INTELIGENCIA ARTIFICIAL PARA JUEGOS

SESIÓN 3

Dr. Edwin Villanueva Talavera

Contenido

- Búsqueda en Profundidad Limitada/iterativa
- Búsqueda de costo uniforme
- Búsqueda con Información
  - Búsqueda codiciosa
  - Búsqueda A*
  - Heurísticas
- Búsqueda Adversarial

Bibliografía:

Capitulo 3.5, 3.6 y 5.1-5.3 del libro:

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

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*

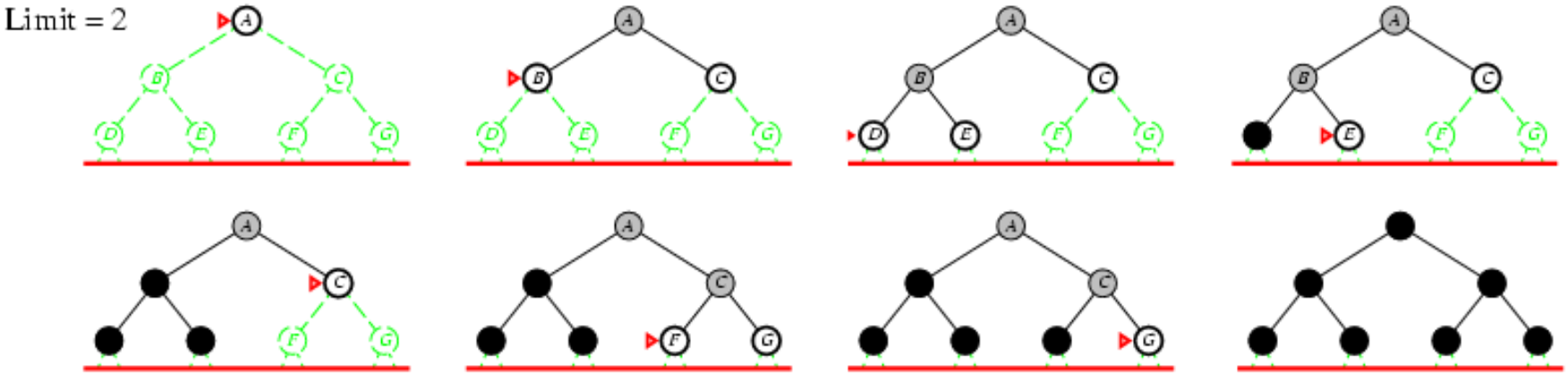
Limit = 0

Búsqueda de profundización iterativa

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

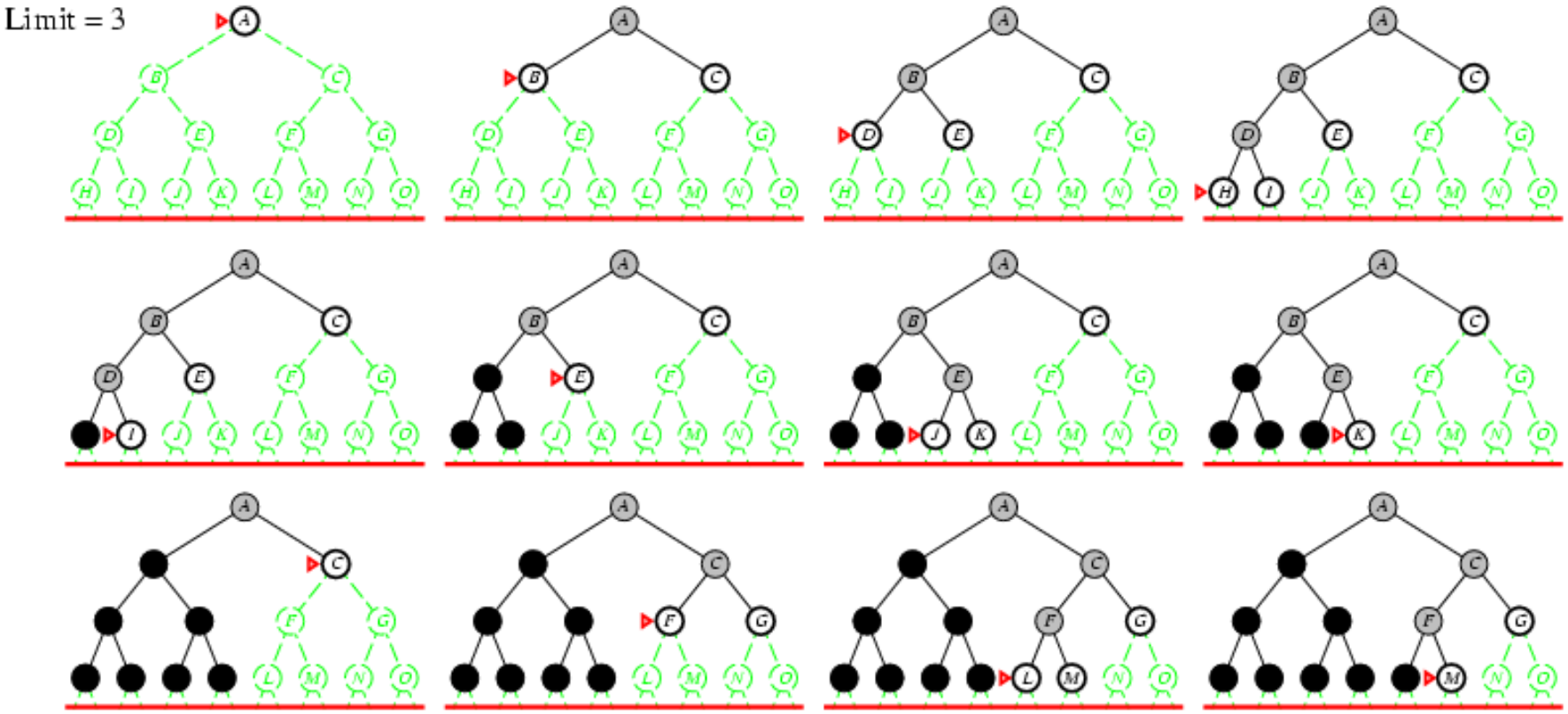
Búsqueda de profundización iterativa

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

Búsqueda de costo uniforme

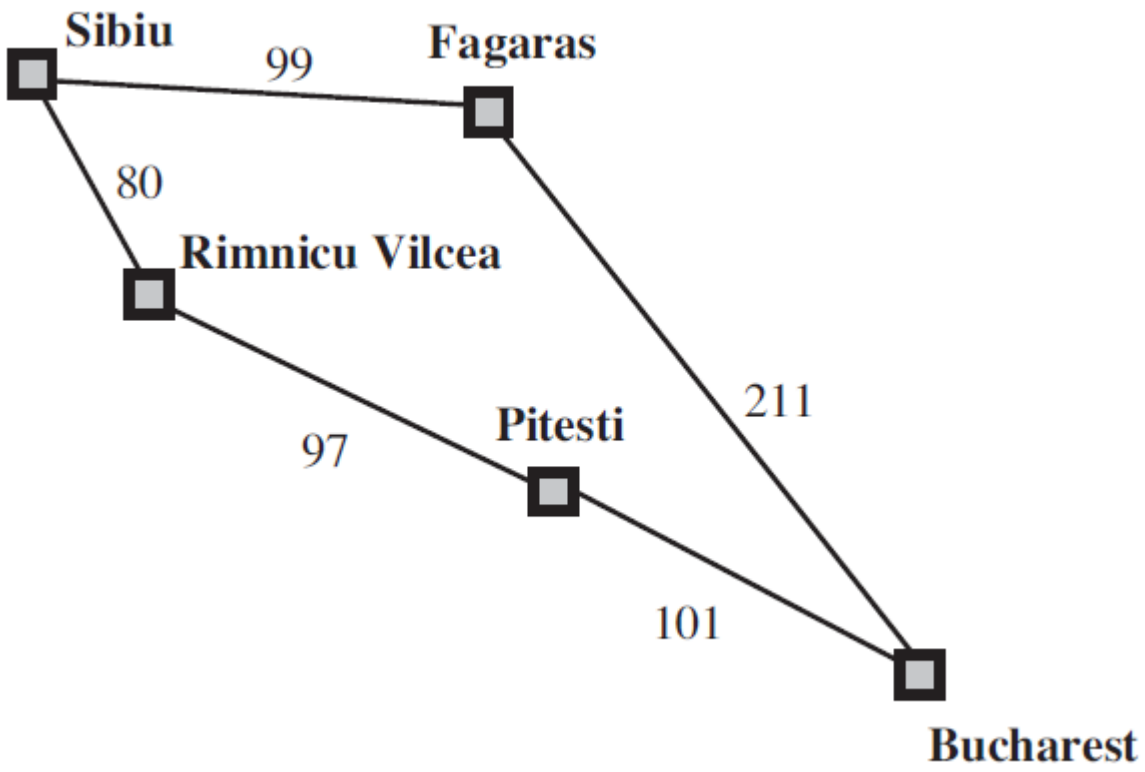
- Expande el nodo no expandido **n** que tenga el costo de camino **g(n)** más bajo
- Implementación: Puede ser GRAPH-SEARCH usando como frontera una lista ordenada por **g(n)** (PATH-COST)

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child
```

Búsqueda de costo uniforme

Ejercicio:

- Aplicar búsqueda de costo uniforme en el mapa de Rumania para llegar a *Bucharest* partiendo de *Sibiu*



Búsqueda de costo uniforme

Propiedades:

- Equivalente a la búsqueda en amplitud si los costos de las acciones son todos iguales
- Completa? SI, si el costo de cada paso es $\geq \varepsilon$
- Complejidad de tiempo:

  # de nodos con g() ≤ costo de solución óptima, $O(b^{1+\lceil C^*/\varepsilon \rceil})$, *donde C* es el costo de la solución optima*
- Complejidad de espacio:

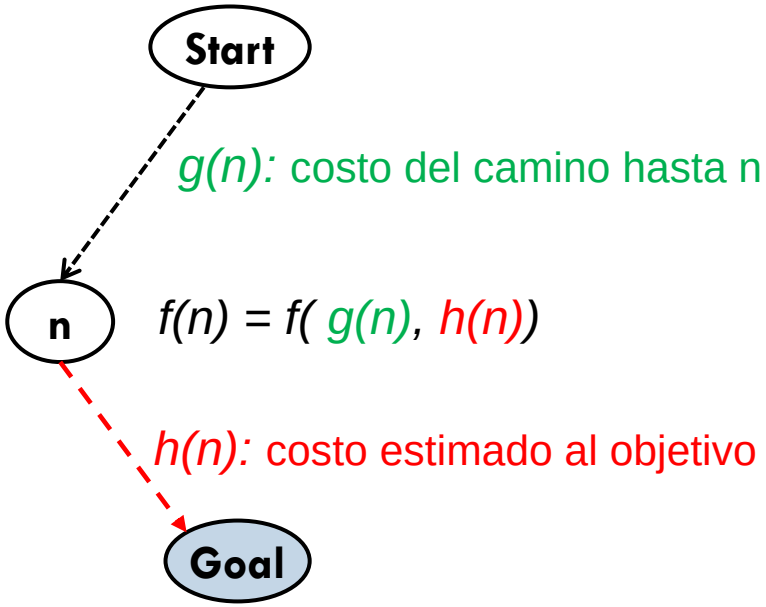
  Igual que la complejidad de tiempo
- Optima? SI, ya que los nodos son expandidos en orden creciente del costo total.

Búsqueda con Información

- Utiliza conocimiento específico sobre el problema para encontrar soluciones de forma mas eficiente que la búsqueda ciega.
  - Conocimiento específico adicional a la definición del problema.
- Enfoque general: búsqueda por la mejor opción
  - Utiliza una función de evaluación para cada nodo.
  - Expande el nodo que tiene la función de evaluación más baja.

Búsqueda por la mejor opción

- Idea: usar una función de evaluación *f(n)* para cada nodo.
  - f(n) es una estimación de cuan deseable es el nodo n
  - Se expande el nodo mas deseable que aún no fue expandido
  - **f(n)** es normalmente una combinación del **costo de camino g(n)** y de una **función de heurística h(n)** que mide el costo estimado para llegar al objetivo desde n



Búsqueda por la mejor opción

- Implementación:
  - Similar que Búsqueda de Costo Uniforme. La diferencia es que los nodos de la frontera son ordenados por la función de evaluación f(n), en vez del costo de camino.
- La selección de *f(n)* determina la estrategia de búsqueda:
  - Busca codiciosa por la mejor opción
  - Busca A*

Búsqueda por la mejor opción

Implementación

```
function BEST-FIRST-GRAPH-SEARCH(problem, f) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE
  frontier ← a priority queue ordered by f, with node as the only element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)  /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored And child.STATE is not in frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in some frontier node n with f(n) > f(child) then
        replace frontier node n with child
```

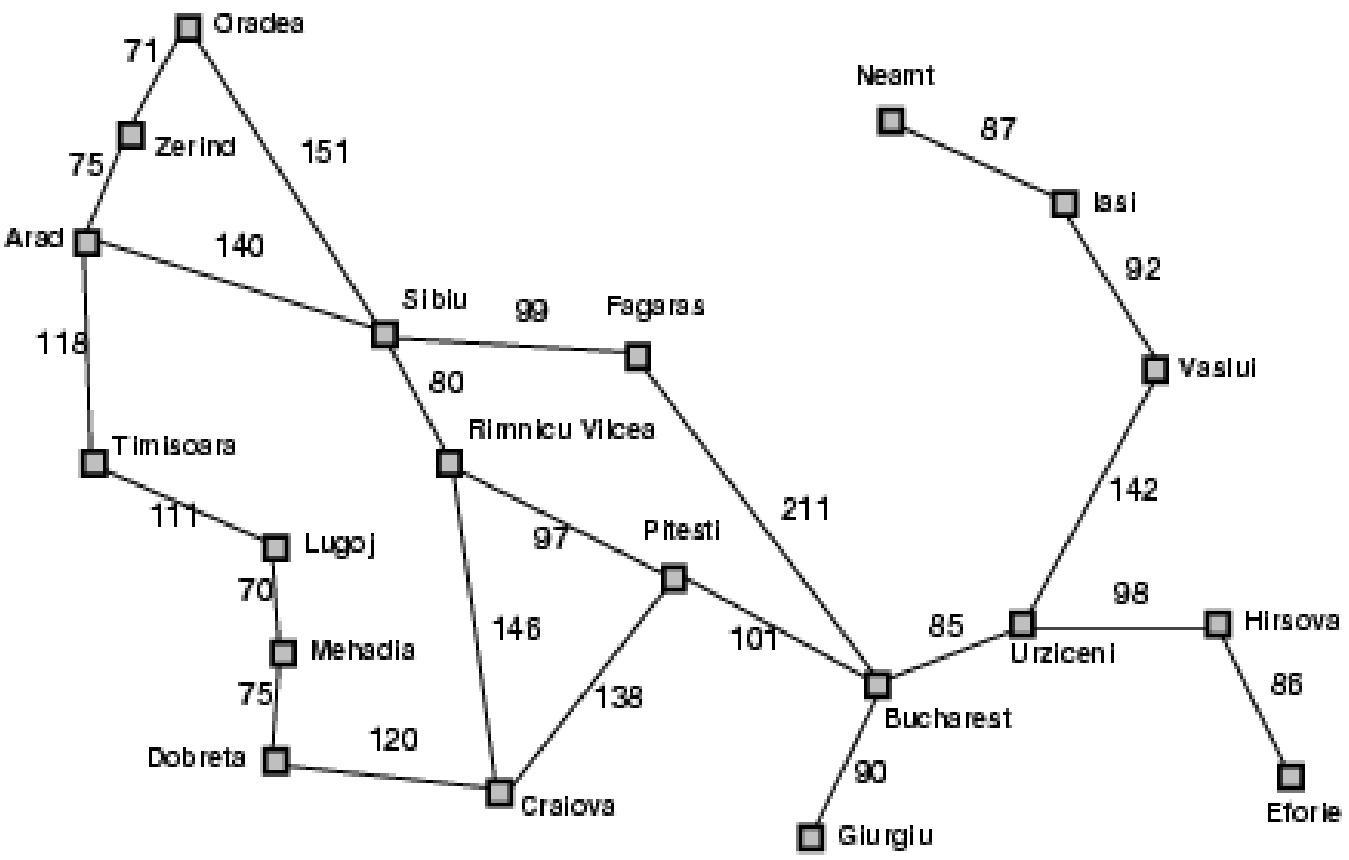
Búsqueda Voraz

- Es un tipo de búsqueda por la mejor opción donde la función de evaluación: $f(n) = h(n)$ (heurística)
  - *h(n): estimado del costo del camino mas barato desde n al objetivo*
  - *h(nodo_objetivo) = 0*

  Ejemplo: h(n) = $h_{DLR}(n)$ = distancia en línea recta desde n hasta el objetivo.
- La búsqueda codiciosa o voraz expande el nodo en la frontera con menor *h(n)*, osea, el que parece que está mas próximo al objetivo de acuerdo a la función heurística.

Búsqueda Voraz

- Ejemplo de búsqueda voraz en el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:

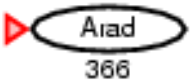


Distancia en linea recta hasta Bucareste

Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Búsqueda Voraz

- Ejemplo de búsqueda voraz en el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:

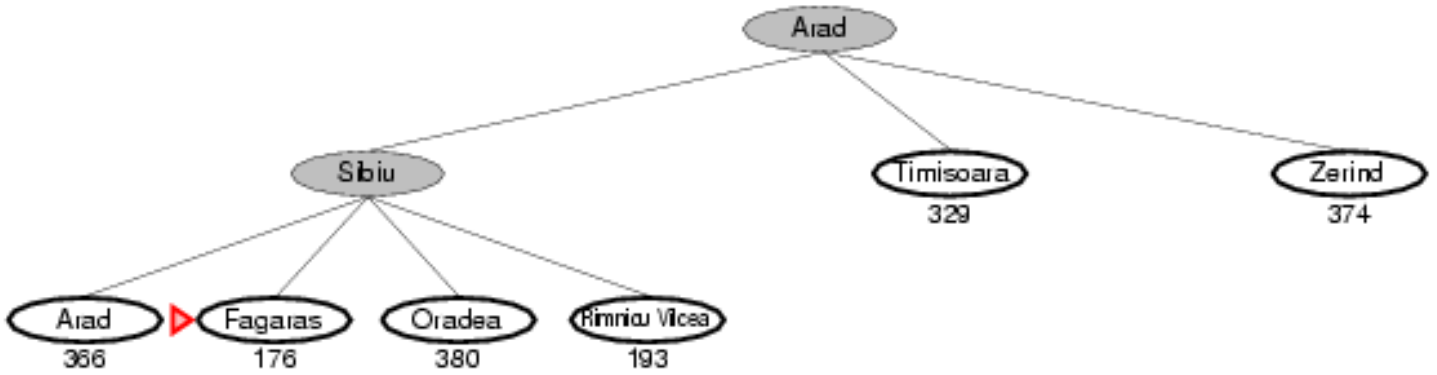


Búsqueda Voraz

- Ejemplo de búsqueda voraz en el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:

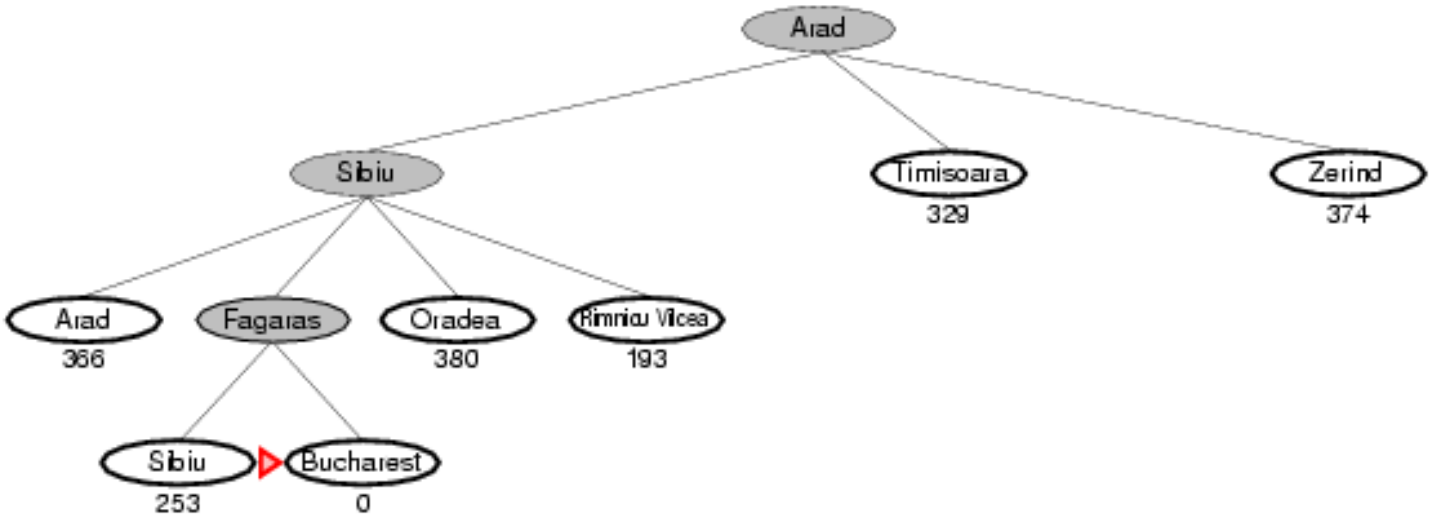
Búsqueda Voraz

- Ejemplo de búsqueda voraz en el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



Búsqueda Voraz

- Ejemplo de búsqueda voraz en el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



El camino via Rimnicu Vilcea y Pitesti es 32 km mas corto!

Búsqueda Voraz

Propiedades de búsqueda voraz

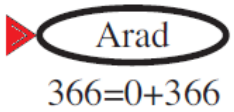
- Completa? Si, si chequea estados repetidos.
- Complejidad de tiempo: $O(b^m)$ en el peor caso, pero una buena función heurística puede llevar a una reducción substancial
- Complejidad de espacio:
  - $O(b^m)$, mantiene todos los nodos en memoria.
- Optima? NO. Puede haber un camino mejor siguiendo algunas opciones peores en algunos nodos del árbol de búsqueda

Búsqueda A*

- Es la forma mas común de búsqueda por la mejor opción
- Função de avaliação $f(n) = g(n) + h(n)$
  - $g(n)$ = costo del camino para alcanzar n
  - $h(n)$ = costo estimado del camino mas barato de n hasta el objetivo
  - $f(n)$ = costo total estimado del camino mas barato hasta el objetivo pasando por n

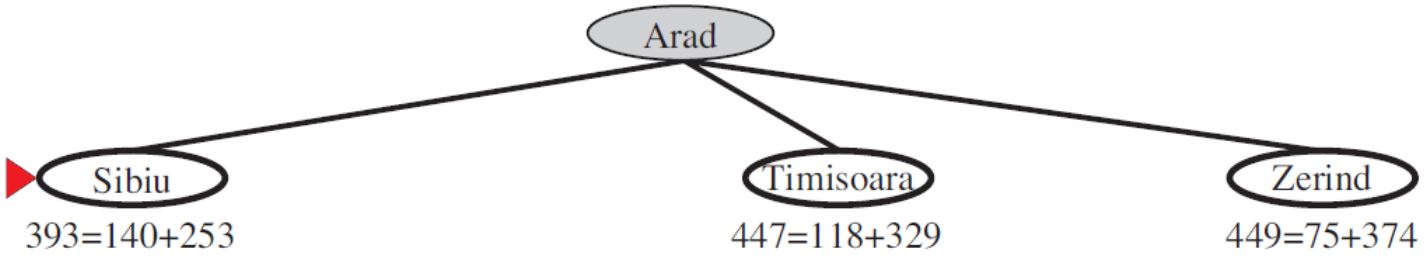
Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



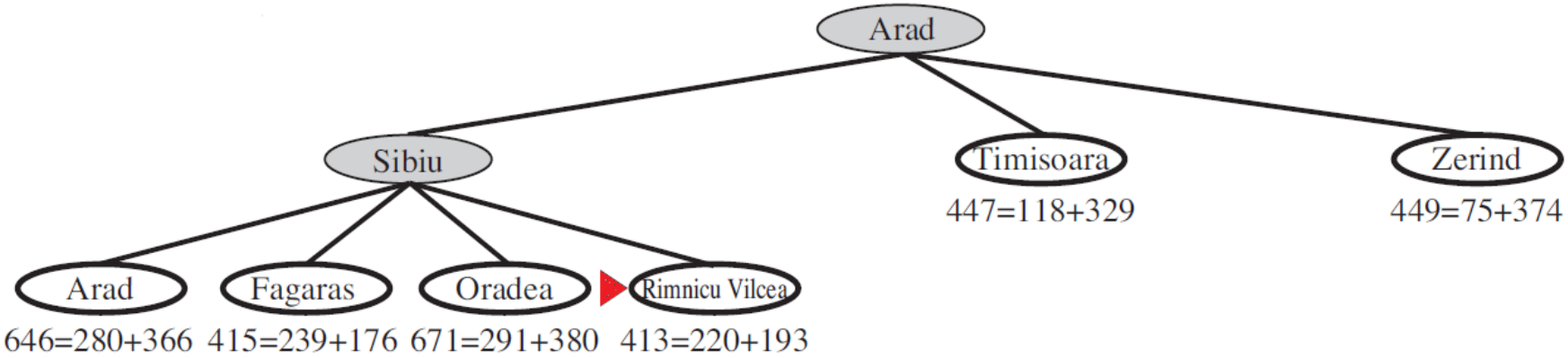
Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



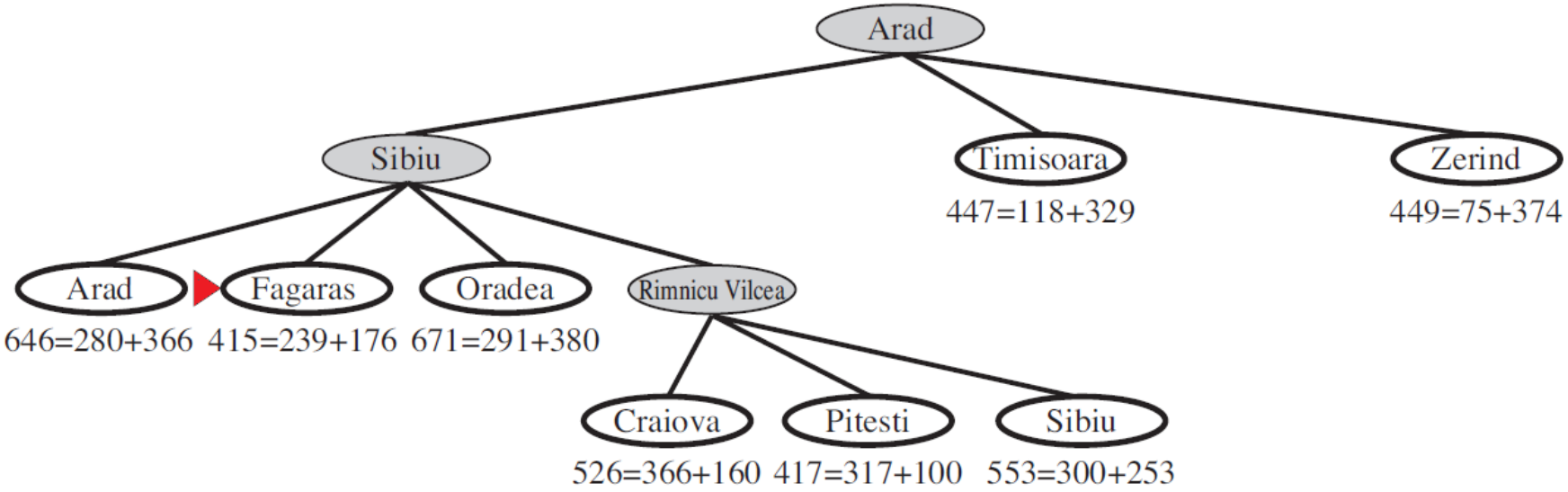
Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



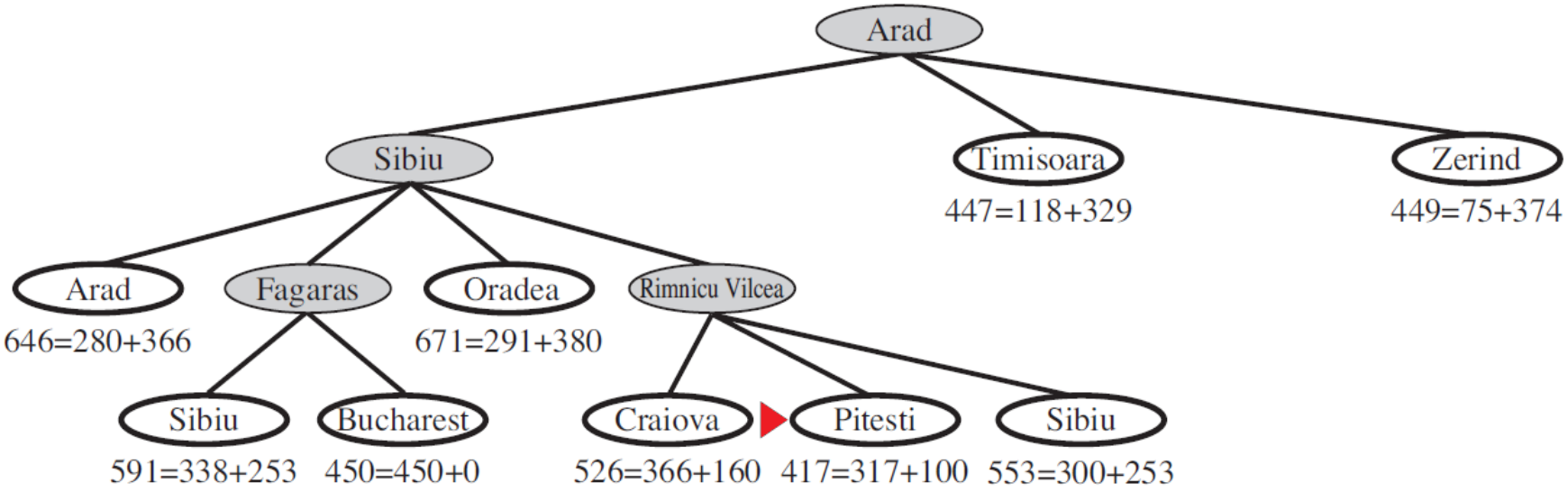
Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



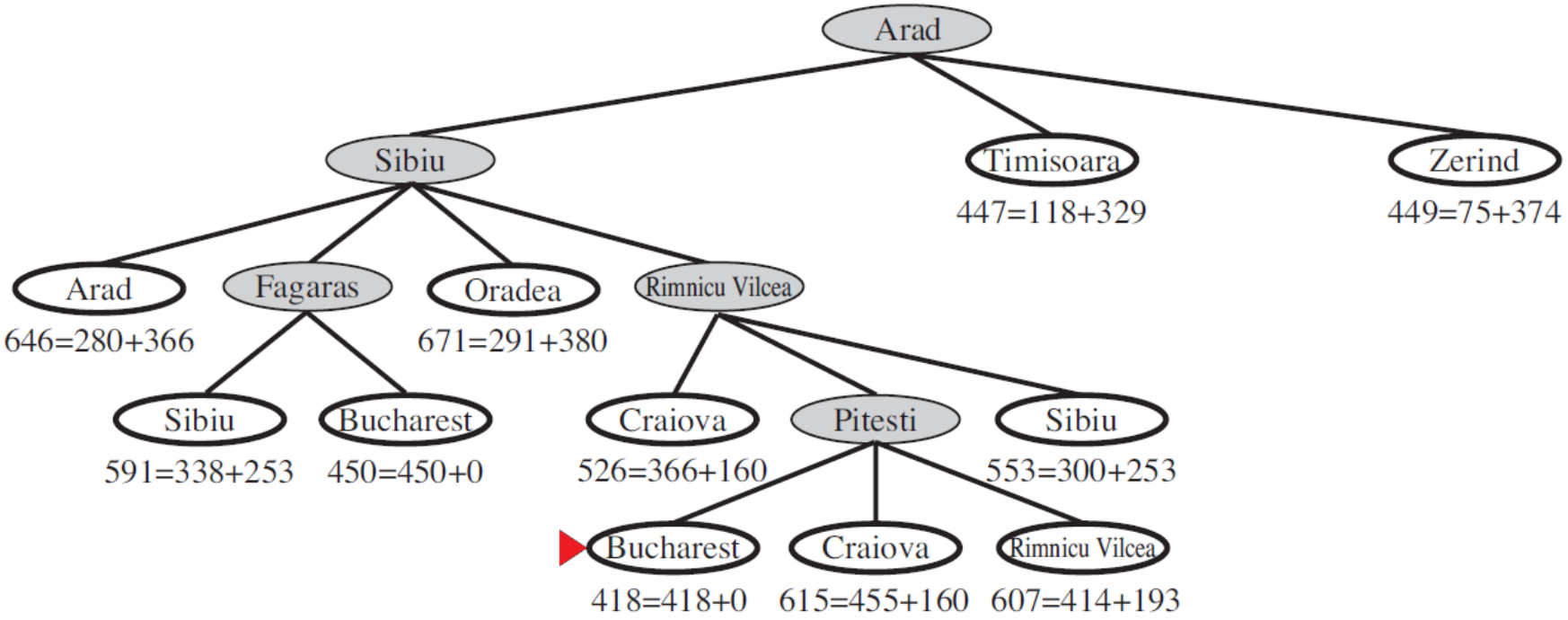
Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



Búsqueda A*

- Ejemplo de búsqueda A* el mapa de Romania siendo el objetivo Bucharest y heurística $h_{DLR}(n)$:



Búsqueda A*

Condiciones de Optimalidad: Admisibilidad

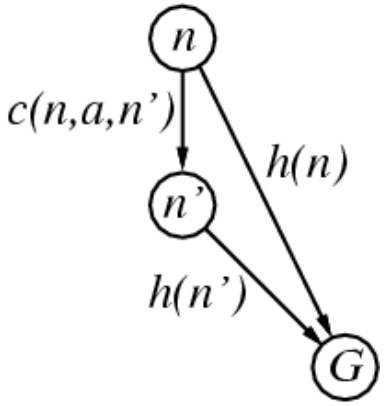
- Una heurística $h(n)$ es admisible si para cada nodo n se verifica $h(n) \leq h^*(n)$, donde $h^*(n)$ es el costo verdadero de alcanzar el estado objetivo a partir de n
- Una heurística admisible nunca sobreestima el costo de alcanzar el objetivo, es decir, la heurística siempre es optimista.
  - Ejemplo: $h_{DLR}(n)$ (distancia en línea recta nunca es mayor que distancia por las calles).
- Teorema: Si $h(n)$ es admisible, A* es optimo con búsqueda en árbol (sin memoria de estados visitados).
- Cuando se usa BEST-FIRST-GRAPH-SEARCH se necesita una condición mas estricta para garantizar optimilidad: Consistencia

Búsqueda A*

Condiciones de Optimalidad: Consistencia

- Una heurística $h(n)$ es consistente (o monotónica) si para cada nodo n y sucesor n' generado por acción a se verifica que:

$$h(n) \leq c(n,a,n') + h(n')$$



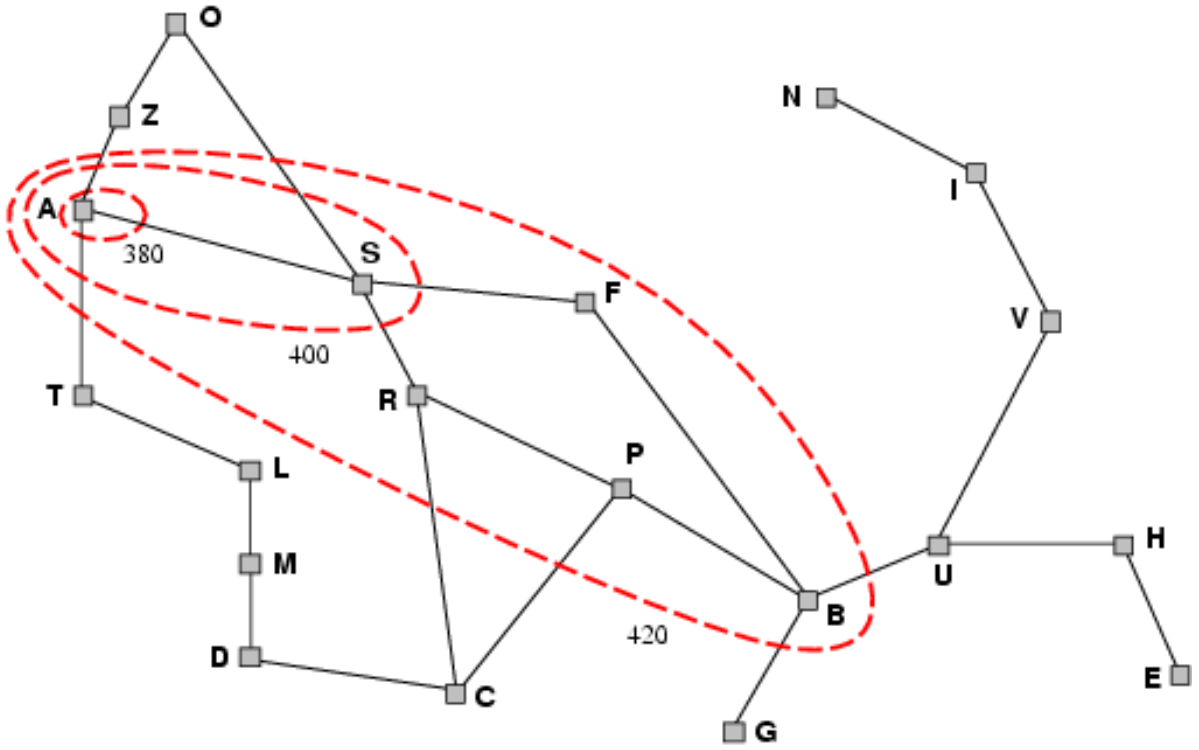
Es una forma de la desigualdad triangular (cada lado del triangulo no puede ser mayor que la suma de los otros lados.

- Toda heurística consistente es también admisible

Búsqueda A*

Contornos de valores *f* en el espacio de estados que A* traza

- A* expande nodos en orden creciente de valores de *f*
- Gradualmente adiciona contornos de nodos
- Los estados fuera del contorno *i* tienen $f > f_i$, donde $f_i < f_{i+1}$
- No expande nodos con f(n)> C* (costo de la solución optima)



Si h(n)=0 tenemos una búsqueda de costo uniforme ⇒ círculos concéntricos.

Cuanto mejor la heurística mas direccionados son los círculos hacia el objetivo

Búsqueda A*

Propiedades de A*

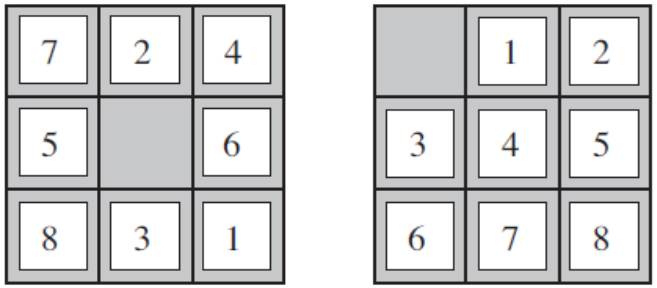
- Completa? Si, a menos que exista una cantidad infinita de nodos con f(n)< C*)
- Complejidad de tiempo: Exponencial en el peor de los casos (heurística no apropiada)
- Complejidad de espacio: También exponencial en el peor caso ya que mantiene todos los nodos en memoria.
- Optima? SI
- Óptimamente Eficiente: Ningún otro algoritmo de búsqueda garantiza expandir un numero menor de nodos que A* y encontrar la solución optima. Esto porque cualquier algoritmo que no expande todos los nodos con f(n) < C* corre el riesgo de omitir una solución optima.

Heurísticas

Ejemplo de heurísticas admisibles

- Para el rompecabezas de 8 piezas:
  - $h_1(n)$ = número de piezas fuera de posición
  - $h_2(n)$ = distancia "Manhattan" total (para cada pieza calcular la distancia en movidas verticales y horizontales hasta su posición objetivo)

Ejercicio



- $h_1(S) = ?$
- $h_2(S) = ?$

Heurísticas

Factor de ramificación efectiva de las heurísticas

- Factor de ramificación efectiva (*b**):
  - Si el numero total de nodos generados por A* es *N* y la profundidad de la solución *d*, entonces *b** es el factor de ramificación que un árbol uniforme de profundidad *d* tendría para contener *N+1* nodos:

$$N + 1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

  - Por ejemplo, si A* encuentra la solución a un profundidad d=5 expandiendo 52 nodos entonces $b^* = 1.92$
  - Mejores heurísticas tienen valores mas próximos a 1

Heurísticas

Factor de ramificación efectiva de las heurísticas h_1 y h_2 para el rompecabezas de 8 piezas

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Heurísticas

Dominancia:

- Una heurística h_2 domina a otra heurística h_1 si para todo nodo *n*:

$$h_2(n) \geq h_1(n)$$

- h_2 es mejor que h_1 cuando h_2 domina h_1
  - La heurística dominante h_2, al tener mayores valores de h en cada nodo, expande nodos mas próximos del objetivo de que h_1, significando menos nodos expandidos.

Heurísticas

Como crear heurísticas admisibles: problema relajado

- El costo de la solución optima de una simplificación del problema (problema relajado) puede ser una heurística para el original.

  Por ejemplo, en el problema del rompecabezas de 8 piezas, la formulación original es:

  Una pieza puede moverse del cuadrado A al cuadrado B si

  A es horizontalmente o verticalmente adyacente a B y B es blanco

  Podríamos generar los siguientes problemas relajados:

  I. Una pieza puede moverse del cuadrado A al cuadrado B

  II. Una pieza puede moverse del cuadrado A al cuadrado B si A es adyacente a B

  El costo de la solución óptima del problema (I) seria h_1, (# piezas fuera de lugar), mientras que el costo de la solución optima del problema (II) seria h_2 (distancia Manhatan)

Heurísticas

Como crear heurísticas admisibles: problema relajado

- Las heurísticas generadas con problemas relajados son admisibles, ya que el costo de la solución del problema relajado nunca va ser mayor que el del problema original
- También son consistentes para el problema original si lo son para el problema relajado
- Si se tiene una colección de heurísticas admisibles $h_1 \ldots h_m$ y ninguna domina a las demás, se puede generar una nueva heurística *h* que domina a todas:

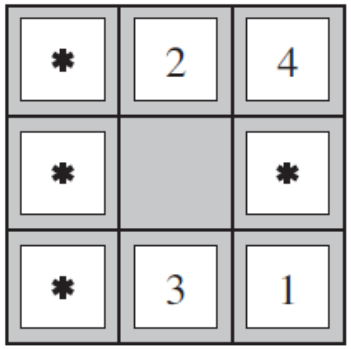
$$h(n) = \max\{h_1(n), \ldots, h_m(n)\}$$

Heurísticas

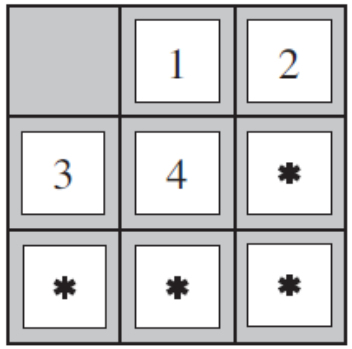
Como crear heurísticas admisibles: Sub-problemas

1. Usar el costo de la solución optima de un sub-problema del problema original.

Ejemplo: Costo de colocar los 4 primeros números en sus lugares, sin importarse como queden los otros números * (aunque sus movidas si cuentan en el costo)



Start State



Goal State

Búsqueda Adversarial

- Los algoritmos de búsqueda estudiados hasta ahora son apropiados para implementar agentes que actuarán en ambientes sin interacción con otros agentes y que poseen total control de sus acciones y de sus efectos
- En entornos multiagentes, donde las acciones de los agentes afectan los objetivos de los otros agentes (entornos conocidos como **juegos**) se necesita otro tipo de algoritmos para guiar la toma de decisiones de los agentes : **algoritmos de búsqueda adversarial**
- Un tipo común de juego que abordaremos son los llamados **juegos de suma-zero** (*zero-sum games*), donde la suma de la utilidad de los agentes (**jugadores**) al final del juego es constante

Búsqueda Adversarial

Consideraciones en juegos de dos jugadores:

- El agente que nos toca programar lo llamamos MAX, al oponente lo llamamos MIN.
- MAX hace el primer movimiento, turnándose con MIN hasta el final
- La movida del oponente MIN es "imprevisible": MAX tiene que considerar todos los movimientos posibles del MIN
- Limite de tiempo: el agente MAX tiene que tomar una decisión, así no sea óptima
- Al final del juego, cada jugador recibe un valor de utilidad que refleja su estado al final del juego (ej. ganador, perdedor, empate)

Búsqueda Adversarial

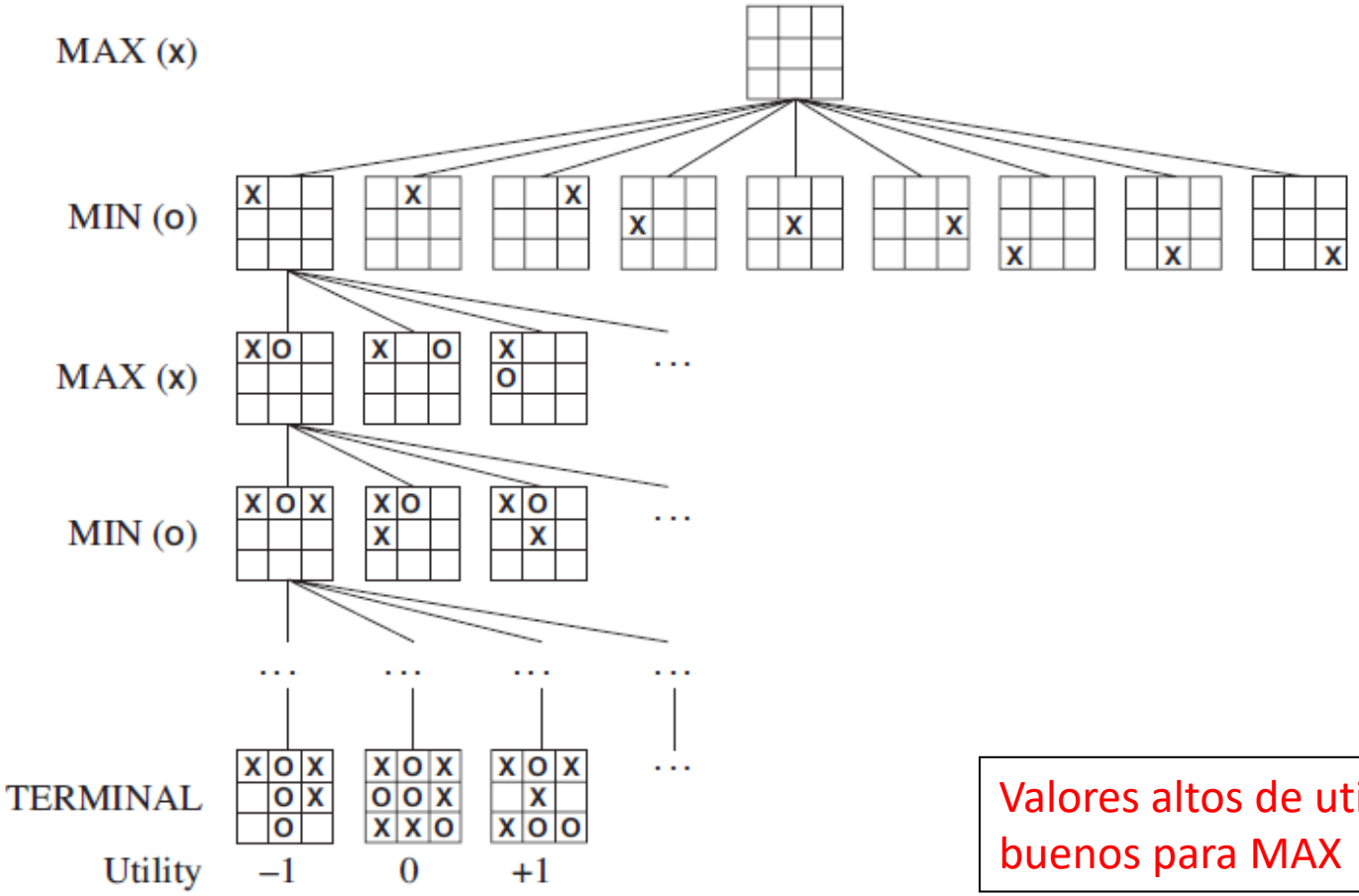
Formulación del Juego:

- Estado Inicial S_0: Situación del juego al inicio
- Player*(s)*: Define que jugador tiene la movida en un estado dado *s*
- Actions*(s)*: Retorna el conjunto de movidas válidas a partir del estado *s*
- Result*(s,a)*: Función sucesora (o modelo de transición). Define a que estado se arriba aplicando acción *a* al estado *s*
- Terminal-Test*(s)*: Determina si estado *s* es un estado final de juego (estado terminal)
- Utility*(s,p)*: Función de utilidad. Retorna el valor de utilidad que recibe jugador *p* al final del juego finalizado en estado terminal *s*

Búsqueda Adversarial

Arbol de Juego (2 jugadores):

- Arbol donde los nodos son estados del juego y aristas son movidas. Nodo raiz es el estado inicial del juego



Valores altos de utilidad son buenos para MAX

Búsqueda Adversarial

Decisiones Óptimas en Juegos (2 jugadores):

- La solución óptima para MAX seria uma sequencia de movidas que MAX haría para llegar a un estado terminal donde el sea ganador
- MAX debe encontrar una **estrategia** de contingencia la cual debe especificar el movimiento en el estado inicial y luego el movimiento en los estados resultantes de cada posible movimiento de MIN y así sucesivamente
- Para encontrar esta estratégia se asume que el oponente MIN hace sus movimientos de forma óptima (queriendo llegar a un estado terminal donde MIN es el ganador)

Búsqueda Adversarial

Decisiones Óptimas en Juegos (2 jugadores): MINIMAX

- Dado un árbol de juego, la estrategia óptima puede ser determinada a partir del **valor Minimax de cada nodo:**

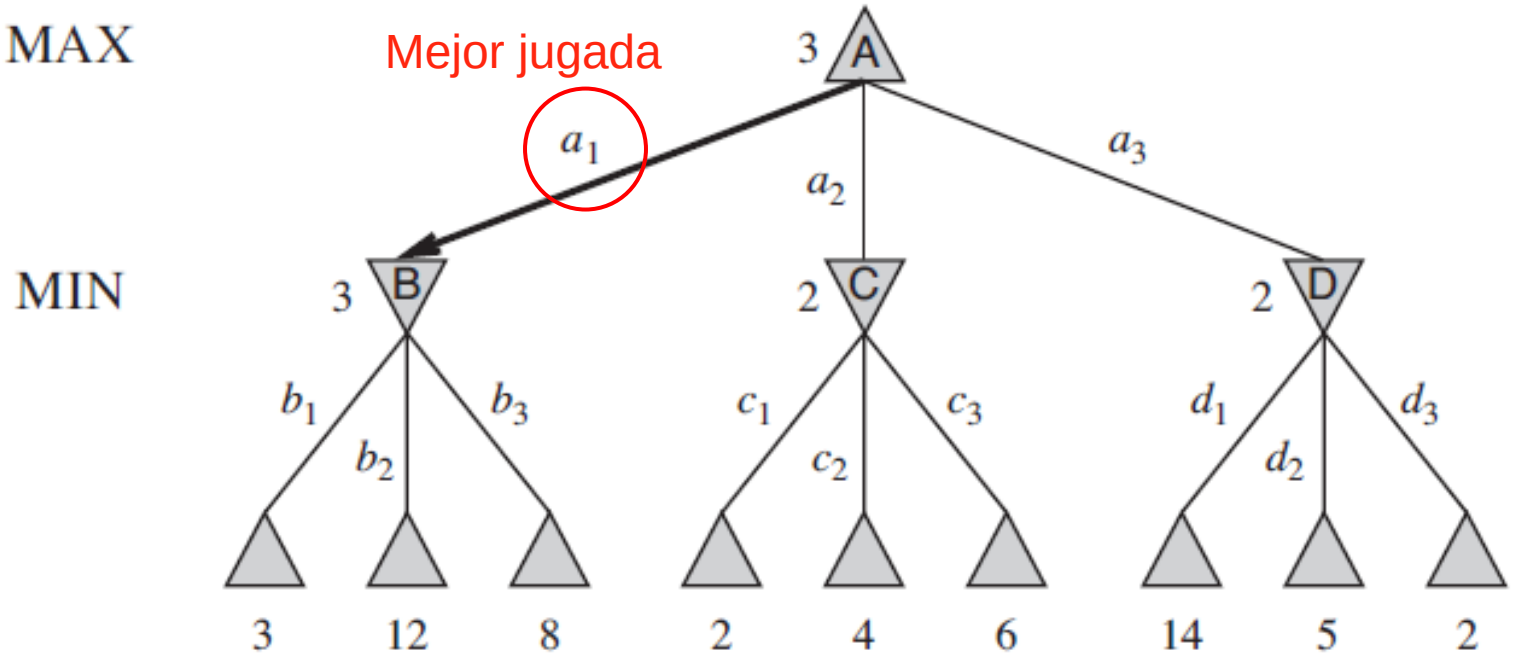
$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{MINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \end{cases}$$

- El valor Minimax(s) (**para MAX**) es la utilidad de MAX de estar en estado s asumiendo que MIN escogerá los estados más ventajosos para el mismo hasta el final del juego (es decir, los estados con menor valor de utilidad para MAX)

Búsqueda Adversarial

Ejemplo de cálculo del valor de MINIMAX:

- Cada jugador hace un solo movimiento



Búsqueda Adversarial

Algoritmo MINIMAX:

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Búsqueda Adversarial

Propiedades del algoritmo MINIMAX:

- Equivale a una búsqueda completa en profundidad en el árbol del juego.
  - m: profundidad máxima del árbol
  - b: movimientos válidos en cada estado

- Completo? Si (Si el árbol es finito)
- Óptimo? Si (contra un oponente óptimo)
- Complejidad de tiempo? $O(b^m)$
- Complejidad de espacio? O(bm)

Para ajedrez, b ≈ 35, m ≈100 para juegos "razonables"
→ solución exacta no es posible

Preguntas?