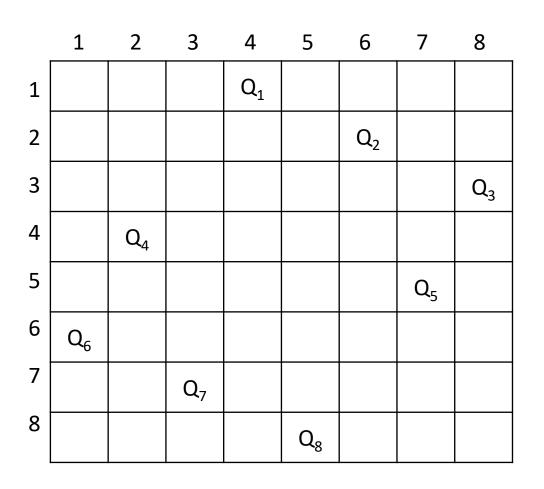
La técnica algorítmica backtracking

Dadas variables x_1, \dots, x_n con dominios finitos d_1, \dots, d_n

... y un conjunto de restricciones R

 \dots encontrar una asignación para cada x que respete R

Cómo posicionar 8 reinas en un tablero de ajedrez, de modo que no se ataquen



Numeramos las filas y columnas del tablero de 1 a 8; también numeramos las reinas 1 a 8

Cada reina debe estar en una fila diferente → suponemos que la reina *i* va en la fila *i*

La solución al problema es una 8-tupla $(x_1, ..., x_8)$ en que x_i es la columna en la que va la reina i \rightarrow en el ej. = (4, 6, 8, 2, 7, 1, 3, 5)

$$d_i = \{1, 2, 3, 4, 5, 6, 7, 8\}, 1 \le i \le 8$$

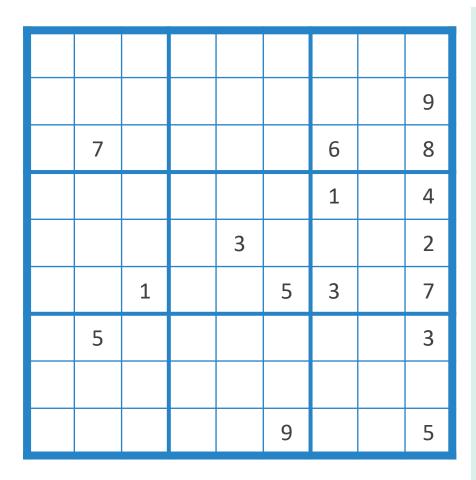
¿Cuáles son las restricciones R?

Modelación de un problema para poder resolverlo mediante backtracking

¿Cuáles son las variables?

¿Cuáles son sus dominios?

¿Cuáles son las restricciones?



¿Cuáles son las variables en el sudoku?

¿Cuáles son sus dominios?

¿Cuáles son las restricciones?

El enfoque de fuerza bruta



Generar todas las tuplas posibles y, para cada una, verificar si cumple todas las restricciones:

- en general, hay $|d_1| \times |d_2| \times ... \times |d_n|$ tuplas
- en el caso del problema de las 8 reinas, hay 88 tuplas
- en el caso del sudoku, hay? tuplas

¿Es posible hacerlo mejor?

Quizás no es necesario generar todas las tuplas...

 backtracking permite resolver el problema evaluando muchas menos tuplas

La "gracia" de backtracking

Una secuencia de asignaciones válidas para las *k* primeras variables no necesariamente llega a una solución:

 podemos usar funciones de acotamiento —versiones de R— para chequear si las asignaciones a las k primeras variables tiene una posibilidad de éxito

Podemos 'retractarnos' cuando una asignación nos lleva a una contradicción:

• así, si la última asignación a x_k nos lleva a una contradicción, nos evitamos tener que chequear $|d_{k+1}| \times ... \times |d_n|$ tuplas

Para 'retractarnos', es necesario deshacer la asignación a x_k

La posibilidad de retractarnos es lo que le da el nombre a la técnica: backtracking

El paso de deshacer la última asignación a x_k se conoce como **backtrack**

La idea es **descartar** tuplas que violan alguna restricción, *mientras* estamos formando la tupla, sin esperar necesariamente hasta que esté terminada

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

¿Es posible?

					9
7				6	8
				1	4
		3			2
	1		5	3	7
5					3
			9		5

¿Es posible?



Si el problema tiene solución, queremos una garantía.

Si no tiene solución, también queremos una garantía.

¿Cómo hacemos esto?

¿Es posible?



La idea es responder recursivamente la pregunta: "Dado un problema, ¿es posible resolverlo?".

Si asignamos una variable, ¿qué nos queda?

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
 for v \in D_x:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

Backtracking en pseudo código.

Sudoku como CSP + backtracking

https://www.geeksforgeeks.org/backtracking-algorithms/

https://www.geeksforgeeks.org/sudoku-backtracking-7/

Complejidad de fuerza bruta para *n* variables:

Complejidad de backtracking para *n* variables:

Cuál se demora menos en la práctica ('con cronómetro'):?

... y su complejidad

Complejidad de fuerza bruta para *n* variables:

$$O(9^n)$$

Complejidad de backtracking para *n* variables:

$$O(9^n)$$

Cuál se demora menos en la práctica ('con cronómetro'):

Muy probablemente backtracking

Backtracking siempre es igual o más rápido que fuerza bruta

N-Queens como CSP + Backtracking

https://www.youtube.com/watch?v=0DeznFqrgAl

(Abre Paréntesis...

La clase pasada hablamos sobre casos de más de una solución...

```
is solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
 for v \in D_x:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

¿Puedo modificar el pseudo código para encontrar todas las soluciones?

```
all-solutions(X,D,R):
if X = \emptyset, return true
x \leftarrow alguna variable de X sin asignar
for v \in D_x:
          if x = v viola R, continue
          x \leftarrow v
          if all - solutions(X, D, R):
                    X es una asignación valida
          x \leftarrow \emptyset
return false
```

all-solutions

Podemos usar variantes del tipo *all-solutions* para cuando quiero (encontrar / saber si hay / contar) más de una solución.

Cuando uso una variante del tipo *all-solutions*, ¿es lo mismo que usar **fuerza bruta**?

...Cierre Paréntesis)

Podemos usar variantes del tipo *all-solutions* para cuando quiero (encontrar / saber si hay / contar) más de una solución

Cuando uso una variante del tipo *all-solutions,* ¿es lo mismo que usar fuerza bruta? → NO; siempre es igual o más rápido en la práctica, gracias al descarte de tuplas en cada *back-track*

Mejoras a Backtracking

- Podas
- Propagación
- Heurísticas

Para cada mejora veremos:

- 1. Contexto
- 2. Definición
- 3. Ubicación en pseudo código

Descarte



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

Una forma de hacer esto es revisar las restricciones del conjunto \boldsymbol{R} de la definición del problema.

¿Hay alguna otra manera?

Mejora: Podas

Son restricciones adicionales que le ponemos al problema.

Se deducen de las restricciones originales.

Pueden ser más costosas de revisar, pero suelen valerlo en la práctica.

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
 for v \in D_x:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
for v \in D_x:
           if x = v no es válida, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

Mejora: Podas

Dominios



No todos los valores de un dominio son siempre válidos.

Depende de las restricciones que afectan a la variable.

¿Cómo va cambiando un dominio a medida que resolvemos?

					9
7				6	8
				1	4
		3			2
	1		5	3	7
5					3
			9		5

1						
						9
	7				6	8
					1	4
			3			2
		1		5	3	7
	5					3
				9		5

Múltiples asignaciones



¿Será posible hacer más de una asignación por paso?

¿En qué circunstancias tiene sentido?

Mejora: Propagación

Al asignar, es posible invalidar valores del dominio de otra variable.

Es útil propagar esta información luego de una asignación.

Si algún $|D_x| = 1$, entonces podemos asignar x y volver a propagar.

Hay que tener más cuidado al deshacer las asignaciones.

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
 for v \in D_x:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

```
is\ solvable(X, D, R):
if X = \emptyset, return true
x \leftarrow alguna variable de X
for v \in D_x:
          if x = v viola R, continue
          x \leftarrow v, propagar
          if is solvable(X - \{x\}, D, R):
                    return true
          x \leftarrow \emptyset, propagar
return false
```

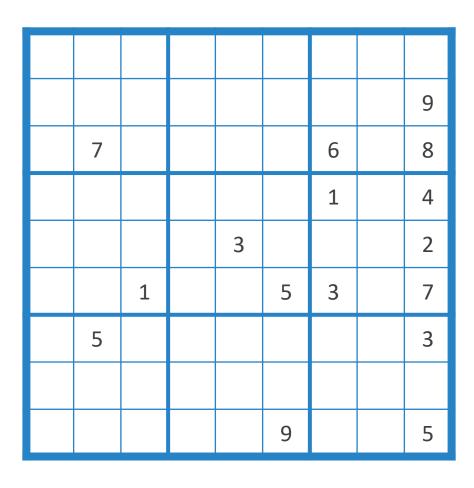
Orden de asignación



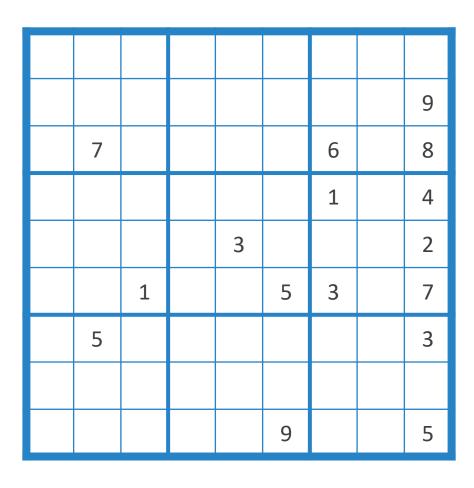
A la hora de resolver un problema de asignación,

¿Afecta el orden en que asignamos las <u>variables</u>?

¿Afecta el orden en que probamos sus posibles valores?



1						
						9
	7				6	8
					1	4
			3			2
		1		5	3	7
	5					3
				9		5



					1
					9
7				6	8
				1	4
		3			2
	1		5	3	7
5					3
			9		5

Sudoku

					1
					9
7				6	8
				1	4
		3			2
	1		5	3	7
5					3
					6
			9		5

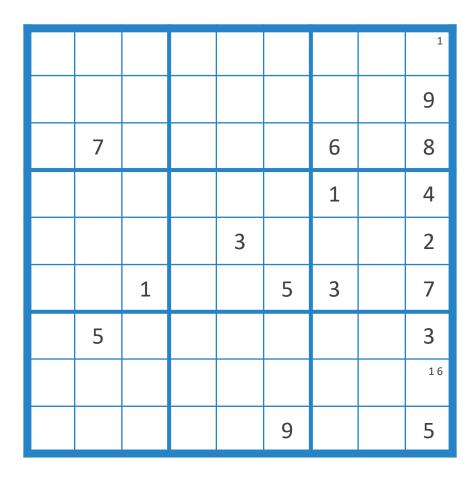
Mejora: Heurísticas

Cuando un problema es muy difícil, usamos heurísticas.

Las heurísticas tratan de aproximar la realidad.

Son una idea de qué tan buena es una opción.

Sudoku



Sudoku

4			2				
8						1	
7		4					
325							
3 2				5			
3 5	8						2
1					3		
9		5					
6							

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow alguna variable de X
 for v \in D_x:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset
 return false
```

```
is\ solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow \text{la mejor variable de } X
 for v \in D_x, de mejor a peor:
           if x = v viola R, continue
           x \leftarrow v
           if is solvable(X - \{x\}, D, R):
                      return true
           x \leftarrow \emptyset
 return false
```

```
is solvable(X, D, R):
 if X = \emptyset, return true
 x \leftarrow \text{la mejor variable de } X
 for v \in D_x, de mejor a peor:
           if x = v no es válida, continue
           x \leftarrow v, propagar
           if is solvable(X - \{x\}, D, R):
                     return true
           x \leftarrow \emptyset, propagar
 return false
```

Mejoras: Heurísticas, Podas y Propagación.

¿Cuándo uso Backtracking?

Sirve siempre cuando es necesario probar todo...

...y siempre es igual o más rápido que fuerza bruta.

Estrategias algorítmicas

Vistas:

- Dividir para conquistar
- Backtracking

Por ver:

- Algoritmos codiciosos
- Programación dinámica

Bonus: Complejidad de CSP



CSP es una familia entera de problemas con las mismas características.

¿Qué tan rápido podrán resolverse los CSP?

SAT

Sea φ una fórmula en lógica proposicional.

 φ se dice **satisfacible** si existe forma de hacerla verdadera.

Averiguar si φ es satisfacible es NP-Completo.

SAT como CSP



Queremos encontrar una asignación a cada variable de φ :

 $lue{}$ La fórmula ϕ debe hacerse verdadera

Conclusión:

CSP es al menos tan difícil como SAT (NP-Completo).