TALLER N°3

Programación entera mixta: Capacitated Facility Location

Taller de programación 1-2025

Fecha: 14/07/2025

Autor: Alfonso Sebastian Palacios Vergara

TALLER N°3

Programación entera mixta: Capacitated Facility Location

Explicación breve del algoritmo

El programa resuelve problemas de localización de instalaciones con capacidades (CFL) usando un algoritmo de ramificación y acotamiento (branch and bound) que combina métodos exactos y heurísticos. La idea principal es explorar el espacio de soluciones mediante la creación de un árbol de decisión, donde cada nodo representa una solución parcial con ciertas variables fijadas (activación de fuentes de producción o facilities). Para cada nodo, se calcula una cota inferior resolviendo una relajación lineal del problema utilizando el solver de programación lineal de COIN-OR (ClpSimplex), lo que permite obtener valores óptimos para las variables continuas sin forzar enteros. Paralelamente, se mantiene una cota superior con la mejor solución entera encontrada, inicializada mediante una solución greedy que prioriza fuentes con mejor relación costo-capacidad. El algoritmo avanza seleccionando nodos con la menor cota inferior, ramificando en variables fraccionarias (por ejemplo, fuentes activadas parcialmente en la relajación) y podando ramas que no pueden mejorar la solución actual. La integración de COIN-OR asegura cálculos precisos para las cotas, mientras que la estructura del árbol de búsqueda evita explorar soluciones redundantes, optimizando eficiencia y convergencia.

Heurísticas o técnicas utilizadas

El programa utiliza un algoritmo de ramificación y acotamiento (Branch and Bound) integrado con técnicas de programación lineal (LP) para resolver problemas de localización de instalaciones con capacidades (CFL). Este enfoque combina métodos exactos y heurísticas para optimizar la búsqueda de soluciones óptimas. La estructura principal se basa en explorar un árbol de decisiones donde cada nodo representa una solución parcial con ciertas variables fijadas (activación de fuentes de producción). Los nodos se priorizan según su cota inferior, calculada mediante relajaciones lineales resueltas con el solver de COIN-OR (ClpSimplex), lo que permite obtener valores óptimos para las variables continuas sin forzar enteros. La cota superior se inicializa con una solución greedy que prioriza fuentes con mejor relación costo-capacidad, considerando también los costos de transporte para asignar clientes de manera eficiente.

La heurística greedy se enfoca en construir una solución factible inicial al seleccionar fuentes de producción basándose en una métrica que combina el costo de activación y la capacidad disponible, optimizando progresivamente la asignación de clientes según su densidad de costo (costo/demanda). Esto proporciona una cota superior rápida para guiar la poda del árbol de búsqueda. Durante la ramificación, se explora el nodo con menor cota inferior y se ramifica en variables fraccionarias (por ejemplo, fuentes activadas parcialmente en la

relajación LP). Cada ramificación genera dos nuevos nodos, fijando la variable a 0 o 1, y se evalúan para determinar si pueden mejorar la solución actual.

La relajación LP es clave para calcular cotas inferiores precisas: se modela el problema como un programa lineal, resolviendo iterativamente con simplex para obtener soluciones óptimas en cada nodo. Esto permite identificar rápidamente nodos inviables y reducir el espacio de búsqueda. La poda se realiza cuando la cota inferior de un nodo supera la cota superior actual, evitando explorar ramas redundantes. Además, se verifica si un nodo tiene una solución entera para actualizar la cota superior si mejora la solución conocida.

El algoritmo también incluye una heurística para calcular el costo exacto de una solución entera, asignando clientes a fuentes activadas de manera que minimice los costos de transporte. Esta heurística ordena clientes y fuentes por eficiencia (costo por unidad de capacidad) para garantizar una asignación óptima. La combinación de métodos exactos (simplex) y heurísticas (greedy y asignación por eficiencia) asegura que el algoritmo converja a una solución óptima sin sacrificar eficiencia, balanceando exploración exhaustiva con poda inteligente. La estructura evita técnicas como backtracking o programación dinámica, centrándose en la interacción entre LP relajado y búsqueda guiada por cotas, típica de los métodos Branch and Bound modernos.

Funcionamiento del programa

El programa resuelve problemas de localización de instalaciones con capacidades (CFL) mediante un algoritmo de ramificación y acotamiento (Branch and Bound) que integra relajaciones lineales resueltas con el solver de COIN-OR (ClpSimplex) y heurísticas para acelerar la convergencia. La lógica central se basa en explorar un árbol de decisiones donde cada nodo representa una solución parcial con ciertas variables fijadas (activación de fuentes de producción). El proceso comienza con una solución inicial obtenida mediante una heurística greedy que prioriza fuentes con mejor relación entre costo de activación y capacidad disponible, considerando también los costos de transporte para asignar clientes de manera eficiente. Esta solución inicial establece una cota superior (mejor solución entera conocida) que guía la poda del árbol.

Para cada nodo del árbol, se calcula una cota inferior resolviendo una relajación lineal del problema usando el simplex de COIN-OR. La relajación permite valores fraccionarios en las variables de activación (0 <= yj <= 1), lo que genera cotas más precisas que métodos heurísticos. Si la cota inferior de un nodo supera la cota superior actual, el nodo se descarta (poda por cota). Si no, se ramifica fijando una variable fraccionaria a 0 o 1, creando dos nuevos nodos hijos. La ramificación se enfoca en la variable más fraccionaria (la más alejada de 0 o 1), asegurando que se prioricen decisiones que impactan significativamente en la solución.

La relajación LP modela el problema como un programa lineal con restricciones de demanda y capacidad. Para cada nodo, se construye una matriz de coeficientes que incluye:

- Restricciones de demanda : Suma de envíos a un cliente debe igualar su demanda.
- Restricciones de capacidad : Suma de envíos desde una fuente no puede exceder su capacidad (relajada para variables libres). La función objetivo minimiza costos de activación y transporte. El solver de COIN-OR (ClpSimplex) resuelve esta relajación, devolviendo la cota inferior y la solución continua.

La heurística greedy para la solución inicial evalúa métricas de eficiencia (costo por unidad de capacidad) y asigna clientes a fuentes activadas de manera que minimice costos. Esta solución factible inicial asegura una cota superior rápida, evitando explorar ramas redundantes.

Durante la ramificación, se verifica si un nodo tiene una solución entera (todas las variables yj son 0 o 1). Si es así, se calcula el costo exacto asignando clientes a fuentes activadas, ordenando clientes por densidad de costo (costo por unidad de demanda) y fuentes por eficiencia (costo por unidad de capacidad restante). Esto garantiza una asignación óptima dentro de la solución entera.

El algoritmo termina cuando la diferencia entre la cota superior e inferior es menor que una tolerancia definida, asegurando que la solución es óptima. La estructura del árbol de búsqueda, junto con la integración de COIN-OR, permite explorar el espacio de soluciones de forma eficiente, evitando la explosión combinatoria de métodos como backtracking o programación dinámica.

El flujo del algoritmo sería:

1. Cargar problema dese archivo
2. Verificar validez (si la capacidad >= demanda)
3. Generar sol. greedy inicial
4. Crear nodo raíz y resolver relajación LP
5. Explorar la cola de nodos priorizados por cota inf
6. Para cada nodo:
    a. Poda si cota inf > cota sup
    b. Ramificar si solución no es entera
    c. Actualizar cotas si se encuentra solución mejor
7. Salida: Solución óptima o mensaje de no factibilidad

Aspectos de implementación y eficiencia

El algoritmo explora un árbol de decisiones donde cada nodo representa una solución parcial con ciertas variables fijadas (activación de fuentes). La priorización de nodos mediante una cola de prioridad (priority_queue) ordenada por la cota inferior asegura que se procesen primero los nodos más prometedores, minimizando la expansión innecesaria

del árbol. Esta estrategia reduce drásticamente el número de nodos evaluados, ya que los nodos con cotas inferiores superiores a la cota superior actual se descartan inmediatamente (poda por cota). Además, la detección de soluciones enteras en tiempo real actualiza la cota superior, acelerando la convergencia hacia la solución óptima.

La relajación LP resuelta con el solver de COIN-OR (ClpSimplex) es clave para calcular cotas inferiores precisas. Al permitir valores fraccionarios en las variables de activación (0 <= yj <= 1), el simplex encuentra soluciones óptimas para el problema relajado, lo que mejora la calidad de las cotas y reduce el número de ramas exploradas, sin embargo, la creación de un nuevo modelo de ClpSimplex para cada nodo implica una sobrecarga computacional, ya que requiere reconstruir matrices y vectores desde cero en cada llamada. Esto podría optimizarse reutilizando estructuras o actualizando solo las partes dinámicas (como limites de variables fijadas), reduciendo el costo de las operaciones repetidas.

- shared_ptr: Facilita la gestión automática de memoria, evitando fugas y simplificando la relación padre-hijo entre nodos. Sin embargo, el uso extensivo de shared_ptr introduce una sobrecarga mínima pero medible, ya que el control de contadores de referencias requiere operaciones atómicas.
- std::vector: Almacena soluciones y capacidades/demandas con acceso O(1), pero su creación y copia frecuente en nodos hijos (por ejemplo, al ramificar) puede generar fragmentación de memoria.
- priority_queue: Utiliza un heap interno para mantener los nodos ordenados por cota inferior, con inserciones y extracciones en O(log(n)), lo cual es eficiente para conjuntos grandes de nodos.

La solución inicial generada mediante una heurística greedy reduce significativamente el tiempo de ejecución. Al priorizar fuentes con mejor relación costo-capacidad y asignar clientes según la densidad de costo (costo por unidad de demanda), se obtiene una cota superior factible en tiempo O(m * log(m) + n * log(n)), donde m es el número de fuentes y n el de clientes. Esta cota superior actúa como umbral para la poda desde el inicio, evitando explorar ramas que no pueden mejorarla. La heurística también evita resolver LPs completas en etapas iniciales, ahorrando recursos computacionales.

La elección de la variable más fraccionaria (la más alejada de 0 o 1) prioriza decisiones con mayor impacto en la solución, reduciendo la profundidad del árbol. Sin embargo, en problemas con múltiples variables similares, este proceso puede no ser determinante y requerir ajustes adicionales.

Al calcular el costo exacto de una solución entera, se ordenan clientes por densidad de costo (costo por unidad de demanda) y fuentes por eficiencia (costo por unidad de capacidad restante). Este paso, aunque O(n * log(n) + m*log(m)), es rápido comparado con resolver un LP y garantiza asignaciones óptimas dentro de la solución entera.

Cada nodo almacena copias completas de soluciones y variables fijadas. En problemas grandes, esto puede llevar a un uso exponencial de memoria, limitando el tamaño máximo de instancia procesable.

Las optimizaciones implementadas son:

- Poda temprana: Al comparar cotas inferiores y superiores con una tolerancia (ej: 1e-6), se evitan cálculos redundantes en nodos que no pueden mejorar la solución actual.
- Relajación LP precisa: La integración de COIN-OR asegura cotas inferiores óptimas, superando métodos heurísticos que podrían subestimar o sobrestimar el potencial de un nodo.
- Priorización de nodos: La cola priorizada por cota inferior minimiza el riesgo de ramificar nodos con soluciones subóptimas, reduciendo el número de iteraciones necesarias.

Con respecto a los tiempos obtenidos, se utilizó un computador HP Pavillion x360 con Ubuntu 24.04.2 para la confección y ejecución del código. De una muestra de 5 ejecuciones por test, el promedio de tiempos para los casos de ejemplo fueron:

- facil1: 1694 microsegundos
- facil2: 3006 microsegundos
- facil3: 3668 microsegundos
- med1: 891639 microsegundos
- med2: 185197 microsegundos
- med3: 1831463 microsegundos

Si bien para casi todos los archivos de prueba el programa funcionó correctamente, para facil1 no se logró que calculara la solución óptima, o cualquier solución. Adicionalmente, no ha sido probado para casos difíciles.

Ejecución del código

El código posee un makefile que permite compilar todos los archivos necesarios. Se recomienda utilizar la versión de Linux utilizada en la confección del código para su compilación y ejecución. Se debe abrir la terminal en la carpeta PalaciosAlfonso217756235 (que contiene todos los archivos .cpp y .h) y se debe escribir el comando “make” y presionar enter. Luego, para ejecutar el código, se debe escribir el comando “./main” y presionar enter. La terminal ofrecerá varias opciones, y entre ellas pedirá el nombre del archivo a ejecutar: se debe introducir por consola el número 1 y posteriormente el nombre y la extensión del mismo, y presionar enter en ambos casos. El archivo a leer debe estar en el mismo directorio que los demás archivos (carpeta PalaciosAlfonso217756235). Posteriormente, para la ejecución de un problema en un archivo cargado, se debe ingresar 3 y, de estar seguro, se

debe posteriormente ingresar n. Una vez ejecutado, se debe presionar la tecla enter para volver al menú principal.

Se poseen 6 archivos de prueba: “facil1.txt”, “facil2.txt”, “facil3.txt”, “med1.txt”, “med2.txt” y “med3.txt”. Alternativamente, se puede escribir el número 5 por pantalla y posteriormente ingresar el nombre del nuevo archivo para crear un nuevo archivo de prueba.

Por otra parte, una vez realizado el make, se pueden ejecutar los comandos “./testProblema”, “./testNodoArbol”, “./testResolveCFL”, “./testComparadorNodos”, y “./testStrategyBranchAndBound” para testear cada una de las clases y su funcionamiento.

Se utilizan las librerías iostream, string, cstring, fstream, sstream, memory, limits, iomanip, vector, queue, algorithm, cmath, cfloat, stdexcept, map, ClpSimplex, CoinPackedMatrix, CoinPackedVector y chrono en el funcionamiento del algoritmo en conjunto.