Planeación

En planeación, dada una descripción del mundo, una situación inicial y una situación deseada, la meta es hallar una secuencia de acciones (las cuales pueden cambiar las situaciones), tal que la situación deseada es alcanzada desde el estado inicial. Además, no todas las acciones son aplicables en todas las situaciones.

Definición 1. La Planeación es una representación de estados describiendo un comportamiento futuro con la ejecución de un conjunto de acciones bajo un orden con limitantes temporales o algún otro tipo de éstas.

Planeación

Definición 2. Un problema de planeación es una cuádrupla

< ***F, A, I, G*** > donde:

- ***F*** es un conjunto de fluentes, los cuales caracterizan las situaciones
- ***A*** es un conjunto de acciones, con una definición de de sus respectivas precondiciones y efectos o causas.
- ***I*** es un conjunto de fluentes describiendo la situación inicial
- ***G*** es un conjunto de fluentes describiendo la situación meta o deseada

Planeación

Definición 3. Dado un PP $P = < \boldsymbol{F, A, I, G} >$ y un entero n, un plan para un PP es una secuencia $A = a_1, \ldots, a_n$ tal que hay n+1 situaciones $S_0, \ldots, S_n$ tales que para cada a_i en A, S_{i-1} es consistente con las precondiciones a_i's y S_i es modificado desde S_{i-1}, por exactamente los efectos de a_i. Para hallar el plan válido para las situaciones dadas, S_0 debe ser implicado por I, y G debe ser implicado por S_n.

El mundo de los cubos
Ejemplo

Configuración Inicial:

sobre(a,mesa)

sobre(c,mesa)

sobre(b,mesa)

brazovacio

libre(a)

libre(c)

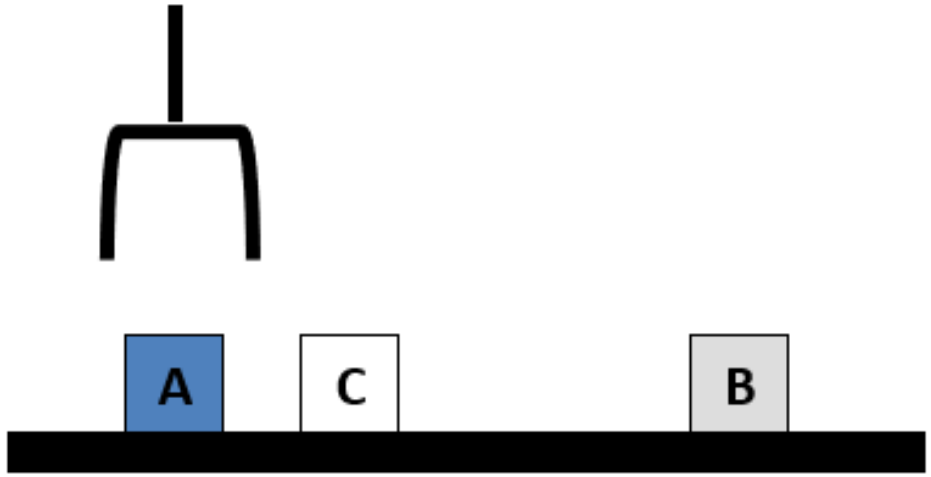
libre(b)

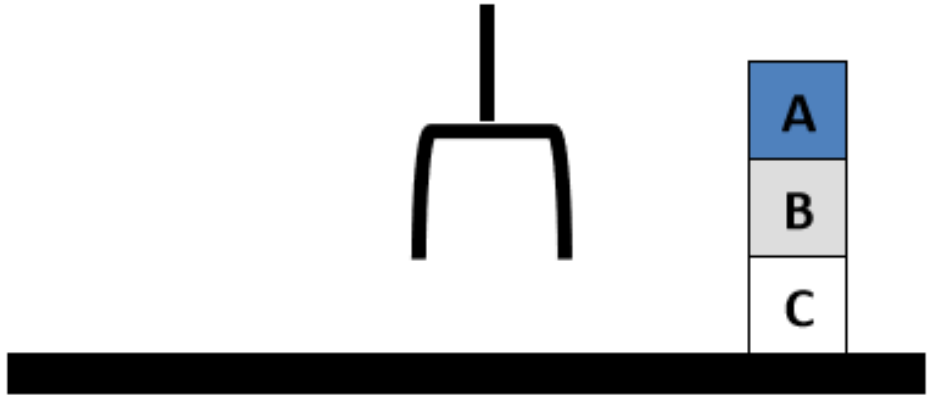
Goal:

sobre(a,b)

sobre(b,c)

sobre(c,mesa)





Plan:

coger(b)

poner(b,c)

coger(a)

poner(a,b)

Sintaxis DLV-k

Declaraciones/Fluentes.- Todas las declaraciones y fluentes deben ser declarados usando la declaración de acción de la forma:

fluents(actions):

$p(X_1, \ldots, X_n)$ requires $t_1, \ldots, t_m$

donde $p \in L_{act}$,

$X_1, \ldots, X_n$ son variables

$t_1, \ldots, t_m \in L_{typ}$,

n es la aridad de p

todas las X_i ocurren en $t_1, \ldots, t_m$ y $m \geq 0$

Si m=0 la parte *requires* puede ser omitido

Sintaxis DLV-k

fluents:

$p(X_1, \ldots, X_n)$ requires $t_1, \ldots, t_m$

sobre(B,L) requires cubo(B), lugardisponible(L).

ocupado(B) requires lugardisponible(B).

actions:

mover(B,L) requires cubo(B), lugardisponible(L).

Sintaxis DLV-k

causation rules: estas son usadas para para definir dependencias estáticas y dinámicas y tiene la forma siguiente:

$$\textit{cuased}\ \ f\ \ \text{if}\ b_1, \ldots, b_k, \text{not}\ b_{k+1}, \ldots, \text{not}\ b_l$$

$$\textit{after}\ a_1, \ldots, a_m, \text{not}\ a_{m+1}, \ldots, \text{not}\ a_n$$

donde f puede ser fluente, tipo y acciones $\cup$ {false}

Una condición de ejecución puede ser de la forma:

$$\textit{executable}\ a\ \text{if}\ b_1, \ldots, b_m, \text{not}\ b_{m+1}, \ldots, \text{not}\ b_n$$

Sintaxis DLV-k

Una condición de ejecución puede ser de la forma:

executable *a* if b_1, …, b_m, not b_{m+1}, …, not b_n

Ejemplo.

always:

executable mover(B,L) *if* not ocupado(B), not ocupado(L), B <> L.

inertial sobre(B,L).

caused ocupado(B) if sobre(B1,B), cubo(B).

caused sobre(B,L) after mover(B,L).

caused -sobre(B,L1) after mover(B,L), sobre(B,L1), L <> L1.

Sintaxis DLV-k

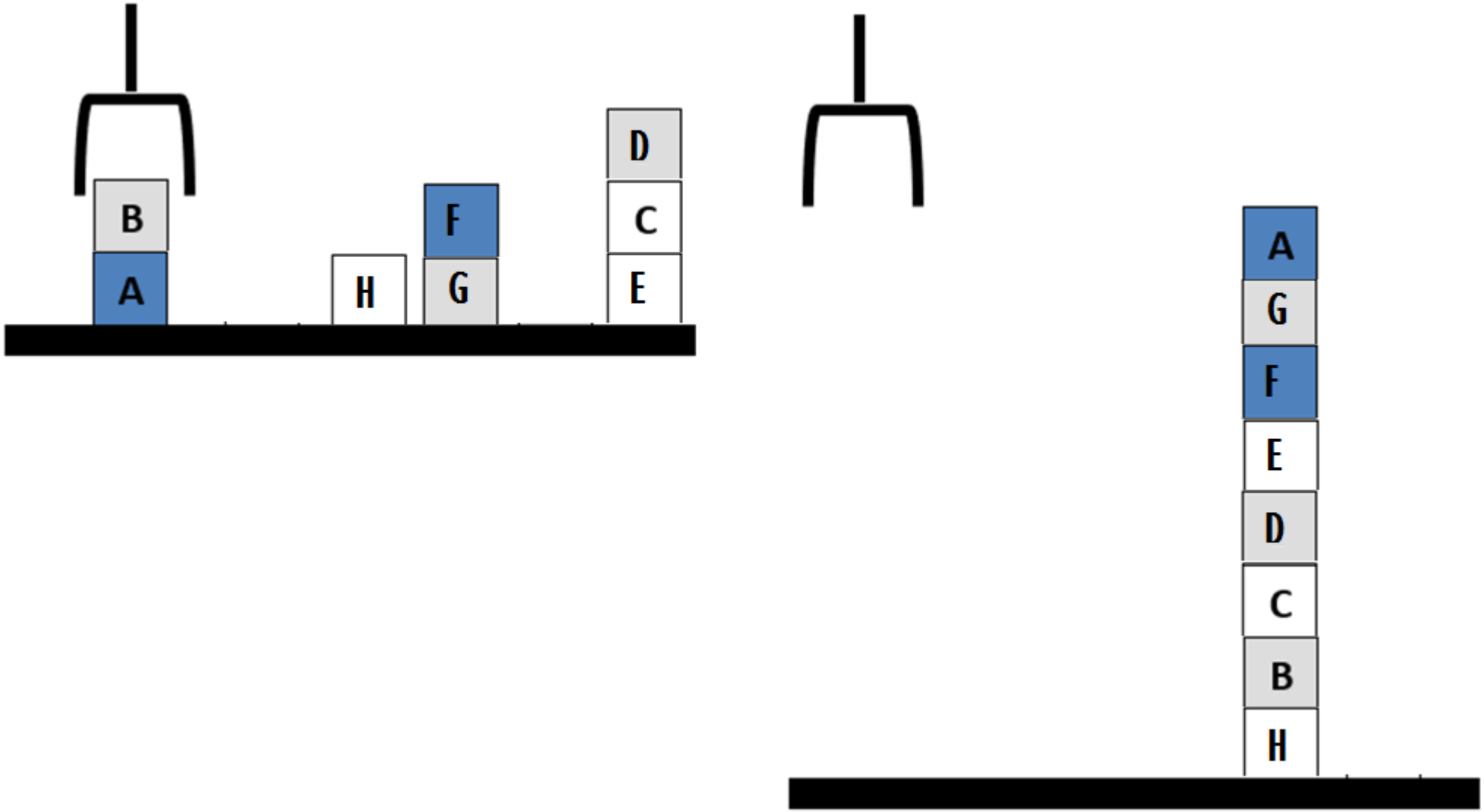
Un query es de la forma:

$$g_1, \ldots, g_m, \text{not } g_{m+1}, \ldots, \text{not } g_n \text{ ? (i)}$$

Donde los g_i variables libres y pertenecen a los fluentes, $i \geq 0$, $n \geq m \geq 0$

Ejemplo.

El mundo de los cubos
Ejemplo



Tipos de planeadores

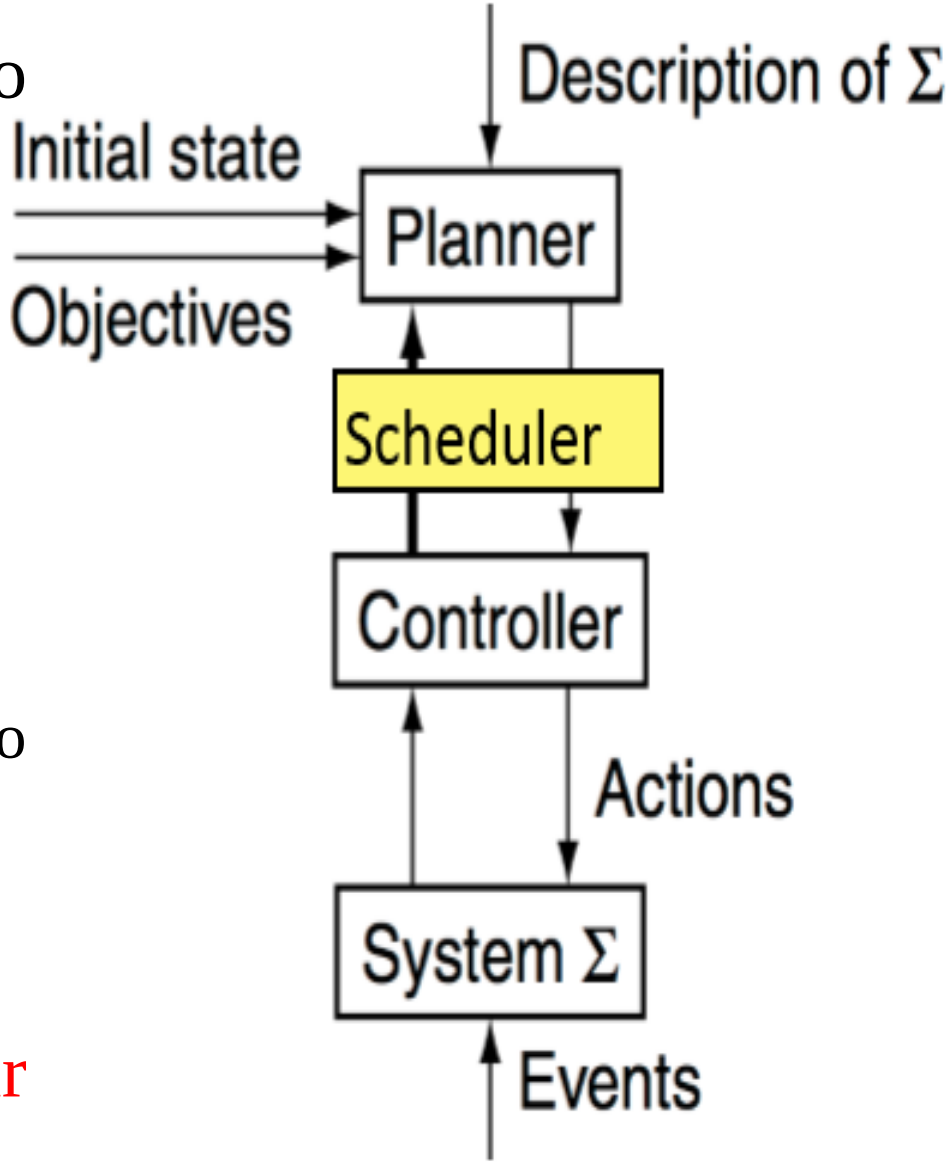
- Scheduling

- Decide cuando y como realizar un conjunto dado de acciones
  - Restricciones de tiempo
  - Restricciones de recursos
  - Funciones objetivo

- Problema NP completo

- Planning
  - Decide que acciones usar para alcanzar algún conjunto de objetivos
  - Puede ser mucho peor que un problema NP completo

- Los problemas de scheduling pueden requerir planeación



Principales tipos de planeadores

- De dominio especifico
  - Hecho para un dominio de planeación especifico
  - No trabaja bien en otros dominios de planeación
- Independiente del dominio
  - En principio, trabaja en cualquier dominio de planeación
  - En la práctica, necesita restricciones sobre la clase de dominio de planeación
- Configurable
  - Máquina de planeación independiente del dominio
  - Las entradas incluyen información acerca de como resolver problemas en algún dominio

Planeadores de dominio especifico

- La mayoría de los sistemas de planificación del mundo real exitosos funcionan de esta manera

- Frecuentemente usan técnicas de problemas especificos que son difíciles de generalizar a otros dominios de planeación

Planeadores independientes del dominio

- En principio, trabajan en cualquier dominio de planificación
- Sin el conocimiento de dominio específico, excepto la descripción del sistema Σ
- En la práctica
  - No es factible hacer planificadores independientes del dominio que trabajen bien en todos los ámbitos de planificación posibles
- Hacer supuestos simplificadores para restringir el conjunto de dominios
  - Planeación clásica
  - Enfoque histórico de mucha investigación sobre la planificación automatizada

Hipotesis restrictivas

1. Sistema finito
   - un número finito de estados, acciones y eventos
2. Observable totalmente
   - El controlador siempre esta en el estado actual Σ's
3. Deterministico
   - Cada acción tiene solo un resultado
4. Estático
   - no hay cambios, para las acciones del controlador
5. Metas de consecución
   - Un conjunto de estados meta u objetivo
6. Planes secuenciales
   - Un plan es una secuencia lineal ordenada linealmente de acciones (a1, a2, …, an)
7. Tiempo implícito
   - no hay duraciones de tiempo; secuencia lineal de estados instantáneos
8. Planeación OFF-line
   7. El planeador no tiene conocimiento del status de ejecución

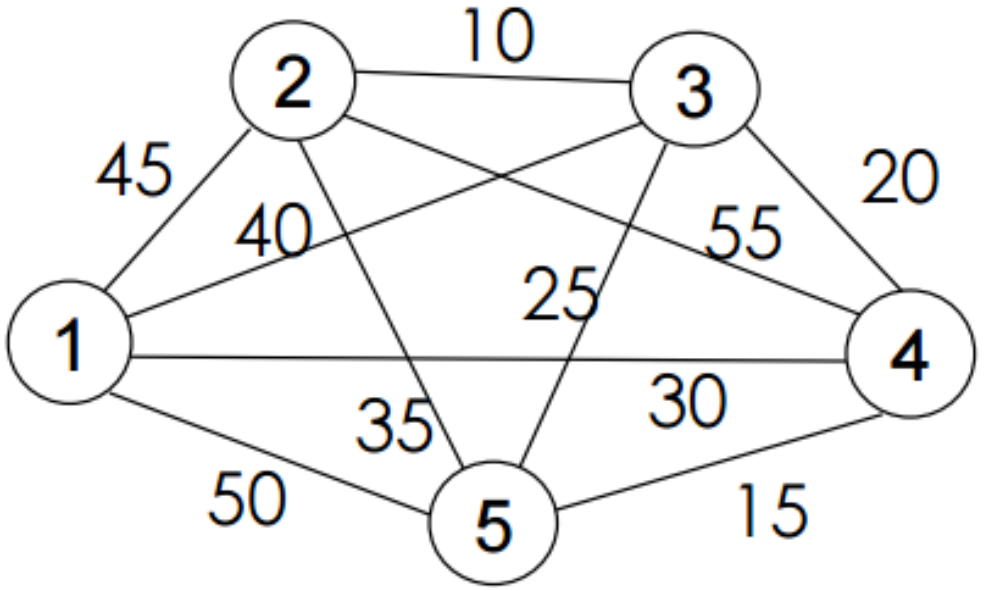
Planeación clásica

- La planeación clásica requiere de todas las hipótesis restrictivas
- La generación off-line de secuencias de acciones para un sistema determinístico, estático y finito, con conocimiento completo, metas de logros y tiempo implícito
- Se reduce al problema siguiente:
  - Dado un problema de planeación P = (Σ, s0, Sg)
- Hallar una secuencia de acciones (a1, a2, …, an) que produzca una secuencia de transiciones entre estados (s1, s2, …, sn) tal que sn está en Sg. Esto es justamente el camino que se busca en un grafo
  - Nodos = estados
  - Ejes = acciones
- Es trivial?????

Agente viajero

Formulación Intuitiva: Un viajante de comercio debe recorrer una serie de ciudades y volver a la de partida recorriendo la menor distancia posible, es decir, sin pasar dos veces por ninguna de ellas. Se supone que hay caminos entre cada par de ciudades.

Formulación matemática: Dado un grafo completo con costos positivos asociados a sus aristas, encontrar un tour (ciclo simple que incluya a todos sus vértices) de costo mínimo (es decir, la suma de los costos de sus aristas es el menor posible).



Problema: Dado un grafo no dirigido y con pesos G = (V, A), encontrar un ciclo simple de costo mínimo que pase por todos los nodos.

- Es un problema NP, pero se necesita una solución eficiente.
- Problema de optimización, donde la solución está formada por un grupo de elementos en cierto orden: Se puede aplicar el esquema voraz.

Agente viajero

inPath(X,Y) v outPath(X,Y) :- arc(X,Y,**Cost**). Guess

:- inPath(X,Y), inPath(X,Y1), Y <> Y1.

:- inPath(X,Y), inPath(X1,Y), X <> X1. Check

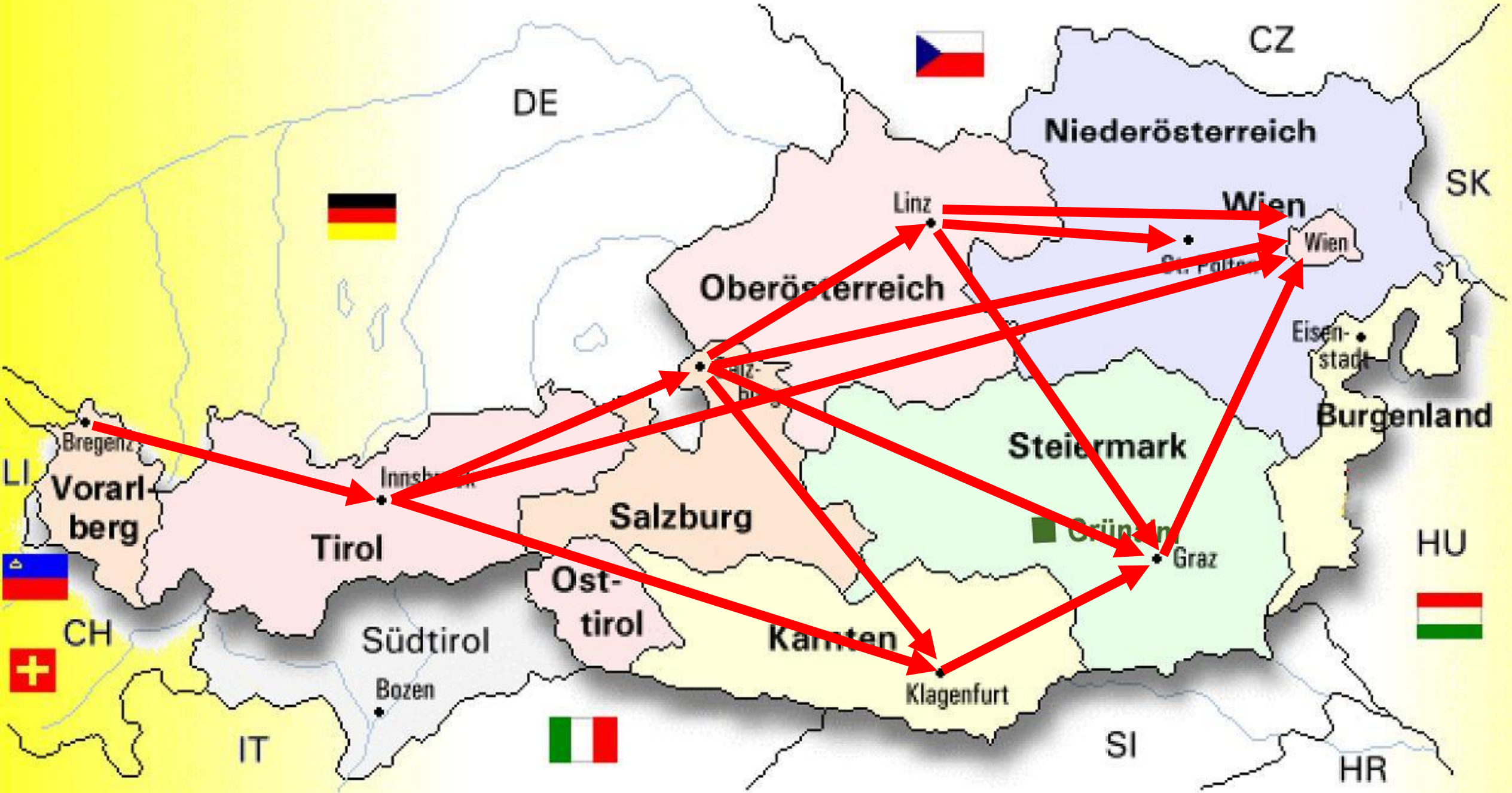
:- node(X), not reached(X).

reached(X) :- start(X). Predicado auxiliar

reached(X) :- reached(Y), inPath(Y,X).

+ Optimización

:~ inPath(X,Y), arc(X,Y,Cost). [Cost:] Optimiza


DE
CZ
SK
HU
LI
CH
IT
SI
HR
Niederösterreich
Oberösterreich
Linz
Wien
Wien
Eisen-
stadt
Burgenland
Steiermark
Graz
Salzburg
Tirol
Vorarl-
berg
Bregenz
Ost-
tirol
Südtirol
Bozen
Klagenfurt

```
conn(brg,ibk,2).
conn(ibk,sbg,2).
conn(ibk,vie,5).
conn(ibk,kla,3).
conn(sbg,kla,2).
conn(sbg,gra,2).
conn(sbg,lin,1).
conn(sbg,vie,3).
conn(kla,gra,2).
conn(lin,stp,1).
conn(lin,vie,2).
conn(lin,gra,2).
conn(gra,vie,2).
conn(gra,eis,1).
conn(stp,vie,1).
conn(eis,vie,1).
conn(stp,eis,2).
```

Regla que indica que las conecciones son en ambas direcciones

```
conn(B,A,C) :- conn(A,B,C).
```

```
weekday(1,1).
 weekday(D,W) :- D=D1+1, W=W1+1, weekday(D1,W1), W1 < 7.
 weekday(D,1) :- D=D1+1, weekday(D1,7).


city(T) :- conn(T,_,_).


cost(A,B,W,C) :- conn(A,B,C), #int(W), 0 < W, W <= 7, not ecost(A,B,W).
ecost(A,B,W) :- cost(A,B,W,C), conn(A,B,C1), C != C1.
cost(stp,eis,2,10).
```

fluents: unvisited.
in(C) requires city(C).
visited(C) requires city(C).
actions: travel(X,Y) requires conn(X,Y,_) costs C
where weekday(time,W), cost(X,Y,W,C).
always: executable travel(X,Y) if in(X). nonexecutable travel(X,Y) if visited(Y).

caused unvisited if city(C), not visited(C). caused in(Y) after travel(X,Y).
caused visited(C) if in(C). inertial visited(C).

noConcurrency.

initially: in(vie).

goal: not unvisited? (8)