- En clases se vió backtracking de manera teórica, donde resolver un problema se trata de buscar la tupla $X = \{x_1, x_2, ..., x_n\}$ que satisface la función de criterio $P(x_1, x_2, ..., x_n)$. P indica que X es una solución al problema.

- Sabemos que existen las restricciones explícitas: $x_i \in S$. Básicamente define el "dominio" de x_i.

  Ejemplo: En el sudoku, tenemos que elegir tuplas (i, j, k) donde la celda (i, j) lleva el valor k. Entonces, $S = \{(i, j, k) \mid 1 \leq i, j, k \leq 9\}$

- Están tambien las restricciones implícitas, que relacionan los x entre ellos. Estas son las que caracterizan al problema.

  Ejemplo: En el sudoku, tenemos las restricciones entre filas, columnas y cuadrantes

- Como se vio en clases, la gracia del Backtracking es ir construyendo X paso a paso, asegurandose de que en el paso j, X satisface la función parcial de criterio $P_j(x_1, x_2, ..., x_j)$. Pj indica si los primeros j elementos de X pueden formar un X solución. Así, si P_j retorna false, sabemos que X es inviable y nos ahorramos el probar muchas de las siguientes permutaciones. Nótese que para un problema que requiere n pasos para resolver, $P_n = P$

  Ojo: P_j puede dar un falso positivo: en muchos problemas detectar si un X es inviable siendo que cumple todas las restricciones puede ser muy difícil. Lo que sí jamás debe pasar es que P_j descarte un estado X que sea viable.

- En cada paso del backtracking elegimos el siguiente x para agregar a X: **Llamaremos a esto una "jugada"**.

  Ejemplo: en el sudoku, poner un 8 en la celda (3,5)

- Por lo mismo, para el paso número j, $X = \{x_1, x_2, ..., x_{j-1}\}$, buscamos un x_j que satisfaga P_j. **Llamaremos a los x_j que satisfacen P_j "jugadas legales"**

  Ejemplo: en el sudoku, poner un 8 en la celda (3,5) dado que no habia ningun 8 en esa fila, ni en esa columna, ni en ese cuadrante

- En backtracking existen dos tipos de problemas: a falta de nombres les diremos los de **"asignación"** y los de **"planificación"**.

  Los de asignación son los problemas en los que se debe buscar literalmente un conjunto de x tal que satisfagan todas las restricciones. *Ejemplo: sudoku, n-queens, particiones* La gracia de estos problemas es que podemos elegir los valores en cualquier orden, mientras cumplan las restricciones. En estos casos deshacer una jugada es trivial, ya que significa simplemente borrar el último x de la lista / eliminar la ultima asignación.

  Los de planificación son problemas donde cada jugada altera el estado del problema, por lo que es de vital importancia el orden en el que se llevan a cabo las jugadas. *Ejemplo: cindy's puzzle, bejeweled, collapse* En estos problemas, deshacer una jugada es no trivial, ya que no basta con eliminar la ultima asignación para devolver el problema a su estado anterior. Para esto es sumamente necesario, junto con la jugada que hiciste, guardar cuales fueron los cambios que se efectuaron en el problema, de manera de poder restaurarlos todos. No solo eso, sino que la validez de una jugada va a depender del estado.

  Ejemplo: En el bejeweled/candy crush un intercambio en la misma posición puede ser o no legal según el estado.

Backtracking inteligente

Podas

Vimos que P_j puede entregar un falso positivo. Esto es debido a que por definición, P_j sólo considera las restricciones explícitas e implícitas del problema. El problema es que todas las jugadas afectan de una u otra manera las jugadas siguientes, de manera **indirecta**, lo cual no es trivial de detectar. Pero estudiando detenidamente el problema, podemos ver casos que si bien no están especificados en las restricciones del problema, garantizan que no es resolvible. La idea es entonces mejorar la función P_j de manera de que considere estas restricciones. Cada uno de estos casos que incluyamos en la función parcial constituye una **poda**.

Ejemplo: En el sudoku, luego de cada jugada podemos revisar si alguna celda sin asignar en alguna parte del tablero quedó sin posibles asignaciones futuras. De ser así, no tiene sentido seguir.

Heurísticas

En cada paso puede existir más de una jugada legal. ¿Cuál probamos primero? Tenemos que hacer una estimación de cual es la jugada que más nos conviene... pero, ¿cómo hacemos eso?

Sabemos que en todo algoritmo de backtracking, buscar la solución significa recorrer el árbol de estados probando las distintas combinaciones. Cada vez que se nos presenta una decisión, podemos equivocarnos o tomar una decision correcta. Nuestro objetivo es equivocarnos lo menos posible, pero no podemos saber con exactitud cual es la opcion que más conviene. Lo que sí podemos hacer es aumentar la probabilidad de tomar una decisión correcta, disminuyendo entonces la probabilidad de explorar gran parte del árbol.

En el contexto de backtracking, una **heurística** consiste en rankear bajo cierto criterio las opciones que tenemos, para saber cual conviene más tomar. Esto es usando únicamente la información conocida hasta el momento. Cómo se hace exactamente dependerá del problema, pero hay ciertos puntos que tienen en común. En general, existen dos tipos, heurísticas de variable y heurísticas de valor.

Las **heurísticas de variable** responden a la pregunta "¿Qué celda asigno ahora?" ó "¿Que pieza debería mover ahora?". Muchos problemas solicitan elegir una variable que debe tomar un valor. Una heurística transversal a todos estos problemas es elegir la variable con el menor dominio, considerando las restricciones. Esto tiene sentido debido a que como tiene menos opciones, se ramifica menos el arbol de búsqueda, por lo que el arbol generado tiene menos nodos.

Ejemplo: en el sudoku siempre te va a convenir asignar primero las celdas que tienen una sola opción, ya que no puedes equivocarte. Si no quedan, entonces pasas a las que tengan 2 opciones, ya que hay menos probabilidades de equivocarte, y así

Las **heurísticas de valor** responden a la pregunta "¿Y que valor le asigno a la celda?" ó "¿Y hacia donde muevo la pieza?". Los problemas anteriores, o problemas donde la siguiente variable a asignar ya viene dada, requieren que se decida que hacer con esa variable. Una heurística transversal es la de elegir la opción que restringe lo menos posible las opciones de las otras variables aun no asignadas. Esto tiene sentido ya que si una variable sin asignar queda con 0 posibles valores, el problema es irresolvible. Es deseable entonces tratar de alejarse de ese estado lo más posible.

El backtracking normal sería

Algorithm 1 *solve*(*state*)

```
if is_solution(state) then
  return true
end if
variable ← get_next_variable_to_assign(state)
S ← get_all_possible_values_for(variable)
for all value ∈ S do
  if is_legal(state, variable, value) then
    make_move(state, variable, value)
    if solve(state) then
      return true
    end if
    undo_move(state, variable, value)
  end if
end for
return false
```

Agregando todo lo que vimos antes, puede quedar algo como lo que sigue:

Algorithm 2 *solve*(*state*)

```
if is_solution(state) then
  return true
end if
// Heurísticas de variable
variable ← get_best_variable_to_assign(state)
// Heurísticas de valor
S ← get_all_legal_and_sorted_values_for(variable)
for all value ∈ S do
  // Podas
  consequences ← make_move_and_check_viability(state, variable, value)
  if consequences ≠ state is unviable and solve(state) then
    return true
  end if
  undo_move(state, variable, value, consequences)
end for
return false
```

No olvides que al final, Backtracking se trata de probar **todas** las opciones, pero detectando cuando hay opciones que definitivamente no sirven para no tener que perder el tiempo. La modelacion de la n-tupla y la función P no es necesaria y para algunos problemas puede ser muy complicada, pero sirve para guiar como hacerlo.

El como implementar el backtracking, donde y como incluir podas y heurísticas queda a tu criterio, pero lo anterior puede servirte de guía para hacerlo.

En caso de que el problema lo permita, usa estructuras de datos para acelerar los procesos, ya sea revisión del estado de término, revisión de podas y o heurísticas, etc.