INTELIGENCIA ARTIFICIAL (INF371)

CAPITULO 2: BÚSQUEDA

Dr. Edwin Villanueva Talavera

Contenido

- Estructura de Agentes
- Agentes de Resolución de Problemas
- Búsqueda de Soluciones
- Estratégias de Búsqueda sin Información
- Estratégias de Búsqueda con Información

Bibliografía:

Capitulo 2.4, 3.1, 3.2, 3.3 del libro:

Stuart Russell & Peter Norvig "Artificial Intelligence: A modern Approach", Prentice Hall, Third Edition, 2010

Estructura de Agentes

- Un agente queda completamente definido si se define la función del agente (mapeo de secuencias de percepciones a acciones)
- En la práctica es difícil especificar explícitamente la función del agente
- Al diseñar agentes se busca una forma concisa de representar la función del agente, esto es, se busca implementar un programa de agente para una arquitectura dada

Agente = arquitectura + programa

Estructura de Agentes

Agente Dirigido por Tabla

```
Function TABLE_DRIVEN_AGENT(percept) return action
    Variables estáticas:
      ■ percepts, una secuencia, inicialmente vacía
      ■ table, tabla de acciones, indexada por secuencias de
        percepciones, de inicio completamente especificada
    append percept to the end of percepts
    action ← LOOKUP(percepts, table)
    return  action
```

- Desventajas:
  - Tabla gigante (ajedrez = 10^{150} entradas)
  - Mucho tiempo para construir la tabla
  - No tiene autonomía
  - Aunque use aprendizaje automático, tardaría mucho en aprender la tabla.

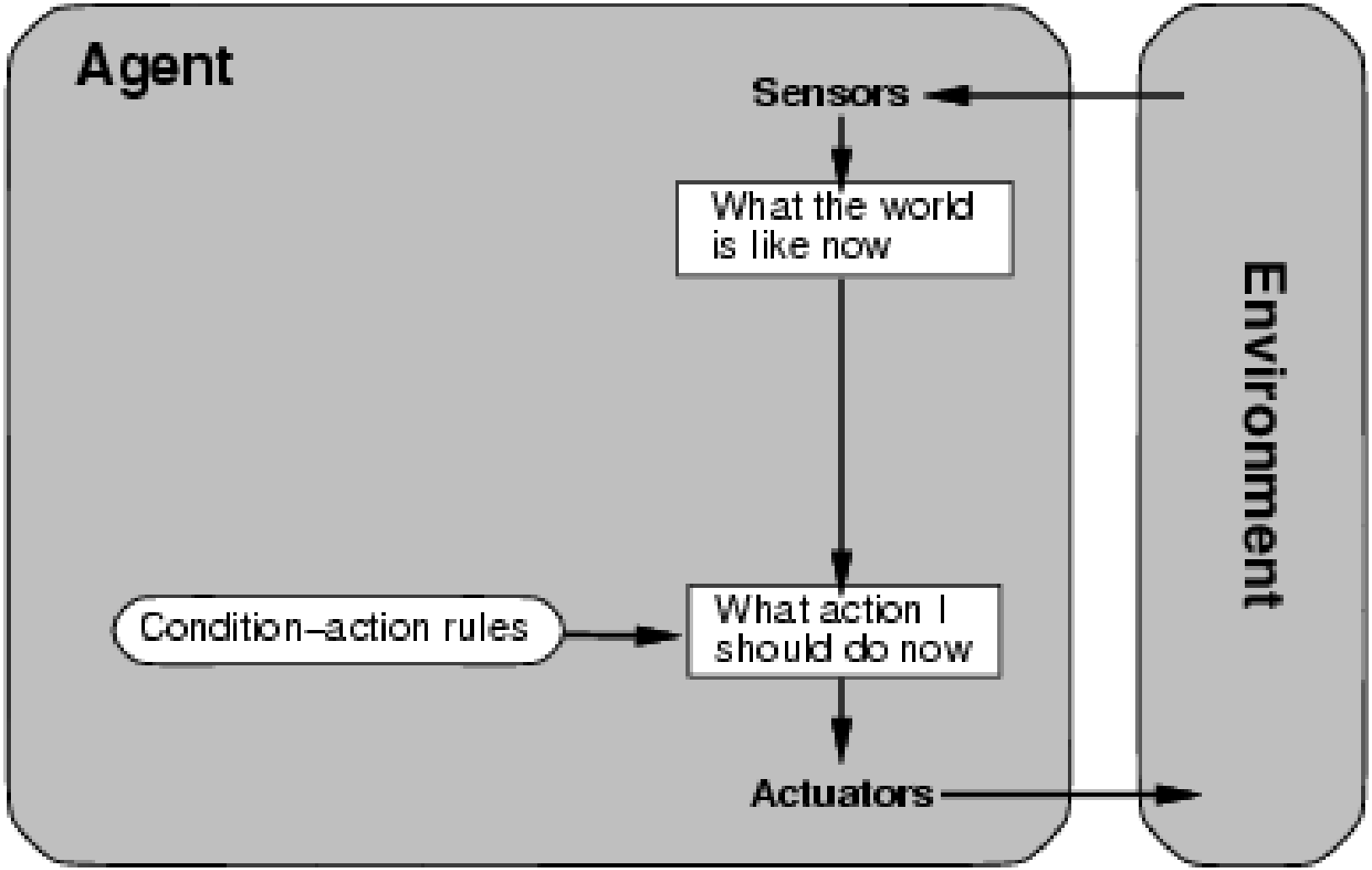
Estructura de Agentes

Tipos básicos de agentes

- Agentes reactivos simples
- Agentes reactivos basados en modelos
- Agentes basados en objetivos
- Agentes basados en utilidad

Estructura de Agentes

Agente reactivo simple



Estructura de Agentes

Programa del agente reactivo simple

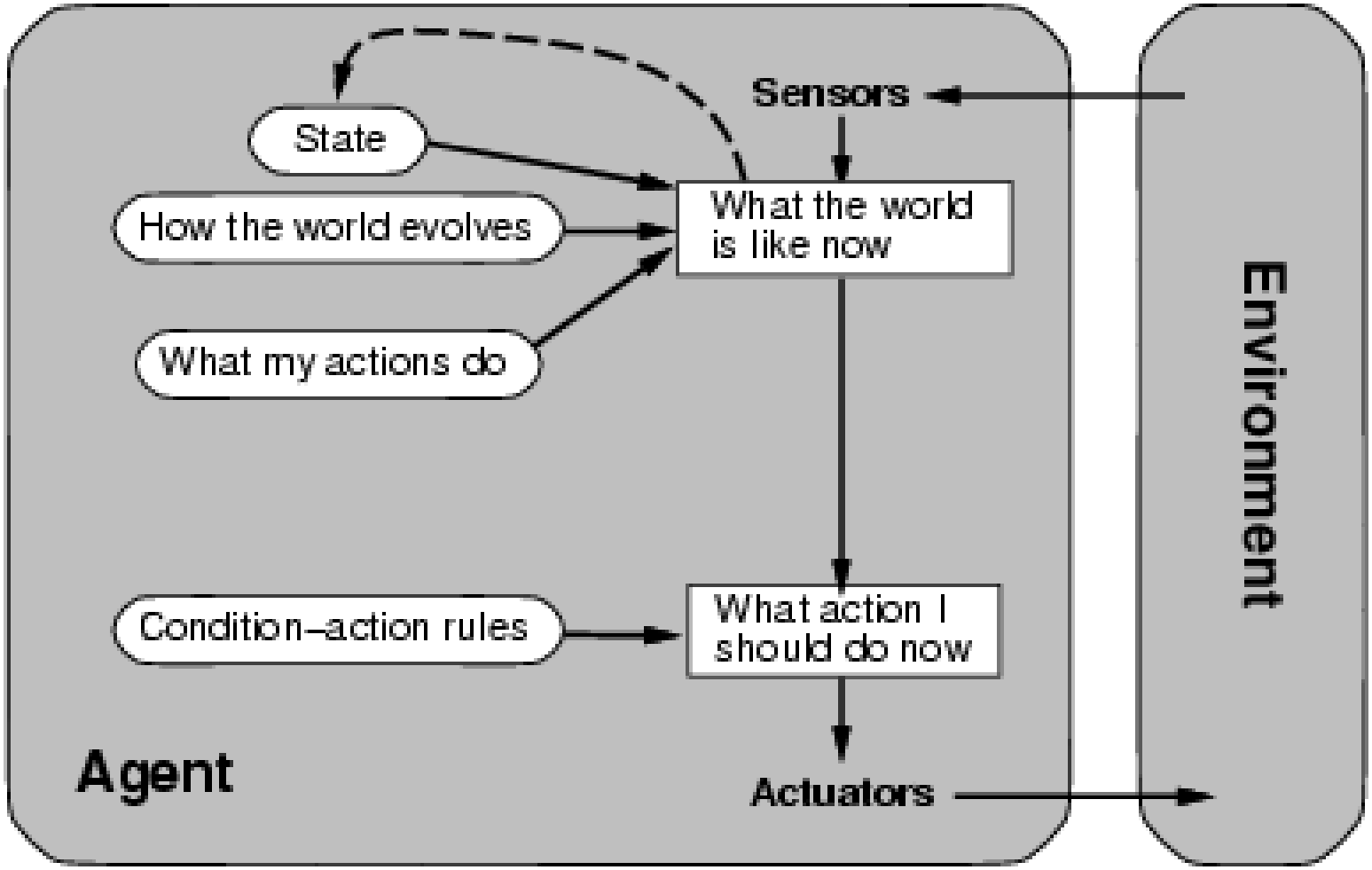
function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
  persistent: *rules*, a set of condition–action rules

  state ← INTERPRET-INPUT(*percept*)
  rule ← RULE-MATCH(*state*, *rules*)
  action ← *rule*.ACTION
  return *action*

- El agente funciona apenas si el ambiente fuese completamente observable y la decisión correcta pudiese ser tomada basada apenas en la percepción actual.

Estructura de Agentes

Agente reactivo basado en modelo



Estructura de Agentes

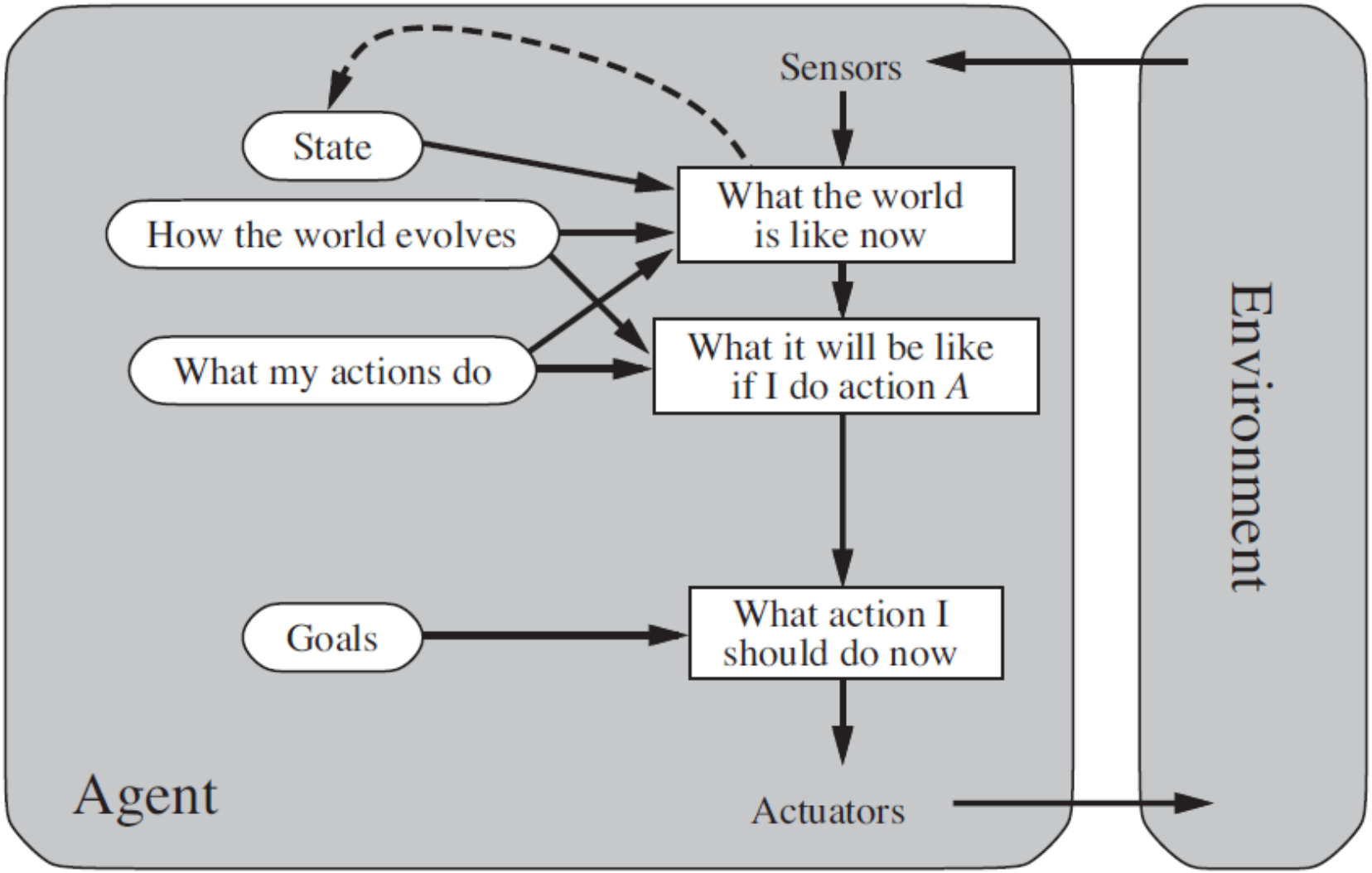
Programa del Agente Reactivo Basado en Modelo

```
function MODEL-BASED-REFLEX-AGENT(percept) returns an action
  persistent: state, the agent's current conception of the world state
              model, a description of how the next state depends on current state and action
              rules, a set of condition–action rules
              action, the most recent action, initially none

  state ← UPDATE-STATE(state, action, percept, model)
  rule ← RULE-MATCH(state, rules)
  action ← rule.ACTION
  return action
```

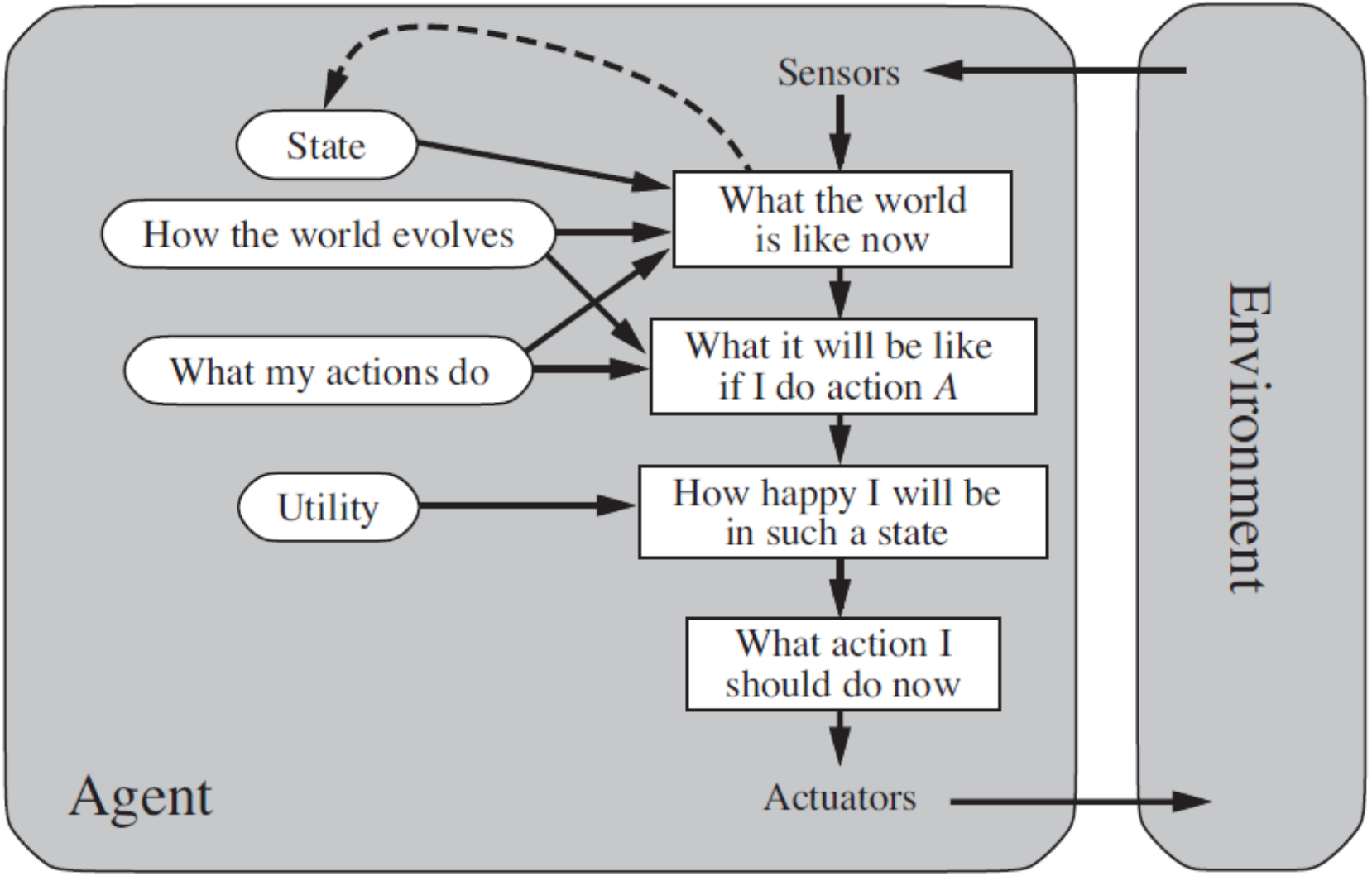
Estructura de Agentes

Agente basado en objetivo



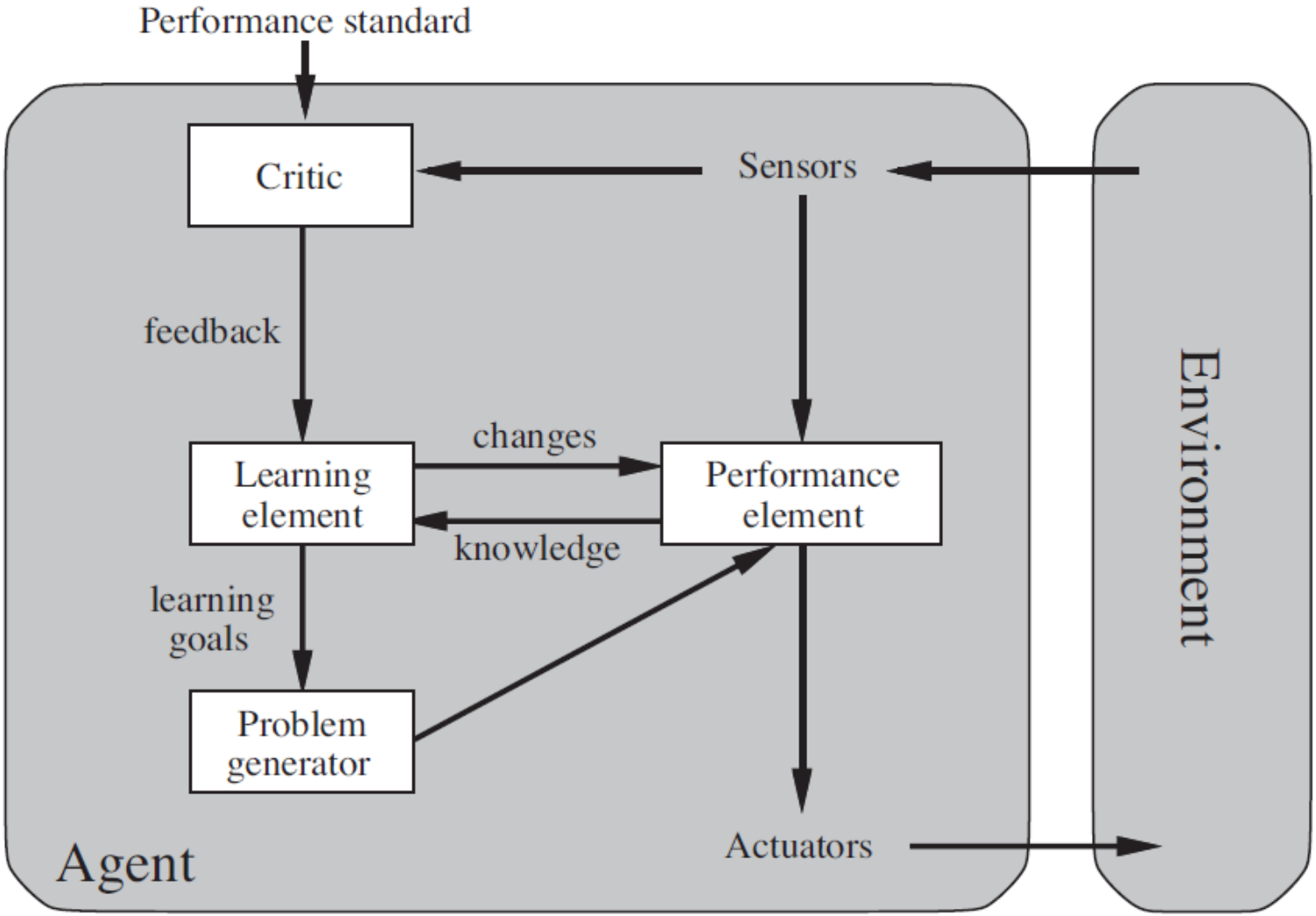
Estructura de Agentes

Agente basado en utilidad



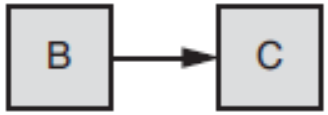
Estructura de Agentes

Agentes que aprenden

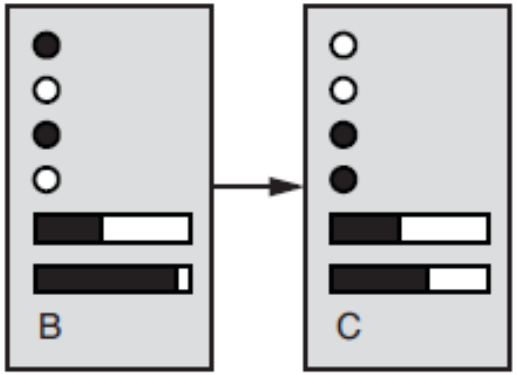


Representación de estados en agentes

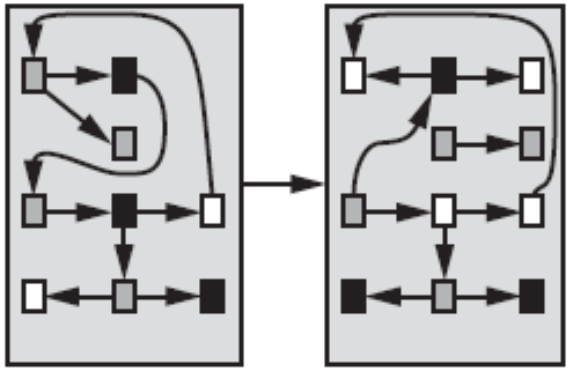
Tipos de representación de estados:



Atómica



Factorada



Estructurada

Representación de estados en agentes

Consideraciones para representación de estados en juegos:

- Representación depende del tipo de salida del juego:
  - juegos textuales -> strings,
  - Juegos de tableros -> listas o matrices de posiciones, etc.
  - Videojuejos -> representación factorada o estructurada
- Puede haber varias formas de representar estados en juegos. Por ejemplo, un juego de carreras de carros:
  - Lista de posiciones y velocidades de todos los carros (vista externa)
  - Lista de distancias y ángulos a todos los otros carros (vista interna)
- Representación es crucial para el resto del diseño del agente

Agentes de resolución de problemas

- Los agentes reactivos no funcionan en entornos para los cuales el numero de reglas condición-acción es muy grande para almacenar
- En ese caso podemos construir un tipo de agente basado en objetivo llamado de agente de resolución de problemas

Agentes de resolución de problemas

Pasos de un agente básico de resolución de problemas

- Formulación de objetivo
- Formulación de problema:
  - Estado inicial, espacio de estados, acciones, modelo de transición, costo de camino
- Búsqueda de solución:
  - encuentra una secuencia de acciones para llegar a un estado objetivo
- Ejecución de solución

Agentes de resolución de problemas

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH(problem)
      if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

La suposición es un ambiente estático, observable, discreto e determinístico.

Agentes de resolución de problemas

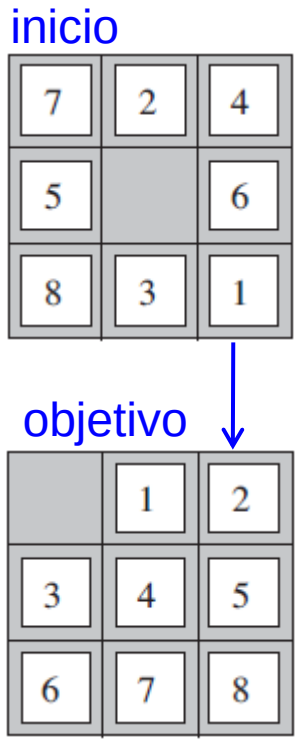
Componentes de la Formulación del Problema:

- Estados: Conjunto de situaciones diferentes que puede estar el problema
- Estado inicial: Situación del problema al inicio
- Acciones: Operaciones que pueden ser realizadas desde un determinado estado *s*, denotada comúnmente como: ACTIONS(*s*)
- Modelo de transición: estados alcanzables desde un estado dado *s* con una determinada acción *a*, comúnmente se denota: RESULT(*s*,*a*)
- Función de prueba de objetivo: Determina si un estado es la solución del problema, comúnmente se denota: GOAL-TEST(*s*)
- Costo del camino: Alguna función que mide cuan difícil o costoso es determinado camino para llegar a un nodo *s* desde el estado inicial, *g*(*s*)

Agentes de resolución de problemas

Ejemplo de Formulación de Problema: 8-puzzle

- Estados: Todas las configuraciones posibles de 8 números y un blanco
- Estado inicial: Alguna configuración dada del puzzle
- Acciones: Movimientos del casillero blanco: **Derecha, Izquierda, Arriba, Abajo**
- Modelo de transición: resultado de alguna acción, dado un estado: Ej. RESULT(inicio, Izquierda) = blanco y 5 intercambiados
- Prueba de Objetivo: Verifica si el estado es el objetivo
- Costo del camino: Cada acción cuesta 1. El costo de la solución seria el costo de todas las acciones



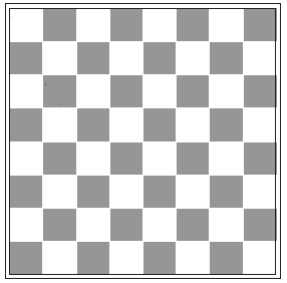
Agentes de resolución de problemas

Ejemplo de Formulación de Problema: 8-queens

- Estados: configuraciones de 0 a 8 reinas en el tablero
- Estado inicial: 0 reinas en el tablero
- Acciones: **Adicionar** una reina a un casillero vacío
- Modelo de transición: Retorna el tablero con la reina añadida
- Prueba de objetivo: Verificar que el estado tenga 8 reinas no atacadas

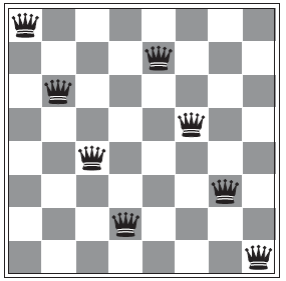
Esta formulación tiene
64x63x…x57 ~**1.8x10^{14}** posibles secuencias a investigar!

inicio



Objetivo: estado donde las 8 reinas no se ataquen

Ej.



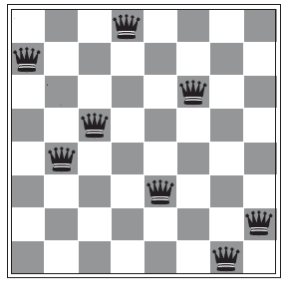
Agentes de resolución de problemas

Ejemplo de Re-formulación de Problema: 8-queens

- Estados: Vectores de 8 números no repetidos (permutaciones). Cada elemento indica la fila en que se encuentra la reina en una columna
- Estado inicial: Permutación aleatoria
- Acciones: Intercambiar 2 elementos
- Prueba de objetivo: Verificar si la nueva permutación tiene reinas no atacadas

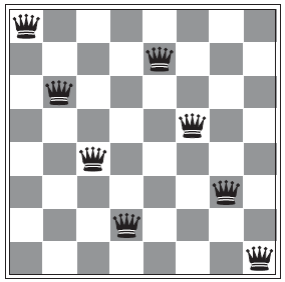
Esta formulación tiene
8x7x…x1 = **40320** posibles secuencias a investigar!

inicio



[2, 5, 4, 1, 6, 3, 8, 7]

Objetivo:
estado donde las 8 reinas no se ataquen
Ej.

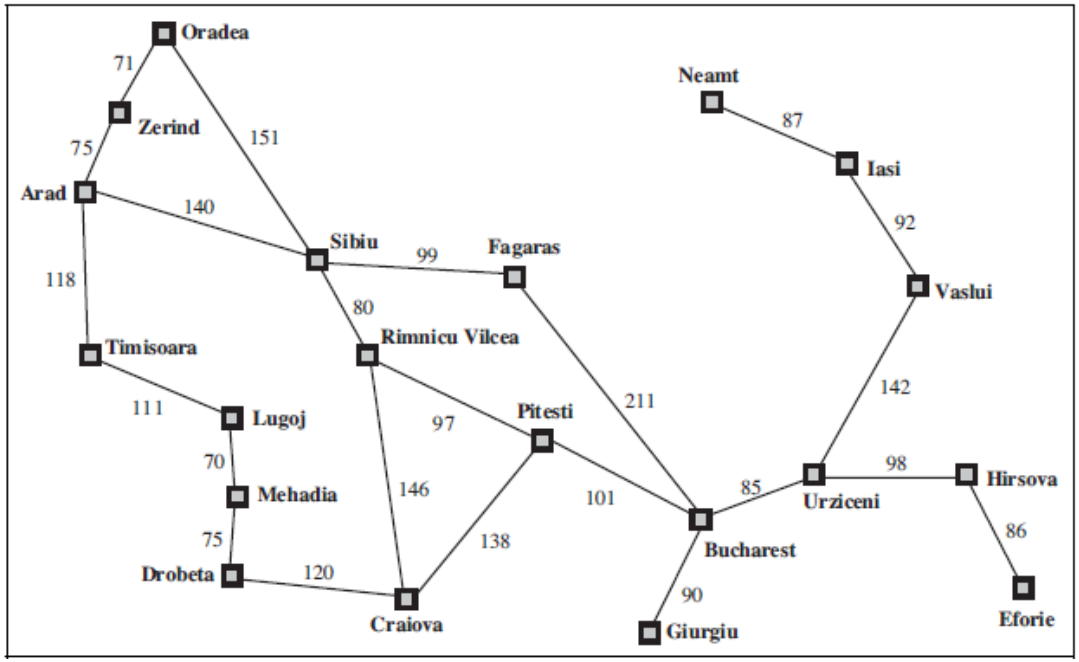


[1, 3, 5, 7, 2, 4, 6, 8]

Agentes de resolución de problemas

Ejemplo de Formulación de Problema: búsqueda de ruta en mapa

- Estados: Todas las posibles ciudades
- Estado inicial: ciudad inicial
- Acciones: Moverse a alguna ciudad vecina
- Modelo de transición: Mapa



- Prueba de Objetivo: Verificar si se llego a la ciudad deseada
- Costo del camino: Puede ser tiempo, distancia recorrida, contaminación emitida, etc

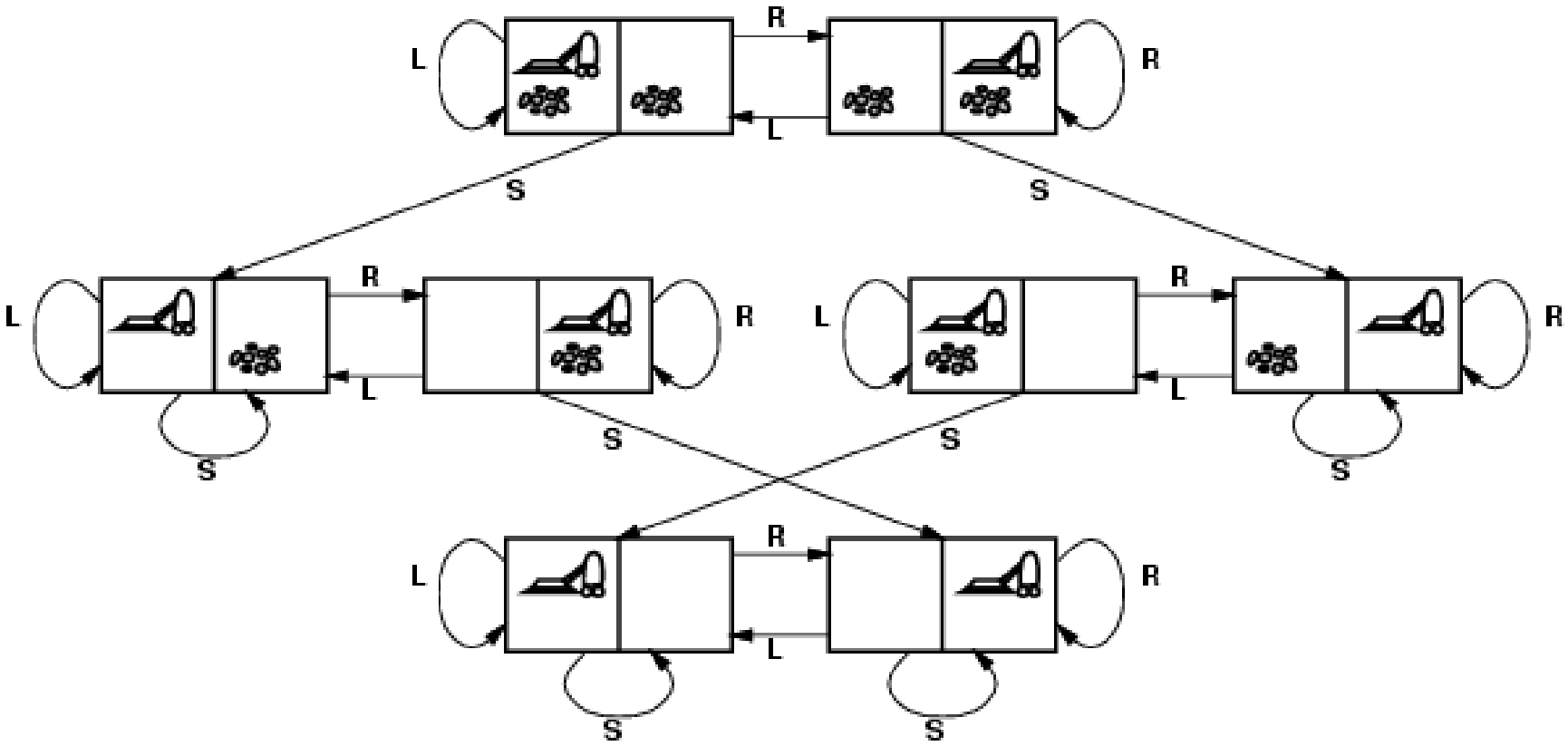
Agentes de resolución de problemas

Espacio de estados

- Conjunto de todos los estados accesibles a partir de un estado inicial
  - El modelo de transición (o **función sucesor**) determina el espacio de estados
- El espacio de estados puede ser representado con un árbol (o grafo) de búsqueda, donde los nodos representan estados y los arcos acciones

Agentes de resolución de problemas

Ejemplo de espacio de estados del mundo de la aspiradora

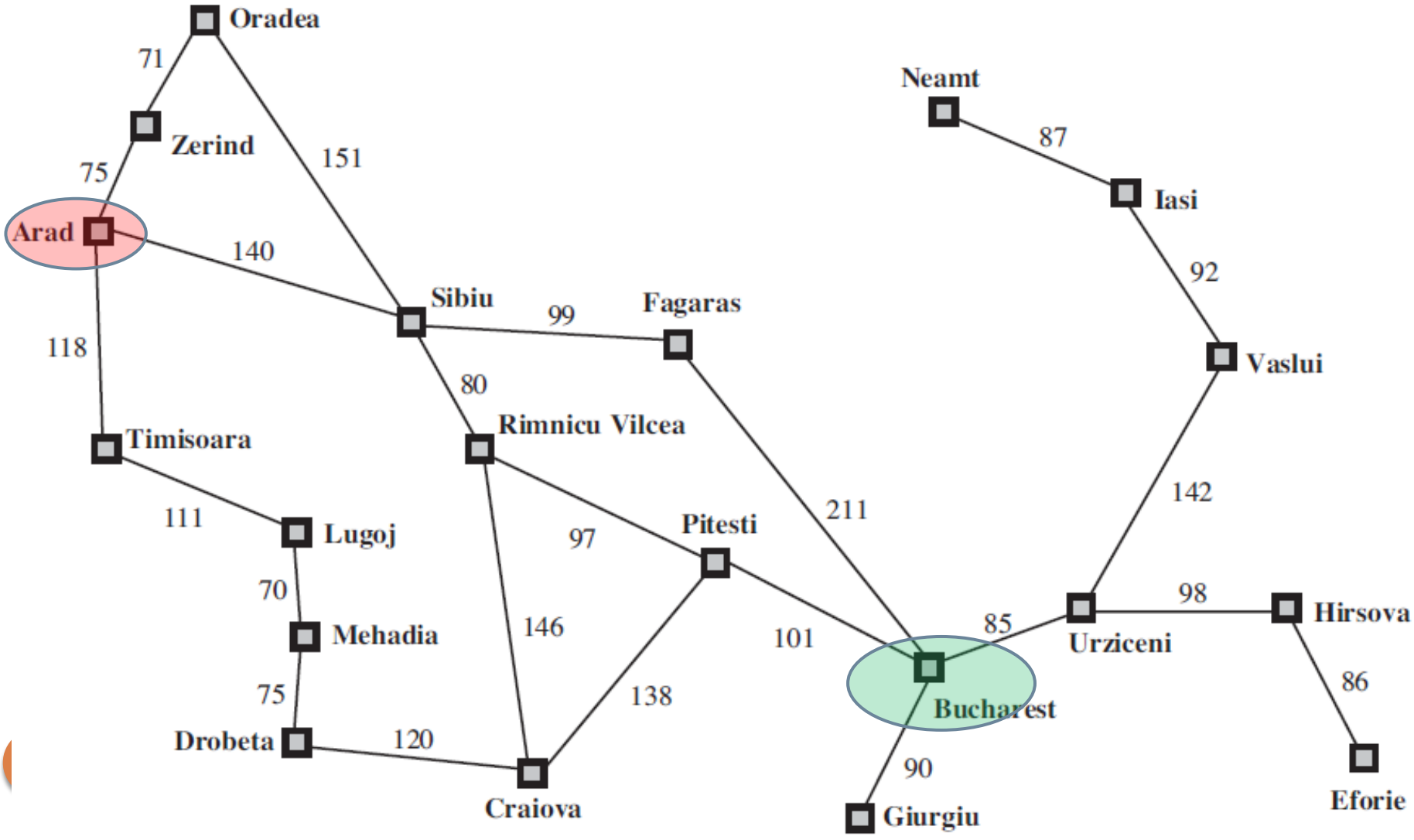


Algoritmos de Búsqueda

- La idea es explorar el espacio de estados mediante el recorrido de un árbol de búsqueda
- Expandir el estado actual aplicando la función sucesor, generando nuevos estados
- La estrategia de búsqueda determina el camino a seguir, esto es, que nodos se exploran primero y cuales se dejan para después.

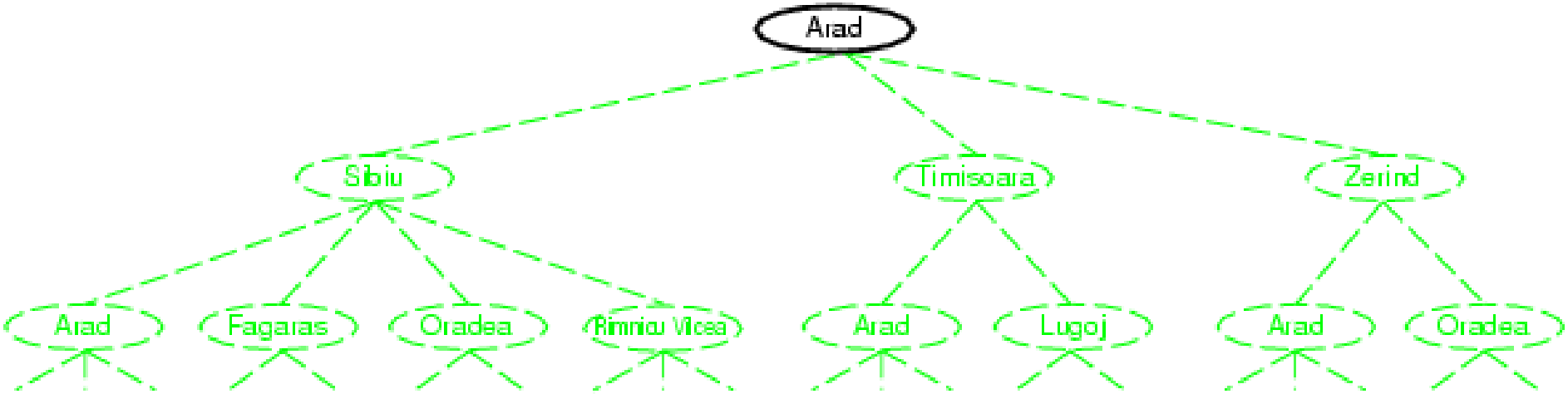
Búsqueda de Soluciones

Ejemplo: Búsqueda en el mapa de Romania



Búsqueda de Soluciones

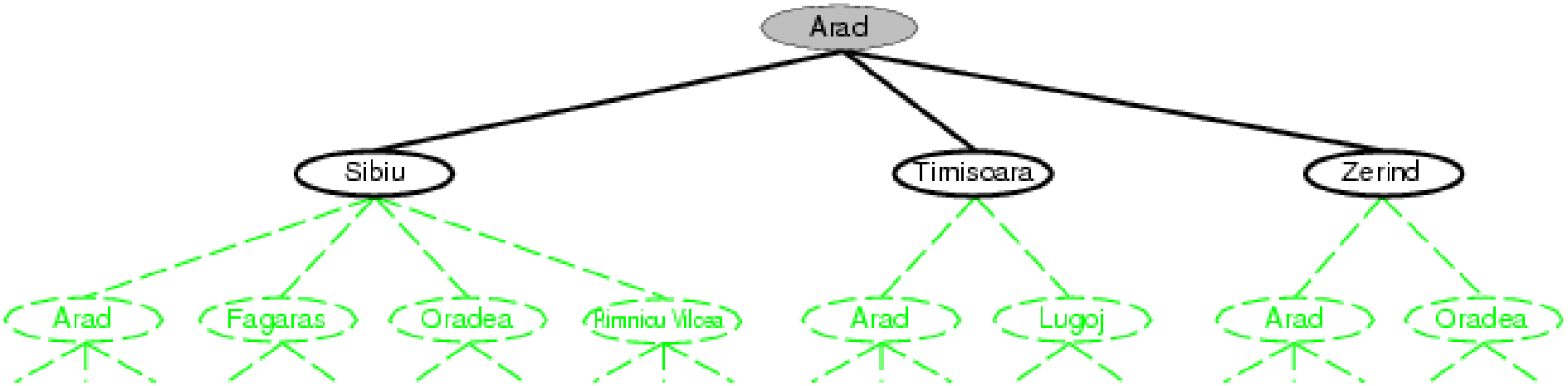
Ejemplo: Búsqueda en el mapa de Romania



Estado Inicial

Búsqueda de Soluciones

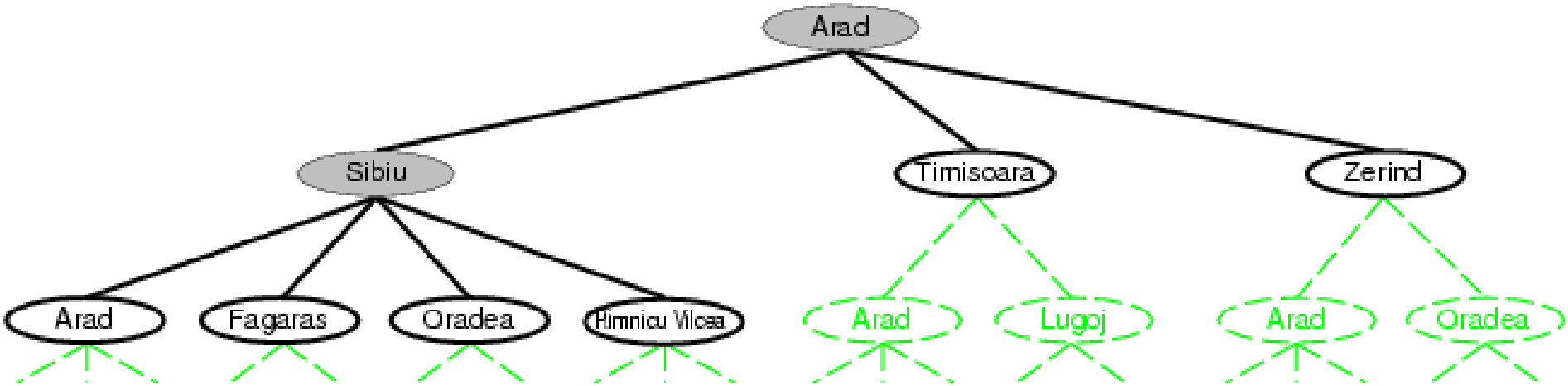
Ejemplo: Búsqueda en el mapa de Romania



Después de expandir Arad

Búsqueda de Soluciones

Ejemplo: Búsqueda en el mapa de Romania

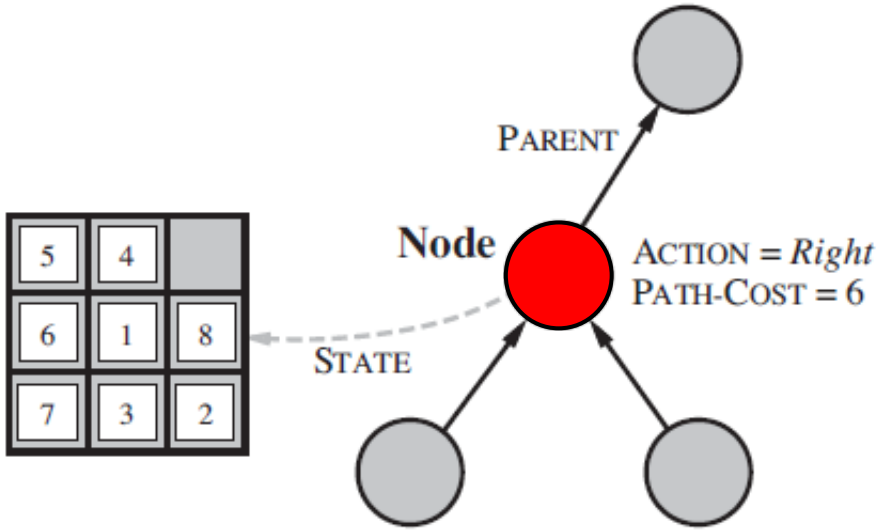


Después de expandir Sibiu

Búsqueda de Soluciones

Estructura de un Nodo:

- Debe incluir información de: estado, nodo padre, la acción que generó el nodo, costo del camino desde el nodo raíz, y profundidad del nodo



- La colección de nodos que fueron generados pero aún no expandidos es llamada de **frontera**
- La forma como colocar/sacar nodos de la frontera define la **estrategia de búsqueda**

Búsqueda de Soluciones

Generación de nodos hijos:

```
function CHILD-NODE(problem, parent, action) returns a node
  return a node with
      STATE = problem.RESULT(parent.STATE, action),
      PARENT = parent, ACTION = action,
      PATH-COST = parent.PATH-COST + problem.STEP-COST(parent.STATE, action)
```

Búsqueda de Soluciones

Estructuras de datos para implementar la frontera: ***queue***

- First-in First Out (FIFO)
- Last-in First-out (LIFO o Pila)
- Cola de Prioridad

Operaciones en la frontera:

- *EMPTY?*(*queue*): Retorna *true* si la cola esta vacía
- *POP*(*queue*): Remueve y retorna el 1er elemento de la cola
- *INSERT*(*element*, *queue*): Inserta un elemento en la cola y devuelve esta

Búsqueda de Soluciones

Algoritmo general de búsqueda en arboles

```
function TREE-SEARCH(problem) returns a solution, or failure
    initialize the frontier using the initial state of problem
    loop do
        if the frontier is empty then return failure
        choose a leaf node and remove it from the frontier
        if the node contains a goal state then return the corresponding solution
        expand the chosen node, adding the resulting nodes to the frontier
```

Búsqueda de Soluciones

Algoritmo general de búsqueda en arboles con memoria de nodos expandidos

```
function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

Estrategias de Búsqueda

Búsqueda sin información o búsqueda ciega

- Estrategias de búsqueda sin información usan solamente la información disponible en la definición del problema
  - Solo generan sucesores verificando si es estado objetivo
- Las estrategias de búsqueda sin información se distinguen por la orden en que los nodos son expandidos.
  - Búsqueda en amplitud (*Breadth-first search*)
  - Búsqueda de costo uniforme
  - Búsqueda en profundidad (*Depth-first search*)
  - Búsqueda en profundidad limitada
  - Búsqueda de profundización iterativa
  - Búsqueda bidireccional

Estrategias de Búsqueda

Evaluación de desempeño

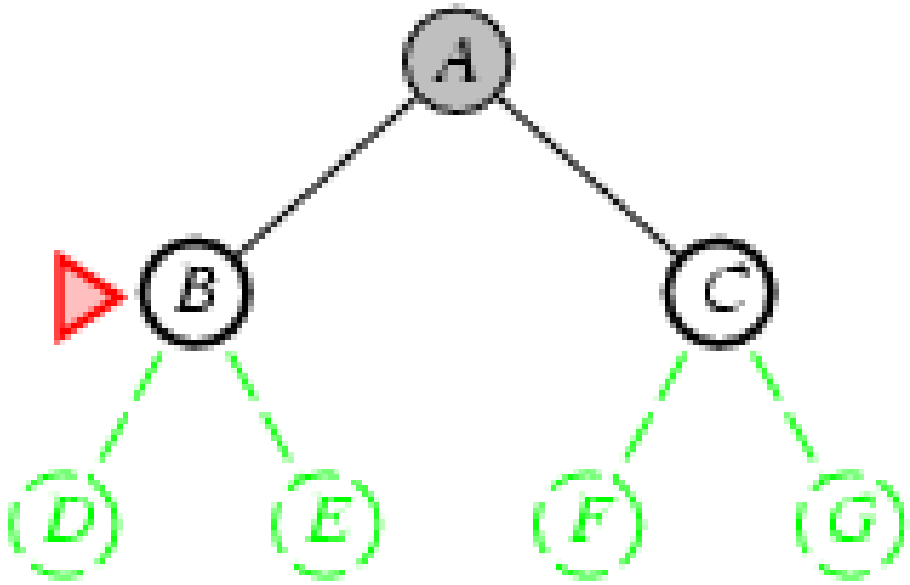
- Estrategias son evaluadas de acuerdo a los siguientes criterios
  - Completitud: el algoritmo siempre encuentra la solución?
  - Complejidad de tiempo: número de nodos generados
  - Complejidad de espacio: número máximo de nodos en memoria
  - Optimalidad: la estrategia encuentra la solución optima?
    - Una solución óptima es una solución con menor costo de camino.
- Complejidad de tiempo y espacio son medidos en función de:
  - *b*: máximo factor de ramificación del árbol (numero máximo de sucesores de cualquier nodo)
  - *d*: profundidad del nodo objetivo menos profundo
  - *m:* tamaño máximo de cualquier camino en el espacio de estados

Búsqueda en Amplitud

- Expandir el nodo aun no expandido mas cerca de la raíz
- Implementación: Puede ser TREE-SEARCH o GRAPH-SEARCH usando como frontera una cola FIFO:

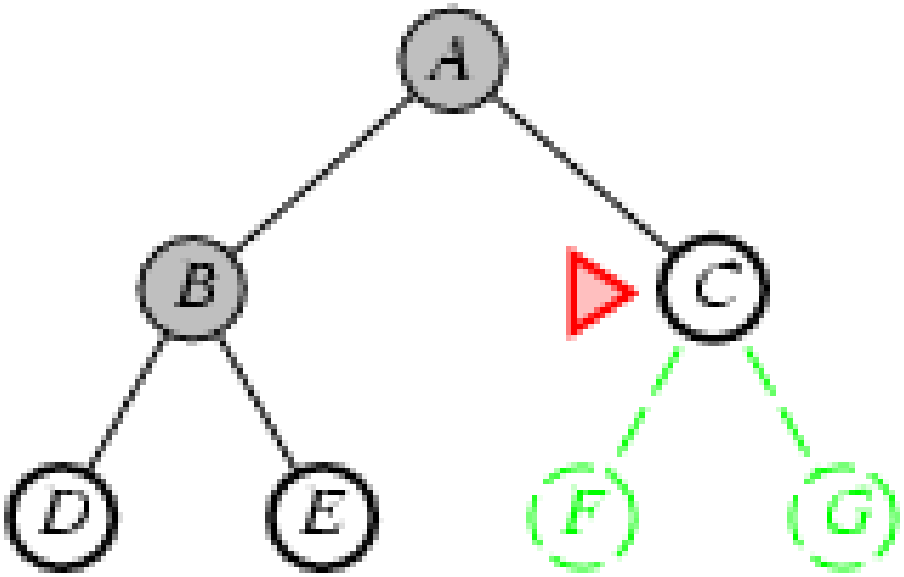
Búsqueda en Amplitud

Ejemplo de exploración de nodos en búsqueda en amplitud



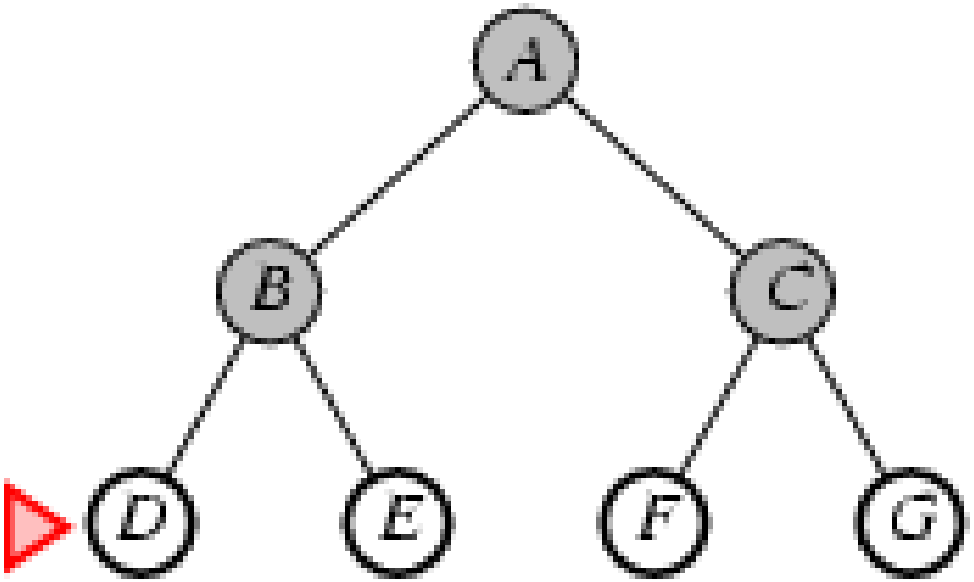
Búsqueda en Amplitud

Ejemplo de exploración de nodos en búsqueda en amplitud



Búsqueda en Amplitud

Ejemplo de exploración de nodos en búsqueda en amplitud



Búsqueda en Amplitud

Propiedades de Búsqueda en amplitud

- Completa? SI, si b es finito
- Complejidad de tiempo:

  $1+b+b^2+b^3+\ldots+b^d + b(b^d) = O(b^{d+1})$ (impl. Graph-Search)

- Complejidad de espacio:
  - Existe $O(b^{d-1})$ nodos em *explored set* y $O(b^d)$ en la frontera, asi que la complejidad espacial es dominada por la frontera: $O(b^d)$
- Optima? SI, si todos las acciones tuvieran los mismos costos

Búsqueda en Amplitud

Propiedades de Búsqueda en amplitud

- Con un factor de ramificación b=10 y suponiendo que puedan ser generados 1 millón de nodos por segundo y que cada nodo requiera 1KB de espacio, se tendría:

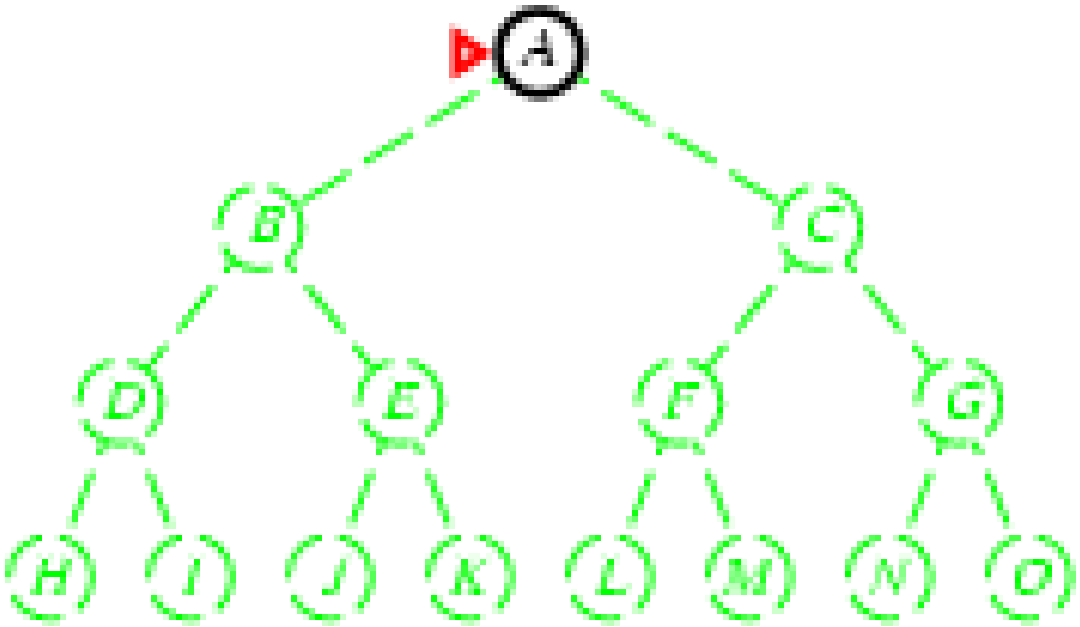
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Búsqueda en Profundidad

- Expande el nodo no expandido mas profundo
- Implementación: Puede ser GRAPH-SEARCH usando como frontera una lista LIFO (last-in, first-out), también conocida como pila:

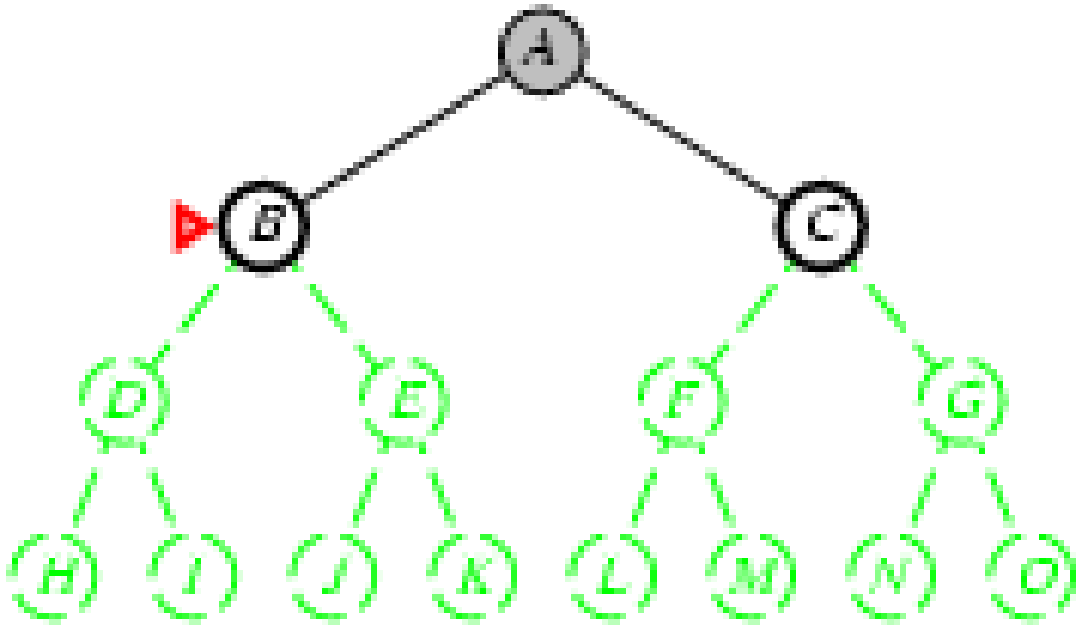
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



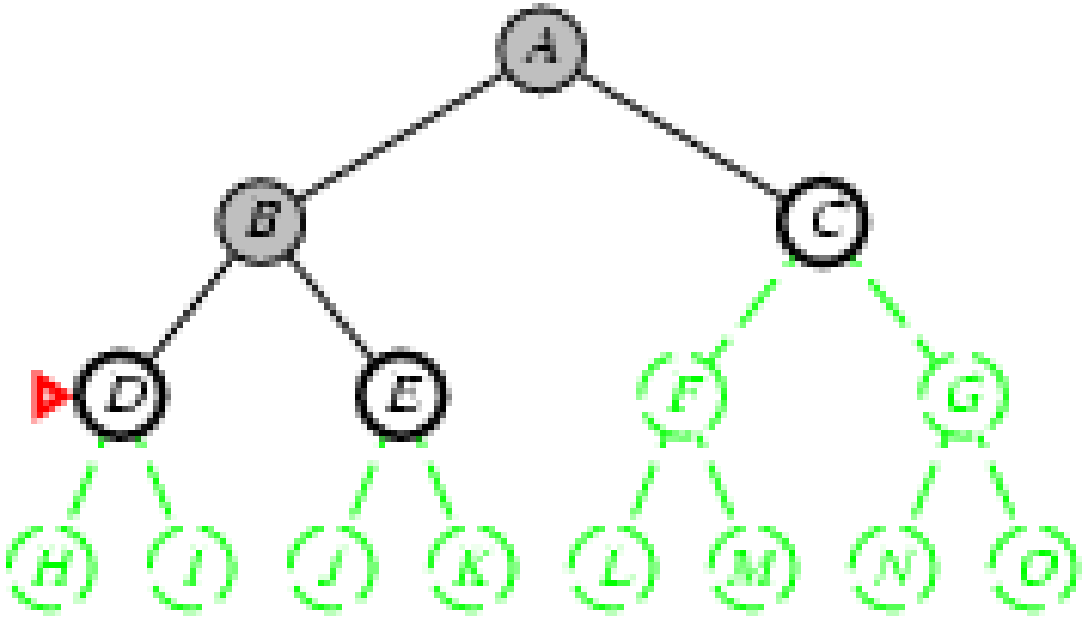
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



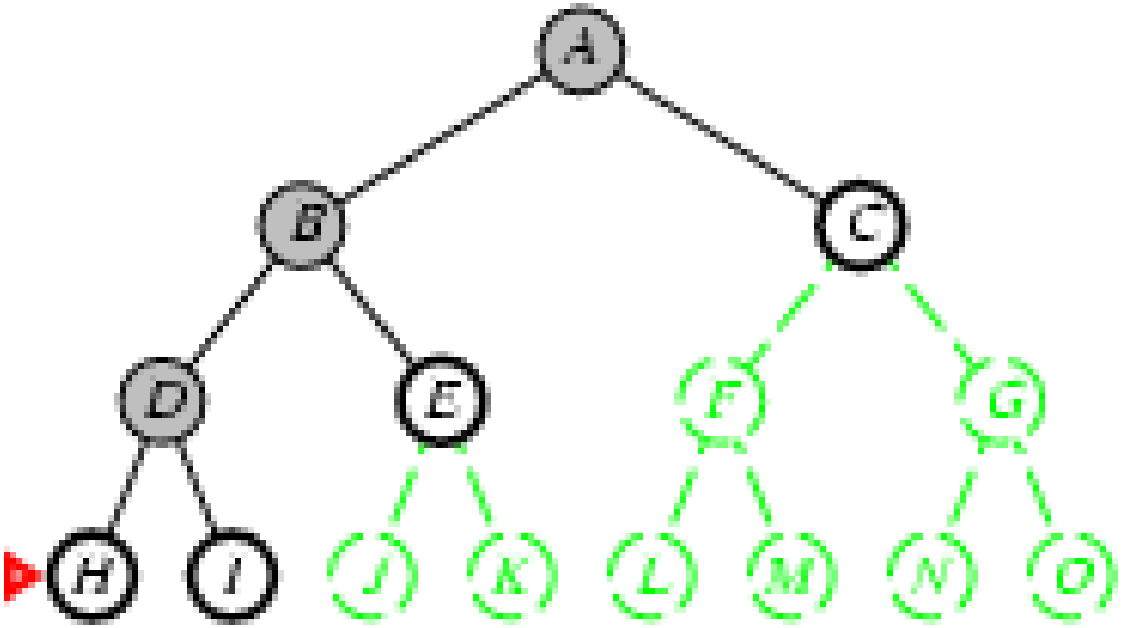
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



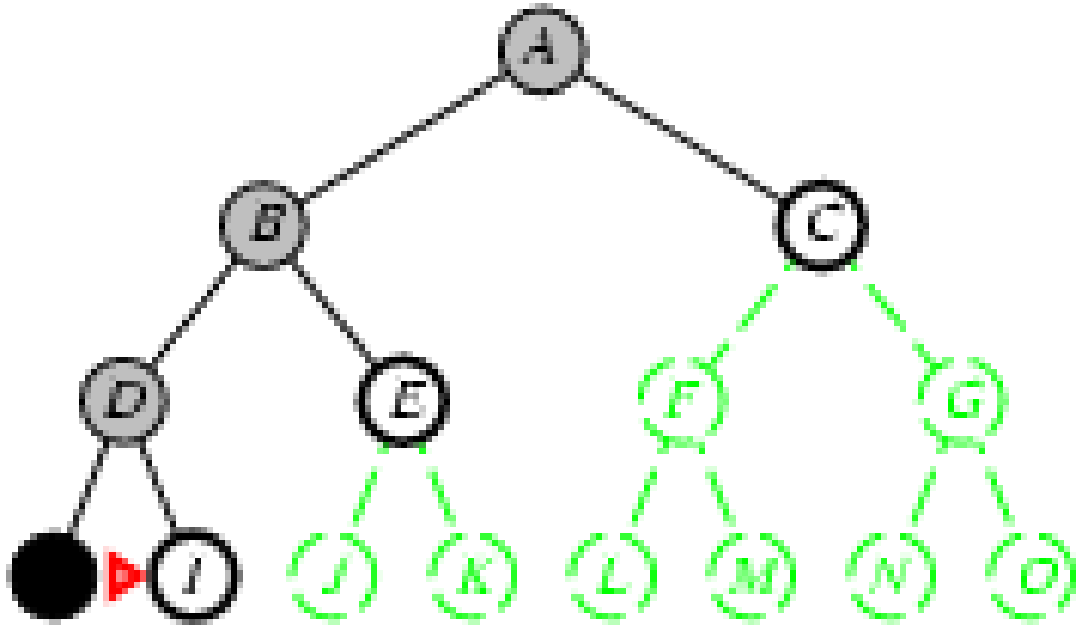
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



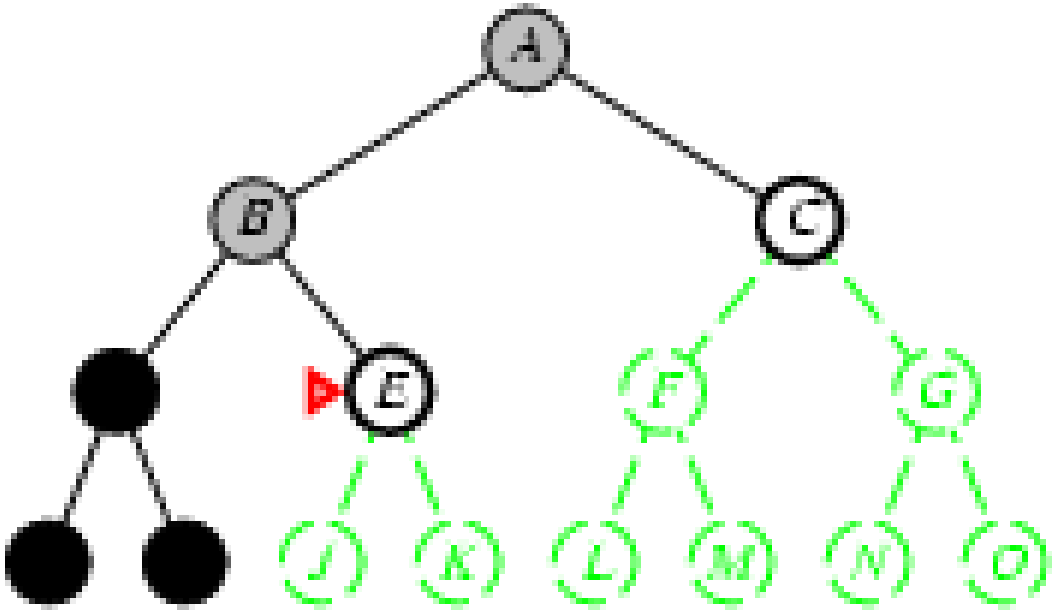
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



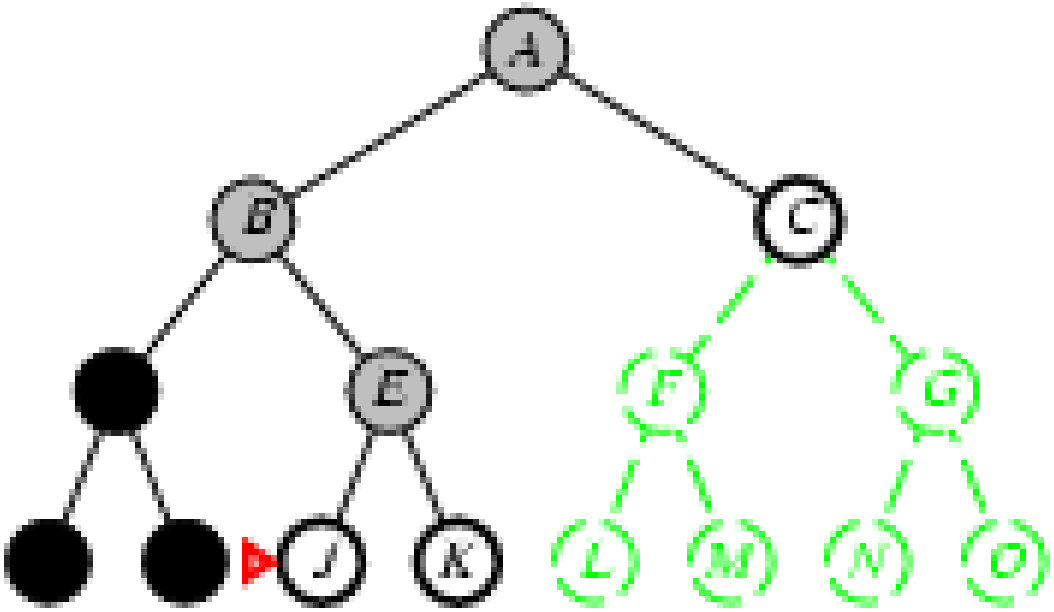
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



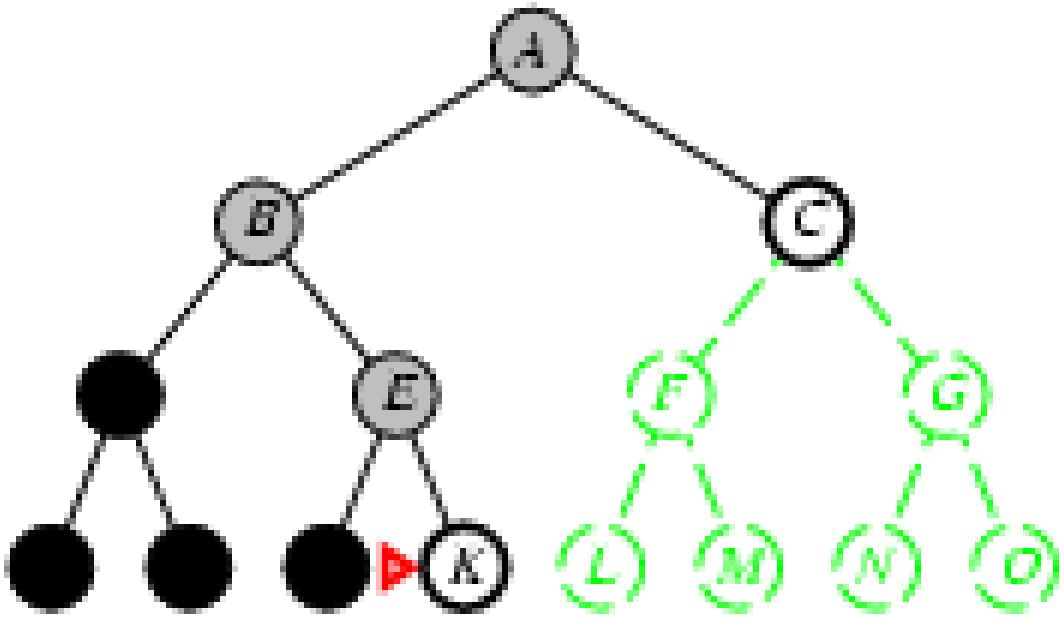
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



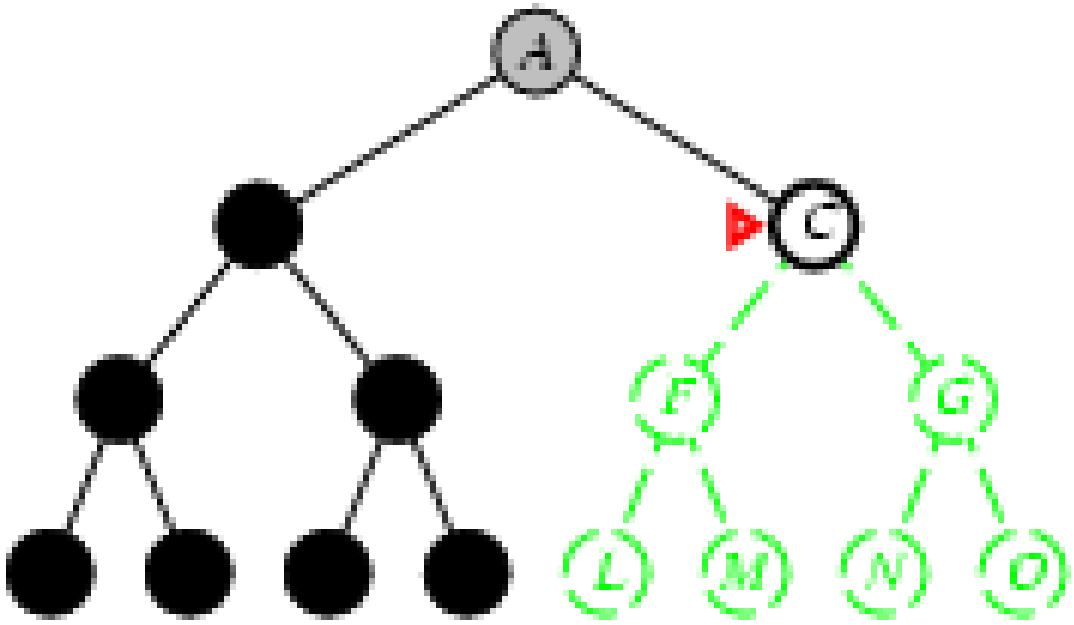
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



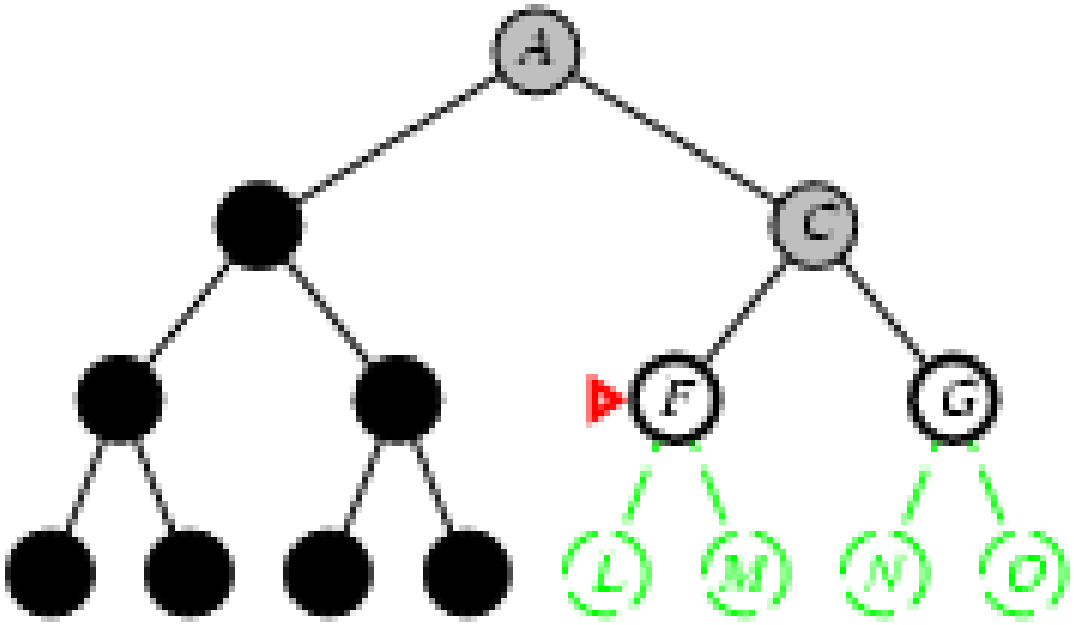
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



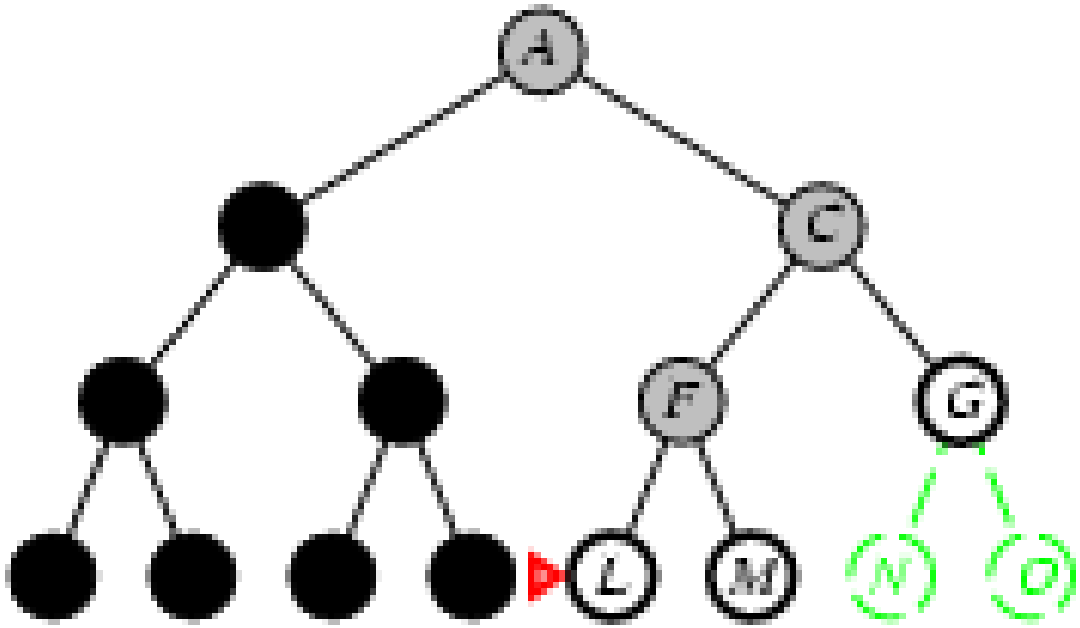
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



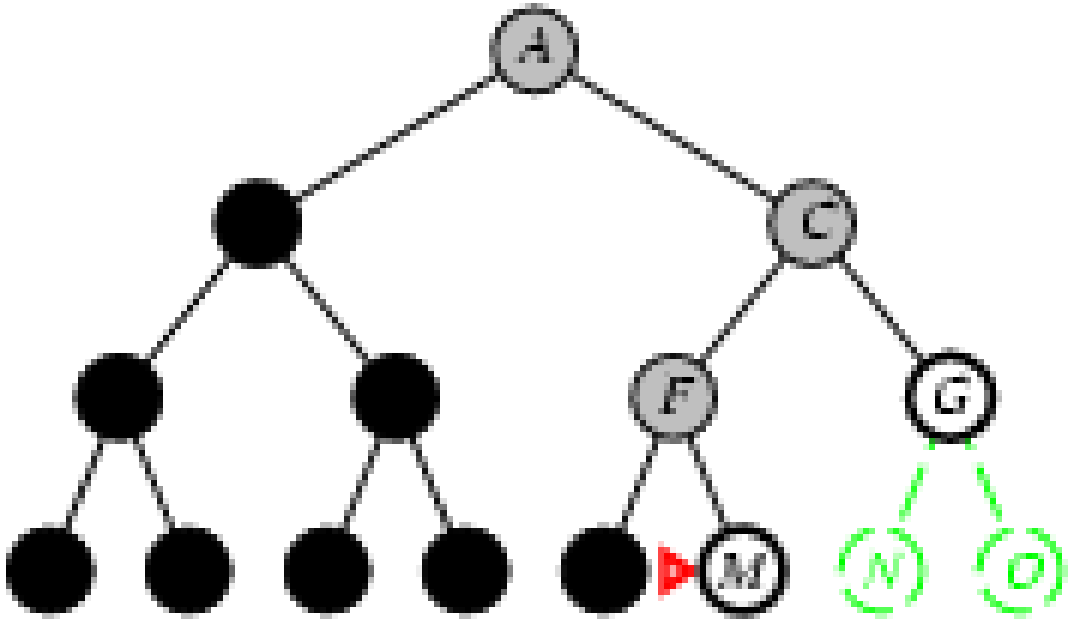
Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



Búsqueda en Profundidad

Ejemplo de exploración de nodos en búsqueda en profundidad



Búsqueda en Profundidad

Propiedades de Búsqueda en Profundidad

- Completa? SI, solo en espacios con profundidad finita
- Complejidad de tiempo:

  $O(b^m)$, pésimo cuando m es mucho mayor que d, pero si hay muchas soluciones puede ser mas eficiente que la búsqueda en amplitud

- Complejidad de espacio:

  $O(bm)$, (complejidad lineal). En el ejemplo anterior con b=10, d=m=16 se tendría 156 kilobytes en lugar de 10 exabytes

- Optima? NO, ya que la búsqueda termina cuando encuentra la 1ra solución, pudiendo haber otra a una profundidad menor.

Búsqueda en Profundidad Limitada

- La búsqueda es hasta un limite de profundidad *l*. Para esto se considera que los nodos de profundidad *l* no tienen sucesores.
- Implementación recursiva:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  else if limit = 0 then return cutoff
  else
      cutoff_occurred? ← false
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          result ← RECURSIVE-DLS(child, problem, limit − 1)
          if result = cutoff then cutoff_occurred? ← true
          else if result ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Búsqueda en Profundidad Limitada

Propiedades:

- Completa? NO, la solución puede estar mas profunda que *l*
- Complejidad de tiempo: $O(b^l)$
- Complejidad de espacio: $O(bl)$,
- Optima? NO

Búsqueda de profundización iterativa

- Llama iterativamente a BFS limitado, aumentando gradualmente el limite de profundidad *l*

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution, or failure
    for depth = 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
```

Búsqueda de profundización iterativa

Ejemplo de búsqueda en profundidad con profundización iterativa: *l=0*

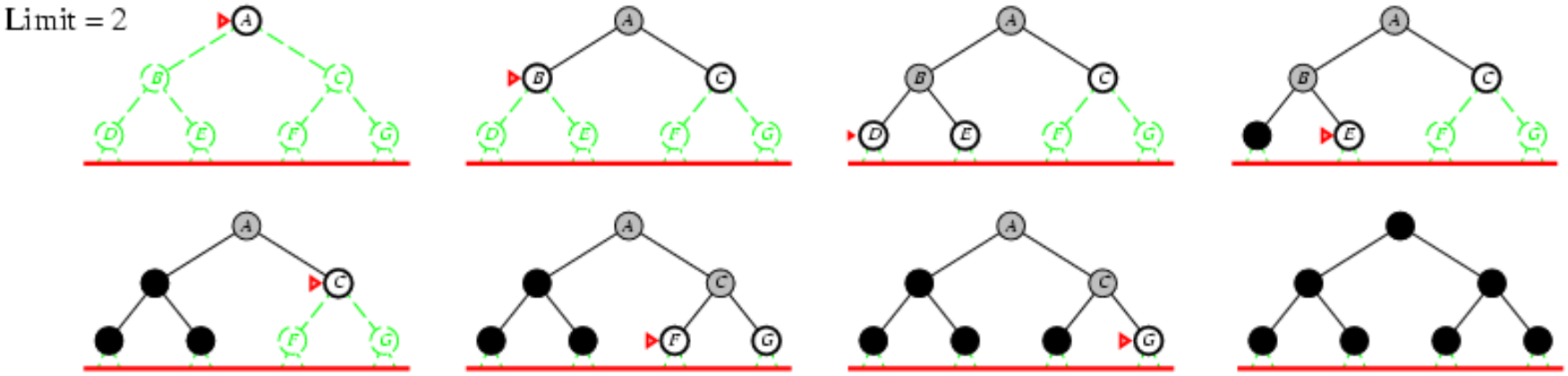
Limit = 0

Búsqueda de profundización iterativa

Ejemplo de búsqueda en profundidad con profundización iterativa: *l=1*

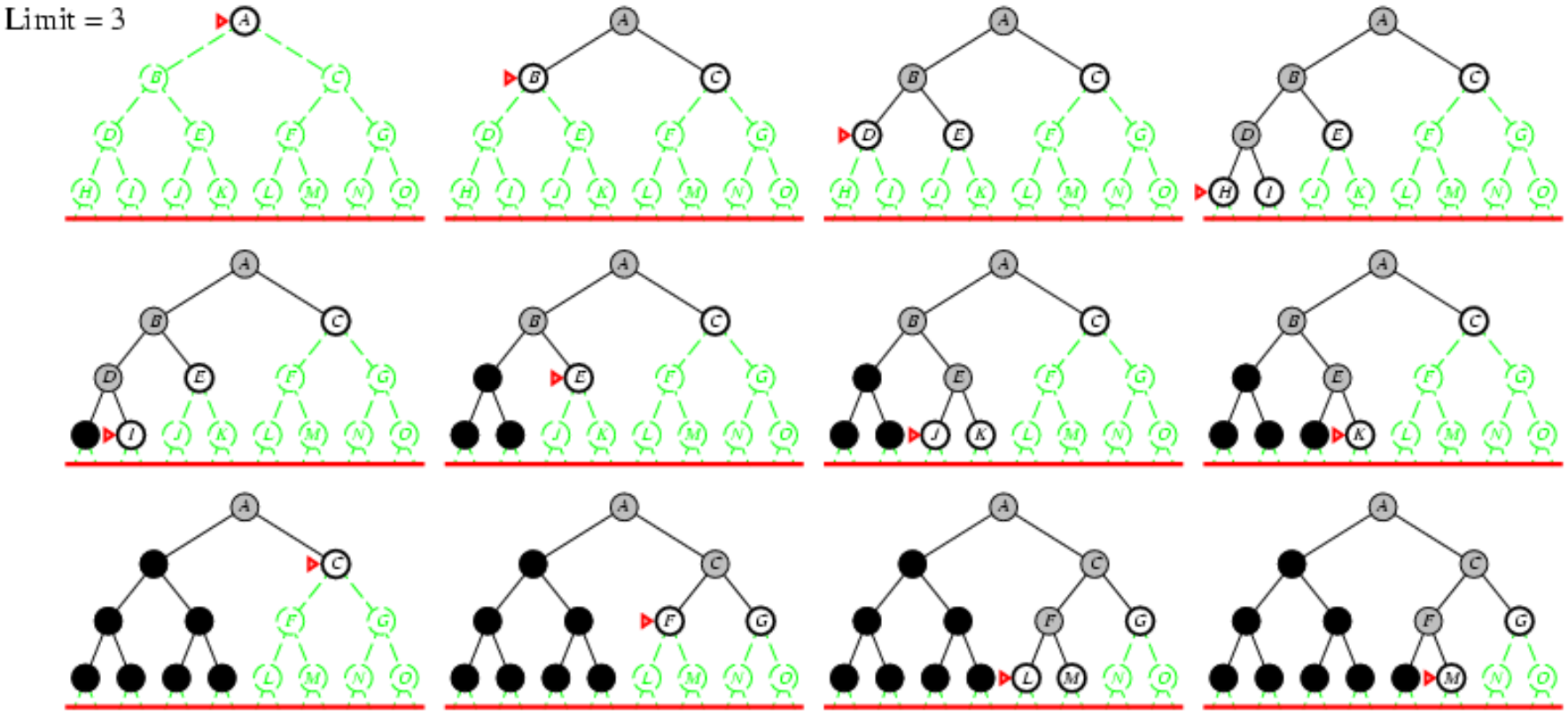
Búsqueda de profundización iterativa

Ejemplo de búsqueda en profundidad con profundización iterativa: *l=2*



Búsqueda de profundización iterativa

Ejemplo de búsqueda en profundidad con profundización iterativa: *l*=3



Búsqueda de profundización iterativa

Propiedades:

- Completa? SI, siempre encontrara un nivel donde este la solución
- Complejidad de tiempo: $O(b^d)$
- Complejidad de espacio: $O(bd)$,
- Optima? SI, si todas las acciones cuestan igual

Preguntas?