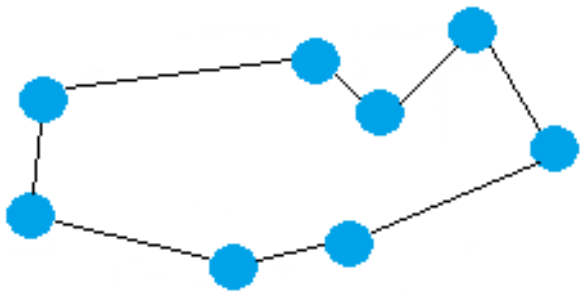


Figura 18: Recorrido vecino más cercano