



UNIVERSIDAD TECNOLÓGICA CENTROAMERICANA

FACULTAD DE INGENIERÍA

ANÁLISIS DE ALGORITMOS

SECCION: 872

PROYECTO

CATEDRÁTICO:

ING. MIGUEL ARDÓN

PRESENTADO POR:

ANA MELISSA CABRERA - 22141111

LIZZIE LETICIA JUAREZ ESPINAL - 22051070

JOSUE FRANCISCO DE JESUS MORENO - 22011187

FECHA: 12/12/2024

LUGAR: CAMPUS SAN PEDRO SULA

Índice

Índice.....	2
Introducción.....	3
Problemas Seleccionados.....	4
Problema de la Mochila (Knapsack).....	4
Problema del Viajante de Comercio (TSP).....	5
Coloración de grafos.....	7
Problema de la Mochila (Knapsack).....	8
Método de Fuerza Bruta ($O(2^n)$).....	8
Problema del Viajante de Comercio (TSP).....	9
Peor de los Casos.....	9
Notación Theta (Θ).....	10
Mejor de los Casos.....	10
Coloración de grafos.....	11
Mejor Caso (Ω -Omega).....	11
3. Peor Caso (O).....	11
4. Caso Promedio (Θ -Theta).....	11
Análisis de los Componentes de Complejidad.....	12
Desarrollo de Algoritmos.....	12
Problema de la Mochila (Knapsack).....	12
Problema del Viajante de Comercio (TSP).....	14
Clusterización de Ciudades (K-means Clustering).....	14
Coloración de grafos.....	15
Descripción del Tiempo de Ejecución.....	17
Experimentación y Resultados.....	18
Problema de la Mochila (Knapsack).....	18
Problema del Viajante de Comercio (TSP).....	20
Coloración de grafos.....	20
Comparación de Ejecuciones (Intentos 3, 9 y 11).....	21
Conclusiones.....	23
Referencias.....	24
Apéndices.....	25

Introducción

En el ámbito del análisis de algoritmos, los problemas no polinomiales representan un desafío central debido a su complejidad computacional y su impacto en diversas aplicaciones prácticas. Estos problemas, conocidos por pertenecer a las clases NP-completo y NP-difícil, presentan una estructura compleja que impide encontrar soluciones óptimas en tiempo polinómico para todas sus instancias conocidas.

Este informe se centra en tres problemas de la teoría de la complejidad computacional: el Problema de la Mochila (Knapsack), el Problema del Viajante de Comercio (TSP) y el Problema de la Coloración de Grafos. Cada uno de estos problemas es un ejemplo representativo de situaciones en las que el crecimiento exponencial de combinaciones posibles genera desafíos grandes para los algoritmos.

El Problema de la Mochila implica seleccionar un subconjunto de objetos con valores y pesos asignados para el beneficio total sin exceder una capacidad limitada. El Problema del Viajante de Comercio busca determinar la ruta más corta para visitar una serie de ciudades exactamente una vez y regresar al punto de partida. Este problema es crucial en la optimización de rutas de transporte y logística, donde la eficiencia puede marcar una diferencia.

El Problema de la Coloración de Grafos donde se asignan colores a los vértices de un grafo de modo que dos vértices adyacentes no comparten el mismo color. Sus aplicaciones incluyen la asignación de recursos en redes de telecomunicaciones y la programación de horarios.

En este proyecto, exploraremos las características esenciales, analizaremos las complejidades computacionales y discutiremos los enfoques algorítmicos utilizados para abordar estos problemas. Se presentarán también resultados obtenidos a través de implementaciones prácticas, destacando sus limitaciones y posibles optimizaciones. El objetivo es proporcionar una comprensión profunda de estos problemas dentro del contexto del análisis de algoritmos y su relevancia en la computación moderna.

Problemas Seleccionados

Problema de la Mochila (Knapsack)

El problema de Knapsack es un problema clásico dentro de la optimización combinatoria en la informática y en las matemáticas. Consiste en un conjunto de objetos, cada uno con su peso y su valor, el objetivo es seleccionar el subconjunto de estos objetos más óptimo de tal manera que se maximice el valor total de los objetos sin excederse de la capacidad máxima de la mochila. Cada objeto debe ser seleccionado solo una vez, pueda estar o no en el subconjunto final.

Importancia y Aplicaciones Prácticas

Por su importancia, el problema de Knapsack sigue siendo estudiado a profundidad, ya que para su solución se incluyen enfoques más óptimos como la programación dinámica, algoritmos recursivos y otros enfoques utilizando algoritmos greedy. El problema es fundamental en la teoría de la optimización combinatoria con aplicaciones extensas dentro de varias ramas de las ciencias así como la economía. Gracias a este modelo se pueden hacer una variedad de representaciones basadas en situaciones de la vida real donde se deban tomar decisiones bajo ciertas restricciones. Debido a que es un problema NP-completo, esto significa que no existe algoritmo eficiente que lo resuelva en tiempo polinómico para todos los casos. Algunas aplicaciones de este problema se pueden encontrar dentro de la planificación de proyectos, optimización de redes, o la inteligencia artificial.

Complejidad Computacional y su Relevancia

Como mencionamos antes, el problema de Knapsack, gracias a su naturaleza de combinatoria y su capacidad para representar situaciones de la vida real donde se involucra la optimización, esto lo hace un problema bastante estudiado dentro de la teoría de la computación. Es un problema cuya complejidad computacional lo convierte en un punto para estudiar conceptos de optimización para el diseño de algoritmos eficientes.

Problema del Viajante de Comercio (TSP)

El Problema del Viajante, o travelling salesman problem (TSP, por sus siglas en inglés), es un desafío que trata de encontrar la ruta más corta para visitar varias ciudades y volver al inicio, pasando por cada ciudad solo una vez.

El Problema del Viajante, o travelling salesman problem (TSP, por sus siglas en inglés), es un desafío que trata de encontrar la ruta más corta para visitar varias ciudades y volver al inicio, pasando por cada ciudad solo una vez.

Este problema, que se descubrió por primera vez en 1930, se considera un problema NP-Hard en la optimización combinatoria. A pesar de ser un problema computacionalmente complicado, existen técnicas heurísticas y precisas que pueden resolver planteamientos específicos del problema, abarcando desde una ciudad hasta miles.

El TSP es importante no solo en teoría; en la práctica, juega un papel crucial en cómo se planifican las rutas de entrega. Por ejemplo, los camiones de reparto usan este principio para elegir el mejor camino que minimice el tiempo en carretera y el uso de combustible, lo que ayuda a reducir los gastos y a que los clientes estén más contentos.

Complejidad

Si pensamos en este problema considerando un único destino, parece muy sencillo de calcular y en teoría lo es. Sin embargo, cuando aumentamos el **número de destinos** automáticamente se vuelve un cálculo muchísimo más complejo.

Por ejemplo, si tenemos un **agente viajero** con 10 destinos, nos entregaría alrededor de 362.880 rutas posibles, las cuales serían demasiadas para que un programa las maneje en un tiempo razonable entregándonos las más rápidas y eficientes.

Es por esta razón que el **Travelling Salesman Problem** se considera como un problema **NP-difícil** (clasificación según la teoría de complejidad computacional) debido a que no existe un **algoritmo de tiempo** que lo resuelva de manera eficiente.

Para visualizar la complejidad y la cantidad de rutas posibles, consideremos lo siguiente:

- Cantidad de Rutas:** La cantidad de posibles rutas que el viajante puede tomar es $(n-1)!(n-1)!$, donde n es el número de ciudades. Esto se debe a que, después de elegir una ciudad inicial, hay $(n-1)(n-1)$ opciones para la siguiente ciudad, $(n-2)(n-2)$ opciones para la ciudad siguiente, y así sucesivamente hasta que todas las ciudades hayan sido visitadas.

Por ejemplo:

- Para $n=3$ ciudades: $(3-1)!=2!=2(3-1)!=2!=2$ rutas.
- Para $n=4$ ciudades: $(4-1)!=3!=6(4-1)!=3!=6$ rutas.
- Para $n=5$ ciudades: $(5-1)!=4!=24(5-1)!=4!=24$ rutas.

Numero de Ciudades (n)	Cantidad de Rutas $((n-1)!)$
2	1
3	2
4	6
5	24
6	120
7	720
8	5,040
9	40,320
10	362,880
11	3,628,800
12	39,916,800
13	479,001,600
14	6,227,020,800
15	87,178,291,200

Coloración de grafos

El problema de la coloración de grafos consiste en asignar colores a los vértices de un grafo de manera que no existan dos vértices adyacentes con el mismo color.

Formalmente, un grafo $G=(V,E)$, donde V es el conjunto de vértices y E es el conjunto de aristas, requiere una función de coloración $f:V \rightarrow C$, donde C es el conjunto de colores, de modo que si dos vértices u y v están conectados por una arista $(u,v) \in E$, entonces $f(u) \neq f(v)$. El objetivo es minimizar el tamaño de C para optimizar el uso de recursos (Rugerio Bretón & Sánchez Ortega, 2018)

Existen distintas aplicaciones para un algoritmo de coloración de grafos, según nuestra investigación las aplicaciones tienen gran relevancia a los de gestión de recursos, la optimización de redes y la asignación de tareas críticas. Por ejemplo un sistema donde se aplica en la programación de clases, exámenes y turnos laborales se usaría para evitar conflictos de horario entre elementos incompatibles En telecomunicaciones, es crucial asignar frecuencias distintas a estaciones de radio vecinas para minimizar las interferencias Otro aplicación seria en el diseño de compiladores, es necesario asignar variables a registros de memoria de manera eficiente para minimizar el tiempo de acceso durante la ejecución de programas.

La coloración de grafos es un problema de optimización combinatoria que pertenece a la clase de complejidad NP-completo. Esto significa que no existe un algoritmo conocido capaz de resolver todas sus instancias en tiempo polinomial. La complejidad del problema radica en el crecimiento exponencial del número de combinaciones posibles a medida que aumenta el número de vértices del grafo.

Análisis de Tiempo

Problema de la Mochila (Knapsack)

En este proyecto la variante de Knapsack que se abordó en este proyecto fue la de 0-1, que consiste en que cada elemento puede ser incluido o excluido dentro de la solución final. Siendo este un problema NP-completo, significando que no existe un algoritmo eficiente para encontrar la solución óptima en el peor de los casos.

La complejidad para encontrar una solución óptima varía según el enfoque que tomemos:

Método de Fuerza Bruta ($O(2^n)$)

Si utilizamos un método de fuerza bruta se requiere explorar todas las combinaciones posibles para el conjunto de elementos para determinar cuál será la mejor solución. Este procedimiento aumenta la complejidad ya que depende del tamaño del arreglo lo que lleva a una complejidad exponencial en el peor de los casos. En este caso se implementó un enfoque greedy basado en recursión y backtracking para identificar la mejor solución al problema. El análisis de la recursividad se puede representar como $T(n) = 2T(n - 1) + C$ donde:

$n - 1$ se refiere a la reducción del tamaño de entrada en cada paso recursivo.

$2T$ es la cantidad de veces que se ejecuta el método recursivo ya que se debe evaluar cuando el item actual se incluye o se excluye.

C representa las operaciones constantes que se ejecutan dentro de la llamada recursiva.

```
function knapsack(items, capacity) {
    let lastValue = 0;
    let lastWeight = 0;
    let bag = [];

    function search(index, currentWeight, currentValue, currentBag) {
        if (index >= items.length) {
            if (currentWeight <= capacity && currentValue > lastValue) {
                lastValue = currentValue;
                lastWeight = currentWeight;
                bag = [...currentBag];
            }
        } else {
            search(index + 1, currentWeight, currentValue, currentBag);
            if (currentWeight + items[index].weight <= capacity) {
                currentBag.push(items[index]);
                search(index + 1, currentWeight + items[index].weight,
                    currentValue + items[index].value, currentBag);
                currentBag.pop();
            }
        }
        return;
    }

    search(0, 0, 0, []);
    return { bag, lastValue, lastWeight };
}
```

$$\begin{aligned} T(n) &= 2T(n-1) + C \\ &= 2(2T(n-2) + C) + C \\ &= 2^2T(n-2) + 2C + C \\ &= 2^2(2T(n-3) + C) + 2C + C \\ &= 2^3T(n-3) + 2^2C + 2C + C \\ &= 2^kT(n-k) + \sum_{i=0}^{k-1} 2^i C \\ &= 2^nT(0) + C \sum_{i=0}^{n-1} 2^i \\ &= 2^nT(0) + C(2^{n+1}) \\ &= 2^nT(0) + C(2^n - 1) \\ O(T(n)) &= O(2^n) \end{aligned}$$

Tras realizar el análisis podemos concluir que la implementación recursiva mediante backtracking indica un tiempo exponencial de $O(2^n)$ en el peor de los casos.

Programación Dinámica ($O(n*C)$)

Al implementar algoritmos más eficientes como la programación dinámica, se puede llegar a una solución con una complejidad de $O(n*C)$ donde:

n es la cantidad de ítems en el conjunto.

C es la capacidad de la mochila.

Siendo esta una técnica más eficiente, se puede resolver el problema de Knapsack reduciendo considerablemente el tiempo de ejecución al evitar cálculos redundantes. La solución se basa en la construcción de una tabla que almacena subresultados para determinar si la inclusión del elemento actual maximiza el valor total sin exceder la capacidad máxima que tiene la mochila.

Problema del Viajante de Comercio (TSP)

La complejidad de este algoritmo se basa en la dispersión, distancia, cantidad y orden de recorrido entre los puntos o ciudades que se estén analizando. Dependiendo de estos y más puntos, el tiempo de ejecución puede variar de caso a caso. A continuación, se presentan aproximaciones de los tiempos de ejecución en el peor y mejor de los casos para 10 ciudades, teniendo en cuenta que el programa selecciona de manera aleatoria la posición de cada punto, por lo que no se puede hacer un ejemplo perfecto de cada caso.

Peor de los Casos

En el peor de los casos, los puntos están dispersos y distantes unos de otros, lo que significa que el grafo que representa las conexiones entre las ciudades es denso y las distancias entre ellos son significativamente grandes. En esta situación, resolver el TSP se convierte en un desafío significativo porque encontrar la ruta más corta para visitar cada ciudad una vez y regresar al punto de partida requiere evaluar todas las combinaciones de rutas posibles.

Los métodos precisos, como la enumeración exhaustiva o la programación dinámica, tienen un tiempo de ejecución prohibitivo en este escenario y aumentan exponencialmente con el número de ciudades. Esto hace que encontrar la solución óptima sea prácticamente y computacionalmente costosa para instancias grandes del problema.

Ejemplo del Peor Caso:

En un ejemplo cercano al peor de los casos, los puntos están relativamente dispersos, haciendo que el programa tenga que recorrer las ciudades de varias maneras con distancias extensas. En este caso, el tiempo de ejecución fue de 3144 ms, lo cual refleja el costo de cálculo para hallar la ruta. Para este caso, el tiempo de ejecución en el peor de los casos sería $O(n!)$ debido a la necesidad de evaluar todas las posibles rutas.

Notación Theta (Θ)

La notación Θ proporciona una descripción más precisa del tiempo de ejecución del algoritmo, indicando tanto un límite superior como inferior que son asintóticamente iguales. Para el TSP utilizando métodos exactos (como fuerza bruta o programación dinámica), la complejidad es:

$$\Theta(n!) \Theta(n!)$$

Esto indica que el tiempo de ejecución crece factorialmente tanto en el mejor como en el peor de los casos.

Mejor de los Casos

La notación Ω describe un límite inferior en el tiempo de ejecución del algoritmo. En el mejor de los casos, aunque los puntos estén ordenados de manera óptima, el tiempo de ejecución sigue estando determinado por la estructura del algoritmo. Sin embargo, es importante notar que en un escenario realista con puntos ordenados, algunas heurísticas o algoritmos podrían encontrar soluciones más rápidamente, pero aún así, el peor de los casos sería una evaluación exhaustiva de rutas.

$$\Omega(n!)$$

Para el TSP exacto, incluso en el mejor de los casos donde los puntos están ópticamente ordenados, el algoritmo necesitaría verificar al menos una fracción significativa de las permutaciones para confirmar la solución, llevando el tiempo de ejecución mínimo a crecer factorialmente.

Coloración de grafos

El tiempo de ejecución para el algoritmo se puede expresar en términos de $T=C \cdot n^k$, donde:

n es el número de nodos del grafo.

k es el número de colores.

C es una constante dependiente de las operaciones internas.

El tiempo máximo de ejecución corresponde al número de combinaciones de colores evaluadas:

$T=C \cdot k^n$ donde T crece exponencialmente con n y k.

Mejor Caso (Ω -Omega)

El mejor caso ocurre si el algoritmo encuentra una solución válida en el primer intento (por ejemplo, al asignar colores en el orden más favorable).

El grafo tiene una estructura sencilla (por ejemplo, muy pocas aristas) y se puede colorear con k colores sin necesidad de retroceso.

Solo funciona si una rama del árbol de búsqueda es explorada:

$$\Omega(T)=O(k^n)$$

3. Peor Caso (O)

El peor caso ocurre cuando el algoritmo explora todas las combinaciones posibles de colores antes de concluir que no hay solución válida o encontrar la última solución posible.

El grafo está altamente conectado (denso) y el número de colores es insuficiente para evitar conflictos. Este es el comportamiento en problemas NP-completos, donde cada nodo tiene k opciones y el árbol tiene una profundidad n. El algoritmo recorre todo el árbol de combinaciones:

$$O(T) = O(k^n)$$

4. Caso Promedio (Θ -Theta)

El caso promedio depende de la distribución de las soluciones en el espacio de búsqueda.

En un grafo con una estructura moderada y un número razonable de colores, el algoritmo encuentra una solución tras explorar parcialmente el árbol.

El árbol de búsqueda no se recorre completamente:

$$\Theta(T) = O(k^{n/2})$$

Esto asume que, en promedio, se explora aproximadamente la mitad del árbol.

Análisis de los Componentes de Complejidad

1. Evaluación de Conexiones DFS en isConnected()

$O(n+E)$, donde E es el número de aristas.

Este paso es lineal respecto al tamaño del grafo.

2. Asignación de Colores (colorGraph()):

- Complejidad: $O(k^n)$ en el peor caso debido al retroceso y las combinaciones.

3. Evaluación de Seguridad (isSafe()):

- Complejidad por nodo: $O(n)$ en el peor caso, ya que verifica todas las aristas conectadas al nodo.

Desarrollo de Algoritmos

Problema de la Mochila (Knapsack)

El algoritmo que se implementó para el problema de Knapsack se basa en técnicas combinatorias. Este enfoque explora todas las combinaciones posibles para un conjunto dado de objetos. Durante el proceso el algoritmo evalúa cada combinación y almacena el valor máximo encontrado en una variable, con el fin de encontrar la solución más óptima al problema.

```

async function knapsack(items, capacity) {
  const start = performance.now();

  let lastValue = 0;
  let lastWeight = 0;
  let bag = [];

```

Los valores lastValue y lastWeight y el arreglo contenido el subconjunto de elementos son los que se van actualizando durante el recorrido de combinaciones.

```

function search(index, currentWeight, currentValue, currentBag) {
  if (index >= items.length) {
    if (currentWeight <= capacity && currentValue > lastValue) {
      lastValue = currentValue;
      lastWeight = currentWeight;
      bag = [...currentBag];
    }
    return;
  }

  search(index + 1, currentWeight, currentValue, currentBag);

  if (currentWeight + items[index].weight <= capacity) {
    currentBag.push(items[index]);

    search(index + 1, currentWeight + items[index].weight, currentValue + items[index].value, currentBag);
    currentBag.pop();
  }
}

```

La función search es una función auxiliar que es la que se encarga de explorar todas las combinaciones posibles de inclusión y exclusión mediante recursión.

```

if (index >= items.length) {
  if (currentWeight <= capacity && currentValue > lastValue) {
    lastValue = currentValue;
    lastWeight = currentWeight;
    bag = [...currentBag];
  }
  return;
}

```

El caso base se encarga de actualizar los valores finales una vez se hayan recorrido todas las combinaciones en caso sea una combinación que cumpla con los requisitos de no pasarse de la capacidad máxima y que sea un valor más grande que el último valor guardado.

```

search(index + 1, currentWeight, currentValue, currentBag);

```

El caso principal se encarga de procesar las combinaciones excluyendo el ítem actual, es decir que el algoritmo omite incluirlo y continúa evaluando las combinaciones con el elemento que le sigue.

```
if (currentWeight + items[index].weight <= capacity) {  
    currentBag.push(items[index]);  
  
    search(index + 1, currentWeight + items[index].weight, currentValue + items[index].value, currentBag);  
    currentBag.pop();  
}
```

El caso secundario es el que se encarga de evaluar las combinaciones incluyendo el elemento actual. Se hace push al elemento actúa para poder evaluar las combinaciones al incluirlo. Llamamos a Search de manera recursiva para así explorar las opciones que tenemos. De ahí se hace pop al elemento para volver al estado anterior de la combinación.

```
search(0, 0, 0, []);
```

Aquí iniciamos la función recursiva con los valores iniciales.

Como se mencionó anteriormente, al utilizar un enfoque basado en la exploración de todas las combinaciones posibles del conjunto de elementos, la complejidad del algoritmo depende directamente del tamaño del conjunto de elementos. Este enfoque tiene una complejidad exponencial en el peor de los casos, ya que el número de combinaciones posibles es 2^n , donde n es el número de elementos en el conjunto. Esto significa que el tiempo de ejecución puede crecer rápidamente a medida que aumente el tamaño del conjunto, lo que puede afectar el rendimiento para grandes cantidades de elementos.

Problema del Viajante de Comercio (TSP)

El algoritmo propuesto para resolver el problema del viajante de comercio (TSP) con un enfoque híbrido de clustering se compone de varias fases que combinan técnicas de clustering, heurísticas de optimización y métodos de refinamiento global. Aquí está una explicación detallada de cómo funciona cada fase y la propuesta de solución en JavaScript.

Clusterización de Ciudades (K-means Clustering)

En esta fase, las ciudades se agrupan en clústeres utilizando el algoritmo K-means. La idea es dividir el problema grande en varios subproblemas más pequeños y manejables.

Para cada clúster, se utiliza la heurística del vecino más cercano para generar una ruta inicial, y luego se optimiza esa ruta utilizando el algoritmo 2-opt.

Conexión de Clústeres

Una vez que tenemos rutas optimizadas para cada clúster, conectamos estos clústeres en una ruta global.

Finalmente, aplicamos el algoritmo de recocido simulado (simulated annealing) para refinar aún más la ruta global.

Coloración de grafos

El sistema que desarrollamos en JavaScript implementa el problema de la coloración de grafos utilizando técnicas combinatorias y algoritmos de backtracking. El código maneja la representación del grafo mediante una lista de adyacencia, realiza verificaciones de conectividad mediante el recorrido en profundidad (DFS) y utiliza asignación recursiva para encontrar combinaciones de colores viables.

El núcleo del sistema está compuesto por:

- **Clase *ColoringGraphs*:** Gestiona la creación del grafo, la verificación de conectividad y la ejecución del algoritmo de coloración.
- **Métodos Clave:**
 - *addEdge(v, w)*: Agrega aristas al grafo.
 - *isConnected()*: Verifica la conectividad usando DFS.
 - *graphColoring(numColors)*: Implementa la asignación de colores con retroceso.

En el caso de el algoritmo de coloracion de grafo se utilizaron 3 partes claves vistas en clase, la primera es el uso de listas enlazadas donde está en la representación del grafo mediante un *adjList*, que es un arreglo donde cada índice contiene una lista de vecinos de cada nodo. Esta estructura permite agregar nodos adyacentes de manera mas simple y realizar consultas rápidas durante el proceso de coloración.

```
8     this.adjList = Array.from({ length: V }, () => []);
9 }
10
```

La función recursiva *colorGraph(node)* realiza un recorrido en profundidad (DFS) para asignar colores a cada nodo y aplicar retroceso cuando se detectan conflictos. La recursión permite explorar todas las combinaciones posibles para encontrar una solución válida, aprovechando el enfoque de retroceso a lo que se le conoce como backtracking.

```
48 const colorGraph = (node) => {
49   if (node === this.V) {
50     return true;
51   }
52   const usedColors = new Set(result);
53   if (usedColors.size >= numColors) {
54     return false;
55   }
56   for (let c = 0; c < numColors; c++) {
57     if (isSafe(node, c)) {
58       result[node] = c;
59       colorUsage[c] += 1;
60       steps.push({
61         vertex: node,
62         color: c,
63         action: "assign",
64         message: `Node ${node} is assigned color ${c}`,
65         result: [...result],
66       });
67       if (colorGraph(node + 1)) return true;
68     } else {
69       steps.push({
70         vertex: node,
71         color: c,
72         action: "unassign",
73         message: `Backtracking: Unassigning color ${c} from node ${node}`,
74       });
75     }
76   }
77 }
```

El método *isConnected()* usa DFS para verificar si el grafo es conexo. El recorrido en profundidad permite verificar si todos los nodos son alcanzables desde el nodo inicial, asegurando la conectividad del grafo.

```

isConnected() {
    if (this.V === 0) return false;

    const visited = Array(this.V).fill(false);

    const dfs = (node) => {
        visited[node] = true;
        for (const neighbor of this.adjList[node]) {
            if (!visited[neighbor]) dfs(neighbor);
        }
    };

    dfs(0);

    return visited.every((node, idx) => node || this.adjList[idx].length === 0);
}

```

Descripción del Tiempo de Ejecución:

Escenario	Complejidad Temporal	Explicación
Mejor Caso (grafo pequeño y fácil de colorear)	$O(V + E)$	El DFS y la verificación de conectividad se ejecutan una vez.
Peor Caso (retroceso completo)	$O(V^{\text{numColors}})$	La recursión explora todas las combinaciones posibles.
Caso Aproximado (heurístico eficiente)	$O(V + E + V * \text{numColors})$	El algoritmo explora una cantidad significativa, pero no todas las combinaciones.

Experimentación y Resultados

Problema de la Mochila (Knapsack)

Cantidad de Items	Capacidad	Tiempo Algoritmo 1 (ms)	Tiempo Algoritmo DP (ms)
10	50	0	0
10	100	0	0.1
20	50	0	0.1
20	100	0	0.1
30	50	0.5	0.1
30	100	0.3	0.1
40	50	2.2	0
40	100	0.5	0
50	50	14.1	0
50	100	1.4	0
60	50	55.1	0.1
60	100	7.4	0

Se realizaron varias pruebas comparativas entre el algoritmo basado en fuerza bruta y el algoritmo optimizado mediante programación dinámica. Estas pruebas incluyeron diferentes cantidades de elementos y dos valores de capacidad distintos, con el objetivo de observar y analizar el comportamiento de cada algoritmo bajo diversas condiciones. Esta comparación permitió identificar las diferencias en rendimiento, eficiencia y escalabilidad entre ambos enfoques.

Como podemos observar al comparar ambos algoritmos implementados, el algoritmo basado en programación dinámica mantiene un rendimiento más constante en comparación al algoritmo que utiliza fuerza bruta. Esto se debe a que el algoritmo con programación dinámica optimiza los cálculos por medio de matrices para evitar operaciones redundantes,

mientras que el tiempo de ejecución del algoritmo que utiliza fuerza bruta crece de manera acelerada dependiendo del tamaño del conjunto.

Cantidad de Items	Capacidad	Tiempo Algoritmo 1 (ms)	Tiempo Algoritmo DP (ms)
100	20	0	0
100	40	7.2	0.2
100	60	751.9	0.1

Al probar con un tamaño constante relativamente alto, se puede observar como el rendimiento de nuestro algoritmo es afectado drásticamente a medida se va experimentando con distintos valores para la capacidad máxima de la mochila.

Cantidad de Items	Capacidad	Tiempo Algoritmo 1 (ms)	Tiempo Algoritmo DP (ms)
20	100	0	0
40	100	0.7	0.3
60	100	9.9	0.4
80	100	808.8	0.4
100	100	1642.4	0.1

Podemos ver un comportamiento similar al usar un valor constante para la capacidad de la mochila.

Durante las pruebas con valores muy grandes tanto en la cantidad de elementos como en la capacidad del problema, se observó que el algoritmo basado en fuerza bruta presenta tiempos

de ejecución muy elevados. Esto provoca que la aplicación deje de responder debido a la sobrecarga computacional generada.

Problema del Viajante de Comercio (TSP)

Coloración de grafos

Los “trials” del algoritmo se realizaron seleccionando diferentes valores para la cantidad de nodos del grafo y el número de colores disponibles. Las combinaciones fueron probadas sistemáticamente, abarcando grafos pequeños y medianos. Una vez que se identificó un cambio significativo en el tiempo de ejecución, se realizaron más pruebas con valores similares.

Para cada combinación de nodos y colores, se ejecutó el algoritmo. Se midió el tiempo de ejecución utilizando el objeto *performance.now()* para registrar los tiempos con precisión.

Los resultados obtenidos se almacenaron, registrando el tiempo de ejecución para cada combinación y observando si el algoritmo fue capaz de generar una solución válida.

Se evaluaron casos exitosos, tiempos extremos (muy largos) y fallas como "overflow" o bloqueos para entender los límites del algoritmo.

Cantidad de Nodos	Cantidad de Colores	Tiempo de Ejecución
4	2	0.03
5	3	0.04
5	5	0.06
10	4	0.03
10	6	0.28
15	5	0.03

15	6	0.06
15	8	2.95
15	10	1575
15	10	1226
15	11	Overflow – crash
15	9	76.6

Comparación de Ejecuciones (Intentos 3, 9 y 11)

Los tiempos de ejecución para los casos 3, 9 y 11 son:

Intento	Cantidad de Nodos (n)	Cantidad de Colores (k)	Tiempo de Ejecución (s)
3	5	5	0.06
9	15	8	2.95
11	15	11	Overflow - Crash

La fórmula del tiempo máximo T se calcula como:

$T = C \cdot n^k$ Donde C es una constante proporcional al tiempo por operación básica del algoritmo.

1. Intento 3 (n=5, k=5):

$$T_3 = C \cdot 5^5 = C \cdot 3125$$

El tiempo real fue 0.06 segundos, lo que indica un C muy pequeño debido al tamaño reducido del grafo.

2. Intento 9 (n=15, k=8):

$$T_9 = C \cdot 15^8 = C \cdot 2562890625$$

Aunque el cálculo teórico predice un tiempo mucho mayor, la implementación puede haberse detenido antes debido a optimizaciones internas o límites de búsqueda, resultando en 2.95 segundos.

3. Intento 11 (n=15, k=11):

$$T_{11} = C \cdot 15^{11} = C \cdot 865041591938125$$

Este caso llevó al "overflow", lo que confirma que el algoritmo no puede manejar combinaciones tan extensas en memoria o tiempo.

La comparación de tiempos muestra cómo el crecimiento exponencial afecta el desempeño del algoritmo:

1. **Intento 3:** El tiempo es insignificante debido al tamaño pequeño del problema (5^5).
2. **Intento 9:** Aumentar los nodos y colores (15^8) incrementó el tiempo de ejecución drásticamente, alcanzando varios segundos.
3. **Intento 11:** El aumento adicional de colores (15^{11}) llevó a una explosión combinatoria que superó las capacidades del sistema.

Casos Rápidos:

- ◆ Grafos con pocos nodos y colores (por ejemplo, 4 nodos con 2 colores) se resolvieron en menos de 0.03 segundos.
- ◆ El tiempo de ejecución se mantuvo constante para grafos de hasta 10 nodos con un número moderado de colores.

Crecimiento Exponencial:

- ◆ A medida que aumentaron los nodos y los colores (por ejemplo, 15 nodos con 8 colores), el tiempo de ejecución creció significativamente, alcanzando 2.95 segundos.

Fallas:

- ◆ Para combinaciones complejas como 15 nodos y 11 colores, el algoritmo no pudo completarse debido a un "overflow", dejando un límite en la capacidad de la implementación actual.

El algoritmo presenta varias limitaciones significativas. Una de las principales es el crecimiento exponencial de su complejidad ($O^{|V| \cdot |C|}$), lo que provoca un aumento increíblemente grande en el tiempo de ejecución a medida que se incrementa el tamaño del grafo y la cantidad de colores disponibles, haciéndolo inviable para grafos grandes o con múltiples combinaciones de colores. Además, en términos de gestión de recursos, la implementación actual no maneja bien escenarios extremos, como grafos con muchos nodos y colores, lo que puede provocar fallas o bloqueos debido al consumo excesivo de memoria o tiempo. Por último, la falta de heurísticas limita la capacidad del algoritmo para encontrar soluciones aproximadas en tiempos razonables, lo que reduce su aplicabilidad en contextos donde el tiempo de respuesta es crítico.

Conclusiones

El proyecto ha demostrado la efectividad de los algoritmos exactos basados en retroceso (backtracking) para resolver problemas de coloración de grafos en casos pequeños y moderadamente complejos. Utilizando listas enlazadas para representar grafos, recorridos en profundidad (DFS) para verificar conectividad, y asignación recursiva para la coloración, se lograron soluciones exactas en tiempos razonables para grafos con menos de 15 nodos y

colores limitados. Sin embargo, el crecimiento exponencial del tiempo de ejecución en problemas más grandes evidencia la necesidad de enfoques más avanzados para aplicaciones prácticas a gran escala.

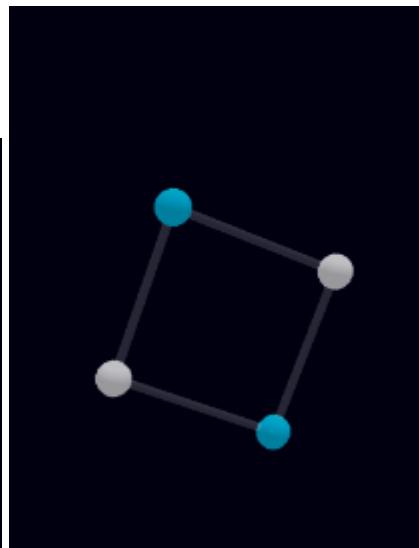
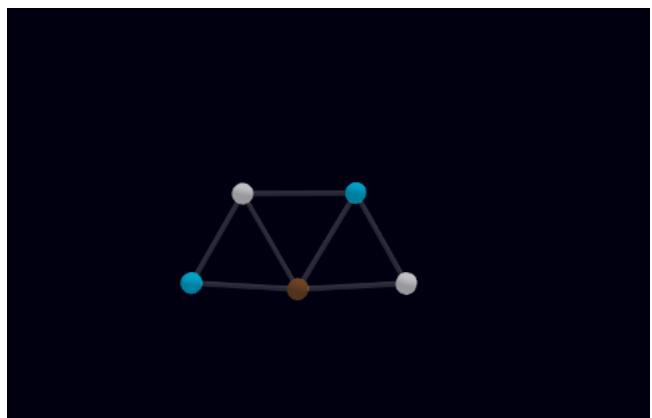
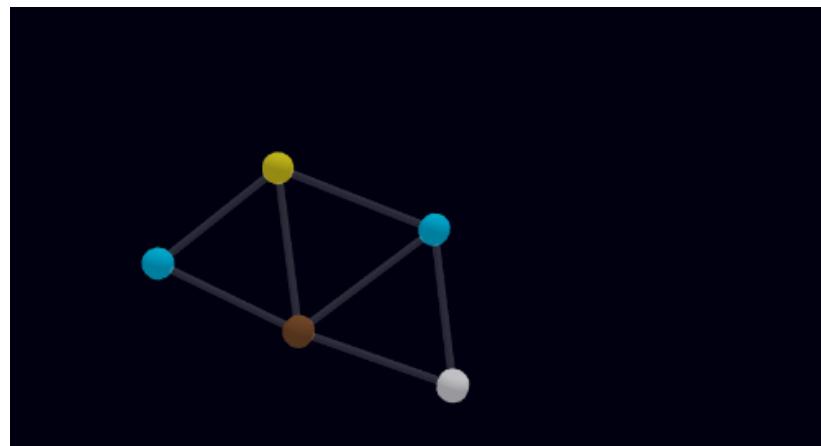
En este estudio se implementó un enfoque basado en fuerza bruta para resolver el problema de Knapsack. Aunque este enfoque resulta más sencillo de implementar en comparación con el que utiliza programación dinámica, sus limitaciones se hicieron evidentes debido a su complejidad exponencial. Esto ocasiona tiempos de ejecución muy elevados y, en algunos casos, puede hacer que la aplicación deje de responder cuando se prueban conjuntos con grandes cantidades de elementos o capacidades extensas. Estos resultados destacan la importancia de diseñar algoritmos más eficientes al abordar problemas con grandes volúmenes de datos, especialmente en aplicaciones prácticas donde el rendimiento es crucial.

El TSP es un problema fundamental y desafiante en el campo de la optimización combinatoria. Si bien encontrar la solución exacta es computacionalmente impracticable para grandes instancias, una combinación de heurísticas, metaheurísticas y enfoques híbridos puede producir soluciones cercanas a óptimas en tiempos razonables. La elección del método depende en gran medida de las características específicas del problema en cuestión y de los recursos computacionales disponibles. La continua investigación y desarrollo de nuevas técnicas aseguran que el TSP siga siendo un área activa y relevante en la ciencia y la ingeniería.

Referencias

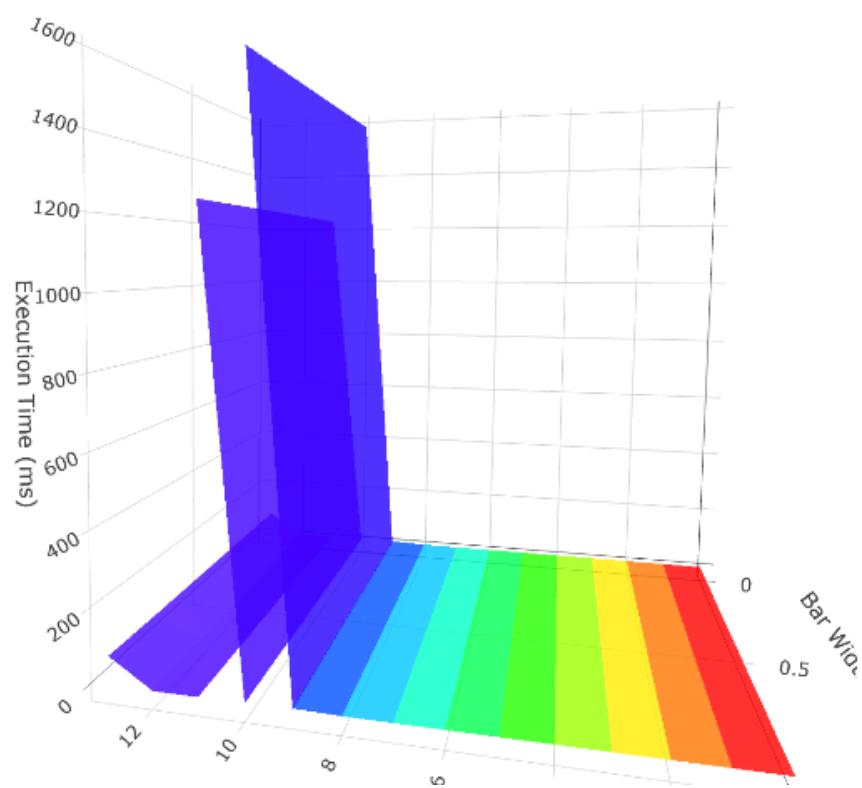
- Aguilar, J. (2011). *Problemas NP-completos y complejidad computacional*. Francia-Venezuela: Cooperación Bilateral.
- Méndez Díaz, I. (2003). *Problema de colooreo de grafos: Un estudio poliedral y un algoritmo Branch-and-Cut* (Tesis doctoral). Universidad de Buenos Aires.
- Rugerio Bretón, J. R., & Sánchez Ortega, M. A. (2018). *Comparación de métodos heurísticos para el problema de coloreado de grafos* (Tesis de Honores). Universidad de las Américas Puebla
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

Apéndices



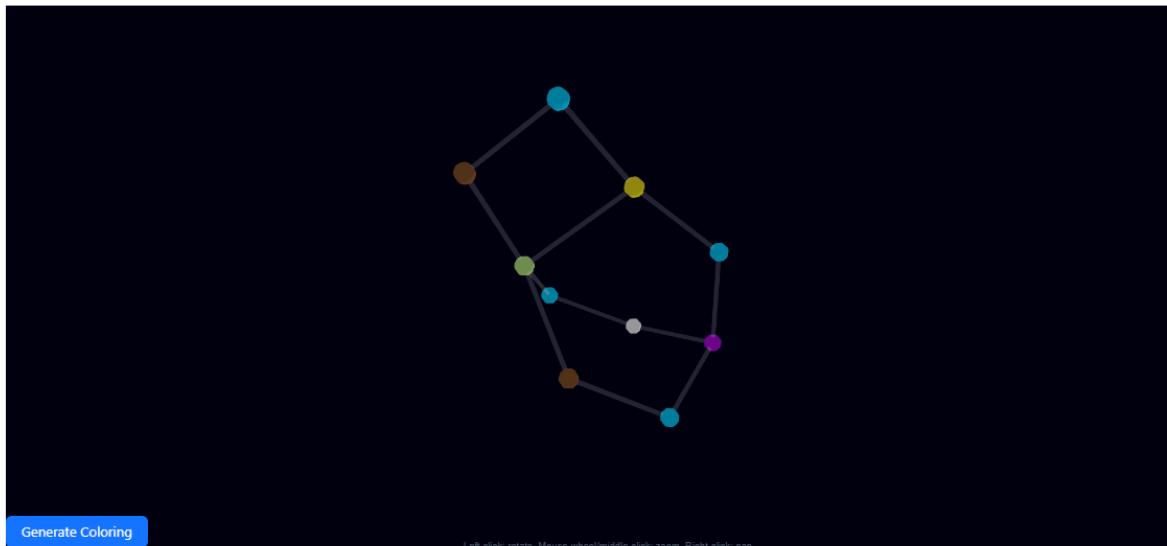
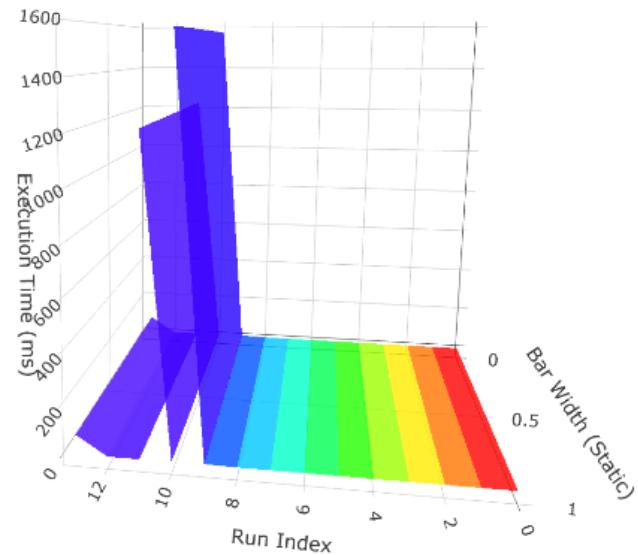
Execution Time Comparison

3D Execution Time Comparison



Execution Time Comparison

3D Execution Time Comparison



Step-by-Step Visualization

Step 272: Vertex 9 was assigned color #bd10e0.

[Previous](#) [Next](#)

Knapsack

Generar Items

Generar

Capacidad

Ejecutar Algoritmos

Items Generados	
Peso	Valor
38	47
16	41
15	84
36	55
50	77
18	80
23	41
8	38

Resultados Knapsack

Peso: 15, Valor: 84
Peso: 8, Valor: 38
Peso: 17, Valor: 80
Peso: 12, Valor: 68
Peso: 7, Valor: 82
Peso: 15, Valor: 100
Peso: 7, Valor: 49
Peso: 6, Valor: 84
Peso: 5, Valor: 56
Peso: 6, Valor: 75
Peso: 1, Valor: 37

Resultados Knapsack DP

Peso: 1, Valor: 37
Peso: 6, Valor: 75
Peso: 5, Valor: 56
Peso: 6, Valor: 84
Peso: 7, Valor: 49
Peso: 15, Valor: 100
Peso: 7, Valor: 62
Peso: 12, Valor: 68
Peso: 8, Valor: 38
Peso: 18, Valor: 80
Peso: 15, Valor: 84

Historial de Resultados

Resetear			
Cantidad de Items	Capacidad	Tiempo Algoritmo 1 (ms)	Tiempo Algoritmo DP (ms)
20	100	0	0
40	100	0.7	0.3
60	100	9.9	0.4
80	100	808.8	0.4
100	100	1642.4	0.1

Generar Items

Generar

Ejecutar Algoritmos

Generados

Peso	Valor
20	
88	
45	
73	
69	
64	
27	

Resultados Knapsack

No data

Peso Total: 0
Valor Total: 0

Resultados Knapsack DP

No data

Peso Total: 0
Valor Total: 0

Historial de Resultados

Resetear			
Cantidad de Items	Capacidad	Tiempo Algoritmo 1 (ms)	Tiempo Algoritmo DP (ms)
20	100	0	0
40	100	0.7	0.3
60	100	9.9	0.4
80	100	808.8	0.4
100	100	1642.4	0.1

This page isn't responding

React App

Wait Exit page