

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



## Tema 1 - Sistemas Inteligentes y Búsqueda

Profesor Antonio González Muñoz, Curso 20-21

### Sistemas Inteligentes

- Un sistema inteligente (SI) es un sistema software que muestra un comportamiento inteligente o interactúa de forma más inteligente con su entorno y con otros sistemas.
- La barrera entre un software normal y un sistema inteligente queda un tanto difusa, al igual que la barrera entre los Sistemas Inteligentes y la Inteligencia Artificial.
- La Inteligencia Artificial se ocupa de la investigación básica en la implementación de cada una de las habilidades relacionadas con la inteligencia humana.
- Los Sistemas Inteligentes se encargan de aplicar la investigación, tratando de solucionar problemas ya existentes o mejorar nuestra forma de trabajar o calidad de vida aplicando técnicas de IA.

#### **Problemas en IA**

- Los humanos normalmente somos más eficientes que las máquinas para resolver problemas NP-Duros.
  - "Si un problema en IA tiene solución es que se ha escogido mal el problema"

#### **Construcción de Sistemas Inteligentes**

- La Inteligencia Artificial es una rama de la informática que estudia y resuelve problemas situados en la frontera de la misma.
- Se basa en dos ideas fundamentales:
  - Representación del conocimiento explícito y declarativo.
  - Resolución de problemas por heurísticas.

### Búsqueda Heurística y Propiedades:

#### **Heurísticas y tipos de problemas**

- Las heurísticas son criterios, métodos o principios para decidir cuál de entre varias acciones promete ser la mejor para alcanzar una determinada meta.
- **Problema de las 8 Reinas**
  - Colocar 8 reinas en un tablero de ajedrez sin que ninguna pueda capturar a otra.
  - Se puede tener una heurística que cuente el número de casillas libres luego de colocar una reina y otra que obtenga el mínimo de casillas libres por cada fila; esta última es una heurística mucho mejor.
  - Cuanto más general sea una heurística a un tipo de problemas, mejor porque se puede adaptar a múltiples problemas.
  - Este es un problema de satisfacción de restricciones.
- **Problema del 8-Puzzle**
  - Se tiene una configuración inicial de casillas y se desean mover hasta que estén en orden.
  - La heurística puede ser el número de casillas mal colocadas; aunque una mejor es la que tiene la suma Manhattan del camino de la casilla del inicio al final ya que de esa manera se estiman mejor los pasos que tomará el problema.
  - Problema de camino. (Igual que el mapa de carreteras y TSP).
- **Problema del Mapa de Carreteras**
  - Dadas unas ciudades conectadas por carreteras, obtener el camino más corto entre una ciudad y otra.
  - La heurística se parece a la de 8-puzzle, se tiene en cuenta lo que ha recorrido más la estimación para llegar al final.
  - Se diferencia porque contempla el costo, porque es un problema de optimización, por eso se tiene que minimizar.
- **Problema del Viajante de Comercio**
  - Dado un conjunto de ciudades conectadas, obtener el camino más corto para recorrer todas las ciudades y retornar a la inicial.

si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

- Hay muchas heurísticas complejas para este problema, se debe contemplar lo que se tiene recorrido y estimar el camino de vuelta al inicio. Como es un problema tan complejo no se tiene una heurística general que funcione sobre él, las heurísticas simples no funcionan bien.
- Una heurística es el grafo óptimo de grado 2 se utiliza ya que se parece al TSP, (orden  $n^3$ )
- Se puede utilizar el árbol generador minimal, (orden  $n^2$ )
- **Problema de la Moneda Falsa**
  - *Se tienen 12 monedas, se sabe que hay una falsa porque, por ejemplo, pesa menos que el resto. Existe un instrumento para pesarla: una balanza, se quiere detectar con el menor número posibles de evaluaciones cual es la falsa, en este caso el máximo es de 3 evaluaciones.*
  - Problema de tipo descomponible, no se desea realizar una simulación, se desea un esquema genérico.
  - El estado se representa con el número de monedas.
  - Una heurística para este problema puede ser la "complejidad" de pesar una  $x$  cantidad de monedas entre sí.

### **Búsqueda Primero El Mejor:**

- Búsqueda con el Algoritmo A\***
- Método más conocido de búsqueda por el mejor nodo.
  - La idea es evitar explorar caminos que ya sabemos que no son interesantes por su costo.
  - La función de evaluación es  $f(n) = g(n) + h(n) \rightarrow A^*$ 
    - $g(n)$  es el costo actual.
    - $h(n)$  es el costo estimado del nodo al objetivo.
    - $f(n)$  el costo total estimado al objetivo obligado a pasar por el nodo.
  - El algoritmo posee una lista de nodos ABIERTOS, donde al principio está el nodo inicial. Existe otra lista, CERRADOS, que está vacío.
  - Comienza un bucle que se repite hasta encontrar solución o hasta que ABIERTOS está vacío.
    - Seleccionar el mejor nodo de ABIERTO.
    - Si es el nodo objetivo terminar.
    - En caso contrario, expandir ese nodo.
    - Para cada uno de sus sucesores:
      - Si está en ABIERTOS, insertarlo manteniendo la información del mejor parente.
      - Si está en CERRADOS, insertarlo manteniendo la información del mejor parente y actualizar la información de los descendientes.
      - En otro caso insertarlo como un nodo nuevo

### **Modelos más generales: Algoritmos de Agendas**

- Consiste en una generalización del algoritmo A\*, surge de manera totalmente independiente, siendo diseñado para un algoritmo llamado AM para teoría de números.
- Este algoritmo propone en vez de Estados, Tareas. El algoritmo realizaba una tarea.
  - Una tarea consiste en una descripción de la misma, una lista de justificaciones -- los padres del nodo-- y una valoración heurística.
  - Una vez que se realizaba una tarea (que sería como expandir un nodo), surgían otras tareas que se le proponían. Por lo tanto debe gestionar dichas tareas, de aquí surge la agenda.
  - La agenda es la lista de tareas, de donde se quiere seleccionar la tarea más importante. Esto es un equivalente a la heurística de decidir cual nodo es el mejor, por lo tanto la agenda es un equivalente al conjunto de ABIERTOS.
  - El conjunto de CERRADOS no existe pero puede pensarse como las tareas ya realizadas.
  - El algoritmo toma una tarea de la agenda, luego de realizarla obtiene otra tarea, la cual puede que ya existe en la agenda pero fue propuesta por un parente diferente o una justificación distinta, si es así añade la nueva justificación a la tarea y continúa.
- Formalmente, este algoritmo y el de A\* son muy similares; pero que no está ligado a encontrar un camino mínimo, se puede pensar que el A\* sería una particularización del Algoritmo de Agendas.
  - Esto quiere decir que si una tarea es generada por padres diferentes, en el Algoritmo de Agendas lo

que se hace es que se añade a las justificaciones, de que es posible por otra tarea llegar a esa, es un algoritmo más general. En el A\*, en cambio, se tiene un operador que discrimina que si un padre es mejor que otro, se selecciona que el nodo se quede con ese padre.

### Modelos con Poda

- Una de las primeras maneras en la que se extendió el algoritmo A\* original es que se le añadieron criterios de poda.
  - **Búsqueda Dirigida (Poda a priori)**
    - La diferencia es que restringe de qué manera se generan los descendientes, originalmente como este algoritmo guarda todos los nodos hijos ya sea en ABIERTOS o CERRADOS, si se tiene un problema en el que el factor de ramificación es muy elevado, entonces el espacio de búsqueda crece mucho.
    - Se restringe artificialmente este factor de ramificación por medio de la heurística  $h$  que utiliza, se generan los descendientes y se limita el número del mismo, el resto de los descendientes al ser podados, esto puede comprometer encontrar la solución.
  - **Algoritmo A\* con Memoria Acotada (Poda a posteriori)**
    - Se delimita la estructura de datos, por ejemplo ABIERTOS puede tener solamente un tamaño determinado fijo, una vez que se llena el espacio nodos generados luego deben comparar su función  $f(n)$  y si son mejores entran eliminando los peores nodos de ABIERTOS.
    - Es posteriori porque se genera el nodo y luego se ve si se poda luego de compararlo con el resto de nodos ya generados.
    - Es más ágil que el A\*: se puede sacrificar obtener la solución óptima pero funciona más rápido.

### Alternativas al A\*

- Modifican de manera más extensiva el A\* original
- Tienen la misma optimalidad que el A\*, pero en ciertos problemas funcionan mejor que el propio A\*.
  - **Descenso Iterativo A\* (IDA\*)**
    - Algoritmo popular y frecuentemente utilizado, se implementó por el autor Korf en 1985 para afrontar problemas que el A\* original no podía, por ejemplo el 8-Puzzle de valores elevados
    - La función heurística se mantiene de igual,  $f(n) = g(n) + h(n)$ , lo que varía es la implementación. Se utiliza una estrategia retroactiva branch and bound, que va a iterar no en profundidad sino en costo de la solución.
    - Se establece un modelo de cotas, se establece una cota inicial y en las siguientes iteraciones se empieza a variar estas cotas.
      - Una vez que inicia el algoritmo, el nodo raíz es comparado con la cota, como será igual a la cota podrá generar hijos ya que la cota es siempre el  $h$  del nodo raíz. Genera un hijo, si el nodo hijo no supera la cota, el algoritmo continúa profundizando por ese nodo recursivamente. Si se llega a un nodo que supera la cota, se para la recursión. Se regresa al nodo anterior y genera otro hijo por el que continúa explorando, todo esto hasta que regresa al nodo raíz y este genera otro hijo.
      - El algoritmo no se detiene si encuentra el objetivo en un nodo que está por encima de la cota porque no está garantizado que sea la manera óptima de llegar a él, se detendrá si consigue el objetivo y este tiene un coste menor que la cota actual.
      - Si el algoritmo no ha encontrado la solución para una cota  $\eta$ , entonces se incrementa la cota. La cota es calculada como el valor mínimo del coste  $f$  de los nodos que han superado la cota anterior.
      - Se repite el proceso utilizando la cota nueva hasta encontrar la solución.

### Procedimiento IDA\* (Estado-inicial Estado-final)

EXITO=False

$\eta = h(s)$

Mientras que EXITO=False

## Procedimiento IDA\* (Estado-inicial Estado-final)

EXITO=Falso

$$\eta = h(s)$$

Mientras que EXITO=Falso

    EXITO=Profundidad (Estado-inicial, $\eta$ )

$$\eta = \min_{i=1,n} \{f(i)\} = \min_{i=1,n} \{g(i) + h(i)\}$$

Solución=camino desde nodo del Estado-inicial  
al Estado-final por los punteros

### Profundidad (Estado-inicial, $\eta$ )

    Expande todos los nodos cuyo coste  $f(n)$  no excede  $\eta$

- ¿Por qué se utiliza la cota de esa manera? Es para mantener la condición que  $h(n) \leq h^*(n) \forall n$ , la admisibilidad. La cota se amplía lo mínimo posible para poder buscar la solución pero estando siempre por debajo de  $h^*(n)$ .
- Notar que en cada iteración se repiten nodos, ya que al volver a la raíz se empiezan a generar todos los nodos explorados de nuevo, es una desventaja pero también una ventaja es que se almacenan muy pocos nodos en memoria.
- **Características:**
  - **Completitud:** Es completo, bajo las mismas condiciones que el A\*, es decir en ciertos tipos de grafos y con coste positivo.
  - **Admisibilidad:** Es admisible, solo cuando  $h \leq h^*$  se encuentra la solución óptima
  - **Eficiencia:** Complejidad en tiempo exponencial, pero complejidad en espacio lineal en la profundidad del espacio de búsqueda. Aunque pareciera ser lo contrario, el número de reexpansiones es solo mayor en un pequeño factor que el número de expansiones de los algoritmos Primero El Mejor
  - Primer algoritmo en resolver óptimamente 100 casos generados aleatoriamente del 15-Puzzle, pero no es mejor como tal que el A\*, pero si puede funcionar mejor en ciertos casos.
- **Búsqueda Primero El Mejor Recursivo**
  - Almacena más nodos que el IDA\* pero muchos menos que el A\* y también, se diseñó para evitar reexpandir nodos, pero si es posible que pueda hacerlo.
  - La función heurística  $f(n) = g(n) + h(n)$  se mantiene, la estructura es similar al resto de los algoritmos.
  - La idea es que en cada nivel del árbol de exploración, se almacenan además de los nodos que se han explorado, también los nodos no explorados de ese mismo nivel. Se tiene un conjunto de ABIERTOS. Solo se admite mantener en memoria los nodos de un camino que se está explorando y los nodos hermanos de aquellos que están en el camino, no se admite expandir esos nodos hermanos en ese camino. Si sucede que un nodo hermano tiene un mejor  $f$  que el nodo expandido actualmente, primero, se almacena el valor  $f$  en el nodo expandido actual y se poda, luego, se continúa el algoritmo por el nodo hermano que ahora pasará a ser el expandido.
  - La función  $f$  de un nodo en principio es como se describe anteriormente, cuando se expande, la función  $f$  del nodo es el mínimo de las funciones  $f$  de sus hijos. Esto se propaga hasta llegar al nodo raíz.
- **Búsqueda por Franjas / Fringe Search**
  - Surge como una variante del IDA\*, se sabe que en cada iteración va regenerando nodos una y otra vez, en ciertos problemas esto es poco eficiente. La idea que se propone aquí es almacenar la frontera de exploración aunque ahora surge el problema de que no se sabe qué camino es el que se utilizó para llegar a esa frontera.

**QUIERES  
CONSEGUIR  
15€??**

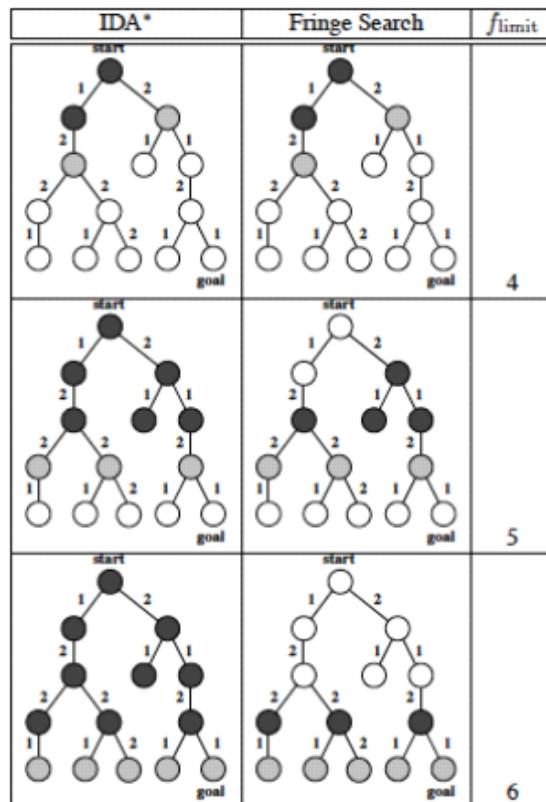
TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



- Se mantiene también una memoria de los nodos que se han visitado, se utiliza una tabla para almacenar los estados repetidos; esto hace que este algoritmo funcione mejor en aquellos problemas en los que se pueda acceder rápidamente a un nodo en la tabla hash, por ejemplo en planificación de caminos.
- Se mantienen también dos listas: la lista *now* y la lista *later*. El algoritmo sigue siendo retroactivo, comienza desde el nodo raíz que se encuentra en la lista *now* y expande los nodos hasta que llega a una cota, pero los nodos que superan la cota -que serían los nodo frontera- se almacenan en la lista *later*. En la siguiente recursión, la lista *later* pasa a ser la lista *now*, por lo tanto la búsqueda se retoma desde la frontera, sin tener que reexplorar los nodos anteriores. Estos nodos se guardan en la tabla hash, las listas *now* y *later* están siendo constantemente sobreescritas.
- Una vez que se llega al objetivo, se recupera el camino con la tabla hash que posee el nodo con mejor *g* por el que se ha pasado para llegar al objetivo.



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos



#### Algoritmos de Búsqueda para Planificación de Caminos con Otros Modelos de Representación:

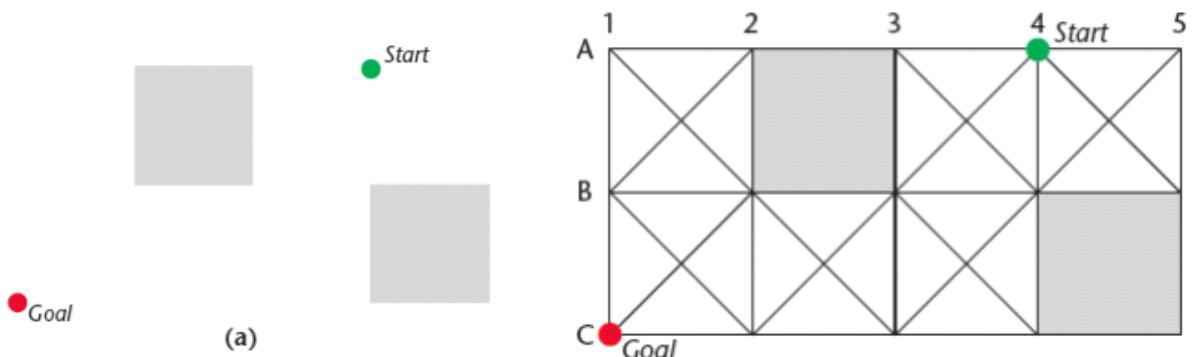
- La planificación de caminos tiene muchas representaciones del espacio, la representación que se lleva hasta ahora es que cada celda del mapa es cuadrículada y el agente ocupa la cuadrícula como tal.
- No todos los algoritmos funcionan con esta representación.

#### **Modelo de Celdas**

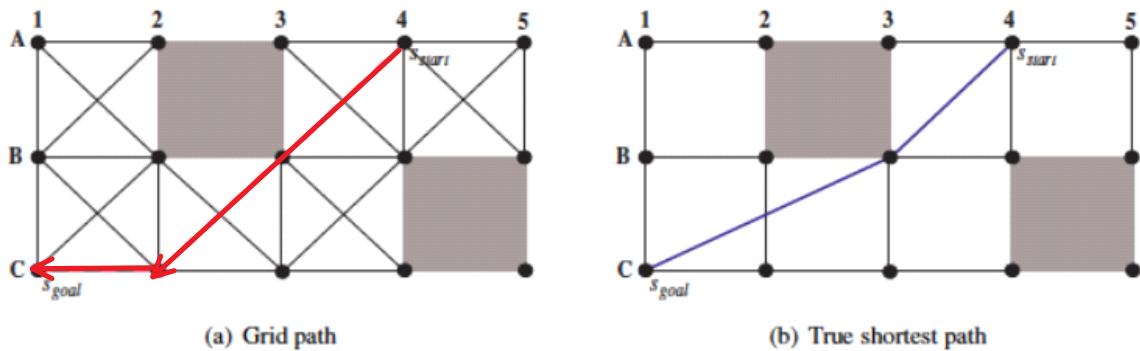
- El agente ahora se localiza en una de las esquinas de la celda en vez de ocuparla entera: es un punto en el espacio, pero los obstáculos si la pueden ocupar en su totalidad, es decir, si en la realidad un objeto ocupa parcialmente una celda se hace que esa celda entera esté ocupada.
- El agente puede pasar cerca de los "obstáculos" porque en realidad existe una sobreestimación del tamaño

de los obstáculos en la representación, por lo que esto no supone un problema.

- El agente puede moverse en 8 direcciones, las 4 originales y 4 diagonales.
- Es propia de planificación de caminos.



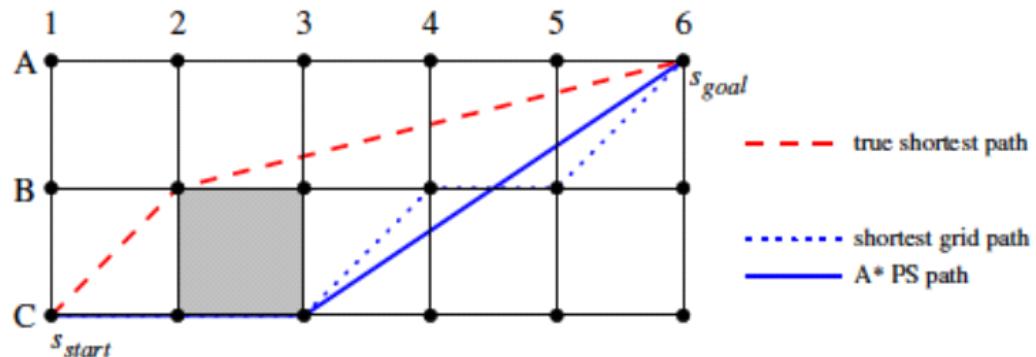
- El A\* consigue el camino óptimo de la representación que tiene, aunque este camino no es necesariamente el camino óptimo real.



- Como esto es un problema, se han diseñado variantes que toman en consideración esto.

#### A\* con Caminos Suavizados a Posteriori

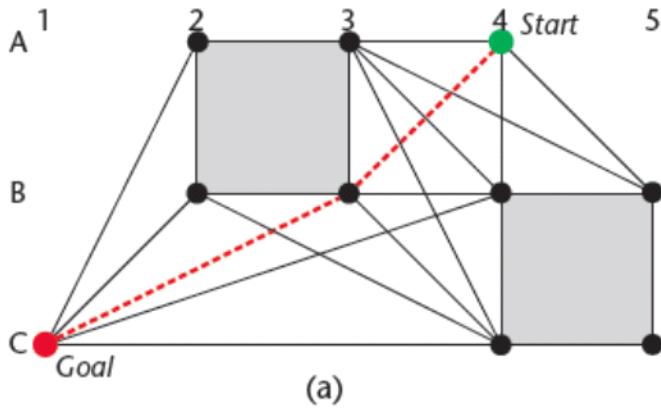
- Esta variante funciona exactamente igual que el A\*, pero una vez que se ha obtenido el camino se revisa intentando conectar directamente algún vértice del camino con el vértice objetivo para acortar la distancia del camino. Se dice que dos vértices son visibles si se puede trazar una línea recta entre ellos sin que pase por un obstáculo.



#### A\* con Grafos de Visibilidad

- Se modifica la representación de la cuadrícula, ahora partiendo desde los vértices de los obstáculos y los

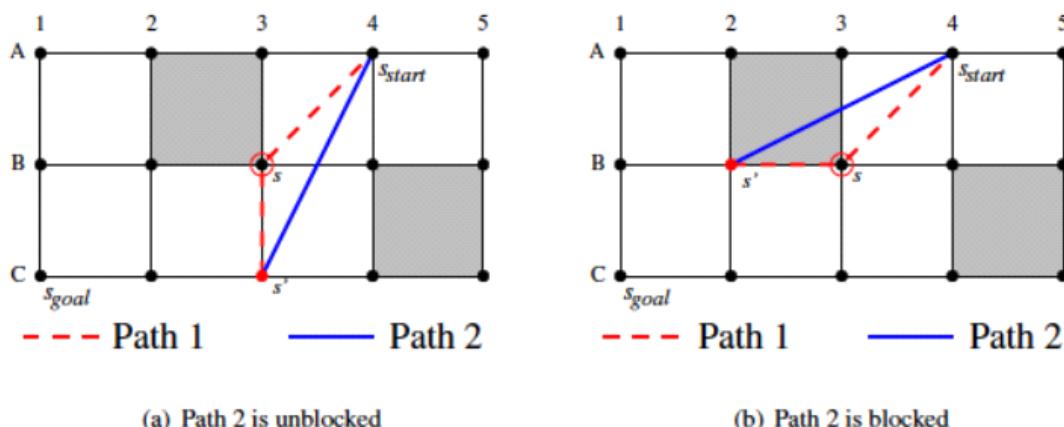
puntos de inicio y de final, se empiezan a trazar arcos entre los puntos que sean visibles de los obstáculos con los puntos de interés. La representación resultante permite que, si se realiza el A\* en el grafo de visibilidad, se obtendrá el camino óptimo real.



- El problema es que esta variante es mucho más lenta que la variante original pero obtendrá el camino más corto, mientras que la discretización con rejilla si bien da caminos más largos, es mucho más rápido en su ejecución.

## Algoritmo Theta\* Básico

- Es una modificación del A\* para que internamente logre hacer el suavizado de caminos por sí mismo.
  - Dado un camino, por ejemplo, Nodo Inicio → Nodo  $s$  → Nodo  $s'$ , se tendrían dos caminos:
    - El **camino 1** es el camino usual que haría el A\*, se considera el coste desde el nodo de inicio al nodo  $s$  más el coste del nodo  $s$  a su sucesor  $s'$ .
    - El **camino 2** es realizar un camino fuera de la rejilla, considera el coste desde el mejor nodo padre de  $s$  a  $s'$  en línea recta. Esto es lo que añade Theta\*.
  - De estas dos opciones, se elige el que tenga el mejor coste o bien algún camino sea no sea visible y entonces se descarta el camino 2.



## **Descomposición de Problemas:**

- Es búsqueda por el mejor nodo pero para problemas descomponibles.
  - La estructura subyacente para estos problemas es un grafo Y/O; se adapta el algoritmo A\* a estos problemas.

## Algoritmo Y/O\*

- **Grafo Y/O**

- Combinación de nodos Y y nodos O que indican el orden de consecución de tareas a realizar para alcanzar un objetivo.
- Para resolver un grafo Y/O cada nodo se resuelve de la siguiente manera:
  - Si es un nodo Y, se resuelven sus hijos. Se combina la solución y se soluciona el nodo, se devuelve la solución
  - Si es un nodo O, se tiene que resolver un solo hijo y ver si devuelve solución. Si no lo hace, resolver otro hijo. Cuando algún hijo esté resuelto, se combina la solución en el nodo y se devuelve.
  - Si es un nodo terminal, se resuelve el problema asociado y se devuelve.

- **Algoritmo**

- El algoritmo puede resolver una parte de los problemas en grafos Y/O pero no todos, ya que el algoritmo espera que cada nodo, cada subproblema, sea independiente del resto. Que se puedan resolver de forma separada, pero esto no pasa en todos los problemas.
- Por esto, no es tan utilizado para problemas complejos pues en estos, los nodos tienen dependencias entre sí.
- En este algoritmo no tiene sentido mantener las listas de ABIERTOS o CERRADOS, se debe mantener el grafo completo ya que los nodos por si solos no permiten analizar el problema entero.

- GRAFO contiene el nodo de inicio.
- Comienza un ciclo que se repite hasta que el nodo inicial quede resuelto o hasta que supere un valor MAXIMO
  - Trazar el mejor camino actual desde inicio
  - Seleccionar un nodo
  - Generar los sucesores del nodo e incluirlos de forma adecuada en el GRAFO
  - Propagar la información obtenida hacia arriba en el GRAFO

- Para que un nodo Y/O quede resuelto basta con que un hijo O esté resuelto o todos los hijos Y estén resueltos, esto es una etiqueta que se traspasa de hijo a padre. Una vez se etiquete que el nodo raíz está resuelto, finaliza el algoritmo.
- Puede que el problema sea irresoluble, en ese caso, se posee una constante llamada MAXIMO que almacena un coste que si lo supera la raíz, termina el algoritmo. Cuando un nodo se etiqueta como irresoluble, se le asigna  $h = \text{MAXIMO}$ , y se combina con el resto de nodos.
- La filosofía del A\* original es que en cada paso, se selecciona el mejor camino disponible. En un problema descomponible, ¿qué significa que sea el mejor camino? El mejor nodo.
- Ahora, aparte de tomar en cuenta la valoración de un nodo, también tiene que verse si es un nodo Y o nodo O, porque si es Y, la valoración de los otros nodos cuenta también.
- Respecto a la función heurística  $f = g + h$ 
  - $h$  está asociada a cada nodo.
  - $g$ , originalmente se asocia al costo de ir de la raíz al nodo. En este caso, como hay nodos que dependen de otros, no tiene sentido una  $g$  individual. Por lo tanto  $g$  es pertenece como tal a los nodos, ahora tiene que ver si es un arco Y o un arco O; esto se propaga de los nodos a su padres. Por defecto, un solo arco tiene coste unidad y un arco multiple el costo es número de problemas, este valor se añade al costo  $h$  y es lo que se propaga hacia la raíz. Los nodos terminales solo poseen un valor  $h$ , pero el resto si posee como tal el valor  $f$  conjunto de  $g + h$  dependiendo de los nodos hijos que tengan y que tipo de arco poseen

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



### Propiedades de los Métodos Heurísticos:

#### Estudio teórico del Algoritmo A\*

- Muchos métodos heurísticos no tienen garantías excepto aquellos de primero el mejor.
  - Para estudiar esto, se necesitan dos propiedades sobre el problema, no el algoritmo.
    1. El espacio de estados del problema debe poderse representar en un **grafo localmente finito**. Un grafo es localmente finito cuando el número de descendientes de un nodo es finito, pero no significa que el grafo en general sea finito.
    2. El coste  $C(N_i, N_j) > 0 \forall N_i \forall N_j \in suc(N_i)$ .
  - **Notación:**
    - Un nodo sucesor de otro nodo se representa como, dado  $N_i$  y  $N_j$  se dice que  $N_j$  es sucesor de  $N_i$  si  $N_j \in suc(N_i)$  y esta conexión tiene un coste  $C(N_i, N_j)$ .
    - Los nodos iniciales se representan como  $S_0, \dots, S_n$ , porque teóricamente puede haber más de uno pero se puede tener un nodo "virtual"  $S$  que se conecte a todos los nodos  $S_0, \dots, S_n$  con un coste 0 para tener siempre un único nodo inicial, sin importar el problema.
    - Los nodos objetivo se representan como  $\gamma_1, \dots, \gamma_m$ . El conjunto de todos se representan como  $\Gamma$  y este conjunto no es lo mismo que  $S$ , no se conectan. Es solo la agrupación de todos.
    - Como en A\* se explora por caminos, se representa un camino entre dos nodos  $N_i, N_k$  cualesquiera que no tienen que ser sucesores directos uno del otro como  $P_{N_i-N_k}$ .  $\mathbb{P}_{N_i-N_k}$  representa uno o más caminos entre los nodos. También se pueden tener los caminos entre un nodo a un conjunto de nodos.
    - $P_{N_i-N_j}^*$  indica el camino óptimo entre  $N_i$  y  $N_j$ . De igual manera se tiene  $\mathbb{P}_{N_i-N_k}^*$ .
    - $k(N_i, N_k)$  representa el coste óptimo del camino  $P^*$  entre  $N_i$  y  $N_k$ .
    - $g^*(N_i)$  es el coste del camino óptimo entre  $S$  y  $N_i$ . Equivalente a  $k(S, N_i)$ . Se diferencia de la  $g$  actual porque  $g^*$  es el mejor camino teórico de todos aunque el A\* no lo haya considerado, se puede pensar que A\* lo que intenta es estimar la  $g^*$  que desconoce.
    - $h^*(N_i)$  es el coste del camino óptimo  $P^*$  entre  $N_i$  y al mejor objetivo. Equivalente a  $\min(k(N_i, \gamma))$ ,  $\gamma \in \Gamma$ .
    - $C^*$  es el coste del camino óptimo entre  $S$  y el o los mejores objetivos, equivalente a  $h^*(S)$ . Es equivalente también a  $g^*(\Gamma^*)$ .
    - $\Gamma^* \subseteq \Gamma$ , es el subconjunto de los nodos objetivo a los que se puede llegar por un camino de coste  $C^*$ .
    - $f^*(N_i) = g^*(N_i) + h^*(N_i)$  es la función teórica del mejor camino que va de  $S$  a  $N_i$  sumado al mejor camino que va de  $N_i$  a  $\gamma$ .
      - $f^*(S) = h^*(S) = g^*(\gamma) = f^*(\gamma) = C^* \forall \gamma \in \Gamma$
      - El óptimo de  $h^*(S)$  es lo mismo que  $g^*(\gamma)$  si es el nodo objetivo óptimo y también coincide con el coste asociado.

#### Propiedades de $f^*$ :

- La propiedad  $f^*(N) = C^* \forall N \in P_{S-\Gamma}^*$ , o sea que la suma de  $h^*$  y  $g^*$  es el Coste óptimo si  $N$  forma parte de un camino óptimo del nodo inicio al objetivo.

$$\begin{aligned} \forall N \in P_{S-\Gamma}^* \\ \rightarrow \exists p \text{ tal que } g_p(N) + h_p(N) = C^* \\ \rightarrow g^*(N) + h^*(N) \leq g_p(N) + h_p(N) = C^* \end{aligned}$$

¿Qué pasa si  $g^*(N) + h^*(N) < C^*$ ?  $\rightarrow \exists p' \text{ tal que } g^*(N) = g_{p'}(N) \text{ y } h^*(N) = h_{p'}(N)$

No puede ser porque no puede existir un camino que sea mejor que el óptimo porque en ese caso ese mismo camino ya sería el óptimo,  $g^*(N) + h^*(N) < C^*$  es imposible.

$$\therefore f^*(N) = g^*(N) + h^*(N) = C^*$$

- La propiedad  $f^*(N) > C^* \forall N \notin P_{S-\Gamma}^*$ , o sea que el mejor camino de un nodo que no forma parte del camino óptimo no podrá ser igual o menor que  $C^*$ .

$\forall N \notin P_{S-\Gamma}^*$  suponer  $f^*(N) \leq C^*$   
 $\rightarrow g^*(N) + h^*(N) \leq C^*$   
 $\rightarrow \exists p' \text{ tal que } g_{p'}(N) + h_{p'}(N) \leq C^*$   
 $\rightarrow N \in P_{S-\Gamma}^*$  lo que es una contradicción, por lo tanto queda demostrado.

- El algoritmo A\* desea obtener un camino de costo óptimo entre  $S$  y  $\Gamma$ . Hace uso de  $f$ ,  $g$  y  $h$ .
  - $f$  es la estimación que hace el A\* de la  $f^*$ , porque si se pudiera obtener la  $f^*$  no habría necesidad realizar una búsqueda porque ya se tiene el mejor camino posible.
  - $g$  es la estimación que hace el A\* de la  $g^*$ , es el mejor camino de los visto hasta ahora.
    - Se tiene que  $g(S) = 0$
    - $g_p(N_i)$  es el coste acumulado de los arcos desde  $S$  hasta  $N_i$  por un camino cualquiera.
    - Propiedad:  $g_p(N_i) \geq g^*(N_i)$  esto es así porque  $g^*$  es el mejor de todos, por lo tanto  $g_p$  no puede ser mejor porque entonces sería ya  $g^*$ .
  - $h$  es una heurística definida que utiliza el A\*.
    - Se tiene que  $h(\gamma) = 0$
    - $h_p(N_i)$  es el coste acumulado desde  $N_i$  hasta  $\gamma$  por un camino cualquiera.
    - Propiedad:  $h_p(N_i) \geq h^*(N_i)$  esto es así porque  $h^*$  es el mejor de todos, por lo tanto  $h_p$  no puede ser mejor porque entonces sería ya  $h^*$ .

#### Compleitud: ¿A\* encuentra solución?

- Se dice que un algoritmo es completo si alcanza una solución cuando esta existe.
- Primero, se comienza suponiendo que el grafo es finito...
  - En cada paso del algoritmo se selecciona un nodo, y junto a ese nodo también se está seleccionando un camino entonces un nodo equivale a un camino. Entonces cada paso equivale a seleccionar un camino y se detiene una vez se llega a la solución.
  - El hecho de que el coste no pueda ser negativo evita que se produzcan ciclos, ya que si un nodo tiene un padre que es un ciclo, tendrá un coste peor a otro padre que no forma un ciclo. Por lo tanto, A\* no explora los ciclos.
  - Como se tiene que es un grafo finito, entonces la cantidad de caminos acíclicos es finito, entonces, A\* está garantizado que finaliza.
  - En un grafo finito, A\* es completo:  
 $\exists P_{S-\gamma}$  pero supongamos que A\* no es completo entonces ABIERTOS se quedará vacío y no encontrará solución, esto no se cumple ya que, en primer lugar,  $S$  entrará en ABIERTOS, y cuando salga, entrarán todos sus sucesores. Por lo tanto, entrará un sucesor que estará en el camino  $P$  solución y eso se repetirá hasta que salga  $\gamma$  y como encuentra solución se contradice con que ABIERTOS se quede vacío.
- Ahora, A\* es completo suponiendo que el grafo es localmente finito y el coste es siempre positivo.
  - Se supone nuevamente que  $\exists P_{S-\gamma}$  pero el algoritmo no la encuentra.
  - Nuevamente, siempre habrá un nodo de  $P_{S-\gamma}$  en ABIERTOS porque inicialmente entra  $S$  en ABIERTOS, luego entran los hijos de  $S$  y eventualmente entrará un hijo que sea parte de ese camino.
  - Ahora, si no encuentra solución pueden pasar dos cosas:
    - El algoritmo termina, esto es una contradicción análoga a la demostración anterior en un grafo finito.
    - El algoritmo no termina, esto quiere decir que está indefinidamente explorando nuevos caminos, pero se sabe que los grafos localmente finitos poseen una cantidad finita de caminos de longitud finita entonces si el algoritmo no termina es porque ya exploró esos caminos sin encontrar solución (que si existe) y está explorando caminos infinitos pero en ese caso pasaría en  $f(n) = g(n) + h(n)$  que  $g(n) = \infty$  lo cual haría que  $f(n) = \infty$  y esto no puede suceder ya que si hay nodos en ABIERTOS que tienen valoración finita e infinita, se seleccionarán siempre los finitos primero por el criterio de selección de A\*, por lo tanto se llega a una contradicción porque eventualmente llegaría a la solución por un camino finito.
  - Por lo tanto, queda demostrado que A\* es completo si el coste es siempre positivo y se exploran grafos localmente finitos.

#### Admisibilidad: ¿A\* encuentra la solución óptima?

- Para que una heurística  $h$  sea admisible se tiene que puede serlo si y solo si  $h(n) \leq h^*(n) \forall n$ .

- **Lema 1:** Para cualquier nodo  $n$  que no esté en CERRADOS, y para cualquier camino óptimo entre  $S$  y  $n$  llamado  $P$  siempre se puede seleccionar un nodo  $n'$  que está en  $P$  y está en ABIERTOS en el que  $g(n') = g^*(n')$ .
  - Partiendo de  $n$  se selecciona un camino óptimo  $P = (n_0 = S, n_1, \dots, n_k = n)$  ahora queremos encontrar  $n'$ .
  - Si inicialmente  $S$  está en ABIERTOS, entonces  $S$  está en  $P$  y en ABIERTOS, y  $g(S) = g^*(S) = 0 \rightarrow n' = S$  luego se demuestra el Lema 1.
  - Si  $S$  no está en ABIERTOS, primero, se define un conjunto  $\Delta$  que contiene nodos  $m$  que están en CERRADOS y en  $P$  para los que  $g(m) = g^*(m)$ .  $\Delta \neq \emptyset$  ya que  $S \in \Delta$ , si de  $\Delta$  se escoge el nodo de mayor índice (de la numeración anterior,  $n_0 = S, n_1, \dots$ ) y se llama  $n^*$  que es un nodo que está en CERRADOS, está en  $P$ , tiene  $g(n^*) = g^*(n^*)$  y tiene el índice más alto. Se tiene un  $n'$  es el sucesor de  $n^*$  en  $P$  pero  $n'$  estaría en ABIERTOS porque si estuviera en CERRADOS, ya sería  $n^*$ .
  - Se tiene que  $g(n') \leq g(n^*) + C(n^*, n')$  porque  $n'$  sería igual al costo  $g$  del padre más el costo o bien podría tener otro parente mejor. Como  $n^*$  está en  $\Delta$  se sabe que  $g(n^*) = g^*(n^*)$ , entonces  $g(n^*) + C(n^*, n') = g^*(n^*) + C(n^*, n') = g^*(n')$  y si se conectan los extremos, se tiene que  $g(n') = g^*(n')$  y por lo tanto se demuestra el Lema 1.
- **Lema 2:** Si  $h$  es admisible y se supone que  $A^*$  no ha terminado entonces para cualquier camino óptimo  $P$  entre  $S$  y  $\gamma$  siempre se puede seleccionar un nodo  $n'$  en  $P$  y en ABIERTOS tal que la  $f(n') \leq C^*$ .
  - Si  $A^*$  no ha terminado, entonces  $\gamma$  no puede estar en CERRADOS entonces tomando el Lema 1, siempre se puede encontrar un camino  $P$  de  $S$  a  $\gamma$  donde  $g(n') = g^*(n')$
  - $f(n') = g(n') + h(n') = g^*(n') + h(n')$  y si  $h$  es admisible entonces  $h(n') \leq h^*(n')$ , entonces  $g^*(n') + h(n') \leq g^*(n') + h^*(n')$  y queda entonces que  $f(n') \leq f^*(n') = C^*$ . Luego, queda demostrado el Lema 2.
  - Lo que dice este lema es que si  $A^*$  no ha terminado siempre habrá nodos por debajo de la cantidad óptima si  $h$  es admisible.
- **Teorema:** Si  $h$  es admisible entonces  $A^*$  es admisible.
  - Si se supone que  $h$  es admisible pero  $A^*$  no es admisible entonces  $A^*$  no encuentra la solución óptima, o sea que es  $> C^*$ . El nodo que da esa solución se le llama  $t \in \Gamma$ .
  - Si se sitúa en el momento que  $t$  está en ABIERTOS y es el mejor nodo en ese instante. Por lo que dice el Lema 2, en ABIERTOS también estará un nodo  $n'$  tal que  $f(n') \leq C^*$ . En ese caso se tendrá que  $f(n') \leq C^* < f(t)$  por lo que no puede pasar que  $t$  sea el mejor nodo de ABIERTOS, se ha encontrado una contradicción.
- Por lo tanto, queda demostrado que  $A^*$  encuentra la solución óptima solamente si la heurística es admisible y que el grafo sea localmente finito y que el coste sea siempre positivo.

#### **Dominancia: Cuando hay dos heurísticas para un problema, ¿cuál es la mejor?**

- Dada una versión del  $A^*$   $A_1$  con una heurística  $h_1$  domina a otra versión  $A_2$  con  $h_2$  si cada nodo expandido por  $A_1$  también es expandido por  $A_2$ ,  $A_1$  domina a  $A_2$  porque si cada nodo que expande  $A_1$  también lo expande  $A_2$ , entonces  $A_2$  expande más nodos luego  $A_1$  llega a la solución expandiendo menos nodos y por lo tanto es más eficiente.
- En general, no hay un estudio teórico que garantice que  $h_1$  sea mejor o peor que  $h_2$  excepto en el caso que ambas sean admisibles y que  $h_1 > h_2$  para todo nodo, se dice que  $h_1$  está más informada que  $h_2$ . La dominancia está relacionada con una heurística más informada y por lo tanto, mejor.
- Esto se puede entender como que, si bien  $h_1$  y  $h_2$  están por debajo del óptimo  $h^*$ , si se tiene que  $h_1 > h_2$  eso quiere decir que  $h_1$  está más cerca de  $h^*$  y por lo tanto expande menos nodos (si  $h_1 = h^*$  sería ir al camino óptimo directamente), por lo tanto, si está más cerca es que la estima mejor. Si está más alejada y más cercana a cero expande más nodos y empieza a asemejarse al costo uniforme.
- ¿Qué ocurre con  $h_1 \geq h_2$ ?
  - Depende del criterio que tome cada versión del algoritmo en el desempate, no se puede concluir claramente una dominancia pero los algoritmos tendrán un comportamiento similar pero no igual, no tienen porque expandir los mismos nodos por el criterio de desempate que aplican.
  - Aunque el número de nodos expandidos por  $h_1$  será más pequeño que  $h_2$ .
- ¿Cuál es el interés de utilizar  $f(n) = g(n) + \omega \cdot h(n)$ ?
  - Se tiene un peso  $\omega$ 
    - Cuando vale 1, es el  $A^*$  normal.
    - Cuando vale 0, es el algoritmo de costo uniforme.

- Generalmente se puede usar 2 o 3 porque potencian la heurística y aceleran mucho el algoritmo.
- Muchas heurísticas no están bien informadas, por lo que están alejadas de  $h^*$ , al multiplicar por un valor positivo lo que se hace es hacerla más informada, aunque esto tiene el riesgo de que si el valor es muy alto se supere  $h^*$  y entonces la heurística deje de ser admisible y no se garantice la solución óptima.
- Son estudios experimentales, el valor de  $\omega$  se va variando y visualizando la reducción de los nodos expandidos para los valores.

### ¿Es posible encontrar heurísticas que hagan más eficiente el A\*?

- Heurísticas Monótonas
  - El algoritmo A\* puede tecnicamente utilizar cualquier función  $h$ , aunque esto no garantiza que funcione bien.
  - Hay una propiedad de  $h^*$  que se va a necesitar que tenga  $h$ .
    - Si se tiene  $S, n$  y  $\gamma$  se puede ver que el coste óptimo de  $S$  a  $\gamma$  será más pequeño o igual que el coste de pasar de  $S$  a  $n$  y de  $n$  a  $\gamma$ . Es una desigualdad triangular  $k(S, \gamma) \leq k(S, n) + k(n, \gamma)$  y como se habla de  $h^*$ ,  $h^*(S) = k(S, \gamma)$  y entonces  $h^*(S) \leq k(S, n) + h^*(n)$ .
    - Se puede generalizar como  $h^*(n) \leq k(n, n') + h^*(n') \forall n, n'$
    - **Consistencia:** Lo que pasa con  $h$  es que no tiene porqué cumplir esta propiedad, pero  $h$  será consistente si cumple que  $h(n) \leq k(n, n') + h(n') \forall n, n'$ .
    - **Monotonía:**  $h$  es monótona si  $h(n) \leq C(n, n') + h(n') \forall n \forall n' \in suc(n)$ .
    - Si se tiene una función consistente, también es monótona, dado que si se verifica para cualquier par de nodos, se verifica para el nodo y su sucesor. Y si una función es monótona, también es consistente. Son propiedades equivalentes.
  - **Teorema:** Toda heurística consistente o monótona es admisible y por lo tanto garantiza la solución óptima.
    - Si  $h$  es consistente entonces  $h(n) \leq k(n, n') + h(n')$ , si  $n' = \gamma \in \Gamma^*$  entonces  $h(n) \leq k(n, \gamma) + h(\gamma)$ ,  $h(\gamma) = 0$ , entonces  $k(n, \gamma) = h^*(n)$  y por lo tanto  $h(n) \leq h^*(n)$ .
  - **Teorema:** Un algoritmo A\* guiado por una heurística monótona alcanza el camino óptimo a todos los nodos expandidos, es decir,  $g(n) = g^*(n) \forall n \in CERRADOS$ .
    - Si suponemos un caso en que se tiene un nodo  $n$  que es el mejor de ABIERTOS, va a entrar en CERRADOS pero no verifica el teorema, entonces  $g^*(n) < g(n)$ .
    - Se tiene un camino  $P_{S-n}$  y hay un nodo  $n'$  que es antecesor de  $n$  en  $P$  y en ABIERTOS que verifica que  $g(n') = g^*(n')$ , luego se tiene que  $f(n') = g(n') + h(n') = g^*(n') + h(n') \leq g^*(n') + k(n', n) + h(n) = g^*(n) + h(n)$ .
    - Por lo tanto,  $f(n') < f(n)$  por lo que no se podría tener que  $n$  es el mejor de ABIERTOS porque  $n'$  es mejor.
  - Esto es importante porque esto evita que se tengan que revisar nodos en CERRADOS, lo que es mucho más costoso que revisarlos en ABIERTOS.

### Heurísticas a través de Modelos Simplificados:

#### ¿Cómo se generan las heurísticas?

- Se tiene un problema donde se desea obtener la  $h$ , entonces se simplifica el problema, ya sea quitar restricciones o simplificar condiciones, que sea básicamente lo mismo pero que pueda resolverse de manera algorítmica, al resolverlo se sabe cuantos pasos hay desde ese estado a la solución y se conoce la  $h^*$  y por lo tanto la  $h$  es la  $h^*$  de un problema simplificado.

#### ¿Una heurística obtenida de esta manera tienen buenas propiedades?

- **Teorema:** Toda heurística obtenida por un modelo simplificado es consistente, por lo tanto, monótona y por lo tanto admisible.
  - $h_s^*$  del problema simplificado cumple con  $h_s^*(n) \leq C_s(n, n') + h_s^*(n')$ , como es la  $h_s^*$  cumple con la condición de monotonía del problema simplificado.
  - El costo de  $C_s(n, n') \leq C(n, n')$
  - Como la  $h$  del problema real es la  $h_s^*$  del simplificado, se tendrá que  $h$  es también monótona y admisible,  $h(n) \leq C(n, n') + h(n')$ .
- Por lo tanto, toda heuristica obtenida de esta manera está garantizada que es monótona y admisible.
- Esta técnica funciona pero si se puede resolver el problema simplificado de manera exacta y algorítmica. Otro

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
**CRUSH DE APUNTES**

ANTES DE QUE  
LOS QUEME



problema es que las heurísticas que se deducen son muy pocas informadas y por lo tanto no son tan buenas.

#### Proceso Sistemático

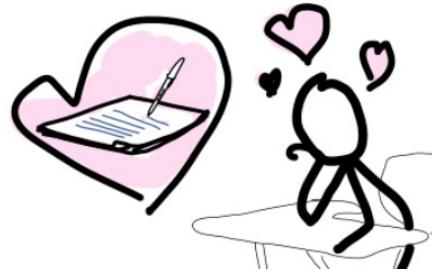
- Es una de las formas de como llegar a un modelo simplificado con una metodología sistemática.
- Para obtener versiones simplificadas de un problema, es necesario tener una descripción formal del problema, por ejemplo, en lógica de predicados.
- Una vez se tienen los predicados y las acciones, esto es lo que realmente determina que acciones pueden realizarse y cuales no.
- Las versiones simplificadas no deben cambiar la esencia del problema, pero deben quitarse las condiciones que no modifiquen la idea general, al menos la que lo hagan en menor grado.
- Esta técnica es muy utilizada en IA, específicamente, muchos programas que resuelven problemas de satisfacción de restricciones generan heurísticas de esta manera.



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



## Tema 2 - Problemas de Satisfacción de Restricciones

Profesor Antonio González Muñoz, Curso 20-21



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

- Es un tipo particular de problemas que van a tener asociado
  - Métodos de búsqueda particulares
  - Heurísticas de propósito general que sirven en general para estos problemas.
- Un problema de este tipo se denomina CSP y está definido por X, D y C.
  - X es el conjunto de las variables  $\{X_1, X_2, \dots, X_n\}$
  - D es el conjunto de dominios  $\{D_1, D_2, \dots, D_n\}$ , uno por cada variable.
  - C es el conjunto de restricciones  $\{C_1, C_2, \dots, C_m\}$  que especifican las combinaciones permitidas de valores.
- Cada dominio  $D_i$  representa un conjunto  $\{V_1, \dots, V_k\}$  de valores permitidos para  $X_i$ , y cada restricción  $C_i = <\text{lista de variables y relación entre ellas}>$

### Satisfacción de Restricciones

- El estado de un problema es una asignación de valores a algunas variables o a todas.
- Para tener un lenguaje que indique que clase de solución se está buscando, se tiene...
  - Asignaciones **consistentes**: Una asignación que no viola ninguna restricción, es decir, que los valores que se les asigna a las variables no están rompiendo ninguna restricción que se tienen en el problema.
  - Asignaciones **completas**: Una asignación en la que aparecen todas las variables.
  - Asignación **parciales**: Una asignación en la que aparecen algunas de las variables.
- Lo que se busca para un problema de esta clase es una asignación completa y consistente.
- Algunos CSPs requieren una solución que maximice o minimice una función objetivo
  - **COP**: Constraint Optimization Problems, problemas de optimización de restricciones. Un subgrupo de los CSPs más complejos por la restricción de optimizar.
- Las restricciones entre las variables se pueden representar como un grafo de restricciones, en donde los nodos son las variables y los arcos entre variables son las restricciones del problema que tengan que ver los dos nodos.

### Tipos de CSPs

- Según el tipo de variables
  - Variables Discretas (Enteros, Símbolos, Cadenas)
    - Dominios Finitos
      - Tienen n variables, el tamaño del dominio es  $d \rightarrow O(d^n)$  asignaciones completas. Es la más simple y aún así, tiene un crecimiento exponencial.
      - Ejemplos: Diseño de circuitos, demostración de teoremas, verificación y validación de software.
    - Dominios Infinitos
      - Utilizan enteros y cadenas con un dominio infinito, es necesario tener un lenguaje de restricciones.
      - Ejemplos: Asignación de tareas y máquinas.
  - Variables Continuas (Números reales)
    - Tienen restricciones lineales
    - Por ejemplo, tiempo de inicio y de fin de las observaciones del Telescopio Espacial Hubble.
- Según su tipo de restricciones
  - Restricciones Unarias, involucran una sola variable.
    - $SA \neq Green$
  - Restricciones Binarias que involucran parejas de variables.
    - $SA \neq WA$
  - Restricciones de Orden Superior que involucran 3 o más variables, restricciones globales.
    - Restricciones Criptogramáticas de Columnas o Puzzles Criptogramáticos
  - Preferencias o Restricciones suaves
    - Rojo es mejor que verde.

### ¿Cómo resolver estos problemas?

- **Formulación Incremental:** Se realiza una asignación paso a paso, se asigna una variable a un valor, luego otra, hasta llegar al final.
- **Formulación Completa de Estados**

#### Formulación Incremental

- Se comienza con un **estado inicial** de una asignación vacía,  $\{\}$ . Ninguna variable tiene valores.
- Se tiene una **función sucesor** en la que un valor se puede asignar a cualquier variable no asignada con la condición de que no suponga ningún conflicto con las variables que han sido anteriormente asignadas. Es un proceso más tradicional de búsqueda.
- El **test del objetivo** es que la asignación actual es completa, dado que mientras se esté ejecutando se tendrá siempre asignaciones parciales.
- El **costo del camino** es constante para cada paso en principio.
- Para  $n$  variables, la solución se encuentra a una profundidad de  $n$ . Puede tener un tamaño gigantesco pero tiene una frontera.

#### Formulación Completa de Estados

- Dado que el camino no interesa para la solución, se pueden manejar estados con asignaciones completas y sobre ella se resuelve con métodos de escalada para problemas que incumplan restricciones.
- Hace uso de métodos de Búsqueda Local, ya que estos métodos funcionan mejor con soluciones completas. A diferencia de la incremental, estas no tienen garantías pero son más eficientes de ejecutar.

#### Complejidad

- Estos problemas son extremadamente complejos, obtener soluciones es en general NP-Completo y la obtención de soluciones óptimas es NP-Duro.
- Se requiere por lo tanto gran eficiencia de los métodos de búsqueda.

#### Búsqueda en la Formulación Incremental

- El uso de Búsqueda en Anchura hace que se genere un árbol con  $n! \times d^n$  hojas (mucho).
- Existe una propiedad crucial que es común en los CSPs: La conmutatividad, es decir, el orden de aplicación de cualquier conjunto de acciones no tiene ningún efecto sobre el resultado. El orden en el que se asignan las variables no importa, porque lo que importa es el resultado final.
- La técnica más eficiente para esta clases de problemas es la Búsqueda en Profundidad Retroactiva (Backtracking), en general las búsquedas retroactivas funcionan bien ya que tienen profundidad limitada pero no expanden tantos nodos.

#### Mejoras del Modelo

- Al backtracking se le puede añadir una heurística para mejorar la funcionalidad del algoritmo
  - ¿Qué variable debe asignarse después, y en qué orden deberían intentarse los valores?
  - ¿Cuáles son las implicaciones de las variables actuales para el resto de variables no asignadas?
  - Cuando un camino falla, ¿puede la búsqueda evitar repetir este fracaso en los siguientes nodos?
- Los problemas CSP disponen de heurísticas genéricas independientes del problema.

#### Variables y Ordenamiento de Valores

- **Mínimos Valores Restantes (MVR) o Variable Más Restringida**
  - En principio no hay ningún criterio en la selección de la variable, por lo que el orden en que se tienen las variables en el problema afecta mucho su rendimiento.
  - Esta heurística toma de primero las variables que poseen más restricciones frente a las que poseen menos, tiene la suerte de ser una heurística sencilla, fácil de calcular e implementar. Funciona mejor porque hay menos alternativas en las que se va a ir a un camino peor y tener que regresar.
  - En los problemas en los que se tienen la misma cantidad de restricciones, se añade más información para mitigar esto de manera que los dominios de las variables se van a ir reduciendo conforme se vayan asignando más variables.
- **Grado Heurístico**
  - Selecciona la variable, entre las que no estén asignadas, que esté implicada en el mayor número de restricciones. La MVR es una mejor heurística, pero esta funciona junto a la MVR para resolver

empates de restricciones.

- **Valor Menos Restringido**

- Una vez seleccionada una variable debe decidirse un orden para examinar sus valores.
- Heurística del **valor** menos restringido: Se prefiere el valor que menos restringe los valores del resto de variables no asignadas.
- Se toman decisiones que eviten en lo posible que restrinja las otras variables porque de este modo se mejoran las posibilidades de llegar a la solución sin realizar una vuelta atrás.

### **Propagación de la Información a través de las Restricciones**

Para mejorar el backtracking, luego de la heurística, se considera la propagación de información por las restricciones.

- **Comprobación hacia delante (Forward-Checking)**

- Es un algoritmo adicional que se incluye en el proceso de búsqueda.
- Modifica los dominios para mantener solo los valores legales para variables no asignadas.
  - Si selecciono y asingo X, compruebo variables relacionadas con X en el grafo de restricciones y elimino los valores no legales.
  - Cuando se tiene un dominio vacío se realiza la vuelta atrás.
- Aun así, este método no logra descubrir todas las inconsistencias que se tienen.

- **Propagación de Restricciones (Constraint Propagation)**

- Es más costoso computacionalmente que la Comprobación hacia Delante pero lo que se ahorra en expandir menos nodos mejora en general el tiempo de ejecución.
- La idea no es comparar una variable asignada con el resto, sino hacer comprobaciones de cada par de variables en el grafo de restricciones después de haber hecho una asignación, para ello se define:
  - Arco Consistencia: X es arco consistente con Y si y solo si
    - Para todo valor de  $x_i$  hay alguno  $y_i$  permitido
    - La arco consistencia es equivalente a la consistencia sobre una condición binaria que representa un arco.
    - Que X sea arco consistente con Y no implica que Y sea arco consistente con X.
- **Grafo de restricciones completo arco consistente:** Si todas sus variables son arco consistentes, es decir, si los dominios de las variables satisfacen todas las restricciones.
- El algoritmo para mantener la arco consistencia, se llama Mantenimiento de Arco Consistencia (**MAC**).
  - Cada vez que se asigna una variable, se realiza sobre todas las restantes variables no asignadas un chequeo de arco consistencia y se fuerza a que sean arco consistentes.
    - Se fuerza eliminando valores de los dominios
    - Si algún dominio queda vacío, se debe hacer vuelta atrás.
  - Se mantiene una cola de arcos dirigidos para todas las variables no asignadas.
  - Se recorre la cola y comprueba/fuerza arco consistencia en cada arco.
- La propagación de restricciones basada en arco-consistencia detecta inconsistencias antes que Forward Checking, puede ejecutarse como preprocesamiento antes de empezar la búsqueda.

- **Vuelta atrás inteligente**

- El Backtracking normal hace uso de la vuelta atrás cronológica, se vuelve al punto de decisión más reciente. Pero puede suceder que al entrar en un callejón sin salida, la decisión que llevó a esto no está un salto hacia atrás, sino mucho más atrás, por lo que hacer varias vueltas atrás es un gasto.
- El método de vuelta atrás inteligente se llama Salto Atrás.
  - Para una variable X, su conjunto conflicto es el conjunto de variables previamente asignadas Y, tales que el valor asignado a Y evita uno de los valores de X por una restricción entre ambas.
  - El método de Salto Atrás retrocede a la variable más reciente del conjunto conflicto aunque esta esté mucho más atrás en el camino actual.

### **Búsqueda Local para Problemas de Satisfacción de Restricciones:**

- Se pueden utilizar Algoritmos de Búsqueda Local utilizando la formulación completa de estados, ya que para esta clase de problemas funcionan muy bien.
- Se utiliza la heurística de mínimos conflictos, se selecciona el valor que cause el menor número de conflictos con otras variables. Es una heurística genérica y no depende del problema.
- La técnica de mínimos conflictos es sorprendentemente eficaz para muchos CSPs, en particular cuando se parte de un estado inicial razonable.

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES

ANTES DE QUE  
LOS QUEME



- Otra ventaja es que se puede utilizar aunque las condiciones del problema cambien mientras se está en ejecución, se puede adaptar.



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
**CRUSH DE APUNTES**  
ANTES DE QUE  
LOS QUEME



## Tema 3 - Sistemas de Planificación en IA

Profesor Antonio González Muñoz, Curso 20-21

### **Razonamiento sobre Acciones y Planificación Clásica:**

- ¿Qué es planificar?
  - Es razonar sobre acciones. El eje central de la planificación son las acciones, conocer como son las acciones, representarlas y razonar sobre las acciones.
- ¿Qué es un plan?
  - Es una secuencia de acciones que resuelve un problema en particular.
- Búsqueda y Planificación
  - El planteamiento de Planificación se asemeja mucho al de búsqueda.
  - La diferencia entre ambos es los tipos de problemas en los que se aplica, y lo que requiere. Cuando se trata de búsqueda, esta va analizando como ordenar las acciones para conseguir un fin determinado y eso lo requiere la planificación, pero la planificación va un paso más allá porque como trata de problemas reales necesita un modelo de representación del conocimiento, algo que la búsqueda no lo utiliza.

### **Problemas de Planificación:**

- Complejidad del mundo real:
  - Frente a un dominio que es real, la búsqueda por sí sola no puede resolver ciertos problemas más complejos; se deben utilizar los modelos basados en la lógica para capturar esa complejidad del mundo real.
- Otros problemas:
  - **Problema del Marco (Frame Problem)**
    - Problema más importante de la planificación, que solamente se puede mitigar pero no evitar del todo.
    - Es un problema esencialmente asociado a la representación del conocimiento porque lo que se plantea es como se puede representar una acción para poder razonar eficientemente como es el mundo antes y después de realizar una acción, ya que las acciones cambian el mundo.
    - Puede parecer un problema sencillo, pero el problema es que primero se tiene un estado antes y un estado después de la acción en el que habrá cambiado algo por la acción; aquí la esencia de la acción es que debe indicar como se modifica ese estado.
    - La dificultad es dual; cuando se describe la acción es normal que se describa que cambia pero el problema surge en poder describir que es lo que no cambia luego de la acción y luego como se razona con eso que no cambia. Por eso se llama del "marco", es del contexto.
    - Una solución sería describir todo, lo que cambia y que cambia pero esto es mucho más complejo y se vuelve inviable.
  - **Efectos Dependientes del Contexto**
    - Este problema tiene que ver, obviamente, con los efectos: los cambios que se tienen que describir
    - Por ejemplo, describir el tiempo es complicado ya que el tiempo que se toma realizar una acción, ir de la casa a la universidad, varía por la hora y el día. Puede que a veces tome más y a veces menos tiempo, entonces, es un efecto que depende del contexto.
    - Se tienen que considerar las acciones raras que pueden que sucedan muy pocas veces para que el razonador sepa cómo lidiar con ellas si llegan a suceder.
  - **Problema de la Cualificación**
    - Tiene que ver con la representación de una acción, es sobre qué condiciones se puede aplicar una acción
    - Todas las acciones tienen precondiciones, los requisitos por los cuales se puede aplicar una acción.
    - El problema es ¿cómo definir las precondiciones?, porque a veces no es sencillo saber cuáles podrían ser y cómo representarlo.
    - **Ejemplo:** Se tiene un coche que se conduce solo, ¿qué precondiciones necesita para que ponga



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

en marcha el coche el agente? Se pueden añadir las usuales, que el motor esté encendido, que esté en neutral, que haya gasolina. Pero, puede suceder que en un día se cumplen todas las precondiciones y no enciende el coche porque alguien le metió un plátano al tubo de escape, entonces, ¿también debería añadirse otra precondición para verificar que no hayan plátanos?

- El problema es cuando existen precondiciones que son muy infrecuentes y que es absurdo comprobarlas siempre, entonces, pueden existir situaciones en la que las acciones no puedan realizarse aunque las precondiciones originales se cumplan.

### Modelos previos a la Planificación

Son modelos anteriores a la Planificación como tal pero que pretendían resolver la misma clase de problemas; no se consideran como Planificación porque solo eran capaces de resolver problemas muy simples pero aportaron mucha información para llegar a la Planificación.

- **El cálculo de situaciones (de Green)**

- Utiliza la lógica de predicados para modelar el problema en vez de los modelos icónicos, ya que se tenía un modelo deductivo muy potente que era independiente del problema; se intentó adaptar esto para planificar sobre un problema.
- Se intenta "demostrar" llegar al estado que deseo por medio de las acciones que serían ahora axiomas, pero entendiendo que los cambios que pasan en el mundo tienen que representarse.
- Para evitar el problema de la monotonía, o sea, que el valor de verdad de los predicados cambie cuando cambia el estado del problema. Se introduce una variable a los predicados de estado, por lo tanto se puede tener que los predicados sean ciertos en un estado pero no en otro.
- Las acciones se representan como una función *hacer*(acción, estado) la cual genera un nuevo estado dependiendo de la acción, los efectos se expresan como fbfs, ya sean efectos positivos o negativos.
- El problema es que surge el **problema del marco**, porque los efectos son locales y por eso no se sabe de lo que no cambia. Esto se intentó mitigar con Axiomas de Marco que son acciones para lo que no cambia
- Sigue el problema de la cualificación, de las precondiciones, si se intentan añadir también predicados para todo lo que puede suceder se vuelve inviable la inferencia lógica.
- A pesar de esto, este sistema puede funcionar en problemas reducidos; ya que era muy costosa computacionalmente y que no realmente se resuelve el problema del marco. Es teóricamente correcto.
- Lo más **importante** que aportó a la planificación es que la **representación por lógica de predicados** es muy potente.

- **Reducción de diferencias**

- Diseñado en paralelo al sistema anterior, se quería diseñar un sistema que resolviera los problemas de manera similar a como lo hace el Humano. Era el General Problem Solver y la Reducción de Diferencias era como el GPS funcionaba.
- Se desea reducir las diferencias entre el estado actual y el estado objetivo, si no hay diferencias entonces el problema está resuelto.
- Es un proceso que se va descomponiendo el problema, se busca la acción más relevante para el problema, se aplica y se repite el proceso con los trozos restantes del problema.
- La idea es encontrar una diferencia entre el objeto inicial y el objeto final. Si no hay diferencias, el problema está resuelto. El objeto es el estado del problema.
- Si hay diferencias, se considera la más importante (las diferencias están ordenadas), se encuentra un operador para reducir esa diferencia y ahora se comparan las precondiciones del operador con el estado actual, si no hay diferencias es que se puede aplicar y se aplica. Si hay diferencias, se tratará de reducirlas.
- Se repite el proceso tomando como objeto inicial el objeto producido por la aplicación del operador y el objeto final.
- Uno de los problemas de este sistema es que requiere de mucha infraestructura para realizar el análisis de diferencias, era muy difícil representar problemas más complejos con listas.
- También no se tenían métodos de búsquedas eficientes, se utilizaban métodos retroactivos. Aun así era capaz de resolver problemas simples.

- **Lecciones aprendidas:** Es muy útil utilizar lógica de predicados para representar esta clase de problemas y también es bueno utilizar heurísticas para dirigir la búsqueda y tener un esquema más sencillo de diferencias.

### Planificación Clásica

- Surge como un descendiente directo de los dos sistemas anteriores. En primer lugar la PC adopta una serie de suposiciones:
  - El sistema tiene un número finito de estados o situaciones.
  - El sistema es perfectamente observable, es decir, se tiene conocimiento completo del estado del sistema.
  - El sistema es determinista, es decir, la aplicación de una acción a un estado conduce siempre a un mismo estado, no hay incertidumbre.
  - El sistema es estático, es decir, el sistema permanece en el mismo estado hasta que se aplique una acción
  - Los objetivos se conocen antes de empezar la planificación y no cambian.
  - Un plan solución de un problema de planificación es una secuencia de acciones finita y linealmente ordenada.
  - No se contempla el razonamiento temporal y numérico, por lo que la calidad del plan se determina por el número de acciones del mismo.
  - La tarea de planificación consiste en construir un plan completo que satisface el objetivo antes de la ejecución de cualquier parte del mismo.
- Como se puede ver, tiene unas cuantas restricciones. Existen otros tipos de planificación que relajan estas suposiciones.
- Un problema en Planificación Clásica se formula a través de los siguientes elementos:
  - Un conjunto de fórmulas atómicas, denominadas hechos o literales, que representan la información relevante del problema.
  - Un conjunto de operadores definidos en el dominio del problema.
  - Un conjunto inicial de hechos que forman la situación inicial del problema.
  - Un conjunto final de hechos que deben formar parte de la situación final del problema.
- El primer sistema de planificación se llamó STRIPS y condicionó la mayoría de trabajos de planificación desde comienzos de los años 70. Es la base de los planificadores actuales pero ahora está en desuso.
- STRIPS se utilizó para planificar las acciones del robot Shakey.

### STRIPS: Standford Research Institute Problem Solver (1971)

- Representación del conocimiento, heredada del Cálculo de Situaciones.
  - Describe **estados** con lógica de predicados, pero sabiendo los problemas que daba anteriormente, se restringe para evitarlos, para ello se utiliza la Hipótesis del Mundo Cerrado.
    - Todo predicado que no está descrito en un estado se supone que es falso, esto simplifica muchísimo más describir los estados, esto aumenta la eficiencia.
    - Esto impide que se maneje incertidumbre.
    - Los estados se describen como una **conjunción** de literales básicos, o sea, son predicados completamente instanciados. El trabajar con solamente con conjunciones, para saber si una acción se puede aplicar en un estado es mucho más eficiente y rápido hacerlo con conjunciones.
    - **No se admiten disyunciones**, porque esto implica incertidumbre. Tampoco se admiten implicaciones (axiomas).

## Estado

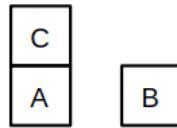


Predicados relevantes:

LIBRE(X)  
SOBRE(X,Y)  
SOBREMESA(X)  
MANOVARIA



\_\_\_\_\_

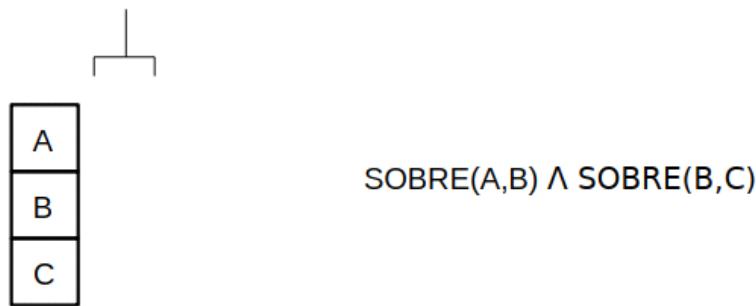


-----  
 SOBRE(X,Y)  
 SOBREMESA(X)  
 MANOVACIA  
 COGIDO(X)

$\text{LIBRE}(B) \wedge \text{SOBRE}(C,A) \wedge \text{SOBREMESA}(A) \wedge \text{LIBRE}(C) \wedge \text{MANOVACIA} \wedge \text{SOBREMESA}(B)$

- **Describe objetivos**

- El objetivo es una orden, lo que se desea que sea cierto. No necesariamente corresponde a un estado.
- En el ejemplo coincide con un estado pero no tiene por qué ser siempre así.
- El objetivo se puede formar por conjunciones de literales básicos (que no tienen variables) y también órdenes que tengan variables pero deben estar cuantificadas existencialmente ( $\exists$ ), por ejemplo:
  - Que exista un bloque que esté sobre A,  $\exists x \text{ sobre}(A,x)$
  - Que exista un bloque que esté sobre A y B,  $\exists x \text{ sobre}(A,x) \wedge \text{sobre}(B,x)$ .
- No se pueden resolver cuantificadores universales ( $\forall$ )



- **Describe operaciones**

- Componente más importante de STRIPS, los estados y objetivos meramente restringen la lógica de predicados.
- Las reglas tienen un nombre y unas variables que utiliza.
- No utiliza como tal la lógica de predicados como lo hace el sistema de Green, la utiliza parcialmente.
- Se describe en base a tres estructuras:
  - Fórmula de **precondición**, que es una fórmula de lógica totalmente.
  - Lista de **supresión**, lista de predicados y con variables libres sin cuantificación. Son aquellos predicados que se eliminan de la conjunción.
  - Lista de **adición**, lista de predicados y con variables libres sin cuantificación. Son aquellos predicados que se añaden a la conjunción.
- Se podrá aplicar una acción sobre un estado siempre que la precondición sea verdad en ese estado.
  - "Si se tiene una acción a que se quiere aplicar al estado S, se podrá hacer si y solo si la precondición de la acción es cierto en ese estado"

**Nombre de la acción y variables usadas**  
**Fórmula de Precondición**  
**Lista de supresión**  
**Lista de Adición**

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



## Lista de Adición

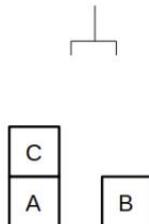
$$(a \in \text{ACTIONS}(s)) \Leftrightarrow s \models \text{PRECOND}(a)$$

$$\text{RESULT}(s, a) = (s - \text{DEL}(a)) \cup \text{ADD}(a)$$

- Para demostrar que una fórmula parte de otra fórmula, se utilizan mecanismos deductivos, pero como los estados y la precondición son una conjunción de literales el mecanismo de deducción es muy eficiente. Se transforma en un esquema simple de emparejamiento del estado con la precondición.
- ¿Cómo se obtiene de una acción y un estado, otro estado nuevo? Para eso se tienen las listas de supresión y de adición.

## Modelo de regla tipo STRIPS

- Coger(x)
- FP:  $\exists x$  tal que Sobremesa(x)  $\wedge$  Libre(x)  $\wedge$  Manovacia
- LA: COGIDO(x)
- LS: SOBREMESA(x), LIBRE(x), MANOVACIA



LIBRE(B)  $\wedge$  SOBRE(C,A)  $\wedge$  SOBREMESA(A)  $\wedge$  LIBRE(C)  $\wedge$  MANOVACIA  $\wedge$  SOBREMESA(B)

LIBRE(B)  $\wedge$  SOBRE(C,A)  $\wedge$  SOBREMESA(A)  $\wedge$  LIBRE(C)  $\wedge$  MANOVACIA  $\wedge$  SOBREMESA(B)

SOBRE(C,A)  $\wedge$  SOBREMESA(A)  $\wedge$  LIBRE(C)  $\wedge$  COGIDO(B)

si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

**WUOLAH**

- STRIPS está hecho de esta manera para poder mitigar el problema del marco, con las listas de supresión y adición y supone que el resto de las cosas que no están en ninguna de esas dos listas no se ve afectado. No hay necesidad de deducir las cosas que no cambian en las acciones.
- Esta es una de las aportaciones más importantes de STRIPS, hace que el razonamiento sea mucho más eficiente ya que no usa como tal la lógica de predicados sino acciones que quitan y ponen cosas en el estado.
- STRIPS está muy limitado y por ello había necesidad de extenderlo.

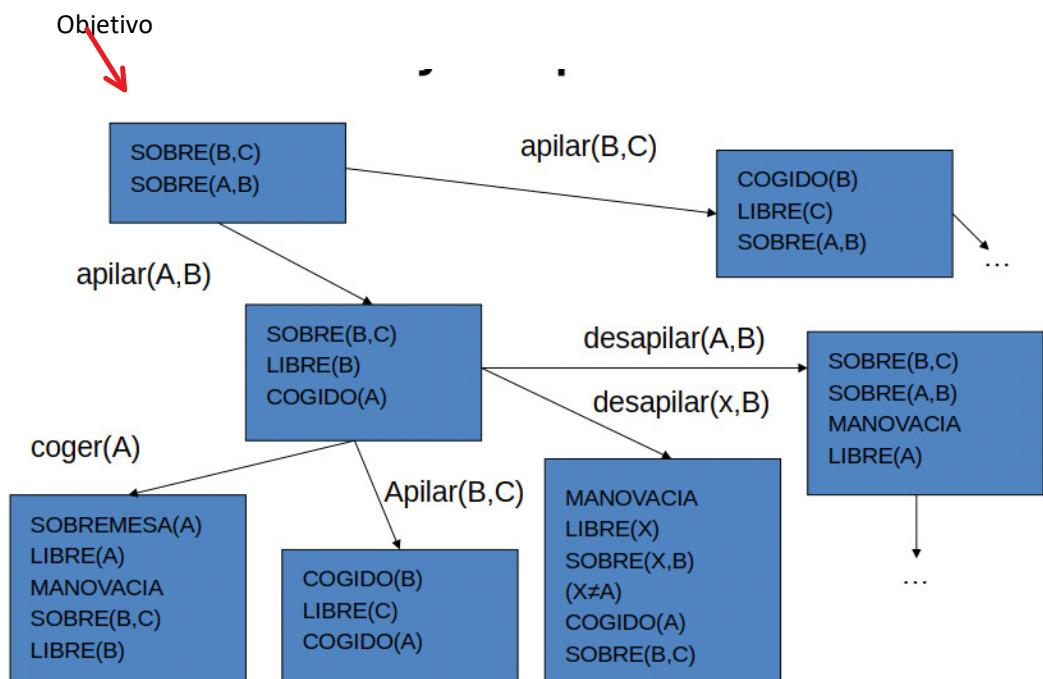
### Lenguaje de Planificación PDDL

- El lenguaje que utiliza STRIPS es otra de las aportaciones más importantes, ya que ha influido en todos los planificadores que se han diseñado a partir de él. El PDDL contiene el propio lenguaje de STRIPS con ADL y extensiones.
- Es un intento de estandarizar los lenguajes para describir los problemas y dominios de planificación, desarrollado para permitir competiciones entre planificadores.
- Una definición en PDDL consiste en:
  - Una definición del dominio.
  - Una definición del problema.
- Requerimientos:

- :strips, :equality, :typing, :adl (Para usar disyunciones, cuantificadores en precondiciones y objetivos, efectos condicionales, etc)
- Otras características:
  - Acciones con duración.
  - Expresiones y variables numéricas.
  - Métricas del problema.
  - Ventanas temporales.
  - Restricciones duras y blandas sobre el plan.

### Planificación como búsqueda en un espacio de estados

- STRIPS usa un modelo de planificación particular, aunque existían dos maneras de realizar esta búsqueda: Por Progresión y Regresión, como algoritmos A\* o similares, ¿por qué no se utilizaron?
  - Existen algoritmos de **progresión** válidos pero no se utilizó por el problema de la complejidad del mundo, al factor de ramificación de un estado puede ser muy muy alto, cuando se desarrolló STRIPS no se había encontrado manera de paliar esto: no se logró encontrar una **heurística potente**, por lo que no se utiliza la **progresión**.
  - Por **regresión**, el factor de ramificación no crece tanto como la progresión, lo cual es muy útil. STRIPS se plantea utilizar regresión, pero no es directo ya que los objetivos no son necesariamente estados entonces, ¿cómo se transforma un objetivo a un estado al que retroceder?
    - Se desea tener un estado donde se verifique el objetivo que se tiene, entonces, lo que se busca es un estado donde si se aplica un operador se obtiene el objetivo o lo que es lo mismo: se aplica el operador a la inversa; ya que esto genera un estado donde el operador original es aplicable y si se aplica se satisface al objetivo.
    - ¿Cómo se construye? Dado el objetivo y un literal del mismo, si se quiere conseguir ese literal se debe buscar un operador que en su lista de adición lo contenga o algo que con una sustitución lo genere.
    - Para los antecesores se le añade al objetivo la fórmula de precondición del estado actual.
    - Si  $Q_u$  es un literal de la lista de adición del operador particularizado la regresión es  $V$ .
    - Si está en la lista de supresión del operador, la regresión es  $F$ , cuando se tiene una conjunción con Falso, es imposible.
    - En otro caso, la regresión es  $Q_u$ .



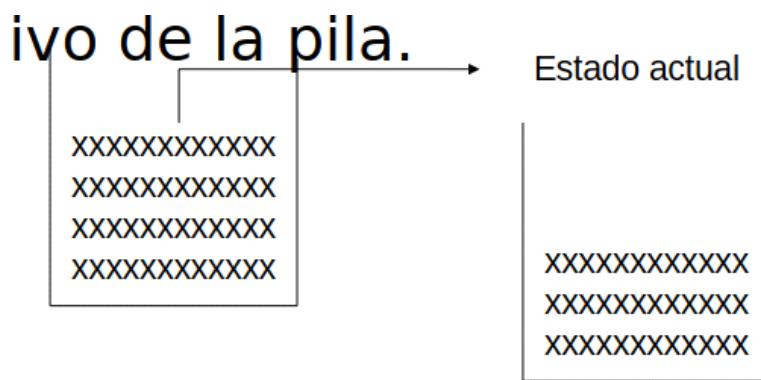
- De nuevo, STRIPS no utiliza la regresión como tal, ya que aunque se pensaba que iba a generar menos

ramificación, se han generado una cantidad considerable y además los estados que se obtienen son bastante más complejos. Se están analizando todos los órdenes posibles para resolver los literales: se decide descomponer el problema para que no intervenga el orden.

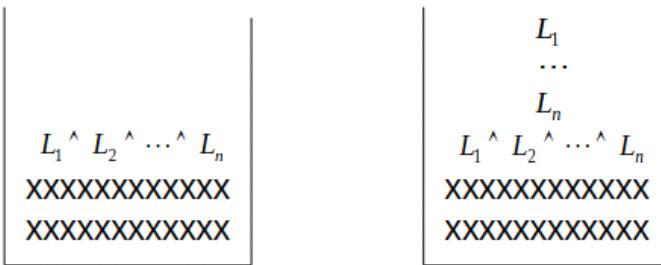
- STRIPS maneja un modelo por **regresión basado en descomposición**.
- Ni los métodos hacia delante ni hacia atrás no podrían ser eficientes sin una buena heurística generalista; posteriormente si se han podido encontrar haciendo uso de la técnica de modelos simplificados.
  - Basadas en ignorar precondiciones o ignorar la lista de supresión.

### Cómo funciona STRIPS

- Regresión por descomposición, como ya se mencionó. Cuando hay varios literales se descompone.
- Resolución de problemas, heredada de la Reducción de Diferencias y el General Problem Solver
  - Incorporan operadores al plan que sean relevantes aunque no se puedan aplicar en el estado actual, lo que le permitirá razonar hacia delante o atrás.
- Utiliza la representación simbólica del conocimiento sobre el dominio del problema: STRIPS no trabajará directamente con el estado y las acciones, se tienen dos estructuras...
  - Utiliza una pila de objetivos: que queda por resolver, que ya se ha resuelto, que acciones se han resuelto.
  - El estado actual del problema.
- Es una especie de metaestado, donde se tiene una pila junto con un estado. La pila se utiliza para dar una descripción de cómo se está resolviendo el problema en cada momento y con esto se realizará la búsqueda. Estos son los nodos del proceso de búsqueda.
- Se realizará un proceso de búsqueda con una serie de operaciones, estas operaciones no son acciones como tal, sino son acciones que trabajan con la pila y el estado actual del problema.
- El nodo inicial contiene en la pila de objetivos el objetivo completo del problema, y el estado actual es estado inicial del problema.
- **Operadores:** (Transforman la pila y/o el estado actual)
  - **Emparejar:**
    - Si el objetivo (literales simples o compuestos) de la parte superior de la pila empareja con el estado actual, se suprime ese objetivo de la pila, o sea que es cierto en ese estado.
    - Puede suceder que sea válido con varias sustituciones diferentes, se anota la sustitución asociada a la pila, entonces recordará la sustitución aplicada.
    - Este operador **solo toca la pila de objetivos**.



- **Descomponer:**
  - Si el objetivo en la parte superior de la pila es compuesto que no es cierto (si lo fuera se aplicaría Emparejar), entonces añadir los literales componentes en la parte superior de la pila.
  - **Importante:** El literal compuesto ***no*** se suprime, porque es un dispositivo de seguridad que da STRIPS para resolver las posibles interacciones que no era capaz de resolver un grafo Y/O.
  - Pueden descomponerse en cualquier orden.



- **Resolver:**

- Cuando el objetivo que se encuentra en la parte superior de la pila es un único literal no resuelto, entonces se busca un operador cuya lista de adición contenga un literal que empareje con él.
- Dado el literal  $L$ , se aplica un criterio de buscar una regla que añada ese literal, que lo tenga en su lista de adición, se elimina  $L$  y se reemplaza por el operador  $O_u$  y se pone su fórmula de precondition  $FP_u$  encima, esto se hereda del GPS. Notar que se introducen ahora operadores a la pila.



- **Aplicar:**

- Cuando el término de la parte superior de la pila es un operador, entonces se suprime de la pila, se aplica sobre el estado actual modificándolo y se anota en el plan. Se ha hecho razonamiento hacia delante.
- **Modifica la pila de objetivos y el estado** actual.
- El proceso se repite hasta que la pila queda vacía, el estado debe ser concordante con el objetivo.

#### Proceso de búsqueda asociado

- Aunque no pareciera que se realiza una búsqueda, se debe de utilizar puesto que los operadores anteriores introducen bifurcaciones en las configuraciones de los nodos, por ejemplo, si un objetivo se empareja con 3 sustituciones diferentes, se generan 3 nodos hijos de ese estado, donde en cada nodo se intentará probar con un tipo de sustitución. Esto sucede también con Descomponer (orden de los literales) o Resolver (que existen varios operadores que resuelven un literal).
- Aquí es necesario tener un algoritmo de búsqueda, como un A\* o de Anchura. Originalmente se utilizó un Algoritmo en Profundidad Retroactivo con una heurística; el problema de la heurística es que no se tenía una que fuera muy potente o generalista.
  - No se pudieron obtener heurísticas buenas para la ordenación de literales; la heurística más relevante y que era genérica que se logró sería la selección de la mejor acción a aplicar.
  - La heurística para determinar el mejor operador es el número de precondiciones ciertas en el estado actual.

#### Interacción entre Subobjetivos

- Puede que cuando se realice una descomposición, puede que la resolución de un literal haga que otro literal que anteriormente era válido en el nodo ahora sea falso; resolviendo un objetivo cambia la resolución del otro.
- Por esto es que se mantiene el literal compuesto, si sucede una interacción, el algoritmo vuelve a

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



descomponerlo en el nuevo estado hasta que se logre tener uno donde se cumpla el literal compuesto. Esto no está garantizado, no es completo.

- Las interacciones que son suaves son aquellas que con el orden de las acciones se puede evitar.
- Las interacciones fuertes son aquellas que suceden sin importar el orden en el que se elijan las acciones.
- STRIPS tiene dificultad con las interacciones entre los subobjetivos, se llama la anomalía de **Sussman** y que el planificador no produce planes óptimos. El problema es que STRIPS descompone los problemas secuencialmente, se necesita intercalar las resoluciones de los subobjetivos para dar la optimalidad.

#### Planificación como búsqueda en un espacio de estados: HSP, FF:

- Existen otra clase de planificadores que lograron resolver el problema haciendo una búsqueda directamente en el espacio de estados.

##### **HSP**

- **HSP** (Heuristic Search Planner) fue el primero desarrollado que logró obtener heurísticas independientes del dominio, la idea es utilizar modelos simplificados; se trata de aplicar operadores e ignorar la lista de supresión; utiliza una estimación para el cálculo de la longitud del plan de la solución simplificada.
- Utilizó como algoritmo de búsqueda el algoritmo de escalada por máxima pendiente, donde los empates se resuelven de manera estocástica, una segunda versión utiliza el A\* con peso.
  - Se define el peso de f en 0 para todos los literales f del estado inicial y al resto se le asigna valor infinito.
  - Se aplican las acciones y se actualizan los pesos.
  - Para cada acción con precondición  $pre(O)$  que añade al literal f se tiene que
    - $peso(f) = \min(peso(f), peso(pre(O)) + 1)$
  - Se está realizando una especie de estimación de cuantos operadores se están requiriendo para obtener un literal concreto.
  - Cuando se tiene un conjunto de literales, se suman los pesos.
  - El proceso se repite hasta que no cambien los pesos en dos iteraciones sucesivas.
  - La heurística queda como  $h_{HSP}(S) = peso(G) = \sum_{g \in G} peso(g)$  donde G es el conjunto de objetivos.
- Como se obtiene por un modelo simplificado, se pensaría que la heurística es admisible y óptima pero no lo es, porque está basada en un modelo simplificado pero la sumatoria del final es la que evita la admisibilidad, porque está suponiendo la independencia del peso en cada uno de los literales, porque no toma en cuenta que ciertos pasos para verificar un literal podrían también ayudar a verificar otro: los toma como independientes, no bastaría sumarlos.
- Aun así es una heurística muy buena.

##### **FF**

- **FF** (Fast-Foward) está basado en **HSP** pero añade tres elementos nuevos:
  - Usa una nueva heurística basada en la idea del planificador Graphplan, basado en grafos. Se utiliza una versión simplificada. En las capas se van añadiendo los literales resultantes de realizar ciertas acciones, incluida la acción de no hacer nada, la ejecución finaliza cuando en la última capa se alcanzan todos los objetivos.
    - Una vez que llega al destino se extrae el plan situándose en la última capa y para cada objetivo se hace lo siguiente en la capa  $i$ .
      - Si el objetivo está presente en la capa  $i - 1$ , entonces se inserta como objetivo para alcanzar en la capa  $i - 1$ .
      - En otro caso, se selecciona una acción de la capa  $i - 1$  que añade el objetivo e insertamos cada precondición de la acción como objetivo de la capa  $i - 1$ .
      - Cuando ya se ha trabajado con todos los objetivos de la capa  $i$ , se continúa con los objetivos de la capa  $i - 1$ .
      - Se para al llegar a la primera capa.
    - El plan simplificado es una secuencia  $\{O_0, O_1, \dots, O_{m-1}\}$  en donde en cada  $O_i$  es el conjunto de acciones seleccionadas en cada capa.
    - La longitud del plan es  $h_{FF}(S) = \sum_{i=0, \dots, m-1} |O_i|$ . Es una mejor estimación de HSP.
  - Nuevo método de búsqueda: Un método de escalada forzado.
    - Este método de escalada evalúa todos los sucesores, se cambia irrevocablemente al mejor de



si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

**WUOLAH**

- todos si mejora al actual, si no es el caso, se realiza una búsqueda en anchura con el objetivo de buscar un nodo mejor que el de partida.
  - La búsqueda continúa desde ese nodo.
  - Guarda la traza de los nodos seleccionados para generar el plan.
- Un proceso para ordenar descendientes.
  - Considera en el proceso de búsqueda un conjunto de acciones útiles
  - Para un estado  $S$ , el conjunto  $H(S)$  de acciones útiles se definen como
    - $H(S) = \{o | pre(o) \subseteq S, add(o) \cap G_1 \neq \emptyset\}$  donde  $G_1$  representa el conjunto de objetivos marcados en la siguiente capa.
    - Es el conjunto de acciones que se pueden aplicar cuando las precondiciones son ciertas en el estado y que añaden alguno de los literales objetivo.
- Estos planificadores anteriores realizan la planificación como la búsqueda en un espacio de estados, pero no ha sido la única vía.

### **Planificación como búsqueda en un espacio de planes:**

- La filosofía que tienen es que se tiene un espacio donde se tienen todos los planes posibles que podrían trazarse para una situación dada y uno se mueve en este espacio de planes, pasando de un plan a otro, se realiza una búsqueda hasta encontrar un plan que encaja con las condiciones del problema. Es una búsqueda por unas características muy específicas.
- Se obtiene la **Planificación por Orden Parcial (POP)**, que también surge inspirado en STRIPS. Retoma la idea de descomposición de STRIPS.
- Aún así, se tienen diferencias: No se toman decisiones hasta que no se sepa como tomarlas, a esto se le denomina la Hipótesis del Menor Compromiso, el problema de STRIPS y de usar una pila es que estaba forzado a ordenar las acciones y ejecutarlas ahí mismo aún si no supiera como hacerla, esto trae muchos problemas.
- Hipótesis del Menor Compromiso:** Es necesario ocuparse de las decisiones que actualmente interesen, dejando cualquier otra decisión para más tarde.
- Se utiliza una estructura de grafo, pero no de búsqueda, sino para representar la solución con la misma idea de la pila de STRIPS: cambiar una estructura ordenada por una parcialmente ordenada para representar esa pila.

#### **Definición de plan**

- El concepto de plan es el concepto base de POP.
- Tiene varias componentes
  - Un conjunto de nodos (operadores)
  - Un conjunto de relaciones de orden:  $s_i < s_j$  significa que  $s_i$  como acción debe producirse en algún momento antes que  $s_j$ . Se tenía como la pila en STRIPS.
  - Un conjunto de restricciones de variables del tipo  $x = A$ .
  - Un conjunto de vínculos causales:  $s_i \rightarrow^c s_j$  que significa que  $s_i$  aporta el literal  $c$  a  $s_j$ .
    - Estructura nueva e importante, se originan en los problemas de interacción. Esta estructura de datos que permite proteger la validez de los nodos.

#### **Plan Inicial**

- Se incluyen dos operadores (nodos) ficticios INICIO y FIN ya que solamente trata con planes no posee estados ni objetivos como tal por lo que se necesitan operadores para indicar el inicio y el final de un plan.
  - INICIO no tiene precondiciones y tiene de efecto el estado inicial.
  - FIN tiene como precondiciones el objetivo y no tiene efectos.
- Este plan inicial tiene solamente estos dos nodos, no hay ninguna clase de restricciones o vínculos, se tiene que el orden es INICIO < FIN.
- Siempre se parte de este plan.

INICIO

INICIO

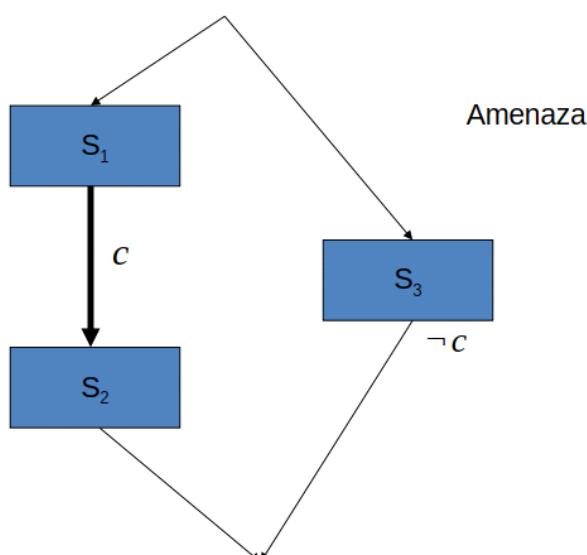
WUOLAH



- La idea es que se va moviendo de un plan a otro por refinamiento de las acciones, añadiendo acciones, se realiza buscando alguna acción que añadir al plan hasta que queda resuelto. Se ordenan las acciones lo que tiene explicitadas el orden, el resto se deja en paralelo, STRIPS en cambio habría obtenido diferentes planes secuenciales para poderlo representar.

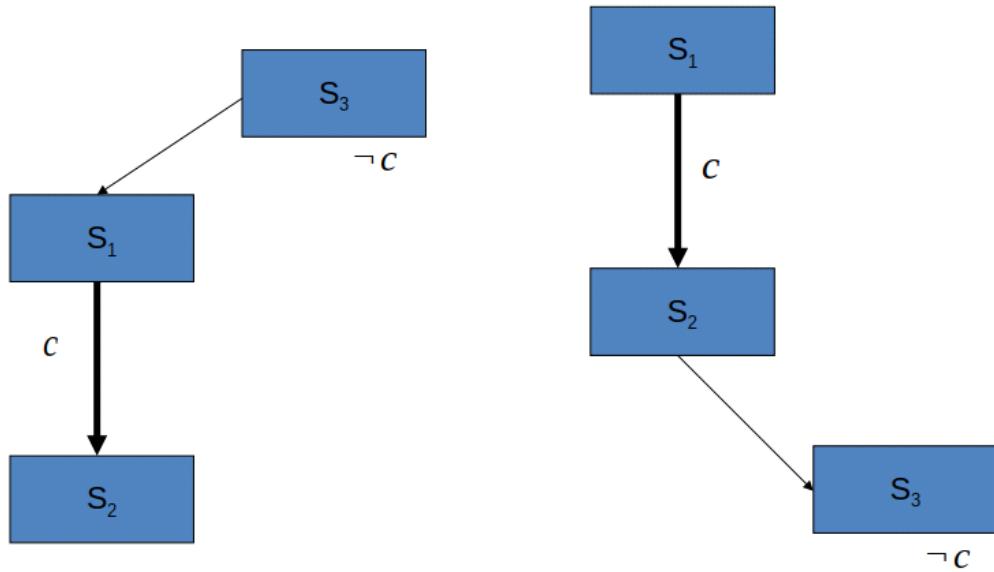
#### Plan Completo y Consistente

- El algoritmo parte del plan inicial y lo refina hasta que se tiene un plan completo y consistente.
- Plan Completo**
  - Cada precondición de cada operador se debe de satisfacer mediante otro operador, tiene que ver con los vínculos causales.
  - $s_i$  logra una precondición  $c$  de  $s_j$  si
    - $s_i < s_j$  y  $c \in EFECTOS(s_i)$
    - $\neg \exists s_k$  con  $\neg c \in EFECTOS(s_k)$  y  $s_i < s_k < s_j$  en alguna linealización del plan.
    - No puede haber una acción  $s_k$  que elimine  $c$  entre  $s_i$  y  $s_j$ , porque entonces  $c$  ya no sería cierta cuando  $s_j$  la requiera, este es el mecanismo de protección.
- Plan Consistente**
  - No hay contradicciones en las restricciones de orden ni de variables.
    - $x = A$  y  $x = B$  sería inconsistente.
    - $s_i < s_j, s_j < s_k, s_k < s_i$  sería inconsistente.
- Amenazas:** Cuando se viola un vínculo causal se denomina de esta manera



- ¿Cómo se resuelven? Dos métodos.
  - Ascenso:** se traslada la acción que produce la amenaza antes del vínculo causal.
  - Degradoación:** se traslada la acción que lo produce después del vínculo causal.

- Se prueba una u otra para ver cual funciona, puede que no se pueda reparar la amenaza y el algoritmo hace una vuelta atrás.



Algoritmo POP(<A,O,L>),agenda,R)

- Terminación:** Si **agenda** esta vacía, devuelve <A,O,L>.
- Selección del objetivo:** Sea <Q,A<sub>e</sub>> un par de la **agenda** (por definición A<sub>e</sub> ∈ A y Q es un elemento de la conjunción de la precondición de A<sub>e</sub>).
- Selección de la acción:** Sea A<sub>nuevo</sub> = **elección** de una acción que añada Q (ya sea mediante una nueva acción de R o mediante una acción existente en A que pueda ordenarse consistentemente antes de A<sub>e</sub>). Si no existiese tal acción entonces devolver fallo. En otro caso, sea L' = L ∪ {A<sub>nuevo</sub>  $\xrightarrow{Q}$  A<sub>e</sub>}, y sea O' = O ∪ {A<sub>nuevo</sub> < A<sub>e</sub>}. Si A<sub>nuevo</sub> es una acción nueva, entonces A' = A ∪ {A<sub>nuevo</sub>} y {A<sub>0</sub> < A<sub>nuevo</sub> < A<sub>∞</sub>} (en otro caso A' = A).
- Actualizar el conjunto de objetivos:** Sea agendanueva=agenda-{<Q,A<sub>e</sub>>}. Si A<sub>nuevo</sub> no estaba antes, entonces para cada elemento Q<sub>i</sub> de la conjunción de su precondición añadir <Q<sub>i</sub>,A<sub>nuevo</sub>> a **agendanueva**.
- Protección de enlaces causales:** Para acción A<sub>t</sub> que pudiese amenazar a un enlace causal A<sub>p</sub>  $\xrightarrow{H}$  A<sub>c</sub> ∈ L' elegir una restricción de orden consistente, o bien
  - ascenso:** añadir A<sub>t</sub> < A<sub>p</sub> a O', o
  - degradación:** añadir A<sub>c</sub> < A<sub>t</sub> a O'.
 Si ninguna restricción es consistente, entonces devolver fallo.
- Recursión:** Llamar a POP(<A',O',L'>),agendanueva,R).

**QUIERES  
CONSEGUIR  
15€??**

TRÁENOS A TU  
CRUSH DE APUNTES  
ANTES DE QUE  
LOS QUEME



#### Heurísticas para la Planificación de Orden Parcial

- Se tienen diferentes heurísticas para la selección de planes
  - Número de acciones en el plan  $N$ , sería la  $g$ .
  - Precondiciones sin resolver  $OP$ , sería la  $h$ . Es una estimación de las acciones que faltan.
- Frecuentemente se utiliza el algoritmo A\* con la heurística  $f(P) = N(P) + OP(P)$ .
- No tiene garantía absoluta que sea admisible porque una misma acción puede resolver dos precondiciones, luego la  $h$  sobreestima lo que falta por resolver, aún así es una heurística muy buena.

#### Planificación Jerárquica

- A menudo es interesante generar un plan inicial compuesto por tareas de alto nivel para luego ir desglosándolo en acciones más simples.
- Esto es especialmente útil en problemas complejos en donde planificar desde cero todas las acciones de bajo nivel puede resultar una tarea muy costosa.
- Existen dos vías generales para esta planificación:
  - **ABSTRIPS**, que es una extensión directa de STRIPS. Se jerarquizaba el problema de STRIPS con la asignación de valores numéricos de criticidad sobre los predicados de las precondiciones. Considera el problema a nivel de criticidad máximo, cuando se tiene que insertar una acción aparecen las de nivel más alto solamente, luego se reduce la criticidad y se comienza de nuevo con el plan que dejó el nivel anterior utilizando acciones del nivel mayor hasta la del nivel actual, se va refinando el plan.
  - **Redes de Planificación Jerárquica de Tareas**: Hace uso intensivo del conocimiento y requiere que el mismo sea descrito de forma jerárquica, es decir, requiere un trabajo más arduo al principio pero los planificadores son mucho más rápidos y eficientes.
    - Se tienen dos tipos de acciones:
      - Métodos (Alto nivel), describen acciones con primitivas.
      - Acciones Primitivas (Bajo nivel)
    - Los métodos se pueden descomponer en subtareas que pueden ser otros métodos o acciones primitivas.
    - Esta descomposición se define mediante una red de tareas, que establece la ordenación total o parcial de las subtareas de los métodos.
    - El funcionamiento de un planificador HTN es muy similar al de un planificador clásico
    - El objetivo del problema consiste en aplicar una serie de tareas mediante descomposición y planificación de una secuencia de métodos que, finalmente, se desglosará en las acciones primitivas.
    - Presenta dos ventajas:
      - El dominio del problema se describe en términos de acciones estructuradas jerárquicamente, resultando más intuitivo para el experto que modela el problema.
      - La función de planificador consiste en refinar estas estructuras generando las expansiones necesarias y simplifica la resolución del problema original.
    - Esto es un grafo Y/O por debajo que resuelve estas tareas.

#### Representación para Planes:

- Se generó al mismo tiempo que surge STRIPS
- El objetivo es conseguir una representación para planes que permita:
  - Usar un plan previamente generado para la resolución de problemas posteriores.
  - Controlar inteligentemente la ejecución de un plan concreto.
- Se basa en que los humanos interiorizan un plan y ya posteriormente tienen internamente una versión general del plan, se entienden como macrooperaciones.
- Se utiliza una vez que se ha generado un plan, permite monitorizarlo y en ocasiones permite reparar un plan si sucede algún inconveniente.

#### Tablas Triangulares

- Dado un plan que ya se haya obtenido se empieza a llenar la tabla, la representación se describe con una estructura de datos llamada una tabla triangular inferior de  $n + 1$  operaciones del plan con una numeración

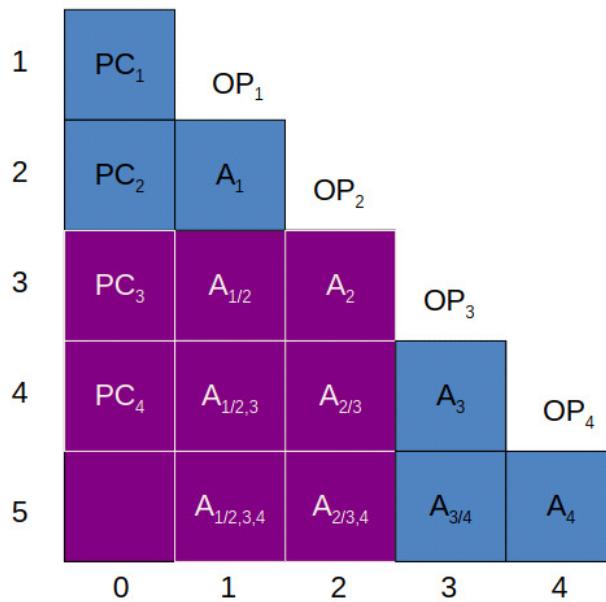


si  
consigues  
que suba  
apuntes, te  
llevas 15€ +  
5 Wuolah  
Coins para  
los sorteos

**WUOLAH**

especial de filas y columnas.

- Las columnas van desde 0 hasta  $n$  de izquierda a derecha.
- En las filas es de 1 hasta  $n + 1$  de arriba abajo.
- Luego de esto, se etiqueta la tabla con los operadores encima del número que le corresponde al operador, ahora se entiende el porqué de la numeración: el operador  $OP_1$  está asociado a la fila 1 y la columna 1, esta propiedad se verifica para el resto de operadores.
- Ahora se rellena el interior:
  - $A_1$  refleja la lista de Adición del  $OP_1$ , las listas que siguen debajo tienen que ver con esa lista de adición pero ahora teniendo en cuenta el resto de los operadores, o sea,  $A_i$  es la lista de adición de  $OP_i$
  - $A_{i/j}$  son los literales de  $A_i$  no suprimidos por  $OP_j$ .
- El significado de una fila menos en la posición 0, son los literales que van añadiendo una lista de operadores.
- Se tendrá toda la tabla menos la columna 0 rellena.
  - Está relacionada con las precondiciones de los operadores del plan, un operador se puede aplicar en un estado si sus precondiciones son ciertas y se podrá aplicar en un estado futuro porque o bien hay otro operador que añade cosas que lo necesita, están relacionados los operadores; o porque el literal que se necesita es cierto en el estado inicial y no ha cambiado a lo largo del tiempo.
- En la fila quinta debe estar el objetivo que se busca.
- Núcleo i-ésimo: Es una fórmula de la lógica y no está perfectamente definido, es una conjunción de literales de los literales marcados que se encuentran en una región de la tabla: es la intersección de los literales de las filas que quedan por debajo de la i-ésima incluida esta y con las columnas que quedan a la izquierda de la i-ésima no incluida.



El núcleo representa la  
precondición de la  
secuencia parcial

$$\{OP_i, OP_{i+1}, \dots, OP_n\}$$

- Permite comprobar si el plan va a fallar o no, el núcleo permite tener una garantía que el plan va a llegar al objetivo. Si no hay ningún núcleo válido, se tiene que replanificar. Si una operación se hace falsa, se busca en los núcleos alguna que se vuelva cierta y se toma desde allí.
- La estructura relevante es el núcleo, permite aprender plantes y para ejecutarlos inteligentemente se puede ver la columna primera o la última fila.