

# Resolución de Problemas

- La resolución de problemas es una capacidad que consideramos inteligente
- Somos capaces de resolver problemas muy diferentes
  - Encontrar el camino en un laberinto
  - Resolver un crucigrama
  - Jugar a un juego
  - Diagnosticar una enfermedad
  - Decidir si invertir en bolsa
  - ...
- El objetivo es que un programa también sea capaz de resolverlos

# Resolución de Problemas

- Deseamos definir cualquier tipo de problema de manera que se pueda resolver automáticamente
- Necesitamos:
  - Una representación común para todos los problemas
  - Algoritmos que usen alguna estrategia para resolver problemas definidos en esa representación común

# Definición de un Problema

- Si abstraemos los elementos de un problema podemos identificar:
  - Un punto de partida
  - Un objetivo a alcanzar
  - Acciones a nuestra disposición para resolver el problema
  - Restricciones sobre el objetivo
  - Elementos que son relevantes en el problema definidos por el tipo de dominio

# Representación de problemas

- Existen diferentes formas de representar problemas para resolverlos de manera automática
- Representaciones generales
  - **Espacio de estados:** un problema se divide en un conjunto de pasos de resolución desde el inicio hasta el objetivo
  - **Reducción a subproblemas:** un problema se puede descomponer en una jerarquía de subproblemas
- Representaciones para problemas específicos
  - **Resolución de juegos**
  - **Satisfacción de restricciones**

# Representación de problemas: Estados

- Podemos definir un problema por los elementos que intervienen y sus relaciones
- En cada instante de la resolución de un problema esos elementos tendrán unas características y relaciones específicas
- Denominaremos **Estado** a la representación de los elementos que describen el problema en un momento
- Distinguiremos dos estado especiales el **Estado Inicial** (punto de partida) y el **Estado Final** (objetivo del problema)
- ¿Que incluir en el estado?

# Modificación del estado: operadores

- Para poder movernos entre los diferentes estados necesitamos operadores de transformación
- **Operador:** Función de transformación sobre la representación de un estado que lo convierte en otro estado
- Los operadores definen una relación de accesibilidad entre estados
- Representación de un operador:
  - Condiciones de aplicabilidad
  - Función de transformación
- ¿Que operadores? ¿Cuántos? ¿Que granularidad?

# Espacio de estados

- Los estados y su relación de accesibilidad conforman lo que se denomina **espacio de estados**
- Representa todos los caminos que hay entre todos los estados posibles de un problema
- Podría asimilarse con un mapa de carreteras de un problema
- La solución de nuestro problema esta dentro de ese mapa

# Solución de un problema en Espacio de Estados

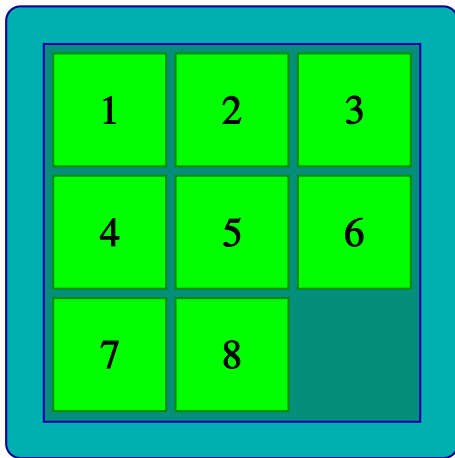
- **Solución:** Secuencia de pasos que llevan del estado inicial al final (secuencia de operadores) o también el estado final
- **Tipos de solución:** una cualquiera, la mejor, todas
- **Coste de una solución:** Gasto en recursos de la aplicación de los operadores a los estados. Puede ser importante o no según el problema y que tipo de solución busquemos



# Descripción de un problema en Espacio de Estados

- Definir el conjunto de estados del problema (explícita o implícitamente)
- Especificar el estado inicial
- Especificar el estado final o las condiciones que cumple
- Especificar los operadores de cambio de estado (condiciones de aplicabilidad y función de transformación)
- Especificar el tipo de solución:
  - La secuencia de operadores o el estado final
  - Una solución cualquiera, la mejor (definición de coste), ...

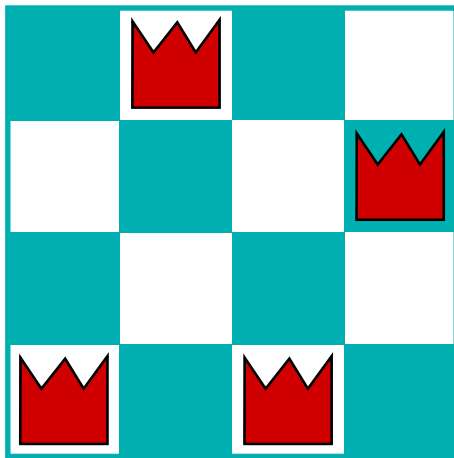
## Ejemplo: 8 puzzle



## Ejemplo: 8 puzzle

- Espacio de estados: Configuraciones de 8 fichas en el tablero
- Estado inicial: Cualquier configuración
- Estado final: Fichas en orden específico
- Operadores: Mover hueco
  - Condiciones: El movimiento está dentro del tablero
  - Transformación: Intercambio entre el hueco y la ficha en la posición del movimiento
- Solución: Qué pasos + El menor número

## Ejemplo: N reinas



## Ejemplo: N reinas

- Espacio de estados: Configuraciones de 0 a n reinas en el tablero con sólo una por fila y columna
- Estado inicial: Configuración sin reinas en el tablero
- Estado final: Configuración en la que ninguna reina se mata entre si
- Operadores: Colocar una reina en una fila y columna
  - Condiciones: La reina no es matada por ninguna ya colocada
  - Transformación: Colocar una reina mas en el tablero en una fila y columna determinada
- Solución: Una solución, pero no nos importan los pasos

# Búsqueda en el espacio de estados

- La resolución de un problema con esta representación pasa por explorar el espacio de estados
- Partimos del estado inicial evaluando cada paso hasta encontrar un estado final
- En el caso peor exploraremos todos los posibles caminos entre el estado inicial del problema hasta llegar al estado final

# Estructura del espacio de estados

- Primero definiremos una representación del espacio de estados para poder implementar algoritmos que busquen soluciones
  - Estructuras de datos: Árboles y Grafos
  - Estados = Nodos
  - Operadores = Arcos entre nodos (dirigidos)
  - Árboles: Solo un camino lleva a un nodo
  - Grafos: Varios caminos pueden llevar a un nodo

# Algoritmo Básico

- El espacio de estados puede ser **infinito**
- Es necesaria una aproximación diferente par buscar y recorrer árboles y grafos (no podemos tener la estructura en memoria)
- La estructura la construimos a medida que hacemos la búsqueda



# Algoritmo Básico

---

**Función:** Búsqueda en espacio de estados()

**Datos:** El estado inicial

**Resultado:** Una solución

Seleccionar el primer estado como el estado actual

**mientras** *estado actual*  $\neq$  *estado final* **hacer**

    Generar y guardar sucesores del estado actual (expansión)

    Escoger el siguiente estado entre los pendientes (selección)

**fin**

---

- La selección del siguiente nodo determinará el tipo de búsqueda (orden de selección o expansión)
- Es necesario definir un orden entre los sucesores de un nodo (orden de generación)

# Algoritmo Básico

- **Nodos abiertos:** Estados generados pero aún no visitados
- **Nodos cerrados:** Estados visitados y que ya se han expandido
- Tendremos una estructura para almacenar los nodos abiertos
- Las diferentes políticas de inserción en la estructura determinarán el tipo de búsqueda
- Si exploramos un grafo puede ser necesario tener en cuenta los **estados repetidos** (esto significa tener una estructura para los nodos cerrados). Merece la pena si el número de nodos diferentes es pequeño respecto al número de caminos

# Características de los algoritmos

- Características:
  - **Complejidad:** ¿Encontrará una solución?
  - **Complejidad temporal:** ¿Cuanto tardará?
  - **Complejidad espacial:** ¿Cuanta memoria gastará?
  - **Optimalidad:** ¿Encontrará la solución óptima?

# Algoritmo General de Búsqueda

---

---

**Algoritmo:** Busqueda General

Est\_abiertos.insertar(Estado inicial)

Actual ← Est\_abiertos.primer()

**mientras no es\_final?(Actual) y no Est\_abiertos.vacia?()** **hacer**

    Est\_abiertos.borrar\_primer()

    Est\_cerrados.insertar(Actual)

    Hijos ← generar\_sucesores(Actual)

    Hijos ← tratar\_repetidos(Hijos, Est\_cerrados, Est\_abiertos)

    Est\_abiertos.insertar(Hijos)

    Actual ← Est\_abiertos.primer()

**fin**

---

- Variando la estructura de abiertos variamos el comportamiento del algoritmo (orden de visita de los nodos)
- La función generar\_sucesores seguirá el orden de generación de sucesores definido en el problema
- El tratamiento de repetidos dependerá de cómo se visiten los nodos

# Tipos de algoritmos

- Algoritmos de búsqueda ciega
  - No tienen en cuenta el coste de la solución en la búsqueda
  - Su funcionamiento es sistemático, siguen un orden de visitas y generación de nodos establecido por la estructura del espacio de búsqueda
  - Anchura prioritaria, Profundidad prioritaria, Profundidad iterativa
- Algoritmos de búsqueda heurística
  - Utilizan una estimación del coste de la solución para guiar la búsqueda
  - No siempre garantizan el óptimo, ni una solución
  - Hill-climbing, Branch and Bound, A\*, IDA\*

# Búsqueda en Anchura Prioritaria

- Los nodos se visitan y generan por niveles
- La estructura para los nodos abiertos es una cola (FIFO)
- Un nodo es visitado cuando todos los nodos de los niveles superiores y sus hermanos precedentes han sido visitados
- Características:
  - Completitud: El algoritmo siempre encuentra una solución
  - Complejidad temporal: Exponencial respecto al factor de ramificación y la profundidad de la solución  $O(r^p)$
  - Complejidad espacial: Exponencial respecto al factor de ramificación y la profundidad de la solución  $O(r^p)$
  - Optimalidad: La solución que se encuentra es óptima en número de niveles desde la raíz

# Búsqueda en Profundidad Prioritaria

- Los nodos se visitan y generan buscando los nodos a mayor profundidad y retrocediendo cuando no se encuentran nodos sucesores
- La estructura para los nodos abiertos es una pila (LIFO)
- Para garantizar que el algoritmo acaba debe imponerse un **límite** en la profundidad de exploración
- Características
  - Completitud: El algoritmo encuentra una solución si se impone un límite de profundidad y existe una solución dentro de ese límite
  - Complejidad temporal: Exponencial respecto al factor de ramificación y la profundidad del límite de exploración  $O(r^p)$
  - Complejidad espacial: En el caso de no controlar los nodos repetidos el coste es lineal respecto al factor de ramificación y el límite de profundidad  $O(rp)$ . Si tratamos repetidos el coste es igual que en anchura. Si la implementación es recursiva el coste es  $O(p)$
  - Optimalidad: No se garantiza que la solución sea óptima

# Búsqueda en Profundidad Limitada

---

**Procedimiento:** Búsqueda en profundidad limitada (límite: entero)

Est\_abiertos.insertar(Estado inicial)

Actual  $\leftarrow$  Est\_abiertos.primer()

**mientras no es\_final?(Actual) y no Est\_abiertos.vacia?() hacer**

    Est\_abiertos.borrar\_primer()

    Est\_cerrados.insertar(Actual)

**si** profundidad(Actual)  $\leq$  límite **entonces**

        Hijos  $\leftarrow$  generar\_sucesores (Actual)

        Hijos  $\leftarrow$  tratar\_repetidos (Hijos, Est\_cerrados, Est\_abiertos)

        Est\_abiertos.insertar(Hijos)

**fin**

    Actual  $\leftarrow$  Est\_abiertos.primer()

**fin**

---

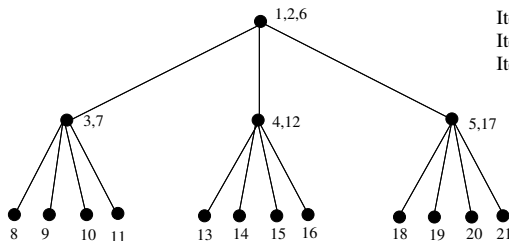
- La estructura de abiertos es ahora una pila
- Se dejan de generar sucesores cuando se llega al límite de profundidad
- Esta modificación garantiza que el algoritmo acaba
- Si tratamos repetidos el ahorro en espacio es nulo



# ID (iterative deepening): profundidad iterativa

- Intenta combinar el comportamiento espacial del DFS con la optimalidad del BFS
- El algoritmo consiste en realizar **búsquedas en profundidad sucesivas** con un nivel de profundidad máximo acotado y creciente en cada iteración
- Así se consigue el comportamiento de BFS pero sin su coste espacial, ya que la exploración es en profundidad, y además los nodos se regeneran a cada iteración
- Además esto permite evitar los casos en que DFS no acaba (existen ramas infinitas)
- En la primera iteración la profundidad máxima será 1 y este valor irá aumentando en sucesivas iteraciones hasta llegar a la solución
- Para garantizar que el algoritmo acaba si no hay solución, se puede definir una cota máxima de profundidad en la exploración

# ID (iterative deepening)



Iteracion 1: 1  
Iteracion 2: 2,3,4,5  
Iteracion 3: 6,7,8,9,...21

# Búsqueda en profundidad iterativa

---

---

**Procedimiento:** Búsqueda en profundidad iterativa (límite: entero)

prof  $\leftarrow$  1

Actual  $\leftarrow$  Estado inicial

**mientras no** *es\_final?(Actual)* **y** *prof* < *límite* **hacer**

    Est\_abiertos.inicializar()

    Est\_abiertos.insertar(Estado inicial)

    Actual  $\leftarrow$  Est\_abiertos.primer()

**mientras no** *es\_final?(Actual)* **y no** *Est\_abiertos.vacia?()* **hacer**

        Est\_abiertos.borrar\_primer()

        Est\_cerrados.insertar(Actual)

**si** *profundidad(Actual)*  $\leq$  *prof* **entonces**

            Hijos  $\leftarrow$  generar\_sucesores (Actual)

            Hijos  $\leftarrow$  tratar\_repetidos (Hijos, Est\_cerrados, Est\_abiertos)

            Est\_abiertos.insertar(Hijos)

**fin**

        Actual  $\leftarrow$  Est\_abiertos.primer()

**fin**

    prof  $\leftarrow$  prof+1

**fin**

---

# Profundidad Iterativa

- Compleitud: El algoritmo siempre encontrará la solución
- Complejidad temporal: La misma que la búsqueda en anchura. El regenerar el árbol en cada iteración solo añade un factor constante a la función de coste  $O(r^p)$
- Complejidad espacial: Igual que en la búsqueda en profundidad
- Optimalidad: La solución es óptima igual que en la búsqueda en anchura