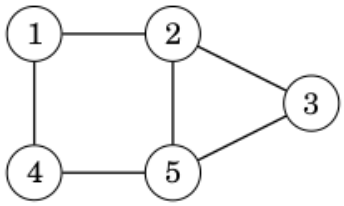
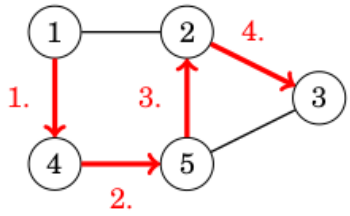
GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: CAMINO Y CICLO DE HAMILTON

1. Introducción

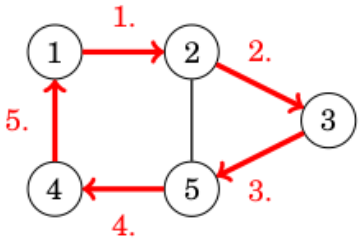
Un **camino hamiltoniano** es un camino que visita cada nodo del grafo exactamente una vez. Por ejemplo, el grafo:



contiene una ruta hamiltoniana desde el nodo 1 al nodo 3:



Si un camino hamiltoniano comienza y termina en el mismo nodo, se llama **circuito o ciclo hamiltoniano**. El grafo anterior también tiene un circuito hamiltoniano que comienza y termina en el nodo 1:



Sobre como determinar si un grafo presenta o no un camino o ciclo de hamilton y como encontrarlo lo veremos en esta guía.

2. Conocimientos previos

2.1. William Rowan Hamilton

Fue un matemático, físico, y astrónomo irlandés, que hizo importantes contribuciones al desarrollo de la óptica, la dinámica, y el álgebra.

Autor: Luis Andrés Valido Fajardo
Email: luis.valido1989@gmail.com
Teléfono: 53694742

2.2. Camino en un grafo

En teoría de grafos, un camino (en inglés, walk,y en ocasiones traducido también como recorrido) es una sucesión de vértices y aristas dentro de un grafo, que empieza y termina en vértices, tal que cada vértice es incidente con las aristas que le siguen y le preceden en la secuencia. Dos vértices están conectados o son accesibles si existe un camino que forma una trayectoria para llegar de uno al otro; en caso contrario, los vértices están desconectados o bien son inaccesibles.

2.3. Ciclo en un grafo

La palabra ciclo se emplea en teoría de grafos para indicar un camino cerrado en un grafo, es decir, en que el nodo de inicio y el nodo final son el mismo.

2.4. Problema *NP-Hard*

Un problema NP-completo es uno que puede convertirse en otro problema NP en un tiempo polinomial razonable. Así, si se tiene un algoritmo P para una incógnita NP-completa, entonces se tiene un algoritmo P para todos los problemas NP y NP=P. Se puede entonces pensar en un ejercicio de este tipo como una representación universal de estos casos.

Un problema NP-difícil es cualquier tipo de problema, no necesariamente en el conjunto de los NP, que puede ser reducido a uno NP-completo en un tiempo polinomial razonable. En este sentido, los problemas NP-difíciles son al menos tan difíciles como los problemas NP-completos, pero podrían ser mucho más difíciles. Por ejemplo, supongamos que se tiene una solución exponencial a un problema NP-difícil que puede ser convertida a una solución exponencial de un problema NP. Se tiene un tiempo exponencial de solución al problema NP, pero puede que tenga otra solución que trabaje en tiempo polinomial.

2.5. Grado del nodo

El grado de un nodo es la cantidad de aristas que inciden en él. Una arista incide en un nodo si lo conecta con otro nodo (o consigo mismo, en el caso de un bucle). Dicho de otro modo: una arista incide en un nodo si lo toca con uno de sus dos lados.

2.6. Algoritmo de retroceso (*backtracking*)

Los algoritmos de vuelta atrás o retroceso (backtracking en inglés) se basan en recorrer el espacio completo de las soluciones posibles al problema planteado. Esta técnica es la aplicación directa del método de búsqueda conocido como primero en profundidad. Típicamente, los algoritmos de vuelta atrás no realizan ningún tipo de optimización y recorren el árbol de soluciones completo. Sin embargo, es posible aplicarles una poda para no descender en aquellas ramas que, de antemano, se sabe que no conducen a una solución.

2.7. Programación Dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas.

La teoría de programación dinámica se basa en una estructura de optimización, la cual consiste en descomponer el problema en subproblemas (más manejables). Los cálculos se realizan entonces recursivamente donde la solución óptima de un subproblema se utiliza como dato de entrada al siguiente problema. Por lo cual, se entiende que el problema es solucionado en su totalidad, una vez se haya solucionado el último subproblema. Dentro de esta teoría, Bellman desarrolla el Principio de Optimalidad, el cual es fundamental para la resolución adecuada de los cálculos recursivos. Lo cual quiere decir que las etapas futuras desarrollan una política óptima independiente de las decisiones de las etapas predecesoras. Es por ello, que se define a la programación dinámica como una técnica matemática que ayuda a resolver decisiones secuenciales interrelacionadas, combinándolas para obtener de la solución óptima.

2.8. Máscara de bits

Esta idea algorítmica permite generar todos los posibles subconjuntos de un conjunto de elementos y puede ser utilizados en problemas en donde se debe contar la cantidad de subcojuntos de un conjunto inicial que cumplan con determinadas condiciones. En este caso la máscara serviría para generar todos los posibles subconjuntos y luego bastaría con chequear cada subconjunto generado y contar con aquellos que cumplan con restricciones o condiciones que imponga el problema.

3. Desarrollo

No se conoce ningún método eficiente para probar si un grafo contiene una ruta hamiltoniana, y el problema es *NP-Hard*. Aún así, en algunos casos especiales, podemos estar seguros de que un grafo contiene un camino hamiltoniano.

Una observación simple es que si el grafo está completo, es decir, hay una arista entre todos los pares de nodos, también contiene un camino hamiltoniano. También se han logrado resultados más fuertes:

- **Teorema de Dirac:** Si el grado de cada nodo es al menos $N/2$ donde N es la cantidad de nodos del grafo, el grafo contiene un camino hamiltoniano.
- **Teorema de Ore:** si la suma de grados de cada par de nodos no adyacentes es al menos N donde N es la cantidad de nodos del grafo, el grafo contiene un camino hamiltoniano.

Una propiedad común en estos teoremas y otros resultados es que garantizan la existencia de una ruta hamiltoniana si el grafo tiene una gran cantidad de aristas. Esto tiene sentido, porque cuantas más aristas contiene el grafo, más posibilidades hay para construir un camino hamiltoniano.

La mejor aportación en este sentido es un teorema publicado en 1976 debido a J. A. Bondy y a V. Chvátal, que generaliza los resultados anteriormente encontrados por G. A. Dirac (1952) y Ø. Ore (1960). Todos estos resultados afirman, básicamente, que un grafo es hamiltoniano si existen suficientes aristas. Para enunciar el Teorema de Bondy-Chvátal es menester definir primero qué es la clausura de un grafo.

Definición Dado un grafo G con n vértices, la clausura de G es el grafo que tiene los mismos vértices que G y que aparece al agregar todas las aristas de la forma u, v para cualquier par de vértices u y v que no sean adyacentes y cumplan que $grado(v) + grado(u) \geq n$.

Teorema de Bondy-Chvátal: Un grafo es hamiltoniano si y sólo si lo es su clausura.

Dado que no hay una forma eficiente de verificar si existe una ruta hamiltoniana, está claro que tampoco hay un método para construir eficientemente la ruta, porque de lo contrario podríamos tratar de construir la ruta y ver si existe.

Una forma simple de buscar una ruta hamiltoniana es usar un algoritmo de retroceso (*backtracking*) que pase por todas las formas posibles de construir la ruta.

Una solución más eficiente se basa en la programación dinámica. La idea es calcular los valores de una función $posible(S, X)$, donde S es un subconjunto de nodos y X es uno de los nodos. La función indica si hay una ruta hamiltoniana que visita los nodos de S y termina en el nodo X.

3.1. Algoritmo de retroceso

3.1.1. Verificar si existe ciclo

Este problema se puede resolver utilizando la siguiente idea. Genere todas las configuraciones posibles de vértices e imprima una configuración que satisfaga las restricciones dadas. Habrá $n!$ configuraciones. Por lo tanto, la complejidad temporal general de este enfoque será O($N!$).

Cree una matriz de ruta vacía y agréguele el vértice 0. Agregue otros vértices, comenzando desde el vértice 1. Antes de agregar un vértice, verifique si es adyacente al vértice agregado anteriormente y si aún no está agregado. Si encontramos dicho vértice, lo agregamos como parte de la solución. Si no encontramos un vértice, devolvemos falso y sacamos el último vertice adicionado y buscamos un nuevo nodo vecino no visitado del nodo que se encuentra al final de la mtriz que indica la matriz. Cuando la cantidad de nodos en la matriz sea igual a la cantidad de nodos y el nodo inicial sea vecino del nodo final entonces encontramos un ciclo.

3.1.2. Imprimir todos los posibles ciclos de Hamilton grafo no dirigido

El problema dado se puede resolver utilizando Backtracking para generar todos los ciclos hamiltonianos posibles. Siga los pasos a continuación para resolver el problema:

1. Cree una matriz auxiliar, digamos $path[]$ para almacenar el orden de recorrido de los nodos y una matriz booleana $visited[]$ para realizar un seguimiento de los vértices incluidos en la ruta actual.
2. Inicialmente, agregue el vértice de origen a $path$.

3. Ahora, agregue recursivamente vértices a $path$ uno por uno para encontrar el ciclo.
4. Antes de agregar un vértice a $path$, verifique si el vértice que se está considerando es adyacente al vértice agregado previamente o no y si aún no se encuentra en $path$. Sino se encuentra dicho vértice, agréguelo a $path$ y marque su valor como verdadero en la matriz $visited[]$.
5. Si la longitud de $path$ se vuelve igual a N y hay una arista desde el último vértice en $path$ hasta el primer vertice almacenado en $path$, imprima la matriz $path$.
6. Después de completar los pasos anteriores, si no existe dicha ruta, entonces no existe ciclo hamiltoniano.

3.2. Programación Dinamica

3.2.1. Verificar si existe camino

El enfoque más simple para resolver el problema dado es generar todas las permutaciones posibles de N vértices. Para cada permutación, verifique si es una ruta hamiltoniana válida verificando si hay una arista entre los vértices adyacentes o no.

El enfoque anterior se puede optimizar mediante el uso de programación dinámica y enmascaramiento de bits, que se basa en las siguientes observaciones:

- La idea es tal que para cada subconjunto S de vértices, comprobar si existe un camino hamiltoniano en el subconjunto S que termina en el vértice v donde $v \in S$.
- Si v tiene un vecino u, donde $u \in S - v$, por lo tanto, existe un camino hamiltoniano que termina en el vértice u.
- El problema se puede resolver generalizando el subconjunto de vértices y el vértice final del camino hamiltoniano.

Siga los pasos a continuación para resolver el problema:

1. Inicialice una matriz booleana $dp[][]$ en la dimensión $N * 2^N$ donde $dp[j][i]$ representa si existe o no una ruta en el subconjunto representada por la máscara i que visita todos y cada uno de los vértices en i una vez y termina en vértice j.
2. Para el caso base, actualice $dp[i][1 << i] =$ verdadero, para i en el rango $[0, N - 1]$
3. Itere sobre el rango $[1, 2^N - 1]$ usando la variable i y realice los siguientes pasos:
   - Todos los vértices con bits establecidos en la máscara i están incluidos en el subconjunto.
   - Itere sobre el rango $[1, N]$ usando la variable j que representará el vértice final de la ruta hamiltoniana de la máscara del subconjunto actual i y realice los siguientes pasos:
     - Si el valor de i y 2^j es verdadero, entonces itere sobre el rango $[1, N]$ usando la variable k y si el valor de $dp[k][i \quad xor \quad 2^j]$ es verdadero, entonces marque $dp[j][i]$

es cierto y sale del ciclo.

- De lo contrario, continúe con la siguiente iteración.

4. Itere sobre el rango usando la variable i y si el valor de $dp[i][2^N - 1]$ es verdadero, entonces existe una ruta hamiltoniana que termina en el vértice i.

3.2.2. Calcular todos los caminos de hamilton del nodo 1 al nodo N en grafo dirigido

El problema dado se puede resolver de igual manera que el anterior usando máscara de bits con programación dinámica con algunas modificaciones, iterando sobre todos los subconjuntos de los vértices dados representados por una máscara de tamaño N y verificando si existe una ruta hamiltoniana que comienza en el vértice 1 y termina en vértice N y contar todos esos caminos. Digamos que para un grafo que tiene N vértices, S representa una máscara de bits donde S esta en una rango $[0, (1 << N) - 1]$ y $dp[i][S]$ representa el número de rutas que visitan cada vértice en la máscara S y terminan en el vertice i entonces la recurrencia válida será dada como $dp[i][S] = \sum dp[j][S \quad XOR \quad 2^i]$ donde $j \in S$ y hay una arista de j a i donde $S \quad XOR \quad 2^i$ representa el subconjunto donde el i-ésimo vértice no esta y debe haber una arista de j a i.

1. Inicialice una matriz 2D $dp[N][2^N]$ con 0 y establezca $dp[0][1]$ como 1.
2. Itere sobre el rango de $[2, 2^N - 1]$ usando la variable i y verifique que la máscara tenga todos los bits configurados.
   - Itere sobre el rango desde $[0, N)$ usando la variable end y recorra todos los bits de la máscara actual y asuma cada bit como el bit final.
     - Inicialice la variable $prev$ como $i - (1 << end)$.
     - Itere sobre el rango $[0, size)$ donde $size$ es el tamaño de la lista de adyacencia al nodo end usando la variable it y recorra los vértices adyacentes del bit final actual y actualice la matriz $dp[][]$ de esta manera $dp[end][i] += dp[it][prev]$.
3. Después de realizar los pasos anteriores, imprima el valor de $dp[N - 1][2^N - 1]$ como respuesta.

4. Implementación

4.1. C++

4.1.1. Verificar si existe ciclo (Algoritmo retroceso)

```
#define V 5 //cantidad de vertices

void printSolution(int path[]);

bool isSafe(int v, bool graph[V][V], int path[], int pos) {
   if (graph [path[pos - 1]][ v ] == 0) return false;
   for (int i = 0; i < pos; i++) if (path[i] == v) return false;
```

```
   return true;
}

bool hamCycleUtil(bool graph[V][V], int path[], int pos){
   if (pos == V) {
      if (graph[path[pos - 1]][path[0]] == 1) return true;
      else return false;
   }

   for (int v = 1; v < V; v++){
      if (isSafe(v, graph, path, pos)){
         path[pos] = v;
         if (hamCycleUtil (graph, path, pos + 1) == true) return true;
         path[pos] = -1;
      }
   }
   return false;
}

//Se invoca este metodo pasando la matriz de adyacencia del grafo
bool hamCycle(bool graph[V][V]){
   int *path = new int[V];
   for (int i = 0; i < V; i++) path[i] = -1;

   path[0] = 0;
   if (hamCycleUtil(graph, path, 1) == false ){
      cout << "\nNo existe una solucion"; return false;
   }
   printSolution(path); return true;
}


void printSolution(int path[]) {
   cout << "Existe solucion: El siguiente es un ciclo de Hamilton \n";
   for (int i = 0; i < V; i++) cout << path[i] << " ";
   cout << path[0] << " ";  cout << endl;
}
```

4.1.2. Imprimir todos los posibles ciclos de Hamilton grafo no dirigido (Algoritmo retroceso)

```
#define V 6 //cantidad de vertices
bool hasCycle;

bool isSafe(int v, int graph[][V], vector<int> path, int pos){
   if (graph[path[pos - 1]][v] == 0) return false;
   for (int i = 0; i < pos; i++) if (path[i] == v) return false;
   return true;
}
```

```
void FindHamCycle(int graph[][V], int pos, vector<int> path, bool visited[],
    int N) {
   if (pos == N) {
      if (graph[path[path.size() - 1]][path[0]] != 0) {
         path.push_back(0);
         for (int i = 0; i < path.size(); i++) cout << path[i] << " ";
         cout << endl;
         path.pop_back();
         hasCycle = true;
      }
      return;
   }
   for (int v = 0; v < N; v++) {
      if (isSafe(v, graph, path, pos) && !visited[v]) {
         path.push_back(v); visited[v] = true;
         FindHamCycle(graph, pos + 1, path, visited, N);
         visited[v] = false; path.pop_back();
      }
   }
}

void hamCycle(int graph[][V], int N){
   hasCycle = false;
   vector<int> path;
   path.push_back(0);
   bool visited[N];
   for (int i = 0; i < N; i++) visited[i] = false;
   visited[0] = true;
   FindHamCycle(graph, 1, path, visited, N);
   if (!hasCycle) {
     cout << "No existe ciclo hamiltoniano " << endl; return;
   }
}
```

4.1.3. Verificar si existe camino (Programación Dinámica)

```
bool Hamiltonian_path( vector<vector<int> >& adj, int N){
   int dp[N][(1 << N)];
   memset(dp, 0, sizeof dp);

   for (int i = 0; i < N; i++) dp[i][(1 << i)] = true;

   for (int i = 0; i < (1 << N); i++) {
      for (int j = 0; j < N; j++) {
         if (i & (1 << j)) {
            for (int k = 0; k < N; k++) {
                if (i&(1<<k) && adj[k][j] && j!=k && dp[k][i ^ (1 << j)]){
```

```
                        dp[j][i] = true; break;
                    }
                }
            }
        }
    }

    for (int i = 0; i < N; i++) {
        if (dp[i][(1 << N) - 1]) return true;
    }
    return false;
}
```

4.1.4. Calcular todos los caminos de hamilton del nodo 1 al nodo N en grafo dirigido

```
int findAllPaths( int N, vector<vector<int> >& graph){
    int dp[N][(1 << N)];
    memset(dp, 0, sizeof dp);
    dp[0][1] = 1;
    for (int i = 2; i < (1 << N); i++) {
        // Si el primer vertice esta ausente
        if ((i & (1 << 0)) == 0) continue;

        // Considere solo los subconjuntos completos
        if ((i & (1 << (N - 1))) && i != ((1 << N) - 1)) continue;

        // Elige la ciudad final
        for (int end = 0; end < N; end++) {
          // Si esta ciudad no esta en el subconjunto
          if (i & (1 << end) == 0) continue;
          // Conjunto sin la ciudad final
          int prev = i - (1 << end);
          // Consultar por las ciudades adyacentes.
          for (int it : graph[end]) {
              if ((i & (1 << it))) {
                 dp[end][i] += dp[it][prev];
              }
          }
        }
    }
    return dp[N - 1][(1 << N) - 1];
}
```

4.2. Java

4.2.1. Verificar si existe ciclo (Algoritmo retroceso)

```
class HamiltonianCycle {
   final int V = 5; //cantidad de vertices del grafo
   int path[]; // secuencia que indica un posible ciclo

   boolean isSafe(int v, int graph[][], int path[], int pos){
      if (graph[path[pos - 1]][v] == 0) return false;
      for (int i = 0; i < pos; i++) if (path[i] == v) return false;
      return true;
   }

   boolean hamCycleUtil(int graph[][], int path[], int pos){
      if (pos == V){
         if (graph[path[pos - 1]][path[0]] == 1) return true;
         else return false;
      }

      for (int v = 1; v < V; v++) {
         if (isSafe(v, graph, path, pos)) {
            path[pos] = v;
            if (hamCycleUtil(graph, path, pos + 1) == true) return true;
            path[pos] = -1;
         }
       }
        return false;
   }

   int hamCycle(int graph[][]) {
      path = new int[V];
      for (int i = 0; i < V; i++) path[i] = -1;

      path[0] = 0;
      if (hamCycleUtil(graph, path, 1) == false) {
         System.out.println("\nNo existe ciclo"); return 0;
      }
      printSolution(path); return 1;
   }

   void printSolution(int path[]){
      System.out.println("Existe solucion : El siguiente" +" es un ciclo de
         Hamilton");
      for (int i = 0; i < V; i++) System.out.print(" " + path[i] + " ");
      System.out.println(" " + path[0] + " ");
   }
}
```

4.2.2. Imprimir todos los posibles ciclos de Hamilton grafo no dirigido (Algoritmo retroceso)

```
import java.util.ArrayList;
```

```
public class Main {

   public boolean isSafe(int v,int graph[][],ArrayList<Integer> path,int pos){
      if (graph[path.get(pos - 1)][v] == 0) return false;
      for (int i = 0; i < pos; i++) if (path.get(i) == v) return false;
      return true;
   }

   public boolean hasCycle;

   void hamCycle(int graph[][]){
      hasCycle = false;

      ArrayList<Integer> path = new ArrayList<>();
      path.add(0);

      boolean[] visited = new boolean[graph.length];

      for (int i = 0; i < visited.length; i++) visited[i] = false;
      visited[0] = true;

      FindHamCycle(graph, 1, path, visited);

      if (!hasCycle) {
         System.out.println("No Hamiltonian Cycle" + "possible "); return;
      }
   }

   void FindHamCycle(int graph[][], int pos, ArrayList<Integer> path, boolean
       [] visited){
      if (pos == graph.length) {
         if (graph[path.get(path.size() - 1)][path.get(0)] != 0) {
            path.add(0);
            for (int i = 0; i < path.size(); i++) {
                System.out.print(path.get(i) + " ");
            }
            System.out.println();
            path.remove(path.size() - 1);
            hasCycle = true;
         }
         return;
      }

      for (int v = 0; v < graph.length; v++) {
         if (isSafe(v, graph, path, pos) && !visited[v]) {
            path.add(v); visited[v] = true;
            FindHamCycle( graph, pos + 1,path, visited);
            visited[v] = false; path.remove(path.size() - 1);
         }
      }
```

```
    }
}
```

4.2.3. Verificar si existe camino (Programación Dinámica)

```
class Main{
   static boolean Hamiltonian_path(int adj[][], int N) {
      boolean dp[][] = new boolean[N][(1 << N)];
      for(int i = 0; i < N; i++) dp[i][(1 << i)] = true;

      for(int i = 0; i < (1 << N); i++){
          for(int j = 0; j < N; j++){
              if ((i & (1 << j)) != 0){
                for(int k = 0; k < N; k++){
                   if ((i & (1 << k)) != 0 &&  adj[k][j] == 1 && j != k && dp[
                       k][i ^ (1 << j)]) {
                      dp[j][i] = true; break;
                   }
                }
              }
          }
       }

       for(int i = 0; i < N; i++) {
          if (dp[i][(1 << N) - 1]) return true;
       }
           return false;
   }
}
```

4.2.4. Calcular todos los caminos de hamilton del nodo 1 al nodo N en grafo dirigido

```
class Main {
   static int findAllPaths(int N, List<List<Integer>> graph){
      int dp[][] = new int[N][(1<<N)];
      for(int i=0;i<N;i++){
         for(int j=0;j<(1<<N);j++){
            dp[i][j]=0;
         }
      }

      // Inicializar para el primer vertice
      dp[0][1] = 1;

      // Iterar sobre todas las mascaras.
      for (int i = 2; i < (1 << N); i++) {
         // Si el primer vertice esta ausente
```

```
            if ((i & (1 << 0)) == 0) continue;
            // Considere solo los subconjuntos completos
            if ((i & (1 << (N - 1)))==1 && (i != ((1 << N) - 1))) continue;
            // Elige la ciudad final
            for (int end = 0; end < N; end++) {
               // Si esta ciudad no esta en el subconjunto
               if ((i & (1 << end)) == 0) continue;
               // Conjunto sin la ciudad final
               int prev = i - (1 << end);
               // Consultar por las ciudades adyacentes.
               for (int it : graph.get(end)) {
                  if ((i & (1 << it))!=0)
                     dp[end][i] += dp[it][prev];
               }
            }
         }
         return dp[N - 1][(1 << N) - 1];
      }
   }
```

5. Aplicaciones

Dentro de las aplicaciones del camino y ciclo de Hamilton podemos mencionar:

1. **Enrutamiento de red:** Los caminos de Hamilton se pueden utilizar para encontrar el camino más corto entre dos puntos en una red, lo que los hace útiles para el enrutamiento en redes de transporte y comunicación.
2. **Diseño de circuitos:** En el diseño de circuitos, las rutas de Hamilton se pueden utilizar para encontrar el diseño más eficiente para conectar componentes, minimizando la longitud de los cables y reduciendo costos.
3. **Secuenciación de ADN:** El problema de la ruta de Hamilton se puede aplicar a la secuenciación de ADN, donde el objetivo es encontrar la secuencia más corta que contenga todos los fragmentos de ADN dados.
4. **Programación:** En problemas de programación, las rutas de Hamilton se pueden utilizar para encontrar la secuencia más eficiente de tareas o eventos, minimizando el tiempo o costo total.
5. **Planificación de viajes:** En turismo y planificación de viajes, los caminos de Hamilton se pueden utilizar para crear rutas óptimas para visitar múltiples destinos, maximizando la cantidad de lugares visitados y minimizando el tiempo de viaje.
6. **Robótica:** En robótica, las rutas de Hamilton se pueden utilizar para planificar la ruta más eficiente que debe tomar un robot mientras completa una tarea, como recoger y entregar artículos en un almacén.

7. **Gestión de la cadena de suministro:** Las rutas de Hamilton se pueden utilizar en la gestión de la cadena de suministro para optimizar el flujo de mercancías y minimizar los costos de transporte.
8. **Diseño VLSI:** en el diseño VLSI (integración a muy gran escala), las rutas de Hamilton se pueden utilizar para determinar el mejor diseño para conectar transistores en un chip, reduciendo el tamaño total y el costo del chip.
9. **Teoría de juegos:** En teoría de juegos, los caminos de Hamilton se pueden utilizar para analizar y optimizar estrategias en juegos como el ajedrez o las damas.
10. **Redes de computadoras:** En las redes de computadoras, las rutas de Hamilton se pueden utilizar para encontrar la ruta más eficiente para la transmisión de datos, minimizando los retrasos y maximizando el uso del ancho de banda.

6. Complejidad

La complejidad del tiempo de usar un algoritmo de retroceso (*backtracking*) para hallar una ruta hamiltoniana es al menos O(N!), porque hay N! Diferentes formas de elegir el orden de N! nodos. La variante utilizando la programación dinámica para generar una posible ruta es posible implementar esta con una complejidad O($2^N N^2$).

7. Ejercicios

A continucación una lista de ejercicios que se puede resolver aplicando los elementos y algoritmos abordados en la presente guía:

- CSES - Hamiltonian Flights