Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Análisis de Algoritmos - Grupo 51

Profesora: Ing. Ana Lorena Valerio Solís

Segundo proyecto: Solución del problema de la mochila utilizando algoritmos genéticos y dinámicos

Estudiantes:

Roosevelt Alejandro Pérez González - 2022119527

José Andrés Lorenzo Segura - 2022136934

Arnold Jafeth Álvarez Rojas - 2023207970

Sede San Carlos

I Semestre, 2024

Introducción

El problema de la mochila es un desafío clásico en la optimización combinatoria, con aplicaciones relevantes en diversos campos como la logística, la gestión de inventarios y la planificación de recursos. En su esencia, implica la selección de un conjunto de elementos para maximizar el valor total, considerando una restricción de capacidad asociada a la capacidad de una mochila o contenedor. Resolver este problema se convierte en un objetivo primordial para empresas y organizaciones que buscan optimizar la asignación de recursos y mejorar la eficiencia operativa.
En el ámbito de los proyectos de optimización, los algoritmos genéticos y dinámicos emergen como enfoques poderosos para abordar el desafío de la mochila. Los algoritmos genéticos, inspirados en los principios de la evolución biológica, son técnicas de búsqueda que emplean conceptos como la selección natural, la mutación y la recombinación para explorar y encontrar soluciones óptimas en un espacio de búsqueda complejo. Estos algoritmos comienzan generando una población inicial de posibles soluciones, representadas como cromosomas, de manera aleatoria. A través de un proceso iterativo que implica operadores genéticos como la selección, la recombinación y la mutación, las soluciones evolucionan hacia configuraciones más óptimas y adaptadas al problema.
Por otro lado, los algoritmos dinámicos se presentan como una estrategia de programación que divide el problema de la mochila en subproblemas más pequeños y manejables. Esta técnica resuelve los subproblemas de manera recursiva, aprovechando la estructura óptima del problema para construir y actualizar eficientemente una tabla de soluciones. La clave de este enfoque radica en identificar la estructura subyacente del problema y utilizarla para descomponerlo en partes más simples y resolubles. Así, los algoritmos dinámicos pueden encontrar la solución óptima global de manera eficiente al evitar la repetición de cálculos y optimizar la búsqueda a través de una exploración sistemática del espacio de soluciones.
En el contexto del problema de la mochila, los algoritmos genéticos emergen como una herramienta poderosa para encontrar la combinación óptima de elementos que maximice el valor total dentro de la capacidad de la mochila. Estos algoritmos inician su proceso generando una población inicial de posibles soluciones, representadas como cromosomas, de forma aleatoria. Cada cromosoma representa una solución potencial, donde cada gen codifica la presencia o ausencia de un elemento en la mochila.
A medida que avanza el proceso iterativo, estos algoritmos aplican una serie de operadores genéticos para mejorar gradualmente las soluciones. La selección identifica las soluciones más prometedoras, las cuales son susceptibles de sobrevivir y reproducirse para la siguiente generación. Durante la recombinación, se combinan características de soluciones padres seleccionadas para producir descendencia con potencial para heredar características ventajosas. La mutación

introduce cambios aleatorios en los cromosomas, permitiendo la exploración de nuevas áreas del espacio de búsqueda.
Este ciclo iterativo de selección, recombinación y mutación conduce a la evolución de las soluciones hacia configuraciones más óptimas. Finalmente, después de un número predeterminado de generaciones o cuando se alcanza un criterio de parada, el algoritmo genético devuelve la mejor solución encontrada, que representa una combinación óptima de elementos para maximizar el valor dentro de la capacidad de la mochila.
Por otro lado, los algoritmos dinámicos ofrecen una perspectiva diferente para abordar el problema de la mochila. Estos algoritmos descomponen el problema en subproblemas más pequeños y resolubles, utilizando una estrategia recursiva para encontrar la solución óptima global. La clave de este enfoque radica en identificar la estructura subyacente del problema y utilizarla para construir una tabla de soluciones que se actualiza incrementalmente.

En el caso de la mochila, los algoritmos dinámicos construyen una tabla que almacena las soluciones óptimas para diferentes combinaciones de elementos y capacidades de mochila. A través de un proceso iterativo, estas tablas se llenan y actualizan con información sobre las soluciones óptimas para subconjuntos de elementos, permitiendo la determinación eficiente de la solución óptima global. Esta estrategia aprovecha la estructura recursiva del problema de la mochila para evitar la repetición de cálculos y encontrar rápidamente la mejor solución posible.

Análisis del problema

Para la resolución de este proyecto se llevó a cabo el uso de los algoritmos genéticos y los algoritmos dinámicos, sin embargo con esto surge la duda de “existen otros algoritmos que pueden funcionar como sustitutos a los algoritmos genéticos y dinámicos?”.
Investigando diversos documentos se llegó a 2 resultados que pueden sustituir funcionar como sustitutos a los algoritmos ya mencionados. El primero siendo el algoritmo de Greedy como sustituto para el algoritmo genético y el algoritmo de branch and bound para el algoritmo dinámico.

El algoritmo Greedy, o Voraz, es una estrategia heurística utilizada para abordar problemas de optimización, como el problema de la mochila. Su enfoque simple consiste en seleccionar en cada paso la opción que parece ser la mejor en ese momento, sin considerar cómo esta elección afectará las futuras decisiones. En el contexto del problema de la mochila, esto significa elegir el objeto con la mejor relación valor-peso y colocarlo en la mochila, repitiendo este proceso hasta que no haya más espacio o no queden más objetos por considerar.

Una de las ventajas más destacadas del algoritmo Greedy es su eficiencia y facilidad de implementación. Dado que solo requiere comparar las relaciones valor-peso de los objetos y seleccionar la mejor opción en cada paso, el algoritmo Greedy puede proporcionar soluciones rápidas para problemas de optimización combinatoria. Además, cuando se trata de objetos que pueden ser fraccionados, es decir, cuando es posible tomar una parte de un objeto en lugar de su totalidad, el algoritmo Greedy garantiza la solución óptima del problema de la mochila. Esto se debe a que puede distribuir los objetos de manera fraccionada para maximizar el valor total dentro de la capacidad de la mochila.

Sin embargo, la principal desventaja del algoritmo Greedy surge cuando los objetos no pueden ser fraccionados, lo que se conoce como el problema de la mochila 0-1. En esta situación, el algoritmo Greedy puede no proporcionar la solución óptima. Su enfoque de selección localmente óptima en cada paso puede no tener en cuenta combinaciones de elementos que conducirán a una solución global óptima. Esto se debe a que el algoritmo Greedy no realiza una exploración exhaustiva del espacio de soluciones y puede perder oportunidades de optimización al tomar decisiones basadas únicamente en información local. En consecuencia, aunque el algoritmo Greedy es eficiente y fácil de implementar, su falta de exhaustividad puede llevar a soluciones subóptimas en ciertos casos.

En la presentación sobre algoritmos voraces del Departamento de Ciencias de la Computación e I.A de la Universidad de Granada (s.f.), cuando se ordenan los elementos antes de aplicar el algoritmo Greedy, se puede garantizar que la selección en cada iteración del ciclo se realice en tiempo constante. Esto se debe a

que, al tener los elementos ordenados, la selección del elemento con la mejor relación valor-peso se vuelve una operación simple de acceso directo a la posición correspondiente en la lista ordenada. El ordenamiento de los elementos tiene una complejidad de O(nlogn), donde nn es el número de elementos en la lista, determinada por el algoritmo de ordenamiento utilizado, como Quicksort o Mergesort.

Con lo mencionado, y según Solís(2006) el cual menciona las ventajas y desventajas que poseen los algoritmos genéticos, se pueden abarcar los puntos cruciales por los que este algoritmo es una buena opción para ser implementado como sustituto del algoritmo genético como, por los siguientes factores:

A diferencia de los algoritmos genéticos, el algoritmo Greedy es mucho más simple de implementar y ejecutar. Para el problema de la mochila, especialmente cuando los objetos son fraccionables, proporciona soluciones óptimas de manera rápida y eficiente. Esto lo hace ideal para situaciones donde la velocidad y la simplicidad son cruciales.

El algoritmo Greedy sobresale en problemas donde los objetos pueden ser fraccionados, garantizando la solución óptima. Los algoritmos genéticos, aunque pueden abordar estos problemas, suelen requerir más tiempo y recursos para llegar a soluciones comparables.

Mientras que los algoritmos genéticos pueden sufrir de convergencia prematura y requieren técnicas adicionales para mantener la diversidad en la población, el algoritmo Greedy no tiene este problema, ya que su enfoque es determinista y directo.

Los algoritmos genéticos son más adecuados para problemas altamente complejos y de gran escala donde no existen soluciones analíticas claras y se requiere una exploración exhaustiva del espacio de soluciones. En contraste, el algoritmo Greedy es más adecuado para problemas de optimización combinatoria donde se busca una solución rápida y razonablemente buena.

Por otro lado, si se utiliza un heap ordenado por la estrategia de selección, el tiempo de inicialización del heap cae a O(n), ya que simplemente implica agregar los elementos al heap. Sin embargo, en cada selección, es necesario mantener la estructura del heap mediante la operación de "heapify". Esto significa que después de seleccionar un elemento, es necesario reorganizar el heap para garantizar que el elemento con la mayor prioridad (o menor, dependiendo del tipo de heap) esté en la raíz. La operación de heapify tiene una complejidad de O(logn) en promedio, lo que resulta en una complejidad total del algoritmo de O(nlogn).

Entre los métodos clásicos para la resolución de problemas de optimización entera, el algoritmo de Branch and Bound se destaca por su eficiencia relativa en encontrar

soluciones óptimas. Este método permite tratar el problema como si fuera lineal y continuo mediante la relajación de las variables, es decir, eliminando temporalmente las restricciones de integralidad. Esta relajación inicial constituye la raíz de un "árbol de búsqueda" que se expande creando subproblemas adicionales. Las variables de decisión, originalmente enteras y ahora continuas, son restringidas nuevamente para alcanzar valores enteros mediante un proceso de ramificación y acotamiento, tanto superior como inferior, de ahí el nombre Branch and Bound.

El objetivo es garantizar que las variables sean enteras mediante el establecimiento de nuevas cotas, lo que da lugar a valores óptimos locales de los subproblemas, conocidos como soluciones factibles. La mejor de estas soluciones se denomina "valor incumbente". Un beneficio clave del algoritmo Branch and Bound es su capacidad para descartar subproblemas infactibles o aquellos con soluciones peores que la incumbente, "podando" el árbol de exploración y acelerando la búsqueda del óptimo global.

Según Larrea Egaña (2020), entre las ventajas del algoritmo Branch and Bound se encuentran su capacidad para proporcionar una solución óptima y su flexibilidad para adaptarse a diferentes tipos de problemas de optimización. Además, el método es intuitivo y se puede mejorar continuamente mediante estrategias de poda y heurísticas más sofisticadas. Otro aspecto positivo es su capacidad para manejar problemas grandes y complejos de manera más eficiente que una búsqueda exhaustiva, gracias a su enfoque estructurado para explorar el espacio de soluciones.

La eficiencia del algoritmo Branch and Bound depende en gran medida del orden en que se exploran los subproblemas. Existen múltiples estrategias para la ramificación y selección de nodos, cada una con sus propias ventajas y desventajas.

Sin embargo, un inconveniente significativo del algoritmo es su naturaleza compleja y no determinística en términos computacionales, lo que puede llevar a un aumento exponencial en los tiempos de resolución a medida que crecen los requerimientos del problema. Esta dependencia del orden de generación y resolución de los subproblemas implica que, en problemas grandes y complejos, los tiempos de ejecución pueden crecer rápidamente, lo cual representa un desafío importante en términos de eficiencia y escalabilidad.

Según Larrea Egaña (2020), el algoritmo Branch and Bound resuelve problemas de optimización, como el de la mochila 0-1, dividiendo el problema original en subinstancias más pequeñas. En el caso de la mochila, esto implica evaluar recursivamente cada objeto, considerando si incluirlo o no en la mochila, y calcular el valor total sin exceder la capacidad máxima. La complejidad computacional de este enfoque es $O(n \cdot c)$, donde n es la cantidad de objetos y c es la capacidad máxima de la mochila (Martello & Toth, 1990). Este algoritmo ofrece una estrategia

eficiente para encontrar la solución óptima en problemas de optimización entera como el de la mochila.

Con lo presentado, y con lo que menciona Lifeder(2020) sobre las ventajas y desventajas de los algoritmos dinámicos, se pueden abarcar los puntos cruciales por los que este algoritmo es una buena opción para ser implementado como sustituto del algoritmo dinámico debido a los siguientes factores:

A diferencia de la programación dinámica, que puede requerir grandes cantidades de memoria para almacenar todos los subproblemas, Branch and Bound se enfoca en los subproblemas más prometedores y descarta los infactibles, lo que puede resultar en un uso más eficiente de la memoria.

Branch and Bound es altamente adaptable y puede ajustarse a diferentes problemas de optimización mediante la implementación de diferentes estrategias de poda y heurísticas, lo que puede mejorar su eficiencia en comparación con la programación dinámica en ciertos escenarios.

El enfoque estructurado de Branch and Bound para explorar el espacio de soluciones mediante la relajación y ramificación permite encontrar la solución óptima de manera más eficiente, especialmente en problemas de optimización entera como el de la mochila 0-1.

Aunque puede tener una complejidad exponencial, el uso de estrategias adecuadas de poda y la relajación inicial pueden hacer que Branch and Bound sea más manejable y escalable en comparación con la programación dinámica, que puede sufrir problemas de memoria y redundancia.

Solución del problema

Inicialización del proyecto

Diagrama con las estructuras realizadas

Programación dinámica:
Matriz:
Item1: Peso = 1, Valor = 10
Item2: Peso = 2, Valor = 20
Item3: Peso = 3, Valor = 30
Capacidad de la mochila: 4.

	1	2	3	4
Item 1	10	10	10	10
Item 2	10	20	30	30
Item 3	10	20	30	40

Ítems Seleccionados: Item1 y Item3

Peso Total: 4

Valor Total: 40

Array:

Items
[0] Item (peso,valor)
[1] Item (peso,valor)
[2] Item (peso,valor)
[3] Item (peso,valor)
[4] Item (peso,valor)
[5] Item (peso,valor)
[6] Item (peso,valor)
[n-1] Item (peso,valor)

Algoritmo Genético:

Individuos	0	1	2	3	4	…	n-1
0	1	0	1	0	1	…	0
1	0	1	0	1	0	…	1
2	1	1	0	0	1	…	0
3	0	0	1	1	0	…	1
…	…	…	…	…	…	…	…
19	1	0	0	1	0	…	1

Individuos	Aptitud
0	1200
1	1150
2	1300
3	1100
…	…
19	1250

Individuos	0	1	2	3	4	…	n-1
Mejor individuo	1	0	1	0	1	…	0
1	0	1	0	1	0	…	1
2	1	1	0	0	1	…	0
3	0	0	1	1	0	…	1
…	…	…	…	…	…	…	…
19	1	0	0	1	0	…	1

Clases para ambos:

Item
Peso int Valor int
getPeso() int getValor() int

Mochila
Capacidad int Items Item[] pesoTotal int valorTotal int
getCapacidad() int getItems() Item[] setPesoTotal(peso int) void setValorTotal(valor int) void

Cuál es el cruce realizado

Pseudocódigos de los algoritmos

Pseudocódigo del algoritmo genético

```
// Inicializar constantes
TAMANO_POBLACION = 20
TASA_MUTACION = 0.05
NUMERO_GENERACIONES = 20
PENALIDAD = 1000.0

FUNCION resolverMochilaGA(mochila)
    // Obtener la capacidad de la mochila
    capacidad = obtenerCapacidad(mochila)
    // Obtener los ítems disponibles
    items = obtenerItems(mochila)
    // Número de ítems
    n = longitud(items)

    // Inicializar la población de manera aleatoria
    INICIALIZAR POBLACION
    PARA i DESDE 0 HASTA TAMANO_POBLACION - 1 HACER
        PARA j DESDE 0 HASTA n - 1 HACER
            poblacion[i][j] = valor aleatorio (0 o 1)

    mejorIndividuo = NULL
    mejorAptitud = -INFINITO

    // Almacenar las mejores poblaciones
    mejoresPoblaciones = array de tamaño TAMANO_POBLACION x (n + 1)

    PARA generacion DESDE 0 HASTA NUMERO_GENERACIONES - 1 HACER
        // Evaluar la aptitud de cada individuo en la población
        EVALUAR APTITUD
        PARA i DESDE 0 HASTA TAMANO_POBLACION - 1 HACER
            aptitud[i] = evaluarAptitud(poblacion[i], items, capacidad)
            SI aptitud[i] > mejorAptitud ENTONCES
                mejorAptitud = aptitud[i]
                mejorIndividuo = copia(poblacion[i])

        // Guardar las mejores poblaciones de la generación actual
        GUARDAR MEJORES POBLACIONES
        PARA i DESDE 0 HASTA TAMANO_POBLACION - 1 HACER
            mejoresPoblaciones[i] = poblacion[i] + [aptitud[i]]
```

```
        // Selección de padres para la nueva población
        SELECCIÓN
        nuevaPoblacion = array de tamaño TAMANO_POBLACION x n
        PARA i DESDE 0 HASTA TAMANO_POBLACION - 1 HACER
            padre1 = seleccionar(aptitud)
            padre2 = seleccionar(aptitud)
            descendiente = cruzar(poblacion[padre1], poblacion[padre2], items, capacidad)
            nuevaPoblacion[i] = descendiente

            IMPRIMIR "Padre 1, Padre 2, Descendiente" con sus puntuaciones

        // Aplicar mutación dirigida y evitar duplicados
        MUTACIÓN DIRIGIDA Y EVITAR DUPLICADOS
        PARA i DESDE 0 HASTA TAMANO_POBLACION - 1 HACER
            SI random < TASA_MUTACION ENTONCES
                mutarDirigida(nuevaPoblacion[i], items, capacidad)
            PARA j DESDE 0 HASTA i - 1 HACER
                SI nuevaPoblacion[i] ES IGUAL A nuevaPoblacion[j] ENTONCES
                    mutar(nuevaPoblacion[i])

        // Aplicar elitismo, manteniendo el mejor individuo de la generación anterior
        ELITISMO
        SI mejorIndividuo != NULL ENTONCES
            nuevaPoblacion[0] = mejorIndividuo

        poblacion = nuevaPoblacion

    // Ordenar las mejores poblaciones por puntuación
    ORDENAR MEJORES POBLACIONES POR PUNTUACIÓN
    ordenar(mejoresPoblaciones, por puntuación descendente)

    // Imprimir las 5 mejores poblaciones y sus puntuaciones
    SI TAMANO_POBLACION == 3 ENTONCES
        IMPRIMIR "Las 3 mejores poblaciones"
    SINO
        IMPRIMIR "Las 5 mejores poblaciones"

    // Construir la lista de ítems seleccionados en la mejor solución encontrada
    CONSTRUIR LISTA DE ITEMS SELECCIONADOS
    pesoTotalFinal = 0
    valorTotalFinal = 0
    count = contar elementos en mejorIndividuo == 1

    itemsSeleccionados = array de tamaño count
    index = 0
    PARA i DESDE 0 HASTA longitud(mejorIndividuo) - 1 HACER
        SI mejorIndividuo[i] == 1 ENTONCES
```

```
            itemsSeleccionados[index++] = items[i]
            pesoTotalFinal += items[i].peso
            valorTotalFinal += items[i].valor

    resultado = nueva Mochila(capacidad, itemsSeleccionados)
    resultado.setPesoTotal(pesoTotalFinal)
    resultado.setValorTotal(valorTotalFinal)

    RETORNAR resultado

// Función para evaluar la aptitud de un individuo
FUNCION evaluarAptitud(individuo, items, capacidad)
    pesoTotal = 0
    valorTotal = 0
    PARA i DESDE 0 HASTA longitud(individuo) - 1 HACER
        SI individuo[i] == 1 ENTONCES
            pesoTotal += items[i].peso
            valorTotal += items[i].valor
    SI pesoTotal <= capacidad ENTONCES
        RETORNAR valorTotal
    SINO
        RETORNAR valorTotal - PENALIDAD * (pesoTotal - capacidad)

// Función para seleccionar un padre basado en la aptitud
FUNCION seleccionar(aptitud)
    totalAptitud = suma(aptitud)
    SI totalAptitud <= 0 ENTONCES
        RETORNAR índice aleatorio en aptitud

    punto = random * totalAptitud
    suma = 0
    PARA i DESDE 0 HASTA longitud(aptitud) - 1 HACER
        suma += aptitud[i]
        SI suma > punto ENTONCES
            RETORNAR i
    RETORNAR longitud(aptitud) - 1

// Función para cruzar dos padres y producir un descendiente
FUNCION cruzar(padre1, padre2, items, capacidad)
    puntoCruce1 = random en rango(padre1.length)
    puntoCruce2 = random en rango(padre1.length - puntoCruce1) + puntoCruce1
    descendiente = array de tamaño padre1.length
    descendienteOrigen = ""

    PARA i DESDE 0 HASTA puntoCruce1 - 1 HACER
        descendiente[i] = padre1[i]
        descendienteOrigen += "P1 "
    PARA i DESDE puntoCruce1 HASTA puntoCruce2 - 1 HACER
```

```
        descendiente[i] = padre2[i]
        descendienteOrigen += "P2 "
    PARA i DESDE puntoCruce2 HASTA longitud(padre2) - 1 HACER
        descendiente[i] = padre1[i]
        descendienteOrigen += "P1 "

    corregir(descendiente, items, capacidad)
    IMPRIMIR "Origen descendiente: " + descendienteOrigen

    RETORNAR descendiente

// Función para mutar un individuo
FUNCION mutar(individuo)
    PARA i DESDE 0 HASTA longitud(individuo) - 1 HACER
        SI random < TASA_MUTACION ENTONCES
            individuo[i] = 1 - individuo[i]

// Función para aplicar una mutación dirigida a un individuo
FUNCION mutarDirigida(individuo, items, capacidad)
    intento = 0
    MIENTRAS intento < 10 HACER
        index = random en rango(individuo.length)
        individuo[index] = 1 - individuo[index]
        pesoTotal = 0
        PARA i DESDE 0 HASTA longitud(individuo) - 1 HACER
            SI individuo[i] == 1 ENTONCES
                pesoTotal += items[i].peso
        SI pesoTotal <= capacidad ENTONCES
            RETORNAR
        SINO
            individuo[index] = 1 - individuo[index]
        intento += 1

// Función para corregir un individuo que excede la capacidad de la mochila
FUNCION corregir(individuo, items, capacidad)
    pesoTotal = 0
    PARA i DESDE 0 HASTA longitud(individuo) - 1 HACER
        SI individuo[i] == 1 ENTONCES
            pesoTotal += items[i].peso
    MIENTRAS pesoTotal > capacidad HACER
        index = random en rango(individuo.length)
        SI individuo[index] == 1 ENTONCES
            individuo[index] = 0
            pesoTotal -= items[index].peso
```

Pseudocódigo de Dinámico

```
INICIAR FUNCION resolverMochilaDP(mochila)
    // Obtener la capacidad de la mochila
    capacidad = mochila.obtenerCapacidad()
    // Obtener los ítems disponibles
    items = mochila.obtenerItems()
    // Número de ítems
    n = longitud(items)

    // Inicializar la tabla dp con dimensiones (n + 1) x (capacidad + 1) a 0
    dp = matriz de tamaño (n + 1) x (capacidad + 1) inicializada a 0

    // Llenar la tabla dp
    PARA i DESDE 1 HASTA n HACER
        PARA w DESDE 0 HASTA capacidad HACER
            // Si el peso del ítem actual es menor o igual al peso máximo permitido
            SI items[i - 1].obtenerPeso() <= w ENTONCES
                // Llenar la celda actual con el máximo valor posible
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - items[i - 1].obtenerPeso()] + items[i -
1].obtenerValor())
            SINO
                // Si el ítem no cabe, se mantiene el valor anterior
                dp[i][w] = dp[i - 1][w]

        // Imprimir los resultados cada 5 etapas o al final
        SI i % 5 == 0 O i == n ENTONCES
            imprimirResultados(dp, i, items, capacidad)

    // Encontrar los ítems seleccionados
    w = capacidad
    valorTotal = dp[n][w]
    pesoTotal = 0
    seleccionados = arreglo de tamaño n inicializado a falso

    // Retroceder para encontrar los ítems que forman la solución óptima
    PARA i DESDE n HASTA 1 HACER
        SI valorTotal > 0 Y valorTotal != dp[i - 1][w] ENTONCES
            seleccionados[i - 1] = verdadero
            pesoTotal += items[i - 1].obtenerPeso()
            valorTotal -= items[i - 1].obtenerValor()
            w -= items[i - 1].obtenerPeso()

    // Filtrar los ítems seleccionados
    itemsSeleccionados = filtrar(items, seleccionados)

    // Crear el objeto Mochila resultado con los ítems seleccionados
    resultado = nueva Mochila(mochila.obtenerCapacidad(), itemsSeleccionados)
```

```
    resultado.setPesoTotal(pesoTotal)
    resultado.setValorTotal(dp[n][capacidad])

    // Retornar la solución
    RETORNAR resultado

FIN FUNCION resolverMochilaDP

INICIAR FUNCION imprimirResultados(dp, etapa, items, capacidad)
    // Imprimir resultados intermedios
    IMPRIMIR "Resultados en la etapa " + etapa + ":"
    valorTotal = dp[etapa][capacidad]
    pesoTotal = 0
    seleccionados = arreglo de tamaño etapa inicializado a falso

    w = capacidad
    // Retroceder para encontrar los ítems seleccionados en la etapa actual
    PARA i DESDE etapa HASTA 1 HACER
        SI valorTotal > 0 Y valorTotal != dp[i - 1][w] ENTONCES
            seleccionados[i - 1] = verdadero
            pesoTotal += items[i - 1].obtenerPeso()
            valorTotal -= items[i - 1].obtenerValor()
            w -= items[i - 1].obtenerPeso()

    // Imprimir valor total y peso total
    IMPRIMIR "Valor total: " + dp[etapa][capacidad]
    IMPRIMIR "Peso total: " + pesoTotal

    // Imprimir los nombres de los ítems seleccionados
    IMPRIMIR "Items seleccionados: "
    PARA i DESDE 0 HASTA longitud(seleccionados) HACER
        SI seleccionados[i] ENTONCES
            IMPRIMIR items[i].obtenerNombre() + " "
    IMPRIMIR nueva línea
    IMPRIMIR nueva línea

FIN FUNCION imprimirResultados
```

Descripción del tipo de mutación realizada

Descripción de la estrategia dinámica

El algoritmo de programación dinámica implementado en el código proporcionado resuelve el problema de la mochila 0-1 de manera eficiente. El enfoque se basa en construir una matriz bidimensional dp, donde dp[i][w] representa el valor máximo que se puede obtener con los primeros i elementos y una capacidad máxima de w. Esta matriz permite descomponer el problema en subproblemas más pequeños y resolverlos de manera iterativa.

Para explicar esto mejor se va a segmentar la información en varios puntos, los cuales son:

Inicialización:

Primero, se determina la capacidad máxima de la mochila y se obtiene el conjunto de elementos disponibles, junto con sus respectivos pesos y valores. La matriz dp se inicializa con dimensiones (n + 1) x (capacidad + 1), donde n es el número de elementos. Cada celda de la matriz inicialmente se llena con ceros, lo que representa que sin elementos o con capacidad cero, el valor máximo alcanzable es cero.

Llenado de la Tabla dp:

El llenado de la matriz dp se realiza a través de dos bucles anidados. El bucle exterior recorre cada elemento disponible, mientras que el bucle interior recorre cada posible capacidad de la mochila desde 0 hasta la capacidad máxima. En cada iteración, el algoritmo verifica si el peso del elemento actual es menor o igual a la capacidad actual. Si es así, el valor de dp[i][w] se actualiza con el máximo entre no incluir el elemento actual (dp[i - 1][w]) e incluirlo (dp[i - 1][w - items[i - 1].getPeso()] + items[i - 1].getValor()). Esto asegura que se selecciona la opción que maximiza el valor alcanzable. Si el peso del elemento es mayor que w, simplemente se copia el valor de dp[i - 1][w], lo que significa que el elemento no puede ser incluido dado que excede la capacidad actual.

Construcción de la Solución Óptima:

Una vez que la matriz dp se ha llenado completamente, el valor máximo alcanzable con todos los elementos y la capacidad total se encuentra en dp[n][capacidad]. A partir de este punto, el algoritmo procede a construir la solución óptima rastreando hacia atrás desde dp[n][capacidad]. Comenzando desde el último elemento y la capacidad máxima, el algoritmo verifica si el valor en dp[i][w] difiere de dp[i - 1][w]. Si hay una diferencia, esto indica que el elemento i-ésimo fue incluido en la solución óptima. El elemento se marca como seleccionado, y tanto el valor total como la capacidad se actualizan restando el valor y el peso del elemento incluido respectivamente. Este proceso se repite hasta que todos los elementos han sido considerados o hasta que el valor total se reduce a cero.

Almacenamiento y Uso de Resultados Previos:

La matriz dp actúa como una memoria para almacenar los resultados de los subproblemas previamente calculados. Cada entrada dp[i][w] se basa en los resultados almacenados de los subproblemas correspondientes a los primeros i-1 elementos y capacidades hasta w. Esto permite que el algoritmo construya la solución óptima de manera eficiente, utilizando resultados previos para calcular los valores óptimos para capacidades mayores. La reutilización de estos resultados evita cálculos redundantes y mejora significativamente la eficiencia del algoritmo en comparación con una búsqueda exhaustiva.

Análisis de Resultados

Resultados finales

El proyecto en su gran mayoría está completo, se logró realizar los dos algoritmos solicitados, de una manera que ambos cumplen con la tarea solicitada de una manera bastante eficiente; sin embargo, hubo requerimientos que, por tiempo, no se logró realizar, como por ejemplo no se puede realizar la 4.ª consulta solicitada que se refiere a la impresión de las mutaciones aplicadas junto a su puntuación.
El algoritmo genético, a pesar de dar resultados considerablemente buenos, en muchas ocasiones se queda por detrás del algoritmo dinámico que en la mayoría de las pruebas realizadas por los estudiantes para comprobar su fidelidad siempre daba la mejor solución posible al problema de la mochila.
También, se encontró un error a la hora de imprimir las puntuaciones de los padres y descendientes de los cromosomas. Sin embargo, no se encontró una solución, se probó cambiar los tipos de datos, de float a Double, pero no mejoró. En consulta con la profesora, se llegó a la conclusión que puede ser una mala inicialización de alguna variable que esté generando este dato sucio. Sin embargo, es importante mencionar que ese dato no afecta a la funcionalidad del sistema, por lo menos no de una manera notable.
En cuanto a las mediciones, no se logró realizar la medición analítica, debido a que los algoritmos son demasiados extensos y complejos y se decidió poner esos esfuerzos en otras areas del proyecto.

El resto del proyecto está completo y funcional. El usuario podrá correr ambos algoritmos mediante un menú para que este pueda elegir la cantidad de objetos que desea analizar.

Medición Empírica

Cantidad de Objetos	Genético				
	Asignaciones	Comparaciones	Cantidad de lineas ejecutadas	Tiempo de ejecucion (ms)	Memoria Usada bits
5	2437	4508	6945	32	118526
10	9454	26251	35705	103	531974
20	16988	88012	105000	302	1732550
30	27469	190326	217795	344	4019486
40	56397	337309	393706	409	8331310
50	71859	517185	589044	886	13937766

Tabla 1: Medición empírica del algoritmo Genético

Cantidad de Objetos	Dinamico				
	Asignaciones	Comparaciones	Cantidad de lineas ejecutadas	Tiempo de ejecucion (ms)	Memoria Usada
5	134	282	416	1	2060
10	696	1685	2381	2	19842
20	2244	5332	7576	5	69944
30	5494	12906	18400	7	173002
40	11946	27021	38967	11	376756
50	22464	50176	72640	13	712226

Tabla 2: Medición empírica del algoritmo Dinámico

En las Tablas 1 y 2, se evidencia claramente la superioridad del algoritmo dinámico sobre el genético en varios aspectos críticos: cantidad de asignaciones, número de comparaciones, líneas ejecutadas, tiempo de ejecución y uso de memoria. Por ejemplo, con solo cinco objetos, el algoritmo genético realiza hasta 18 veces más asignaciones que su contraparte dinámica, lo que nos indica una mayor eficiencia del último en este parámetro. Este dato sugiere que el algoritmo dinámico podría escalar mejor con un aumento en la cantidad de ítems, y efectivamente así es. Al incrementar el número de objetos a 50, el algoritmo dinámico efectúa solo 22,464 asignaciones frente a las más de 70,000 del genético.

A su misma vez podemos observar que el algoritmo dinámico presenta un mejor rendimiento en términos de comparaciones y tiempo de ejecución. En comparación, el algoritmo genético realiza cerca de 16 veces más comparaciones y a su misma vez el tiempo de ejecución tiene un comportamiento similar. Para este problema específico, el algoritmo dinámico resulta ser mucho más eficiente que el genético.

En términos de uso de memoria, el algoritmo dinámico también demuestra ser más eficiente. Al comparar el desempeño de ambos algoritmos con 50 objetos, se observa que el genético consume más de un megabyte de memoria, mientras que el dinámico se mantiene por debajo de ese umbral. Estos resultados subrayan la superioridad del algoritmo dinámico en eficiencia de recursos, especialmente en escenarios con un mayor número de objetos.

Tamaño de objetos	Factor Talla	Factor Asignación	Factor Comparaciones	Factor lineas ejecutadas	Factor Tiempo Ejecución
10/5	2	3,87935987	5,82320319	5,14110871	3,21875
20/10	2	1,79691136	3,35271037	2,9407646	2,93203883
30/20	0,66666667	1,61696492	2,16250057	2,0742381	1,13907285
40/30	0,75	2,05311442	1,77226968	1,80769072	1,18895349
50/30	0,6	2,61600349	2,71736389	2,70457999	2,5755814
50/5	10	29,4866639	114,726043	84,8155508	27,6875

Tabla 3: Factor de crecimiento del algoritmo Genético

Tamaño de objetos	Factor Talla	Factor Asignación	Factor Comparaciones	Factor lineas ejecutadas	Factor Tiempo Ejecución
10/5	2	5,19402985	5,9751773	5,72355769	2
20/10	2	3,22413793	3,16439169	3,18185636	2,5
30/20	0,66666667	2,4483066	2,42048012	2,42872228	1,4
40/30	0,75	2,17437204	2,09367736	2,11777174	1,57142857
50/30	0,6	4,08882417	3,88780412	3,94782609	1,85714286
50/5	10	167,641791	177,929078	174,615385	13

Tabla 4: Factor de crecimiento del algoritmo Dinámico

En la tabla que muestra el factor de crecimiento del algoritmo Genético, se observa que este parece tener una complejidad cuadrática. Esto expone que, aunque el algoritmo Genético utiliza más recursos en comparación con el algoritmo Dinámico, su complejidad sigue siendo bastante aceptable. A pesar de la mayor demanda de recursos, el rendimiento temporal del algoritmo Genético sigue siendo eficiente. De hecho, como se destacó en la tabla 1, donde se presentaron los tiempos de ejecución, ninguna de las pruebas superó el segundo de duración. Esto sugiere que, a pesar de su complejidad, el algoritmo Genético puede resolver problemas en tiempos razonables, manteniéndose dentro de límites de rendimiento prácticos y manejables.
En cuanto al algoritmo Dinámico, a partir del factor de crecimiento de este, podemos observar que parece tener una complejidad cuadrática. Aunque podría parecer que su rendimiento es peor que el del algoritmo genético, recordemos que la programación dinámica busca siempre la mejor solución posible, lo que puede hacerla parecer menos eficiente en algunos aspectos. Sin embargo, en ninguna de las pruebas realizadas se observó este problema. En todas las pruebas, el algoritmo Dinámico demostró ser el mejor.

De hecho, si se observan las tablas 5 y 6, se podrá observar cómo el algoritmo dinámico, supera en eficiencia al genético.

Clasificación de los algoritmos

Notación Genético

Clasificación en notación O Grande según sus comparaciones, asignaciones, líneas ejecutadas y tiempo de ejecución. Contemplando la cantidad de ítems:

Usar la notación O	

Clasificación del comportamiento de las **asignaciones**	O(n^2)
Clasificación del comportamiento de las **comparaciones**	O(n^2)
Clasificación del comportamiento de las **líneas ejecutadas**	O(n^2)
Clasificación del comportamiento en el **tiempo de ejecución**	O(n^2)

Notación Dinámico

Clasificación en notación O Grande según sus comparaciones, asignaciones, líneas ejecutadas y tiempo de ejecución. Contemplando la cantidad de items:

Usar la notación O	O(n^2)
Clasificación del comportamiento de las **asignaciones**	O(n^2)
Clasificación del comportamiento de las **comparaciones**	O(n^2)
Clasificación del comportamiento de las **líneas ejecutadas**	O(n^2)
Clasificación del comportamiento en el **tiempo de ejecución**	O(n^2)

Instrucciones ejecutadas

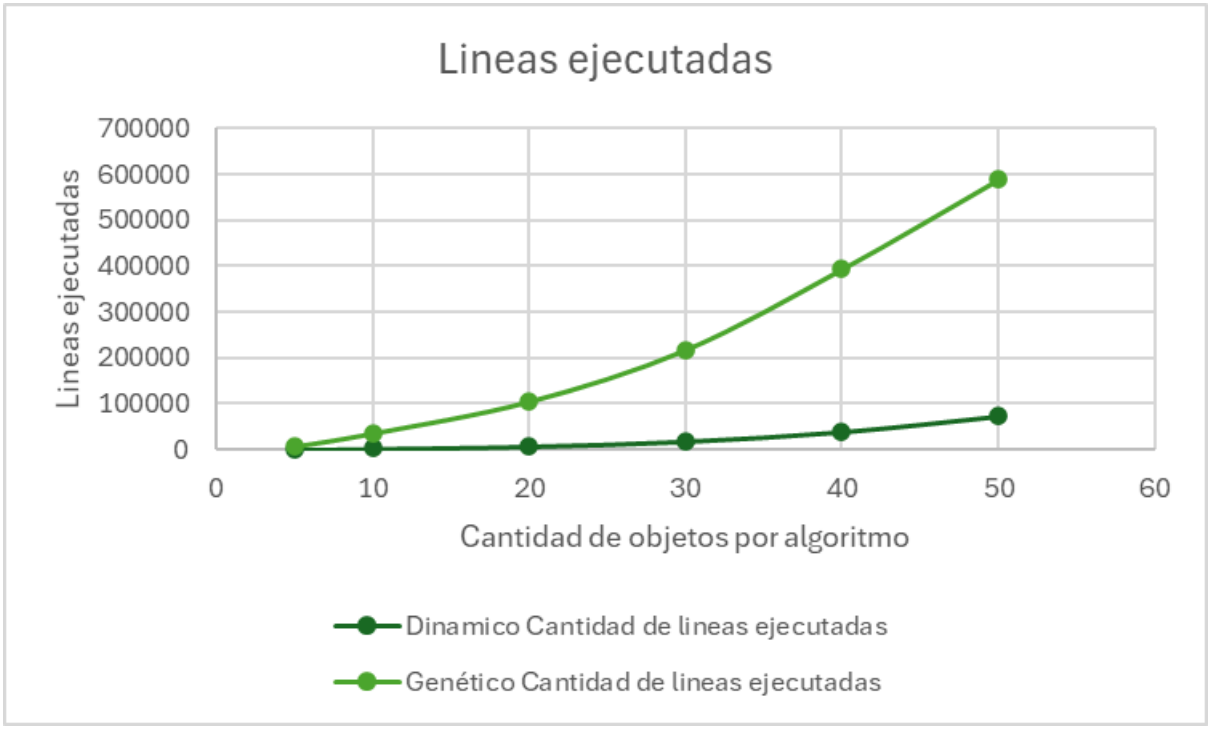

Tabla 5: Líneas ejecutadas

Como se mencionó anteriormente, las líneas ejecutadas por el algoritmo genético superan ampliamente las del algoritmo dinámico, un aspecto que queda evidenciado con mayor claridad en este gráfico. Esta disparidad refleja la diferencia en la complejidad y la cantidad de operaciones realizadas por cada algoritmo durante la ejecución.

Memoria consumida

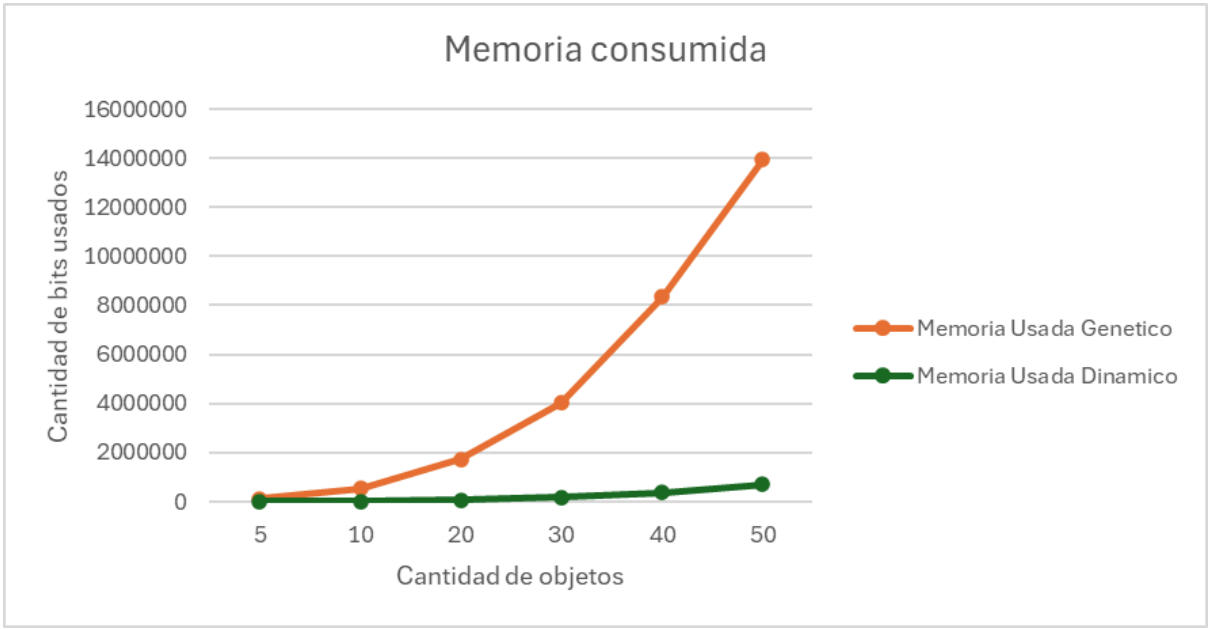

Tabla 6: Memoria consumida

Conclusiones

¿Cuál estrategia es más eficiente?

Para abordar este problema, la estrategia más eficiente es emplear el algoritmo dinámico. Este algoritmo supera al algoritmo genético en varios aspectos críticos, incluyendo la cantidad de asignaciones y comparaciones realizadas, el uso de memoria, y el tiempo de ejecución. En términos de asignaciones y comparaciones, el algoritmo dinámico realiza significativamente menos operaciones, lo que se traduce en un uso más eficiente de los recursos del sistema. Además, su consumo de memoria es considerablemente menor, lo que lo hace ideal para entornos con recursos limitados o entornos con grandes cantidades de datos.

El tiempo de ejecución es otro factor clave donde el algoritmo dinámico demuestra su superioridad. Al resolver el problema más rápidamente, permite una respuesta más ágil y eficiente, lo cual es crucial en aplicaciones donde el tiempo es un factor determinante. Además, el algoritmo dinámico no solo es más rápido y económico en términos de recursos, sino que también tiende a obtener mejores resultados en la mayoría de los casos. Esta consistencia en el rendimiento lo convierte en la opción preferida para resolver este tipo de problemas, asegurando las mejores soluciones de manera más eficiente y confiable.

¿Cuál algoritmo se hace más eficiente conforme crece la talla de crecimiento?

Según las tablas elaboradas, se puede observar un resultado sorprendente: a pesar de las expectativas iniciales, el algoritmo genético resulta ser el más eficiente cuando se consideran diversos factores de rendimiento. Estos factores incluyen el número de asignaciones y comparaciones realizadas, las líneas de código ejecutadas y el tiempo total de ejecución.

Aunque pueda parecer contraintuitivo, los datos muestran que, a medida que aumenta la cantidad de objetos que se intentan ordenar en la mochila, el algoritmo genético no solo mejora en términos de tiempo de ejecución, sino que también maximiza el valor añadido. Esto se debe a la naturaleza heurística del algoritmo genético, que le permite explorar una mayor cantidad de soluciones potenciales de manera más rápida y efectiva en comparación con el algoritmo dinámico.

El algoritmo genético, con su capacidad para realizar un mayor número de iteraciones y adaptarse mejor a los cambios en el tamaño y complejidad del problema, demuestra una eficiencia superior en contextos donde la escalabilidad es crucial. Mientras que el algoritmo dinámico es robusto y eficiente en problemas de menor escala, el genético muestra su verdadero potencial cuando se enfrenta a tareas más complejas y de mayor envergadura. Así, el algoritmo genético no solo proporciona soluciones rápidas, sino que también logra

optimizar el valor obtenido, lo que lo convierte en una opción destacada para problemas de optimización con grandes conjuntos de datos.

Recomendaciones

Es crucial realizar mediciones analíticas precisas para obtener una referencia sólida al analizar datos relacionados con el desempeño de algoritmos o sistemas computacionales. Estas mediciones proporcionan una base empírica para comprender el comportamiento del sistema y evaluar su eficiencia en diferentes situaciones.

Cuando se realizan las mediciones analíticas, es esencial seguir un enfoque sistemático y detallado. Esto implica establecer protocolos claros para recopilar datos, asegurándose de que las mediciones se realicen de manera consistente y precisa en diferentes ejecuciones y escenarios.

En casos donde los números de puntuación pueden salir incorrectos o fluctuar significativamente, es fundamental realizar análisis adicionales para identificar y abordar posibles fuentes de error. Esto puede implicar el investigar posibles cuellos de botella de rendimiento y realizar pruebas adicionales para validar los resultados.

Importante, recordarle al profesor, que en cada ejecución del problema, este tarda unos segundos para que el usuario pueda leer la información correctamente, no es que los algoritmos tarden todo ese tiempo para dar la solución.

Literatura citada

Departamento de Ciencias de la Computación e I.A, Universidad de Granada. Algoritmos Greedy: Análisis y diseño de algoritmos. Recuperado de https://elvex.ugr.es/decsai/algorithms/slides/4%20Greedy.pdf

Larrea Egaña, P. I. (2020, agosto 28). Desarrollo de algoritmo Branch & Bound en paralelo para la resolución de problemas de optimización combinatorial [Memoria de pregrado, Universidad Técnica Federico Santa María, Departamento de Industrias]. Extraído del repositorio digital USM recuperado de content (usm.cl)

Lifeder. (14 de marzo de 2020). Programación dinámica: características, ejemplo, ventajas, desventajas. Recuperado de: https://www.lifeder.com/programacion-dinamica/.

Rubio Solís (2006) propuso un método innovador para el control difuso autosintonizable de voltaje en generadores sincrónicos utilizando la búsqueda tabú y algoritmos genéticos (págs. 66-68).

Bitácora y minutas

Bitácora de Actividades

1-Nombre del Estudiante: **Arnold Jafeth Alvarez Rojas**
Fecha: 17/05/24
Actividad: Búsqueda de artículos
Duración: 2:30 horas
Descripción de la actividad: Se buscaron artículos sobre los algoritmos genético y dinámico

2-Nombre del Estudiante: **Arnold Jafeth Alvarez Rojas**
Fecha: 24/05/24
Actividad: Creación del análisis del problema
Duración: 3 horas
Descripción de la actividad:
Se indago por google scholar, google books y otros medios para poder generar la información para el análisis del problema
Observaciones:
Se logró generar la mitad de la información faltando únicamente la comparativa de las ventajas y desventajas de los algoritmos dados con los propuestos como sustitutos.

3-Nombre del Estudiante: **Arnold Jafeth Alvarez Rojas**
Fecha: 26/05/24
Actividad: Finalizar el análisis del problema y comenzar con solución del problema
Duración: 2:30 horas
Descripción de la actividad:
Se realizó una indagación por google scholar, google books y otros medios para poder generar la información faltante para el análisis del problema y se comenzó a analizar el código de programación dinámica para explicar su funcionamiento.
Observaciones:
Se logró generar la información faltante para análisis del problema y se creó la base para la explicación de la programación dinámica.

4-Nombre del Estudiante: **Arnold Jafeth Alvarez Rojas**
Fecha: 03/06/24
Actividad: Finalizar explicación de la programación dinámica
Duración: 1 horas
Descripción de la actividad:

Se realizó una indagación por google scholar, google books y otros medios para poder generar la información faltante para la explicación de la programación dinámica.
Observaciones:
Se logró generar la información faltante para la programación dinámica.

5-Nombre del Estudiante: **Arnold Jafeth Alvarez Rojas**
Fecha: 04/06/24
Actividad: Finalizar documento del proyecto
Duración: 6 horas
Descripción de la actividad:
Se realizó una indagación por google scholar, google books y otros medios para poder, a parte de analizar los códigos y los resultados obtenidos para finalizar la creación de los puntos faltantes.

1- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 10/05/2024
Actividad: Búsqueda de artículos
Duración: 2:30 horas
Descripción de la actividad: Se buscaron artículos sobre los algoritmos genético y dinámico

2- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 17/05/2024
Actividad: Se empezaron los códigos genético y dinámico Duración: 3 horas
Descripción de la actividad: Se realizó el codigo dinamico casi completo y el genético está pendiente de arreglar

3- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 26/05/2024
Actividad: Se realiza un poco de la documentación y se continúa arreglando el código genético
Duración: 5 horas
Descripción de la actividad: Se le hicieron varias modificaciones al algoritmo genético pero hay que arreglar las mutaciones, verificar porque siempre los tamaños sobrepasan la mochila y se mostraron los cromosomas del genetico

4- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 30/05/2024
Actividad: Se arreglaron las mutaciones en el algoritmo genético

Duración: 3 horas
Descripción de la actividad: Se descubrió que la forma de hacer las mutaciones era errónea y solo hacia ciertos cambios a la hora de mutar

5- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 31/05/2024
Actividad: Se arreglo el problema de la mochila ya que el algoritmo favoreció el valor por encima del peso y se hicieron las consultas del algoritmo dinámico
Duración: 5 horas
Descripción de la actividad: Se realizaron todas las consultas del algoritmo dinámico solo quedó 1 la cual no se está seguro si está funcionando.

6- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 02/06/2024
Actividad: Se realizaron todas las consultas del algoritmo genético y se arreglo la última consulta del algoritmo dinámico
Duración: 5 horas

7- Nombre del Estudiante: **José Andrés Lorenzo Segura**
Fecha: 04/06/2024
Actividad: Se terminó la documentación
Duración: 5 horas

1- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 10/05/2024
Actividad: Se buscó algunos artículos
Duración: 2:30 horas
Descripción de la actividad: Se buscaron artículos sobre los algoritmos genético y dinámico

2- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 17/05/2024
Actividad: Se empezaron los códigos genético y dinámico
Duración: 3 horas
Descripción de la actividad: Se realizó el código dinámico casi completo y el genético está pendiente de arreglar

3- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 26/05/2024
Actividad: Se realiza un poco de la documentación y se continúa arreglando el código genético
Duración: 5 horas
Descripción de la actividad: Se le hicieron varias modificaciones al algoritmo genético, pero hay que arreglar las mutaciones, verificar porque siempre los tamaños sobrepasan la mochila y se mostraron los cromosomas del genético

4- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 30/05/2024
Actividad: Se arreglaron las mutaciones en el algoritmo genético
Duración: 3 horas
Descripción de la actividad: Se descubrió que la forma de hacer las mutaciones era errónea y solo hacía ciertos cambios a la hora de mutar

5- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 31/05/2024
Actividad: Se arregló el problema de la mochila, ya que el algoritmo favoreció el valor por encima del peso y se hicieron las consultas del algoritmo dinámico
Duración: 5 horas
Descripción de la actividad: Se realizaron todas las consultas del algoritmo dinámico, solo quedó 1 la cual no se está seguro si está funcionando.

6- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 02/06/2024
Actividad: Se realizaron todas las consultas del algoritmo genético y se arregló la última consulta del algoritmo dinámico
Duración: 5 horas

8- Nombre del Estudiante: **Roosevelt Alejandro Pérez González**
Fecha: 0/06/2024
Actividad: Se realizaron todas las consultas del algoritmo genético y se arregló la última consulta del algoritmo dinámico
Duración: 6 horas

Minuta de Reunión

Lugar: Salón de estudio 24/7	**Fecha y hora: 10/05/2024, 12:40 pm**	
Objetivo de la reunión: Inicio del proyecto de Análisis de Algoritmos		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Búsqueda de literatura relacionada con el proyecto

Acuerdos

Los estudiantes deberán buscar literatura relacionada con el proyecto antes de la próxima sesión.

Tema 2

Se establecen los días de reunión para realizar el proyecto.

Acuerdos

Los dos estudiantes se comprometen a reunirse los días martes y miércoles de 5 pm - 8 pm.

Tema 3

Se establecen los días de reunión para realizar el proyecto.

Acuerdos

Los dos estudiantes se comprometen a reunirse los días martes y miércoles de 5 pm - 8 pm.

Minuta de Reunión

Lugar: Laboratorio 1	**Fecha y hora: 17/05/2024, 12:40 pm**	
Objetivo de la reunión: Empezar los algoritmos		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Se empezó a trabajar en los algoritmos solicitados en el proyecto

Acuerdos

Buscar los mejores algoritmos y compararlos entre sí.

Minuta de Reunión

Lugar: Salón de estudio 24/7	**Fecha y hora: 26/05/2024, 1:00 pm**	
Objetivo de la reunión: Comenzar la documentación y hacer mejoras		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Iniciar la documentación

Acuerdos

Los estudiantes se reparten partes del documento para dejar una carga similar para todos.

Tema 2

Hacer mejoras al código.

Acuerdos

Los estudiantes buscan cómo arreglar el código genético, debido a que los resultados no son los esperados.

Minuta de Reunión

Lugar: Cubículo	**Fecha y hora: 30/05/2024, 5:00 pm**	
Objetivo de la reunión: Mejorar algoritmos		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Mejorar el algoritmo genético

Acuerdos

Debido a que el algoritmo genético sigue dando problemas, se trabaja en equipo para mejorar el algoritmo en el apartado de mutaciones.

Minuta de Reunión

Lugar: Cubículo	**Fecha y hora: 31/05/2024, 3:00 pm**	
Objetivo de la reunión: Arreglar algoritmo genético		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Arreglar algoritmo genético

Acuerdos

Se arregla el algoritmo genético debido a que éste favorecía el valor por encima del peso. Además, se empezaron las consultas acerca de este algoritmo.

Minuta de Reunión

Lugar: Cubículo	**Fecha y hora: 2/06/2024, 12:00 pm**	
Objetivo de la reunión: Finalización de la parte programada		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Terminar consultas del genético y del dinámico

Acuerdos

Los estudiantes terminan la mayoría de las consultas de ambos algoritmos, logrando así la finalización de la parte programada.

Minuta de Reunión

Lugar: Discord	**Fecha y hora: 04/05/2024, 4:00 pm**	
Objetivo de la reunión: Terminar Proyecto		
Participantes	**Cargo**	**Firma**
Roosevelt Alejandro Pérez González		Presente
José Andrés Lorenzo Segura		Presente
Arnold Jafeth Álvarez Rojas		N/A
N/A		N/A

Ausentes	**Cargo**	**Justificación**
N/A		N/A

Temas por tratar

Tema 1

Terminar la documentación del proyecto

Acuerdos

Los estudiantes se reúnen al ser las 4 de la tarde para terminar la documentación necesaria del proyecto.