



# UNIVERSIDAD MAYOR REAL Y PONTIFICIA DE SAN FRANCISCO XAVIER DE CHUQUISACA

FACULTAD DE CIENCIA Y TECNOLOGIA

INGENIERIA DE SISTEMAS,  
INGENIERIA EN CIENCIAS DE LA COMPUTACION E  
INGENIERIA EN DISEÑO Y ANIMACION DIGITAL

SIS 420 – INTELIGENCIA ARTIFICIAL

---

***BUSQUEDAS***

*ING. CARLOS WALTER PACHECO LORA PH.D.*

*SUCRE - BOLIVIA, 2020*

# Enfoques

---

## Busqueda en grafos

- Planificacion
  - Busqueda no informada
  - Busqueda informada (heurística)
  - Algoritmos genéticos
  - Satisfaccion de restricciones
- Juegos
- Utilidad y toma de decisiones
- Aprendizaje por refuerzo

---

## Probabilidad

- Incertidumbre y probabilidad
- Razonamiento probabilístico: red bayesiana
- Razonamiento probabilístico en el tiempo
- Reconocimiento del habla
- Aprendizaje probabilístico
- Aprendizaje profundo
- Redes neuronales
- Tratamiento probabilístico del lenguaje
- Percepcion

---

## Lógica

- Lógica proposicional
- Lógica de 1er orden
- Lógica modal, temporal y difusa
- Representación del conocimiento
- Planificación
- Aprendizaje basado en el conocimiento
- Tratamiento lógico del lenguaje

# Enfoque: Búsqueda en grafos

---

## Búsqueda en grafos

- Planificación
  - Búsqueda no informada
  - Búsqueda informada (heurística)
  - Algoritmos genéticos
  - Satisfacción de restricciones
- Juegos
- Utilidad y toma de decisiones
- Aprendizaje por refuerzo

# Busqueda no informada

---

Busqueda en anchura

Busqueda en anchura de costo uniforme

Busqueda en profundidad

Busqueda en profundidad limitada

Busqueda en profundidad iterative

Busqueda bidireccional

Busqueda en grafos

# Busqueda informada

---

Heurísticas

Busqueda voraz primero el mejor

Busqueda A\* y AO\*

Busqueda de ascencion de colinas

Busqueda Tabu

Busqueda de temple simulado

Busqueda de has local

Algoritmos Geneticos

Busqueda online

# Juegos

---

Teoría de Juegos

Minimax

Poda Alfa-Beta

Función de Evaluación

Corte de Búsqueda: Efecto horizonte

Posibilidad: MinimaxEsperado

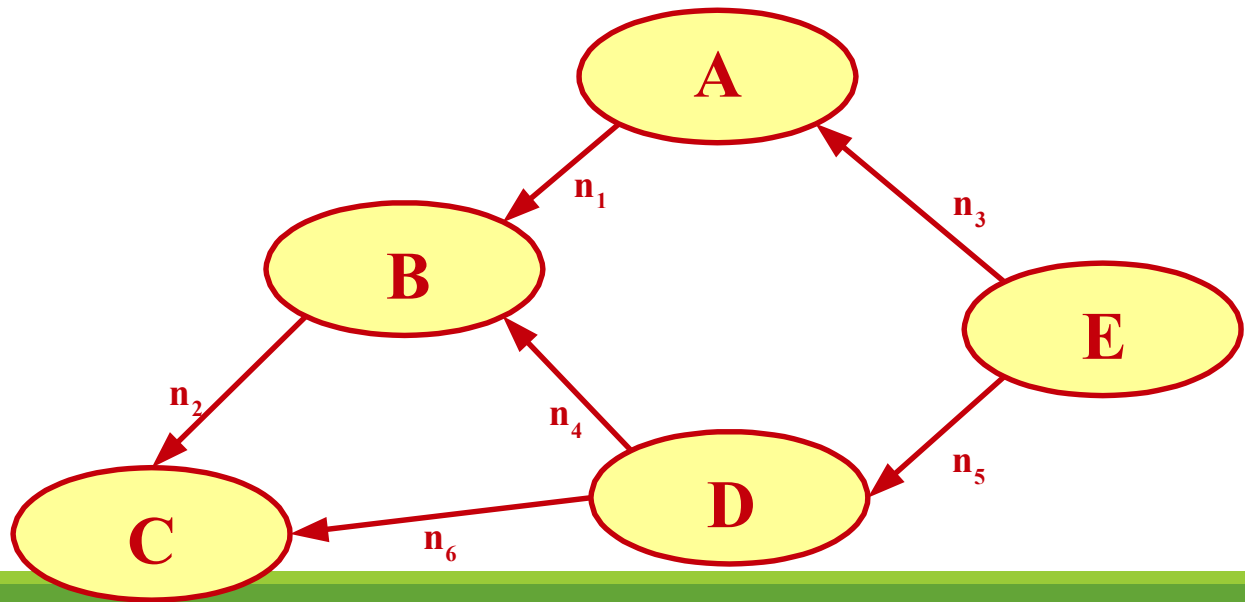


# Busqueda en grafos (Teoria de grafos)

---

Grafo, compuesto por:

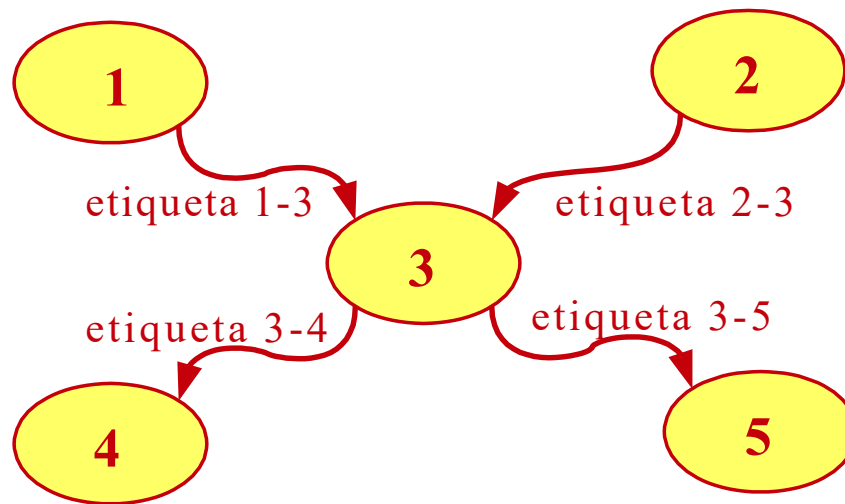
- Un conjunto de nodos (vértices)
- Un conjunto de enlaces (aristas) que unen los nodos de diferentes maneras
- Si dos nodos son los extremos de un enlace se dice que son nodos adyacentes.



# Grafo etiquetado

---

Grafo donde cada enlace tiene asociado una etiqueta, ya sea un símbolo o un numero



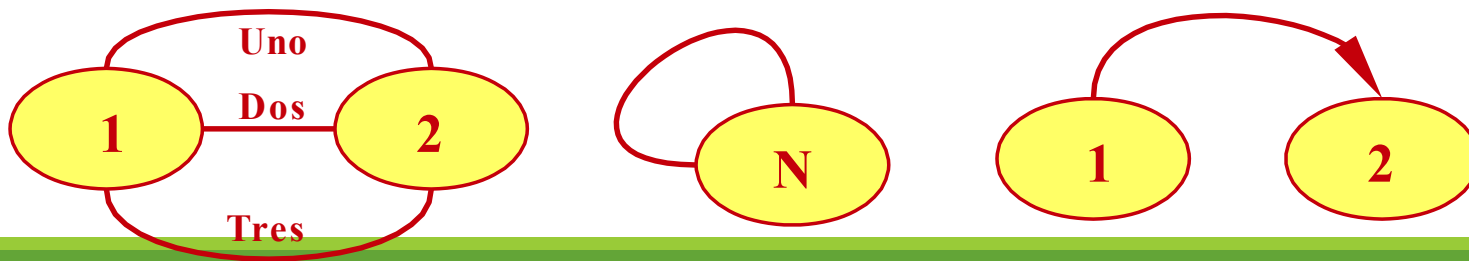
# Variantes de un grafo

---

Multigrafo: grafo que puede tener varios enlaces entre 2 nodos adyacentes.

Pseudografo: grafo que pose enlaces cuyos extremos son un mismo nodo (lazos).

Digrafo: grafo dirigido (flechas) donde los extremos de los enlaces poseen un orden

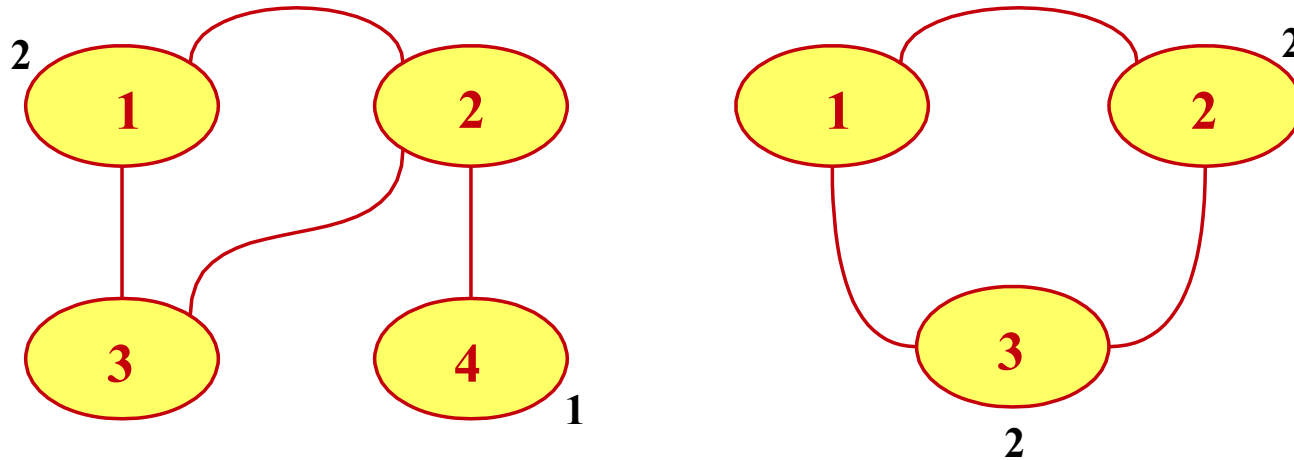


# Grado de un nodo

---

Es el numero de enlaces que tiene a un nodo por extremo.

Grafo regular: todos los nodos tienen el mismo grado



# Camino

---

Sucesión finita de nodos y enlaces.

Grafo conexo: si para cada par de nodos existe un camino entre ellos

Longitud: Numero de enlaces de un camino

Ciclo: camino cerrado en el que coinciden el primer y el ultimo nodo

Circuito: ciclo en el que no se repiten enlaces

# Arbol

---

Digrafo conexo sin ciclos. Entre cada par de nodos existe un único camino

Nodo raíz: nodo desde el que parten todos los caminos

Nodo\_interno: nodo intermedio de un camino

Nodo hoja: nodo donde termina un camino

Frontera: conjunto de todos los nodos hojas de un arbol

# Jerarquía dentro de un árbol

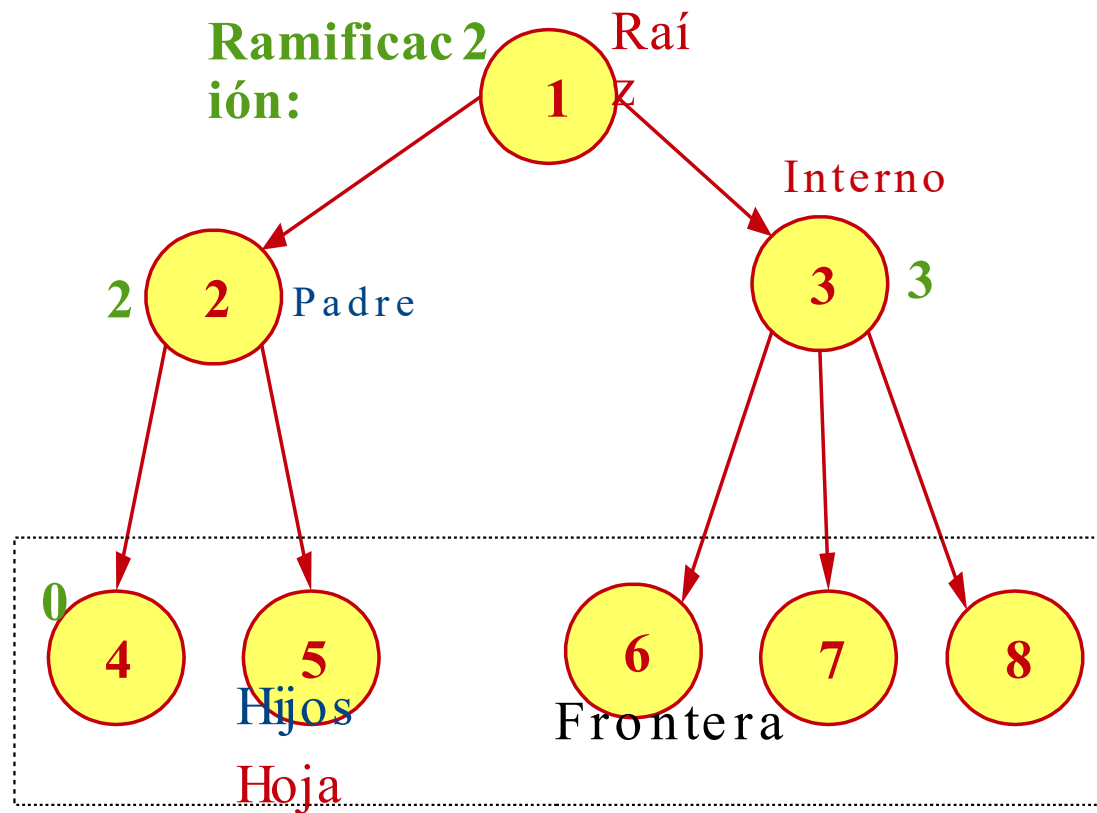
Nodos hijos: nodos destino de los enlaces que parten de un nodo

- Ramificación: número de hijos de un nodo
- Expandir nodo: crear y asignar nodos hijos

Nodo padre: nodo origen del enlace que llega hasta un nodo

Nodos Ascendentes: todos los nodos que hay desde un nodo hasta el nodo raíz

Nodo descendentes: todos los nodos que hay desde un nodo hasta sus nodos hojas




# Formulacion del problema

---

Proceso para decidir qué acciones y estados se deben considerar dado un objetivo (modelización).

Abstraccion (Modelo): proceso de eliminar detalles de una representación y dejar solo lo justo para poder solucionar un problema.

Un agente que tiene distintas opciones inmediatas pero con valores desconocidos puede decidir qué hacer evaluando las posibles diferentes acciones futuras que al final le lleven a estados de valores conocidos.





# Busqueda

---

Proceso de encontrar una secuencia de acciones (solución) para alcanzar un objetivo.

Ejecución: llevar a cabo la solución. Este tipo de agentes ignoran el resto de percepciones que recibe mientras están calculando y ejecutando una solución (lo que se llama lazo abierto en Teoría de control).

# Definicion de problemas de busqueda

---

Estado inicial

Acciones

Modelo de transición

Funcion objetivo

Funcion coste de camino

# Estado inicial

---

Estado que percibe el agente racional al comenzar a solucionar el problema.

problema.ESTADO-INICIAL → estado

# Acciones

---

Descripcion de las posibles acciones que pueden realizar (aplicar) el agente en cada estado (conjunto).

`problema.ACCIONES(estado)-> acciones`

# Modelo de transiciones

---

Descripción de lo que hace cada acción en cada estado, siendo sucesor el nuevo estado resultante.

problema.RESULTADO(estado, acción) -> estado-sucesor

Funcion sucesor:

Sucesor(estado, accion, estado-sucesor)

Sucesor(e, a, e)

# Funcion objetivo

---

Determina si un estado es un objetivo o no.

Define el conjunto de estados finales.

problema.ES-OBJETIVO(estado) -> CIERTO/FALSO

Funcion test:

Test(estado)

# Funcion coste de camino

---

Asigna un coste a cada camino.

Permite medir el rendimiento.

`problema.COSTE-ACCIÓN(estado, acción)`

`problema.COSTE-CAMINO(estado) -> coste`

Normalmente, a cada acción se le asigna un coste positivo, y el coste del camino será la suma de las que componen el camino.

Solución Optima: camino de coste mínimo.

# Espacio de estados

---

Se compone de:

- Estado Inicial.
- Acciones.
- Modelo de Transiciones.
- Conjunto de estados alcanzables desde estado inicial

Se modeliza como un grafo dirigido, los enlaces son las acciones y los caminos son secuencias de estados y acciones.



# Grafico del espacio de estados

---

Representación de un problema.

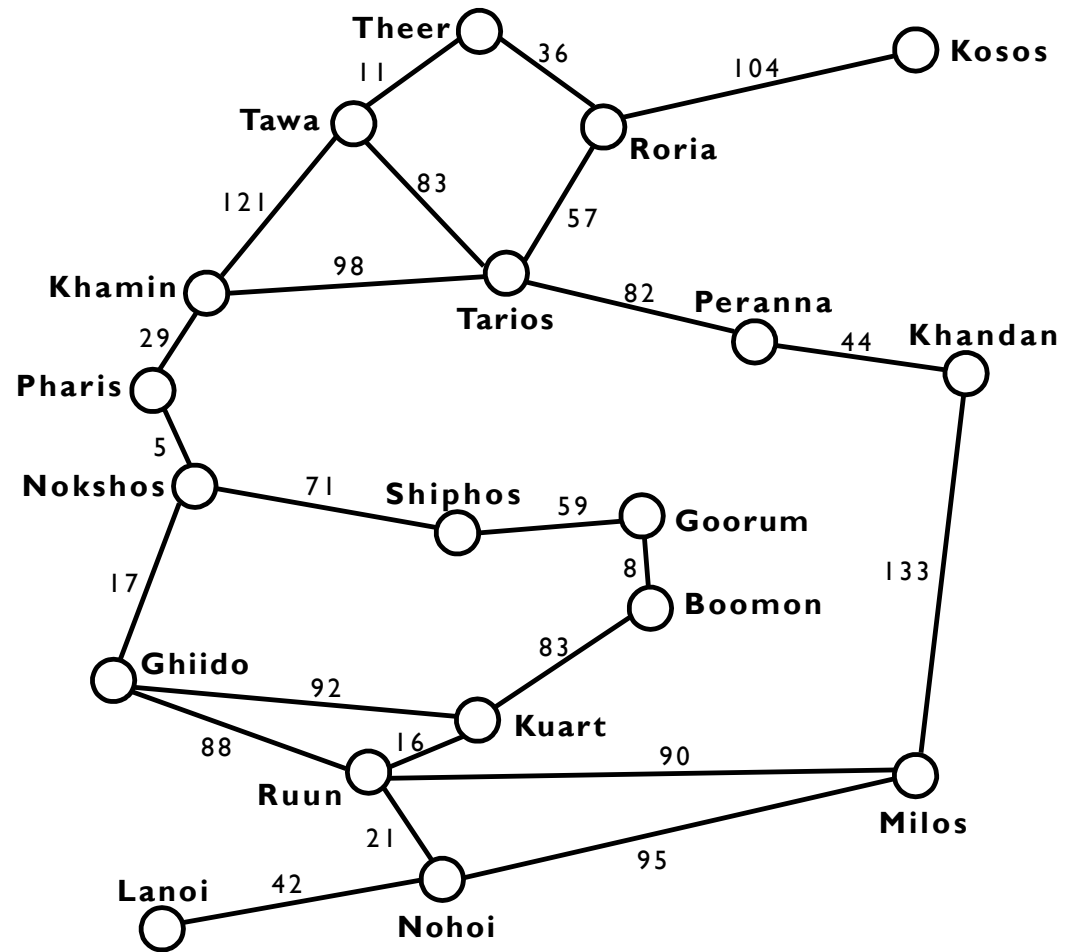
Cada nodo es un estado y las enlaces son acciones realizadas.

Los objetivos serán uno o más estados (subconjunto).

Cada estado aparece una y sólo una única vez.

Suele ser demasiado grande para construirlo completo.

Puede ser infinito.



# Arbol de busqueda

---

Los nodos también son estados y las ramas, acciones.

El nodo raíz es el estado inicial.

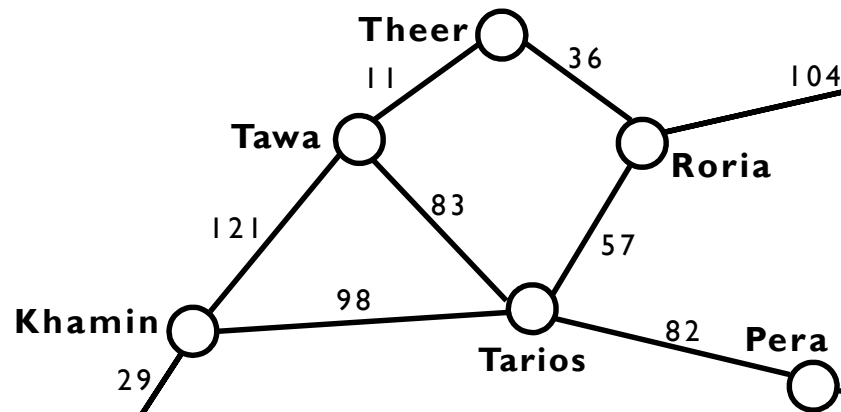
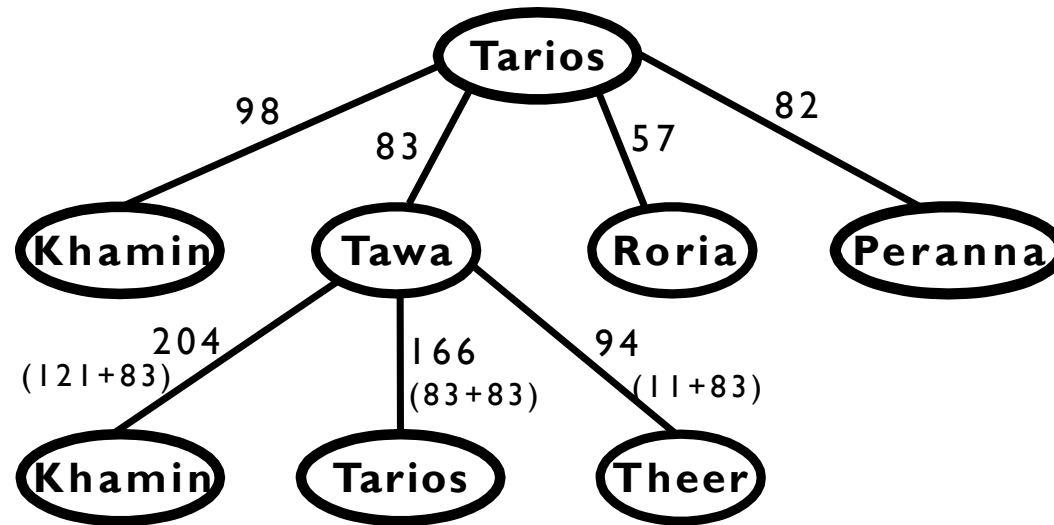
Los caminos son secuencias de acciones.

No se necesita expandir entero, sólo la parte que permita alcanzar un solución.

Los estados se repiten.

Son infinitos.





# Estados repetidos

---

Nodos distintos que representan al mismo estado

Aunque el espacio de estados sea infinito, provocan que el árbol sea infinito

Aparecen cuando hay más de una manera de ir desde un estado a otro (grafos)

# Infraestructura para algoritmos de busqueda

---

Nodo.

Frontera (Lista).

Tipos de Listas.

Medida del Rendimiento

# Nodo

---

Nodo.ESTADO -> estado

Nodo.PADRE -> nodo

Nodo.ACCION -> acción

Nodo.ACCIONES -> Lista-acciones

Nodo.COSTE -> coste-camino

$G(n)$ : coste del camino

# Frontera (Lista)

---

**frontera.ESTÁ-VACÍA() → T/F**

**frontera.POP() → nodo**

**frontera.AGREGAR(nodo) → lista**

**frontera.QUITAR(nodo) → lista**



# Tipos de lista

---

**Listas FIFO (Colas):** el primer elemento en entrar es el primero en salir (más antiguo).

**Listas LIFO (Pilas):** el último elemento en entrar es el primero en salir (más nuevo).

**Listas de Prioridad:** sale el de mayor prioridad según función de ordenación.

# Medida de rendimiento

---

Completo: encuentra solución.

Óptimo: encuentra la óptima.

Complejidad (Tiempo/Espacio).

## Tamaño del Problema

- Factor de Ramificación (b).
- Profundidad (d).
- Camino Máximo (m).

# BÚSQUEDA NO INFORMADA

---

Sólo se aporta como información la definición del problema.

Sólo se puede ir generando nodos y comprobando si alguno de ellos es la solución al problema.

Los algoritmos se diferencian en cuanto al siguiente nodo a expandir.

# BÚSQUEDA PRIMERO EN ANCHURA (Breadth-First Search)

---

Descripción:

- Se expande el nodo raíz.
- Luego, todos sus nodos hijos. Luego, los hijos de los hijos, así hasta encontrar la solución.
- Se expande cada nivel antes de expandir los del siguiente nivel.

Implementación: lista FIFO.

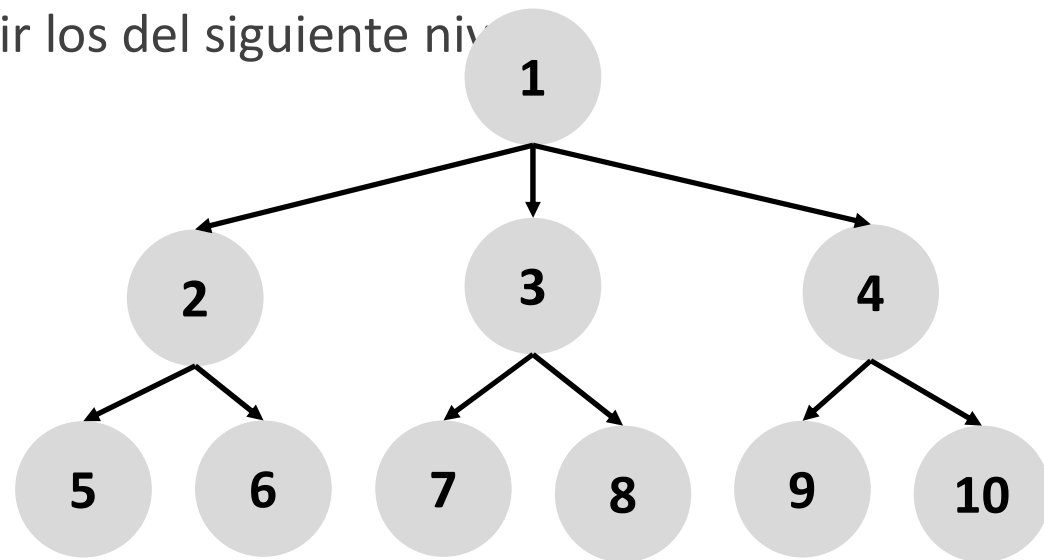
[1] →

[2, 3, 4] →

[3, 4, 5, 6] →

[4, 5, 6, 7, 8] →

[5, 6, 7, 8, 9, 10]



# BÚSQUEDA PRIMERO EN ANCHURA (Breadth-First Search)

---

## Medida de rendimiento:

- **Completo:** sí, si existe solución, la encuentra. Comprueba si un nodo es solución cuando se genera y no cuando se expande.
- **Óptimo:** sí, porque la solución encontrada es la más superficial (condición: coste de acciones iguales y no negativos).
- **Complejidad:**
  - Tiempo: exponencial  $O(bd) = bd + \dots + b^2 + b + 1$ .
  - Espacio: exponencial  $O(bd)$  en frontera y  $O(bd-1)$  en explorada.

# ALGORITMO - PSEUDOCÓDIGO

**función** BÚSQUEDA-PRIMERO-ANCHURA(problema)

**devuelve** solución o fallo

---

nodo-raíz ← CREAR-NODO-RAÍZ(problema)

**si** problema.ES-OBJETIVO(nodo-raíz.ESTADO) **entonces**

**devolver** nodo-raíz

frontera ← CREAR-FIFO()

frontera.AGREGAR(nodo-raíz)

explorada ← CREAR-CONJUNTO()

**repetir**

**si** frontera.ESTÁ-VACÍA() **entonces devolver** fallo

    nodo ← frontera.POP()

    explorada.AGREGAR(nodo)

**por cada** acción **en** problema.ACCIONES(nodo.ESTADO) **hacer**

        hijo ← CREAR-NODO-HIJO(problema, nodo, acción)

**si** hijo.ESTADO **no está en** explorada **y**

            hijo.ESTADO **no está en** frontera.ESTADOS() **entonces**

**si** problema.ES-OBJETIVO(hijo.ESTADO) **entonces**

**devolver** hijo

                frontera.AGREGAR(hijo)

# IMPLEMENTACIÓN

---

# BÚSQUEDA PRIMERO EN ANCHURA (Breadth-First Search)


---

## Análisis:

- **Ventajas**

- Si hay solución, la encuentra.
- Encuentra la solución óptima.

- **Desventajas**

- Expande muchos nodos inútiles.
  - Orden exponencial en espacio.
  - Coste constante y no negativo.
  - Sólo para problemas muy simples.
- 



# BÚSQUEDA DE COSTO UNIFORME (Uniform-Cost Search)

---

Descripción:

- Basado en Primero en Anchura.
- Costes de acciones variables.
- Costes no negativos.

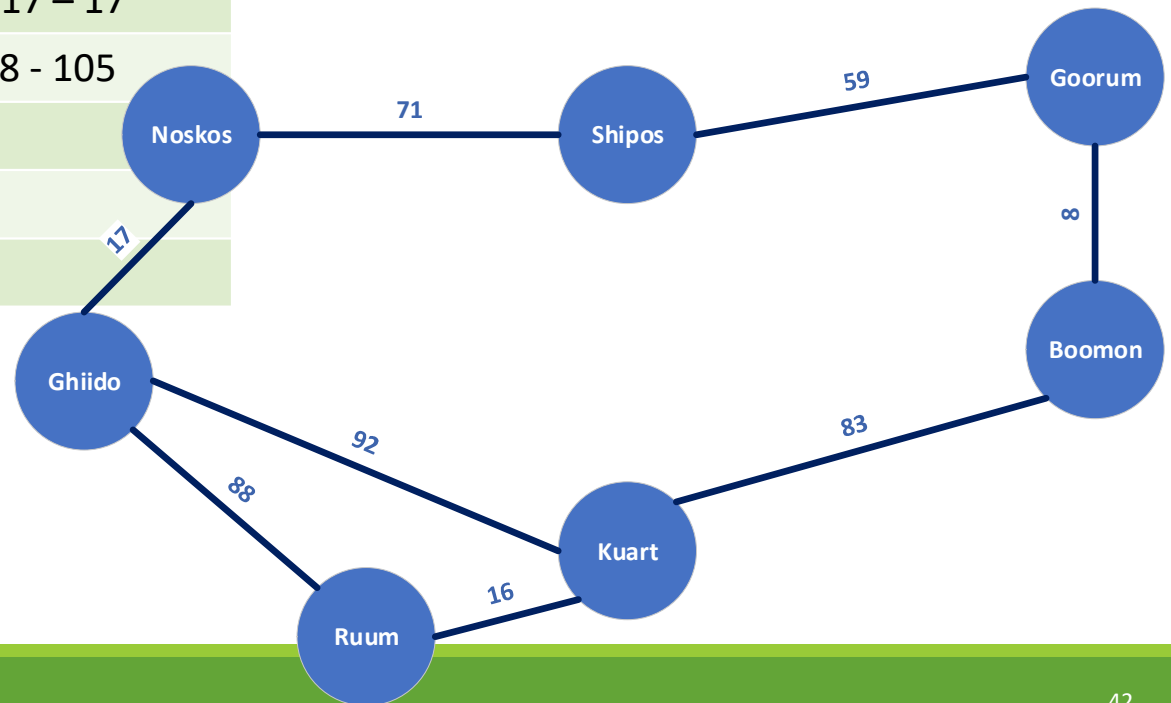
Expande nodo con menor coste asociado (no el más superficial).

Implementación: lista de prioridad.

# BÚSQUEDA DE COSTO UNIFORME (Uniform-Cost Search)

## Llegar de Nokshos a Ruun

CAMINO 1	CAMINO 2	CAMINO 3
Shipos 71 – 71	Ghiido 17 – 17	Ghiido 17 – 17
Goorum 59 - 130	Kuart 92 – 109	Ruun 88 - 105
Boomon 8 - 138	Ruun 16 - 125	
Kuart 83 - 221		
Ruun 16 - 237		



# ALGORITMO

**función** BÚSQUEDA-COSTE-UNIFORME(problema) **devuelve** solución o fallo

nodo-raíz ← CREAR-NODO-RAÍZ(problema)

frontera ← CREAR-PRIORIDAD()

---

frontera.AGREGAR(nodo-raíz)

explorada ← CREAR-CONJUNTO()

**repetir**

**si** frontera.ESTÁ-VACÍA() **entonces** devolver fallo

  nodo ← frontera.POP()

**si** problema.ES-OBJETIVO(nodo.ESTADO) **entonces** devolver nodo

  explorada.AGREGAR(nodo)

**por cada** acción **en** problema.ACCIONES(nodo.ESTADO) **hacer**

    hijo ← CREAR-NODO-HIJO(problema, nodo, acción)

**si** hijo.ESTADO **no está en** explorada **y**

      hijo.ESTADO **no está en** frontera.ESTADOS() **entonces**

        frontera.AGREGAR(hijo)

**sino**

      nodo-frontera ← frontera.BUSCAR(hijo.ESTADO)

**si** hijo.COSTE < nodo-frontera.COSTE **entonces**

        nodo-frontera ← hijo

# IMPLEMENTACIÓN

---

# BÚSQUEDA DE COSTO UNIFORME (Uniform-Cost Search)

---

Medida de Rendimiento:

Completo: sí, si existe solución, la encuentra. Nota: Comprueba al expandir y no al generar.

Óptimo: sí, porque obtiene el camino con el menor coste.

Complejidad:

- Tiempo: exponencial  $O(b^{1+C^*/\epsilon})$ 
  - $C^*$  coste de camino óptimo,
  - $\epsilon$  coste mínimo de acción.
  - Nota: puede ser superior a  $O(b^d)$
  - $\text{coste}=\text{cte.} \rightarrow C^*/\epsilon=d \rightarrow O(b^{d+1})$
- Espacio: exponencial  $O(b^{1+ \lceil C^*/\epsilon \rceil})$ .

# BÚSQUEDA DE COSTO UNIFORME (Uniform-Cost Search)


---

Análisis:

- **Ventajas**

- Completo y óptimo.
- Admite costes variables.

- **Desventajas**

- Orden exponencial en espacio.
  - Costes no negativos.
  - Sólo para problemas muy simples.
- 

# BÚSQUEDA PRIMERO EN PROFUNDIDAD (Depth-First Search)

---

## Descripción:

- Se expande el nodo raíz.
- Luego, uno de sus hijos.
- Luego, uno de los hijos del hijo, etc.
- Cuando se llega a una hoja, si no es solución, se retrocede y se prueba con el siguiente hijo.
- Se expande el nodo más profundo de la frontera.

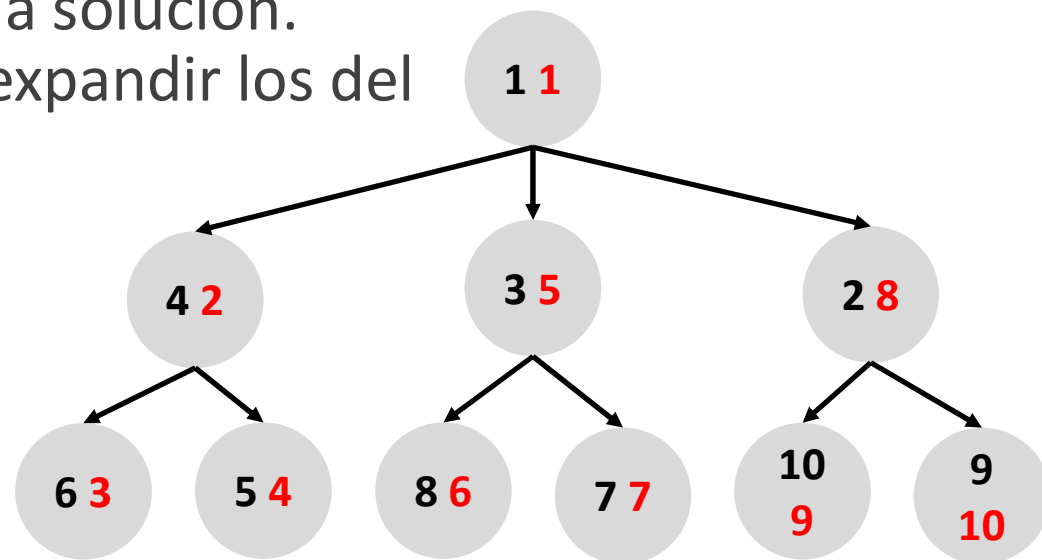
# BÚSQUEDA PRIMERO EN PROFUNDIDAD (Depth-First Search)

Descripción:

- Se expande el nodo raíz.
- Luego, todos sus nodos hijos. Luego, los hijos de los hijos, así hasta encontrar la solución.
- Se expande cada nivel antes de expandir los del siguiente nivel.

Implementación: lista FIFO.

[1] → [2, 3, 4] → [2, 3, 5, 6] → [2, 3, 5] →  
[2, 3] → [2, 7, 8] → [2, 7] → [2] →  
[9, 10] → [9]





# ALGORITMO

```
función BÚSQUEDA-PRIMERO-PROFUNDIDAD(problema)
devuelve solución o fallo
  nodo-raíz ← CREAR-NODO-RAÍZ(problema)
  si problema.ES-OBJETIVO(nodo-raíz.ESTADO) entonces
    devolver nodo-raíz
  frontera ← CREAR-LIFO()
  frontera.AGREGAR(nodo-raíz)
  explorada ← CREAR-CONJUNTO()
  repetir
    si frontera.ESTÁ-VACÍA() entonces devolver fallo
    nodo ← frontera.POP()
    explorada.AGREGAR(nodo)
    por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
      hijo ← CREAR-NODO-HIJO(problema, nodo, acción)
      si hijo.ESTADO no está en explorada y
        hijo.ESTADO no está en frontera.ESTADOS() entonces
          si problema.ES-OBJETIVO(hijo.ESTADO) entonces
            devolver hijo
          frontera.AGREGAR(hijo)
```

# ALGORITMO

```
función BÚSQUEDA-PRIMERO-PROFUNDIDAD(problema)
devuelve solución o fallo
    explorada ← CREAM-CONJUNTO()
    nodo-raíz ← CREAM-NODO-RAÍZ(problema)
    devuelve BPP-RECURSIVA(nodo-raíz, problema, explorada)

función BPP-RECURSIVA(nodo, problema, explorada)
devuelve solución o fallo
    si problema.ES-OBJETIVO(nodo.ESTADO) entonces
        devolver nodo
    por cada acción en problema.ACCIONES(nodo.ESTADO) hacer
        hijo ← CREAM-NODO-HIJO(problema, nodo, acción)
        si hijo.ESTADO no está en explorada entonces
            resultado ← BPP-RECURSIVA(hijo, problema, explorada)
            si resultado ≠ fallo entonces devolver resultado
    devolver fallo
```

# IMPLEMENTACIÓN

---

# BÚSQUEDA PRIMERO EN PROFUNDIDAD (Depth-First Search)

---

## Medida de Rendimiento:

- Completo: si evita caminos redundantes y el espacio de estados es finito.
  - Nota: comprobar estados repetidos no evita los caminos redundantes.
  - En un espacio de estados infinito no es completo porque puede no encontrar nunca la solución.
- Óptimo: no, puede encontrar otras soluciones antes de la óptima.
- Complejidad:
- Tiempo: exponencial  $O(b^m)$ , donde  $m$  puede ser mayor que  $d$  (incluso puede ser infinito).
- Espacio: lineal  $O(b \cdot m)$ , ya que sólo almacena el camino y los nodos hijos de los nodos intermedios.
  - Si sólo se expande un nodo hijo:  $O(m)$
  - Si no hace falta guardar el camino:  $O(1)$
  - Nota: en estos 2 últimos casos no hay frontera.

# BÚSQUEDA PRIMERO EN PROFUNDIDAD (Depth-First Search)

---

Análisis:

- **Ventajas**

- Ocupa muy poco espacio.

- **Desventajas**

- No es completo ni óptimo.
  - Puede probar muchos caminos inútiles.
  - Puede quedar atrapado en bucles infinitos.
  - Coste constante y no negativo.
  - Sólo para problemas simples.



# BÚSQUEDA EN PROFUNDIDAD LIMITADA (Depth-Limited Search)

---

## Descripción:

- Basada en Primero en Profundidad.
- Se establece una profundidad máxima ( $p$ ) como diámetro del espacio de estados (camino máximo).
- Cuando se llega a la profundidad máxima  $p$ , si no hay solución, se retrocede.
- Se expande el nodo más profundo de la frontera hasta un límite.

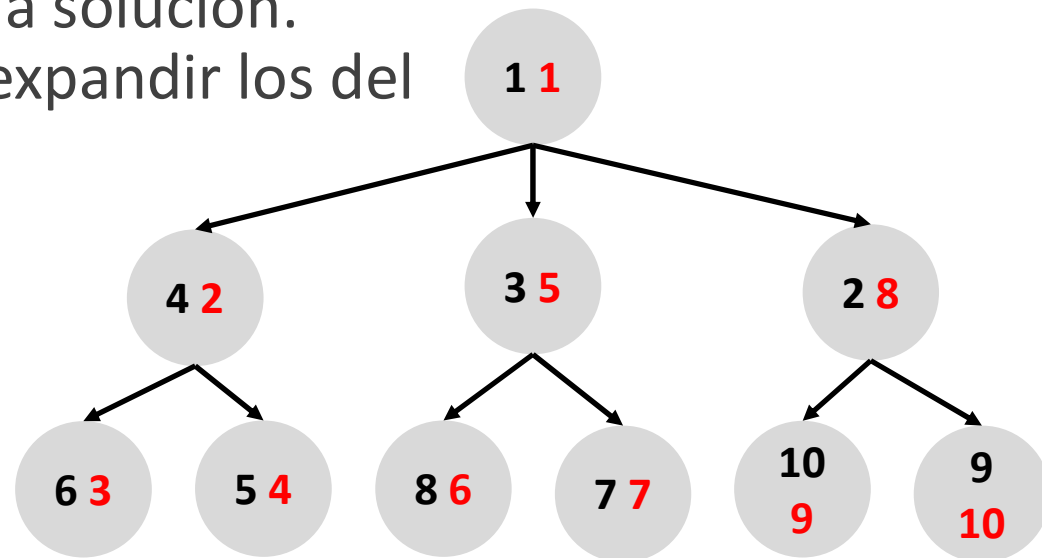
# BÚSQUEDA EN PROFUNDIDAD LIMITADA (Depth-Limited Search)

Descripción:

- Se expande el nodo raíz.
- Luego, todos sus nodos hijos. Luego, los hijos de los hijos, así hasta encontrar la solución.
- Se expande cada nivel antes de expandir los del siguiente nivel.

Implementación: lista FIFO.

[1] → [2, 3, 4] → [2, 3, 5, 6] → [2, 3, 5] →  
[2, 3] → [2, 7, 8] → [2, 7] → [2] →  
[9, 10] → [9]





# ALGORITMO

```
función BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, límite)  
devuelve solución o fallo o corte  
    explorada ← CREAR-CONJUNTO()  
    nodo-raíz ← CREAR-NODO-RAÍZ(problema)  
    devuelve BPL-RECURSIVA(nodo-raíz, problema, límite, explorada)  
  
función BPL-RECURSIVA(nodo, problema, límite, explorada)  
devuelve solución o fallo o corte  
    si problema.ES-OBJETIVO(nodo.ESTADO) entonces devolver nodo  
    si límite = 0 entonces devolver corte  
    por cada acción en problema.ACCIONES(nodo.ESTADO) hacer  
        hijo ← CREAR-NODO-HIJO(problema, nodo, acción)  
        si hijo.ESTADO no está en explorada entonces  
            resultado ← BPL-RECURSIVA(hijo, problema,  
                                     límite - 1, explorada)  
            si resultado ≠ fallo entonces devolver resultado  
    devolver fallo
```

# IMPLEMENTACIÓN

---

# BÚSQUEDA EN PROFUNDIDAD LIMITADA (Depth-Limited Search)

---

Medida de Rendimiento:

- Completo: no, las soluciones pueden estar más allá del límite máximo  $p$ .
- Óptimo: no, puede encontrar otras soluciones antes de la óptima, que podría quedar además por debajo del límite máximo  $p$ .
- Complejidad:
- Tiempo: exponencial  $O(b^p)$ .
- Espacio: lineal  $O(b \cdot p)$ ,  $O(p)$ ,  $O(1)$ .

# BÚSQUEDA EN PROFUNDIDAD LIMITADA (Depth-Limited Search)


---

Análisis:

- **Ventajas**

- Ocupa muy poco espacio.
- No cae en bucles infinitos.

- **Desventajas**

- No es completo ni óptimo.
  - Puede probar muchos caminos inútiles.
  - Coste constante y no negativo.
  - Sólo para problemas simples.
- 

# BÚSQUEDA EN PROFUNDIDAD ITERATIVA (Iterative Deepening Search)

---

Descripción:

- Ampliación de Profundidad Limitada.
- De forma gradual, se va ampliando el límite máximo  $p$ .

Se va ampliando el límite hasta alcanzar una solución, que además será la óptima.

# BÚSQUEDA EN PROFUNDIDAD ITERATIVA (Iterative Deepening Search)

---

Medidas de rendimiento:

**Completo:** sí, si hay solución, la encuentra.

**Óptimo:** sí, porque cuando encuentra una solución, es la más superficial.

**Complejidad:**

- Tiempo: exponencial  $O(bd)$   
 $(d) \cdot b + (d-1) \cdot b^2 + \dots + (2) \cdot b^{d-1} + (1) \cdot b^d$
- Espacio: lineal  $O(b \cdot d)$ ,  $O(d)$ ,  $O(1)$ .

# BÚSQUEDA EN PROFUNDIDAD ITERATIVA (Iterative Deepening Search)

---

```
función BÚSQUEDA-PROFUNDIDAD-ITERATIVA(problema)
devuelve solución o fallo
  para límite de 0 a máximo hacer
    resultado ← BÚSQUEDA-PROFUNDIDAD-LIMITADA(problema, límite)
    si resultado = solución entonces devolver solución
  devolver fallo
```

# BÚSQUEDA EN PROFUNDIDAD ITERATIVA (Iterative Deepening Search)

---

Análisis:

- **Ventajas**

- Ocupa muy poco espacio.
- No cae en bucles infinitos.
- Es completo y óptimo

- **Desventajas**

- Puede probar muchos caminos inútiles.
- Visita muchas veces los nodos superficiales.
- Coste constante y no negativo.



# BÚSQUEDA EN PROFUNDIDAD ITERATIVA (Iterative Deepening Search)

---

Es el mejor algoritmo de búsqueda no informada, sobre todo si el espacio de búsqueda es muy

grande y no se conoce a qué profundidad está la solución.

Mejora: usar primero en anchura hasta agotar memoria y, luego, usar búsqueda en profundidad iterativa desde cada uno de los nodos frontera (Evita visitar los nodos superficiales de nuevo).

# IMPLEMENTACIÓN

---