Complejidad computacional

Departamento de Informática
Universidad de Valladolid

César González Ferreras

Introducción

- Algoritmia:
  - Diseño y análisis sistemático de algoritmos específicos para resolver un problema dado.
  - Permite demostrar que un algoritmo concreto resuelve un problema en tiempo $O(f(n))$
- Complejidad computacional:
  - Considera globalmente todos los posibles algoritmos para resolver un problema dado.
  - Permite demostrar que cualquier algoritmo capaz de resolver correctamente un problema debe necesariamente requerir un tiempo que está en $\Omega(g(n))$.
  - La función $g(n)$ es la cota inferior de la complejidad del problema.
- El objetivo es conseguir que $f(n) \in \Theta(g(n))$.
  - Es el algoritmo más eficiente posible (salvo quizás la constante multiplicativa).

Técnicas complejidad computacional

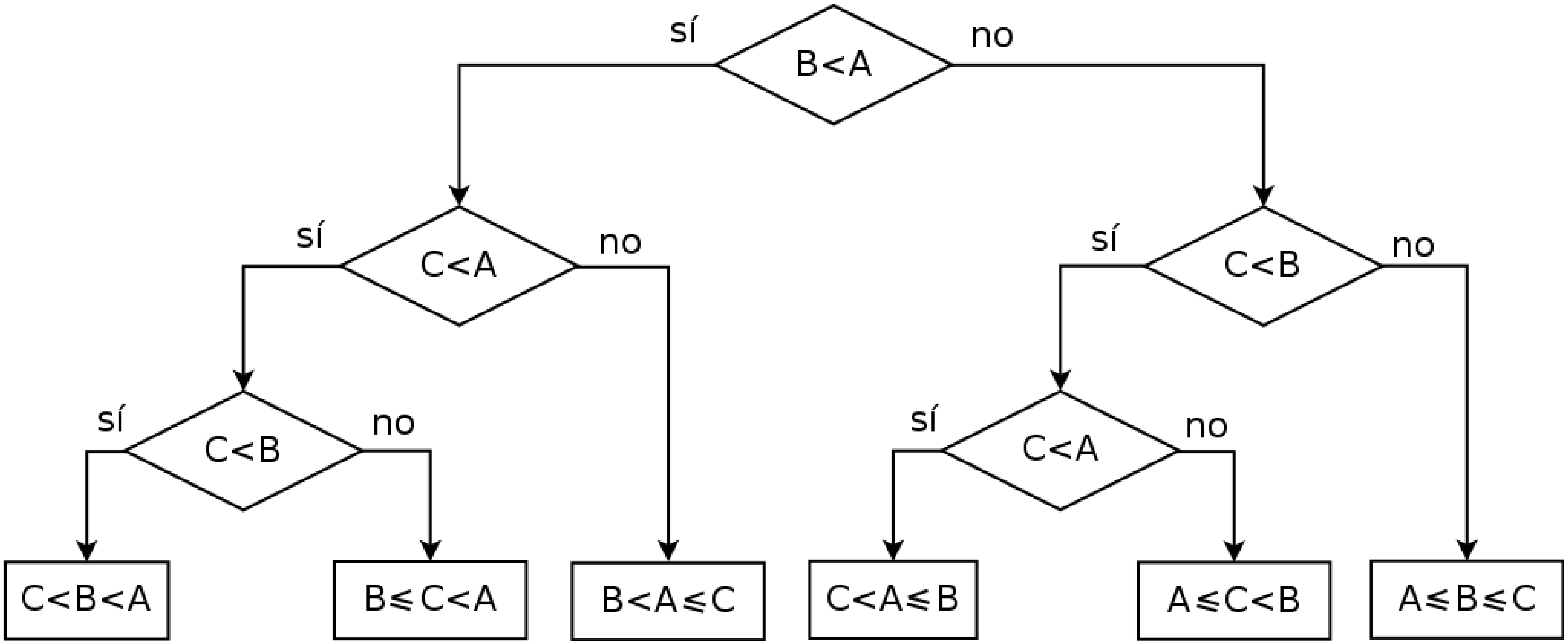
- Argumentos de teoría de la Información:
  - Se suele aplicar a problemas con comparaciones.
  - Se basa en representar el algoritmo mediante un árbol de decisión para todos los datos posibles de un tamaño dado.
  - Medir a partir del árbol el coste en el peor caso.
  - Ejemplo: ordenación por comparaciones.
- Argumentos del adversario:
  - Diseñar un caso de entrada que haga que el algoritmo se encuentre en el peor caso.
  - Ejemplo: hallar el máximo entre n elementos.

Complejidad de la ordenación

- Usaremos argumento de teoría de la Información.
- No puede existir ningún algoritmo de ordenación basado en comparaciones cuyo tiempo de ejecución tenga una cota inferior menor que $\Omega(n \log n)$
  - A todo algoritmo de ordenación por comparación le corresponde, para cada valor de n, un árbol de decisión que es válido para ordenar n elementos.

Complejidad de la ordenación

- Árbol de decisión para la ordenación por inserción de 3 elementos.



Complejidad de la ordenación

- El espacio de posibles soluciones tiene un tamaño n! (posibles permutaciones de un vector de n elementos)
- Todo árbol de decisión válido para ordenar n elementos debe contener al menos n! hojas.
- Por tanto, dicho árbol tendrá una altura mínima de $\log_2(n!)$.
- La altura del árbol de decisión da el número de comparaciones efectuadas por el algoritmo en el peor caso: $\log_2(n!)$.
- $\log_2(n!) \in \Theta(n \log(n))$
- **Para ordenar n elementos se necesita un tiempo que está en $\Omega(n \log(n))$ independientemente del algoritmo basado en comparaciones que se utilice.**

Máximo de un vector

- Usaremos argumento del adversario.
- **Problema**: determinar el máximo de un vector V empleando un algoritmo basado en comparaciones.
- El algoritmo evidente:

```
Función índicemax(T[1..n])
    m ← T[1]
    k ← 1
    Para i ← 2 hasta n hacer
        si T[i] > m entonces
            m ← T[i]
            k ← i
    devolver k
```

- **¿Es posible encontrar un algoritmo mejor?**

Máximo de un vector

- Al comparar los elementos *x* e *y* llamamos perdedor al menor de ellos.
- Cada comparación genera un perdedor.
- Debe haber n-1 perdedores para determinar el máximo de n elementos.
- Demostración:
  - Si asumimos que hay 2 elementos *x* e *y* que nunca han perdido una comparación, siendo *x>y*
  - El algoritmo declara *x* como el máximo elemento.
  - Es posible construir otro vector V’ que es igual a V con la diferencia de que *y* es reemplazado por *y’*, siendo *y’>x>y*.
  - El algoritmo se comportará de igual manera con V’ que con V, puesto que *y’* ganará también todas las comparaciones.
  - El algoritmo declarará *x* como el elemento máximo, lo cual es una contradicción.
- **Para determinar el máximo de un vector empleando un algoritmo basado en comparaciones es necesario al menos n-1 comparaciones.**

LAS CLASES P Y NP

Reducciones lineales

- Es difícil determinar la complejidad exacta de la mayoría de problemas.
- Es útil comparar la dificultad de distintos problemas.
- Diremos que **un problema se reduce a otro**:
  - si podemos transformar eficientemente casos del primer problema en casos del segundo
  - de tal manera que la resolución del caso transformado proporcione la respuesta del caso original.
  - Entonces, los problemas tienen aproximadamente la misma complejidad.
- Ejemplo:
  - Factorización.
  - Comprobación de números primos.

Reducciones lineales

- Existen problemas para los que no se conoce algoritmo eficiente, pero tampoco se ha conseguido demostrar su complejidad intrínseca.
- Ejemplos:
  - Viajante de comercio.
  - Coloreado de grafos.
  - Problema de la mochila.
  - Ciclos hamiltonianos.
  - Satisfacción de una fórmula booleana.
- ¿Existen algoritmos eficientes para estos problemas?

Reducciones lineales

- ¿Son problemas intrínsecamente difíciles aunque no se haya podido demostrar todavía?
- Un algoritmo eficiente para resolver cualquiera de estos problemas proporciona automáticamente algoritmos eficientes para todos ellos.
  - Son problemas de complejidad parecida.
- Si dado un problema se puede demostrar que es computacionalmente equivalente a uno de estos problemas:
  - Podemos concluir que se trata de un problema difícil y que de momento nadie sabe resolverlo eficientemente.

Las clases P y NP

- ¿Qué es un algoritmo eficiente?
- Un algoritmo es eficiente si existe un polinomio p(n) tal que el algoritmo puede resolver cualquier caso de tamaño n en un tiempo que está en O(p(n)).
- Diremos que estos algoritmos son de tiempo polinómico.
- **P es la clase de problemas de decisión que se pueden resolver mediante un algoritmo de tiempo polinómico.**
- Un problema de decisión es un problema que requiere como solución una respuesta **SÍ** o **NO**.

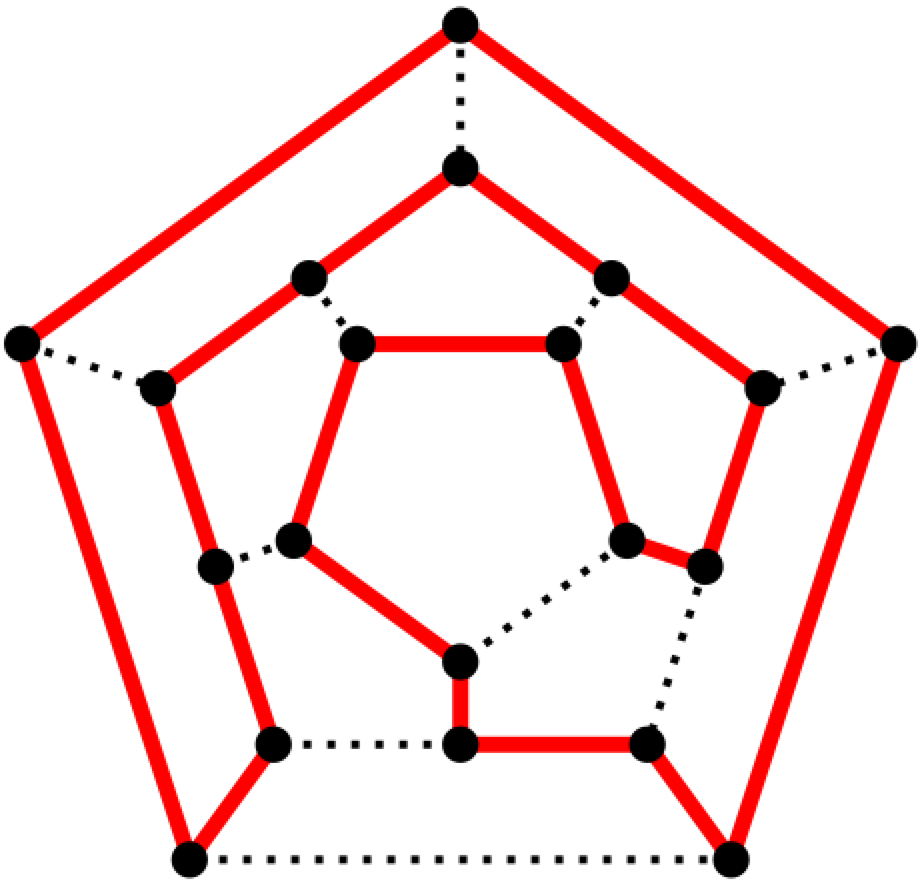
Las clases P y NP

- La **clase NP** consiste de todos aquellos problemas de decisión cuyas soluciones positivas/afirmativas pueden ser verificadas en tiempo polinómico a partir de ser alimentadas con la información apropiada *(definición informal)*.
- La abreviatura NP hace referencia a:
  - "Nondeterministic Polynomial time".
  - **Cuidado: ¡NP no significa "no polinómico"!**

Ciclo Hamiltoniano

- Dado un grafo G=<N,A>.
- El problema consiste en hallar un camino que comience el algún nodo, visite cada nodo exactamente una vez y vuelva al nodo de partida.
- Un grafo es hamiltoniano si existe uno de estos ciclos.
- El problema de determinar si un grafo es hamiltoniano está en NP:
  - Se puede verificar en tiempo lineal si una sucesión de nodos forma un ciclo hamiltoniano.

Ciclo Hamiltoniano



Las clases P y NP

- Relación entre P y NP:

$$P \subseteq NP$$

- ¿Es posible que P = NP?
  - No ha sido posible demostrar si es o no cierto.
  - Aunque se piensa que P ≠ NP.

La clase NP-completo

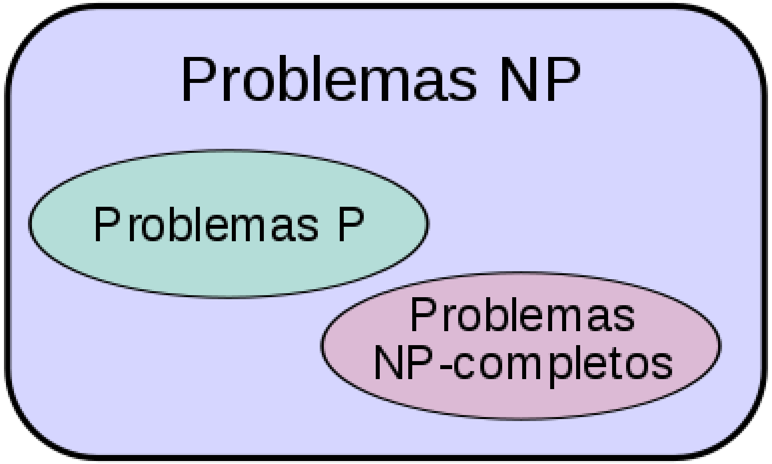
- La clase de complejidad **NP-completo** es el subconjunto de los problemas de decisión en NP tal que todo problema en NP se puede reducir en cada uno de los problemas de NP-completo.
- Se puede decir que los problemas de NP-completo son los problemas más difíciles de NP.
- Con la conjetura $P \neq NP$, sabemos que los problemas NP-completos no se pueden resolver en tiempo polinómico.

La clase NP-completo

- Existe un número elevado de problemas NP-completos.
- Si cualquiera de ellos estuviera en P, entonces P=NP.
- Hasta ahora no ha sido posible encontrar un algoritmo de tiempo polinómico para resolver un problema NP-completo.
- **Eso nos lleva a pensar que P ≠ NP (aunque no está demostrado).**

Clases P, NP y NP-completo

- Si la conjetura P ≠ NP es cierta, entonces:

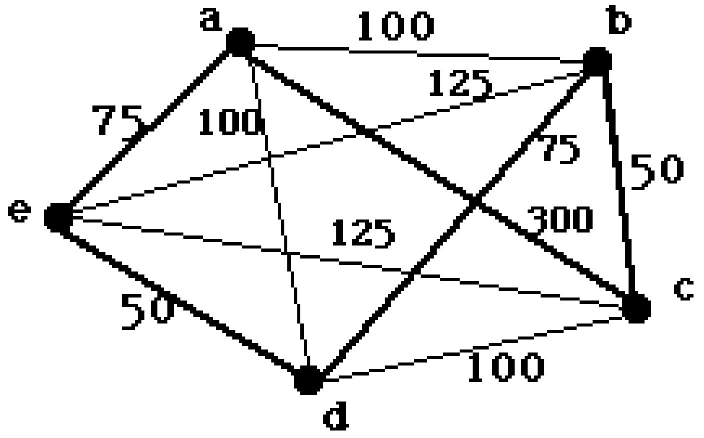


Problemas NP-completos

- Problemas NP-completos:
  - Encontrar un ciclo hamiltoniano.
  - Viajante de comercio.
  - Satisfacibilidad booleana.
  - Coloreado de grafos.
  - Mochila.
  - Subconjunto suma.

Viajante de comercio

- Encontrar un recorrido del grafo que empiece y termine en algún nodo, después de haber visitado todos los demás nodos exactamente una vez y cuyo coste sea el mínimo posible.



Coste del camino AEDBCA = 550

Satisfacibilidad booleana

- Una fórmula boolena es satisfactoria si existe al menos una forma de asignar valores a sus variables de tal modo que resulte ser verdadera.
- El problema de decidir si una fórmula booleana es o no satisfactoria es NP-completo.
- Por ejemplo:

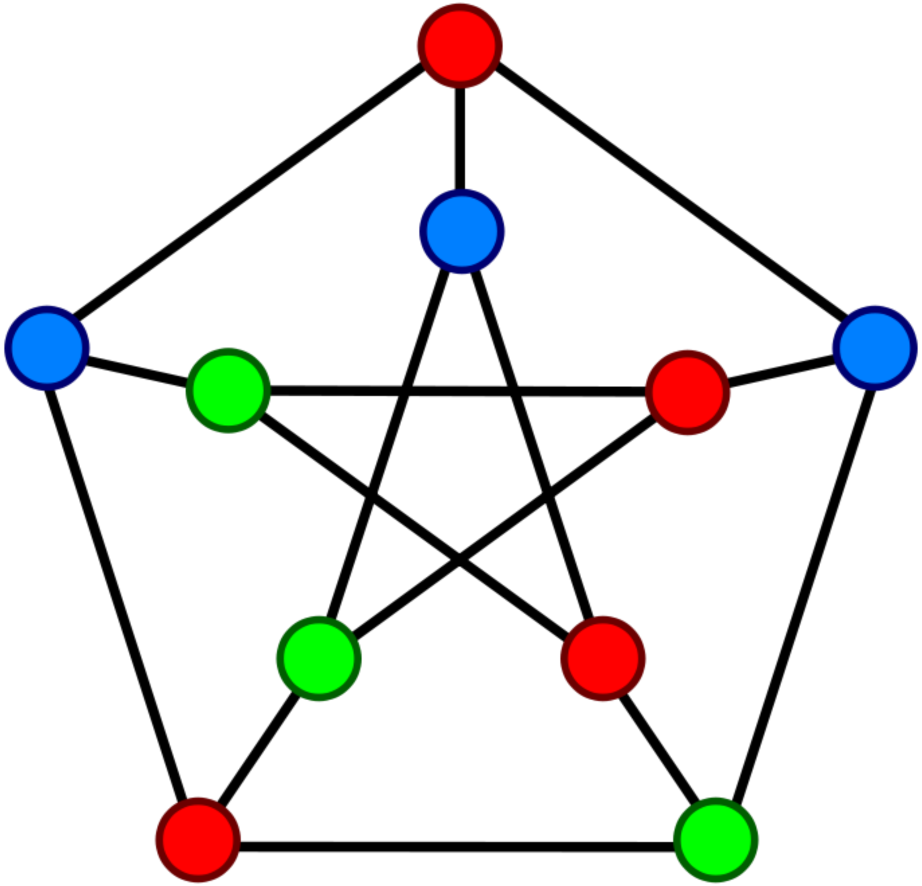
$$(p \vee q) \Rightarrow (p \wedge q)$$

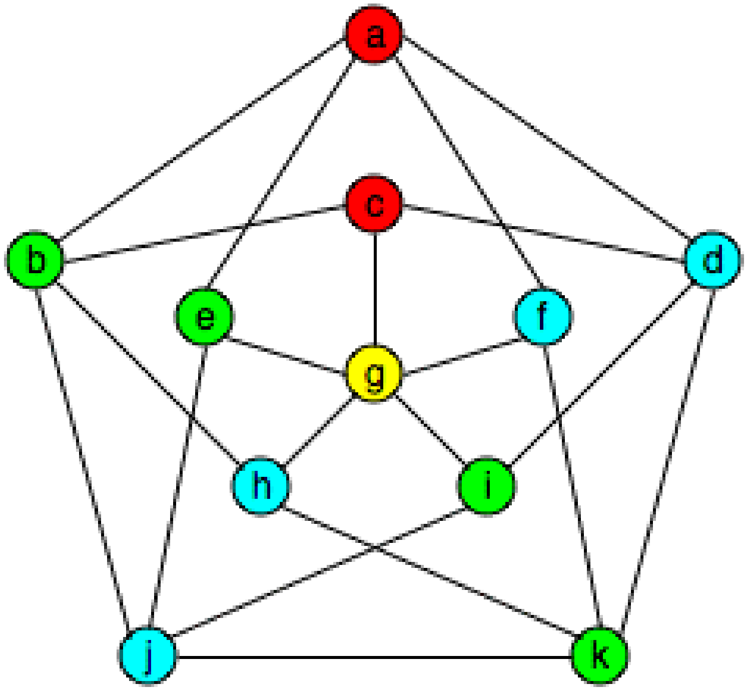
  - Es satisfactoria porque es *verdadero* si asignamos *verdadero* a *p* y a *q*.

Coloreado de grafos

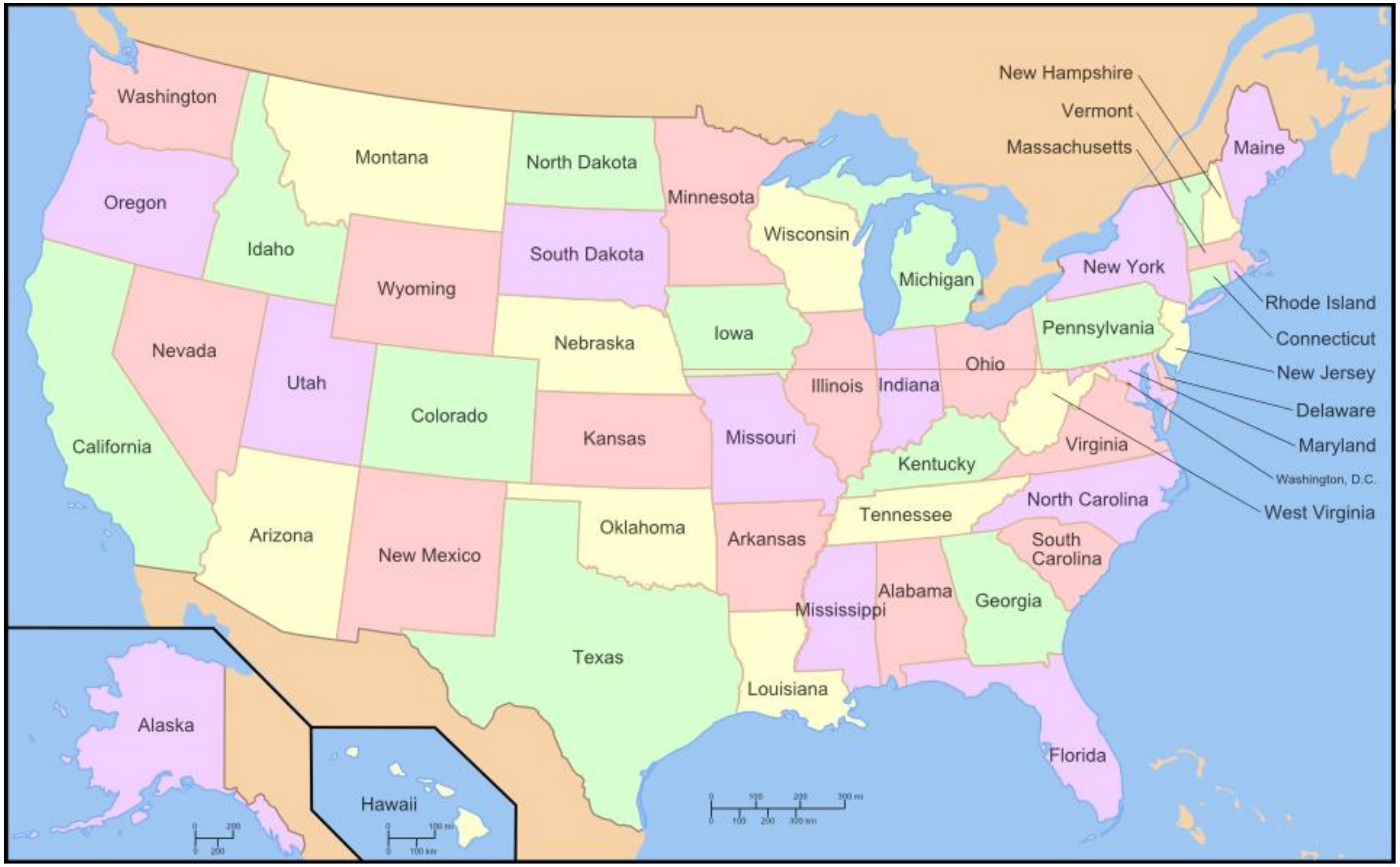
- Sea G un grafo no dirigido.
- Un coloreado de G es una asignación de colores a los nodos de G de tal manera que dos nodos cualesquiera que estén conectados por una arista sean siempre de diferentes colores.
- Dado un grafo G y un entero k ¿se puede colorear G con k colores diferentes?
- El problema es NP-completo excepto para k=1 y k=2.

Coloreado de grafos

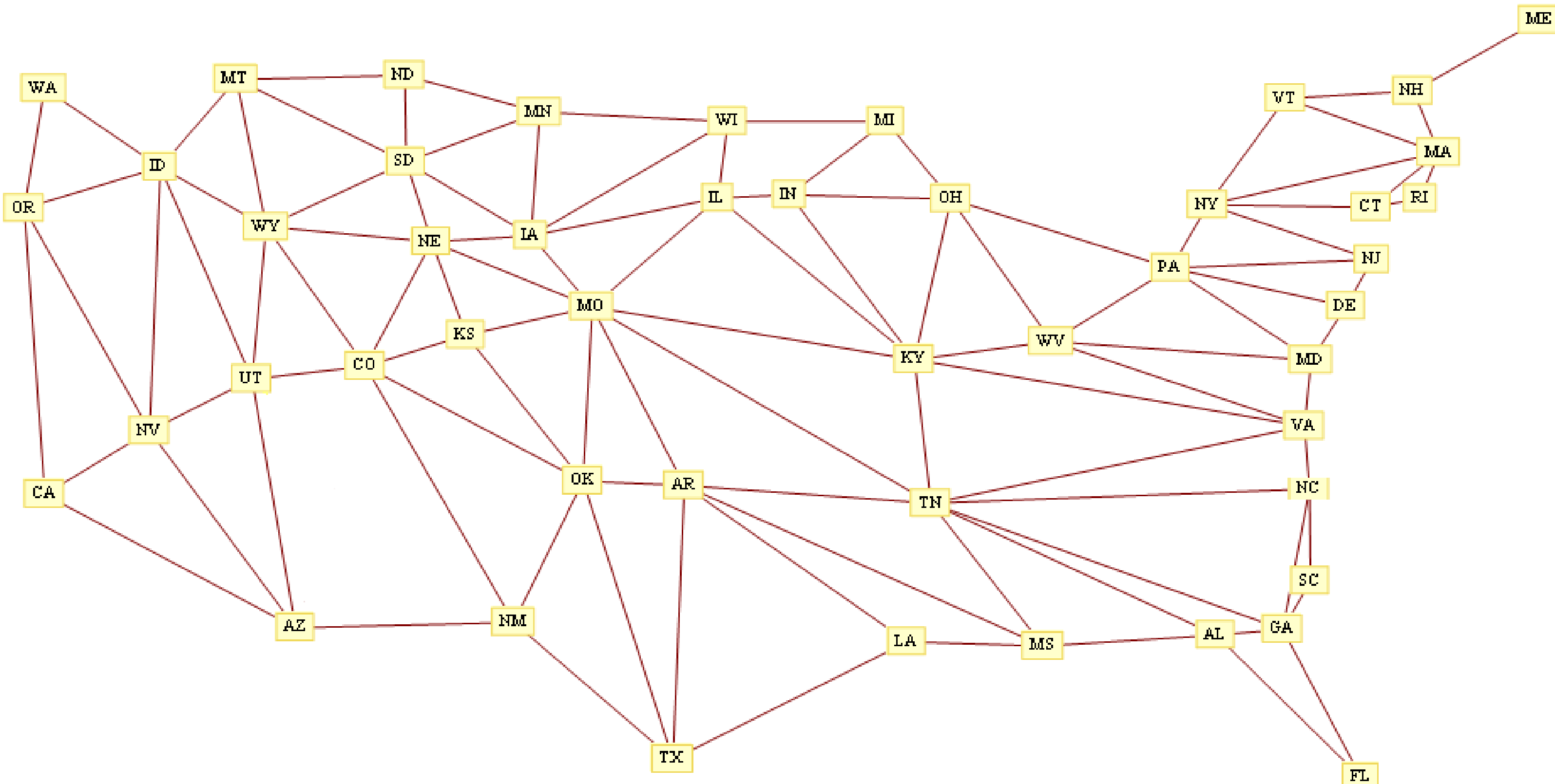




Coloreado de grafos



Coloreado de grafos



Subconjunto suma

- Dado un conjunto de enteros y un entero s, ¿existe algún subconjunto cuya suma sea s?
- Ejemplo:
  - Dado el conjunto {3, 34, 4, 12, 5, 2}
  - ¿Algún subconjunto suma 9?
  - Solución: {4, 5}