BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

PLANEACIÓN DECLARATIVA CLÁSICA CON EL LENGUAJE DE ACCIÓN K

TESIS

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

C. MARÍA JOSÉ ÁNGELES LÓPEZ

DIRECTOR Y ASESOR DE TESIS:

DR. FERNANDO ZACARÍAS FLORES

Puebla, Pue. Octubre 2019 Otoño

Dedico este trabajo principalmente a mi abuelita, a pesar de la distancia física siento que estás conmigo siempre, nos faltaron muchas cosas por compartir en especial este último paso, trato de ser mejor cada día para que te sientas más orgullosa de lo que sé que estas ahora.

A mis papás, hermanos, abuelito, tíos y tías, primos, sobrinos y demás familia que me alientan y me apoyan en las decisiones que tomo, creo que de no ser por eso este trabajo no sería posible.

Primero gracias a Dios por haberme dado la vida y permitirme el haber llegado hasta este momento tan importante de mi formación profesional.

A mis abuelitos Raúl y Jose por todas sus enseñanzas y lecciones de vida, espero algún día tener el amor que ustedes tienen, son un ejemplo a seguir.

A mis papás Miguel Ángel y Rosario por costear mis estudios y darme esta valiosa oportunidad que ustedes no pudieron tener pero que espero que ese sueño lo cumplan conmigo y mis hermanos.

A mi tía Cony, a quien quiero como a una madre, por siempre estar dispuesta a escucharme y ayudarme en cualquier momento, eres también parte fundamental para que esto se lograra.

A mis hermanos Juan e Isabela por ser mis compañeros de vida, ustedes son el mejor regalo que me pudieron dar.

A mis tías Lety, More, a mis tíos, a mis primos Sebas, Omar, Yuliana, a mi ahijada Monse; tal vez me falten por mencionar, pero agradezco que están conmigo siempre, me han visto crecer a lo largo de estos casi 25 años, gracias por compartir mis logros, mis tropiezos y todo lo que lleva esta vida, gracias por darme palabras de aliento cuando más las necesité; gracias por seguir aquí.

Gracias infinitas a cada uno de ustedes espero algún día poder devolver un poco de lo que me han dado, no tienen idea de lo importante que son en mi vida y el inmenso amor que le tengo cada uno.

A mi compañera de universidad, estrés, pero sobre todo más risas, Bety, ¡si se pudo!

Al Dr. Fernando Zacarías por ser director de este trabajo de tesis, gracias por sus consejos y sugerencias para que este trabajo saliera adelante.

Índice

Capítulo 1

Introducción

Cuando planeas, planeas tener éxito. Por lo tanto, no es una sorpresa que la planificación eventualmente te lleve a tener éxito y lograr una mejor posición dentro de su entorno. Cualesquiera que sean sus objetivos, con una planificación correcta puede tener más posibilidades de éxito. La planeación es una de las áreas más importantes en la vida e incluso en muchos ámbitos, tanto en empresas, organizaciones e incluso en la vida cotidiana de un individuo. Si planea tener éxito, el plan incluirá su progreso, así como el mejor camino para lograr el objetivo. La gestión de la organización siempre mantiene los objetivos por sí mismos que quiere lograr. Estos objetivos, y la planificación, siempre conducirán al progreso de la organización. El progreso de la organización es la razón por la cual la planificación es importante para la administración. La importancia de la planificación aumenta en una organización donde no se ha observado estabilidad.

Así, en el presente trabajo se aborda el problema de planeación desde la perspectiva de la planeación clásica declarativa basada en la programación lógica a través de juegos de lógica. Analizaremos como llevar a cabo la representación de conocimiento, así como el razonamiento que nos permita poder modelar y programar problemas de planeación clásica. Para esto, se revisará el estado del arte con la finalidad de justificar el lenguaje elegido para el modelado y solución de problemas de planeación. Las técnicas de planificación se aplican en una variedad de tareas que incluyen robótica, planificación de procesos, recopilación de información basada en internet, agentes autónomos, control de misiones de naves espaciales, etc [1].

Finalmente, en esta investigación analizaremos y modelaremos algunos problemas de planeación que permitan sentar las bases para en trabajos futuros se puedan resolver problemas en otras áreas de aplicación como las mencionadas previamente. De esta manera contribuiremos en el desarrollo de metodologías actuales y con una mejor eficiencia en la solución de muchos problemas de la inteligencia artificial en general. La planificación es una habilidad clave para los sistemas inteligentes, incrementando su autonomía y flexibilidad. Cuando pensamos en sistemas inteligentes es común pensar

en hacer uso de agentes inteligentes que nos permitan modelar problemas en la planificación [2].

1.1 Antecedentes del Proyecto

Los sistemas de planeación existentes hasta hace unos años han sido desarrollados en lenguajes como STRIPS y PDDL [3, 4]. Sin embargo, en los últimos años, se ha visto el desarrollo de lenguajes de acción que proveen herramientas flexibles y expresivas para describir la relación entre fluentes y acciones. Este tipo de lenguajes han recibido considerable atención entre los investigadores de la comunidad de razonamiento y representación de conocimiento. El estudio de este tipo de lenguajes ha sido profundo y se han desarrollado lenguajes que han mejorado en mucho el modelado de problemas de planeación en diversas áreas del conocimiento. En este trabajo de tesis, analizaremos la eficiencia y expresividad en la representación y el razonamiento del conocimiento con lenguajes de acción que son significativamente diferentes de los marcos estrictos basados en operadores de STRIPS y PDDL.

Particularmente, analizaremos Smodels y DLV^K [5, 6], dos de los lenguajes de acción que han mostrado ser excelentes lenguajes para el diseño y modelado de problemas de planeación en los últimos años.

Smodels es un sistema que incluye dos módulos que definen de manera clara y eficiente un lenguaje de acción. i) un algoritmo para implementar la semántica de modelos estables para programas base, y ii) un algoritmo para calcular una versión con conexión a un programa normal base sin funciones de rango restringido. Estas implementaciones son capaces de resolver un rango de problemas computacionales relacionados a programas normales. Este puede calcular el modelo bien fundado de un programa. Es capaz de decidir si un programa tiene un modelo estable. Puede generar todos o un número dado de modelos estables de un programa. Es capaz de decidir si una literal dada es satisfecha en alguno o en todos los modelos estables de un programa. Este lenguaje ha demostrado ser eficiente en áreas cubiertas por la programación lógica, problemas combinatorios de grafos, diagnósticos de circuitos y satisfactibilidad proposicional.

Además, Smodels ha puesto un énfasis especial en desarrollar una implementación de la semántica de modelos estables que puede ser usada en aplicaciones reales. Smodels tiene una novedosa técnica de implementación que conduce a una complejidad de espacio lineal que le ha llevado a ser una herramienta competitiva cuando es comparado con otras propuestas existentes.

Por otro lado, DLV^K es una novedosa propuesta de un lenguaje de planeación basado en lógica llamado *K*. En esta propuesta, las transiciones entre los estados de conocimiento pueden ser descritos claramente. Un estado es caracterizado por los valores de verdad de un número de fluentes, describiendo propiedades relevantes del dominio del discurso. Además, permite hacer planeación bajo conocimiento incompleto, aunque, también permite la representación de transición entre estados de conocimiento completo. Así, DLV^K permite resolver problemas de planeación complejos y difíciles, incluyendo planeación segura bajo estados iniciales incompletos. Lo cual no puede ser resuelto por los sistemas de planeación basados en lógica tales como planeadores de satisfacibilidad.

1.2 Objetivo General y Objetivos Específicos

Modelar e implementar soluciones de planeación a través del lenguaje de acción llamado DLV^K.

Objetivos específicos:

- Analizar la representación de conocimiento en el lenguaje de acción *K*.
- Experimentar con ejemplos sencillos de planeación.
- Determinar las principales características de representación de conocimiento así, como la metodología de razonamiento en el lenguaje de acción *K*.
- Establecer una metodología para el modelado de problemas de planeación.

1.3 Metodología

Se revisará la literatura que sobre el estado del arte existe para identificar aquellos estándares pertinentes y útiles para el modelado de problemas de planeación en lenguajes de acción. Posteriormente, se estudiarán y analizarán los diferentes sistemas existentes, de tal forma que podamos justificar nuestra elección del lenguaje de acción

para el modelado de problemas de planeación clásica. Además, se determinarán los requerimientos necesarios para la representación de conocimiento, considerando la abstracción necesaria del entorno a modelar. Por otro lado, se instalará el lenguaje de acción elegido para el modelado de los problemas de planeación.
Por otro lado, para la evaluación del sistema, utilizaremos diversas configuraciones que nos permitan validar la robustez de nuestra solución, así como la flexibilidad y generalidad de nuestra solución.

Finalmente, se caracterizarán las principales propiedades de cada uno de los elementos que conforman la solución hallada. Así como, plantear trabajo a futuro basado en nuestra implementación.

Para esto, la metodología empleada consistirá en las fases siguientes:

1. **Definición del problema:** Es decir, en esta primera fase nos interesa mostrar cómo debemos determinar cuál es la representación más adecuada del entorno en que se presenta el problema para poder hacer una correcta abstracción de este.
2. **Determinación de objetos:** incluye como determinar que objetos son los mínimos necesarios para modelar correctamente nuestro problema.
3. **Determinación de Fluentes:** Identificar la mejor manera de caracterizar los diversos escenarios como el estado inicial, la configuración final, así como los escenarios intermedios o parciales alcanzados con la ejecución de las diversas acciones que modifican un estado.
4. **Análisis y Determinación de Acciones:** Definir las acciones necesarias para la solución del problema, así como los diversos efectos que las acciones provocan y que deben ser consideradas.

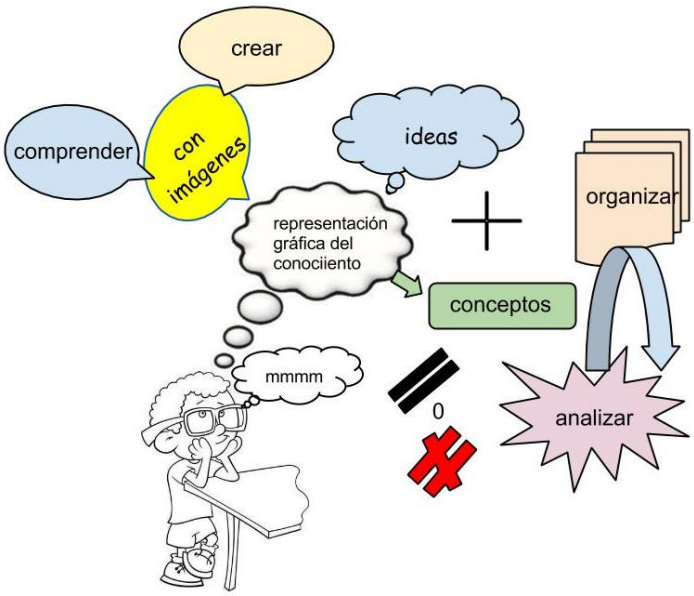


Figura 1.1 Procesos mentales para determinar la representación de conocimiento

En la figura 1.1 se muestra cómo podemos a través de herramientas que nos permiten representar el conocimiento, mediante símbolos que producen un impacto visual, con el objeto de ayudar, al trabajar con ideas y conceptos, a pensar más efectivamente, jerarquizando y simplificando la información, además de captar, comprender y representar la realidad. Para poder utilizar las herramientas de la construcción simbólica del conocimiento se deben unir con "imaginería"; como lo señala Chan [3], con una serie de lenguajes y códigos bajo diversos entornos audiovisuales, gráficos e informáticos (mapas conceptuales, mapas mentales, línea de tiempo, organigrama, esquemas, etc.).

1.4 Justificación

El problema de planeación ha sido abordado recientemente desde la programación basada en el cálculo de modelos por diversos investigadores [5], [6] y [7]. En particular, las reglas disyuntivas son empleadas para buscar una solución candidata, las cuales son entonces checadas por su validez. Es importante señalar que muchas de las propuestas existentes en esta área buscan soluciones de manera exhaustiva o ingenua. Son pocos los sistemas que desarrollan algoritmos eficientes para entrelazar las fases de búsqueda y chequeo de soluciones tan buenas como sea posible, tal que la generación de candidatas pueda ser podada efectivamente [8], [9].

El lenguaje STRIPS fue un buen punto de partida para la planificación de la representación de problemas, pero hubo espacio para mejoras. ADL (Action Description Language) es una de las extensiones de STRIPS que eliminó algunas de sus restricciones para manejar problemas más cercanos a la realidad. A diferencia de STRIPS, ADL no asume que los literales no mencionados son falsos, sino más bien desconocidos, lo que se conoce mejor como Suposición del mundo abierto. También admite literales negativos, variables cuantificadas en objetivos (por ejemplo, Atx At (P1, x) ∧ At (P2, x)), efectos condicionales y disyunciones en objetivos (todo esto no está permitido en STRIPS).

STRIPS y ADL fueron la inspiración para otra extensión de lenguajes de representación: PDDL (Planning Domain Definition Language). Fue un intento de estandarizar los lenguajes de planificación lo que hizo posible la serie del Concurso Internacional de Planificación (IPC). En otras palabras, PDDL contiene STRIPS, ADL y muchos más lenguajes de representación.

Gracias a un lenguaje común, la competencia de planificación puede comparar el rendimiento de los sistemas de planificación utilizando un conjunto de problemas de referencia. Lo más importante es que al tener un estándar formal, podemos comparar sistemas y enfoques, pero también acelerar el progreso en ese campo. Un formalismo común es un compromiso entre el poder expresivo y el progreso de la investigación básica (que fomenta el desarrollo a partir de fundamentos bien entendidos). Considerando que la planeación está profundamente arraigada en nuestras vidas (robots de fábrica, aspiradoras inteligentes, agentes de planificación y más) y su desarrollo da forma a nuestro futuro (automóviles autónomos, aviones no tripulados), definitivamente no se debe ignorar [9]. Finalmente, podemos concluir que estos esfuerzos han desembocado en lenguajes formales basados en programación lógica tales como DLV^{K}, Smodels, ASSAT, etc.

Este tipo de desarrollos nos brindan el contar con problemas muy cercanos a la realidad modelados a través de lenguajes proposicionales. Esto contribuye de manera directa a detonar el amor e involucramiento en el área de la programación lógica, una de las áreas que en nuestro plan de estudios se contempla. Con esta propuesta lograremos llevar el uso de la programación lógica al modelado de problemas de planeación. Y poder

considerar este tipo de propuestas en áreas como la robótica o el desarrollo de agentes inteligentes.

El sistema permitirá caracterizar como se modela la planeación clásica desde lenguajes basados en la programación lógica y por lo tanto tendremos aportaciones relevantes.

1.5 Narrativa de Capítulos

Esta tesis se compone de cuatro capítulos. En el primer capítulo se presenta la introducción donde se da a conocer la importancia de la planeación para poder alcanzar el objetivo y tener éxito en todo lo que se propone, objetivos tanto general como los específicos, la metodología, los antecedentes y la justificación. El capítulo dos presenta el paradigma *Answer Set Programming* basado en lógica y el software que ha implementado, así como definiciones y ejemplos. En el tercer capítulo se aborda a DLV (*Disjunctive Datalog System*) sistema de base de datos deductivo donde se implementa el paradigma ASP para la representación de conocimiento, elementos básicos de DLV y ejemplos de programas realizados con este sistema. En el capítulo cuatro se explica la sintaxis de DLV^K con ayuda de un ejemplo y así poder llegar al problema central de esta tesis que es planeación a través del problema "La travesía del puente", cuáles fueron los pasos que se siguieron para el desarrollo del mismo y el resultado final de la implementación. Después de los capítulos se presenta el trabajo a futuro y por último las conclusiones a las que se llegaron en esta tesis.

Capítulo 2

Answer Set Programming (ASP)

El propósito de este capítulo es el de introducir al lector en uno de los nuevos paradigmas que más éxito ha tenido en los últimos años, *Answer Set Programming* o *A-Prolog*. Este paradigma basado en programación lógica y particularmente en el cálculo de respuestas provee un *backbone* computacional que ha permitido el desarrollo de poderosas implementaciones de software tales como: DLV, Smodels, ASSAT y Cmodels, entre otros [10]. Para el caso proposicional disyuntivo, DLV provee un agradable panorama general acerca de la complejidad del tipo de problemas que resuelve.

Platform			Features					Mechanics
Name	OS	Licence	Variables	Function symbols	Explicit sets	Explicit lists	Disjunctive (choice rules) support	
ASPeRiX	Linux	GPL	Yes				No	on-the-fly grounding
ASSAT	Solaris	Freeware						SAT-solver based
Clasp Answer Set Solver	Linux, macOS, Windows	MIT License	Yes, in Clingo	Yes	No	No	Yes	incremental, SAT-solver inspired (nogood, conflict-driven)
Cmodels	Linux, Solaris	GPL	Requires grounding				Yes	incremental, SAT-solver inspired (nogood, conflict-driven)
DLV	Linux, macOS, Windows[15]	free for academic and non-commercial educational use, and for non-profit organizations[15]	Yes	Yes	No	No	Yes	not Lparse compatible
DLV-Complex	Linux, macOS, Windows	GPL		Yes	Yes	Yes	Yes	built on top of DLV — not Lparse compatible
GnT	Linux	GPL	Requires grounding				Yes	built on top of smodels
nomore++	Linux	GPL						combined literal+rule-based
Platypus	Linux, Solaris, Windows	GPL						distributed, multi-threaded nomore++, smodels
Pbmodels	Linux	?						pseudo-boolean solver based
Smodels	Linux, macOS, Windows	GPL	Requires grounding	No	No	No	No	
Smodels-cc	Linux	?	Requires grounding					SAT-solver based; smodels w/conflict clauses
Sup	Linux	?						SAT-solver based

Figura 2.1 Software desarrollados para el cálculo de *answer sets* (modelos)

Como se muestra en la figura 2.1 ASSAT es un software de uso libre, trabaja en sistema operativo Solaris y utiliza solucionadores SAT (Problema de satisfacibilidad booleana). Cmodels es un software con licencia pública general de GNU, trabaja en los sistemas operativos Linux y Solaris, requiere de otro paquete para hacer las instanciaciones base, su mecánica es incremental y también utiliza solucionadores SAT. DLV es un software un poco más completo que los que se mencionan anteriormente, ya que tiene variables,

símbolos de función, soporte disyuntivo, pero no es compatible con el front-end Lparse; DLV es un software gratuito y trabaja en los sistemas operativos Linux, macOS y Windows. Smodels necesita de otro paquete para hacer las instanciaciones base, también es software con licencia pública general de GNU y trabaja en los sistemas operativos Linux, macOS y Windows.

2.1 Panorama ASP

Answer set programming (ASP) es un paradigma lógico que en los últimos años ha tenido un gran desarrollo [11, 12, 13, 14]. Además, se han obtenido resultados relevantes que establecen relaciones directas entre ASP y la lógica intuicionista y otras lógicas. Esto ha dado a este paradigma un gran impulso a su uso entre los miembros de la comunidad científica. También, han surgido diversas aplicaciones en el campo de la Inteligencia Artificial. En este capítulo se da un panorama general sobre ASP con la finalidad de establecer el marco teórico empleado en este trabajo.

Se considera que una teoría está compuesta por tres elementos fundamentalmente que son: Primero, el conjunto de axiomas lógicos (correspondientes a los axiomas que conforman la lógica elegida). Segundo, el conjunto de axiomas propios (en este contexto lo conforman el conjunto de cláusulas comprendidas en un programa lógico). Tercero, las reglas de inferencia empleadas para su aplicación en la obtención de nuevo conocimiento (en este caso la regla de *modus ponens*). Más adelante se define de manera formal estos tres ingredientes que son la parte fundamental del conocimiento básico necesario para entender este trabajo. Enseguida, se dan algunos conceptos y definiciones que serán utilizados a lo largo del mismo.

2.2 Lógica proposicional

El lenguaje de la lógica proposicional tiene un alfabeto que consiste de:

- Símbolos proposicionales p0, p1, ...
- Conectivos $\wedge$, $\vee$, $\rightarrow$, $\perp$
- Símbolos auxiliares “(”, “) ”, “ , ” , “ . ”

Donde $\wedge,\vee,\rightarrow$, son conectivos binarios y $\perp$ es un conectivo sin argumentos. También, el conectivo $\perp$ denota falso. Los símbolos proposicionales son también llamados átomos o proposiciones atómicas.

Una fórmula T (también llamada *Top*) es introducida como una abreviación de $\perp \leftarrow \perp$. La negación clásica es representada con $\neg$ (también conocida como "not"). Se dice que un programa es un conjunto de cláusulas tomando en consideración la siguiente convención:

$A \leftarrow B$ es otra forma de escribir $B \rightarrow A$.

Se denota una cláusula de la siguiente manera:

$$H \leftarrow B$$

Refiriéndose a H como la cabeza (*Head*) y a B como el cuerpo (*Body*).

2.3 Tipos de cláusulas

Enseguida se definen los diferentes tipos de cláusulas que se puede encontrar en la literatura [13].

- Si $H = A$ *y* $B = B = B_1, B_2,..., Bk, Bi, Bi$ letra proposicional entonces $H \leftarrow B$ se le denomina cláusula positiva.
- Si $H = \perp$ y $B = B_1, B2,..., Bk, Bi, Bi$ letra proposicional entonces $H \leftarrow B$ se le denomina cláusula Horn.
- Si $H \leftarrow B$ donde $H = A$, $B = B_1, B2,..., Bk, \neg B_{k+1},..., \neg B_m$ le denominamos cláusula normal.
- Si tenemos $H \leftarrow B$ con $H = A_1 \vee ... \vee A_n$, $B = B_1, B_2,..., Bk, \neg B_{k+1},..., \neg B_m$ le llamamos cláusula disyuntiva.
- Si tenemos $H \leftarrow B$ donde $H = A_1 \vee ... \vee A_n, \neg A_{n+1} \vee ... \vee \neg A_{n+m}$,
  $B = B_1, B_2, ..., B_k, \neg B_{k+1},..., \neg B_{k+m}$ le denominamos clausula libre.
- Si $H \leftarrow B$, con H y B anidaciones de $\vee,\wedge$ y $\neg$, es una cláusula aumentada.

En la figura 2.2 se muestra cómo se conciben los diferentes tipos de cláusulas descritas anteriormente. Como se puede observar, existen diversos tipos de reglas, sin embargo, a nivel de software, es complejo poder aceptar todos estos tipos de reglas. La mayoría de software que existen aceptan solo hasta el nivel de las cláusulas disyuntivas, este es el caso del software que se utilizó, DLV. Algunos otros, aceptan todos los tipos de cláusulas, sin embargo, primero transforman las cláusulas libres y aumentadas a sus equivalentes cláusulas disyuntivas.

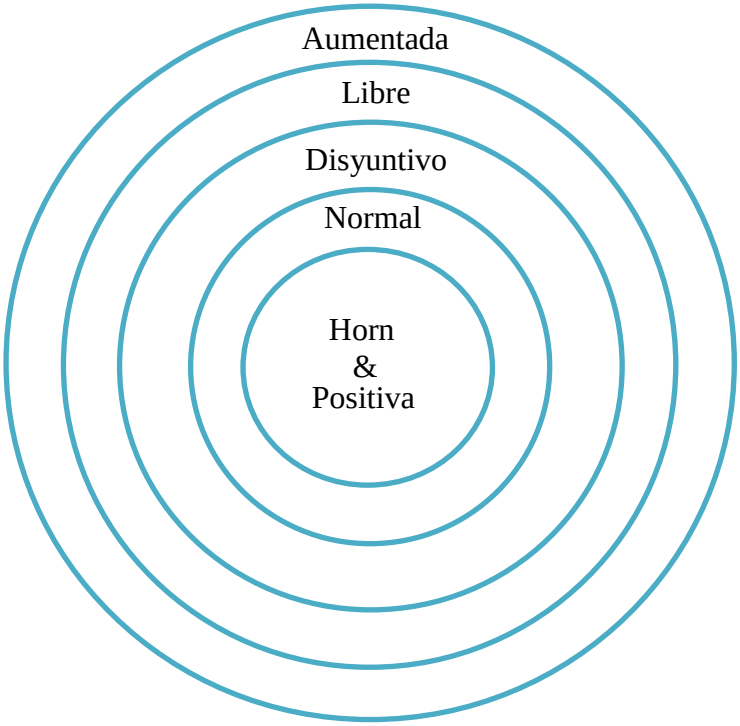


Figura 2.2 Tipos de cláusulas lógicas gráficamente

Cada uno de estos tipos de cláusulas define una clase de programa [13], por ejemplo:

$a \vee b \leftarrow e \wedge d \wedge \neg c$	Disyuntiva, Libre
$\bot \leftarrow p \vee q.$	Restricción (a)

Existen algunos tipos de cláusulas especiales tales como las siguientes [13]:

Si $B_1, B_2,..., Bk, \neg B_{k+1},..., \neg B_{k+m}$ es T entonces se identifica la regla con A y le llamamos hecho.

Si A es $\perp$ entonces decimos que la regla es una restricción. Tal como la mostrada en (a).

Una literal L es un átomo A o un átomo fuertemente negado -A.

Así, un programa P es un conjunto finito de reglas normales, donde A y cada *Bi* son literales.
A es llamada la cabeza de un regla *r*, denotada por *H(r)*, y a B_1, B_2,..., Bk,$\neg B_{k+1}$,..., $\neg B_{k+m}$ le llamamos el cuerpo de la regla *r* y lo denotamos por *B(r)*.

2.4 Conceptos teóricos de ASP

Dentro del paradigma ASP se considera que una teoría se compone de tres elementos fundamentalmente [13] como lo son:

1. Conjunto de axiomas lógicos, que normalmente son definidos de manera concreta para cada sistema axiomático existente.
2. El conjunto de axiomas propios, que en este caso lo constituye el programa lógico representado en DLV.
3. El tercer y último componente lo constituye la regla de inferencia llamada *modus ponens*.

Axiomas lógicos. - Existen diferentes sistemas axiomáticos.
Axiomas propios. - Pueden ser las cláusulas de un programa.
Regla de inferencia. - Modus Ponens.

1) $A \rightarrow B$ Proposición verdadera (Teorema)
2) A Proposición verdadera (Teorema)
3) B Proposición

Esta regla significa que si se tiene como hipótesis A y $A \rightarrow B$, ambas verdaderas, entonces se puede concluir B verdadera.

Véase un ejemplo de lo que sería una teoría en este marco de trabajo.

Ejemplo 2.1 Supóngase que P es un programa que solo contiene átomos positivos en el cuerpo, *A* y *B* letras proposicionales, entonces:

$P = \{ A_1 \leftarrow B_{11},\ldots, B_{1n}.$

...

...

...

$A_k \leftarrow B_{k1},..., B_{kn}.\}$

Considérese a la lógica clásica como el contexto teórico, por tanto, los axiomas lógicos son:

Axioma 1)	$A \rightarrow (B \rightarrow A)$	A1
Axioma 2)	$(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$	A2
Axioma 3)	$(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$	A3

Y como axiomas propios las cláusulas en el programa *P*:

$A_1 \leftarrow B_{11}, ..., B_{1n}.$

...

...

...

$A_k \leftarrow B_{k1}, ..., B_{kn}.$

Como única regla de inferencia el *modus ponens*. Así, se obtiene esta teoría.

Enseguida, se definen algunos de los conceptos teóricos empleados en el desarrollo de este trabajo.

2.5 Definiciones básicas

Definición 2.1 FORM (conjunto de fórmulas bien formadas) [13, 14]:
A es una fórmula bien formada (fbf):

Si *A* y *B* son fórmulas bien formadas, así también lo son:

$$A \vee B,\ A \wedge B, \qquad A \leftarrow B, \qquad A \leftrightarrow B, \qquad A \rightarrow \bot$$

Definición 2.2 Dado σ un conjunto de cláusulas (fbf) se dice que existe una deducción de *A* supuesto σ, notación $\sigma \vdash A$ si existe $B_1, B_2, ..., B_n$ fórmulas bien formadas tales que para cada $i \in \{1, ..., n\}$, B_i es un axioma (A1, A2, A3) o $B_i \in \sigma$ o existe B_k [13]

$B_k :=$ $\dfrac{B_j \rightarrow B_i}{B_j}$

Entonces B_i. Donde k, j < i.

Definición 2.3 Toda letra proposicional, ⊥ y T, (T := ⊥ → ⊥) son denominadas fórmulas elementales. Cualquier fórmula construida con ∧ y ∨ sobre fórmulas elementales es denominada fórmula básica [13].

Definición 2.4 Un conjunto de letras proposicionales *X* satisface una fórmula básica *F* ($X \models F$) si:

Para *F* fórmula elemental, $X \models F$ si $F \in X$ o F = T [11].

$X \models F \vee G$ si $X \models F$ y $X \models G$

$X \models F \wedge G$ si $X \models F$ o $X \models G$

Definición 2.5 Dado *P* un programa básico constituido por fórmulas básicas. Un conjunto de letras proposicionales *X* se dice cerrado bajo *P* si para toda cláusula $H \leftarrow B \in$ P tenemos que $X \models H$ siempre y cuando $X \models B$ esto es, si $X \models B$ entonces $X \models H$ [13].

Definición 2.6 Sea *X* un conjunto de letras proposicionales y *P* un programa constituido por fórmulas básicas, *X* es denominado un *Answer Set* de *P* si el conjunto *X* es mínimo entre los conjuntos de átomos cerrados bajo *P* [13].

Definición 2.7 (*Answer Set*) La reducción de un programa normal (disyuntivo, libre, aumentado) relativo al conjunto de letras proposicionales *X*, es definido de la siguiente manera:

Para una formula elemental *F* [13]:

$F^x = F$

$(F \wedge G)^x = F^x \wedge G^x$

$(F \vee G)^x = F^x \vee G^x$

$(\neg F)^x = \bot$ si $X \models F$; T en otro caso.

$P^x = (H \leftarrow B)^x$ tal que $H \leftarrow B \in P$

Definición 2.8 Sea *P* un programa y *X* un conjunto de átomos. *X* es un *Answer Set* de *P* si este es un *Answer Set* de la reducción P^x [13].

Sea *P* un programa, $M \subseteq L_P$, M *Answer Set* de *P* y L_P un conjunto de átomos:
$M \cup \neg\{L_P - M\}$ es el modelo de *P* [13].

Definición 2.9 Dado un conjunto de literales *A* y *P* un programa. Denotamos $\neg A = \{\neg a \mid a \in A\}$. También se define $A^* = A \cup \neg\{Lit_P - M\}$ [13].

Definición 2.10 Decimos que dos reglas r_1 y r_2 están en conflicto (denotado por $r_1 \nabla \sim r_2$) sí y sólo si ambas tienen la misma cabeza, pero una es la contrapuesta de la otra, es decir, sí $H(r_1) = -H(r_2)$ [13].

Definición 2.11 (Principio de rechazo causal). Sean P_1 y P_2 programas y sea *S Answer Set* de la actualización de P_1 y P_2. Definimos y denotamos al conjunto de reglas a rechazar como:

$Rej(S, (P_1, P_2)) = \{r \in P_1 \mid \exists\ r' \in P_2$, tal que $r \nabla r'$ y $S \models B(r) \cup B(r')\}$ [13]

2.6 Cálculo de *answer sets*

Un avance importante en el área de razonamiento no-monótono fue el descubrimiento de una fuerte conexión entre la lógica predeterminada y la programación lógica (LP) que permitió leer el operador *Prolog* de la negación como fallo como una formalización del razonamiento no-monótono de la negación predeterminada

Esto abrió la posibilidad de reinterpretar la lógica. Los programas como formalismos de representación de conocimiento con razonamiento no-monótono incorporado, conducen a una fertilización cruzada entre ambas áreas.

Entre las diferentes semánticas de programación lógica propuestas para la negación como falla, una de ellas ha tenido un impacto notable en la aplicación práctica del razonamiento no-monótono. En [14], Michael Gelfond y Vladimir Lifschitz introdujeron la llamada semántica del modelo estable que subsumía muchas de las alternativas de LP existentes, pero muy importante, sin las restricciones sintácticas establecidas por los enfoques anteriores. La orientación basada en modelos de esta semántica eventualmente llevó a un paradigma adecuado para problemas de satisfacción de restricciones que se conoce hoy en día como el conjunto de respuestas (ASP) [14] y se convirtió en uno de los enfoques más destacados y exitosos para representación de conocimiento práctico. En la figura 2.3 se muestra la configuración típica del uso de solucionadores de *Answer Sets* en el desarrollo de aplicaciones para modelar problemas reales.

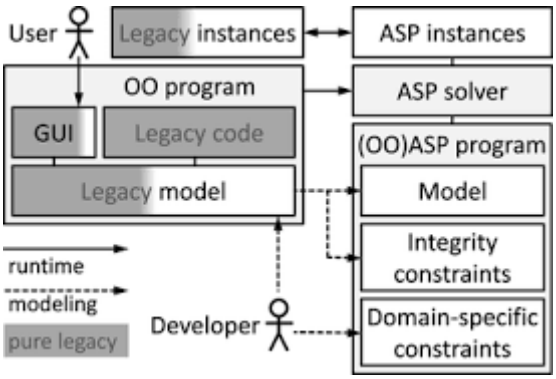


Figura 2.3 Uso general de *Answer Sets* en desarrollo de aplicaciones

Gran parte de este éxito se debe a los impresionantes avances en la implementación de solucionadores eficientes [12, 13, 14] y el desarrollo de aplicaciones.

2.1 Ejemplos de cálculo de *answer sets*

Enseguida, se dan algunos ejemplos que muestran la forma en que se aplica la reducción anterior a las reglas de un programa para calcular sus *Answer Sets*.

Ejemplo 2.2 Considérese el siguiente ejemplo.

Sea P: $a \leftarrow \neg \neg a$.

$\neg b \leftarrow c \vee b$.

Propóngase que el *Answer Set* sea X = ∅ entonces, aplicando la reducción al programa *P* para verificar si lo satisface y aplicando la definición 2.7 a P se obtiene:

P^X: $a^X \leftarrow \neg\neg a^X$

$\neg b^X \leftarrow (c \vee b)^X$

P^X: $a \leftarrow T$

$T \leftarrow c \vee b$

Así, X = ∅ no satisface a P^X por lo tanto X no es un A*nswer Set* de *P*

Si X = {*a*}

P^X: $a^X \leftarrow \neg \neg a^X$

$\neg b^X \leftarrow (c \vee b)$ *

P^X: $a^X \leftarrow T$

$T \leftarrow c \vee b$

Por lo tanto, X es un A*nswer Set* de P.

Ejemplo 2.3 Ahora, considere el siguiente ejemplo.

P: $a \leftarrow \neg a$

Supongamos que X = ∅, aplicando la definición 2.7 a P:

P^X: $a^X \leftarrow (\neg a)^X$

Del cual resulta el programa siguiente:

P^X: $a \leftarrow T$

Por lo tanto, X = ∅ no es un A*nswer Set* de *P*.

Ejemplo 2.4 Considere el siguiente ejemplo.

Sea P: $a \leftarrow \neg b$

$b \leftarrow \neg a$

Sea X = ∅, aplicando la definición 2.7 calculemos el nuevo programa P^X

P^X: $a \leftarrow$ T

$b \leftarrow$ T

Por lo tanto, X = ∅ no es un A*nswer Set* de *P*

Ahora, probemos con X = {a}

P^X: $a \leftarrow$ T

$b \leftarrow \perp$

por lo tanto, X = {a} es un answer set de P

Sea X = {b}, véase si este es un *Answer Set* del programa

P^X: $a \leftarrow \perp$

$b \leftarrow$ T

por lo tanto, X = {b} también es un A*nswer Set* de *P*

Ejemplo 2.5 Considere el siguiente ejemplo.

Sea *P*: $a \leftarrow \neg b$

$b \leftarrow \neg a$

$p \leftarrow \neg p$

$p \leftarrow \neg a$

Sea X = ∅, trataremos de ver si este es un A*nswer Set* de *P* aplicando la definición 2.7:

P^X: $a \leftarrow$ T

$b \leftarrow$ T

$b \leftarrow$ T

$b \leftarrow a$

por lo tanto, X = ∅ no es un *Answer Set* de *P*.

Ahora, sea X = {*a*}

P^x: $a \leftarrow$ T

$b \leftarrow \perp$

$b \leftarrow$T

$b \leftarrow$ a

por lo tanto, X = {*a*} no es un A*nswer Set* de *P*.

Ahora veamos si X = {*b*} es un A*nswer Set* de *P*

P^x: $a \leftarrow \perp$

$b \leftarrow$ T

$b \leftarrow$ T

$b \leftarrow a$

por lo tanto, X = {*b*} no es un A*nswer Set* de *P*

Sea X = {*p*}

P^x: $a \leftarrow$T

$b \leftarrow$ T

$b \leftarrow \perp$

$b \leftarrow a$

por lo tanto, X = {*p*} no es un A*nswer Set* de *P*

Sea X = {*p, a*} y busquemos probar que es *Answer Set* de P

P^x: $a \leftarrow$ T

$b \leftarrow \perp$

$b \leftarrow \perp$

$b \leftarrow a$

por lo tanto, X = {*p, a*} si es un *answer set* de *P*

Finalmente, veamos si X = {*p, b*} también es *Answer Set* de *P*

P^x: $a \leftarrow \perp$

$b \leftarrow$ T

$b \leftarrow \bot$

$b \leftarrow a$

por lo tanto, X = {*p, b*} no es un *Answer Set* de *P*

Ejemplo 2.6 Considere el siguiente ejemplo.

Sea *P*: $\neg b \leftarrow \neg a$

Veamos si X = ∅ es un *Answer Set* de *P* y aplicando la definición 2.7:

P^X: T ← T

por lo tanto, X = ∅ es un modelo (*answer set*) de *P*.

2.7 Intuicionismo

La lógica intuicionista tiene características excelentes para el desarrollo de la teoría de A*nswer Set Programming*. De la misma forma es la base para desarrollar teorías que son usadas en la caracterización de programas lógicos y transformaciones de estas [13].

Intuicionismo está basado en un concepto de prueba, más que en verdad como en el caso de lógica clásica para explicar el significado y uso de conectivos lógicos en términos de conocimiento o demostrabilidad. Enseguida se da la definición formal de Intuicionismo [15].

Definición 2.12 La teoría formal del cálculo proposicional intuicionista es: Símbolos lógicos $\leftarrow\vee$, $\wedge$ y la negación clásica "not" (denotada también como $\neg$). *Modus ponens* como la única regla de inferencia y el siguiente esquema de axiomas [15]:

1. $A \rightarrow (B \rightarrow A)$
2. $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$
3. $A \wedge B \rightarrow A$
4. $A \wedge B \rightarrow B$
5. $A \rightarrow (B \rightarrow (A \wedge B))$
6. $A \rightarrow (A \vee B)$

7. $B \rightarrow (A \vee B)$
8. $(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C))$
9. $(A \rightarrow B) \rightarrow ((A \rightarrow \neg B) \rightarrow \neg A)$
10. $\neg A \rightarrow (A \rightarrow B)$

David Pearce [16] demostró que **M** un conjunto de átomos, **M** es un *Answer Set* de **P** si y solo si $P \cup \neg\mathbf{M} \setminus \mathbf{M}$.

Se ha seleccionado *Answer Set Programming* en esta tesis debido a que ésta es la realización de mucho trabajo teórico y que cuenta con software siguiendo esta teoría sobre razonamiento no monótono y aplicaciones de inteligencia artificial de programación lógica. Además, existen contribuciones recientes y relevantes que establecen relaciones estrechas entre *answer set programming* y lógica intuicionista. Esto nos permite que podamos utilizar toda la maquinaria existente de intuicionismo en ASP.

Capítulo 3

DLV: *Disjunctive Datalog System*

Como hemos visto en el capítulo 2, *answer set programming* es un nuevo desarrollo que ha mostrado tener un impacto significativo en la práctica de la programación declarativa y de la representación de conocimiento. Esto obedece primordialmente a que sus fundamentos teóricos son bien entendidos, incluyendo complejidad y poder expresivo. Esto le permite ser aplicada a un amplio rango de problemas de búsqueda y decisión. Además, está estrechamente relacionado a la programación con restricciones. Finalmente, pero no por eso menos importante, actualmente contamos con implementaciones prometedoras que han mostrado en la práctica ser muy eficientes, obteniendo resultados con tiempos iguales o incluso mejor que implementaciones ad hoc desarrolladas en lenguajes como C++ [18].

El sistema en su primera versión ha estado disponible desde 1997 y se ha ido mejorando después de varios años de investigación teórica. DLV admite un lenguaje basado en formalismos lógicos con un alto poder expresivo para que los programas puedan representar problemas prácticos relevantes. Ha tenido mejoras año con año y se han incorporado nuevas características y técnicas de optimización relevantes en todos los módulos [17].

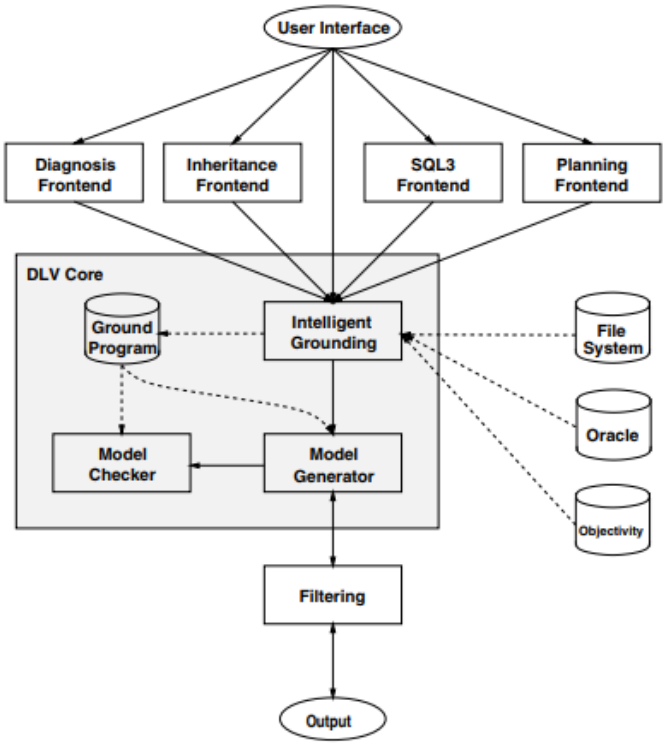


Figura 3.1 Arquitectura de DLV

La figura 3.1 muestra la arquitectura del sistema DLV. El orden en esta imagen es de arriba hacia abajo. La Interfaz Principal del usuario está orientada en línea de comandos, pero también cuenta con una GUI (Interfaz Gráfica de Usuario) y está apto para la mayoría de front-end. Los datos de entrada pueden ser dados por archivos regulares, bases de datos de Oracle y de Objetividad. El núcleo de DLV (DLV core) va dando uno por uno los *answer sets* y cada vez que lo encuentra invoca a "*Filtering*", el cual se encarga del procesamiento posterior y verifica si continúa o no. También el núcleo consta de tres componentes principales: "*Intelligent Grounding*", "*Model Generator*" y "*Model Checker*", estos módulos comparten una estructura de datos principal llamada "*Ground Program*", se crea a partir de *Intelligent Grounding* utilizando técnicas de base de datos diferenciales junto con estructuras de datos adecuadas que son utilizadas por *Model Generator* y *Model Checker*. Se garantiza que *Ground Program* tiene exactamente el mismo *answer set* que el programa principal. Para algunas clases de programas sintácticamente registrados, el módulo *Intelligent Grounding* ya calcula los *answer sets* correspondientes. *Model Generator* y *Model Checker* se encargan de la mayor parte del cálculo en problemas más difíciles. *Model Checker* verifica si el modelo en cuestión es un *answer set* [17].

DLV es una de las implementaciones más prolíferas que introduce algoritmos para calcular modelos estables (*answer sets*) de programas lógicos disyuntivos. DLV es un sistema de bases de datos deductivas basado en programación lógica y que ofrece varios *front-ends* para varios formalismos avanzados de representación de conocimiento. Formalmente, DLV es *datalog* disyuntivo extendido con restricciones, negación fuerte y consultas (*querys*) [19].

3.1 Lenguaje central de DLV

Los elementos básicos de DLV son las constantes, que refieren entidades u objetos que son almacenados en bases de datos relacionales.

Definición 3.1 Una constante debe iniciar con una letra minúscula y le pueden seguir letras, guión bajo y dígitos. Además, los números son por definición constantes [20].

not es una palabra reservada y no es una constante [20].

Definición 3.2 Una variable inicia con una letra mayúscula y le pueden seguir letras, guión bajo y dígitos. Una variable especial es la llamada variable anónima denotada por "_" (guión bajo). Cada ocurrencia de "_" representa una nueva y única variable [20].

Definición 3.3 Un término es una constante o una variable [20].

Los siguientes son ejemplos de elementos básicos en DLV:

constantes: a1, 1, 9862, aBc1, c

variables: A, V2f, Vi X3

literales: a, not ~b(8,K), not weight(X,1,kg) [20]

Definición 3.4 Átomos. Una literal es un átomo. DLV a diferencia de otros sistemas permite dos tipos de negación: la negación verdadera (o explícita) y la negación como

fallo. Escribiremos la negación verdadera con los siguientes símbolos – o ~. Mientras que la negación por fallo se escribirá con la palabra "*not*" [20].

Ejemplos de átomos válidos:
a
b(8,k)
weight(X,1,kg) [20]

Definición 3.5 Con fines de simplicidad diremos que un hecho se trata de una proposición y lo que encontramos dentro del paréntesis serán los argumentos. De manera sintáctica se escribirá primero el nombre de la proposición comenzando con una letra minúscula seguido de los argumentos dentro de un paréntesis, los argumentos estarán separados por comas, al final del hecho deberá ir un punto [20].

Ejemplos de hechos válidos:
weigth(apple,100, gram).
-valid(1, equals, 0). [20]

Definición 3.6 En DLV se pueden hacer comentarios entre las líneas de código. Para esto, utilizaremos el símbolo "%" el cual denotará que es un comentario y no una línea de código [20].

Ejemplo de comentario válido:
%Esto es un comentario
%----------Esto sigue siendo un comentario-----

3.2 Base de datos y Programación lógica

DLV combina dos paradigmas poderosos ampliamente probados en aplicaciones reales. Por tal razón, DLV puede ser visto como un sistema de programación lógica o como un sistema de base de datos deductivo [21,22]. Visto de esta manera, DLV consta de dos partes fundamentalmente: La bien conocida "Base de datos Extensional" y la "Base de datos Intencional" [20].

3.2.1 Base de Datos Extensional

Respecto a la base de datos extensional (BDE), DLV permite exportar e importar bases de datos relacionales a través del controlador ODBC. La BDE puede contener solo hechos tal como se muestra con el siguiente ejemplo [23].

Ejemplo 3.1 Supongamos que deseamos representar un grafo dirigido como el siguiente:

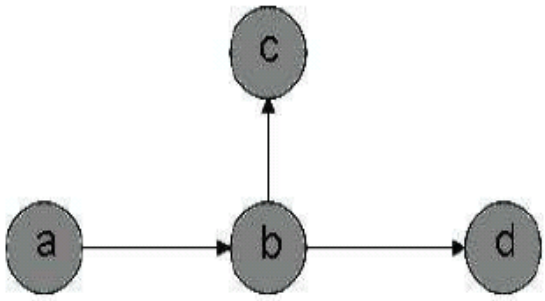


Figura 3.2 Grafo dirigido

Como podemos observar (figura 3.2), lo que deseamos representar es cada uno de los arcos dirigidos de que consta el grafo. Por tanto, la representación sería:

arco(a,b).
arco(b,c).
arco(b,d).

Aquí, arco es llamado un símbolo predicado o símbolo de relación, las partes entre paréntesis son constantes. Estas constantes, como se mencionó previamente, deben iniciar con una letra minúscula.

El anterior símbolo predicado constituye la BDE que puede ser guardado en el archivo "grafo.bde" que podría ser interpretado por DLV de la manera siguiente:

> DLV -silent grafo.bde

Obteniendo el siguiente answer set: arc(a,b), arc(b,c), arc(b,d)

O bien de la manera siguiente, donde le indicamos a DLV que no nos interesa conocer los símbolos predicados contenidos en la BDE.

> DLV -silent -nofacts grafo.bde

Obteniendo el *answer se*t: fg, que significa que la respuesta es vacía, es decir, DLV no despliega los hechos definidos en la BDE. Por tanto, no hay información adicional que pueda deducir DLV [20].

3.2.2 Base de Datos Intensional

En esta sección, mostraremos como definir conocimiento en que puede hacer uso de la BDE. A este conocimiento, se le conoce generalmente como el programa, o Base de Datos Intencional - BDI. Así, las reglas denotan relaciones entre literales. La BDE corresponde a los datos de entrada para la BDI o programa. Las reglas tienen la forma siguiente:

h1 v ... v hn :- b1, ..., bm.

donde, h1 a hn representa explícitamente literales, donde n > 0 (es decir, debe haber al menos uno de éstos). Los símbolos " :- " denotan la implicación y b1, ..., bm representan literales en general, donde m ≥ 0 debe suceder (es decir, el cuerpo puede ser omitido completamente). La parte antes de " :- " es referida como cabeza, la parte después de " :- " como cuerpo. Note que la negación como falla puede ocurrir sólo en el cuerpo. Además, las reglas deben ser seguras.

De manera informal, si el cuerpo evalúa verdadero, la cabeza debe evaluar verdadero también. Los símbolos de negación como falla pueden leerse como "No hay evidencia de que xxx evalúe a verdadero", como en la lógica de dafault. Así mismo, la cabeza representa una disyunción y el cuerpo una conjunción.

Normalmente, una BDI depende de los datos de entrada definidos en la BDE [20]. Enseguida, comenzaremos presentando programas simples a través de ejemplos.

Reglas y hechos

Supongamos que deseamos modelar que en todo momento alguien nos dice un chiste, reímos. Además, alguien ahora nos dice un chiste. Esto se puede hacer de la forma siguiente:

```
broma.
rie :- broma.
```

La primer línea es un hecho y expresa que una broma es verdadera (una palabra o hecho que tiene el valor de verdad verdadero es llamada una proposición). La segunda línea es llamada una regla. Esta se lee como sigue: Si la broma es cierta, rie debe ser cierto. El signo ":-" representa una flecha hacia la izquierda, en la versión de programación lógica es una implicación [20].

3.3 Negación y suposición de un mundo completo

Este tipo de negación es conocida como negación como falla y es empleada cuando no estamos conscientes sobre si algo sucede o no. Esta es una de las negaciones ampliamente aceptada como una propuesta razonable: La semántica de modelos estables (*answer set programming* – ASP) [20].

La negación es tratada como "negación como falla", en otras palabras, Si un átomo no es cierto en algún modelo, entonces su negación deberá ser considerada como cierta en ese modelo [22]. La idea de cómo aplica este tipo de negación lo veremos con el siguiente ejemplo:

```
nodo(X) :- arco(X,_).
nodo(Y) :- arco(_,Y).
complemento_arcos(X,Y) :- nodo(X), nodo(Y), not arco(X,Y).
```

Aquí, la regla complemento_arcos describe el conjunto de arcos en el grafo complementario. Tal que un arco debe ir desde un nodo a otro (posiblemente el mismo), y este arco no debe estar contenido en el arco original.

Note que nodo(X) y nodo(Y) necesitan estar incluidos en el cuerpo en el orden de satisfacer los siguientes requerimientos de seguridad para las reglas. Variables, las cuales ocurren en una literal negada, deben también ocurrir en una literal positiva en el cuerpo.

Ejecutando la regla anterior con el nodo de la figura 3.2 la ejecución será como la siguiente:

$ DLV -silent -nofacts simple_graph compl_graph
Dando como respuestas:
{node(1), node(2), node(3), node(4), comparc(1,1), comparc(1,3), comparc(1,4), comparc(2,1), comparc(2,2), comparc(3,1), comparc(3,2), comparc(3,3), comparc(3,4), comparc(4,1), comparc(4,2), comparc(4,3), comparc(4,4)}

3.4 Negación verdadera

DLV implementa otra noción de negación: Negación verdadera.
La negación como falla, la cual fue introducida antes, no soporta aseveración explicita. Más bien, Si no hay evidencia de que un átomo es verdadero, este es considerado como falso [20].

Sin embargo, hay varias situaciones en las cuales la negación como falla no es apropiada porque es necesario que en algunas ocasiones sepamos que algo es explícitamente conocido que es falso. Por esta razón, la negación verdadera es algunas veces referida como *negación explícita* [22].

Esta negación es denotada precediendo un átomo con "~" o "-".

Ejemplo 3.2 Imagine una simple situación, en la cual un agente tiene que cruzar una vía de ferrocarril. El agente deberá cruzar esta si ningún tren se aproxima. Con esta descripción, uno puede especificarlo a través del siguiente programa:

cruza_vía :- not tren_aproxima.

Dado que no se tiene ninguna otra información, tren_aproxima no sucederá, y cruza_vía será derivado, es decir, si ejecutamos ese programa tendremos:

```
$ DLV -silent rail_naf
  {cross_railroad}
```

Pero esto es un problema, ya que la regla indica que el agente cruce, lo que puede causar que el agente sea arrollado por el tren si este se aproxima dado que el agente cruzará (según indica el programa).
La forma de resolver este tipo de situaciones es a través de la negación verdadera, la cual nos permite modelar este mismo problema de la manera siguiente:

```
cruza_vía :- -tren_aproxima.
```

En este caso, -tren_aproxima puede ser considerado como un átomo separado, y dado que no hay información de que esto suceda, cruza_vía no puede ser derivado, es decir, si ejecutamos el programa este dará:

```
$ DLV -silent rail_naf
  { }
```

3.5 Restricciones de Integridad

Las restricciones en nuestro marco de trabajo especifican que no puede ser verdadero en un modelo. En otras palabras, Las restricciones son formulaciones de posibles inconsistencias. Este mecanismo es muy útil en conjunción con reglas disyuntivas. Las reglas disyuntivas sirven como generadores para diferentes modelos y las restricciones son usadas para seleccionar solo lo deseado.

La sintaxis de las restricciones es similar a la de las reglas, solo que no tienen cabeza. Como con las reglas, las restricciones deben reunir requerimientos de seguridad [20]. Veamos el siguiente ejemplo:

Ejemplo 3.3 El problema consiste en colorear un mapa que es representado por un grafo. La restricción es, que dos nodos conectados por un arco no pueden tener el mismo color.

Para formular este problema con reglas DLV, reconsideraremos algunas de las reglas previas.

```
nodo(X) :- arco(X,_).
nodo(Y) :- arco(_,Y).
color(X,rojo) v color(X,verde) v color(X,azul) :- nodo(X).
```

Los modelos (respuestas) de este programa corresponden a todas las posibles combinaciones de coloreado. Entonces, nosotros solo necesitamos agregar una restricción la cual descartará las combinaciones donde dos nodos adyacentes tengan el mismo color. Esto corresponde a la siguiente restricción.

```
:- arcoX,Y), color(X,C), color(Y,C).
```

Esta restricción nos brindara el resultado deseado, ya que eliminará las combinaciones que no deseamos (nodos adyacentes no pueden ser del mismo color).

3.6 Restricciones Débiles

Una de las características sobresalientes de DLV lo son las restricciones llamadas "débiles". Estas nos permiten formular muchos problemas de optimización de una manera natural y sencilla. Mientras las restricciones estándar siempre tienen que ser satisfechas, las restricciones débiles expresan el objetivo, es decir, estas deben ser satisfechas sólo si es posible, pero su violación no elimina modelos (respuestas) [20].

Los *answer sets* (conjunto de respuestas) de un programa P con un conjunto W de restricciones débiles son aquellos *answer sets* de P los cuales minimizan el número de restricciones violadas. Estas son llamadas los mejores modelos de (P,W). Notese que un programa P puede tener varios mejores modelos (violando el mismo número de restricciones débiles) [20].

Sintácticamente, las restricciones débiles son especificadas como sigue:

:~ Conj. [Weight:Level]

Donde Conj es una conjunción de literales, y Weight y Level son enteros positivos.

Los pesos y niveles de prioridad se permiten que sean variables, previendo que esas variables también aparezcan en una literal positiva en Conj. El usuario puede omitir el peso o la prioridad o ambos, pero todas las restricciones débiles deben tener la misma forma sintáctica (es decir, el usuario es libre de especificar solo pesos o solo prioridades, o ambas, pero todas las restricciones del programa deben tener la misma forma sintáctica.

Ejemplo 3.4 Considere el siguiente programa contenido en el archivo example1.

```
a v b.
c :- b.
:~ a.
:~ b.
:~ c.
```

Dado que los pesos y niveles de prioridad son omitidos, sus valores son asignados a 1 por default. Si ejecutamos este programa en DLV, obtenemos la siguiente salida.

```
$ DLV -silent example1

Best Model: {a}
Cost ([Weight:Level]): <[1:1]>
```

Note que los *answer sets* de { a v b. c :- b. } son {a} y {b, c}. La presencia de restricciones débiles descarta {b, c} porque este viola dos restricciones (mientras que {a} viola solo una restricción).

3.7 Aritmética de predicados en DLV

El razonamiento y cómputo sobre un conjunto finito de rangos enteros es posible con los predicados de DLV.

#int, #succ, +, *.

Estos operadores aritméticos son solo posibles si la aritmética entera es turnada a activa dando un número límite N sobre la línea de comando. DLV solo trabaja con números enteros, es decir, mayores o iguales que cero y menores o iguales que la N definida en la línea de comando [20].

Ejemplo 3.5 Definamos el predicado de Fibonacci.

```
fibonacci(N,F) :-    #succ(N2,N1),
                     #succ(N1,N),
                     fibonacci(N1,F1),
                     fibonacci(N2,F2),
                     +(F1,F2,F).
```

Como podemos observar, la manera en que se formula esta definición es literalmente escrita de manera directa sobre DLV.

Ejemplo 3.6 La división y módulo no han sido implementados aún. Sin embargo, estos pueden ser fácilmente simulados usando las primitivas existentes como sigue:

```
div(X,Y,Z) :- XminusDelta = Y*Z,
              X = XminusDelta + Delta,
              Delta < Y.

mod(X,Y,Z) :- div(X,Y,XdivY),
              XminusZ = XdivY * Y,
              X = XminusZ + Z.
```

Note que, al contrario de la construcción de funciones, la adición de estas reglas predicado div y mod en el modelo (esto es, si #maxint es 100, entonces cada *answer set* del programa contendrá 100*100 hechos para ambos div y mod).

Existe otro conjunto de características igual de importantes que las tratadas en este capítulo, sin embargo, invitamos al lector a referirse a los documentos originales que pueden ser accesados en la siguiente dirección electrónica con acceso libre, http://www.dlvsystem.com/html/DLV_User_Manual.html.

3.8 Comparación de implementaciones

Como hemos podido observar en *answer set programming*, las soluciones a un problema son representadas por *answer sets* (conjuntos de respuestas), y no por sustituciones de respuestas producidas en respuesta a un *query* como en la programación lógica convencional (como prolog). Este método de programación hace uso de calculadores de *answer sets*, tales como Smodels, Cmodels, ASSAT, DLV y GNT. Estos sistemas eficientes hacen posible llevar a nivel de software el paradigma ASP. DLV y GNT son el software más general que trabajan con clases de programas lógicos disyuntivos mientras que otros solo cubren el caso no disyuntivo. Por otro lado, los sistemas ASSAT y Cmodels usan resolvedores SAT como máquinas de búsqueda. Estos están basados en la relación entre la completación de semánticas y *answer set programming* para programas lógicos.

Así, DLV es el sistema de software que hemos elegido como sistema formal para modelar nuestro problema de tesis señalado previamente.

Como hemos podido observar, DLV puede ser usado como una implementación para resolver problemas de representación declarativa, esto quiere decir que no se escribe un programa que resuelva un problema, sino que se especifica cómo debería ser la solución a ese problema. En este trabajo de tesis hemos empleado DLV como un lenguaje para la representación de conocimiento para diversos problemas de planeación, este lenguaje también es utilizado para dar una solución óptima a juegos [24], por mencionar alguno, al juego conecta 4. Particularmente, se representa el conocimiento de un agente cuyo objetivo es el de buscar la mejor tirada en el juego bipersonal llamado cuatro en línea (conecta 4) [25]. Este tipo de problemas son muy comunes en diversas áreas del conocimiento, tal como se verá en el siguiente capítulo.

Capítulo 4

Planeación a través del problema La travesía del puente

En el capítulo anterior hablamos sobre el sistema DLV, el cual nos dice que es un sistema de base de datos deductiva basado en la programación lógica y sirve para el cálculo de *answer sets*, también se menciona que este sistema ofrece varios *front-ends* para representar el conocimiento, uno de ellos es DLV^K el cual sirvió para desarrollar y dar solución al problema central de este trabajo de tesis y será descrito paso a paso a lo largo de este capítulo.

4.1 Planeación

Algunos autores definen la planeación de la siguiente forma:

"Planear es el proceso para decidir las acciones que deben realizarse en el futuro, generalmente el proceso de planeación consiste en considerar las diferentes alternativas en el curso de las acciones y decidir cuál de ellas es la mejor" Robert N. Anthony.

"La planeación consiste en fijar el curso concreto de acción que ha de seguirse, estableciendo los principios que habrán de orientarlo, la secuencia de operaciones para realizarlo, y la determinación de tiempos y números necesarios para su realización". A. Reyes Ponce.

En el enfoque computacional, la planeación consiste en la siguiente tarea: dado un estado inicial, varias acciones, las condiciones previas y efectos de estas, se encontrará una secuencia de acciones o sea un plan, para un estado en el que se logró alcanzar la situación deseada [26], tal como se muestra gráficamente en la figura 4.1.

Figura 4.1 Representación gráfica de planeación

Definición 4.1 Planeación es la representación de estados que describen como procederá con la ejecución de un conjunto de acciones bajo un orden con limitaciones temporales o algún otro tipo de estas [27].

Definición 4.2 Un problema de planeación es una cuádrupla de la siguiente forma [27]:

<F, A, I, G>

Dónde:

- ***F*** es un conjunto de fluentes, los cuales caracterizan las situaciones
- ***A*** es un conjunto de acciones, con una definición 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

4.1.1 Planeación Clásica

La planeación clásica es el problema para encontrar una secuencia de acciones para que se logre el objetivo, a partir de un estado inicial particular. La planeación clásica se puede convertir como un problema de búsqueda de ruta en un grafo, cuyos nodos son los estados posibles y las aristas son las transiciones posibles con las acciones que se tienen disponibles. Se puede expresar problemas diferentes de esta forma [28]. Por ejemplo el problema del Agente Viajero descrito a continuación.

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.
Dado un grafo completo no dirigido 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).

También cabe mencionar que la planeación clásica ha tenido éxito en tratar de encontrar un plan con N pasos o estados creados a partir de proposiciones.

4.2 Lenguaje de Acción K

Antes de entrar de lleno al sistema DLV^K es conveniente mencionar que es lenguaje de acción K y la importancia de este. K es un lenguaje de planeación declarativa basado en programación lógica, con este lenguaje pueden describirse las transiciones entre estados de conocimiento completo e incompleto. El uso de estados de conocimiento permite una representación de problemas de forma natural y compacta. Los resultados forman la base teórica para el sistema DLV^K. [29]

Como se menciona en el párrafo anterior, el lenguaje K puede trabajar y razonar con estados incompletos, esto es una característica que lo distingue con otros lenguajes de acción. Singularmente, se hace la distinción entre negación por falla y la negación fuerte, el lenguaje K permite razonar sobre estados de conocimiento en el que un fluente puede ser verdadero, falso o desconocido, y estados del mundo en los que el fluente es verdadero o falso. Esto permite diferentes enfoques de planeación, incluida la planeación clásica (con la información y conocimiento tratados de forma tradicional) y la planeación con supuestos predeterminados o incompletos [26].

4.3 DLV^K

Dicho anteriormente el sistema DLV^K es la implementación de un *front-end* para el lenguaje K en el sistema de programación lógica DLV y así es como se obtiene este eficaz sistema de planificación declarativa [19].

La semántica formal de este sistema se basa en la transición. El lenguaje C y muchos otros se basan en extensiones de lógicas clásicas y describen transiciones entre posibles estados del mundo. Aquí, un estado del mundo se caracteriza por los valores de verdad de varios fluentes, es decir, predicados que describen propiedades relevantes del dominio del discurso, donde cada fluente necesariamente es verdadero o falso. Una acción es aplicable solo si alguna precondición (fórmula sobre los fluentes) es verdadera en el estado actual, y la ejecución de esta acción cambia el estado actual al modificar los valores de verdad de algunos fluentes [26].

DLV^K cuenta con conocimientos previos explícitos: el dominio de planificación tiene antecedentes (representados por un programa estratificado de registro de datos) que

describe predicados estáticos. Declaraciones de tipos: se escriben los argumentos de predicados cambiables, llamados fluentes, y átomos de acción. Negación fuerte y débil. Los estados en este sistema son conjuntos consistentes de literales básicos, en los que no todos los átomos deben aparecer y, por lo tanto, representan estados de conocimiento. Ejecución paralela / secuencial de acciones: es posible la ejecución simultánea de acciones, y de hecho el modo predeterminado. Todas las acciones para ejecutar deben cumplir con una condición de ejecución. La exclusión mutua de acciones puede hacerse cumplir en un modo de planificación secuencial. DLV^K puede calcular planes seguros (a menudo llamados planes conformes en la literatura). Informalmente, un plan es seguro, si es aplicable a partir de cualquier estado inicial legal y hace cumplir la meta, independientemente de cómo evolucione el estado. Con esta función, también podemos modelar la planificación de mundos posibles con un estado inicial incompleto, donde el mundo inicial solo se conoce parcialmente, y estamos buscando un plan que alcance la meta deseada de cada mundo posible de acuerdo con el estado inicial.

Sin embargo, los agentes de planificación generalmente no tienen una visión completa del mundo. Incluso si su conocimiento es incompleto, es decir, se desconocen varios fluentes, deben tomar decisiones, ejecutar acciones y razonar sobre la base de su información (incompleta) disponible. Por ejemplo, imagina un robot frente a una puerta. Si no se sabe si la puerta está abierta, el robot puede decidir empujar hacia atrás. Alternativamente, podría decidir detectar el estado de la puerta para obtener información completa. Sin embargo, esto requiere que haya disponible una acción de detección adecuada y, lo que es más importante, realmente ejecutable (es decir, el sensor no está roto). Por lo tanto, incluso en presencia de detección, algunos fluidos pueden permanecer desconocidos y dejar a un agente en un estado de información incompleta. A continuación se muestra más a detalle cómo funciona el sistema DLV^K [26].

4.3.1 Sintaxis de DLV^K

Para ayudar describir la sintaxis de un programa en DLV^K tomaremos como ejemplo un problema de planeación común llamado “el acertijo del pastor” descrito a continuación.

Ejemplo 4.1 El acertijo del pastor

El objetivo es ayudar al pastor a cruzar al otro lado del rio a una oveja, un lobo y una caja de coles, debemos tener en cuenta que el lobo no puede quedarse solo con la oveja y la oveja no se puede quedar sola con la caja de coles. En un inicio los objetos estarán en la posición 2 que representa el lado derecho y la meta es cruzarlos a la posición 1 que representa al lado izquierdo, tal como se muestra en la figura 4.2 y 4.3.

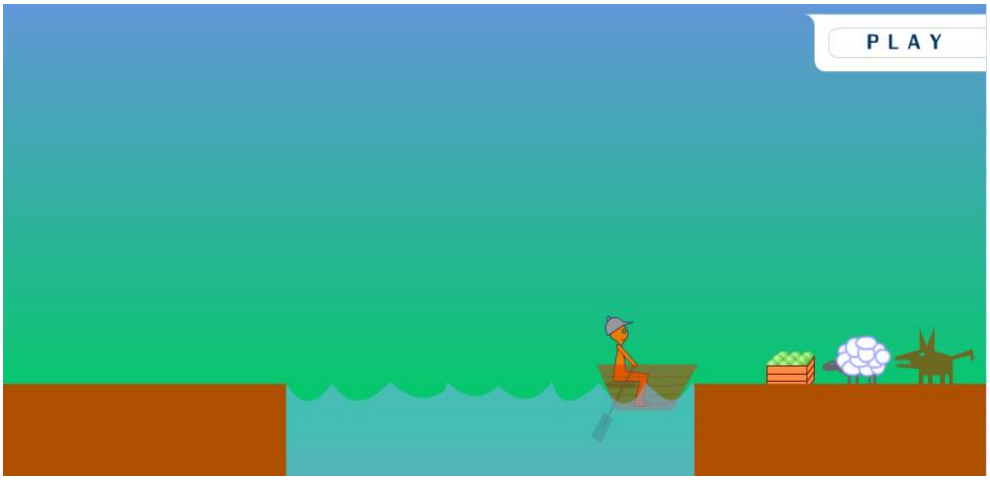


Figura 4.2 Estado inicial

Figura 4.3 Meta

Una vez presentado el escenario, se modelan cada uno de los elementos necesarios que definen a todo el problema de planeación como: fluentes, acciones, condiciones de ejecución, reglas que describen las causas de las acciones (*causation rules*), el inicio y la meta, para así el programa pueda calcular un plan óptimo el cual contenga la solución al problema.

En este caso los elementos para modelar el problema son: el lobo, la oveja, la caja de coles y la locación (para indicar si ya cruzó alguno de los anteriores o no).

- **Fluentes.** Representan propiedades básicas del mundo (caracterizan el contexto) [26]. Se declaran de la siguiente forma:

  fluents: p(X1, …, Xn) requires t1, …, tm [27]

Para el problema de ejemplo los fluentes son:

fluents: onl(X,Y) requires lobo(X), location(Y).
ono(X,Y) requires oveja(X), location(Y).
onr(X,Y) requires repollo(X), location(Y).

Cada fluente caracteriza la descripción que tiene cada uno de los elementos en el contexto, el primero para el lobo, el segundo para la oveja y el tercero para la caja de coles; donde X puede ser cualquiera de los tres mencionados y Y es la posición donde esta cada uno y el que nos determina si el objeto ya cruzó el rio o no.

- **Acciones.** Estos son los eventos que cambian el estado del problema y de manera formal se declaran de manera similar como los fluentes [26]. En este caso las acciones quedan de la siguiente forma:

actions: moveo(X,Y) requires oveja(X), location(Y).
movel(X,Y) requires lobo(X), location(Y).
mover(X,Y) requires repollo(X), location(Y).

Cada una de las acciones indica que para mover a uno de los objetos se requiera una proposición X del tipo oveja, lobo o caja de coles y un Y de tipo locación para saber a qué lugar se moverá.

- **Condiciones de ejecución.** Estas van dentro de la sección “always”, se ejecutará una acción siempre y cuando se cumpla la condición [26]. Son de la forma:

  executable a if b1, …, bm, not bm+1, …, not bn [27]

Una de las condiciones de ejecución del problema es la siguiente:

executable moveo(X,Y) if ono(A,B), Y<>B.

Quiere decir que la acción "moveo(X,Y)" para mover a la oveja al sitio Y, se va a ejecutar si la oveja aún no ha cruzado, es por eso que Y debe ser diferente de B porque son sitios distintos.

O también puede ser que no se ejecute una acción si se cumple alguna condición. Son como la forma anterior solo reemplazando la palabra reservada *executable* por *nonexecutable*.

- **Causation rules (Reglas que describen las causas de las acciones).** Son las causas que se tienen al ejecutar una acción y de manera formal se declaran [26]:

  caused f if b1, ..., bk, not bk+1, ..., not bl
  after a1, ..., am, not am+1, ..., not an [27]

Las reglas que describen las causas de las acciones para el problema se representan como:

caused ono(X,Y) after moveo(X,Y).
caused -ono(B,L1) after moveo(B,L), ono(B,L1), L <> L1.

Al ejecutar la acción "moveo(X,Y)" causará que la posición de la oveja cambie al moverla, ya sea que estaba la posición 1 (lado izquierdo) y cruzó a la posición 2 (lado derecho) o viceversa, la segunda consecuencia es que la posición de la oveja ya no es L1, esto se modela poniendo el símbolo de negación "-" al fluente.
Para los dos objetos faltantes (oveja y caja de coles) las condiciones de ejecución y causation rules son de manera similar solo cambiando el nombre de la proposición y el fluente.

- **Estado inicial.** Se definen las condiciones iniciales del problema, están precedidas por la palabra reservada *initially*, tal como se muestra a continuación [26]:

initially: ono(a,2). onl(b,2). onr(c,2).

Significa que inicialmente la oveja "a", el lobo "b" y la caja de coles "c" están en la posición 2, como podemos recordar representa el lado derecho.

- **Meta.** Objetivo que se quiere alcanzar, precedido por la palabra reservada *goal*:

goal: onl(b,1), ono(a,1), onr(c,1) ? (5)

En este caso queremos que el lobo "b", la oveja "a" y la caja de coles "c" queden en la posición 1 (lado izquierdo) en ese orden para que el problema sea resuelto.

A continuación se muestra la ejecución del código.

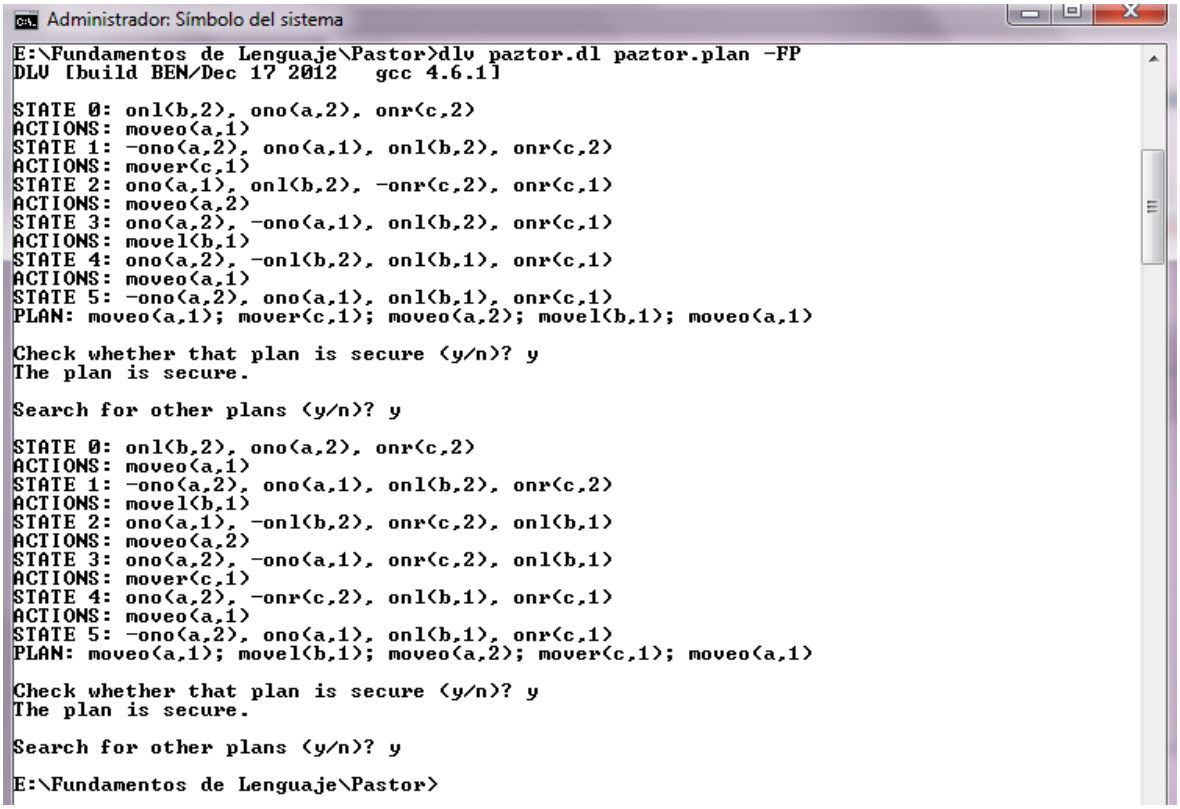


Figura 4.4 Ejecución del problema El acertijo del pastor

Al ejecutar el programa se obtuvieron dos soluciones al problema tal como se muestra en la figura 4.4. Para la primera solución el *STATE 0* muestra que la oveja, el lobo y la caja de coles se encuentran en la posición 2, ya que fue como se declaró en *initially*; en el *STATE 1* ahora la oveja ya no está en la posición 2 porque antes se ejecutó la acción moveo(a,1) es por eso que la proposición ono tiene el signo "-" pero delante de este se muestra el cambio de posición en dicha proposición; en el *STATE 2* ahora el lobo fue el que cruzó al sitio 1 ya que se ejecutó la acción movel(b,1); recordemos que una de las condiciones es que el lobo no puede estar solo con la oveja es por eso que en el *STATE 3* la oveja regresa a la posición 2; en el *STATE 4* podemos ver otra de las condiciones, que la oveja no puede quedarse sola con la caja de coles pero el lobo si entonces el programa ejecuta la acción mover(c,1) para cruzar a la caja de coles al sitio 1; finalmente la única por cruzar al lado 1 y se cumpla el objetivo es la oveja por lo tanto se ejecuta su acción y los tres quedan en el lado izquierdo cumpliéndose la meta.
Como podemos observar al final de los estados se muestra el plan que es el resumen de las transiciones entre estados.

Asimismo la segunda solución, sólo que aquí el plan muestra que primero se movió a la oveja al sitio 1, después al lobo, se regresa a la oveja al sitio 2, se cruza a la caja de coles al sitio 1 y finalmente se cruza a la oveja para que los tres objetos estén en la meta.

Como podemos apreciar, la sintaxis de DLV^K no es tan compleja de utilizar y entender, entonces ahora podemos abordar el problema principal de este trabajo de tesis el cual se presenta a continuación.

4.4 La travesía del puente

Ya vista la sintaxis que se utiliza en DLV^K para solucionar un problema de planeación, ahora en este punto nos adentraremos al problema principal de esta tesis para obtener la solución o soluciones para el problema "la travesía del puente", la descripción de dicho problema es la siguiente, se tiene que cruzar al otro lado del puente una familia conformada por 4 personas, cada miembro de la familia cruza a velocidades distintas (1 seg., 3 seg., 6 seg., y 8 seg.), en el puente solo puede cruzar 1 o 2 personas máximo y una de ellas forzosamente debe llevar la linterna ya que es de noche, cabe mencionar que el tiempo de la linterna dura 20 segundos encendida; si cruza un par de personas se

tomara el tiempo del que tenga la mayor velocidad, es por eso que se debe encontrar la solución más óptima para que los 4 miembros de la familia crucen a un velocidad que no sobrepase los 20 segundos.

Se realizó un análisis previo de la codificación en DLV^K para encontrar solución o posibles soluciones al problema y definir los fluentes, acciones y las causas que se tendrían al cambiar el estado de cada integrante de la familia y el tiempo de la linterna para así poder modelar este problema de planeación.

Iniciamos con establecer los elementos principales en este caso la familia con su velocidad, para diferenciar a cada uno se le dará un nombre: delgado = 1segundo, mediano = 3 segundos, mujer = 6 segundos, hombre = 8 segundos; el tiempo de la linterna inicia con 0 segundos y después se irá sumando la velocidad mayor si se da el caso de que cruce un par de personas, si no, solamente se sumará la velocidad de la persona que cruza sola; para diferenciar el lado inicial donde está situada la familia que gráficamente es el lado derecho se ocupará el nombre "aquí" y cuando hayan cruzado al lado izquierdo se cambiará por "¬aquí", esto aparte de diferenciar los lugares facilitará al programa porque se usa la misma variable, solo que cambiara su valor de verdad cuando sea necesario, tal como se muestra a continuación.

estado	**¬ aquí**	**linterna**	**aquí**
0		linterna=0	**delgado(1), mediano(3), mujer(6), hombre(8)**
1	delgado(1), mediano(**3**)	← linterna=3 segundos	mujer(6), hombre(8)
2	mediano(3)	→ linterna=4 segundos	delgado(**1**), mujer(6), hombre(8)
3	mediano(3), mujer(6), hombre(**8**)	← linterna=12 segundos	delgado(1)
4	mujer(6), hombre(8)	→ linterna=15 segundos	delgado(1), mediano(**3**)
5	**delgado(1), mediano(3), mujer(6), hombre(8)**	← **linterna=18 segundos**	

Tabla 4.1 Estados

En la tabla 4.1 se muestra que al realizar este procedimiento tuvimos un buen resultado ya que el tiempo de la linterna con todos los movimientos sumó 18 segundos, por lo tanto no excede el tiempo límite. Nos ayudó a definir un poco más las proposiciones que se ocuparán y las causas que se tendrán, por ejemplo, cuando cruce una sola persona el programa deberá utilizar la proposición "cruzar(delgado)" o cualquier otro de la familia, cuando cruce un par de integrantes de la familia se utilizará "cruzardos(delgado, mediano)" o cualquier combinación de pares posibles.
Las causas serán definidas por los cambios de cada acción, por ejemplo, siguiendo la tabla anterior del estado 0 al estado 1 cruzaron dos personas con la linterna, delgado y mediano, esto causa que ambos cambien su estado a "¬aquí" y que el tiempo de la linterna aumente a 3 segundos porque se toma la velocidad mayor, pasa lo mismo en la transición del estado 1 al 2 cruza solamente delgado con la linterna, causa que ahora su estado cambie a "aquí" y se sume su tiempo a la linterna.

Se tendrán tres archivos, "declaracion.dl" donde se declaran las variables que se utilizaron, "cruzaflu.plan" contiene las acciones y las causas, y "meta.plan" el estado inicial y el estado al que se quiere llegar.

Tomando en cuenta el análisis anterior, se hizo la prueba solo para cruzar a una persona, sin ninguna restricción aún, pero sí con las causas que tendrá. En el primer archivo se declaró la variable persona y la velocidad de esta, como se muestra en la figura 4.5, al declararlas por separado ayudará a que el programa tenga una mejor respuesta. En la figura 4.7 se define como estado inicial que la persona delgado este en "aquí" y el estado final se quiere que la persona ahora este en "-aquí".

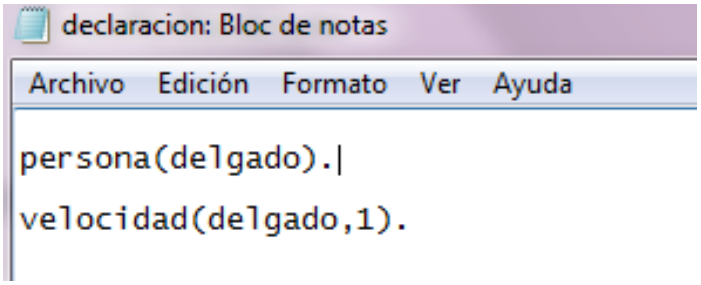


Figura 4.5 Variables

La figura 4.6 muestra cómo se declararon los fluentes que describen el entorno del problema, en este caso la proposición aquí, requiere a una variable X de tipo persona (por el momento solo tomará el valor de la persona delgado porque es el único declarado hasta ahora), la acción de cruzar a uno requiere una variable X de tipo persona; en la sección *always* se ejecutará la acción cruzar (sin ninguna condición aun) y las causas que se mencionaban, la variable X al estar en "aquí" y después de cruzar ahora esté en "-aquí" o si X estaba en "-aquí", después de cruzar ahora cambia a "aquí".

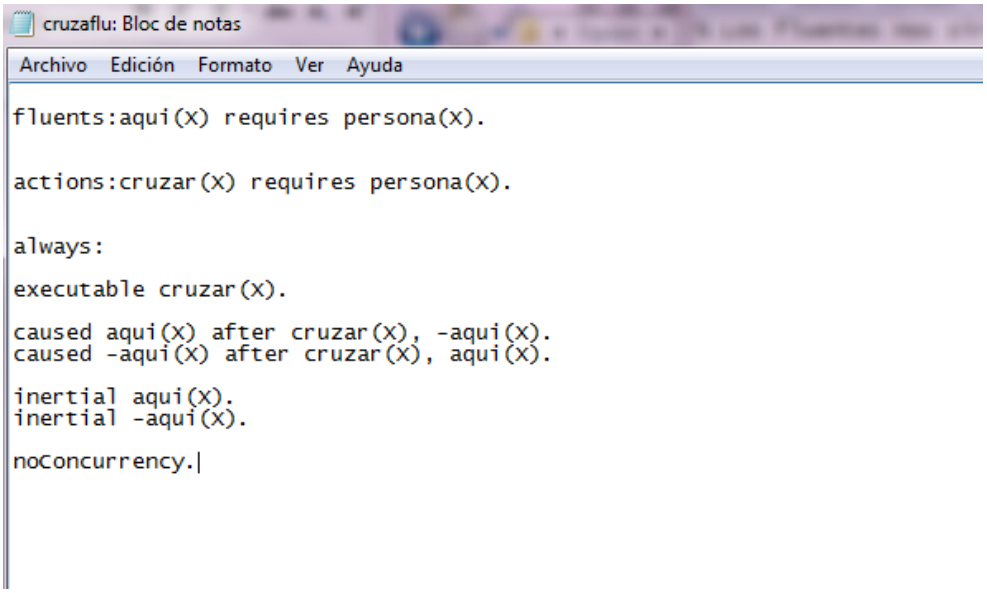


Figura 4.6 Fluentes

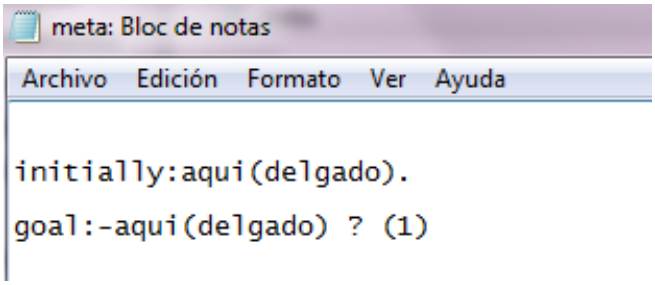


Figura 4.7 Inicio y meta

Al ejecutar en línea de comandos los tres archivos se deben poner en orden, primero el archivo donde están las variables, después el que contiene los fluentes, acciones, etc. y por último el archivo que tiene la inicialización y la meta; ya que de no ser así no se lleva a cabo la ejecución, la forma es la siguiente:

>dlv declaracion.dl cruzarflu.plan meta.plan –FP

Tenemos la salida que se muestra en la figura 4.8, se obtuvo el resultado que se esperaba, en el *STATE 0* delgado estaba en “aquí”, al realizar la acción de cruzar cambió su valor por “-aquí” tal como se muestra en el *STATE 1*.

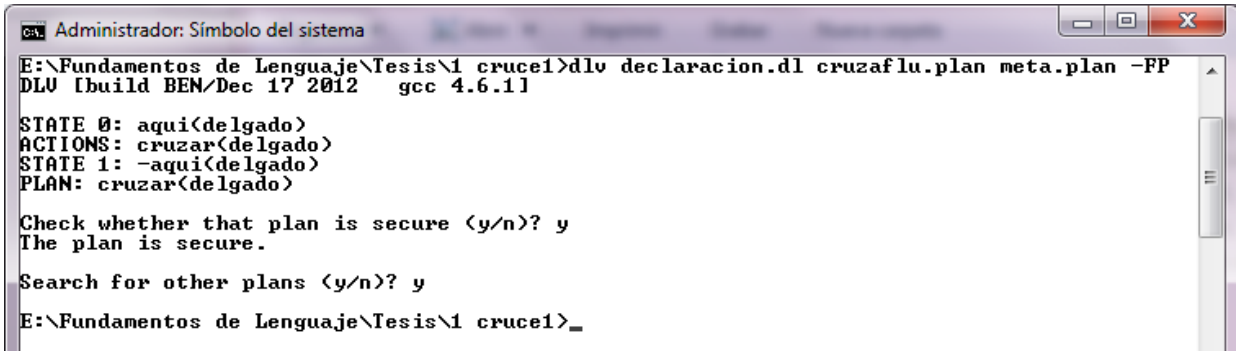


Figura 4.8 Resultado

Después se agregó otro integrante de la familia a la declaración de variables para hacer la prueba de cruzar un par de personas, así como lo muestra la figura 4.9.

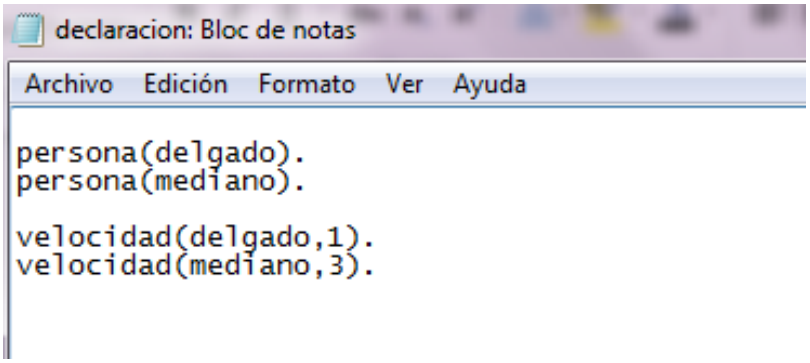


Figura 4.9 Variables

Al archivo meta en la sección de *initially* se le agregó la otra persona para indicar que esta al inicio en el lado izquierdo “aquí(mediano)” y en la sección *goal* también se agrega de la forma “-aquí(mediano)” para decir que ya cruzó.

En la figura 4.10 se puede ver como se agregó la función para la acción de mover dos personas que requiere de dos variables X y Y de tipo persona, también que ésta se ejecuta aun sin ninguna restricción y las causas que tiene dicha acción, nótese que para cada variable se tienen dos casusas posibles para cambiar su valor de verdad.

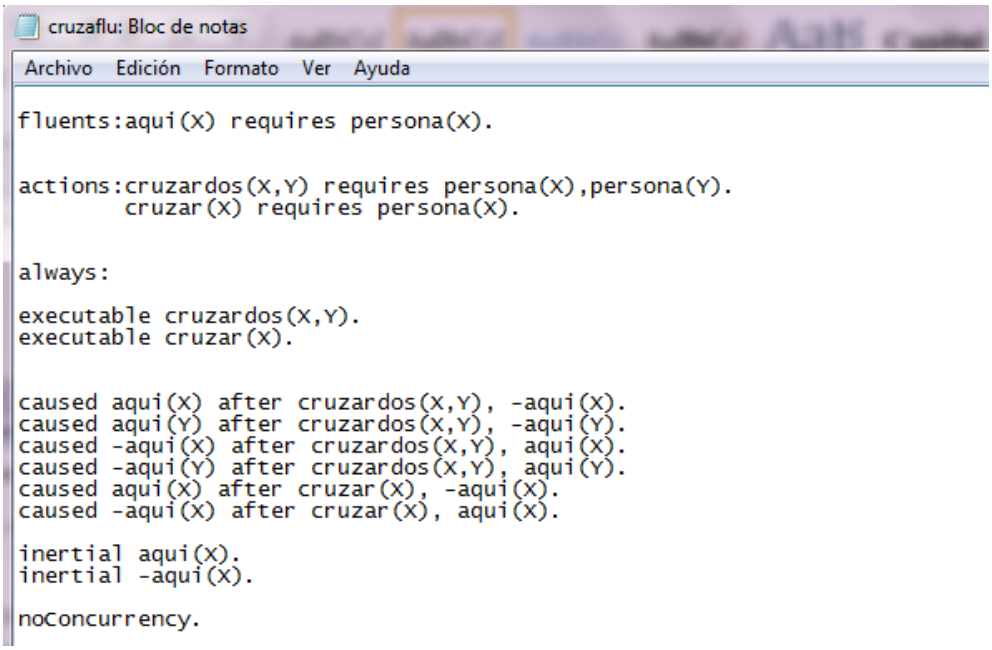


Figura 4.10 Fluentes

El resultado de la ejecución del programa se muestra en la figura 4.11. En el *STATE 0* la persona delgado y mediano se encuentran en "aquí" después ejecuta la acción "cruzardos" pero como podemos observar la función toma a mediano dos veces como si fueran personas distintas, esto es porque no tenemos una restricción al momento de cruzarlos, pero en el *STATE 1* ahora cambió mediano a "-aquí"; luego no hace ninguna acción por lo tanto *STATE 2* no tiene ningún cambio; por último vuelve a utilizar la acción de "cruzardos" y pasa lo mismo que el caso anterior, cruzó a delgado tomándolo como variables distintas y en el *STATE 3* ya se muestra que ambas personas están en "-aquí".

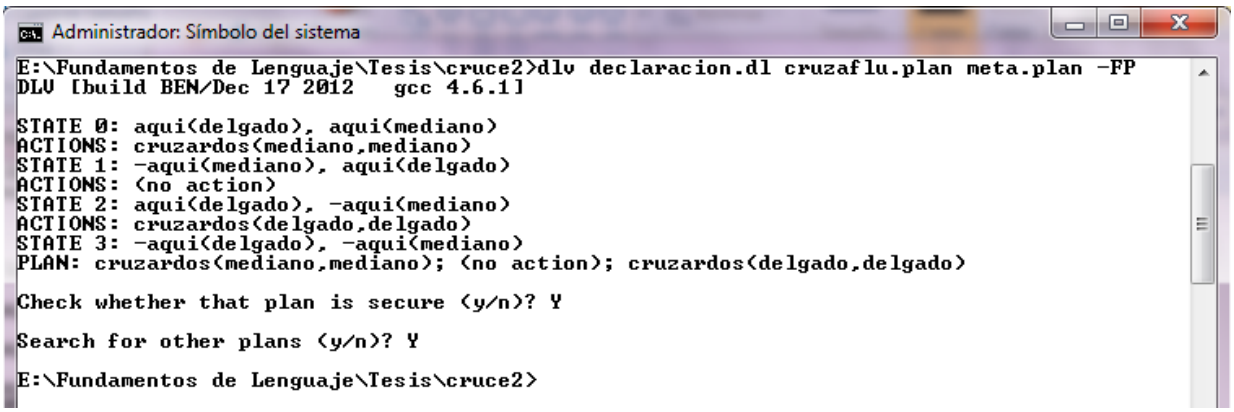


Figura 4.11 Resultado

Ahora se hace la prueba para cruzar a las cuatro personas. En las variables se declara a la persona mujer y hombre, también se agrega la velocidad de cada una, como se muestra en la figura 4.12.

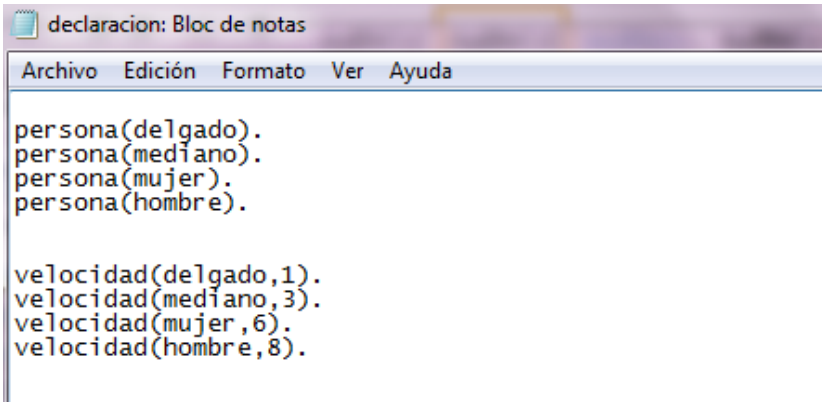


Figura 4.12 Variables

El archivo donde se encuentran los fluentes y las acciones se queda igual porque todavía no agregamos ninguna restricción para ejecutar las acciones y en el archivo donde está la inicialización y la meta también son agregados las dos personas faltantes en la sección *initially* como "aqui(mujer). aqui(hombre)" y en la sección *goal* quedan como "-aquí(mujer), -aquí(hombre)".

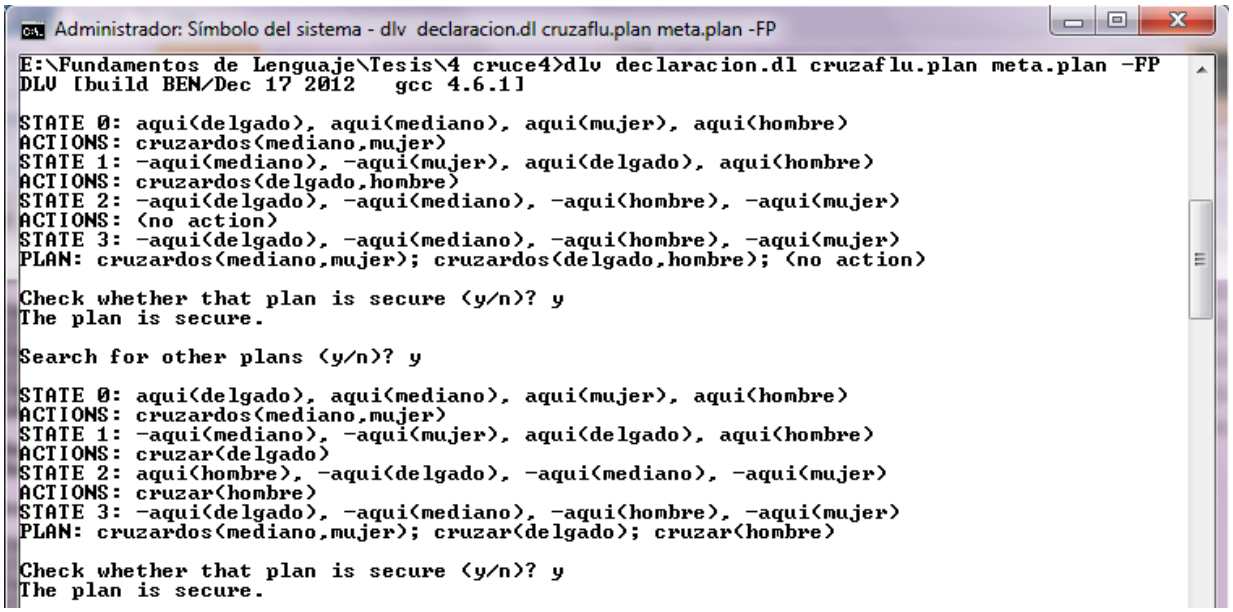


Figura 4.13 Resultado

En la figura 4.13 se muestran dos soluciones, en la primera el plan dice que cruzo a dos personas, mediano y mujer; después volvió a utilizar esa función, cruzando a las dos personas restantes, delgado y hombre.
La segunda solución, en el *STATE 0* las cuatro personas se encuentran en "aquí", después de hacer la acción "cruzardos(mediano, mujer)" en el *STATE 1* se muestra que ambas personas ahora están en "-aquí", luego se ejecuta la acción "cruzar(delgado)" y en el *STATE 2* se muestra el cambio que tuvo, por ultimo cruza la persona hombre y el *STATE 3* muestra que las cuatro personas ya están en "-aquí".

Una vez realizada la prueba y verificando que cruzan las cuatro personas, ahora se agrega la función que verificará cuando crucen dos personas, cuál de ellos es el que tiene mayor velocidad.

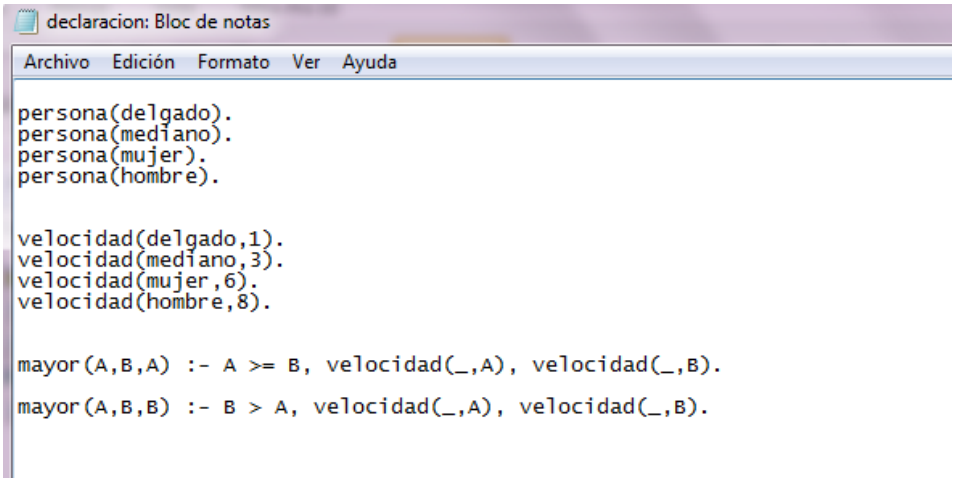


Figura 4.14 Función mayor

La figura 4.14 muestra cómo fue agregada la función mencionada en el párrafo anterior. La primera declaración es para el caso en el que la velocidad de la primer persona es mayor o igual a la segunda persona, sólo se tomará como mayor a la primer persona, ya que la que tiene menor velocidad deberá esperar a la de mayor velocidad. Para la segunda declaración ahora se verifica que la segunda persona sea la que tiene mayor velocidad y así poder almacenarla.

El archivo de inicialización y la meta no tiene ninguna modificación por el momento.

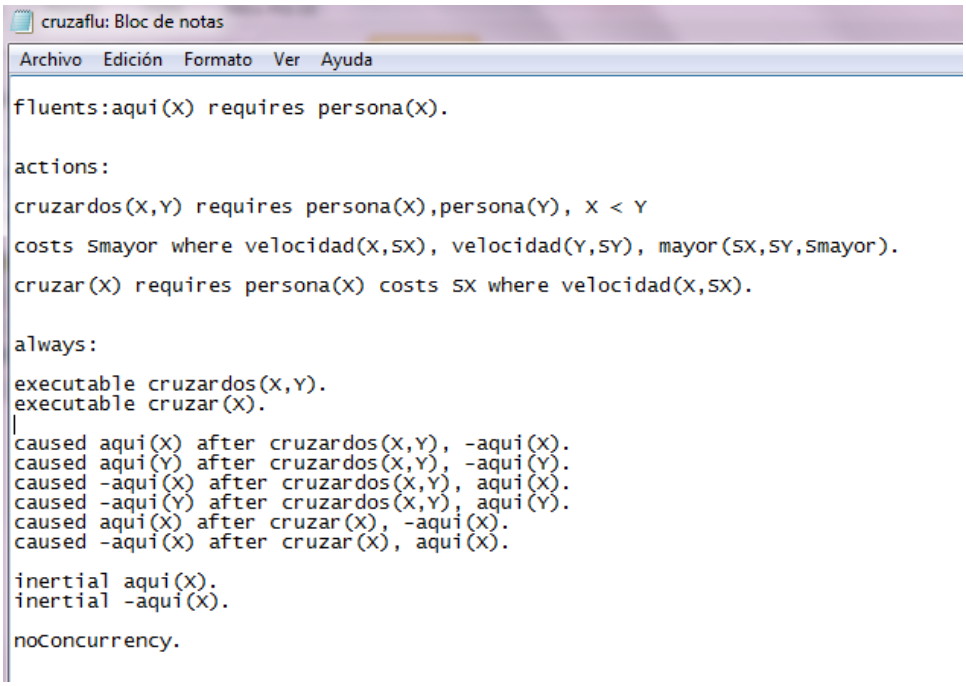


Figura 4.15 Costo

Como se muestra en la figura 4.15 la acción "cruzardos" tiene modificaciones, se agregó que X < Y, ya que se sabe que esta condición se cumple siempre por el orden en que se declararon inicialmente las personas; seguido se declara "*costs*" el cual lleva la cuenta de las velocidades ya que no se debe sobrepasar los 20 segundos. Siendo más específicos "Smayor" toma como resultado la velocidad mayor del cruce de un par de personas, las variables "SX" y "SY" se usan para obtener los valores de las dos personas y "mayor(SX, SY, Smayor)" declarado anteriormente en el archivo "declaracion.dl" regresará el costo total de ese cruce. De igual forma para la acción que cruza a una persona en caso de que el programa la ejecute, sólo se suma la velocidad de esa persona por ello no se llama a la función "mayor" porque no es necesario comparar nada.

El resultado del programa completo se muestra en la figura 4.16.

Al ejecutar el programa dio dos soluciones pero observemos que son similares sólo que cambia el orden al cruzar a las personas, al igual que el "*COST*" dio el mismo resultado.

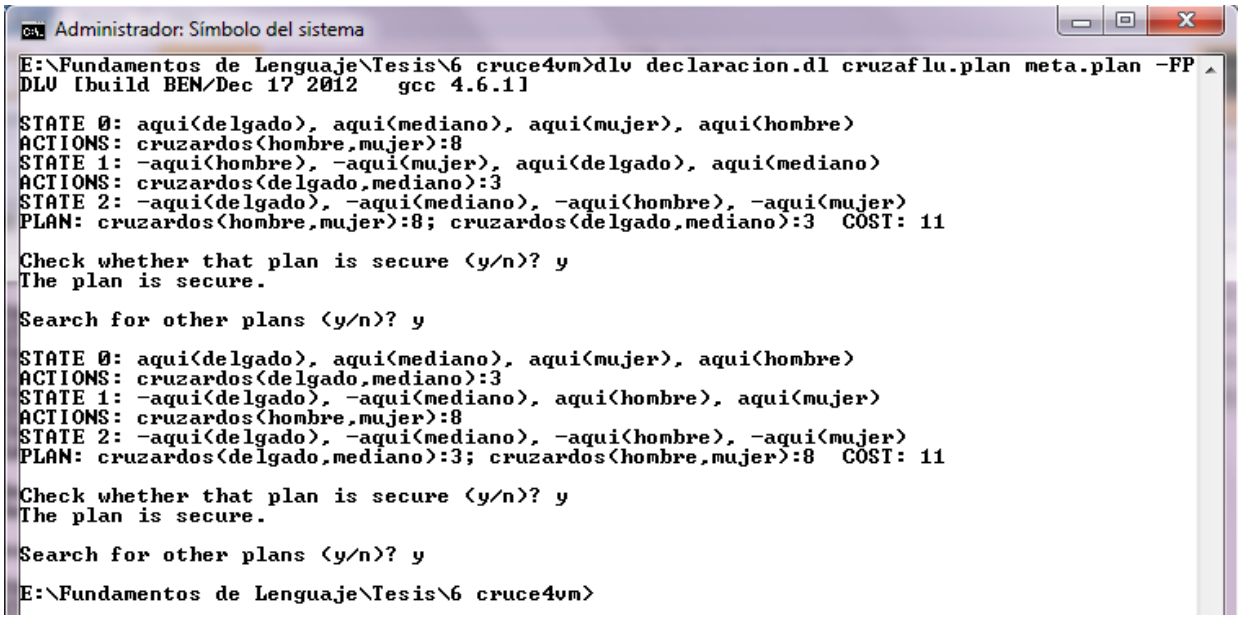


Figura 4.16 Resultado verificando velocidad mayor

Para el primer plan, en el *STATE 0* las cuatro personas se encuentran situadas en "aquí", la primera acción ejecutada es "cruzardos(hombre, mujer): 8", el número 8 es la velocidad mayor de los dos, como podemos recordar la persona hombre tarda en cruzar 8 segundos; en el *STATE 1* ya se ve que hombre y mujer están en "-aquí" y al realizar la nuevamente la acción que cruza a las otras dos personas "delgado y mediano" da como resultado de mayor velocidad 3, éste se suma al anterior gracias a la función que se implementó en el código, por lo tanto en el resumen del plan el costo total fue 11.

Siguiendo con las especificaciones del problema, dice que para cruzar un par de personas o solo una, deben llevar la lámpara, entonces ahora se agrega en el archivo "meta.plan" una función que nos diga quién es el que tendrá la lámpara en un inicio, en este caso será "delgado" porque es el que tiene la menor velocidad de todos y posiblemente será el que cruce más veces y en ocasiones cruzará solo. Todo lo anterior lo podemos ver en la figura 4.17.

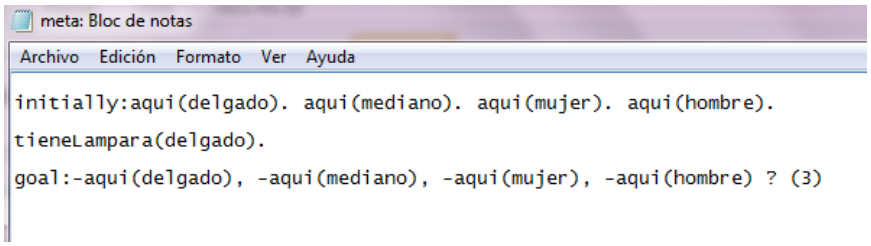


Figura 4.17 Declaración de "tieneLampara"

También en el archivo “cruzarflu.plan” se tiene que agregar la función en la sección de los fluentes especificando los parámetros que necesita, en la sección del *always* ahora se ejecutarán las acciones de “cruzardos” y “cruzar” si tienen la lámpara, para la primer acción ya sea que la primer persona tenga la lámpara o la segunda. Si este programa se ejecuta con esos cambios el plan no da que cruza pares de personas, pero en cada par el que cruza siempre es “delgado” porque en un inicio se declara que él tiene la lámpara y no hay otra condición todavía que nos diga que puede darle la lámpara a otra persona.

Por lo anterior es que se incluye en el programa la función “darlampara” en la sección *actions*, esta función requiere una variable X de tipo persona y en la sección *always* se especifica que se debe ejecutar esa acción o que no se ejecute si otra persona ya tiene la lámpara. “tieneLampara” y “darlampara” al usarlas causan que una persona tenga la lámpara después de que se la dieron o que no tenga la lámpara después de que se la dio a otra persona.

Aunque las últimas dos funciones fueron agregadas al código, todavía no se tiene un resultado óptimo al momento de la ejecución, es por eso se tuvo que pensar en otra función que diera solución a este problema. En los resultados anteriores vemos que al cruzar un par pasaba la misma persona, o que una cruzaba todas las veces con las demás personas, o por ejemplo si una persona ya estaba en “-aquí” cruzaba con una persona que estaba en “aquí” y eso no es posible porque para cruzar ambas personas deben estar en el mismo sitio, entonces se tiene que verificar cuales personas están en sitios distintos, para ello se agrega la función “sitiosDistintos” en los fluentes y requiere una variable X y una Y de tipo persona.

En la sección *always* ahora sí estará la opción que no se ejecute la acción “cruzardos” si las dos personas están en sitios distintos y también que no se ejecute “darlampara” si la persona ya tiene la lámpara y está en otro sitio, ya que tampoco es posible que si una persona ya cruzó le dé la lámpara a una que aún no ha cruzado.

Lo que causa que un par de personas están en sitios distintos es que la primera esté en el origen (aquí) y la segunda esté en el destino (-aquí) o al revés, que la primera este en el destino y la segunda en el origen.

Todo lo anterior se muestra en la figura 4.18

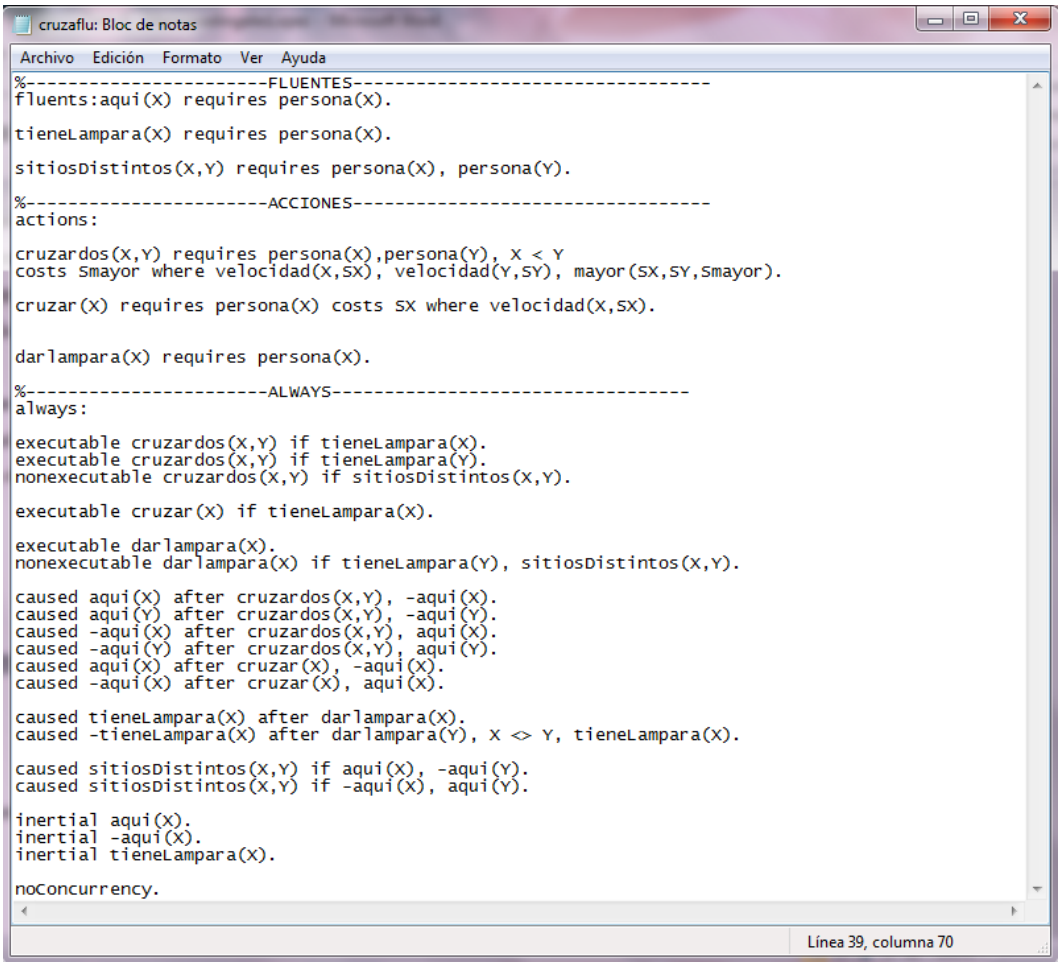


Figura 4.18 Código completo, contiene todos los fluentes, acciones, condiciones y causas

La salida del programa se muestra en la figura 4.19. En el *STATE 0* todas las personas se encuentran en el origen y el que tiene la lámpara es “delgado”, al ejecutar la acción “cruzados(delgado, mediano): 3” la velocidad mayor es de “mediano = 3 seg”; en el *STATE 1* se puede ver que “delgado y mediano” ya están en el destino y se muestran las combinaciones de las personas que se encuentran en sitios distintos, después se ejecuta “cruzar” ya que el que regresará es “delgado” y él es el que todavía tiene la lámpara, su velocidad (1 segundo) se suma a la del cruce anterior; en el *STATE 2* “delgado” se encuentra en el origen nuevamente y otra vez se hace la combinación de personas que están en lugares diferentes; la siguiente acción es que “delgado” le dio la lámpara a “hombre” y este cambio se muestra en el *STATE 3*; cruzan al destino dos personas “hombre y mujer” y la velocidad mayor que se suma a los otros cruces es 8 segundos,

por lo tanto en el *STATE 4* vemos que el único que se encuentra en el origen es "delgado", la lámpara la tiene aún la persona "hombre" es por eso que la persona anterior no podría cruzar porque no tiene la lámpara, en este estado las combinaciones de personas en sitios distintos también se hace; luego se le da la lámpara a "mediano" ya que es el que tiene menor velocidad y ese cambio se nota en el *STATE 5*, entonces regresa al origen y su velocidad (3 segundos) se suma; por último cruzan "delgado y mediano" y la velocidad del mayor se suma a todos los cruces anteriores, en el *STATE 7* ahora si las cuatro personas están en el destino, "mediano" se quedó con la lámpara y el costo de todos los cruces fue 18 segundos, es un buen resultado ya que es menor al tiempo que tarda encendida la lámpara (20 segundos).

Para finalizar se muestra el resumen del plan.

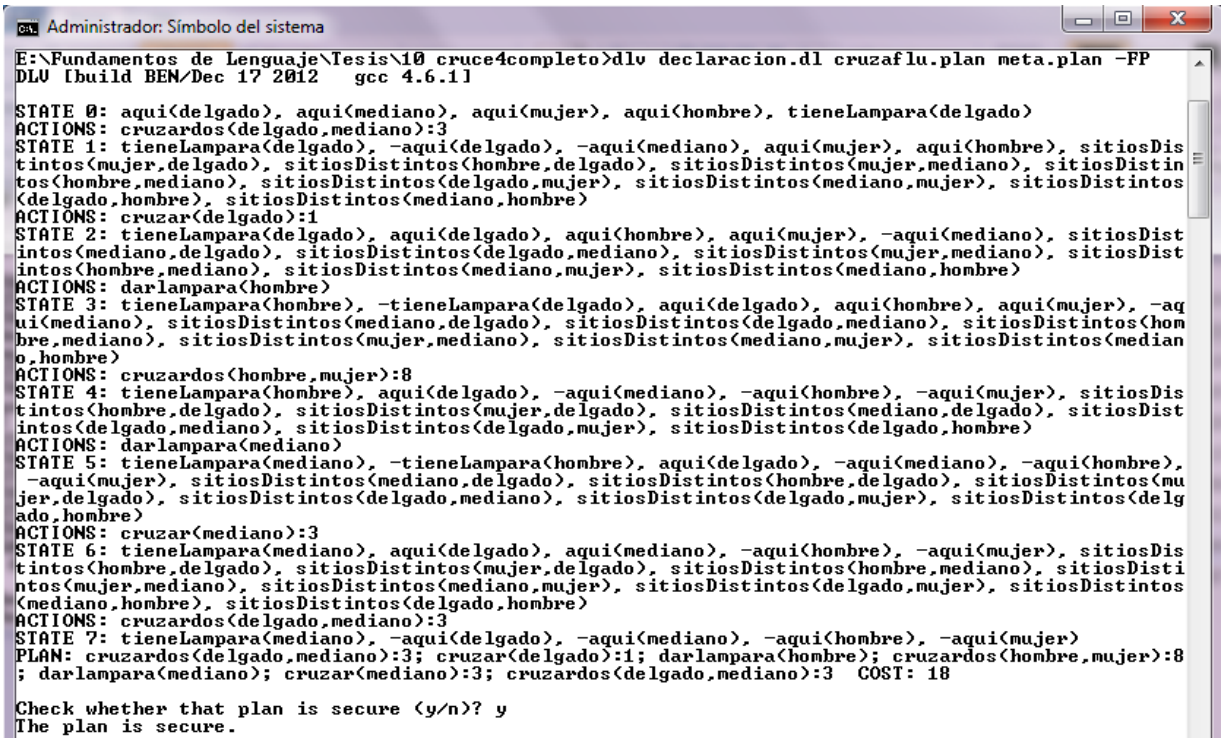
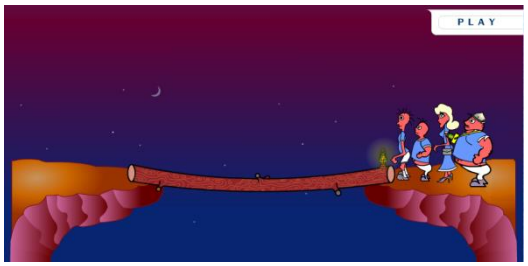


Figura 4.19 Plan optimo

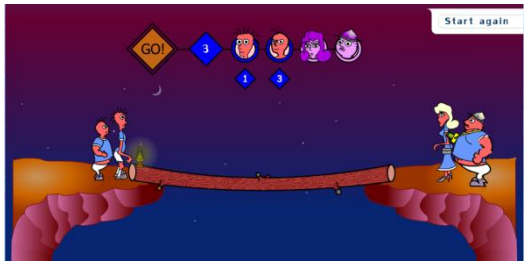
Para fines prácticos a continuación se muestra de forma gráfica algunos estados del plan anterior para tener mejor entendimiento de cómo se fueron desarrollando las acciones para llegar a la meta. En el *STATE 0* inicialmente la familia se encuentra en el lado derecho que es "aquí", al realizar la acción "cruzardos(delgado,mediano)" ahora en el *STATE 1* podemos observar que dichos miembros de la familia ya se encuentran del lado izquierdo, o sea en "-aquí"; en el *STATE 2* vemos que "mediano" aún se encuentra

en “-aquí” y “delgado” regresó al lado derecho ya que se realizó la acción “cruzar(delgado)”. Siguiendo en el *STATE 4* las personas “mujer” y “hombre” ya se encuentran en “-aquí” pero “delgado y “mediano” se encuentran en el lado derecho. Finalmente el *STATE 7* muestra a todos los miembros de la familia en el lado izquierdo y vemos que se logró alcanzar la meta de este problema.

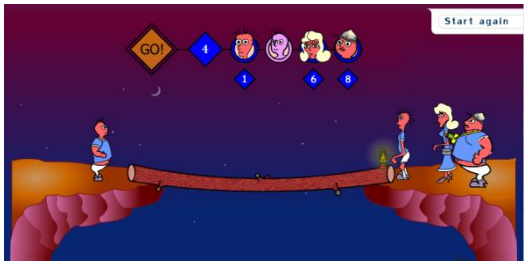
STATE 0:



STATE 1:

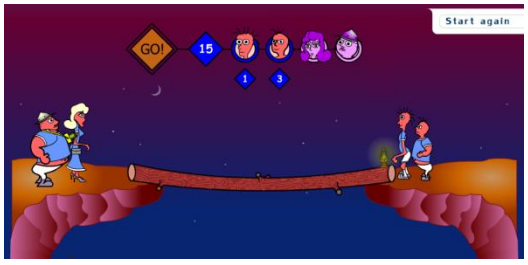


STATE 2:

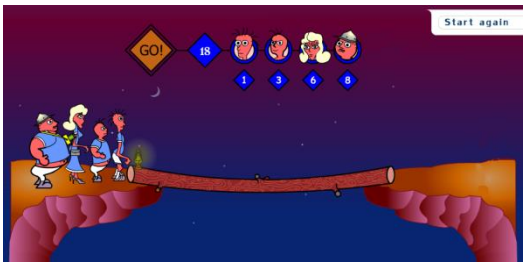


.
.
.

STATE 4:



STATE 7:



Trabajo a futuro

Para el problema “La travesía del puente”, el objetivo es que el número de personas a cruzar en el puente ahora sean 5 o 6, ya que por el momento sólo cruza a 4 personas. Si esto es posible se podrá lograr llevar el uso de la programación lógica al modelado de cualquier otro problema de planeación y que se considere como una opción este tipo de propuestas en áreas como la robótica o el desarrollo de agentes inteligentes.

El sistema permitirá caracterizar como se modela la planeación clásica desde lenguajes basados en la programación lógica como DLV^{K}, el cual se utilizó en esta tesis.

Conclusiones

En este trabajo de tesis la planeación fue un punto muy importante en todo, ya que para alcanzar cualquier objetivo si se tiene una buena planeción podremos seguir un camino optmimo para que haya mayor probabilidad de que se tenga éxito. La planeación declarativa clásica es la que nosotros usamos para poder encontrarle solución a nuestro problema.

Al inicio se presentaron las bases teoricas de *Answer Set Progamming* (ASP), DLV y DLV^K con el fin de dar a conocer las características de cada uno.

Como el paradigma ASP está basado en programación lógica al igual que la planeación declarativa clásica, vimos que es más sencillo el cálculo de *answer sets* para la representación del conocimiento, ya que la teoría no es dificil de entender y que para ello se implementó software que ayuda a cumplir esta tarea, una de las implementaciones de ASP en software y que nosotros trabajamos fue DLV, conocimos su sintaxis para conocer como se modela en este sistema.

Se utilizó el front-end DLV^K de DLV para el lenguaje de acción K, con el fin de modelar los problemas de planeación y con ayuda de un ejemplo se estudio la sintaxis del mismo. Empezamos por definir cuales eran los elementos importantes para describir nuestro entorno y poder colocarlos en las secciones que conforman el programa, se declararon las variables necesarias para solucionar el problema, los fluentes, las acciones, las causas que tendrían esas acciones y las condiciones para ejecutarlas; también definimos el estado inicial y la meta.

Con ayuda de todo lo anterior se pudo alcanzar el objetivo principal de esta tesis que es darle solución al problema “La travesía del puente”, la ejecución del programa nos dio el mejor plan para cruzar a los cuatro integrantes de la familia al otro lado del puente y que no se excediera el tiempo que se mantenía encendida la lámpara, de hecho el costo de todo el plan fue 18 segundos, dos segundos menos que el tiempo límite.

Como nos podemos dar cuenta con el lenguaje de acción K tiene una semántica formal clara con una base lógica, lo que es útil para las tareas de razonamiento, fácil aprender, de entender y de implementar para representar de manera más simple los problemas de

planeación. A diferencia de otros programas en DLV^K el código se puede simplificar y no es necesario que todos los objetos del mundo sean declarados como lo pudimos ver en el ejemplo del “Acertijo del pastor” no fue necesario declarar el bote, el rio ni al mismo pastor para que el problema encontrara la solución. También el tiempo de ejecución no es mucho como si se ejecutara un problema en otro lenguaje de programación.

Referencias

[1] Stuart Russell and Peter Norvig. "*Inteligencia Artificial: Un enfoque Moderno*", Prentice Hall, 2002.

[2] Kowalski Robert, Logic for Problem Solving. *Artificial Intelligence series*. The computer science library, 1990.

[3] CHAN, S. Y. *The use of graphs as decision aids in relation to information overload and managerial decision quality*. Journal of Information Science, v. 27, n. 6, p. 417-425, 2001.

[4] McCarthy John. *Programs with common sense*. In Proc. Teddington conference on the Mechanization of Thought Processes. pp. 75 – 91 London 1959.

[5] Ghallab, M., Howe, A., Knoblock, C., McDermott, D., Ram, A., Veloso, M., Weld, D., & Wilkins, D. *PDDL — The Planning Domain Definition language (Tech. Rep.)*. Yale Center for Computational Vision and Control. (Available at http://www.cs.yale.edu/pub/ mcdermott/software/pddl.tar.gz), 1998.

[5] Niemelä, Ilkka & Simons, Patrik. *SMODELS – an implementation of the stable model and well-founded semantics for normal logic programs*. 10.1007/3-540-63255-7_32. 2006.

[6] Eiter T., Faber W., Leone N., Pfeifer G., Polleres A. *System Description: The DLVK Planning System*. In: Eiter T., Faber W., Truszczyński M.. (eds) Logic Programming and Nonmotonic Reasoning. LPNMR 2001. Lecture Notes in Computer Science, vol 2173. Springer, Berlin, Heidelberg. 2001.

[7] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. *A deductive system for nonmonotonic reasoning*. In J. Dix, et. al. Proc. Of 4th International Conference on Logic Programming and Nonmonotonic Reasoning, number 1265 in Lectures Notes in AI, pp. 363-374, Springer, Berlín 1997.

[8] W. Faber, N. Leone and G. Pfeifer. *Pushing Goal derivation in DLP computations*. In Proc. Of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning, LNAI, Springer, USA, December 1999.

[9] JENNINGS, N.; WOOLDRIDGE, M. Agent Technology - Foundations, Applications, and Markets. Springer-UNICOM. 1998.

[10] Answer Set Programming. https://en.wikipedia.org/wiki/Answer_set_programming

[11] M. Osorio and J.A. Navarro and J. Arrazola, *Applications of Intuitionistic Logic in Answer Set Programming (Extended version)*, TPLP 2003.

[12] J.J. Alferes and J.A. Leite and L.M. Pereira and H. Przymusinska and T.C.Przymusiski. *Dynamic logic programming*, *Procs*, of 6th the International Conference on Principles of Knowledge Representation and reasoning(KR 98). Anthony Cohn and Lenhart Schubert and Stuart Shapiro, Morgan Kaufmann Publishers

[13] T. Eiter and M. Fink and G. Sabattini and H. Thompits. *Considerations on Updates of Logic Programs.* Proceedings of the seventh European Workshop on Logic in Artifitial Intelligence (JELIA 00), 2000, M.O. Aciego and I.P. de Guzman and G. Brewka and L. M. Pereira, Springer LNAI, Vol. 1, 1919.

[14] M. Gelfond and V. Lifschitz. *The Stable Model Semantics for Logic Programming*, 5th Conference on Logic Programming, 1988, R. Kowalski and K. Bowen, MIT Press.

[15] Wooldridge, M. y Jennings, N. R. Intelligent agents: theory and practice. http://pattie.www.media.mit.edu/people/pattie/CHI97/sld001.htm

[16] David Pearce. *Equilibrium logic, Annals of Mathematics and Artificial Intelligence* 47 (2006), 3-41.

[17] Leone, Nicola & Pfeifer, Gerald & Faber, Wolfgang & Calimeri, Francesco & Dell'Armi, Tina & Eiter, Thomas & Gottlob, Georg & Ianni, Giovambattista & Ielpa, Giuseppe & Koch, Christoph & Perri, Simona & Polleres, Axel. (2002). *The DLV system*. 537-540. 10.1007/3-540-45757-7_50.

[18] Gelfond M. and Lifschitz V. *The Stable Model Semantics for Logic Programming*, 5th Conference on Logic Programming, R. Kowalski and K. Bowen, Mit Press. 1988

[19] Leone N. The DLV Project: A disjunctive Datalog System http://www.dbai.tuwien.ac.at/proj/dlv/

[20] Bihlmeyer, Robert & Faber, Wolfgang & Ielpa, Giuseppe & Lio, Vincenzino & Pfeifer, Gerald. *Manual de usuario de DLV en línea*. http://www.dlvsystem.com/html/DLV_User_Manual.html

[21] J.J. Alferes and J.A. Leite and L.M. Pereira and H. Przymusinski and T.C.Przymusinski, *Dynamic logic programming*, Procs, of 6th the International Conference on Principles of Knowledge Representation and reasoning(KR 98), Anthony Cohn and Lenhart Schubert and Stuart Shapiro, Morgan Kaufmann Publishers

[22] M. Gelfond and V. Lifschitz, *The Stable Model Semantics for Logic Programming*, 5th Conference on Logic Programming, 1988, R. Kowalski and K. Bowen, MIT Press.

[23] Leone, Nicola & Pfeifer, Gerald & Faber, Wolfgang & Perri, Simona & Alviano, Mario & Terracina, Giorgio. (2005). *The Disjunctive Datalog System DLV**. Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy.

[24] Bierman, H. S. y L. Fernández, *Game Theory with economic applications*, Addison-Wesley, 1998.

[25] Gintis, Herbert (2000): *Game Theory Evolving*. Princeton University Press, ISBN 0691009430

[26] Eiter, Thomas & Faber, Wolfgang & Pfeifer, Gerald & Polleres, Axel. (2004). *Declarative Planning and Knowledge Representation in an Action Language.* 10.4018/9781591404507.ch001.

[27] T. Eiter and W. Faber and N. Leone and G. Pfeifer and A. Polleres. *A Logic Programming Approach to Knowledge-State Planning, II: The DLV^K System*. Institut für Informationssysteme, Abteilung Wissensbasierte Systeme, INFSYS Research Report 1843-01-11. December 2001; October 2002

[28] Héctor Luis Palacios Verdes. (2009). Translation-based approaches to Conformant Planning (Tesis Doctoral UPF). Universitat Pompeu Fabra, Barcelona, España

[29] T. Eiter and W. Faber and N. Leone and G. Pfeifer and A. Polleres, A Logic Programming Approach to Knowledge-State Planning: Semantics and Complexity, Institut für Informationssysteme, Abteilung Wissensbasierte Systeme, INFSYS Research Report 1843-01-11. December 2001; October 2002