UNIVERSIDAD SIMÓN BOLÍVAR
Ingeniería de la Computación
Diseño de Algoritmos II - CI-5652

One-Dimensional Cutting Stock Problem (CSP)

1ra. Entrega

Juan García 05-38207
Federico Flaviani 99-31744

8 de junio de 2011

Índice

1. Introducción **3**
1.1. Breve descripción del problema 3

2. Diseño **4**
2.1. Modelo utilizado para representar el problema 4
2.2. Estructuras y algoritmos involucrados en la aplicación 4
2.2.1. Clase Piece 5
2.2.2. Clase Pattern 5
2.3. Representación de la solución 7
2.4. Función Objetivo 7
2.5. Operadores 8

3. Detalles de implementación **9**
3.1. Pseudocodigo de las estructuras 9
3.1.1. Algoritmo de Local Search 9
3.1.2. Algoritmo de Local Search - Mejor Mejor 9
3.1.3. Algoritmo de ILS 10

4. Instrucciones de operación **11**

5. Estado actual **11**
5.1. Estado final de la aplicación 11
5.2. Errores 11

6. Conclusiones y recomendaciones **12**
6.1. Mejoras 12

7. Tablas **13**

8. Referencias bibliográficas **14**

1. Introducción

1.1. Breve descripción del problema

Cutting Stock Problem (CSP) o Problema de Corte y Empaquetamiento es un problema de optimización orientado al área de la programación entera, en donde el objetivo es minimizar el desperdicio generado al cortar una serie de patrones en un área dada. Este problema surge regularmente en el área de la industria siderúrgica o del papel, en donde se busca disminuir las pérdidas monetarias por desperdicios de material.

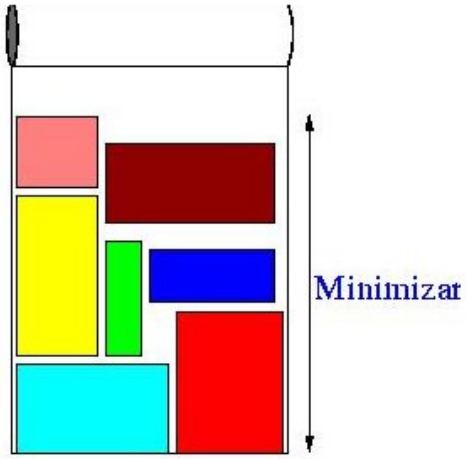

Figura 1: Forma general de CSP en dos dimensiones

En CSP se dispone de un área la cual deseamos cortar y existen muchas maneras de representarla. Puede ser una tela de tamaño finito en todos los sentidos, o una tela de ancho fijo con altura infinita, que es el caso que trataremos en este proyecto. Luego se tienen una serie de formas o patrones que queremos cortar de la mencionada tela. Dado que nuestro problema es la versión unidimensional (Figura 2), las piezas que se quieren cortar de la tela seran rectangulos con base constante igual a uno y con alturas variadas. Además no se permitirá el cambio de orientación de las piezas. Instancias del problema con mas nivel de dificultad se encuentra con la versión de dos dimensiones y piezas de tamaño y formas variables, asi como en tres dimensiones, en donde el problema se basa en optimizar la forma de empaquetar piezas en un volumen dado, tipo un contenedor de mercancia.

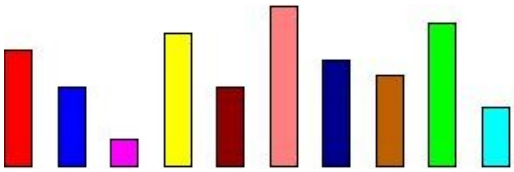

Figura 2: Generalización a cortes unidimensionales

2. Diseño

2.1. Modelo utilizado para representar el problema

Como se mencionó anteriormente la forma en que representamos el problema es usando una tela ïnfinitaçon ancho fijo. Para representar la infinitud de la tela hallamos una cota superior que cubra todos los posibles cortes que se pueden realizar, de esta forma nuestra cota superior CS sera:

$$CS = \sum_{i=1}^{N} L_i$$

donde N es el número de piezas a cortar y L_i es el largo de la pieza i.
En nuestro programa crearemos una serie de estructuras para representar cada parte del problema, comenzando con las piezas que queremos cortar, pasando por patrones de corte ya establecidos, los cuales representaran en nuestro caso una solución factible.

2.2. Estructuras y algoritmos involucrados en la aplicación

Dentro de las principales estructuras que utilizamos, estan aquellas que nos permiten representar un estado o solución factible. Para ellos necesitamos piezas y un patrón de corte, por lo tanto se definirán las estrucutras *Piece* y *Pattern* en conjunto con su tanda de atributos y diferentes métodos.

En cuanto a los algoritmos utilizados, encontramos en primera instancia un algoritmo greedy que nos permitira crear una solución inicial. Luego se aplicarán una serie de procedimientos con el fin de determinar una vecindad en un cierto espacio. Para finalmente aplicar los algortimos de las diferentes meta-heuristicas que se van a aplicar para resolver el problema.

El algoritmo greedy para generar una solución inicial consiste en ir aplicando cortes sucecivos a la tela, de la manera mas eficiciente posible. Primero

se ordenan los cortes a realizar según el tamaño de mayor a menor. Se comenzará por la base he iremos colocando las piezas por fila. Existirá un índice que llevará la información de la altura lograda de menor valor, esto con el fin de comenzar una nueva tanda de corte por fila a la mínima altura posible, sin tener que recorrer toda la matriz para insertar un corte. Finalmente cuando el algoritmo termina, nos entregará una dispocisión de cortes donde las piezas mas grandes estan en la parte inferior, y las mas pequeñas en el tope de la tela.

2.2.1. Clase Piece

Esta clase representa una pieza a ser cortada de la tela. Posee dos atributos, *large* que representa el largo del rectangulo a cortar y *pos* que permite conocer la posición de la pieza cortada en la tela total. Además, como todo objeto tiene un constructor que recibe como parámetros un entero que representa el largo del bloque y un arreglo bidimensional para inicializar la posición del mismo en la tela. Finalmente posee un método *clone* que permite crear una copia de una pieza.

```
class Piece {
 public:
  int large;
  int pos[2][2];
  Piece(int, int[2][2]);
  Piece clone();
};
```

2.2.2. Clase Pattern

La clase *Pattern* representa un patrón de corte de una isntancia del problema del CSP. Es decir es una solucion factible, en donde estan posicionadas todas las piezas de corte sobre la tela. A partir de una instancia de patter se obtienen otras instancias de la misma aplicacando los operadores de vecindad, asi como se puede obtener el nivel de calidad de la misma, para hacer comparaciones entre un patrón y otro.

La clase tiene una serie de atributos y métodos, pero dentro de los mas importantes encontramos a *width* y *height* que representan el alto y ancho de la tela a cortar. *num_pieces* y *pieces* son el número de piezas y la lista que contiene a las mismas, repectivamente. Además otros atributos que modelan el area ocupada y área libre, el número de lineas concretadas en el patrón y

variables enteras que soportan la generación de las vecindades.

Dentro de los métodos mas importantes encontramos los constructores de clase, que permiten inicializar un patrón cualquiera, también el método *vicinityNext* que permite ir generando la vecindad dinamincamente, basado en los métodos *vicinityFirstLevel*, *vicinitySecondLevel* y *vicinityThirdLevel*, el método *perturbar* que permite pertubar una solución iniciada para aplicar la metaheuristica *ILS*, entre otros métodos de apoyo para hacer revision de calidad y de la función objetivo.

```
class Pattern {
 public:
  int width;
  int height;
  int num_pieces;
  list<Piece>* pieces;
  int area_ocup;
  int area_no_ocup;
  int lines;
  int iteTermino;
  int x,y,x2,y2,x21,y21,x3,y3,x32,y32,x31,y31;
  Pattern();
  Pattern(int, int);
  Pattern(list<Piece *>, int, int);
  Pattern perturbar(Pattern);
  int quality();
  int distance(Pattern);
  Pattern clone();
  void remove(int);
  void add(int,Piece);
  Pattern vicinityFirstLevel(bool);
  Pattern vicinitySecondLevel(bool);
  Pattern vicinityThirdLevel();
  Pattern vicinityNext();
  list<Pattern> vicinity();
  void updateHeight();
```

2.3. Representación de la solución

La solución al problema de CSP se encuentra representada mediante la clase *Pattern*, en donde se encuentra una lista de piezas que fueron cortadas y poseen mediante la clase *Piece* una serie de atributos que denotan la posición en la tela, y forman el patrón como fue cortada. Además podemos saber la altura y el número de lineas creadas con el corte, y analizar una función objetivo.

Esto es de manera general, pero la estructura que en concreto representará nuestra solución se basa en un arreglo de listas de piezas. El tamaño del arreglo viene definido por el ancho fijo de la tela a cortar, y cada una de las listas se compone por la piezas colocadas al estilo de una pila, en la columna i del arregla o tela. Esta representación nos será de conveniencia al momento de realizar un movimiento para generar un estado vecino.

2.4. Función Objetivo

La función objetivo que vamos a utilizar esta definida por diferentes parámetros. De manera general definimos una métrica de calidad de los estados solución basada en la máxima altura generada por el corte de cada una de las piezas. Es decir que si comparamos dos estados solución, en donde ambos tengan un patrón con el mismo número de piezas cortadas, es decir, la misma área ocupada, aquel patrón que tenga la altura menor sera la mejor. En primera instancia esa es la métrica que vamos a utilizar en nuestro problema.

Otra función objetivo que podemos usar esta basada en el número de lineas completas generadas en el patrón, es decir, todas las lineas completas de que tengan el ancho total de la tela, y como altura una unidad. El problema de esta métrica es que podemos encontrar dos patrones con el mismo número de lineas, pero con alturas muy diferente, en donde el malgasto sea excesivo.

Una idea mejor es crear una función objetivo que relacione las dos métricas anteriores, aunque actualmente no hemos definido esta estrategia para ser usada en las metaheuristicas a implementar.

2.5. Operadores

Para generar la vecindad para un estado solución utilizamos un conjunto de tres operadores, pero que pueden ser ultilizados por separado para generar tres tipos de vecindades. Los tres operadores se basan en el moviento de piezas, el operador de primer nivel genera una vecindad moviendo una pieza selecciona de los niveles superiores. El siguiente operador, de segundo nivel, fija una pieza seleccionada y realiza una llamada recursiva al operador de primer nivel, al final se recolocará la pieza fijada, resultando asi un movimiento de dos piezas. El tercer operador es una llamada al operador de tercer nivel, en donde se fija una pieza y se llama a el operador de segundo nivel.

3. Detalles de implementación

3.1. Pseudocodigo de las estructuras

3.1.1. Algoritmo de Local Search

Require: Patron inicial : actual
  $k \leftarrow 0$
  Patron next
  while $k < 100 \quad \wedge \quad !actual.iteTermino$ **do**
    while $k < 100 \quad \wedge \quad !actual.iteTermino$ **do**
      $next \leftarrow actual.vicinityNext()$
      if $next \neq NULL \quad \wedge \quad next.quality() < actual.quality()$ **then**
        $actual.destroy()$
        $actual \leftarrow next$
        k++
      else
        $next.destroy()$
      end if
    end while
  end while
  return actual

3.1.2. Algoritmo de Local Search - Mejor Mejor

Require: Patron inicial: actual
  $k \leftarrow 0$
  Patron next
  Patron best
  $best \leftarrow actual$
  while $k < 100$ **do**
    while $!actual.iteTermino$ **do**
      $next \leftarrow actual.vicinityNext()$
      if $next \neq NULL \quad \wedge \quad next.quality() < actual.quality()$ **then**
        $best \leftarrow next$
      else
        $next.destroy()$
      end if
    end while
    if $actual = best$ **then**
      break

```
    else
      actual ← best
    end if
    k++
  end while
  return actual
```

3.1.3. Algoritmo de ILS

```
Require: Patron inicial: s_mejor
  k ← 0
  s_mejor ← localSearch(s_mejor)
  while k < 100 do
    Patron s1
    s1 ← s_mejor.perturbar()
    s1 ← localSearch(s1)
    if s1.quality() < s_mejor.quality() then
      s_mejor.destroy()
      s_mejor ← s1
    else
      s1.destroy()
      break
    end if
    k++
  end while
  return s_mejor
```

4. Instrucciones de operación

Para compilar el programa debe realizar la llamada:

```
$> make
```

Y para realizar la corrida:

```
$> ./CSP
```

5. Estado actual

5.1. Estado final de la aplicación

La aplicación no se encuentra del todo operativa, sin embargo fue posible implementar el algoritmo greedy para generar la solución inicial, los operadores mencionados anteriormente para generar la vecindad, y las metaheurísticas de Local Search, Local Search con Mejor-Mejor e ILS.

5.2. Errores

Existe un problema en la implementación de las metaheurísticas que no permite salir de los óptimos locales que se encuentra en la solución greedy inicial. Esto es para casos, en que los movimientos que deban realizarse sean mas de cuatro. Básicamente hace falta un operador que logre concretar movimientos a mas de cuatro niveles, o que gracias a las sugerencias de los compañeros en el aula, se logre realizar una perturbación mayor a la instancia, para asi llegar de otra manera al optimo global.

6. Conclusiones y recomendaciones

Durante el desarrollo de esta primera entrega, pudimos notar el nivel de dificultad que posee este problema, sobre todo para encontrar una representación ideal. Ádemas en la literatura existen una gran cantidad de variaciones del mismo, lo que a veces puede resultar en una complicación al momento de definir las bases para nuestro problema específico.

Sin embargo fue posible realizar una implementación que logre representar efectivamente nuestro problema, y a partir de alli utilizar algunas metaherísticas para encontrar una buena solución al mismo. Aunque dado a las complicaciones generadas no fue posible realizar un análisis exaustivo de las metaheurísticas.

6.1. Mejoras

Es necesario mejorar nuestros operadores de vecindad, dado que en estos momentos se esta creando una vecindad muy grande, y que no permite hallar una buena solución. Con recomendaciones de los colegas en la disciplina y la profesora, se procederá a mejorar nuestros operadores de vecindad, y ademas aplicar las metaheurísticas con tendencia probabilisticas, las cuales deberian ayudar a mejorar las soluciones obtenidas hasta ahora.

7. Tablas

Parametro	Valor
k = Nro. iteraciones LS	100
k = Nro. iteraciones LS Mejor-Mejor	100
k = Nro. iteraciones ILS	100

Cuadro 1: Parametros

La instancia que se presenta a continuacion tenia 66 piezas, y la solucion inicial tenia como funcion de calidad 20.

Heuristica	Optimo Alcanzado	Tiempo (seg)
Local Search	4	6.616
LS Mejor-Mejor	4	17.558
ILS	4	12.370

Cuadro 2: Tiempo de las heuristicas para instancia de 66 piezas

8. Referencias bibliográficas

Referencias

[1] **Solving the Cutting Stock Problem in the Steel Industry**
KARELAHTI J.

[2] **A Genetic Solution for the Cutting Stock Problem**
ANDRAS P., ANDRAS A, SZABO Z.

[3] **A Progressive Heuristic Search for the Cutting Stock Problem**
ONAINDIA E., BARBER F., BOTTI V., CARRASCOSA C., HERNANDEZ M., REBOLLO M.