Estrategias algorítmicas

- Dividir para conquistar
- Backtracking
- Algoritmos codiciosos
- Programación dinámica
- Branch & bound

Estrategias algorítmicas

- **Backtracking**
- Algoritmos codiciosos
- Programación dinámica

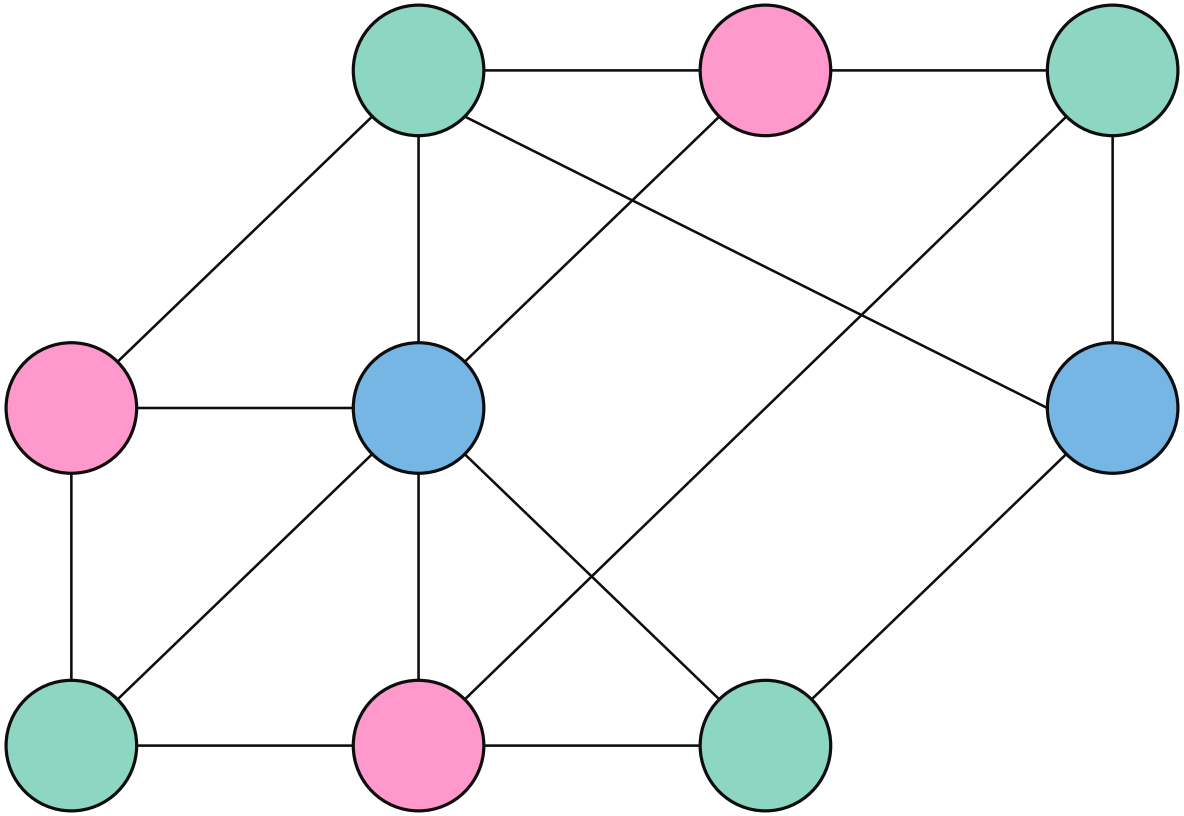
Motivación: Coloración de grafos

Dado un grafo no dirigido $\boldsymbol{G(V, E)}$ y 3 colores, llamamos una **3-coloración** del grafo a una asignación tal que

- Cada nodo tiene asignado color y sólo uno
- Nodos vecinos tienen colores distintos

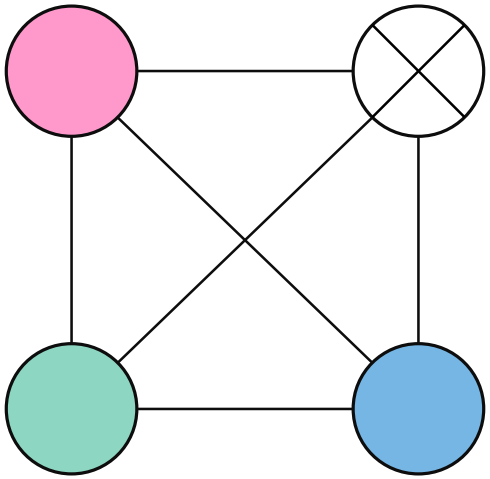
Queremos determinar si un grafo es 3-coloreable

Coloración de grafos



Este grafo es 3-coloreable. ¿Cómo podemos probarlo?

Coloración de grafos



Este grafo no es 3-coloreable. ¿Cómo podemos probarlo?

Constraint Satisfaction Problems

Problemas como este se llaman de **satisfacción de restricciones**

Es una familia entera de problemas de combinatoria

¿Cómo podríamos generalizarlo?

CSPs en general

Tenemos un conjunto de **variables**

Cada variable puede tomar ciertos valores: tiene un **dominio**

Las **restricciones** prohíben ciertas combinaciones de valores

¿Cómo se vería esto en el problema de coloración de grafos?

Modelación de 3-coloración

Tenemos una variable por cada nodo del grafo

El dominio de cada variable es $\{1, 2, 3\}$

Cada arista prohíbe que dos variables tengan el mismo color

¿Cómo lo resolvemos?

Si el problema tiene solución, queremos una garantía

Si no tiene solución, también queremos una garantía

¿Cómo hacemos esto?

Fuerza bruta

Una forma es generar todas las permutaciones posibles

Luego para cada permutación verificar si cumple todas las restricciones

Esta estrategia se conoce como **fuerza bruta**

Sin embargo no es necesario probar **todas** las permutaciones posibles…

¿Es posible?

Dado un problema, ¿es posible resolverlo?

La idea es responder esa pregunta recursivamente

Si asignamos una variable, ¿qué nos queda?

$3 - col\big(G(V, E)\big)$:

$3 - col\big(G(V, E)\big)$:

Sea v un nodo sin color en V

$for\ c \in \{1, 2, 3\}$:

$if\ is\ valid(v, c, E)$:

$v.\text{color} = c$

$3 - col\big(G(V, E)\big)$:

Sea v un nodo sin color en V

$for\ c \in \{1, 2, 3\}$:

$if\ is\ valid(v, c, E)$:

$v.\text{color} = c$

$if\ 3 - col\big(G(V, E)\big)$:

$return\ true$

$v.\text{color} = 0$

$return\ false$

$3 - col\big(G(V, E)\big)$:

    if Todos los nodos tienen color:

        return true

    Sea v un nodo sin color en V

    for $c \in \{1, 2, 3\}$:

        if is valid(v, c, E):

            $v.\text{color} = c$

            if $3 - col\big(G(V, E)\big)$:

                return true

            $v.\text{color} = 0$

    return false

Backtracking

Esta estrategia se conoce como **backtracking**

La idea es **descartar** permutaciones que violan alguna restricción

Eso significa que **siempre** es igual o más rápido que fuerza bruta

```
is solvable(X, R):
    if is solution(X, R):
        return true
    x = choose unassigned variable(X)
    for v ∈ x.d:
        if is valid(x, v, R):
            assign(x, v)
            if is solvable(X, R):
                return true
            unassign(x, v)
    return false
```

Modelación

Para resolver un problema siempre es necesario:

- Identificar las componentes del problema
- Expresarlas en términos de variables y restricciones
- Preocuparse de que las **operaciones** sean eficientes

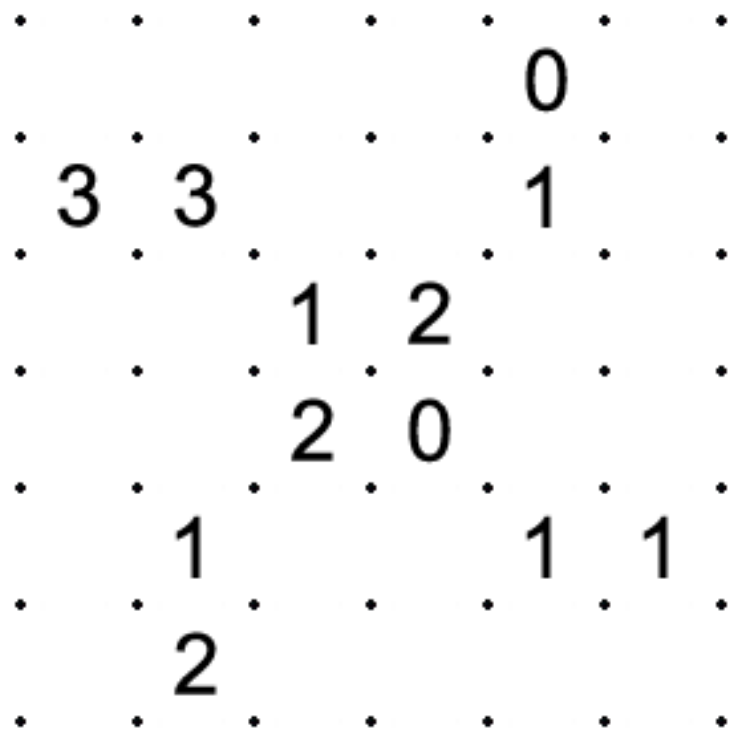
N-Queens

En un tablero de ajedrez de $n \times n$ se quieren poner n reinas…

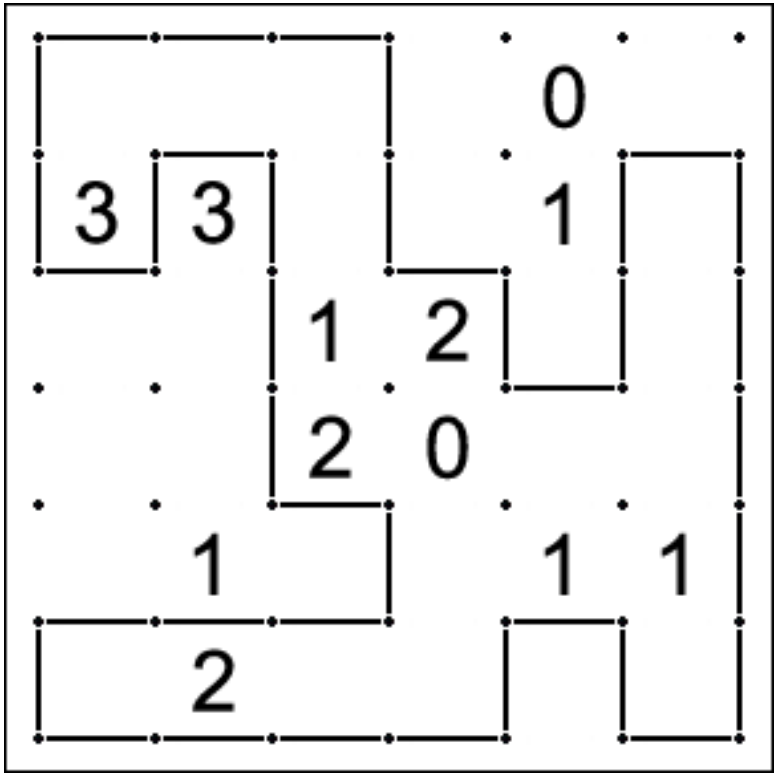
… tal que ninguna reina pueda atacar a otra reina

¿Cómo modelamos esto?

Slitherlink



Slitherlink



Descarte

La idea es **descartar** permutaciones que no llevan a una solución

Una forma de hacer esto es revisar las restricciones

¿Hay alguna otra manera?

```
is solvable(X, R):
    if is solution(X, R):
        return true
    x = choose unassigned variable(X)
    for v ∈ x.d:
        if is valid(x, v, R):
            assign(x, v)
            if is solvable(X, R):
                return true
            unassign(x, v)
    return false
```

Podas

Una **poda** es una restricción adicional

Se **deducen** de las restricciones originales

¿Cuándo nos conviene hacer una poda?

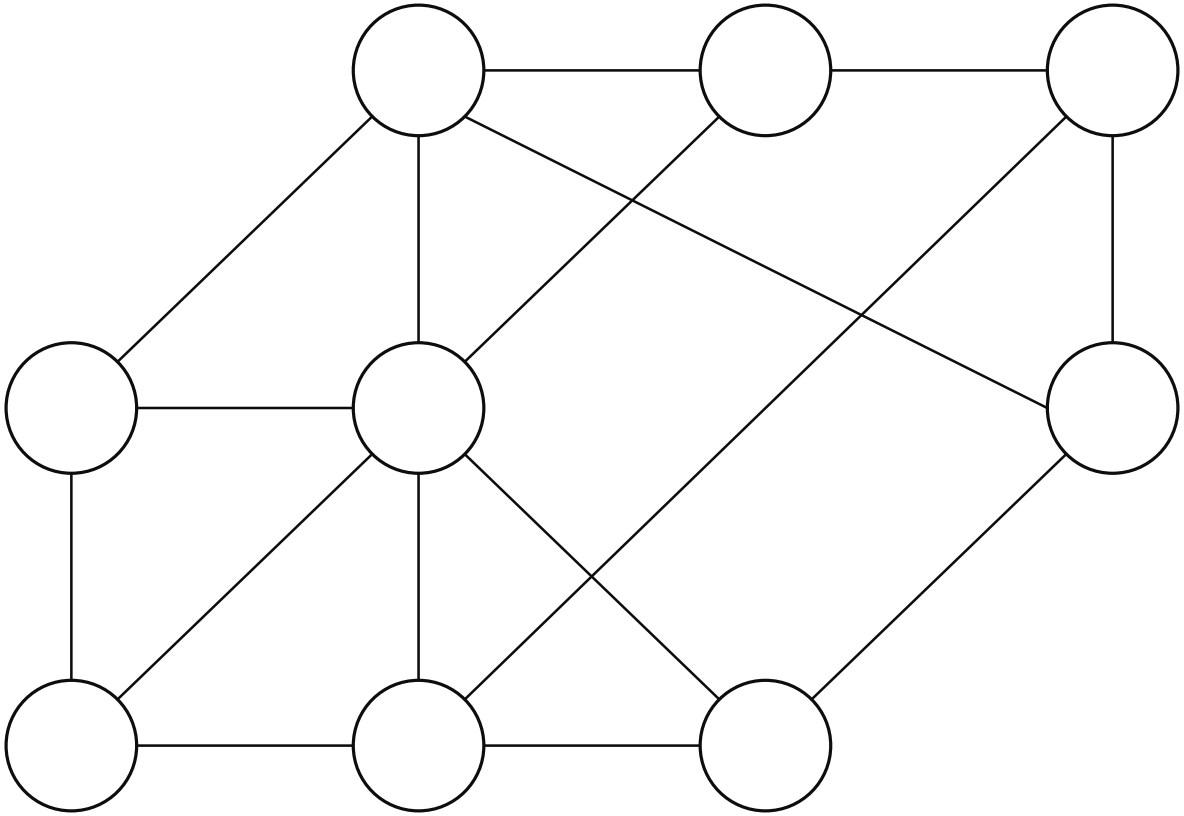
Estrategia

¿Afecta el orden en que asignamos las variables?

¿Afecta el orden en el que probamos los valores?

¿Será posible formular una estrategia para resolver el problema?

Coloración de grafos



¿Qué estrategia podemos usar para colorear este grafo?

```
is solvable(X, R):
    if is solution(X, R):
        return true
    x = choose unassigned variable(X)
    for v ∈ x.d:
        if is valid(x, v, R):
            assign(x, v)
            if is solvable(X, R):
                return true
            unassign(x, v)
    return false
```

Heurísticas

Formalmente, estas estrategias se conocen como **heurísticas**

Las heurísticas nos dicen que opciones nos convienen más

¿Cuándo nos conviene usar heurísticas?

Usos de backtracking

Sirve cuando es necesario probar combinaciones, ya sea

- Problemas de **asignación**
- Problemas de **planificación**
- Problemas de **optimización** combinatorial

Mejoras adicionales

El mundo de Backtracking tiene muchas más cosas interesantes:

- propagación
- back-jumping
- arc-consistency
- etc…

Para efectos de este curso nos quedaremos con lo visto hoy