Tema 2.
Resolución de problemas mediante búsqueda

Búsqueda ciega

Lecturas :

- CAPÍTULO 3 de Russell & Norvig
- CAPÍTULOS 7 y 8 de Nilsson

Algunas figuras tomadas de http://aima.cs.berkeley.edu/

Búsqueda en el espacio de estados

Solucionar un problema mediante búsqueda:

Formulación + búsqueda + ejecución

- **Formulación del problema:**
  - Definir **estados**
  - Especificar el **estado inicial**
  - Especificar las **acciones** que puede realizar el agente
    - Reglas para las acciones permitidas.
    - Función sucesor:

      Estado actual → Lista de estados directamente accesibles.
  - Definir los estados **objetivo**
    - Definición extensiva: Lista
    - Definición intensiva: **Test de objetivo**
  - Definir **utilidad**: Función de coste del camino.

Camino: Secuencia de estados conectados por acciones.

Espacio de estados: Conjunto de estados accesibles desde el estado inicial

Se representa mediante un grafo conexo cuyos nodos ≡ estados, arcos ≡ acciones.

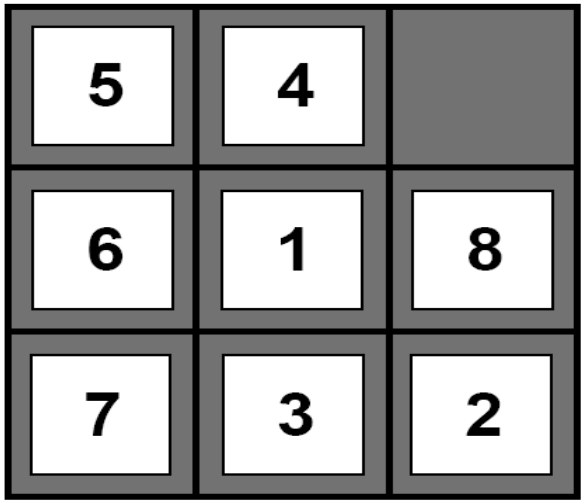
Solución: Camino desde el estado inicial al estado(s) objetivo(s).

Solución óptima: Solución con el mínimo coste.

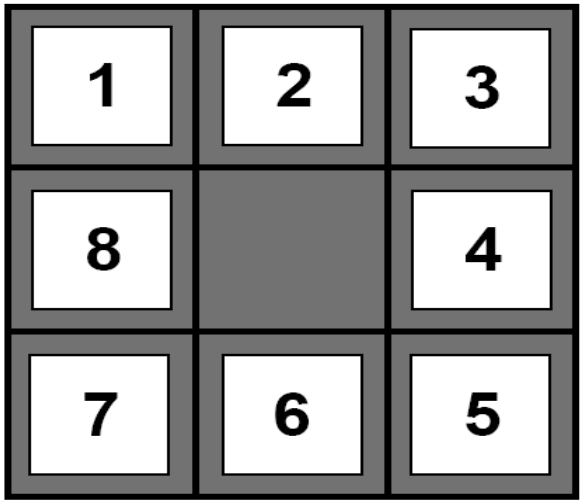
Problemas sencillos, I

8-puzzle.

- **Estados:** Disposiciones de 8 casillas numeradas de 1 a 8 + casilla vacía en un tablero de 3 x 3.
- **Estado inicial:** Una disposición dada (arbitraria).
- **Acciones:** Desplazar una pieza adyacente a la casilla vacía, a esa casilla.
  El hueco estará ahora en donde estaba la pieza que se ha movido.
- **Estado objetivo: Una disposición ordenada, con la casilla vacía en el medio.**
- **Utilidad**: Coste unidad por movimiento



Estado inicial

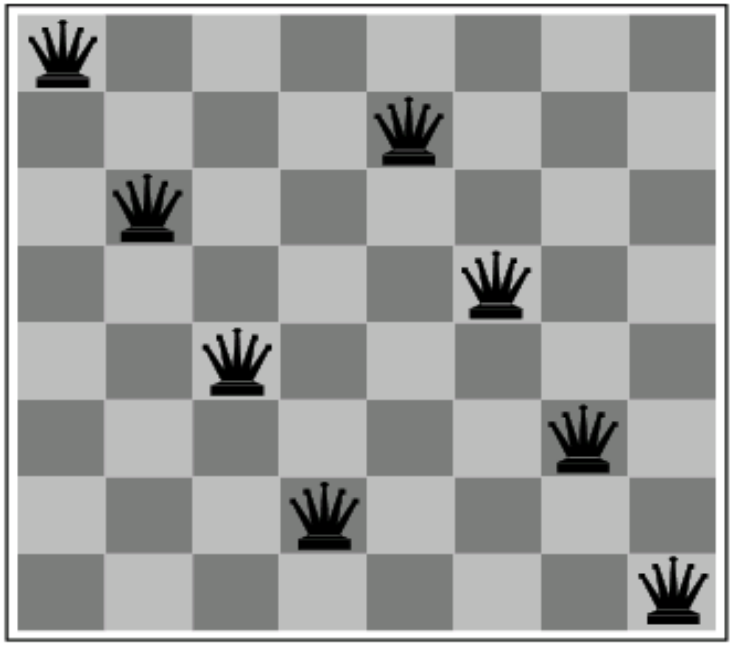


Estado objetivo

Problemas sencillos, II

Problema de las N reinas (formulación con estados completos)

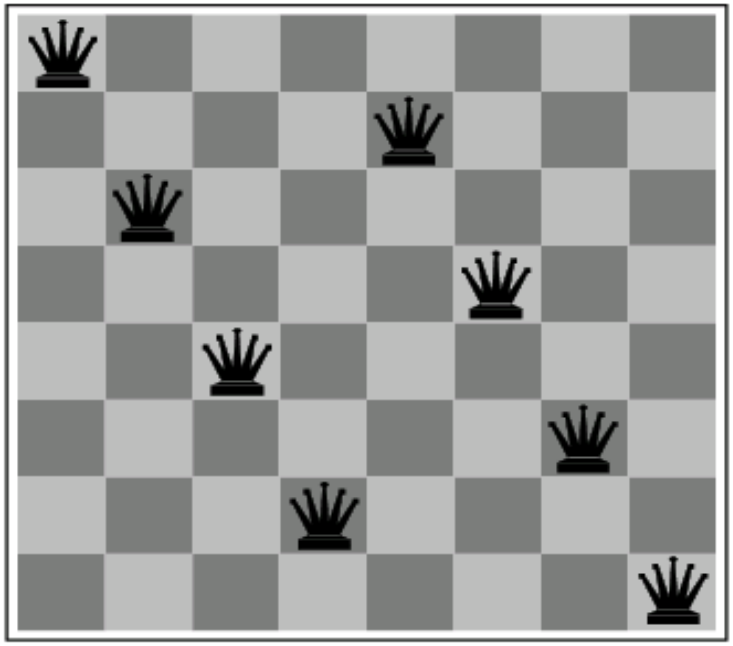
- **Estados:** Disposiciones de N reinas en un tablero de ajedrez de N x N casillas.
- **Estado inicial:** Una disposición dada (arbitraria) de las N reinas en el tablero.
- **Acciones:** Mover una reina a una casilla vacía.
- **Estado final:** N reinas que no se atacan entre ellas.
- **Utilidad**: Sólo es importante el estado final.



Problemas sencillos, III

Problema de las N reinas (formulación incremental)

- **Estados:** Disposiciones de n = 1,2, 3,… N reinas en un tablero de ajedrez de N x N.
- **Estado inicial:** Un tablero vacío.
- **Acciones:** A partir de un estado con n reinas, situar una reina nueva en una casilla vacía en la columna vacía más a la izquierda, que no ataque a las reinas previamente situadas.
- **Estado final:** N reinas que no se ataquen entre sí.
- **Utilidad**: Todos los caminos tienen igual longitud (N pasos) e igual coste.



Problemas sencillos, IV

Criptoaritmética.

- **Estados:** Letras + dígitos.
- **Estado inicial:** Todas las letras.
- **Acciones:** Sustituir una letra por un dígito no usado.
- **Estado final:** Todas las letras se sustituyen por dígitos de tal forma que la aritmética es correcta.
- **Utilidad**: Todos los caminos tienen igual longitud (10 pasos) e igual coste

```
    FORTY              29786
+     TEN          +     850
    + TEN              + 850
  ------             ------
  SIXTY              31486
```

Problemas reales I

- Buscador de rutas (por ej. viaje en avión)
  - **Estados:** Aeropuerto + hora de llegada.
  - **Estado inicial:** Aeropuerto de salida + hora de partida
  - **Acciones:** Estados resultantes de tomar uno de los vuelos programados disponibles desde el aeropuerto actual, saliendo más tarde que la hora actual + tiempo de tránsito del aeropuerto.
  - **Estado final:** Aeropuerto destino + hora de llegada deseada.
  - **Utilidad**: Coste económico, tiempo total de viaje, número de transbordos, confortabilidad, etc.

- Problemas de planificación de rutas:
  - **PVC** : Problema del viajante de comercio.

    Problema de diseño óptimo de tours: Encontrar un recorrido a través de N ciudades dadas en el que cada ciudad se visite sólo una vez, minimizando la longitud del camino.
    - **Estados:** N ciudades.
    - **Estado inicial:** Ciudad de partida.
    - **Acciones:** Viajar a una ciudad aún no visitada que sea directamente accesible por carretera desde la ciudad actual.
    - **Estado final:** Ciudad de partida, habiendo visitado exactamente una vez cada una de las otras ciudades
    - **Utilidad**: Distancia total recorrida.

Problemas reales II

- Diseño de distribución de componentes en chips VLSI

  Posicionar componentes y conexiones en un chip para:

  - Minimizar: área, latencias, capacitancias no deseadas, generación de calor, costes.
  - Maximizar: nivel de producción.
- Navegación de robots
- Problemas de ensamblaje secuencial en fábricas (razonamiento espacial)
- Diseño de proteínas (predecir la estructura terciaria a partir de la estructura primaria).
- Búsqueda en Internet

Árbol de búsqueda, I

- En casi todos los casos, la búsqueda es en un **grafo de búsqueda** (puede haber múltiples caminos desde el nodo inicial a un nodo dado).
- Enfoquémonos en búsqueda en un **árbol** (un sólo camino desde el nodo raíz a un nodo dado)
  - Los **nodos de un árbol de búsqueda** corresponden a estados de búsqueda.
  - El **nodo raíz** de un árbol de búsqueda corresponde al estado de búsqueda inicial.
  - **Acciones**: **Expandir** el nodo de búsqueda actual: Generar nodos hijo (correspondientes a nuevos estados) aplicando la función **sucesor** al nodo actual.
  - **Estado objetivo:** Un nodo correspondiente a un estado que satisface el **test de objetivo**.
  - **Utilidad**: Coste del camino desde el nodo raíz al nodo actual.

Terminología:

- **Nodo padre**: Nodo del árbol desde el cual se ha generado el nodo actual aplicando una sola vez la función sucesor.
- **Nodo ancestro**: Un nodo en el árbol desde el cual el nodo actual ha sido generado aplicando una o varias veces la función sucesor.
- **Profundidad**: Longitud del camino desde la raíz al nodo actual.
- **Nodo hoja**: Nodo generado que no se ha expandido aún
- **Frontera**: Conjunto formado por los nodos hoja.

Árbol de búsqueda, II

- Pseudocódigo para búsqueda en árbol

  ***problema* = {nodo-raíz, *expandir*, *test-objetivo*}**

  estrategia

function búsqueda-en-árbol (*problema*, e*strategia*)
;; devuelve *solución* o *fallo*
;; *lista-abierta* contiene los nodos de la frontera del árbol de búsqueda
Inicializar *árbol-de-búsqueda* con *nodo-raíz*
Inicializar *lista-abierta* con *nodo-raíz*
Iterar
  If (*lista-abierta* está vacía) **then return** *fallo*
  Elegir de *lista-abierta*, de acuerdo a la *estrategia*, un *nodo* a expandir.
    If (*nodo* satisface *test-objetivo*)
      then return *solución* (camino desde el *nodo-raíz* hasta el *nodo actual*)
      else eliminar *nodo* de *lista-abierta*
        expandir nodo
        añadir nodos hijo a *lista-abierta*

Estrategias de búsqueda

- **Búsqueda no informada (ciega):**
  - Se usa en la búsqueda solamente **la definición del problema**
  - Las distintas estrategias de búsqueda difieren en el **orden** en el que los nodos se van expandiendo.
    - Búsqueda primero en anchura.
    - Búsqueda de coste uniforme.
    - Búsqueda primero-en-profundidad.
    - Búsqueda de profundidad limitada.
    - Búsqueda primero en profundidad con profundidad iterativa
    - Búsqueda bidireccional.

- **Búsqueda informada (heurística)**
  - Usa **heurística** (estimaciones de cómo de lejos está un estado dado del estado objetivo) para guiar la búsqueda.

Rendimiento

Rendimiento resolviendo problemas

- Completitud.
- Optimalidad.
- Costes:
  - Coste del camino: dado por la función de utilidad.
  - Coste de la búsqueda
    - Complejidad temporal.
    - Complejidad espacial (uso de memoria).

Coste de la búsqueda (análisis de complejidad):

- Factor de ramificación (***b***): número máximo de sucesores de cualquier nodo.
- Profundidad del nodo objetivo más superficial (***d***)
- Profundidad máxima del árbol de búsqueda (***m***)
- Coste mínimo de una acción (ε)

- Hipótesis:
  - Factor de ramificación (*b*) finito.
  - Desde el punto de vista del análisis de coste, todas las operaciones tienen el mismo coste
  - La profundidad máxima del árbol de búsqueda (*m*) puede ser infinita.
  - Coste del camino = suma de costes de cada paso (que no son negativos).

Búsqueda primero-en-anchura

Expandir todos los nodos en una **profundidad** dada antes de expandir nodos en una profundidad mayor.

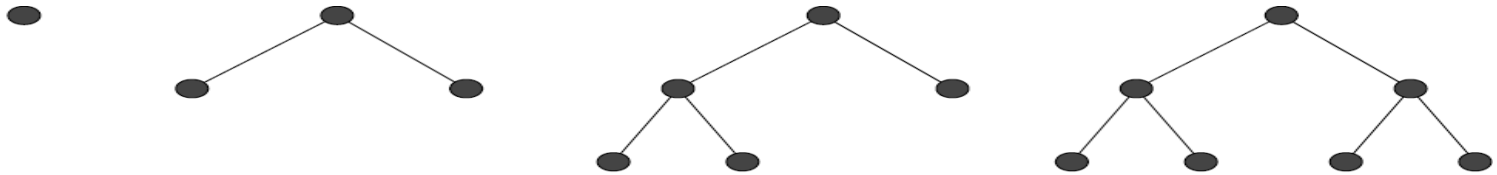
- Implementación:
  - Usar búsqueda-en-árbol utilizando una cola FIFO (first-in-first-out) para la lista de candidatos a expandir (*lista-abierta*).

    Los nodos nuevos generados se meten al final de la cola ⇒ los nodos más superficiales se expanden antes que los más profundos.
- Rendimiento:
  - **Completo**: Siempre encuentra el nodo objetivo que está a una profundidad menor.
  - Óptimo sólo si el coste del camino es una función no decreciente de la profundidad (por ej., todas las acciones tienen igual coste)
  - Complejidad temporal y espacial: Exponencial en *b*
    - Número máximo de nodos **expandidos:**

$$1 + b + b^2 + \ldots + b^d - 1 = \frac{b^{d+1} - b}{b - 1} = O(b^d)$$

    - Número máximo de nodos **generados:**

$$b + b^2 + \ldots + b^d + \left(b^{d+1} - b\right) = \frac{b^{d+2} - b^2}{b - 1} = O(b^{d+1})$$



Búsqueda de coste uniforme

Expandir nodo con el **coste de camino más bajo**

- Implementación:
  - Ordenar la lista de candidatos a expandir de acuerdo a sus costes de camino.

    Se expande primero el nodo con menor coste

- Rendimiento:

  La búsqueda puede entrar en un bucle infinito si hay un bucle con coste cero en el espacio de estados.
  - **Completo y óptimo**: Siempre encuentra el nodo objetivo con el menor coste de camino [C*] sii el coste de cada paso es mayor que cero (es decir, si el coste siempre crece si la longitud del camino crece).
  - **Complejidad**:

    El caso más complejo espacialmente y temporalmente es

$$O\left(b^{\lceil C^*/\varepsilon \rceil}\right)$$

$C^* \equiv$ Coste de camino de la solución óptima

$\varepsilon \equiv$ Coste mínimo (>0) de una acción

- Si todos los pasos tienen igual coste, la búsqueda de coste uniforme es equivalente a la búsqueda primero-en-anchura.

Búsqueda primero-en-profundidad

Expandir primero los **nodos más profundos** de la frontera.

- Implementación:
  - Usar búsqueda-en-árbol con una cola LIFO (last-in-first-out), (pila) para implementar la lista de candidatos a ser expandidos

    Los nodos nuevos generados se introducen al principio de la cola ⇒ los nodos más profundos se expanden primero.
  - Alternativa: Función recursiva que se llama a sí misma por cada uno de sus hijos.

- Rendimiento:
  - **No es completa:** podría no parar en árboles que no tienen una profundidad máxima.
  - **No es óptima**
  - Complejidad:

    Asumamos que la profundidad máxima del árbol, m, es finita
    - **Complejidad temporal**: Peor caso $O(b^m)$
    - **Complejidad espacial**: Se debe llevar cuenta del camino desde nodo raíz a nodo hoja + hermanos no expandidos de cada nodo del camino, es decir, $(b \cdot m+1)$ nodos

Búsqueda con vuelta atrás (backtracking)

Variante de la búsqueda con profundidad limitada, con **uso eficiente de la memoria**

- Sólo se genera un sucesor en cada expansión. En cada nodo parcialmente expandido se recuerda cuál es el siguiente sucesor a generar.

  $\Rightarrow O(m)$ estados

- Generar sucesor modificando el estado actual (en vez de copiar + modificar). Se deben poder deshacer modificaciones cuando se vaya hacia atrás para generar los siguientes sucesores.

  $\Rightarrow$ un solo estado + $O(m)$ acciones

Búsqueda de profundidad limitada

Expandir primero los nodos más profundos del conjunto frontera, hasta una profundidad máxima (l)

- Implementación:
  - Igual que la búsqueda primero-en-profundidad, asumiendo que los nodos con profundidad **igual a** l no tienen sucesores.
- Rendimiento:
  - **No es completo** si $l < d$
  - **Optimalidad**
    - **No óptimo** si $l > d$.
    - **Óptimo** si $l = d$ y los costes de cada paso son iguales.
  - Complejidad:
    - **Complejidad temporal**: Peor caso $O(b^l)$
    - **Complejidad espacial**: $O(b \cdot l)$
- Selección de la **profundidad límite** (l)
  - Usar conocimiento específico del problema.

    Por ej., el problema de las N reinas tiene como mucho N pasos en la formulación de estado-completo, y exactamente N pasos en la formulación incremental
  - **Diámetro** del problema: Número máximo de pasos necesarios para alcanzar cualquier estado desde cualquier otro estado.

Búsqueda primero-en-profundidad con profundidad iterativa

Expandir nodos hasta una **profundidad máxima**, e ir **incrementando** esta profundidad límite.

- Implementación:
  - A menudo combinado con búsqueda limitada en profundidad $l = 0,1,2,...$
- Rendimiento: Combina ventajas de búsqueda primero-en-anchura y búsqueda primero-en-profundidad

  - **Completa** si el factor de ramificación (b) es finito.
  - **Óptima** si los costes de cada paso son iguales.
  - Complejidad:
    - **Complejidad temporal**: Número de nodos generados

      $$(d)b + (d-1)b^2 + (d-2)b^3 + \ldots + (2)b^{d-1} + (1)b^d$$

      Esto puede ser menor que la complejidad de búsqueda en anchura

      $$b + b^2 + \ldots + b^d + \left(b^{d+1} - b\right)$$

    - **Complejidad espacial**: $O(b \cdot d)$
- Es la estrategia de búsqueda ciega preferible cuando el espacio de búsqueda es grande y se desconoce el valor de d.
- **Búsqueda con alargamiento iterativo**: Limitar el coste máximo del camino e irlo incrementando, en vez de incrementar el límite a la profundidad (normalmente costoso)

Búsqueda bidireccional

Combinar búsqueda hacia delante (del estado inicial al estado final) y **búsqueda hacia atrás** (del estado final al estado inicial)

- Implementación:
  - Se encuentra la solución cuando el nodo a expandir en una búsqueda está en el **conjunto frontera** del otro árbol de búsqueda.
  - Puede usarse en combinación con cualquier estrategia de búsqueda.
- **Rendimiento**:

Asumamos que ambas búsquedas son primero-en-anchura

  - El chequeo de la pertenencia de un nodo puede realizarse en tiempo constante a través de una **tabla hash**.
  - **Completa** si el factor de ramificación (b) es finito.
  - **Óptima** si los costes de cada paso son iguales.
  - Complejidad:
    - **Complejidad temporal**: $O(b^{d/2})$
    - **Complejidad espacial**: Se debe mantener en memoria al menos un árbol de búsqueda $O(b^{d/2})$
- Dificultades:
  - La **función predecesor** debería ser computable de manera eficiente.
  - Definición **implícita** de **estados objetivo** (por ej. N reinas)

Resumen de algoritmos de búsqueda no informada

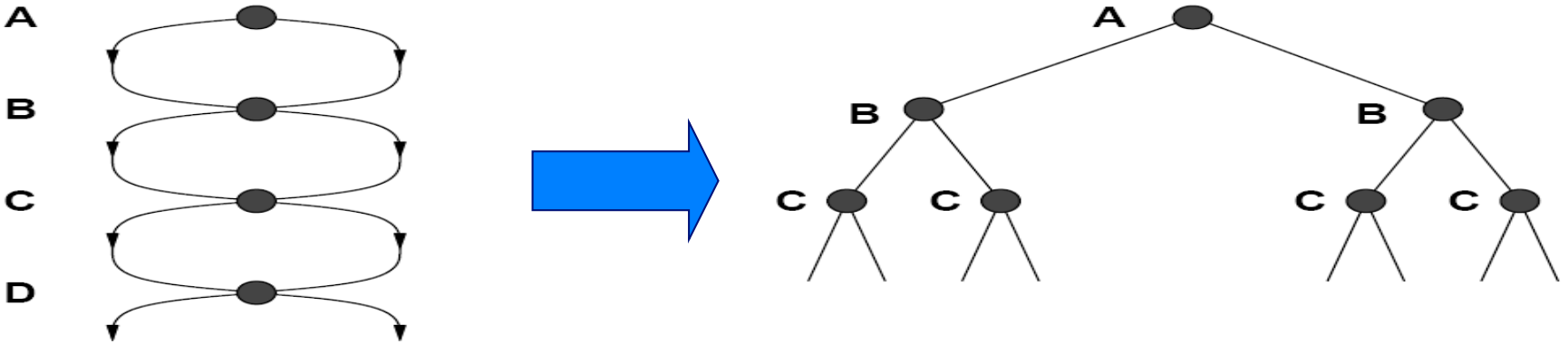
Criterio	Primero en anchura	Coste uniforme	Primero en profundidad	Profundidad limitada	Profundidad iterativa
Completo?	Sí *	Sí *	No	Sí, para $l \geq d$	Sí
Tiempo	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	b^{m}	b^{l}	b^{d}
Espacio	b^{d+1}	$b^{\lceil C^*/\varepsilon \rceil}$	$b \cdot m$	$b \cdot l$	$b \cdot d$
Óptimo?	Sí *	Sí	No	No	Si*

Estados repetidos

- Elegir una representación que evite repetición de estados.
- Si la repetición de estados no se detecta, la **complejidad** del problema de búsqueda puede llegar a ser **exponencial**
  - Por ej., búsqueda en una malla regular
  - Ejemplo trivial

Asumiendo una profundidad máxima **m**

El árbol de búsqueda tiene 2^m nodos hoja



- **Eliminación de estados repetidos:**
  Guardar los estados expandidos en *lista-cerrada*.
  - No expandir un nodo candidato si ya está en *lista-cerrada*
    - Requerimientos extra de memoria.
    - Óptimo en búsqueda con coste uniforme, y en búsqueda primero en anchura cuando el coste del paso es constante.
    - No se garantiza encontrar el camino óptimo con otras estrategias.

Búsqueda en un grafo

- Pseudocódigo para búsqueda en grafo

  problema* = {*nodo-raíz*, *expandir*, *test-objetivo*}; e*strategia

function búsqueda-en-grafo (*problema*, *estrategia*)
;; devuelve *solución* o *fallo*
;; *lista-abierta* contiene los nodos de la frontera de *árbol-de-búsqueda*
Inicializar *árbol-de-búsqueda* con *nodo-raíz*
Inicializar *lista-abierta* con *nodo-raíz*
Inicializar *lista-cerrada* a una lista vacía
Iterar
If (*lista-abierta* está vacía) **then return** *fallo*
Elegir dentro de *lista-abierta*, de acuerdo a la *estrategia*, un *nodo* a expandir.
If (*nodo* satisface *test-objetivo*)
then return *solución* (camino desde *nodo-raíz* a *nodo*)
eliminar *nodo* de *lista-abierta*
If (*nodo* no está en *lista-cerrada*)
then añadir *nodo* a *lista-cerrada*
expandir *nodo*
añadir nodos hijo a *lista-abierta*

Tipos de problema

- Problema con estados unívocamente definidos:

  Espacio **determinista**, **completamente observable** (las percepciones determinan unívocamente el estado del entorno)

- Problema conformado:

  No observable (problemas **sin sensores**).

- Problema de contingencia

  No determinista, y posiblemente parcialmente observable.

- Problema de exploración

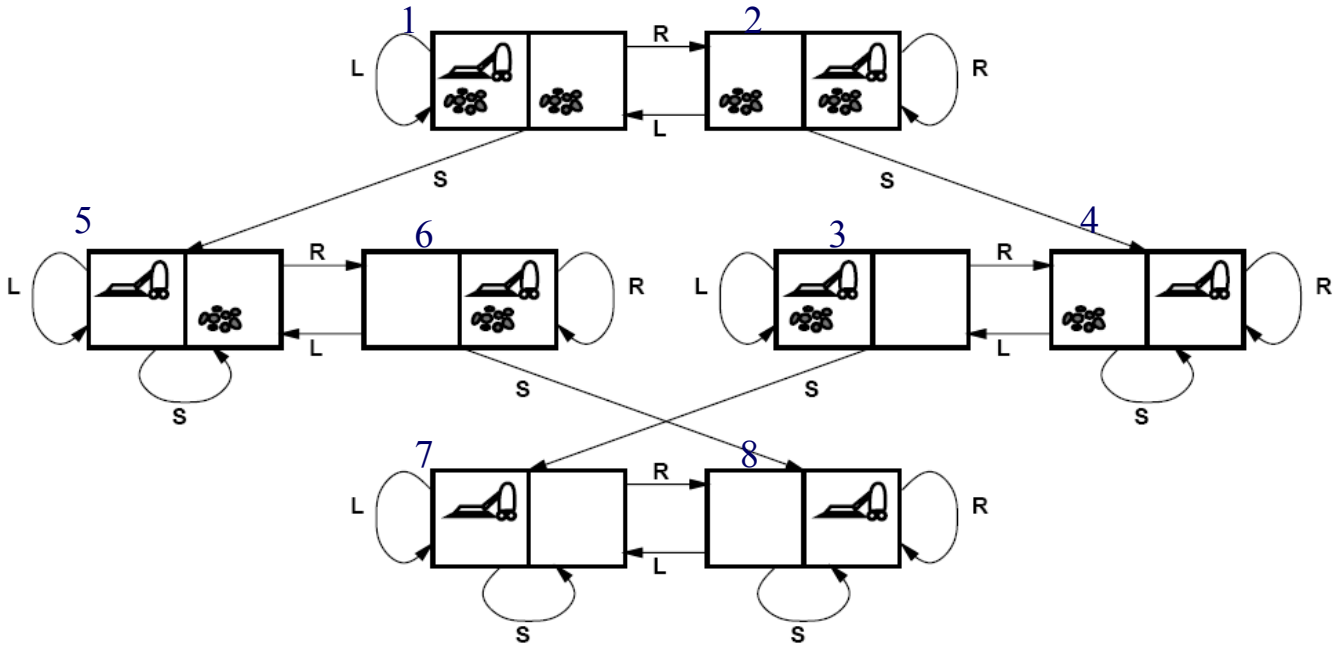
  Espacio de estados desconocido.

Problemas de **contingencia** y **exploración**

- Los algoritmos **estándar de búsqueda no son, en general, adecuados** para resolver este tipo de problemas.
- El agente puede recopilar **información** a través de sus sensores **después de actuar**.
- Proposición: **alternar búsqueda y ejecución.**

Mundo de la aspiradora

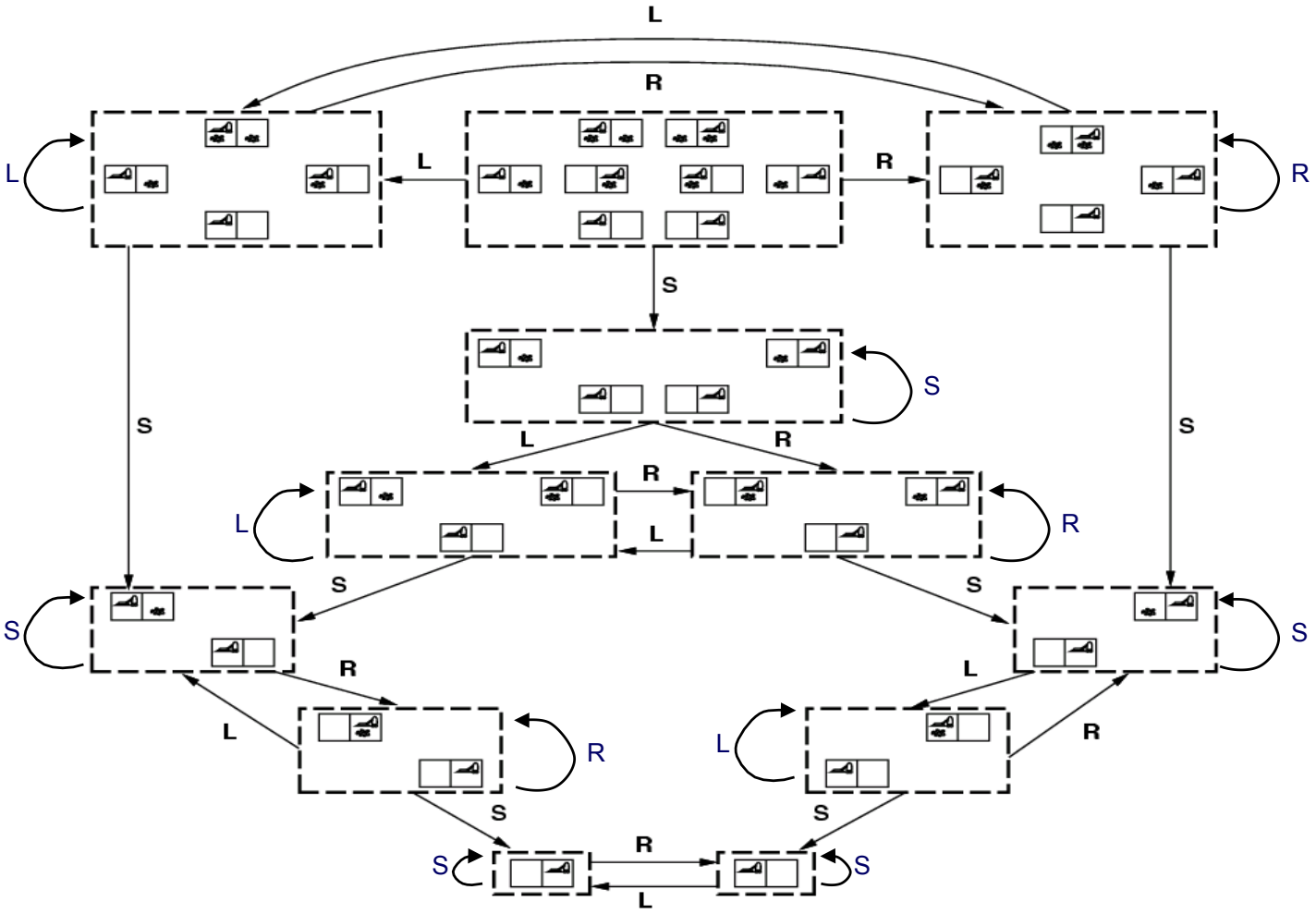
- Estados físicos:
  - Agente: Aspiradora en uno de las dos estancias adyacentes.
  - Cada estancia puede estar sucia / limpia
- Acciones:
  - L / R: mover a la izquierda / derecha
  - S: Aspirar suciedad
  - NoOp: No hacer nada
- Objetivo: Mundo limpio (estado 7 u 8)



Problema conformado

Si el agente **no tiene sensores**, entonces

- Los estados de búsqueda representan **estados de creencia: subconjuntos** de los posibles estados físicos del mundo.
- A través de una **secuencia de acciones** uno puede ser capaz de **forzar** al mundo a un estado objetivo.



Problema de contingencia

- La solución no puede ser formulada en general como una secuencia fija de acciones.
- **Plan de contingencia**: La solución puede ser dada como un árbol, donde se elige una rama u otra dependiendo de las percepciones

Ejemplo:

- Estado sencillo, se empieza en estado #5. **¿Solución?**
  [*Right*, *Suck*]
- Conformado, se empieza en {1, 2, 3, 4, 5, 6, 7, 8}
  Por ejemplo, la acción *Right* conduce al estado {2, 4, 6, 8}. **¿Solución?**
  [*Right*, *Suck*, *Left*, *Suck*]
- Contingencia, se empieza en estado #5.
  No interesa aspirar si el suelo está limpio.
  Sensores: localización, suciedad. **¿Solución?**
  [*Right*, **if** *suelo-sucio* **then** *Suck*]