

# Backtracking II

Clase 16

IIC 2133 - Sección 1

Prof. Diego Arroyuelo

# Sumario

**Introducción**

Un ejemplo

Extensiones de Backtracking

Cierre

# Problema de las 8 reinas

# 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

# 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  $T[0 \dots 7]$ ,

índice  $0 \leq i \leq 8$

**output:** **true** ssi hay solución

$\text{Queens}(T, i)$ :

```
1  if  $i = 8$  : return true
2  for  $v = 0 \dots 7$  :
3      if  $\text{Check}(T, i, v)$  :
4           $T[i] \leftarrow v$ 
5          if  $\text{Queens}(T, i + 1)$  :
6              return true
7  return false
```

# 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  $T[0 \dots 7]$ ,

índice  $0 \leq i \leq 8$

**output:** true ssi hay solución

Queens( $T, i$ ):

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

**input** : Arreglo  $T[0 \dots 7]$ ,

índices  $0 \leq i, j \leq 7$

**output:** false ssi es ilegal

Check( $T, i, v$ ):

```
1  for  $j = 0 \dots i - 1$  :
2      if  $v = T[j]$  :
3          return false
4      if  $|(v - T[j]) / (i - j)| = 1$  :
5          return false
6  return true
```

# 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  $T[0 \dots 7]$ ,

índice  $0 \leq i \leq 8$

**output:** true ssi hay solución

Queens( $T, i$ ):

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

**input** : Arreglo  $T[0 \dots 7]$ ,

índices  $0 \leq i, j \leq 7$

**output:** false ssi es ilegal

Check( $T, i, v$ ):

```
1  for  $j = 0 \dots i - 1$  :
2      if  $v = T[j]$  :
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



# Complejidad

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

- En un conjunto de  $n$  variables  $X = \{x_1, \dots, x_n\}$

# Complejidad

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

- En un conjunto de  $n$  variables  $X = \{x_1, \dots, x_n\}$
- con valores posibles en dominios  $D = \{D_1, \dots, D_n\}$

# Complejidad

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

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

# Complejidad

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

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

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

# Complejidad

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

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

# Complejidad

La complejidad de las posibles soluciones para CSP cumplen,

- la estrategia de fuerza bruta revisa **todas las tuplas**

# Complejidad

La complejidad de las posibles soluciones para CSP cumplen,

- la estrategia de fuerza bruta revisa **todas las tuplas**

$$\mathcal{O}(K^n)$$



# 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

# 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  $\mathcal{O}(K^n)$

# 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

# 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?

# 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

# Otra interpretación del backtracking

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

# 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  $X$



# 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

# 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

# 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**

# Otra interpretación del backtracking

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

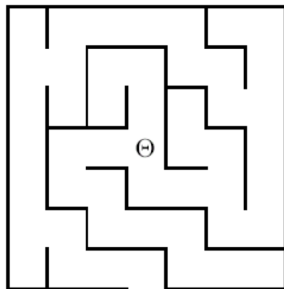
Los CSP generan muchas tuplas posibles como asignaciones para las variables de  $X$

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

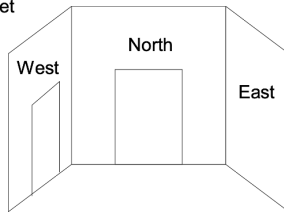
El ejemplo por excelencia para visualizar el grafo implícito es el **problema de recorrer un laberinto**

# Recorrido del laberinto

Supongamos que nos interesa salir de un laberinto dado que estamos en  $\Theta$



Which way do  
I go to get  
out?

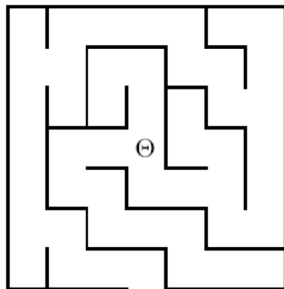


Behind me, to the South  
is a door leading South

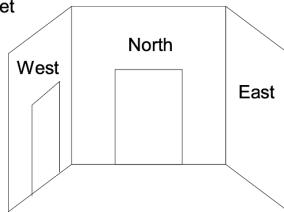
CS314

# Recorrido del laberinto

Supongamos que nos interesa salir de un laberinto dado que estamos en  $\Theta$



Which way do  
I go to get  
out?

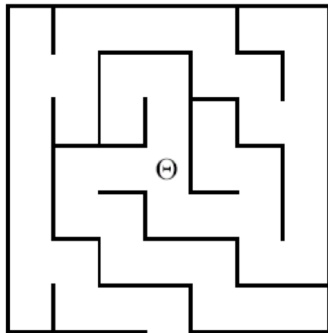


Behind me, to the South  
is a door leading South

CS314

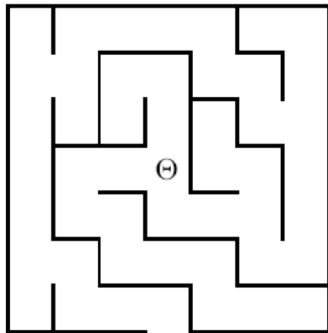
Podemos resolver este problema con *backtracking*

# Recorrido del laberinto



Planteamos el problema como un CSP

# Recorrido del laberinto

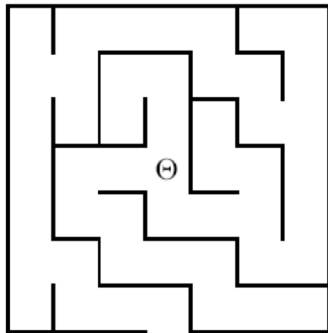


Planteamos el problema como un CSP

- Variables?



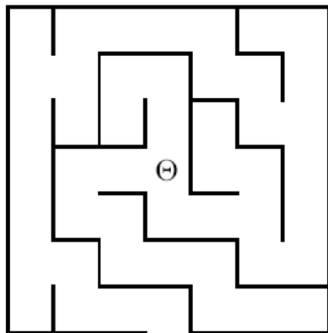
# Recorrido del laberinto



Planteamos el problema como un CSP

- Variables?
- Dominios?

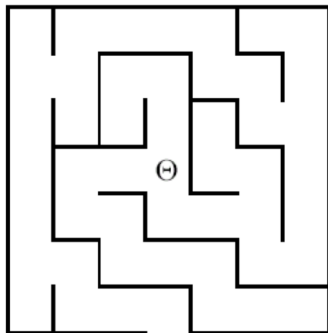
# Recorrido del laberinto



Planteamos el problema como un CSP

- Variables?
- Dominios?
- Restricciones?

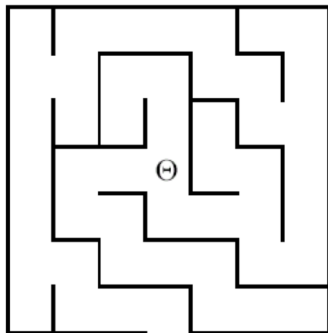
# Recorrido del laberinto



Planteamos el problema como un CSP

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

# Recorrido del laberinto



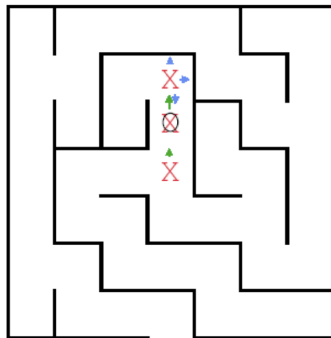
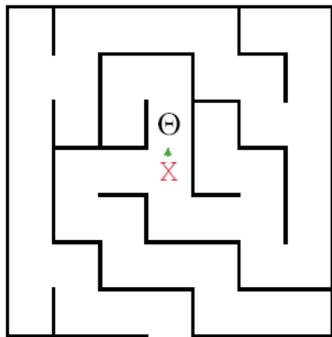
Planteamos el problema como un CSP

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

Caracterizamos por  $\Theta$  la posición actual

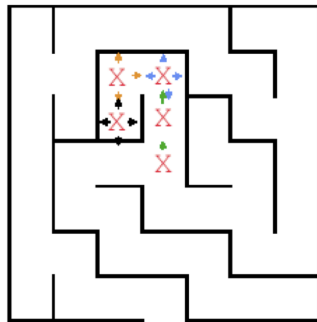
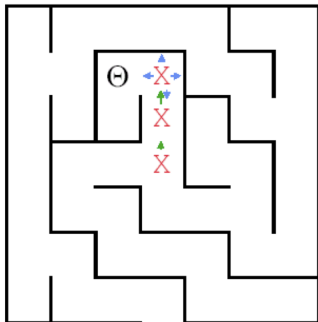
# Recorrido del laberinto

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



# Recorrido del laberinto

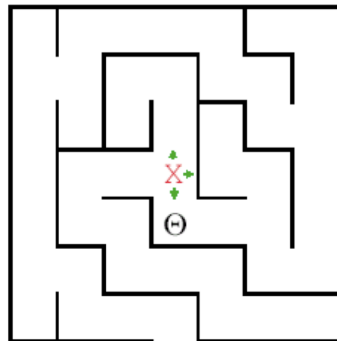
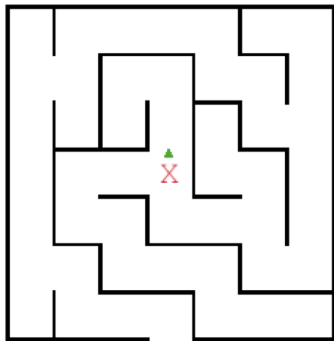
Debemos hacer backtrack cuando llegamos a un camino sin salida: solo muros y celdas ya visitadas



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

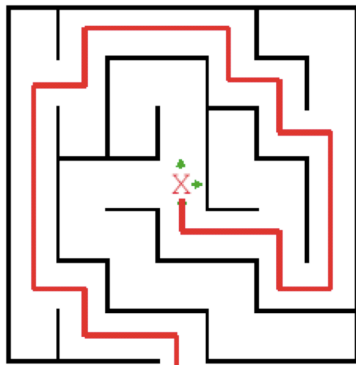
# Recorrido del laberinto

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  $x = \text{exit}$  : return true
2  if  $x = \text{visited}$  : return false
3   $x \leftarrow \text{visited}$ 
4  for  $v \in \{N, E, S, W\}$  :
5      if  $x + v \neq \text{wall}$  :
6           $x \leftarrow x + v$ 
7          if isSolvable( $X, x, D, R$ ) :
8              return true
9   $x \leftarrow \text{nonvisited}$ 
10 return false
```

## Otros problemas habituales

## Otros problemas habituales

Hay varios problemas clásicos que se resuelven mediante backtracking

# Otros problemas habituales

Hay varios problemas clásicos que se resuelven mediante backtracking

- Recorrido del caballo de ajedrez (*Knight's tour problem*)

# 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)

# 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

# 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)

# 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

# Objetivos de la clase

- Identificar pseudocódigo base para backtracking y sus partes

# Objetivos de la clase

- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas

# Objetivos de la clase

- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas
- ☐ Identificar mejoras de desempeño para backtracking

# Sumario

Introducción

**Un ejemplo**

Extensiones de Backtracking

Cierre

## Ejemplo: Backtracking

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Para asegurar la conectividad del transporte en el extremo sur del país existen tramos en los cuales se utilizan barcasas para llevar vehículos (autos particulares y camiones) entre dos puntos que no tienen conectividad por tierra. La capacidad de la barcaza se define en función de los metros lineales de vehículos que puede acomodar (4 filas de vehículos de máximo 15 metros cada fila son 60 metros lineales de capacidad máxima) y el peso máximo total que puede transportar (por ejemplo 240.000 kilos de carga). Así una barcaza B se define como

$(B.n\_filas, B.m\_por\_fila, B.max\_carga)$ .

Los vehículos V que están a la espera de transporte están en una fila y tienen determinado su largo y peso ( $V.largo, V.peso$ ) expresados en metros y kilogramos.

## Ejemplo: Backtracking

### Ejercicio (I2 P4 - 2022-2)

- (a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.



## Ejemplo: Backtracking

### Ejercicio (I2 P4 - 2022-2)

- (a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

## Ejemplo: Backtracking

### Ejercicio (I2 P4 - 2022-2)

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

- Variables  $X = \{x_1, \dots, x_n\}$ , una para cada auto indicando si se sube o no a la barcaza

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

- Variables  $X = \{x_1, \dots, x_n\}$ , una para cada auto indicando si se sube o no a la barcaza
- Dominios idénticos para cada variable:  $\{0, 1\}$ , donde 1 indica que sí se sube

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

- Variables  $X = \{x_1, \dots, x_n\}$ , una para cada auto indicando si se sube o no a la barcaza
- Dominios idénticos para cada variable:  $\{0, 1\}$ , donde 1 indica que sí se sube
- Restricciones

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

- Variables  $X = \{x_1, \dots, x_n\}$ , una para cada auto indicando si se sube o no a la barcaza
- Dominios idénticos para cada variable:  $\{0, 1\}$ , donde 1 indica que sí se sube
- Restricciones
  - Peso máximo:  $W = B.\text{max\_carga}$  tal que

$$\sum_{i=1}^n x_i w_i \leq W$$

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

(a) [1 pto.] Identifique las Variables, Dominios y Restricciones del problema.

Denotaremos por  $w_i$  y  $\ell_i$  el peso y largo del auto  $i$ -ésimo.

- Variables  $X = \{x_1, \dots, x_n\}$ , una para cada auto indicando si se sube o no a la barcaza
- Dominios idénticos para cada variable:  $\{0, 1\}$ , donde 1 indica que sí se sube
- Restricciones

- Peso máximo:  $W = \text{B.max\_carga}$  tal que

$$\sum_{i=1}^n x_i w_i \leq W$$

- Largo máximo:  $L = (\text{B.n\_filas}) \cdot (\text{B.m\_por\_fila})$  tal que

$$\sum_{i=1}^n x_i \ell_i \leq L$$

## Ejemplo: Backtracking

### Ejercicio (I2 P4 - 2022-2)

- (b) [3 ptos.] Diseñe un algoritmo para definir qué vehículos de la fila transportar de modo de maximizar la cantidad de vehículos sin superar la capacidad de la barcaza (en metros lineales totales y la carga máxima de la misma). **No considere** la capacidad de cada fila de la barcaza, sino la **capacidad total**.

# Ejemplo: Backtracking

Ejercicio (I2 P4 - 2022-2)

Supondremos que



# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Supondremos que

- $X[0 \dots n - 1]$  es el arreglo para guardar los valores binarios

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Supondremos que

- $X[0 \dots n-1]$  es el arreglo para guardar los valores binarios
- $Y[0 \dots n-1]$  es el arreglo para guardar la asignación óptimo, inicializado con ceros

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Supondremos que

- $X[0 \dots n-1]$  es el arreglo para guardar los valores binarios
- $Y[0 \dots n-1]$  es el arreglo para guardar la asignación óptimo, inicializado con ceros
- $\#(X)$  entrega el número de autos asignados en  $X$

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Supondremos que

- $X[0 \dots n-1]$  es el arreglo para guardar los valores binarios
- $Y[0 \dots n-1]$  es el arreglo para guardar la asignación óptimo, inicializado con ceros
- $\#(X)$  entrega el número de autos asignados en  $X$
- $\ell(i)$  entrega el largo del auto  $i$

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

Supondremos que

- $X[0 \dots n-1]$  es el arreglo para guardar los valores binarios
- $Y[0 \dots n-1]$  es el arreglo para guardar la asignación óptimo, inicializado con ceros
- $\#(X)$  entrega el número de autos asignados en  $X$
- $\ell(i)$  entrega el largo del auto  $i$
- $w(i)$  entrega el peso del auto  $i$

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

**input** :  $X[0 \dots n-1]$  arreglo,  $L$  largo permitido,  
 $W$  peso permitido,  $i$  índice a asignar en  $X$

Backtrack ( $X, L, W, i$ ):

**if**  $i = n$  :

**if**  $\#(X) > \#(Y)$  :

$Y \leftarrow$  copia de  $X$

**else:**

**for**  $j \in \{0, 1\}$  :

**if** *asignar*  $X[i] \leftarrow j$  *no supera restricciones* :

$X[i] \leftarrow j$

                Backtrack( $X, L - j \cdot \ell(i), W - j \cdot w(i), i + 1$ )

# Ejemplo: Backtracking

## Ejercicio (I2 P4 - 2022-2)

**input** :  $X[0 \dots n-1]$  arreglo,  $L$  largo permitido,  
 $W$  peso permitido,  $i$  índice a asignar en  $X$

Backtrack ( $X, L, W, i$ ):

**if**  $i = n$  :

**if**  $\#(X) > \#(Y)$  :

$Y \leftarrow$  copia de  $X$

**else**:

**for**  $j \in \{0, 1\}$  :

**if** *asignar*  $X[i] \leftarrow j$  *no supera restricciones* :

$X[i] \leftarrow j$

                Backtrack( $X, L - j \cdot \ell(i), W - j \cdot w(i), i + 1$ )

El algoritmo se llama con Backtrack( $X, L, W, 0$ ) y una vez que termina,  $Y$  contiene la asignación óptima.

# Sumario

Introducción

Un ejemplo

**Extensiones de Backtracking**

Cierre



# Primera extensión de Backtracking

# Primera extensión de Backtracking

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

# Primera extensión de Backtracking

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

- Nos interesan las soluciones explícitamente

# 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

# 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  $X = \emptyset$  : return true
2   $x \leftarrow$  alguna variable de  $X$ 
3  for  $v \in D_x$  :
4      if  $x = v$  no rompe  $R$  :
5           $x \leftarrow v$ 
6          if isSolvable( $X - \{x\}, D, R$ ) :
7              return true
8           $x \leftarrow \emptyset$ 
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  $X = \emptyset$  : return true
2   $x \leftarrow$  alguna variable de  $X$ 
3  for  $v \in D_x$  :
4      if  $x = v$  no rompe  $R$  :
5           $x \leftarrow v$ 
6          if isSolvableAll( $X - \{x\}, D, R$ ) :
7              Se marca  $x \leftarrow v$  como solución
8           $x \leftarrow \emptyset$ 
9  return false
```

Incluso en este escenario, Backtracking es mejor que fuerza bruta

# Mejoras de desempeño de Backtracking



# Mejoras de desempeño de Backtracking

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

# 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

# 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

# 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

# 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

# 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

# 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

# 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

# Podas

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

- Las soluciones inviables se **descartan** según las restricciones  $R$  del CSP

# 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

# 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

# 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?

# 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  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$   
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8        $x \leftarrow \emptyset$   
9   return false
```

# Podas

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

```
isSolvable( $X, D, R$ ):  
1   if  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$   
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8        $x \leftarrow \emptyset$   
9   return false
```



# Podas

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

```
isSolvable( $X, D, R$ ):  
1   if  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$   
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8        $x \leftarrow \emptyset$   
9   return false
```

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

# Dominios

Consideremos el siguiente tablero de Sudoku parcialmente completado

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

# 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

# Propagación

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

# Propagación

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

- Existen restricciones que invalidan ciertos valores de  $D_i$

# 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

# 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



# 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?

# 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( $X, D, R$ ):  
1   if  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$   
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8        $x \leftarrow \emptyset$   
9   return false
```

# Propagación

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

```
isSolvable( $X, D, R$ ):  
1   if  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$ , propagar  
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8            $x \leftarrow \emptyset$ , propagar  
9   return false
```

# Propagación

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

```
isSolvable( $X, D, R$ ):  
1  if  $X = \emptyset$  : return true  
2   $x \leftarrow$  alguna variable de  $X$   
3  for  $v \in D_x$  :  
4      if  $x = v$  no rompe  $R$  :  
5           $x \leftarrow v$ , propagar  
6          if isSolvable( $X - \{x\}, D, R$ ) :  
7              return true  
8           $x \leftarrow \emptyset$ , 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

# 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

# Heurísticas

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

# 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

# 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

# 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**!

# 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  $X = \emptyset$  : return true  
2    $x \leftarrow$  alguna variable de  $X$   
3   for  $v \in D_x$  :  
4       if  $x = v$  no rompe  $R$  :  
5            $x \leftarrow v$   
6           if isSolvable( $X - \{x\}, D, R$ ) :  
7               return true  
8        $x \leftarrow \emptyset$   
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  $X = \emptyset$  : return true  
2   $x \leftarrow$  la mejor variable de  $X$   
3  for  $v \in D_x$  de mejor a peor :  
4      if  $x = v$  no rompe  $R$  :  
5           $x \leftarrow v$   
6          if isSolvable( $X - \{x\}, D, R$ ) :  
7              return true  
8           $x \leftarrow \emptyset$   
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  $X = \emptyset$  : return true  
2   $x \leftarrow$  la mejor variable de  $X$   
3  for  $v \in D_x$  de mejor a peor :  
4      if  $x = v$  no rompe  $R$  :  
5           $x \leftarrow v$   
6          if isSolvable( $X - \{x\}, D, R$ ) :  
7              return true  
8       $x \leftarrow \emptyset$   
9  return false
```

Las heurísticas tratan de aproximar la realidad, pueden equivocarse

# Heurísticas

Posible heurística: partir por la variable con dominio más pequeño

								1
								9
	7					6		8
						1		4
				3				2
		1			5	3		7
	5							3
								16
					9			5

# Heurísticas

Posible heurística: partir por el valor con menos apariciones

4				2				
8							1	
7			4					
3 2 5								
3 2					5			
3 5		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  $X = \emptyset$  : return true
2   $x \leftarrow$  la mejor variable de  $X$ 
3  for  $v \in D_x$  de mejor a peor :
4      if  $x = v$  no rompe  $R$  :
5           $x \leftarrow v$ , propagar
6          if isSolvable( $X - \{x\}, D, R$ ) :
7              return true
8           $x \leftarrow \emptyset$ , propagar
9  return false
```

# Sumario

Introducción

Un ejemplo

Extensiones de Backtracking

**Cierre**

# Objetivos de la clase



# Objetivos de la clase

- Identificar pseudocódigo base para backtracking y sus partes

# Objetivos de la clase

- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas

# Objetivos de la clase

- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas
- ☐ Identificar mejoras de desempeño para backtracking