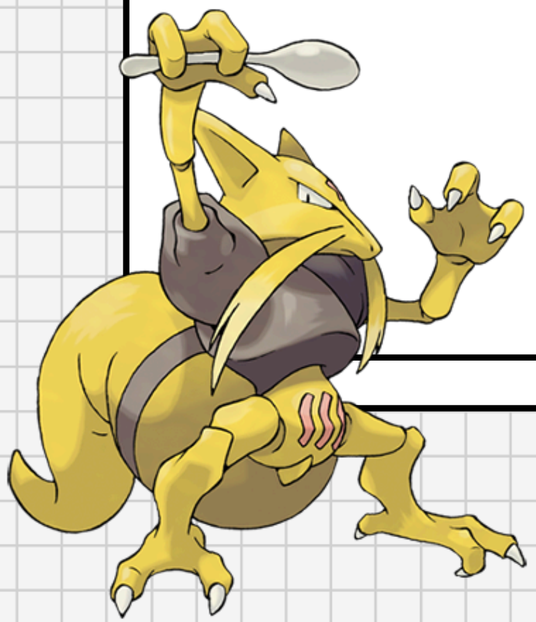
Ayudantía 11

Backtracking



Definición

Satisfacción de restricciones
(Constraint Satisfaction Problems o CSP)

Son problemas que contienen:

- Un conjunto X de variables
- Un conjunto D de dominios de las variables
- Un conjunto C de restricciones

$$X = \{x_1, \ldots, x_n\}$$

$$D = \{D_1, \ldots, D_n\}$$

$$C = \{C_1, \ldots, C_m\}$$

Una **solución** es darle un valor a {x_1,...,x_n} dentro de sus dominios y que se sigan cumpliendo todas las restricciones.

Ejemplo

Algunos CSP...

Problemas de optimización

Quiero conocer la mejor solución al problema

Problemas de decisión

Quiero conocer si existe alguna solución

Problemas de enumeración

Quiero conocer cuántas soluciones existen

Definición

Backtracking

Es una técnica algorítmica para CSP que nos permite encontrar soluciones paso a paso, que **retrocede un paso cuando encuentra un camino inválido**.

Traducido a los conjuntos de un CSP, vamos a ir **probando valores para las variables x_i**, y si alguno rompe alguna restricción (C), damos un paso hacia atrás y volvemos a probar con el siguiente valor posible en su dominio (D_i)

Backtracking

Pseudocódigo

- **X**: Variables
- **D**: Dominios
  - Dx: dominio de **X**
- **R**: Restricción
  - c/u para un subconjunto de **X**

```
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
```

Definición

Podas

Es una mejora a backtracking! Permite reducir el espacio de búsqueda, **se podan ramas** del decision tree que no conducen a una solución válida.

```
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
```

Definición

Propagación

Es una mejora a backtracking!

Cuando a una variable se le asigna valor, se puede **propagar esta información** para luego poder reducir el dominio de valores de otras variables.

```
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
```

Definición

Heurísticas

Es una mejora a backtracking!

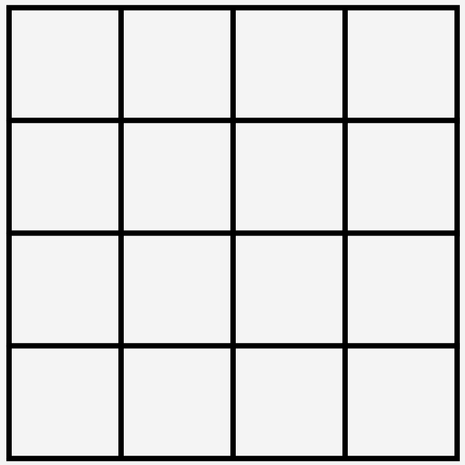
Es una **aproximación al mejor criterio** para abordar un problema

```
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
```

Rep.Gráfica

Backtracking

Veamos el queens visto en clase, pero en 4x4!

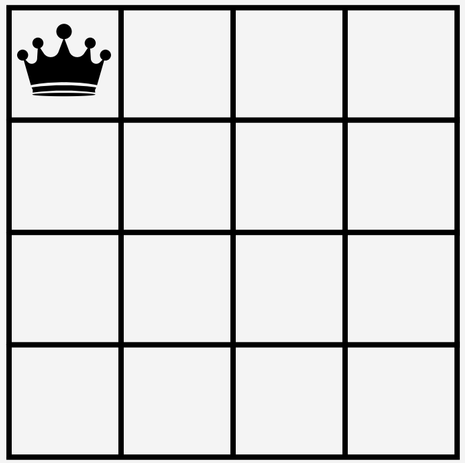


Parto con mis 16 casillas completamente vacía. Nuestras variables pasan a ser cada una de las reinas, ya que les podemos asignar una posición.
¿Cuál es la primera casilla que haría sentido asignar?

Rep.Gráfica

Backtracking

Asignamos la reina en el primer espacio disponible

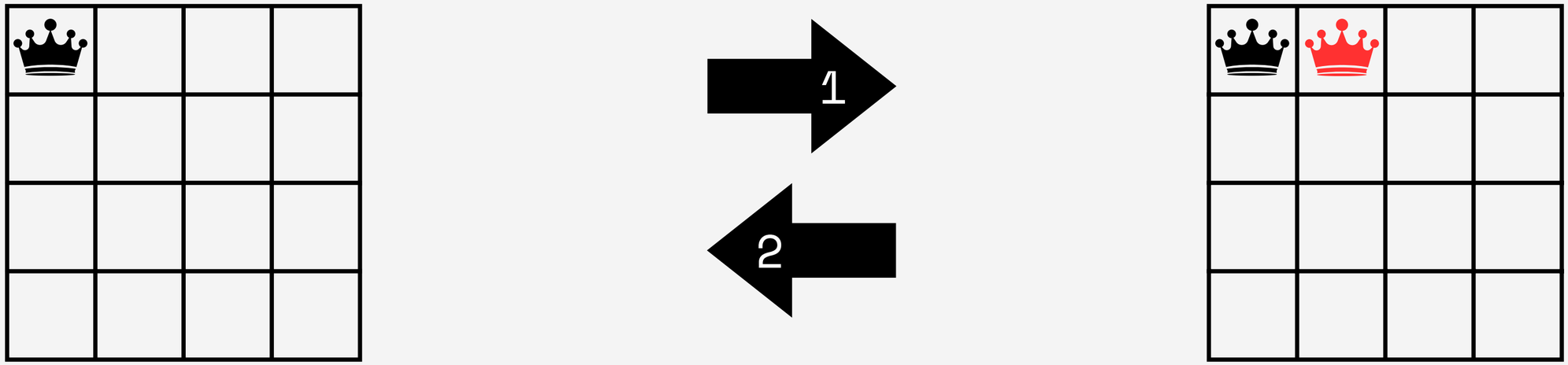


Dado que no rompo restricciones, paso a asignar la siguiente reina!

Rep.Gráfica

Backtracking

Asignamos la reina en el primer espacio disponible

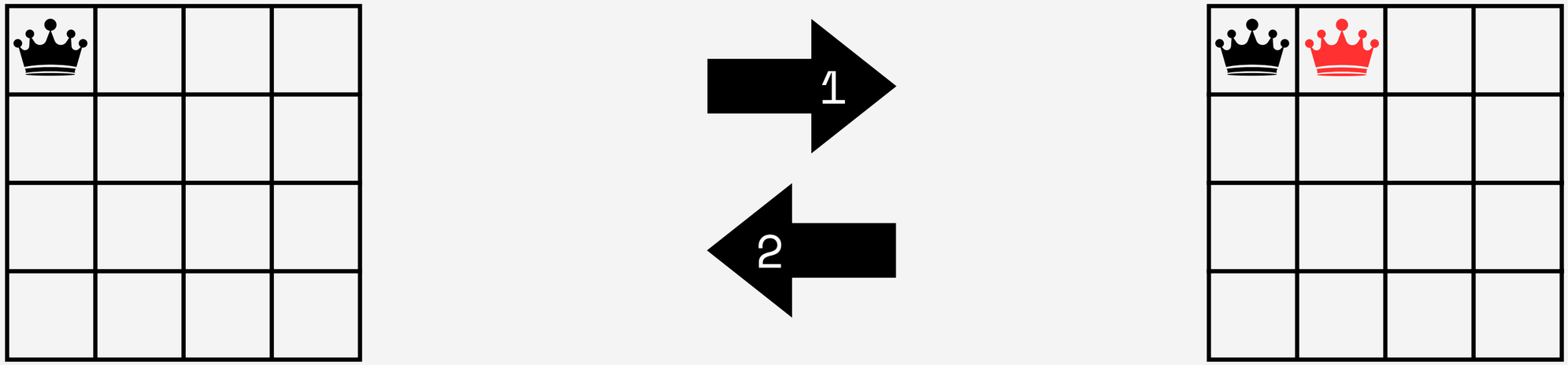


Al ponerla a un lado rompo la primera asignación, así que me devuelvo al paso anterior

Rep.Gráfica

Backtracking

Asignamos la reina en el primer espacio disponible

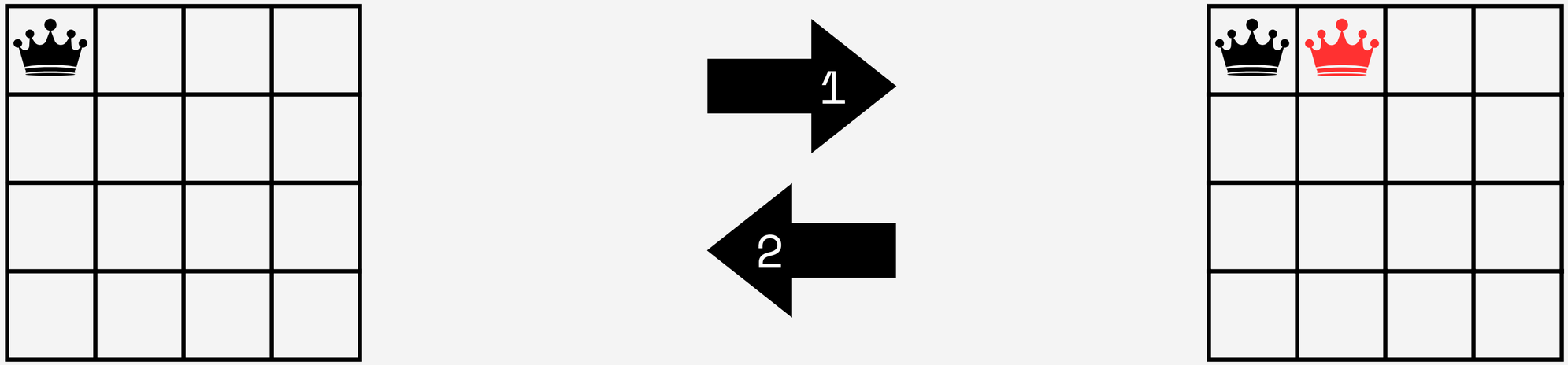


OJO! Si nos fijamos, cualquier ejecución que siguiera tras esta reina roja no sirve, así que **se poda**

Rep.Gráfica

Backtracking

Asignamos la reina en el primer espacio disponible

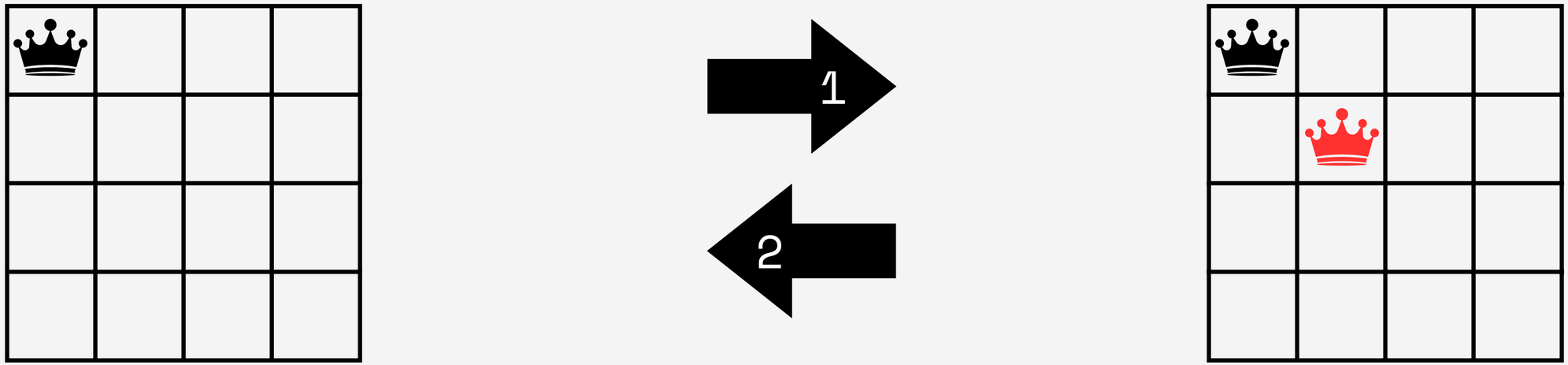


OJO! Si nos fijamos, cualquier ejecución que siguiera tras esta reina roja no sirve, así que **se poda**

Rep.Gráfica

Backtracking

Asignamos la reina en el segundo espacio disponible

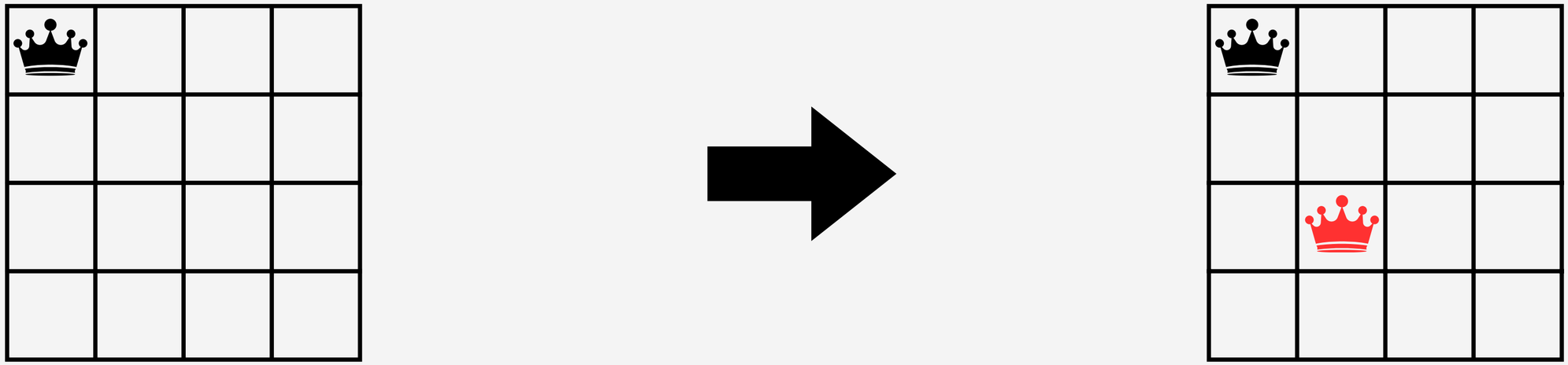


Al ponerla a una esquina rompo la primera asignación, así que me devuelvo al paso anterior

Rep.Gráfica

Backtracking

Asignamos la reina en el tercer espacio disponible

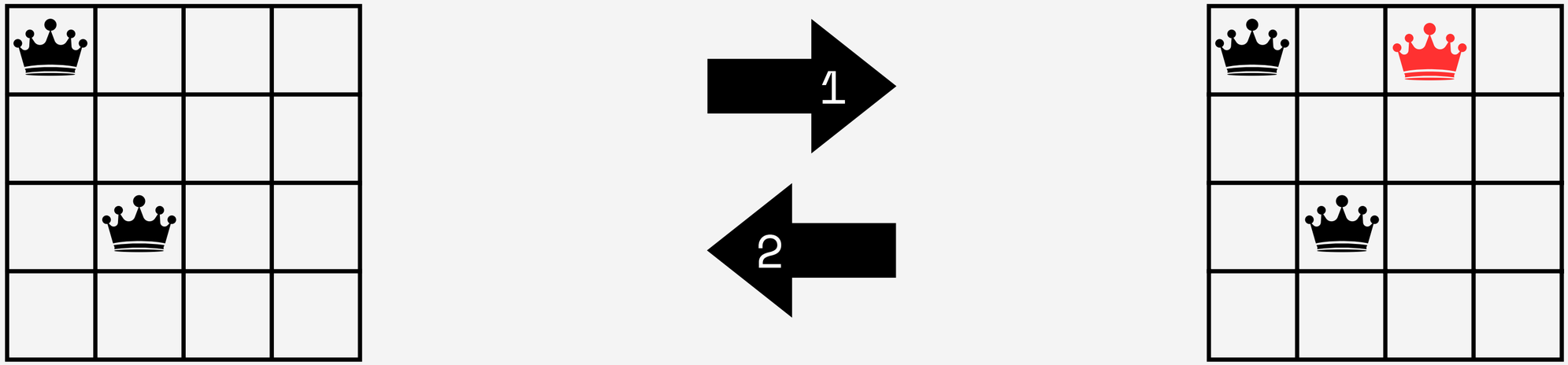


Dado que no rompo restricciones, paso a la siguiente columna!

Rep.Gráfica

Backtracking

Asignamos la reina en el primer espacio disponible

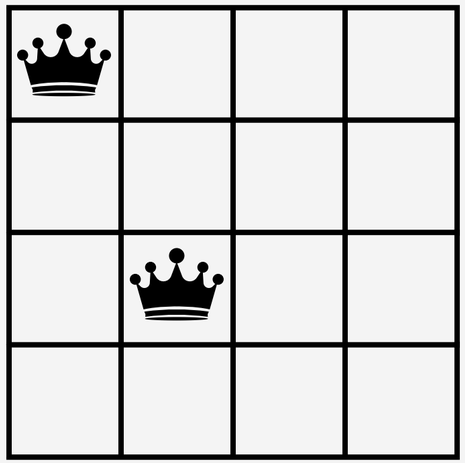


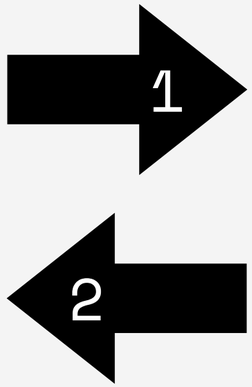
Rompo la primera asignación! (2 reinas en una fila)
Entonces, vuelvo al paso anterior

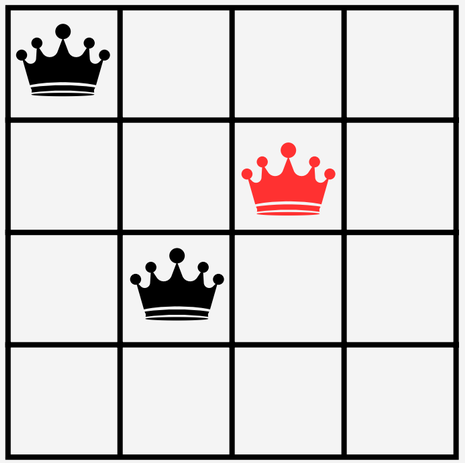
Rep.Gráfica

Backtracking

Asignamos la reina en el segundo espacio disponible





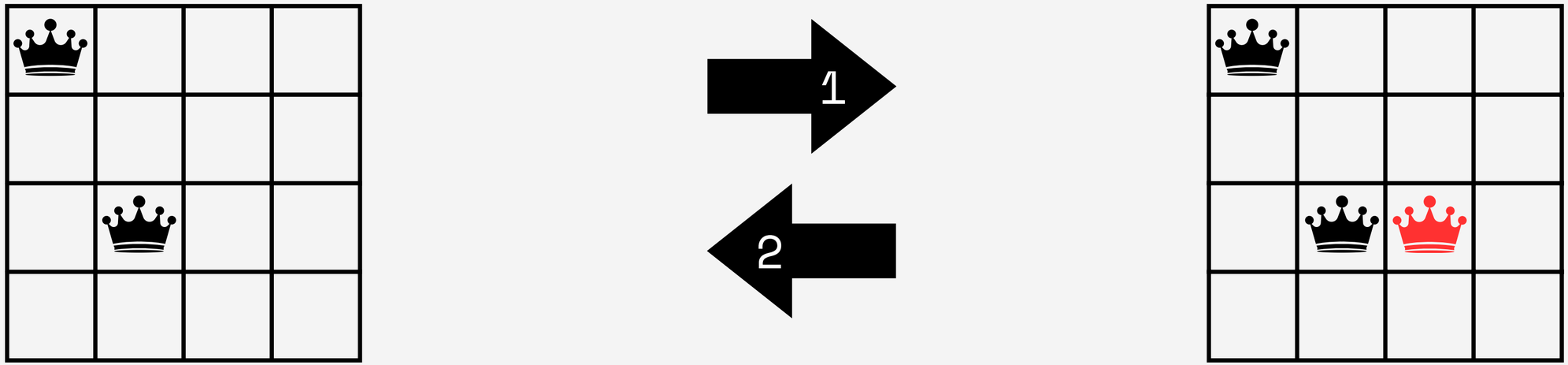


Rompo la segunda asignación! (choca en los bordes)
Entonces, vuelvo al paso anterior

Rep.Gráfica

Backtracking

Asignamos la reina en el tercer espacio disponible

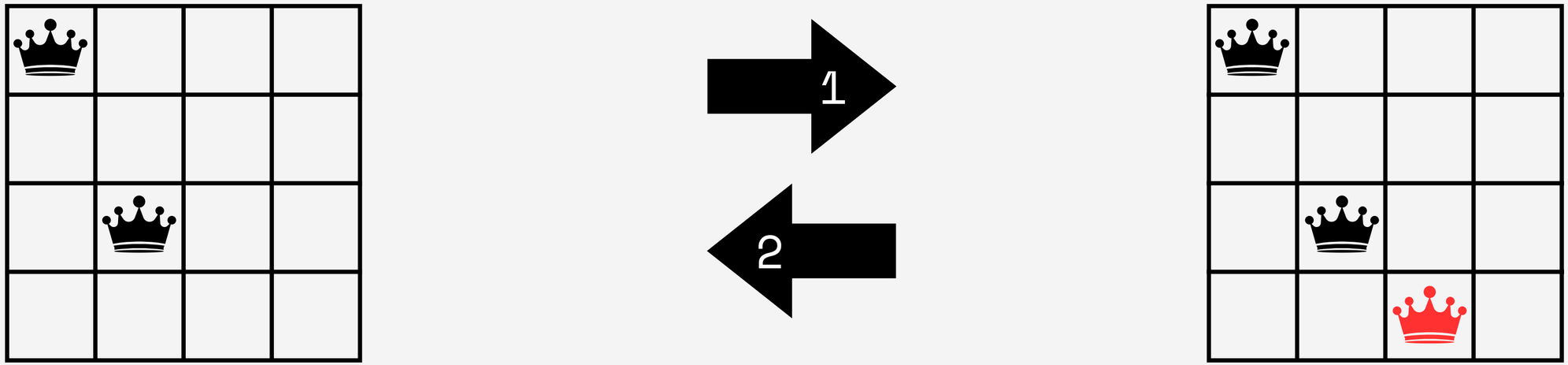


Rompo la segunda asignación! (reina en misma fila)
Entonces, vuelvo al paso anterior

Rep.Gráfica

Backtracking

Asignamos la reina en el cuarto espacio disponible

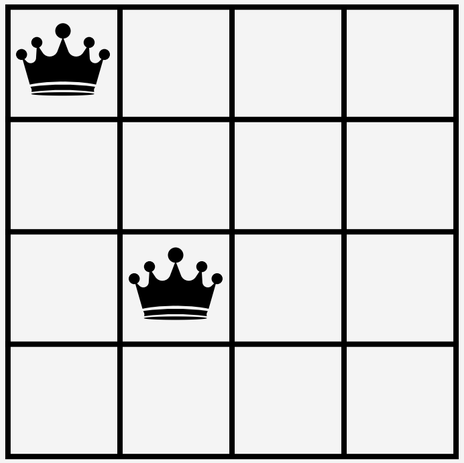


Rompo la segunda asignación! (reina en misma fila)
Entonces, vuelvo al paso anterior

Rep.Gráfica

Backtracking

Chuta pero... No tengo más opciones de reina para la tercera columna :(

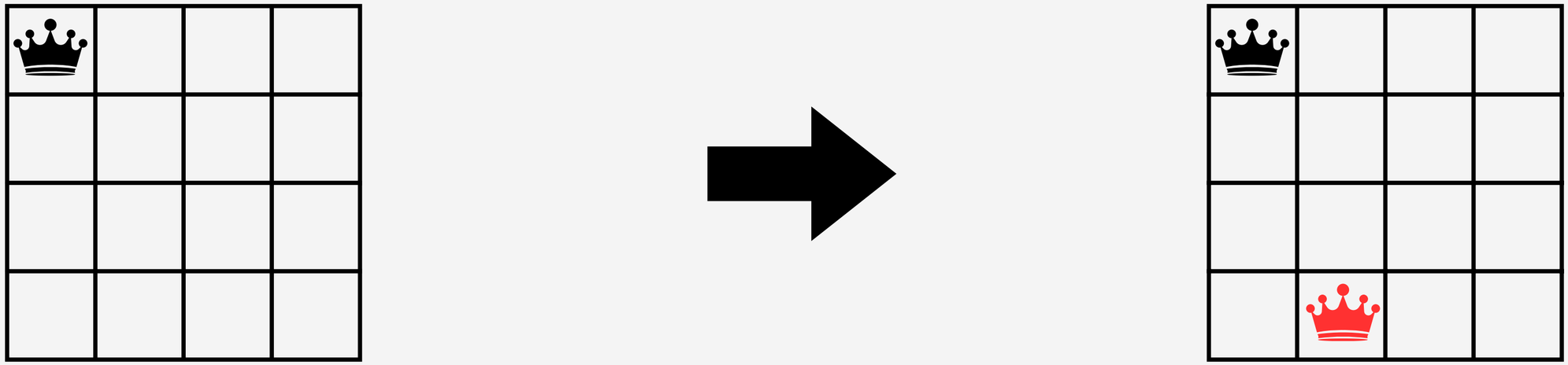


¡Tenemos que devolvernos otro paso más hacia atrás!

Rep.Gráfica

Backtracking

Habíamos probado hasta la tercera fila, entonces, asignamos la reina en el cuarto espacio disponible!



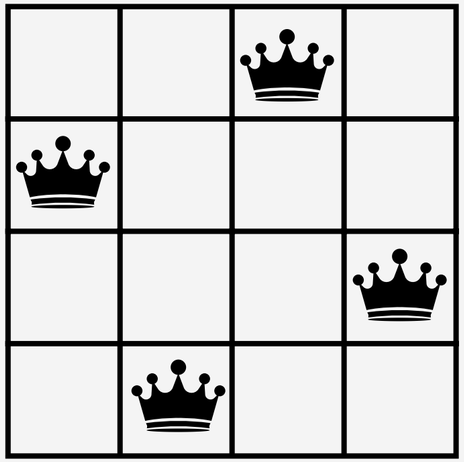
Dado que no rompo restricciones, paso a la siguiente columna!

Rep.Gráfica

Backtracking

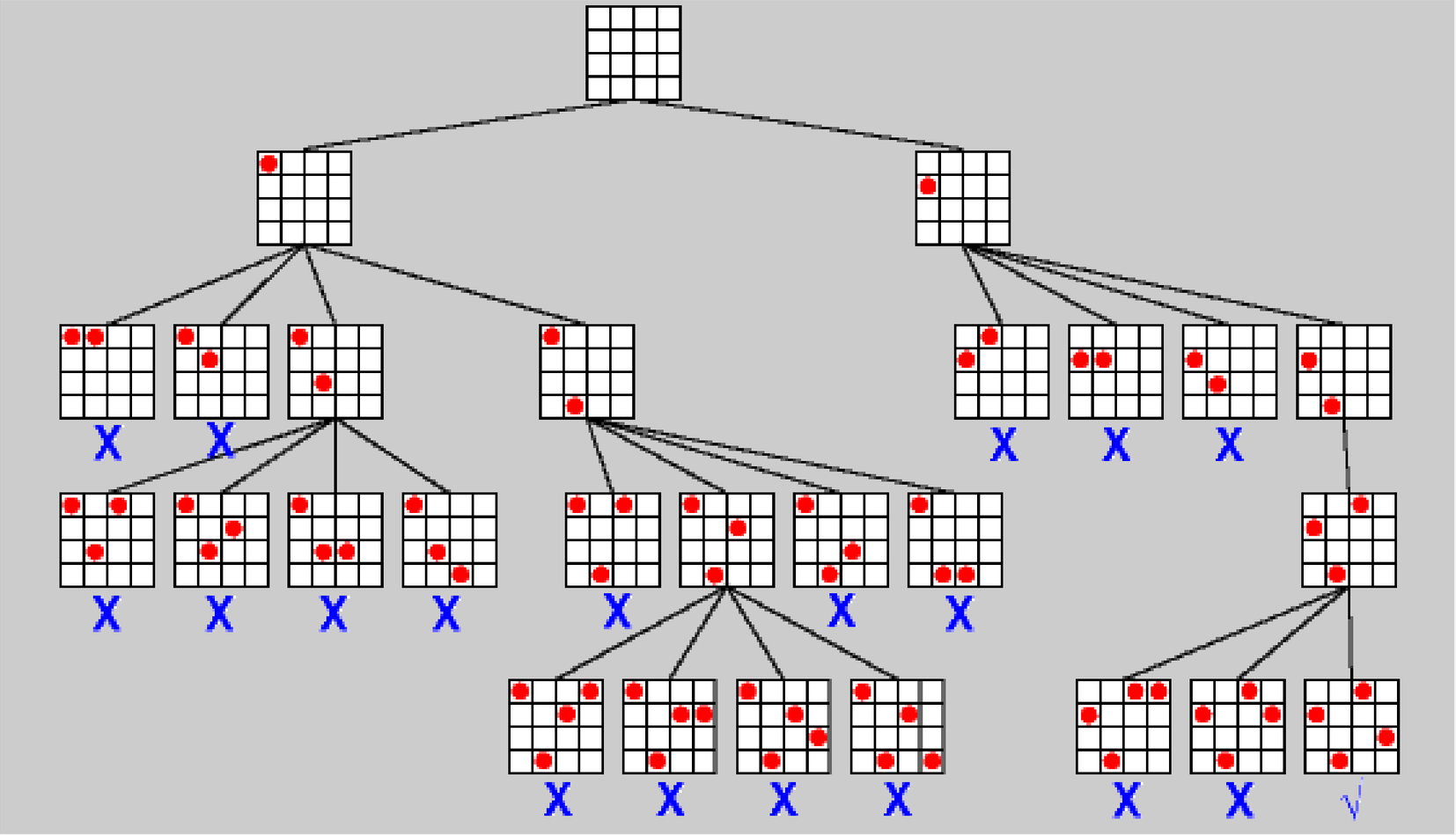
No vamos a hacer todo el árbol de ejecución o estaríamos en la ayudantía atrapados por mucho rato :)
Lo importante es que sepan la idea de **devolverse y asignar el siguiente valor**!
También podrían usar **heurísticas** para optimizarlo

Solución de este queens :D



Rep.Gráfica

Backtracking



Por si de todas maneras les daba curiosidad ver el árbol completo <3

Ejercicio 1

En una ciudad se quiere instalar estaciones de carga para vehículos eléctricos. La ciudad está dividida en una cuadrícula M de $n \times n$ celdas y cada celda puede contener una estación de carga o estar vacía. Cada estación de carga E_i permite cubrir un cuadrante de $c \times c$ celdas con la estación en su centro y se dispone de una lista Z de las zonas (celdas) de la ciudad que es prioritario que estén cubiertas por al menos una estación de carga. De igual forma, se dispone de una lista S que indica todas aquellas celdas de la ciudad en las que no se pueden instalar estaciones de carga por razones de seguridad. Finalmente, por limitaciones de presupuesto se puede instalar un máximo de k estaciones de carga.

Pregunta 2 - I2-2024-1

(a) (1 punto) El problema es claramente un CSP, identifique Variables, Dominios y Restricciones.

Pregunta 2, Inciso a - I2-2024-1

Pregunta 2.a) - I2-2024-1

Variables: Cada estación es una, y debemos asignarle un valor tal que E[i]=(x_i,y_i), para todo 1<=i<=k

Dominios: Para cada E[i] el dominio es las posiciones en las que puede estar la estación, M[nxn]

Restricciones: E[i] no pertenece a S para todo i en 1<=i<=k

No consideramos el cubrir las celdas necesarias Z ya que ese es el **objetivo** de nuestro CSP (queremos saber si se pueden asignar o no las estaciones)

(b) (3 puntos) Diseñe un algoritmo en pseudocódigo que permita ubicar las estaciones de carga para cubrir las zonas Z de la ciudad, sin ubicarlas en las celdas seguras S, utilizando a lo más de k de ellas.

Pregunta 2, Inciso b - I2-2024-1

Pregunta 2.b) - I2-2024-1

Dado el pseudocódigo de backtracking, necesitamos:

```
   isSolvable(X, D, R):                          ——— Algo que haga llamadas recursivas
1      if X = ∅ : return true                     ——— Algo que compruebe que la sol es correcta
2      x ← alguna variable de X
3      for v ∈ D_x :
4          if x = v no rompe R :                  ——— Algo que nos compruebe que no se rompe R
5              x ← v
6              if isSolvable(X − {x}, D, R) :
7                  return true
8              x ← ∅
9      return false
```

Vamos a plantear entonces 3 funciones para ello!

Pregunta 2.b) - I2-2024-1

Partimos por lo más simple: comprobar que no se rompan las restricciones.
Recordemos nuestra única restricción:

“E[i] no pertenece a S para todo i en 1<=i<=k”

Entonces, planteamos una función que verifique que la celda en la que estamos poniendo una estación no sea de las prohibidas:

```
safeCel(celda,S)
    for s in S
        if s == celda
            return false
    return true
```

Pregunta 2.b) - I2-2024-1

Como segundo paso, necesitamos tener la función que verifica si una solución es correcta. Recordemos que el objetivo del CSP es que **se cubran las celdas prioritarias de la ciudad.** Planteamos entonces una función que verifique esto:

```
testSol(Z,E)
    Aux[n x n] // Matriz auxiliar para la verificacion
    for celda in E // id que celdas estan cubiertas
        Aux[(c x c)*celda] <- cubierta // centradas en celda
    for zona in Z // verificar celdas prioritarias
        if Aux[zona] <> cubierta
            return false
    return true
```

Pregunta 2.b) - I2-2024-1

Entonces nos queda lo último! Usar las funciones que tenemos para la estructura de backtracking :)

M es la ciudad, Z lo prioritario, S lo prohibido, E las asignaciones de estaciones y k la cantidad por asignar

```
grid(M,Z,S,E,k) // E parte vacío y k en el numero de estaciones
    if testSol(Z,E) // es solución válida
        return true
    elseif k == 0 // no hay estaciones disponibles
        return false
    else
        for celda in M // el dominio completo
            if safeCel(celda,S)     // cumple restricciones
                E[k] <- celda // Estación k en esa celda
                if grid(M,Z,S,E,k-1) // una estacion menos para asignar
                    return true
                E[k] <- null ——— Esta es la linea para devolverse! Dejo en blanco
        return false                el espacio y en la siguiente iteración del for
                                    asigno el sgte valor
```

(c) (2 puntos) Un experto aconseja que se minimice la distancia media desde las celdas de Z a la estación más cercana, siempre utilizando a lo más k estaciones. Modifique su algoritmo anterior de acuerdo con lo señalado. *Hint: Puede asumir que dispone de las funciones: distancia(E_i, Z_i) que le entrega la distancia entre la Zona prioritaria Z_i y la Estación de carga E_i, y la función masCerca(Z_i) que entrega la estación de carga más cercana a Z_i.*

Pregunta 2, Inciso c - I2-2024-1

Pregunta 2.c) - I2-2024-1

Como ya hicimos safeCel(celda, S) y testSol ¿es necesario que las repitamos?
Nos fijamos entonces en lo que necesitamos cambiar para minimizar la distancia media:

- ¿Se cambian las restricciones? No! Aún hay que verificar si las celdas de E (asignadas) no están en S (prohibidas)
- ¿Se cambia la verificación del objetivo? No! solo necesitamos revusar si las celdas de E (asignadas) cubren las de Z (prioritarias)

Entonces, nos enfocaremos en cambiar la estructura de la función que contiene la recursividad!

Pregunta 2.c) - I2-2024-1

Entonces, partamos viendo lo que necesitamos cambiar en pseudocódigo! Primero, necesito conocer cuál es la distancia media hacia la estación más cercana! Para ello disponemos de distancia(E_i,Z_i) y de la función másCerca(Z_i)

```
dMedia(Z,E)
    D <- 0
    m <- cardinalidad(Z)
    for i in Z
        D <- D + distancia(masCerca(i),i)
    return D / m
```

Cuántas estaciones prioritarias hay

Distancia desde cada prioritaria hasta la asignada más cercana

Pregunta 2.c) - I2-2024-1

Ahora, la idea es que cada vez que encontremos una solución válida, vayamos guardando la menor!

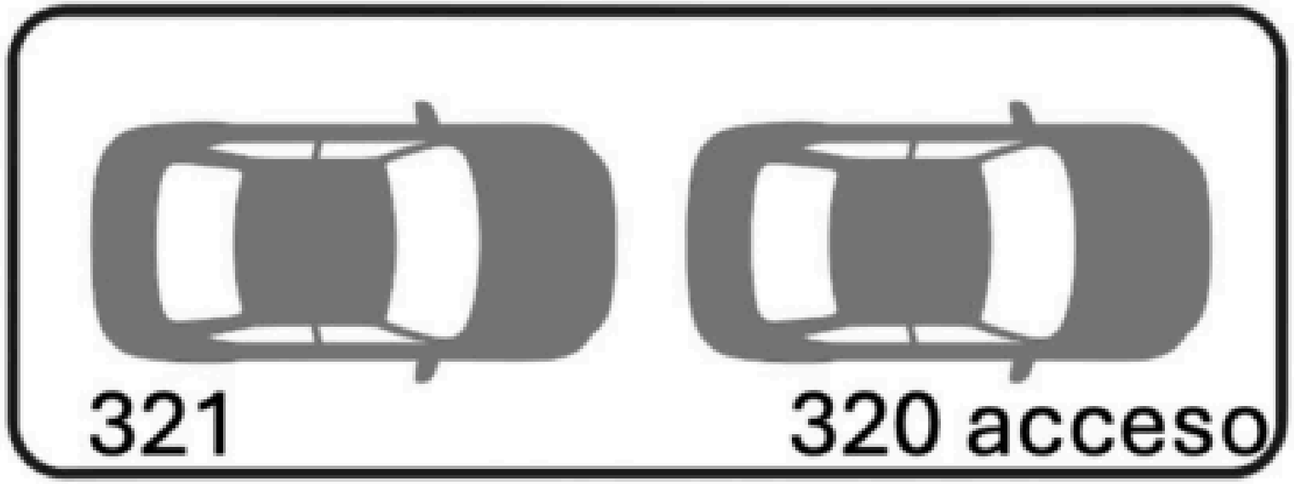
```
grid(M,Z,S,E,k,dMin) // E parte vacío, k numero de estaciones, dMin +infinito
    if testSol(Z,E) // es solución válida
        d = dMedia(Z,E)
        if d < dMin // es la de distancia minima
            dMin <- d
            minE <- E // guardamos la solucion
        return true
    elseif k == 0 // no hay estaciones disponibles
        return false
    else
        for celda in M // el dominio completo
            if safeCel(celda,S)     // cumple restricciones
                E[k] <- celda // Estación k en esa celda
                if grid(M,Z,S,E,k-1) // una estacion menos para asignar
                    // E[k] es una solucion, next
                E[k] <- null
        return false
```

La diferencia se da en que acá no retornamos

Ejercicio 2

En un subterráneo existen varios estacionamientos organizados en una configuración tándem, donde algunos espacios están dispuestos uno detrás de otro. Si dos estacionamientos (por ejemplo, 320 y 321) están en tándem, el vehículo que ocupe el espacio delantero (320) bloquea la entrada y salida del vehículo del espacio trasero (321). Esto implica que para que un vehículo pueda entrar o salir del espacio 321, el espacio 320 debe estar desocupado en ese momento. Existen también estacionamientos que no están en tándem, y los pares que están en tándem son siempre de a lo más dos espacios.

Pregunta 2 - I2-2025-02



Pregunta 2 - I2-2025-02

La administración del estacionamiento requiere asignar los N estacionamientos disponibles a M (M <= N) vehículos. Para realizar esto, cada vehículo tiene asociada una hora de llegada y una hora de salida, la cual se debe considerar para que sea posible el acceso y salida desde los estacionamientos en tándem. Así, la administración debe determinar una asignación de vehículos a estacionamientos que cumpla las siguientes condiciones:

- Un mismo estacionamiento no puede ser asignado a más de un vehículo en intervalos de tiempo superpuestos.
- En los estacionamientos tándem, un vehículo no puede ser asignado al espacio trasero si durante su llegada o salida el espacio delantero estará ocupado.
- Todos los vehículos deben poder estacionar y retirarse sin bloqueos, respetando sus horarios.

Pregunta 2 - I2-2025-02

En este contexto:

a) (1 pt.) Modele este problema como un CSP, definiendo claramente variables, dominios y restricciones.

Pregunta 2, Inciso a - I2-2025-02

Pregunta 2.a) - I2-2024-1

Variables: Cada vehículo/estacionamiento es una, y debemos asignarle un valor tal que V[i]=pos y se cumple que 1<=i<=M

Dominios: Si escogimos vehículos en variables, los estacionamientos son los correspondientes dominios. Análogo al revés.

Restricciones: Tenemos varias:

- Cada E[i] tiene un atributo blockBy
- Cada V[i] tiene atributos hIn, hOut
- E[i] puede ser asignado a V[j] si:
  - E[i] está libre en [V[j].hIn,V[j].hOut]
  - E[i].blockBy está libre en V[j].hIn y V[j].hOut

Pregunta 2.a) - I2-2024-1

Además, el **objetivo del CSP** que vamos a querer lograr es que pueda haber una asignación exitosa de los N vehículos en los M estacionamientos (considerando que N<=M), sujeto a las horas de llegada y salida de cada estacionamiento

b) (3 pts.) Proponga el pseudocódigo de la función setParking() para resolver la asignación de estacionamientos, indicando además los parámetros adecuados para realizar la llamada inicial.

Pregunta 2, Inciso b - I2-2025-02

Pregunta 2.b) - I2-2025-02

Miramos la estructura general de Backtracking:

```
   isSolvable(X, D, R):                          ← Algo que haga llamadas recursivas
1      if X = ∅ : return true                    ← Algo que compruebe que la sol es correcta
2      x ← alguna variable de X
3      for v ∈ D_x :
4          if x = v no rompe R :                 ← Algo que nos compruebe que no se rompe R
5              x ← v
6              if isSolvable(X − {x}, D, R) :
7                  return true
8              x ← ∅
9      return false
```

Vamos a plantear entonces 4 funciones para ello!

Pregunta 2.b) - I2-2025-02

Partimos por lo más simple de nuevo: comprobar que no se rompan las restricciones.
Recordemos lo primero que debemos cumplir en cualquier estacionamiento: **Si el estacionamiento está vacío en el intervalo que lo usaremos, entonces está libre**.
Entonces, planteamos una función que verifique esto:

```
libre(e,hIn,hOut):
1 if e no tiene un vehiculo asignado en el intervalo (hIn,hOut):
2      return true                          // se puede asignar
3 return false
```

Pregunta 2.b) - I2-2025-02

Dado que ahora no tenemos solo una restricción, tenemos que además hacer otra función que revise que también se cumpla. Revisamos antes si un espacio estaba vacío, pero ahora nos fijaremos en si está bloqueado! Definimos esto en una función:

```
libreBlock(e,hIn,hOut):
1 if e no tiene un vehiculo asignado en hIn && hOut :
2      return true                          // se puede asignar
3 return false
```

*No usamos blockBy ya que lo llamaremos con args de e.blockBy. Se ve en el siguiente paso:D

Pregunta 2.b) - I2-2025-02

Ya tenemos funciones que comprueben el requisito del estacionamiento, pero por separado. Por comodidad, haremos una función que compruebe ambos requisitos al mismo tiempo:

```
test(v,e):                                  // vehiculo, estacionamiento
1 if libre(e,v.hIn,v.hOut)&& libreBlock(e.blockBy,v.hIn,v.hOut):
2      return true                          // se puede asignar
3 return false
```

Esto comprueba que, dado un estacionamiento e, este esté libre en el intervalo [v.hIn,v.hOut] y que aparte, su entrada no esté bloqueada en v.hIn y en v.hOut para poder entrar y salir.

Pregunta 2.b) - I2-2025-02

Finalmente, incorporamos todo a la estructura general de Backtracking!

```
setParking(V,E):                          // {vehículos},{estacionamientos}
1 if V=Ø: return true
2 v ↤ un vehículo de V                    // el vehículo a asignar
3 for e ∈ E:
4       if test(v,e) :                    // si e puede ser asignado a v
5             v ↤ e                       // se asigna e a v
6             if setParking(V-{v},E):     // intenta asignar el siguiente v
7                   return true
8             v ↤ Ø                       // se intenta nuevo e
9 return false                            // No hay solución válida
```

Es importante mencionar que no necesitamos función para ver si la solución es válida ya que cuando asignamos todo valor ya podemos afirmarlo!

c) (2 pts.) Explique cómo aplicaría en su solución propagación, de modo de mejorar el desempeño de su algoritmo.

Pregunta 2, Inciso c - I2-2025-02

Pregunta 2.c) - I2-2025-02

Recordemos que **propagación** es cuando pasamos a futuras variables el valor de asignaciones anteriores para acotar su dominio. Hay varias maneras de hacerlo, pero una opción es que al asignar válidamente a un vehículo v el estacionamiento e, cuando e es en tándem, **se registren como utilizados en ese rango ambos estacionamientos.**

Feedback