Backtracking I

Clase 13

IIC 2133 - Sección 2

Prof. Mario Droguett

Sumario

El algoritmo `CountingSort()`

input : Arreglo $A[0 \ldots n-1]$, natural k
output: Arreglo $B[0 \ldots n-1]$

`CountingSort` (A, k)**:**

```
1   B[0...n-1] ← arreglo vacío de n celdas
2   C[0...k] ← arreglo vacío de k+1 celdas
3   for i = 0...k :
4       C[i] ← 0
5   for j = 0...n-1 :
6       C[A[j]] ← C[A[j]] + 1
7   for p = 1...k :
8       C[p] ← C[p] + C[p-1]
9   for r = n-1...0 :
10      B[C[A[r]] - 1] ← A[r]
11      C[A[r]] ← C[A[r]] - 1
12  return B
```

El algoritmo `CountingSort()`

```
   CountingSort (A, k):
1      B[0...n-1] ← arreglo vacío
2      C[0...k] ← arreglo vacío
3      for i = 0...k :
4          C[i] ← 0
5      for j = 0...n-1 :
6          C[A[j]] ← C[A[j]] + 1
7      for p = 1...k :
8          C[p] ← C[p] + C[p-1]
9      for r = n-1...0 :
10         B[C[A[r]] - 1] ← A[r]
11         C[A[r]] ← C[A[r]] - 1
12     return B
```

- La complejidad del algoritmo es $\Theta(n+k)$
- Si $k \in \mathcal{O}(n)$, entonces `CountingSort()` es $\Theta(n)$

¡Este es un mejor tiempo que $\mathcal{O}(n \log(n))$!

RadixSort

Otro algoritmo de ordenación en tiempo lineal es `RadixSort`

- Usado por las máquinas que ordenaban tarjetas perforadas
- Cada tarjeta tiene 80 columnas y 12 líneas
- Cada columna puede tener un agujero en una línea

El algoritmo ordena las tarjetas revisando una columna determinada

- Si hay un agujero en la columna, se pone en uno de los 12 compartimientos
- Las tarjetas con la perforación en la primera columna quedan encima
- La misma idea funciona para d columnas

Podemos generalizarla para un número natural de d dígitos

Ordenando por dígito

Consideremos un número natural de d dígitos $n_0 n_1 \cdots n_{d-1}$

- Podemos ordenar según el dígito más significativo d_0
- Según este dígito, separamos los números en *compartimientos*
- Luego, ordenados recursivamente cada compartimiento por su segundo dígito más significativo d_1
- Finalmente, combinamos los contenidos de cada compartimiento

Problema: recursión busca que no mezclemos compartimientos antes de terminar

Ordenación estable

Un ingrediente fundamental para el algoritmo que plantearemos es el siguiente

Definición

Dada una secuencia $A[0 \ldots n-1]$, sea $B[0 \ldots n-1]$ la secuencia resultante de ordenar A usando un algoritmo de ordenación $\mathcal{S}$. Sean a, a' elementos en A tales que para el algoritmo $\mathcal{A}$ son equivalentes y a aparece antes que a' en A. Diremos que $\mathcal{S}$ es **estable** si los elementos correspondientes b y b' aparecen en el mismo orden relativo en B.

Si ordenamos por el segundo dígito, un orden estable dejaría elementos que comparten segundo dígito en el mismo orden en que nos llegaron

RadixSort

El algoritmo `RadixSort` ordena por dígito **menos significativo**

- Ordena por dígito n_{d-1}
- Luego, usando el mismo arreglo, ordena por dígito n_{d-2}, **con un algoritmo estable**
- Luego de ordenar k dígitos, los datos están ordenados si solo miramos el fragmento $n_{d-k} \cdots n_{d-1}$
- Se requieren solo d pasadas para ordenar la secuencia completa

```
RadixSort(A, d):
    for j = 0 ... d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
         j-ésimo dígito menos significativo
```

Ejemplo de ejecución

	Arreglo inicial	Ordenado por unidad	Ordenado por decena	Ordenado por centena
0	0 6 4	0 0 **0**	0 **0** 0	**0** 0 0
1	0 0 8	0 0 **1**	0 **0** 1	**0** 0 1
2	2 1 6	5 1 **2**	0 **0** 8	**0** 0 8
3	5 1 2	3 4 **3**	5 **1** 2	**0** 2 7
4	0 2 7	0 6 **4**	2 **1** 6	**0** 6 4
5	7 2 9	1 2 **5**	1 **2** 5	**1** 2 5
6	0 0 0	2 1 **6**	0 **2** 7	**2** 1 6
7	0 0 1	0 2 **7**	7 **2** 9	**3** 4 3
8	3 4 3	0 0 **8**	3 **4** 3	**5** 1 2
9	1 2 5	7 2 **9**	0 **6** 4	**7** 2 9

RadixSort

```
RadixSort(A, d):
    for j = 0 . . . d − 1 :
        StableSort(A, j)    ▷ algoritmo de ordenación estable por
                              j-ésimo dígito menos significativo
```

Supongamos que A tiene n datos naturales con d dígitos y `StableSort` toma tiempo $\Theta(n + k)$

- Si cada dígito puede tomar k valores distintos
- Entonces `RadixSort` toma tiempo $\Theta(d \cdot (n + k))$
- Si d es constante y $k \in \mathcal{O}(n)$, entonces `RadixSort` es $\Theta(n)$

Dos implementaciones

Estas ideas tiene dos implementaciones

LSD string sort (*Least Significant Digit*)

- Si todos los strings son del mismo largo (patentes, IP's, teléfonos)
- Funciona bien si el largo es pequeño

MSD string sort (*Most Significant Digit*)

- Si los strings tienen largo diferente
- Ordenamos con `CountingSort()` por primer caracter
- Recursivamente ordenamos subarreglos correspondientes a cada caracter (excluyendo el primero, que es común en cada subarreglo)
- Como `Quicksort`, puede ordenar de forma independiente
- **Pero** particiona en tantos grupos como valores del primer caracter

MSD en acción

she	are	are	are	...	are
sells	by	by	by		by
seashells	she	sells	seashells		sea
by	sells	seashells	sea		seashells
the	seashells	sea	seashells		seashells
sea	sea	sells	sells		sells
shore	shore	seashells	sells		sells
the	shells	she	she		she
shells	she	shore	shore		she
she	sells	shells	shells		shells
sells	surely	she	she		shore
are	seashells	surely	surely		surely
surely	the	the	the		the
seashells	the	the	the		the

Cuidados de MSD

En la ejecución de MSD string sort se debe considerar

- Si un string s_1 es prefijo de otro s_2, s_1 es menor que s_2

$$\text{she} \leq \text{shells}$$

- Pueden usarse diferentes alfabetos
  - binario (2)
  - minúsculas (26)
  - minúsculas + mayúsculas + dígitos (64)
  - ASCII (128)
  - Unicode (65.536)
- Para subarreglos pequeños (e.g. $|A| \leq 10$)
  - cambiar a `InsertionSort` que *sepa* que los primeros k caracteres son iguales

Objetivos de la clase

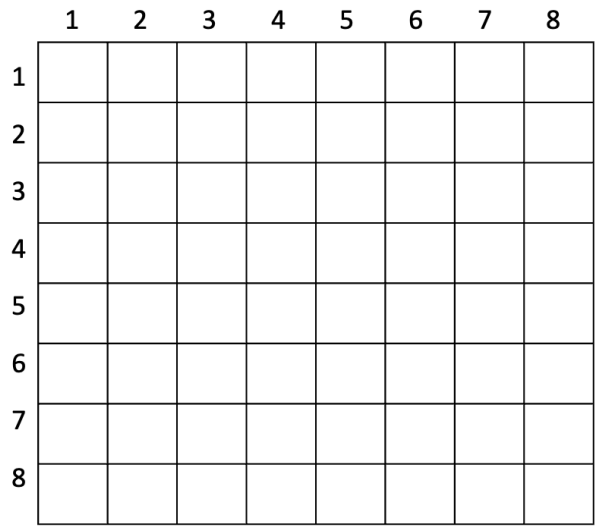
- ☐ Comprender el algoritmo `RadixSort` para ordenar strings
- ☐ Definir la clase de problemas de satisfacción de restricciones
- ☐ Comprender la dificultan inherente a los CSP
- ☐ Comprender la estrategia de backtracking
- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas

Sumario

Un problema clásico

Consideremos el problema de posicionar 8 reinas en un tablero de ajedrez de modo que no se ataquen

- Las reinas se desplazan por filas, columnas y diagonales
- Para lograr el objetivo: deben estar en columnas, filas y diagonales diferentes



¿Qué tan fácil es resolverlo?

Un problema clásico

Para modelar este problema, podemos numerar las filas y columnas

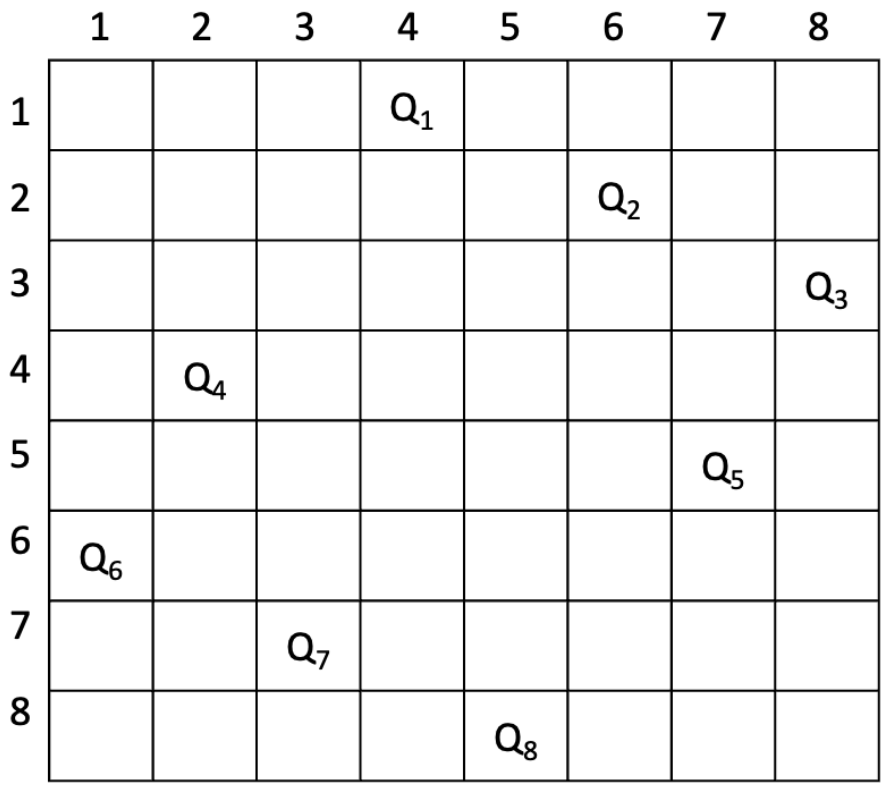
- Cada fila y columna en el rango $1 \ldots 8$
- Denotamos por x_i a la columna de la reina en la fila i
- Las posiciones de las 8 reinas se describe como un vector

$$(x_1, \ldots, x_8)$$

¿Cómo sabemos si $(4, 6, 8, 2, 7, 1, 3, 5)$ es una solución?

Un problema clásico

El vector $(4, 6, 8, 2, 7, 1, 3, 5)$ representa la siguiente configuración

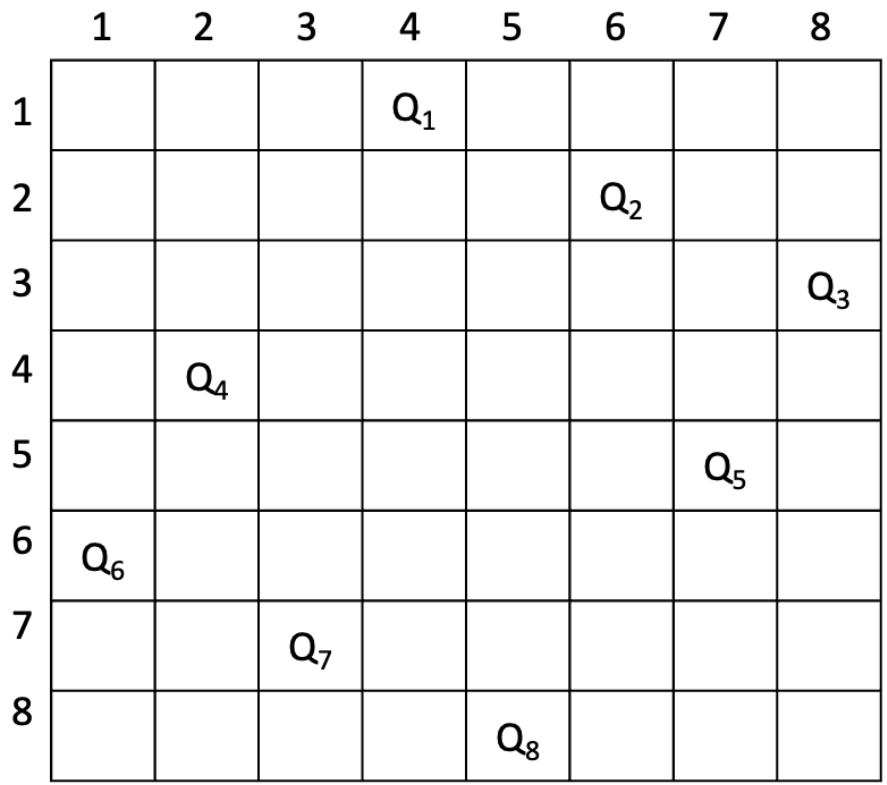


Efectivamente es solución al problema

Un problema clásico

Este problema tiene un conjunto de **restricciones**

1. Dos reinas no deben estar en la misma fila
2. Dos reinas no deben estar en la misma columna
3. Dos reinas no deben estar en un camino diagonal



Problemas de satisfacción de restricciones

Definición

Un problema de satisfacción de restricciones o *constraint satisfaction problem* **(CSP)** es una tripleta (X, D, C) tal que

- $X = \{x_1, \ldots, x_n\}$ es un conjunto de **variables**
- $D = \{D_1, \ldots, D_n\}$ es un conjunto de **dominios** respectivos
- $C = \{C_1, \ldots, C_m\}$ es un conjunto de **restricciones**

donde cada restricción involucra un subconjunto de variables de X. Una **solución** es una asignación de las variables en sus dominios tal que se satisfacen todas las restricciones.

Observemos que

- No necesariamente las variables son del mismo dominio
- Una restricción C_j puede involucrar 1, 2 o más variables de X

Problemas de satisfacción de restricciones

Ejemplo

El **problema de las 8 reinas** efectivamente es un CSP

Variables

- $X = \{x_1, \ldots, x_8\}$
- Cada variable x_i se interpreta como columna de la reina en la fila i

Dominios

- $D = \{B, \ldots, B\}$ con $B = \{1, \ldots, 8\}$
- En este caso el dominio de cada variable en X es el mismo

Restricciones

- La restricción sobre las filas está implícita en la elección de las variables
- Para las columnas: $i \neq j \rightarrow x_i \neq x_j$
- Para las diagonales: $i \neq j \rightarrow |(x_j - x_i)/(j - i)| = 1$

Otro problema clásico

Consideremos un tablero de sudoku parcialmente completado

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

¿Podemos ver el sudoku como un CSP? ¿X, D, C?

¿Es fácil resolver los CSP?

Las 8 reinas y el sudoku son ejemplos de la clase de problemas CSP

¿Qué tan rápido pueden resolverse los problemas de esta clase?

Existe un problema central en computación que puede ayudarnos

Definición

El problema de decisión **SAT** toma como input una fórmula en lógica proposicional $\varphi \in \mathcal{L}(P)$ y responde si φ es satisfacible

Ejemplo

Para el conjunto $P = \{p\}$

- $\varphi_1 = p \to \neg p$ es satifacible, pues $\sigma(\varphi_1) = 1$ para la valuación $\sigma(p) = 0$
- $\varphi_2 = p \wedge \neg p$ no es satifacible, pues no existe valuación que la haga verdadera

¿Es fácil resolver los CSP?

Ahora, para $\varphi \in \mathcal{L}(P)$, podemos interpretar la pregunta

$$¿\varphi \text{ es satisfacible?}$$

como un CSP donde

- $X = P$, conjunto de variables proposicionales
- $D = \{B \ldots, B\}$ con $B = \{0, 1\}$
- Restricción de que el valor de verdad de φ sea 1 al evaluar los valores asignados a cada variable

Si tuviéramos una forma eficiente de resolver un **CSP**,
podríamos usarla para resolver **SAT**

¿Es fácil resolver los CSP?

Teorema

El problema de decisión **SAT** es **NP-completo**

Los problemas NP-completos son considerados difíciles

- Es un problema abierto saber si se pueden resolver de manera eficiente
- Además, todo problema NP-completo sirve para resolver otro problema NP-completo

Con esto, los CSP servirían para resolver cualquier problema NP-completo

Conclusión: los CSP son difíciles

Resolviendo CSPs

Para resolver un CSP, podemos partir con **fuerza bruta**

- Generar todas las asignaciones de variables
- Verificar cada asignación para ver si cumple **todas** las restricciones
- Si se encuentra una asignación que cumple, se retorna como solución

Para un CSP (X, D, C), esto requiere revisar en general las tuplas de

$$D_1 \times D_2 \times \cdots \times D_n$$

Ejemplo

Para el problema de las 8 reinas, hay

$$8^8 = 16.777.216$$

tuplas posibles de la forma $(x_1, \ldots, x_8)$. ¿Cuántas hay en el sudoku?

¿Cómo mejoramos esto?

Sumario

Resolviendo CSPs

Quizás no es necesario generar todas las tuplas

- Podemos *informar* la búsqueda en el espacio de tuplas posibles
- Esa búsqueda puede *arrepentirse* si se rompe una restricción

Utilizaremos la estrategia algorítmica de **backtracking**, que incluye

- un conjunto de variables $X = \{x_1, \ldots, x_n\}$
- un conjunto de dominios **finitos** $D = \{D_1, \ldots, D_n\}$
- un conjunto de restricciones sobre variables

Backtracking es la forma central para resolver CSPs
(también se usa para otros problemas)

Backtracking

La estrategia de *backtracking* se basa en el siguiente principio

1. Realizar una asignación de la variable x_k cuando ya se han asignado $x_1, \ldots, x_{k-1}$
2. Se verifica si la nueva asignación **parcial** $x_1, \ldots, x_{k-1}, x_k$ puede terminar en una solución al problema
3. Si no es así, nos **retractamos** y deshacemos la asignación de x_k

El paso de retractarse se conoce como **backtrack**

- Permite descartar tuplas que violan alguna restricción
- Lo hacemos sin necesidad de conocer la tupla completa
- Nos ahorramos revisar $|D_{k+1}| \times \cdots \times |D_n|$ tuplas

Backtracking es igual o más rápido que la fuerza bruta

Backtracking

¿Tiene solución el siguiente tablero?

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

Backtracking

Queremos garantías sobre la existencia de soluciones

- Si el problema tiene solución, queremos saberlo
- Si no tiene, también queremos saberlo

Podemos responder recursivamente la pregunta

Dado un problema, ¿es posible resolverlo?

aprovechando que extender una asignación parcial

$$(x_1, \ldots, x_{k-1}) \to (x_1, \ldots, x_{k-1}, x_k)$$

genera una nueva instancia del problema

Hacemos *Backtracking* para la nueva instancia

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**

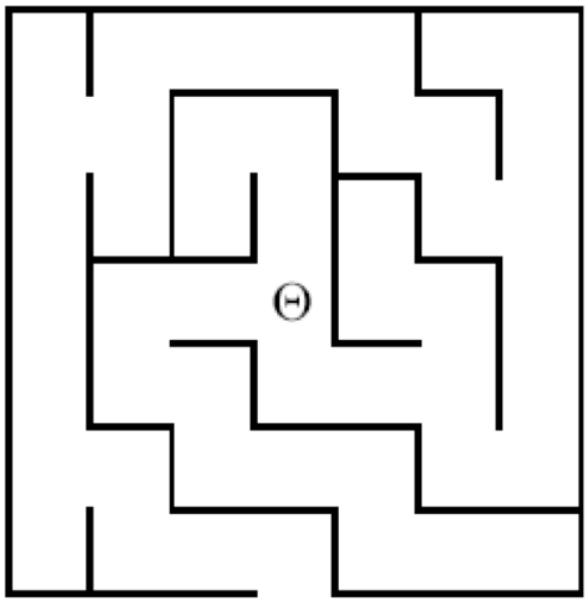
Los CSP generan muchas tuplas posibles como asignaciones para las variables de X

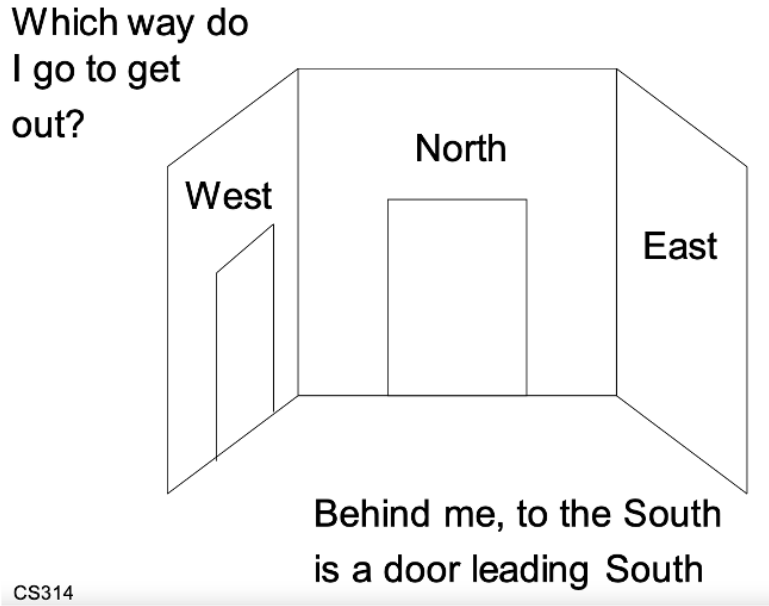
- Cada posible asignación genera un camino
- Las nuevas asignaciones abren nuevos caminos
- A la colección de todas estas alternativas le llamamos **grafo implícito**

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

Recorrido del laberinto

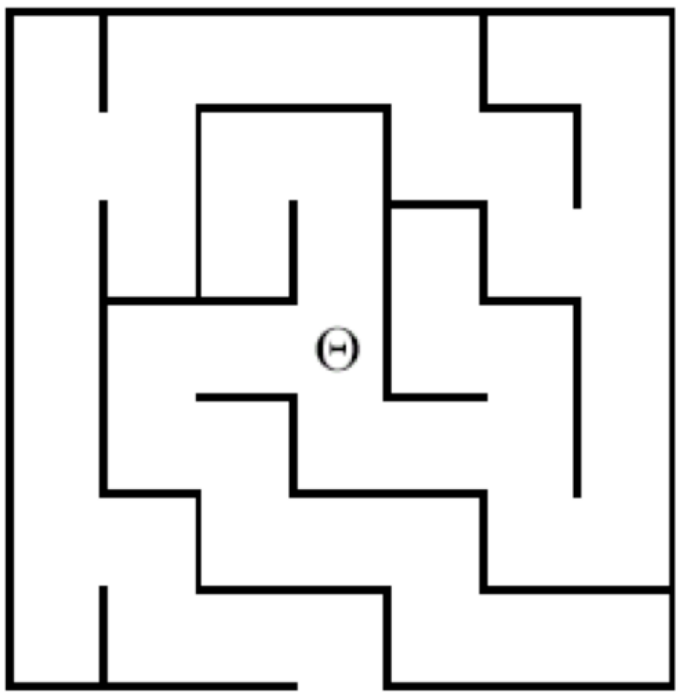
Supongamos que nos interesa salir de un laberinto dado que estamos en Θ





Podemos resolver este problema con *backtracking*

Recorrido del laberinto



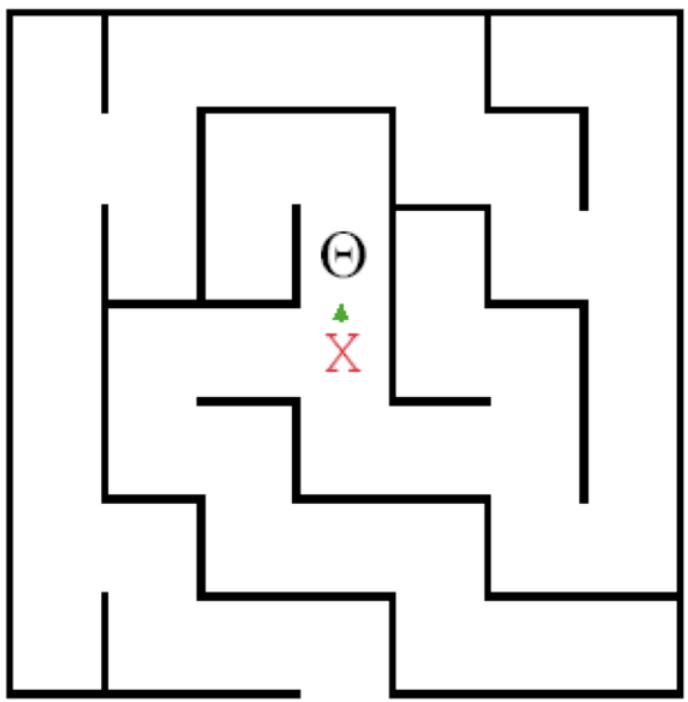
Planteamos el problema como un CSP

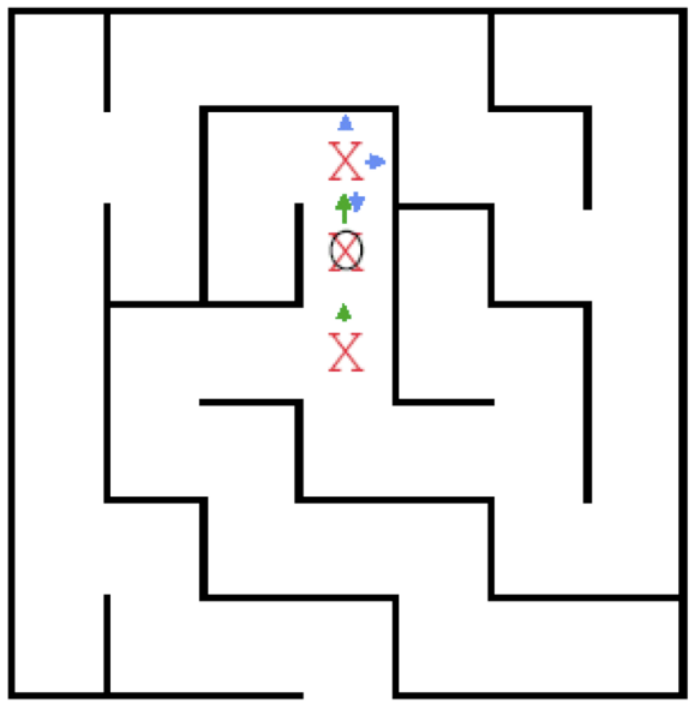
- Variables?
- Dominios?
- Restricciones?
- Qué define el *éxito*?

Caracterizamos por Θ la posición actual

Recorrido del laberinto

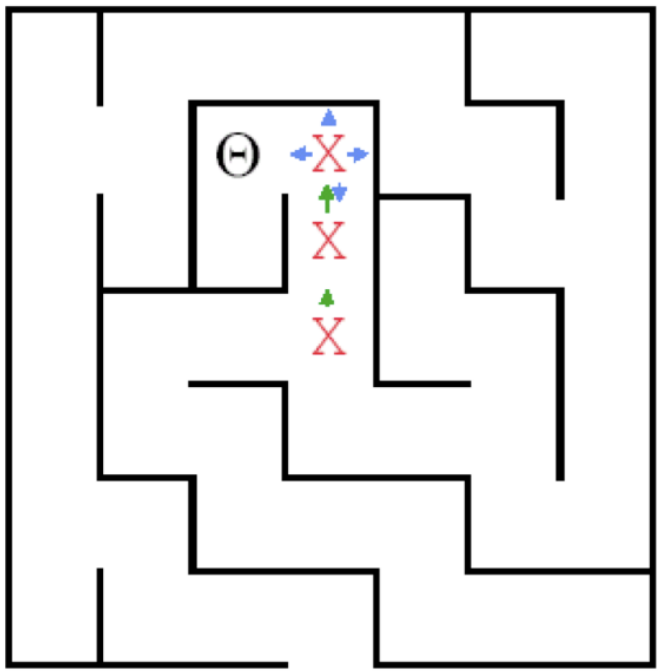
En cada nueva posición Θ solo podemos elegir dar un paso en las direcciones libres y distintas de aquella de la cual venimos

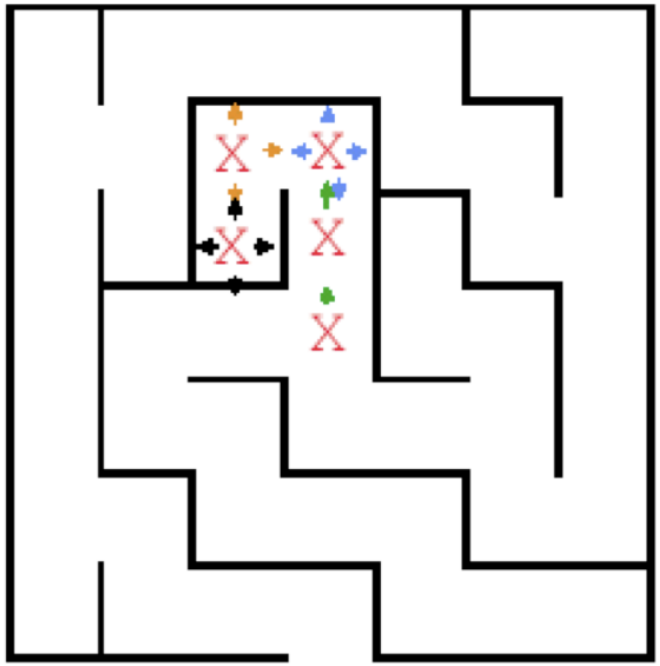




Recorrido del laberinto

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

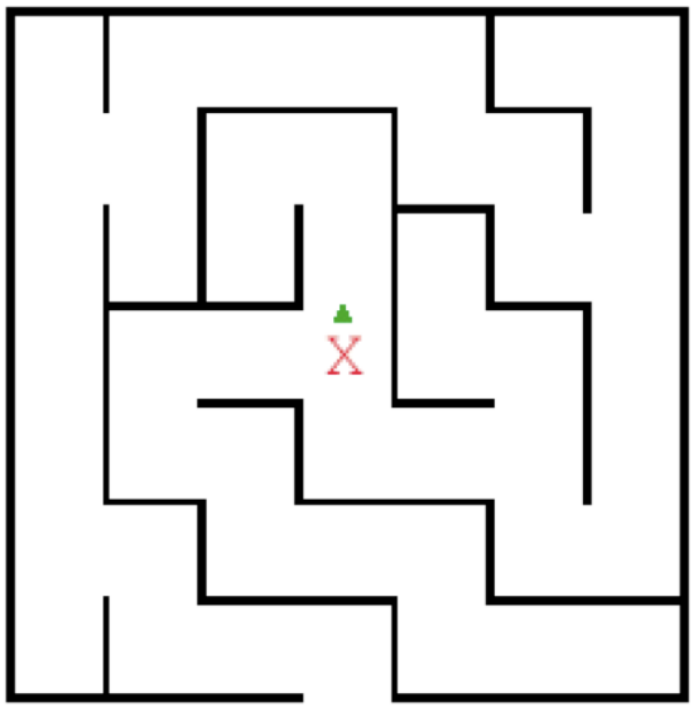


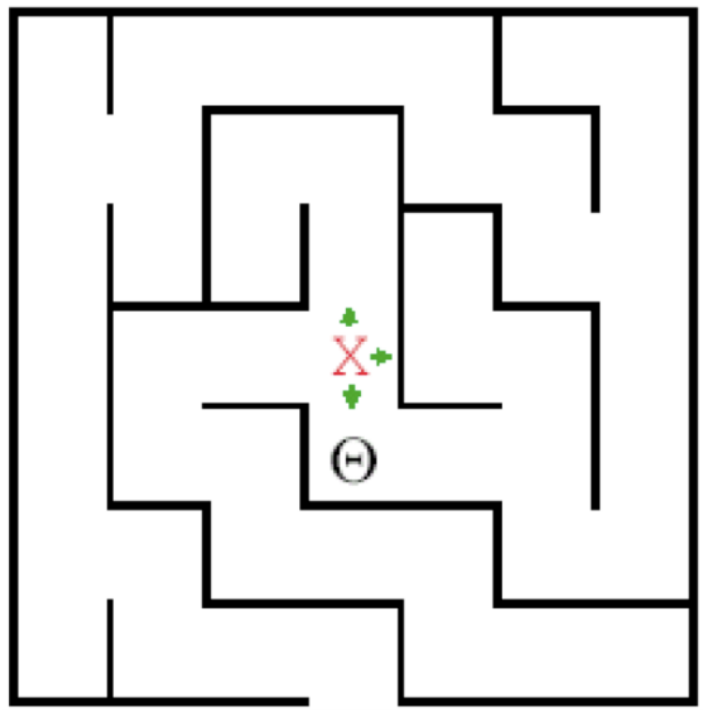


No hay más opciones: ¿hasta dónde nos *arrepentimos* con el backtrack?

Recorrido del laberinto

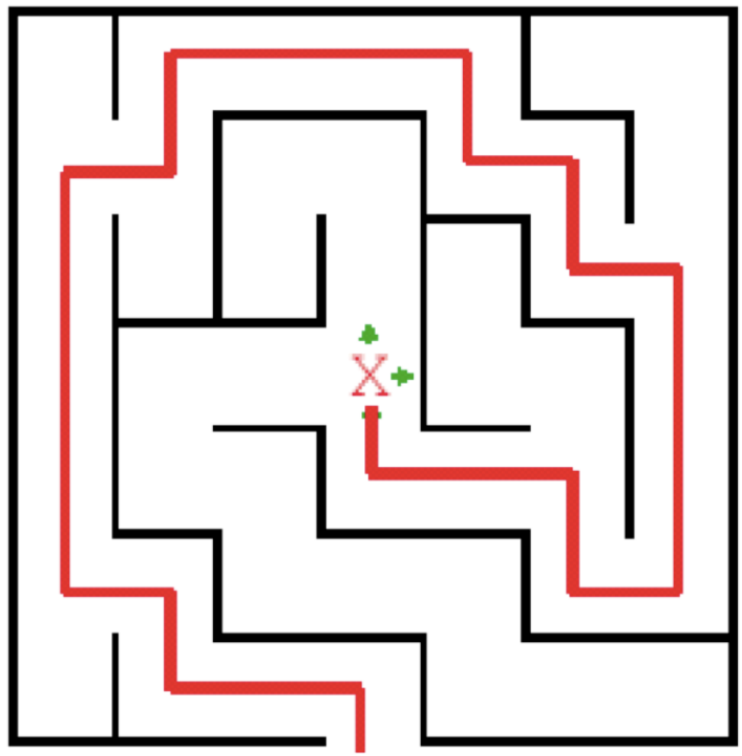
Sabemos que ir al norte no funcionó. Probamos otra opción yendo al sur.





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

Sumario

Objetivos de la clase

- ☐ Comprender el algoritmo `RadixSort` para ordenar strings
- ☐ Definir la clase de problemas de satisfacción de restricciones
- ☐ Comprender la dificultan inherente a los CSP
- ☐ Comprender la estrategia de backtracking
- ☐ Identificar pseudocódigo base para backtracking y sus partes
- ☐ Aplicar las ideas de backtracking para resolver algunos problemas