

**Resolución del problema del Granjero mediante el método de Búsqueda en Amplitud  
con Costo en Prolog**

**Grupo 5**

**Daniel Santiago Avila Medina  
Alejandra Abaunza Suárez  
Jeison Camilo Alfonso Moreno  
Santos Alejandro Arellano Olarte**

**Introducción a la Inteligencia Artificial**

**Laura Mora Paez**

**Pontificia Universidad Javeriana**

**Bogotá D.C**

Para “desglosar” el programa en Prolog que se realizó para esta entrega, vamos a explicar paso a paso:

### 1. Estado inicial y final

```
% Definimos nuestro estado inicial y final  
estado_inicial(estado(i,i,i,i)).  
estado_final(estado(d,d,d,d)).
```

Se representa el estado como una tupla de 4 posiciones, las cuales son:

**estado(G,L,C,H) = (granjero, lobo, cabra, heno).**

Cada posición puede ser **i (izquierda)** o **d (derecha)**, siendo el estado inicial el que se observa en la primera imagen:

**estado(I,I,I,I).**

Y el estado final:

**estado(D,D,D,D).**

- 2. Lados opuestos:** En este caso son dos hechos que creamos para definir la relación de oposición entre orillas. Se utilizan para mover al granjero (y a quien lleve) al otro lado.

```
% Declaramos Lados opuestos  
opuesto(i,d).  
opuesto(d,i).
```

### 3. Estados inseguros y seguros:

```
% Estados inseguros  
inseguro(estado(G,L,C,_)) :- L = C, G \= L.   % Lobo se come cabra  
inseguro(estado(G,_,C,H)) :- C = H, G \= C.   % Cabra se come al heno  
seguro(S) :- \+ inseguro(S).
```

Definimos que un estado es inseguro si en este ocurre una de las siguientes condiciones (restricciones que teníamos en el taller), las cuales son: el lobo y la cabra quedan solos sin el granjero o que la cabra y el heno quedan solos.

```

inseguro(estado(G,_,C,H)) :- C
seguro(S) :- \+ inseguro(S).

```

En este fragmento definimos que un estado es seguro si no se puede demostrar que es inseguro, adicional a la presentación que se vió en clase, obtuvimos más claridad para esta parte en: [SWI-Prolog -- \(\+\)/1](#)

#### 4. Movimientos posibles:

```

% Movimientos que podemos hacer:
% Granjero cruza solo (m1)
mover(estado(G,L,C,H), estado(G2,L,C,H), m1_solo) :-
    opuesto(G,G2),
    seguro(estado(G2,L,C,H)).

% Granjero cruza con el Lobo (m2)
mover(estado(G,L,C,H), estado(G2,G2,C,H), m2_lobo) :-
    G = L, opuesto(G,G2),
    seguro(estado(G2,G2,C,H)).

% Granjero cruza con la cabra (m3)
mover(estado(G,L,C,H), estado(G2,L,G2,H), m3_cabra) :-
    G = C, opuesto(G,G2),
    seguro(estado(G2,L,G2,H)).

% Granjero cruza con el heno (m4)
mover(estado(G,L,C,H), estado(G2,L,C,G2), m4_heno) :-
    G = H, opuesto(G,G2),
    seguro(estado(G2,L,C,G2)).

```

Estos son todos los movimientos que podemos realizar tal y como definimos en el taller del granjero, de igual manera, en todos estos casos verificamos que el nuevo estado sea seguro.

#### 5. Función de costo

```

% Costo del estado == cuántos objetos siguen en la orilla izquierda
costo_estado(estado(_,L,C,H), Costo) :- contar_i([L,C,H], Costo).
contar_i([],0).
contar_i([i|T],N) :- !, contar_i(T,N1), N is N1+1.
contar_i([_|T],N) :- contar_i(T,N).

```

Aquí es donde definimos el costo de un estado como el número de cosas(lobo, cabra, heno) que todavía están en la orilla izquierda, por ejemplo:

- Si todos están a la izquierda el costo = 3.
- Si solo falta por mover el heno (O cualquier otro) el costo = 1.
- Y si todos están en la derecha el costo = 0.

También es importante destacar que no se cuenta al granjero porque él siempre debe moverse, también es importante mencionar que la función de costo no reordena toda la frontera porque eso ya sería convertir la estrategia en un Best First Search (que no es lo que se nos pidió) Lo que hicimos fue mantener el recorrido en amplitud pura, es decir, expandiendo siempre nivel por nivel para garantizar el camino mínimo en número de pasos, La diferencia está en que, cuando generamos los hijos de un nodo, antes de encolarlos al final, los ordenamos localmente con la función de costo.

De esta manera, se sigue cumpliendo la idea de amplitud, pero la búsqueda se orienta un poco mejor hacia los estados que tienen más avance (menos objetos en la orilla izquierda) donde respetamos la lógica del BFS y al mismo tiempo usamos una heurística sencilla para decidir el orden de los hijos.

## 6. Cola para BFS

```
% Cola para el BFS
cola_vacia([]).
encolar(E,Q,Q2) :- append(Q,[E],Q2).
encolar_lista(Q,[],Q).
encolar_lista(Q,[H|T],Qf) :- encolar(H,Q,Q1), encolar_lista(Q1,T,Qf).

enCola(E,[nodo(E,_,_,_)|_]).
enCola(E,[_|R]) :- enCola(E,R).
```

Aquí básicamente implementamos una cola basándonos en: [Lists in Prolog - GeeksforGeeks](#) y también en la primera respuesta de: [Creating a queue structure in Prolog - Stack Overflow](#) para explicar un poco más a detalle tenemos que:

- cola\_vacia/1: cola inicial.
- encolar/3: mete un elemento al final.
- encolar\_lista/3: mete una lista de elementos.
- enCola/2: verifica si un estado ya está en la cola (evita que haya duplicados).

## 7. Búsqueda en amplitud con costo

```

% -Hacemos BFS con costo
resolver_bfs_costo(CaminoEstados,CaminoMovs,Expandidos) :-
    estado_inicial(E0),
    cola_vacia(Q0),
    encolar(nodo(E0,[E0],[],0),Q0,Q1),
    bfs_costo(Q1,[],NodoSol,ExpandidosRev),
    NodoSol = nodo(_,CERev,CMRev,_),
    reverse(CERev,CaminoEstados),
    reverse(CMRev,CaminoMovs),
    reverse(ExpandidosRev,Expandidos).

```

Cuando construimos cada nodo, además del estado guardamos el camino de estados y de movimientos que nos llevaron hasta ahí. Lo que pasa es que Prolog trabaja mucho más rápido cuando agregamos elementos por la cabeza de la lista, porque eso es una operación  $O(1)$ , mientras que meterlos al final costaría mucho más.

Por eso nosotros vamos guardando el camino “al revés”: cada vez que generamos un hijo, simplemente lo añadimos adelante en la lista. Eso hace que al final tengamos el camino completo pero invertido. Para dejarlo en el orden correcto (del estado inicial al estado final) usamos reverse/2.

## 8. Caso meta

```

% Caso meta que queremos
bfs_costo([nodo(E,CE,CM,D)|_],_,nodo(E,CE,CM,D),[E]) :-
    estado_final(E), !.

```

En el caso meta lo que hacemos es revisar siempre el primer nodo de la cola, porque como estamos trabajando en amplitud, el primer nodo que salga de la cola será el que tenga el menor número de pasos. Si ese estado ya es el estado final, significa que encontramos la solución óptima y no necesitamos seguir explorando más niveles.

Por eso la condición es que E sea igual al estado\_final. Cuando eso se cumple, simplemente devolvemos ese nodo como la solución. El operador ! (corte) está ahí para asegurar que Prolog no intente buscar más soluciones alternativas, ya que la primera vez que encontramos la meta en BFS sabemos que necesariamente es la de menor costo en pasos.

En otras palabras, ese es el caso meta porque representa la condición de parada el cual es llegar al estado donde el granjero, el lobo, la cabra y el heno están todos en la orilla derecha. Y lo evaluamos en la cabeza de la cola porque BFS garantiza que es el camino más corto posible.

## 9. Paso recursivo

```
% Paso recursivo
bfs_costo([Nodo|Resto],Cerrados,NodoSol,[E|ExpTail]) :-
    Nodo = nodo(E,_,_,_),
    \+ member(E,Cerrados),
    expandir_hijos(Nodo,Resto,Cerrados,Hijos),
    encolar_lista(Resto,Hijos,NuevaCola),
    bfs_costo(NuevaCola,[E|Cerrados],NodoSol,ExpTail),
    E = E.
```

Aquí se toma el primer nodo de la cola, en el caso de que no este en Cerrados:

- Se generan sus hijos.
- Se ordenan localmente según costo\_estado/2.
- Se encolan al final de la cola.
- Se continúa recursivamente.

Y como explicamos anteriormente, aquí es donde está la diferencia clave de nuestro algoritmo de amplitud con costo frente al de Best-First: en nuestro caso no reordenamos toda la frontera, sino que únicamente ordenamos los hijos de un nodo entre ellos antes de encolarlos al final. De esta forma seguimos respetando la lógica del BFS, expandiendo nivel por nivel y garantizando siempre el camino mínimo en pasos, pero al mismo tiempo usamos la función de costo que hicimos para dar prioridad local a los estados que tienen más avance hacia la meta.

## 10. Saltar estados cerrados

```
% Si ya estaba cerrado
bfs_costo([nodo(E,_,_,_)|Resto],Cerrados,NodoSol,Exp) :-
    member(E,Cerrados), !,
    bfs_costo(Resto,Cerrados,NodoSol,Exp).
```

Si el estado ya está en Cerrados, se ignora y se pasa al siguiente nodo.

## 11. Expansión y ordenamiento de hijos

```

% Expandimos y ordenamos hijos por costo
expandir_hijos(nodo(E,CE,CM,D),Cola,Cerrados,Hijos) :-
    findall(K-Hijo,
        ( mover(E,E2,Acc),
          \+ member(E2,Cerrados),
          \+ en_cola(E2,Cola),
          costo_estado(E2,K),
          Hijo=nodo(E2,[E2|CE],[Acc|CM],D+1)
        ),
        Pares),
    keysort(Pares,Ordenados),
    pares_valores(Ordenados,Hijos).

pares_valores([],[]).
pares_valores([_V|T],[V|R]) :- pares_valores(T,R).

```

- findall/3 genera todos los hijos válidos.
- Cada hijo se guarda como K-Hijo, donde K es el costo.
- keysort/2 ordena por clave (menor costo primero).
- pares\_valores/2 elimina las claves y deja solo los hijos ordenados.

## 12. Imprimir solución

```

imprimir_solucion(Estados,Movs) :-
    writeln('Estados'),
    forall(member(E,Estados),(write(' '),writeln(E))),
    writeln('Movimientos'),
    forall(member(M,Movs),(write(' '),writeln(M))).

```

Aquí no hay mucho que explicar, se imprimen todos los estados y los movimientos que llevaron de uno al siguiente y usamos forall/2 para recorrer las listas.

## 13. Prueba

```

prueba :- resolver_bfs_costo(Estados,Movs,_), imprimir_solucion(Estados,Movs).

```

Aquí básicamente creamos un atajo que lanza la búsqueda e imprime la solución para poder verificar que todo esté bien, dando el siguiente output:

Estados

estado(i,i,i,i)

estado(d,i,d,i)

estado(i,i,d,i)

estado(d,d,d,i)

estado(i,d,i,i)

estado(d,d,i,d)

estado(i,d,i,d)

estado(d,d,d,d)

Movimientos

m3\_cabra

m1\_solo

m2\_lobo

m3\_cabra

m4\_heno

m1\_solo

m3\_cabra

true

Next

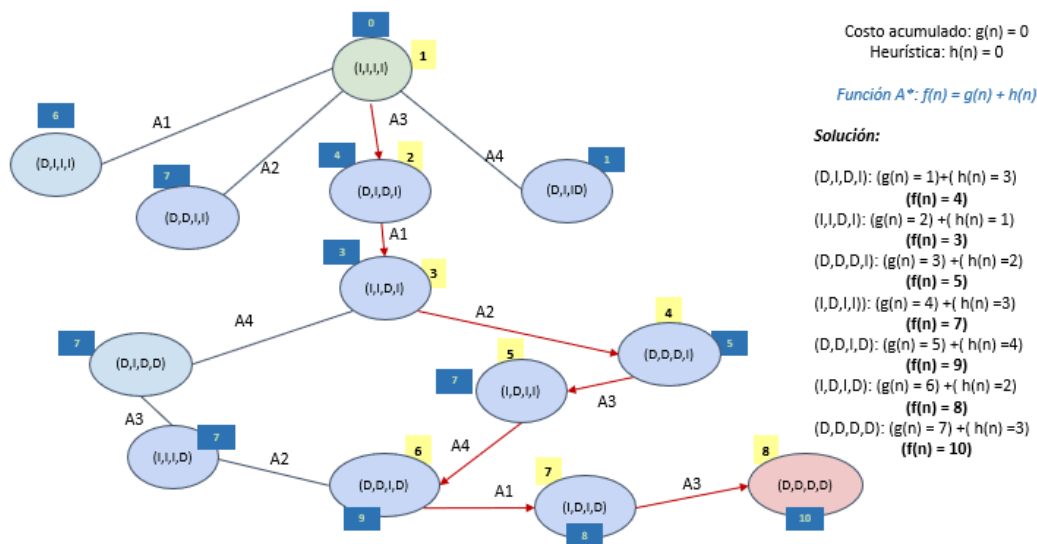
10

100

1,000

Stop

Para verificar que quedo bien nos basamos en nuestro taller y encontramos que:



En efecto, está bien ya que los movimientos que se realizan son los mismos que en el taller que realizamos!

### Comandos de prueba que utilizamos:

Adicional a **prueba.** que mencionamos anteriormente, también ejecutamos:

**Ver estados, movimientos y orden de expansión:**



```
resolver_bfs_costo(Estados,Movs,Expandidos),
imprimir_solucion(Estados,Movs),
writeln('Expandidos:'), writeln(Expandidos).
```

En este caso Expandidos nos muestra en qué orden se sacaron nodos de la cola:

Movimientos

m3\_cabra

m1\_solo

m2\_lobo

m3\_cabra

m4\_heno

m1\_solo

m3\_cabra

Expandidos:

```
[estado(d,d,d,d), estado(i,d,i,d), estado(d,d,i,d), estado(i,i,i,d), estado(i,d,i,i), estado(d,i,d,d),
estado(d,d,d,i), estado(i,i,d,i), estado(d,i,d,i), estado(i,i,i,i)]
```

**Estados** =

```
[estado(i,i,i,i), estado(d,i,d,i), estado(i,i,d,i), estado(d,d,d,i), estado(i,d,i,i), estado(d,d,i,d),
estado(i,d,i,d), estado(d,d,d,d)]
```

,

**Expandidos** =

```
[estado(d,d,d,d), estado(i,d,i,d), estado(d,d,i,d), estado(i,i,i,d), estado(i,d,i,i), estado(d,i,d,d),
, estado(d,d,d,i), estado(i,i,d,i), estado(d,i,d,i), estado(i,i,i,i)]
```

,

**Movs** = [m3 cabra, m1 solo, m2 lobo, m3 cabra, m4 heno, m1 solo, m3 cabra]

Comprobar cantidad de pasos (7 para llegar a el estado final)

```
resolver_bfs_costo(Estados,Movs,_),
length(Movs,7),|
last(Estados,estado(d,d,d,d)).
```

Y nos da el siguiente resultado:

**Estados** =

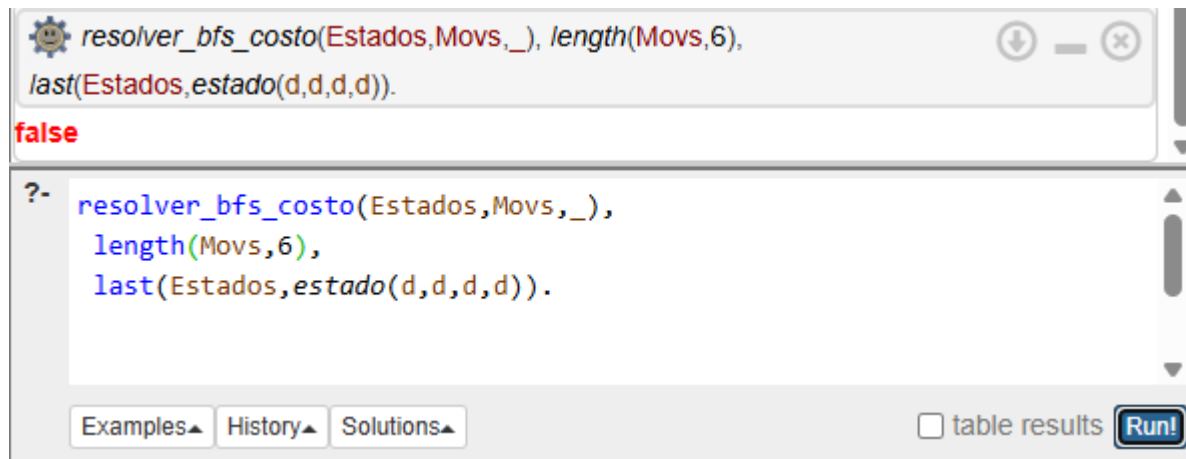
```
[estado(i,i,i,i), estado(d,i,d,i), estado(i,i,d,i), estado(d,d,d,i), estado(i,d,i,i), estado(d,d,i,d),
estado(i,d,i,d), estado(d,d,d,d)]
```

,

**Movs** = [m3\_cabra, m1\_solo, m2\_lobo, m3\_cabra, m4\_heno, m1\_solo, m3\_cabra]

Next 10 100 1,000 Stop

Por lo que en efecto, está bien, si en vez de ponerle un 7 le ponemos un 8 (u otro número) nos retornará false ya que se hace en 7 pasos



```
resolver_bfs_costo(Estados,Movs,_), length(Movs,6),
last(Estados,estado(d,d,d,d)).

false

?- resolver_bfs_costo(Estados,Movs,_),
   length(Movs,6),
   last(Estados,estado(d,d,d,d)).

Examples History Solutions table results Run!
```

Siendo estas las pruebas que realizamos.

### SUSTENTACIÓN AQUÍ ABAJO (Implementación DFS)

Para la sustentación realizamos el siguiente código para implementar DFS

*% Sustentación de La implementación de DFS*

*% Resolver el camino con dfs*

```
resolver_dfs(CaminoEstados, CaminoMovs) :-
    estado_inicial(E0),
    dfs(E0, [E0], [], CamEstRev, CamMovRev),
    reverse(CamEstRev, CaminoEstados),
    reverse(CamMovRev, CaminoMovs).
```

*% si Estado ya es final, devolvemos el camino que armamos (en reversa)*

```
dfs(E, Visitados, MovsRev, Visitados, MovsRev) :-
    estado_final(E), !.
```

*% Paso recursivo: tomamos un sucesor E2 mediante un movimiento "Accion"*

```
dfs(E, Visitados, MovsRev, CamEstRev, CamMovRev) :-
    mover(E, E2, Accion),
    \+ member(E2, Visitados), % Hacemos esto para evitar ciclos.
    dfs(E2, [E2|Visitados], [Accion|MovsRev], CamEstRev, CamMovRev).
```

Para realizarlo, nos basamos en la solución de la primera respuesta de este thread de StackOverflow: [Depth First Search Algorithm Prolog - Stack Overflow](#), no hay mucho para explicar, esta es la versión clásica de DFS, solo la implementamos y reutilizamos el mismo modelo de estados que ya teníamos (estado\_inicial/1, estado\_final/1, inseguro/1, seguro/1 y mover/3).

Para entrar un poco más a detalle en su funcionamiento, nos queda que:

- resolver\_dfs/2: inicia la búsqueda desde el estado inicial. Llama a dfs/5 y luego invierte los caminos con reverse/2 para devolverlos en orden desde el inicio hasta la meta (misma razón de eficiencia que en la versión de amplitud con costo: agregamos por cabeza,  $O(1)$ , y al final invertimos una sola vez).
- dfs/5 (caso meta): si el estado actual ya es el estado\_final, se detiene y devuelve el camino acumulado (en reversa). El corte (!) evita seguir buscando más alternativas, porque ya encontramos una solución por esta rama.
- dfs/5 (paso recursivo): selecciona un sucesor E2 aplicando un operador mover/3, evitamos ciclos como en la solución que vimos en Stack Overflow con member(E2, Visitados), acumula el estado y la acción por cabeza ([E2|Visitados], [Accion|MovsRev]) y sigue profundizando por esa rama hasta llegar a su final.

Para probarlo creamos un comando igual al de la primera prueba pero que resuelve con DFS, no hay mucho que explicar aquí (Ya explicamos el funcionamiento de **prueba** antes):

*% Prueba para DFS*

**prueba\_dfs** :-resolver\_dfs(Estados, Movs), imprimir\_solucion(Estados, Movs).

El cual nos da el siguiente output:

```

prueba_dfs.
Estados
estado(i,i,i,i)
estado(d,i,d,i)
estado(i,i,d,i)
estado(d,d,d,i)
estado(i,d,i,i)
estado(d,d,i,d)
estado(i,d,i,d)
estado(d,d,d,d)
Movimientos
m3_cabra
m1_solo
m2_lobo
m3_cabra
m4_heno
m1_solo
m3_cabra
true

```

Next 10 100 1,000 Stop

Por lo que, funciona bien!!

**Referencias:**

1. [Lists in Prolog - GeeksforGeeks](#)
2. [Creating a queue structure in Prolog - Stack Overflow](#)
3. [SWI-Prolog -- \(\+\)/1](#)
4. [Prolog - Search](#) (Subido al Campus Virtual)
5. [Depth First Search Algorithm Prolog - Stack Overflow](#) (Base para el DFS de la sustentación)