Joaquín Dotor Ruiz
bo0017

Memoria Bin Packing Problem 2D

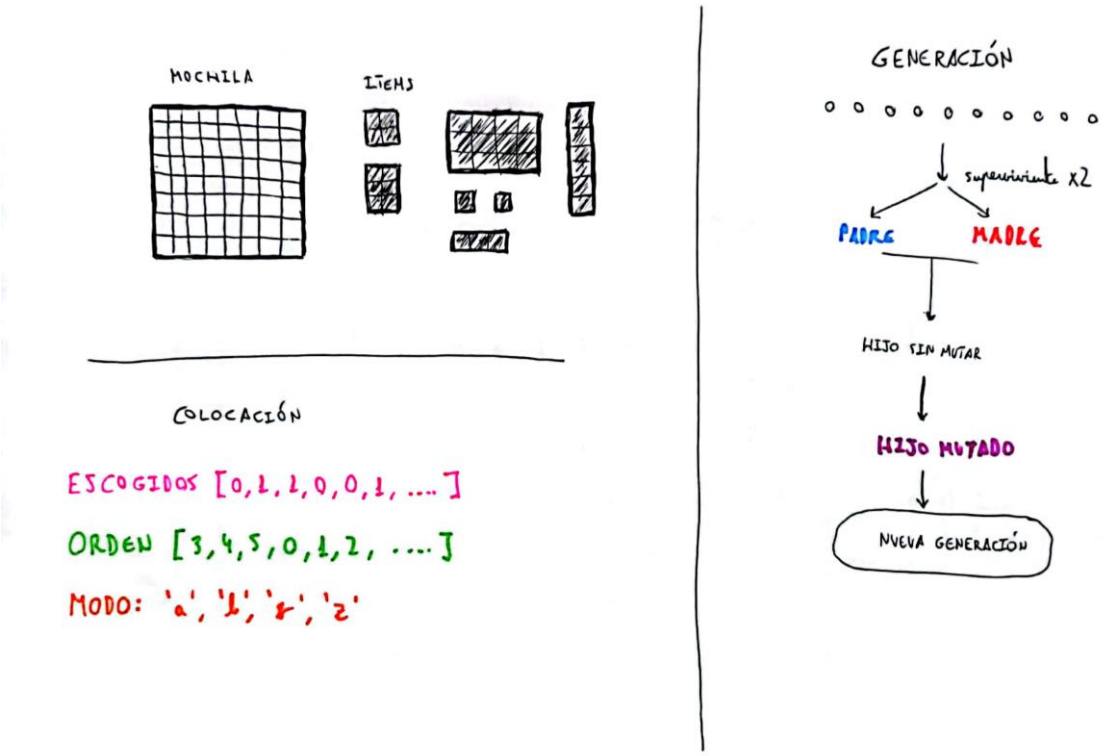
Introducción:

El problema de la mochila es un problema de optimización en el que disponemos de una serie de objetos y una mochila con un tamaño determinado, siendo objetivo minimizar el espacio desaprovechado de dicha mochila o, lo que es lo mismo, ocupar el máximo espacio posible. Este problema tiene numerosas aplicaciones como cargar un camión, almacenar copias de seguridad, algoritmos criptográficos, etc.

Para este caso, consideramos dos dimensiones, por lo que el espacio de la mochila se determinará por su alto y ancho, al igual que el espacio que ocupan los objetos.

La solución para el problema de la mochila se puede abordar de diferentes formas, una de ellas es mediante computación evolutiva, haciendo uso de algoritmos genéticos, que es la que se utiliza para este caso.

Explicación del funcionamiento:



Para introducir los objetos en la mochila, el procedimiento consiste en recorrer todos los objetos y para cada uno de ellos comprueba todas las posiciones posibles dentro de la mochila y las almacena.

Empieza con una colocación no escogiendo ningún objeto, el orden en el que se van guardando aleatorio y el modo de almacenaje (“a”, “b”, “r” o “z”) aleatorio. Valida esa colocación, repite este proceso tantas veces como individuos tenga una generación y establece la lista de colocaciones como primera generación.

Cabe destacar que el cromosoma orden es importante, ya que sin el estaríamos limitados a ir guardando los objetos en el orden lógico (primero, segundo, tercero, …), con el que se perderían muchas posibles colocaciones válidas.

El funcionamiento del algoritmo genético consiste en que a partir de esta primera generación se va evolucionando mediante cruce y mutación para conseguir nuevas generaciones cada vez más eficientes.

El criterio para cruzar a dos individuos es seleccionarlos con una probabilidad asociada a su puntuación entre la mitad de mejores, usando el espacio ocupado como parámetro de medición.

Una vez obtenido cada hijo, se muta con unas probabilidades fijas antes de incluirlo en la nueva generación.

Cuando la nueva generación se haya completado, este proceso se repite hasta obtener las generaciones deseadas.

La solución final será un objeto colocación, que nos indicará qué objetos escogemos y cómo colocarlos, siendo esta la mejor colocación.

Explicación del código:

-Clase Item

En esta clase se define el objeto a meter en la mochila, con atributos alto y ancho, getters y setters correspondientes y un constructor.

-Clase Mochila

En esta clase se define la mochila, con sus correspondientes atributos sizeX, sizeY y espacio[][], que indican, el alto, el ancho y una matriz con el espacio en el que se puede meter un objeto, valiendo ‘1’ si hay un objeto ocupando esa posición o ‘0’ si está vacía.

<u>En esta clase también se definen los siguientes métodos:</u>

- **void vaciar()** //Pone a ‘0’ todos los espacios de la mochila (la vacía).
- **boolean libre(int x, int y)** //Indica si en esa posición exacta de la mochila hay o no un objeto.
- **boolean cabe(int x, int y, Item item)** //Comprueba si un objeto cabe en esa posición exacta de la mochila.
- **boolean cabeGirado(int x, int y, Item item)** //Comprueba si un objeto cabe girado 90º en esa posición exacta de la mochila.

- **void colocarItem(Posicion posicion, Item item)** //Coloca el objeto en la mochila en base a una posición, que tiene información de sus coordenadas en la mochila y si está girado o no.
- **Getters y setters** y un método **mostrar()** para mostrar el estado actual de la mochila.

Clase Posicion

En esta clase indicaremos cómo guardar un objeto en la mochila, con los atributos X, Y y girado, que indican las coordenadas dentro de la mochila y si el objeto está girado o no.

En esta clase también se definen los siguientes métodos:

- **List<Posicion> posicionesPosibles(Mochila mochila, Item item)** //Con este método lo que obtenemos es una lista de todas las posiciones en las que podemos guardar un objeto en la mochila, según el estado actual de la mochila.
- **Posicion primeraPosicionPosible(Mochila mochila, Item item)** //Devuelve la primera posición posible.
- **Posicion ultimaPosicionPosible(Mochila mochila, Item item)** // Devuelve la última posición posible.
- **Posicion randomPosicionPosible(Mochila mochila, Item item)** //Devuelve una posición aleatoria posible.
- **Getters y setters** correspondientes

Clase Colocacion

Con el uso de esta clase obtenemos una colocación válida de los objetos en la mochila en base a tres cromosomas, que son: objetos escogidos, orden en el que se introduce cada objeto y modo.

Hay cuatro modos:

‘a’, los objetos se van introduciendo en la primera posición posible de la mochila.

‘b’, los objetos se van introduciendo en la última posición posible de la mochila.

‘r’, los objetos se van introduciendo en una posición aleatoria de la mochila.

‘z’, los objetos se van alternando, introduciéndose primero uno en la primera posición posible, siguiente en la última posición posible y así sucesivamente.

Además, con esta clase podemos mutar cada uno de los cromosomas y obtener una puntuación de la colocación.

Métodos importantes:

- **Colocacion(Mochila mochila, List<Item> items)** //Constructor que con los parámetros mochila y lista de objetos genera una colocación que no escoge ningún objeto, dejando aleatorio el orden y el modo de almacenaje.
- **boolean valida()** //Método que introduce los objetos en la mochila en base a los atributos escogidos[] orden[] y modo en caso de que sea posible. Si no lo es devuelve false.

- **Strting mochilaLlena()** //Invoca al método valida para llenar la mochila y la imprime por pantalla.
- **void mutarEscogidos(double prob)** //Muta el array de escogidos con una probabilidad indicada por parámetro.
- **void mutarOrden(double prob)** //Muta el orden de objetos introducidos con una probabilidad indicada por parámetro.
- **void mutarModo(double prob)** //Muta el modo de colocar los objetos con una probabilidad indicada por parámetro.
- **void mutar(double probE, double probO, double probM)** //Método que invoca a todos los métodos de mutación con sus respectivas probabilidades.
- **int puntuacion()** //Método que asocia una puntuación a la colocación actual.
- **Getters y setters** correspondientes.

Clase Generacion

La clase generación gestiona una lista de individuos (Colocaciones) con una serie de operaciones que nos permiten obtener la mitad de mejores individuos (contando la puntuación como espacio ocupado), obtener un individuo superviviente y cruzar dos individuos para la obtención de un individuo hijo.

Métodos importantes:

- **List<Colocacion> mitadMejores()** //Devuelve una lista de la mitad de colocaciones mejores de la generación usando la mediana de la puntuación para ello.
- **int sumaPuntos()** //Recorre la generación sumando la puntuación de cada individuo para obtener el total.
- **int sumaPuntosMitadMejores()** //Recorre la mitad de individuos mejores de la generación sumando la puntuación de cada uno para obtener el total.
- **Colocacion superviviente()** //Escoge un individuo de la mitad de mejores de la generación con una probabilidad de ser escogido asociada a cada uno.
- **Colocacion cruce(Colocacion padre, Colocacion madre)** //Cruza de manera aleatoria la colocación padre y la colocación madre para obtener un hijo con una nueva colocación.
- **int mejorPuntuación()** //Obtiene la puntuación del individuo con mayor puntuación de la generación.
- **double medianaPuntos()** //Obtiene la mediana de puntuación de la generación.

Clase AlgoritmoGenetico

La clase AlgoritmoGenetico nos permite gestionar todas las generaciones y obtener nuevas haciendo uso del método evolucionar.

Métodos importantes:

- **void evolucionar()** //Escoge un padre y una madre de la generación actual, los cruza para generar un hijo, añade el hijo a una nueva generación y repite hasta que la nueva generación tenga el mismo numero de individuos que la actual.
- **Colocacion mejorColocacionGeneracionActual()** //Obtiene la colocación con mejor puntuación de la generación actual.
- **void actualizarMejorColocacion()** //Actualiza el atributo con mejor colocación si en la generación actual hay alguna mejor.

Evolución de las probabilidades de mutación:

Los experimentos se han hecho con mochilas de entre 5x5 y 20x20 para obtener un equilibrio entre complejidad y tiempo de ejecución, probando ocasionalmente el algoritmo con mochilas excesivamente pequeñas o grandes (3x3 y 30x30) pero sin hacer mucho hincapié en ellas.

Para los ítems disponibles siempre se ha experimentado con más ítems que la capacidad total, aproximadamente el doble de espacio de los ítems que el espacio disponible en la mochila, incluyendo también ítems que no caben, ítems del tamaño exacto de la mochila, etc.

Las probabilidades de mutación que he ido tanteando han ido enfocadas a afinar el funcionamiento en las mochilas objetivo (5x5 – 20x20), no funcionando igual para todo el rango, pero si razonablemente bien para todo él.

La primera probabilidad escogida fue 0.1 para todos los cromosomas (escogidos, orden y modo) y el resultado fue que la probabilidad era excesivamente alta, produciendo resultados aleatorios, siendo más notable en mochilas de 10x10 en adelante.

La siguiente probabilidad probada fue 0.01 que funcionaba mejor en mochilas pequeñas (6x6, 7x7, etc.) ya que en esas mochilas habrá que meter menos objetos para llenarlas, entonces, al ser el vector de escogidos y orden más pequeño, la probabilidad no es necesario que sea tan pequeña ya que en el caso de que el vector tenga tamaño 10, de media mutará una componente, pero en el caso de mochilas grandes, vector de tamaño 100, mutarán 10 componentes de media, lo cual es muy aleatorio.

Como 0.01 funcionaba bien para mochilas pequeñas, pero no para las grandes, probé con 0.001 pero esta probabilidad resultó ser demasiado pequeña. Funciona más o menos bien para las mochilas de 20x20, pero la mutación era mínima. Para las mochilas pequeñas no funcionaba, prácticamente no mutaba nada.

Finalmente probé un valor intermedio entre 0.01 y 0.001, que fue 0.005 y resultó funcionar razonablemente bien para mochilas pequeñas y grandes.

Las probabilidades de mutación de los tres cromosomas han sido siempre iguales, siendo más influyente la probabilidad de mutar los objetos escogidos, y la menos influyente, la de mutar el modo de introducir los objetos ya que cualquiera de los modos es igual de eficiente excepto el aleatorio (‘r’), que es el menos eficiente.

El tamaño de la población y el número de generaciones se escogió teniendo en cuenta el tiempo de ejecución, por lo que no pueden ser generaciones muy grandes (más de 30), pero tampoco muy pequeñas (5 o menos) porque no funciona el algoritmo.

Al probar generaciones de entre 10 y 20 individuos, se observa que cuanto mayor sea la generación el proceso será más lento, ya que al principio todas las soluciones son malas y bajan la media de puntuación y por ello avanza más lento. Por ello he decidido que la población será de 10 individuos, que funciona bien para cualquier tamaño de mochila de nuestro rango, aunque para mochilas grandes funciona mejor con una mayor población.

El número de generaciones debe ser grande, al menos 100, pero hay casos en los que se alcanza un 80% de ocupación con pocas iteraciones. Con 100 generaciones nos aseguramos llegar siempre al 80% de ocupación. Si la mochila es grande funcionará mejor con más generaciones.

Pruebas:

Para las pruebas se proporcionan conjuntos de datos en ficheros de texto que solo hay que sustituirlos en la clase launcher desde la declaración de la mochila hasta la ultima sentencia de añadir un objeto a la lista de objetos.

Para modificar la población o el número de generaciones hay que cambiar el segundo campo de cada bucle for que hay en la clase launcher, siendo el primer bucle el que determina la población y el segundo el que determina el número de generaciones.

El formato que se muestra en la consola es el siguiente:

- Durante la ejecución se va mostrando la última generación ejecutada, la media de puntuación obtenida y la máxima de esa generación, expresada como un porcentaje del espacio ocupado.
- Cuando ha finalizado la ejecución podemos observar los resultados de cada generación en detalle con el siguiente formato:
    1. Número de generación.
    2. Una línea para cada individuo en la que se muestra el modo de colocación, si la colocación ha sido válida, la puntuación de ese individuo, el vector de objetos escogidos y el vector de orden a introducir ese objeto en caso de estar escogido.
    3. Total de espacios ocupados por esa generación.
- Puntuaciones de cada generación de media y de mejor solución expresadas en espacios ocupados y en porcentaje ocupado.
- Mejor colocación, con formato individuo anteriormente explicado (modo, valida, puntos, escogidos, orden) y una ilustración de los huecos ocupados por esta colocación en la matriz mochila.

En la carpeta “Notas” hay una carpeta con algunos resultados obtenidos durante las pruebas y el conjunto de datos a introducir para probar el programa.

Joaquín Dotor Ruiz
bo0017

Posible mejora:

Como posible mejora se me ocurre distinguir los objetos en dos categorías, que serían, objetos principales y objetos para rellenar, siendo los primeros objetos grandes, y los segundos los que son pequeños y/o cuadrados, que se intentarían introducir los últimos para intentar aprovechar los pequeños espacios no ocupados.