Backtracking II

Clase 14

IIC 2133 - Sección 1

Prof. Sebastián Bugedo

Sumario

Introducción

Extensiones del Backtracking

Backtracking: idea de pseudocódigo

input : Conjunto de variables sin asignar X, dominios D, restricciones R

`isSolvable`(X, D, R)**:**

```
1  if X = ∅ : return true
2  x ← alguna variable de X
3  for v ∈ D_x :
4      if x = v no rompe R :
5          x ← v
6          if isSolvable(X − {x}, D, R) :
7              return true
8          x ← ∅
9  return false
```

Esto es solo una orientación: las variables, argumentos y estructura dependerá del problema particular

Problema de las 8 reinas

A continuación, un algoritmo para determinar si una asignación parcial de las 8 reinas puede dar lugar a una solución válida

input : Arreglo $T[0 \ldots 7]$,
índice $0 \leq i \leq 8$
output: **true** ssi hay solución

```
Queens(T, i):
1    if i = 8 : return true
2    for v = 0 ... 7 :
3        if Check(T, i, v) :
4            T[i] ← v
5            if Queens(T, i + 1) :
6                return true
7    return false
```

input : Arreglo $T[0 \ldots 7]$,
índices $0 \leq i, j \leq 7$
output: **false** ssi es ilegal

```
Check(T, i, v):
1    for j = 0 ... i − 1 :
2        if v = T[i] :
3            return false
4        if |(v − T[j])/(i − j)| = 1 :
5            return false
6    return true
```

¿Cómo podemos modificar el algoritmo para obtener una solución?

Complejidad

El análisis de complejidad del *backtracking* involucra el conteo de tuplas posibles

- En un conjunto de n variables $X = \{x_1, \ldots, x_n\}$
- con valores posibles en dominios $D = \{D_1, \ldots, D_n\}$
- tenemos $|D_1| \times |D_2| \times \cdots \times |D_n|$ tuplas posibles

Luego, en el caso particular de que $|D_i| = K$ para todo i,

- revisar todas las tuplas es $\mathcal{O}(K^n)$

Complejidad

La complejidad de las posibles soluciones para CSP cumplen,

- la estrategia de fuerza bruta revisa **todas las tuplas** $\mathcal{O}(K^n)$
- el backtracking puede revisar menos tuplas, pero sigue siendo proporcional $\mathcal{O}(K^n)$

Es decir, asintóticamente estas estrategias tienen la misma complejidad

¿Cuál es más rápido en la práctica?

No olvidar: *Backtracking* es igual o más rápido que la fuerza bruta

Otra interpretación del backtracking

Podemos pensar en la estrategia de backtracking como **búsqueda en un grafo implícito**

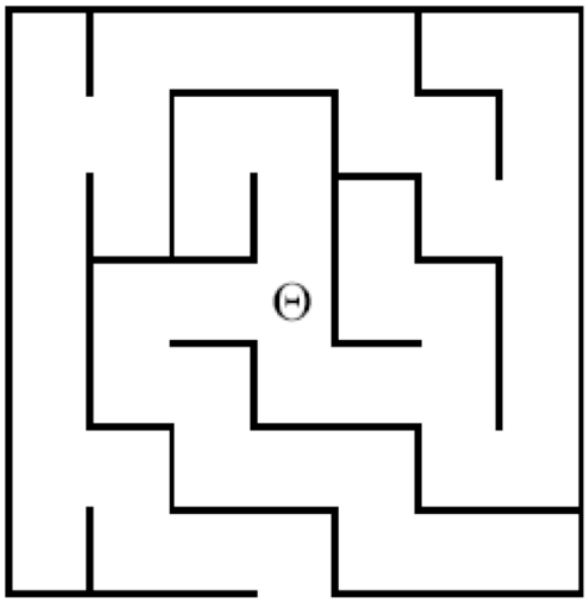
Los CSP generan muchas tuplas posibles como asignaciones para las variables de X

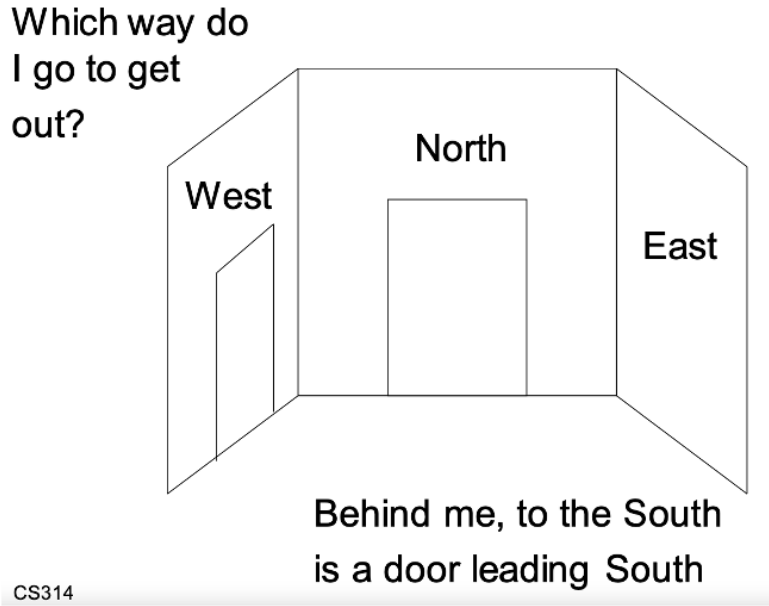
- Cada posible asignación genera un camino
- Las nuevas asignaciones abren nuevos caminos
- A la colección de todas estas alternativas le llamamos **grafo implícito**

El ejemplo por excelencia para visualizar el grafo implícito
es el **problema de recorrer un laberinto**

Recorrido del laberinto

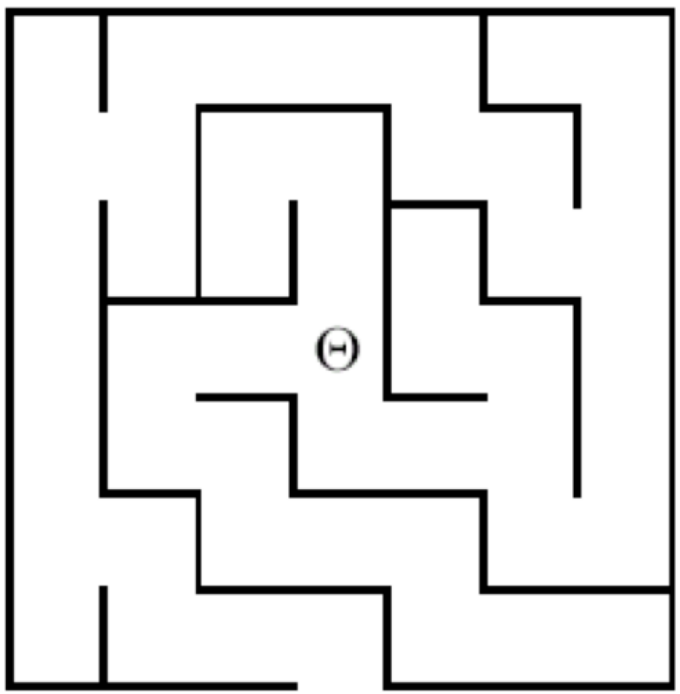
Supongamos que nos interesa salir de un laberinto dado que estamos en Θ





Podemos resolver este problema con *backtracking*

Recorrido del laberinto



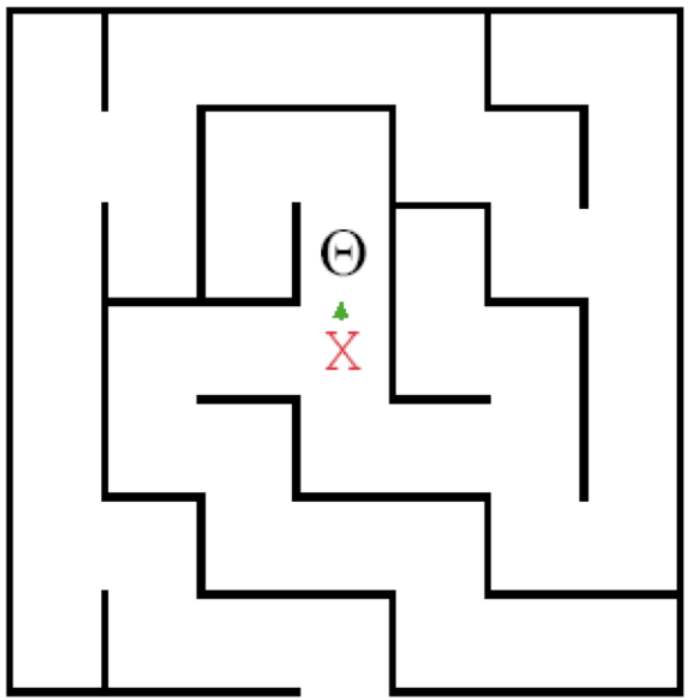
Planteamos el problema como un CSP

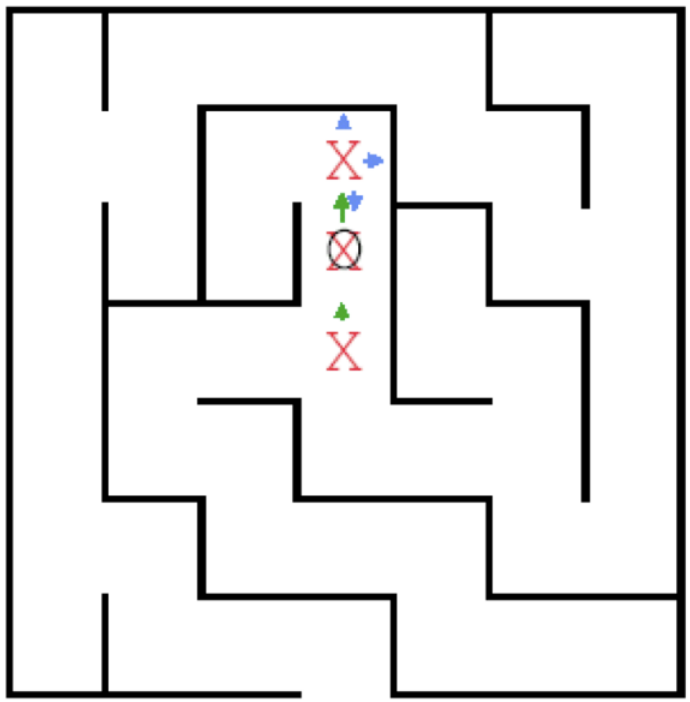
- Variables?
- Dominios?
- Restricciones?
- Qué define el *éxito*?

Caracterizamos por Θ la posición actual

Recorrido del laberinto

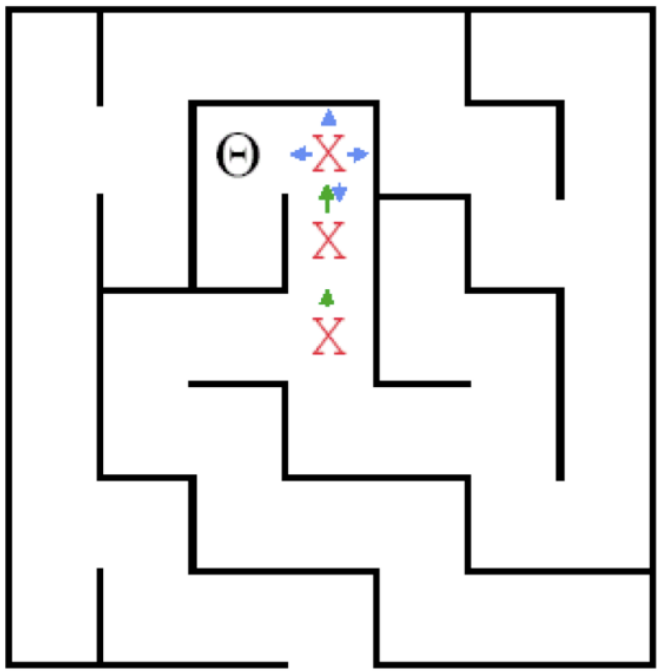
En cada nueva posición Θ solo podemos elegir dar un paso en las direcciones libres y distintas de aquella de la cual venimos

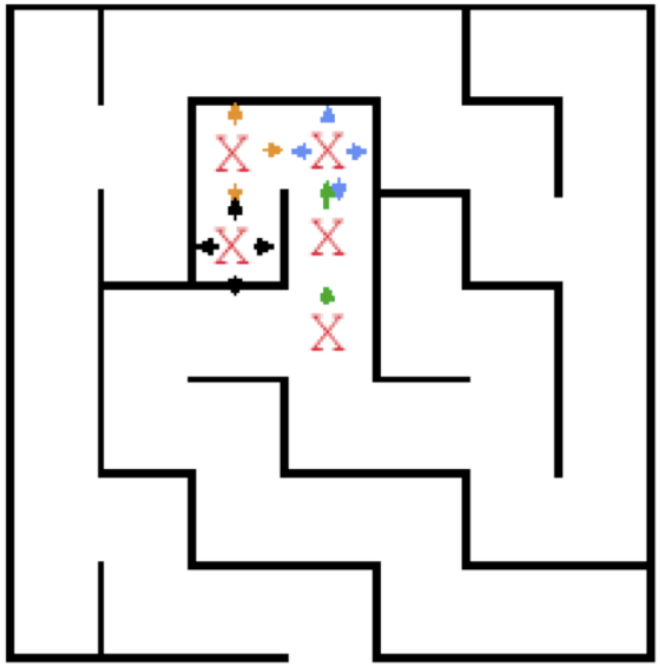




Recorrido del laberinto

Debemos hacer backtrack cuando llegamos a un camino sin salida: solo muros y celdas ya visitadas

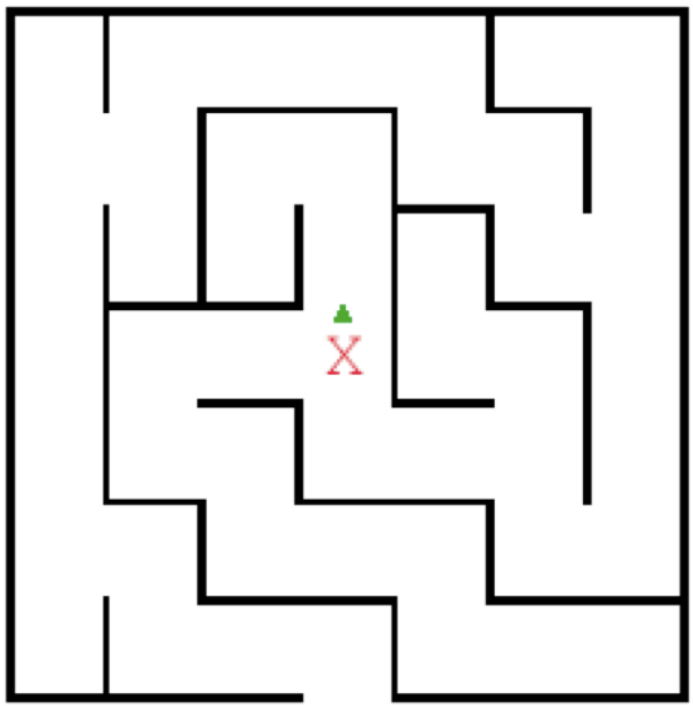


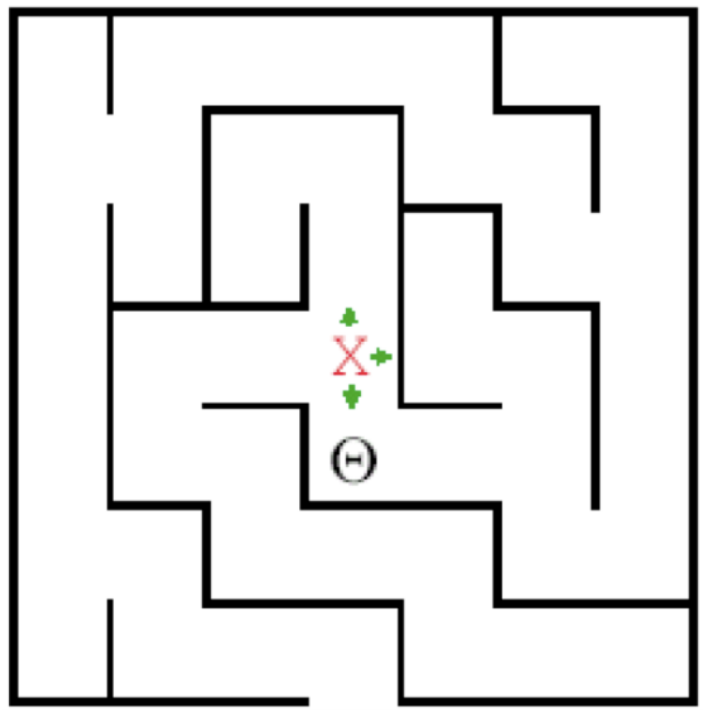


No hay más opciones: ¿hasta dónde nos *arrepentimos* con el backtrack?

Recorrido del laberinto

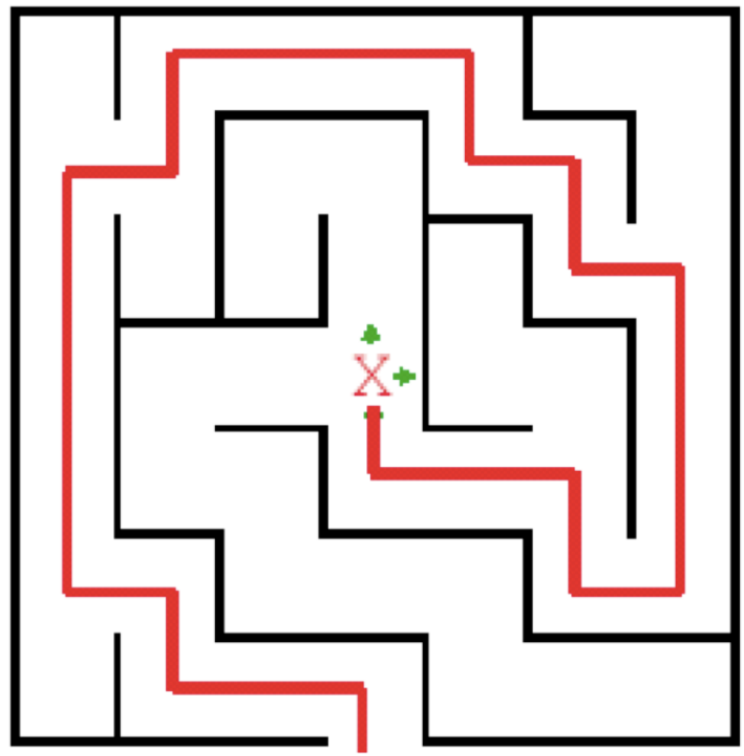
Sabemos que ir al norte no funcionó. Probamos otra opción yendo al sur.





Recorrido del laberinto

En este caso, logramos llegar a una solución que encuentra la salida



Recorrido del laberinto

Le agregamos etiquetas a las posiciones, de modo que sabemos cuáles hemos visitado (visited). Todas comienzan como nonvisited y la salida se marca como exit

input : Conjunto de variables sin asignar X, posición x, dominios D, restricciones R

`isSolvable`(X, x, D, R)**:**

```
1   if x = exit : return true
2   if visited : return false
3   x ← visited
4   for v ∈ {N, E, S, W} :
5       if x + v ≠ wall :
6           x ← x + v
7           if isSolvable(X, x, D, R) :
8               return true
9           x ← nonvisited
10  return false
```

Otros problemas habituales

Hay varios problemas clásicos que se resuelven mediante backtracking

- Recorrido del caballo de ajedrez (*Knight's tour problem*)
- Problema de la mochila (capacidad versus número de items)
- Balance de carga
- Coloreo de mapas (Sudoku es un caso particular)

En general, puzzles NP-completos podemos atacarlos
con alguna idea de backtracking

Objetivos de la clase

- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Reconocer necesidad de modificaciones al esquema de backtracking
- ☐ Comprender el concepto de poda
- ☐ Comprender el concepto de propagación
- ☐ Comprender el concepto de heurística


CONCIERTOS
ORQUESTA CIUDADANA DE SANTIAGO
CORO ALUMNI UC
15 de octubre - 12:30
Iglesia de los Sacramentinos - Santa Isabel
entre San Diego y Arturo Prat
Santiago Centro, metro Parque Almagro
16 de octubre - 17:30
Centro Cultural de España - Providencia 927
Providencia, metro Salvador
ENTRADA LIBERADA
Vivamos bien
STGO
ILUSTRE MUNICIPALIDAD
Fundación
Musical
Lagar
EMBAJADA
DE ESPAÑA
EN CHILE
aecid
Cooperación
Española
CULTURA / SANTIAGO DE CHILE

Conversemos de la tarea

Sumario

Introducción

Extensiones del Backtracking

Primera extensión de Backtracking

Consideremos ahora el problema de determinar **todas** las soluciones a un CSP

- Nos interesan las soluciones explícitamente
- O solo queremos contarlas

En ambos casos, necesitamos que el algoritmo **no se detenga** al encontrar la primera solución

Encontrar todas las soluciones

input : Conjunto de variables sin asignar X, dominios D, restricciones R

`isSolvable`(X, D, R)**:**

```
1  if X = ∅ : return true
2  x ← alguna variable de X
3  for v ∈ D_x :
4      if x = v no rompe R :
5          x ← v
6          if isSolvable(X − {x}, D, R) :
7              return true
8          x ← ∅
9  return false
```

¿Cómo modificar el algoritmo genérico para encontrar **todas** las soluciones?

Encontrar todas las soluciones

input : Conjunto de variables sin asignar X, dominios D, restricciones R

`isSolvableAll`(X, D, R)**:**

```
1  if X = ∅ : return true
2  x ← alguna variable de X
3  for v ∈ D_x :
4      if x = v no rompe R :
5          x ← v
6          if isSolvableAll(X − {x}, D, R) :
7              Se marca x ← v como solución
8          x ← ∅
9  return false
```

Incluso en este escenario, Backtracking es mejor que fuerza bruta

Mejoras de desempeño de Backtracking

Ahora, nos interesa poder **informar mejor** al Backtracking

- Gracias a las características del problema, sabemos que hay caminos que ya no es necesario revisar
- El dominio para x_i quizás no es D_i completo
- Puede haber *mejores* elementos de D_i para elegir primero

Estos casos nos permiten proponer las siguientes mejoras que detallaremos

- Podas
- Propagación
- Heurísticas

Podas

Backtracking es capaz de determinar si una asignación puede terminar en solución

- Las soluciones inviables se **descartan** según las restricciones R del CSP
- Requiere llamados recursivos
- Posiblemente, **muchos** llamados

¿Podemos hacerlo mejor?

Agregaremos nuevas restricciones que se deducen de las iniciales

Podas

Llamaremos **podas** a estas nuevas restricciones y se revisan junto a las originales

```
  isSolvable(X, D, R):
1     if X = ∅ : return true
2     x ← alguna variable de X
3     for v ∈ D_x :
4         if x = v no rompe R :
5             x ← v
6             if isSolvable(X − {x}, D, R) :
7                 return true
8             x ← ∅
9     return false
```

Podas

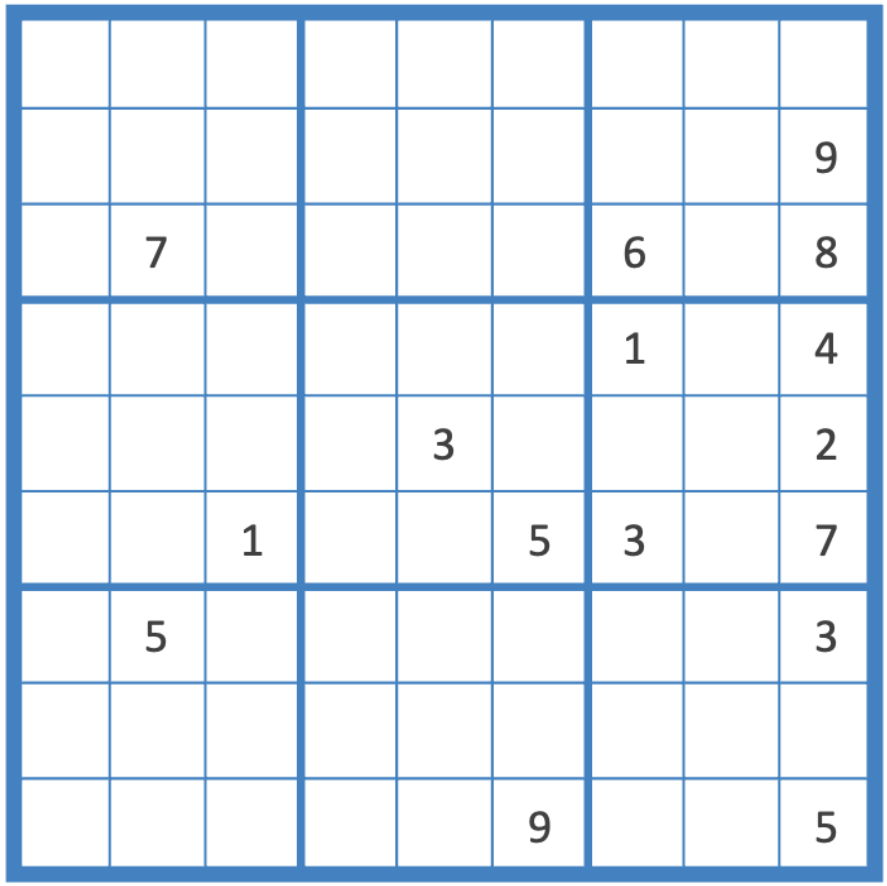
Llamaremos **podas** a estas nuevas restricciones y se revisan junto a las originales

```
isSolvable(X, D, R):
    if X = ∅ : return true
    x ← alguna variable de X
    for v ∈ D_x :
        if x = v no rompe R :
            x ← v
            if isSolvable(X − {x}, D, R) :
                return true
            x ← ∅
    return false
```

Pueden ser más costosas de checkear,
pero vale la pena en la práctica

Dominios

Consideremos el siguiente tablero de Sudoku parcialmente completado



Dominios

Si asignamos el valor 1 a la posición $(0, 0)$, ¿cambió el dominio válido para alguna variable?

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

Propagación

Backtracking chequea todos los valores posibles en el dominio D_i de la variable x_i

- Existen restricciones que invalidan ciertos valores de D_i
- Backtracking clásico los revisa igual
- Esas soluciones parciales nunca serán válidas

¿Podemos hacerlo mejor?

Cambiaremos los dominios de las demás variables luego de una asignación

Propagación

Llamaremos **propagación** a la acción de modificar dominios luego de una asignación

```
isSolvable(X, D, R):
    if X = ∅ : return true
    x ← alguna variable de X
    for v ∈ D_x :
        if x = v no rompe R :
            x ← v
            if isSolvable(X − {x}, D, R) :
                return true
            x ← ∅
    return false
```

Propagación

Llamaremos **propagación** a la acción de modificar dominios luego de una asignación

```
isSolvable(X, D, R):
1   if X = ∅ : return true
2   x ← alguna variable de X
3   for v ∈ D_x :
4       if x = v no rompe R :
5           x ← v, propagar
6           if isSolvable(X − {x}, D, R) :
7               return true
8           x ← ∅, propagar
9   return false
```

Ojo al deshacer asignaciones,
pues hay que reestablecer dominios propagados

Orden

Consideremos el siguiente tablero de Sudoku parcialmente completado: ¿por qué celda partimos llenando?

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

Nos interesa minimizar la posibilidad de fracasar

Orden

¿Será mejor la $(0, 8)$?

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

Orden

¿Ahora cuál sería razonable escoger?

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

Orden

¿Ahora cuál sería razonable escoger?

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

Heurísticas

Backtracking chequea los valores válidos en el dominio D_i de la variable x_i en un orden arbitrario

- No solo puede afectar el orden en que se asignan valores
- También puede afectar el orden en que se itera sobre las variables disponibles

De hecho, si dispusiéramos de un **oráculo** que nos dice el mejor orden de asignación, el problema se vuelve **lineal**!

Guiaremos la búsqueda según algunos criterios (falibles)

Heurísticas

Llamaremos **heurísticas** a las estrategias para catalogar variables y valores según *qué tan buenos son*

```
  isSolvable(X, D, R):
1     if X = ∅ : return true
2     x ← alguna variable de X
3     for v ∈ D_x :
4         if x = v no rompe R :
5             x ← v
6             if isSolvable(X − {x}, D, R) :
7                 return true
8             x ← ∅
9     return false
```

Heurísticas

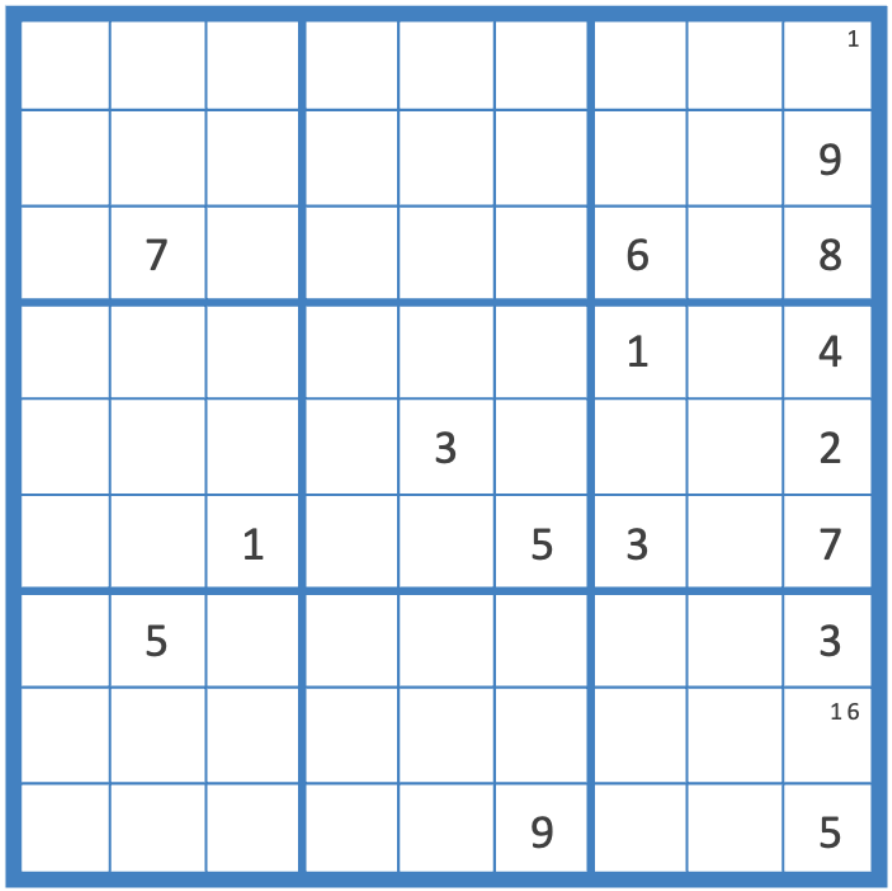
Llamaremos **heurísticas** a las estrategias para catalogar variables y valores según *qué tan buenos son*

```
isSolvable(X, D, R):
1     if X = ∅ : return true
2     x ← la mejor variable de X
3     for v ∈ D_x de mejor a peor :
4         if x = v no rompe R :
5             x ← v
6             if isSolvable(X − {x}, D, R) :
7                 return true
8             x ← ∅
9     return false
```

Las heurísticas tratan de aproximar la realidad, pueden equivocarse

Heurísticas

Posible heurística: partir por la variable con dominio más pequeño



Heurísticas

Posible heurística: partir por el valor con menos apariciones

4				2				
8							1	
7			4					
325								
32					5			
35		8						2
1						3		
9			5					
6								

Backtracking mejorado

Podemos incorporar estas mejoras según convenga en un problema particular

```
isSolvable(X, D, R):
1     if X = ∅ : return true
2     x ← la mejor variable de X
3     for v ∈ D_x de mejor a peor :
4         if x = v no rompe R :
5             x ← v, propagar
6             if isSolvable(X − {x}, D, R) :
7                 return true
8             x ← ∅, propagar
9     return false
```