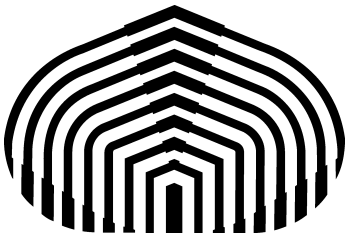

Universidad Simón Bolívar

Diseño de Algoritmos II

Proyecto II

Lorenzo Fundaró - 0639559
Germán Jaber - 0639749

25 de mayo de 2010

Índice

1. Breve descripción del problema **3**
1.1. Especificación formal 3

2. Descripción de las heurísticas empleadas **3**
2.1. Representación de la solución 3
2.2. Función Objetivo 3
2.3. Operadores 4

3. Pseudocódigos **4**
3.1. Heurística Finite Best Strip 4
3.2. Búsqueda Local 4
3.2.1. Method: FirstBest 5
3.2.2. Method: BestBest 6

4. Instancias **7**

5. Valores iniciales **7**

1. Breve descripción del problema

En teoría de complejidad computacional, el "Bin Packing Problem" es un problema combinatorio de dificultad NP-HARD. Se tienen objetos que tienen asignados un costo C e infinitos contenedores idénticos con capacidad V, el problema consiste en colocar los objetos dentro de la mínima cantidad posible.

Existen muchas variaciones de este problema, éstas difieren, básicamente, en las dimensiones con las que se trabaja. Entre las versiones existentes tenemos 1D, 2D, Strip Packing, Cutting Stock y 3D. Estas tienen muchas aplicaciones, como por ejemplo llenar containers con fines de exportación, llenar camión es con una capacidad determinada de peso, crear respaldos de archivos en medios portátiles (pendrives, disco duro portátil, etc), encontrar el orden de ejecución que menos tarde para programas que comparten memoria. Nosotros, particularmente, abordaremos la variación 2D.

La variación de 2-dimensiones trata de colocar rectangulos(objetos) dentro de rectangulos más grandes(contenedores). Tiene aplicaciones en la industria metalúrgica, maderera, papelera...básicamente, en cualquier industria donde se necesite cortar rectángulos de tamaño estándar en varios pedazos. También se aplica a la industria de imprenta, para encajar artículos de periódicos o revistas en la mínima cantidad de hojas posible.

En nuestro caso nos concentramos en una variación específica de 2D, que prohibe que los objetos se roten y exige que los 'cortes' se puedan hacer con una serie de cortes que van de un extremo a otro del bin (requisito clásico para industrias como la papelera).

1.1. Especificación formal

Dado un contenedor de capacidad V y una lista $a_1, ..., a_n$ de elementos de capacidad variable, se busca encontrar un entero B junto con una B-partición $S_1 \cup S_B$ de $\{1, ..., n\}$ tal que $\sum_{i \in S_k} a_i \leq V$ para todo $k = 1, .., B$. Una solución es óptima si posee un B mínimo.

2. Descripción de las heurísticas empleadas

2.1. Representación de la solución

Una solución puede verse como un vector de elementos empacados en un contenedor determinado. Cada elemento indica sus coordenadas en el contendor al que pertenece, el número de contendor donde está y las características respectivas al item como su ancho y alto.

2.2. Función Objetivo

La función objetivo consiste en comparar si luego de aplicar el operador de vecindad se ha reducido el número de contenedores utilizados.

Si esto sucede se procede a utilizar la nueva solución como la mejor obtenida hasta el momento.

2.3. Operadores

El operador que permite moverse de una solución a otra obedece a un principio que permite alcanzar una solución óptima. Este principio consiste en elegir un contenedor objetivo (Target Bin) el cuál tenga la mayor posibilidad de ser vaciado por sucesivas extracciones de sus elementos, esta posibilidad se determina a través de la función: [1]

$$\varphi(S_i) = \alpha \frac{\sum_{j \in S_i} w_j h_j}{WH} - \frac{|S_i|}{n}$$

donde S_i denota el conjunto de el conjunto de items que están actualmente empacados en el bin i y α es un valor predefinido de peso positivo.

Una vez se determina el Target Bin se procede a extraer un elemento del mismo, y junto con los elementos del resto de los contenedores (sin incluir al Target Bin) se trata de empacar dichos elementos. Luego la función objetivo se utilizará para decidir si cambiamos si cambiamos de solución o no.

3. Pseudocódigos

3.1. Heurística Finite Best Strip

Este algoritmo consiste en ir poniendo objetos en la zona inferior izquierda del contenedor. Se comienza con el primero y se prosigue con el resto hasta agotar el ancho del contenedor. Al agotar el ancho del bin hemos conseguido lo que se denomina un "strip" o también conocido como una capa y se procede a crear un nuevo strip para colocar el resto de los elementos. Cuando se han colocado todos los elementos se procede entonces a tratar de colocar strips encima de otros siempre que quepan en un bin.

3.2. Búsqueda Local

```
LocalSearch(Items I, int Hbin, int Wbin)
  int targetBin
  Packing pack
  Packing bestPack
  pack = initialSolution(I)
  bestPack = pack

  long max_iterations = 10000 //diez mil
  int terminationLimit = 4
  int iterations = 0

  int bins_in = 1
```

```
  int bins_out

  int done//Parámetro de terminación
  int div //Parámetro de diversificación

  //Ciclo principal
  while (iterations < max_iterations && done < terminationLimit){
    //Si la vecindad ya es muy grande
    if (bins_in == pack.total_bins){

      div = newDiv(pack,div) //Actualizo parámetro de diversificación
      done = newDone(pack,div) //Actualizo parámetro de terminación
      bins_in = 1 //Reduzco vecindad

      targetBin = diversify(pack,div) //Diversifico
    }
    else
      targetBin = targetBin(pack)

    bins_out = Method(targetBin,pack,Hbin,Wbin,bins_in)
    //Method puede ser FirstBest o BestBest

    if (bins_out > bins_in){
      bins_in = max(pack.total_bins,bins_in+1) //Aumento vecindad
    }
    else{
      bestPack = pack //Actualizo solución
      bins_in = min(1,bins_in-1) //Reduzco vecindad
    }
    ++iterations
  }
  return bestPack
}
```

3.2.1. Method: FirstBest

```
int FirstBest(int targetBin, Packing pack, int Hbin, int Wbin, int bins_in){
  for (item IN pack) {
    if (item IS IN targenBin){
      for (comb IN every combination of bins that
                  does not include the targetBin){

        //Tomo los items de la combinacion
        itemsToPack = getItems(comb,targetBin,items)

        //Incluyo el item que se 'sacó' del targetBin
        itemsToPack = itemsToPack + item

        temp_pack = FBS(itemsToPack) //Los empaco
```

```
        //Si logre poner todos los objetos en k bins o menos,
        //es que quite el objeto del targetBin
        if (temp_pack.total_bins <= bins_in){
          pack = merge(pack,temp_pack)
          return pack
        }
      }
    }
  }

  //Fallé en encontrar algo mejor, retorno infinito
  return INF
}
```

3.2.2. Method: BestBest

```
int BestBest(int targetBin, Packing pack, int Hbin, int Wbin, int bins_in){
  Packing bestPack
  bestPack.total_bins = INF

  for (item IN pack) {
    if (item IS IN targenBin){
      for (comb IN every combination of bins that
                  does not include the targetBin){

        //Tomo los items de la combinacion
        itemsToPack = getItems(comb,targetBin,items)

        //Incluyo el item que se 'sacó' del targetBin
        itemsToPack = itemsToPack + item

        temp_pack = FBS(itemsToPack) //Los empaco

        if (temp_pack.total_bins < bestPack.total_bins){
          bestPack = temp_pack
        }
        else if (temp_pack.total_bins == bestPack.total_bins){
          bestPack = breakTie(temp_pack,bestPack)
        }
      }
    }
  }

  if(bestPack.total_bins <= bins_in)
    return bestPack
  else
    return INF
}
```

4. Instancias

Se usaron instancias propuestas por Berkey y Wang. En particular, se usaron las instancias de clase I para 20 objetos y las de clase II para 40 objetos.
Vamos a presentar ahora la descripción de las intancias usadas, pero antes introduciremos ciertas notaciones:

- wj y hj: representan las dimensiones de los objetos individuales
- W y H: representan las dimensiones de los contenedores infinitos e iguales

Las instancias usadas son, entonces, las siguientes:

- Clase I: wj y hj uniformemente distribuidas en [1,10], W = H = 10
- Clase I: wj y hj uniformemente distribuidas en [1,10], W = H = 30

5. Valores iniciales

El valor inicial utilizado fué el de un objeto por contenedor.