Universidad Simón Bolívar

CI-5437 Inteligencia Artificial 1

Resumen de las Clases

Alumno:
Tony Lattke

Profesor:
Prof. Blai Bonet

28 de octubre de 2012

Resumen

1. Oficina: MYS-215A

2. Correo: bonet@ldc.usb.ve

3. Plan de Evaluación:

   3 Proyectos y final 50 %

   Examen 50 % martes semana 12

4. Bibliografía:

   Russell S., Norvig P. Artificial intelligence - A modern approach

5. Objetivos:

   Búsqueda Heurística

   BFS, DFS, DFID

   $A^*, IDA^*, LRTA^*, BnB$

   Descomposición de problemas en subproblemas

   Grafos AND/OR

   AO^*

   $aclos, LAO^*, LDFS$

   Arboles de juego

   Min Max X

   $\alpha\beta$ - Pruning

   Scoot, MID, UCT

   Planificación

   CSP

   Representación

   Solución

   Representación de conocimiento, lógica

   Representación de inferencia: Lógica proposicional (SAT) y Lógica de 1^{er} orden: Resolución

   Manejo de incertidumbre (representación de inferencia)

   Redes de Markov

   Redes Bayesianas

   Grafos de factores

Índice general

Clase 1

1.1. ¿Qué es IA?

Historia

1. Todo este campo empezó con la primera computadora
2. Test de Turing
3. Mini Test de Turing, ejemplo: Captcha, preguntas de fácil resolución
4. IBM

Sub disciplina

1. Visión
2. NLP (Natural Language Processing)
3. Representación de conocimiento
4. Planificación
5. Machine Learning

Visión moderna

1.2. Lenguajes

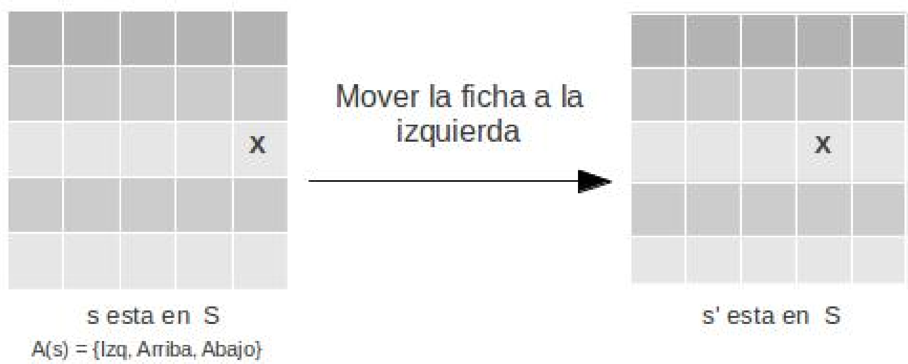
Todo empezó con el desarrollo de lenguajes Lisp, Scheme, Prolog. Luego por querer más portabilidad y eficiencia se empezó a usar C, C++, Java

1.3. Búsqueda

$$Graph = (Vertex, Edges) \tag{1.1}$$

Espacio de búsqueda es caracterizado por:

1. Conjunto finito de estados S
2. Estado inicial $S_0 \in S$
3. Subconjunto de estados objetivos (Goals), $S_G \subseteq S$
4. Conjunto finito A de operadores para cada $s \in S$, tenemos un subconjunto $A(s) \subseteq A$ de soperadores aplicables en S
5. Función de transición $f(.,.)$ tal que $f(s, a)$ es el estado que resulta de aplicar a en el estado S, para todo $s \in S$ y $a \in A(s)$ Ejemplo:



6. Costos $C(s, A)$ de aplicar la acción aplicable a en el estado $s \in S$

1.4. Soluciones

Una solución es una secuencia de acciones tal que:

1. $a_0 \in A(s_0)$
2. $a_1 \in A(s_1)$ donde $s_1 = F(s_0, a_0)$
3. $a_i \in A(s_i)$ donde $s_i = F(s_{i-1}, a_{i-1})$ para $1 \leq i \leq n+1$
4. $S_{n+1} \in S_G$

Costo de $\pi : C(T) = C(s_0, a_0) + C(s_1, a_1) + \ldots + C(s_n, a_n) = \Sigma_{0 \leq i \leq n} C(s_i, a_i)$

1.5. Ejemplos

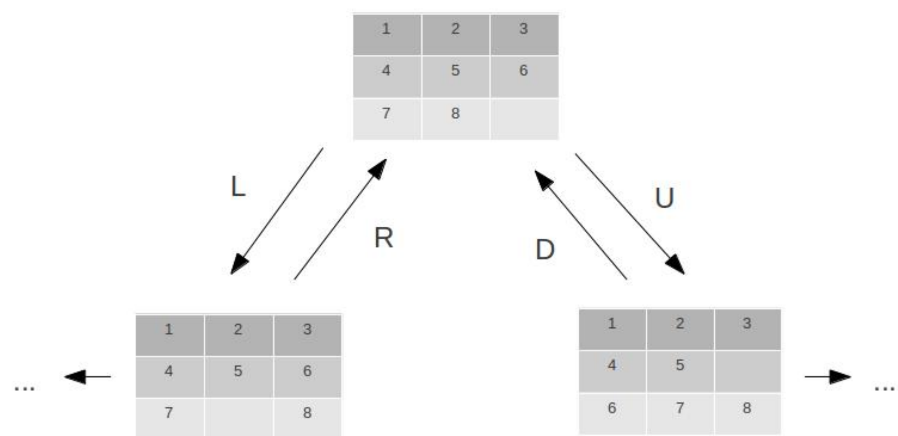
Existen muchos casos en los que podemos calcular el árbol de búsqueda:

1.5.1. Puzzle

Podemos ver el 8 puzzle, 15 puzzle, 24 puzzle, $(n^2 - 1)$ puzzle, $(nm - 1)$ puzzle

La acciones disponibles en este juego son: mover el blanco L, R, U, D

En el caso de un 8 puzzle en el estado mostrado se tienen las siguientes opciones:



Número de estados es 9! eso es aproximadamente 100000 para el 8 puzzle

Para un 15 puzzle se necesitan 16 TB para representar todos los estados del grafo

1.5.2. Grafos implícitos

1.5.3. Cubo de rubik

1.5.4. Jarras de agua

1.5.5. Torres de Hanoi

Para 3 discos y 3 barras se tienen 3^n estados y $2^n - 1$ pasos, donde n es el número de discos.

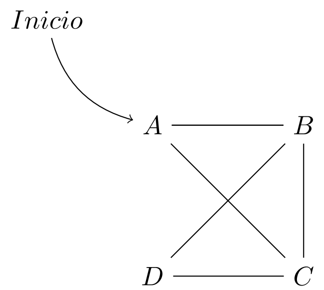
Que pasaría si tienes 4 barras y 3 discos, se tienen muchas soluciones y no puedes tener una sola solución buena.

1.5.6. Río, cabra, lobo, paja

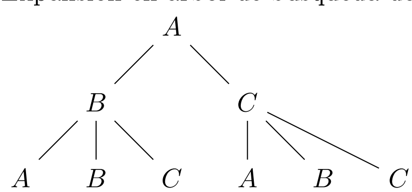
1.6. Nociones básicas

Nodo es una representación del estado.

Un estado puede estar representado más de una vez, eso se le llama duplicado.
Al explorar un grafo se genera un árbol de búsqueda.
Ejemplo:
Los lados son en ambas direcciones.



Expansión en árbol de búsqueda de un grafo.



1.7. Terminología

Cuando se construye un nodo para S, se dice que se genera S. Cuando se generan todos los sucesores de un estado S, se dice que se expande S

1.8. Algoritmos

Se hace un análisis en función de:

1. Completitud (si existe un camino el algoritmo lo consigue).
2. Optimalidad (es la solución óptima).
3. Complejidad en tiempo.
4. Complejidad en espacio (uso de la memoria), en el mundo real se prefiere darle prioridad.

Clase 2

Durante esta clase veremos unas definiciones básicas para poder empezar explicar los algoritmos sobre grafos. Además veremos los algoritmos BFS, BFS sin duplicados, DFS y DFID.

2.1. Definiciones

Nodos corresponden a estructuras de datos que representan datos y se guardan en memoria.

La información que se guarda en un nodo n contiene:

1. $state(n)$ = estado representado por n
2. $parent(n)$ = apunta al nodo i "padre" de n
3. $action(n)$ = operador que lleva $state(parent(n))$ en $state(n)$
4. $g(n)$ = costo de llegar a n desde la raíz del arbol de búsqueda

Procedimiento asociados a estados:

1. $init()$ genera una estructura de datos que representa el estado inicial
2. $is\_goal(g)$ chequea si es un estado objetivo
3. $succ(s)$ genera una lista con los sucesores del estado s y las acciones correspondientes

Procedimiento asociados a nodos:

1. $make\_root(s)$ construye la raíz del arbol de búsqueda
   n :=new nodo
   $state(n) := s$
   $parent(n) := null$
   $action(n) := null$
   $g(n) := null$
   return n

2. $make\_node(n, a, s)$ construye un nodo que representa al estado s que es hijo del estado n a traves de la acción a

   n' :=new nodo
   $state(n') := s$
   $parent(n') := n$
   $action(n') := a$
   $g(n') := g(n) + cost(state(n), a)$
   return n'

3. $extract\_solution(n)$ construye el unico camino de la raíz a al nodo n

   $path$:=new lista
   while $parent(n)! = null$ **do**
     $path.push - front(action(n))$
     $n := parent(n)$
   end while
   return $path$

2.2. Búsqueda en amplitud (BFS)

2.2.1. Algoritmo

$queue :=$ new FIFO-queue
$queue.push(node\_root(init()))$
while !queue.empty() **do**
  $n := queue.pop_first()$
  if $is\_goal(state(n))$ **then**
    return $extract\_solution(n)$
  end if
  for all $< s, a >\in succ(state(n))$ **do**
    $n' := make\_node(n, a, s)$
    $queue.push(n')$
  end for
end while
return $null$

2.2.2. Posible mejoras

1. Colocar el if dentro del forEach, es decir chequear la terminación durante la generación

2. Eliminar duplicados

2.3. BFS con eliminacion de duplicados

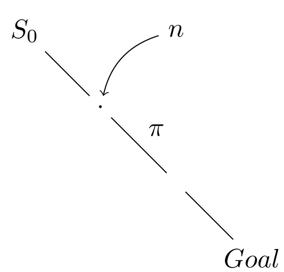
2.3.1. Algoritmo

$queue$:= new FIFO-queue
$queue.push(node\_root(init()))$
$closed$:= new set //conjunto que contiene estados expandidos
while !queue.empty() **do**
  $n := queue.pop_first()$
  if $state(n) \notin closed$ **then**
    if $is\_goal(state(n))$ **then**
      return $extract\_solution(n)$
    end if
    for all $< s, a > \in succ(state(n))$ **do**
      $n' := make_node(n, a, s)$
      $queue.push(n')$
    end for
    $closed.include(state(n))$
  end if
end while
return $null$

2.3.2. Análisis

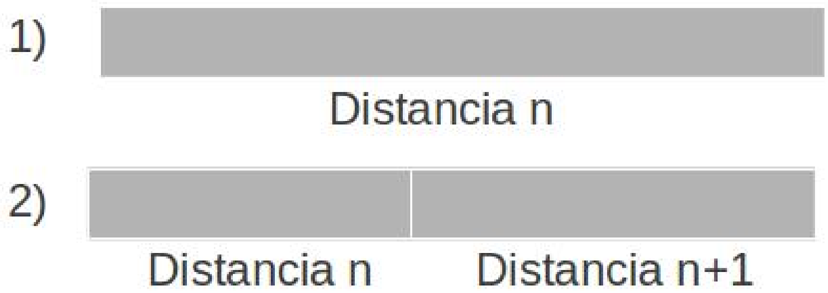
1. Completitud

   Supongamos que el grafo tiene un camino desde el estado incial a un estado goal. Sea π un camino tal.
   Invariante: Al inicio de la iteración queue contiene un nodo (no expandido) que representa un estado en π



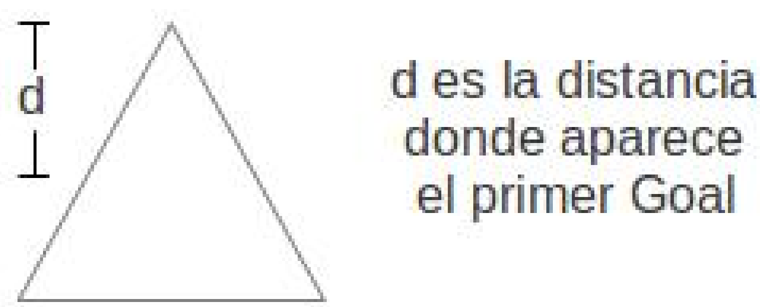
2. Optimalidad

   BFS calcula un camino de menor longitud que es optimo si todos los costos son iguales (costos uniformes). Para ver esto tenemos:
   Invariante: Los nodos en queue estan ordenados por distancia a la raíz. de hecho al inicio de cada iteración todos los nodos de la cola estan a la misma distancia de la raíz ó la cola solo contiene nodos a distancia n y a distancia n+1



3. Complejidad en tiempo y en espacio

   Suponemos que el arbol de búsqueda es un arbol regular de factor de ramificación b

   En el peor caso:

   a) Es el último de la profundidad d
   b) Espacio requerido para la cola es $O(b^d)$
   c) Espacio requerido para closed es $O(\Sigma_{0\leq k\leq d} b^k) = O(b^d)$
   d) Tiempo es $O(\Sigma_{0\leq k\leq d} b^k) = O(b^d)$

2.4. Búsqueda en profundidad(DFS)

2.4.1. Algoritmo

$queue$:= new LIFO-queue
$queue.push(node\_root(init()))$
while !queue.empty() **do**
  $n := queue.pop_first()$
  if $is\_goal(state(n))$ **then**
    return $extract\_solution(n)$
  end if

for all $< s, a > \in succ(state(n))$ **do**
  $n' := make\_node(n, a, s)$
  $queue.push(n')$
end for
end while
return $null$

2.4.2. Análisis

1. Completitud

   Es completo solo para arboles finitos (grafos sin ciclos)

2. Optimalidad

   No existe garantia

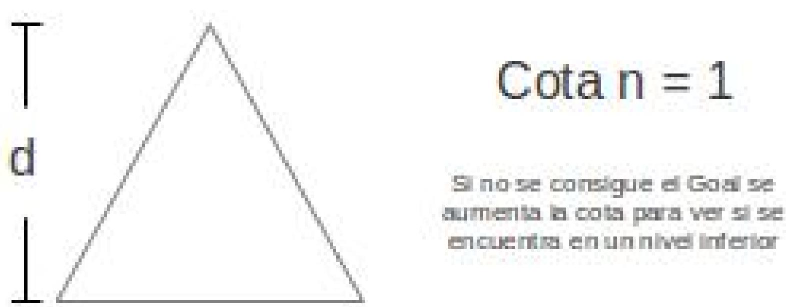
3. Complejidad en tiempo y en espacio

   Espacio solo tiene en la cola los hijos de los nodos no explorados. $O(b * n)$ (espacio lineal)

   Tiempo = $O(b^n)$

2.5. Depth First Iterative Deeploing (DFID)

El arbol de búsqueda se ve asi:



2.5.1. Algoritmo

for all $k = 0$ to co **do**
  $n := make_root(init())$
  $\pi := DFS\_acotada(n, k)$
  if π es camino **then**
    return π
  end if
end for

Donde el algoritmo $DFS\_acotada$ es:

$DFS\_acotada(n, k)$
if $g(n) > k$ **then**
  return $null$
end if
if $is\_goal(state(n))$ **then**
  return $extract\_solution(n)$
end if
for all $< s, a > \in succ(state(n))$ **do**
  $\pi = DFS\_acotada(make\_node(n, a, s), k)$
  if $\pi \neq null$ **then**
    return π
  end if
end for
return $null$

2.5.2. Análisis

1. Completitud

   Es completo

2. Optimalidad

   Optimo para costos iguales (comunes)

3. Complejidad en tiempo y en espacio

   $Espacio = O(b * d)$
   $Tiempo = O(\Sigma_{0 \leq \leq d} b^k) = O(b^d)$

Clase 3

3.1. Uniform Cost Search (UCS) (se eliminan los duplicados)

3.1.1. Algoritmo

$queue := priority\_queue()$ //prioridad de n es g(n)
$queue.insert(make\_root(state(n)))$
$closed := (/)$
while $!queue.empty()$ **do**
  $n = queue.extract\_first()$
  if $state(n) \notin closed$ **then**
    $closed.insert(state(n))$
    if $is\_goal(state(n))$ **then**
      return $extract\_solution(n)$
    end if
    for all $< s, a > \in succ(state(n))$ **do**
      $queue.insert(make\_node(n, a, s))$
    end for
  end if
end while
return $null$

3.1.2. Análisis

1. Completitud

   Es completo si existe una solución la consigue

2. Optimalidad

   Siempre selecciona el camino de menor costo

3. Complejidad en tiempo y en espacio

   Es exponencial en profundidad del goal

3.2. Búsqueda Heurística Informada

Utiliza una función h(,) que mapea estados en números tal que h(s) es un "estimado" del costo de alcanzar un estado goal desde s

Durante la búsqueda la hurística se calcula sobre nodos n como $h(n) = h(state(n))$

En general uno puede generalizar la noción de heurística para que sea una función de nodos en enteros

3.3. Heuristica perfecta h^*

$h^*(s)$ es el costo del camino de menor costo desde s a un estado goal. Si no existe un camino desde s a un estado goal, $h^*(s) = \infty$

Propiedades de las funciones heuristicas:

1. h es segura si $h(s) = \infty => h^*(s) = \infty$ (es segura si las cosas que te dices que descartes son realmente descartables)

2. h "conoce el goal" si $h^*(s) = 0 => h(s) = 0$

3. h es "admisible" si $h(s) \leq h^*(s)$. La propiedad 3 implica la 1

4. h es "monótona" si $h(s) \leq h(s') + c(s, a)$ desde $< a, s' >\in succ(s)$. La propiedad 4 implica al resto

3.4. Greedy Best-First Search (GBFS)

3.4.1. Algoritmo

```
queue := priority_queue() //es ordenada por h(,)
queue.insert(make_root(state(n)))
closed := (/)
while !queue.empty() do
  n = queue.extract_first() //esto sacaria el de menor valor h
  if state(n) ∉ closed then
    closed.insert(state(n))
    if is_goal(state(n)) then
      return extract_solution(n)
    end if
    for all < s, a >∈ succ(state(n)) do
      n' := make_node(n, a, s)
      if h(n') < ∞ then
        queue.insert(n')
      end if
    end for
  end if
```

end while
return *null*

3.4.2. Observaciones

No toma en cuenta la distancia de la raíz al nodo

3.4.3. Análisis

1. Completitud

   Si h es segura (y elimina duplicados)

2. Optimalidad

   No hay garantía

3. Complejidad en tiempo y en espacio

   Es igual al número de estados en el espacio de búsqueda en el peor caso, exponencial.

3.5. Best-First Search (BFS)

3.5.1. Algoritmo

$queue := priority\_queue()$ //es ordenada por g(,)+h(,)
$queue.insert(make\_root(state(n)))$
$closed := (/)$
while $!queue.empty()$ **do**
  $n = queue.extract\_first()$
  if $state(n) \notin closed$ or $g(n) < distancia(state(n))$ **then**
    $distancia(state(n)) := g(n)$
    $closed.insert(state(n))$
    if $is\_goal(state(n))$ **then**
      return $extract\_solution(n)$
    end if
    for all $< s, a > \in succ(state(n))$ **do**
      $n' := make\_node(n, a, s)$
      if $h(n') < \infty$ **then**
        $queue.insert(n')$
      end if
    end for
  end if
end while
return *null*

3.5.2. Observaciones

Si $h(,)$ es 0 entonces la cola de prioridades seria solo ordenada por $g(,)$. Tambien se tiene que siempre se insertan cosas en la cola ya que el ultimo if siempre se cumple y otra cosa que va a ocurrir es que la parte de la condicion $g(n) < distancia(state(n)$ nunca se cumple.

Existen grafos y h admisible y no monótona donde BFS es mucho peor que UCS

Si h es monótona no existen re-expansiones

BFS con h admisible se conoce como A^*

El valor $g + h$ de un nodo se llama el valor f, $f(n) = g(n) + h(n)$

Es recomendable romper empates entre nodos con mismo valor f favoreciendo a los de menor valor h

3.5.3. Análisis

1. Completitud

   Si h es segura

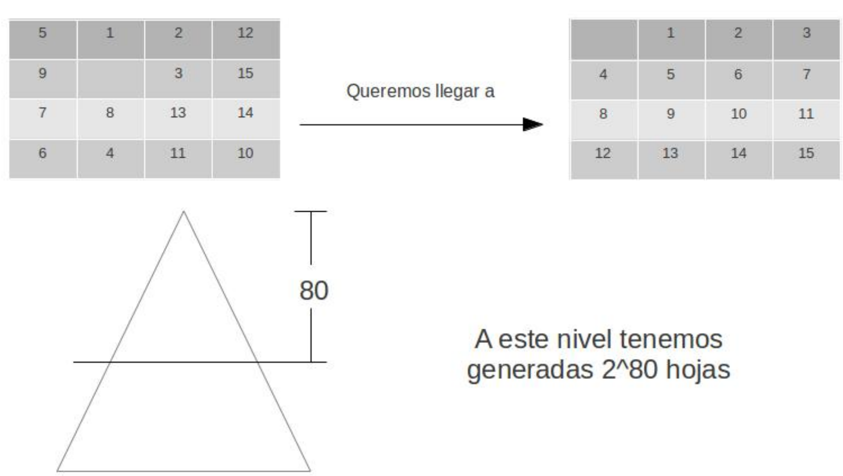
2. Optimalidad

   Si h es admisible

3. Complejidad en tiempo y en espacio

   Exponencial en la profundidad de goal por heurística monótonas

3.6. Ejemplo para 15 puzzle



3.7. Función heurística distancia Manhattan

Suma de las distancias individuales de cada ficha (tile) a su destino final

1. Es monótona

2. Se puede calcular eficientemente precompilando dichas distancias

   Ejemplo de distancia para la ficha 5:

   $dist5 = [2, 1, 2, 3, 1, 0, 1, 2, 2, 1, 2, 3, 3, 2, 3, 4]$

   Hago lo mismo para cada ficha. Posiblemente mejor se representa con un arreglo de arreglos tal como: $dist[i][] = \{\ldots\}$

   El codigo luce así:

   $h = 0$
   for all $i = 0$ to 15 **do**
   $\quad h+ = dist[T[i]][i]$
   end for

Clase 4

4.1. Weighted A^* (WA^*)

Igual a A^* excepto que la heurística h es multiplicada por un peso W.
$f(n) = g(n) + W * h(n)$

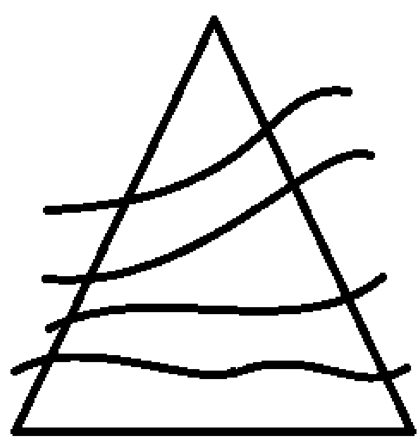
4.1.1. Propiedades

1. W cumple las siguentes características

Si W = 1	=>	Recuperamos A^*
Si W = 0	=>	Recuperamos UCS
Si W = ∞	=>	Recuperamos GBFS

2. Si h es admisible, WA^* retorna una solución que es a lo sumo W veces más costosa que la óptima

DFID es a BFS lo que IDA^* es a A^*
Recordemos que DFID es un algoritmo que incrementa las cotas



4.2. IDA^*

4.2.1. Algoritmo

$n := make\_root\_node(init())$
$t := f(n)[= h(n)]$
$plan := null$
while $plan = null\ \&\ t < \infty$ **do**
  $< plan, new - t >:= DFS\_acotada(n, t)$
  $t := new - t$
end while
return $< plan, t >$

Donde el algortimo DFS_acotada es:

$DFS\_acotada(n, t)$
if $g(n) + h(n) > t$ **then**
  return $< null, g(n) + h(n) >$
end if
if $is\_goal(n)$ **then**
  return $< extract\_solution(n), g(n) >$
end if
$new - t := \infty$
for all $< a, s >\in succ(n)$ **do**
  $n' := make\_node(n, a, s)$
  $< plan, cost >:= DFS\_acotada(n', t)$
  if $plan \neq null$ **then**
    return $< plan, cost >$
  end if
  $new - t := min(new - t, cost)$
end for
return $< null, new - t >$

4.2.2. Análisis

1. Completitud

   Es completo si existe una solución

2. Optimalidad

   Es óptimo si h es admisible

3. Complejidad en tiempo y en espacio

   Espacio: Es lineal en la profundidad de la solución.
   Tiempo: Es similar a A^*, es decir exponencial en profundidad.
   Sin embargo hay un caso patológico cuando se generan solo un nodo, esto ocurre normalmente se trata con numeros reales. Eso tiene una solución con un algoritmo llamado TSP.

4.3. Resumen de los algoritmos

Algoritmo	Completitud	Optimalidad	Tiempo	Espacio
BFS	$\surd$	Si costo $= 1$ $\surd$	b^d	b^d
DFID	Si existe sol $\surd$	Si costo $= 1$ $\surd$	b^d	$b * d$
DFS	X	X	b^n	$b * n$
UCS	$\surd$	$\surd$	b^d	b^d
GBFS	$\surd$	X	b^d	b^d
A^*	$\surd$	$\surd$	b^d	Falta
IDA^*	$\surd$	$\surd$	b^d	b^d

Resultado: A^* expande todo nodo n con $f(n) <$ costo óptimo y algunos nodos n con $f(n) =$ costo óptimo

4.4. Hill Climbing

4.4.1. Algoritmo

$n := make\_root\_node(init())$
while $true$ **do**
  if $is\_goal(state(n))$ **then**
    return $< extract\_solution(n), g(n) >$
  end if
  $succ := make\_node(n, a, s) :< a, s >\in succ(n)$
  $n :=$ Selecciónar nodo n en succ que minimiza h(,)
end while
return $< plan, t >$

4.5. Enforced Hill Climbing

Lo que esta buscando siempre es el nodo que vaya mejorando la heurística, es decir $h(a') < h(n)$. Esto sería saltar al nodo n ya que mejora la heurística (Se le conoce como $Improve$ a esta parte)

4.5.1. Algoritmo

$n := make\_root\_node(init())$
while $n \neq null$ **do**
  if $is\_goal(state(n))$ **then**
    return $< extract\_solution(n), g(n) >$
  end if
  $n := Improve(n)$
end while
return $< null, \infty >$

Donde el algoritmo $Improve$ es:

```
Improve(n_0)
queue := new FIFO-queue
closed := (/)
queue.insert(make_root_node(state(n)))
while !queue.empty() do
  n := queue.pop_first
  if h(n) < h(n_0) then
    return n
  end if
  if state(n) ∉ closed then
    closed.insert(state(n))
    for all < a, s >∈ succ(n) do
      n' := make_node(n, a, s)
      queue.push_back(n')
    end for
  end if
end while
return null
```

4.5.2. Propiedades

Completo para espacios fuertemente conectados y $h(n) = 0$ si y solo si n es un Goal

Clase 5

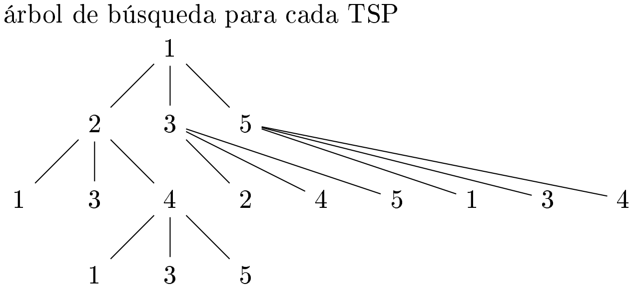
5.1. Problema del agente viajero

Dado un grafo $G = (v, e)$ con costo $c(e)$ para cada $edge \in E$

Dado un vertice $v \in C$, se quiere conseguir un tour con el camino de menor costo.

(dibujo 2)

árbol de búsqueda para cada TSP



El árbol de busqueda es finito para cada TSP y las hojas del árbol representan soluciones

5.2. Branch and bound (ramifica y poda)

Es un algoritmo tipo DFS

5.2.1. Algoritmo

$\alpha := \infty$
$best := null$
$DFS\_BnB(make\_root\_node(init()))$
return $< extract\_solution(best), \alpha >$

Donde el algoritmo DFS_BnB es:

$DFS_BnB(n)$
if $f(n) > \alpha$ **then**

return no es un camino bueno (quiere decir que podamos)
end if
if $is\_goal(n)$ **then**
if $g(n) < \alpha$ **then**
$\alpha := g(n)$
$best := n$
end if
end if
for all $< s, a > \in succ(n)$ **do**
$DFS\_BnB(make\_node(n, a, s))$
end for

5.3. Búsqueda en tiempo reales

Suponen un agente que se mueve (toma decisiones) en el ambiente en tiempo real y quiere alcanzar un estado objetivo.

Ahora veamos algoritmos entrelazan planificación y ejecución:

5.3.1. Learning Real-Time $A^*(LRTA^*)$

Retiene a que la heurística que se modifica a medida que se toman decisiones

$LRTA^*$: H es la heurística
$n := make\_root\_node(init())$
repeat
$LRTA^*\_Trial(n)$
until Alguna condició

Donde el algoritmo $LRTA^*\_Trial$ es:

$LRTA^*\_Trial(n)$
//Es codicioso porque buscaría el costo de la acción mas el $h(n)$ más pequeño

while $!is\_goal(n)$ **do**
for all $< s, a > \in succ(n)$ **do**
$next[a] := c(state(n), a) + H(s)$
$res[a] := s$
end for
Seleccionar a^* que minimiza $next[.]$
$H(state(n)) := next[a^*]$
$n := res[a^*]$
end while

H es una tabla (de Hash) que guarda valores asociados a estados:

1. Cuando se busca el valor para s en H, si no existe una entrada para s, se retorna h(s). Si existe la entrada se retorna el valor de esta

2. Cuando se escribe un valor para s. Si H no es entrada para s, se crea una entrada con el valor dado. Sino se modifica la entrada

Si el ambiente se puede explorar de forma segura, cada ejecución termina en un goal.

Si h es admisible y H se preserva a lo largo de distintas ejecuciones, eventualmente el agente recorre caminos optimos.

5.4. ¿De donde vienen la heurísticas?

Queremos heurísticas que sean:

- Admisibles(mejor si son consistentes)
- Eficientes: i.e. computables en tiempo polinomial (teoria) o constante o lineal o cuadratico a lo sumo(en el practica)

Veamos 2 casos de la heurística de Manhattan

15 puzzle:

Recordemos que dada una configuración lo que hacemos es contar por tile el número de casillas que faltan para llegar a una posición

Este estimado es admisible, por que? -> porque cada tile debe tener un numero de movimientos finito

Idea: esto es una solución optima al problema simplificado

5.5. Simplificación

Problema corresponde a grafo $G = (v, e)$ con costos $c : E- > Reales$ Una simplificación es un grafo $G' = (v', e')$ con costo $c' : E'- > Reales$ tal que:

1. $v \subset v'$
2. $E \subset E'$
3. $c'(e) \leq c(e)$ para $e \in E$

   $(v, v') \in E => (\alpha(v), \alpha(v')) \in E'$

5.6. Heurísticas basadas en patrones (Pattern Database)

Dada una configuracion cualquiera del 15 puzzle y queremos llegar a la configuracion en la que estan ordenados los numeros lo que hago es agarrar un grupo de fichas, digamos 4 y el vacio y el resto lo marco como X (don't care) entre ellas indistiguibles y lo mismo hago con el tablero final, esdecir tomo esos mismos numeros y el blanco y el resto lo marco con X igualmente indistinguibles ahora me quedan $(16!/11!) = (15 * 14 * 15 * 13 * 12)$ configuraciones que esto es mucho menor que el problema original que es 16!

Clase 6

6.1. Problemas de descomposición

Asociadas a tareas que pueden resolverse con una estrategia ”dividir y conquistar” en donde la tarea se divide en subtareas que deben resolverse para obtener una soluciones.

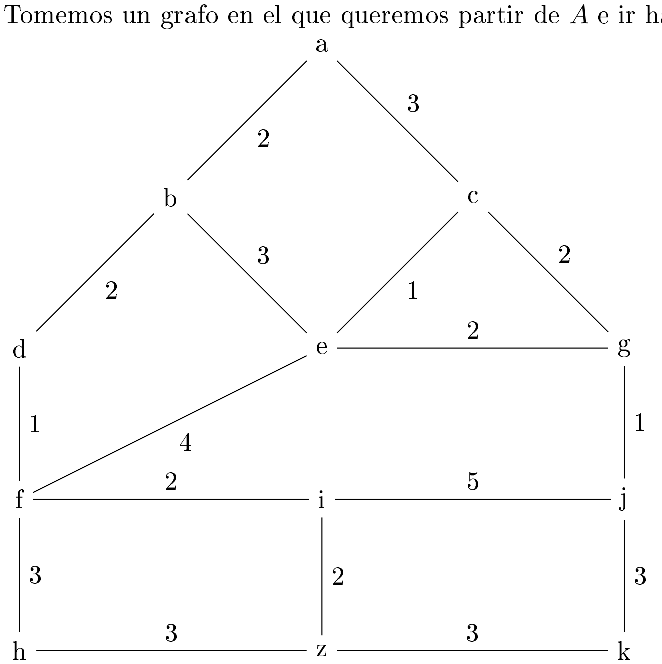
Esta división recursica de tareas en subtareas crea una jerarquía de dependencias entre subtareas que se representa con un grafo

Ejemplos:

Torres de Hanoi de 3 astas Move(1..n,1,3): Move(1..(n-1),1,2); Move(n,1,3);Move(1..(n-1),2,3)

Move(1..(n-1),1,2): Move(1..(n-2),1,3); Move(n-1,1,2);Move(1..(n-2),3,2)

Tomemos un grafo en el que queremos partir de A e ir hasta Z



Las tareas serían:
$Path(a->z):$
$Path(a-z,f)$
$Path(a-z,g)$
$Path(a-z,f):$
$Path(a-f); Path(f-z)$
$Path(a-z,g):$
$Path(a-g); Path(g-z)$
$Path(a-f)$

6.2. Grafo AND/OR

Grafo dirigido con costo compuesta por dos tipos de nodos: AND y OR
Cada nodo representa una tarea a resolver
AND: Representa una tarea cuya solución necesita de la solución de todas las tareas que corresponden a sus hijos
OR: Representa una tarea cuya solución necesita de la solución de uno de sus hijos
Suposición: El grafo AND/OR de dependencias entre tareas es acíclico

6.2.1. ¿Que es una solución para cada grafo AND/OR?

En general una solución de un grafo G es un subgrafo G' tal que:

1. La raiz de G es raiz de G'
2. Si $n \in G'$ y es AND, entonces todos sus hijos pertenecen a G'
3. Si $n \in G'$ y es OR, entonces un hijo de n pertenecen a G'

(dibujo 2)

6.2.2. Costo de la solución

Podemos tener varios criterios:

- La suma de los costos de todas las aristas en la solución
- El costo del camino mas largo en la solución
- Costo promedio
- etc

6.3. AO^*

Es un algoritmo Best-First para grafos AND/OR (utiliza heuristicas)

Best-First significa que se mantiene un conjunto de soluciones "parciales". Se selecciona la mejor solución parcial y se expande. Esot termina cuando la mejor solución parcial es una solución

AO^* mantiene una representación G explicita del grafo AND/OR implicito (dibujo 3)

G es un grafo AND explicito y "parcial", i.e. que tiene nodos en la frontera que nos son terminales

El grafo NO puede tener ciclos, el algoritmo solo funciona para grafos aciclicos

La mejor solución de G es el grafo G^*

6.3.1. Algoritmo

1. Inicialización
2. G contiene la raiz S_0 del grafo
3. $G^* = G$
4. $V(S_0) = h(S_0)$
5. lazo
   - *a*) Seleccionar un nodo n frontera en G^* que no es terminal. Si no existe tal nodo no terminal. Entonces G^* es la solución [implementado un DFS]
   - *b*) Expandir n: para cada hijo n' de n, Si n' existe en G, agregar la arista $n- > n'$. Sino agregar n' en G, la arista $n- > n'$ y inicializar $V(n') = h(n')$
   - *c*) Recomputamos el valor de n y todos sus ancestros(recursivamente):
     Caso AND:
     $$V(n) = \Sigma_{hijosDeN} c(n, n') + V(n')$$
     Caso OR:
     $V(n) = MIN_{hijosDeN} c(n, n') + V(n')$ [Adicionalmente la mejor arista la marcamos]
     Donde hijosDeN es $= n'$ hijos de n

Para mejores referencias leer NILSSON: Principles of AI

Clase 7

7.1. Sistema de transición no deterministico

Caracterizado por:

1. Un espacio S de estados
2. Un estado inicial $S_0 \in S$
3. Un conjunto $S_G \subset SdeGoals$
4. Acciones $A(s)$ aplicables es estado $s \in S$
5. Una funció de transición no deterministica $F(s,a) \subset S$ para $s \in S$ y $a \in A(s)$
6. Costos positivos $c(s,a)$ para $s \in S$

Una solución no puede ser una secuencia de acciones. En general, una solución estrategica que indica qie acción tomar en cada estado del problema. Se representa con una función $\pi : S- > A$ (Con la restricción que $\pi(s) < -A(s)$ para todo $s \in S$)

Ejemplo:

I$\downarrow$	$\downarrow$	Lleno	G
$\downarrow$	$\downarrow$	Lleno	$\uparrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$	$\uparrow$
$\rightarrow$	$\rightarrow$	$\rightarrow$	$\uparrow$

Robot que navega un ambiente no determinado (Las acciones tienen efectos no deseados)

Dada una politica $\pi : S- > A$, definimos las ejecuciones de π que comienzan en S_0

Una ejecucción que comienza en S_0 es una secuencia $< S_0, S_1, S_2, \ldots >$ de estados tal que: $S_{i+1} \in F(S_i, x(s_i))$ para $i \geq 0$

Definimos cuando π es solucion en terminos de sus ejecuciones. Si todas las ejecuciones terminan en un estado goal, decimos que π es una ”solucion fuerte”.

Una estrategia π es "solucion fuertemente ciclica" si todas las ejecuciones que no terminan en un goal son injustas.

Una ejecucion $< S_0, S_1, \ldots >$ es injusta ssi existe un estado s que aparece un numero infinito de veces en la ejecucion y un estado $s' \in F(s, \pi(s))$

7.2. Conectarlo con la idea de los grafos AND/OR

$NodosOR$: estados del problema donde se selecciona accion
$NodosAND$: Son de la forma (s, a) donde $s \in S$ y $a \in A(s)$
$Aristas$: Conectan nodos OR s con nodos AND (s, a) para $s \in S$ y $a \in A(s)$
Nodos AND (s, a) se conectan con nodos OR s' tal que $s' \in F(s, a)$

$$
\begin{tikzcd}
& (1,1) \arrow[dl] \arrow[dr] & \\
(1,1,\leftarrow) & & (1,1,\downarrow) \arrow[d] \\
& & (1,2)
\end{tikzcd}
$$

Grafo AND/OR tiene ciclos pero existen soluciones fuertes
Consideramos el árbol AND/OR de posibles acciones:

- La raiz es $< s_0 >$ es un nodo OR
- Los sucesores de la raiz son de la forma $(< s_0 >, a)$ para cada $a \in A(s_0)$ y corresponde a nodo AND
- Los hijos $(< s_0 >, a)$ son nodos OR de la forma $< s_0, s_1 >$ para cada $s \in F(s_0, a)$

En general los nodos OR son de la forma $< S_0, S_1, \ldots, S_n >$ y los nodos AND son de la forma $(< S_0, \ldots, S_n >, a)$ con $a \in A(S_n)$

Grafos AND/OR con ciclos y soluciones fuertemente aciclicas.

Dada una politica $\pi : S- > A$, definimos dos funciones que mapean estados en costos.

$V^{\pi}(S) = c(S, \pi(S)) + MAX_{s' \in F(s, \pi(s))} V^{\pi}(S')$
$V^{\pi}_{opt}(S) = (S, \pi(S)) + MIN_{s' \in F(s, \pi(s))} V^{\pi}_{opt}(S')$
Propiedades:

- π es solucion fuerte ssi $V^{\pi}(S_0) < \infty$
- π es solucion fuertemente ciclica ssi $V^{\pi}_{opt}(s') < \infty$ para cada s' que es "alcanzable" a partir de s_0 usando π

7.2.1. Solucion de $V^{\pi}(.)$

Dado π fijo comenzamos con un estimado $V_0(S) \equiv 0$ Iterativamente conseguimos un nuevo estimado
$V_{RTI}(S) = c(S, \pi(S)) + MAX_{s' \in F(s,\pi(s))} V_k(S')$
$Resultado : V_0, V_1, V_2, \ldots -> V^{\pi}$

7.2.2. ¿Como consigo un π que cumpla las propiedades?

Ecuaciones de Bellman:
$V^*(S) = MIN_{a \in A(S)}[c(s,a) + MAX_{s' \in F(s,a)} V^*(S')]$
$V^*_{op}(S) = MIN_{a \in A(S)}[c(s,a) + MINV^*_{opt}(S')]$
Estas ecuaciones se pueden resolver de una forma iterativa

7.3. Computo de solucion fuertemente ciclica

1. Sea $s' = s$ y $A'(s) = A(s)$ para cada $s \in S$
2. Encontrar solucion $V^*_{opt}(.)$ para s' y $A'(.)$
3. Eliminar de s' todos los estados s tal que $V^*_{opt}(s) = \infty$
4. Eliminamos acciones a de $A's$ tales que $s' \in F(s,a)$ sea un estado eliminado
5. Ir a 2

El problema tiene solucion (fuertemente ciclica) ssi $S_0 \in S'$ (al finalizar el algoritmo). En dicho ciclo la solucion es:
$\pi(S) = ARGMIN_{a \in A'(S)}[c(s,a) + MIN_{s' \in F(s,a)} V^*_{opt}(S')]$
Para todo $s \in S'$

Clase 8

8.1. Arboles de juego

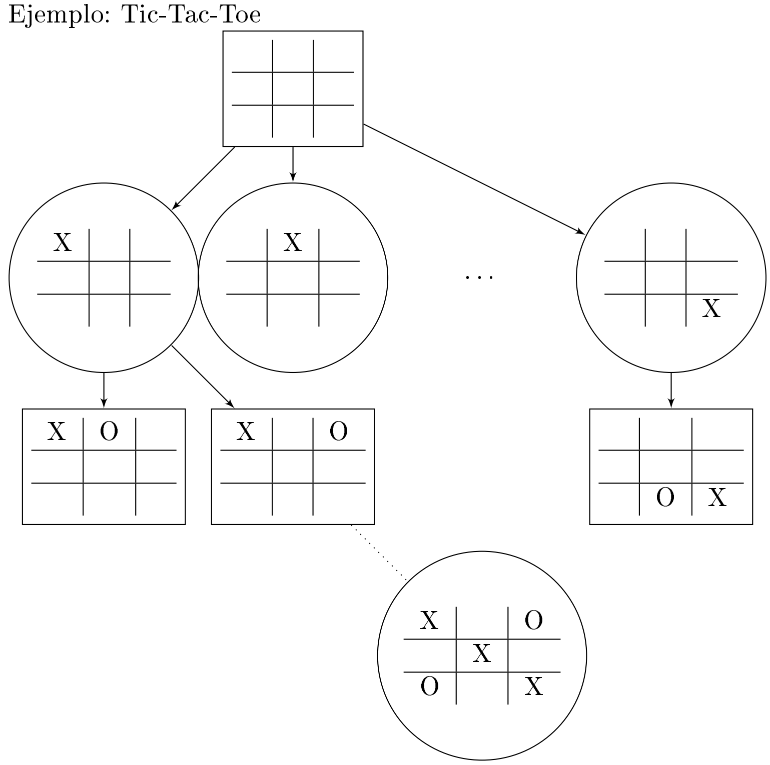
Modelo para juegos de 2 personas de suma cero, deterministicos y con información completa.

Ejemplos: Ajedrez, Damas, Tic-Tac-Toe, Nim, Go, etc

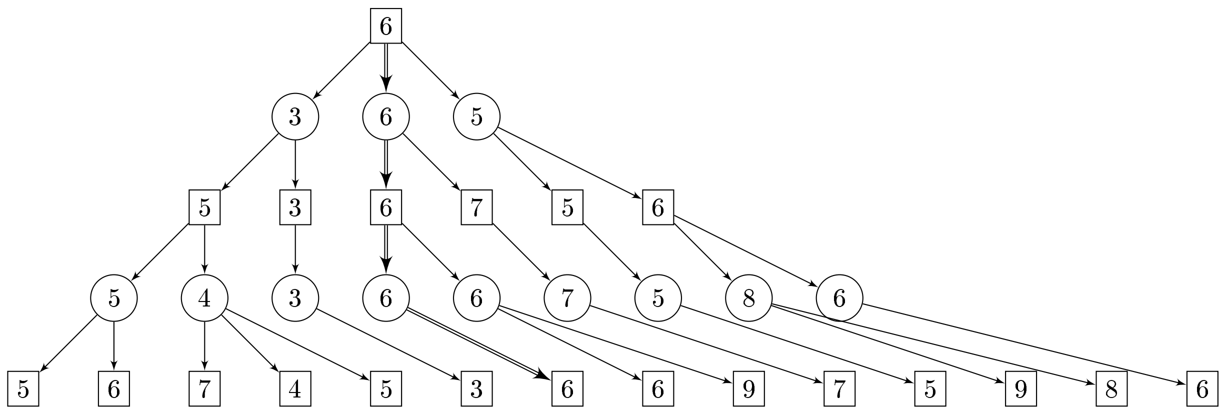
Se representan con un arbol con dos tipos de nodos llamada MAX y MIN que representan a los dos jugadores

El arbol es bipartito donde los hijos de nodos MAX son nodos MIN y viceversa, ya que el arbol representa las jugadas

Ejemplo: Tic-Tac-Toe



Otro juego



Todo este arbol se le conoce como el valor MINMAX del juego o valor de juego. La ruta que tiene doble flecha se le conoce como variación principal.

8.2. Minmax

Computa el valor del juego

Observe que el $max(a, b) = -min(-a, -b)$ ejemplo: $max(5, 8) = -min(-5, -8)$

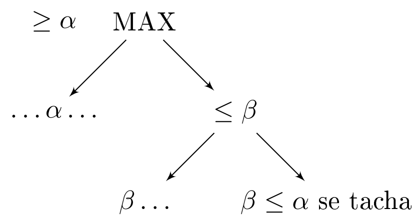
8.2.1. Algoritmo

$minmax(n, d)[Negamax]$
if n es terminal o $d = 0$ **then**
  return α
end if
$\alpha := -\infty$
for all $n' \in succ(n)$ **do**
  $\alpha := max(\infty, -minmax(n', d-1))$
end for
return α

8.2.2. Análisis

- Arbol de ramificación b.
- Tiempo $= O(b^d)$
- Espacio $= O(bd)$

Si yo estoy explorando un nodo MAX y llego a una configuración donde el hijo tiene valor α y otro hijo tiene un hijo β no hace falta revisar los hermanos de β si $\beta \leq \alpha$



8.3. $\alpha\beta Pruning$

Calcula el valor minmax del juego tratando de reducir al maximo el Hsadg de evaluaciones

8.3.1. Algoritmo

La llamada inicial del algoritmo debe ser: $\alpha\beta{-}pruning(raiz, d, -\infty, \infty, MAX)$

$\alpha\beta - pruning(n, d, \alpha, \beta, t)$
if n es terminal o $d = 0$ **then**
  return $h(n)$
end if
if $t = MAX$ **then**
  for all $n' \in succ(n)$ **do**
    $\alpha := max(\alpha, \alpha\beta - pruning(n', d-1, \alpha, \beta, MIN))$
    if $\alpha \geq \beta$ **then**
      $break$
    end if
  end for
  return α
end if
if $t = MIN$ **then**
  for all $n' \in succ(n)$ **do**
    $\beta := min(\beta, \alpha\beta - pruning(n', d-1, \alpha, \beta, MAX))$
    if $\alpha \geq \beta$ **then**
      $break$
    end if
  end for
  return β
end if

8.3.2. Análisis

- Espacio $= O(bd)$
- Tiempo

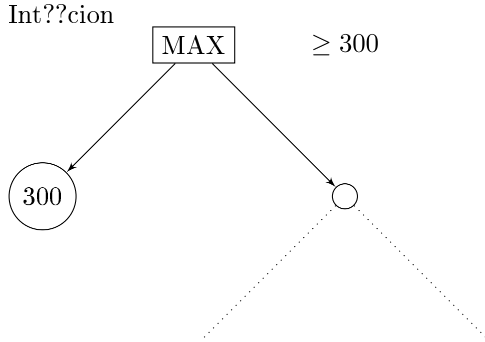
- Peor caso: no existe poda y tiempo es el mismo de minmax $O(b^d)$
- Mejor caso: $O(b^{d/2}) = O(\sqrt{b^d})$, gracias a esto puedo ver el doble de posiciones hacia delante
- Caso promedio: $O(b^{(3d)/4})$

Clase 9

9.1. Arboles de juego

- Minmax: $O(b^d)$
- $\alpha\beta$ Pruning: $O(b^{d/2})$, $O(b^{d3/4})$, $O(b^d)$
- Scout: Negascout

Int??cion



9.1.1. Algoritmo - Primera vista

$Test(n, d, v, >) \rightarrow return$ true
if $d = 0$ o $terminal(n)$ Si el valor minmax el nodo n es $> v$ **then**
  if $h(n) > v$ **then**
    return $true$
  else
    return $false$
  end if
end if
for all $n' \in Succ(n)$ **do**
  if n es MAX & $test(n', d-1, v, >) = true$ **then**
    return $true$
  end if

if n es MIN & $test(n', d-1, v, >) = false$ **then**
return $false$
end if
end for
if n es MAX **then**
return $false$
end if
if n es MIN **then**
return $true$
end if

Test necesita de al menos $\Omega(b^{d/2})$
Ahora usamos $Test(.)$ para calcular el valor minmax de un nodo

9.1.2. Algoritmo - Segunda vista

$Scout(n, d)$ (Calculando el valor minmax de un nodo)
if $d = 0$ o $terminal(n)$ **then**
return (n)
end if
$let n_1, n_2, \ldots, n_m$ los sucesores de n
$v := scoot(n_1, d-1)$
for $i = 2$ to m **do**
if n es MAX & $test(n_i, d-1, v, >) = true$ **then**
$v := scoot(n_i, d-1)$
end if
if n es MIN & $test(n', d-1, v, >) = false$ **then**
$v := scoot(n_i, d-1)$
end if
end for

Empiricamente se observa que scoot es bueno en juegos bajo factor de ramificación y que son profundos El pseudocodigo que nos coloco de "minmax" era Negamax

(Combinando la idea de hacer un test antes de calcular los valores minmax...)

Minmax - Negamax
Scoot $\alpha\beta$Pruning - Negascoot

No va a colocar el codigo si quieren se los piden va a colocar propiedades

$\alpha\beta$ - Pruning

$\alpha < \beta$ Si puede llamar con otros valores ¿Que pasa cuando es así?

- Una falla por arriba de $\alpha\beta$ - Pruning con una ventana $[\alpha, \beta]$ sucede cuando el valor retornado es $\geq \beta$. Si $\alpha\beta$ no se llama con una ventana suficiente amplia no retorna el valor de juego ...
- Una falla por abajo con una ventana $[\alpha, \beta]$ sucede cuando el valor retornado es $\geq \alpha$

- Una ventana nula es una ventana de la forma $[m, m+1]$
  - Falla por arriba ssi el valor del nodo es $\geq m+1$ Es equivalente a que $test(n, d, m, >)$ retorne true
  - Falla por abajo ssi el valor del nodo es $\geq m$ ssi $test(n, d, m, >) = false$
- En conclusion llamar a $\alpha\beta - pruning$ con la ventana nula es equivalente a $test(n, d, m, >)$

Recordemos Minmax (2 procedimientos mutuamente recursivos F y G)

$F(n, d) \leftarrow n$ es MAX
if $d = 0$ o terminal(n) **then**
  return $h(n)$
end if
$v := -\infty$
for all $n' \in Succ(n)$ **do**
  $q := G(n, d-1)$
  $v := max(q, v)$
end for
return v

$G(n, d)$
if $d = 0$ o terminal(n) **then**
  return (n)
end if
$v := +\infty$
for all $n' \in Succ(n)$ **do**
  $q := F(n', d-1)$
  $v := min(q, v)$
end for
return v

Scoot $\alpha\beta$ pruning implementado con 2 procedimientos mutuamente recursivos F' y G'

$F'(n, d, \alpha, \beta)$
if $d = 0$ o terminal(n) **then**
  return $h(n)$
end if
$m := -\infty$
$Letn_1, n_2, \ldots, n_k$ //m es una cota inf
$m := max(m, G'(n_1, d-1, \alpha, \beta))$
if $m \geq \beta$ **then**
  return m
end if

```
for i = 2 to k do
   t := G'(n_i, d − 1, m, m + 1)
   if t > m then
      if t ≥ β then
         m := t
      else
         m := G'(n_i, d − 1, t, β)
      end if
   end if
   if m ≥ β then
      return m
   end if
end for
return m

G'(n, d, α, β)
if d = 0 o terminal(n) then
   return h(n)
end if
m := +∞
Letn_1, n_2, . . . , n_k //sucesores de n
m := min(m, F'(n_1, d − 1, α, β))
if m ≤ α then
   return m
end if
for i = 2 to k do
   t := F'(n_i, d − 1, m, m + 1)
   if t ≤ m then
      if t ≤ α then
         m := t
      else
         m := F'(n_i, d − 1, t, β)
      end if
   end if
   if m ≥ d then
      return m
   end if
end for
return m
```

El proyecto:

Informe y una tabla que resuma los resutados de la corrida del algoritmo. En la pagina hay muchos problemas. Empaquetado y README para que sepa compilar y correr

Clase 10

10.1. Proyecto 2: Juego de otello

Correr los siguientes algoritmos y tomar tiempo en cada uno y dar resultados:

- minmax
- $\alpha\beta - pruning$
- $negascout(\alpha\beta - pruning + scout)$

Tiempo: 2 Semanas

Podemos agregar una tabla de trasposición (si me encuentro un estado y otro es igual, entonces ya no tengo que volver a calcular). Variación principal debe tener el mismo valor, por ejemplo diferencia de 5 fichas se mantiene durante toda la variación principal

10.2. Planificación automatica

Enfoque basado en modelo para el comportamiento autonomo

$$\xrightarrow{\text{Descripción de un problema}} \text{Planificación} \xrightarrow{\text{Solución}}$$

La descripción del problema es en un lenguaje de alto nivel*

Planificador = Solucionador de problemas pertenecientes a la clase de problemas expresables en el lenguaje.

Existe un compromiso entre la expresibilidad del lenguaje de descripción y la eficiencia del planificador

10.3. Planificación clásica

Es el modelo mas simple que corresponde a problemas deterministicos con información completa

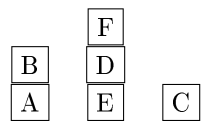
Lenguajes de representación: Strips, ADL, SAS

La planificación clasica se corresponde en problemas que se pueden modelar como problemas de busqueda de caminos en grafos

10.3.1. STRIPS

SU caracteristica basica es que los estados del mundo se representan como asignaciones en valores de verdad a un conjuntos de proposiciones

Ejemplo Black world, cuyo estado inicial luce asi:



$$\begin{array}{llll} on\_table(A) = T & clear(B) = T & on(B,A) = T & handempty = T \\ on\_table(E) = T & clear(A) = F & on(B,C) = F & Hand(C) = F \\ on\_table(V) = F & clear(F) = T & on(D,E) = T & Hand(A) = F \\ & & on(F,B) = T & Hand(B) = F \\ & & on(A,B) = F & \end{array}$$

Supongamos que agarro la caja C entonces la variable $handempty$ tomaria el valor F y la proposición $Hand(C)$ tomaria valor T

Ahora supongamos que nuestra meta final es que $on(A,B)$ tome el valor T, a esto lo llamaremos Situación goal.

Una acción en STRIPS es una tabla de forma a =<Pre,Add,Del> (conjunto de proposiciones)

La acción a es aplicable en estado S ssi $\forall p \in Pre,\ S \models p$ (i.e S asigna true a p)

El resultado de aplicar a en S es un estado $s' = res(a,S)$ tal que

$$s'[p] = \begin{cases} s[p] & si \quad p \notin Add \bigcup Del \\ True & si \quad p \in Add \\ False & si \quad p \in Del \end{cases} \tag{10.1}$$

Ejemplo:

$$Put\_on\_table(c) = \begin{cases} Pre & = & \{hand(c)\} & \text{//esto tiene que ser true} \\ Add & = & \{handempty, on_table(c), clear(c)\} & \text{//esto se transforma en true} \\ Del & = & \{hand(c)\} & \text{//esto se vuelve false} \end{cases} \tag{10.2}$$

Una valuación S sobre un conjunto de proposiciones F se puede representar como el subconjunto siguiente: $\{p \in F : s[p] = true\}$

En el caso del ejemplo seria:

$S = \{clear(B), on(B,A), on_table(A), clear(F), on(F,D), on(D,E), on\_table(E), hand(C)\}$

Tomando un estado S como subconjunto de proposiciones:

- a es aplicable en $s <=> PRE <= S$

- $Res(a,s) = (S/Del) \bigcup Add$

P = (F,I,G,A) donde:

- F es un conjunto de proposiciones
- $I \subseteq F$ define situación inicial
- $G \subseteq F$ define situaciones goal
- A es un conjuunto de acciones de la forma $a =< Pre, Add, Del >$ donde $Pre, Add, Del \subseteq F$

Continuemos con el ejemplo:
$F = \{on\_table(A), \ldots, On(A,B), \ldots, clear(A), \ldots, hand(A), \ldots, handempty\}$
Definamos unas cuantas funciones extras para ayudarnos a resumir operaciones:

$Put\_on\_table(A)$
$Pre = \{hand(A)\}$
$Add = \{on\_table(A), clear(A)\}$
$Del = \{hand(A), handempty\}$

$Pick\_from\_table(A)$
$Pre = \{handempty, clear(A), on\_table(A)\}$
$Add = \{hand(A)\}$
$Del = \{handempty, clear(A), on\_table(A)\}$

$Put\_on\_block(A,B)$
$Pre = \{hand(A), clear(B)\}$
$Add = \{on(A,B), handempty, clear(a)\}$
$Del = \{clear(B), hand(A)\}$

$Pick\_from\_block(A,B)$
$Pre = \{on(A,B), handempty, clear(A,B)\}$
$Add = \{hand(A), clear(B)\}$
$Del = \{on(A,B), handempty, clear(A)\}$

10.4. STRIPS

Los problemas STRIPS $P = (F, I, G, A)$ define un espacio de busqueda:

- Estados son subconjuntos de F
- Estado inicial $S_0 = F$
- Estados goal $S_G = \{s \in S | G \subseteq S\}$
- Acciones $A(s) = \{a \in A | Pre(a) \subseteq s\}$
- Función M transición $f(n,s)$ esta dado por $f(a,s) = [S - Del(a)] \bigcup Add(a)$
- Costos iguales a 1

Soluciones son caminos que llenan S_0 a un estado en S_G

10.5. Problema de decision

Dado unproblema STRIPS $P = (F, I, G, A)$ ¿existe un plan que lo solucione?¿Cuanto recurso computacional se necesita para resolver el problema? $=> STRIPS \in PSPACE$

Clase 11

11.1. Heuristicas para planificación

Problema STRIPS ... donde:

- F es un conjunto ...
- $I \subseteq F$ ...
- $G \subseteq F$ ...
- A es el conjunto de acciones de forma $a = (Pre, Add, Del)$ donde $Pre, Add, Del \subseteq F$

P define un problema de busqueda

- Estados spm subconjunto de F
- $S_0 = I$
- $S_G = \{s \in 2^F : g \in s\}$
- $A(s) = \{a : Pre(a) \subseteq s\}$
- $f(a, s) = (S - Del(a)) \bigcup Add(a)$ para cada $a \in A(s)$
- Costos unitarios (o dos costos)

11.2. Algoritmos para planificación

- Tipicamente lo que se quiere es una solución sin importar la calidad
- Se utilizan algoritmos suboptimos
  - $WA^*, WIDA^*, GBFS, EHC$
- Heuristicas:Se obtienen a partir de una simplificación o relajación del problema

11.3. Delete Relaxation

$P =< F, I, G, A >$ si aplicamos Delete Relaxation nos queda $P^+ =< F, I, G, A^+ >$ donde $A^+ = \{(Pre, Add,) : (Pre, Add, Del) \in A\}$

Heuristicas $h(s) = h_{P^+}(s) < - - -$ Heuristicas sobre la ... P^+

11.3.1. Observaciones sobre P^+

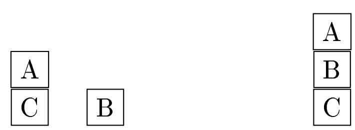
Veamos algunas pistas sobre la simplificación:

- P^+ no se puede resolver de forma optima (de forma eficiente)
- Sin embargo si exite un plan para P es facil conseguir uno para P^+
- Los planes para P^+ son de tamano lineal (i.e son ”costos”)
- P^+ es descomponible y P no lo es

11.3.2. Calcular un plan en P^+

P^+ es descomponible. Ejemplo considere un goal $G = \{p, q\}$. Entonces podemos descomponer G en dos subproblemas $G_1 = \{p\}$ y $G_2 = \{q\}$

Ejemplo $p = on(A, B)$ y $q = on(B, C)$. Si tenemos planes π_1 para G_1, ¿Podemos combinarlos en un plan para $G_1 \bigcup G_2$?



Estado Inicial   Estado Final

$\}- > on(A, B)$

$$\left.\begin{array}{l} pick\_from(A, B) \\ put\_table(A) \\ pick(B) \\ put\_on(B, C) \end{array}\right\} - > on(B, C) \tag{11.1}$$

11.4. Heuristica aditiva

$h_{add}(s) \doteq h_{add}(G, s)$ El costo de obtener G a partir de s en P^+

$h_{add}(c, s) \doteq \Sigma_{p \in C} h_{add}(p, s)$

$$h_{add}(p, s) \doteq \begin{cases} 0 & p \in S \\ min_{a \in O(p)}(c(a) + h_{add}(Pre(a), s)) & p \notin S \end{cases} \tag{11.2}$$

Recordar: $O(p)$ son las acciones que agregan p, i.e. $O(p) = \{a : p \in Add(a)\}$

11.4.1. Algoritmo

h es un arreglo que guarda los valores $h_{add}(p,s)$ para todo $p \in F$
Inicializar:

$$h[p] \doteq \begin{cases} 0 & p \in S \\ \infty & p \notin S \end{cases} \tag{11.3}$$

$Repetir$:
foreach $p \in F$ tal que $h[p] > 0$ do
$h[p] := min_{o \in O(p)} c(a) + \Sigma_{q \in Pre(a)} h[q]$
$until$ no exista cambio en h
$return \Sigma_{p \in G} h[p]$

Sin embargo h_{add} no es admisible, para verlo considere el problema:
$P =< F = \{p,q\}, G = \{p,q\}, I = VACIO, A >$
$A = \{a = (VACIO, p, q, VACIO)\}$
$h_{add}(I) = 2$
$h^*(I) = 1$
Si yo quiero una heuristica admisible entonces busco el h_{MAX}

11.5. Heuristica Max

$h_{max}(s) \doteq h_{MAX}(G,s)$ El costo de obtener G a partir de s en P^+
$h_{add}(c,s) \doteq Max_{p \in C} h_{add}(p,s)$

$$h_{MAX}(p,s) \doteq \begin{cases} 0 & p \in S \\ min_{a \in O(p)}(c(a) + h_{MAX}(Pre(a),s)) & p \notin S \end{cases} \tag{11.4}$$

11.5.1. Algoritmo

El algoritmo quedaria así:
h es un arreglo que guarda los valores $h_{max}(p,s)$ para todo $p \in F$
Inicializar:

$$h[p] \doteq \begin{cases} 0 & p \in S \\ \infty & p \notin S \end{cases} \tag{11.5}$$

$Repetir$:
foreach $p \in F$ tal que $h[p] > 0$ do
$h[p] := min_{o \in O(p)} c(a) + MAX_{q \in Pre(a)} h[q]$
$until$ no exista cambio en h
$return Max_{p \in G} h[p]$

Esta heuristica es admisible e informativa

11.6. Construcción de un plan para G en P^+

Observaciones:

- un plan optimo o "bueno" en P^+ no repite acciones
- Un plan sin acciones repetidas, es decir un plan bueno puede representarse como el conjunto de acciones en el plan

Planes en P^+ se representan como un conjunto de acciones. Veamos la construcción del grafo de planificación para P^+ (este es un grafo que tiene niveles)
$P_0 =$ "las proposiciones en S"
$A_0 = \{a \in A : Pre(a) \subseteq\}$
$P_1 = P_0 \bigcup \{Add(a) : a \in A_0\}$
$A_1 = \{a \in A : Pre(a) \subseteq P_1\}$
.
.
$P_i = P_{i-1} \bigcup \{Add(a) \in A_{i-1}\}$
$A_1 = \{a \in A : Pre(a) \subseteq P_i\}$
El # de capas esta acotado por $\mid F \mid$. Sea P_m la primera capa tal que $P_m = P_{m+1}$

1. Si $!(\subseteq P_m)$ no existe plan para s en $P^+ =>$ no existe plan para s en P
2. Si $G \subseteq P_m$, asuma que m es el menor indice tal que $G \subseteq P_m$ ["basta generar P_m hasta que $G \subseteq P_m$"]

$P \mid$
$A \mid$
$P \mid$
$A \mid$
.
.
$P_{m-1} \mid <-$ no tiene P_k
$A_{m-1} \mid a_k$
$P_m = \{P_1, P2, \ldots, P_k\}$
$G' = [G/Add(a_k)] \bigcup Pre(a_k)$
Despues de calcular el grafo de planificación se calcula un plan π para s en P^+ (plan relajado)
La heuristica se define: $h_{FF}(s) = \Sigma_{a \in \pi} c(a)$
h_{FF} es la heuristica usada por el planificador FF