BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

"MODELADO DE PREFERENCIAS UTILIZANDO OPERADORES DE OPTIMIZACIÓN EN ANSWER SET PROGRAMMING."

TESIS PARA OBTENER EL TÍTULO DE

LICENCIADA EN INGENIERÍA EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:

ELIZABETH SEDEÑO OSORIO

ASESOR: DRA. CLAUDIA ZEPEDA CORTÉS

Índice General

Introducción

Las preferencias nos acompañan en nuestra vida diaria, a cada individuo, según la Real Academia Española preferir es la elección de alguien o algo entre varias personas o cosas, así cada uno tiene sus propias preferencias de música, moda, sabores, colores, aromas, etc. Algunas preferencias están sobre otras, es decir nos puede gustar el helado de chocolate y fresa pero preferir el de fresa. El concepto de preferencia surge de la modelación de disyuntivas a las que se ve enfrentado cada individuo diariamente, durante toda nuestra vida.

Actualmente el concepto de preferencia se ha desarrollado de diferentes áreas de investigación, por ejemplo, en la teoría de decisión matemática, las preferencias (frecuentemente expresadas como utilidades), usadas para el modelo económico de comportamiento humano. En inteligencia artificial, los agentes se apoyan de las preferencias para obtener sus objetivos, En bases de datos, las preferencias ayuda a la reducción de la cantidad de información regresada en la respuesta de las consultas de usuario. En filosofía, las preferencias son usadas para el razonamiento sobre los valores, deseos y deberes. En computación, el interés está en la reducción al mínimo de los recursos de cómputo (tiempo, espacio, comunicación, etc. para realizar una tarea bajo ciertas características [1].

Ahora bien, si deseamos optimizar o sistematizar dichas preferencias entonces nos dirigimos a la programación lógica que permite formalizar hechos del mundo real.

La contribución de este trabajo se enfoca en comprender el funcionamiento de los operadores de optimización Minimize y Maximize en problemas de preferencia usando el solucionador de answer set `clasp`, que pertenece a la suite de Potassco.

Esta tesis tiene como objetivo principal analizar y comprender el funcionamiento de los operadores de optimización para modelar problemas de modelado de preferencias de una implementación de Answer Set Programming(ASP), en este caso la que se usará es Potassco y los operadores a analizar y utilizar serán Minimize y Maximize.

Se modelaron un conjunto de problemas de preferencias en ASP utilizando operadores de optimización para después poder calcular su answer set, lo que se quiere es entender cómo funciona `clasp`, como es que resuelve los programas y obtiene el answer set óptimo.

Se analizaron y discutieron los resultados basados en la experiencia obtenida durante el desarrollo del modelado y solución del problema, tomando en cuenta los criterios que son te interés para llevarlas a cabo tales como: la facilidad de codificación, la facilidad de modificar un programa para obtener nuevos y mejores resultados, que tan entendible es la lectura de cada una de las codificaciones, la facilidad de traslación de un código a otro.

En el capítulo 1 se describen conceptos de programación lógica, lógica G'_3, semántica estable, answer set programming, se describen los diferentes solucionadores de ASP, la optimización en `clasp`, las preferencias que serán parte fundamental de este trabajo, así como una explicación de cómo se modelan problemas en ASP usando los operadores de optimización.

En el capítulo 2 se enuncian los problemas de preferencias que son modelados en ASP, se muestra su solución y la explicación a cada answer set arrojado por `clasp`, así como el cálculo de los valores de optimización. Todos los problemas tienen la misma estructura para su mejor comprensión, y son explicados claramente, el modelado se lleva a cabo paso a paso con una explicación del porqué de cada línea, y los resultados son analizados y explicados al final de cada problema.

En el capítulo 3 se describen los resultados obtenidos del análisis realizado a los problemas de preferencias modelados, se da una breve explicación del funcionamiento de `clasp` en problemas de preferencias usando los operadores de optimización #minimize y #maximize.

Se mencionan sus restricciones y fallos, también se menciona brevemente y sin profundizar que es lo que usa `clasp` para arrojar los answer sets.
En el capítulo 4 están las conclusiones obtenidas en el desarrollo de este trabajo. También se presentan las propuestas de trabajos futuros. La contribución de este trabajo está en los capítulos 3 y 4, que es en donde radica el cumplimiento de los objetivos planteados.

En el apéndice A presenta el código de los problemas de preferencia completos, aunque en el capítulo 3 al final de cada problema se presenta el código completo, el usado con clasp lleva una línea o más líneas para que no se muestren todos los datos, sólo el resultado.

En apéndice B presenta las conversaciones que se tuvieron con el soporte técnico de Potassco, dichas conversaciones se llevaron a cabo para una mejor comprensión del funcionamiento de `clasp`. Todas las conversaciones se pueden ver también en [26], ya que en el apéndice B sólo se presentan las recibidas vía correo electrónico.

Capítulo 1

Marco teórico

En este capítulo se describen conceptos básicos de programación lógica, lógica $\boldsymbol{G'_3}$, semántica estable se hace una descripción de los diferentes solucionadores (solvers). En particular se profundiza en `clasp` que es el solucionador que se usa en esta tesis, el que estudiamos a detalle, con el que se prueban diferentes problemas de preferencia, se hace una descripción de los operadores de optimización que son usados en el conjunto de problemas de preferencia modelados en el siguiente capítulo, se muestra un ejemplo tomado de [25], el cuál es un problema de preferencia que usa los operadores de optimización, en el que nos basaremos para explicar más adelante como se modelan problemas usando ASP y los operadores Maximize y Minimize.

Las tres primeras secciones fueron tomadas de [2]. Estás primeras secciones nos dan las bases para el modelado de los problemas de preferencias. En la sección 1.4 de este capítulo se explica que es ASP, las aplicaciones en el área de Inteligencia Artificial. En la sección 1.5 se presentan tres solucionadores (solvers) de ASP: DLV, SMODELS y `clasp`. Este último es el que estudiaremos y analizaremos. En la sección 1.6 se describe más a detalle la optimización en `clasp`, en un subsección de esta sección se describe el uso de los operadores de optimización #minimize y #maximize, y se agrega un ejemplo descrito a detalle para su mejor comprensión y modificación, que a la vez sirve como base para el modelado de todos los problemas que se van a analizar. La última sección habla de las preferencias, en las cuales estarán basados nuestros problemas.

1.1 Programación lógica

Una firma $\mathcal{L}$ es un conjunto finito de elementos que llamamos *átomos,* o símbolos *proposicionales* [2]. El lenguaje de la lógica proposicional tiene un alfabeto que consiste de

Símbolos proposicionales: $p_0, p_1 \ldots;$
Conectivos: $\wedge, \vee, \leftarrow, \neg;\ and$
Símbolos auxiliares: (,),

Donde $\wedge, \vee, \leftarrow$ son conectivos binarios y $\neg$ es conectivo unario. Las formulas son construidas así usualmente en lógica. Un *literal* puede ser un átomo *a,* llamada *literal positiva;* o la negación de un átomo $\neg a$, llamada *literal negativa.*
Una clausula *normal* es una clausula de la forma

$$a \leftarrow b_1 \wedge \ldots \wedge b_n \wedge \neg\, b_{n+1} \wedge \ldots \wedge \neg b_{n+m}$$

Donde a y cada b_i son átomos para $1 \leq i \leq n + m$. En un pequeño abuso de la notación

$$a \leftarrow B^+ \cup \neg B^-$$

Donde el conjunto $\{b_1, \ldots, b_n\}$ será denotado por B^+ , y el conjunto $\{b_{n+1}, \ldots, b_{n+m}\}$ será denotado por B^-. Definimos un *normal programa P,* como un finito conjunto de clausulas normales.

El cuerpo de una clausula normal podría estar vacía, en cuyo caso la cláusula es conocido como un hecho y se puede denotar simplemente como:$a \leftarrow$

Dado un conjunto de átomos de M y una firma $\mathcal{L}$, definimos$\neg\widetilde{M} = \{\neg a \mid a \in \mathcal{L} \setminus M\}$
Limitaremos nuestra discusión a los programas proposicionales, damos por sentado que los programas con símbolos de predicados son únicamente para abreviar el ground del programa.

Finalmente, damos dos definiciones antes de definir la semántica de stable y p-stable para programas normales.

Definición 1. [3] Sea P un programa normal y *M* un conjunto de átomos. Definimos $RED(P,M) = \{a \leftarrow B^+ \cup \neg(B^- \cap M) \mid a \leftarrow B^+ \cup \neg B^- \in P\}$

Ejemplo 2. *Tomando el siguiente programa normal P*

$$b \leftarrow \neg a.$$
$$a \leftarrow \neg b.$$
$$p \leftarrow \neg a.$$
$$p \leftarrow \neg p.$$
$$c \leftarrow p.$$

x	$\neg x$
0	2
1	2
2	0

$\rightarrow$	0	1	2
0	2	2	2
1	0	2	2
2	0	1	2

Tabla 1. Tabla de verdad de las conectivas en G'_3

Dado $M = \{a, p\}$,$RED(P,M)$

$$b \leftarrow \neg a.$$
$$a \leftarrow.$$
$$p \leftarrow \neg a.$$
$$p \leftarrow \neg p.$$
$$c \leftarrow p.$$

Para cualquier programa *P,* la parte positiva de *P,* denotada por *POS(P)* es el programa consiste exclusivamente de esas reglas en *P* que no tienen literales negativas.

1.2 Lógica G'_3

Como mencionamos en la introducción de la sección, la semántica p-stable para programas normales basados en terminaciones débiles fue definida con tres valores lógicos llamada lógica G'_3en [3]. Por esta razón, damos una breve descripción de esta lógica.

La lógica G'_3 es definida como una lógica de valor 3 con valores verdaderos en el dominio $D = \{ 0, 1, 2 \}$ donde 2 es el valor designado. La función de evaluación de los conectivos lógicos es definido como sigue:

$$x \wedge y = \min(x, y)$$
$$x \vee y = \max(x, y)$$

Los conectivos ¬ y → son definidos de acuerdo a la tabla de verdad dada en la *Tabla 1.*
En [4], los autores ofrecen una axiomatización de la lógica G'_3juntocon una solidez y teorema de lo completo, es decir, cada teorema es una tautología y viceversa.

1.3 Semántica estable

Desde ahora, asumimos que el lector es familiarizado con el concepto modelo clásico mínimo.
Ahora, damos la definición de la semántica estable para programas normales [2].

Definición 3. [5] *Sea P un programa normal y* $M \subseteq \mathcal{L}_p$. *Supongamos que* $P^M = POS(RED(P, M))$ *, entonces decimos que M es un modelo estable de P si M es un modelo clásico minio de* P^M.

Ejemplo 4. Sea P el siguiente programa normal:

$$b \leftarrow \neg a.$$
$$a \leftarrow \neg b.$$
$$p \leftarrow \neg a.$$
$$p \leftarrow a.$$

Y sea $M\{a, p\}$*, entonces tenemos* $RED(P, M)$ *es el siguiente programa:*

$$b \leftarrow \neg a.$$

$$a \leftarrow.$$

$$p \leftarrow \neg a.$$

$$p \leftarrow a.$$

Y $POS(RED(P, M))$ *es el programa*

$$a \leftarrow.$$

$$p \leftarrow a.$$

El conjunto M*es un modelo clásico minimo de* $POS(RED(P, M))$*. Por lo tanto de acuerdo a la definición,* $M = \{a, p\}$ *es un modelo estable de P.*

1.4 Answer Set Programming

Answer Set Programming (ASP) es un paradigma de programación declarativa orientado a tareas en conocimiento intensivo y problemas combinatorios de búsqueda. Su idea principal es reducir la búsqueda del problema para resolver modelos estables de un programa lógico, y para usar solucionador ASP para realizar la búsqueda.ASP es particularmente adecuado para modelos incompletos, inconsistentes y de dominio dinámico. Se ha convertido en una importante representación del conocimiento y se ha aplicado a varias áreas en AI (Inteligencia Artificial) como la planeación, el diagnostico, integración de la información y la bioinformática. Extensas aplicaciones de ASP motivan varias extensiones del lenguaje e implementaciones, incluyendo integraciones de ASP con restricciones de satisfacción y descripciones lógicas. Recientemente, la semántica del modelo estable, una base matemática de ASP, es muestra que es estrechamente relacionada con la lógica clásica [6].

1.5 Solvers de ASP

Actualmente podemos encontrar diferentes sistemas de software para calcular la semántica estable tales como: DLV [7], SMODELS [7] y `Clasp` [8], esta última pertenece a Potassco.

DLV (datalog con disyunción) es un poderoso sistema de base de dados deductivo disponible gratuitamente. Es basado en el lenguaje de programación declarativo datalog que es conocido por ser una herramienta conveniente para la representación de conocimiento. Es un traductor de bases de datos deductivas y puede verse por consiguiente como una manera de realizar consultas en bases de dados, pero también se describe como un sistema para Answer Set Programming.

Smodels consta de dos partes, smodels y lparse. La primera parte, es la programación lógica actual que hace el trabajo dura y lparse sólo agrega una capa sintáctica en el tope de este. Smodels es desarrollado en el Laboratorio de Ciencias Computacionales de Teóricas de la Universidad de Helsinki por PAtrik Simons, y el lparse fue desarrollado por Tommi Syrjänen [7].

`Clasp` es un sistema solucionador de Answer Sets normal que forma parte de la suite de los laboratorios de Potassco. `Clasp` está originalmente diseñado y optimizado para el manejo de conflicto para la solución de ASP. Para ello, cuenta con cuenta con una serie de sofisticadas técnicas de razonamiento y aplicación, algunos específicos de ASP y otros tomados de CDCL basada en solucionadores SAT. Combina las capacidades de modelado de alto nivel de ASP con técnicas de estado del arte de la zona de problemas de restricción Boolean. El principal algoritmo de `clasp` se basa en el manejo de conflicto de nogood learning, una técnica que fue un éxito para la comprobación de satisfabilidad (SAT).

A diferencia de otros solucionadores de aprendizaje ASP, `clasp` no se basa en el software de legado, como un solucionador SAt o cualquier otro programa de solución existente. Más

bien `clasp` ha sido realmente desarrollado para resolver answer sets basados en el nogood learning[24].

`Clasp` provee diferentes modos de razonamiento:

- Enumeración de (Proyectos) soluciones
- Optimización de soluciones

1.6 Optimización con clasp

`Clasp` es originalmente diseñado y optimizado para conflicto de manejo de solución de ASP, como se describe en [9]. Para ello, cuenta con una seria de sofisticadas técnicas de razonamiento y aplicación, algunos específicos de ASP y otros tomados de solucionadores CDCL-basedSAT.

`Clasp` aplica el enfoque de solución ASP, combina las capacidades de modelado de alto nivel de ASP con las técnicas del estado del arte del área de las restricciones de solución Booleana.

Los answer set pueden ser resueltos por clasp, que es un programa de solución.

1.6. 1 Operadores #minimize y #maximize

Las declaraciones de optimización se derivan de la pregunta básica de si un conjunto de átomos es un answer set si este es un conjunto óptimo de respuesta. Gringo y clingo adoptan las declaraciones de optimización de *lparse*, indicado a través de las palabras clave #maximize y #minimize. Como una declaración de optimización no admite un cuerpo, cualquier variable (local) que debe ocurrir en un átomo(a través de un dominio o incorporando un predicado) del lado derecho de la condición dentro de la declaración de optimización.

Una declaración de optimización tiene la forma:

$$\text{opt } [L_1 = w_1@p_1; : : : ; L_n = w_n@p_n \}$$

$$\text{opt } \{L_1@p_1; : : : ; L_n@p_n \}$$

donde *opt* es #maximize o #minimize, L_i son literales cona pesos w_i (enteros) *y* prioridades p_i (enteros).

La semántica de una declaración de optimización es intuitiva: un answer set es *óptimo* si la suma de sus valores de las literales es el máximo o mínimo, como se requiere en la declaración, entre todos los answer sets. Esta definición es suficiente si una declaración de optimización es especificada junto con un programa lógico. Si ocurren diferentes prioridades, entonces depende del tipo de declaración de optimización, el answer set cuya suma de los valores asignados a las prioridades más alta es maximizado o minimizado.

Tomando en cuenta la compatibilidad con *lparse*, si varias instrucciones se optimizaban, por defecto se asignan prioridades. La declaración n-ésima tiene prioridad n, por lo tanto, las declaraciones posteriores tienen mayor prioridad. Se sugiriere que si desea utilizar más de una sentencia de optimización, especificar siempre las prioridades para que el programa sea más legible y el orden independiente.

Ejemplo. Para ilustrar la optimización, se considera una situación de reserva de hotel, donde queremos elegir uno de los cinco hoteles disponibles. Los hoteles se identifican mediante números asignados en orden descendente de estrellas. Por supuesto, entre más estrellas tiene un hotel, más que cuesta la noche. Como información complementaria, se sabe que el hotel 4 se encuentra en una calle principal, por lo que esperamos que las habitaciones sean ruidosas.

Inicializamos los cinco hoteles que se quieren elegir:

```
1 { hotel(1..5) } 1.
```

Se muestran los hoteles con su respectivo número de estrellas

```
star(1,5). star(2,4). star(3,3). star(4,3). star(5,2).
```

Ahora se muestran los hoteles con sus precios.

```
cost(1,170). cost(2,140). cost(3,90). cost(4,75). cost(5,60).
```

Decimos que el hotel 4 esta en la calle principal

```
main_street(4).
```

Ahora decimos que el hotel X que esta en la calle principal es ruidoso

```
noisy :- hotel(X), main_street(X).
```

Como se prefiere el hotel con el mayor número de estrellas, entonces decimos que queremos maximizar el número de estrellas con el operador `#maximize,` y en sus parámetros ponemos que se prefiere un `hotel(X)` tomando en cuentas el número de estrellas `star(X,Y)` y como las prioridades van de forma descendente asignamos prioridad 1 con `@ 1`, de esta manera decimos que no es tan importante preferir el hotel, ya que hay otras situaciones a considerar. Por lo tanto nuestra línea quedaría así:

```
#maximize [ hotel(X) : star(X,Y) = Y @ 1 ].
```

Ahora queremos ahorrar dinero, es decir queremos minimizar el costo del hotel que deseamos elegir. Por esto, hacemos uso del operador `#minimize,` y decimos que preferimos un `hotel(X),` tomando en cuenta el costo `cost(X,Y)` pero también el número de estrellas `star(X,Z)` y le damos una prioridad 2 `@2,` porque es más importante minimizar el costo que preferir un hotel con el mayor número de estrellas, pero aún hay que considerar otra situación, el ruido que existe en la calle principal.

```
#minimize [ hotel(X) : cost(X,Y) : star(X,Z) = Y/Z @ 2 ].
```

Finalmente preferimos un hotel donde no haya ruido y sea tranquilo, y como en la calle principal hay ruido, debemos restringir la elección del hotel ubicado en esa calle, por eso ahora minimizamos el ruido y usamos `#minimize,` y en sus parámetros ponemos la regla que anteriormente citamos sobre el ruido `noisy` , y le damos una prioridad mayor que a las otras declaraciones de optimización `@ 3,` porque preferimos un lugar tranquilo para hospedarse, a un hotel con el mayor número de estrellas o el menor costo.

```
#minimize { noisy @ 3 }.
```

El código completo queda de la siguiente manera:

```
1    1 { hotel(1..5) } 1.
2    star(1,5). star(2,4). star(3,3). star(4,3). star(5,2).
3    cost(1,170). cost(2,140). cost(3,90). cost(4,75).
     cost(5,60).
4    main_street(4).
5    noisy :- hotel(X), main_street(X).
6    #maximize [hotel(X) : star(X,Y) = Y @ 1].
7    #minimize [hotel(X) : cost(X,Y) : star(X,Z) = Y/Z @ 2].
8    #minimize { noisy @ 3 }.
```

Líneas 6-8 contribuyen a las declaraciones de optimización en orden inverso de importancia, de acuerdo con el cual queremos elegir el mejor hotel para reservar. La sentencia más importante en la optimización de la línea 8 establece que evitar el ruido es nuestra prioridad principal. El criterio de optimización secundario en la línea 7 consiste en minimizar el coste por estrella. Por último, la tercera declaración de optimización en la línea 6 especifica que queremos maximizar el número de estrellas especifica que queremos maximizar el número de estrellas entre los hoteles. Las declaraciones de optimización en las Lineas 6-8 se crea una instancia de la siguiente manera:

```
6    #maximize [ hotel(1)=5@1, hotel(2)=4@1,
```

```
      hotel(3)=3@1, hotel(4)=3@1, hotel(5)=2@1 ].
7     #minimize [ hotel(1)=34@2, hotel(2)=35@2,
       hotel(3)=30@2, hotel(4)=25@2, hotel(5)=30@2 ].
8     #minimize [ noisy=1@3 ].
```

Utilizando `clasp` para calcular un `answer set` óptimo, nos encontramos con que el hotel 4 no es elegible porque implica ruidoso. Así, hotel de 3 y 5 permanecen como óptima, la segunda declaración de optimización más importante en la línea 7. Este lazo se rompe a través de al menos una declaración de optimización significativa en la Línea 6, porque tiene un hotel de 3 estrellas, es más que un hotel 5. Por lo tanto decide reservar hotel 3 que ofrece 3 estrellas con un costo de 90 por noche.

1.7 Preferencias

Las preferencias son un concepto fundamental que guía nuestras decisiones y acciones. Elegir el color de un coche o elegir entre los préstamos hipotecarios son diferentes ejemplos en que las preferencias pueden guiar a los humanos por una simple de decisiones muy importantes. Las preferencias son un tema multidisciplinario y han sido ampliamente estudiados en economía, psicología, filosofía, lógica y otras disciplinas centradas en el ser humado. Sin embargo, son un tema relativamente nuevo en Inteligencia Artificial (IA) [10,11].

Las preferencias pueden proporcionar una manera efectiva para elegir entre las mejores soluciones a un problema. Dichas soluciones pueden representar los estados del mundo más estimables cuando hablamos de representación de información incompleta, los estados del mundo más satisfactorios cuando hablamos de preferencias de usuario, o decisiones óptimas cuando estamos hablando de tomas de decisión con incertidumbre.

El uso de las preferencias ha beneficiado a dominios, como razonamiento en presencia de información incompleta e incierta, modelado de preferencias, y toma de decisión con

incertidumbre. En la literatura, distintos enfoques simbólicos de razonamiento no clásico han sido creados. Entre ellos, la programación lógica con la semántica de *answer set* ofrece un buen acercamiento entre representación y procesamiento simbólico del conocimiento, y diferentes extensiones para manejar preferencias.

La noción de preferencia es generalizada en el razonamiento común, en parte porque las preferencias constituyen una natural y efectiva manera de resolver situaciones indeterminadas. En la toma de decisiones, por ejemplo, se pueden tener varias desideratas, de las cuales no todas pueden cumplirse simultáneamente, en tal situación, las preferencias entre desideratas pueden permitir llegar a una solución adecuada.

En la lógica basada en Inteligencia Artificial, un enfoque estándar para el manejo de preferencias es tener un sistema de razonamiento no monotónico y, de una manera o de otra, dotarlo de preferencias. Por ejemplo, las preferencias son agregadas de tal manera a la lógica por default [12,13], la lógica autoespisctica [14,15], circunscripción [16,17], y lógica de programación[18,19].

Sin embargo, aunque la noción de "preferencia" es intuitivamente fácil, hay una variedad sorprendente como esta noción se realiza en varios enfoques. Por lo tanto, algunos enfoques toman una preferencia ordenada como la expresión de una "conveniencia" que una propiedad es adoptada mientras en otros el ordenamiento expresa el orden con en el que las propiedades(o lo que sea) se han de considerar. Al describir más adelante, algunos enfoques combinan la noción de herencia de las propiedades con la noción general de la preferencia. El resultado, por supuesto que es, dependiendo de cómo la noción de preferencia es interpretada, las diferentes conclusiones pueden ser aproximadas. Al mismo tiempo, mientras la manipulación de la preferencia lógica constituya un indispensable significado para los sistemas de razonamiento legal (cf. [20,21]), también se está usando en otras áreas de aplicación como agentes inteligentes y comercio electrónico [22], y la resolución de ambigüedades gramaticales [23].

Capítulo 2

Modelado de problemas de preferencia.

En este apartado se muestran siete problemas de preferencias, el primero es el único que tiene una variante, es decir, se modificó el primer problema y así se generó el segundo. Estos problemas son solucionados con `clasp` usando los operadores #minimize y #maximize. Los siete problemas son simples y fáciles de entender y modificar. Los problemas tienen entre tres y siete opciones, `clasp` tiene que encontrar la mejor de acuerdo a las declaraciones de optimización que se escriba, es decir respecto a las condiciones que el problema presente. Todos los ejemplos cuentan con la misma estructura para su fácil manipulación y modificación. La estructura de cada problema es la siguiente:

- Descripción del problema: se muestra el problema de preferencias que se desea solucionar. Las restricciones que tiene. En particular se pone una tabla que contiene los datos del problema para proseguir con el modelado. Y el resultado que se espera que dé `clasp` en base a la descripción proporcionada del problema.

- Análisis del problema: en esta parte del problema se analiza que operador de optimización se debe usar, cuántos y para que se usan. Se debe considerar que declaración tendrá más prioridad, es decir cuál tiene mayor importancia.

- Diseño e implementación del problema: en esta parte de modela el problema en ASP, tomando en cuenta los datos de la tabla que se presenta en la descripción del problema. Se describe que significa cada línea. Primero se escribe cuantas opciones existen, después se describen los datos de la tabla. Finalmente se presentan las declaraciones de optimización necesarias para solucionar el problema y llegar al

resultado deseado. En esta última parte del modelado se relacionan los datos y se asignan prioridades dentro de las declaraciones de optimización.

- Resultados: Finalmente se analizan los resultados obtenidos, analizando cada answer set arrojado por `clasp`, así como los valores de optimización de cada answer set. Así podemos decir que significa cada answer set con relación al problema de preferencias.

Ejemplo 1.

Descripción del problema

Se desea ir dela ciudad A a la ciudad B, y se prefiere ir en el menor tiempo posible. Existen tres caminos para llegar, los cuales son de 5km, 2 km y 4km, y se recorren en un tiempo de 3hrs, 1 hr y 2 hrs respectivamente, como se muestra en la siguiente tabla.

Camino	Distancia	Tiempo
1	5 km	3 hrs
2	2 km	1 hr
3	4 km	2 hrs

Se espera que el viajero prefiera irse por el camino 2, ya que es el más corto, y llega en poco tiempo.

Análisis del problema

Se presentaran los tres caminos que se pueden elegir, describiendo los tiempos y distancias de cada uno, para así darle preferencia al que realice el recorrido en el menor tiempo y recorra la menor distancia. Para esto, utilizaremos el operador ***#minimize***, porque queremos reducir (minimizar) tanto el tiempo como la distancia, le daremos mayor prioridad a minimizar el tiempo, porque nos interesa llegar rápido de A-B.

Diseño e implementación del problema

Modelaremos el problema con ASP, para ello representaremos todos los datos dados anteriormente.

Para empezar, decimos que se tienen tres caminos y lo representamos como sigue:

```
1 { camino(1..3) } 1.
```

Donde `camino(1..3)` indica que se tienen 3 caminos, también podría escribirse camino(1), camino(2) y camino(3) separados por un "." , para reducir un poco el código lo dejamos como esta.

Ahora describiremos las distancias

```
distancia(1,5). %Distancia 1 es de 5 km
distancia(2,2). %Distancia 2 es de 2 km
distancia(3,4). %Distancia 3 es de 4 km
```

Ahora continuamos con los tiempos que dura el recorrido en cada camino:

```
tiempo(1,3).
tiempo(2,1).
tiempo(3,2).
```

El recorrido del camino 1 dura 3hrs, el camino 2 dura 1 hr y el camino 3 dura 2 hrs.

Finalmente hacemos las declaraciones de minimización de tiempo y distancia, y así definir que se prefiere reducir el tiempo antes que la distancia.

En esta parte se asocian el camino con la distancia y con el tiempo, en los parámetros del operador `#minimize.`

Decimos entonces que queremos elegir un `camino(X)` considerando la `distancia(X,Y)`, el camino X y la distancia Y del camino X, aquí se relaciona la distancia a cada camino, y le asignamos una prioridad "`@ 1`", menor que a la declaración que minimiza el tiempo.

```
#minimize [camino(X): distancia(X,Y) = Y @ 1].
```

Ahora escribimos la declaración que minimiza el tiempo, y decimos que elegimos un `camino(X)` considerando el `tiempo(X,Y)`, es decir el tiempo de recorrido Y del camino X, y le daremos una prioridad de 2, que es mayor que la anterior, porque es más importante minimizar el tiempo que la distancia.

```
#minimize [camino(X): tiempo(X,Y) = Y @ 2].
```

El código completo queda de la siguiente manera y el programa final se puede ver en el Apéndice A:

```
1    1 { camino(1..3) } 1.
2    distancia(1,5).
3    distancia(2,2).
4    distancia(3,4).
5    tiempo(1,3).
6    tiempo(2,1).
7    tiempo(3,2).
8    #minimize [camino(X): distancia(X,Y) = Y @ 1].
9    #minimize [camino(X): tiempo(X,Y) = Y @ 2].
```

Donde la Línea 8 y 9 indican que se desea minimizar la distancia a recorrer y el tiempo que durará el viaje. Se usa dos veces el operador #minimize para indicar que se desea no sólo minimizar el tiempo sino también la distancia, también se podría tomar sólo la Línea 8 y así no tomaríamos en cuenta el tiempo del recorrido, por lo tanto los answer set darían los caminos con la distancia más corta.

El valor que sigue a @ indica la prioridad que `clasp` dará a la sentencia, es decir se quiere minimizar la distancia y el tiempo, pero es más importante llegar en el menor tiempo posible que reducir la distancia del recorrido, por eso se le asigna un valor mayor a la sentencia de la Línea 9.

Si cambiamos 1 por 3, o por un valor entero mayor a 2 en la Línea 8 indicaríamos preferimos reducir la distancia al tiempo, sin embargo clasp tomará las dos condiciones para calcular el mejor answer set.

Resultados

`Clasp` ha calculado dos answer set que minimizan el tiempo y distancia del recorrido.

El primer answer set indica que el camino que minimiza el tiempo y la distancia es el *camino 3*, cuyos valores de optimización son: 2 y 4 en la parte de `Optimization`, donde 2 es por la declaración que minimiza el tiempo indicado en la línea 9; esto significa que el recorrido se haría en 2 hrs, y el 4 por la declaración de minimización de la distancia escrito en la línea 8, indicando que el viajero recorrería 4 km.

```
Answer: 1
camino(3)
Optimization: 2 4
```

El segundo answer set indica que el mejor camino que minimiza el tiempo y la distancia es el *camino 2*, cuyos valores de optimización son: 1 y 2 mostrados en la parte de `Optimization`, donde 1 es por la declaración que minimiza el tiempo indicado en la línea 9; esto significa que el recorrido se haría en 1 hr, y 2 por la declaración de minimización de la distancia escrito en la línea 8; indicando que el viajero recorrería 2 km.

```
Answer: 2
camino(2)
```

```
Optimization: 1 2
```

Los resultados en la parte de `Optimization` serán dados de forma descendente dependiendo de la prioridad dada a las restricciones de minimización, por ejemplo la Línea 9 tiene mayor prioridad que la Línea 8, entonces se muestra primero el valor de optimización de la Línea 9 que es 1, y luego el de la Línea 9 que es 2, y lo presenta así `Optimization: 1 2`.

Ejemplo 2.

Descripción del problema

Este ejemplo es una variante del Ejemplo 1, debido a que sólo cambiamos los valores de distancias, tiempos y agregamos una restricción, ahora consideramos que cada camino tiene un nivel de peligro, y está asignado de menor a mayor peligro, es decir el camino menos peligroso tendrá un valor menor. Preferimos un camino con el menor peligro posible.

La siguiente tabla muestra los nuevos datos de los caminos:

Camino	Distancia	Tiempo	Nivel de Peligro
1	30	1	1
2	20	2	3
3	10	5	2

Se espera que se prefiera viajar por el camino 1, ya que recorre 30 km en 1 hr,y no es un camino peligroso, y como se prefiere minimizar el tiempo que la distancia.

Análisis del problema

Dado que este ejemplo es una variante del ejemplo 1, sólo vamos a considerar los nuevos valores y a tomar en cuenta el peligro de cada camino, por esta razón queremos minimizar el nivel de peligro para evitar posibles accidentes.

Diseño e implementación del problema

Declaramos los tres caminos

```
1{camino(1..3)}1.
```

Las distancias de los caminos

```
distancia(1,30).  distancia(2,20). distancia(3,10).
```

Ahora los tiempos que se tarda en ir de A-B

```
tiempo(1,1). tiempo(2,2). tiempo(3,5).
```

Indicamos los niveles de peligro

```
peligro(1,1). peligro(2,3).peligro(3,2).
```

Finalmente definimos las declaraciones de optimización.

Asociamos el camino X con su nivel de peligro Y. Y le damos mayor prioridad a minimizar la preferencia de un camino peligroso, es decir elegir un camino con un nivel de peligro bajo para evitar accidentes, por eso ponemos `@ 5.`

```
#minimize[camino(X): peligro(X,Y)= Y @ 5].
```

Ahora minimizamos el tiempo del recorrido y le damos una prioridad menor que a reducir el peligro, pero mayor que a la declaración que reduce la distancia. Asociamos el camino X con su respectivo tiempo Y.

```
#minimize[camino(X): tiempo(X,Y) = Y @ 3].
```

Escribimos la declaración que minimiza la distancia a recorrer, y le asignamos una prioridad `@ 2`. Asociamos el camino X con su distancia Y.

```
#minimize[camino(X): distancia(X,Y) = Y @ 2].
```

El código completo queda de la siguiente manera y el programa final se puede ver en el Apéndice A:

```
1    1{camino(1..3)}1.
2    distancia(1,30).  distancia(2,20). distancia(3,10).
3    tiempo(1,1). tiempo(2,2). tiempo(3,5).
4    peligro(1,1). peligro(2,3).peligro(3,2).
5    #minimize[camino(X): peligro(X,Y)= Y @ 5].
6    #minimize[camino(X): tiempo(X,Y) = Y @ 3].
7    #minimize[camino(X): distancia(X,Y) = Y @ 2].
```

Es importante minimizar tres aspectos el tiempo, la distancia y el peligro, y esto se hace mediante las líneas 5, 6 y 7 respectivamente.

Las prioridades están dadas de acuerdo a que se quiere llegar más rápido no importando la distancia a recorrer. Por esto la Línea 6 tiene mayor prioridad que la Línea 7. Y la línea 5 tiene mayor prioridad porque queremos reducir el nivel de peligro del camino a elegir.

Resultados

`Clasp` ha calculado dos answer sets.

Este primer answer set resultante muestra el camino 3, en la parte de `Optimization` se obtienen los siguiente valores `2 5 10`, donde 2 indica que el camino 3 tiene un nivel de peligro 2, el 5 indica que el recorrido por este camino se hace en 5 horas, y el 10 indica que se recorren 10 km por este camino.

```
Answer: 1
camino(3)
Optimization: 2 5 10
```

El segundo answer set muestra que se prefiere tomar el camino 1. Cuyos valores en la parte de `Optimization` son: `1 1 30`, donde el primer 1 indica que el nivel de peligro de camino 1 es de 1, el nivel más bajo escrito en la tabla. El segundo 1 indica el tiempo que dura el recorrido por este camino es de 1 hora, y finalmente el 30 indica que se recorren 30 km en este camino.

```
Answer: 2
camino(1)
Optimization: 1 1 30
```

Ejemplo 3.

Descripción del problema

Se desea elegir una universidad entre las 5 mejores de un ranking, teniendo en cuenta los costos. Se prefiere una universidad ubicada en los primeros lugares del ranking. La siguiente tabla muestra el ranking de las universidades con sus costos respectivos.

Código universidad	Universidad	Ubicación en el ranking	Costo
1	A	1	1500
2	B	2	1000
2	C	3	2000
3	D	4	1800
4	E	5	9000

Se espera que se prefiera estudiar en la universidad con código 1, puesto que esta ubicada en el primer lugar y tiene un bajo costo.

Análisis del problema

Se modelará un programa que pueda elegir la mejor universidad y a un costo accesible.Para esto se tomará en código de la universidad. Para esto le asignaremos una prioridad mayor la ubicación de la universidad en el ranking, como el número de lugar en el ranking es de forma ascendente minimizaremos el lugar.

Diseño e implementación del problema

En este ejemplo usaremos el código de la universidad para asociarla con su lugar en el ranking y su precio.

Definimos las cinco universidades:

```
1{ universidad(1..5) } 1.
```

Escribimos el lugar en el ranking

```
ranking(1,1).
ranking(2,2).
ranking(3,3).
ranking(4,4).
ranking(5,5).
```

Ahora los precios de las universidades

```
precio(1,1500).
precio(2,1000).
precio(3,2000).
precio(4,1800).
precio(5,9000).
```

Finalmente definimos las declaraciones de optimización.

Para minimizar el lugar en el ranking decimos que preferimos una `universidad(X)` que este en lugar del `ranking(X,Y)`, es decir cada universidad X con su lugar Y en el ranking y le daremos una prioridad mayor a esta declaración porque nos importa más elegir una universidad colocada en los primeros lugares del ranking.

```
#minimize [universidad(X):ranking(X,Y) = Y @ 3].
```

Ahora queremos reducir el costo de la universidad, para esto usamos `#minimize` y en sus parámetros ponemos que tome en cuenta la universidad X y con su precio Y, y le damos una prioridad de 2, menor que a la declaración anterior.

```
#minimize [universidad(X): precio(X,Y) = Y @ 2].
```

El código completo queda como sigue :

```
1 { universidad(1..5) } 1.

2    ranking(1,1).
3    ranking(2,2).
4    ranking(3,3).
5    ranking(4,4).
6    ranking(5,5).
7    precio(1,1500).
8    precio(2,1000).
9    precio(3,2000).
10   precio(4,1800).
11   precio(5,9000).
12   #minimize [universidad(X):ranking(X,Y) = Y @ 3].
13   #minimize [universidad(X): precio(X,Y) = Y @ 2].
```

El programa final se puede ver en el Apéndice A.

En este ejemplo se quiere elegir la mejor universidad, con un costo accesible. Por lo tanto como en el ranking el menor valor indica que una universidad es mejor que otra, queremos minimizar la posición en el ranking, obteniendo así la mejor universidad que se puede elegir. Esto se representa en la Línea 12, y así le damos mayor prioridad al ranking que al precio de la universidad indicado en la Línea 13.

Resultados

`Clasp` da 5 answer sets.

El primero indica que `clasp` eligió la universidad 5, y con sus valores de optimización mostrados en la parte de `Optimization: 5 9000` dice que la universidad 5 ocupa el 5° lugar en el ranking y tiene un costo de $5900.

```
Answer: 1
universidad(5)
Optimization: 5 9000
```

El *answer 2* indica que `clasp` eligió la universidad 4, y con sus valores en la parte de `Optimization: 4 1800` dice que la universidad 4 ocupa el 4° lugar en el ranking y tiene un costo de $1800.

```
Answer: 2
universidad(4)
Optimization: 4 1800
```

El *answer 3* indica que `clasp` eligió la universidad 3, y con sus valores en la parte de `Optimization: 3 2000` dice que la universidad 3 ocupa el 3° lugar en el ranking y tiene un costo de $2000.

```
Answer: 3
universidad(3)
Optimization: 3 2000
```

El *answer 4* indica que `clasp` eligió la universidad 2, y con sus valores en la parte de `Optimization: 2 1000` dice que la universidad 2 ocupa el 2° lugar en el ranking y tiene un costo de $1000.

```
Answer: 4
universidad(2)
Optimization: 2 1000
```

El *answer 5* es el mejor answer set e indica que la mejor universidad es la 1, porque está ubicada en el primer lugar en ranking, con un costo de $1500, esto está indicado en la parte de `Optimization: 1 1500.` Así se cumple que se minimice el lugar en el ranking y el costo de la universidad.

```
Answer: 5
universidad(1)
Optimization: 1 1500
```

Ejemplo 4.

Descripción del problema

Se desea invertir $30,000 y obtener el mayor número de ganancias. Para esto se tienen tres opciones para invertir (1) donde se obtiene una ganancia de 6%, en (2) se obtiene una ganancia del 10% y (3) que obtiene una ganancia del 3%, como se muestra en la siguiente tabla.

Opción	Ganancia
1	6 %
2	10 %
3	3 %

Se espera que el inversionista elija la opción 2, ya que es la opción que le da más ganancias.

Análisis del problema

Como estamos buscando un answer set que prefiera la inversión que obtenga más ganancias usaremos **#maximize**, porque queremos maximizar las ganancias. Así representaremos las tres opciones con sus respectivos porcentajes de ganancia, sólo usaremos el porcentaje y no la cantidad real para simplificar la elección.

Diseño e implementación del problema

Modelaremos el problema con ASP , y así representaremos los datos dados en la tabla anterior.

```
1{opcion(1..3)}1.
```

Donde opción(1..3) indica que se tienen 3 opciones.

Describimos las ganancias de cada opción

```
ganancia(1,6).
ganancia(2,10).
ganancia(3,3).
```

Finalmente hacemos la declaración que indica que deseamos que `clasp` prefiera la opción que deje más ganancias al inversionista. Así decimos que queremos maximizar eligiendo una opción(X) considerando su respectiva ganancia(Y), así relacionamos cada opción con su ganancia. Y decimos que queremos que imprima el valor de la ganancia.

```
#maximize[opción(X):ganancia(X,Y)=Y].
```

Como sólo existe una declaración de optimización no usaremos la opción de prioridad.

El código completo queda de la siguiente manera:

```
1    1{opcion(1..3)}1.
2    ganancia(1,6).
3    ganancia(2,10).
4    ganancia(3,3).
```

```
5     #maximize[opcion(X):ganancia(X,Y)=Y].
```

Resultados

`Clasp` ha calculado dos answer sets.

El primer answer set indica que la mejor opción que maximiza las ganancias es la opción 3, cuyo valor de optimización es 16 (Optimization: 16), que es la suma de las dos opciones restantes 6 + 10, 6 % de la opción 1 y 10% de la opción 2, que da como resultado 16.

```
Answer: 1
opcion(3)
Optimization: 16
```

El segundo answer set indica que el mejor camino que maximiza las ganancias es la opción 2, cuyo valor de optimización es 9, que es la suma de la opción 1 y 3, 6+ 3 que dan como resultado 9.

```
Answer: 2
opcion(2)
Optimization: 9
```

Como podemos apreciar el operador **#maximize** a diferencia de **#minimize** cambia a la hora de dar el valor de optimización, éste realiza una suma de los datos que no pertenecen al answer set. En este ejemplo suma las ganancias.

Ejemplo 5

Descripción del problema

Un profesor tiene que elegir a uno de 3 alumnos que salieron empatados en un examen de selección, y tiene que preferir a uno de ellos para enviarlo a una olimpiada de matemáticas,

va a tomar en cuenta el promedio general y la asistencia, esta última se tomará en porcentaje. La información de cada alumno se muestra a continuación:

Alumno	Promedio	Asistencia
1	9	60%
2	9	90%
3	8	100%

Se espera que prefiera al alumno 2, porque tiene un promedio alto, y ha asistido a la mayoría de las clases.

Análisis del problema

Como se desea que el profesor prefiera enviar a la olimpiada al alumno que tenga el mejor promedio y además haya asistido más, usaremos el operador **#maximize**, y le daremos mayor prioridad al promedio que a la asistencia, pues que haya asistido no garantiza que tendrá un buen desempeño en la olimpiada.

Diseño e implementación del problema

Modelaremos el problema con ASP, para ellos representaremos los datos de la tabla anterior.

Primero decimos que existen tres alumnos de los que vamos a elegir uno, y lo presentamos así:

```
1{alumnos(1..3)}1.
```

Donde alumnos(1..3) indica que se tienen tres alumnos.

Describimos los promedios:

```
promedio(1,9).
```

```
promedio(2,9).
promedio(3,8).
```

Continuamos con la asistencia:

```
asistencia(1,60).
asistencia(2,90).
asistencia(3,100).
```

Finalmente escribimos las declaraciones de maximización de promedio y asistencia, y definimos mayor prioridad al promedio.

Decimos que queremos elegir un alumno considerando el promedio y dándole una prioridad 2, que es mayor que a la que se le dará a la siguiente declaración. Así asociamos al alumno con su promedio.

```
#maximize[alumnos(X):promedio(X,Y)=Y@2].
```

Ahora escribimos la declaración para maximizar la asistencia, y decimos que elija al alumno(X), y asocie al alumno X con su asistencia Y. A esta declaración le daremos una prioridad menor.

```
#maximize[alumnos(X):asistencia(X,Z)=Z@1].
```

El código completo queda de la siguiente manera:

```
1    1{alumnos(1..3)}1.
2    promedio(1,9).
3    promedio(2,9).
4    promedio(3,8).
5    asistencia(1,60).
```

```
6    asistencia(2,90).
7    asistencia(3,100).
8    #maximize[alumnos(X):promedio(X,Y)=Y@2].
9    #maximize[alumnos(X):asistencia(X,Z)=Z@1].
```

Resultados

`Clasp` calculó dos answer set, el óptimo es alumno 2.

El primer answer set indica que el profesor debe preferir enviar al alumno 3 a la olimpiada de matemáticas, y lo eligió porque tiene el cumple con las dos declaraciones de optimización, aunque después es descartado porque existe otro alumno que tiene un promedio alto y como tiene más prioridad, entonces existe un answer set mejor. En este primer answer set observamos que los valores de optimización `Optimization: 18 150` coinciden con la suma de las asistencias y promedios restantes, es decir 18 es la suma del promedio del alumno 1 y 2, cuyos valores son 9 y 9, y la suma da como resultado 18. Y 150 es la suma de la asistencia: 60% del alumnos 1 y 90% del alumno 2 que dan como resultado 150.

```
Answer: 1
alumnos(3)
Optimization: 18 150
```

El segundo answer set indica que el mejor alumno es el alumno 2, porque tiene un buen promedio y ha asistido a casi todas las clases. Los valores de `Optimization: 17 160` corresponden primero 17 a la suma de los promedios del alumno 1 que es 9 y del alumno 3 que es 8. Y 160 corresponde a la suma de las asistencias de los alumnos 1 y 3, que son 60 y 100 respectivamente.

```
Answer: 2
alumnos(2)
Optimization: 17 160
```

A diferencia de **#minimize, #maximize** realiza una suma para dar los valores de optimización, y esta suma es de los datos que no tienen relación con el answer set calculado.

Ejemplo 6

Descripción del problema

Se desea elegir entre 3 plantas que producen motores, tomando en cuenta la capacidad y costo de producción prefiriendo primero la que produzca la mayor cantidad de motores al menor costo posible, a continuación se muestra una tabla que muestra la capacidad mensual de producción y costo de producción de cada panta:

Planta	Capacidad mensual de Producción	Costo total de la producción
1	800	13600
2	600	12000
3	700	16800

Se espera que se prefiera al planta 1 porque produce más motores.

Análisis del problema

Como deseamos preferir la planta que produzca la mayor cantidad de motores usaremos el operador **#maximize** para representarlo, y para representar que se prefiera el menor costo usaremos el operador **#minimize**, que no ayudará a elegir el menor precio.Le daremos mayor prioridad a producir más motores.

Diseño e implementación del problema

Modelaremos el problema con ASP, y comenzaremos representando las plantas, así decimos que tenemos tres plantas que producen motores:

```
1{planta(1..3)}1.
```

También podríamos representar las plantas de la siguiente manera: camino(1,), camino(2) y camino(3).

Ahora describimos la capacidad de producción mensual:

```
capacidad(1,800).
capacidad(2,600).
capacidad(3,700).
```

Continuamos con los costos de producción de cada planta:

```
costo(1,13600).
costo(2,12000).
costo(3,16800).
```

Finalmente hacemos las declaraciones de minimización del costo y maximización de la producción de motores, y le daremos mayor prioridad a producir más motores que a reducir el costo de producción.

Decimos que queremos elegir una planta(X) que produzca Y motores al mes(capacidad de producción), con la siguiente declaración relacionamos cada planta con su producción mensual, y como queremos maximizar usaremos el operador #maximizar, y le daremos prioridad 2 (@2):

```
#maximize[planta(X):capacidad(X,Y)=Y @ 2].
```

Ahora representamos con el operador #minimize que deseamos preferir la planta que tenga el menor costo de producción, y le damos una prioridad menor que en la declaración anterior, en este caso prioridad 1. Representamos la declaración de minimización diciendo que elegiremos una planta X con su respectivo costo de producción Y que tenga el menor costo:

```
#minimize[planta(X):costo(X,Y)=Y@1].
```

El código completo queda de la siguiente manera:

```
1    1{planta(1..3)}1.
2    capacidad(1,800).
3    capacidad(2,600).
4    capacidad(3,700).
5    costo(1,13600).
6    costo(2,12000).
7    costo(3,16800).
8    #minimize[planta(X):costo(X,Y) = Y @ 1].
9    #maximize[planta(X):capacidad(X,Y) = Y @ 2].
```

Resultados

`Clasp` ha calculado dos answer sets.

El primero indica que se debe preferir la planta 3, y sus valores de `Optimization: 1400 16800`; el primero es la suma de las capacidades de producción de la planta 1 y 2, y 16800 es el costo que tiene la planta 3.

```
Answer: 1
planta(3)
Optimization: 1400 16800
```

El segundo answer set indica que la major planta es la 1, porque esta es la que produce más motores y además el costo de producción es bajo. Sus valores de `Optimization: 1300 13600,` significan que el primer valor es la suma de las capacidades de producción de la planta 2 y 3, y el segundo valor indica el costo de producción de la planta 3.

```
Answer: 2
planta(1)
Optimization: 1300 13600
```

Como podemos observar aunque se usen en el mismo problema los dos operadores (#maximize y #minimize) los resultados de optimización no cambian, es decir, #maximize sigue realizando las suma correspondiente y #miminize busca el valor minimo.

Ejemplo 7

Descripción del problema

Un comerciante desea elegir un día de descanso para esto ha realizado un análisis de sus ventas, y ha conseguido la siguiente tabla de ventas promedio. El comerciante prefiere descansar un día en que no haya muchas ventas para no tener pérdidas.

Día de la semana	Ventas promedio por día
1 – Domingo	300
2- Lunes	500
3 - Martes	100
4 - Miércoles	300
5- Jueves	600
6 - Viernes	700
7 – Sábado	200

Se espera que el comerciante prefiera el día Martes como día de descanso, ya que de acuerdo a las ventas promedio es el día en que no hay muchas ventas.

Análisis del problema

Usaremos el operador **#minimize** ya que estamos buscando un answer set que prefiera como día de descanso en el que hayan menos ventas. Lo que haremos es modelaremos un problema que minimize las ventas, es decir que busqué entre las opciones el día con menos ventas.

Diseño e implementación del problema

Modelaremos el problema en ASP, así representaremos los datos de la tabla anterior.

Decimos que tenemos 7 posibles opciones, en este caso los días de la semana que son enumerados del 1-7.

```
1{descanso(1..7)}1.
```

Describimos las ventas de cada día

```
venta(1,300). venta(2,500).
venta(3,100). venta(4,300).
venta(5,600). venta(6,700).
venta(7,200).
```

Finalmente hacemos la declaración de optimización que indica que seamos que `clasp` prefiera el día con menos ventas. Así decimos que queremos minimizar eligiendo un día descanso(X) considerando su respectivo promedio de ventas(Y), así relacionamos cada opción con su venta.

```
#minimize[descanso(X):venta(X,Y)=Y].
```

El código complete queda de la siguiente manera:

```
1    1{descanso(1..7)}1.
2    venta(1,300). venta(2,500).
3    venta(3,100). venta(4,300).
4    venta(5,600).venta(6,700).
5    venta(7,200).
6    #minimize[descanso(X):venta(X,Y)=Y].
```

Resultados

`Clasp` ha calculado dos answer set que minimizan las ventas y así se elige el día con menos ventas.

El primer answer set indica que el día con menos ventas, es decir el día que se debería destinar para descanso es el día 7, que corresponde al día sábado, y el valor de optimización representada en la parte de `Optimization:200`, indica el valor de las ventas en el día 7.

```
Answer: 1
descanso(7)
Optimization: 200
```

El segundo answer set indica que el día con menos ventas y que el comerciante debe preferir para descansar es el día 3, que corresponde al día martes, en la parte de `Optimization` podemos ver que muestra `100` que pertenece al promedio de ventas que el comerciante tiene los martes.

```
Answer: 2
descanso(3)
Optimization: 100
```

Capítulo 3

Discusión de resultados

Durante el desarrollo de este trabajo y después de realizar varias pruebas con los problemas planteados logramos identificar como funciona `clasp`, las diferencias entre #minimize y #maximize. La comprensión del operador #minimize es más fácil ya que como su nombre lo dice busca el menor entre un conjunto de datos dados y el menor es el mejor, y también el valor de optimización. Por el contrario en el estudio del funcionamiento de `clasp` nos encontramos que gringo no soporta las declaraciones de maximización [26], gringo es un modelador capaz de traducir los programas proporcionados por los usuarios a su equivalente a programas básicos. La salida de gringo puede ser enviada al solucionador clasp, que calcula el conjunto de respuesta [25]. Tras encontrarnos con esta respuesta se procedió a realizar pruebas y analizas los resultados encontrándonos qué #maximize busca el mayor, pero para encontrar el valor de optimización realiza una suma de los pesos de los literales excepto el actual.

Si hay varios operadores (#minimize o #maximize) iguales no hay necesidad de poner prioridad. En los ejercicios planteados se dejó así para entender lo que queríamos resolver. Pero podría o no ponerse prioridad, podría en cambio se pude usar "," (conjunción) para unir las declaraciones de optimización que usan el mismo.

La prioridad sólo se usa cuando son diferentes operadores de optimización, es decir cuando tengamos que minimizar (#minimize) y maximizar (#maximize), en este caso tenemos que decir que es más importante minimizar o maximizar.

Al iniciar este trabajo suponíamos que el operador #maximize trabajaba de manera contraria a #minimize, y en cierta parte así lo hace, #minimize busca el menor y #maximize el mayor, varía al mostrar y obtener los valores de optimización.

Minimize busca el menor y el valor que tenga ese answer set será el valor de optimización. Maximize por el contrario realiza una suma de todas las opciones excepto la actual y esa suma es el valor de optimización.

El número de answer sets no está bien definido, los answer sets arrojados por `clasp` son definidos por el algoritmo de Ramificación y Poda (Branch an Bound)[26] que se aplica normalmente para resolver problemas de optimización. Realiza una enumeración de solucionas basándose en la generación de un árbol de expansión. Lo que hace es seleccionar el nodo que en principio parece más prometedor. La técnica de Ramificación y Poda utiliza cotas para podar aquellas ramas del árbol que no conducen a la solución óptima. Para ello calcula en cada nodo una cota del posible valor de aquellas soluciones alcanzables desde ése. Si la cota muestra que cualquiera de las soluciones ten que ser necesariamente peor que la mejor solución hallada hasta el momento no necesitamos seguir explorando por esa rama del árbol, lo que permite realizar el proceso de poda[27].

Como podemos observar en el Apéndice B sección **B.4** `clasp` realmente resuelve el #maximize usando #minimize, y lo que hace es negar lo que está dentro del #maximize, Esto no es mostrado, lo hace internamente.

Capítulo 4

Conclusiones y Trabajo Futuro

En este documento se trabajó con ASP un lenguaje declarativo para la representación de conocimiento y razonamiento del sentido común. Los problemas que se modelaron fueron de preferencias donde se tenía que elegir entre diferentes opciones obteniendo la mejor dependiendo del contexto del problema, primero que se quería realizar minimizar o maximizar, o ambas. Después cuál tenía mayor prioridad. Todos los problemas modelados fueron sencillos para poder entender cómo se modelan en ASP, también para reducir la complejidad del análisis del funcionamiento de `clasp`,el solucionador usado en este trabajo.

La codificación fue bastante sencilla, basándonos en los ejercicios de optimización propuestos en la guía de Potassco [25]. Para ello primero se analizó a profundidad el ejercicio de los hoteles mostrado en el capítulo 2, hasta lograr comprender como se realizan las declaraciones de optimización y de los datos. Con estas bases se logró cumplir con el objetivo principal de este trabajo: modelar un conjunto de problemas de preferencias utilizando operadores de optimización en Answer Set Programming. También se cumplió con el estudio y análisis de los operadores de optimización. Ahora se comprende cómo se usan y funcionan dichos operadores para la implementación de Answer Set Programming.

Implementamos siete problemas de preferencia que cumplen con los criterios que mencionamos en la introducción de este documento, son fáciles de codificar, fáciles de modificar para obtener nuevos y mejores resultados, son entendibles y con facilidad se pueden trasladar de un código a otro.

En un trabajo futuro podríamos adentrarnos más en el funcionamiento del algoritmo de Ramificación y Poda en `clasp`, es decir, explicar cómo se realiza el árbol con las opciones modeladas, para luego realizar la poda y obtener el answer set óptimo.

También se podría llevar el modelado de problemas de preferencias a otro nivel, modelar problemas más complejos, con grandes cantidades de información, por ejemplo para ser usados en la toma de decisiones.

Apéndice A

Programas completos

A.1 Ejemplo1

```
1 { camino(1..3) } 1.

distancia(1,5).

distancia(2,2).

distancia(3,4).

tiempo(1,3).

tiempo(2,1).

tiempo(3,2).

#hide distancia(X,Y).

#hide tiempo(X,Y).

#minimize [camino(X): distancia(X,Y) = Y @ 1].

#minimize [camino(X): tiempo(X,Y) = Y @ 2].
```

A.2 Ejemplo 2

```
1{camino(1..3)}1.

distancia(1,30).  distancia(2,20). distancia(3,10).

tiempo(1,1). tiempo(2,2). tiempo(3,5).

%niveles de peligrosidad

peligro(1,1). peligro(2,3).peligro(3,2).

#hide distancia(X,Y).
```

```
#hide tiempo(X,Y).
#hide peligro(X,Y).
#minimize[camino(X): peligro(X,Y) = Y @ 5].
#minimize[camino(X): tiempo(X,Y) = Y @ 3].
#minimize[camino(X): distancia(X,Y)= Y @ 2].
```

A.3 Ejemplo 3

```
1{ universidad(1..5) } 1.
ranking(1,1).
ranking(2,2).
ranking(3,3).
ranking(4,4).
ranking(5,5).
precio(1,1500).
precio(2,1000).
precio(3,2000).
precio(4,1800).
precio(5,9000).
#hide ranking(X,Y).
#hide precio(X,Y).
#minimize [universidad(X):ranking(X,Y) = Y @ 3].
#minimize [universidad(X): precio(X,Y) = Y @ 2].
```

A.4 Ejemplo 4

```
1{opcion(1..3)}1.

ganancia(1,6).

ganancia(2,10).

ganancia(3,3).

#hide ganancia(X,Y).

#maximize[opcion(X):ganancia(X,Y)=Y].
```

A.5 Ejemplo 5

```
1{alumnos(1..3)}1.

promedio(1,9).

promedio(2,9).

promedio(3,8).

asistencia(1,60).

asistencia(2,90).

asistencia(3,100).

#hide promedio(X,Y).

#hide asistencia(X,Z).

#maximize[alumnos(X):promedio(X,Y)=Y@2].

#maximize[alumnos(X):asistencia(X,Z)=Z@1].
```

A.6 Ejemplo 6

```
1{planta(1..3)}1.

capacidad(1,800).

capacidad(2,600).

capacidad(3,700).

costo(1,13600).

costo(2,12000).

costo(3,16800).

#hide capacidad(X,Y).

#hide costo(X,Y).

#minimize[planta(X):costo(X,Y) = Y @ 1].

#maximize[planta(X):capacidad(X,Y) = Y @ 2].
```

A.7 Ejemplo 7

```
1{descanso(1..7)}1.

venta(1,300). venta(2,500).

venta(3,100). venta(4,300).

venta(5,600).venta(6,700).

venta(7,200).

#hide venta(X,Y).

#minimize[descanso(X):venta(X,Y)=Y].
```

Apéndice B

Conversaciones en Foro de Potassco

B.1

[potassco:discussion] Help with optimisation

"Elizabeth Sedeño" <elizhx9@users.sf.net> 12 de noviembre de 2012 21:52

Responder a: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Para: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Well, I've been working the # minimize, intuitively I think the # maximize is the opposite of # minimize, but try the following example:
1 { university(1..5) } 1.

ranking(1,1).
ranking(2,2).
ranking(3,3).
ranking(4,4).
ranking(5,5).

maximize[university(X):ranking(X,Y)=Y](**.**)

I see in the Optimization value does not match what I suspected, I expected as a result 5 because it is the maximum number in the ranking, but clasp throws 10,
I wonder why?

B.2.

[potassco:discussion] Help with optimisation

---

rkaminski <rkaminski@users.sf.net> 13 de noviembre de 2012 05:43

Responder a: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Para: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

This has historical reasons. The output format of gringo does not support maximize statements. So they are translated into minimize statements (all signs are flipped). This translation preserves answer sets but the optimal value reported by clasp no longer corresponds to the sum of weights of true literals in the maximize statements.

B.3.

[potassco:discussion] Help with optimisation

"Elizabeth Sedeño" <elizhx9@users.sf.net> 28 de noviembre de 2012 20:55

Responder a: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Para: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Well, Now I have a question, if that is not the sum,what is it?

B.4

[potassco:discussion] Help with optimisation

---

rkaminski <rkaminski@users.sf.net> 29 de noviembre de 2012 02:04

Responder a: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Para: "[potassco:discussion]" <863629@discussion.potassco.p.re.sf.net>

Look as the following example:

```
2 { a,b,c } 2.
:- b,c.
#maximize [ a=1, b=2, c=3 ].
```

The optimal answer is {a,c} with value 4. clasp really solves:

```
2 { a,b,c } 2.
:- b,c.
#minimize [ not a=1, not b=2, not c=3 ].
```

And thus reports {a,c} with value 2=(1+2+3)-(1+3). That is the sum of all weights in the maximize constraints minus the actual optimum 4.

Regards, Roland

Bibliografía

[1] Diego Rodríguez Vllanueva. Utilización de los enfoques para preferencias en answer set programming para definir familias de problemas con preferencias. Tesis. http://jupiter.utm.mx/~tesis_dig/10300.pdf. Fecha de última consulta: 20/enero/2013.

[2] Claudia Zepeda and José Luis Carballido. P-stable models of Strong kernel programs.2010.

[3] M. Osorio, J. A. Navarro, J. Arrazola, and V. Borja. Logics with common weak completions.*Journal of Logic and Computation*, 16(6):867-890, 2006.

[4] M. Osorio and J. L. Carballido. Brief study of G'3 logic. *Journal of Applied Non-Classical Logic*, 18(4):79-103, 2009.

[5] M. Osorio, J. Arrazola, and J. L. Carballido. Logical weak completions of paraconsistent logics.*Journal of Logic and Computation*, Published on line on May 9, 2008.

[6] Esra Erdem, Lee Joohyung y Lierler Yuliya. MP3: Theory and Practice of Answer Set Programming. http://peace.eas.asu.edu/aaai12tutorial/Home.html. Fecha de última consulta: 28/septiembre/2012.

[7] Matilde Hernández Salas. Modelado y planeación con ASP (Answer Set Programming) http://catarina.udlap.mx/u_dl_a/tales/documentos/msp/hernandez_s_m/capitulo1.pdf. Fecha de última consulta: 28/septiembre/2012.

[8] Universidad de Potsdam. Potassco. http://potassco.sourceforge.net/labs.html. Fecha de última consulta: 28/septiembre/2012.

[9] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub.Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence* (IJCAI'07), pages 386–392. AAAI Press/The MIT Press, 2007.

[10] Kaci, S. *Working with Preferences: Less Is More*. Springer-Verlag. 2011.

[11] Domshlak, C., Hüllermeier, E., Kaci, S., and Prade, H. (2011). *Preferences in AI: An overview. Artifical Intelligence*, 175(7-8):1037–1052.

[12] G. Brewka. Adding priorities and specificity to default logic. In L. Pereira and D. Pearce, editors, *European Workshop on Logics in Artificial Intelligence (JELIA'94)*, Lecture Notes in Artificial Intelligence, pages 247–260. Springer Verlag, 1994

[13] J. Delgrande and T. Schaub. Expressing preferences in default logic. *Artificial Intelligence*, 123(1-2):41–87, 2000

[14] K. Konolige. Hierarchic autoepistemic theories for nonmonotonic reasoning. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, pages 439–443.Morgan Kaufmann Publishers, 1988.

[15] J. Rintanen. Prioritized autoepistemic logic. In C. MacNish, D. Pearce, and L. M. Pereira, editors, *Proceedings of the European Workshop on Logics in Artificial Intelligence*, volume 838 of *Lecture Notes in Artificial Intelligence*, pages 232–246, Berlin. Springer-Verlag. September 1994.

[16] [McCarthy, 1986] J. McCarthy. Applications of circumscription to formalizing common-sense knowledge. *Artificial Intelligence*, 28:89–116, 1986.

[17] V. Lifschitz. Closed-world databases and circumscription. *Artificial Intelligence*, 27:229–235, 1985.

[18] Y. Zhang and N. Foo. Answer sets for prioritized logic programs. In J. Maluszynski, editor, *Proceedings of the International Symposium on Logic Programming (ILPS'97)*, pages 69–84. MIT Press, 1997.

[19] [Brewka and Eiter, 1999] G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.

[20] T. Gordon. *The Pleading Game: An Artificial Intelligence Model of Procedural Justice.* Dissertation, Technische Hochschule Darmstadt, Alexanderstraße 10, D-64283 Darmstadt, Germany, 1993.

[21] H. Prakken. *Logical Tools for Modelling Legal Argument*. Kluwer Academic Publishers, 1997.

[22] B. Grosof. Business rules for electronic commerce. http://www.research.ibm.com/rules/papers.html , 1999. IBM Research

[23] B. Cui and T. Swift. Preference logic grammars: Fixed-point semantics and application to data standardization. *Artificial Intelligence*, 138(1-2):117–147, 2001.

[24] Universidad de Potsdam. Clasp. http://www.cs.uni-potsdam.de/clasp . Fecha de última consulta: 20/enero/2013.

[25] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub y Sven Thiele. A User's Guide to gringo, clasp, clingo, and iclingo. http://www.cs.utexas.edu/~vl/teaching/lbai/clingo_guide.pdf. Fecha de última consulta: 20/enero/2013.

[26] Potassco.(2012,Agosto 30). Help with optimisation[Mensaje de Foro]. Recuperado de: http://sourceforge.net/p/potassco/discussion/863629/thread/a7e58913/ .Fecha última de consulta: 20/enero/2013.

[27] Rosa Guerequeta y Antonio Vallecillo. Técnicas de Diseño de Algoritmos. Servicio de Publicaciones de la Universidad de Málaga. 1998.
http://www.lcc.uma.es/~av/Libro/indice.html . Fecha de la última consulta: 21/enero/2013.