BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

"MODELADO DEL PROBLEMA DE SECUENCIACIÓN MÍNIMA CON TIEMPO DE REALIZACIÓN UTILIZANDO UN ENFOQUE DE PLANIFICACIÓN BASADO EN ANSWER SET PROGRAMMING Y PREFERENCIAS"

TESIS PROFESIONAL PRESENTADA POR

ADRIANA HUITZIL TELLO

COMO REQUISITO PARA OBTENER EL TÍTULO DE
LICENCIADA EN CIENCIAS DE LA COMPUTACIÓN

BENEMÉRITA UNIVERSIDAD AUTÓNOMA DE PUEBLA

FACULTAD DE CIENCIAS DE LA COMPUTACIÓN

"MODELADO DEL PROBLEMA DE SECUENCIACIÓN MÍNIMA CON TIEMPO DE REALIZACIÓN UTILIZANDO UN ENFOQUE DE PLANIFICACIÓN BASADO EN ANSWER SET PROGRAMMING Y PREFERENCIAS"

Tesis profesional presentada por

Adriana Huitzil Tello

Como requisito para obtener el título de
Licenciada en Ciencias de la Computación

Jurado Calificador

Dr. Iván Olmos

Dr. José Luis Carballido

Dra. Claudia Zepeda Cortés

Índice general

Introducción

Actualmente en la vida diaria se realizan diversos trabajos que requieren el uso de precisión, de actividades que pueden ser peligrosas para el ser humano, de procesos repetitivos o tediosos, por tal razón las personas necesitan de máquinas, mecanismos y programas que realicen el trabajo de una forma rápida y eficiente, automatizar tareas para que se obtenga una mayor y mejor producción en el menor tiempo posible y al mismo tiempo se minimicen errores o baja producción, evitar posibles accidentes y trabajo forzado.

En los últimos años la gama de aplicaciones prácticas y casos exitosos para resolver problemas de la vida real ha crecido de manera impresionante, en los que destacan principalmente aplicaciones para resolver problemas combinatorios, proveer esquemas de planeación y el modelado de agentes lógicos por mencionar algunas de las aplicaciones típicas. En este sentido, se encuentra la Programación Lógica la cual tiene su origen en aplicaciones de Inteligencia Artificial o relacionadas por ejemplo: sistemas expertos, bases de datos, demostración automática de teoremas, reconocimiento del lenguaje natural, etc.

En 1969, la planeación surge como un problema en lógica cuando John McCarthy propuso el primer programa comprobador o verificador de teoremas adaptado a la planificación [20]. La planificación es una de las formas en que un agente puede tomar ventaja del conocimiento que posee, y la habilidad de razonar acerca de las acciones y sus efectos en el ambiente [4]. A través de la Programación Lógica es posible hacer uso de la Planificación, la cual se ha aplicado en una variedad de tareas que incluyen la robótica, la planificación de procesos, agentes autónomos y control de la misión en naves espaciales, entre otros, que permiten especificar una secuencia de pasos o acciones que deben cumplirse para poder obtener determinados resultados que den solución al problema planteado. Así mismo han surgido herramientas de cómputo que permiten obtener una o varias respuestas que solucionan el problema llamadas modelos estables o Answer Set. Estos Answer Set pueden obtenerse en base a ciertas características o reglas para indicar si se trata de un Answer Set o un Answer Set óptimo, mediante el uso de preferencias.

La contribución de este trabajo se enfoca en implementar un problema de Planificación en un Programa Lógico en el cual se controlan los tiempos en que se realizan las acciones en base al problema de Secuenciación Mínima con Tiempos de Realización y haciendo uso de preferencias, con lo que se debe medir el tiempo en que se realizan las entregas para que éstas sean recibidas en el tiempo que se ha indicado. Para obtener la respuesta a este problema se hará uso de resolvedores de programas lógicos. Este tipo de problemas en donde se hace uso de la Planificación permite automatizar sistemas en los que es necesario planear la forma en que se pueden obtener diversos resultados en base a una cierta información, como los ejemplos que se han mencionado anteriormente.

En el capítulo 1 se describen diversos conceptos que se consideran importantes tales como nociones generales de la programación lógica, la sintaxis de un programa lógico, la semántica que permite dar un significado a los programas lógicos y la descripción de las herramientas con las cuales se trabajarán para el desarrollo de este problema. En el capítulo 2 se describe el lenguaje A, el cual es parte fundamental de la Programación Lógica y se muestra la codificación del problema de reparto de oficios en este lenguaje. En el capítulo 3 se realiza la descripción detallada del problema particular de reparto de oficios, la codificación en Smodels para la primera versión y las modificaciones realizadas para llegar a una versión final. En el capítulo 4 se presenta la explicación de la sintaxis básica para codificar en Clasp y posteriormente la codificación del problema, indicando las diferencias con respecto a Smodels. En el capítulo 5 se define el uso de preferencias y la codificación en Clasp. En las conclusiones se muestra una comparación con el uso de estas dos herramientas: Smodels y Clasp. En el *Apéndice* se encuentran algunas de las pruebas realizadas de cada software. Finalmente se muestra la bibliografía utilizada en esta tesis.

Capítulo 1

Preliminares

Como ya se ha mencionado la Programación Lógica tiene las bases en la Inteligencia Artificial. Ofrece un modelo alternativo para enfrentar y tratar de solucionar problemas con la ayuda de la computadora, este modelo indica a la computadora la descripción del problema para que ésta sea la responsable de analizar el problema y determinar las posibles soluciones. En esta tesis se plantea el problema de reparto de oficios analizando la secuencia con la que se realizan ciertas tareas y con ello se propone un modelo general del mismo utilizando un enfoque de planificación basado en Answer Set Programming, donde los Answer Set del problema planteado se obtendrán con el uso de herramientas computacionales como Smodels y Clasp, los cuales se describen en este capítulo.

1.1 Programación Lógica

La Programación Lógica (Logic Programming o LP) ofrece una manera más natural de representar conocimiento declarativo utilizando el lenguaje de la lógica matemática [17], construyendo una base de conocimientos a través de hechos y reglas (lógica) entre diversos objetos que pueden ser capturados en un programa lógico para que con la ayuda de herramientas computacionales seamos capaces de contestar preguntas sobre los objetos descritos y encontrar soluciones a problemas particulares. La finalidad de un programa lógico es buscar todos los valores que hacen verdadero a este programa y como resultado de las combinaciones de los valores obtenidos resulte un modelo estable [7].

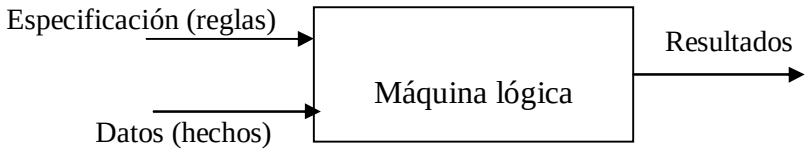
El concepto de un modelo estable o conjunto de respuestas (Answer Set), se utiliza para definir una semántica declarativa de programas lógicos, que están condicionados a una sintaxis particular muy restringida, el cual se define a continuación:

Definición *[14]. Dado un programa normal P, decimos que* $M \subseteq L_P$ *es un Modelo Estable (Answer Set) de P si M es un modelo para* P^M.

Intuitivamente, un modelo es estable si todo átomo en él tiene "alguna razón" para estar ahí, es decir, para cada átomo en el modelo tiene que existir alguna regla que tiene a ese átomo en la cabeza, de tal forma que el cuerpo de la regla es verdadera en el modelo. Un modelo estable es un conjunto de átomos y es una respuesta al programa lógico, esto indica qué átomos son verdaderos.

En los lenguajes tradicionales la programación consiste en indicar cómo resolver un problema mediante sentencias; en la programación lógica, se trabaja de una forma descriptiva, es decir, estableciendo relaciones entre entidades (objetos del mundo), sin establecer cómo se debe hacer, sino más bien qué hacer ya que es el programador quien suministra la parte lógica y el intérprete realiza la parte del control, es el que indica el orden en que se aplican las reglas [24].

La programación lógica es aquel tipo de programación que permite al software "razonar". Dada una base de datos consistente en un conjunto de entidades, propiedades de esas entidades y relaciones de unas entidades con otras, el sistema es capaz de hacer razonamientos. Básicamente, este proceso se expresa de la siguiente forma [13]:



Resultados = reglas + hechos

Donde se entiende que "hechos" son el conjunto de datos que conoce el sistema a priori (o que va adquiriendo a lo largo de su ejecución) y "reglas" son un conjunto de operaciones que se pueden aplicar a dichos datos para sacar un resultado lógico.

1.1.1 Sintaxis de la Programación Lógica

La Programación Lógica está basada en la lógica de primer orden o lógica de predicados donde [11]:

- Un predicado o su negación es llamado literal.
- Un hecho es una literal positiva.
- Una cláusula o regla es una fórmula de la forma [5]:

$$A \leftarrow . \qquad (I)$$
$$A \leftarrow B_1,\ldots,B_m. \qquad (II)$$
$$\leftarrow B_1,\ldots,B_m. \qquad (III)$$

La primera regla (I) representa un hecho porque no tiene cuerpo, donde el símbolo "←" se lee como "si" o como implicación y la parte izquierda de "←" se llama cabeza. La segunda es una regla donde $B_1,\ldots,B_m$ son un conjunto de literales llamados cuerpo de la regla. La tercera regla (III) indica una restricción debido a que no tiene cabeza. Las reglas son un conjunto de operaciones que se pueden aplicar a los datos existentes para obtener un resultado lógico.

Además la Programación Lógica hace uso de la Lógica Proposicional la cual se define de la siguiente manera [8, 9]:

- Variables proposicionales (P): p, q, r,…
- Conectivos lógicos: ∧ (conjunción), ∨ (disyunción), ¬ (negación), → (condicional), ↔ (bicondicional).
- Símbolos auxiliares: (,)

1.1.2 Lenguaje A

El lenguaje A es un elemento muy importante dentro de la Programación Lógica, ya que es el pseudocódigo de un programa para Planning (Planificación). El lenguaje A consiste de un alfabeto el cual consta de 2 conjuntos disjuntos de símbolos no vacíos llamados F y A [1], los cuales corresponden al conjunto de Fluents y el conjunto de Acciones respectivamente. A su vez el lenguaje de acción A está representado a través de tres sub-lenguajes que son: lenguaje de descripción del dominio, lenguaje de observación y lenguaje de consulta [1, 22]. Un ejemplo de un programa en lenguaje A es el siguiente:

```
robotEstaEnLugar(L).%fluent
robotMove(L1,L2). %acción mover
initially estaEnLugar(recepcion). %estado inicial
finally estaEnLugar(oA). %estado final

%execución de la acción
robotMove(L1,L2)
     causes estaEnLugar(L2)
          if holds(estaEnLugar(L1),T), T < length, time(T),
               lugar(L1), lugar(L2), neq(L1,L2).
```

donde la primera línea indica un fluent, la segunda la acción a ejecutar, la tercera y cuarta línea corresponden al lenguaje de observación y las líneas siguientes al lenguaje de descripción del dominio en donde se está indicando que cuando el robot ejecute la acción de moverse del lugar `L1` al lugar `L2` el efecto causado será que está en el lugar `L2` si se encuentra en el lugar `L1` y estos lugares son diferentes.

En el capítulo 2 se muestra la descripción completa del lenguaje A y la codificación para el problema de reparto de oficios.

1.2 Answer Set Programming (A-Prolog)

Answer Set Programming (ASP, también conocida como Programación Lógica Estable o A-Prolog) es un área de representación del conocimiento enfocado en la lógica, ba-

sado en lenguajes para el modelado de problemas de cálculo en términos de restricciones [1, 16]. Los orígenes de Answer Set Programming están en la programación lógica [2, 18] y el razonamiento no monótono [21, 25]. ASP tiene sus inicios cuando estas dos áreas se fusionaron en el trabajo de Gelfond y Lifschitz [23] donde se define la semántica estable.

Un problema es modelado por un programa lógico de manera que el Answer Set del programa corresponde directamente a la solución del problema [16, 19]. Los primeros sistemas de software que se desarrollaron para calcular Answer Set de los programas lógicos: DLV [3] y lparse/Smodels [15], demostraron que el paradigma de Answer Set Programming tiene un potencial para ser la base de la práctica del cálculo declarativo. Estos dos sistemas de software, sus descendientes y esencialmente todos los otros sistemas que han desarrollado y aplicado hasta ahora ASP, contienen dos componentes principales [6]: un grounder que es una entrada del programa que se encarga de producir su equivalente proposicional y un solver (solucionador) que acepta el programa de entrada y calcula los Answer Set correspondientes.

La técnica de Answer Set Programming se basa en la posibilidad de representar ciertas colecciones de conjuntos como colecciones de modelos estables [12]. Gracias a la existencia de implementaciones eficientes para calcular modelos estables tales como Smodels[1], DLV[2], Clasp[3], enfocados a resolver problemas en base a la semántica estable, la gama de problemas que podían enfrentarse con este nuevo paradigma creció rápidamente para incluir problemas combinatorios, establecer y resolver problemas de planeación, modelar el comportamiento de agentes lógicos y aplicaciones de inteligencia artificial en general [17].

1.2.1 Semántica

La introducción de la semántica de modelos estables (Stable Models), por Gelfond y Lifschitz [23], significó un importante adelanto en el área de la programación lógica, debi-

[1] http://www.tcs.hut.fi/Software/smodels/
[2] http://www.dbai.tuwien.ac.at/proj/dlv/
[3] http://www.cs.uni-potsdam.de/clasp/

do a que se ha convertido en la base de Answer Set Programming. La semántica de Answer Set Programming presenta una característica innovadora, que es el uso de dos tipos de negación [17]: la negación como falla y la negación clásica o explícita. Esta segunda negación se denota con el conectivo ~, que permite la especificación explícita de conocimiento que de antemano se sabe que es falso, por otra parte, la negación como falla que se denota por el símbolo ¬, significa que una fórmula ¬F es cierta si no es posible mostrar que esta fórmula F es verdadera con la información existente.

Por lo tanto, para que un programa lógico tenga un significado y pueda ser interpretado por programas computacionales se debe asignar una semántica, intuitivamente una semántica es una forma de determinar el tipo de conclusiones que se pueden establecer a partir de un conjunto de reglas. De tal manera, un problema debe codificarse en la forma de un programa lógico buscando que la semántica del programa capture precisamente las soluciones del problema en cuestión [17]. Debido a esto, surgen lenguajes que permiten escribir programas lógicos que soportan dichas expresiones tales como el uso de Answer Set Programming.

El éxito de la semántica de los modelos estables y posteriormente de los Answer Sets, se debió en gran medida a su capacidad para resolver problemas.

1.3 Sistemas ASP

Actualmente existen varios solucionadores para Answer Set, para este caso se hará mención de 2 de ellos: Smodels y Clasp, debido a que serán los solucionadores que se usarán para el desarrollo de la implementación.

1.3.1 Smodels

Smodels fue desarrollado por Patrik Simons en el Laboratorio de Ciencias Computacionales Teóricas de la Universidad Tecnológica de Helsinki, es una eficiente implementación de la semántica de modelos estables para programas lógicos, es un sistema para

Answer Set Programming. A su vez Smodels necesita de lparse, el cual es un front – end que se encarga de transformar los programas del usuario de tal manera que Smodels pueda entenderlos.

1.3.2 Clasp

Clasp es un sistema solucionador de Answer Sets para extender programas lógicos normales que forma parte de la suite de los laboratorios de Potassco[4]. Clasp permite además el uso de declaraciones de optimización mediante las herramientas gringo y clingo de Potassco, para establecer preferencias entre los answer sets resultantes y poder definir si se trata de un answer set óptimo para el programa lógico definido.

[4] http://potassco.sourceforge.net/

Capítulo 2

Modelado de reparto de oficios utilizando el lenguaje A

Recordemos que el problema a implementar está enfocado a la entrega de oficios para un caso particular, en donde específicamente se describe un robot que debe tomar los oficios correspondientes al empleado de una oficina que deberá entregar en un periodo de tiempo determinado mediante la ejecución de ciertas acciones y llevando el control de los tiempos en que realiza cada acción para que éste pueda realizar la entrega correspondiente en un horario que ha especificado el empleado. La implementación de este problema se ha realizado primero para el caso en que el robot entrega los oficios en una oficina y poco a poco se ha ido incrementando el número de oficinas de acuerdo a las reglas que se irán definiendo. Algo importante que se debe mencionar es, que es posible aplicar este modelo de entrega a diversos objetos y no únicamente a oficios.

2.1 Descripción del Lenguaje A

Como ya se ha mencionado, el lenguaje A consiste de dos conjuntos disjuntos llamados F y A.

Los *fluents* forman parte de la descripción del estado del mundo y representan las propiedades de éstos objetos. Un *fluent literal* es un fluent el cual puede estar precedido por la negación ¬ o simplemente el fluent, es decir un fluent positivo o un fluent negado. Un conjunto de fluents forman un *estado* σ, y decimos que un fluent *f* se cumple en el estado σ si $f \in \sigma$, por el contrario, decimos que un fluent literal ¬*f* se cumple en σ si *f* no está contenido en el estado σ.

Las acciones, representan las operaciones aplicadas a los fluents que permiten realizar cambios de estados. Cuando una acción se ejecuta, indica que se está mostrando la si-

tuación de lo que sucede en el transcurso de esa ejecución. Si la acción se ejecuta con éxito, cambia el estado del mundo. Las situaciones son términos que identifican un instante dado. Son del estilo [10]: situación inicial (S_0) y situación después de ejecutar alguna acción *a* en una situación *s* (*Result(a, s)*).

Al momento de indicar la situación inicial, ninguna acción se ha ejecutado, lo que se representa como una lista vacía [ ]. La situación [a_n,…a_1] indica la historia en donde se ejecuta la acción a_1 en la situación inicial, seguida por la siguiente acción a_2 y así sucesivamente hasta a_n, esto indica una relación entre situación y estados; el estado del mundo es el estado que corresponde a la situación *s*, en cada situación *s*, algunos fluents son verdaderos y otros falsos.

Los tres lenguajes que representan el lenguaje de acción A son los siguientes:

Lenguaje de descripción del dominio. Este lenguaje es usado para indicar lo que ocurre entre estados al momento de ejecutar las acciones. La descripción del dominio D consiste de proposiciones de efecto que son de la siguiente forma:

$$a \textbf{ causes } f \textbf{ if } p_1,\ldots,p_n,\ \neg q_1,\ldots,\neg q_r \qquad (1)$$

donde a indica la acción, f y $p_1,\ldots,p_n,\ q_1,\ldots,q_r$ son fluents, los cuales también pueden estar negados $\neg q_1,\ldots,\neg q_r$. La proposición anterior significa que si los fluents literals $p_1,\ldots,p_n,\ \neg q_1,\ldots,\neg q_r$ se cumplen en el estado correspondiente a una situación *s*, entonces en el estado correspondiente a la situación alcanzada por ejecutar la acción *a* en la situación *s*, el *fluent literal f* debe cumplirse. Si n=0 y r=0 en (1) entonces tenemos [7]:

$$a \textbf{ causes } f \qquad (2)$$

También es posible agrupar varias proposiciones de efecto, es decir, $\{a \text{ causes } f_1,\ldots,\ a \text{ causes } f_m\}$ por:

$$a \textbf{ causes } f_1,\ldots,\ f_m \qquad (3)$$

Las proposiciones de efecto definen una función de transición que va de los estados y las acciones, a los estados. Sea una descripción del dominio D, una función de transición Φ debe satisfacer las siguientes propiedades para que pueda ser definida dicha transición, por lo tanto, para todas las acciones *a*, fluents *g* y estados σ se debe cumplir:

1. Si D incluye una proposición de efecto como en (1) en donde *f* es el fluent *g* y si $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ se cumplen en σ entonces $g \in \Phi(a, \sigma)$.

2. Si D incluye una proposición de efecto de la forma (1) en donde *f* es un fluent negativo $\neg g$ y si $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ se cumplen en σ entonces *g* no está en $\Phi(a, \sigma)$.

3. Si D no cuenta con tales proposiciones de efecto, entonces $g \in \Phi(a, \sigma)$ si y solo si se cumple que $g \in \sigma$.

Una descripción del dominio D, tiene a lo más una función de transición que satisface las propiedades anteriores. Si existe esa función de transición, entonces podemos decir que D es consistente y representamos a la función de transición mediante Φ_D. Un ejemplo de una descripción inconsistente del domino D es el siguiente:

a **causes** f
a **causes** ¬f

La parte de descripción del dominio, también puede incluir ejecutabilidad de condiciones, de la siguiente forma:

executable a **if** $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$

donde a es una acción y $p_1, \ldots, p_n, q_1, \ldots, q_r$ son fluents. Esto significa que si los fluents literals $p_1, \ldots, p_n, \neg q_1, \ldots, \neg q_r$ se cumplen en el estado σ de una situación *s*, entonces la acción *a* es ejecutada en *s*.

Lenguaje de observación: El lenguaje de observación es un conjunto de observaciones O, formado por valores de proposiciones de la siguiente forma [26]:

$$\mathtt{f}\ \textbf{after}\ \mathtt{a_1,..,a_m} \qquad (4)$$

en donde f es un fluent y $a_1,..,a_m$ son las acciones. Esta proposición significa que si las acciones $a_1,..,a_m$ se ejecutan en el estado inicial entonces en el estado correspondiente a la situación $[a_1,..,a_m]$ el fluent f se cumple. En otro caso, cuando la secuencia de acciones $a_1,..,a_m$ es una secuencia vacía, únicamente se indica el fluent marcándolo como estado inicial:

$$\textbf{initially}\ \mathtt{f} \qquad (5)$$

Dada la descripción consistente del domino D, el conjunto de observaciones O es usado para determinar los estados correspondientes a la situación inicial, referido como estados iniciales y denotado por σ_0.

Se dice que (σ_0, Φ_D) satisface O, si y sólo si, σ_0 es un estado inicial que corresponde a una descripción consistente del dominio D y a un conjunto de observaciones O, si para todas las observaciones de la forma (4) en O, el fluent literal f se cumple en el estado $\Phi(a_m, \Phi(a_{m-1},\ldots\ \Phi(a_1, \sigma_0)\ldots))$, donde σ_0 es un estado inicial que corresponde a (D, O) y Φ_D es la función de transición. De esta forma se puede decir que (D, O) es consistente si este tiene un modelo y si tiene un modelo único decimos que es completo.

Lenguaje de consulta. Una consulta Q consiste de proposiciones de la forma (4). Decimos que un dominio consistente de descripción D, en la presencia de un conjunto de observaciones O implica una consulta Q de la forma (4) si para todos los estados iniciales σ_0 que corresponde a (D, O), el fluent literal *f* se cumple en el estado $[a_m,\ldots,a_1]\,\sigma_0$. Lo cual denotamos como $D \models_O Q$.

Se dice que un conjunto de observaciones O es un estado inicial completo, si este conjunto de observaciones consiste de proposiciones de la forma (5) y además se cumple

que para todos los fluents `f`, cada *initially f* o *initially* ¬*f* está en el conjunto O, pero no ambos.

Por lo tanto, con el fin de modelar un problema de planificación usando el lenguaje A, se debe especificar una tercia (D, O, G), donde D es una descripción del dominio, O es un conjunto de observaciones, y G es una colección de fluents literals $G = \{g_1,..., g_l\}$, a la que nos referiremos como una meta. Así que se requiere encontrar una secuencia de acciones $a_1, ..., a_n$ tal que para todo $1 \leq i \leq l$, $D \models_0 g_i$ **after** $a_1, ..., a_n$. Entonces, decimos que $a_1, ..., an$ es un plan para lograr la meta G con respecto a (D, O).

2.2 Modelado del problema de reparto de oficios en una oficina utilizando el lenguaje A

En esta sección se hace uso del lenguaje A para modelar el problema enfocado al caso particular de reparto de oficios, correspondiente a la primera versión donde se entregan los oficios a una oficina. Mediante el uso del símbolo % indicaremos los comentarios dentro del código. En la descripción del código presentamos los fluents, las acciones, las condiciones de ejecutabilidad, los efectos de acciones y los estados iniciales y finales, los cuales se definen a continuación:

Fluentes

`tiempoSalida(T).` %Representa el tiempo `T` en que el robot sale de un lugar.
`tiempoLlegada(T).` %Representa el tiempo `T` en que el robot llega a un lugar.
`estaEnLugar(L).` %Indica el lugar `L` en que se encuentra el robot
`robotTieneOficios.` %Indica que el robot tiene los oficios que va a entregar.
`entregoOficiosEnLugar(L).` %Establece que el robot ha entregado los oficios en el lugar `L`.

Acciones

`move(L1,L2).` %Representa que el robot se moverá del lugar `L1` al lugar `L2`.
`tomarOficios.` %Indica que el robot debe tomar los oficios a entregar.
`entregarOficios(L,T_entrega).` %Indica que el robot debe hacer entrega de los oficios en el lugar `L` en el tiempo de entrega `T_entrega` que se haya establecido.

Efectos y ejecutabilidad de condiciones

```
move(L1,L2)
     causes estaEnLugar(L2)
          if holds(estaEnLugar(L1),T), T < length, time(T),
               lugar(L1), lugar(L2), neq(L1,L2).
```

esto significa que si se cumple que el robot se encuentra en el lugar `L1` en el tiempo `T` indicado en la regla `holds(estaEnLugar(L1),T)`, y ambos lugares `L1` y `L2` son diferentes `neq(L1,L2)`, entonces es posible ejecutar la acción `move(L1,L2)`. La ejecución de la acción `move(L1,L2)` causa que el robot se encuentre en el lugar `L2`, es decir, `estaEnLugar(L2)`. Para la ejecución de la acción tomar oficios será de la siguiente manera:

```
tomarOficios
     causes robotTieneOficios
          if holds(estaEnLugar(recepcion),T),
             holds(neg(robotTieneOficios),T),
             T < length, time(T).
```

esto nos indica que si el robot desea tomar los oficios debe cumplirse primero que se encuentre en la recepción, el cual es el único lugar de donde podrá tomar los oficios, indicado por la regla `holds(estaEnLugar(recepcion),T)` y que el robot no tenga los oficios `holds(neg(robotTieneOficios),T)`, solo de esta manera podrá ejecutarse la acción `tomarOficios` lo que causará que el robot tenga los oficios `robotTieneOficios`.

Para esta última ejecución de la acción entregar oficios, necesitaremos considerar más reglas, como se muestra:

```
entregarOficios(L,T_entrega)
     causes neg(robotTieneOficios), entregoOficiosEnLugar(L)
          if T < length, time(T), holds(estaEnLugar(L),T),
               holds(robotTieneOficios,T),
               holds(neg(entrego_oficios_en_lugar(L)),T),
               holds(tiempo_llegada(T_llegada),T),
               horario_oficios(L,T0,T1),
               T0<= T_llegada, T_llegada <= T1,
```

```
assign(T_entrega,T_llegada),
lugar(L), timeh(T_entrega),
timeh(T_llegada), timeh(T0), timeh(T1).
```

de esta manera indicamos que si se cumple que el robot está en el lugar `L` en donde realizará la entrega `holds(estaEnLugar(L),T)`, tiene los oficios correspondientes para entregar `holds(robotTieneOficios,T)`, además también se sabe que el robot no ha entregado los oficios en ese lugar `holds(neg(entregoOficiosEnLugar(L)),T)`, y se conoce el tiempo en que el robot llegó al lugar `L` indicado por `holds(tiempoLlegada(T_llegada),T)`, con lo que es posible que a través del horario que el empleado indicó para recibir los oficios `horario_oficios(L,T0,T1)` se realice una comparación entre el tiempo de llegada con los tiempos establecidos, es decir, `T0 <= T_llegada, T_llegada <= T1`, y si se cumple esta comparación se asigne el tiempo de llegada al tiempo de entrega `assign(T_entrega,T_llegada)`. Por lo que todo esto dará como resultado que sea posible ejecutar la acción de entregar los oficios. La ejecución de esta acción permitirá establecer que el robot ya no tiene los oficios `neg(robotTieneOficios)`, ya que se ha realizado la entrega de los oficios en el lugar correcto `entregoOficiosEnLugar(L)`.

Estado inicial

El estado inicial permite conocer el lugar en que se encuentra el robot antes de realizar la entrega, el tiempo en que saldrá de ese lugar y a que oficinas se debe entregar los oficios. Esto se establece de la siguiente manera:

```
initially estaEnLugar(recepcion).  %Lugar de partida que es la recepcion.
initially neg(robotTieneOficios).  %El robot no tiene los oficios a entregar.
initially tiempoSalida(1).  %El robot inicia en el tiempo 1.
initially neg(entregoOficiosEnLugar(oA)).  %Aún no se han entregado
los oficios en el lugar indicado.
```

Estado final

El estado final representa la meta del plan e indica el resultado obtenido después de haber aplicado las acciones correspondientes al estado inicial, por lo que se obtiene lo siguiente:

```
finally entregoOficiosEnLugar(oA).
```
%Entrega de los oficios en el lugar indicado.

```
finally estaEnLugar(oA).
```
%Al realizar la entrega el robot se encuentra en ese lugar.

Como indicamos al principio de este capítulo sólo se mostrará el modelado de la entrega de oficios en una oficina pero es posible agregar más oficinas por lo que sería necesario también agregar algunas reglas más.

Capítulo 3

Modelado del problema de reparto de Oficios

En este capítulo se presenta la descripción del problema de reparto de oficios para un caso general a través del problema de Secuenciación Mínima con Tiempo de Realización mediante la aplicación de Answer Set Programming y el uso de preferencias, las implementaciones del problema particular, la descripción del código y las modificaciones realizadas en las implementaciones. Como se había mencionado, para obtener los correspondientes Answer Sets se usarán dos herramientas: Smodels y Clasp. Una vez que se hayan obtenido los Answer Set de cada implementación será necesario comparar dichos modelos, así como analizar las variantes de cada uno y posteriormente realizar los cambios necesarios para obtener mejores resultados en las siguientes pruebas, teniendo en cuenta los tiempos de realización de las acciones. Esta aplicación fue tomada por el interés en el estudio del modelado de preferencias en Answer Set Programming. Nuestra atención se centró en modelar problemas de secuenciación mínima con tiempos de realización aplicando preferencias para obtener answer set que den una solución óptima a un problema de reparto.

3.1 Descripción del problema

En esta sección se describe el caso particular del problema de Secuenciación Mínima, la cual consiste en el reparto de oficios a los empleados de oficinas. Este problema particular está enfocado en un robot que tiene como principal tarea llevar dichos oficios a los empleados de las diferentes oficinas.

Los elementos que componen el mundo para este problema son el robot que será el encargado de entregar los oficios, las oficinas en donde el robot hará la entrega correspondiente de los oficios a cada uno de los empleados y la recepción que es el lugar en donde el

robot podrá tomar los oficios. Para llevar a cabo esta tarea, se describirá el estado inicial del programa al cual se le aplicarán las acciones correspondientes para obtener los Answer Set que indiquen la solución de cómo el robot logra entregar a tiempo los oficios solicitados.

Inicialmente, el robot está en la recepción y los oficios se encuentran sobre una mesa, los cuales están organizados en carpetas donde cada carpeta tiene escrito el nombre del empleado al cual deberán dirigirse los oficios, por lo tanto, si el robot se encuentra en la recepción solo debe tomar los oficios correspondientes, teniendo claro que eso representa la carpeta con el nombre del empleado al que se entregarán los oficios. Una vez que el robot haya tomado los oficios deberá proceder a su correspondiente entrega, la cual realizará a través de una secuencia de acciones definidas y en base a los horarios que el empleado haya indicado para recibir sus oficios y que el robot ya conoce, además el robot tiene indicados los tiempos que ocupa para trasladarse (tiempo de viaje) del estado inicial (recepción) a cada oficina y viceversa, estos tiempos de ida y vuelta deben ser los mismos. El robot solo puede llevar una carpeta de oficios y hacer una entrega en un periodo de tiempo, entonces cada vez que el robot deba llevar una carpeta de oficios a varias oficinas tendrá que regresar a la recepción por otra carpeta de oficios para la nueva entrega.

Para este problema particular se establece la entrega de oficios, sin embargo, este modelado puede amoldarse a la entrega de diversos objetos, por ejemplo: herramienta, instrumental, paquetes, sobres, café, etc. También se muestra la versión en donde se entregan los oficios en una oficina pero el modelado puede crecer en base al número de oficinas que se establezcan. Las pruebas que se realizaron para la entrega de oficios en ambas herramientas (Smodels y Clasp) fueron de una a cinco oficinas, algunas de estas pruebas se muestran en el Apéndice y todas las pruebas realizadas se pueden observar en la siguiente página con sus respectivos resultados: *https://sites.google.com/site/codigoentregadeoficios/*

La finalidad de este trabajo es modelar la forma en que el robot reparte dichas entregas, es decir, el robot debe entregar el oficio asignándole un tiempo a cada entrega para que cada empleado reciba su oficio en el periodo de su preferencia.

3.2 Implementaciones

Primero se muestra la codificación en Smodels que fue la primera parte con la que se trabajó realizando varias pruebas para obtener una versión final, y posteriormente la codificación en Clasp, usando restricciones de cardinalidad y preferencias. En ambos casos se obtuvieron los Answer Sets de cada prueba para comparar los modelos obtenidos y analizar las variantes de cada una.

3.2.1 Implementación para el reparto de oficios en una oficina utilizando Smodels

Siguiendo con lo planteado anteriormente, se describe ahora la formulación del código correspondiente a la primera versión modelada en Smodels que posteriormente se codificará en Clasp para su comparación. En esta primera versión el robot tiene como primera meta entregar el oficio a solo una oficina.

Por lo tanto, se describirán los elementos del mundo en el cual se va a trabajar. Para esta parte es necesario saber cuáles son las constantes, los lugares (oficinas), los fluents, las acciones que debemos usar para llegar a la meta y los estados iniciales y finales a los que se quiere llegar. El código es el siguiente:

Declaración de constantes

Mediante el uso de la palabra `const` se definen las constantes que se usarán, las cuales son:

```
const length=4. %Indica el tiempo máximo para llegar a la meta
time(1..length).

const reloj=4. %Indica la longitud de los pasos para realizar la acción
timeh(0..reloj).
```

a través de las constantes `length` y `reloj` se muestra el tiempo máximo que debe transcurrir para llegar a la meta, el uso de `time(1..length)` y `timeh(0..reloj)` son abreviaturas que ayudan a reducir código e incluyen 4 reglas respectivamente, que son las siguientes:

```
time(1). time(2). time(3). time(4). y
timeh(0). timeh(1). timeh(2). timeh(3).
```

Declaración de hechos

Para definir los lugares correspondientes a las oficinas entre los cuales se moverá el robot, estos se declaran de la siguiente manera:

```
lugar(recepcion). % Lugar recepción donde el robot tomará los oficios
lugar(oA).  %Indica el lugar (oficina A)
```

la declaración de los lugares son hechos, debido a que son datos que el robot conoce y por lo tanto no tienen cuerpo. Como se había mencionado, el robot también conoce los horarios establecidos por los empleados, por lo que estos también forman parte de los hechos y se escriben de la siguiente forma:

```
horario_oficios(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).
```

donde `horario_oficios(oA,2,3)` indica que el empleado de la oficina `oA` desea recibir el oficio en el horario de 2 a 3. En la segunda y tercera línea `tiempo_viaje(recepcion,oA,1)` y `tiempo_viaje(oA,recepcion,1)` se indican las unidades de tiempo que el robot utilizará para moverse de la recepción a la oficina y viceversa, las cuales deberán ser iguales, en ambos casos le tomará al robot una unidad de tiempo.

Declaración de fluentes

Ahora se definen los fluents, que indican las propiedades del mundo, estos son:

```
fluent(tiempo_salida(T)) :- timeh(T).
fluent(tiempo_llegada(T)) :- timeh(T).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(robot_tiene_oficios).
fluent(entrego_oficios_en_lugar(L)) :- lugar(L).
```

los dos primeros fluents `tiempo_salida(T)` y `tiempo_llegada(T)` marcan los tiempos en que el robot sale y llega respectivamente de un lugar. El fluent `esta_en_lugar(L)` indica la ubicación del robot, la cual puede ser la recepción o alguna oficina, `robot_tiene_oficios` significa que el robot ha tomado los oficios correspondientes y está listo para entregarlos, y finalmente el fluent `entrego_oficios_en_lugar(L)`, indica que el empleado de la oficina ha recibido a tiempo sus oficios.

Declaración de acciones

Para que el robot pueda llegar a la meta de entregar los oficios a una oficina, es necesario definir las acciones, por lo tanto estás serán las acciones que el robot deberá ejecutar:

```
action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficios).
action(entregar_oficios(L,T_entrega)):-
              timeh(T_entrega,lugar(L).
```

en la primera acción `move(L1,L2)` se indica que el robot se mueve del lugar `L1` al lugar `L2` para la correspondiente entrega de los oficios, la acción `tomar_oficios` significa que el robot debe tomar los oficios que va a entregar, y por último, la acción `entregar_oficios(L,T_entrega)` señala que el robot debe entregar los oficios a la oficina `L` en el tiempo de entrega indicado.

Declaración de los estados iniciales

El estado inicial se establece mediante el uso de la palabra `initially` para indicar de donde partirá el robot:

```
initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).
initially(neg(entrego_oficios_en_lugar(oA))).
```

con el uso de la regla `esta_en_lugar(recepcion)` se indica que el robot se encuentra en la recepción que es el lugar en donde tomará los oficios y saldrá inicialmente para entregarlos, mediante la regla `neg(robot_tiene_oficios)` se indica que el robot aún no ha tomado los oficios de la mesa en la que se encuentran, con la regla `tiempo_salida(1)` se establece que el tiempo en que el robot saldrá de la recepción es 1, mediante `neg(entrego_oficios_en_lugar(oA))` se indica que el empleado de la oficina `oA` aún no ha recibido sus oficios.

Declaración de los estados finales

Una vez definidos los estados iniciales y a través de la ejecución de las acciones ya mencionadas es posible llegar a una meta. La meta a alcanzar es que el empleado obtenga sus oficios en el horario de su preferencia indicado anteriormente, por lo que se definen de la siguiente manera los estados finales mediante la palabra reservada `finally`:

```
finally(entrego_oficios_en_lugar(oA)).
finally(esta_en_lugar(oA)).
```

en la primera regla `entrego_oficios_en_lugar(oA)` se indica que el robot ha entregado los oficios satisfactoriamente en la oficina `oA` y que por lo tanto el empleado de esa oficina los ha recibido en el horario indicado. La segunda regla `esta_en_lugar(oA)` establece que si el robot ha entregado los oficios en la oficina `oA`, por lo tanto se encuentra en esa oficina.

Declaración de las condiciones de ejecutabilidad

Ahora es necesario indicar de qué forma o qué condiciones deben cumplir las acciones que se han definido para que estas puedan ser ejecutadas en los fluents y podamos obtener resultados correctos para la entrega satisfactoria de los oficios. Mediante el uso de la palabra reservada `executable` se define la ejecución de las tres acciones que se describen de la siguiente manera:

Para poder ejecutar la primera acción `move(L1,L2)` en el tiempo `T`, que indica el tiempo actual en que se realizan las acciones, implica que debe cumplirse que este tiempo sea menor al tiempo máximo establecido en la constante `length`, dado por `T < length`, y que esta variable `T` sea del tipo `time` para que pueda ser comparada con la constante, definido como `time(T)`, además también se debe cumplir que el robot se encuentre en el lugar `L1` para que pueda moverse al lugar `L2`, de lo contrario no podría realizarse el movimiento, esto se define mediante el uso de la regla `holds(esta_en_lugar(L1),T)`, y también se debe indicar que `L1` y `L2` sean lugares y a la vez sean diferentes, ya que no tendría caso moverse al mismo lugar. Esta descripción corresponde a la declaración de ejecutabilidad que se muestra a continuación:

```
executable(move(L1,L2),T) :- T < length, time(T),
                    holds(esta_en_lugar(L1),T),
                    lugar(L1), lugar(L2), neq(L1,L2).
```

La ejecución de la acción `tomar_oficios` se cumple si el robot está en la recepción para que sea posible tomar los oficios, indicado en la regla `holds(esta_en_lugar(recepcion),T)` y que en ese mismo momento el robot no haya tomado los oficios, definido mediante la regla `holds(neg(robot_tiene_oficios),T)` porque no tendría caso tomar los oficios si ya los tiene y recordemos que solo puede tomar los oficios de un empleado a la vez. Esto se representa en la declaración de ejecutabilidad siguiente:

```
executable(tomar_oficios,T) :- T < length, time(T),
                        holds(esta_en_lugar(recepcion),T),
                        holds(neg(robot_tiene_oficios),T).
```

Por último se muestra la ejecución de la acción `entregar_oficios`, la cual se cumple bajo las siguientes restricciones: `L` debe ser un lugar y `T_entrega` de tipo `timeh` que está definida para llevar el conteo de los pasos para que se realicen las acciones, el robot debe estar en el lugar `L` en donde realizará la entrega indicada en la regla `holds(esta_en_lugar(L),T)`, el robot debe tener los oficios a entregar indicado en la regla `holds(robot_tiene_oficios,T)`, el robot no ha entregado los oficios en el lugar indicado con la regla `holds(neg(entrego_oficios_en_lugar(L)),T)` sino ya no tendría caso llevar los oficios a esa oficina. Es importante también considerar el tiempo en que el robot llega a la oficina ya que ayudará a verificar que se entregó el oficio en el horario indicado, manejando ese tiempo de llegada con la regla `holds(tiempo_llegada(T_llegada),T)` y considerar el horario establecido para recibir los oficios mediante `horario_oficios(L,T0,T1)` que es parte del conocimiento previo del robot. Por otra parte, debemos comprobar que efectivamente el tiempo en que llegó el robot a la oficina está dentro del intervalo de tiempo que el empleado indicó, esto se muestra de la siguiente manera: `T0 <= T_llegada, T_llegada <= T1.` Una vez realizada esta comparación, tenemos que asignar el tiempo de llega a una nueva variable llamada `T_entrega`, esto con el fin de que el robot no pierda el conteo de los pasos realizados, y finalmente que las variables `T_llegada, T0` y `T1` sean del mismo tipo `timeh`. Esta condición de ejecutabilidad queda definida de la siguiente manera:

```
executable(entregar_oficios(L,T_entrega),T):- T < length,
               time(T), lugar(L), timeh(T_entrega),
               holds(esta_en_lugar(L),T),
               holds(robot_tiene_oficios,T),
               holds(neg(entrego_oficios_en_lugar(L)),T),
               holds(tiempo_llegada(T_llegada),T),
               horario_oficios(L,T0,T1),
               T0<= T_llegada, T_llegada <= T1,
               assign(T_entrega,T_llegada),
               timeh(T_llegada), timeh(T0), timeh(T1).
```

Declaración de las reglas de causa

Cada una de las ejecuciones anteriores causará diferente efecto en los fluents correspondientes, por tal motivo es necesario definir reglas de causa, estas reglas de causa permitirán obtener el resultado final de cada ejecución. Para la ejecución de la acción `move(L1,L2)` tenemos las siguientes reglas de causa:

```
causes_dd(esta_en_lugar(L2),move(L1,L2),esta_en_lugar(L1)):-
lugar(L1), lugar(L2).
```

en esta primera regla de causa se establece la ubicación del robot, indicando que se encuentra en el lugar `L2` definida mediante el fluent `esta_en_lugar(L2)` si se ha ejecutado la acción `move(L1,L2)`, y en ese mismo tiempo el robot se encuentra en el lugar `L1` indicado en el fluent `esta_en_lugar(L1).`

Por otra parte, la segunda regla establece lo contrario de la primera, es decir:

```
causes_dd(neg(esta_en_lugar(L1)),move(L1,L2),
esta_en_lugar(L1)) :- lugar(L1), lugar(L2), neq(L1,L2).
```

se indica que ya no está en el lugar `L1` definido mediante el fluent `neg(esta_en_lugar(L1))`, después de que se ha ejecutado la acción `move(L1,L2)` y en ese momento el robot estaba en el lugar `L1` indicado en el fluent `esta_en_lugar(L1)`, puesto que ahora se encuentra en el lugar `L2`. Debemos tener en cuenta que para que esto suceda los lugares a los cuales se moverá el robot deben ser diferentes porque no tendría caso moverse a un lugar si se encuentra en el.

La tercera regla de causa que resulta de la ejecución de la acción `move(L1,L2)` es:

```
causes_dd(tiempo_llegada(T_llegada), move(L1,L2),
tiempo_salida(T_salida)) :- tiempo_viaje(L1,L2,Tv),
T_llegada = Tv + T_salida, timeh(T_salida), timeh(T_llegada),
timeh(Tv), lugar(L1), lugar(L2), neq(L1,L2).
```

en esta regla de causa se obtiene el tiempo en que el robot llega a una oficina o a la recepción, el cual se va actualizando cada vez que se ejecuta la acción mover, teniendo conocimiento del tiempo en que salió de algún lugar. Para obtener este tiempo, debemos sumar el tiempo en que el robot tardó en trasladarse al lugar llamado `Tv` el cual indica el tiempo de viaje más el tiempo en que salió `T_salida`, es decir, `T_llegada = Tv + T_salida`. De lo contrario como se indica en la siguiente regla, no podrá obtenerse el tiempo de llegada si ya se sabe ese tiempo:

```
causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2),
tiempo_llegada(T_llegada)) :- timeh(T_llegada), lugar(L1),
lugar(L2), neq(L1,L2).
```

Este mismo caso se establece en la siguiente regla de causa que define el tiempo de salida:

```
causes_dd(neg(tiempo_salida(T_salida)), move(L1,L2),
tiempo_salida(T_salida)) :- timeh(T_salida), lugar(L1),
lugar(L2), neq(L1,L2).
```

en donde también se niega haber obtenido el tiempo de salida si ya se tenía ese tiempo, lo que quiere decir que no se puede tener el mismo tiempo después de haberse ejecutado una acción, ya que cada ejecución consume un tiempo determinado.

La única regla de causa para la ejecución de la acción tomar oficios es la siguiente:

```
causes(tomar_oficios, robot_tiene_oficios).
```

la cual establece que si el robot ejecuta la acción de tomar los oficios, causa que el robot tenga los oficios a entregar.

Para la ejecución de la acción entregar oficios, se tienen dos reglas que son las siguientes:

```
causes(entregar_oficios(L,T_entrega),
neg(robot_tiene_oficios)) :- timeh(T_entrega), lugar(L).
```

en esta primera regla se establece que si el robot ejecuta la acción de entregar los oficios en el lugar y tiempo indicados `entregar_oficios(L,T_entrega)`, el efecto causado será que el robot ya no tiene los oficios `neg(robot_tiene_oficios)`. Por otra parte, en la segunda regla:

```
causes(entregar_oficios(L,T_entrega),
entrego_oficios_en_lugar(L)) :- timeh(T_entrega), lugar(L).
```

se establece que al ejecutar la misma acción `entregar_oficios(L,T_entrega)`, causará que el robot haya entregado los oficios en el lugar indicado `entrego_oficios_en_lugar(L)`.

Declaración de la meta

Las reglas para definir la meta nos sirven para indicar en qué condiciones queremos que una meta sea alcanzada, especificando el tiempo máximo para que el plan sea concluido. Las reglas son las siguientes:

```
not_goal(T):- time(T),literal(X), finally(X), not holds(X,T).
goal(T) :- time(T),not not_goal(T).
```

en la primera regla se indica que no se ha llegado a la meta en el tiempo indicado `not_goal(T)` si se cumple que `X` es un literal `literal(X)` y un estado final `finally(X)` y no hay evidencia de que `X` se cumpla en el tiempo establecido `not holds(X,T)`, por lo tanto no se ha llegado a la meta. La segunda regla establece que se llegó a la meta `goal(T)` si no hay evidencia de que no se ha llegado a la meta `not not_goal(T)`.

Verificación y restricción del un plan

Ya que se ha indicado que debe existir una meta y en qué condiciones debe cumplirse, también es necesario verificar que exista un plan para poder llegar a esa meta, por lo que se define la siguiente regla:

```
exists_plan :- goal(T).
```

la cual establece que existe un plan si hay una meta que debe cumplirse en un determinado tiempo.

Por otro lado también se debe establecer una restricción cuando ocurra el caso de que no se tiene un plan, ya que no podríamos llegar a una meta si no contamos con un conjunto de pasos que nos lleven a un objetivo. Entonces con esta restricción:

```
:- not exists_plan.
```

se establece que no se quiere que no haya evidencia de que existe un plan. Con esto estamos forzando de alguna manera a que siempre se tenga un plan para cumplir una meta determinada, que en nuestro caso es que el robot logre hacer entrega de ciertos oficios a los empleados de oficinas.

Declaración de literal

Como ya se había mencionado en el capítulo 1, una literal es un fluent que puede ser positivo o negativo. En este caso la declaración de `literal`, nos indica exactamente eso, que podemos hacer uso de la negación y no negación de una sentencia `G` si `G` es un fluent, lo cual declaramos como:

```
literal(G)  :- fluent(G).
literal(neg(G)) :- fluent(G).
```

Declaración de contrary

Es necesario también la declaración de `contrary` (contrario), debido a que en esta formulación no se hace uso de la negación fuerte "-" como lo es en otros lenguajes que se utiliza para representar cuando un fluent es negado, por tal motivo, definimos un contrario para simular el comportamiento de esta negación:

```
contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).
```

esto nos indica que F es lo contrario de neg(F) y a su vez neg(F) es también lo contrario de F si se cumple que F es un fluent.

Declaración de holds

Ahora también se debe indicar que fluents se cumplen en el tiempo 1:

```
holds(F, 1) :- literal(F), initially(F).
```

en esta regla se define que un fluent F se cumple en el tiempo 1 si F es un literal y además ese fluent F esta declarado como estado inicial.

Axiomas de efecto

Los axiomas de efecto son reglas que indican las condiciones en que es posible que se cumplan las reglas de causa. De la misma manera en que se han declarado dos tipos diferentes de reglas de causa, tenemos esos dos tipos de axiomas de efecto que corresponden a las reglas descritas anteriormente, las cuales son:

```
holds(F, T+1) :- literal(F), time(T), T < length, action(A),
                 executable(A,T), occurs(A,T), causes(A,F).
```

en este axioma de efecto se define que el fluent F se cumple en el siguiente tiempo T+1, si se ejecuta la acción A en un tiempo anterior T y esa misma acción también ocurre en el mismo tiempo T y además la ejecución de esa acción causa el fluent F.

El otro axioma de efecto es:

```
holds(F, T+1) :- literal(F), time(T), T < length, action(A),
                 executable(A,T), occurs(A,T),
                 causes_dd(F,A,F_inicial),
                 literal(F_inicial), holds(F_inicial,T).
```

en este axioma de efecto se indica algo muy similar que en el anterior, que el fluent F se cumple en el siguiente tiempo T+1, si se ejecuta la acción A en un tiempo anterior T y esa

misma acción también ocurre en el mismo tiempo T y con la diferencia del anterior de que el fluent F es causado por la ejecución de esa acción A si se cumple el fluent F_inicial en el tiempo actual T.

Reglas de inercia

Recordemos que la inercia se refiere a que los fluents permanecen sin cambios. Para definir las reglas de inercia de todos los fluentes lo hacemos con la siguiente sentencia:

```
holds(F, T+1) :- literal(F), literal(G), contrary(F,G),
                 time(T), T < length, holds(F,T),
                 not holds(G, T+1).
```

la cual indica que el fluent F se cumple en el siguiente tiempo T+1, si el literal F es el contrario del literal G, y F se cumple en un tiempo anterior T y por otra parte no hay evidencia de que G se cumple en el siguiente tiempo T+1, en otras palabras, el literal F se va a mantener en el siguiente tiempo T+1 si no hay evidencia de que otro literal G se cumpla.

No concurrencia

Para evitar que varias acciones se ejecuten o cumplan en un mismo tiempo, es necesario agregar reglas que no permitan que exista concurrencia de acciones, por lo que agregamos las siguientes sentencias:

```
possible(A,T):-action(A),time(T),executable(A,T),not goal(T).

occurs(A,T) :- action(A), time(T), possible(A,T),
               not not_occurs(A,T).

not_occurs(A,T) :- action(A), action(AA), time(T),
                   occurs(AA,T),  neq(A,AA).
```

en la primera sentencia se establece que la acción A es posible en el tiempo T, si se ha ejecutado esta acción en ese tiempo y además no se ha llegado a la meta. En la segunda sentencia se establece que la acción A ocurre en el tiempo T si A es posible en T y no hay evi-

dencia de que la acción A no ha ocurrido en T. Y la tercera sentencia indica que la acción A no ocurre en el tiempo T si otra acción diferente AA ocurre en ese mismo tiempo T.

Declaración de hide

Finalmente queda indicar cuáles de las sentencias que estamos manejando queremos que se muestren al momento de ejecutar los códigos, para esto, se hace uso de la palabra `hide` la cual evitará que se muestren muchas líneas en el resultado y sea más fácil ver lo que realmente necesitamos, esto se declara de la siguiente manera:

```
hide initially(F).
hide contrary(F,G).
hide fluent(F).
.
.
.
```

Solución para el problema de reparto de oficios en una oficina

Como modelo estable obtenido en esta primera prueba aplicado a una oficina se tiene lo siguiente:

```
Stable Model:
occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
```

lo que se observa en este resultado son los `occurs`, los cuales indican las acciones que ocurrieron y que el robot realizó para lograr entregar los oficios a la oficina oA, es decir, primero ocurrió que el robot tomara los oficios en el tiempo1, después se movió de la recepción a la oficina oA en el tiempo 2 y finalmente en el tiempo 3 entregó los oficios en la oficina oA, indicando que llegó a esa oficina en el tiempo 2.

Como se ha explicado, la meta para entregar los oficios a una oficina para esta primera versión se completó satisfactoriamente. Sin embargo, aún se deben realizar algunos cambios necesarios para que el robot pueda realizar estas entregas en más de una oficina.

3.2.2 Agregación de condiciones de ejecutabilidad para dos oficinas

Cada vez que se agregue una nueva oficina es necesario modificar el tiempo para llegar a la meta, la longitud de los pasos para realizar la acción, agregar el nuevo lugar junto con su horario de preferencia para recibir los oficios, así como cambiar los valores iniciales y finales dependiendo cual queremos que sea la nueva meta.

Ahora tomaremos el mismo código que se tenía para una oficina y lo modificaremos para la entrega de oficios en dos oficinas.

Declaración de constantes

Los nuevos tiempos serán `length=8` y `reloj=6` para el tiempo en que debe cumplirse la meta y los pasos en los que el robot debe realizar las acciones respectivamente, quedando de la siguiente manera:

```
const length=8. %Tiempo máximo para llegar a la meta
time(1..length).

const reloj=6. %Longitud de los pasos para realizar la acción
timeh(0..reloj).
```

Declaración de hechos

Ahora los nuevos hechos serán, la oficina C `(oC)` junto con sus respectivos tiempos de preferencia de la entrega de los oficios y el tiempo de viaje para trasladarse a la oficina y a la recepción, de tal forma que se tiene lo siguiente:

```
lugar(recepcion). %Lugar recepción donde el robot tomará los oficios.
```

```
lugar(oA).  %Lugar oficina A.
lugar(oC).  %Lugar oficina C.

horario_oficios(oA,2,3).  %Recibir oficios en la oficina A de 2 a 3
tiempo_viaje(recepcion,oA,1).  %Tiempo para trasladarse a la oficina A.
tiempo_viaje(oA,recepcion,1). %Tiempo para trasladarse a la recepción.

horario_oficios(oC,4,5). %Recibir oficios en la oficina C de 4 a 5.
tiempo_viaje(recepcion,oC,2). %Tiempo para trasladarse a la oficina C.
tiempo_viaje(oC,recepcion,2). %Tiempo para trasladarse a la recepción.
```

Declaración de fluentes y acciones

Los fluents y las acciones no serán modificados, ya que deben ser los mismos para cualquier número de oficinas.

Declaración de los estados iniciales y finales

En la parte del estado inicial se agregará la información de que aún no se han entregado los oficios en la nueva oficina `C` y para el estado final que ya se han entregado los oficios en ambas oficinas, esto queda de la siguiente manera:

```
initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).
initially(neg(entrego_oficios_en_lugar(oA))).
initially(neg(entrego_oficios_en_lugar(oC))).

finally(entrego_oficios_en_lugar(oA)).
finally(entrego_oficios_en_lugar(oC)).
```

Declaración de condiciones de ejecutabilidad

Para la parte de declaraciones de ejecutabilidad se tendrá que modificar y agregar otro executable para la acción de mover, de lo contrario el robot no se mueve a la segunda oficina cuando ya entregó los oficios en la primera, el código modificado será el siguiente:

```
executable(move(L1,L2),T) :- T < length,
               holds(esta_en_lugar(L1),T),
               holds(robot_tiene_oficios,T),
               holds(neg(entrego_oficios_en_lugar(L2)),T),
               time(T),lugar(L1), lugar(L2),neq(L1,L2).
```

este primer executable lo aplicaremos para el caso donde el robot se mueve de la recepción a la oficina, y el segundo executable se aplica cuando el robot tenga que moverse de alguna oficina a la recepción:

```
executable(move(L1,recepcion),T) :- T < length,
                    holds(esta_en_lugar(L1),T),
                    holds(entrego_oficios_en_lugar(L1),T),
                    holds(neg(robot_tiene_oficios),T),
                    time(T), lugar(L1), neq(L1,recepcion).
```

Declaración de las reglas de causa

Al igual que las acciones y los fluents, las reglas de causa no serán modificadas por el momento.

A partir de la declaración de la meta hasta la declaración de hide, el código será siempre el mismo, ya que ese código corresponde a la estructura de cada una de las reglas para que puedan cumplirse.

Solución para el problema de reparto de oficios en dos oficinas

Haciendo estos cambios, se obtienen ahora 3 modelos estables, los cuales son:

```
Answer: 1
Stable Model: occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2) occurs(entregar_oficios(oA,2),3)
occurs(move(oA, recepcion),4) occurs(tomar_oficios,5)
occurs(move(recepcion,oC),6)
```

en este primer modelo, se observa que primero ocurre que el robot toma los oficios en el tiempo 1, después en el tiempo 2 se mueve de la recepción a la oficina oA, en el tiempo 3

logra hacer entrega de los oficios a la oficina oA con tiempo de llegada 2, posteriormente en el tiempo 4 se mueve de regreso a la recepción, en el tiempo 5 el robot toma los oficios y finalmente en el tiempo 6 ocurre que el robot se mueve de la recepción a la oficina oC para realizar la entrega de los oficios, pero este último paso para entregar los oficios no ocurre, por lo que no es posible concluir el plan.

El segundo modelo estable que se obtuvo fue el siguiente:

```
Answer: 2
Stable Model: occurs(tomar_oficios,1)
occurs(move(recepcion,oC),2) occurs(move(oC,oA),3)
occurs(move(oA,oC),4) occurs(move(oC,oA),5)
occurs(move(oA,oC),6) occurs(move(oC,oA),7)
```

en este modelo, en el tiempo 1 el robot toma los oficios, en el tiempo 2 ocurre que el robot se mueve de la recepción a la oficina oC, pero a partir del tiempo 3 en adelante solo se mueve de una oficina a otra y ya no logra realizar las demás acciones correspondientes para la entrega de los oficios.

El tercer modelo estable que se obtuvo fue:

```
Answer: 3
Stable Model: occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2) occurs(move(oA,oC),3)
occurs(move(oC,oA),4) occurs(move(oA,oC),5)
occurs(move(oC,oA),6) occurs(move(oA,oC),7)
```

aquí sucede algo similar que en el modelo estable anterior, en donde a partir del tiempo 3 el robot solo se va moviendo de una oficina a otra y ya no logra realizar las demás acciones.

Como se puede observar, en este caso para la entrega de oficios en dos oficinas, no es posible realizar la entrega de los oficios satisfactoriamente, por lo que es necesario realizar nuevos cambios.

3.2.3 Cambio de reglas de causa

Como se ha mostrado anteriormente, el robot aún no realiza la entrega de los oficios para dos oficinas. Analizando el problema se pudo observar que se deben verificar los tiempos de llegada ya que no se está haciendo correctamente y esto interviene para que no sea posible entregar los oficios.

Agregando al código anterior lo siguiente nos ayudará a llevar el control del tiempo en que el robot sale de la recepción o de alguna oficina, porque no se estaba contemplando el caso en que hubiera un tiempo de llegada al momento en que se ejecuta la acción mover. Por lo tanto, la única regla que debe agregarse es:

```
causes_dd(tiempo_salida(T_salida),move(L1,L2),
tiempo_llegada(T_llegada)) :-  tiempo_viaje (L1,L2,Tv),
T_salida = Tv + T_llegada,   timeh(T_salida),
timeh(T_llegada),timeh(Tv),lugar(L1), lugar(L2), neq(L1,L2).
```

en esta regla de causa se está indicando el tiempo en que el robot sale de algún lugar `tiempo_salida(T_salida)` si se ejecuta la acción `move(L1,L2)` y se sabe además el tiempo en que había llegado al lugar del que va a salir `tiempo_llegada(T_llegada)` el cual ya se había calculado previamente. Para la obtención del tiempo de salida debemos sumar el tiempo de traslado al lugar destino o tiempo de viaje `Tv` y el tiempo en que había llegado `T_llegada`, teniendo con esto la siguiente asignación `T_salida = Tv + T_llegada`. De tal manera que así se irá actualizando el tiempo de salida de la misma forma en que lo hacemos para el tiempo de llegada.

Realizados los cambios indicados, se obtiene el siguiente modelo estable, en el cual ya es posible entregar los oficios en dos oficinas:

```
Answer: 1
Stable Model: occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2) occurs(entregar_oficios(oA,2),3)
occurs(move(oA,recepcion),4) occurs(tomar_oficios,5)
occurs(move(recepcion,oC),6) occurs(entregar_oficios(oC,5),7)
```

este modelo indica lo siguiente: primero el robot toma el café en el tiempo 1, después se mueve de la recepción a la oficina `oA` en el tiempo 2, en el tiempo 3 entrega los oficios en la oficina `oA` con tiempo de llegada 2, luego en el tiempo 4 regresa a la recepción, toma los otros oficios a entregar en el tiempo 5, después se mueve a la oficina `oC` en el tiempo 6 y por último en el tiempo 7 entrega los oficios en esta segunda oficina `oC` con tiempo de llegada 5.

Por el momento solo se ha probado esta codificación para la entrega de oficios en 2 oficinas pero aún es necesario realizar cambios en los fluents.

3.2.4 Eliminación y agregación de nuevos fluentes

Hasta el momento vemos que este programa cumple con las entregas correspondientes dentro del periodo solicitado, sin embargo, aún se tienen que verificar los tiempos de salida cuando el robot se mueve de alguna oficina a la recepción y también los tiempos de llegada al momento de tomar los oficios, porque cuando el robot se traslada de una oficina a la recepción se está indicando el tiempo de llegada a la oficina en donde se encuentra, en lugar de indicar el tiempo en que sale el robot de esa oficina y por otro lado, cuando el robot toma los oficios se está indicando el tiempo de salida y eso es incorrecto también puesto que debería de indicarse el tiempo en que el robot llega a ese lugar.

Para corregir estos errores de los tiempos de salida y llegada es necesario que se eliminen los fluents anteriores del tiempo de llegada y tiempo de salida y agregar otros fluents indicando por separado cuales son los tiempos de salida y llegada de la oficina y de la recepción.

El problema era que se estaba manejando la misma variable de ambos tiempos para referirnos a las oficinas y a la recepción, lo que originaba el error, es por eso que ahora tienen que ser cuatro fluents, dos para la recepción y dos para referirse a las oficinas. De tal manera que es necesario cambiar la forma de llevar el control de los tiempos y sustituir los fluents `tiempo_salida(T)` y `tiempo_llegada(T)` por los siguientes fluents:

- `t_llegada_aoficina(T)` y `t_llegada_arecepcion(T)` para indicar el tiempo en que el robot llega a la oficina y a la recepción respectivamente siempre y cuando se cumplan sus restricciones correspondientes.
- `t_salida_doficina(T)` y `t_salida_drecepcion(T)` quienes marcan los tiempos en que el robot sale de alguna oficina o de la recepción.

Entonces, estos nuevos fluents permitirán controlar mejor los tiempos de cada lugar.

Declaración de nuevos fluentes

Con la explicación anterior se tiene lo siguiente en el código:

```
fluent(t_llegada_aoficina(T)) :- timeh(T).
fluent(t_llegada_arecepcion(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion (T)) :- timeh(T).
```

Remplazo de fluentes en el estado inicial

Ahora como se ha definido el cambio de los fluents es necesario cambiar también el resto de las reglas en donde aparezcan estos. En las condiciones iniciales reemplazamos `initially(tiempo_salida(1))` por:

```
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
```

esto indica que ahora el tiempo en que el robot sale de la recepción es 1 ya que se encuentra en ella y por lo tanto el tiempo de salida de la oficina es 0 porque no se encuentra en una oficina sino en la recepción, este fluent se verá afectado cuando el robot haya entregada los oficios y salga de la oficina para dirigirse a la recepción y tomar otros oficios.

Remplazo de fluentes en las condiciones de ejecutabilidad

Por lo tanto, el código para las sentencias de ejecutabilidad solo se aplica cuando el robot entrega los oficios, debido a que en las otras dos reglas de ejecutabilidad no aparecen los fluents modificados, entonces el código quedará de la siguiente manera:

```
executable(entregar_oficios(L,T_entrega),T):-
                T < length, time(T), lugar(L),
                timeh(T_entrega),
                holds(esta_en_lugar(L),T),
                holds(robot_tiene_oficios,T),
                holds(neg(entrego_oficios_en_lugar(L)),T),
                holds(t_llegada_aoficina(T_llegada_O),T),
                horario_oficio(L,T0,T1),
                T0<= T_llegada_O, T_llegada_O <= T1,
                assign(T_entrega,T_llegada_O),
                timeh(T_llegada_O), timeh(T0), timeh(T1).
```

Remplazo de fluentes para las reglas de causa

La verificación de los tiempos para las reglas de causa ahora será con los nuevos fluents que se han definido anteriormente, los cuales son `t_llegada_aoficina(T)`, `t_llegada_arecepcion(T)`, `t_salida_doficina(T)` y `t_salida_drecepcion(T)`, entonces se cambiarán las reglas de efecto donde aparecen los fluents `tiempo_salida(T)` y `tiempo_llegada(T)` con lo que se obtendrá lo siguiente para conocer el tiempo en que el robot sale de la recepción:

```
causes_dd(t_salida_dmaquina(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m),
timeh(T_llegada_O), timeh(Tv), lugar(L1), lugar(L2),
neq(L1,L2).

causes_dd(neg(t_salida_dmaquina(T_salida_m)), move(L1,L2),
t_salida_dmaquina(T_salida_m)) :- timeh(T_salida_m),
lugar(L1), lugar(L2), neq(L1,L2).
```

Y estas dos reglas siguientes son para obtener el tiempo de llegada a las oficinas:

```
causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_dmaquina(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), ti-
meh(T_llegada_O), timeh(Tv), lugar(L1), lugar(L2),
neq(L1,L2).

causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- timeh(T_llegada_O),
lugar(L1), lugar(L2), neq(L1,L2).
```

El resto del código será el mismo. Realizando estos cambios con respecto a los fluents, se tendrá ahora un mejor control de las variables y los tiempos correspondientes, además de que permitirá saber a cual lugar hace referencia la llegada y salida del robot. Este código completo se puede observar en el *Apéndice* indicado en la página 75.

3.2.5 Agregación de reglas de causa

Por ahora ya se realizó el cambio correspondiente de las variables y todo va saliendo bien, pero aún se muestran algunos errores en los tiempos de salida que se corregirán en el código que se tiene agregando las siguientes reglas de causa.

Estas 3 reglas de causa son las que hacían falta para dar por terminado el programa:

```
causes_dd(t_salida_doficina(T_salida_O),
entregar_oficios(L,T_entrega),
t_llegada_aoficina(T_llegada_O)) :- T_salida_O = T_llegada_O,
lugar(L),timeh(T_entrega), timeh(T_salida_O),
timeh(T_llegada_O).
```

en esta primera regla de causa se obtiene el tiempo en que el robot sale de la oficina al momento en que se ejecuta la acción `entregar_oficios` y conociendo también el tiempo en que el robot llegó a esa oficina indicado por el fluent `t_llegada_aoficina(T_llegada_O)`, si el tiempo de llegada a la oficina

`T_llegada_O` se asigna al tiempo de salida de esa misma oficina `T_salida_O` para que de esta manera se lleve un control se vayan actualizando los tiempos en que el robot sale de las oficinas en las que se encuentra. La segunda regla de causa está definida de la siguiente manera:

```
causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficios(L,T_entrega), t_salida_doficina(T_salida_O))
:- timeh(T_salida_O), timeh(T_entrega), lugar(L).
```

esta regla está negando el tiempo en que el robot ha salido de la oficina después de que se ha ejecutado la acción entregar oficios y ya se conocía este tiempo, esto se establece porque ahora se necesita saber el tiempo en que el robot llegó a la oficina para entregar los oficios, más no el tiempo en el que salió de la oficina anterior. Para la tercera regla de causa tenemos un caso similar a la regla anterior, la cual es:

```
causes_dd(neg(t_salida_doficina(T_salida_O)), move(L1,L2),
t_salida_doficina(T_salida_O)) :- timeh(T_salida_O),
lugar(L1), lugar(L2), neq(L1,L2).
```

ahora en esta regla también se está negando el tiempo en que el robot sale de la oficina con la diferencia de que se ejecuta la acción `move(L1,L2).`

Con la agregación de estas tres reglas de causa, el problema de secuenciación mínima con tiempos de realización para el caso particular de reparto de oficios queda terminado y se puede verificar la entrega correcta de los oficios de 1 a 5 oficinas, que fueron el número de oficinas con las que se trabajó.

En el *Apéndice* es posible observar el código completo correspondiente a esta versión final y una tabla en donde se indican los tiempos en que Smodels obtiene los Answer Sets con 5 oficinas.

Capítulo 4

Codificación en Clasp

En este capítulo se dará una explicación de la sintaxis básica correspondiente a la codificación en Clasp y se mostrará la implementación para el modelado del problema de reparto de oficios utilizando esta herramienta, problema que se ha resuelto en el capítulo 3 utilizando Smodels.

4.1 Sintaxis Básica

La sintaxis en Clasp es muy parecida a Smodels. Describiremos parte de la sintaxis básica que se ha utilizado para el modelado del problema de secuenciación mínima.

Declaración de constantes

Las constantes que aparecen en un programa lógico son valores que el usuario define. La declaración de las constantes se hace a través de la declarativa `#const`, con la que se puede definir un valor predeterminado para una constante. Sintácticamente, `#const` debe estar seguido por una asignación que tenga una constante en el lado izquierdo y un término en el lado derecho [5]. Por ejemplo:

```
#const length = 8.
#const reloj = 6.
```

en donde `length` y `reloj` son las constantes, `8` y `6` son los términos, o bien, también pueden ser de la forma:

```
#const x = 42.
#const y = f(x,h).
```

en donde, por razones de eficiencia, la declaración de constates son en orden dependiente, es decir, la constante `x` se sustituye por 42 en el término de la constante `y` porque el valor

de esta constante depende del valor de la constante `x`, pero cuando se invierte el orden de la declaración de estas constantes ya no es válido.

Declaración de hechos

Los hechos pueden resumirse o abreviarse, es decir, en lugar de escribir los hechos por separado utilizando una línea de código por cada hecho, se pueden escribir todos juntos en una sola línea. Por ejemplo:

```
#const length = 4.
time(1..length).
```

donde `time(1..length)` es la abreviación de los hechos siguientes:

```
time(1) time(2) time(3) time(4)
```

También es posible indicar en una sola línea una variedad de hechos numéricos, alfabéticos, alfanuméricos (pero que empiecen con letra o de lo contrario provocara un error, por ejemplo: 45fg, no es válido) en un solo paréntesis que tengan el mismo nombre de cabeza. No permite cadenas de caracteres especiales como $, &, @…etc. Por ejemplo, esta abreviación de hechos:

```
time(a;1;cadena;1500;r;r45)
```

dará como resultado lo siguiente:

```
time(a) time(r45) time(r) time(1500) time(cadena) time(1)
```

Las declaraciones de hechos que tienen una secuencia o serie también es posible abreviarlas. Por ejemplo, en lugar de indicar uno por uno los hechos del 1 al 10, se pueden indicar los 10 hechos en una sola línea, abreviando esta secuencia de la siguiente manera: `numero(1..10)`. Sin embargo, aunque de “a” a “z” parece ser también una serie aceptada por lenguajes de programación de alto nivel, potassco no la reconoce como tal, solo es posible abreviar caracteres mediante el uso del operador ‘;’ es decir, declararlo de esta forma `lugar(a;b;c)` con lo que obtenemos:

```
lugar(a) lugar(b) lugar(c)
```

Una sucesión no necesariamente inicia desde 1 o 0 puede ser cualquier intervalo como `timeh(150..1504)`. La abreviación de hechos por así decirlo se puede utilizar dentro de hechos junto con otros parámetros para indicar la combinación entre ellos. Un ejemplo sería la combinación de hechos para obtener la serie del uno al cuatro con "a" pero en una sola línea, entonces esto se puede definir de la siguiente manera `entrega(1..4,a)`. De tal forma que se obtienen los siguientes resultados:

```
entrega(1,a) entrega(2,a) entrega(3,a) entrega(4,a)
```

Negación débil y fuerte

En los programas lógicos existen dos tipos de negación: la negación fuerte y la negación débil. El conectivo `not` expresa la negación por defecto o débil, es decir, un literal de la forma `not A` se mantiene a menos que `A` se derive. En cambio la negación fuerte de alguna proposición se mantiene si el complemento de la proposición se deriva. La negación clásica o fuerte se denota por el símbolo "-" y siempre va frente a los átomos. Es decir, si `A` es un átomo, entonces `-A` es el complemento de `A`. Semánticamente, `-A` es simplemente un nuevo átomo, con la condición adicional de que `A` y `-A` no deben contenerse en forma conjunta [5], por ejemplo, en el siguiente programa:

```
1 bird(tux).     penguin(tux).
2 bird(tweety).  chicken(tweety).

3 flies(X) :- bird(X), not -flies(X).
4 -flies(X) :- bird(X), not  flies(X).
```

en la tercera línea se utilizan los dos tipos de negación, en donde, `flies(X)` quiere decir que `X` vuela (el símbolo ':-' indica una implicación y se lee 'si se cumple') que `X` es un ave `bird(X)`, y no hay evidencia de que `X` no vuela `not -flies(X)`, donde `X` es una variable, la cual siempre se denota con letras mayúsculas. En la cuarta línea `-flies(X)` significa que `X` no vuela, si se cumple que `X` es un ave `bird(X)` y no hay evidencia de que `X` vuele `not flies(X)`. Por lo tanto el resultado es:

```
bird(tux) penguin(tux) bird(tweety) chicken(tweety)
flies(tweety) flies(tux)
```

Restricciones de cardinalidad

Una regla es una abreviación de un conjunto de intuiciones correspondientes a un problema real. Otro tipo de regla es la restricción de cardinalidad que indica las combinaciones posibles que se pueden hacer de un conjunto de valores. La sintaxis de este tipo de reglas es:

```
límite inferior{constante/variable1, constante/variable2,
constante_variable_n }límite superior.
```

Siempre el límite izquierdo debe ser menor que el derecho. Por ejemplo en el siguiente código:

```
1 #const length = 4.
2 time(1..length).
3 lugar(recepcion;oA).
4 action(move(L1,L2)):- lugar(L1), lugar(L2).

5 1{executable(move(L1,L2),T) : T < length: time(T) :
  lugar(L1): lugar(L2)} 1.
```

observemos la quinta línea que tiene la regla entre llaves `1{executable(move(L1,L2),T) : T < length: time(T) : lugar(L1): lugar(L2)} 1`. A esta regla se le llama restricción de cardinalidad, la cual tiene dos condiciones que son `move(L1,L2)`y `T`, donde la primera condición nos indica que se mueva de `L1` a `L2` y la segunda indica que sea en el punto de tiempo `T` pero si `T` es menor a la constante `length` y `T` es una variable de tipo `time` y también si `L1` y `L2` son lugares. Las restricciones son verdaderas si y solo si el número de literales verdaderos que están dentro del conjunto se encuentran entre el límite superior e inferior [5].

Como se observa en el ejemplo anterior, `1{.....}1` indica que sólo se puede mover una vez del lugar `L1` al lugar `L2`, ya que esto es lo que pide el problema. La ausencia del

número a la derecha del símbolo "}" no afecta ya que por default lo toma como un uno (1). Ahora bien, este no puede ser cero, la cardinalidad debe ser indicada con dos números, uno a la izquierda y otro a la derecha de las llaves y siempre el número a la izquierda debe ser menor que el de la derecha, de lo contrario se producirá un error lógico indicando que el programa es insatisfactible.

También se debe saber que al poner cardinalidad cero – cero no produce ningún error pero tampoco tiene sentido ya que no proporciona ningún resultado. Los símbolos "{" y "}" no pueden faltar porque estos denotan un conjunto resultante denominado dominio de predicados y al omitirlos produciría un error en la ejecución.

4.2 Implementación del problema de reparto de oficios para una oficina de Smodels a Clasp

En esta sección se describen algunas de las variantes que se presentaron durante el desarrollo del código para pasarlo de Smodels a Clasp. Para este traslado de código fue necesaria la realización de varias pruebas con diferentes modificaciones para obtener los resultados que finalmente se obtuvieron. Algunos de estos códigos pueden verse en el *Apéndice*.

Se describen a continuación cada una de las reglas en ambos modelos para la primera versión que ya se tenía en Smodels donde el robot solo entregaba los oficios a una oficina.

Diferencia de la declaración de constantes

En Smodels definimos la longitud del plan a través del uso de la palabra reservada `const` de la siguiente manera:

```
const length = 4.
```

A diferencia de Clasp, solo tenemos que agregar el operador "#" antes de la palabra `const` para que se puedan tomar como constantes, de lo contrario se marcará un error.

```
#const length = 4.
```

Diferencia de la declaración de hechos

Para la declaración de los hechos, en Smodels lo hacemos por separado, es decir, cada hecho en una línea aparte:

```
lugar(recepcion).
lugar(oA).
```

En Clasp podemos unirlos o abreviarlos de tal forma que solo sea una línea de código haciendo uso del operador "***;***" para marcar la separación de cada hecho como se muestra:

```
lugar(recepcion;oA).
```

Declaración de fluentes

La declaración de las sentencias para los fluents es de la forma:

```
fluent(f).
fluent(f):- b1,…,bn.
```

que es la misma para Smodels y Clasp:

```
fluent(robot_tiene_oficios).
fluent(esta_en_lugar(L)):- lugar(L).
```

Declaración de las acciones

Para las acciones se tiene que son de la forma:

```
action(a).
action(a) :- b1,…,bn.
```

en ambos modelos también se declaran de la misma manera:

```
action(tomar_oficios).
action(move(L1,L2)):- lugar(L1), lugar(L2).
```

Declaración del estado inicial

El estado inicial tiene la siguiente forma:

```
initially(g1).  … initially(gn).
initially(neg(h1)).  … initially(neg(hr)).
```

Por lo que en los dos modelos declaramos el estado inicial de la misma manera:

```
initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).
initially(neg(entrego_oficios_en_lugar(oA))).
```

Declaración del estado final

De la misma manera que el estado inicial, la declaración del estado final tiene la misma estructura:

```
finally(g1).  … finally(gn).
finally(neg(h1)).  … finally(neg(hr)).
```

Por lo tanto, la declaración del estado final es el mismo en Smodels y Clasp:

```
finally(entrego_oficios_en_lugar(oA)).
finally(esta_en_lugar(oA)).
```

Declaración de las reglas de ejecución

La declaración para las reglas de ejecución de acciones se hace de la siguiente manera:

```
executable(a,T) :- T < length, time(T), holds(f1,T),..,
                   holds(fn,T).
```

Para Smodels y Clasp los declaramos de esta forma:

```
executable(move(L1,L2),T) :- T < length, time(T),
holds(esta_en_lugar(L1),T), lugar(L1), lugar(L2), neq(L1,L2).
```

Con la única diferencia, de que en Clasp indicamos la diferencia de valores con el símbolo "!=":

```
executable(move(L1,L2),T) :- T < length, time(T),
holds(esta_en_lugar(L1),T), lugar(L1), lugar(L2), L1 != L2.
```

donde `T<length` nos indica que `T` debe ser menor a la longitud permitida por el plan, en este caso `length`.

Declaración de las reglas de causa

Las reglas de causa se declaran de dos formas. La primera es:

```
causes_dd(f, a, f_inicial) :- p1,..,pn, neg(q1),..,neg(qr).
```

donde `f` es un fluent que se cumplirá cuando la acción `a` se ejecute y `f_inicial` es también un fluent que debe cumplirse en el mismo momento en que se ejecuta la acción. En Smodels y Clasp declaramos esta regla de causa de la siguiente manera:

```
causes_dd(neg(esta_en_lugar(L1)), move(L1,L2),
esta_en_lugar(L1)) :- lugar(L1), lugar(L2), L1 != L2.
```

La segunda forma es:

```
causes(a, f) :- p1,..,pn, neg(q1),..,neg(qr).
```

donde al ejecutar la acción `a` se causará el fluent `f`. Esto lo declaramos en Smodels y Clasp de la forma siguiente:

```
causes(tomar_oficios, robot_tiene_oficios).

causes(entregar_oficios(L,T_entrega),
neg(robot_tiene_oficios)) :- timeh(T_entrega), lugar(L).
```

Diferencia del uso de hide

Para Smodels ocultamos las sentencias con el uso de la palabra reservada `hide` de la siguiente manera:

```
hide initially(F).
hide contrary(F,G).
hide fluent(F).
```

Para Clasp solo basta con agregar el símbolo "#" delante de `hide`:

```
#hide initially(F).
#hide contrary(F,G).
#hide fluent(F).
```

Solución en Clasp

Con estos cambios realizados, el código en Clasp para la entrega de oficios en una oficina es el siguiente:

```
#const length = 4.
time(1..length). % tiempo maximo para llegar a la meta

#const reloj = 4.
timeh(0..reloj). % Longitud de los pasos para realizar la accion

lugar(recepcion;oA).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

fluent(robot_tiene_oficios).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(tiempo_llegada(T)) :- timeh(T).
fluent(tiempo_salida(T)) :- timeh(T).
fluent(entrego_oficios_en_lugar(L)) :- lugar(L).

action(tomar_oficios).
action(move(L1,L2)):- lugar(L1), lugar(L2).
action(entregar_oficios(L,T_entrega)) :-timeh(T_entrega),lugar(L).

initially(neg(entrego_oficios_en_lugar(oA))).
initially(esta_en_lugar(recepcion)).
```

```
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).

finally(entrego_oficios_en_lugar(oA)).
finally(esta_en_lugar(oA)).

executable(tomar_oficios,T) :- T < length, time(T),
                               holds(esta_en_lugar(recepcion),T),
                               holds(neg(robot_tiene_oficios),T).

executable(move(L1,L2),T) :- T < length, time(T),
                             holds(esta_en_lugar(L1),T),
                             lugar(L1), lugar(L2), L1 != L2.

executable(entregar_oficios(L,T_entrega),T):-
                T < length, time(T), lugar(L), timeh(T_entrega),
                holds(esta_en_lugar(L),T),
                holds(robot_tiene_oficios,T),
                holds(neg(entrego_oficios_en_lugar(L)),T),
                holds(tiempo_llegada(T_llegada),T),
                horario_oficios(L,T0,T1),
                T0<= T_llegada, T_llegada <= T1,
                T_entrega == T_llegada,
                timeh(T_llegada), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1))
:- lugar(L1), lugar(L2), L1 != L2.

causes_dd(tiempo_llegada(T_llegada), move(L1,L2),
tiempo_salida(T_salida)) :- tiempo_viaje(L1,L2,Tv),
T_llegada = Tv + T_salida, timeh(T_salida), timeh(T_llegada),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2),
tiempo_llegada(T_llegada)) :- timeh(T_llegada), lugar(L1),
lugar(L2), L1 != L2.

causes_dd(neg(tiempo_salida(T_salida)), move(L1,L2),
tiempo_salida(T_salida)) :- timeh(T_salida), lugar(L1), lugar(L2),
L1 != L2.

causes(tomar_oficios, robot_tiene_oficios).

causes(entregar_oficios(L,T_entrega), neg(robot_tiene_oficios)) :-
timeh(T_entrega), lugar(L).
```

```
causes(entregar_oficios(L,T_entrega), entrego_oficios_en_lugar(L))
:- timeh(T_entrega), lugar(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

not_goal(T):- time(T),literal(X), finally(X), not holds(X,T).
goal(T) :- time(T),not not_goal(T).
exists_plan :- goal(T).
:- not exists_plan.

literal(G)  :- fluent(G).
literal(neg(G)) :- fluent(G).
contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).

holds(F, 1) :- literal(F), initially(F).

holds(F, T+1)   :-
    literal(F), time(T), T < length,
    action(A),executable(A,T),
    occurs(A,T),causes(A,F).

holds(F, T+1)   :- literal(F),
    time(T), T < length,
    action(A), executable(A,T),
    occurs(A,T),causes_dd(F,A,F_inicial),
    literal(F_inicial), holds(F_inicial,T).

holds(F, T+1)   :-
    literal(F), literal(G), contrary(F,G),
    time(T), T < length, holds(F,T),
    not holds(G, T+1).

possible(A,T) :- action(A), time(T), executable(A,T),
                 not goal(T).

occurs(A,T) :- action(A), time(T), possible(A,T),
               not not_occurs(A,T).

not_occurs(A,T) :- action(A), action(AA), time(T),  occurs(AA,T),
                   A != AA.

#hide time(T).
#hide timeh(T).
#hide lugar(L).
#hide horario_oficios(P,T1,T2).
#hide tiempo_viaje(L1, L2,D).
#hide action(A).
```

```
#hide causes(A,F).
#hide causes_dd(F,A,F_inicial).
#hide initially(F).
#hide contrary(F,G).
#hide fluent(F).
#hide literal(L).
#hide executable(A,T).
#hide holds(F,T).
#hide not_occurs(A,T).
#hide possible(A,T).
#hide exists_plan.
#hide finally(X).
#hide goal(T).
#hide not_goal(T).
```

Al ejecutar el código mostrado en Clasp se obtiene el siguiente Answer Set:

```
Answer: 1
occurs(tomar_oficios,1) occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
```

en donde los pasos que el robot ejecuta son: tomar los oficios, moverse a la oficina donde los entregará y entregar los oficios.

De esta forma podemos ver que la meta ha sido alcanzada con éxito de la misma manera que en la versión de Smodels. Ahora modificaremos un poco más este código, utilizando una herramienta de Clasp. La codificación de las demás pruebas en Clasp se pueden observar en el *Apéndice* o también en el siguiente sitio, donde se presentan todas las pruebas realizadas junto con sus impresiones de pantalla correspondientes: *https://sites.google.com/site/codigoentregadeoficios/*

4.3 Codificación para el reparto de oficios en una oficina con restricciones de cardinalidad.

En esta sección se hace uso de una herramienta de Clasp llamada restricción de cardinalidad, que ya se ha descrito anteriormente en la *sección 3.7*. Para esto, únicamente se realizarán cambios en las condiciones de ejecutabilidad. Lo cual se muestra de la siguiente manera:

La condición de ejecutabilidad de la acción tomar oficios es:

```
executable(tomar_oficios,T) :- T < length, time(T),
holds(esta_en_lugar(recepcion),T),
holds(neg(robot_tiene_oficios),T).
```

y al hacer uso de la restricción de cardinalidad, la condición de ejecutabilidad quedará de la siguiente manera:

```
1{executable(tomar_oficios,T) : T < length: time(T) }1.

executable(tomar_oficios,T) :-
                    holds(esta_en_lugar(recepcion),T),
                    holds(neg(robot_tiene_oficios),T).
```

La condición de ejecutabilidad de la acción mover es:

```
executable(move(L1,L2),T) :- T < length, time(T),
                    holds(esta_en_lugar(L1),T),
                    lugar(L1), lugar(L2), L1 != L2.
```

aplicando la restricción de cardinalidad a esta condición de ejecutabilidad, se tiene lo siguiente:

```
1{executable(move(L1,L2),T) : T < length: time(T) :
lugar(L1): lugar(L2)} 1.
```

Finalmente, para la última condición de ejecutabilidad para la acción entregar oficios, se tiene:

```
executable(entregar_oficios(L,T_entrega),T):-
                T < length, time(T), lugar(L), ti-
                meh(T_entrega),
                holds(esta_en_lugar(L),T),
                holds(robot_tiene_oficios,T),
                holds(neg(entrego_oficios_en_lugar(L)),T),
                holds(tiempo_llegada(T_llegada),T),
                horario_oficios(L,T0,T1),
                T0<= T_llegada, T_llegada <= T1,
```

```
                T_entrega == T_llegada,
                timeh(T_llegada), timeh(T0), timeh(T1).
```

al hacer uso de la restricción de cardinalidad, esta condición de ejecutabilidad para la acción entregar oficios queda de la siguiente manera:

```
1{executable(entregar_oficios(L,T_entrega),T) : T < length:
time(T): lugar(L): timeh(T_entrega;T_llegada;T0;T1)}1.

executable(entregar_oficios(L,T_entrega),T):-
                timeh(T_entrega),
                holds(esta_en_lugar(L),T),
                holds(robot_tiene_oficios,T),
                holds(neg(entrego_oficios_en_lugar(L)),T),
                holds(tiempo_llegada(T_llegada),T),
                horario_oficios(L,T0,T1),
                T0<= T_llegada, T_llegada <= T1,
                T_entrega == T_llegada.
```

Solución en Clasp con restricciones de cardinalidad

El resultado que se obtiene utilizando las restricciones de cardinalidad que se han definido es el mismo que el caso anterior en donde no se utilizaban las restricciones de cardinalidad, pero es una forma de codificar que más adelante nos ayudará a reducir código. De esta forma, el Answer Set obtenido es el siguiente:

```
Answer: 1
occurs(tomar_oficios,1) occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
```

Capítulo 5

Declaraciones de optimización para indicar preferencias

Si se quisiera modelar algún tipo de preferencia para el problema de reparto de oficios se puede hacer uso de las declaraciones de optimización que ofrecen las herramientas de Potassco[5]. En este capítulo se mostrará la implementación de preferencias mediante estas declaraciones de optimización usando la herramienta Clasp. La idea es asignar pesos a los literales para preferir cuál de ellos tiene el valor máximo o mínimo.

5.1 Uso de declaraciones de optimización en Clasp

Las declaraciones de optimización se encargan de verificar si un Answer Set es óptimo. Un Answer Set es óptimo si la suma de los literales es el máximo o el mínimo entre todos los Answer Set del programa lógico, de acuerdo a las declaraciones de optimización.

Por ejemplo, supongamos que una persona debe ir a tres lugares y tiene que elegir a cuál de ellos irá primero. Se sabe que el lugar s_1 está a 5 calles, el lugar s_2 está a 2 calles y el lugar s_3 está a 4 calles del lugar donde se encuentra actualmente. El tiempo que esta persona tarda en trasladarse desde el punto en que se encuentra a los lugares s_1, s_2 y s_3 son 3, 1 y 2 minutos respectivamente. Esta representación se especifica en las líneas 1-7 en el programa que se muestra más adelante. Entonces si se desea saber:

a) ¿Cuál es el lugar que le queda más lejos a esta persona desde el lugar en que se encuentra?, diríamos que es el lugar s_1 porque está a 5 calles del punto en que se encuentra y porque el lugar s_1 se ubica a 3 calles más del lugar s_2 y a una calle más del lugar s_3.

[5] http://potassco.sourceforge.net/

b) ¿A cuál de los lugares tarda menos tiempo en llegar esta persona desde el lugar donde se encuentra actualmente?, sabremos que es al lugar s_2, porque el tiempo de traslado a este lugar desde donde se encuentra actualmente al lugar s_2 es de un minuto a diferencia del lugar s_1 que tarda en llegar 3 minutos y el lugar s_3 que tarda 2 minutos.

En particular las herramientas de Answer Set gringo y clingo que forman parte de Potassco, adoptan las declaraciones de optimización de lparse [27], las cuales se indican mediante las palabras reservadas `#maximize` y `#minimize` que permiten encontrar answer sets máximos o mínimos de un programa lógico. Por ejemplo, si tuviéramos 3 answer sets y suponiendo que el primer answer set tiene el valor 3, el segundo answer set tiene el valor 5 y el tercer answer set tiene el valor 1, entonces, si aplicamos `#maximize` el answer set máximo será el segundo answer set porque tiene el valor 5 que es el más grande de los tres answer set. Si utilizamos `#minimize` el answer set mínimo será el tercer answer set porque tiene el valor 1 y es el valor más pequeño de los tres answer sets. Cabe aclarar que los valores de estos tres answer sets pueden obtenerse mediante diversas reglas que se apliquen a los literales del programa.

La sintaxis de estas declaraciones de optimización `#maximize` y `#minimize` se establece de la siguiente manera [27]:

```
#maximize [L1 = w1 @ p1,…, Ln = wn @ pn].
#minimize [L1 = w1 @ p1,…, Ln = wn @ pn].
```

donde L_i son literales, w_i es el peso que se asigna a cada literal y $@\ p_i$ es la prioridad asignada a cada literal.

Los pesos y las prioridades asignados a cada literal son de tipo entero mayores que 0, y la literal que tenga mayor valor, tendrá mayor prioridad.

Por ejemplo, si se tienen 3 salones a, b y c, con los siguientes lugares de sillas disponibles y ocupadas:

Salón	Lugares ocupados	Lugares vacios
a	2	4
b	3	5
c	5	2

Teniendo en la literal *lugares_o* los lugares que ya están ocupados y en la literal *lugares_v* los lugares aún disponibles. Y si a cada literal se le asigna cierta prioridad, suponiendo que a la primera literal *lugares_o* se le asigna prioridad 3 y a la segunda literal *lugares_v* se le asigna prioridad 2.

Entonces teniendo estos datos, si en una declaración de optimización se quisiera saber ¿cuáles salones tienen mayor número de lugares ocupados? y ¿cuáles salones tienen mayor número de lugares vacios? El answer set resultante sería el valor del literal con mayor prioridad, en este caso el literal *lugares_o* porque tiene una prioridad 3 la cual es mayor que la prioridad 2 del literal *lugares_v*, indicando que es el salón *c* quien tiene 5 lugares ocupados, además suma el resto de lugares ocupados del salón *a* y el salón *b* que resultan ser 5 lugares más ocupados y como el salón *c* es resultado del literal con mayor prioridad, entonces, para el aswer set del literal *lugares_v* se tiene únicamente la suma de los lugares vacios de los mismos salones que son el salón *a* con 4 lugares vacios y el salón *b* con 5 lugares vacios lo que resulta que son 9 lugares vacios. Esto se puede expresar usando la siguiente declaración de optimización:

```
#maximize [salon(X): lugares_o(X,Y) = Y @ 3,
           salon(X): lugares_v(X,Y) = Y @ 2].
```

donde `#maximize` indica que se va a obtener los valores máximos de los literales, `salon(X)` es alguno de los 3 salones, `lugares_o(X,Y)` y `lugares_v(X,Y)` indican los lugares ocupados y vacios respectivamente, `= Y` indica el valor que tenga esa literal porque es un valor entero, `@ 3` y `@ 2` es la prioridad asignada a cada literal.

Para este caso en que los literales tienen diferente prioridad, los answer set resultantes se van a basar en los literales con mayor prioridad y en base a estos valores será el resul-

tado de los demás literales. También se debe especificar que es lo que se quiere como resultado, en este caso se quiere que muestren los salones, por tal razón se define `salon(X)`.

Por otra parte, si estos literales *lugares_o* y *lugares_v* tuvieran la misma prioridad, es decir, que *lugares_o* tenga prioridad 2 y *lugares_v* también tenga prioridad 2 dentro de la misma declaración de optimización, entonces ahora el resultado del answer set para esta declaración de optimización será el valor máximo de la literal *lugares_o* que sigue siendo el salón *c* con 5 lugares ocupados, y la suma de los lugares ocupados y los lugares vacios de los salones *a* y *b*, que son: 2 lugares ocupados y 4 lugares vacios del salón *a*, más 3 lugares ocupados y 5 lugares vacios del salón *b*, lo que resulta que son 14 lugares. Esto se puede representar con la siguiente declaración de optimización:

```
#maximize [salon(X): lugares_o(X,Y) = Y @ 2,
           salon(X): lugares_v(X,Y) = Y @ 2].
```

En este caso, donde los literales tienen la misma prioridad, los anwer sets resultantes se basan en el valor máximo del primer literal y la suma del resto de los valores de los demás literales.

Retomando el ejemplo anterior donde una persona debe elegir a cuál lugar debe ir primero dependiendo del número de calles en que se encuentre el lugar con respecto al lugar en donde se encuentra actualmente esta persona y haciendo uso de estas declaraciones de optimización de Potassco para contestar las preguntas que se habían planteado, se tiene lo siguiente:

- Para la pregunta a) que dice: ¿Cuál es el lugar que le queda más lejos a esta persona desde el lugar en que se encuentra? La representamos con la siguiente regla:

  ```
  #maximize [lugar(X): distancia(W,X,Y) = Y @ 1].  (a)
  ```

  esta declaración de optimización con `#maximize` y corchetes, obtiene los literales con valores más grandes y suma los valores de los literales restantes. Esta declaración consiste en mostrar el lugar `X`, indicado con `lugar(X)`, tal que la distancia `Y`

desde `W` hasta `X`, (indicada mediante `distancia(W,X,Y) = Y`, donde `= Y` significa que se va a tomar el valor `Y` que corresponde a la distancia desde el punto `W` en que se encuentra hasta el lugar `X`, porque es un valor de tipo entero), sea la distancia máxima con prioridad 1 indicado mediante (`@ 1`). Esta prioridad se ha definido con valor 1 porque es el único literal y declaración de optimización que se ha definido, sin embargo, si se quisiera agregar otro literal dentro de esta misma declaración de optimización o definir una nueva declaración de optimización se puede cambiar el valor de la prioridad dependiendo cual literal se quiere que tenga mayor prioridad. Por lo tanto, con esto se obtiene que el lugar más lejos es s_1 porque se había dicho que se encuentra a 5 calles de distancia del punto inicial y otro de los lugares que le queda más lejos también es el lugar s_3 porque se encuentra a 4 calles de distancia del punto inicial.

- Para la pregunta b) que dice ¿A cuál de los lugares tarda menos tiempo en llegar esta persona desde el lugar donde se encuentra actualmente?, usaremos `#minimize` y corchetes:

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

esta regla permite obtener los literales que tienen los valores más pequeños y su respectivo valor. Esta declaración consiste en mostrar el lugar `X`, indicado con `lugar(X)`, tal que el tiempo `Y` que tarda en trasladarse desde `W` hasta `X`, (indicado mediante `tiempo(W,X,Y) = Y`, donde `= Y` significa que se va a tomar el valor `Y` que corresponde al tiempo que ocupa la persona en trasladarse desde el punto `W` en que se encuentra hasta el lugar `X`, porque es un valor de tipo entero), sea el tiempo mínimo con prioridad 2 indicado mediante (`@ 2`). En este caso la prioridad asignada a la literal `Y` es 2, pero como se mencionó en el inciso a), es posible cambiar el valor de la prioridad dependiendo cuál literal, si se tienen varias literales en una misma declaración de optimización o cual declaración de optimización si solo se tiene un literal, se desea que tenga mayor prioridad. Con esta regla se obtiene como

resultado que es el lugar s_2 con un tiempo de 1 minuto en llegar desde el lugar en donde se encuentra actualmente y también el lugar s_3 con un tiempo de 2 minutos en llegar desde el lugar en donde se encuentra actualmente.

Entonces de lo anterior se puede ver de qué forma es posible utilizar declaraciones de optimización.

Ahora bien, el modelado del problema que se ha propuesto al inicio de este capítulo para elegir a qué lugar debe ir primero una persona es el siguiente, el cual está codificado en Clasp:

```
1 1 { lugar(s3;s2;s1) } 1.

2 distancia(p,s1,5).
3 distancia(p,s2,2).
4 distancia(p,s3,4).

5 tiempo(p,s1,3).
6 tiempo(p,s2,1).
7 tiempo(p,s3,2).
```

en la primera línea se describen los lugares a los cuales debe ir esta persona abreviando hechos. Para el uso de declaraciones de optimización es muy importante el uso de restricciones de cardinalidad como se describió en el capítulo 4, que en este caso debe ser de 1 a 1. De la línea 2 a la línea 4 se especifica la distancia que hay desde el punto `p` en que se encuentra esta persona a los lugares `s1, s2` y `s3`. En las líneas 5 a 7 se indica el tiempo en que esta persona tarda en trasladarse desde el punto `p` en que se encuentra a los lugares `s1, s2` y `s3`. Como es posible observar, en este código solo se está definiendo el problema pero aún no se ha dado solución al mismo, por lo que el resultado que se obtiene con este código es el siguiente:

```
Answer: 1
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s3)
Answer: 2
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
```

```
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s2)
Answer: 3
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s1)
```

estos 3 answer sets muestran las distancias y los tiempos que se han definido y en cada answer set un lugar diferente, esto por el uso de las restricciones de cardinalidad.

Ahora, si a este código le agregamos la primera declaración de optimización `(a)` que se estableció para responder a la pregunta ¿Cuál es el lugar que le queda más lejos a esta persona desde el lugar en que se encuentra? la cual es:

```
#maximize [lugar(X): distancia(W,X,Y) = Y @ 1].
```

entonces se obtendrán los siguientes answer sets:

```
Answer: 1
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s3)
Optimization: 7
Answer: 2
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s1)
Optimization: 6
```

en el primer answer set se muestran las distancias, los tiempos y el lugar `s3` porque como se había dicho, uno de los lugares que le quedaba más lejos a esta persona es el lugar `s3` porque se encuentra a 4 calles del punto en que se encuentra y como usamos `#maximize` y corchetes, en la parte de `Optimization` se muestra el valor 7 que resulta de sumar los valores de las literales restantes que son: la distancia del lugar `s1` con 5 calles de distancia más la distancia del lugar `s2` que son 2 calles, por lo que la suma de estos dos lugares da 7 calles. Para el segundo answer set se tiene que el lugar `s1` es otro de los lugares más lejos porque se encuentra a 5 calles del punto en que se encuentra esta persona y en el resultado de `Optimization` se obtiene el valor 6 porque resulta de sumar la distancia del lugar `s3`

con 4 calles de distancia más la distancia del lugar `s2` con 2 calles de distancia desde donde se encuentra actualmente, por lo que la suma de estos dos lugares da 6 calles.

Como se puede observar, estos resultados si coinciden con lo que se había planteado anteriormente.

Ahora, si al código que se había descrito anteriormente le agregamos la declaración de optimización `(b)` que se estableció para responder a la pregunta ¿A cuál de los lugares tarda menos tiempo en llegar esta persona desde el lugar donde se encuentra actualmente? la cual es:

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

entonces se obtendrán los siguientes answer sets:

```
Answer: 1
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s3)
Optimization: 2
Answer: 2
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s2)
Optimization: 1
```

al igual que en los answer set anteriores, en este primer answer set se muestran las distancias, los tiempos y el lugar `s3` porque se había definido que uno de los lugares a los que tarda menor tiempo en trasladarse desde el lugar en que se encuentra esta persona es el lugar `s3` porque el tiempo de traslado es de 2 minutos y ese tiempo se indica en la parte de `Optimization`. Para el segundo answer set se tienen también las distancias y los tiempos pero ahora el lugar `s2` porque es otro de los lugares a los que tarda menor tiempo en trasladarse desde el lugar en que se encuentra esta persona con un tiempo de 1 minuto el cual se observa en la parte de `Optimization`.

Aquí también se observa que estos resultados si coinciden con lo que se había planteado anteriormente.

Si juntamos las dos declaraciones de optimización `(a)` y `(b)` con el mismo código que se tenía, entonces el código completo será:

```
1 1 { lugar(s3;s2;s1) } 1.

2 distancia(p,s1,5).
3 distancia(p,s2,2).
4 distancia(p,s3,4).

5 tiempo(p,s1,3).
6 tiempo(p,s2,1).
7 tiempo(p,s3,2).

8 #maximize [lugar(X): distancia(W,X,Y) = Y @ 1].
9 #minimize [lugar(X): tiempo(W,X,Y) = Y @ 2].
```

Y los answer sets que se obtendrán serán los siguientes:

```
Answer: 1
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s3)
Optimization: 2 7

Answer: 2
distancia(p,s1,5) distancia(p,s2,2) distancia(p,s3,4)
tiempo(p,s1,3) tiempo(p,s2,1) tiempo(p,s3,2) lugar(s2)
Optimization: 1 9
```

en estos dos answer sets se obtienen los resultados de la misma manera que cuando se agregaron una a una las declaraciones de optimización, pero observemos que en la línea 9 del código la declaración de optimización tiene la literal con mayor prioridad, por lo tanto el primer answer set que se obtiene corresponde al valor mínimo de la literal con mayor prioridad que esta declarada en la línea 9, y en base a esto en la parte de `Optimization` se muestran los valores correspondientes de cada literal.

5.2 Agregación de reglas de preferencias en Clasp al problema de reparto de oficios en una oficina

Si se quisieran aplicar algunas preferencias al problema de reparto de oficios que se ha establecido en los capítulos 3 y 4, para poder saber por ejemplo a cuál oficina entregar primero los oficios en base al horario que ha establecido el empleado o en base al tiempo de traslado que el robot ocupa para llegar a la oficina, sería aplicando declaraciones de optimización para así poder preferir el mejor tiempo o el mejor horario.

De esta manera se pueden plantear las siguientes preguntas que hemos definido:

1. ¿Cuál oficina tiene el menor horario para recibir oficios? Para dar respuesta a esta pregunta se necesita la siguiente declaración de optimización, a la cual llamaremos O_1:

```
#minimize [oficina(X) : horario_oficios(X,Y,Z) = Y @ 1].
```

esta declaración de optimización mostrará la oficina que tenga el horario de oficios `Y` más pequeño.

2. ¿Cuál es la distancia mínima que debe recorrer el robot para llegar a la oficina? Esta pregunta la podemos resolver utilizando esta declaración de optimización, a la cual llamaremos O_2:

```
#minimize [oficina(X) : tiempo_viaje(recepcion,X,Z) = Z @ 2].
```

con esta declaración de optimización podremos obtener la oficina que tiene el menor tiempo de viaje `Z`, desde la recepción a la oficina.

3. Si se tuvieran que entregar oficios con algún un grado de importancia, podríamos preguntarnos, ¿A qué oficina se le van a entregar los oficios que tienen mayor grado de importancia? Esto lo definimos mediante la declaración de optimización siguiente, a la cual llamaremos O_3:

```
#maximize [oficina(X) : oficio_a_entregar(X,Y) = Y @ 3].
```

mediante esta declaración de optimización se obtiene el mayor grado de importancia de los oficios que deberán entregarse a cierta oficina.

Entonces de esta manera se están aplicando algunas preferencias para ver a cual oficina es preferible entregar los oficios.

Recordemos que el código que se tenía para el problema de secuenciación mínima para el caso particular de entrega de oficios en una oficina es el siguiente:

```
#const length = 4.
time(1..length). % tiempo maximo para llegar a la meta

#const reloj = 4.
timeh(0..reloj). % Longitud de los pasos para realizar la accion

lugar(recepcion;oA).

horario_oficios(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

fluent(robot_tiene_oficios).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(tiempo_llegada(T)) :- timeh(T).
fluent(tiempo_salida(T)) :- timeh(T).
fluent(entrego_oficios_en_lugar(L)) :- lugar(L).

action(tomar_oficios).
action(move(L1,L2)):- lugar(L1), lugar(L2).
action(entregar_oficios(L,T_entrega)) :- timeh(T_entrega), lugar(L).

initially(neg(entrego_oficios_en_lugar(oA))).
initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).

finally(entrego_oficios_en_lugar(oA)).
finally(esta_en_lugar(oA)).

executable(tomar_oficios,T) :- T < length, time(T),
holds(esta_en_lugar(recepcion),T), holds(neg(robot_tiene_oficios),T).

executable(move(L1,L2),T) :- T < length, time(T),
holds(esta_en_lugar(L1),T), lugar(L1), lugar(L2), L1 != L2.

executable(entregar_oficios(L,T_entrega),T):-
```

```
                    T < length, time(T), lugar(L), timeh(T_entrega),
                    holds(esta_en_lugar(L),T),
                    holds(robot_tiene_oficios,T),
                    holds(neg(entrego_oficios_en_lugar(L)),T),
                    holds(tiempo_llegada(T_llegada),T),
                    horario_oficios(L,T0,T1), T0<= T_llegada,
                    T_llegada <= T1, T_entrega == T_llegada,
                    timeh(T_llegada), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(tiempo_llegada(T_llegada), move(L1,L2),tiempo_salida(T_salida))
:- tiempo_viaje(L1,L2,Tv), T_llegada = Tv + T_salida,timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2), tiem-
po_llegada(T_llegada)) :-      timeh(T_llegada), lugar(L1), lugar(L2),
L1 != L2.

causes_dd(neg(tiempo_salida(T_salida)), mo-
ve(L1,L2),tiempo_salida(T_salida)) :- timeh(T_salida), lugar(L1),
lugar(L2), L1 != L2.

causes(tomar_oficios, robot_tiene_oficios).

causes(entregar_oficios(L,T_entrega), neg(robot_tiene_oficios)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficios(L,T_entrega), entrego_oficios_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

donde el resultado que se obtenía de este código es el siguiente:

```
Stable Model:
occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
```

el cual ya se ha explicado en el capítulo 3. Pondremos por nombre *P* a este código para referirnos a él posteriormente.

Ahora bien, si agregamos a este programa *P* las declaraciones de optimización `#maximize` y `#minimize` que hemos declarado anteriormente, primero debemos declarar nuevos hechos de la siguiente manera:

```
1 { oficina(oA) } 1.
```

esta línea de código indica la declaración de las oficinas con las cuales se trabajarán para las preferencias y debemos indicarlas con el uso de restricciones de cardinalidad. Los demás hechos a agregar son los tipos de oficios:

```
tipo_oficio(solicitud,3). %Mayor grado de importancia
tipo_oficio(informe,2).   %Grado de importancia intermedio
tipo_oficio(carta,1).      %Menor grado de importancia
```

aquí se indica que se tienen tres tipos de oficios que el robot debe llevar a cada empleado y dependiendo del oficio, se establece un grado de importancia, nótese que ha mayor valor mayor será la prioridad.

Una vez definidos los oficios, se deben asignar los tipos de oficios que se entregarán a cada oficina, esto se indica de la manera siguiente:

```
oficio_a_entregar(oA,2).
```

en esta regla se establece que el tipo de oficios que se le van a entregar a la oficina `oA` tienen un grado de importancia 2, por lo que corresponde a un informe, de acuerdo a los oficios que se han definido.

Una vez que se han definido estos nuevos hechos en el programa P, si unimos a este programa la declaración de optimización O_1 que ya hemos planteado para indicar cuál oficina tiene el menor horario para recibir oficios, obtendremos el siguiente resultado:

```
oficina(oA)
occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
Optimization: 2
```

en donde la primera línea establece que la oficina `oA` tiene el horario de oficios más pequeño y por lo tanto en la parte de `Optimization` se da el valor 2 que indica que el empleado de esta oficina desea recibir sus oficios a partir de las 2 de la tarde. Las siguientes tres líneas que inician con `occurs` indican los pasos de cómo el robot va ejecutando las acciones, lo cual también ya se ha descrito en el capítulo 4.

Si al programa *P* lo unimos ahora con la declaración de optimización O_2 en la que se busca la distancia mínima que debe recorrer el robot para llegar a la oficina, obtenemos el siguiente answer set:

```
oficina(oA)
occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
Optimization: 1
```

en este answer set se indica que la oficina `oA` tiene la distancia mínima ya que el tiempo de viaje desde la recepción a la esta oficina es 1 y este 1 corresponde al valor de `Optimization`.

Si al programa *P* lo unimos ahora con la declaración de optimización O_3 en donde se desea saber a qué oficina se le van a entregar los oficios que tienen mayor grado de importancia, dará como resultado este answer set:

```
oficina(oA)
occurs(tomar_oficios,1)
occurs(move(recepcion,oA),2)
occurs(entregar_oficios(oA,2),3)
Optimization: 0
```

esto indica que la oficina `oA` es la oficina a la que se le van a entregar los oficios con mayor grado de importancia y como no hay más valores porque es la única oficina, entonces devuelve 0 en la parte de `Optimization` ya que no hay valores que sumar.

Con estos cambios realizados, se concluye la parte de preferencias para el problema de secuenciación mínima de reparto de oficios en una oficina.

Agregando estas mismas reglas al código de reparto de oficios para 2, 3, 4 y 5 oficinas se puede observar de acuerdo a estas preferencias, cuál es la oficina óptima, lo cual es posible ver en el *Apéndice*.

Conclusiones

Durante el desarrollo de este trabajo se explicaron algunas diferencias en la implementación de ambas herramientas Smodels y Clasp. La codificación del problema de Secuenciación Mínima correspondiente al reparto de oficios en Smodels resultó ser un tanto difícil por la cuestión de no conocer la sintaxis correcta de esta herramienta, también como se ha observado en los ejemplos descritos, son más las líneas de código que en Clasp, y al momento de ejecutar las pruebas realizadas para obtener los Answer Sets correspondientes, Smodels ocupa un tiempo mayor y conforme se va incrementando el número de oficinas el tiempo de ejecución también se incrementa.

La codificación en Clasp fue más fácil, como ya lo hemos visto en las pruebas realizadas, es muy parecida a la codificación de Smodels con la diferencia que en Clasp reducimos el código mediante el uso de abreviaciones en la declaración de los tiempos y lugares, por otra parte, para la obtención de los Answer Sets, Clasp es mucho más rápido, obtiene los resultados en un tiempo mucho menor que en Smodels no importando el número de oficinas que se tengan. Con esto se puede concluir que Clasp es una herramienta mucho más eficiente pero no más importante que Smodels.

Por otra parte también debemos señalar que el uso de declaraciones de optimización para establecer las preferencias en Clasp, proporcionan answer sets óptimos en los cuales se indica cuál es la oficina con mayor o menor importancia de acuerdo a la prioridad que se establezca en los literales de las declaraciones de optimización definidas.

Apéndice

En esta sección se presentan algunos de los códigos completos correspondientes a las pruebas realizadas para el problema de Secuenciación Mínima para el caso particular de reparto de oficios en las herramientas Smodels y Clasp. Primero se muestran las versiones en Smodels, una tabla que muestra los tiempos en que se obtienen los modelos estables y otra tabla de los resultados obtenidos en la versión final de Smodels. Posterior a esto, se muestran los códigos de las implementaciones en Clasp, las implementaciones utilizando restricciones de cardinalidad y las implementaciones haciendo uso de las declaraciones de optimización para representar preferencias.

En el sitio que ya se ha mencionado podemos encontrar todas las pruebas realizadas junto con las impresiones de pantalla y una breve explicación de lo que hace cada versión: *https://sites.google.com/site/codigoentregadeoficios/*

Versiones en Smodels

Cuarta versión en Smodels para 2 oficinas

```
% EN ESTA VERSIÓN SE OBTUVO EL TIEMPO DE SALIDA DE LA OFICINA AL MOMENTO
% DE QUE SE EJECUTA LA ACCION MOVER DE UNA OFICINA A LA RECEPCIÓN.

const length=8. % Tiempo maximo para llegar a la meta
time(1..length).

const reloj=6. % Longitud de los pasos para realizar la accion
timeh(0..reloj).

lugar(recepcion).
lugar(oA).
lugar(oC).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
```

```
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(t_llegada_aoficina(T)) :- timeh(T).
fluent(t_llegada_amaquina(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega), lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).

finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).

% Se ejecuta mover de recepcion a una oficina
executable(move(L1,L2),T) :- T < length, time(T),
       holds(esta_en_lugar(L1),T), holds(robot_tiene_oficio,T),
       holds(neg(entrego_oficio_en_lugar(L2)),T),
       assign(L1,recepcion), lugar(L1), lugar(L2), neq(L1,L2).

% Se ejecuta mover de una oficina a recepcion
executable(move(L1,L2),T) :- T < length, time(T),
                             holds(esta_en_lugar(L1),T),
                             holds(neg(robot_tiene_oficio),T),
                             holds(entrego_oficio_en_lugar(L1),T),
                             assign(L2,recepcion), lugar(L1),
                             lugar(L2), neq(L1,L2).

executable(tomar_oficio,T) :- T < length, time(T),
                                holds(esta_en_lugar(recepcion),T),
                                holds(neg(robot_tiene_oficio),T).

executable(entregar_oficio(L,T_entrega),T):- T < length, time(T),
            lugar(L), timeh(T_entrega), holds(esta_en_lugar(L),T),
            holds(robot_tiene_oficio,T),
            holds(neg(entrego_oficio_en_lugar(L)),T),
            holds(t_llegada_aoficina(T_llegada_O),T),
            horario_oficio(L,T0,T1), T0 <= T_llegada_O,
            T_llegada_O <= T1, assign(T_entrega,T_llegada_O),
            timeh(T_llegada_O), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), neq(L1,L2).
```

```
causes_dd(t_salida_drecepcion(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m),
timeh(T_llegada_O), timeh(Tv), lugar(L1), lugar(L2), neq(L1,L2).

causes_dd(neg(t_salida_drecepcion(T_salida_m)), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- timeh(T_salida_m), lugar(L1),
lugar(L2), neq(L1,L2).

causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), neq(L1,L2).

causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- timeh(T_llegada_O), lugar(L1),
lugar(L2), neq(L1,L2).

% MODIFICACION
causes_dd(t_salida_doficina(T_salida_O), entregar_oficio(L,T_entrega),
t_llegada_aoficina(T_llegada_o)) :- T_salida_O = T_llegada_o,
lugar(L),timeh(T_entrega), timeh(T_salida_O), timeh(T_llegada_o).

causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficio(L,T_entrega), t_salida_doficina(T_salida_O)) :-
timeh(T_salida_O), timeh(T_entrega), lugar(L).

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
not_goal(T):- time(T),literal(X), finally(X), not holds(X,T).
goal(T) :- time(T),not not_goal(T).
exists_plan :- goal(length).
%:- not exists_plan.

literal(G)  :-fluent(G).
literal(neg(G)) :- fluent(G).
contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).

holds(F, 1) :- literal(F), initially(F).

holds(F, T+1) :- literal(F), time(T), T < length,action(A),
                 executable(A,T), occurs(A,T),causes(A,F).

holds(F, T+1) :- literal(F), time(T), T < length, action(A),
                 executable(A,T), occurs(A,T), causes_dd(F,A,F_inicial),
                 literal(F_inicial), holds(F_inicial,T).
```

```
holds(F, T+1) :- literal(F), literal(G), contrary(F,G), time(T),
T<length, holds(F,T), not holds(G, T+1).

possible(A,T) :- action(A), time(T), executable(A,T), not goal(T).

occurs(A,T) :-action(A),time(T),possible(A,T),not not_occurs(A,T).

not_occurs(A,T) :- action(A),action(AA),time(T),occurs(AA,T), neq(A,AA).

hide time(T).
hide timeh(T).
hide lugar(L).
hide horario_oficio(P,T1,T2).
hide tiempo_viaje(L1,L2,D).
hide action(A).
hide causes(A,F).
hide causes_dd(F,A,F_inicial).
hide initially(F).
hide contrary(F,G).
hide fluent(F).
hide literal(L).
hide executable(A,T).
hide holds(F,T).
hide not_occurs(A,T).
hide possible(A,T).
hide exists_plan.
hide goal(T).
hide not_goal(T).
```

En la siguiente tabla se muestran los tiempos de ejecución obtenidos de este programa pero para diferente número de oficinas. El código para cada programa solo cambia en la agregación de las oficinas por lo que el nombre es diferente.

Archivo	Número oficinas	Tiempo	Numero modelos
oficio_v4_2oficinas	2	0.717 s	1
oficio_v4_3oficinas	3	4.507 s	1
oficio_v4_4oficinas	4	19.452 s	1
oficio_v4_5oficinas	5	103.320 s	1

Quinta versión en Smodels para 5 oficinas

```
% ESTA ES LA VERSIÓN FINAL EN SMODELS. SE NEGÓ EL TIEMPO DE SALIDA DE LA
% OFICINA DESPUÉS DE EJECUTAR LA ACCION MOVER DE UNA OFICINA A LA
% RECEPCIÓN.

const length=20.
time(1..length).

const reloj=18.
timeh(0..reloj).

lugar(recepcion).
lugar(oA).
lugar(oC).
lugar(oM).
lugar(oL).
lugar(oS).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

horario_oficio(oM,7,8).
tiempo_viaje(recepcion,oM,1).
tiempo_viaje(oM,recepcion,1).

horario_oficio(oL,10,11).
tiempo_viaje(recepcion,oL,2).
tiempo_viaje(oL,recepcion,2).

horario_oficio(oS,15,16).
tiempo_viaje(recepcion,oS,2).
tiempo_viaje(oS,recepcion,2).

fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(t_llegada_aoficina(T)) :- timeh(T).
fluent(t_llegada_arecepcion(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega),lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
```

```
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).
initially(neg(entrego_oficio_en_lugar(oM))).
initially(neg(entrego_oficio_en_lugar(oL))).
initially(neg(entrego_oficio_en_lugar(oS))).

finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).
finally(entrego_oficio_en_lugar(oM)).
finally(entrego_oficio_en_lugar(oL)).
finally(entrego_oficio_en_lugar(oS)).

% Se ejecuta mover de recepcion a una oficina
executable(move(L1,L2),T) :-  T < length, time(T),
            holds(esta_en_lugar(L1),T), holds(robot_tiene_oficio,T),
            holds(neg(entrego_oficio_en_lugar(L2)),T),
            assign(L1,recepcion), lugar(L1), lugar(L2), neq(L1,L2).

% Se ejecuta mover de una oficina a recepcion
executable(move(L1,L2),T) :- T < length, time(T),
      holds(esta_en_lugar(L1),T), holds(neg(robot_tiene_oficio),T),
      holds(entrego_oficio_en_lugar(L1),T), assign(L2,recepcion),
      lugar(L1), lugar(L2), neq(L1,L2).

executable(tomar_oficio,T) :- T < length, time(T),
                              holds(esta_en_lugar(recepcion),T),
                              holds(neg(robot_tiene_oficio),T).

executable(entregar_oficio(L,T_entrega),T):-
                  T < length, time(T), lugar(L), timeh(T_entrega),
                  holds(esta_en_lugar(L),T),
                  holds(robot_tiene_oficio,T),
                  holds(neg(entrego_oficio_en_lugar(L)),T),
                  holds(t_llegada_aoficina(T_llegada_O),T),
                  horario_oficio(L,T0,T1), T0<= T_llegada_O,
                  T_llegada_O <= T1, assign(T_entrega,T_llegada_O),
                  timeh(T_llegada_O), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), neq(L1,L2).

causes_dd(t_salida_drecepcion(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), neq(L1,L2).

causes_dd(neg(t_salida_drecepcion(T_salida_m)), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- timeh(T_salida_m), lugar(L1),
lugar(L2), neq(L1,L2).
```

```
causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), neq(L1,L2).

causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- timeh(T_llegada_O), lugar(L1),
lugar(L2), neq(L1,L2).

causes_dd(t_salida_doficina(T_salida_O), entregar_oficio(L,T_entrega),
t_llegada_aoficina(T_llegada_O)) :- T_salida_O = T_llegada_O,
lugar(L),timeh(T_entrega), timeh(T_salida_O), timeh(T_llegada_O).

causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficio(L,T_entrega), t_salida_doficina(T_salida_O)) :-
timeh(T_salida_O), timeh(T_entrega), lugar(L).

% MODIFICACION
causes_dd(neg(t_salida_doficina(T_salida_O)), move(L1,L2),
t_salida_doficina(T_salida_O)) :- timeh(T_salida_O), lugar(L1),
lugar(L2), neq(L1,L2).

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).
causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

Omitiremos la parte de código que va después de esta línea, ya que en todos los códigos se define lo mismo

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Aquí se muestran los resultados obtenidos de la versión final para 5 oficinas en donde es posible verificar que realmente el robot hace entrega de los oficios dentro de los periodos de preferencia de los empleados.

Oficina	Periodo de preferencia	Resultado de la ejecución
oA	horario_oficios(oA,2,3). tiempo_viaje(recepcion,oA,1). tiempo_viaje(oA,recepcion,1).	occurs(tomar_oficios,1) occurs(move(recepcion,oA),2) occurs(entregar_oficios(oA,2),3) occurs(move(oA,recepcion),4)

oC	horario_oficios(oC,4,5). tiempo_viaje(recepcion,oC,2). tiempo_viaje(oC,recepcion,2).	occurs(tomar_oficios,5) occurs(move(recepcion,oC),6) occurs(entregar_oficios(oC,5),7) occurs(move(oC,recepcion),8)
oM	horario_oficios(oM,7,8). tiempo_viaje(recepcion,oM,1). tiempo_viaje(oM,recepcion,1).	occurs(tomar_oficios,9) occurs(move(recepcion,oM),10) occurs(entregar_oficios(oM,8),11) occurs(move(oM,recepcion),12)
oL	horario_oficios(oL,10,11). tiempo_viaje(recepcion,oL,2). tiempo_viaje(oL,recepcion,2).	occurs(tomar_oficios,13) occurs(move(recepcion,oL),14) occurs(entregar_oficios(oL,11),15) occurs(move(oL,recepcion),16)
oS	horario_oficios(oS,15,16). tiempo_viaje(recepcion,oS,2). tiempo_viaje(oS,recepcion,2).	occurs(tomar_oficios,17) occurs(move(recepcion,oS),18) occurs(entregar_oficios(oS,15),19)

Versiones en Clasp

Segunda versión en Clasp para 3 oficinas

```
% EN ESTA VERSIÓN SE MODIFICARON LOS TIEMPOS DE LLEGADA Y SALIDA

#const length = 12.
time(1..length).

#const reloj = 9.
timeh(0..reloj).

lugar(recepcion;oA;oC;oM).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

horario_oficio(oM,7,8).
tiempo_viaje(recepcion,oM,1).
tiempo_viaje(oM,recepcion,1).

fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
```

```
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(tiempo_llegada(T)) :- timeh(T).
fluent(tiempo_salida(T)) :- timeh(T).
fluent(tiempo_total(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega),lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
initially(tiempo_salida(1)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).
initially(neg(entrego_oficio_en_lugar(oM))).

finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).
finally(entrego_oficio_en_lugar(oM)).

executable(tomar_oficio,T) :- T < length,
                        holds(esta_en_lugar(recepcion),T),
                        holds(neg(robot_tiene_oficio),T),
                        time(T).

% Se ejecuta mover de recepcion a una oficina
executable(move(L1,L2),T) :- T<length, holds(esta_en_lugar(L1),T),
                  holds(robot_tiene_oficio,T),
                  holds(neg(entrego_oficio_en_lugar(L2)),T),
                  time(T), lugar(L1), lugar(L2), L1 != L2,
                  holds(tiempo_salida(T_salida),T),
                  horario_oficio(L,T0,T1), tiempo_viaje(L1,L2,Tv),
                  T_total = T_salida + Tv, T0 <= T_total,
                  T_total <= T1, timeh(T_total), timeh(T0),
                  timeh(T1),  timeh(Tv).

% Se ejecuta mover de una oficina a recepcion
executable(move(L1,recepcion),T) :- T < length,
                        holds(esta_en_lugar(L1),T),
                        holds(entrego_oficio_en_lugar(L1),T),
                        holds(neg(robot_tiene_oficio),T),
                        time(T), lugar(L1), L1 != recepcion.

executable(entregar_oficio(L,T_entrega),T):- T < length,
            holds(esta_en_lugar(L),T), holds(robot_tiene_oficio,T),
            holds(tiempo_llegada(T_llegada),T),
            horario_oficio(L,T0,T1),
            holds(neg(entrego_oficio_en_lugar(L)),T),
            T_entrega := T_llegada,T0 <= T_llegada, T_llegada <= T1,
            timeh(T_llegada), timeh(T0), timeh(T1), time(T),
            lugar(L), timeh(T_entrega).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
```

```
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(tiempo_llegada(T_llegada), move(L1,L2),tiempo_salida(T_salida))
:- tiempo_viaje(L1,L2,Tv), T_llegada = Tv + T_salida, timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2),
tiempo_llegada(T_llegada)) :-timeh(T_llegada),lugar(L1),lugar(L2),L1!=L2.

% MODIFICACION
causes_dd(tiempo_salida(T_salida), move(L1,L2),tiempo_llegada(T_llegada))
:- tiempo_viaje(L1,L2,Tv), T_salida = Tv + T_llegada, timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_salida(T_salida)), move(L1,L2),
tiempo_salida(T_salida)) :- timeh(T_salida),lugar(L1),lugar(L2),L1 != L2.

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

not_goal(T):- time(T),literal(X), finally(X), not holds(X,T).
goal(T) :- time(T),not not_goal(T).
exists_plan :- goal(length).
%:- not exists_plan.

literal(G)  :-fluent(G).
literal(neg(G)) :- fluent(G).
contrary(F, neg(F)) :- fluent(F).
contrary(neg(F), F) :- fluent(F).

holds(F, 1) :- literal(F), initially(F).

holds(F, T+1) :- literal(F), time(T), T < length,action(A),
                 executable(A,T), occurs(A,T),causes(A,F).

holds(F, T+1) :- literal(F), time(T), T < length, action(A),
                 executable(A,T), occurs(A,T), causes_dd(F,A,F_inicial),
                 literal(F_inicial), holds(F_inicial,T).

holds(F, T+1) :- literal(F), literal(G), contrary(F,G), time(T),
T<length, holds(F,T), not holds(G, T+1).

possible(A,T) :- action(A), time(T), executable(A,T), not goal(T).

occurs(A,T) :-action(A),time(T),possible(A,T),not not_occurs(A,T).
```

```
not_occurs(A,T) :- action(A), action(AA),time(T),occurs(AA,T), neq(A,AA).

#hide time(T).
#hide timeh(T).
#hide lugar(L).
#hide horario_oficio(P,T1,T2).
#hide tiempo_viaje(L1, L2,D).
#hide action(A).
#hide causes(A,F).
#hide causes_dd(F,A,F_inicial).
#hide initially(F).
#hide contrary(F,G).
#hide fluent(F).
#hide literal(L).
#hide executable(A,T).
#hide holds(F,T).
#hide not_occurs(A,T).
#hide possible(A,T).
#hide exists_plan.
#hide finally(X).
#hide goal(T).
#hide not_goal(T).
```

Segunda versión en Clasp para 3 oficinas con restricciones de cardinalidad

```
% SE AGREGARON LAS RESTRICCIONES DE CARDINALIDAD

#const length = 12.
time(1..length).

#const reloj = 9.
timeh(0..reloj).

lugar(recepcion;oA;Oc;oM).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

horario_oficio(oM,7,8).
tiempo_viaje(recepcion,oM,1).
tiempo_viaje(oM,recepcion,1).


fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(tiempo_llegada(T)) :- timeh(T).
```

```
fluent(tiempo_salida(T)) :- timeh(T).
fluent(tiempo_total(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega),lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
initially(tiempo_salida(1)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).
initially(neg(entrego_oficio_en_lugar(oM))).


finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).
finally(entrego_oficio_en_lugar(oM)).

1{executable(tomar_oficio,T) : T < length : time(T) } 2 :-
holds(esta_en_lugar(recepcion),T), holds(neg(robot_tiene_oficio),T).

% Se ejecuta mover de recepcion a una oficina
1{executable(move(L1,L2),T) : T < length : time(T) : lugar(L1): lugar(L2)
: L1 != L2} 2 :-  holds(esta_en_lugar(L1),T),holds(robot_tiene_oficio,T),
         holds(neg(entrego_oficio_en_lugar(L2)),T),
         holds(tiempo_salida(T_salida),T),
         horario_oficio(L,T0,T1), tiempo_viaje(L1,L2,Tv),
         T_total = T_salida + Tv, T0 <= T_total, T_total <= T1,
         timeh(T_total), timeh(T0), timeh(T1), timeh(Tv).

% Se ejecuta mover de una oficina a recepcion
1{executable(move(L1,recepcion),T) : T < length : time(T) : lugar(L1) :
L1 != recepcion } 2.

executable(move(L1,recepcion),T) :- holds(esta_en_lugar(L1),T),
                                    holds(entrego_oficio_en_lugar(L1),T),
                                    holds(neg(robot_tiene_oficio),T).

1{executable(entregar_oficio(L,T_entrega),T) : T < length : time(T):
lugar(L): timeh(T_entrega)} 2 :-
%: timeh(T_entrega;T_llegada;T0;T1)}1 :-%TAMBIEN ES POSIBLE DE ESTA FORMA
            holds(esta_en_lugar(L),T), holds(robot_tiene_oficio,T),
            holds(tiempo_llegada(T_llegada),T),
            horario_oficio(L,T0,T1),
            holds(neg(entrego_oficio_en_lugar(L)),T),
            T_entrega := T_llegada,T0 <= T_llegada, T_llegada <= T1,
            timeh(T_llegada), timeh(T0), timeh(T1),time(T),lugar(L).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.
```

```
causes_dd(tiempo_llegada(T_llegada), move(L1,L2),tiempo_salida(T_salida))
:- tiempo_viaje(L1,L2,Tv), T_llegada = Tv + T_salida, timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2), tiem-
po_llegada(T_llegada)) :- timeh(T_llegada),lugar(L1),lugar(L2), L1 != L2.

% MODIFICACION
causes_dd(tiempo_salida(T_salida), move(L1,L2),tiempo_llegada(T_llegada))
:- tiempo_viaje(L1,L2,Tv), T_salida = Tv + T_llegada, timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_salida(T_salida)), move(L1,L2),
tiempo_salida(T_salida)) :- timeh(T_salida),lugar(L1),lugar(L2),L1 != L2.

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

Tercera versión en Clasp para 2 oficinas

```
% SE HIZO EL CAMBIO CORRESPONDIENTE DE LAS VARIABLES PARA UN MEJOR
% ENTENDIMIENTO Y SE NEGO EL TIEMPO DE SALIDA DE LA OFICINA DESPUES
% DE EJECUTAR LA ACCION MOVER DE UNA OFICINA A LA RECEPCION.

#const length = 8.
time(1..length).

#const reloj = 6.
timeh(0..reloj).

lugar(recepcion;oA;oC).

horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(t_llegada_aoficina(T)) :- timeh(T).
fluent(t_llegada_arecepcion(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion(T)) :- timeh(T).
```

```
fluent(tiempo_total(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega),lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).

finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).

executable(tomar_oficio,T) :- T < length,
                        holds(esta_en_lugar(recepcion),T),
                        holds(neg(robot_tiene_oficio),T),
                        time(T).

% Se ejecuta mover de recepcion a una oficina
executable(move(L1,L2),T) :- T<length, holds(esta_en_lugar(L1),T),
                        holds(robot_tiene_oficio,T),
                        holds(neg(entrego_oficio_en_lugar(L2)),T),
                        time(T), lugar(L1), lugar(L2), L1 != L2,
                        holds(t_salida_drecepcion(T_salida),T),
                        horario_oficio(L,T0,T1),
                        tiempo_viaje(L1,L2,Tv),
                        T_total = T_salida + Tv,
                        T0 <= T_total, T_total <= T1, timeh(T_total),
                        timeh(T0), timeh(T1),   timeh(Tv).

% Se ejecuta mover de una oficina a recepcion
executable(move(L1,recepcion),T) :- T < length,
                        holds(esta_en_lugar(L1),T),
                        holds(entrego_oficio_en_lugar(L1),T),
                        holds(neg(robot_tiene_oficio),T),
                        time(T), lugar(L1), L1 != recepcion.

executable(entregar_oficio(L,T_entrega),T):- T < length,
                  holds(esta_en_lugar(L),T),
                  holds(robot_tiene_oficio,T),
                  holds(t_llegada_aoficina(T_llegada_O),T),
                  horario_oficio(L,T0,T1),
                  holds(neg(entrego_oficio_en_lugar(L)),T),
                  T_entrega := T_llegada_O,
                  T0 <= T_llegada_O, T_llegada_O <= T1,
                  timeh(T_llegada_O), timeh(T0), timeh(T1),
                  time(T), lugar(L), timeh(T_entrega).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).
```

```
causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- timeh(T_llegada_O), lugar(L1),
lugar(L2), L1 != L2.

causes_dd(t_salida_drecepcion(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(t_salida_drecepcion(T_salida_m)), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- timeh(T_salida_m), lugar(L1),
lugar(L2), L1 != L2.

% MODIFICACION DE LOS 3 CAUSES_dd
causes_dd(t_salida_doficina(T_salida_O), entregar_oficio(L,T_entrega),
t_llegada_aoficina(T_llegada_o)) :- T_salida_O = T_llegada_o,
lugar(L),timeh(T_entrega), timeh(T_salida_O), timeh(T_llegada_o).

causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficio(L,T_entrega), t_salida_doficina(T_salida_O)) :-
timeh(T_salida_O), timeh(T_entrega), lugar(L).

causes_dd(neg(t_salida_doficina(T_salida_O)), move(L1,L2),
t_salida_doficina(T_salida_O)) :- timeh(T_salida_O), lugar(L1),
lugar(L2), L1 != L2.

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

Tercera versión en Clasp para 2 oficinas con restricciones de cardinalidad

```
#const length = 8.
time(1..length). % tiempo maximo para llegar a la meta

#const reloj = 6.
timeh(0..reloj). % Longitud de los pasos para realizar la accion

lugar(recepcion;oA;oC).
```

```
horario_oficio(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficio(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

fluent(entrego_oficio_en_lugar(L)) :- lugar(L).
fluent(robot_tiene_oficio).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(t_llegada_aoficina(T)) :- timeh(T).
fluent(t_llegada_arecepcion(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion(T)) :- timeh(T).
fluent(tiempo_total(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficio).
action(entregar_oficio(L,T_entrega)) :- timeh(T_entrega),lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficio)).
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
initially(neg(entrego_oficio_en_lugar(oA))).
initially(neg(entrego_oficio_en_lugar(oC))).

finally(entrego_oficio_en_lugar(oA)).
finally(entrego_oficio_en_lugar(oC)).

1{executable(tomar_oficio,T) : T < length : time(T)} 2 :-
holds(esta_en_lugar(recepcion),T), holds(neg(robot_tiene_oficio),T).

% Se ejecuta mover de recepcion a una oficina
1{executable(move(L1,L2),T) : T < length : time(T): lugar(L1):
lugar(L2): L1 != L2} 2 :- holds(esta_en_lugar(L1),T),
                          holds(robot_tiene_oficio,T),
                          holds(neg(entrego_oficio_en_lugar(L2)),T),
                          holds(t_salida_drecepcion(T_salida),T),
                          horario_oficio(L,T0,T1), tiempo_viaje(L1,L2,Tv),
                          T_total = T_salida + Tv, T0 <= T_total,
                          T_total <= T1, timeh(T_total), timeh(T0),
                          timeh(T1),  timeh(Tv).

% Se ejecuta mover de una oficina a recepcion
1{executable(move(L1,recepcion),T) : T < length: time(T):
lugar(L1): L1 != recepcion} 2.

executable(move(L1,recepcion),T) :- holds(esta_en_lugar(L1),T),
holds(entrego_oficio_en_lugar(L1),T), holds(neg(robot_tiene_oficio),T).

1{executable(entregar_oficio(L,T_entrega),T) : T < length:
time(T): lugar(L): timeh(T_entrega)} 2: holds(esta_en_lugar(L),T),
```

```
                    holds(robot_tiene_oficio,T),
                    holds(t_llegada_aoficina(T_llegada_O),T),
                    horario_oficio(L,T0,T1),
                    holds(neg(entrego_oficio_en_lugar(L)),T),
                    T_entrega := T_llegada_O,
                    T0 <= T_llegada_O, T_llegada_O <= T1,
                    timeh(T_llegada_O), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.
causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- timeh(T_llegada_O), lugar(L1),
lugar(L2), L1 != L2.

causes_dd(t_salida_drecepcion(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m),
timeh(T_llegada_O), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(t_salida_drecepcion(T_salida_m)), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- timeh(T_salida_m), lugar(L1),
lugar(L2), L1 != L2.

% AGREGACION DE LOS 3 ULTIMOS causes_add
causes_dd(t_salida_doficina(T_salida_O), entregar_oficio(L,T_entrega),
t_llegada_aoficina(T_llegada_o)) :-
T_salida_O = T_llegada_o, lugar(L),timeh(T_entrega),
timeh(T_salida_O), timeh(T_llegada_o).

causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficio(L,T_entrega), t_salida_doficina(T_salida_O)) :-
timeh(T_salida_O), timeh(T_entrega), lugar(L).

causes_dd(neg(t_salida_doficina(T_salida_O)), move(L1,L2),
t_salida_doficina(T_salida_O)) :- timeh(T_salida_O), lugar(L1),
lugar(L2), L1 != L2.

causes(tomar_oficio, robot_tiene_oficio).

causes(entregar_oficio(L,T_entrega), neg(robot_tiene_oficio)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficio(L,T_entrega), entrego_oficio_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

Versiones en Clasp con preferencias

Como se ha explicado en el capítulo 5, se establece el uso de preferencias para el problema de secuenciación mínima de reparto de oficios en la codificación de Clasp mediante el uso de declaraciones de optimización. Estas son algunas de las versiones en donde se muestra el uso de preferencias.

Primera versión en Clasp para 1 oficina con restricciones de cardinalidad y preferencias

```
#const length = 4.
time(1..length). % tiempo maximo para llegar a la meta

#const reloj = 4.
timeh(0..reloj). % Longitud de los pasos para realizar la accion

lugar(recepcion;oA).

1 { oficina(oA) } 1.

%tipo de oficios consta de(tipo, grado de importancia)
tipo_oficio(solicitud,3). %Mayor grado de importancia
tipo_oficio(informe,2).   %Grado de importancia intermedio
tipo_oficio(carta,1).      %Menor grado de importancia

%Tipo de oficio a entregar en la oficina (oficina,grado de importancia)
oficio_a_entregar(oA,2).

horario_oficios(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

fluent(robot_tiene_oficios).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(tiempo_llegada(T)) :- timeh(T).
fluent(tiempo_salida(T)) :- timeh(T).
fluent(entrego_oficios_en_lugar(L)) :- lugar(L).

action(tomar_oficios).
action(move(L1,L2)):- lugar(L1), lugar(L2).
action(entregar_oficios(L,T_entrega)) :- timeh(T_entrega), lugar(L).

initially(neg(entrego_oficios_en_lugar(oA))).
initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(tiempo_salida(1)).
```

```
finally(entrego_oficios_en_lugar(oA)).
finally(esta_en_lugar(oA)).

%%%%%%%%%%%%%%%%%%  OPTIMIZACIÓN %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Se minimiza el tiempo del horario de oficios
#minimize [ oficina(X) : horario_oficios(X,Y,Z) = Y @ 1].

%Se obtiene la menor distancia de oficina
#minimize [ oficina(X) : tiempo_viaje(recepcion,X,Z) = Z @ 2].

%Se obtiene el mayor grado de importancia del oficio
#maximize [ oficina(X) : oficio_a_entregar(X,Y) = Y @ 3].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

1{executable(tomar_oficios,T) : T < length: time(T) }1.

executable(tomar_oficios,T) :- holds(esta_en_lugar(recepcion),T),
                               holds(neg(robot_tiene_oficios),T).

1{executable(move(L1,L2),T) : T<length: time(T) : lugar(L1): lugar(L2)}1.

1{executable(entregar_oficios(L,T_entrega),T) : T < length: time(T):
lugar(L): timeh(T_entrega;T_llegada;T0;T1)}1.

executable(entregar_oficios(L,T_entrega),T):- timeh(T_entrega),
      holds(esta_en_lugar(L),T), holds(robot_tiene_oficios,T),
      holds(neg(entrego_oficios_en_lugar(L)),T),
      holds(tiempo_llegada(T_llegada),T), horario_oficios(L,T0,T1),
      T0<= T_llegada, T_llegada <= T1, T_entrega == T_llegada.

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(tiempo_llegada(T_llegada), move(L1,L2),tiempo_salida(T_salida))
:- tiempo_viaje(L1,L2,Tv), T_llegada = Tv + T_salida, timeh(T_salida),
timeh(T_llegada), timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(tiempo_llegada(T_llegada)), move(L1,L2),
tiempo_llegada(T_llegada)) :-timeh(T_llegada),lugar(L1),lugar(L2),L1!=L2.

causes_dd(neg(tiempo_salida(T_salida)), move(L1,L2),
tiempo_salida(T_salida)) :- timeh(T_salida),lugar(L1),lugar(L2),L1 != L2.

causes(tomar_oficios, robot_tiene_oficios).

causes(entregar_oficios(L,T_entrega), neg(robot_tiene_oficios)) :-
timeh(T_entrega), lugar(L).
```

```
causes(entregar_oficios(L,T_entrega), entrego_oficios_en_lugar(L)):-
timeh(T_entrega), lugar(L).
```

Quinta versión en Clasp para 5 oficinas con restricciones de cardinalidad y preferencias

```
#const length = 20.
time(1..length). % tiempo maximo para llegar a la meta

#const reloj = 16.
timeh(0..reloj). % Longitud de los pasos para realizar la accion

lugar(recepcion;oA;oC;oM;oL;oS).

1 { oficina(oA;oC;oM;oL;oS) } 1.

%tipo de oficios consta de (tipo, grado de importancia)
tipo_oficio(solicitud,3). %Mayor grado de importancia
tipo_oficio(informe,2).   %Grado de importancia intermedio
tipo_oficio(carta,1).       %Menor grado de importancia

%Tipo de oficio a entregar en la oficina (oficina, grado de importancia)
oficio_a_entregar(oA,2).
oficio_a_entregar(oC,1).
oficio_a_entregar(oM,3).
oficio_a_entregar(oL,1).
oficio_a_entregar(oS,3).

horario_oficios(oA,2,3).
tiempo_viaje(recepcion,oA,1).
tiempo_viaje(oA,recepcion,1).

horario_oficios(oC,4,5).
tiempo_viaje(recepcion,oC,2).
tiempo_viaje(oC,recepcion,2).

horario_oficios(oM,7,8).
tiempo_viaje(recepcion,oM,1).
tiempo_viaje(oM,recepcion,1).

horario_oficios(oL,10,11).
tiempo_viaje(recepcion,oL,2).
tiempo_viaje(oL,recepcion,2).

horario_oficios(oS,15,16).
tiempo_viaje(recepcion,oS,2).
tiempo_viaje(oS,recepcion,2).

fluent(entrego_oficios_en_lugar(L)) :- lugar(L).
fluent(robot_tiene_oficios).
fluent(esta_en_lugar(L)):- lugar(L).
fluent(t_llegada_aoficina(T)) :- timeh(T).
```

```
fluent(t_llegada_arecepcion(T)) :- timeh(T).
fluent(t_salida_doficina(T)) :- timeh(T).
fluent(t_salida_drecepcion(T)) :- timeh(T).
fluent(tiempo_total(T)) :- timeh(T).

action(move(L1,L2)):- lugar(L1), lugar(L2).
action(tomar_oficios).
action(entregar_oficios(L,T_entrega)) :- timeh(T_entrega), lugar(L).

initially(esta_en_lugar(recepcion)).
initially(neg(robot_tiene_oficios)).
initially(t_salida_drecepcion(1)).
initially(t_salida_doficina(0)).
initially(neg(entrego_oficios_en_lugar(oA))).
initially(neg(entrego_oficios_en_lugar(oC))).
initially(neg(entrego_oficios_en_lugar(oM))).
initially(neg(entrego_oficios_en_lugar(oL))).
initially(neg(entrego_oficios_en_lugar(oS))).

finally(entrego_oficios_en_lugar(oA)).
finally(entrego_oficios_en_lugar(oC)).
finally(entrego_oficios_en_lugar(oM)).
finally(entrego_oficios_en_lugar(oL)).
finally(entrego_oficios_en_lugar(oS)).

%%%%%%%%%%%%%%%%%%%%%%%%%  OPTIMIZACIÓN %%%%%%%%%%%%%%%%%%%%%%%%%

%Se minimiza el tiempo del horario de oficios
#minimize [ oficina(X) : horario_oficios(X,Y,Z) = Y @ 1].

%Se obtiene la menor distancia de oficina
#minimize [ oficina(X) : tiempo_viaje(recepcion,X,Z) = Z @ 2].

%Se obtiene el mayor grado de importancia del oficio
#maximize [ oficina(X) : oficio_a_entregar(X,Y) = Y @ 3].

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

1{executable(tomar_oficios,T) : T < length : time(T)} 5 :-
holds(esta_en_lugar(recepcion),T), holds(neg(robot_tiene_oficios),T).

% Se ejecuta mover de recepcion a una oficina
1{executable(move(L1,L2),T) : T < length : time(T): lugar(L1): lugar(L2):
L1 != L2} 5 :- holds(esta_en_lugar(L1),T), holds(robot_tiene_oficios,T),
               holds(neg(entrego_oficios_en_lugar(L2)),T),
               holds(t_salida_drecepcion(T_salida),T),
               horario_oficios(L,T0,T1), tiempo_viaje(L1,L2,Tv),
               T_total = T_salida + Tv, T0 <= T_total, T_total <= T1,
               timeh(T_total), timeh(T0), timeh(T1), timeh(Tv).

% Se ejecuta mover de una oficina a recepcion
1{executable(move(L1,recepcion),T) : T < length: time(T): lugar(L1): L1
!= recepcion} 5.
```

```
executable(move(L1,recepcion),T) :- holds(esta_en_lugar(L1),T),
holds(entrego_oficios_en_lugar(L1),T), holds(neg(robot_tiene_oficios),T).

1{executable(entregar_oficios(L,T_entrega),T) : T < length: time(T): lu-
gar(L): timeh(T_entrega)} 5 :- holds(esta_en_lugar(L),T),
holds(robot_tiene_oficios,T), holds(t_llegada_aoficina(T_llegada_O),T),
horario_oficios(L,T0,T1), holds(neg(entrego_oficios_en_lugar(L)),T),
T_entrega := T_llegada_O, T0 <= T_llegada_O, T_llegada_O <= T1,
timeh(T_llegada_O), timeh(T0), timeh(T1).

causes_dd(esta_en_lugar(L2), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2).

causes_dd(neg(esta_en_lugar(L1)), move(L1,L2), esta_en_lugar(L1)) :-
lugar(L1), lugar(L2), L1 != L2.

causes_dd(t_llegada_aoficina(T_llegada_O), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- tiempo_viaje(L1,L2,Tv),
T_llegada_O = Tv + T_salida_m, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(t_llegada_aoficina(T_llegada_O)), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :-  timeh(T_llegada_O), lugar(L1),
lugar(L2), L1 != L2.

causes_dd(t_salida_drecepcion(T_salida_m), move(L1,L2),
t_llegada_aoficina(T_llegada_O)) :- tiempo_viaje(L1,L2,Tv),
T_salida_m = Tv + T_llegada_O, timeh(T_salida_m), timeh(T_llegada_O),
timeh(Tv), lugar(L1), lugar(L2), L1 != L2.

causes_dd(neg(t_salida_drecepcion(T_salida_m)), move(L1,L2),
t_salida_drecepcion(T_salida_m)) :- timeh(T_salida_m), lugar(L1),
lugar(L2), L1 != L2.

causes_dd(t_salida_doficina(T_salida_O), entregar_oficios(L,T_entrega),
t_llegada_aoficina(T_llegada_o)) :- T_salida_O = T_llegada_o,
lugar(L),timeh(T_entrega), timeh(T_salida_O), timeh(T_llegada_o).

causes_dd(neg(t_salida_doficina(T_salida_O)),
entregar_oficios(L,T_entrega), t_salida_doficina(T_salida_O)) :-
timeh(T_salida_O), timeh(T_entrega), lugar(L).

causes_dd(neg(t_salida_doficina(T_salida_O)), move(L1,L2),
t_salida_doficina(T_salida_O)) :- timeh(T_salida_O), lugar(L1),
lugar(L2), L1 != L2.

causes(tomar_oficios, robot_tiene_oficios).

causes(entregar_oficios(L,T_entrega), neg(robot_tiene_oficios)) :-
timeh(T_entrega), lugar(L).

causes(entregar_oficios(L,T_entrega), entrego_oficios_en_lugar(L)) :-
timeh(T_entrega), lugar(L).
```

Bibliografía

[1] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving whit Answer Sets*. Cambridge University Press, Cambridge, 2003.

[2] Colmerauer A., Kanoui H., Pasero R., Roussel P. Un systeme de communication home machine en Francais. *Technical report, University of Marseille*, 1973.

[3] Eiter T., Leone N., Mateis C., Pfeifer G., Scarcello F. A deductive system for nonmonotonic reasoning. *In Dix J., Furbach U., Nerode A., eds. Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer. 364–375, 1997.

[4] Francisco Javier E. y Santiago R. Control de un robot móvil utilizando un planificador. Tesis de licenciatura. Universidad Juárez Autónoma de Tabasco, 2010. Online http://es.scribd.com/doc/54606270/13/Planificacion-clasica (última fecha de verificación mayo 2012), páginas 91.

[5] Gebser M., Kaufmann B., Kaminski R., Ostrowski M. Schaub T., and Thiele S. A User's Guide to gringo, clasp, clingo, and iclingo, 2010. Online http://www.cs.utexas.edu/~vl/teaching/lbai/clingo_guide.pdf (última fecha de verificación mayo 2012), páginas 55.

[6] Gebser M., Liu L., Namasivayam G., Neumann A., Schaub T., and Truszczyński A. The First Answer Set Programming System Competition. Journal. Online http://cs.engr.uky.edu/ai/papers.dir/asp-contest.pdf (última fecha de verificación mayo 2012), páginas 15.

[7] Guillermo Didier Bravo. *Desarrollo y modelado de algoritmos para la genotipificación de secuencias: el caso de los transplantes, alelos HLA*. Tesis de Licenciatura. Universidad de las Américas Puebla, 2005. Online http://catarina.udlap.mx/u_dl_a/tales/documentos/lis/bravo_a_gd/capitulo3.pdf (última fecha de verificación mayo 2012), páginas 77.

[8] http://web.ing.puc.cl/~marenas/iic2212-08/clases/lp-b.pdf

[9] http://www.cs.us.es/~jalonso/cursos/li-03/temas/tema-1.pdf

[10] http://www.dc.fi.udc.es/muc/sites/www.dc.fi.udc.es.muc/files/slides-KR.pdf

[11]http://www.docstoc.com/docs/280335/Programacion-logica-y-funcional-incluye--Ejemplos-en-Prolog

[12]http://www.ia.urjc.es/cms/sites/default/files/userfiles/file/ia3/teoria/Intro-ASP.pdf

[13] http://www.it.uc3m.es/jvillena/irc/practicas/estudios/Lenguajes_Logicos.pdf

[14] http://www.oscarestrada.com.mx/archivos/Platica-Tetela-Presen.pdf

[15] Ilkka Niemelä and Patrick Simmons. Smodels – an implementation of the stable model and well-founded semantics for normal logic programs. In Dix J., Furbach U., Nerode A., eds. Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning. Springer, 420–429, 1997.

[16] Ilkka Niemelä. Logic programming with stable model semantics as a constraint programming paradigm. Annals of Mathematics and Artificial Intelligence. 25:3-4, 241–273, 1999.

[17] Juan Antonio Navarro. *Lógica Aplicada a Answer Sets*. Tesis de Licenciatura. Universidad de las Américas Puebla, 2003. Online http://www.mpi-sws.org/~jnavarro/papers/uthesis.pdf (última fecha de verificación mayo 2012), páginas 61.

[18] Kowalski R. Predicate logic as a programming language. In Rosenfeld J., ed. *Proceedings of the Congress of the International Federation for Information Processing*. North Holland. 569–574, 1974.

[19] Marek V., Truszczyński M. *Stable models and an alternative logic programming paradigm*. In Apt K., Marek W., Truszczyński M., Warren D., eds. The Logic Programming Paradigm: a 25-Year Perspective. Springer. 375–398, 1999.

[20] McCarthy J. and Hayes P. J. *Some philosophical problems from the standpoint of artificial intelligence*. In Machine Intelligence 4. B. Meltzer and D. Michie, Eds. Edinburgh University Press. 463-502, 1969.

[21] McCarthy J. Circumscription – a form of nonmonotonic reasoning. Artificial Intelligence. 13:1-2, 27–39, 1980.

[22] Michael Gelfond and Vladimir Lifschitz. *Action languages. Electron. Trans. Artif. Intell*. 2:193-210, 1998.

[23] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors. 5th Conference on Logic Programming. MIT Press, pages 1070–1080, 1988.

[24] Ravi Sethi. *Programming Languages: Concepts & Constructs*. 2nd Ed. Addison-Wesley, 1997.

[25] Reiter R. *A logic for default reasoning*. Artificial Intelligence 13:1-2, 81–132, 1980.

[26] Subrahmanian V. and Zaniolo C. Relating stable models and AI planning domains. *In Proceedings of the 12 th International Conference on Logic Programming, L. Sterling*. Pages 233-247. Tokyo, Japan, 1995. MIT Press.

[27] Tommy Syrjänen. Lparse 1.0 user's manual. http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz. 4, 18, 19, 22, 25, 31, 36, 41, 43, 48, 49, 54