Técnicas de Búsqueda de Soluciones

Jose Aguilar
Cemisid, Facultad de Ingeniería
Universidad de los Andes
Mérida, Venezuela
aguilar@ula.ve

Agentes y Búsqueda

- Algoritmo de búsqueda:
  - Permitir la transición entre estados usando los operadores
  - Controlar esos movimientos
- Proceso de Búsqueda
  - **búsqueda ciega (Búsquedas sin contar con información)**: no utiliza información sobre el problema. No existe información acerca de los pasos necesarios o costos para pasar de un estado a otro. Normalmente, se realiza una búsqueda exhautiva.
  - **Búsqueda heuristicas (Búsqueda respaldada con información)**: usan información sobre el problema como costos, etc. se posee información muy valiosa para orientar la búsqueda para que sea mas óptima

Estrategias de Búsqueda

Búsqueda No Informada (Ciega)

1. Búsqueda por amplitud
2. Búsqueda de costo uniforme
3. Búsqueda por profundidad
4. Búsqueda limitada por profundidad
5. Búsqueda por profundización iterativa
6. Búsqueda bidireccional

Búsqueda Informada (Heurística)

1. Búsqueda avara
2. Búsqueda A*
3. Búsqueda A*PI
4. Búsqueda A*SRM

Muchas otras heurísticas!!!

Criterios de rendimiento

Las estrategias de búsqueda se evalúan según los siguientes criterios:

- **Completitud:** si garantiza o no encontrar la solución si es que existe.

  ¿La estrategia garantiza encontrar una solución, si ésta existe?
- **Complejidad temporal:** cantidad de tiempo necesario para encontrar la solución.

  ¿Cuánto tiempo se necesitará para encontrar una solución?
- **Complejidad espacial:** cantidad de memoria necesaria para encontrar la solución.

  ¿Cuánta memoria se necesita para efectuar la búsqueda?
- **Optimalidad: si** se encontrará o no la mejor solución en caso de que existan varias.

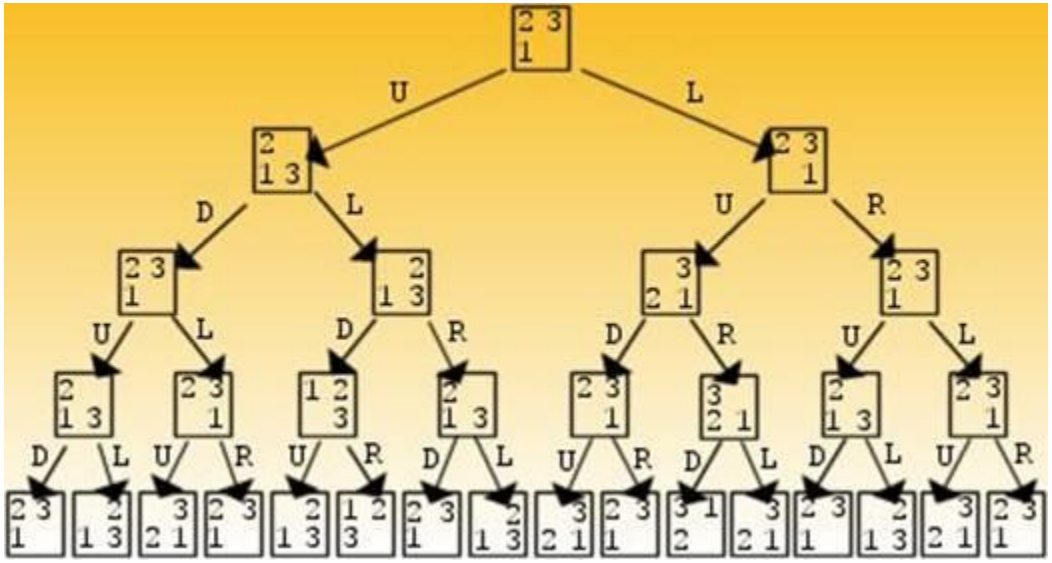
¿Con esta estrategia se encontrará una solución de la más alta calidad, si hay varias soluciones?

Búsqueda ciega

- La búsqueda consiste **escoger uno de los estados posibles** conocidos (una de las opciones).
- Si hay varios estados que se pueden expandir, la elección de cual se expande primero se hace según una **estrategia de búsqueda**.
- El proceso de búsqueda se concibe como la construcción de un **árbol de búsqueda** sobrepuesto al espacio de estados.
- Las técnicas **se diferencian** en el orden en que expanden los nodos en el árbol de búsqueda.

Árbol de Búsqueda

- La **raíz** del árbol de búsqueda corresponde al **estado inicial**
- Los **nodos hoja** del árbol de búsqueda o no se han expandido, o no tienen sucesores, por lo que al expandirlos generaron un conjunto vacío



Árboles de búsqueda

Componentes de los árboles de búsqueda:

- El **estado** al que corresponde cada nodo,
- El **nodo padre (estado inicial)**,
- El **operado**r que se aplica para generar cada nodo,
- La **profundidad** del nodo (distancia hasta la raíz),
- El **costo** desde el estado inicial hasta un nodo dado.

Algoritmo General de búsqueda ciega

- Iniciar árbol de búsqueda con el edo. inicial del problema
- Si no hay candidato para expandir entonces
  - **falla**
- de lo contrario
  - escoger nodo hoja para expandir según estrategia
  - Si nodo es edo. objetivo entonces
    - **Solución** del problema
  - de lo contrario
    - expandir nodo y añadir nuevo nodo al árbol de búsqueda

Búsqueda no informada o ciega

- Estrategia de búsqueda (a quien expandir)
  - Por extensión o amplitud

    Padre, sus hijos, etc. Completa pero no optima
  - Uniforme

    Expande nodo más barato
  - Por Profundidad

    Expande un nodo hasta su hoja y regresa. Completa pero no optima
  - Por Profundidad limitada. Ni completa ni optima
  - Profundidad Iterativa

    Prueba diferentes profundidades. Completa y optima
  - Bidireccional

Búsqueda por amplitud o extensión

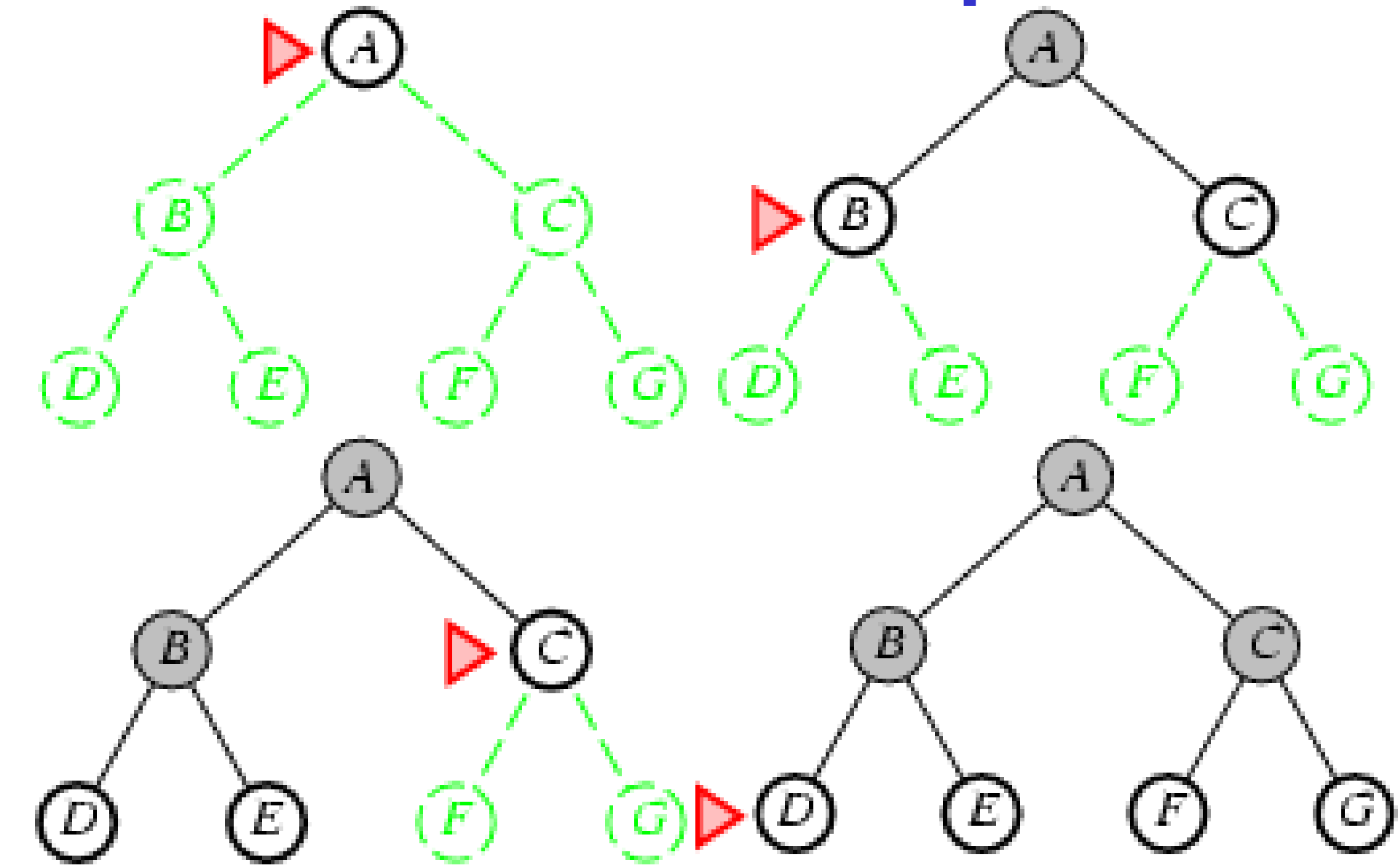
Todos los nodos que están en la profundidad *d* del árbol de búsqueda se expanden antes de los nodos que estén en la profundidad *d+1*.

- Si son varias las soluciones, este tipo de búsqueda permitirá siempre encontrar primero el estado meta más próximo a la raíz.

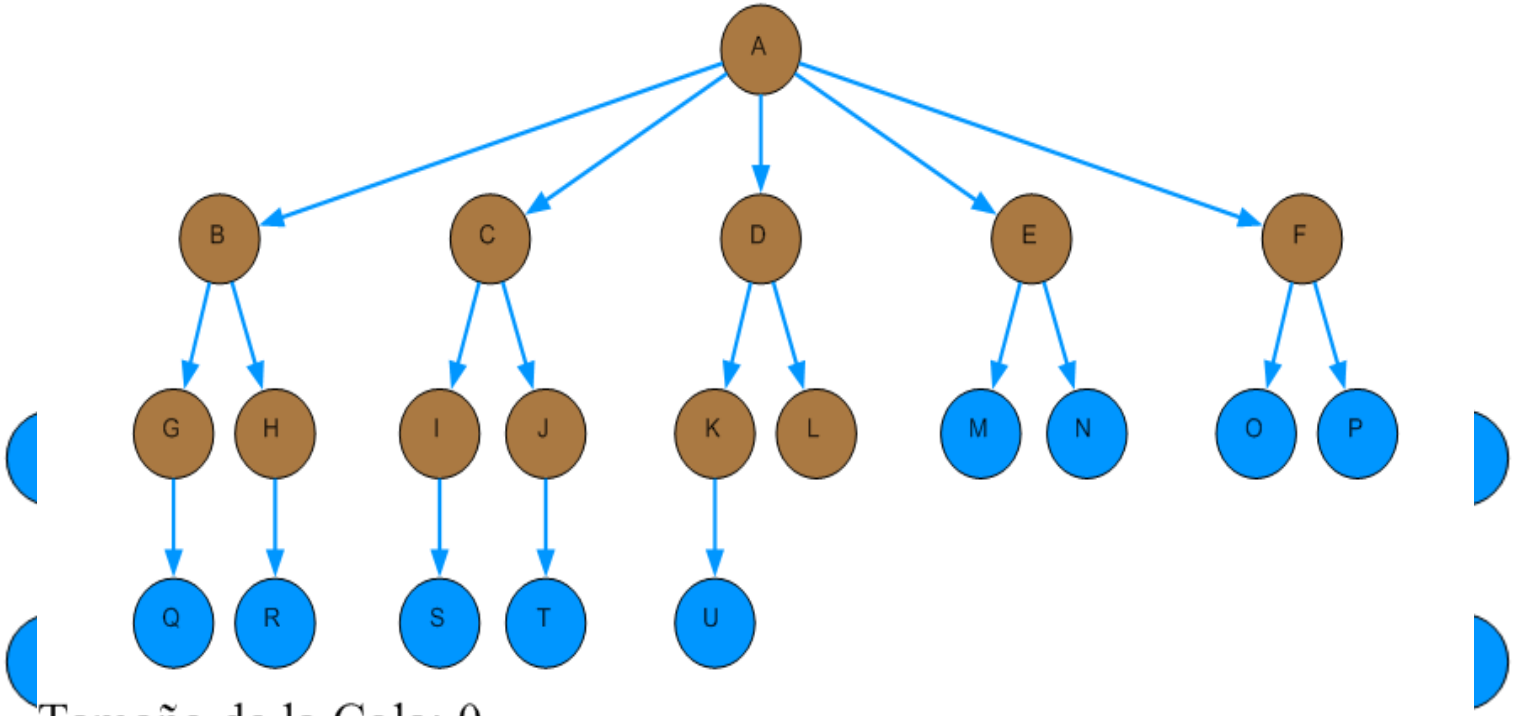
- El tiempo y la cantidad de memoria necesaria crece exponencialmente con respecto a la profundidad.

Es sub-óptima y completa.

Por extensión o amplitud



Por extensión o amplitude o a lo Ancho



Tamaño de la Cola: 0
Nodos Expandidos: 11

Cola: vacía

Búsqueda por profundidad

Siempre se expande uno de los nodos que se encuentren en los mas profundo del árbol.

Solo si la búsqueda conduce a un callejón sin salida, se revierte la búsqueda y se expanden los nodos de niveles menos profundos.

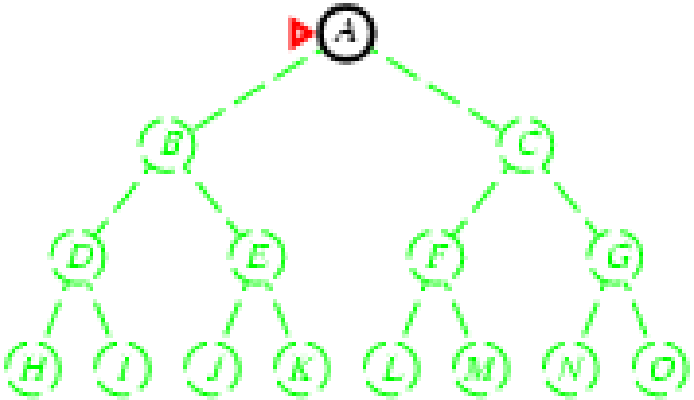
Esta búsqueda,

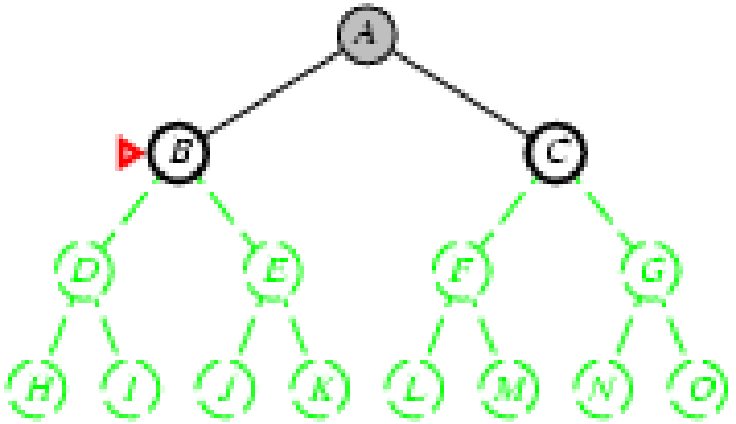
- o se queda atorada en un bucle infinito, y nunca es posible regresar al encuentro de una solución,
- o encuentra una ruta de solución más larga que la solución óptima.

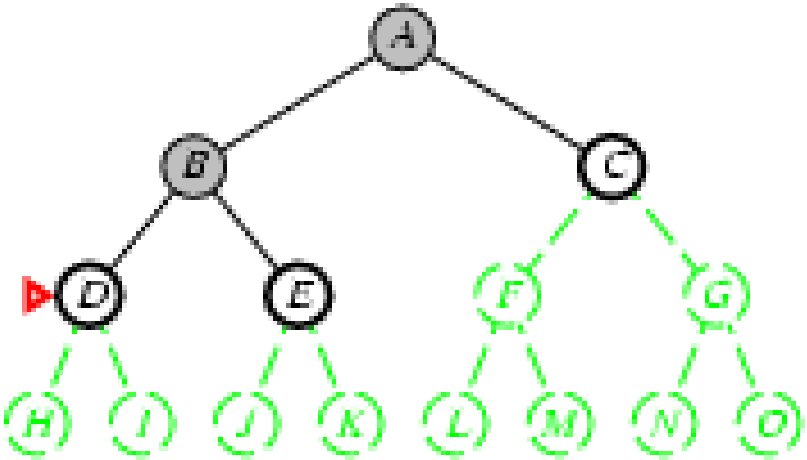
El tiempo necesario crece exponencialmente con respecto a la profundidad, mientras que el espacio requerido en memoria lo hace en forma lineal

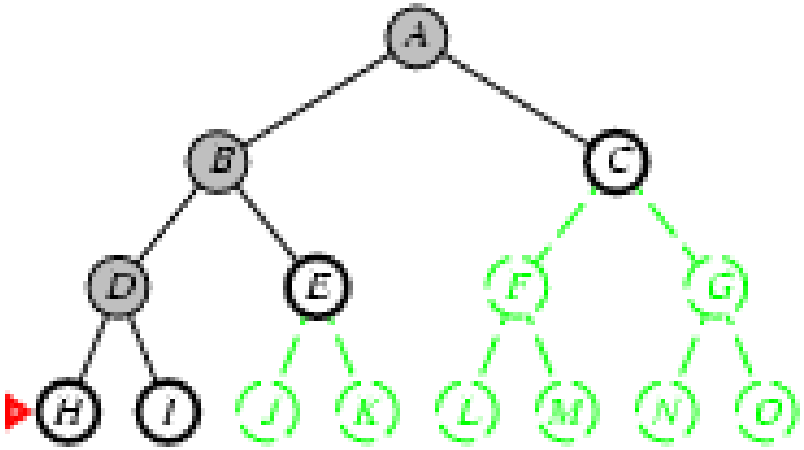
No es óptima pero completa.

Por profundidad

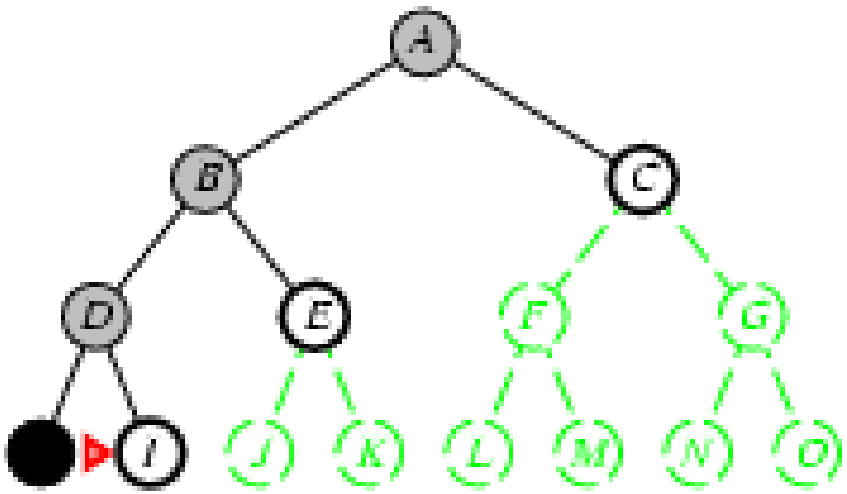


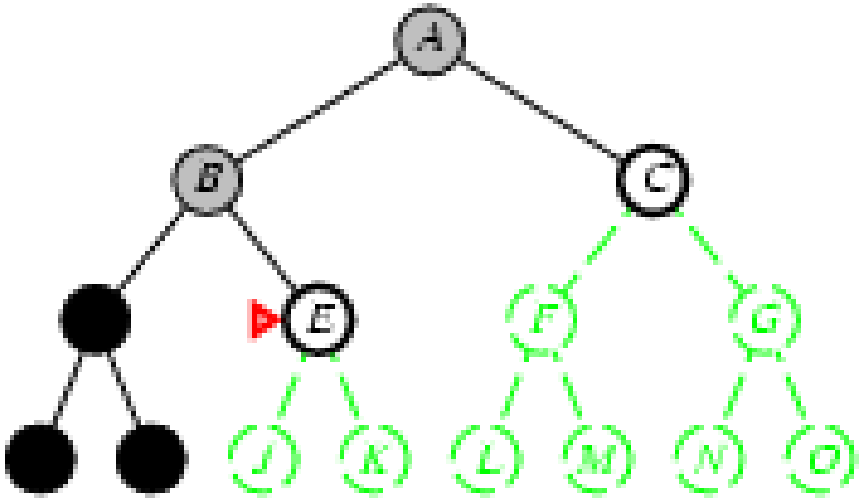


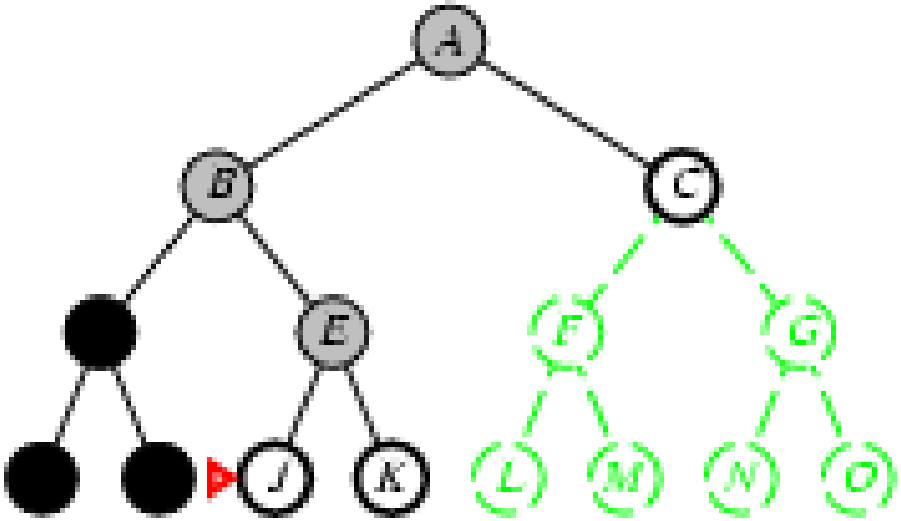


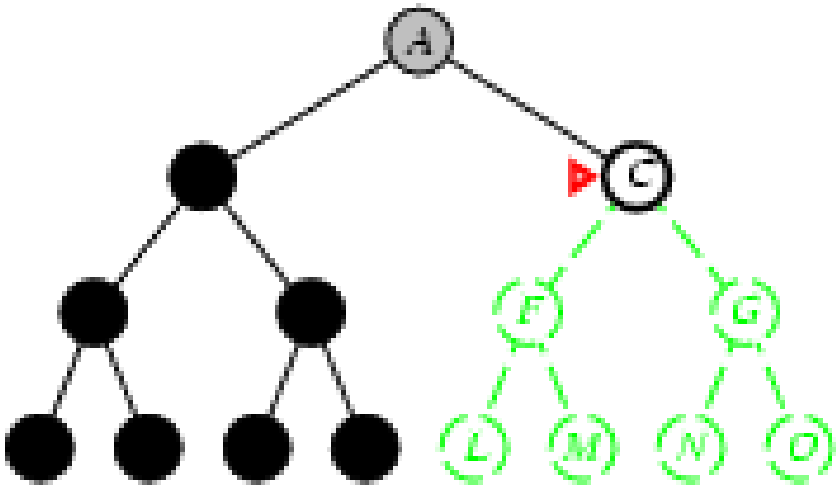


Por profundidad

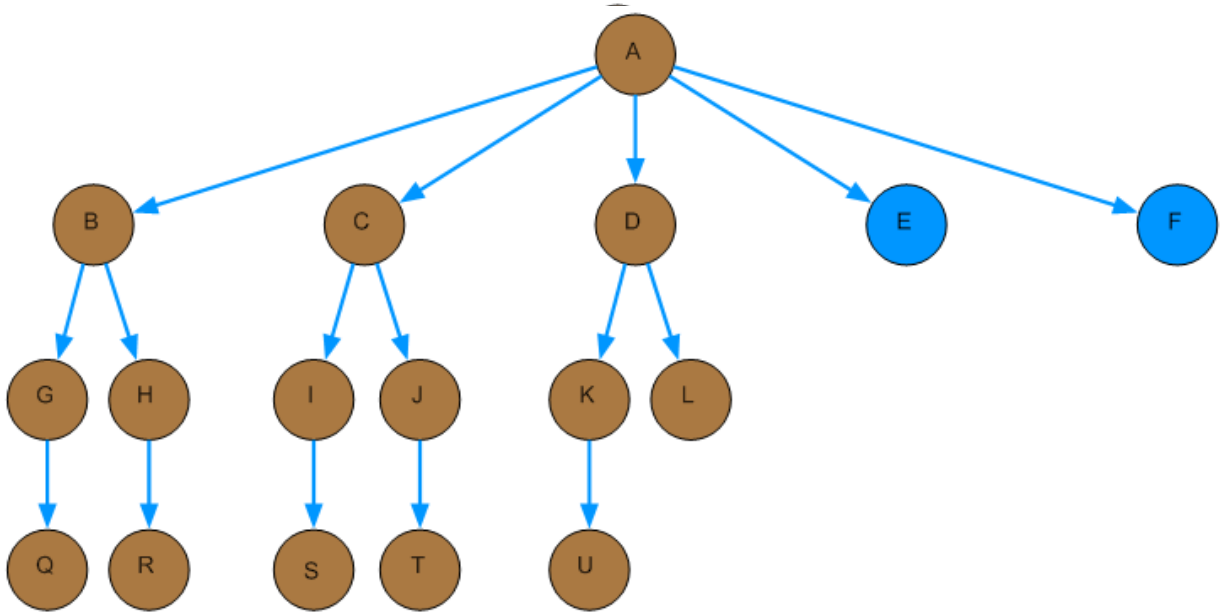








Búsqueda en Profundidad



Tamaño de la Cola: 0
Nodos Expandidos: 15

Cola: vacía

Búsqueda por profundidad limitada

Es similar a la búsqueda por profundidad, con la diferencia que se impone un límite a la profundidad máxima de una ruta.

Se utilizan operadores que informan constantemente de la profundad del nodo:

- El tiempo necesario crece exponencialmente con respecto a la profundidad,
- El espacio requerido en memoria lo hace en forma lineal

No es óptima, pero si completa cuando la profundidad del límite es menor o igual a la profundidad de la solución.

Por profundidad limitada

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Condición de parada

recursividad

Búsqueda por profundización iterativa

Similar a la búsqueda limitada por profundidad con la diferencia que se repiten las búsquedas dando en cada iteración un valor distinto de profundidad para la misma.

- el tiempo necesario crece exponencialmente con respecto a la profundidad,
- el espacio requerido en memoria lo hace en forma lineal

Es óptima y completa.

profundización iterativa

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

inputs: *problem*, a problem

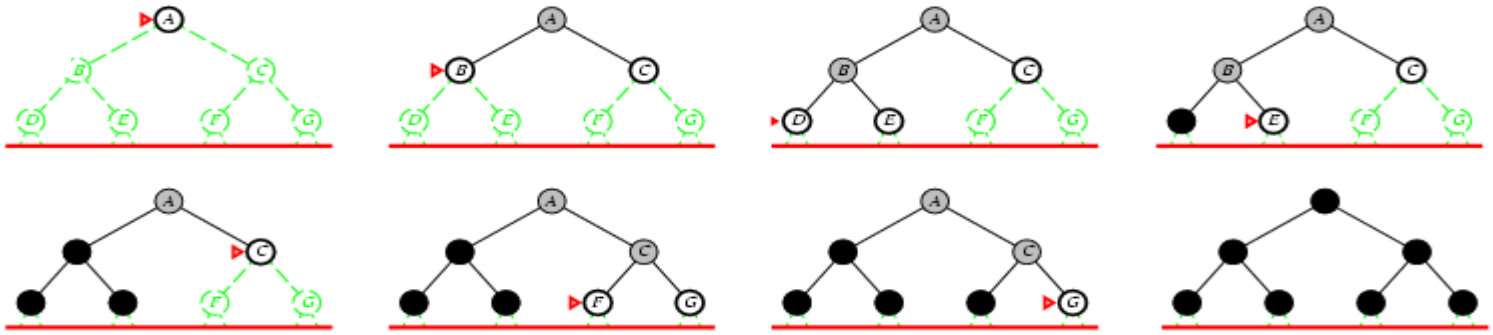
for *depth* ← 0 **to** ∞ **do**

result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)

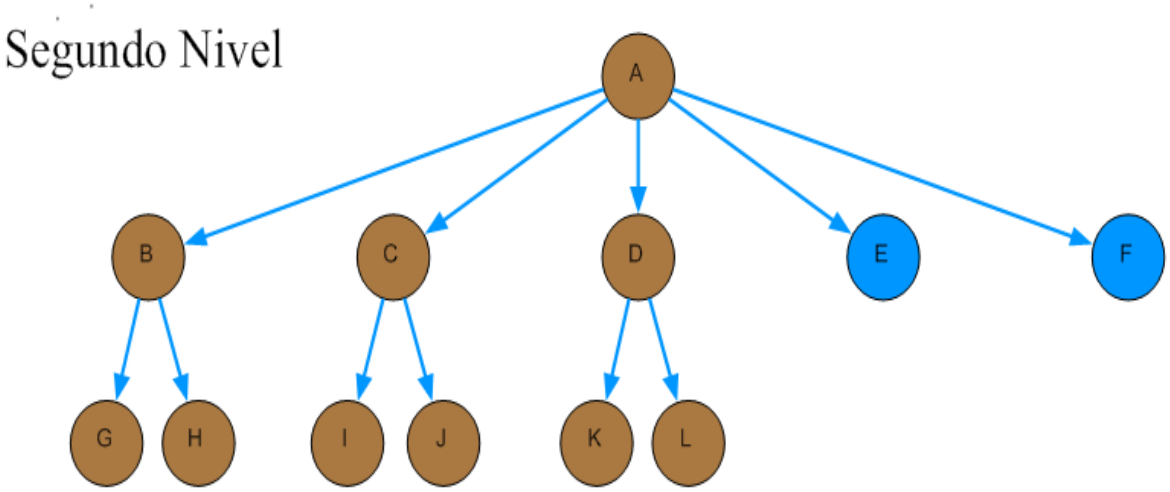
if *result* ≠ cutoff **then return** *result*

Llamada a algoritmo anterior

limite=2



Búsqueda en Profundidad Iterativa



Tamaño de la Cola: 0
Nodos Expandidos: 17

Cola: vacía

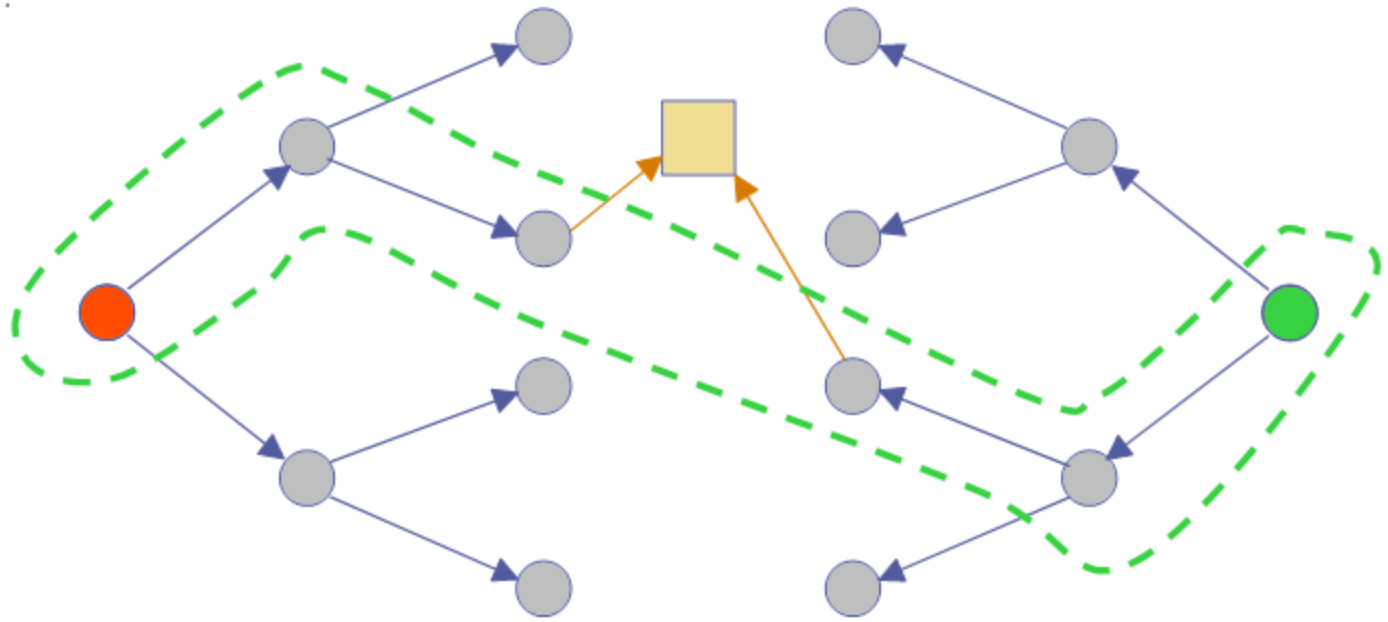
Búsqueda bidireccional

Búsqueda que avanza a partir del estado inicial y que retrocede a partir de la meta, y que se detiene cuando ambas búsquedas se encuentran en algún punto intermedio.

El tiempo y el espacio requerido en memoria crecen exponencialmente con respecto a la mitad de la profundidad (bd/2).

Es óptima y completa.

Búsqueda Bidireccional



Agentes y Búsqueda

	t	espacio	Opt	Comp
– Por extensión	b^d	b^d	no	si
– Uniforme	b^d	b^d	si	si
– Por Profundidad	b^m	b^m	no	si
– Profundidad limitada	b^l	b^l	no	quizás
– Profundidad Iterativa	b^d	b^d	si	si
– Bidireccional	$b^{d/2}$	$b^{d/2}$	si	si

b: factor de ramificación

d: profundidad de la solución

m: máxima profundidad del árbol

l: limite de profundidad

Algoritmos de Búsqueda Informados

- Usan **funciones de evaluación** y se detienen al conseguir una *buena* solución

- Algoritmos heurísticas
  - Minimizar costo estimado para alcanzar objetivo desde donde estoy
  - Ejemplo: Algoritmo de Búsqueda "*Primero el Mejor para expandir*"

- **Clasificación**
  - Iterativo (Descenso de Gradiente, Algoritmos Genéticos, Temple Simulado) vs. Constructivo (Primero el Mejor)
  - Búsqueda Local (Temple Simulado) vs. Búsqueda Global (Algoritmos Genéticos)

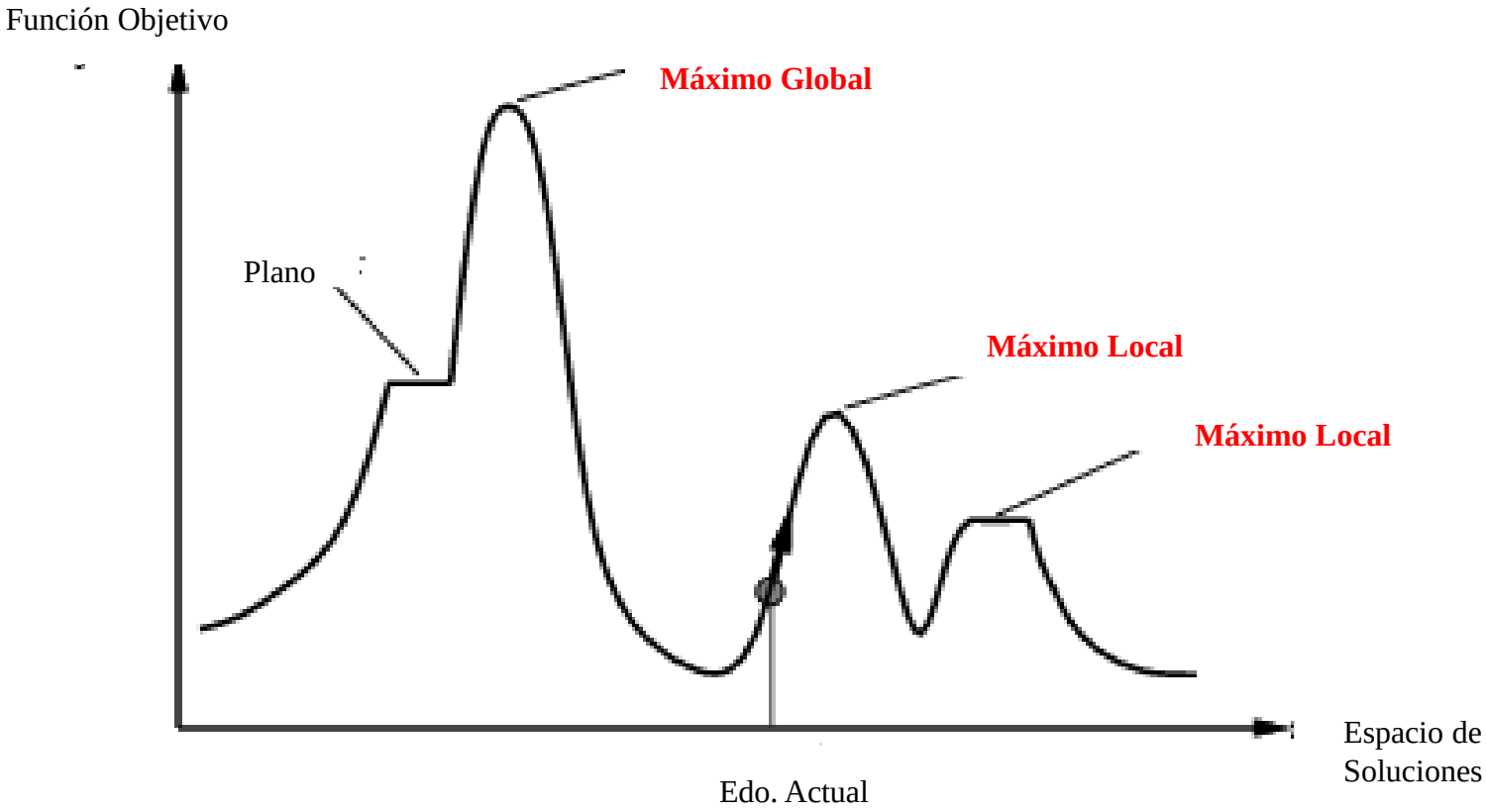
Agentes y Búsqueda

- Algoritmo Heurística General (**CONTRUCTIVO**)
  - Crear árbol de búsqueda solo con nodo raíz
  - Crear una lista de vecinos al nodo actual para expandir
  - Seleccionar cada nodo de la lista y evaluar solución parcial (uso **Función de Evaluación**)
  - Escoger para **expandir mejor sucesor** y repetir si no es edo. objetivo

- Hillclimb (**ITERATIVO**: mejora solución inicial)
  - Actual = Solución-Inicial(problema)
  - Lazo hasta converger
    - próximo= solución vecina a la actual (problema)
    - Si valor(actual) >valor(próximo)
      - actual=próximo

Búsqueda Local

- Problema: cuidado con los máximos locales



Primero el Mejor (constructivo)

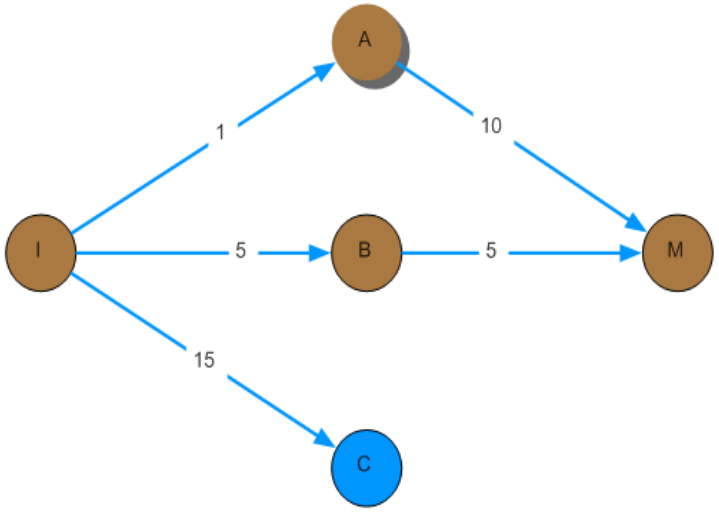
- Idea: usar una función de evaluación *f(n) p*or nodo
  - Estima que tan "deseable“ es

- Algoritmo:
- Tomar nodo actual
  - **Evaluar** los nodos sucesores
  - Si alguno es el estado final
    - Detenerse
  - de lo contrario
    - Expandir nodo no expandido **mas deseable**
  - Regresar al paso inicial hasta no tener a quien expandir

Búsqueda de Costo Uniforme
Primero el Mejor (constructivo)

- Variación de la búsqueda extendida pero basado en **Camino mas barato**
- Cada camino tienen asociado un costo
- Siempre se expande el nodo actual con el menor costo
- Consigue solución sub-optima

Búsqueda de Costo Uniforme



Tamaño de la Cola: 0
Nodos Expandidos: 3

Cola: vacía

Primero el Mejor

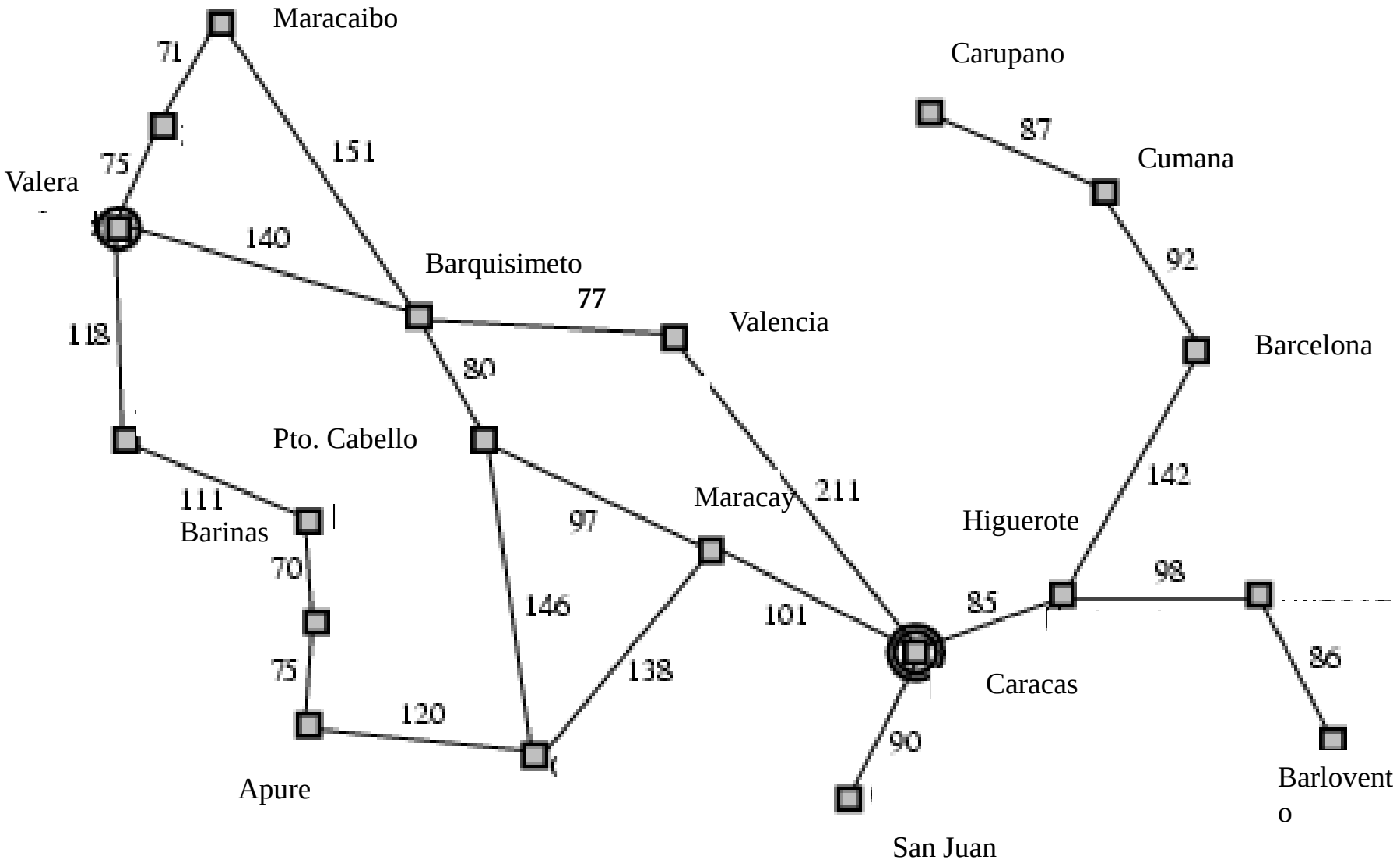
Ejemplo: problema del viajero

- **Función de evaluación** *f(n)*= estima costo desde *n* hasta *objetivo*

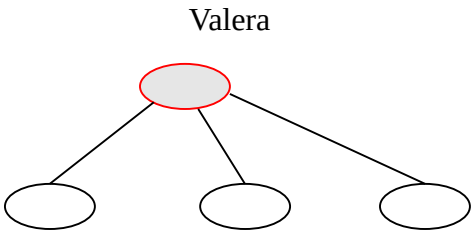
  Por ejemplo: *f(n)* = distancia desde *n* a Caracas

- Algoritmo **expande nodo** que parece estar mas cerca del objetivo

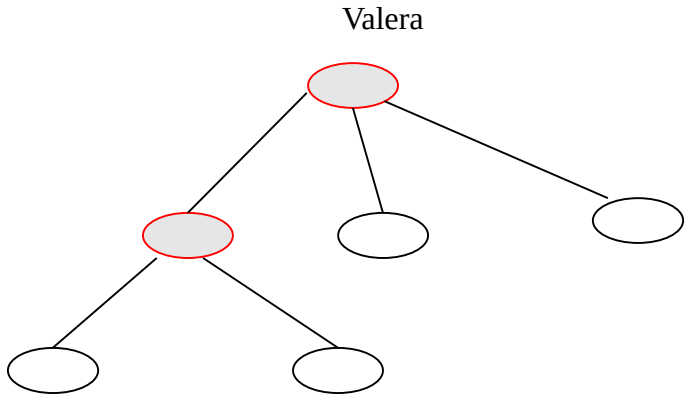
Ejemplo: Venezuela



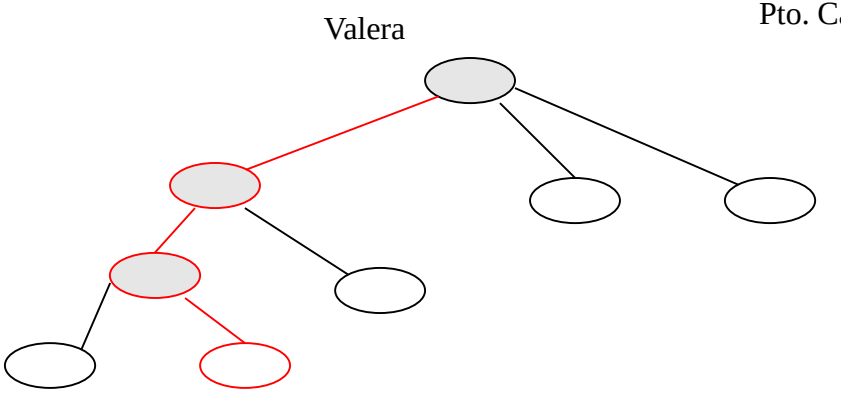
Solución



Bqto: 288 Macbo: 549 Barinas: 504



Pto. Cabello: 197 Valencia: 211



Caracas 0

Búsqueda Avara

h(nodo).

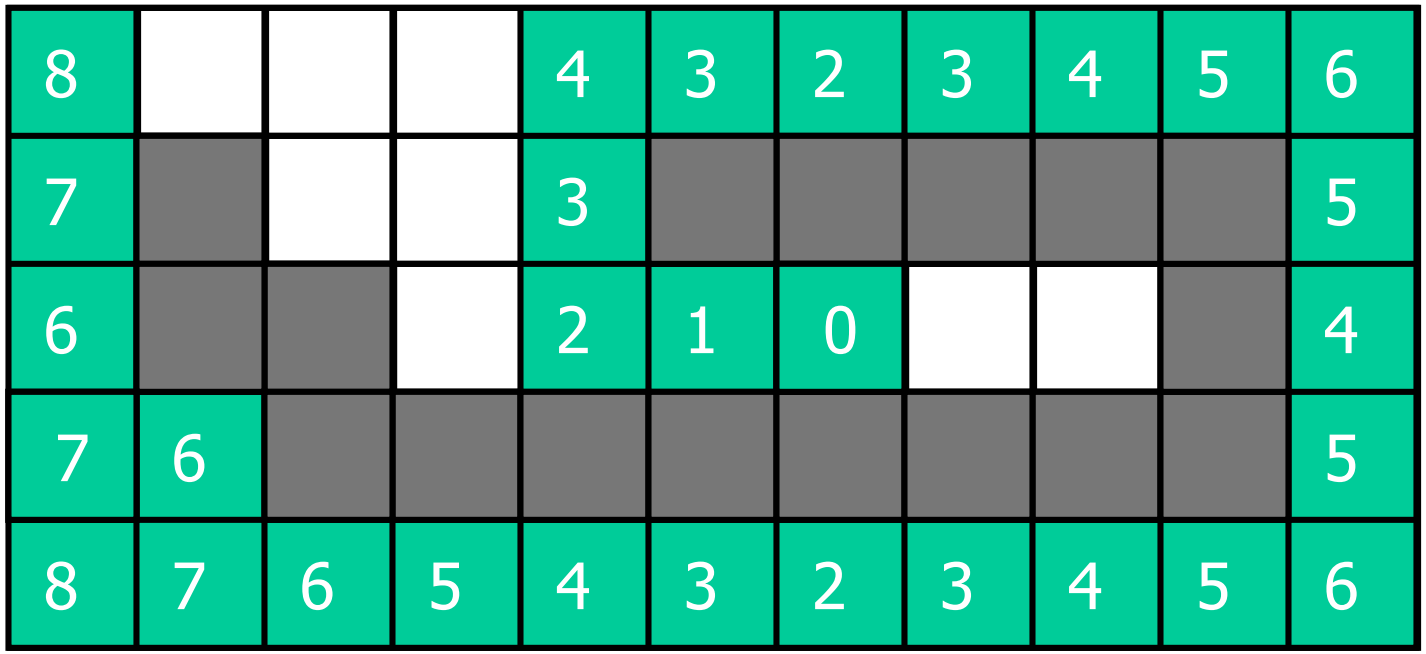
- Función heurística (h)

- h(nodo) = costo estimado del camino más corto desde un nodo hasta el objetivo

- Todas las funciones heurísticas deben cumplir al menos:
  - h(n) ≥0, para todo nodo n
  - h(n) = 0, si “n” es un nodo objetivo

Navegación de un Robot

f(N) = h(N), con h(N) = distancia Manhattan a la meta



Búsqueda más Informada

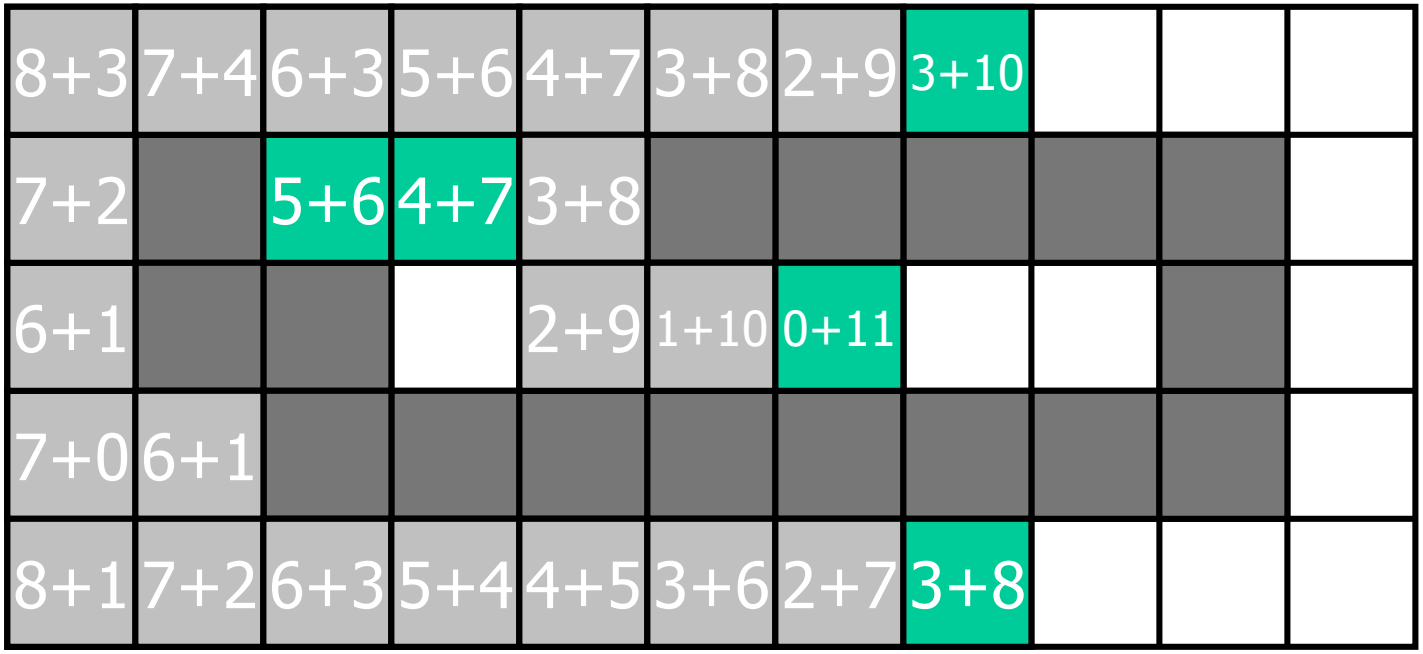
nodo en el camino hasta la meta, lo que queremos es minimizar el largo global del camino a la meta.

- Dado g(N) como el costo del mejor camino encontrado hasta el momento entre el nodo inicial y N

- f(N) = g(N) + h(N)

Navegación de un Robot

f(N) = g(N)+h(N), con h(N) = distancia Manhattan a la meta



Búsqueda en Ascenso a la Cima

- Seguir el camino que parece estar mejorando más rápidamente.
- Continuar mientras que el camino siga mejorando.
- Necesita de una función local que indique que tan bueno es el camino, conocida como función de evaluación (similar al f en el mejor primero)
- Nunca retrocede
- Rápido, pero no garantiza encontrar una solución
- La solución encontrada no es necesariamente óptima.
- Funciona si el camino a la solución es monotónico

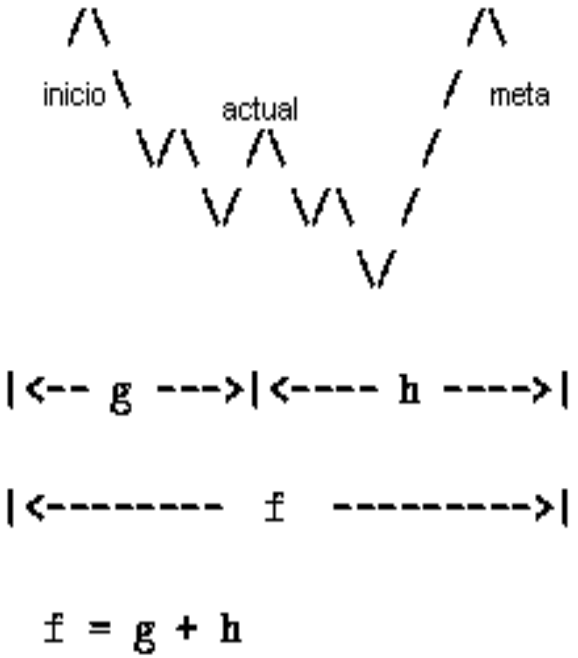
Búsqueda en Ascenso a la Cima

- Se escoge un nodo de la lista de sucesores del último nodo expandido con el valor mínimo de f (una vez que se decide expandir un nodo, solo considere sus sucesores y nunca se trata un camino alternativo)

- Problemas:
- Puede pararse en mínimos locales
- Planicies no hay mejoría local en f, por lo que se puede andar sin rumbo.

Búsqueda A*

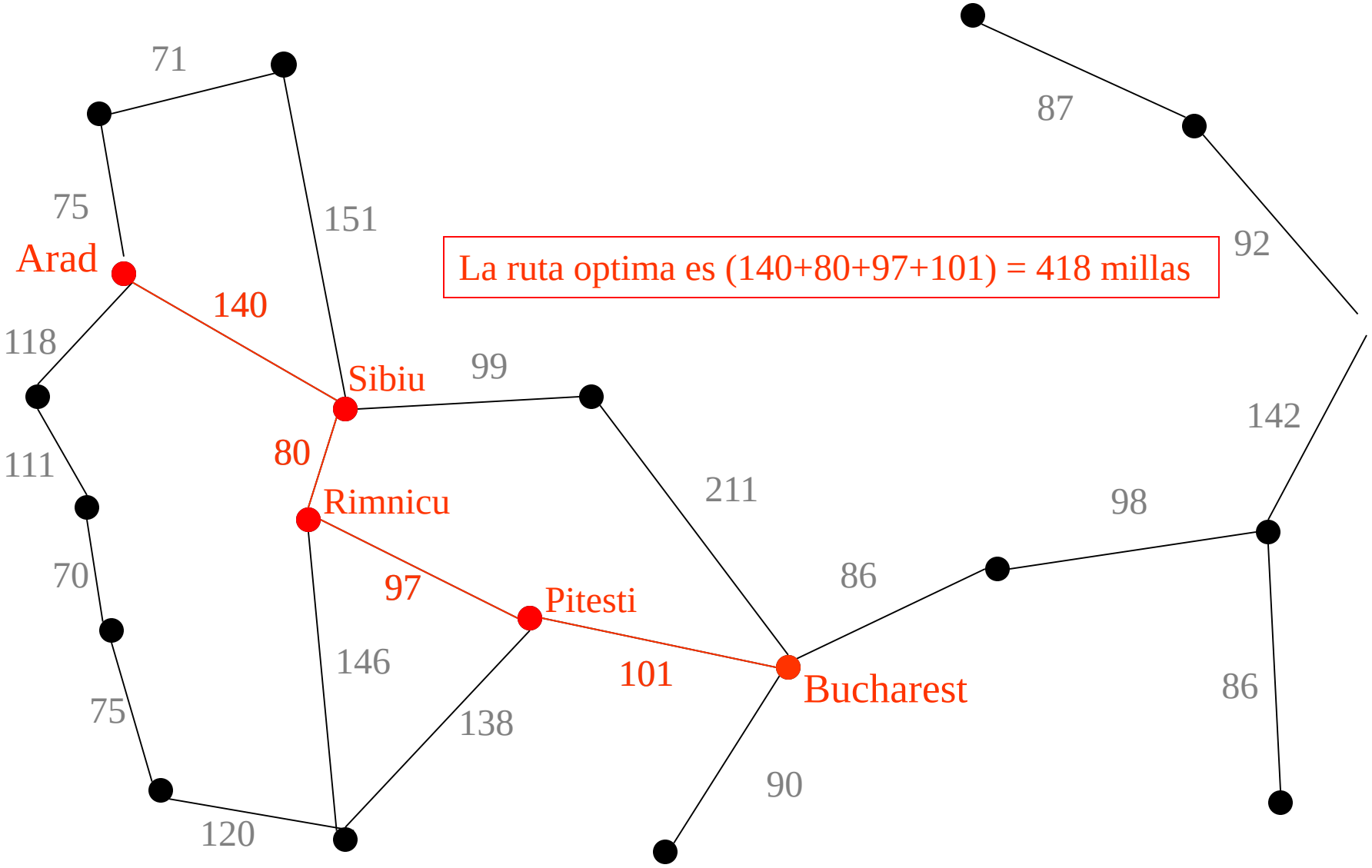
- Búsqueda óptima para solución óptima
- Parecida a el mejor primero, con:
  - f(n) = g(n) + h(n)
- Donde:
  - g(n) = costo mínimo del camino desde el nodo inicial al nodo n
  - h(n) = costo mínimo estimado desde el nodo n al nodo meta



Búsqueda A*

- Se requiere conocer el costo de cada uno de los movimientos.
- Se requiere una función de evaluación heurística (h) para juzgar que tan difícil es llegar desde el nodo actual al nodo final.
- g se conoce exactamente mediante la suma de todos los costos del camino desde el inicio hasta el nodo actual (costo uniforme)
- Se utiliza la distancia euclidiana en línea recta como heurística para h

Ejemplo Búsqueda A*

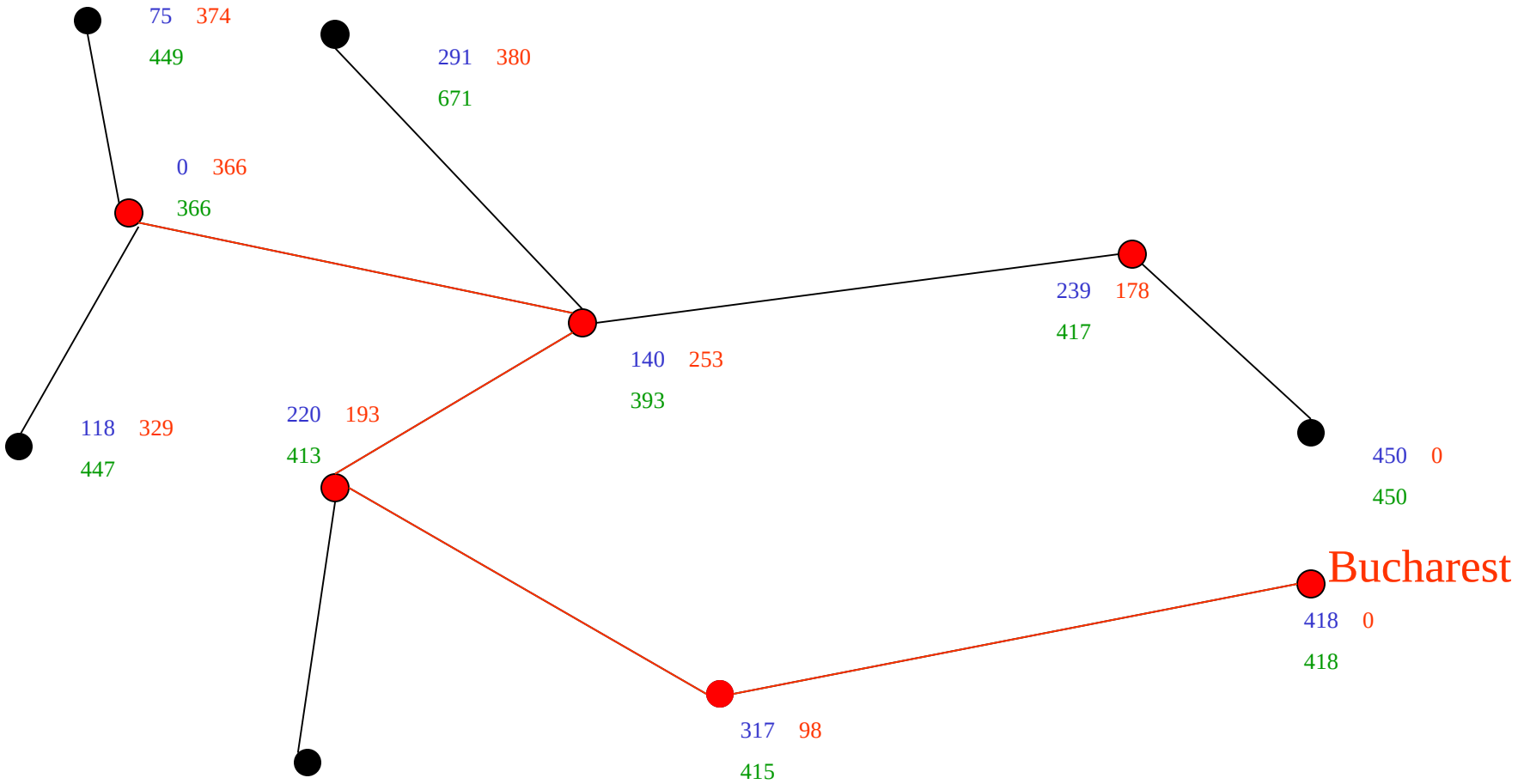


Distancia en Línea Recta a Bucarest

Ciudad	DLR
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244

Ciudad	DLR
Mehadai	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Resultado Búsqueda A*



Algoritmo A*

1) N lista de todos los nodos iniciales

2) Si N esta vacío, no hay solución

3) Seleccione n en N tal que f(n) = g(n) + h(n) es mínimo

4) Si n es un nodo meta, fin meta encontrada

5) De lo contrario remueva n de N, y agregue todos los hijos de

Recocido Simulado

- Es un método de descenso de gradiente
- Usa aleatoriedad para remover los problemas de mínimos locales.
- La temperatura es disminuida en el tiempo: a mayor T, mayor la aleatoriedad de la selección del sucesor

Un **esquema de enfriamiento** que regula cómo va disminuyendo gradualmente la temperatura.

$$T(k) = \frac{T_o}{1 + \ln k}$$

Un algoritmo que se utiliza para encontrar la distribución de equilibrio para cada nuevo valor de la temperatura obtenido por el esquema de enfriamiento.

Temple Simulado (iterativo)

- actual =solución_inicial[problema]
- T= alta temperatura
- repita hasta T= congelado
  - próximo= aleatoria selección desde actual de una solución vecina
  - E= costo[próximo]- costo[actual]
  - Si T= congelado y E> 0 entonces ← **parada**
    - regresar actual

  de lo contrario

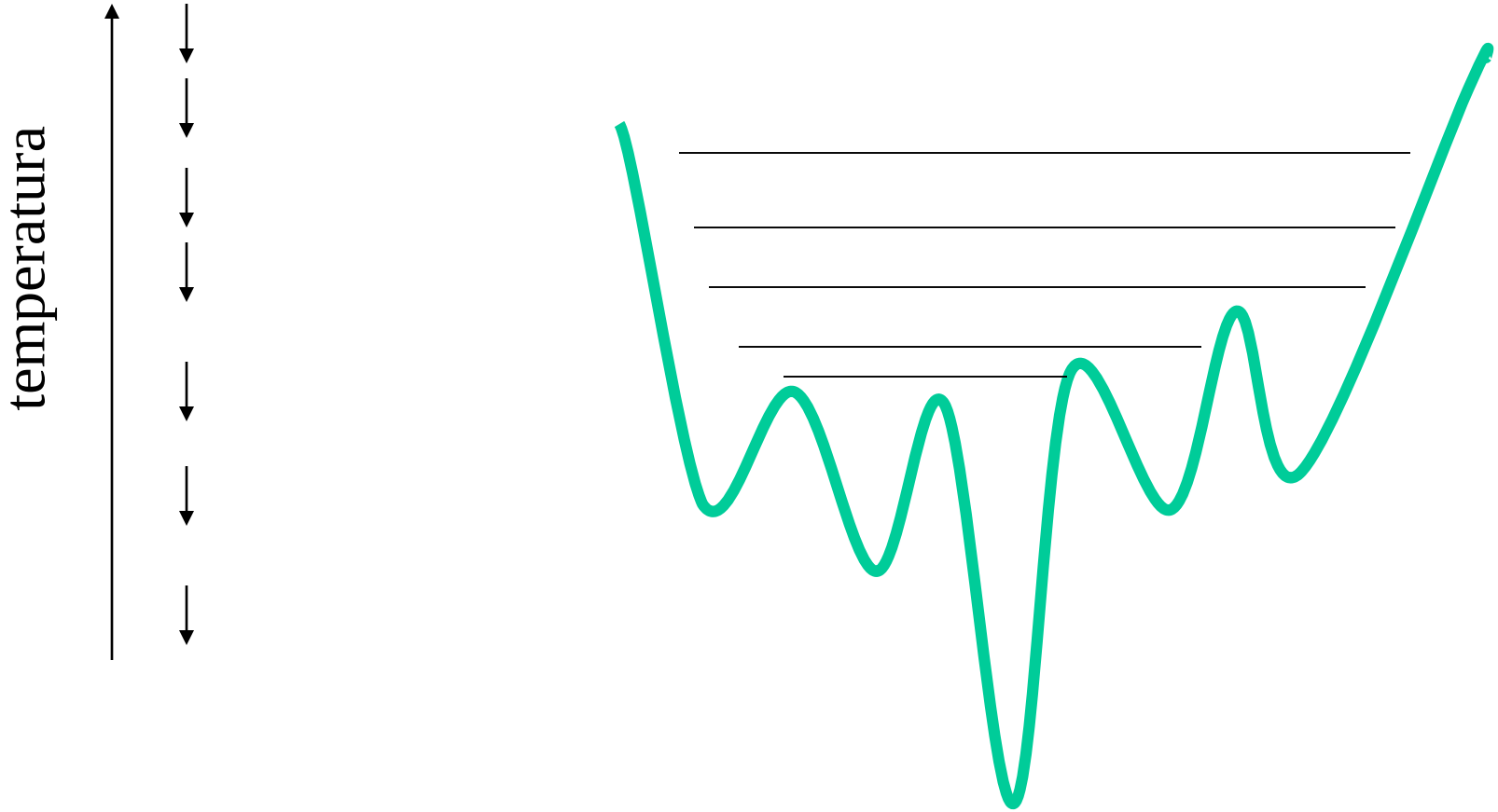
    - Si E>0 entonces
      - actual= próximo con probabilidad $e^{E/T}$
    - Si E< 0 entonces
      - actual= próximo

Actualiza solución

  - descender T si ya han sido aceptados un número dado de movimientos para ese T

Enfriamiento (equilibrio) →

Búsqueda del Mínimo Global : Recocido Simulado



COMPUTACION EVOLUTIVA (iterativo)

- ENFOQUES DE BUSQUEDA Y APRENDIZAJE BASADOS EN MODELOS COMPUTACIONALES DE PROCESOS EVOLUTIVOS
- BUSQUEDA ESTOCASTICA,
  - EVOLUCIONA UN CONJUNTO DE *ESTRUCTURAS*
  - SELECCIONA DE MODO ITERATIVO LAS *MAS APTAS*

***FINALIDAD*:** SUPERVIVENCIA DEL MAS APTO

***MODO*:** ADAPTACION AL ENTORNO

COMPUTACION EVOLUTIVA

- OBJETIVOS CONFLICTIVOS QUE SE SIGUEN:
  - EXPLOTAR LAS MEJORES SOLUCIONES
  - EXPLORAR EL ESPACIO DE BUSQUEDA
- PARADIGMAS:
  - ALGORITMOS GENETICOS (HOLLAND)
  - PROGRAMAS EVOLUTIVOS (MICHALEWICZ)
  - PROGRAMACION GENETICA (KOZA)
  - ESTRATEGIAS EVOLUTIVAS (RECHENBERG SCHWEFEL)
  - PROGRAMACION EVOLUTIVA (FOGEL)

COMPUTACION EVOLUTIVA

MACROALGORITMO:

1.- POBLACION INICIAL DE INDIVIDUOS

2.- EVALUACION DE LOS INDIVIDUOS

3.- REPRODUCCION (generación de nuevos individuos)

4.- REEMPLAZO DE INDIVIDUOS

5.- CONDICION DE FINALIZACION O REGRESA A PASO 2

COMPUTACION EVOLUTIVA

ASPECTOS DE IMPLEMENTACION:

- REPRESENTACION DE LOS INDIVIDUOS
- MEDIDA DE ADAPTACION (FUNCION DE EVALUACION DE LOS INDIVIDUOS)
- MECANISMOS DE SELECCIÓN Y REEMPLAZO
- INTERPRETACION DE LOS RESULTADOS
- PARAMETROS Y VARIABLES QUE CONTROLAN EL PROCESO
- OPERADORES GENETICOS A UTILIZAR

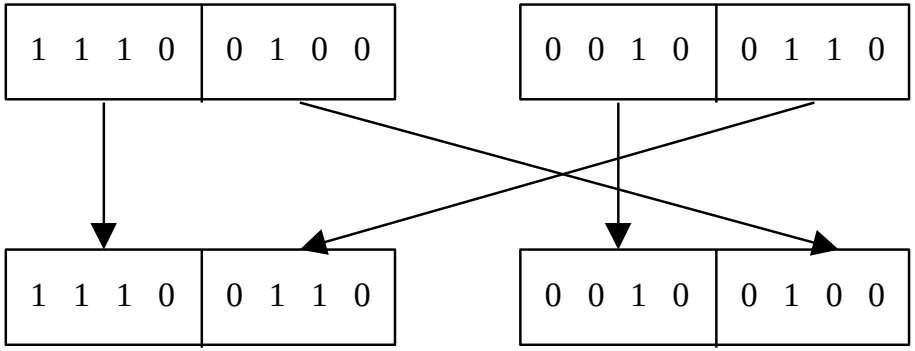
Operadores Genéticos

- Mecanismos de transformación que se utilizan para recorrer el espacio de las soluciones de un problema.

  realizan los cambios de la población durante la ejecución de un algoritmo evolutivo.

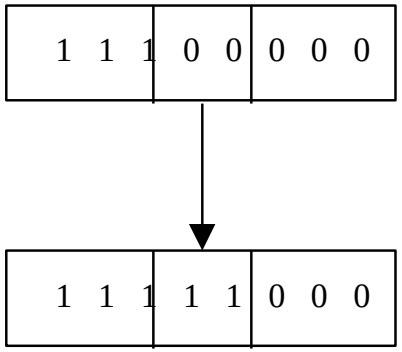
- En general, se basan en los operadores genéticos biológicos.
- Operadores genéticos clásicos:
  - Mutación
  - Cruce

Operador de Cruce



- Algunos Tipos:
  - Cruce multipunto
  - Cruce segmentado

Operador de Mutación



- Tipos
  - Mutaciones sobre genes
  - Mutaciones no estacionarias
  - Mutaciones no uniformes

Operadores Avanzados

- Dominación
- Segregación
- Inversión
- Duplicación

Problema del agente viajero (Problema de optimización combinatoria)

"Visitar todas las ciudades importantes de Vzla. por lo menos una vez, comenzando y terminando en Mérida"

- Requiere más información sobre el espacio de estados:
  - Además de la ubicación del agente,
  - un registro de las ciudades visitadas.

- El test objetivo consiste en verificar si el agente está en Mérida y se ha visitado todas las ciudades una sola vez.

- El objetivo es determinar cuál es el recorrido más corto,

es un problema de complejidad NP.

El Agente Viajero

- ***Formalización General:*** Dada N ciudades, el viajero de comercio debe visitar cada ciudad una vez, teniendo en cuenta que el costo total del recorrido debe ser mínimo.

G=(N, A)

N={1, ...,n}: conjunto de *n* nodos (vértices)

A={a_{ij}}: matriz de adyacencia.

d_{ij} = { ∞ Si $a_{ij} = 0$

Lij Si $a_{ij} = 1$

Lij: distancia entre las ciudades i y j.

El Agente Viajero:

Formas de representar soluciones

- Suponiendo que las ciudades son numeradas desde 1 hasta n, una solución al problema puede expresarse a través de una matriz de estado *E*

$$e_{ij} = \begin{cases} 1 & \text{Si la ciudad } j \text{ fue la } i^{esima} \text{ ciudad visitada} \\ 0 & \text{En otro caso} \end{cases}$$

- La matriz E permite definir un arreglo unidimensional V de dimensión *n*;

$v_j = i$ **Si la ciudad i fue la j^{esima} ciudad visitada**

El Agente Viajero

Función objetivo

$$F1 = \sum_{i=1}^{n}\sum_{k=1}^{n}\sum_{j=1}^{n} L_{ik} e_{ij} e_{kj+1}$$

$$F2 = C\left(\sum_{i=1}^{n}\sum_{j=1}^{n}\left(e_{ij} - n\right) + \sum_{i=1}^{n}\sum_{j=1}^{n}\left(e_{ij} - 1\right) + \sum_{j=1}^{n}\sum_{i=1}^{n}\left(e_{ij} - 1\right)\right)$$

$$FC = F1 + F2$$

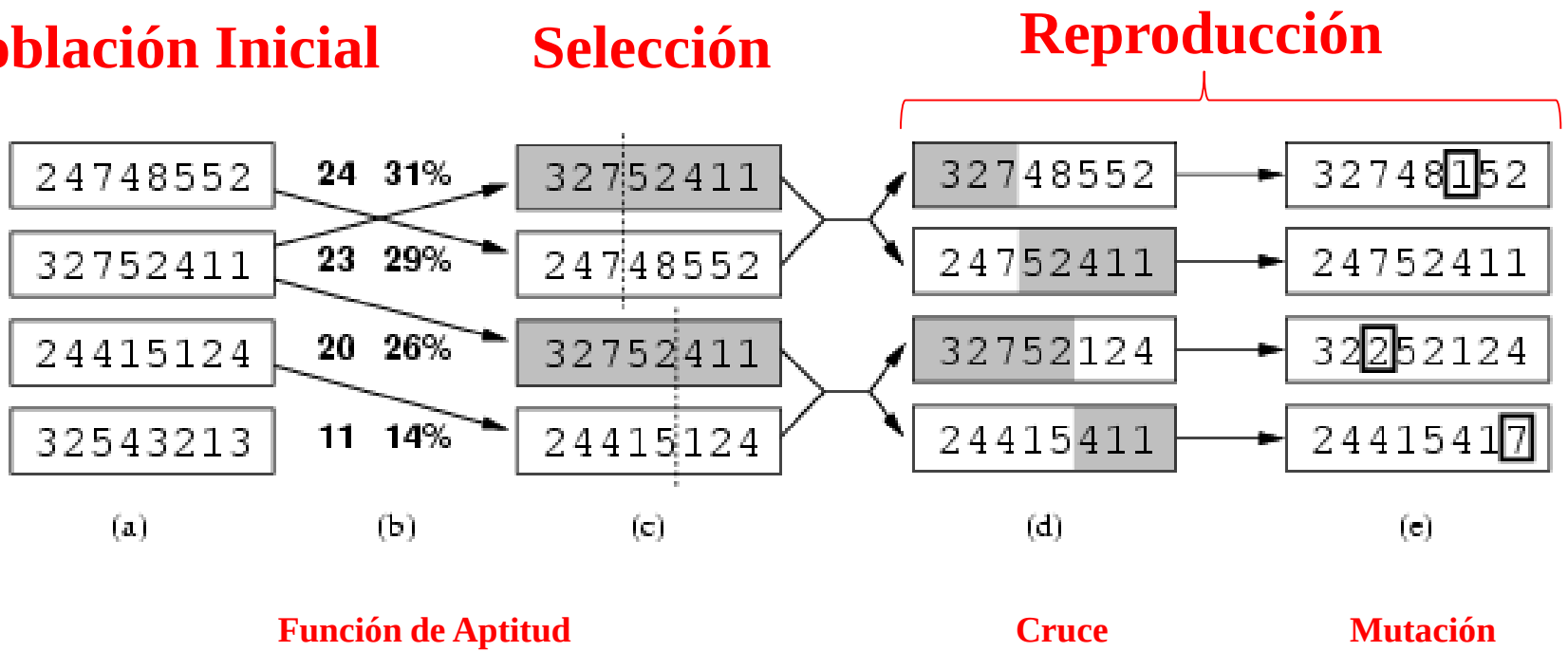
C = n * Max(L_{ik}) factor de penalización.

El Agente Viajero modelado con AG

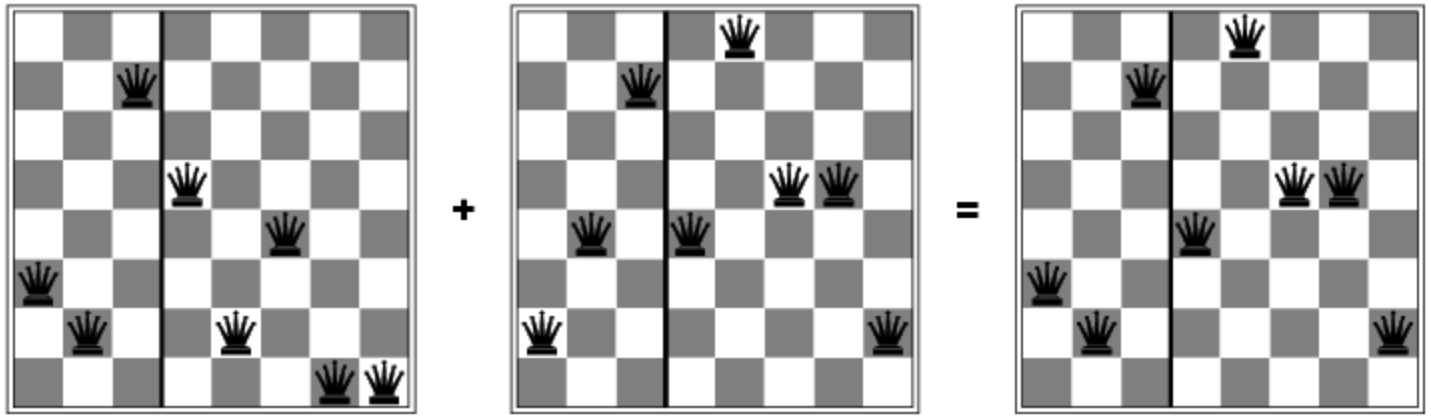
- **Codificación de los individuos:**
  - Una solución viene dada como un recorrido que cumple con las restricciones del problema.
  - Se puede realizar por medio del arreglo unidimensional V de longitud *n*
- **Función Objetivo**: F1 o FC
- **Operadores genéticos:**

  Si es F1 => Inversión.

Algoritmos Genéticos (AGs): evolución



Algoritmos Genéticos (AGs): operadores



Puedo tener operadores particulares a un problema dado

=> Cumplen con las restricciones del problema

Problemas con Restricciones: Constraint satisfaction problems (CSP)

- Componentes:
  - Las variables del problema
  - Los conjuntos posibles de valores de esas variables
  - El conjunto de restricciones que deben ser satisfechas

- Estados definidos por variables X_i con valores desde un conjunto fijo D_i

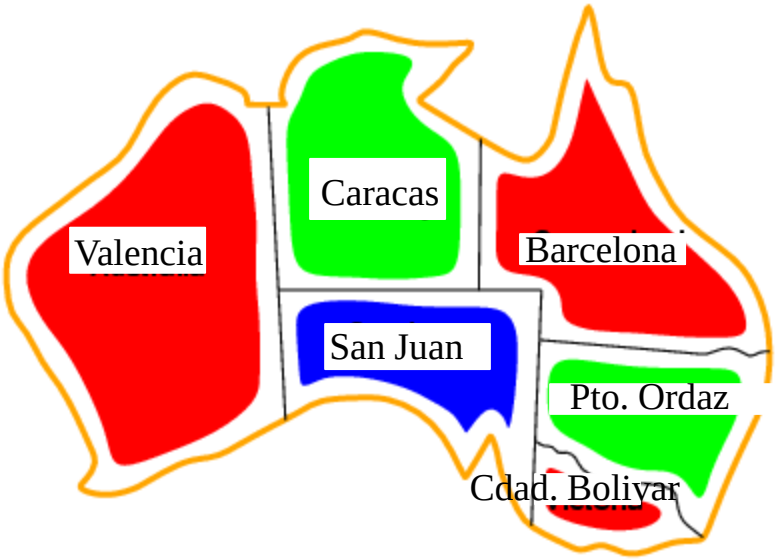
- **Problema de Búsqueda General:**
  - Test objetivo definido por un conjunto de restricciones que indican las combinaciones permitidas en los valores de las variables

Ejemplo: Coloreado de Mapas

- Variables Valencia, Caracas, San Juan, …
- Dominio D_i = {rojo, verde, azul}
- Restricciones: regiones adyacentes deben tener diferentes colores

- Ejm., Valencia ≠ Caracas,
  => (Valencia, Caracas) puede ser {(rojo,verde),(rojo,azul),(verde,rojo),
  (verde,azul),(azul,rojo),(azul,verde)}

Ejemplo: Coloreado de Mapas



- Soluciones deben ser completas y consistentes,
- Ejm., Valencia = rojo, Caracas = verde, Barcelona = rojo, Pto. Ordaz = verde, Cda. Bolivar = rojo, San Juan = azul,

Tipos de Restricciones

- Restricciones Unarias: una simple variable,
  - Caracas ≠ verde
- Restricciones Binarias: pares de variables,
  - Valencia ≠ Caracas
- Restricciones Monarias

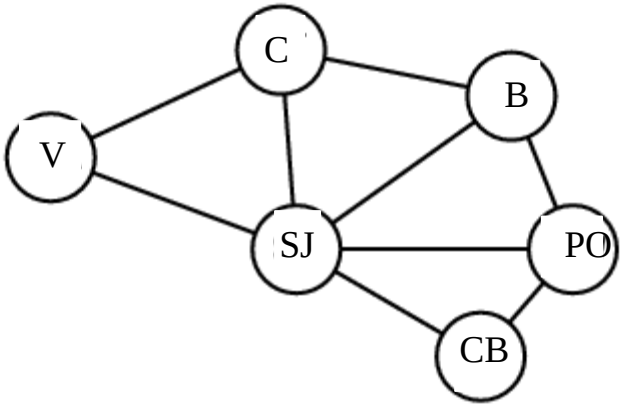
- Restricciones duras y blandas

- Asignación Variable es conmutativa,

p.e., [V = rojo=> C = verde] = [C = verde => V = rojo]

Grafo de Restricciones

- CSP Binario: cada restriccion relaciona 2 variables
- Grafo de Restricciones : nodos son variables, arcos son restricciones



Búsqueda por profundidad en CSP

Se puede retroceder en la busqueda para conseguir solución que satisfaga restricciones (Backtracking)

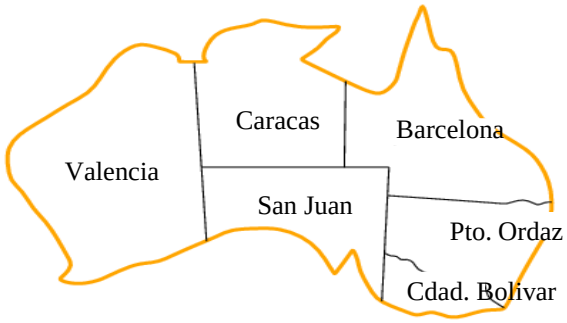
```
function BACKTRACKING-SEARCH( csp) returns a solution, or failure
    return RECURSIVE-BACKTRACKING({}, csp)

function RECURSIVE-BACKTRACKING( assignment,csp) returns a solution, or
failure
    if assignment is complete then return assignment
    var ← SELECT-UNASSIGNED-VARIABLE( Variables[csp], assignment, csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment according to Constraints[csp] then
            add { var = value } to assignment
            result ← RECURSIVE-BACKTRACKING(assignment, csp)
            if result ≠ failue then return result
            remove { var = value } from assignment
    return failure
```

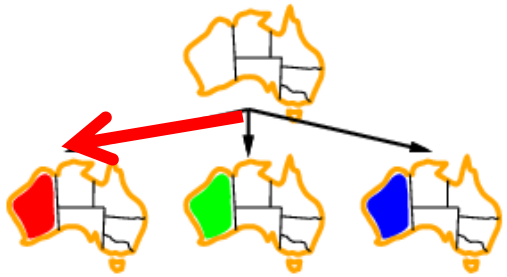
retrocede

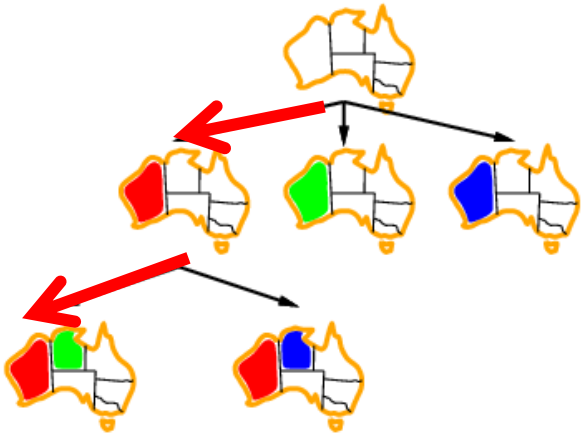
parada

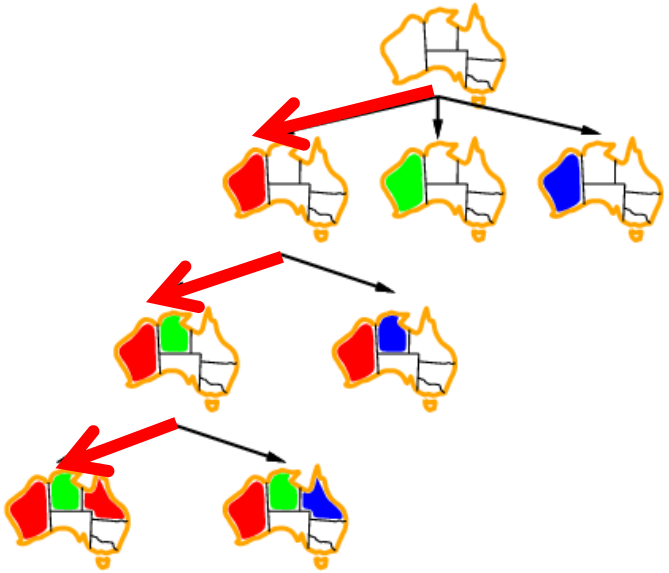
recursividad

Búsqueda por profundidad en CSP



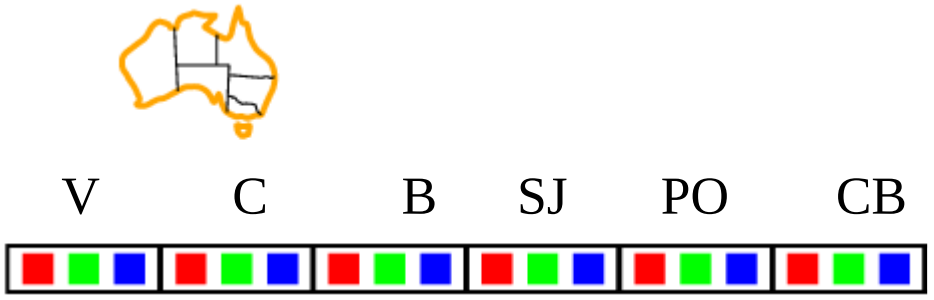




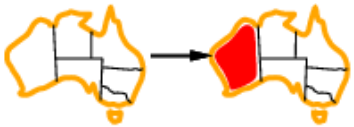


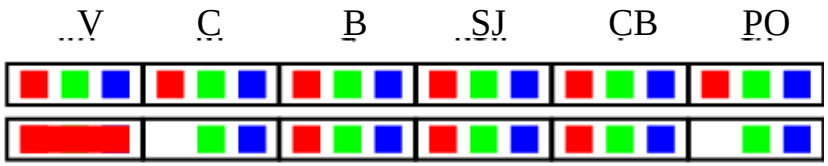
Chequeo Hacia Atras

- Ideas:
  - Guardar **traza de valores legales remanentes** para variables no asignadas
  - **Terminar búsqueda** en una rama cuando cualquier variable tenga valores no legales

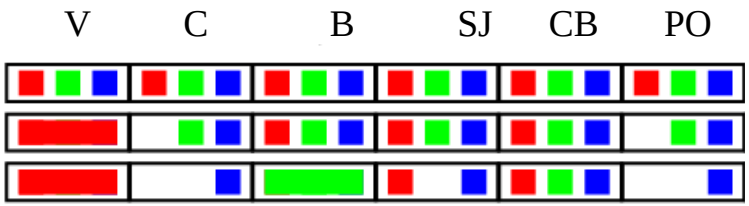


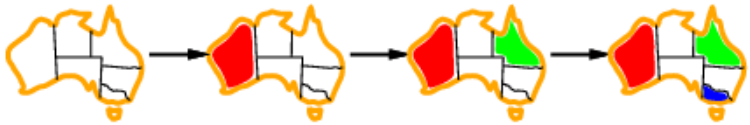
Chequeo Hacia Atras

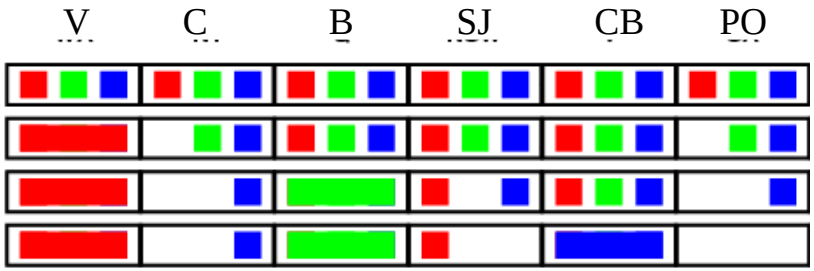




Recorrido de una rama por profundidad







Busqueda Local para CSPs

- Hill-climbing, AG, Temple simulado trabajan típicamente con estados "completos“, es decir, todas las variables asignadas
- Para aplicar a CSPs:
  - Permitir estados que no satisfagan (violen) restricciones
- Selección de variables: escoger aleatoriamente cualquiera.
- Seleccionar valores que violen menos las restricciones. **Se penaliza en FO**
  - operador reasignar valores a variables

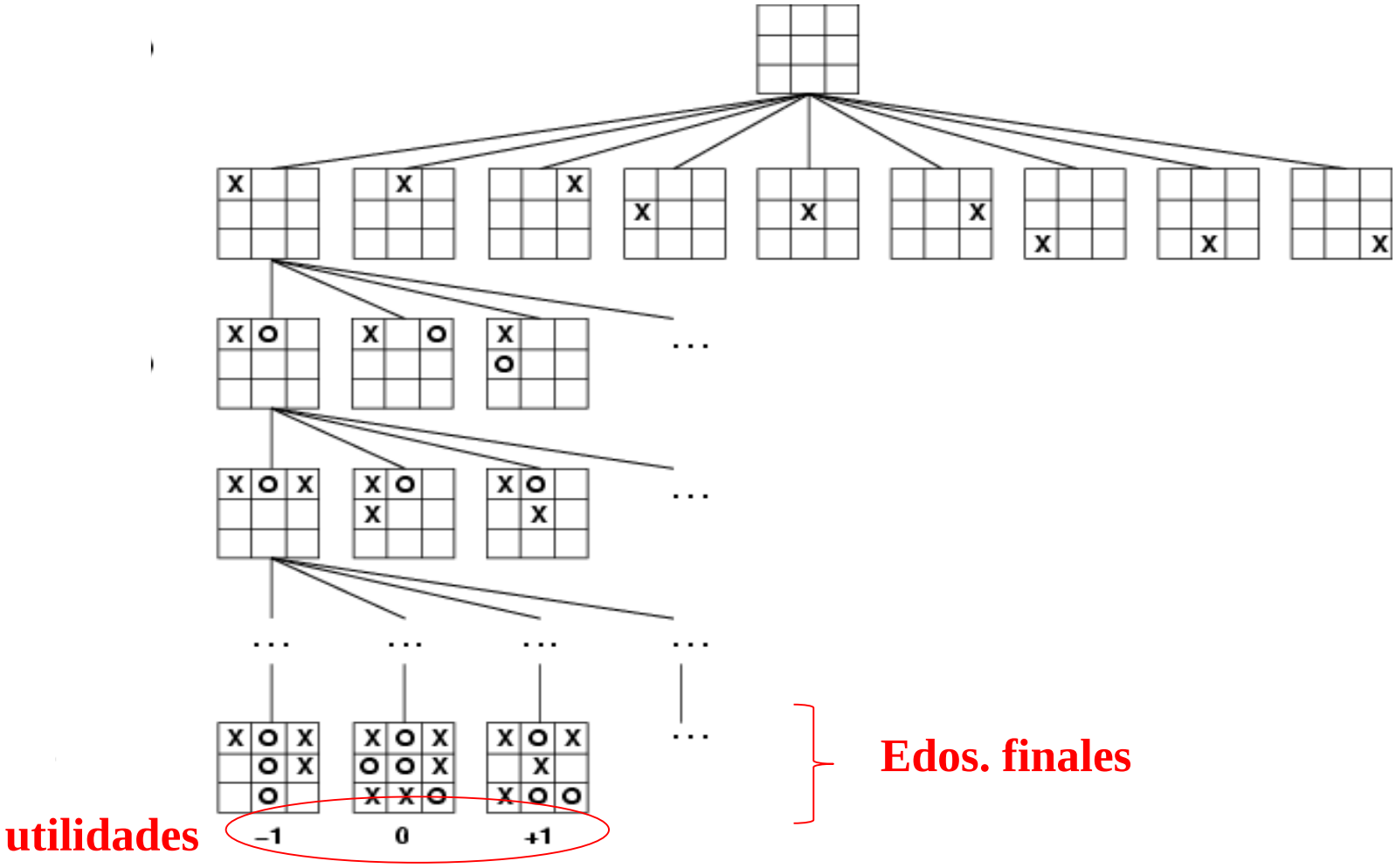
Juegos vs. Problema de búsqueda

- Juego gran similitud con búsqueda
- Oponente "imprevisto" → se debe especificar un movimiento por cada posible replica del oponente
- Limite de tiempo

Juegos vs. Problema de búsqueda

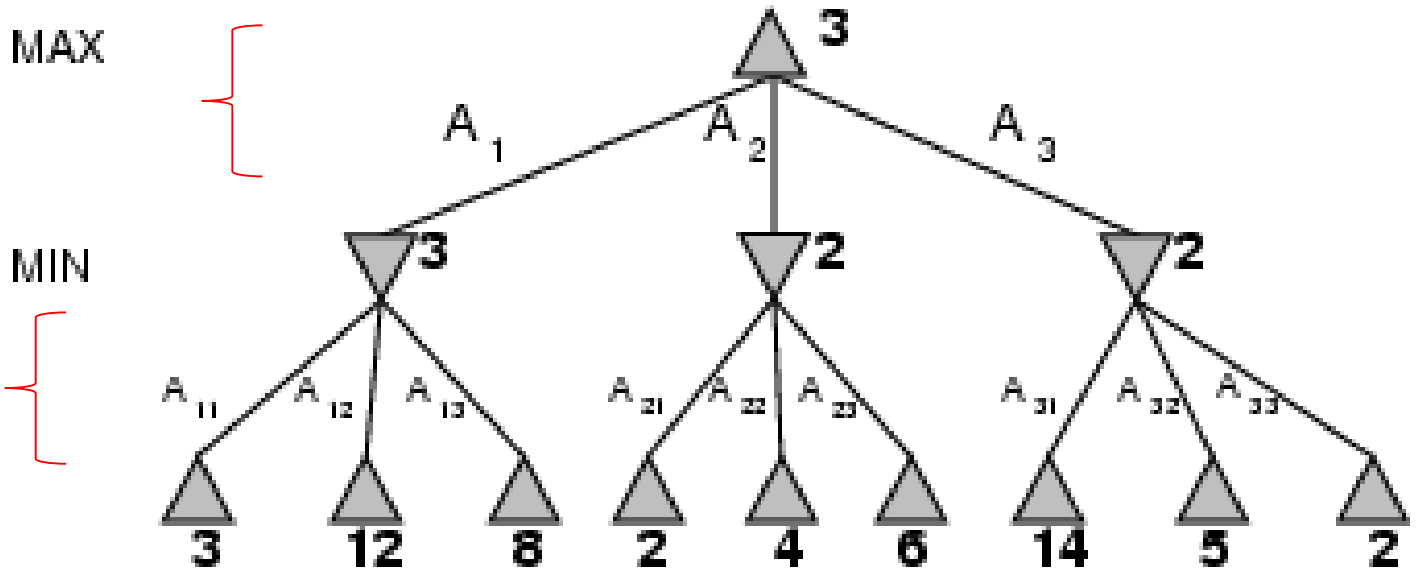
- Estado Inicial: (tablero, laberinto, ...)
- Operadores (jugadas o movimientos permitidos)
- Funcion de Utilidad (puntuacion asociada a una jugada)
- Estado Final. Estados terminales del juego

Juegos vs. Problema de búsqueda



Minimax

- **Idea:** escoger moverse a posición con mas alto valor de mimimax
- **minimax** = mejor jugada mía que posibilite peor jugada del oponente
- Algoritmo minimax: aplicación a dos participantes



Minimax

- Basado en un algoritmo de búsqueda en profundidad limitada
- **Algoritmo minimax:**
  1. Generar *Árbol de búsqueda*, desde la raíz hasta los estados terminales
  2. Obtener valor *función de utilidad* en cada estado terminal
  3. Aplicar la función de utilidad en los nodos *inmediatamente superiores* a los terminales
  4. Continuación de la *propagación hacia el nodo raíz*, una capa cada vez
  5. Al llegar a la raíz, *escoger la jugada* que permita obtener el *mas alto valor*

Minimax

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Escoger sucesor max del min.

Escoger sucesor min. del max

Propiedades de minimax

- Completo? Si
- Optimo? Si
- Complejidad Temporal? $O(b^m)$
- Complejidad Espacial? O(bm)
- Extensión al Método Poda Alfa-beta

Juegos

- Crear programas en el computador para jugar
- Emular el razonamiento humano en el computador
- Construir sistemas que sean capaces de tomar decisiones

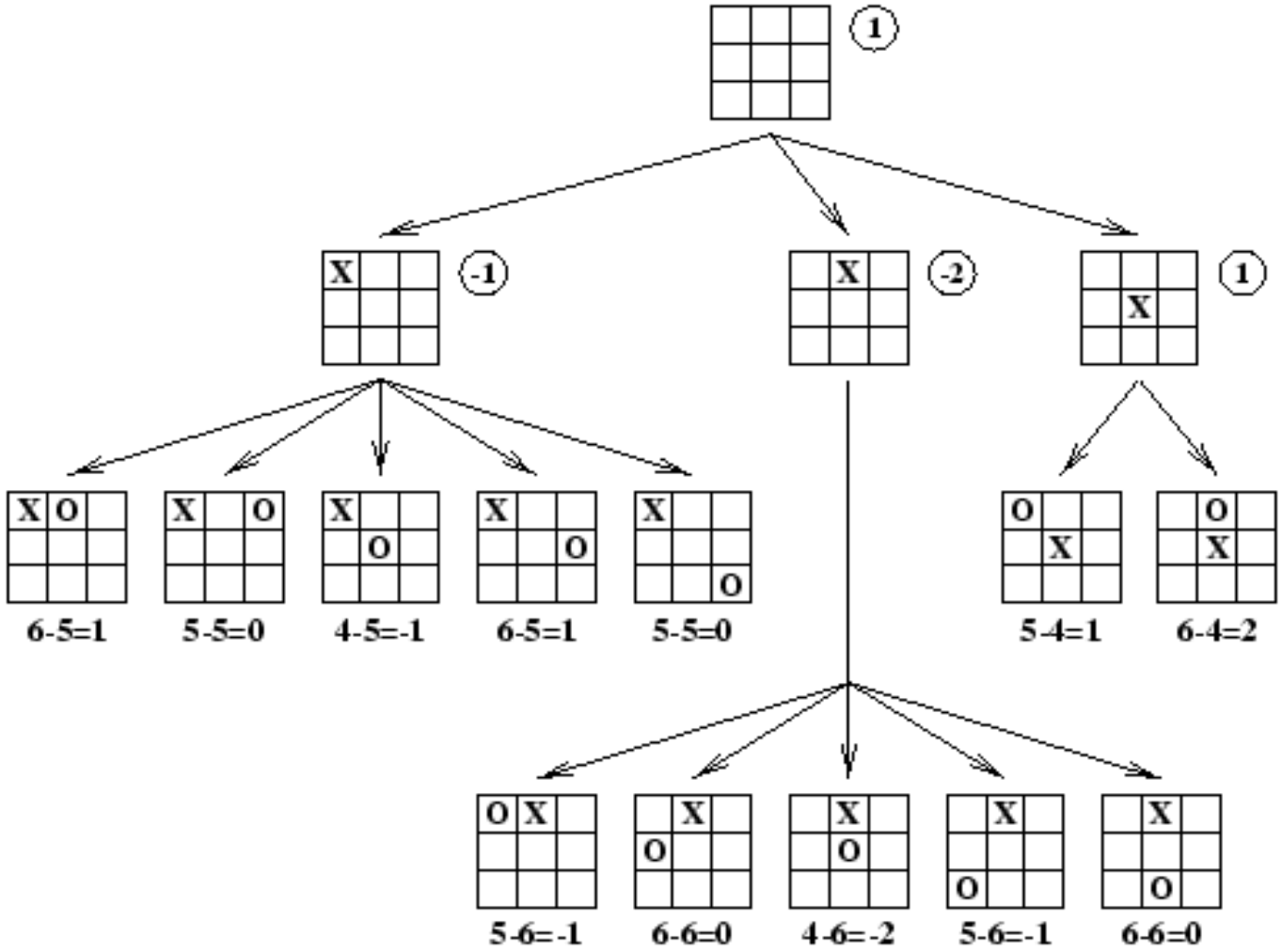
Juegos: Características

- Juegos bipersonales
- Los jugadores mueven alternativamente
- La ventaja para un jugador es desventaja para el otro
- Los jugadores poseen toda la información sobre el estado del juego
- Hay un número finito de estados y decisiones
- No interviene el azar (dados, cartas)

Juegos vs Problema de búsqueda

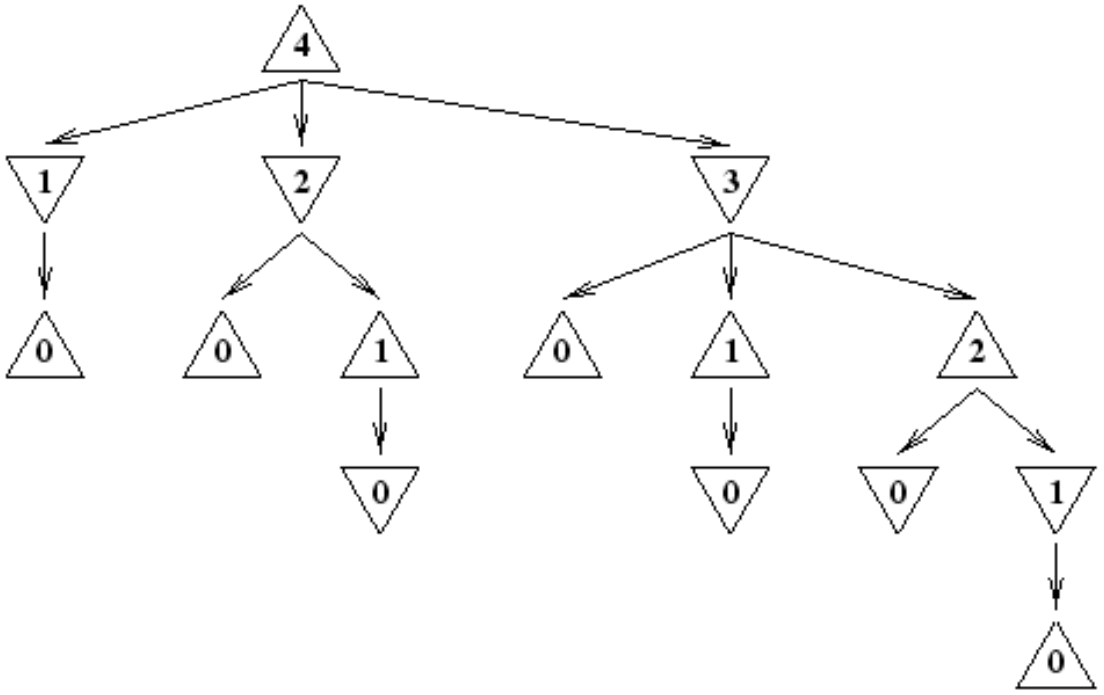
- Juego gran similitud con búsqueda
- Oponente “imprevisto” → se debe especificar un movimiento por cada posible replica del oponente
- Limite de tiempo
- Estado Inicial: (tablero, laberinto)
- Operadores (jugadas o movimientos permitidos)
- Función de Utilidad (puntuación asociada a una jugada)

Ejemplos: 3 en raya



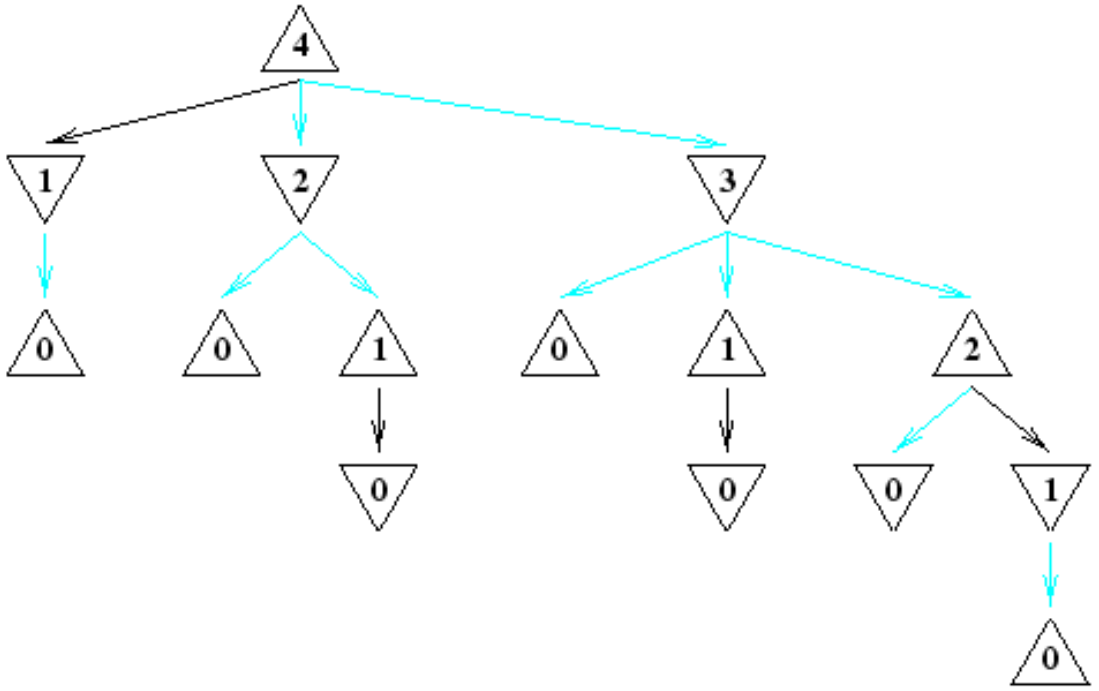
EJEMPLO DE JUEGO: NIM

- Situación inicial: Una pila con N fichas.
- Jugadas: Coger 1, 2 ó 3 fichas de la pila.
- Objetivo: Obligar al adversario a coger la última ficha.
- Desarrollo completo del juego con 4 piezas:



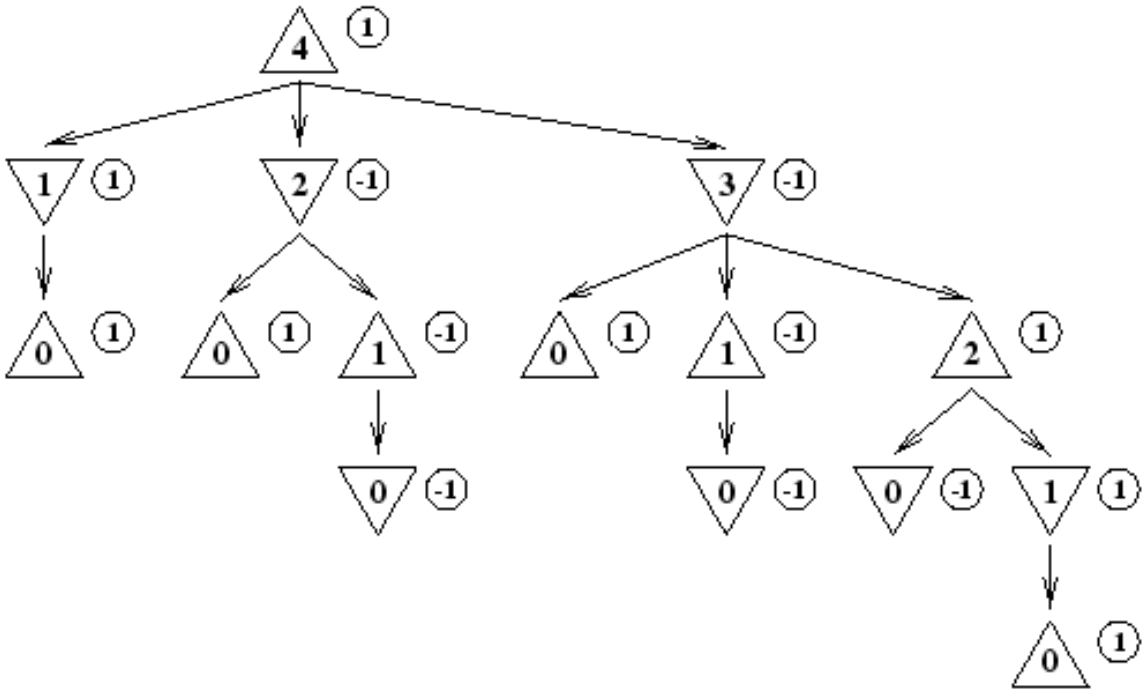
Estrategia ganadora: NIM

- Estrategia ganadora: El movimiento que, haga lo que haga el adversario, nos lleve a una situación ganadora o a la que nos favorezca más.
- Estrategia ganadora en el NIM



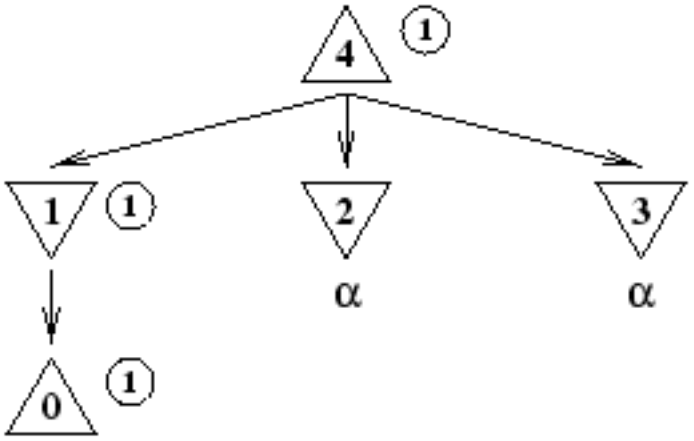
Desarrollo Incompleto

- Función de evaluación estática
  - Límites inferior y superior
- Valores programados, máximo y mínimo.
- Desarrollo del NIM con valores programados.

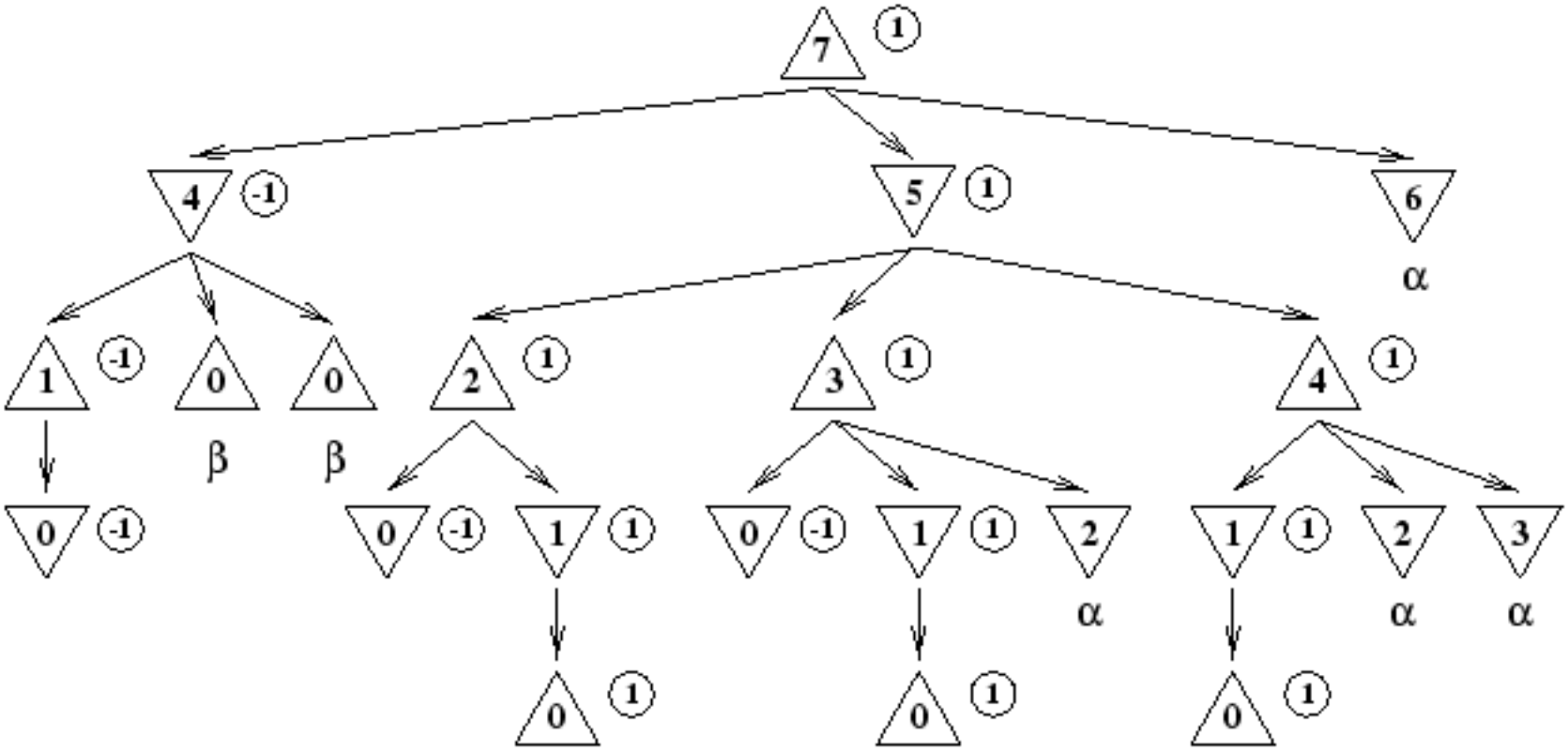


Ejemplo de poda alfa-beta: NIM 4

- Principio alfa-beta: si se tiene una buena (mala) idea, no perder tiempo en averiguar lo buena (mala) que es.



Ejemplo de poda alfa-beta: NIM 7



Complejidad de minimax y alfa-beta

- Complejidad:
  - R : factor de ramificación.
  - P : profundidad de la búsqueda.
  - Complejidad en tiempo de minimax: $O(r^p)$.
  - Complejidad en tiempo de minimax con poda alfa-beta, en el mejor caso: $O(r^{p/2})$.

Complejidad de minimax y alfa-beta

- Aplicación al ajedrez
  - Factor de ramificación : 35
  - Número de movimientos en una partida media: 50
  - Numero de nodos analizados por minimax: $35^{100} \approx 10^{154}$
  - Numero de nodos analizados por minimax con poda alfa-beta : $35^{50} \approx 10^{77}$
  - Número de posiciones legales: 10^{40}

Complejidad en el NIM

- Estados evaluados:
  - Empezando la máquina,
  - Con profundidad 20 y
  - Eligiendo el humano siempre 1.

	8	**12**	**16**	**20**
Minimax	192	2223	25472	291551
Minimax-a-b	16	67	294	1023

Tiempo y espacio para orden 16

	tiempo	espacio
Minimax	14.55 sec.	3245720 bytes
Minimax-a-b	0.23 sec	69812 bytes