El problema del agente viajero
Recocido Simulado

Seminario de Ciencias de la Computación B

Canek Peláez Valdés

Sánchez Correa Diego Sebastián

Índice general

Capítulo 1

Introducción

1.1. El problema

...la pregunta es encontrar, para un conjunto finito de puntos de los cuales se conocen las distancias entre cada par, el camino más corto entre estos puntos. Por supuesto, el problema es resuelto por un conjunto finito de intentos. ***Schrijver (2005)*** [1]

1.2. Heurística

[1] Alexander (Lex) Schrijver (4 de Mayo 1948, Ámsterdam) es un matemático y científico en computación holandés, profesor de matemáticas discretas y optimización en la Universidad de Ámsterdam

Capítulo 2

Diseño

El diseño siguió una estructura orientada a objetos. Contar con la información de las ciudades en una base de datos relacional, sugirió la implementación de un objeto **`database_loader`**. De la msima manera, el contar con soluciones compuestas por permutaciones de ciudades, hizo claro el uso de las clases **`city`** y **`path`** (ciudad y camino, respectivamente). De la misma manera, se ha planteado el modelado del problema y de la heurística como clases compuestas a partir las unidades más fundamentales y con atributos y comportamiento asociado.

2.1. La base de datos

El problema se planteó como una base de datos relacional cuyo contenido está compuesto de las ciudades (1092 en total) y de sus conexiones (que representan la gráfica del problema).

Es importante mencionar que la matriz de adyacencias represnetada por la tabla de conexiones no es cuadrada, sin embargo, el objeto **`database_loader`** la cargará a una matriz de enteros (**`int`** de C) haciéndola cuadrada.

La tabla de ciudades se carga a un arreglo de apuntadores a objetos de tipo **`city`**. Normalmente se usaría un puntero a un objeto de tipo **`city`**, pero el constructor de la clase, naturalmente, aloja los objetos en el heap, devolviendo, por lo tanto, un puntero a un objeto; por ello, el arreglo solo reúne los punteros de los objetos que ya se encuentran guardados dinámicamente; evitando, en conjunto, una posible ambigüedad al tratar con ambos tipos que, en esencia, no comparten características adicionales a ser un apuntador a una ciudad.

2.2. Las ciudades

El objeto **`city`** tiene como propiedades los campos incluidos en la base de datos, con excepción de la población. La clase abstrae el concepto de una ciudad, omitiendo una representación total a partir de atributos asociados (como lo hace la base de datos), sino involucrando en la composición un comportamiento particular.

Esta abstracción se da al decidir que una ciudad proporcione el comportamiento para calcular la distancia con otra ciudad en conjunto con su uso como unidad fundamental del problema.

Cabe destacar que la clase **`city`** también es la encargada de crear los arreglos de ciudades, resultado de una ”modularización”del diseño : ninguna clase conoce el tamaño de la estructura **`city`**, por lo tanto, ninguna clase (a excepción de city) es capaz de crear dinámicamente un arreglo de ciudades; esto podría evitarse si se incluyera la declaración de la estructura dentro del encabezado (header) de la clase, pero considero se trataría de una violación a la encapsulación de los datos.

2.3. Los caminos

Al tener la idea de una ciudad ya abstraída en una clase, era natural que los caminos consitieran de, al menos, un arreglo de ciudades. El primer diseño que realicé se trataba de una representación sin atributos, ni comportamiento; solo un arreglo de ciudades. Poco tiempo después escribí la clase **`path`**. Esta ahora no solo se trataba de un arreglo de ciudades, sino de una estructura con atributos y comportamiendo asociados: esto facilitó (desde un punto de vista del diseñño) la implementaciones de funciones como el intercambio de ciudades dentro de un camino, el cálculo del normalizador y la función de costo.

La creación de un camino implica el cálculo de la distancia máxima de esa instancia, así, como el normalizador y la suma de los costos de sus ciudades. Estos atributos se calculan a partir de operaciones lineales, por ello, son valores que incializo dentro del constructor de la clase y, si se quiere acceder a ellos una vez creado al objeto, la operación se reducirá a una consulta lineal del atributo deseado del objeto. Esta solución implicaba la disminución de la cantidad de invocaciones de funciones para crear una camino válido (dejó de ser necesario invocar la función, seguido de usar un setter del objeto para guardar el resultado), sin embargo, me impedía utilizar el mismo constructor para hacer copias de un objeto (usadas en la heurística) si pretendía mantener el tiempo de ejecución en un estado óptimo, por supuesto.

La implementación de la función copia de la clase **`path`**, entonces, no invoca al constructor de la clase, sino que accede a las propiedades del objeto parámetro y aquellas que eran computadas linealmente, se asociarían como copias al nuevo objeto.

2.4. TSP

El problema del agente viajero fue, como lo han sido todas las estructuras del proyecto, modelado siguiendo un diseño orientado a objetos, sin embargo, no sería un error afirmar que este no era su propósito inicial. La clase comenzó siendo un objeto central de ejecución que administraría la invocación y liberación de todos los objetos mencionados anteriormente, incluso, sería la encargada de ejecutar la heurística. Durante una refactorización, precisamente una que preparaba el terreno para adaptar la implementación de la huerística sobre el diseño del proyecto (y además, cuando decidí elegir crear una clase camino por sobre los arreglos de ciudades), esta clase dejó de ser

solo el paquete que tenía el ambiente global del proyecto.

La clase terminó siendo quien inicia la ejecución del problema, creando un objeto **`database_loader`** y se encarga de proveer la información contenida en este, además de liberarlo de la memoria dinámica. Como atributo, tiene un camino (el único creado en la ejecución del programa; con el constructor de la clase, al menos) que también será usado para la heurística. Por ello, una ejeución, no estará completa sin una invocación a la clase **`sa`**.

2.5. Recocido Simulado

Capítulo 3

Implementación

3.1. Hilos de ejecución

3.1.1. Generador de números aleatorios

Capítulo 4

Experimentación

Parámetro	*Valor*
Semilla	102
Temperatura	14
M	122000
L	1200
Épsilon	0.002
Phi	0.95

Capítulo 5

Conclusiones

5.1. Bibliografía