Práctica 1

Pedro José Piquero Plaza

16 de marzo de 2014

Universidad de Córdoba

Metaheurísticas

Índice

1. Descripción de los problemas

1.1. Max Min Diversity Problem

El problema Max Min Diversity Problem trata de escoger dentro de un determinado conjunto de individuos, un subconjunto de tamaño m, de manera que la diversidad entre cada uno de los individuos sea lo más grande posible.

También es posible tratar de obtener un subconjunto de manera que la diversidad entre cada uno de los individuos que lo integren sea mínima, pero en este caso, me he centrado es obtener el máximo.

1.2. Travelling Salesman Problem

El problema del viajante de comercio (Travelling salesman problem), trata de establecer una ruta entre una serie de ciudades que cumpla con las siguientes condiciones:

- Se debe de escoger todas las ciudades integrantes del problema.
- Se debe empezar y acabar en la misma ciudad.

De entre todas las posibles soluciones al problema se debe escoger aquella ruta que tenga menor coste.

1.3. CutWidth Problem

El problema de la minimización del ancho de corte (CutWidth problem) es un problema NP-duro (Gavril 1977) y consiste en encontrar un diseño lineal de un grafo no dirigido, de manera que se reduzca al máximo el numero de aristas consecutivas entre dos nodos.

1.4. P-hub problem

El problema del P-hub es un problema de ubicación de las instalaciones que pueden ser visto como un tipo de problema de diseño de red. Cada nodo, dentro de un determinado conjunto de nodos, envía y recibe algún tipo de tráfico hacia y desde los otros nodos. Los nodos concentradores deben ser elegidos entre estos nodos para actuar como puntos de conmutación para el tráfico. Los enlaces de red se colocan entre pares de concentradores de manera que los centros están completamente interconectados. Cada uno de los nodos restantes, a su vez, está conectado a uno de los concentradores.

El tráfico puede representar el tráfico de telecomunicaciones, transmisión de datos, los pasajeros de avión, paquetes express, etc.

2. Particularidades de la práctica

2.1. Lenguaje de programación

Para programar está practica he estado usando el lenguaje de programación Ruby, en lugar de C++. Esto significa que si las prácticas siguientes están relacionadas con ésta tendré que seguir programando en Ruby.

La pregunta entonces sería, ¿por qué Ruby? La respuesta es por sencillez. Ruby permite trabajar de una manera mucho más clara y más despreocupada que su contrapartida en C++.

Esto implica por un lado que la codificación se hace más rápida y por otro lado, también permite centrarse más en el problema y menos en cómo se van a representar los datos.

Por supuesto todo esto tiene un precio, y se paga en tiempo de ejecución. Para poner un ejemplo, el tiempo medio de ejecución de una instancia del problema Capacited P-Hub con una implementación de Ruby vainilla ha sido experimentalmente de algo más de 1 minuto 30 segundos. Seguramente se habrá dado cuenta de que el coste por tener ciertas comodidades es bastante alto.

Como es obvio que no se puede realizar una entrega con tiempos de ejecución tan altos, para programas tan relativamente simples (desde el punto de vista de que no se trata de optimizar una función) se han realizado las siguientes operaciones:

- Se ha revisado el diseño de las clases con tiempo de ejecución alto, tratando de realizar modelos que sean sencillos computacionalmente en lugar de modelos muy complicados.
- Sí aún el tiempo de ejecución era elevado, se ha analizado el problema con herramientas "profiler"para tratar de descubrir dónde se desaprovechaba la mayor parte del tiempo.
- Se han seleccionado los métodos que usaban más tiempo computacional.
- Se ha realizado la implementación de dichos metodos en el lenguaje C, usando la API que Ruby ofrece para ello.

De esta manera, se ha pasado, por ejemplo, en la ejecución del problema Capacited P-Hub de tener un tiempo de ejecución que superaba el minuto y treinta segundos a un tiempo se ejecución de alrededor de 1.59 segundos.

En el momento de escribir esta memoria, el 91.8 % del código se escribió en Ruby, un 5.7 % se escribió en C y el 2.4 % restante corresponde al script en bash proporcionado por el profesor.

2.2. Controlando los cambios

Es bueno tener un control sobre los cambios que se producen en el código, especialmente porque puede ocurrir que se produzcan cambios que estropeen el buen funcionamiento de un programa y no saber en qué momento se produjeron puede ser peor que un dolor de muelas.

Es por ello, que a lo largo de toda la práctica he estado (y estaré) usando la herramienta git como controlador de versiones.

La práctica además está subida a un repositorio que he creado en GitHub y puede ser consultada en cualquier momento por el profesor o cualquier otra persona. El enlace actual de la página es: `https://github.com/PracticasUCO/Practica1-Metaheuristica.git`

También puede descargarse el repositorio usando git con el comando:
git clone git@github.com:PracticasUCO/Practica1-Metaheuristica.git

2.3. Comprobando el trabajo

También puede ser útil saber si lo que uno está programando produce los resultados esperados, por ejemplo, si estoy haciendo una función que me devuelva la raíz cuadrada de dos números no viene nunca mal asegurarse de que la función hace lo que tiene que hacer.

Esto se puede hacer de muchas formas, la más común (por desgracia) es esperar a que el programa esté completo o casi completo y después introducir una serie de entradas para las cuales sabemos la salida que debe producir nuestro programa, y, esperar que todo haya salido bien.

En esta práctica he introducido una manera diferente de trabajar, conocido en inglés como *Test Driven Development* el ciclo de desarrollo se puede puede resumir como sigue:

- Se escribe una serie de test que compruebe una determinada funcionalidad que se quiere implementar. Inicialmente estos test fallarán porque la nueva funcionalidad aún no ha sido implementada (a no ser que el test escrito sea muy malo).

- Se escribe el código de la nueva funcionalidad. Este código no tiene por qué ser eficiente.
- Se vuelven a ejecutar los test. Si el resultado del test es correcto el programador puede estar seguro de que el código escrito anteriormente funciona como es debido.
- Se trata de optimizar el código que escribiste.
- Se repite el ciclo.

Para realizar esto, he estado usando el framework de Ruby *MiniTest*. Tenga en cuenta de que soy nuevo en esto, esto supone que probablemente me haya dejado casos sin probar y que haya mejores formas de realizar esto. Aunque esto sale de los objetivos de la asignatura, animo al profesor a darme orientación en estos temas si lo ve preciso.

2.4. Seguimiento de la práctica

La práctica se debe de entregar a través de la plataforma moodle de la universidad antes de los días señalados.

Pero si el profesor tiene interés, puede seguir en tiempo real los avances que voy llevando en la práctica a través de GitHub (o incluso puede llegar a usar el sistema de bugs para hacer notar deficiencias o preguntas en la práctica). Todo ello a través del enlace:
`https://github.com/PracticasUCO/Practica1-Metaheuristica`

3. Ejecución

Para ejecutar la práctica debe de tener Ruby instalado en su sistema. Las versiones soportadas son a partir de Ruby 2.0.0 en adelante. En la Universidad de Córdoba actualmente tienen instalado la versión de Ruby 1.9.3 por lo que trataré de hablar con quien lleve la instalación de programas en la facultad para tratar de que ofrezcan también una versión más actualizada.

Además es necesario tener instaladas las siguientes gemas:

- **minitest**: gem install minitest (es opcional).
- **getopt**: gem install getopt (es obligatoria para que el código ejecute).

Observe que no necesita tener privilegios de administrador para instalar gemas en Ruby.

Por ultimo, debido a que se ha realizado parte de la implementación en C, es necesario compilar esa pequeña parte. Para ello, utilize el archivo *Makefile* que le proporciono.

Si tiene alguna duda acerca de cómo usar la práctica escriba:
$ ruby practica1.rb –help

Si quiere comprobar los test de la practica escriba:
$ ruby test/all.rb

4. Representación de las soluciones

Cada uno de los problemas se ha representado internamente como una clase, que ha sido denotada como BasicProblem, donde Basic hace referencia a que la clase solo tiene en cuenta los datos más importantes del problema y devuelve una solución sin tratar de *optimizarla*, y Problem representa el problema a tratar.

De esta manera se han creado 4 clases que son:

- BasicMMDP: Representación básica del problema Max Min Diversity Problem.
- BasicTSP: Representación básica del problema Travelling Salesman Problem.
- BasicCWP: Representación básica del problema CutWidth Problem.
- BasicPHub: Representación básica del problema P-Hub.

Como ya he comentado estas clases definen una representación interna del problema y una serie de métodos para generar soluciones a dicho problema. Debido a que se trataba de hacer las clases lo más sencillas posible, estás clases no tratan de encontrar soluciones óptimas, aún más, ni siquiera lo intentan, simplemente devuelven la primera solución que se encuentran.

4.1. Representación del problema Max Min Diversity Problem

Para poder representar de una manera eficaz el problema MMDP se ha creado una clase con la siguiente estructura:

- **total_nodes**: Se trata de un atributo numérico que indica el número total de nodos que hay en la solución.

- **solution_nodes**: Se trata de un atributo numérico que indica el número de nodos que deben aparecer en la solución.
- **lista_nodos**: Se trata de un Array que contiene todos los nodos del problema
- **punto_ruptura**: Un atributo de tipo entero que indica cuándo dejar de buscar soluciones. Este atributo se optimiza automáticamente.
- **Nodes**: Se trata de una tabla de hash que usa como llave un Array de dimensión dos y que devuelve la diversidad entre los nodos del Array o nil si no se tiene conocimiento sobre la diversidad de los nodos indicados en el Array.

Con esto ya hay información suficiente para construir una solución, ya que se pueden elegir nodos al azar de la *lista_nodos* hasta alcanzar los *solution_nodes* necesarios y se puede calcular la diversidad que hay en la solución haciendo uso de la tabla de hash *Nodes*.

4.2. Representación del problema Travelling Salesman Problem

La estructura interna de BasicTSP es muy sencilla:

- **caminos**: es un atributo de clase de tipo Array que indica los caminos posibles que hay desde una ciudad a otra (en otros lenguajes esto es una estructura matricial).
- **numero_ciudades**: es un atributo que representa el numero de ciudades leídas en total.

Con esto ya hay información suficiente para obtener todas las soluciones del problema de manera clásica.

4.3. Representación del problema CutWidh Problem

Ésta es sin duda la representación más simple realizada en todos los ejercicios propuestos.

Simplemente se almacena todo en un atributo llamado *grafo* que viene a ser una tabla de Hash que usa como llave un nodo leído de la base de datos en modo texto y pasado a mayúsculas (por lo que puede cambiarse la representación del problema en la base de datos y usarse ciudades en lugar

de números), y devuelve como resultado un Array con todos los nodos a los que se puede ir a partir de la llave.

En caso de que el nodo no tenga aristas o no se conozca, *grafo* devolverá un Array vacío.

No hace falta más información para poder construir la solución ni la función objetivo.

4.4. Representación del problema P-Hub

Esta sin duda, la representación más compleja de todos los problemas dados. Tanto que al final se ha construido una clase que represente a los nodos de la red a tratar llamada CapacitedPHub.

La clase CapacitedPHub representa en si un nodo de la red y le dota de funcionalidades básicas, pero en ningún momento es capaz de realizar funciones complejas como comprobar que las conexiones se han realizado correctamente, etc.

Esta clase tiene la siguiente estructura:

- **id**: Un atributo de valor entero que es único para cada nodo. Se asigna automáticamente y no puede ser modificado ni establecerse inicialmente.
- **coordenadas**: Es un array de dos elementos que indican las coordenadas del nodo. El contenido del array es de dos números flotantes (se asume que se está en un espacio euclidiano).
- **reserva:** Indica la cantidad de recursos que tiene disponibles el concentrador en un momento dado (si el nodo actúa como cliente el valor sera igual a la capacidad).
- **capacidad_servicio**: Indica que capacidad puede ofrecer el nodo cuando éste actúa como concentrador.
- **demanda**: Indica la demanda de servicios del nodo cuando éste actúa como cliente.
- **id_concentrador**: Indica el id del concentrador al que está conectado el nodo cuando actúa como cliente.

Con todo ello se tiene información suficiente como para realizar tareas sencillas con los nodos. Además la clase BasicPHub contiene:

- **Nodos**: este atributo es un Array con todos los nodos leídos de la base de datos encapsulados dentro de la clase CapacitedPHubNode.

- **numero_concentradores**: Indica el número de concentradores que debe de aparecer en la solución.
- **capacidad_concentrador**: Desde que se añade la restricción de que todos los nodos tiene la misma capacidad se añade este atributo a la clase para poder acceder con más facilidad a esta información.

Con esta enorme estructura se puede realizar muy fácilmente el problema propuesto.

5. Descripción de la función objetivo

5.1. Max Min Diversity Problem

La función objetivo de MMDP esta dividida en dos funciones internas: *obtener_suma_costes* y *obtener_coste_entre*.

La idea básica detrás de la evaluación que se realiza es la siguiente:

Si tenemos información sobre la diversidad que produce el unir dos elementos cualesquiera distintos, podríamos calcular la diversidad que se produce de un elemento concreto frente a cualquier otro de un determinado conjunto.

Sea $D_{i,j}$ la diversidad que hay entre el nodo i y el nodo j. Entonces tenemos que la diversidad que produce el nodo i en un Array de n elementos (Dt_i) es la que se ven la expresión 1

$$Dt_i = \sum_{j=0}^{n-1} D_{i,j} \tag{1}$$

Asumiendo que $D_{i,i}$ vale cero. La expresión dos muestra el resultado de efectuar la misma operación con todos los nodos, sin repetirlos, esta es la expresión usada para calcular el valor de la función objetivo.

$$Dt = \sum_{i=0}^{n-1} \sum_{j=i}^{n-1} D_{i,j} \tag{2}$$

Se asume que $D_{i,i}$ es cero (un individuo con respecto a sí mismo no aporta diversidad) y Dt indica Diversidad Total.

5.2. Travelling Salesman Problem

La función objetivo debe devolver el coste de viajar a través de todas las ciudades que representa el vector solución que se devuelve para ello, si

$D_{i,j}$ indica el coste de ir de la ciudad i del vector a la ciudad j del vector (que tiene un tamaño n), la función objetivo (llamada coste_solucion en la clase BasicTSP), se puede resumir tal y como se indica en la expresión 3. CT representa el coste total.

$$CT = (\sum_{i=0}^{n-2} D_{i,(i+1)}) + D_{(n-1),0} \tag{3}$$

La función objetivo de esta clase por tanto hace uso de esta expresión mirando en la matriz que tiene almacenada para coger las $D_{i,j}$.

Como se deben incluir todas las ciudades en el recorrido del viajante de comercio, n (el tamaño del vector) es igual al número de ciudades totales.

5.3. CutWidth Problem

La función objetivo de este problema trata de contar cuántas aristas se rompen a través de una línea imaginaria cuando se ordenan los nodos de un grafo de forma lineal.

Previamente se declaran:

- Un vector llamado *procesados* que indica los nodos ya procesados.
- Una tabla de hash llamada *para_cerrar* que usa como llave un nodo y devuelve un entero que indica el número de aristas que cierran en dicho nodo. Si el nodo usado como llave no está en la tabla de Hash *para_cerrar* devolverá un cero.
- Una variable de tipo entero que se inicializa a cero y que llamaremos *opened*. Esta variable indica el número de aristas que están abiertas en este momento, es decir, que salieron de un nodo pero no llegaron a su destino aún.
- Una variable de tipo entero llamada *count* que se inicializa a cero y que indica el número de cortes que hay por el momento.

Ahora vamos recorriendo el vector solución y en cada iteración haremos lo siguiente: (usaré el termino *nodo_d* para referirme al nodo que se está iterando en dicho momento):

- Se añade *nodo_d* al vector de procesados.
- Se crea un Array llamado aberturas que contiene todos los nodos a los que se puede ir a partir del *nodo_d* menos los nodos que ya están en procesados.

- Se suma a *opened* la longitud del Array *aberturas* y se le resta el valor obtenido de *para_cerrar[nodo_d]*.
- Se borra a *nodo_d* de la tabla de Hash *para_cerrar*.
- Se le suma a *count* el valor de *opened*.
- Sumamos uno al valor devuelto por *para_cerrar* pasándole como llave cada uno de los nodos que estén en *aberturas*. Este valor se guarda en la tabla de hash.

Se devuelve el valor *count*

5.4. PHub Problem

En la función objetivo del PHub Problem lo que se trata de de evaluar la distancia que tiene todos los nodos a su concentrador, para ello se realiza la suma de todas las distancias (esto es una tarea sencilla ya que cada cliente devuelve a quien esta conectado) y se fuerza ha que haya el número de concentradores que se estipula.

6. Experimentos

Se han realizado 1000 iteraciones con todas y cada una de las instancias que ha proporcionado el profesor. Se ha medido tanto el valor de la función objetivo en cada una de ellas como el tiempo de ejecución.

Además se ha sacado unas estadísticas básicas para el tiempo que ha tardado el pc en ejecutar todas y cada uno de los problemas. Puede consultar esta información en una hoja de calculo llamada *Experimentos Practica 1.ods* o en el pdf *Experimentos Practica 1.pdf*

En cada una de las iteraciones se observaba el valor de la función objetivo nos quedamos con aquellas que diesen el mayor valor y el menor valor.

Bibliografía

- Greedy Randomized Adaptive Search Procedure (GRASP), una alternativa valiosa en la minimización de la tardanza total ponderada en una máquina. Por Juan Pablo Caballero Villalobos y Jorge Andrés Alvarado Valencia. Ing. Univ. Bogotá (Colombia). 25 de enero de 2010.
- Algoritmo Grasp híbrido para resolver una nueva variante del problema de la diversidad máxima. Por Fernando Sandoya y Rafael Martí. Congreso Latino-Iberoamericano de investigación operativa. Simpósio Brasileiro de Pesquisa Operacional. 24-28 Septiembre 2012 en Río de Janeiro, Brasil.
- Programming Ruby 1.9 The Pragmatic Programmers Guide by Dave Thomas with Chad Fowler and Andy Hunt November 2010