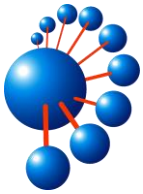


Informe Final:
Proyecto Bin Packing Problem y Cutting Stock

(Inteligencia Artificial 2024-1)

Integrantes: Diego Joaquín Andrés Venegas Anabalón

Pedro Ignacio Palacios Rossi

Francy Pilar Jélvez Jen

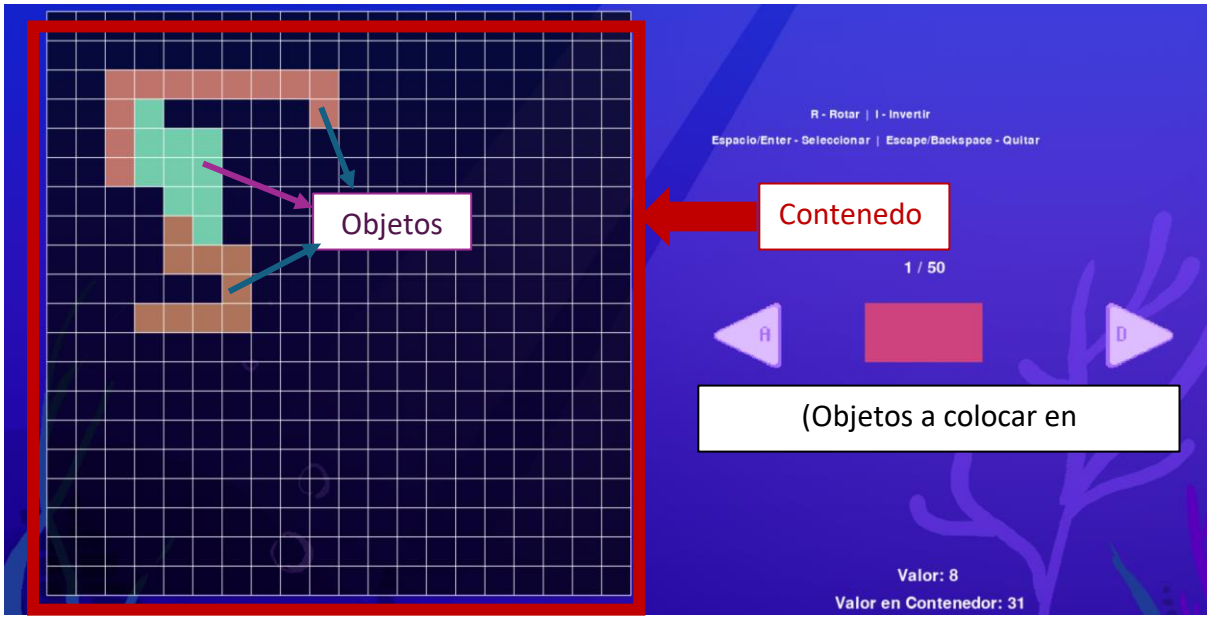
Profesor: Julio Godoy

Ayudante: Felipe Cerda

1. Introducción

En este informe, se analizará como resolver los problemas de “Cutting Stock” y ‘Bin Packing Problem’. Se resolverá mediante el algoritmo de Simulated Annealing, con distintas heurísticas.

La manera de representar este problema será con un programa que simula un juego. El objetivo del juego es llenar un contenedor con las n piezas que son otorgadas por el juego (Cabe mencionar que tanto el tamaño del contenedor como la cantidad de piezas son dadas por el jugador, y que la cantidad de piezas son siempre menos que los espacios disponibles por el contenedor). El código de este juego fue 100% propio de nosotros.



La función objetivo de nuestro agente a implementar será la de minimizar la cantidad de espacios vacíos en el contenedor, y las restricciones serán:

1. No se pueden superponer objetos sobre otros en el contenedor.
2. Todos los objetos en el contenedor deben estar completamente dentro de este.
3. No se permitirá la rotación de objetos.

Este programa es una variante del ya mencionado ‘Bin Packing Problem’, un problema clásico en informática y matemáticas, especialmente en la teoría de la optimización combinatoria. Consiste en que, dado un conjunto de elementos, cada uno con un volumen, y una capacidad máxima para unos contenedores, se debe determinar la combinación de elementos que maximice el uso del espacio total sin exceder la capacidad máxima del contenedor. En otras palabras, se trata de minimizar el espacio sobrante, haciendo que quepan los objetos de la mejor forma en el contenedor. Nuestro juego tiene objetos con un valor que equivale al área del objeto, y un sólo contenedor.

Sin embargo, mientras realizábamos la implementación de nuestro problema, nos dimos cuenta de que también podíamos solucionar el problema de optimización “Cutting Stock Problem”. Este problema consiste en que, dado una pieza de papel y una cantidad de cortes que se deben cortar si o si, se busca minimizar los espacios vacíos que quedan en dicho papel por los cortes. La variante de Cutting Stock que resuelve es la de 2 dimensiones sin guillotina (Que no hace un corte completo de ancho y/o largo).

Las propiedades del ambiente de estos problemas son:

- Observable (Se va a poder ver todas las elecciones a elegir).

- Determinista (No tiene componentes aleatorios).

- Secuencial (Pasa fase por fase).

- Estático (El espacio de la mochila y de las variables a elegir no va a cambiar de las preestablecidas).

- Discreto (Sólo hay tres acciones en total; Agregar/Quitar objetos de contenedor e ir al destino).

El agente será singular y dinámico.

2. Revisión Bibliográfica

La interfaz del problema fue 100% codificado por nosotros, y nos basamos en el siguiente repositorio (Originalmente sobre travelling salesman, pero fue adaptado a nuestro problema) para desarrollar el algoritmo de Simulated Annealing (Está en referencias también).

También nos basamos en los papers de las referencias para determinar cuál método servía para resolver este problema, y usamos el algoritmo para actualizar la temperatura del paper "Developing a simulated annealing algorithm for the cutting stock problem" de Kin Keung Lai y Jimmy W.M. Chan.

3. Método Propuesto

Como mencionado anteriormente, se propone el algoritmo de búsqueda Simulated Annealing. Simulated Annealing es un método estocástico de búsqueda local en el que, a partir de un estado actual, se elige al azar un sucesor, y si este sucesor es mejor que el del estado actual, se procede. Si es peor, se le da una oportunidad para que sea seleccionado o no. Esta oportunidad es dada por la temperatura, una variable que a medida que avanza la búsqueda, va permitiendo menos cantidad de estados peores.

Fue elegido este método debido a la naturaleza estática del problema (Las variables, como los objetos en el juego, no cambian) y porque requiere poca memoria, como todo método local. Además, en nuestra investigación de posibles soluciones al problema, encontramos que era posible hacerlo con Simulated Annealing, pero con modificaciones a la Temperatura, las cuales las aplicamos a nuestro proyecto.

Ahora, ¿Qué es lo que determinará que un estado del problema sea mejor o peor que el otro?, para esto, hemos determinado dos heurísticas:

1. readingOrder, colocar el objeto según el "Orden de lectura". Leyendo de izquierda a derecha en el contenedor, colocar el primer objeto, entre los objetos seleccionables, que pueda caber en ese espacio.
2. ContactSurface, colocar el objeto en la posición que tenga mayor cantidad de 'contactos' posibles con el contenedor y otros objetos.

A lo largo del informe, analizaremos los resultados que obtuvimos de ambas.

4. Implementación

4.1 Instrucciones

En el archivo .zip adjunto a este informe se encuentran los códigos. Para correr el juego, hacer que funcione el agente, elegir la heurística a ocupar y el problema a elegir, se necesita entrar a main.py y elegir las variables acorde a lo que se necesite. *Una vez elegidas, se ejecuta sólo main.py.*

```
def main(mode, heuristic, showAnimation, Problem):
    pygame.init()
    ##Variables importantes aquí. Para ajustar tamaño del contenedor y resolución, ir a Global.py
    if mode == None:
        mode = 1    ## modo == 0 es para jugar, modo == 1 es para una iteración de SA, modo == 2 es para mostrar SA entero
    if heuristic == None:
        heuristic = 0 ## heuritic == 0 es para resolver con countingOrder, heuristic == 1 es para resolver con contactSurface
    if showAnimation == None:
        showAnimation = 1 ## showAnimation == 0 corre el agente sin animación, showAnimation == 1 da la animación
    if Problem == None:
        Problem = 0 ## Problem == 0 permite resolver el problema de Cutting Stock, Problem == 1 el de Bin Packing Problem
```

mode: Para elegir el modo del programa. Si se desea "Jugar" (Manualmente colocar los objetos en el contenedor), se elige mode = 0. Si se desea correr una sola iteración de Simulated Annealing (El primer episodio del algoritmo), se elige mode = 1. Si se desea ejecutar todas las iteraciones de Simulated Annealing hasta encontrar la solución del problema elegido, se elige mode = 2.

heuristic: Para elegir la heurística a utilizar. Si se desea ocupar countingOrder, se elige heuristic = 0. En cambio, si se desea ocupar ContactSurface se elige heuristic = 1.

showAnimation: Mostrar o no la animación de la interfaz mientras es resuelta por el agente. Si se desea ver sin la animación, se elige showAnimation = 0, y si se desea ver la animación se elige showAnimation = 1.

Problem: Elegir el problema a resolver. Si se desea solucionar el de Cutting Stock, se elige Problem = 0. Si se desea solucionar Bin Packing Problem, se elige Problem = 1.

Otras variables importantes se encuentran en el archivo Global.py:

```
""" Parametros Que se pueden modificar """
WIDTH, HEIGHT = 1920, 1080  # tamaño /resolución de la pantalla.
dimContenedor = 10, 10    # tamaño del contenedor.
randConj = 5        # 0 para seed aleatoria, otro para fijar la seed en ese valor.
bg = 0  # 0 para fondo sharkcat, 1 para fondo axolote
```

WIDTH, HEIGHT = El ancho y el alto de la resolución de la ventana, respectivamente.

dimContenedor = Las dimensiones de ancho y alto del contenedor.

randConj = Semilla con la cual se crean los objetos. Si se desea una seed aleatoria, se elige randConj = 0.

bg = Fondo ilustrado del juego.

4.2 Implementación de Interfaz

La implementación del problema fue creada por nosotros y una parte destacable en el contexto del curso es la generación procedural de los objetos. Contamos con dos métodos de generación, uno los genera de tamaños y formas aleatorias y el otro se asegura de que entre todos los objetos se pueda rellenar un contenedor a la perfección.

- **Aleatorización de objetos:** Se utiliza un algoritmo que opera con dos listas: una comienza con un cuadrado inicial, mientras que la otra contiene todas las posiciones adyacentes. Durante un número específico de iteraciones, se selecciona aleatoriamente un cuadrado de la lista de adyacentes y se añade a la lista de cuadrados. Posteriormente, se actualiza la lista de adyacentes. El tamaño máximo de las listas es 16, para nuestra implementación.

```
def CrearObjetos(tmno_contenedor, area_max):
    areas = []
    vol_sum = 0
    while vol_sum < tmno_contenedor[0] * tmno_contenedor[1] * 3:
        areas.append(math.ceil(random.triangular(1,
min(tmno_contenedor[0] * tmno_contenedor[1], area_max), 4)))
        vol_sum += areas[-1]

    objetos = []
    for i in range(len(areas)):
        objetos.append(Objeto(i + 1, areas[i], tmno_contenedor))

    return objetos
```

```python
def ConstruirMatriz(self, area, tmno_max):
    tmno_max = (tmno_max[0] // 2, tmno_max[1] // 2)  # dividir por
2, más facil de manejar
    lista_cuadrados = []  # Lista de cuadrados que forman el objeto.
    lista_posibles_cuadrados = [(0, 0)]  # Lista de cuadrados que
pueden ser añadidos.
    for _ in range(area):
        # Elegir un cuadrado aleatorio.
        nuevo_cuadrado = random.choice(lista_posibles_cuadrados)
        lista_cuadrados.append(nuevo_cuadrado)
        while nuevo_cuadrado in lista_posibles_cuadrados:
            lista_posibles_cuadrados.remove(nuevo_cuadrado)
        # Añadir los cuadrados adyacentes al nuevo cuadrado.
        for c in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
            vecino = (nuevo_cuadrado[0] + c[0], nuevo_cuadrado[1] +
c[1])
            if (vecino not in lista_cuadrados
                    and -tmno_max[0] <= vecino[0] <= tmno_max[0]
                    and -tmno_max[1] <= vecino[1] <= tmno_max[1]):
                lista_posibles_cuadrados.append(vecino)

    matriz = matrizDesdePuntos(lista_cuadrados, self.id)
    return matriz
```

- **Relleno Perfecto:** Funciona con trazos que parten en pedazos el contenedor, la cantidad de trazos depende del tamaño del contenedor y los trazos funcionan como un agente que se mueve en direcciones aleatorias. Una vez recortado el contenedor, se arman los objetos a partir de los recortes, si se encuentra un objeto muy grande, se parte por la mitad.

```
def crearObjetosRellenoPerfecto(tmno_contenedor, maxArea=16,
numTrazos=None):
  if numTrazos is None:
        numTrazos = tmno_contenedor[0] * tmno_contenedor[1] // 8

    def armarMatrizObjeto(cuadradoInicial, id, maxArea):
        lista_cuadrados = []
        cuadrados = [cuadradoInicial]
        matrizContenedor[cuadradoInicial[0]][cuadradoInicial[1]] = 1
        while len(cuadrados) > 0:
            cuadrado = cuadrados.pop(0)
            lista_cuadrados.append(cuadrado)
            if not grilla[0][cuadrado[0]][cuadrado[1]]:
                c = cuadrado[0] - 1, cuadrado[1]
                if matrizContenedor[c[0]][c[1]] == 0:
                    matrizContenedor[c[0]][c[1]] = 1
                    cuadrados.append(c)
            if not grilla[1][cuadrado[0]][cuadrado[1]]:
                c = cuadrado[0], cuadrado[1] - 1
                if matrizContenedor[c[0]][c[1]] == 0:
                    matrizContenedor[c[0]][c[1]] = 1
                    cuadrados.append(c)
            if not grilla[0][cuadrado[0] + 1][cuadrado[1]]:
                c = cuadrado[0] + 1, cuadrado[1]
                if matrizContenedor[c[0]][c[1]] == 0:
                    matrizContenedor[c[0]][c[1]] = 1
                    cuadrados.append(c)
            if not grilla[1][cuadrado[0]][cuadrado[1] + 1]:
                c = cuadrado[0], cuadrado[1] + 1
                if matrizContenedor[c[0]][c[1]] == 0:
                    matrizContenedor[c[0]][c[1]] = 1
                    cuadrados.append(c)



        while len(lista_cuadrados) > maxArea:
            half = len(lista_cuadrados) // 2
```

```
            for c in lista_cuadrados[half:]:
                matrizContenedor[c[0]][c[1]] = 0
            lista_cuadrados = lista_cuadrados[:half]

        if len(lista_cuadrados) == 1:
            print(cuadradoInicial, lista_cuadrados[0])

        return matrizDesdePuntos(lista_cuadrados, id)


    grilla = looplist([looplist([looplist([0 for _ in
range(tmno_contenedor[1])]) for _ in range(tmno_contenedor[0])])
                       for _ in range(2)])

    for i in range(tmno_contenedor[0]):
        grilla[1][i][0] = 1
    for j in range(tmno_contenedor[1]):
        grilla[0][0][j] = 1

    numTrazos = [_ for _ in range(numTrazos)]
    for _ in numTrazos:
        trazo = []
        while True:
            inic = (random.randint(0, tmno_contenedor[0] - 1),
random.randint(0, tmno_contenedor[1] - 1))
            if grilla[0][inic[0]][inic[1]] == 1 or
grilla[1][inic[0]][inic[1]] == 1:
                trazo.append(inic)
                break

        direccion = random.choice([(0, 1), (0, -1), (1, 0), (-1,
0)])
        signt = inic
        while True:
            if random.random() < 0.4:
                direcciones = [(0, 1), (0, -1), (1, 0), (-1, 0)]
                direcciones.remove((direccion[0] * -1, direccion[1]
* -1))
                direccion = random.choice(direcciones)

            signt = (signt[0] + direccion[0], signt[1] +
direccion[1])
```

```
            trazo.append(signt)

            if (grilla[0][signt[0]][signt[1]] == 1 or
grilla[1][signt[0]][signt[1]] == 1
                    or grilla[0][signt[0]][signt[1] - 1] == 1 or
grilla[1][signt[0] - 1][signt[1]] == 1):
                break
            elif signt in trazo[:-1]:
                for _ in range(trazo.index(signt) - 1):
                    trazo.pop(0)
                break

        # si el trazo es muy corto, probabilidad de ignorarlo
        if len(trazo) <= 6 and random.random() < 0.9:
            numTrazos.append(_)
            continue
        elif len(trazo) <= 5 and random.random() < 0.99:
            numTrazos.append(_)
            continue
        elif len(trazo) <= 3 and random.random() < 0.99:
            numTrazos.append(_)
            continue
        elif len(trazo) <= 2 and random.random() < 0.99:
            numTrazos.append(_)
            continue

        # grabar trazo en la grilla
        for i in range(len(trazo) - 1):
            if trazo[i][0] == trazo[i + 1][0]:
                if trazo[i][1] < trazo[i + 1][1]:
                    grilla[0][trazo[i][0]][trazo[i][1]] = 1
                else:
                    grilla[0][trazo[i + 1][0]][trazo[i + 1][1]] = 1
            else:
                if trazo[i][0] < trazo[i + 1][0]:
                    grilla[1][trazo[i][0]][trazo[i][1]] = 1
                else:
                    grilla[1][trazo[i + 1][0]][trazo[i + 1][1]] = 1
```

4.3 Simulated Annealing

La implementación depende de dos partes; la del algoritmo de búsqueda y la heurística a utilizar.

- **Búsqueda:** Para la búsqueda, nos basamos en la de un repositorio de Github ya antes mencionado. Lo que hicimos fue modificarlo para que manejara las variables de nuestra interfaz, con el contenedor y la lista de objetos seleccionables, y cambiar las condiciones de avance, ya que esta implementación originalmente fue hecha para solucionar el problema del Travelling Salesman. Además, lo modificamos siguiendo criterios de un paper llamado "Developing a simulated annealing algorithm for the cutting stock problem", que busca resolver el Cutting Stock Problem.
  Funciona con un loop principal y uno interno, en el que el primero es para ir disminuyendo la temperatura y el segundo es donde ocurre la selección del vecino, donde va buscando a todos los posibles vecinos, y elige al primero que 1. Sea mejor según la heurística o 2. Sea aceptado según la probabilidad de la temperatura.

```
def search(self):
    print("num objetos: " + str(self.n))
    Lesgo = False
    punActual = self.get_function_value(self.ConjObjetos)   #
Evaluamos el conjuntos actual y guardamos su puntaje.
    bestConj = self.ConjObjetos.copy()      # Guarda la mejor
permutacion encontrada hasta el momento.
    bestPun = punActual     # Guarda el puntaje de la mejor
permutación hasta el momento.
    puntajes = [bestPun]    # Guarda el valor de la funcion para
cada iteracion.


    # Bucle principal del Simulated Annealing, en el cual disminuira
la Temperatura.
    for outite in range(self.outIte):

        uphillCount = 0
        # Bucle interno del Simulated Annealing, en el cual se
evaluaran los vecinos de la permutacion actual.
        inite = 0
        while inite < self.inIte:
            nuevoConj = self.get_neighbor(self.ConjObjetos, outite)
# Busca un vecino de ConjObjetos.
            nuevoPun = self.get_function_value(nuevoConj)       #
Guarda el puntaje del vecino.
            # Si el vecino es mejor que el conjunto actual nos
```

```
quedamos con el nuevo conjunto.
            if nuevoPun <= punActual:
                self.ConjObjetos = nuevoConj
                punActual = nuevoPun
            # Si el vecino no es mejor que el conjunto actual
reemplazamos con una probabilidad p.
            else:
                delta = (nuevoPun - punActual) / punActual  #
Calcula el delta.
                p = np.exp(-delta / self.T)                      #
Calcula la probabilidad p.
                if random.random() <= p:         # Reemplazamos
segun p.
                    uphillCount += 1
                    self.ConjObjetos = nuevoConj
                    punActual = nuevoPun
            # Si esta solucion es la mejor hasta el momento, la
guardamos.
            if punActual < bestPun:
                inite = 0
                bestConj = self.ConjObjetos
                bestPun = punActual
            # si se llego a una solucion optima, se termina el
algoritmo.
            if punActual == 0:
                Lesgo = True
                break

            inite += 1

        print(f'\rite={outite}, f_value = {bestPun}, T = {self.T},
upHill = {uphillCount}'
              f', VD =
{int(np.floor(np.exp((np.log(self.n)/self.outIte)*outite)) - 1)}',
end="")
        if outite % 30 == 0:
            print()  # salto de linea cada 30 iteraciones.

        self.T = self.T/(1 + self.beta*self.T)        # Baja la
temperatura.
        puntajes.append(bestPun)  # Guarda el valor de la funcion
para cada iteracion. (Esto es para graficar)
```

```
        if(Lesgo):
            break
```

- **Heurísticas:**
    - **readingOrder:** Se recorre el contenedor desde la esquina superior izquierda, colocando objeto por objeto en el contenedor, y se elige al primer objeto que se pueda colocar en el primer espacio sin romper con ninguna restricción.

```
def readingOrder(cont: Contenedor.Contenedor, objs: list,
showAnimation=False, screen=None):
    # Se coloca objeto por objeto en el contenedor.
    for obj in objs:
        flag = False
        # Se recorre el contenedor de izq a der, se baja una unidad
y se repite.
        for i in range(cont.tamano[0] - obj.tamano()[0] + 1):
            for j in range(cont.tamano[1] - obj.tamano()[1] + 1):
                if obj.verificarColisionConLista(cont, (i, j)):
                    colocarObjeto(cont, obj, (i, j))
```

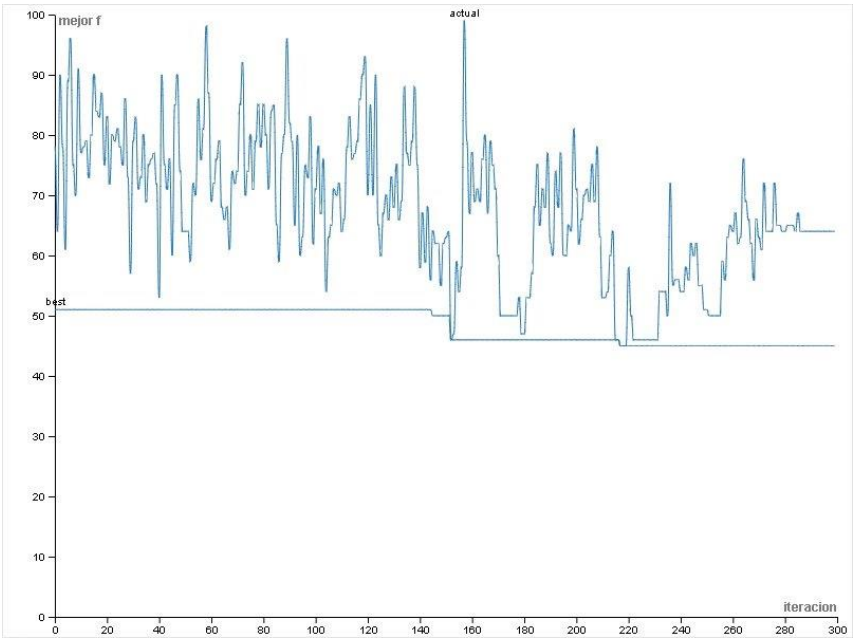
- ContactSurface: Se coloca objeto por objeto en el contenedor, y se elige al objeto que tenga la mayor cantidad de colisiones en el espacio analizado del contenedor.

```
def ContactSurface(cont: Contenedor.Contenedor, objs: list,
showAnimation=False, screen=None):
    # Se coloca objeto por objeto en el contenedor.
    for obj in objs:
        maxCo = -1
        pos = (-1, -1)
        # Se recorre el contenedor de izq a der, se baja una unidad
y se repite.
        for i in range(cont.tamano[0] - obj.tamano()[0] + 1):
            for j in range(cont.tamano[1] - obj.tamano()[1] + 1):
                if obj.verificarColisionConLista(cont, (i, j)):
                    contactos = obj.contarContactosConLista(cont,
(i, j))
                    if maxCo < contactos:
                        maxCo = contactos
                        pos = (i, j)
        if maxCo != -1:
            obj.colocarObjetoConLista(cont, pos)
```

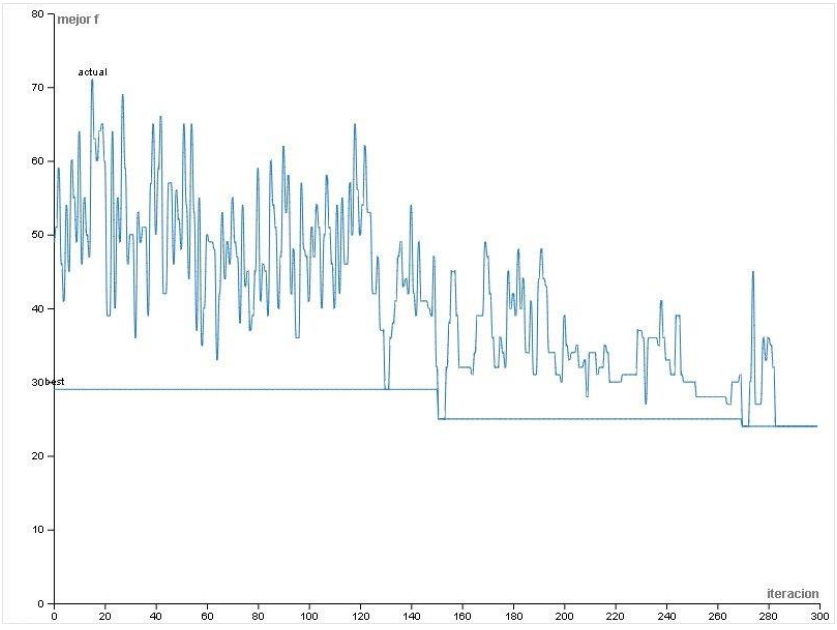
5. Experimentación: Resultados y Análisis

5.1 Gráficos para Cutting Stock Problem (Usando crearRellenoPerfecto)

Reading Order, mejor valor 45:

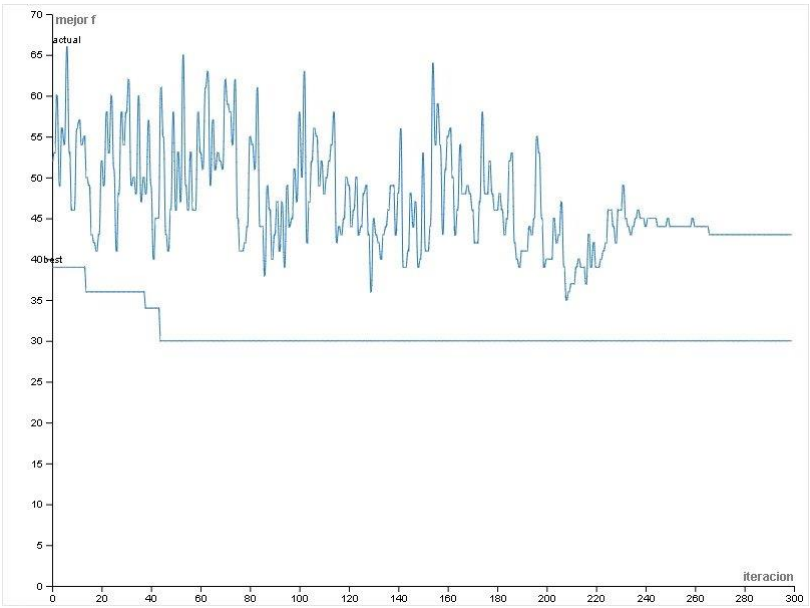


ContactSurface, mejor valor 24:

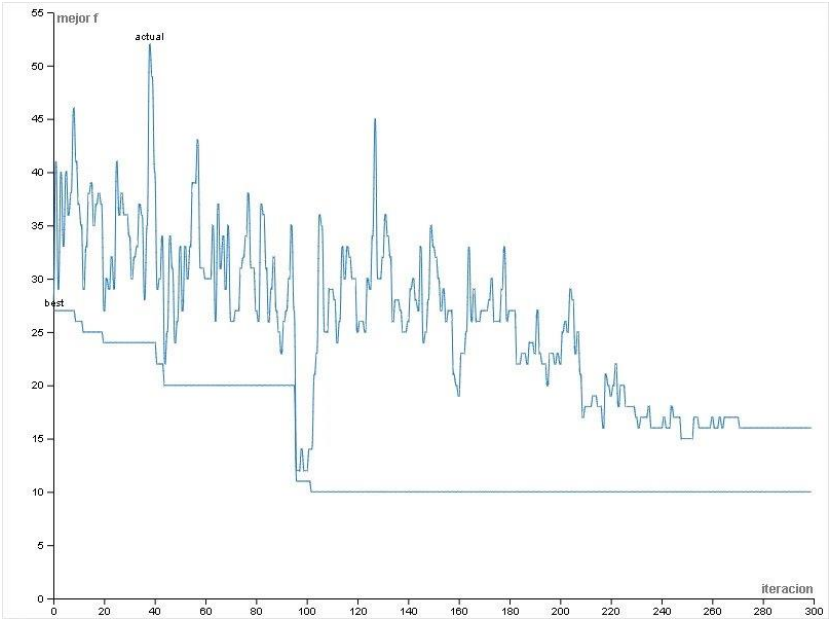


5.2 Gráficos para Bin Packing Problem (Usando Elementos Aleatorización de objetos)

ReadingOrder, mejor valor: 30



ContactSurface, mejor valor: 10



5.3 Análisis

En ambos problemas, ContactSurface da mejores resultados en menos iteraciones, aunque el tiempo por iteración aumente comparado con ReadingOrder. Esto es debido a que la heurística ContactSurface es más compleja, pues realiza una comparación de todos los elementos antes de colocar el objeto indicado, mientras que ReadingOrder elige el primero que encaje en el contenedor.

6. Conclusión

En conclusión, el algoritmo logra buenas aproximaciones a la solución óptima en poco tiempo y con pocos recursos computacionales.

Realizamos una investigación para buscar algoritmos de aproximación que pudiéramos adaptar a nuestro caso e implementamos conceptos teóricos complejos relacionados con el simulated annealing.

Observamos cómo afectan los elementos estocásticos, la solución inicial y el balance entre efectividad y costo computacional de una heurística en la búsqueda con simulated annealing.

7. Referencias

- Simulating annealing extraído de: https://github.com/alwaysbyx/Optimization-and-Search
- Artículo científico: “An effective heuristic for the two-dimensional irregular bin packing problem”: https://www.researchgate.net/publication/256497455_An_effective_heuristic_for_the_two-dimensional_irregular_bin_packing_problem
- Artículo científico: “Local Search Algorithms for the Bin Packing Problem and Their Relationships to Various Construction Heuristics”: https://www.researchgate.net/publication/220403638_Local_Search_Algorithms_for_the_Bin_Packing_Problem_and_Their_Relationships_to_Various_Construction_Heuristics
- Artículo científico: “Developing a simulated annealing algorithm for the cutting stock problem”: https://www.researchgate.net/publication/223090176_Developing_a_simulated_annealing_algorithm_for_the_cutting_stock_problem
- Artículo científico: “An Optimal Cooling Schedule Using a Simulated Annealing Based Approach”: https://www.scirp.org/journal/paperinformation?paperid=78834