Técnicas Algorítmicas

Gonzalo Sainz-Trápaga (GomoX)

26 de Julio de 2008 - http://www.pcmasmas.com

Este documento es el resumen de la charla sobre técnicas algorítmicas del 26 de Julio de 2008 organizada por los usuarios de PC++ (www.pcmasmas.com). A continuación se discuten varias de las técnicas habituales utilizadas en diseño de algoritmos para lograr programas eficientes. Para cada una de las ideas, se presentan algunos ejemplos de algoritmos conocidos que la usan, y se discute su utilidad y sus limitaciones.

1. Tipos de problemas

Podemos clasificar los problemas de cómputo según el resultado que se busca obtener de una solución. Por ejemplo, dada la secuencia de números [7,-2,8,3,-8,3,11,54], tenemos los siguientes:

- **Problemas de decisión**: ¿Existe algún número primo mayor que 5 en esta secuencia? ¿Todos los números de la secuencia son pares? ¿Existe alguna subsecuencia que sume 0?
- **Problemas de optimización**: Hallar el número primo más grande de esta secuencia. Ordenar la secuencia de menor a mayor. Hallar la subsecuencia de suma mínima.

Los casos de arriba consisten en, de alguna manera, recorrer un universo de posibles candidatos y examinar cada uno de ellos. Para los problemas de decisión y el primero de optimización, el universo son los números de la secuencia. Para el problema de ordenamiento, el universo de posibles candidatos son todas las permutaciones de la secuencia (habrá que elegir una permutación que esté ordenada y devolverla para resolver el problema).

2. Técnicas exactas

Para problemas cuya dificultad de cómputo sea moderada o baja, es razonable exigir soluciones exactas a los problemas. Veamos algunas ideas para el diseño de algoritmos que pueden ser de utilidad a la hora de atacar un problema.

2.1. Fuerza Bruta

El algoritmo de fuerza bruta es el más trivial para un problema de optimización combinatoria: consiste en generar todo el universo de candidatos y visitarlos uno por uno, para hallar (o no) la solución válida a nuestro problema. Su aplicación más común es para buscar colisiones en funciones de *hash*, donde se utiliza precisamente porque no se conoce prácticamente nada sobre el comportamiento de las funciones.

2.2. Backtracking

Los algoritmos de *backtracking* son una refinación de los de fuerza bruta, donde nuevamente se explora todo el universo de candidatos a solución para un problema para elegir uno apropiado. La diferencia está en que se explora de forma ordenada, lo cual permite, dado un conocimiento del problema, filtrar de antemano grandes porciones del espacio de candidatos. Este recorte se denomina "poda", y puede mejorar mucho el rendimiento del algoritmo. Se puede podar una porción del espacio cuando se sabe que es imposible que un elemento de dicha porción supere al mejor candidato que fue hallado hasta el momento. Sin embargo, en general no se puede garantizar la efectividad de las podas, por lo tanto pueden existir instancias del problema donde el tiempo sea peor que el del algoritmo de fuerza bruta.

2.3. Dividir y Conquistar

La técnica de *divide and conquer* se basa en la idea de que es más fácil atacar un problema por partes que hacerlo todo a la vez. Un algoritmo consta de tres etapas: dividir, conquistar y combinar. En ellas se divide una instancia

del problema en dos o más instancias más pequeñas, que se resuelven (generalmente con una llamada recursiva) y luego se combinan sus soluciones para obtener la solución del problema original. Algoritmos que utilizan la técnica de dividir y conquistar son *Merge Sort*, *Quick Sort*, el algoritmo de Strassen para multiplicación de matrices y la multiplicación de polinomios que es la base de la transformada rápida de Fourier (FFT). Este tipo de algoritmos es naturalmente apropiado para ejecución paralela.

2.4. Programación dinámica

En muchos casos un problema no puede ser dividido en problemas disjuntos de forma útil para la utilización de un algoritmo de *divide and conquer*. La técnica de programación dinámica se basa en el aprovechamiento del **principio de optimalidad**. Este principio dicta que la solución óptima de un problema contiene soluciones óptimas a problemas más chicos (por ejemplo, cualquier subsecuencia de una secuencia ordenada está ordenada). Si bien no todos los problemas exhiben esta propiedad, se puede resolver de forma eficiente aquellos problemas que sí lo hacen comenzando a resolver desde los subproblemas más pequeños y continuando hacia los más grandes. En muchos casos, los resultados de los subproblemas pueden almacenarse en una tabla en memoria porque son necesarios muchas veces, evitando así rehacer cálculos innecesarios. Ejemplos de algoritmos de programación dinámica son el algoritmo de Dijkstra para caminos mínimos, el algoritmo (anónimo) para el problema de la mochila (KNAPSACK PROBLEM) y mucho más, pero hay aplicaciones muy sencillas como por ejemplo la implementación del cálculo de números de Fibonacci.

2.5. Programación lineal

La programación lineal es una técnica que consiste en expresar el problema como la maximización de una función lineal sujeta a una serie de restricciones lineales[1]. El problema puede estar definido con variables continuas o enteras (en este último caso se denomina programación entera). Si se logra expresar de forma concisa el problema a resolver como ecuaciones de este tipo, existen algoritmos eficientes (como SIMPLEX) para resolver los problemas. El problema de flujo en redes puede resolverse como un problema de programación lineal.

2.6. Utilidad y limitaciones

Si bien utilizando las técnicas mencionadas se pueden hacer algoritmos eficientes, en muchos contextos obtener la solución exacta puede tomar un tiempo prohibitivo. Por ejemplo, para el problema de guardas las cosas en una caja para mudarse (donde se desea desperdiciar una cantidad mínima de espacio por caja), si bien es preferible usar menos cajas, no es razonable esperar una semana para tener la mejor solución posible. Además, aplicar correctamente algunas de estas técnicas puede ser difícil ya que muchas veces requiere un entendimiento profundo del problema.

3. Técnicas aproximadas

En algunos casos, los problemas pueden ser muy difíciles para resolver de forma exacta en un tiempo razonable. Esto corresponde en general a problemas que se denominan NP-completos o NP-duros (NP-*hard*). En estos casos, se usan **heurísticas**.

3.1. Heurísticas y Metaheurísticas

Una heurística es una regla de decisión, que puede no ser ideal, pero debe ser medianamente rápida. Por ejemplo, para llenar una caja con la mayor cantidad posible de cosas, una heurística puede ser "poner primero las cosas más grandes". Esto no es necesariamente lo óptimo, pero permite llegar rápidamente a una solución. Las llamadas **metaheurísticas** son heurísticas para el diseño de heurísticas (o sea, ideas para construir algoritmos heurísticos).

[1] Se dice que una función es lineal cuando es un polinomio de grado máximo 1. Por ejemplo, $f(x, y) = 4x + 3y$ es una función lineal, pero $f(x, y) = x^2 + 7\sqrt(y)$ no lo es. De la misma manera, $f(x, y) = log(x) + 2^y$ no es una función lineal ya que no es un polinomio.

3.2. Algoritmos golosos

Una heurística golosa o *greedy* reacciona frente a una sucesión de decisiones tomando aquella que le provee más beneficio inmediato, sin importar lo que ocurra más tarde. Por ejemplo, para el problema de recorrer todas las ciudades de un mapa y volver al inicio caminando la menor distancia total (problema del viajante de comercio o Traveling Salesman), un algoritmo goloso posible consiste en caminar cada vez hacia la ciudad más próxima de las que todavía no fueron visitadas. Si bien en algunos casos un algoritmo goloso puede ser óptimo, por lo general provee a lo sumo una mala aproximación. Sin embargo, es muy rápido y por su sencillez puede ser usado como punto de partida para otros algoritmos.

3.3. Búsquedas con lista tabú

Una búsqueda con lista tabú o *taboo search* consiste en partir de un candidato al azar (o generado con alguna otra heurística) y modificarlo progresivamente (mediante un segundo algoritmo) hasta que no sea posible obtener mejoras haciendo esa modificación. Esto tiene el problema de que encuentra un mínimo o máximo local, mientras que en los problemas de optimización se buscan extremos globales. Cuando se halla un candidato "inmejorable", se agrega a una lista tabú y se reinicia el proceso, con la premisa de que no se podrá generar un candidato que ya esté en la lista. Esto impide que el algoritmo se bloquee en extremos locales. Como todas las metaheurísticas, este método tiene sus bemoles porque es simplista, pero puede presentar un comportamiento muy bueno en la práctica.

3.4. Algoritmos "bobos"

En el mismo espíritu que *taboo search*, existe otro tipo de algoritmo que jamás se bloquea en extremos locales que le impiden obtener candidatos mejores. Este algoritmo consiste simplemente en producir soluciones al azar y conservar la mejor de las que se observaron. Si bien puede parecer primitivo, dada la potencia de cálculo de las computadoras actuales este método no es despreciable y en muchos casos puede dar un resultado válido en un tiempo aceptable, sobre todo si se tiene en cuenta que insume muy poco tiempo codificar un algoritmo de este tipo (por lo tanto es particularmente apto para algoritmos de un único uso).

3.5. Algoritmos genéticos

Los algoritmos genéticos modelan un problema de optimización como un proceso evolutivo del tipo que se observa en la naturaleza. Se crea una población (un conjunto de candidatos, generalmente elegidos al azar) y se realiza una simulación en la que a medida que el tiempo va pasando, algunos candidatos procrean entre sí, y otros van muriendo. En cada generación, un porcentaje de los recién "nacidos" puede experimentar mutaciones al azar. En cada generación, se seleccionan los candidatos con mayor aptitud, de la misma manera que ocurre en la naturaleza. Para resolver un problema con un algoritmo de este tipo, debe entonces definirse una función de aptitud (que determine cuan "bueno" es un candidato), una función de procreación (que tome dos candidatos y devuelva un tercero combinando las características de los dos primeros) y una última función de mutación (que modifica de forma aleatoria alguna característica de un candidato). El proceso es análogo a la situación natural y se suele usar cuando hay muchas variables en juego y la relación que existe entre ellas no está clara.

3.6. Algoritmos de colonia de hormigas

Los algoritmos de colonia de hormigas modelan el comportamiento de una colonia de hormigas (increíblemente!). Estos animales son especialmente aptos para el problema de caminos mínimos: cualquiera que haya paseado por un parque pudo observar como centenarse de hormigas recorren un camino relativamente directo entre el hormiguero y la fuente de alimento. Lo curioso es que cada individuo se maneja de manera independiente y la comunicación se hace mediante feromonas, sustancias producidas por las hormigas que dejan una suerte de marca olfativa en el lugar donde pasan. Así, un individuo puede depositar una cantidad de feromona proporcional a cuan "bueno" sea el camino que halló. A la hora de elegir un camino en una intersección, otras hormigas utilizan este "olor" como mecanismo de *feedback* sobre la experiencia de otros individuos. Esta simulación se utiliza con excelentes resultados para el problema del viajante de comercio, con el beneficio añadido sobre otros algoritmos de que una simulación en curso puede reaccionar a cambios en el terreno a recorrer en tiempo real.

3.7. Otras metaheurísticas

Existen decenas de otras metaheurísticas similares a las que acabamos de enunciar. En general, existen dos tipos: por un lado aquellas que utilizan procedimientos más formales, similares a *taboo search*, como es el caso de GRASP (*Greedy Randomized Adaptive Search Procedure*). Por otro lado están aquellas que simulan procesos biológicos o naturales en los que la observación mostró que tienen comportamientos apropiados. Si bien el ejemplo clásico es el de los algoritmos de colonia de hormigas, existen varios más:

- **Redes neuronales** que simulan el comportamiento del cerebro humano
- **Enjambres de partículas** (*particle swarm*) que se acomodan en la posición más estable dentro de un sistema, comunicándose solo con sus vecinas para obtener *feedback*

3.8. Utilidad y limitaciones

Los mecanismos heurísticos son buenos cuando no es crítico hallar una solución realmente óptima al problema, y es razonable conformarse con una solución "bastante buena". En general las heurísticas insumen tiempos muy inferiores a los requeridos por los algoritmos exactos.

Sin embargo, cuando intervienen factores aleatorios (como es el caso con la mayoría de las metaheurísticas), existe una probabilidad de obtener resultados arbitrariamente malos. Para evitar eso, se busca lograr lo que se denomina algoritmos aproximados, que son algoritmos en los que el margen de error está acotado. Probar que un algoritmo es aproximado para un cierto problema puede ser muy difícil.

4. Conclusiones

Dependiendo de las condiciones del problema a resolver, hay diferentes técnicas que pueden ser utilizadas. En general, las técnicas exactas insumen tiempos más grandes tanto de desarrollo como de ejecución, y por lo tanto pueden no funcionar en contextos donde haga falta una respuesta rápida.

Las técnicas aproximadas basadas en metaheurísticas tienen un comportamiento probabilístico: tienen un buen rendimiento en tiempos más cortos, pero no dan garantías sobre la calidad de los resultados. Sin embargo, ofrecen cualidades especiales que pueden ser críticas para producir resultados utilizables.

El desarrollo más avanzado de algoritmos contempla algoritmos híbridos, donde se utilizan tanto técnicas exactas como aproximadas, y se las combina intentando obtener lo mejor de cada una.